From High-Level Synthesis to Bundled-Data Circuits

Yoan Decoudu[1(✉)], Jean Simatic[2], Katell Morin-Allory[1], and Laurent Fesquet[1]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, 38000 Grenoble, France
{yoan.decoudu,katell.morin,laurent.fesquet}@univ-grenoble-alpes.fr
[2] HawAI.tech, 38000 Grenoble, France
jean.simatic@hawai.tech

Abstract. In order to spread asynchronous circuit design to a large community of designers, High-Level Synthesis (HLS) is promising option because it requires limited technical skills. Common HLS operations quickly provide a synchronous RTL description, which is usually split in two parts: a data-path and a control-path. In order to desynchronize such a circuit, the desynchronization process is only applied to the control-path, which is no more than a Finite State Machine (FSM). Our approach helps designers for quickly designing data-driven circuits while maintaining a reasonable cost, a similar area and a short time-to-market. To demonstrate our technique, the HLS tool, Catapult HLS from Mentor Graphics, has been used. Once the control-path has been extracted, the corresponding FSM is simply analyzed and desynchronized. On the other hand, the data-path is kept as it is. The resulting circuit is a bundled-data circuit requiring a particularly low design effort. Some samples illustrate the method and show its relevance in terms of area and performance.

Keywords: Event-driven circuits · Desynchronization · Low-power circuits · High-level synthesis

1 Introduction

Asynchronous circuits are today considered as relevant alternatives to synchronous design for many purposes. Indeed, unlike typical synchronous architectures, they have local synchronizations instead of a global synchronization signal. This brings several advantages depending on the implementation template: a reduced dynamic power consumption, robustness [2], low-voltage operations [9] or security. Despite all these favorable characteristics asynchronous circuits are not today widely spread in the industry, probably due to the lack of dedicated knowledge, know-how and EDA tools.

High-Level Synthesis (HLS) enables fast circuit design from a high-level description. This description, usually written in a C-like language, is compiled in order to synthesize a Register Transfer Level (RTL) description. The mostly

Grenoble INP–Institute of Engineering Univ. Grenoble Alpes

A. Orailoglu et al. (Eds.): SAMOS 2020, LNCS 12471, pp. 200–212, 2020.
https://doi.org/10.1007/978-3-030-60939-9_14

used commercial HLS tools cover from ASIC to FPGA implementations, but only generate classical synchronous circuits. For large circuits, this approach helps to explore several architectures and meet the required performances in terms of area, power and speed. Moreover, HLS demands few technical skills in hardware design. Therefore, this approach is ideal for spreading asynchronous circuits in the industry: it is an automated method, which implements asynchronous circuits without important changes in the standard design flow.

This paper proposes a new design method for synthesizing bundled-data circuits based on the use of most synchronous HLS tools. Our approach uses the HLS tool, Catapult HLS from Mentor Graphics, that partitions the resulting circuit into a data-path and a control-path. Then we only desynchronize the control-path, which is replaced by a specific asynchronous Finite State Machine (FSM). This leads to an asynchronous circuit synthesis requiring no specific knowledge on bundled-data circuits while taking advantage of the asynchronous logic features. Section 2 presents the bundled-data circuit principles and the related works on the HLS. Section 3 describes the proposed desynchronization method. Section 4 applies our method on a FIR Filter and a GCD calculator and then compares the synchronous circuits generated from Catapult and their asynchronous counterparts.

2 Related Works

2.1 Bundled-Data Circuits in a Nutshell

In this paper, we focus on a specific class of asynchronous circuits: the bundled-data circuits. In the sequel, we give a brief overview of this class of circuits [17,18].

Bundled-data circuits look very similar to synchronous circuits but the clock tree has been removed and replaced by a control circuit, whereas the data-path is kept as it is (see Fig. 1). The control circuit is locally composed of distributed controllers communicating with each others thanks to a 4-phase handshake protocol.

