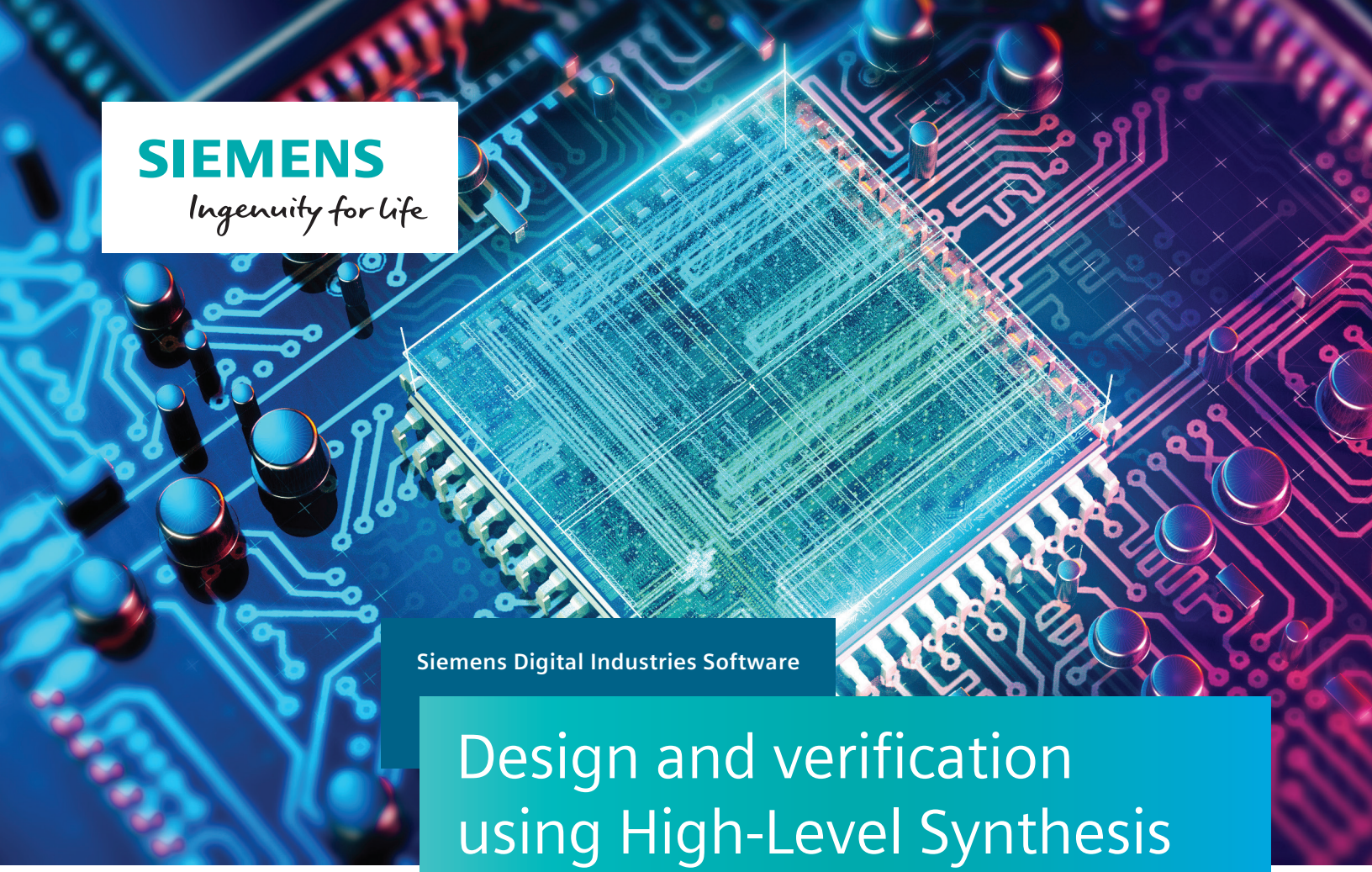

SIEMENS
Ingenuity for life

Siemens Digital Industries Software

Design and verification using High-Level Synthesis

Executive summary

The adoption of High-Level Synthesis (HLS) has been driven by the need to tackle growing verification costs in traditional RTL design flows. This paper presents an overview of design, optimization and verification using HLS. It also outlines some of the requirements for HLS design to fit into existing design and verification flows and ways in which such flows might be adapted as HLS is more widely deployed.

