# Designing a Clock Cycle Accurate Application With High-level Synthesis

Sakari Lahti, Jarno Vanne, Timo D. Hämäläinen
Department of Pervasive Computing
Tampere University of Technology
Tampere, Finland
sakari.lahti@tut.fi, jarno.vanne@tut.fi, timo.d.hamalainen@tut.fi

***Abstract*—During the recent years, high-level synthesis (HLS) has gained traction as a viable alternative to traditional handwritten register transfer level code in describing digital systems. This has been attributed to the maturing of the HLS tools and improving quality of their results. However, most published applications are data path intensive as HLS offers good tools for loop optimization, such as pipelining and loop unrolling. HLS is seldom applied to control-oriented applications since clock is not explicitly present in HLS source code. In this paper, we show how a clock cycle accurate application can be described with HLS. We give as a proof of concept an implementation of an FPGA-based I2C bus controller for an audio codec using Catapult C, and present a generalized work flow. Compared with a corresponding handwritten VHDL implementation, the HLS version consumes 84% more area at the same performance but productivity is increased by 100% at the first design time and even more with further design iterations.**

***Keywords—High-Level Synthesis; C-to-RTL; Catapult-C; FPGA; I2C***

## I. Introduction

For decades now, the register-transfer level (RTL) has been the dominant method for describing digital systems. Problematically, RTL languages such as VHDL and Verilog require expertise that is unfamiliar to engineers from the software domain who greatly outnumber hardware engineers. Furthermore, writing RTL code is tedious and time consuming, which is becoming a challenge in a world where transistors are cheap and abundant but engineering hours are not.

High-level synthesis (HLS) is the emerging method to address these problems by raising the abstraction level of the description language [1]–[5]. In HLS, familiar programming languages such as C and C++ are used to describe the system at a behavioral level. An HLS tool takes the high-level source code as an input along with a platform-specific hardware library, allocates required computing resources from the platform, schedules operations, and binds them to the resources [4]. The output is an RTL language description of the system that can be synthesized on, e.g., FPGA using downstream tools. In addition to reducing development time, the benefits of HLS include fast design space exploration, flexibility in targeting new platforms, and faster verification [3].

Usually, the reported quality of HLS results, such as area usage and performance, has been worse than with RTL [6]–[11]. Recently, there have been promising approaches where this gap has been lessened or even reversed [12]–[16]. However, most of the reported applications are data path oriented since HLS is amenable to fast design space exploration and optimization with loops that transform data. With control-intensive applications, the benefits are less prominent, but if the productivity improvement is still there, the transition from RTL to HLS should be made possible.

Most control-oriented applications require accurate operations synchronized to a clock. However, most HLS languages, such as C++, are timeless, i.e., they do not contain the concept of time that synchronizes the operations. HLS tools do the operation scheduling automatically with little possibility by the designer to affect it. This is an obvious gap between the need to design clock-accurate applications and the abilities offered by the HLS tools.

In this paper, we show that clock-accurate systems can be realized using such timeless source code. Our main contributions are:

1) guidelines on how to write the HLS code to support clock-accurate behavior;
2) an I2C bus [17] based test case application (an audio synthesizer) including real-time constraints like user interaction;
3) comparison of area and development effort between HLS and handwritten VHDL RTL using the test case synthesized on FPGA.

The rest of this paper is organized as follows. In Section 2, we show the hardware architecture of the case study system. Section 3 introduces our HLS design flow. Section 4 discusses the relevant implementation aspects, based on which a recommended work flow is presented in Section 5. Section 6 shows our results and finally, Section 7 contains conclusions and future research directions.