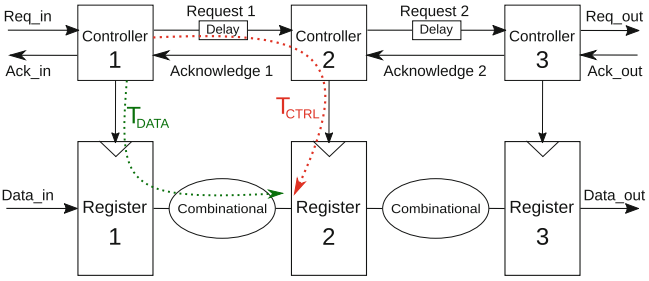

Fig. 1. Bundled-data circuit architecture.

The local timing assumptions in bundled-data circuits are between two connected controllers. Indeed, as shown in Fig. 1, the data must be sampled by the

registers of the next pipeline stage register once the data computation is completed. Therefore the delay of the control path T_{CTRL} has to be longer than the computation in the data-path T_{DATA} in Fig. 1. Thus, it is necessary to add a delay element on the request signal for covering the logical gate delays. In the control path, the controllers are made with C-elements or Muller gates. Figure 2 shows its symbol and its truth table. This component allows the synchronization between 2 signals.

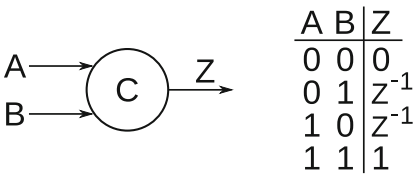

Fig. 2. C-element truth table and symbol.

Late-Capture Protocol. There are many handshake protocols for bundled-data circuits offering different advantages. Most of them activate the registers when the request signal goes high. In order to meet the timing assumption requirements, it is mandatory to have a delay element greater than the critical logical path. In a 4-phase protocol, the cycle time is two times the delay because the request rising edge is propagating through the delay but also its falling edge during the return-to-zero phase. This leads to an important speed drop (2 times) compared to the synchronous version.

In order to be more efficient, a protocol activating the registers on the request falling edge is preferable such as the late-capture protocol [13]. Figure 3 shows the late-capture waveform. The request signal rising and falling edges are propagating through the delay before activating the registers. In this way, the delay is half of the delay required for a protocol activating the registers on its rising edge. This has two advantages: the speed is maintained as in synchronous circuits and the delay power consumption is minimized.

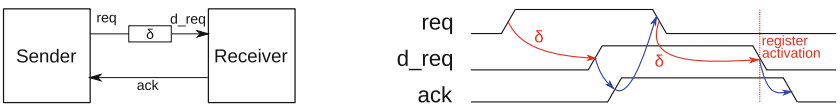

Fig. 3. Handshaking signals with late-capture protocol.

Figure 4 presents the Signal Transition Graph (STG) [3] of the late-capture protocol and its delay-insensitive implementation. This protocol decouples the handshake protocol at the input and at the output of the controller to enhance

the speed of the communication. Hence, the late-capture protocol waits for the deactivation of the request R_{in} before activating the output request R_{out}.

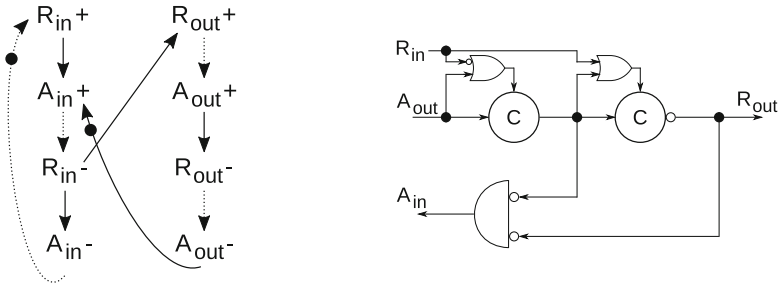

Fig. 4. (a) STG of a late-capture protocol. (b) Late-capture controller implementation.

