2009-01-0519

Using Model-Based Design to Accelerate FPGA Development for Automotive Applications

Sudhir Sharma
The MathWorks

Wang Chen
The MathWorks

Copyright © 2009 SAE International

ABSTRACT

A recent Gartner Dataquest study predicts that the total worldwide automotive semiconductor market will grow from $20.1 billion in 2007 to $25.9 billion by 2010. The study also predicts that revenue from automotive usage of FPGAs will triple to approximately $312 million during that same period[1].

Many of these FPGAs will be deployed in safety applications such as back-up cameras, lane departure warning systems, blind-spot warning system, and adaptive cruise control. FPGAs will also be deployed in next-generation engine electronics, emissions control, navigation, and entertainment applications.

Automotive systems engineers are adept at using Model-Based Design for implementing some of these embedded applications on DSPs and microcontrollers. Many of these engineers are new to FPGA design and waking up to a fragmented workflow that is making it harder to meet time-to-market and cost objectives.

For example, engineers who are migrating their systems designs from DSPs to FPGAs are discovering that additional verification steps such as bit-true, cycle-accurate simulations are required to ensure that the FPGA functions the same as the system specification. This is a time-consuming and error-prone activity involving file exchanges between the system designer and the FPGA designer. Geographically distributed teams face an even bigger challenge since the system engineer and FPGA designer may be many miles away from each other.

This paper illustrates how Model-Based Design integrates the world of system designers, FPGA designers, and verification engineers to increase productivity and produce correct-by-construction designs that match the system specification. Using the concept of executable design specification, this paper discusses how Model-Based Design streamlines both design and verification of FPGAs for automotive applications in two important automotive workflows:

• FPGA design and production deployment to low-volume high-processing power applications such as driver-assistance and infotainment systems.

• FPGA use for prototyping in high-volume applications such as engine and steering control, where the final production deployment will be an ASIC. In this workflow, the proof of concept work is done using FPGAs.

INTRODUCTION

Many new cars now include electronic safety systems such as collision avoidance, adaptive cruise control, and lane-departure system. These new features often require fast processing power and large memory to handle video data streaming, image recognition, or other signal processing, which greatly increase the hardware and software complexity in automotive electronic systems. Many of these complex functions can not be implemented in software running on traditional DSP and microcontrollers. Instead, automotive hardware designers are now focusing on FPGAs and ASICs.

Compared to traditional DSPs and microcontrollers (MCUs), FPGAs and ASICs offer faster processing speed and more functionality to support more advanced features. Choosing between an ASIC and an FPGA implementation depends on the application and is beyond the scope of this paper, but, broadly speaking, an FPGA implementation can be a faster time-to-market and lower-cost solution than an ASIC design. FPGAs also offer the added benefit of reconfigurability when the design specification changes. On the other hand, an ASIC may be the right solution for a large volume, very high-speed, or power-sensitive application.

FPGA usage is growing rapidly because it satisfies the automotive industry's demands for faster processing

speed, higher logic density, shorter time-to-market cycle, and reconfigurability[2].

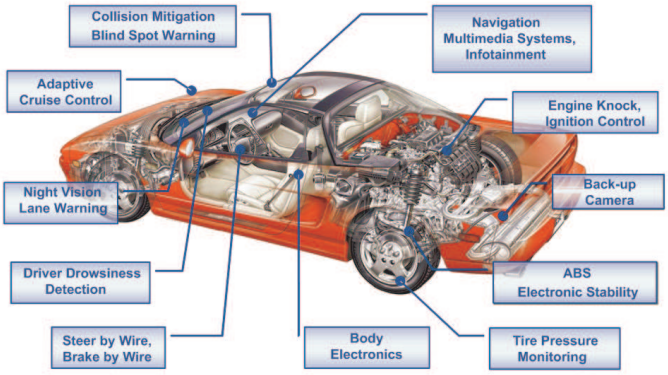


Figure 1. FPGA applications in the automotive industry.

Figure 1 lists a few of the applications where FPGAs may be deployed. These include active safety system, powertrain, chassis, body electronics, and infotainment systems.

In Table1, we capture five major automotive applications and their suitability for FPGA-based implementation.

Active-safety systems and infotainment systems are the most popular areas for FPGA applications. These applications need to process large amounts of streaming input data and provide responses in real time. Their image processing and radar signal processing algorithms may easily consume the processing power of an entire DSP processor. In such applications, instead of using a fast, stand-alone DSP, the adoption of an FPGA as a hardware coprocessor can offer a more compelling solution that allows the designer to use a smaller DSP. This can result in greater system performance at a lower cost[3,4].

