Timing-Abstract Circuit Design in Transaction-Level Verilog

Steven Hoover
Redwood EDA
Shrewsbury, MA, USA
steve.hoover@redwoodeda.com

***Abstract*—Given the complexity of modern integrated circuits, design reuse is essential, but current hardware description languages do not adequately address reuse challenges for many classes of design. Processor cores, as an example, are shaped by cycle-level interactions, and leveraging such designs into environments with different timing constraints requires retiming, repipelining, and microarchitectural changes. Making these changes at the register-transfer level requires significant rewriting. Abstraction is needed, but the abstractions of SystemC and OpenCL are more appropriate for loosely-coupled microarchitectural interactions.**

A *timing-abstract* modeling approach is presented that separates the concerns of behavior and timing. Timing-abstract behavior is specified within the context of pipelines, and logic within pipelines is assigned to pipeline stages as a matter of implementation detail. Sequential elements are generated by tools from the pipelined specification. Logic can be retimed easily, without the risk of introducing functional bugs, so design and verification effort can be focused on the behavioral changes required to retarget a design to a context with different timing constraints. As a secondary benefit, significantly less source code is required to specify register-transfer-level detail.

Keywords—electronic design automation; digital logic; circuit; hardware; software; language; compiler; pipeline; retiming; productivity

I. Introduction

Not long after the introduction of register-transfer-level (RTL) logic in the mid 1980's, there was a recognized need for languages that provided a higher level of abstraction in the design process. By the 1990's, research into higher-level modeling languages was very active [1][2], but, as new languages failed to gain broad adoption, optimism waned, and, even today, RTL remains the predominant design methodology. The need for better methodology, however, has continued to escalate under the pressures of Moore's Law [3]. Transistor counts have grown 25,000x, since the adoption of RTL in the mid-1980s, and clock frequencies have increased by a factor of 100. Over the past decade alone, despite significant advancements in electronic design automation, the effort to design a single chip has increased by nearly a factor of five [4], and today, the design of a single chip can occupy a team of hundreds of engineers for several years. A continuation in this trend is unsustainable.

The most significant productivity improvement over the past decade has come from the adoption of system-on-chip (SoC) design methodology [5]. Rather than designing full-chip RTL code from scratch, a full-chip model is assembled from intellectual property (IP) building blocks, delivered by other teams or other companies. This focus on modularity and reuse is essential to managing complexity.

SoC methodology, however, faces significant challenges and limitations when done at the register-transfer level. When designs were monolithic, RTL code was written for one specific implementation with specific physical constraints in mind. The clock speed, performance target, floorplan, and power budget became deeply reflected in the RTL code. A subsequent generation of a design might implement a similar microarchitecture, but the RTL code would be largely rewritten to meet new physical design constraints. A reusable IP block, on the other hand, cannot be designed with the actual physical constraints of any one implementation in mind. It must be designed with assumptions about its constraints, and it cannot be easily leveraged outside of these assumptions. Among these assumptions, the depth of logic that can fit within a clock period most-significantly influences RTL IP. This is dictated by the clock speed and the technology, which typically differ between implementations.

For these reasons, it is desirable to express designs at a higher level of abstraction and to allow tools to synthesize the abstract designs into gates based on design constraints. This is the goal of high-level synthesis (HLS). Various high-level modeling approaches have been explored along with their potential to synthesize to gates [1][2][16]. Industry momentum currently centers around synthesizing hardware from C/C++-based models using SystemC [6][7][8][9] or OpenCL. Several other languages are available that provide incremental improvements in abstraction, including Bluespec SystemVerilog [10][11] and Chisel [12]. These remain explicit about sequential elements (flip-flops and latches) and therefore do not provide sufficient abstraction to avoid redesign.

SystemC and OpenCL have gained momentum because of

the significant role that software modeling plays in the design and verification of hardware. Synthesizing software into hardware is a challenging task that has taken the industry a few decades to bring to fruition. Tools must bridge the gap from software, which is fundamentally sequential, to hardware, which is fundamentally parallel. Creativity has been applied to the challenge of providing constructs and concurrency models appropriate for hardware within a software language. Synthesis from software is an important enabler for several activities. These include converting existing software to run as hardware, developing code that can run as either software or hardware, enabling software developers to develop hardware, and cleanly integrating hardware models with software and verification models. For the specific task of modeling hardware, however, the use of C++ language semantics presents complications for both designers and tools.

For certain design spaces, C/C++-based HLS tools successfully enable the development of flexible hardware models that can be implemented under different design constraints. HLS is good for constructing optimized computational pipelines such as those found in digital signal processors and graphics processing units, and it is good for timing-insensitive communication channels.