2.2 High-Level Synthesis

HLS tools usually compile a high-level circuit description with no architectural nor timing information. This high-level description is synthesized into a RTL code. In addition, HLS is able to perform architectural exploration to optimize the design according to the design constraints. HLS is typically an attractive way to quickly design synchronous circuits implementing algorithms [4]. In case of asynchronous circuits, several tools already exist [16]. These tools use three different synthesis strategies.

Syntax Directed Translation. This strategy uses a dedicated high-level hardware description language (HDL) and map the code syntactic structures onto hardware components. For instance, the synthesis tools TiDE [10] and Balsa [1] translates the language syntax into handshake components. Circuit speed and area depend a lot on the designer's ability to write an optimized HDL code.

Pipelined Process Decomposition. HLS tools use intermediate representations of the algorithm to abstract and capture the function as independently as possible from the syntax. Data-driven decomposition [20] and CASH compiler [19] respectively use dynamic and static single assignment forms. These approaches are suitable for high-throughput applications as they generate highly pipelined components. But they are ill-adapted to typical IoT applications, which require low-power circuits and can afford low-speed.

Scheduling Based Flows. To enable optimization strategies, synchronous HLS is decomposed in distinct yet interdependent steps [4]. The main technical steps are allocation and scheduling, which can be done by different approaches

depending on the design objectives. Most of the HLS tools use a control/data-path decomposition style. This approach fits well the HLS decomposition because the data-path results from the allocation, and the control-path from the scheduling. Following this model asynchronous HLS raises two issues:

- The scheduling time is continuous for asynchronous circuits. Thus, the optimal asynchronous scheduling may be different. An efficient algorithm is proposed in [8] that solves this question for two optimization criteria.
- For implementing the asynchronous FSM, there are many proposed architectures including locally-clocked [5,11] with optimized state coding, and one-hot state coding [15] allowing direct mapping from the state graph.

To the best of our knowledge, only the BUDASYN [5] flow, which only targets FPGA platforms, integrates a solution to both problems by implementing the algorithm of [8]. The FSM is a centralized locally-clocked implementation of an Extended Burst Mode (XBM) specification [21].

For asynchronous circuits, HLS tools help bridging the gap between circuit designer skills and the specific asynchronous techniques. Our approach is also based on a scheduling flow. However, the FSM is implemented thanks to interconnected distributed controllers. Compared to centralized implementations, such as locally-clocked FSMs, distributed AFSMs are likely less compact but offer a better scalability, ease the place and route operations, and are more appropriated for fine-grain pipelining.

This work is an improvement of the work presented in [12]. Moreover, our method is applied on a commercial tool, which allows more design possibilities. It relies on the synchronous tool Catapult from Mentor Graphics [7] for scheduling and allocation. It provides support for a standard programming language (a subset of ANSI C) and a wide set of possible transformations such as loop unrolling and pipelining. Catapult generates a synchronous synthesizable RTL code of the data-path and synchronous FSM. In order to generate an asynchronous control, the synchronous FSM is simply desynchronized.

3 Desynchronization Method

The HLS tool Catapult from Mentor Graphics generates a synchronous circuit divided in two parts: a control- and a data-path. This section presents the implemented method for desynchronizing the circuit just by replacing its control-path by its asynchronous counterpart. Therefore, the only modification of the data-path is the renaming of the flip-flop clock signal.

3.1 Desynchronization Principle

Figure 5 shows how the control-path and the data-path are interconnected within Catapult. The control-path activates multiplexers and enable signals on the flip-flops in the data-path. Each state is one-hot encoded. A dedicated signal controls the activated part of the data-path, represented by the signals C^M in Fig. 5.