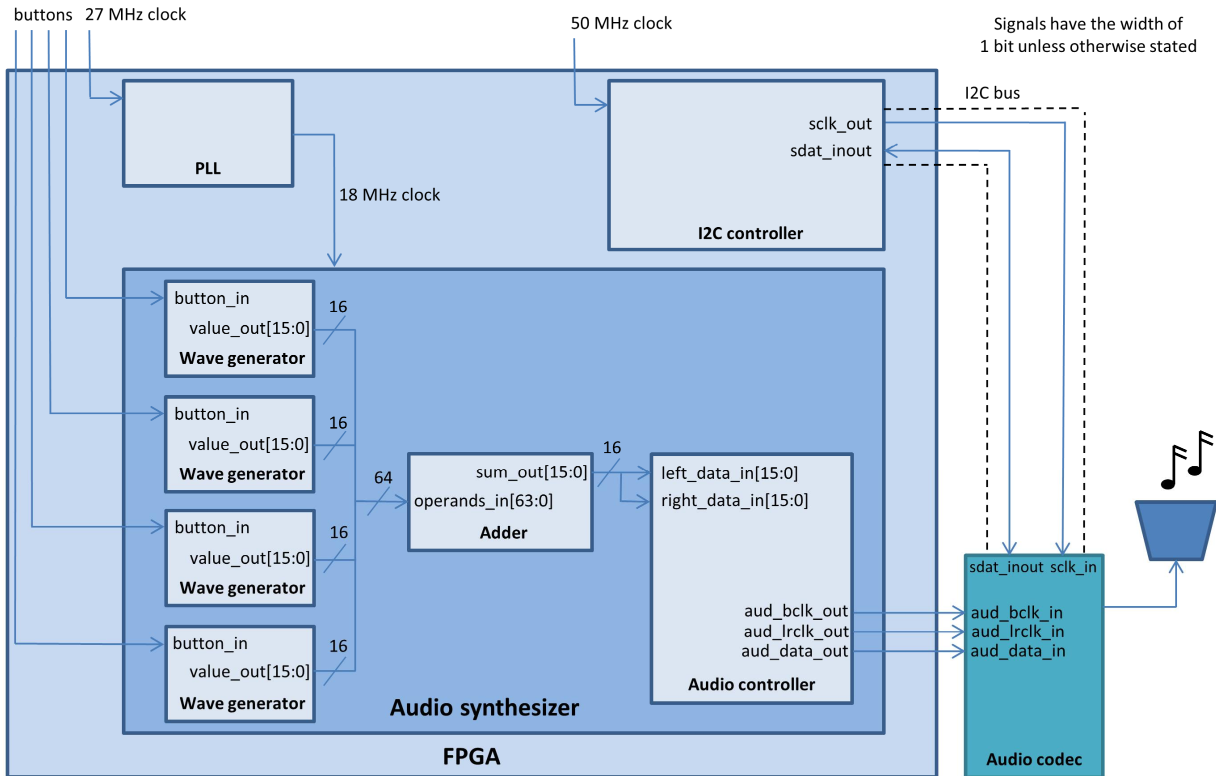


Fig. 1. System architecture.

## II. SYSTEM DESCRIPTION

As a test case application, we created a simple audio synthesizer that is able to generate four different tones and their combinations by pressing push-buttons. More importantly, the system's audio codec is configured using an I2C bus requiring clock-accurate signal handling. Both HLS and VHDL versions of the system were created for comparison.

We used Altera DE2 FPGA board as the platform for our application. It provides an FPGA chip, buttons and leds for user interaction, and an audio codec. Fig. 1 shows the architecture of the system. The main components are the Altera Cyclone II FPGA chip and Wolfson Microelectronics WM8731 audio codec. The FPGA contains the audio synthesizer, the I2C controller, and a phase-locked loop (PLL) circuit as independent blocks.

The synthesizer contains four triangle wave generators that are controlled by corresponding push-buttons on the FPGA board. Pressing a button causes the wave generator connected to it to start generating the signal, and releasing a button resets the signal to zero until it is pressed again. Each wave generator is configured to generate a signal of different frequency. The output of each generator is connected to an adder unit, which adds the signals together but drops the two most significant bits from the result to conform to the audio codec's bit width. This effectively results in overflow causing distorted sound if many buttons are pressed simultaneously. However, sound quality was not our focus in this study.

The adder output is fed to an audio controller which performs a parallel-to-serial conversion of the data. Since we use mono sound, the same data snapshot is taken into both left and right channel inputs at the start of each sample cycle. The audio controller generates two clock signals and a data signal that are fed to the external audio codec. The left-right clock signal (aud_lrclk_out) informs the codec whether left or right channel data is present in its input and the bit clock signal (aud_bclk_out) signifies a bit change in the serial audio data signal.

The I2C controller configures the audio codec when the system reset is released. It writes ten configuration bytes into internal registers of the codec using the I2C bus. For that purpose, a serial data (sdat_inout) and clock signal (sclk_out) are generated for the bus in the controller. The serial data line is bidirectional requiring an inout port, which introduced some interesting design issues that are discussed later.

