

Automated Pipelining in ASIC Synthesis Methodology: Gate Transfer Level

Alexandre Smirnov, Alexander Taubin,
Mark Karpovsky
{alexbs, taubin, markkar}@bu.edu

The paper presents Gate Transfer Level (GTL) as a general framework for synthesis of industrial complexity asynchronous quasi-delay-insensitive (QDI) circuits. The GTL flow automatically provides the finest degree of pipelining (gate-level) resulting in extremely high-performance designs. Automatic gate level pipelining is not possible for synchronous design due to the stage balance problem and clock related overheads (latches, clock skew and jitter). Experimental results show average 4.3x performance increase on MCNC benchmarks compared to synchronous RTL implementation.

1. Introduction

Register Transfer Level (RTL) synthesis method simplified clocked circuitry design and allowed its automation by effectively separating logic optimization from timing. This boosted VLSI industry for more than a decade. However, process variation, signal integrity problems and others physical limitations of synchronous designs are encouraging a search for alternative solutions. Synchronous design is turning out now to become a costly proposition. ITRS has noticed this trend and cited an impending crisis. It predicts that in the near future, the industry will be able to manufacture chips so complex that timing analysis issues will become completely intractable, requiring a shift from traditional synchronous to asynchronous architecture [1].

Asynchronous design can be an alternative for:

- its modularity and as a consequence simplified IP core interfacing;
- delay-insensitive (DI) designs are functionally correct regardless of the gate/wire delays making designs robust to environment and manufacturing variations;
- ‘asynchronous’ means ‘no clock’ therefore it eliminates clock-related problems.

Despite clear advantages of asynchronous designs their acceptance in industry is extremely low.

Asynchronous disadvantage. The major technical barrier is the lack of a design methodology that can be relatively easily understood and adopted by teams experienced in designing synchronous logic using RTL ASIC synthesis flow. Although academia has developed research tools for asynchronous design [2, 3] and demonstrated the

feasibility of asynchronous ICs [4, 5], electronics industry has been reluctant to adopt them for two main reasons:

- using those tools is resource consuming (design methodology is different from RTL, specification format is usually neither HDL nor any other of those used in industry, usage of asynchronous methods requires re-training of engineers, etc);

- no guarantee for success – some tools cannot handle industrial size applications, others exhibit poor results.

This creates a chicken-egg problem: asynchronous EDA is not mature enough for industrial applications resulting in low or no demand for it in industry, this, in turn, results in the absence of commercial quality asynchronous EDA tools.

Thus, a methodology scalable enough to allow synthesis of robust asynchronous implementation for large industrial designs from HDL specification supported by a commercial quality EDA flow could be a solution.

State of the art. The first attempt to provide such a methodology and an EDA flow was undertaken by Theseus Logic and presented in [6-8]. Null Convention Logic (NCL) flow by Theseus includes the synthesis and simulation based on existing, off-the-shelf EDA tools allowing HDL design specification, high complexity and familiar interface. Unfortunately, the NCL circuits synthesized by the flow from [6] still lose the competition with both: synchronous ASIC and asynchronous full-custom template based dynamic ‘integrated pipelining’ from Fulcrum Microsystems [9]. The latter is based on highly pipelined fast QDI domino-like logic circuitry without latches. However the lack of high-level automatic synthesis slows down design and requires special training of involved engineers.

Solution. We propose Gate Transfer Level (GTL) – a general methodology to replace RTL for asynchronous implementations. In support of the methodology we are developing Weaver – a GTL EDA tool. The GTL flow starts from regular HDL. It is based on commercial EDA tools and can handle circuits of the same complexity as contemporary synchronous RTL synthesis flows. Considering the place of our approach in the area/performance trade-off we need to mention that the higher the pipeline granularity the lower the area overhead for completion and handshake circuitry. However going to lower levels of granularity provides a tremendous gain in circuit performance. It is important to note, that in automated synchronous designs pipelining is difficult to implement because it changes the