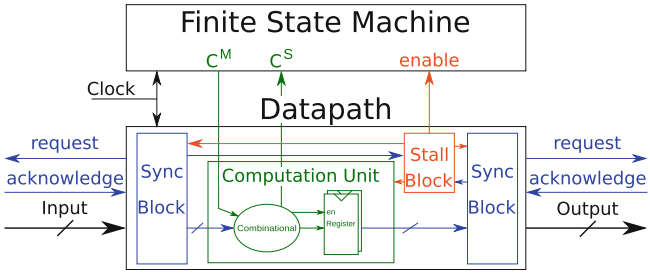

Fig. 5. Circuit architecture generated from catapult.

The next state is chosen at each cycle thanks to the signals C^S coming from the combinational part of the data-path.

In a circuit generated with Catapult, the synchronization between the system and the I/O is done thanks to a Synchronization Block (*Sync Block* in Fig. 5) and its handshake signals inserted at the input and output of the data-path. If the external environment does not activate the Synchronization Block, the *Stall Block* freezes the whole circuit including the FSM thanks to the signal *enable* in Fig. 5.

To desynchronize such a circuit, the FSM is replaced by an Asynchronous Finite State Machine (AFSM). In order to generate an AFSM, the RTL description is parsed to extract the FSM states. Thanks to this AFSM representation, a Petri Net modeling its behavior is generated. Then the Petri Net model is used for validating that the AFSM has no deadlocks and its liveliness is formally ensured. Once these verifications are done, an AFSM netlist is generated with late-capture controllers. The data-path is kept as it is. The unique modification removes the clock tree and substitutes the clock by the signals coming from the AFSM. By only substituting the FSM by an AFSM and connecting it to the data-path as presented in Fig. 6, the whole circuit behavior becomes asynchronous. Thus, the registers are only activated when needed (depending on the current AFSM state) while the computation results remain the same.

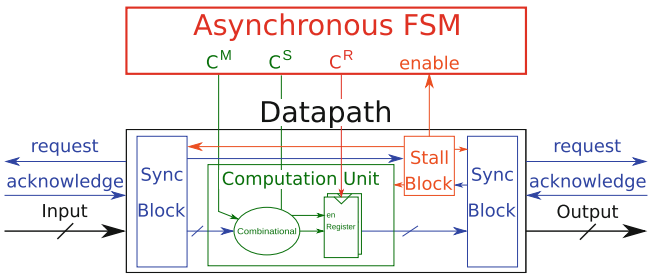

Fig. 6. Circuit architecture after the desynchronization.

Notice that there are now two types of control signals driven by the AFSM. The C^M signals activate the combinational part and the C^R signals directly activate the registers (in replacement of the clock).

3.2 AFSM Architecture

Each AFSM state corresponds to a controller. When there are several successors (respectively predecessor) a demultiplexer (respectively a multiplexer) component is used. Only one controller is activated at one because this latter represents a FSM state. Thus, the handshake protocol needs deactivating the previous controller before activating the next one. This is also a reason for choosing the late-capture protocol. The state activation begins with the request signal of the protocol. As the combinational part processes the data during the whole state duration, the signal C^M is activated on the request rising edge as shown in Fig. 7. According to our protocol, the data capture (represented by C^R) is initiated with the delayed request deactivation. Then the deactivation of C^M comes with the falling edge of the acknowledgement signal *ack*.

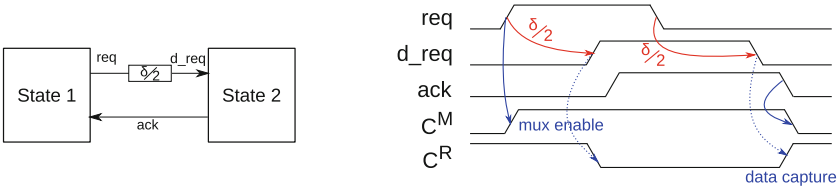

Fig. 7. State activation according to the handshake protocol.

