Efficient Circuit Design with uDL

Steve Hoover
Intel
Massachusetts Microprocessor Design Center
Hudson, MA 01749
steve.hoover@intel.com

Abstract

uDL (Microarchitecture Design Language) is an RTL-replacement concept language that separates the cycle-accurate behavioral description of a design from the physical details necessary to implement it. Physical details such as pipeline staging, physical partitioning, and signal encoding augment the behavioral model and do not affect behavior. Furthermore, a refinement mechanism provides a generic mechanism for abstraction. Equivalence of refinements is verified formally or via shadow simulation.

The behavioral model is the target of dynamic verification. It is both simpler than RTL and less subject to physically motivated design changes. uDL designs can be more-easily leveraged and retargeted to new technologies as the high-level design is not obscured by the physical detail, and that detail can easily be changed without introducing bugs. A uDL model has information about the flow and validity of data. This meta-information provides opportunities that can fuel the next generation of design automation tools.

1. Introduction

The design of a large, high-performance integrated circuit, such as a microprocessor chip, can employ hundreds of people for several years. The investment, especially in verification, continues to grow from one generation to the next, as designs become more complex, and as fabrication technologies advance, allowing more logic to fit onto a single microchip. To combat this trend design teams increasingly reuse and leverage previous designs and integrate third-party IP. However, retargeting these designs for new process technologies and new constraints, especially frequency, can also involve significant effort, including re-verification.

Many high-level modeling languages and methodologies have been introduced by academia, from EDA companies, and within design teams over the years in an attempt to bring the design focus to a level above that of RTL. Previous attempts have run into several technical obstacles:

- Those such as Esterel [1] and, to a lesser extent, BlueSpec [2], which rely on synthesizing designs from higher-level models, sacrifice control over physical details.

- Performance modeling and design space exploration languages, like ADL [9] and Asim [11], represent an additional model and an additional language beyond RTL in the design process, which imposes a significant development and maintenance cost. They also do not provide a verification path to RTL and thus do not eliminate verification effort.
- Models requiring cycle-accuracy to validate versus RTL must incorporate complexity as physical details begin to impact behavior.

Non-technical obstacles exist as well. Academic pursuits have generally failed to integrate into complex real-world environments, and endeavors motivated from within design teams have been constrained by project deliverables. The Microarchitecture Design Language (uDL) project is an attempt to encapsulate ideas motivated by real-world experience and share them with the industry. uDL is an experimental RTL-replacement language concept that introduces several mechanisms for abstraction in a manner that reduces design (and redesign) effort.

The rest of the paper covers uDL as follows. Section 2 describes the methodology enabled by uDL. Section 3 describes the mechanisms for mapping the abstract model to physical details. A "refinement" mechanism is described in Section 4. Section 5 describes the uDL data flow model and its benefits. Treatment of reset and other mode transitions is discussed in Section 6. The applicability of uDL to performance modeling is discussed in Section 7. And Section 8 provides a summary.

2. uDL Methodology

uDL addresses the technical challenges identified above. Rather than introducing an additional model and language, it replaces RTL, providing the same level of detail. It does so in two distinct (though interwoven) parts: the **behavioral model**, which, though cycle-accurate, is void of physical detail, and the **mappings** and **refinements** from that model to the physical details needed to feed downstream tools and flows. This is pictured in Figure 1. Mappings are correct by construction (with a path to verify this assumption through simulation) and refinements are verified individually, formally or in shadow simulation.

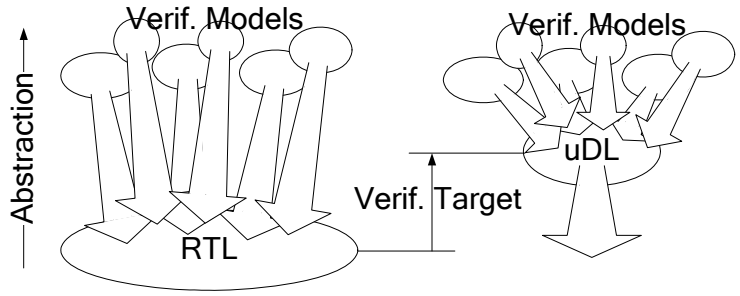

Figure 1: uDL Methodology

Total effort is reduced in several dimensions:

- The behavioral model is the target of verification, and this model is more abstract than RTL, so the verification effort is reduced.
- The entire uDL model (behavioral model plus mappings) tends to be about half the size of RTL, based on our exploration, and provide the same information, so the RTL modeling effort is reduced.