Control-intensive designs, however, have complex cycle-level interactions that require fundamental microarchitectural changes when design constraints change. As an example, a central processing unit (CPU) must contend with structural, control, and data hazards. It must schedule and slot instructions around register dependencies and resources, it must provide register bypass paths, and it must make predictions and recover from mispredictions. At a higher frequency, a new pipeline stage and a new bypass path might be required, with potential impact to these mechanisms. Though there has been research toward automating certain microarchitectural transformations [16], generalized fully-automated optimizations are not in sight. Designers make tradeoffs with workloads in mind, and this information is not available to HLS tools.

This paper describes a modeling approach where cycle-level detail is provided within the context of a *timing-abstract* behavioral model. This approach addresses the conflicting goals of having explicit cycle-level detail in the source code and having the ability to change details without disruptive rewrites. Section II of this paper introduces a pipeline construct that provides a timing-abstract context for specifying cycle-level details. Section III describes how complex logic with feedback and feed-forward paths, as well as interactions between pipelines, can be expressed with cycle-level detail. The potential for cycle-approximate and untimed models is explored in Section IV. Section V addresses the need to control sequential elements, or "sequentials," and clocking networks, which are no longer explicit in the model. Section VI presents analysis of the impact of microarchitectural changes and the compactness of the timing-abstract code compared with SystemVerilog RTL. Finally, the paper is summarized in Section VII.

II. Timing-Abstract Pipelines

Pipelines are a fundamental part of any high-performance digital integrated circuit design. Fig. 1 illustrates a simple pipeline that is performing a Pythagorean theorem calculation ($c = \text{sqrt}(a^2 + b^2)$).

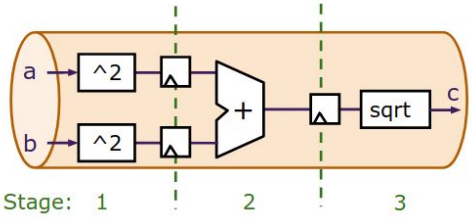

Fig. 1. Pipelined Pythagorean theorem calculation diagram.

Various coding styles are possible with SystemVerilog. The SystemVerilog expression of this pipeline shown in Fig. 2 adheres to one recommended coding style in which sequentials are kept separate from combinational logic, and signals are named to reflect their pipeline stages.

```
// Declarations
logic [3:0] a_a1;
logic [7:0] a_sq_a1, a_sq_a2;
logic [3:0] b_a1;
logic [7:0] b_sq_a1, b_sq_a2;
logic [4:0] c_a3;
logic [8:0] c_sq_a2, c_sq_a3;

// Stage 1 Logic
always_comb begin
   a_sq_a1[7:0] = a_a1[3:0] ** 2;
   b_sq_a1[7:0] = b_a1[3:0] ** 2;
end
// Stage 1 Flops
always_ff @(posedge clk) a_sq_a2[7:0] <= a_sq_a1[7:0];
always_ff @(posedge clk) b_sq_a2[7:0] <= b_sq_a1[7:0];
// Stage 2 Logic
assign c_sq_a2[8:0] = a_sq_a2 + b_sq_a2;
// Stage 2 Flops
always_ff @(posedge clk) c_sq_a3[8:0] <= c_sq_a2[8:0];
// Stage 3 Logic
assign c_a3[4:0] = sqrt(c_sq_a3);
```

Fig. 2. Pipelined Pythagorean theorem calculation in SystemVerilog.

This same design is coded in a timing-abstract representation in Fig. 3, using a language extension of SystemVerilog, called Transaction-Level Verilog, or TL-Verilog. TL-Verilog was developed originally at Intel Corporation, driven in large part by the author, and is now an evolving open language standard.

```
|calc
   @1
      $aa_sq[7:0] = $aa[3:0] ** 2;
      $bb_sq[7:0] = $bb[3:0] ** 2;
   @2
      $cc_sq[8:0] = $aa_sq + $bb_sq;
   @3
      $cc[4:0] = sqrt($cc_sq);
```

Fig. 3. Pipelined Pythagorean theorem calculation in TL-Verilog.

Several TL-Verilog constructs are introduced in Fig. 3.

Pipesignals: Identifiers such as `$aa` in Fig. 3, are termed *pipesignals*. A pipesignal represent a signal and all staged versions of that signal. The SystemVerilog code has two signals corresponding to `$aa_sq`, a version for stage 1 and a version for stage 2.

Pipelines: TL-Verilog introduces a scope construct for pipelines -- `|calc` in Fig. 3. These pipelines are free-flowing without back-pressure. They provide a mechanism for abstracting time.