Figure 8 shows an AFSM controller and its connection to the data-path. The signal C^R is connected after the delay on the request wire to ensure the local timing assumptions. As the registers sample on a rising edge, an inverter is added. Once the request is activated, the signal C^M immediately launches the computation. In this way, the local data-path has two times the delay duration for computing the data. The reset of C^M is done by an OR gate between the request and acknowledgement. As shown on Fig. 6, the AFSM can be frozen by the *Stall Block*, which also deactivates the whole data-path when no new data are available on the *Sync Block*. For this purpose, a latch has been added after each controller for disabling the request. Its clock pin is connected to the *Stall Block* output to be in transparent mode during the circuit operation and in latch mode when the circuit is idle.

In Fig. 8, the next states are chosen thanks to a demultiplexer and a selection signal C^S coming from the data-path. The design of a demultiplexer implementing the late-capture protocol is given on Fig. 9a. Notice that the selection signal is always valid when the request is deactivated. Therefore the selection coming from the data-path is not yet arrived on the request rising edge, which is propagated without control to the next branches. Thanks to a register, a delay and

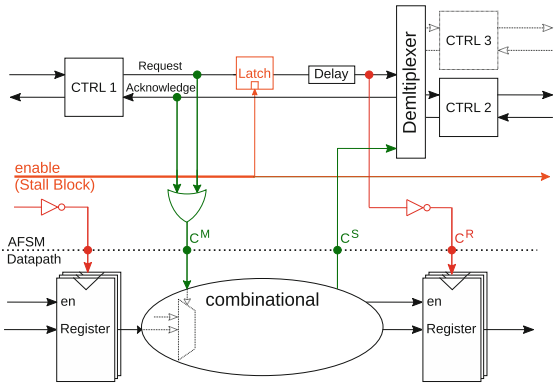

Fig. 8. Asynchronous FSM architecture.

a NAND gate, the request falling edge will only be transmitted to the selected branch. The delay ensures that the selection signal arrives first at the NAND input, so that the demultiplexer only activates one controller. A similar property should also be guarantied by the multiplexer. The latter just propagates the incoming request and acknowledges the adequate successor. Its implementation is given in Fig. 9b. Compared to the demultiplexer letting the request rising edge propagating to the next branches, the combinational logic transmitting the request to the next controller is not so obvious. Indeed, this functionality is obtained thanks to the equation shown in Fig. 9b.

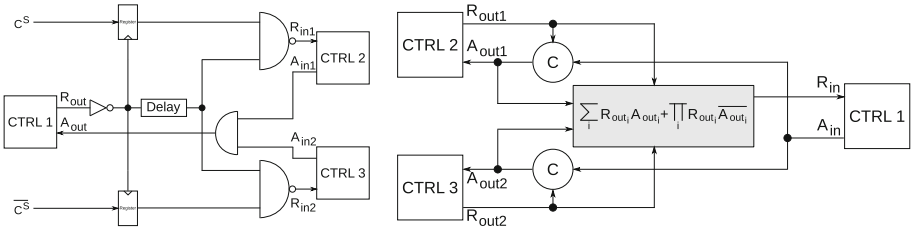

Fig. 9. (a) Demultiplexer architecture. (b) Multiplexer architecture.

The AFSM liveliness can be proven thanks to its Petri net model [14]. As only one token, representing the current state, evolves in the Petri net, the multiplexer and demultiplexer must generate only one token at their output. The deadlocks can also checked. In order to guarantee their absence, note that it is required to add a controller in each one-controller loop. This added controller will not be associated to C^M nor C^R. This case only happens when a state loops on itself.

4 Testcases: FIR Filter and GCD Calculator

In order to demonstrate our design flow, this section presents the desynchronization flow applied to a Greatest Common Divisor (GCD) calculator and a Finite Impulse Response (FIR) Filter. They are first generated with Catapult and then desynchronized. The area and speed performances have been evaluated in the FD-SOI 28 nm technology from STMicroelectronics.