number and position of registers, which results in a completely new specification. There are no tools capable of establishing functional equivalence between pipelined and non-pipelined synchronous implementations. Besides, synchronous design pipelining is reasonable only for more than eight levels of logic. Further reducing the amount of logic per pipeline stage reduces the amount of useful work per cycle while not affecting the overheads associated with latches, clock skew and jitter [10, 11]. For asynchronous circuits these issues do not exist because ‘local clocks’ are internal for a design and its external behavior remains the same independently of the granularity. This is a unique feature of clockless systems. We developed GTL flow as a way to automatically generate high performance extremely fine-grain (gate level) pipeline circuits using Fulcrum style (logic without latches) templates from standard HDL specifications. In GTL flow we are keeping the main idea of dual-rail expansion from [8] with major modification required by shift from coarse-grain to fine-grain pipelining. Carefully designed stages that satisfy in-cell timing assumptions can be placed in library as cells. Such cells communicate in a delay-insensitive manner. This makes the timing analysis only necessary for performance optimization while the functionality is ensured by the DI property. The paper presents the GTL concept, design flow, HDL coding style and some experimental results.

2. Implementation Basics

This work focuses on QDI implementations constituting the largest practical class of designs that effectively tolerate delay variations and are appealing for deep submicron technology. QDI implementation assumes a two-phase discipline in which data communication alternates between *set* and *reset* phases [12]. Data changes from the spacer (*NULL*) to a proper codeword (*DATA*) in the set phase, and then back to *NULL* in the reset phase. The simple delay-insensitive scheme may be obtained by coding *DATA* codewords by one-hot codes, and the spacer *NULL* - by a vector with all entries equal to ‘0’. Particular examples of delay-insensitive encoding based on one-hot codes are: 1) **dual-rail encoding**, in which each signal *a* is represented by two wires *a.0* and *a.1* (i.e. *a*=1 encoded as *a.0*=0, *a.1*=1, and *a*=0 encoded as *a.0*=1, *a.1*=0), or 2) ***n*-rail encoding**, in which a *n*-value signal *a* is encoded by *n* wires *a.0*, ..., *a.n*. An attractive property of delay-insensitive encoding is the capability for a receiver to determine that a codeword has arrived by the codeword itself, without appealing to timing assumptions. For

example, in the DI bus of Figure 1, as soon as one of the wires in each dual-rail pair (*a.0* or *a.1* and *b.0* or *b.1*) goes high, a valid dual-rail codeword is received. Detection of a code completion for every dual-rail pair is implemented by OR gate while the completion of the whole bus is implemented by a latch with the function $g=x_1x_2 + g(x_1+x_2)$, known as a Muller’s C-element [13].

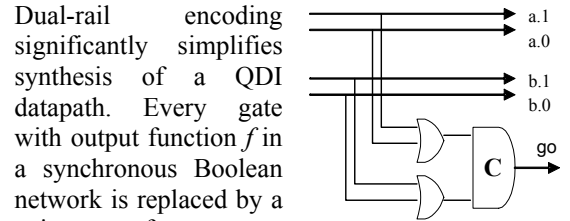


Figure 1 Completion detection for DI bus

Dual-rail encoding significantly simplifies synthesis of a QDI datapath. Every gate with output function f in a synchronous Boolean network is replaced by a pair of gates implementing direct and inverse functions $f.1=f$ and $f.0=f'$ in dual-rail implementation. Handshake control may be implemented uniformly independently from its granularity. This suggests a synthesis approach based on a set of pre-designed templates [5, 14-16], where the inter-stage handshake circuit is considered as a template parameterized by the stage function.

Static standard gate level and dynamic transistor level implementations of a GTL gate representing an AND2 gate are shown within a general QDI template example for a single dual-rail gate on the Figure 2. *LReq* and *LAck* stand for left and *RReq* and *RAck* for right request (*req*) and acknowledgement (*ack*) signals, ACK for handshake circuitry, PC for phase control and CD for completion detection. Storage is shown implemented with Muller C-gates [13] in static implementation. Parasitic capacitances can retain the value for long enough in dynamic implementations. By adding weak inverters ‘staticizers’ can be formed to keep the value for an unlimited time. Such implementation yields no timing assumptions.

Staticizers store the stage output value, solve the problem of charge sharing and improve the circuit noise margin of a pre-charge style implementation of F. The *req* line is used to signal data availability to the following stages while the *ack* indicates that the data portion has been consumed. Depending on the communication protocol, some or all of the handshake events can be transmitted over the data lines so *req* and/or *ack* lines may not even be needed.