FPGAs are also found in powertrain systems as a coprocessor for engine controller tasks such as knock detection and injector control. While ASICs are often used in high speed applications such as gasoline direct injection, diesel multiple injection, and electronic valve lifting, newly introduced flash-based FPGAs may be an attractive option for these applications[1,5].

As of two years ago, premium vehicles such as BMW 5 Series employed well over 100 MCUs to control the various systems in the car. Automotive engineers have been working to consolidate the functions of multiple

Applications	Examples	Key Requirements	Present Technology	FPGA Viability
Active Safety Systems	Adaptive cruise control Collision mitigation Lane departure warning Back-up camera Blind spot warning.	Throughput Reconfigurability	DSP and ASIC/FPGA New systems have more FPGAs in them.	Yes. As a coprocessor for video and image processing. For high processing speed and reconfigurability.
Powertrain	Engine control module: – Fuel injection – Knock detection – Ignition timing Power electronics control Transmission control Vehicle energy management.	Cost/Memory Speed Complexity Reconfigurability	MCU (high end) ASICs are used today extensively in engine control and power electronics control.	Yes. As a co-processor for tasks such as knock detection, fuel injector control, and power electronics control. For HW and SW codesign.
Chassis	Steering control ABS Electronic stability (ESP) Ride control Brake-by-wire	Cost/Memory Speed Redundancy	MCU (mid end) DSPs are sometimes used for steering control.	Yes. Possible for power steering control, which requires high processing speed.
Body Electronics	Roof, window, mirror, seat Climate control Instrument cluster Central body control Network gateway.	Cost/Memory Low Power Reconfigurability	MCU (low end)	Yes. To consolidate the functions of multiple MCUs that are distributed around the vehicle into one FPGA [3].
Infotainment Systems	Navigation Voice recognition Digital audio/TV Rear-seat entertainment.	Throughput Reconfigurability	MCU and DSP	Yes. For high processing speed and reconfigurability.

MCUs into one FPGA to reduce the system complexity[6]. Even as the unit costs of both DSP and FPGA devices are falling, the growing system complexity and increasing need for raw processing power make FPGAs a compelling solution for many automotive applications.

For example, Saab's trionic engine combustion control processing is straining the abilities of MCUs[7]. Conceivably, the engine can burn hydrocarbons and CO in polluted air that is sucked into the intake[8]. The ECU uses a Motorola 683xx 32-bit processor, 4Mbit of memory, an 8-bit coprocessor, barometric sensor, and other components[9].

By using the spark plug as an ionization probe during each idle cycle in the four-cycle process and sensing temperature, pressure, and unburned exhaust, everything can be adjusted to its optimum to wring out every bit of power available in the gasoline, based on real-time combustion results and driving conditions.

Taking this technology farther will require more than a faster embedded processor if it is to be done efficiently for economy autos as well as high-end luxury vehicles.

MODEL-BASED DESIGN

Model-Based Design improves design quality and accelerates design and verification tasks by employing an executable specification. This executable specification is elaborated to create hardware and software partitioning, automatically create hardware and software implementation code, and verify the hardware and software implementations in the context of the complete system (as shown in Figure 2). Significant advantages of Model-Based Design include the fact that it facilitates rapid design iterations and it moves the verification process all the way to the beginning of the design cycle. This helps detect system specification related errors, design errors, and implementation errors early.

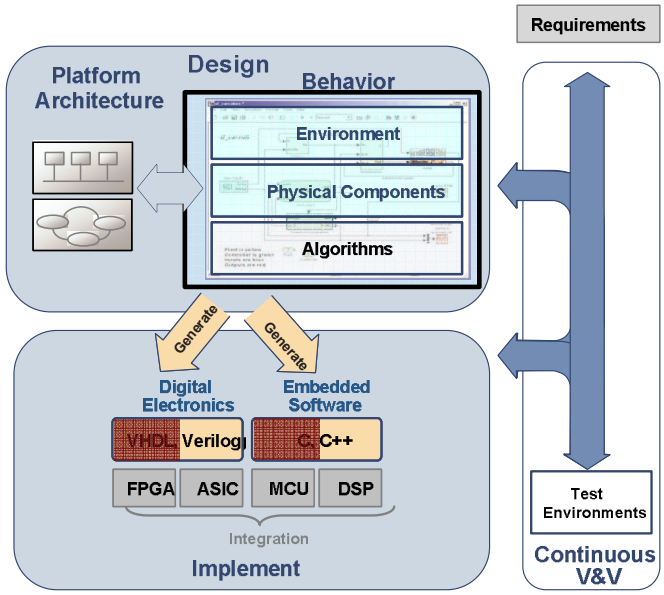


Figure 2. Model-Based Design that uses an executable specification and allows continous system-level verification.