Andres Takach
Siemens Digital Industries Software

Contents

Introduction

High-Level Synthesis (HLS) has been in use in industry for a number of years. Its adoption has been driven primarily due to the advantages that raising the level of abstraction of design has on reducing the ever increasing costs of functional verification.

Adoption of HLS has been most widespread for the creation of new complex subsystems of SoC designs where time to market is an important consideration. Often such new subsystems are driven by new and evolving standards such as standards for wireless communication, image and video encoding. One of the recent publicly available successes of HLS is in the design of the VP9 video decoder[1]]. Video decoders for the H.264 standard and its successor the HEVC[2] standard have also been designed using HLS. It is worth noting that HLS has enabled more aggressive schedules in incorporating more capabilities (e.g, profiles in video decoders) than it would have been possible with manual RTL design methodologies. Users that have adopted it now view HLS as important to their competitiveness in the market. The success on such complex blocks is also driving HLS to be applied to blocks that have existing RTL IP, by writing synthesizable high-level descriptions for them. This is done to get better hardware for new technology nodes and to get the verification benefits of raising the level of abstraction.

High-Level Synthesis tools currently used in industry use a specification of the behavior or algorithm written in C++ or SystemC and generate RTL specifications that can be used in existing design flows. Throughout this paper, the term C++ specification will be used to cover both pure C++ and SystemC specifications. Section II provides an overview of important considerations when coding the C++ specification. Section III provides an overview of the micro-architectural high-level controls that are provided in HLS to generate an RTL specification that meets the design goals.

Traditional design and verification methodologies based on manual creation of the RTL has continued to evolve over the years and there is an expectation that HLS works within and supports existing RTL design methodologies. In addition there is an expectation that more tools and methodologies should be developed around the high-level specification in C++. Sections IV, V and VI cover some of the ongoing challenges and trends on the way HLS is getting deployed and how design methodologies may evolve around HLS.

While verification has been the main driver for adopting HLS, the need to design within certain power budgets is increasingly important. The highest power saving are obtained at the design decisions taken at the highest abstraction levels. Exploration of the design space can lead to significant power reductions as the best architecture is not usually evident. While an existing RTL IP block can be reused for a smaller technology node, the best results are obtained by tailoring the architecture to the target technology node.

A standard power optimization used for RTL is clock gating. During the HLS process, sequential analysis is done that enables it to reduce the conditions under which registers are enabled. By doing so, the switching activity of datapath may also be reduced, and HLS generates the RTL so that downstream synthesis flows can readily perform clock gating to reduce power. Section VIII presents results obtained on power reduction with this low-power HLS feature.

Coding C++ for synthesis

In general, it is best to start from the simplest and most compact code rather than code that has been optimized for software execution. It is also important to understand the hardware implications of different C++ constructs. There are two main areas to consider to create the C++ input that captures hardware intent:

- Perform numerical refinement if necessary. Floating point variables need to be replaced by bit-accurate fixed-point or floating-point variables tailored with the minimal bitwidth characteristics that preserve the numerical performance of the algorithm. For example, the AC Datatype package[3] provides integer, fixed-point and floating-point and complex datatypes to facilitate making the numerical refinement while keeping the abstraction level of the description.
- Consider how arrays are accessed and re-write the description to capture the memory architecture that is required to minimize hardware. For example, image applications go over an image by operating on a one dimensional or two dimensional window whose center point slides over consecutive pixels. A window C++ class can encapsulate the buffering of array accesses for the window operations. Array accesses are thus minimized resulting in a much better hardware implementation than from a generic description that has not been written for that hardware intent.

Modelling of concurrent communicating processes (blocks) can be done in pure C++ using a Khan process network (KPN) modelling style. This style is quite compact and retains a high-level of abstraction. An alternative is to use the thread or method processes in SystemC. Modular IO interfaces provides an encapsulation of cycle and pin accurate IO SystemC interfaces for synthesis while also providing a transaction (TLM) view for simulation. Such interfaces provide a way to separate out the cycle accurate communication so that the rest of the behavior can retain its high-level of abstraction.