The function implementation (F) is the only block specifying the gate logical function.

Out of several asynchronous pipeline styles developed [15-19] in this work we’ve chosen the

simplest data driven style presented in [20] for its minimal and local (inside the stage) delay assumptions and robust (delay insensitive) inter-stage communication.

One way of designing such a pipeline is straightforward – utilizing a standard cell library extended by a Muller C-element (Figure 2 left). Two major concerns with this approach are the area overhead and satisfying the in-stage isochronic fork assumptions. We used such an implementation for testing the flow and the dynamic characteristics of the resulting implementations.

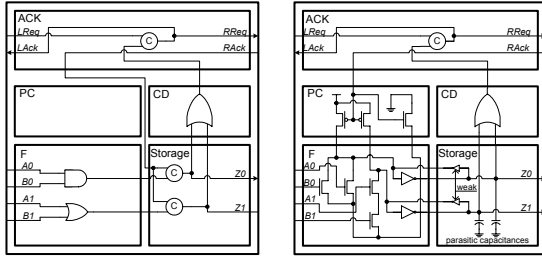


Figure 2 Naive (left) and dynamic implementations of a GTL gate

A dedicated library with each cell representing a pipeline stage (Figure 2 right) can solve both mentioned problems. However the dynamic library is under development at this time.

The use of dynamic logic is attractive for synchronous designs but no dynamic gate standard cell libraries exist so far mostly due to the late input arrival, charge sharing and noise problems. These problems are eliminated in GTL designs thanks to monotonic data transitions, completion detection and data-dependent control.

Memory and logic function implementation are as low-cost and as fast as synchronous domino-like circuits. The overhead comprises the dynamic C-element for handshake implementation (ACK), CD and *ack/req* synchronization.

In addition to low overhead, carefully developed GTL dynamic cells implementing entire stage functionality address the isochronic fork condition: assumptions become local (inside the cells).

3. Design flow

We have implemented a flow maximizing the use of commercial design tools. The flow executes two synthesis steps. On the first step a single-rail synchronous implementation is synthesized for a high-level behavioral specification. It is optimized and mapped into the library composed of conventional single-rail gates functionally equivalent to the target GTL library. Like in [7, 8] a commercial RTL synthesis tool (e.g. Synopsys

DC Ultra) is used. On the second step, similarly to [7], this netlist is expanded into a dual-rail implementation which is mapped into the asynchronous pipeline template library. It consists of the following sub-steps:

1. Reduction of a network to unate gates through considering direct and inverse values of variable a as two different variables $a.0$ and $a.1$. The obtained unate network implements *rail.1* of a dual-rail combinational circuit.

2. Dual-rail expansion of the combinational logic by creating for each gate in the *rail.1* network a corresponding dual gate in the *rail.0* network.

3. Ensuring delay-insensitivity by providing local completion detectors (as described in section 2) at each stage boundary, connecting them to the ACK block (Figure 2) and synchronizing *req* and *ack* lines (inserting forks and joins) where applicable (Figure 3). Figure 3 inherits the notation of Figure 2.

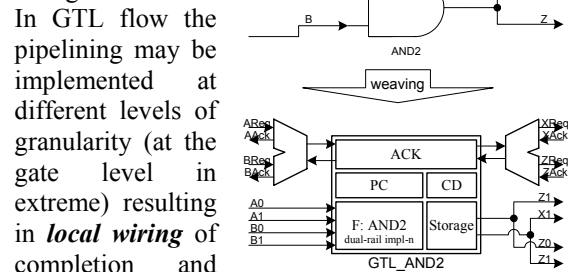


Figure 3 Weaving

signals between adjacent gates (using *ack* and *req* signals). **Neither special completion networks nor registers with completion trees are needed to be built.**

3.1. Standard HDL specification

Complete synthesizable HDL subset is supported by the GTL flow. Unlike in HDL-based asynchronous attempts to express asynchronous formal models in HDL (Martin's CHP in case of [21] and [22] or Signal transition Graphs in case of [23]).

Contrary to [6-8] GTL flow generates asynchronous mechanisms (including handshake circuitry) automatically hiding it from the user.