FPGA SYSTEM DESIGN CHALLENGE

As stated in the introduction, automotive systems engineers are adept at using Model-Based Design for implementing embedded applications on microcontrollers and DSPs. They are typically new to FPGA design. Hardware design workflow is significantly different from software design workflow and the transition can be difficult. The following sections outline some of the typical problems with the manual FPGA design workflow and propose an integrated system-level design approach to FPGA development.

MULTI-PASS WORKFLOW

In the typical hardware system design workflow, a system designer designs the algorithm and creates a text-based design document and corresponding I/O vectors for hardware engineers. FPGA designers then translate this specification into a hardware realizable model by hand-coding either Verilog or VHDL code, the two common hardware description languages (HDLs). To verify that the hand-coded HDL behavior matches the system specification, they write extensive HDL test benches to exercise the I/O test vectors provided by the system designer, as shown in Figure 3.

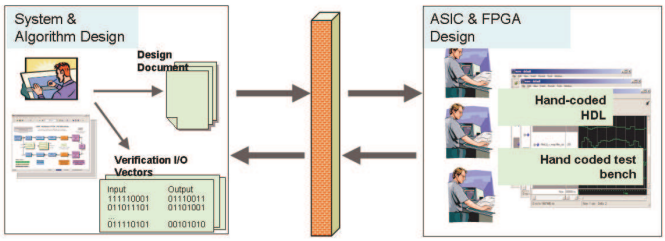


Figure 3. Typical text-based system specification to hardware design workflow that leads to many errors.

While this workflow looks very straightforward, the reality is that it is a laborious and inefficient process.

The system designers need to spend extra effort to create and maintain the text-based design document and test I/O vectors which are only used by FPGA designers. Additional effort is wasted by FPGA designers to create the module level HDL test benches that are not usable for chip-level verification.

This workflow requires close collaboration between system engineers and hardware engineers. However, this level of collaboration is not always easy because system engineers and hardware engineers may be physically located far away from each other.

As we know, design specification changes are inevitable. This workflow breaks down even more when design iterations are required due to specification changes. Working through hundreds or thousands of lines of code is invariably more inefficient than working at a higher level of abstraction.

VERIFICATION

Ensuring that the FPGA implementation matches the system specification in bit-true, cycle-accurate simulations is a time-consuming and error-prone activity involving many file exchanges between the system designers and the FPGA designers. As many as 10 lines of test code may be needed for each line of hardware implementation code. Moreover, these module level test benches and verification scripts are often not directly usable for FPGA system level verification.

HARDWARE AND SOFTWARE CODESIGN

Making the right partitioning choice is a complex decision process and often requires multiple iterations. In many automotive applications, a portion of the design may be running on a DSP or an MCU and some time-critical application may be running on the FPGA coprocessor.

Since the end application requires a seamless interface between hardware and software, engineers need to work in an integrated design and verification environment that allows them to evaluate various hardware and software portioning options to achieve an optimal implementation.

HOW MODEL-BASED DESIGN HELPS

Model-Based Design integrates the world of system design engineers, FPGA designers, and verification engineers to increase productivity and produce correct-by-construction designs that match the executable system specifications.

Figure 4 illustrates a typical Model-Based Design workflow where MATLAB and Simulink are used as the environments for capturing system-level algorithms and design specifications. The steps in this workflow include:

1. Create an executable specification consisting of implementable algorithms, system model, and system-level verification environment.
2. Verify the system model against functional requirements using simulation.
3. Automatically generate production software for embedded processors and synthesizable HDL code for ASICs and FPGAs.
4. Employ the executable specification as a test bench to verify software and hardware implementations.

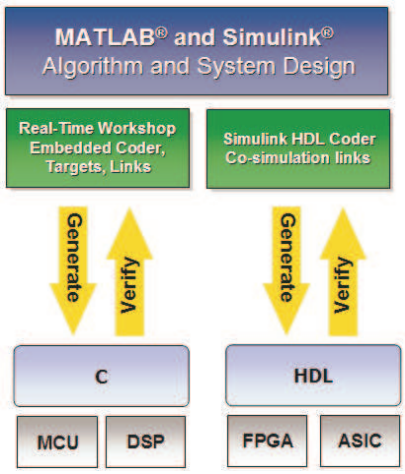


Figure 4. MATLAB and Simulink as the foundation of Model-Based Design.