- A plethora of verification models map into the RTL: transactors, reference models, coverage monitors, visualization tools, protocol models, instruction set models, and more. The mappings can, at times, be more complex than the verification model themselves. With uDL, these mappings have less "distance" to cover vs. RTL, so they are simpler. More importantly, they are subject to far less churn as they are isolated from most physically-motivated changes.
- Since most physically-motivated changes do not impact the behavioral model, they cannot introduce bugs, do not require regression testing, and can be incorporated much faster.
- The design is easier to understand as the physical details can easily be identified and filtered.
- Designs are much more easily leveraged due to their ease of understanding and the ability to retarget the design through uDL mapping capabilities.

3. Mappings

Design elements which are absent from the behavioral model include: cycle (and phase) timing (pipe stages), physical partitioning, and signal encodings. Each of these is provided via mappings, which provide the physical detail while preserving cycle-accurate functionality. As illustrated in Figure 2, every element in the design, such as an "opcode" variable, is provided these three properties through mappings.

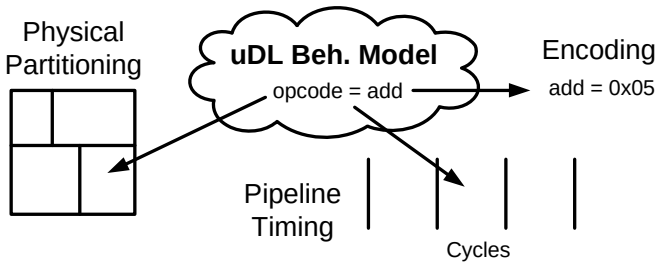

Figure 2: Forms of uDL Mappings

While Lava [3] and Wired [4] introduce mechanisms for providing data-path logic with physical layout properties, uDL focuses on mappings which provide the same information that can be found in RTL and does not necessitate changes to implementation methodology. uDL does provide a generic attribute mechanism that can propagate arbitrary attributes to downstream implementation flows.

Time Abstraction

Clocked logic, especially in high-performance designs, is generally thought of as belonging to a pipeline. The pipeline performs a function, and that function is distributed in time amongst the stages of the pipeline, so that every stage contains a depth of logic which can be evaluated within the period of the clock. This division is a physical detail. In uDL, the behavioral model describes the function of the pipeline, and each behavioral statement is bucketed to a pipeline stage as a physical mapping.

This is straight forward for pipelines which simply perform a function based on inputs and output results some time later. Pipelined logic, however, interacts with state, contains feedback and feed-forward paths, and feeds into and out from other pipelines. It is possible to describe timing-abstract functionality of these situations. In fact, it is

possible to describe arbitrary logic in a timing-abstract way, while still preserving overall functionality that is cycle-accurate. The key is to define timing relationships between pipelines (and feedback/feed-forward within a pipeline) with relative alignment. Figure 3.A shows an example containing stall logic with two feedback paths with relative cycle deltas of 1 and 2, and an input into another pipeline with a particular alignment of the pipelines. After a timing fix (Figure 3.B) to alleviate broken input timing, the overall function is preserved because the logic functions and the relative deltas remain unchanged.