3.2. Flow architecture

The Weaver flow architecture (Figure 4) is an extension and generalization of that by Theseus [6-8] exploiting the idea of on-line compilation of a synchronous netlist synthesized by an industrial synthesis engine (currently Synopsys DC Ultra) into a QDI asynchronous circuit and relying on

industrial tools for the final technology mapping, P&R, etc of the final GTL netlist. As a result, a QDI implementation is compiled for a specification in standard behavioral HDL. The notation of Figure 4 is self-explanatory except for

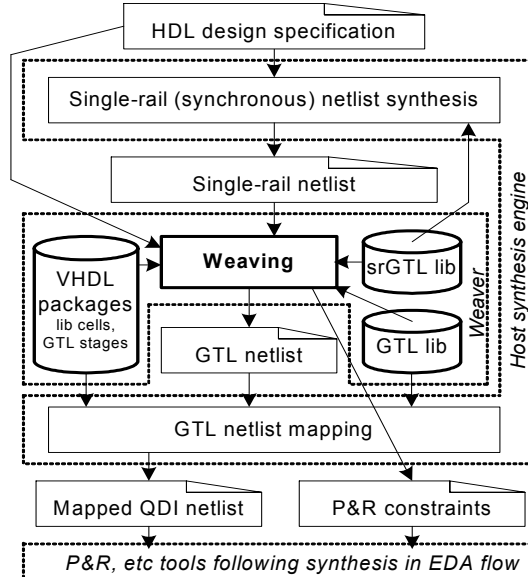


Figure 4 Weaver EDA flow architecture

the srGTL lib which stands for a *rail.1* (see section 3) of the target GTL library.

The flow consists of the Weaver Engine (WE), a set of Tcl scripts to automate the engine interaction with the host synthesis flow environment and a set of VHDL packages in conjunction with physical library specifying the target pipeline architecture.

Tcl scripts introduce new commands to the host compiler command set to automate library retargeting, calling WE etc. For instance the *wvr_acs_compile_design* command implements the same functionality as the *acs_compile_design* from Synopsys Automated Chip Synthesis but synthesizes GTL implementation.

VHDL packages specify particular GTL stage architecture as they are composed out of the cells available in the physical target library. These packages are also used for VHDL simulation of the design. The use of packages as opposed to hard coding the architecture in the WE facilitates synthesis retargeting from one template to another.

Weaver engine – the heart of the flow is a VHDL compiler and a synthesis engine on its own. It is based on the Savant VHDL compiler [24] using AIRE (Advanced Intermediate Representation with Extensibility) – standard internal VHDL representation. Currently WE is relying on a commercial HDL compiler and synthesis engine to perform synthesis and mapping. WE mostly

operates on the synthesized netlist to perform dual-rail expansion, reset routing, pipeline balancing and handshake circuitry synthesis.

The main WE function is dual-rail expansion and synthesis of inter-stage handshake communication circuitry called ‘weaving’ illustrated for an AND2 gate on the Figure 3. The GTL netlist is mapped to the library using the host synthesis engine. Place & Route constraints are not generally needed if a dedicated library is used (timing assumptions are satisfied inside the cells and the inter-stage communication is delay-insensitive) but can be helpful for better performance.

Like RTL, GTL does not define any particular kind of circuitry but rather a wide-known idea of gates communicating through handshakes. This is reflected in the Weaver flow which is built with the assumption that gates are converted to stages communicating with *req/ack* signals requiring synchronization.

3.3. GTL impact

A different circuit methodology makes the Weaver flow significantly different from both synchronous RTL and NCL flows. The biggest impact results from the fine-grain nature of GTL circuit.

1. Coarse-grain pipeline is replaced by the variable granularity pipeline – by default gate level fine-grain with independent bit-level data waves’ propagation and completion. Combinational logic is still synthesized in portions, but their size is no longer dictated by the clock frequency or the timing closure achieved by balancing the size of combinational logic portions as in synchronous RTL designs. It is defined solely by the structure and hierarchy of the design.

2. In RTL, data path merging/branching is relatively rare so NCL preserving the registers structure could afford relying on the designer to explicitly specify the synchronization of *done* signals [6] to ensure proper inter-register communication.