4.1 GCD Calculator

The GCD calculator uses the Euclid's algorithm (given in Fig. 10a) to compute the GCD of two input numbers. The circuit FSM generated by HLS is shown in Fig. 10b. State 1 recovers the input data, State 2 computes the data and State 3 sends the result. When desynchronizing, a Petri net of the FSM is constructed to verify the liveliness and the absence of deadlocks. The self-loop on State 2 requires an additional controller to avoid deadlocks. Two GCD calculators have been implemented: a 8-bit and a 64-bit. For both implementations, the control path is exactly the same.

```
#pragma hls_design top
void gcd(
  ac_channel<dType> &in1,
  ac_channel<dType> &in2,
  ac_channel<dType> &out) {

   dType a = in1.read();
   dType b = in2.read();
   while(b > 0){
     if(a >= b){
       a -= b;
     }
     else{
       dType r;
       r = b;
       b = a;
       a = r;
     }
   }
   out.write(a);
}
```

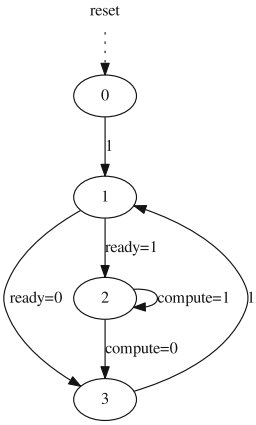

Fig. 10. (a) Source code of the GCD calculator. (b) GCD Calculator FSM.

4.2 FIR Filter

FIR Filters are very usual digital circuit. They process sampled data with filter coefficients. Let x_i the i^{th} sample of the input signal and h_i the i^{th} coefficient of the filter. The output sample y_k is given by Eq. (1):

$$y_k = \sum_i x_{k-i} h_i \tag{1}$$

This equation has been implemented for a 8-bit filter written in C. We choose a sequential architecture with one multiplier and one accumulator. This leads to the architecture given in Fig. 11a. The samples (resp. coefficients) are stored in DL (resp. ROM) and then computed by MULT/ACC. The generated circuit by Catapult has a FSM with only two states (see Fig. 11b): a reset state 0 and a computation state 1. During the desynchronization, the Petri net is extracted as in the previous example and a controller is added to the self-loop on State 1.

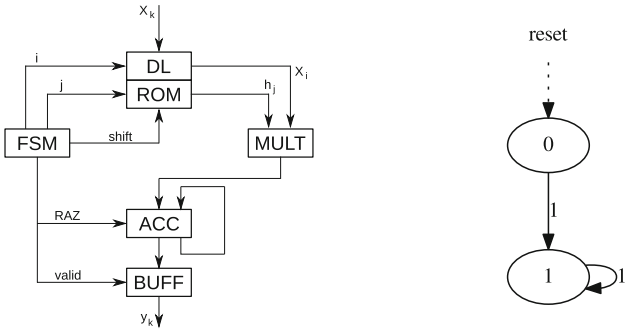

Fig. 11. (a) FIR Filter Architecture. (b) FIR Filter FSM.

4.3 Results

All the three circuits have been synthesized thanks to Design Compiler from Synopsys and validated with back-annotated logical simulations. The timing constraints are resolved thanks to the method described in [6] that takes advantage on the synchronous static timing analysis to check the specific timing constraints imposed by the asynchronous circuits. Thus, this allows us to use traditional EDA tools to synthesize and validate our circuits. Their synchronous counterpart have also been designed in order to compare them in terms of area and speed. For the FIR filter, the input signal is a pulse. For the GCD calculator, a random number generator generates the two input signals. The simulations use sufficiently long stimuli in order to average the results.