Engineers developing control algorithms that target microcontrollers and DSPs make extensive use of automatic production code generation technologies. Additionally, off-the-shelf interfaces between the code generator and integrated design environments (IDE) enable them to perform equivalence testing using, for example, processor In-the-loop (PIL).

On the hardware side, recent advances in automatic HDL code generation technology, such as Simulink HDL Coder from The MathWorks provides bit-true and cycle-accurate synthesizable HDL code for ASIC or FPGA implementation.

Reusing the executable specification as a system level test bench allows the engineers to ensure that final hardware and software implementations match the true intent of the system design engineer.

In this workflow, cosimulation tools enable you to verify the correct functionality of your HDL code with industry leading HDL simulators from Mentor Graphics, Cadence, and Synopsys.

Together, HDL code generation and cosimulation help engineers shorten the two most time consuming and error prone aspects of system design—coding and verification.

The iterative nature of embedded system design and chip design requires use of automated workflows that allow you to do rapid prototyping of your ideas before committing to a particular implementation.

Model-Based Design enables systems designers to do just that. You can iterate your design to achieve optimal area-speed-power implementation, focus on design architecture and value-added IP, and then automatically generate the implementation or prototyping code.

By elevating the abstraction level from HDL code to system level design, Model-Based Design enables chip designers to focus on the many other tasks that complex IC design involves, including IP integration and verification for the rest of the chip.

HARDWARE AND SOFTWARE PARTIONING

Successful system level design requires engineers to have a thorough understanding of their application, the environment where it will operate, and other factors such as the baseline performance.

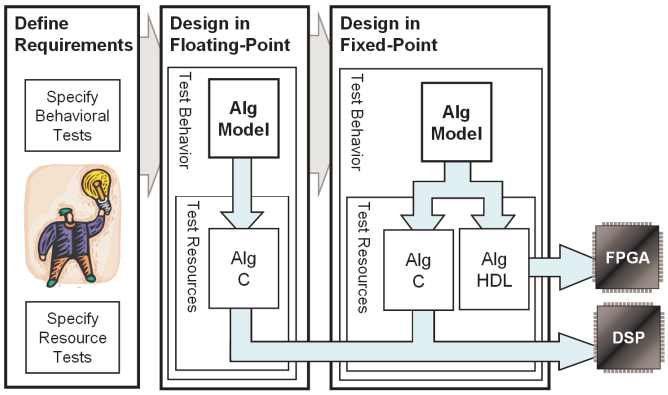


Figure 5. System-level specifications that drive hardware and software partitioning and implementation.

As illustrated in Figure 5, these high level requirements then drive the decisions of choosing hardware and software partitioning, hardware and software design and implementation, and the choice of target platform, such as DSP vs. microcontroller or FPGA vs. ASIC.

Using Model-Based Design, engineers can use the executable specification to generate C code for a DSP, evaluate the DSP performance, and then evaluate the same algorithm on an FPGA. In contract to working at the C and HDL level, engineers gain enormous productivity by working at the high level of abstraction afforded by Model-Based Design.

REUSABILITY

Since both system design and software and hardware implementation are using one golden reference model, it is very easy to maintain the model and reuse it later.

WORKFLOW WITH CASE STUDY

We illustrate the implementation of a Model-Based Design workflow by developing a knock detection algorithm. In this case study, we use MATLAB and Simulink as the foundation tools for creating the executable specification, elaborating this specification for hardware and software code generation, and verifying that the hardware and software implementations match the original executable specification.

We are motivated to use a knock detection algorithm, as many automotive engineers are already using a Model-Based Design workflow for embedded system design with DSPs and MCUs. We show how they can easily target an FPGA using the same concepts.

SYSTEM LEVEL DESIGN

The internal combustion engine relies on precise timing to burn the air and fuel mixture. Premature ignition of the air and fuel mixture causes the engine to produce a knocking in the engine, as shown in Figure 6. This knocking can lead to engine damage if left uncorrected.

Fortunately, engineers have developed signal processing spectral analysis techniques to easily identify engine knock. By leveraging this knowledge, engineers can optimize the performance of ignition timing control logic engine, improve the engine performance, and prevent engine knocking.