The PLL circuit on the FPGA was generated using a configurable Altera megafunction block. The purpose of the PLL is to generate an 18 MHz clock signal for the audio synthesizer from a 27 MHz signal provided by the oscillator chip on the FPGA board. The I2C controller directly uses a 50 MHz clock signal from the oscillator.

The architecture in Fig. 1 is the one used in the VHDL implementation of the system. With the HLS implementation, it was discovered that the architecture could be simplified as described next.

## III. HLS DESIGN FLOW

In this work, we used Mentor Graphic's Catapult 8.0 tool (University Version) for HLS [18]. Catapult accepts as a source language either ANSI C++ or SystemC and produces VHDL and Verilog output. For this study, we chose C++ as the input language as it is inherently timeless and more familiar to most programmers, thus lowering the learning curve and easing adoption.

The first step was to write a C++ description of the system. Initially, the system was partitioned as with the VHDL version (Fig. 1), but it was soon discovered that this was unnecessarily complicated for the HLS paradigm. For example, the adder shown in Fig. 1 does not require its own sub-block as it is a simple addition operation in C++. It was eventually decided that the simplest way was to implement the audio synthesizer in its entirety as a single design unit. The I2C controller remained a separate unit as in the VHDL design.

Test benches were written at the same time with the system source codes. The first idea was to use the same VHDL test benches as for the reference VHDL implementations, but this was not possible as they contained generic parameters, which Catapult does not generate into its VHDL output. However, creating C++ test benches was also beneficial since functional correctness could be verified on the source code level instead of the RTL level. Time was also saved in this way as Catapult did not need to transform all source code iterations into VHDL descriptions. On the other hand, writing the test benches took an amount of time comparable to writing the source codes themselves. After verifying the functionality with C++ simulation, RTL simulation was an easy task. Catapult automatically creates an RTL test bench from the C++ version and checks that both representations create the same output.

The modules were synthesized using Mentor Graphic's Precision Synthesis software, which can be directly invoked from Catapult. The clock frequencies used in the synthesis were the same as shown in Fig. 1. The synthesized designs were mapped to physical resources on the FPGA using Altera's Quartus II program, which was also used for synthesizing the connected system and programming the FPGA.

Catapult was unable to create a single inout port for the I2C data signal, but generated separate input and output ports instead along with a data direction signal. Hence, it was necessary to create an inout buffer in Quartus II to connect the I2C data pin to the controller as shown in Fig. 2. For this purpose, an ALT_IOBUF wrapper component was used with the SDAT_DIR signal controlling the direction of data.

The functionality of the system on FPGA was finally verified by sending different configuration values to the audio codec and checking that they had the desired effect on the sound.

## IV. IMPLEMENTATION DETAILS

Writing the HLS source code for the synthesizer part of the system was rather straightforward since it is data path oriented.

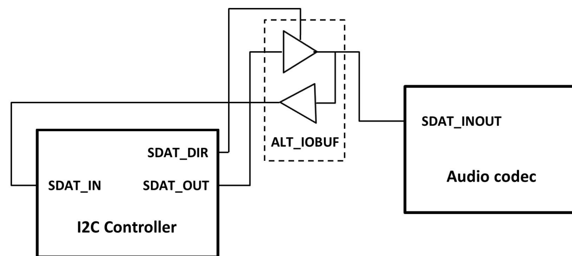


Fig. 2. Serial data connection between the I2C controller and the audio codec.

Because the synthesizer was created as a single design block, Catapult was able to schedule its whole operation within one clock cycle without pipelining.

### *A. The I2C controller functionality*

When the system reset is lifted, the I2C controller automatically sends ten configuration values to the audio codec, setting parameters such as volume, audio data format, and sampling rate. These values are sent as sets of three bytes on the I2C bus. The first byte contains the device address and a read/write-bit. The second byte defines the internal register address in the codec and the third byte the configuration value.

A sample data transfer on the I2C bus is shown in Fig. 3. The I2C controller creates a clock signal (SCLK) that synchronizes the transfer. We operated the I2C bus in standard mode, which defines a maximum SCLK frequency of 100 kHz, and created a 50 kHz SCLK signal. The data signal (SDAT) is bidirectional, i.e., it can be driven by both the I2C controller and the audio codec. When the controller-created SDAT_DIR signal is high, the I2C controller drives the SDAT line, and when it is low, the audio codec drives it. This signal is not specified in the I2C standard but was necessitated by Catapult's inability to create a true inout port.