Architectural synthesis

Architectural choices have a great impact on the performance and area of a design. They also have a great impact on power, though often in less predictable ways than for area and performance. Exploring different choices and measuring their impact is greatly facilitated by HLS.

Hierarchy boundaries

Hierarchy provides a way to both divide the design in more manageable pieces and to model concurrency in a more explicit way. A function that is called repeatedly can be extracted out in its own hierarchy. For example, in *Catapult*[4], such a block with a simple IO protocol is called a CCORE and may be either combinational or sequential, with or without state. It is possible to characterize the timing of such a block using the target ASIC or FPGA RTL synthesis tool to get an estimate that is consistent with the RTL tool's optimizations. The use of such a hierarchy increases the capacity, reduces runtime and helps get better quality of results by making it easier for the designer to focus on the larger picture.

A methodology supported in RTL synthesis is *bottom-up* design: sub-blocks are synthesized and brought in as part of a larger design. This is a design flow that is also expected in HLS and has been recently productized[4]. A number of alternative implementations for a hierarchy block can be created with different interfaces and later used as part of a larger design.

Interface synthesis

Interface synthesis converts the way the C++ function communicates with the outside world. During synthesis, the best interface can be selected to transfer the data at the rate that is required to meet a certain performance. For example, an array that is accessed sequentially in the behavior can be transferred four elements at a time to increase performance. In SystemC, interfaces are determined by ports on the module and alternative transfer widths need to be coded explicitly in C++ under the control of template parameters or macro defines.

Variable/array mapping and memory architecture

Power, area and performance are highly dependent on the memory architecture. It is important to code the C++ with hardware intent to achieve the best results. Within the the same C++, there are many choices that can be selected during synthesis to actually define the memory architecture of the design.

Interface or local variables or arrays may be mapped to memories or may be split into registers. Smaller arrays are typically mapped to registers while larger arrays are typically mapped to memories. The required read and write bandwidth of the memory depends on the algorithm and the performance requirements on the design.

Loop unrolling

Partially or fully unrolling a loop exposes parallelism that exist across subsequent loop iterations. In some cases, partial unrolling may also be used in a coordinated way with memory mapping and interface synthesis to increase the effective bandwidth for data transfer. For example, unrolling may expose the possibility of accessing even and odd elements of an array as one memory word when the array is mapped to memory.

Loop pipelining

Loop pipelining provides a way to increase the throughput of a loop (or decreasing its overall latency) by initiating the $(i+1)^{th}$ iteration of the loop before the i^{th} iteration has completed. Overlapping the execution of subsequent iterations of a loop exploits parallelism across loop iterations. In many cases loop pipelining may improve the resource utilization, thus increasing the performance/area metric of the design.

The pipeline initiation interval *(II)* is the number of cycles a pipeline stage takes to complete before initiating the execution for the next loop iteration. It is common to use and *II*=1 for highest performance, but other *II* values are used depending on the application. For example, a loop may be partially unrolled so the new body has two copies of the body of the original loop and an *II* of two may be used. Partially unrolling a loop may help expose some optimization potential across loop iterations. For example, it could expose the possibility to merge two arithmetic operators into a more optimized implementation. It may also expose

behavior that is specific to even and odd loop iterations and better optimize the copies of the loop body.

Loop pipelining can be applied at any loop level. The *process* from a SystemC process (e.g, SC_THREAD) or a function body that is synthesized as a module is considered the *top* level loop for that process. That loop can be pipelined in which case the loop will ramp up (fill all its pipeline stages) and continue execution accepting new inputs and producing new outputs. Alternatively, one or more inner loops may be pipelined. The pipeline ramps up to fill all stages and ramps downs as the each stage for the last iteration of the loop is completed.

A loop that contains other loops is pipelined by first flattening the nested loop into a single loop. Synthesis automates this transformation so there is no need for a manual rewrite of the behavior.