Area. Figure 12 reports the area of the circuits presented above. The area of the asynchronous circuits is a little bit larger than their synchronous counterparts. Indeed an asynchronous control circuit replaces the FSM. The extra-area of the AFSM results from a small increase of the FSM complexity. However, looking at the whole circuit, it is important to notice that the clock tree has not been implemented in the synchronous versions of the FIR and GCD circuits. The lack of the clock tree will impact unfavorably the comparison for the asynchronous circuits.

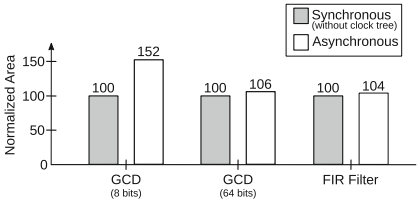

Fig. 12. Areas of the circuits normalized by the synchronous circuits.

Figure 12 reveals that the higher area increase is obtained with the 8-bit GCD calculator. The area is increased of 52% compared to its synchronous counterpart but, in this case, the tiny data-path is quite simple compared to the AFSM. Hence the area increase is huge. Changing the area ratio between the data-path and the control-path gives a completely different view. For the GCD, a 64-bit GCD only gives a 6% growth of the area. For the FIR filter, the FSM is rather simple and the data-path is quite huge compared to the control circuit. Therefore the area is only of 4% larger. As HLS is commonly used for designing large signal processing circuits, the control-path remains usually small compared to the data-path. Hence the area overhead will be negligible with most of applications designed with Catapult.

Speed. The synchronous circuit speed is imposed by the critical path, while, in asynchronous circuits, the speed results from an average of the stage speeds. Therefore the asynchronous circuits are faster than their synchronous counterparts, as shown in Fig. 13.

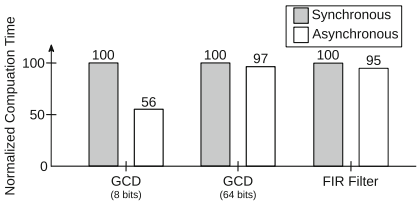

Fig. 13. Normalized computation time of the circuits.

The asynchronous FIR filter is a little bit faster with a speed increase of 5%. As the FSM remains in the same state during the computation (see Fig. 11b), only one controller activates the registers. Hence, the delay of the control circuit matches the circuit critical path. Therefore the speed of the two versions are almost the same.

For both GCD, the critical path corresponds to the computation stage, which is controlled by State 2 (State 2 is fed back on itself). Thus the performance

strongly depends on the iteration number on State 2. For the 8-bit GCD, the computation requires a few iterations, so that the circuit often communicates with its environment activating the other states. Hence the performance enhancement is huge and reaches 44% compared to its synchronous counterpart. For the 64-bit GCD, the computation iteration number is larger and the system spends most of the time in State 2 where the critical path is. The speeds of the synchronous and asynchronous circuits are very similar and we only notice a slight speed increase of 3% for the asynchronous circuit.

5 Conclusion

For most of the designers, asynchronous design is challenging. Nevertheless, asynchronous circuits provide interesting features making them attractive for many applications. The proposed automated design flow takes advantage of the synchronous HLS from Mentor Graphics, Catapult HLS, to achieve asynchronous circuit design with very limited technical skills. This approach avoids learning HDL and keeps the design framework unchanged. This method is an opportunity for non-specialists to quickly design asynchronous circuits in a very standard framework.

The synthesized asynchronous circuits present a little area overhead, which has to be mitigated by the lack of the clock tree in the synchronous versions. The results also show a slight speed increase. As the results are strongly correlated to the FSM and data-path architectures, there is no doubt that a data-path mostly computing outside its critical path will enhance the speed of the asynchronous version. The worst case corresponds to the desynchronized circuits which only activate their critical path (our FIR filter).

Asynchronous circuits are well-suited for processing non-uniformly sampled signals and low-voltage applications thanks to their intrinsic robustness. Future work will target the design flow enhancement and the power analysis of the desynchronized circuits.