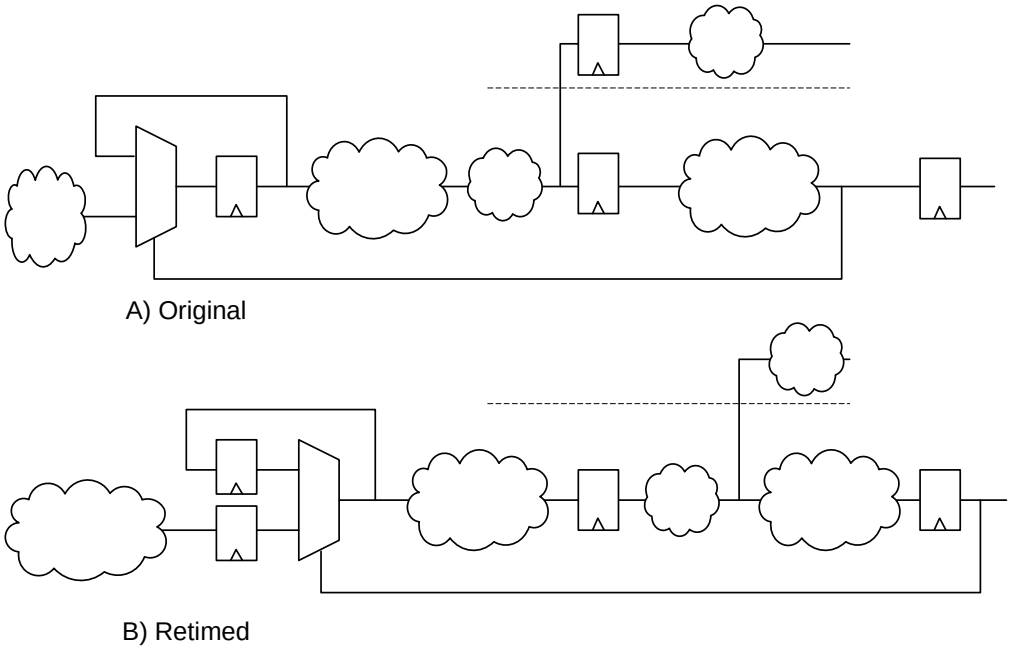

Figure 3: Example of Logic Retiming

Stage bucketing is entirely absent from the uDL behavioral model, and exists only as a physical mapping. Sequential elements are implicit, though they can be referenced to provide physical information such as attaching arbitrary attributes for downstream tools or bucketing them to physical partitions.

Physical Partitioning

A design is partitioned into a hierarchy of subcomponents to enable a divide-and-conquer approach to physical implementation. These subcomponents are often restricted to rectangles for ease of integration. So the boundaries are physically, not behaviorally, motivated. uDL creates a complete separation between the behavioral hierarchy and the physical hierarchy (though often the behavioral partition provides a general guideline for physical partitioning). As with pipeline stages, every behavioral expression is also bucketed to physical hierarchy. Logic can be replicated by bucketing it to multiple places in the physical hierarchy.

Encoding Abstraction

The data type of a uDL variable is defined as the set of values the variable may have (its range). These values may be integers or tokens. An encoding mapping defines the encodings of these values to physical signals. It is up to uDL compilation tools to generate the logic for each uDL expression appropriate for the encodings of its inputs and outputs. Logic synthesis tools tend to do a good job optimizing the type of combinational logic that would be generated to satisfy these encodings in most situations. As an example, an addition of two one-hot values becomes a shift operation, and it is up to the tool flow to generate an optimal shift. (If the tool flow does not generate acceptable results, the refinement mechanism, described in Section 4, can be used to provide further detail.)

This form of abstraction simplifies the behavioral model in several ways. For example, functions like encoders and decoders need not be present in the behavioral model. As another example, experiments can be done with state encodings for state machines to meet timing requirements without impacting the behavioral model.

Each expression understands its range based on the ranges of its inputs. So a variable's type can be implied. If an explicit range is specified, it can imply an assertion if the specification is a subset of what is implied. The assertion can be checked formally, or in simulation. A fair number of assertions seen in today's RTL are unnecessary or are simpler to capture in uDL, and many assertions that would be implicit in uDL are never coded in today's RTL due to the tedious and time-consuming nature of doing so. Examples of such assertions might be: checking that the 4-bit read index into a 10 entry array is less than 10; or checking that the array's read word line is one-hot (the read index and the read word line could actually be the same variable in uDL, differentiated only through physical mappings).