Finer granularity implies smaller but many more synchronization points. Therefore GTL handshake circuitry is on one hand more complex but on the other more systematic what allows for complete synthesis automation leaving the designer to concentrate on the application.

3. Timing analysis is no longer an issue with GTL from the functionality point of view but if performance is an issue pipeline balancing – balancing the token capacity of concurrent pipeline channels by inserting additional buffer stages in shorter portions of the data path – becomes very important. Currently WE implements a simple topological sort based algorithm.

4. Weaving: case study

The sections above explain the general approach for converting a combinational single-rail netlist into a GTL implementation. However not all circuits are combinational. The most important special case is mapping latches and flip-flops.

4.1. Mapping latches and flip-flops

Deep combinational logic (CL) is often pipelined in synchronous design to increase the circuit performance. Figure 5 (top) illustrates that. We assume here either flip-flops or pairs of latches e.g. D-latches (DL) with alternate clocks to be used for pipelining. Empty and shaded circles in the latches reflect alternative clock phases of ‘master’ and ‘slave’ latches. Such a pair of circles represents one data token. By pipelining (breaking the CL into two stages) the clock cycle is reduced by the factor of two. Suppose a half-buffer (HB) [14] pipeline stages

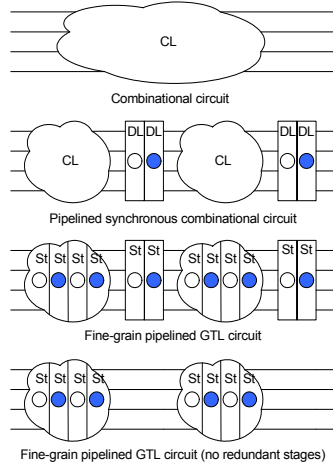


Figure 5 Latches and flip-flops mapping: deep CL

are used. Then GTL turns every CL gate level into another pipeline layer. (See the next level of Figure 5 where St stands for pipeline stage and the circles represent *NULL* and *DATA* parts of data tokens). More data tokens can simultaneously coexist in the pipeline hence the performance is increased. Note that every new layer implements some function whereas the layers corresponding to latches implement no functionality. Therefore if the combinational portions are not too shallow (as shown below) the stages inferred from the latches become redundant and can be removed (fed through). Note that the same result is achieved if the non-pipelined logic is compiled. Thus, existing RTL code can be reused. However the same result is obtained from the initial specification saving the pipelining effort.

Another example (Figure 6) presents a shift register or counter style example where the combinational logic between flip-flops is shallow (one level to none) in the synchronous implementation. The stages corresponding to the flip-flops in synchronous implementation are redundant unless the intermediate lines branch

(e.g. a counter with a parallel output). Since the fork modules are only *ack* synchronizers and represent no data buffering (not stages) the second implementation achieved by simply removing the flip-flops has the functionality distinct from the specification and the synchronous implementation: the last three bits are the same bit branched to 3 while the synchronous implementation is a shift register.

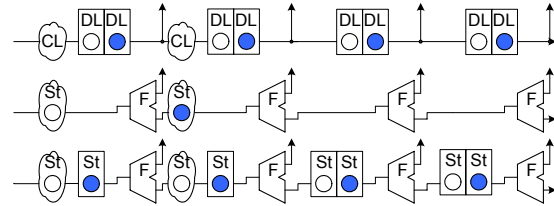


Figure 6 Latches and flip-flops mapping: shallow CL

Clearly the bottom implementation (Figure 6) is correct as long as every other stage is initialized with data tokens (HB pipeline is full).

It is easy to see and it can be shown formally that since in synchronous implementation a pair of latches or a flip-flop can hold one data token the *GTL implementation must provide not smaller token capacity per pipeline stage* to preserve the original functionality. This condition is clearly met for the bottom implementation on the Figure 6. For the stages with no combinational logic every flip-flop has been mapped to two HB (one full-buffer) stages. However in the first bits the combinational gate mapped to the corresponding GTL gate already provides one stage so only one additional HB stage is needed on the place of the flip-flop. Register elimination (Figure 5) and buffer insertion (Figure 6) are performed automatically during weaving in our GTL flow.