Acknowledgment. This work has been partially supported by the OCEAN12 European Project (Ecsel JU, Grant Agreement N°783127).

References

1. Bardsley, A.: Balsa: an asynchronous circuit synthesis system. Master's thesis, University of Manchester (1998)
2. Chang, K.L., Chang, J., Gwee, B.H., Chong, K.S.: Synchronous-logic and asynchronous-logic 8051 microcontroller cores for realizing the Internet of Things: a comparative study on dynamic voltage scaling and variation effects. IEEE J. Emerg. Sel. Top. Circuits Syst. **3**(1), 23–34 (2013)
3. Chu, T.A.: On the models for designing VLSI asynchronous digital systems. Integr. VLSI J. **4**(2), 99–113 (1986)
4. Coussy, P., Gajski, D., Meredith, M., Takach, A.: An introduction to high-level synthesis. IEEE Des. Test Comput. **26**(4), 8–17 (2009)

5. Garcia, K., Oliveira, D.L., d'Amore, R., Faria, L.A., Oliveira, J.L.V.: FPGA implementation of optimized XBM specifications by transformation for AFSMs. In: International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–6 (2016)
6. Gimenez, G., Cherkaoui, A., Cogniard, G., Fesquet, L.: Static timing analysis of asynchronous bundled-data circuits. In: 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), May 2018
7. Graphics, M.: Catapult HLS. Available: http://www.mentor.com
8. Hansen, J., Singh, M.: A fast branch-and-bound approach to high-level synthesis of asynchronous systems. In: 16th IEEE Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 107–116 May 2010
9. Jorgenson, R.D., et al.: Ultralow-power operation in subthreshold regimes applying clockless logic. Proc. IEEE **98**, 299–314 (2010)
10. Nielsen, S., Sparsø J., Jensen, J., Nielsen, J.: A behavioral synthesis frontend to the Haste/TiDE design flow. In: 15th IEEE Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 185–194 (2009)
11. Nowick, S.M., Dill, D.L.: Synthesis of asynchronous state machines using a local clock. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors, pp. 192–197 (1991)
12. Simatic, J., Bastos, R.P., Fesquet, L.: High-level synthesis for event-based systems. In: 2nd International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP), pp. 1–7 June 2016
13. Simatic, J., Cherkaoui, A., Bastos, R.P., Fesquet, L.: New asynchronous protocols for enhancing area and throughput in bundled-data pipelines. In: 29th Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 1–6 Aug 2016
14. Simatic, J., Cherkaoui, A., Bertrand, F., Bastos, R.P., Fesquet, L.: A practical framework for specification, verification, and design of self-timed pipelines. In: 2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp. 65–72 May 2017
15. Sotiriou, C.R.: Direct-mapped asynchronous finite-state machines in CMOS technology. In: 14th Annual IEEE International ASIC/SOC Conference, pp. 105–109 (2001)
16. Sparsø J.: Current trends in high-level synthesis of asynchronous circuits. In: 16th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), pp. 347–350 (2009)
17. Sparsø, J., Furber, S.: Principles of Asynchronous Circuit Design: a Systems Perspective, 1st edn. Springer Publishing Company, Boston (2010)
18. Sutherland, I.E.: Micropipelines. Commun. ACM **32**(6), 720–738 (1989)
19. Venkataramani, G., Budiu, M., Chelcea, T., Goldstein, S.C.: C to asynchronous dataflow circuits: an end-to-end toolflow. In: 13th IEEE International Workshop on Logic Synthesis (IWLS). Temecula, CA, June 2004
20. Wong, C., Martin, A.: High-level synthesis of asynchronous systems by data-driven decomposition. In: Design Automation Conference (DAC), pp. 508–513 June 2003
21. Yun, K., Dill, D.: Automatic synthesis of extended burst-mode circuits.i. (specification and hazard-free implementations). computer-aided design of integrated circuits and systems. IEEE Trans. **18**(2), 101–117 (1999)