Pipestages: TL-Verilog also introduces a scope construct for pipeline stages, or *pipestages*, within pipelines, such as `@1`.

Fig. 4 illustrates how the timing-abstract representation differs from the RTL representation. Flip-flops are implied where pipesignals cross pipestage boundaries.

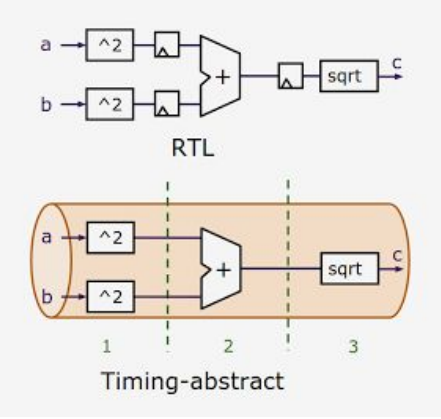

Fig. 4. RTL versus timing-abstract.

Strictly speaking, the timing-abstract behavioral model does not include the pipestage specifications. These have no impact on the overall behavior of the model. A timed signal, such as the pipeline output `$cc` at stage `@3`, exhibits an identical waveform regardless of the staging of the logic that produces it. Pipeline staging is considered to be a physical attribute, or *augmentation*, describing an implementation of the behavioral model. A timing-abstract model, together with timing augmentation is termed a *timing-augmented* model.

Logic retiming (described in [13]) at a statement level is a simple matter of changing timing augmentation and does not carry the risk of introducing bugs. Even if a pipesignal is consumed in an earlier stage than it is produced, this is a reflection of an infeasible physical implementation and does not reflect upon the validity of the timing-abstract behavioral model. In TL-Verilog, every logic statement belongs to a pipeline and can therefore be retimed. (For convenience, a default pipeline and pipestage are assumed for statements that are not explicitly scoped.)

A few other aspects of TL-Verilog that are apparent in Fig. 3 warrant explanation. A pipesignal's type, such as `[31:0]`, is included in its assignment statement; a separate type declaration is not required. Assignment statements use Verilog `assign` statement syntax, but the `assign/always_comb` keyword is not required, and pipesignals are generally used in place of signals.

III. Pipeline Interactions

Expressing cycle-level detail in the source code is most important for designs with a large number of feedback and feed-forward paths and interactions among pipelines. Fig. 5 shows an example in the context of a CPU instruction execution pipeline that includes such interactions. The shaded multiplexer (mux) is selecting an operand for an arithmetic logic unit (ALU). The four sources to the mux, ordered from top to bottom in both Fig. 5 and Fig. 6, are:

1. the result from the previous instruction, one stage ahead in the instruction pipeline
2. a register value from the register file
3. an immediate value embedded in the instruction
4. a value being returned from memory, from the `|mem` pipeline.

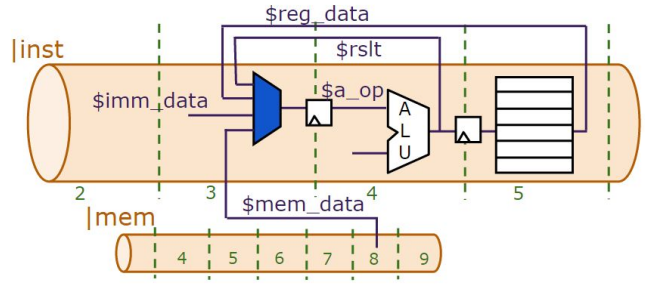

Fig. 5. Operand mux logic.

```
|inst
   @3
      $op_a[63:0] =
         ($src_a == BYP) ? >>1$rslt     :
         ($src_a == REG) ? >>2$reg_data :
         ($src_a == IMM) ? $imm_data    :
         ($src_a == MEM) ? /top|mem>>5$mem_data :
                           64'b0;
```

Fig. 6. Operand mux code.

This example introduces:

Pipeline Alignment: The `>>` (*ahead*) or `<<` (*behind*) syntax, or *pipeline alignment* specifier, provides the stage of a

referenced pipesignal relative to the stage of the assignment statement.

As will be discussed further, the use of relative alignments is key to enabling safe retiming of logic where transactions do interact. It also clarifies the nature of the interaction. Each mux-source expression demonstrates a different alignment.