The *clock* signal is optimized out in the final design. It does not, result in any loss of functionality. Time separation of data tokens is replaced by controlled separation – instead of supplying a clock signal and squeezing the data portions between its transitions indicating the presence of data with an enable signal every data portion is signaled as soon as it is ready. Weaver uses *clock* recipients to automatically determine the data initialized stages (1 for set and 0 for reset).

4.2. Coding for GTL

Coding for GTL currently is not any different from conventional “synchronous” HDL coding style. The above described procedure of mapping registers makes it clear that the effort should not be spent on pipelining which is done automatically

or particular clock related issues which are ignored.

Some additional customization (at this point only for the format of module interface – *req/ack* per signal or synchronized for the whole bus etc) is available through special form comments (like in Synopsys DC).

Host compiler special form comments affecting the intermediate single-rail design architecture (FSM encoding style etc.) can still be used.

Requirement to accept standard “synchronous”

HDL coding style enables reuse of existing RTL HDL as well as makes design specification easier for ‘RTL engineers’. It also makes possible to use standard synchronous synthesis engine(s) for HDL synthesis in conjunction with the supplement technique (weaving) described above.

Figure 7 shows a sample HDL specification of a sequence detector and the Figure 8 sketches its corresponding implementations, where HB and FB stand for half and full buffers while F – for fork.

```

entity sequence_detector is
port (D_out : out std_logic;
      D_in, clk : in std_logic);
end sequence_detector
architecture behavior of sequence_detector is
signal data : std_logic_vector(2 downto 0);
begin
process(clk)
begin
if (clk='0' and clk'event) then
data <= (D_in, Data(2), Data(1));
end if;
if (data = "111") then
D_out <= '1';
else
D_out <= '0';
end if;
end process;
end behavior;

```

Figure 7, Sequence detector behavioral VHDL specification

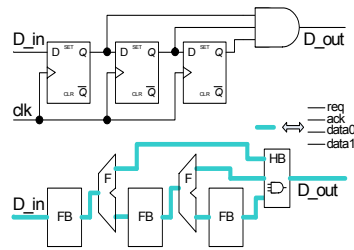


Figure 8 Sequence detector synchronous (top) and GTL implementations

where HB and FB stand for half and full buffers while F – for fork.

5. Preliminary experimental results

To estimate the efficiency of the flow we compare the performance of several synchronous and GTL implementations of benchmark circuits from the set [25]. The examples in the Table 1 are combinational multilevel circuits. These are used to optimize pipeline balancing technique. Similarly to slack matching [26] it balances the number of tokens in the propagation paths increasing the performance up to the performance of the slowest stage. With gate-level pipelining the number of

pipeline stages equals the logic depth after synthesis. Synchronous implementations are not pipelined so the gain is proportionate to the logic depth – the deeper the logic – the greater the pipelining effect. The results are still preliminary since the pipeline balancing implementation is not fully complete.

Table 1. Performance comparison on MCNC benchmarks

Benchmark	gates #	Performance, MHz		
		rtl	gtl	g/r
C17	6	2857	667.6	0.23
C1355	546	37	120.6	3.26
C1908	880	21.7	105.2	4.84
C432	160	189.9	351.0	1.85
C499	202	43.0	122.0	2.84
C5315	2307	44.8	181.3	4.04
C880	383	66.5	246.4	3.71
apex6	238	65.8	158.9	2.80
cm162a	19	63.0	388.0	6.16
cm163a	16	63.2	491.2	7.78
cordic	102	47.8	289.1	6.04
dalu	1131	38.2	268.0	7.02
frg2	526	44.7	223.3	5.0
lal	71	40.2	215.6	5.36
sct	40	55.0	242.6	4.41
X4	136	60.9	283.0	4.65

Avg 4.37

We also compared the synthesis results for an Advanced Encryption Standard (AES) [27] implementation. Unlike the logic synthesis benchmarks the AES example requires hierarchical pipeline balancing. AES was chosen for a design example because: (1) it is a rather large and complex hierarchical design involving various synthesis aspects including state machine and non-linear pipelines in data path design; (2) fine-grain pipeline asynchronous implementation of a security application is potentially advantageous for being less prone to side-channel attacks [28] because it has a balanced power dissipation independent from the data patterns at bit lines.