The data transfer begins with a start condition, which is defined as a high-to-low transition on the SDAT line while SCLK is high. This is followed by eight data bits forming a byte which is sent most significant bit first. The bit values are valid during the high period of the SCLK signal. After a byte has been sent, the I2C controller releases the SDAT line for the next high period of the SCLK. During this period, the audio codec acknowledges the reception by pulling the SDAT line low. If the audio codec, for some reason, is not able to receive or parse the data, the SDAT line is driven high during this clock period. In this case, the audio controller sends all the bytes of the current configuration value again until it receives an acknowledge signal. After the third byte has been successfully sent, the controller creates a stop signal by a low-to-high transition on the SDAT line while SCLK is high. After this, another configuration value can be sent by creating a new start signal followed by the data bytes.

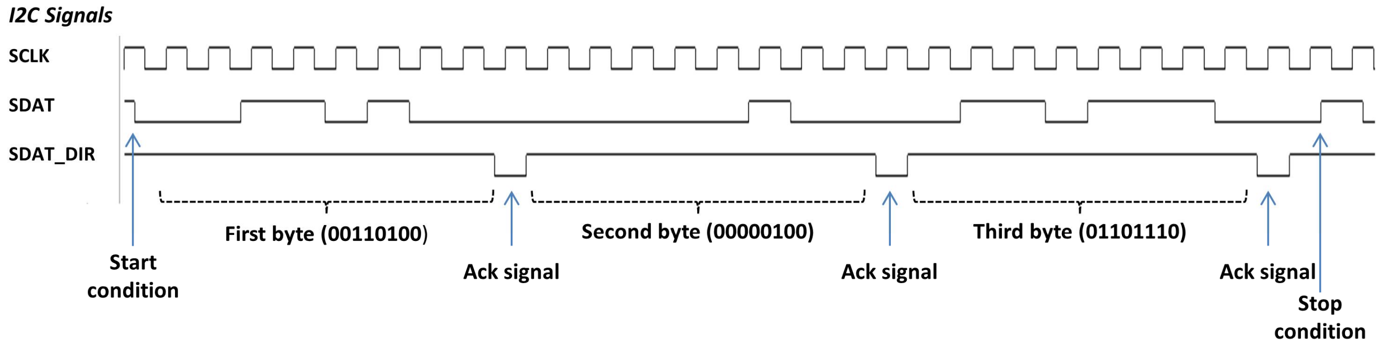


Fig. 3. Complete I2C data transfer cycle for the codec.

### B. *Catapult implementation*

In VHDL, creating the I2C controller is straightforward. The code explicitly defines the system clock signal, which is used to synchronize the operations. With C++, even in HLS environment, there is no such clock signal on the source code level. We needed to overcome this restriction for our application with the intent that the solution is also generalizable to other control oriented systems.

Catapult interprets the *main* function of the source code as the system top-level block. Its function arguments are interpreted as the system's input/output ports depending on whether they are read or written to in the code. Functions that are called in the *main* function can either be inlined to the top level or treated as sub-blocks. The I2C controller consists only of the *main* function.

During scheduling, Catapult chains the operations in the code together and divides them into different clock periods if they cannot be performed during a single period with the given clock frequency. However, when writing the source code, the designer is unware of how the operations implied by it will be scheduled and thus accurate control of timing is difficult.

One solution would be to write the code initially so that the whole system is assumed to be scheduled during one clock cycle and then iteratively change the code according to the real clock cycle count. However, this solution turned out to be problematic in our case as described later.

We created the SCLK signal using the code shown in Fig. 4[1]. Declaring the variables as static preserved their value during different calls to the I2C controller function. The *sclkCounter* variable is incremented during each iteration of the function, and when it reaches the limit value, the variable holding the SCLK value is toggled from 0 to 1 or vice versa. The limit value is calculated by dividing the system clock frequency by the doubled I2C bus frequency, since SCLK value is flipped every half cycle.

The rest of the I2C controller *main* function contains a finite state machine implemented as a switch-case structure, with different states for sending the start condition, sending

```
static bit_t sclkValue;
static int sclkCounter = -1;
const int sclkLimit = REF_CLK_FREQ / (I2C_FREQ * 2);
sclkCounter++;
if (sclkCounter == sclkLimit) {
        sclkCounter = 0;
        sclkValue = (sclkValue == 0) ? 1 : 0;
}
```

Fig. 4. Source code for SCLK generation.

data bits, listening to the acknowledge signal, sending the stop condition, and for staying idle after sending all the configuration values. This structure was functionally identical to the one in the VHDL version and contained a comparable amount of code. The state transitions are primarily dictated by changes in the SCLK signal. Output is written at the end of the function.