Loop merging

Two loops that are sequentially adjacent may be merged into a single loop that executes the same behavior. The merged loop can result in a hardware execution with less latency. It can also save hardware. For example, if the first loop fills up an array that is consumed by the second loop, the lifetime of that array will be reduced and may in fact go away altogether.

Technology library

While the target technology node might be a given, there are cases where a new technology node was not seen to provide a good benefit for the cost and the design was reoptimized for an older technology. That exploration would not have been possible in a manual RTL creation flow. Within the same technology node, choices of low or high V^{th} can also be considered.

Clock frequency

The choice of clock frequency can result in significant power savings. It is possible to change the clock frequency in conjunction with other architectural choices such as pipelining initiation interval to get the same overall design throughput.

Using hierarchy can be used to tailor the clock period to the specific data rate transfer of each block of the design. In DSP designs the input and output data rate for a block may be quite different. A good example is decimation. For a streaming design, blocks downstream and upstream from a block that have different input and output data rates could be clocked at different clock frequencies.

Scheduling

Scheduling transforms the untimed behavior (or partially timed in SystemC designs) into an architecture with a well defined cycle-by-cycle behavior. It takes into account required synthesis directives such as the clock frequency and the target technologies. In addition, it takes into account cycle and resource constraints that are either explicitly provided by the user or implied by interface synthesis directives, variable/array mapping directives and loop pipelining/unrolling directives. Scheduling selects among combinational, sequential and pipelined components that implement the operations in the algorithm.

Verification in an HLS design flow

The source input to HLS is written in C++ in a far more abstract form than an RTL implementation. The high-level of abstraction enables a designer to run orders of magnitudes more vectors than on an RTL implementation. The net effect is significant savings in verification because functional issues are caught earlier in the design cycle.

In an HLS design flow, the HLS tool creates the RTL from the high-level C++ specification in a way that preserves bitaccurate behavior equivalence under a set of assumptions of how interfaces get mapped and scheduled. Either RTL simulation or sequential formal equivalence checking[6] can be used to verify that the expected correct-by-construction property is not invalidated by an issue in the HLS tool. Until formal equivalence becomes commonplace, a full simulation or emulation based verification is still required for the generated RTL. Nonetheless, there is a verification saving as compared to manually created RTL since many verification cycles to debug functional bugs in the RTL are avoided.

Validating the C++ input description

Validating the input description is needed to create the initial design that is correct and to verify that it remains correct after numerical refinement (e.g. float to fixed-point) and after any rewrites that are meant to make the input more suitable for HLS.

Simulation is the most commonly used testing methodology for the C++ specification. The verification is done at different levels: it may start with a block and/or as sets of blocks in a subsystem. For example, a video decoding subsystem can run many frames of video to verify that the decoding is working correctly. The speed of the simulation allows to regression test the design with far fewer compute resources and in a far shorter time than what is required for an equivalent RTL design.

As the complexity of systems increase, it also becomes more difficult to have a high degree of confidence that the test vectors applied are sufficient to cover all the interesting scenarios. Generic code coverage is used as a metric in C++ for software development but lacks important features that have been developed for measuring coverage in RTL as described in Section C.

Formal tools can be used to check properties of the source C++ to make sure there are no ill-defined behavior such as out-of-bound accesses of arrays.

Verifying the generated RTL

Ideally verification of the design would be done fully in C++ and formal verification tools would prove that the RTL specification generated by HLS is functionally equivalent to the C++ design.

Sequential equivalence checking[6] has shown promise to formally verify that the generated RTL is functionally equivalent to the C++ specification. As it get deployed for block-level verification it promises to reduce some of the simulation-based verification requirements to more narrowly focus on testing the correctness of the integration of the blocks and subsystems in the SoC. Formal techniques will likely evolve to target aspects of integration such as correctness of protocols, deadlock-free operation etc.

Until formal techniques can address verification of the full SoC, traditional RTL verification methodologies will still be used to varying degrees to cover at least some verification aspects. Currently RTL verification methodologies typically include a mixture of simulation and emulation.