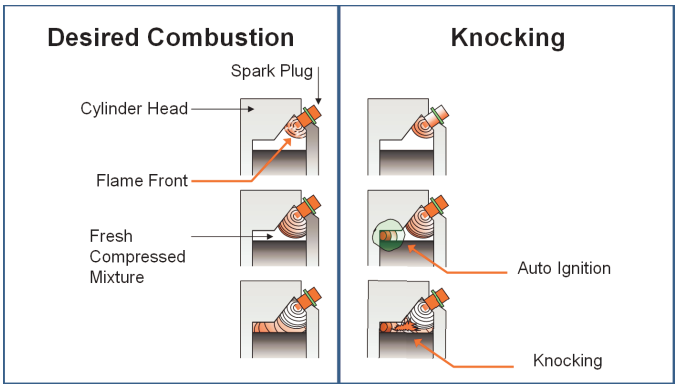
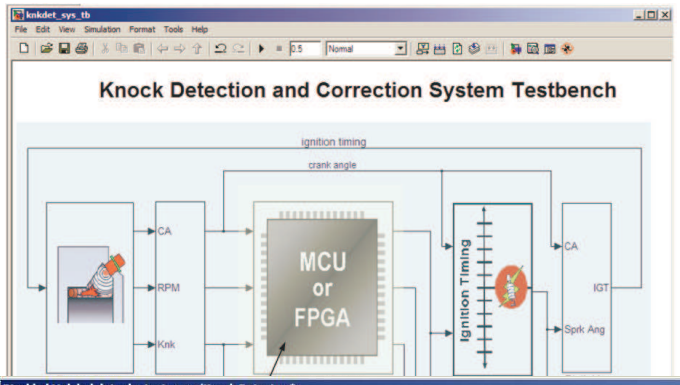


Figure 6. Premature ignition of the air and fuel mixture in internal combustion engine that can lead to engine knocking.

Executable Design Specification

The model in Figure 7 shows the executable specification of an engine knock detection and correction system. This model includes the system specification and the knock detection algorithm. The system specification provides the system-level constraints and operating environment for the algorithm. At this point in the workflow, the executable specification is used to model, simulate, and iterate the algorithm in the context of the complete system. The focus of the executable specification is the correct functional development of the knock detection algorithm, with little consideration for implementation detail.



The executable specification shown in Figure 7 includes the engine spark knock simulator and the analog-to-digital converter (ADC), the electronic spark advance (ESA), and digital-to-analog converter (DAC) subsystems as the testing environment for the knock detector algorithm. The ADC is used to sample the knock sensor signal coming from the engine knock simulator model. This parameterized model is based on a combination of theoretical and empirical data. It contains elements to represent a fundamental knocking frequency of 6 KHz, as well as first and second harmonics. Moreover, the model can be configured to inject additional noise into the system.

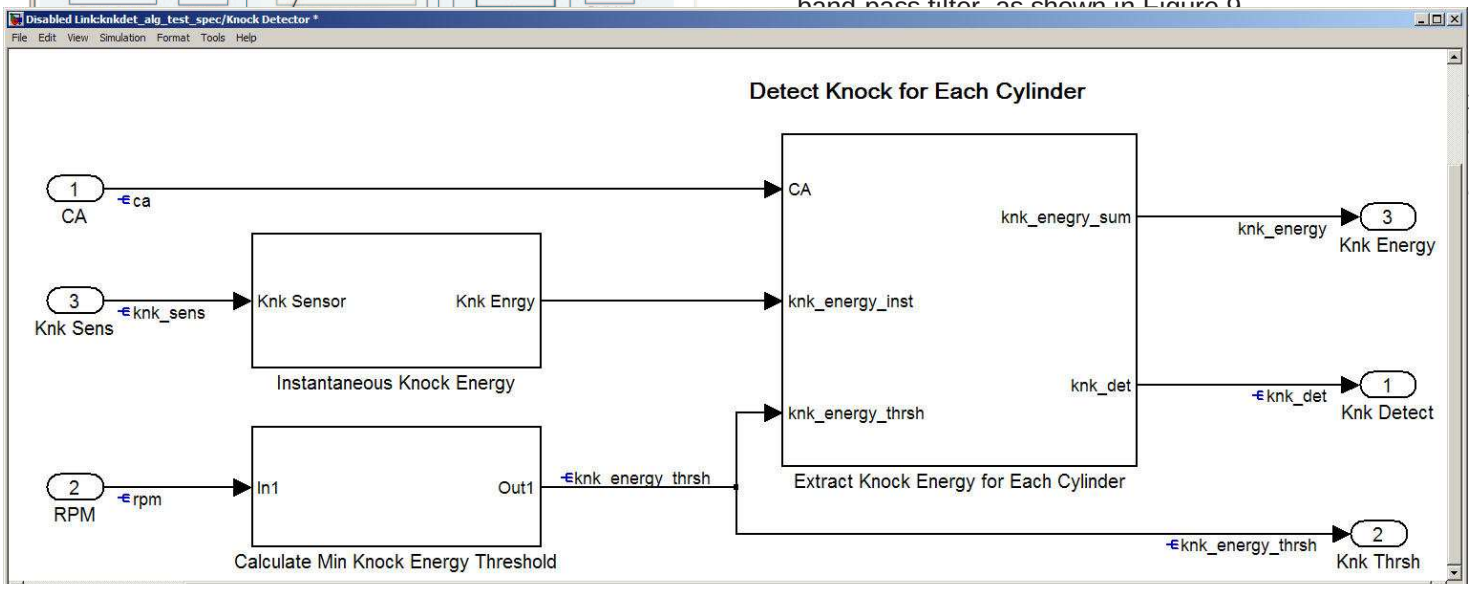
The knock detector subsystem, which is the target design of this case study, is responsible for detecting engine knock independently for each cylinder. The electronic spark advance subsystem adjusts the ignition timing signal to compensate for engine knock. If engine knock is detected, the spark timing is reduced to prevent knock. If no knock is detected, the spark timing is advanced to optimize engine performance and emissions.