### C. *Scheduling results and I/O behavior*

After running the Catapult scheduling for the I2C controller function, it was discovered that with the operating frequency of 50 MHz, it took 2 clock cycles to complete each iteration. This meant that the effective I2C bus frequency would have been half the intended, i.e., 25 kHz. While the bus can operate normally at this frequency, we wanted to maintain the desired frequency, as meeting the timing specification was the main purpose of our study. By dividing the *sclkLimit* value by two, we could have effectively reached the intended SCLK frequency, but this was prevented by the I/O behavior of our system.

As was mentioned in Section 3, Catapult synthesized separate in and out ports for the SDAT signal, which is a true bidirectional signal with three-state logic in the VHDL implementation. To fix this, we instantiated the ALT_IOBUF wrapper component in Quartus II as shown in Fig. 2. The data direction is driven by the separate SDAT_DIR signal, which was synthesized by Catapult for the I2C controller. This signal is driven high when output (SDAT_OUT) is written. However, with the throughput period of 2 clock cycles, the output is written only every second clock period when the end of the

[1] In the actual code, bit-accurate Algorithmic C data types were used, but they are omitted here for clarity

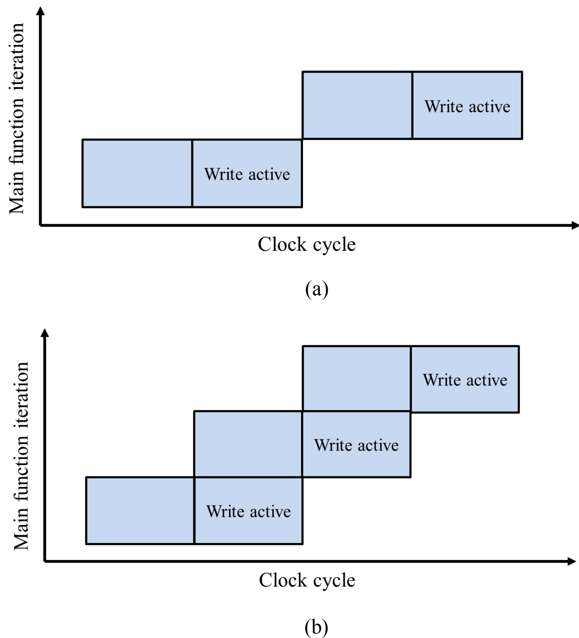


Fig. 5. Schedule of a system with a throughput period of 2. (a) Unpipelined schedule. (b) Pipelined schedule.

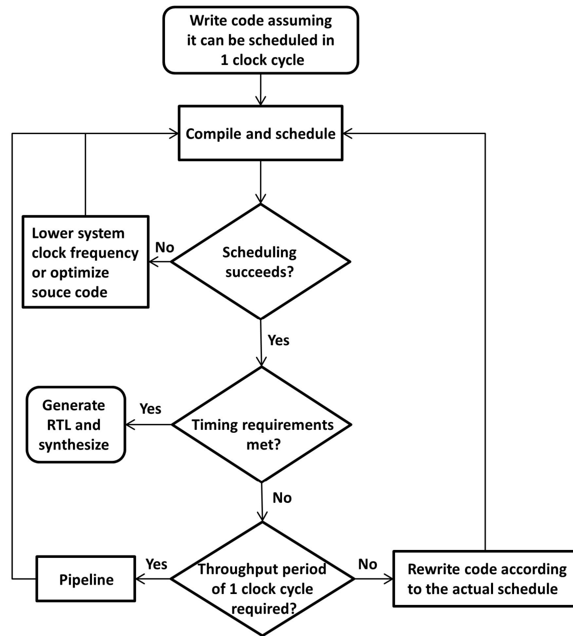


Fig. 6. Workflow for creating clock accurate systems with HLS.

*main* function is reached. This causes SDAT_DIR to oscillate, when the controller is driving a value to the SDAT signal. The oscillation could be eliminated by adding additional logic between the I2C controller and the ALT_IOBUF component, but that would be an ad hoc solution.