4. Refinements

The above mapping techniques will sometimes be capable of providing the level of physical detail necessary to implement a desired function. In other cases they will be insufficient, or awkward. In such cases, a generic refinement mechanism can be used. Logic requiring a more detailed representation can be bucketed (similar to pipeline stage or physical partition bucketing) for refinement. An alternate representation of the logic can be provided for use in implementation, while the behavioral model serves as the functional verification target. Equivalence of the two models is verified either via formal equivalence verification or via simulation where the refinement is built as a shadow model.

Refinements can be nested. Because the mechanism allows logic to be isolated and incrementally refined with nested refinements, each refinement step is generally small in scope, and it should generally be possible to verify refinements formally.

The utility of refinements can be broken into two categories. First, are scenarios where a block of logic is implemented very differently from its simplest behavioral expression. For example, a large first-in-first-out queue might be implemented using two register files (broken apart due to size) with a bypass path. Second, are scenarios where the pipeline stage or physical partitioning mappings must be applied to part of a single

expression. For example, an adder might be distributed across multiple cycles or physical partitions.

More generally, the refinement mechanism is an alternative-representation mechanism, where compilation options can select among the alternatives. It could be used to explore behaviorally-distinct design alternatives. It could also be used to provide abstraction above cycle-accuracy. Though not explored in our research, support for non-deterministic behavior is desirable. This can be introduced through free variables. An arbiter, for example, may provide fairness, but not correctness, and the arbitration choice can be left free to convey this. BlueSpec [2] allows the compiler to make such decisions. Various heuristics could be applied to the choice in simulation. A corresponding cycle-accurate refinement can provide true behavior and formal tools must be capable of proving conformance. The refinement mechanism could also provide performance-approximate models for estimating performance, as discussed in Section 7.

5. Data Flow

While RTL is purely an expression of the logic that implements a design, uDL has a higher-level understanding of the data and its flow through the pipelines of the design. [5] and [6] have shown in different ways that knowledge about pipelines and dataflow can be beneficial to formal analysis. BlueSpec [2] has shown that it can avoid bugs and enable synthesis of data flow control logic.

In uDL's data flow model, all logic is retimable and belongs to a pipeline (possibly a default and/or degenerate one). A pipeline may contain multiple partitions of logic, called "parcel logic", that process parcels (machine instructions or packets for example), generally one per cycle. Each parcel (or sub-parcel) has an independent indication of whether it is valid. In fact this is the real distinction among parcels/sub-parcels. A Boolean variable called "valid" can be assigned to define the valid condition for the parcel/sub-parcel. If unassigned, it is always considered valid. The valid variable does not correspond to a physical signal unless it is consumed.

Parcels can traverse multiple pipelines. For example, an instruction can be enqueued as the output of a fetch pipeline and read from the queue as the input to an issue pipeline. Parcel logic in this case, can be moved across the queue (if it has no interaction with other parcels), affecting the state values that are captured in the queue, similar to the way moving logic across pipeline stages affects the sequential elements that stage the parcel.

The data flow model, and knowledge of validity in particular has several potential benefits, including: implicit checking, safer and easier use of clock gating for power savings, improved visualization, and improved simulation speed, each described below.

Implicit checking: Logic corresponding to a valid parcel may not consume a variable of an invalid parcel. This implies a fair number of assertions and ensures that the values held by signals corresponding to invalid parcels are truly dont-care states. It also simplifies the coding of assertions as they need only apply for valid parcels. In particular, today's assertions must often be conditioned off during reset and for invalid parcels. Conditioning based on reset is unnecessary in uDL as there is no simulation of valid parcels during reset. (See Section 6.)