RTL coverage metrics

One of the challenges with verification is knowing if the input vectors that are used cover all the interesting scenarios and would therefore expose a functional issue in the design. Coverage metrics on the RTL are used an indicator of the quality of the testbench. Most verification engineers expect that the set of vectors that achieve full structural and functional coverage in the C++ specification also result in the same coverage when applied to the RTL specification generated by HLS.

There are a number of challenges in coverage metrics that artificially lowers the coverage obtained in the RTL specification:

- It is primarily intended to cover control conditions since fully covering arithmetic and datapath is extremely challenging. That means that the metric is sensitive on how functionality is expressed. For

example, a <= 0 may deliver different coverage than the equivalent a==0 && a < 0, because in the second case hitting the case a==0 is required for full coverage, but it is not required for a <= 0. Optimizations during synthesis may inadvertently introduce points that are reported as not being covered.

- Combinational and sequential redundancies in the logic can lead to a reduction of coverage, even if the testbench could be fully exhaustive. Synthesis can take care of eliminating such redundancies, though in some cases it may need to understand don't care conditions that come from the environment. For example an input may be encoded in a 1- hot fashion and if synthesis does not account for that information then some logic redundancies may be produced by synthesis.
- A coverage hole for an RTL signal can be reported as many coverage holes on the fanout of that signal. More formal tools to relate derivative coverage holes would be useful. Identifying such unique holes would both improve the metric and facilitate finding how to enhance the vector set to cover it or to find out if the a whole group of related coverage holes can be waived.

Some of the RTL verification methodologies are migrating to C++. Coverage on the C++ can be used to measure how well the testbench is exercising corner cases to see how well the source description is validated. The same input vectors can then be used to exercise the RTL. Traditional line coverage in C++ may not lead to full coverage as measured by RTL coverage tools:

- Coverage of the body of a function does not distinguish the coverage by different callers to that function. Synthesis, on the other hand inlines functions and the coverage may be less for some call contexts.
- Loop unrolling in synthesis replicates the body of a loop and it may expose a coverage hole that did not exist in the loop body prior to loop unrolling. For example:

```
for(int i=0; i < 2; i++) {
  wait();
  if(x & y)
    f(a);
}
```

After full loop unrolling becomes:

```
wait();
if(x & y)
  f(a);
wait()
if(x & y)
  f(a);
```

The input vectors for the second case need to be more extensive for covering the behavior for each replication of the expression x & y and of the inlined function f(a).

- Synthesis adds micro architectural details to the design and that introduces control that needs additional vectors to cover. For example, while in a pure C++ specification the interface is a function call, the synthesized RTL can have interfaces that wait for data to become available. The additional logic for stalling the design while it is waiting for data needs to be explicitly exercised in the testbench.

One of the sources of sequential redundancies is control that is distributed between an explicit finite-state machine (FSM) and a shift register that indicates which stage has valid data in a pipeline. It is possible to have combinations of the FSM state and valid states that are not reachable. If combinational logic is built without taking into account the unreachable scenarios, then sequential redundancy will be present. A similar source of sequential redundancies results from merging of two loops. The combined loop exits when the behavior of each loop has been fully executed. The exit condition of the combined loop is a logical AND of the exit conditions of each of the merged loops. Such conditions are often correlated and could result in a sequential redundancy that will result in coverage holes for the AND gate.

As stated earlier, the way that the RTL is expressed has an impact on whether is included as part of the coverage metric. For example, fixed-point datatypes provide and saturation behavior:

```
ac_fixed<24,24,true> y = ...;
ac_fixed<16,16,true,AC_WRAP,AC_SAT> x = y;
```

will check for overflow and perform saturation accordingly. In order to check overflow it will check if any of the bits 16-23 of y is not identical to bit 15 of y (since the MSB of x is bit 15). A natural expectation for a testbench is to check whether the condition for saturation is ever exercised. However, if the RTL specification for that check is written in terms of

discrete logical gates, the coverage metrics will indicate holes unless the overflow is exercised by a difference in each of the bits.

Coverage of stalling logic can be improved by directing HLS to add an optional stalling pin to any block. This provides a way to directly force stalls to exercise behavior that may otherwise be hard to reach with a testbench.