The knock detector algorithm should identify the engine knock fast and correctly. As engine events occur at over 1 KHz at 6000 RPM, the knock detector system should meet the timing requirement for engine controller to fine tune the ignition timing and fuel injection.

Design the Floating Point Algorithm

With these design constraints in mind, we next create the floating-point model of the knock detection algorithm, shown in Figure 8.

Fundamental signal processing techniques are used to extract knock frequency content from the digitized knock sensor and estimate the knock energy. In this system, only one knock sensor is used. The crank angle (CA) signal is required to determine the knock window associated with each cylinder. The knock energy associated with each cylinder's knock window is compared with a threshold to determine if cylinder knock is present. The knock energy is extracted using a simple band-pass filter, as shown in Figure 9.



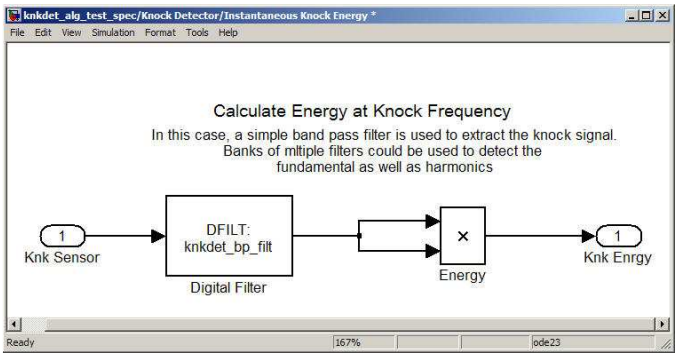


Figure 9. Knock energy calculated using a band-pass digital filter.

As shown in Figure 11, the pulses of sinusoid signals in the first axis represent engine knock. The second axis represents the output of the knock sensor, which will output ”true” or digital logic ”1” when engine knock is detected. The spark ignition for this cylinder starts at a given crank angle (in degrees). During the intervals where spark is detected the cylinder spark injection signal is retarded (corresponding crank angle is decreased). Once no knock is detected, the spark injection signal is then advanced (corresponding crank angle is increased) at a step size less aggressive than the when spark is detected. As shown in this example, if the ignition timing continues to advance and knocking occurs, the ignition timing is again retarded.

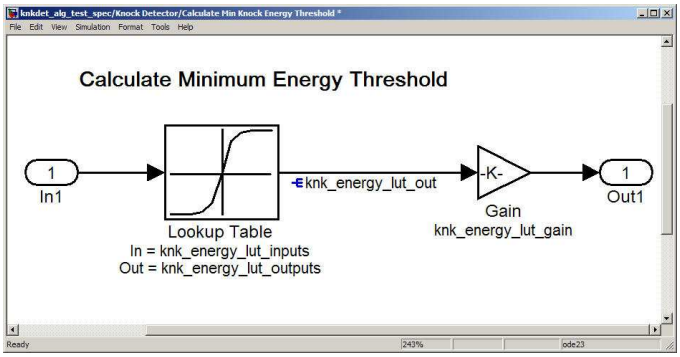


Figure 10. Lookup table implementation of knock detection energy threshold.

FIXED-POINT CONVERSION

Algorithms can be implemented in digital hardware to process either floating-point numbers or fixed-point numbers. At the expense of dynamic range, hardware implementations with fixed-point data type result in a smaller, more power efficient, faster, and cost-effective solution.