1. Source 1 (`>>1$rslt`) is the result from the previous instruction, which is in stage 4 of the pipeline. Since it is consumed by the mux in stage 3, the reference to `$rslt` is given an alignment of (4 - 3), or `>>1`. `>>1` references the transaction that is one cycle ahead in its pipeline.
2. Source 2 (`>>2$reg_data`) is from the register logic in stage 5, and so has an alignment of (5 - 3), or `>>2`, suggesting that the source reflects the transaction that is two cycles ahead.
3. Source 3 (`$imm_data`) is immediate data from the instruction flowing through the pipeline, used by the instruction itself. Therefore, there is no offset for this interaction (3 - 3 = 0), and an offset specification of zero is assumed. This can be referred to as a *naturally-aligned* reference as it reflects the natural flow of the pipeline. (All references in the Pythagorean theorem example were naturally-aligned.)
4. Source 4 (`/top|mem>>5$mem_data`) is from a different pipeline altogether. `/top|mem` provides the pipeline scope of the referenced pipesignal, `$mem_data`, and the pipestage is given by the alignment, `>>5`, which gives us a pipestage of 3 + 5, or 8.

To illustrate logic retiming in the face of pipeline interactions, Fig. 6 modifies Fig. 5 to address a scenario where a timing path into the mux exceeds the cycle time. The issue is addressed by retiming the mux to cycle 4.

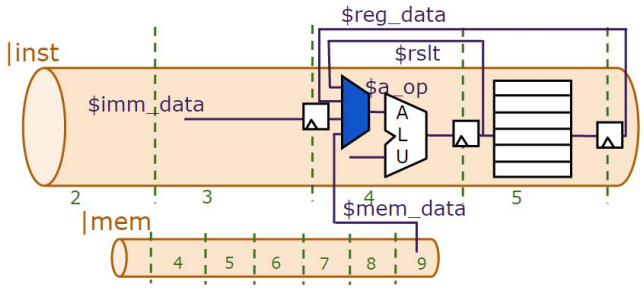

Fig. 7. Operand mux moved to stage 4.

The impact of this change is that each mux source is taken from a delayed version of its prior input, and the mux output no longer requires staging. Implementing this simple change in RTL code is non-trivial and rather bug prone. Two (64-bit-wide) flip-flops must be added and one removed; one signal must be added and one removed. Signal names in the mux expression must be change to reflect their new stage. In TL-Verilog, however, the change is trivial. The stage scope of the mux is simply changed from `@3` to `@4`, and, since the behavioral model has not been changed, functionality is preserved. The use of signal references with relative stage alignments has preserved the ease of logic retiming in the face of pipeline interactions.

IV. Timing-Precision

It is not always necessary to be precise about timing in the source code. Logic synthesis and HLS tools are quite capable of retiming a design to optimize its implementation. Where they can be allowed to do so, the effort of timing closure can be significantly reduced. Reflecting a degree of reality in the source code however does enhance a logic designer's ability to reason about cycle-level interactions. It also presents synthesis tools with less work to do with each synthesis run. An approximately-timed model can provide a good balance, especially as it eliminates the need to break up single logic statements that would physically span cycle boundaries.

In some cases, precise timing is called for. Some design teams, especially those implementing very aggressive designs, employ tools and flows that manipulate sequentials after synthesis based on pre-synthesis knowledge of the sequentials. These flows can include scan chain insertion and clock network generation. When such flows are employed synthesis cannot be permitted to retime logic. Precise timing is also valuable for correlating the physical design back to the source code. Timing paths, for example are reported from one sequential element to another. Even if allowing synthesis retiming is feasible, it can be reasonable to achieve a certain level of timing closure without synthesis retiming and enable it as timing targets are narrowed.

At varying levels of detail, retiming and repipelining will continue to be a significant part of the process of targeting an IP block to a particular implementation. Safe and easy retiming results in fewer builds, fewer simulations, fewer bugs, and less debug effort. Regression testing is unnecessary if timing changes can be shown at the source-code level to have no impact on behavior. Physical timing closure effort can be substantially reduced if physical designers or automated tools can be empowered to apply timing changes to the source code with minimal or no involvement from the logic designer.

Though this paper focuses on cycle-based design, in which all sequentials are flip-flops, phase-granular timing augmentation is also possible. Staging can be specified on the alternate phase of the clock, and latch-based logic can be implied. Retiming of level-sensitive circuits is explored in [14]. The transparency of latches avoids the need to partition logic to precise cycle boundaries and avoids some overhead from setup and hold time requirements. Phase-based design generally suffers a small area penalty, more so in field-programmable gate arrays, where latches may not be available. Furthermore, since phase-based pipelines have roughly double the number of sequentials to manage, design and maintenance costs are a significant deterrent. Using timing-augmentation, the design overhead is minimal. The remainder of this paper, however, focuses on cycle-based design to retain focus on the core contribution.

V. Controlling the Clock Network and Sequentials

When introducing abstraction, it is important to understand the degree of control that is sacrificed. Though the timing-augmented models are precise about the existence of every sequential, it is up to tools to create these sequentials, as well as their controlling inputs. While RTL code can be explicit about reset, enable, clock, and scan inputs, these require special consideration for timing-augmented models.