Emulation

A methodology that has been adopted by some HLS users is to limit simulation based verification for integration testing and move verification to emulation earlier. Skipping a lot of simulation based functional verification of the RTL is a pragmatic approach given that there is a higher confidence with HLS that the design is functionally correct. This is a trend that will likely grow as more verification tools and methodologies are developed to target the verification of the C++ source description.

Prototyping

One of the benefits of an HLS design methodology is that the same C++ source can be easily retargeted to a new technology. This enables prototyping an ASIC design using FPGAs. For example a video encoding design could be evaluated using such a prototyping flow.

ECO flows

In the traditional manual RTL creation methodology, ECOs are quite common. The main objective of an ECO methodology is to reduce the cost of implementing a functional change or a change to address a timing closure issue. For example, it is possible for functional verification to uncover a functional issue after place-and-route has been completed. Ideally, the change can be done with the least impact to all the work that has already been completed.

ECOs in RTL generated by HLS are quite rare relative to ECOs in manually generated RTL. Nonetheless, in order for HLS to fit into existing design flows, a similar ability to perform ECOs is generally expected by new users. In an HLS context, the change could be a functional change that needs to be done on the C++ design or a change in some synthesis directive. ECOs are facilitated by incremental synthesis flows that result in the least number of changes to the generated RTL. There are two challenges with incremental flows in HLS flows:

- Large designs can be done in HLS from an abstract compact description. The most effective methodology to bound changes is to divide the design using hierarchy, so any potential ECO change is isolated to a smaller subset of the design.
- A small change can be amplified if it is in code that is inlined more than one or is part of a body of a loop that is unrolled many times. As a general guideline, code that is inside of a loop that is not dependent on the loop iteration should be pre-computed before the loop. If the change happens to be required on that code, loop unrolling will not replicate the change in that code.

Incremental synthesis provides the least changes when the change is isolated within a single control step (combinational) and does not cross clock register boundaries.

RTL linting

RTL linting is designed to enforce certain styles to catch common mistakes that may lead to functional bugs or tool issues on that RTL. Unfortunately, the checks in some tools are fairly basic (pattern driven) and can raise many false alarms in RTL code that is well written. A lint-complaint rewrite of the RTL code can often seem more obfuscated and in fact be more likely to have an actual error that the original code. Lint rule decks used by different companies can be contradictory making it impossible for HLS to generate the same RTL that passes all lint rule checks in use.

In HLS, the RTL code is machine generated and RTL linting is still done as part of the same policy that applies to manually generated RTL code even though there appears to be little value for such checking. It is likely that as HLS is further deployed, linting will be adapted to a smaller subset of lint rules that is done with more sophisticated/formal analysis.

RTL synthesis flows

It is important for HLS to support the user's synthesis flow. That support can be provided by component characterization flows with the target synthesis tool to obtain relevant area and delay estimates for that specific tool. Both ASIC and modern FPGA technologies and synthesis tools are supported. Memory and other complex IP can also be characterized so that HLS understands how they can be used as part of the design.

In addition to the generated RTL, the RTL synthesis scripts are produced for the target tool. Timing constraints are produced in standard format for the generated RTL.

Power optimization

Using architectural exploration an HLS user can generate a number of designs and perform power analysis to find the design that consumes the least power. For example *Catapult* LP[4] has the analysis capability built-in. In addition it provides power optimization based on clock-gating. The optimizations *strengthen* the enables of registers, by restricting the conditions under which registers are enabled:

- Extract register feedback path conditions and re-express the feedback by adding/strengthening the enable condition.
- Strengthen the enable of a register based on the enable conditions of registers that are in its input cone of logic. This is stability based enable strengthening. Additional 1-bit registers may be required to hold the enable conditions from a prior cycle.
- Strengthen the enable based on the conditional use of the register in its fanout. This is an observability based[7,8,9] strengthening of enables that also reduces
- switching activity on the register and its fanout.

Table I shows the power savings achieved in industrial designs. The savings are highly dependent on the design. Designs that are pipelined and always active have less opportunity for saving power with clock gating. Designs that have communicating blocks where some blocks are stalled while waiting for data from other blocks can see more significant gains. Observability analysis delivers the most benefit when behavior is written as unconditional behavior whose result is used conditionally in a later cycle and that condition is available in an earlier cycle.