The Fixed-Point Conversion Advisor utility is used in this case study to help automatically determine the precision needed.

To ensure equivalency, floating-point models and fixed-point models are compared side-by-side, as shown in Figure 12. When we execute this model, we see the output of the “golden specification,” the output from the fixed-point model, and the difference between the two models. We can rapidly iterate this fixed-point model and try different fixed-point settings to achieve the right balance for your application.

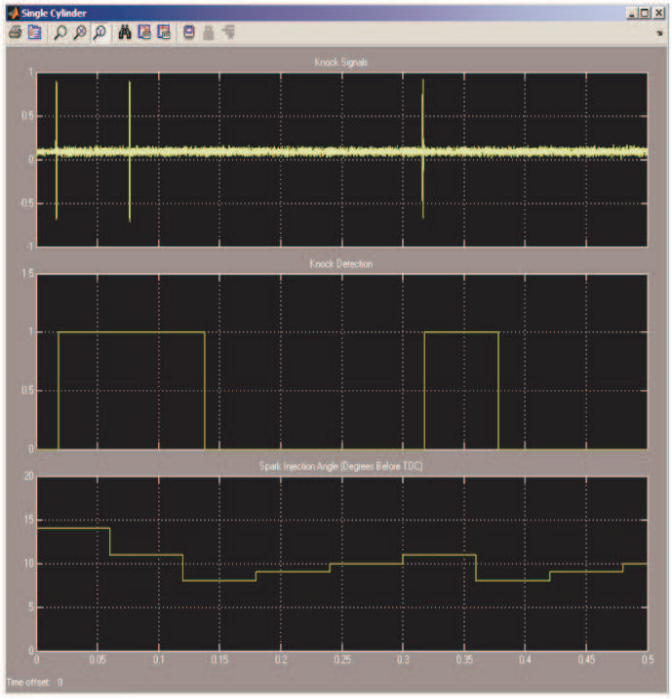


Figure 11. Engine knock waveforms.

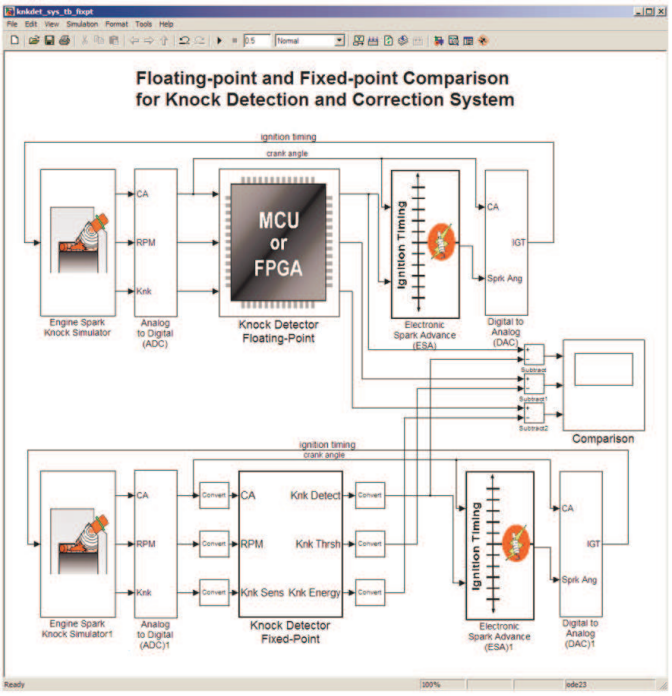


Figure 12. Comparison of floating-point and fixed-point models side-by-side to achieve optimal performance.

HARDWARE-SOFTWARE PARTITION

In this case study, we implement the knock detector algorithm on an FPGA and implement the spark control algorithm in software running on a microcontroller.

AUTOMATIC CODE GENERATION

Once the fixed-point model meets design requirements, we can invoke the Simulink HDL Coder to automatically generate HDL code and test benches directly from the fixed-point model. The generated HDL code matches the fixed-point model in bit-true, cycle-accurate simulations.

The automatically generated HDL code is correct by construction, enabling the designer to save initial hand-coding time and debugging time. Since the designers are working at the level of a system model and not at the level of HDL code, they are able to create quick prototypes and design iterations to keep up with rapidly changing specifications.

For many reasons, including verification, it is important that automatically generated code is readable and integrates seamlessly with hand-written HDL code. Imagine trying to debug your design and tracing a signal into a block where you can not read the HDL code.

SYSTEM LEVEL VERIFICATION

A corresponding HDL test bench is also generated by Simulink HDL Coder. The test bench input/output vectors are generated directly from the fixed-point design specification model to verify that the generated FPGA implementation meets the functional specifications.