Clock gating: Clock gating is an important technique for saving power. Clock gating avoids driving a clock pulse to sequential elements in cycles where data does not need to propagate (or must be held). This is the case, when the values do not belong to valid parcels. Based on timing constraints of generating the gating function, the function that is implemented could be a subset of the cases that could be gated. The choice of gating function is restricted by physical constraints. Since the uDL model understands legal clock gating functions, it would be possible for tools to either synthesize clock gating logic within the constraints of timing (and area, and power), or to verify the choice of gating functions in hand-drawn schematics or uDL attributes.

Visualization: Significant time is spent in the debug process trying to understand simulations. Waveform viewers are one common type of visualization tool. Finding an activity of interest in a waveform view can be a bit like finding a needle in a haystack. A viewer that understands the uDL data flow model could filter out a significant amount of data based on knowledge of validity. It could represent data as meaningful integer and mnemonic values, and could group these as parcels. And this can all be automatic. Attempts to represent a design at this level today are highly customized and require a significant development and maintenance investment.

Simulation speed: All forms of abstraction offered by uDL represent opportunities for faster simulation. Refinements eliminate detail, as does the absence of physical partitioning. Encodings can be chosen by a compiler to optimize simulation speed, as can stage bucketing. Furthermore stage bucketing can be an enabler for multithreaded simulation. One of the challenges with partitioning logic for multithreaded simulation is the tight communication between partitions. Careful stage bucketing can potentially eliminate intra-cycle dependencies, back-to-back cycle dependencies, and so on. Additionally, a parcel can be modeled as a single data structure, and a pointer to it can be tracked through its pipeline(s) rather than staging the data of the parcel. Invalid parcels need not be allocated or simulated at all. Parcel logic can be grouped for both temporal and spatial locality. On a cautionary note, these opportunities also introduce irregularity, which results in branch mispredictions and cache misses, so these thoughts represent a careful balancing act for the simulator to target the simulation platform.

6. Mode Transitions

Much of the logic in a high-performance design is there for “special” modes of operation, in particular, to support reset, test, and debug capabilities. Retaining cycle accuracy in a model that includes these features can limit the ability to abstract away details of the primary functionality. Eliminating the need to model these modes in uDL will significantly improve the ability to model abstractly. Furthermore, this logic tends to be motivated heavily by physical constraints, so it is appropriate for the schematics to be their definition or for the logic to be synthesized from an understanding of the mode’s requirements.

Functional verification of these modes would have to be done on the gate-level model, where simulation is significantly slower. Fortunately, special modes do not generally require the breadth of coverage that is required of the primary functionality. And it is generally easier to divide the logic into isolated pieces. Verification must be

done not only of the function of the mode, but also of the transitions into and out of the mode.

At a minimum, it would be valuable to avoid modeling SCAN and reset logic in uDL. Various methodologies have been employed to date which eliminate SCAN from RTL [10]. For reset, the uDL model would understand the various levels of reset that are supported and the conditions that would trigger them. Each state would have a reset value and a reset level at which the reset value would be applied. The specific logic that assigns the reset values would not be specified. It is reasonable to imagine flows to synthesize such logic. It is also reasonable to imagine flows to verify the behavior in three-state, gate-level simulation. Each level of reset can be validated in simulation by considering the machine to be in an unknown state, simulating the reset flow, and checking that the correct reset state values were applied.

7. Performance Modeling

Performance modeling and design space exploration is generally done today in general-purpose programming languages like C++, often augmented with a library specifically for performance modeling. These languages are more appropriate for performance modeling than RTL as RTL does not support run-time parameterization, easy model configurability, and data collection. So the performance model is a separate model to develop, maintain, and verify against the RTL. This is a significant investment. Generally, there is no formal linkage between the RTL and a performance model. There is always question as to how closely the performance model matches RTL performance. For these reasons, RTL is often used in addition to performance modeling to evaluate the performance of a design. However, by the time performance feedback from RTL models is available it is costly to rework the design to address issues.

A few capabilities have been considered for uDL in support of using uDL for performance modeling. The refinement mechanism is one such capability for easy modeling and configuring of alternatives. Run-time parameterization is another. The ability to insert pipeline stages (at compile-time or runtime) is an important form of parameterization for studying the impact of latency on performance.