### *D. Pipelining*

Fortunately, Catapult and other state-of-the-art HLS tools provide an easy way to control pipelining for loop structures, and the program's *main* function is treated as a loop with an infinite number of iterations. Fig. 5 shows the difference between an unpipelined and a pipelined solution in a system with a throughput period of 2, as was the case with our I2C controller. In Fig. 5(a), the unpipelined solution is shown. Each iteration of the *main* function takes 2 clock cycles. Write is active every second clock cycle causing the SDAT_DIR signal to oscillate in our case.

Fig. 5(b) shows the pipelined version. Each iteration still takes 2 clock cycles, but with a 2-stage pipeline, two consequent iterations overlap and one write is always active causing stable output behavior. Furthermore, with pipelining, one does not need to change the source code since the operation incrementing the *sclkLimit* variable is now performed every clock cycle.

It should be noted that the pipelining solution works even if the unpipelined throughput period were 3 clock cycles or more. After pipeline ramp-up, more iterations would be active concurrently but a new write would still execute every clock period.

## V. Recommendations

Based on this study, we suggest the work flow depicted in Fig. 6 for designing clock cycle accurate systems. First, the HLS source code is written with the assumption that it can be scheduled within one clock cycle. After compiling and scheduling, it is checked if this was actually the case. If not, there are two options. The code can either be rewritten to take the real clock cycle count into account or the system can be pipelined. Rewriting the code is preferable to pipelining especially in resource-constrained systems but if a throughput period of one clock cycle is required as in our example, then pipelining cannot be avoided.

If the code is rewritten, it is possible that the cycle count of one iteration changes. In this case, the code has to be rewritten again to accommodate the new cycle count. The aim is to iterate towards a converging cycle count so adding or removing time-consuming operations should be avoided when rewriting the code for this purpose.

As a concrete example of code rewriting, let us return to the code snippet shown in Fig. 4 and assume that the I/O direction signal oscillation discussed in the previous Section was not an issue. In this case, to achieve the correct SCLK frequency with the unpipelined schedule shown in Fig. 5(a), we could simply divide the *sclkLimit* constant by two. The *sclkLimit* is a constant value determined at compile time, so this would not change the cycle count taken by one iteration of the I2C controller function. The rewriting process would thus converge immediately.

With more complex applications, the scheduling can fail under the system clock frequency constraint. This can happen either before or after applying pipelining. In this case, the

frequency must be lowered or the source code optimized for allowing scheduling under tight clock frequency limit. For example, reducing the bit width of data could reduce the delay of arithmetic operations, albeit at the expense of accuracy.

## VI. RESULTS AND EVALUATION

The area results for both the VHDL and Catapult version of the system are shown in Table 1. Both solutions were synthesized using the clock frequencies shown in Fig. 1. In total, the Catapult version consumes about 84% more logic cells (LCs) on the FPGA than the VHDL version. The audio synthesizer takes 50% more area and the I2C controller 155% more area. The relatively larger area of the I2C controller is due to the pipelining that requires additional resources for pipeline registers, computation, and control. If pipelining were not necessary, the I2C controller would have consumed only 72% more LCs (the figure in parentheses in the Table), which is similar to the figure with the audio synthesizer.

The lines of code of the Catapult version including test benches is roughly half of that of the VHDL version. It can be approximated that the amount of code inversely correlates with productivity, and thus HLS productivity would be twice the RTL productivity with this kind of application. Furthermore, design space exploration and making modifications is much faster with HLS. For instance, pipelining was simply a matter of a few mouse clicks in Catapult, whereas it would have required a significant code revision with the VHDL version.

TABLE I. COMPARISON BETWEEN THE VHDL AND CATAPULT VERSIONS OF THE SYSTEM

| | **VHDL** | **Catapult** | **Difference** |
|---|---|---|---|
| Audio synth. area | 233 | 349 | 50% |
| I2C ctrl. area | 110 | 281 (189) | 155% (72%) |
| Total area | 343 | 630 | 84% |
| Lines of Code | 1400 | 700 | -50% |

## VII. CONCLUSION

In this paper, we have shown how HLS can be used to implement a clock-accurate system. As is common for HLS with the contemporary tools, the design consumes more area than an RTL implementation, but on the other hand there is a considerable boost in productivity, and design space exploration is much faster. Furthermore, it requires less expertise to use HLS than RTL methodology, since learning a hardware description language is more time consuming than learning a HLS tool flow. We thus recommend considering HLS as a viable alternative to RTL methodology even with control oriented applications unless there are strict area requirements.