EDA simulator link products enable you to cosimulate the HDL code with MATLAB and Simulink so you can ensure that your FPGA implementation matches the original executable design specification.

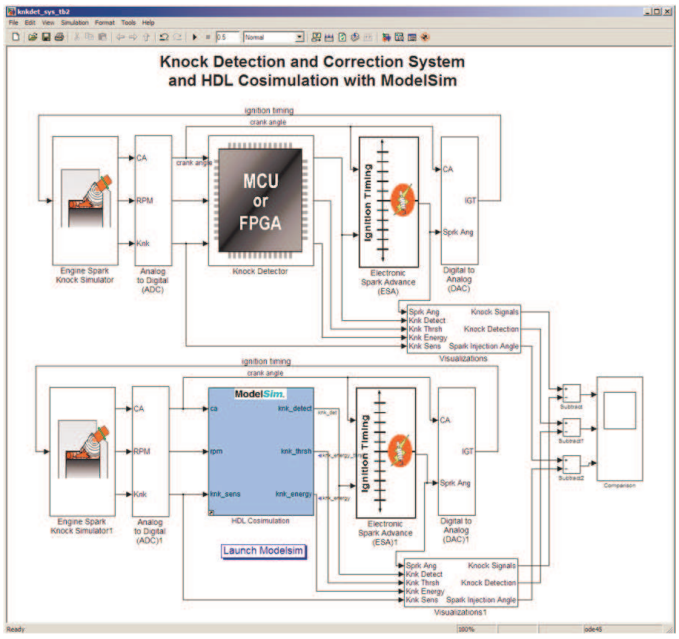


Figure 13. Cosimulating the HDL code with the fixed-point model for ensuring correct hardware implementation.

As shown in Figure 13, the knock detector module is implemented in an FPGA, and the spark advance algorithm is implemented in a microcontroller. We use this model as a test bench to verify both the HDL and C implementations. For the HDL cosimulation, we used EDA Simulator Link MQ (for use with ModelSim).

RAPID ITERATION

Nearly 60%[10] of integrated circuit designs require rework. In addition to functional errors, performance issues, or a changed specification could be the cause. Size, power, or speed may need to be optimized. To address these factors, designers may need to go back to the fixed-point model and adjust the bit-widths, simplify the algorithm, or choose another implementation.

Because engineers using Model-Based Design are invested in the model and not the HDL code, they can readily iterate the fixed-point model to quickly create alternative implementations. System engineers can focus their time on refining the algorithm, and hardware engineers can focus their time on optimizing the implementation for specific targets.

CONCLUSION

In summary, Model-Based Design provides a design flow that directly maps executable system specification into hardware. Model-Based Design permits system engineers and HDL engineers to collaborate using functional models of the design specification that can be executed and understood more easily, thus significantly speeding up design and verification activities.

Starting from a golden Simulink system specification, the design engineers can quickly generate synthesizable, target-independent, human-readable, and correct-by- construction HDL code. This workflow enables design engineers to quickly prototype and iterate their algorithm to keep up with rapidly changing specifications and standards.

Using the cosimulation tools, engineers can do system-level verification to ensure that the HDL code is functionally equivalent to the system specification.

REFERENCES

1. C. Maxfield, “FPGAs gear up for new automotive application”, *EETimes*, Oct 16, 2007.
2. M. Gabrick, et al, “FPGA Considerations for Automotive Applications,” *2006 SAE World Congress*, April 3-6, 2006.
3. T. Costlow, “Looking forward to safer highways,” *Automotive Engineering International Online*, Sep, 2008.
4. T. Mehta, “How FPGAs Enable Automotive Systems,” *Altera Corporation*.
5. M. Mason, et al, “FPGA Reliability in Space-Flight and Automotive Applications”, *FPGA and Programmable Logic Journal*, 2005
6. M. Traub, et al, “Generating hardware descriptions from automotive function models for a FPGA-based body controller: A case study”, *MathWorks Automotive Conference 2008*.
7. http://www.saabhistory.com/2007/11/19/saab-trionic-saab-innovation/
8. http://www.autobloggreen.com/2007/11/13/video-saab-trionic-actually-cleans-the-air/
9. http://pikkupossu.1g.fi/tomi/projects/trionic/trionic.html
10. Dr. Jack Horgan, “Hardware/Software Co-verification, Dr. Jack Horgan, March 29, 2004. http://www10.edacafe.com/nbc/articles/view_weekly.php?newsletter=1&run_date=29-Mar-2004