Sequentials with synchronous reset support simply include combinational logic, such as `AND NOT reset`. This logic can be provided explicitly in the timing-abstract models and synthesis tools are capable of choosing sequential logic cells with reset inputs where appropriate. Since timing abstraction deals only with synchronous logic, asynchronous reset conditions are outside the scope of timing-abstract modeling. So, no special language support is required to support reset sequentials.

Scan logic is rarely explicit in RTL in modern methodologies. It is generally inserted automatically. Partial-scan methodologies might require explicit identification of the sequentials to which to apply scan, and regardless of whether the source code is RTL or timing-augmented, this remains possible only if synthesis retiming is not permitted. Scan is not an obstacle for timing-abstraction.

Enable sequentials use their enable inputs to internally generate conditioned versions of the clock with pulses only for enabled cycles. Power is saved by the removal of transitions on the clock as well as the prevention of transitions on the outputs. An enable flip-flop has the same behavior as an unconditioned flip-flop that has its output recirculated when the enable condition is not asserted. As with synchronous reset sequentials, enable sequentials can also be inferred by modern logic synthesis tools, so again no special provision is required (though TL-Verilog does provide an explicit construct for recirculation).

Lastly, clock gating is similar to clock enabling, but the conditions are applied further upstream in the clocking network, and conditions influence multiple sequentials. Clock gating saves more power than clock enabling by avoiding distribution of the unneeded clock pulses. Gating conditions must be explicit in the source code, and, depending upon the tool flow in use, it may be necessary for the generation of gated clock signals to be explicit to some degree as well. Information must be exposed in timing-augmented models that enables generation of high-quality clock-gating networks.

In TL-Verilog this need is addressed by exposing information as to when logic expressions are computing meaningful results. This information determines the need for clock pulses. To illustrate this, Fig. 8 is a modification of Fig. 3. It introduces a *when condition* scope, **`?$valid`**, that applies to all the logic of the pipeline, across all stages. It indicates that the pipeline logic is valid only when **`$valid`** (whose assignment is not included in the code snippet) is correspondingly asserted. When a pipesignal's value is known to be invalid, it is not necessary to propagate it through sequentials. Thus the various staged versions of **`$valid`** provide an enable or gating condition for the sequentials. It is up to the tools processing TL-Verilog code to provide appropriate specification of the clock gating network from this condition information. Clock-gating logic, which is typically a disruptive afterthought, can be in place in a design from the start.

```
|calc
   ?$valid
      @1
         $aa_sq[7:0] = $aa[3:0] ** 2;
         $bb_sq[7:0] = $bb[3:0] ** 2;
      @2
         $cc_sq[8:0] = $aa_sq + $bb_sq;
      @3
         $cc[4:0] = sqrt($cc_sq);
```

Fig. 8. Pythagorean theorem with validity.

When conditions are motivated not only by the need for control over the clocking network but also from a functional modeling perspective. Knowledge of validity enables detection of inadvertent consumption of invalid signal values. This can be implemented by generating assertions or by modeling invalid states as don't care (X) values, which will propagate. Conveying invalidity is useful for debug activities as well. In Fig. 7, the pipeline flow of the Pythagorean theorem example is clear because of the X states in the waveform, and the meaningful transactions are easily identified.

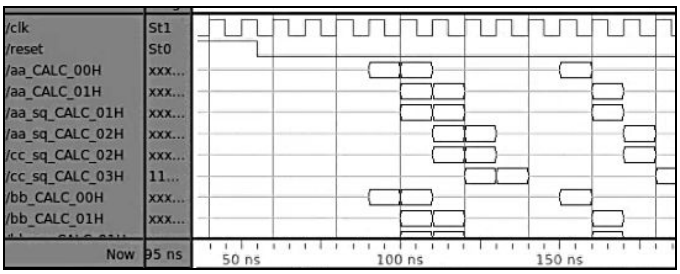

Fig. 9. Pythagorean theorem pipeline trace.

In summary, all necessary control over sequential elements is retained in timing-augmented TL-Verilog models. *When conditions* are a lightweight mechanism to convey design intent which provides clock gating or clock enabling, enables additional checking, and simplifies debug.

VI. Results

This section compares TL-Verilog models to equivalent SystemVerilog models and characterizes code in order to isolate the impact of timing abstraction. First a contrived long-division example is presented and analyzed in its completion. Analysis includes a breakdown of TL-Verilog and generated SystemVerilog code as well as analysis of the impact of a repipelining change. Subsequently, statistics are shared from three industry examples that were converted by

hand from Verilog/SystemVerilog to TL-Verilog. In all cases, Redwood EDA's SandPiper™ code generator was used which reads in TL-Verilog code and produces SystemVerilog code.