Table 1 – Estimated power savings with clock gating

Design	Base power µW	LP power µW	Savings %
Video encoder 1	2707	1338	51%
FFT	5383	5007	7%
Video encoder 2	966	805	17%
Motion estimation block	1786	1630	9%
Automotive	42	37	12%
JPEG	8935	8302	7%
Image scaler	25021	12370	51%

Conclusion

This paper summarizes the current design and verification methodologies using HLS and how it fits into existing design flows. The high costs of verification in traditional manual RTL flows continues to be the main reason to move up design and verification to a higher level of abstraction. Verification is expected to be a major source of innovation as formal techniques and ways to complement it with simulation-based verification and a more rapid transition to emulation become mainstream. Some of techniques developed for RTL such as assertions and coverage will be applied to the C++ specification.

Another driver for HLS is low-power design. As HLS facilitates reuse of the C++ IP, it enables targeting an existing C++ to a new technology node to get hardware more highly optimized for power. As results show, HLS optimizations that understand the sequential properties of the design are able to better optimize the design and deliver additional power saving.

References

1. http://www.webmproject.org/hardware/vp9
2. J.-R. Ohm, W.-J. Han, T. Wiegand. "Overview of the high efficiency video coding (HEVC) standard". *IEEE Transactions on Circuits and Systems for Video Technology*. Volume 22, Issue 12. Dec. 2012.
3. Algorithmic C (AC) Datatypes. https://hlslibs.org/HLSLibs/ACDatatypes/#.
4. Catapult synthesis, Siemens Digital Industries Software, https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/
5. P. Coussy and A. Takach. "Raising the abstraction level of hardware design," *IEEE Design &Test of Computers*, vol. 26, no. 4, pp. 4–6, Jul.–Aug. 200.
6. Anmol Mathur, Masahiro Fujita, Edmund Clarke and Pascal Urard. "Functional equivalence verification tools in High-Level Synthesis flows," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 88-95, July/August, 2009
7. Mitsuhisa Ohnishi, Akihisa Yamada, Hiroaki Noda, and Takashi Kambe. "A method of redundant clocking detection and power reduction at RT level design". In Proceedings of the 1997 international symposium on Low power electronics and design (ISLPED '97). ACM, 1997.
8. Babighian, Pietro, Luca Benini, and Enrico Macii. "A scalable algorithm for RTL insertion of gated clocks based on ODCs computation." *Computer-Aided Design of Integrated Circuits and Systems*, IEEE Transactions on 24.1 (2005): 29-42.
9. Jason Cong, Bin Liu, Rupak Majumdar and Zhiru Zhang. "Behavior-level observability analysis for operation gating in lowpower behavioral synthesis". *ACM Trans. Des. Autom. Electron*. Syst. 16, 1, Article 4 (November 2010).

Siemens Digital Industries Software

Headquarters
Granite Park One
5800 Granite Parkway
Suite 600
Plano, TX 75024
USA
+1 972 987 3000

Americas
Granite Park One
5800 Granite Parkway
Suite 600
Plano, TX 75024
USA
+1 314 264 8499

Europe
Stephenson House
Sir William Siemens Square
Frimley, Camberley
Surrey, GU16 8QD
+44 (0) 1276 413200

Asia-Pacific
Unit 901-902, 9/F
Tower B, Manulife Financial Centre
223-231 Wai Yip Street, Kwun Tong
Kowloon, Hong Kong
+852 2230 3333

About Siemens Digital Industries Software

Siemens Digital Industries Software is driving transformation to enable a digital enterprise where engineering, manufacturing and electronics design meet tomorrow. Xcelerator, the comprehensive and integrated portfolio of software and services from Siemens Digital Industries Software, helps companies of all sizes create and leverage a comprehensive digital twin that provides organizations with new insights, opportunities and levels of automation to drive innovation. For more information on Siemens Digital Industries Software products and services, visit siemens.com/eda or follow us on LinkedIn, Twitter, Facebook and Instagram. Siemens Digital Industries Software – Where today meets tomorrow.

siemens.com/eda

81422-C1 3/21 H