While there are benefits to targeted performance modeling solutions, the integration of performance modeling and “RTL” development would eliminate the second model and its associated costs. It would make the transition from performance modeling to design more graceful as some performance model components could be used or leveraged for design components. And it would enable easier functional verification model creation, as performance modeling components could be used to provide stimulus for designs under test. And lastly, and possibly most importantly, the performance model could now be synthesized into logic. This means it could be targeted for hardware emulation, to get dramatically improved performance, which is an area of much current research [7] [8].

8. Summary

uDL introduces several mechanisms for enabling more abstract cycle-accurate microarchitectural modeling. A separation is established between the behavioral model, which is the target of functional verification, and the details required for physical implementation of that model, with a verification path between the two. Pipeline timing, physical partitioning, and encoding are uDL's three main forms of correct-by-construction mappings to physical detail. Refinement adds a generic mechanism to abstract a design in ways that are not enabled by the other mechanisms, and these are verified either formally or in simulation.

Furthermore knowledge of the flow of data provides implicit assertion checking (as does the encoding mapping), and it is an enabler for future improvements in visualization, logic synthesis (especially of clock-gating and reset logic), and simulation speeds. There is a need for new innovation which will enable "special mode" logic to be absent from the abstract model. And lastly, uDL presents opportunities for unifying performance models and hardware models, simplifying the design process and enabling hardware emulation of performance models.

By bridging the gap between abstract modeling (at a cycle-accurate level) and implementation details, uDL presents a viable direction for the evolution of microarchitecture design over the next several decades. By enabling abstraction and stability in verification target models, it can help to tame the growth in verification effort that threatens the industry. It improves the ability to leverage designs and their verification collateral through multiple technology generations, which is of extreme importance for today's system-on-chip designs.

Acknowledgements

uDL brings together ideas found in many past efforts within Intel. Members of the working group exploring uDL were: Eugene Earlie, Sergey Bogdanov, Steve Hoover, Timothy Leonard, Gabriel Bischoff, Mark Tuttle, Peter Morrison, Armaghan Naik, and Emily Shriver.

References

[1] http://www.esterel-technologies.com

[2] http://www.bluespec.com

[3] P, Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware Design in Haskell," International Conference on Functional Programming, September 1998.

[4] E. Axelsson, K. Claessen, and M. Sheeran, "Wired: Wire-aware circuit design," Conference on Correct Hardware Design and Verification Methods, 2005.

[5] J. Higgins and M. Aagaard, "Simplifying the Design and Automating the Verification of Pipelines with Structural Hazards," ACM Transactions on Design Automation of Electronic Systems, October 2005.

[6] J. Cortadella, M. Kishinevsky, and W. Grundmann, "Synthesis of Synchronous Elastic Architectures," Design Automation Conference 43rd ACM/IEEE, 2006.

[7] M. Pellauer and J. Emer, "HAsim: Implementing a Partitioned Performance Model on an FPGA," Proceedings of the Fifth Annual Boston Area Architecture Workshop, January, 2007.

[8] D. Chiou, H. Sunjeliwala, D. Sunwoo, J. Xu, and N. Patil. "FPGA-based Fast, Cycle-Accurate, Full-System Simulators," Tech Report Number UTFAST-2006-01, Austin, TX, 2006.

[9] P. Mishra, A. Shrivastava, and N. D. Dutt, "Architecture Description Language (ADL)-driven SoftwareToolkit Generation for Architectural Exploration of Programmable SOCs," ACM Transactions on Design Automation of Electronic Systems, July 2006.

[10] S. Barbagallo, M. Lobetti Bodoni, D. Medina, F. Corno, P. Prinetto, and M. Sonza Reorda, "Scan Insertion Criteria for Low Design Impact," Proceedings of the 14th IEEE VLSI Test Symposium, 1996.

[11] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and A. Juan, "Asim: A Performance Model Framework," Computer, 35(2): pp. 68-72, 2002.