The long-division example in Fig. 10 computes A/B to four fractional hexadecimal digits. A and B are each a single hexadecimal digit, where A < B. In each of four successive cycles, a new digit of the quotient is calculated. This calculation is placed within a pipeline with control logic in stage 0 and the calculation in stage 1.

```
|calc
   @0
      // Carry A and B from previous iteration.
      $aa[3:0] = $in_valid ? $aa_in : >>1$aa;
      $bb[3:0] = $in_valid ? $bb_in : >>1$bb;
      // Valid iteration for this pipeline?
      $calc_valid = $reset ? 0 :
           $in_valid || (>>1$calc_valid && (>>1$iteration != 3));
      // Track which iteration we're processing, 0-3.
      $iteration[1:0] = >>1$calc_valid ? (>>1$iteration + 1) : 0;
   // Main calculation. Computes next hex digit of quotient (Q).
   ?$calc_valid
      @1
         // Remainder for this iteration, so, A, R1, R2, R3.
         $remainder[3:0] = $in_valid ? $aa : >>1$next_remainder;
         // The digit of the quotient, computed in this iteration.
         $quotient_digit[3:0] = {$remainder, 4'b0} / $bb;
         // The next value of $remainder, so R1, R2, R3, R4.
         $next_remainder[3:0] = {$remainder[3:0], 4'b0} -
                                ($quotient_digit * $bb);
         // Accumulate $quotient_digit by shifting into $quotient.
         $quotient[15:0] = {>>1$quotient[11:0], $quotient_digit};
```

Fig. 10. Long Division TL-Verilog Code (Single-Cycle)

Fig. 11 provides a breakdown of both the TL-Verilog code and the generated SystemVerilog code. Though the SystemVerilog code is machine generated, industry data will provide evidence that generated code reasonably approximates hand-generated code, following the chosen industry-recommended coding conventions.

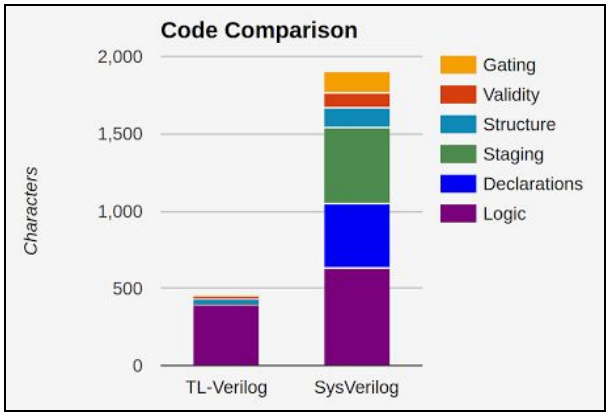

Fig. 11. Divider code breakdown.

The data shows there is very little in the TL-Verilog code beyond assignment statements. The timing abstraction constructs for pipeline and pipestage scope account for very little code and provide context to generate staging flip-flops and clock gating logic. Though the SystemVerilog code is too large to include, TABLE I. provides examples of both TL-Verilog and SystemVerilog code in each category.

TABLE I. Code Categories by Example

	TL-Verilog	*SystemVerilog*
Clock Gating	-	`clk_gate gen_Clk_F_calc_valid_CALC_02H(Clk_F_calc_valid_CALC_02H, clk, calc_valid_CALC_01H, 1'b1, 1'b0);`
Validity	`?$calc_valid`	`` `WHEN(calc_valid_CALC_01H) ... ``
Code Struct.	`|calc`	`always_comb, begin, end, ...`
Staging	`@2`	`always_ff @(posedge Clk_F_calc_valid_CALC_02H) Result_CALC_02H[15:0] <= Result_CALC_01H[15:0];`
Decl. Logic	-	`logic [3:0] Aa_CALC_01H;`
	`$iteration[1:0] = >>1$calc_valid ? (>>1$iteration + 1) : 0;`	`assign CALC_iteration_a0[1:0] = CALC_calc_valid_a1 ? (CALC_iteration_a1 + 1) : 0;`

The TL-Verilog code, while preserving RTL detail and separating the concerns of timing and behavior, reduces code size substantially. Excluding comments and whitespace, the TL-Verilog code is smaller than the generated SystemVerilog code by a factor of 3.4. To a close approximation, the changes in the Declarations and Code Structure categories represent syntactic benefits of TL-Verilog, while the other changes reflect timing abstraction. Excluding the syntactic categories, the estimated reduction from timing abstraction for this example is 3.0. Research has shown a correlation, perhaps a superlinear one, between code size and bugs [15].