In the future, we will create a more complex control-oriented application using HLS. One such application would be a multi-module system with registers between the sub-blocks. The accurate timing of such a system should be studied block by block. In the presented system, there are also no external control signals that affect the I2C controller after reset has been lifted. Examining the effect of such signals on design considerations of clock-accurate systems is of great interest.

## REFERENCES


[1] G. Martin and G. Smith, “High-level synthesis: past, present, and future,” IEEE Des. Test Comput., vol. 26, pp. 18-25, July-Aug. 2009.

[2] H. Ren, “A brief introduction on contemporary high-level synthesis,” in 2014 IEEE Int. Conf. IC Design & Technology, Austin, TX, 2014.

[3] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers and Z. Zhang, “High-level synthesis for FPGAs: from prototyping to deployment,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 30, pp. 473-491, Apr. 2011.

[4] P. Coussy, D. D. Gajski, M. Meredith and A. Takach, “An introduction to high-level synthesis,” IEEE Des. Test Comput., vol. 26, pp. 8-17, July-Aug. 2009.

[5] Y. Pan, S. K. Pilakkat, K. B. Tan and W. Mok, “A user's reflections on the art of high level synthesis,” in 2014 14th Int. Symp. Integrated Circuits, Singapore, 2014, pp. 67-70.

[6] S. Skalicky, C. Wood, M. Lukowiak and M. Ryan, “High level synthesis: where are we? A case study on matrix multiplication,” in 2013 Int. Conf. Reconfigurable Computing and FPGAs, Cancun, 2013.

[7] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do and D. Chen, “High-Level Synthesis: productivity, performance, and software constraints,” Journal of Electrical and Computer Engineering, Article ID 649057, 14 pages, 2012. doi:10.1155/2012/649057

[8] T. Damak, I. Werda, N. Masmoudi and S. Bilavarn, “Fast prototyping H.264 deblocking filter using ESL tools,” in 2011 8th Int. Multi-Conf. Systems, Signals and Devices, Sousse, 2011.

[9] T. Hussain, M. Pericàs, N. Navarro and E. Ayguade, “Implementation of a reverse time migration kernel using the HCE high level synthesis tool,” in 2011 Int. Conf. Field-Programmable Technology, New Delhi, 2011.

[10] L. Piga and S. Rigo, “Comparing RTL and high-level synthesis methodologies in the design of a theora video decoder IP core,” in 5th Southern Conf. Programmable Logic, Sao Carlos, 2009, pp. 135-140.

[11] E. Homsirikamol and K. Gaj, “Can high-level synthesis compete against hand-written code in the cryptographic domain? A case study,” in 2014 Int. Conf. ReConfigurable Computing and FPGAs, Cancun, 2014.

[12] G. Wang, H. Lam, A. George and G. Edwards, “Performance and productivity evaluation of hybrid-threading HLS versus HDLs,” in 2015 IEEE High Performance Extreme Computing Conf., Waltham, MA, 2015.

[13] D. J. Pagliari, M. R. Casu and L. P. Cartoni, ”Acceleration of microwave imaging algorithms for breast cancer detection via high-level synthesis,” in 33rd IEEE Int. Conf. Computer Design, New York, NY, 2015, pp. 475-478.

[14] T. Ahmed, N. Sakamoto, J. Anderson and Y. Hara-Azumi, “Synthesizable-from-C embedded processor based on MIPS-ISA and OISC,” in *IEEE 13th Int. Conf. Embedded and Ubiquitous Computing*, Porto, pp. 114-123.

[15] Q. Zhu and M. Tatsuoka, “High quality IP design using high-level synthesis design flow,” in *21st Asia and South Pacific Design Automation Conf.,* Macau, 2016, pp. 212-217.

[16] M. Sharafeddin *et al.*, “On the efficiency of automatically generated accelerators for reconfigurable active SSDs,” in *26th Int. Conf. Microelectronics*, Doha, 2014, pp. 124-127.

[17] NXP Semiconductors, Eindhoven, Netherlands. *I2C-bus specification and user manual.* (2014) [Online]. Available: http://www.nxp.com /documents/user_manual/UM10204.pdf, Accessed on: Jul. 6, 2016

[18] Mentor Graphics, Wilsonville, OR, USA. *Catapult High-Level Synthesis.* [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/, Accessed on: Jul. 6, 2016.