Reducing IP block development time, however, is not the primary goal of timing abstraction. The primary goal is to increase IP reuse by simplifying and reducing bugs in the process of optimizing IP blocks for specific implementations. A scenario is analyzed, next, in which the long division code is leveraged in a design that is running at a higher clock frequency. The calculation of each iteration involves a division, a multiplication, and a subtraction, which, in this scenario, no longer fit within a clock period. The multiplication and subtraction are moved to a new cycle, and successive iterations must be two cycles apart, not one. To maintain an average throughput of one result every four cycles, this redesign permits the interleaving of calculations on even and odd cycles. The resulting code is shown in Fig. 12, with modifications highlighted.

There were two changes applied, one behavioral, and one only impacting the implementation of the behavior. The behavioral change was the change from a single cycle per iteration to two cycles per iteration. In other words, the alignment of one iteration to the next changed from one cycle

(**>>1**) to two (**>>2**). This alignment change can be seen eight places in Fig. 12. Had the original code been designed in anticipation of this scenario, it could have easily been parameterized to support any alignment. The second change is the retiming of logic within the pipeline. This change is implemented with the addition of the ***@2*** line (which could also have been parameterized).

```
|calc
   @0
      // Carry A and B from previous iteration.
      $aa[3:0] = $in_valid ? $aa_in : >>2$aa;
      $bb[3:0] = $in_valid ? $bb_in : >>2$bb;
      // Valid iteration for this pipeline?
      $calc_valid = $reset ? 0 :
          $in_valid || (>>2$calc_valid && (>>2$iteration != 3));
      // Track which iteration we're processing, 0-3.
      $iteration[1:0] = >>2$calc_valid ? (>>2$iteration + 1) : 0;
   // Main calculation. Computes next hex digit of quotient (Q).
   ?$calc_valid
      @1
         // Remainder for this iteration, so, A, R1, R2, R3.
         $remainder[3:0] = $in_valid ? $aa : >>2$next_remainder;
         // The digit of the quotient, computed in this iteration.
         $quotient_digit[3:0] = {$remainder, 4'b0} / $bb;
      @2
         // The next value of $remainder, so R1, R2, R3, R4.
         $next_remainder[3:0] = {$remainder[3:0], 4'b0} -
                                ($quotient_digit * $bb);
         // Accumulate $quotient_digit by shifting into $quotient.
         $quotient[15:0] = {>>2$quotient[11:0], $quotient_digit};
```

Fig. 12. Long Division TL-Verilog Code (Two-Cycle)

The number of lines and the number of characters of resulting change to the TL-Verilog and to the SystemVerilog produced by the SandPiper code generator are provided in Table II (excluding comments and whitespace). The SystemVerilog code, which is now 6.4 times the size of the TL-Verilog code, contains new flip-flops, new signal declarations, and two new gated clocks. Also, since the generated SystemVerilog signals reflect their pipestages, assignment statements that have been moved to new pipestages are impacted.

TABLE II. CODE IMPACT OF CHANGES

	TL-Verilog	***SystemVerilog***
Lines changed/added	7	34
Chars changed/added	30	1122

It is important to evaluate real-world scenarios as well. Fig. 13. shows code-size data from three stable, real-world examples of Verilog or SystemVerilog modules that were converted by hand to TL-Verilog. No data is currently available to analyze real-world IP block modifications, but these three examples provide meaningful code-size comparison with real-world hand-coded modules. Note that source lines left as Verilog or SystemVerilog were not included in the data. Most notably, this includes module interfaces.

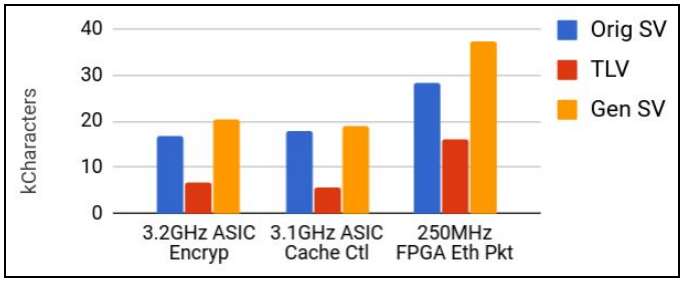

Fig. 13. Real-World Designs

While significant code reduction was achieved, no detail was lost. Logic synthesis was run for two of the designs, and results demonstrated consistency with the original code. The original SystemVerilog of the first two cases adhered to a coding style similar to the generated SystemVerilog code. Clock-gating and X injection were introduced in the first case and improved in the second case in the process of converting. These aspects of the generated code provide value beyond the hand-written SystemVerilog and account for some of the code growth between the generated and original SystemVerilog. The third case shows less code reduction. The hand-coding style in this case was a denser style than that of the generated SystemVerilog. This style is common of FPGA designs and less common among high-frequency designs. It is also typical to see less reduction at lower clock frequencies as there are fewer sequentials to manage.

The resulting pipelined expression of these designs exposed some benefits. For the third case, the designer was able to easily remove one of three stages from the main pipeline after conversion, which was reportedly a valuable improvement that would have been impractical to implement in the Verilog source code. One conceptual bug was known to be eliminated in the course of the conversions as a natural outcome of cleaner pipeline expression, though it is unknown whether this bug could manifest in the context of its full-chip model.

VII. CONCLUSION

The described modeling techniques expose a timing-abstract behavioral model that provides context for specifying cycle-level timing as a physical implementation detail. A pipeline construct provides the timing-abstract context, and a pipestage construct provides timing information. Expressions reference signals with an *alignment* that specifies a relative pipestage offset. Expressions in pipeline and pipestage context using relative references can be safely retimed without impacting functionality. Sequential elements are implied. Clock gating for these sequentials is derived from knowledge of the *validity* of pipelined computations. Validity information also results in cleaner modeling, error checking, and easier debug.

The most impactful consequence of these techniques is an improved ability to reuse IP blocks in contexts with different timing constraints. Control-intensive designs, where the

microarchitecture is a reflection of cycle-level considerations, call for reasonably precise modeling of timing. By isolating timing from behavior, and by enabling tools to manage the sequentials, microarchitectural changes motivated by repipelining can be made with minimal effort and risk.

Acknowledgment

The author would like to thank the many folks at Intel Corporation who contributed to this work and opened the technology to the world. The author also thanks a growing community of open-source supporters.

References

[1] P. Gammie, “Synchronous digital circuits as functional programs,” ACM Computing Surveys (CSUR), 46(2):21, November 2013.

[2] G. Berry and G. Gonthier. “The Esterel Synchronous Programming Language: Design, Semantics, Implementation,” Science of Computer Programming, vol. 19, n°2, pp. 87-152, 1992.

[3] D. Brock, ed., Understanding Moore's law : four decades of innovation. Philadelphia, Pa: Chemical Heritage Press, 2006.

[4] R. Collett and D. Pyle, “What happens when chip-design complexity outpaces development productivity?,” McKinsey on Semiconductors, pp. 24-33, Autumn 2013.

[5] P. Ashenden, ed., J. Mermet, ed. and R. Seepold, ed., System-on-Chip Methodologies & Design Languages, Springer Science & Business Media, March 2013.

[6] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel and D. Stroobandt, “An overview of today’s high-level synthesis tools,” Design Automation for Embedded Systems, vol. 16, no. 3, pp. 31-51, September 2012.

[7] R. Gupta, F. Brewer, “High-level synthesis: A retrospective,” High-level Synthesis, pp 13-28, Springer, 2008.

[8] K. Marquet, B. Karkare and M. Moy, "A theoretical and experimental review of SystemC front-ends", Proc. Specification & Design Languages (FDL), pp.124 -129, 2010.

[9] 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 archive, vol. 30, iss. 4, pp. 473-491, April 2011.

[10] D. Rosenband, J. Schwartz and Arvind, “Modular scheduling of guarded atomic actions,” 41st Design Automation Conference (DAC), 2004.

[11] F. Gruian and M. Westmijze, “VHDL vs. Bluespec system verilog: a case study on a Java embedded architecture,” Proc. of the 2008 ACM Symposium on Applied Computing, pp. 1492-1497, March 2008.

[12] J. Bachrach et al, “Chisel: Constructing Hardware in a Scala Embedded Language”, Proc. of the 49th Design Automation Conference, June 2012.

[13] C. E. Leiserson, F. Rose, and J. B. Saxe, “Optimizing synchronous circuitry by retiming,” Proc. of the 3rd Caltech Conference on VLSI, March 1983.

[14] B. Lockyear and C. Ebeling, “Optimal Retiming of Multi-Phase, Level-Clocked Circuits,” Advanced Research in VLSI and Parallel Systems: Proc. of the Brown/MIT Conference, 1992, pp. 265-280.

[15] F. Brooks, “The Mythical Man-Month, Anniversary Edition,” Addison-Wesley, 1995.

[16] E. Nurvitadhi, “Automatic Pipeline Synthesis and Formal Verification from Transactional Datapath Specifications” (Doctoral dissertation), Retrieved from https://users.ece.cmu.edu/~jhoe/distribution/2010/nurvitadhi.pdf, 2010.