Table 2 Performance comparison

Example	Performance, MHz			Area, um ² xE+06		
	gtl	rtl	g/r	gtl	rtl	g/r
Inverter	350	9.58	36.5	0.17	0.02	12.0
mix_128	666	176	3.78	0.50	0.06	8.23
sbox_128	350	47.9	7.3	0.31	0.27	11.4
keyexpansn	353	44.5	7.93	1.00	0.08	12.0
normal_rnd	350	44.8	7.81	3.78	0.35	10.9
last_round	352	47.4	7.42	3.25	0.29	11.4
aes10rnds	349	9.58	36.4	47.9	4.28	11.2

With no dedicated library, a straightforward stage implementation using a standard cell library [29] (TSMC 0.25 process) extended with Muller C-

elements was used in experiments. Such an implementation generates significant area overhead which will be much smaller with future dynamic logic library. Performance parameters were obtained from simulating a mapped netlist with timing parameters generated for the library (VITAL VHDL specifications).

The synchronous implementation numbers were obtained with the same library ($1/\text{delay} \cdot 10^6$). The synchronous circuit should have been pipelined to achieve better results (at least by placing registers between rounds) but this requires manual design with pipeline stage balancing where as asynchronous design is pipelined automatically.

Even in the stage of preliminary (without dynamic logic library) design where our area results are very far away from our final estimation we could observe the trade-off curve Figure 9 that may be explored using different timing assumptions and different granularity of pipelining. In our AES10

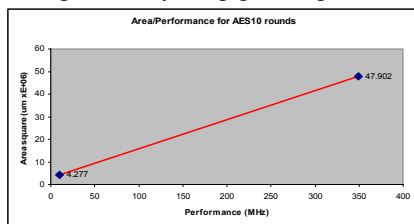


Figure 9 Performance/area trade-off

example synchronous design is not pipelined and use non-local timing assumptions, asynchronous fine-grain pipelined implementation do not rely on timing assumptions and reach a finest degree of pipelining with high area expenses. We did not compare it with NCL implementation because NCL tool required significant code change and is not able to accept hierarchical designs. But we know from previous experience and [6-8] that performance of circuits generated by NCL flow is not better than that of the synchronous counterpart and the area is 2.5-3.5 times bigger.

6. Conclusion and future work

We present GTL – a complete synthesis framework: methodology and the Weaver EDA synthesis flow for fully automatic gate-level pipelining.

Weaver flow provides HDL interface and synthesis scalability facilitating its integration in industrial environments as well as performance and modularity of GTL circuitry contributing to design reuse and robustness.

Our automatically synthesized asynchronous AES implementation (*aes10rounds*) with static standard cell library demonstrated a performance increase

up to 36.4X compared to automated synchronous RTL implementation of the same VHDL specification, and reached performance of the fastest to our knowledge AES IP core from North Pole Engineering performing at 350MHz specifically designed for performance.

Our future research will concentrate on a reliable rich dynamic standard cell library. The use of such a library in the GTL flow will combine the high performance of the fine-grain pipelines with competitive area overhead of dynamic library with design automation provided by the Weaver engine. In another paper [30] we delivered a formal proof of functional and behavior equivalence of our asynchronous pipelined circuits to their synchronous counterparts. This work is based on Marked Graph theory [31], notions of flow equivalence from [32] and synchronous behavior from [33]. However, we are using a very different circuit structure and implementation philosophy from [32] and [33].

References

1. *International technology roadmap for semiconductors*. (<http://public.itrs.net/Files/2003ITRS/Home2003.htm>). 2003.
2. Nowick, S., *MINIMALIST CAD Package*. <http://www.cs.columbia.edu/~rmf/MINIMALIST-get.html>.
3. Cortadella, J., *Petrify home page*. <http://www.ac.upc.es/~vlsi/petrify/petrify.html>.
4. Furber, S.B., D.A. Edwards, and J.D. Garside, *AMULET3: a 100 MIPS Asynchronous Embedded Processor*, in *Proc. International Conf. Computer Design (ICCD)*. 2000.
5. Martin, A.J., et al., *The Design of an Asynchronous MIPS R3000 Microprocessor*, in *Advanced Research in VLSI*. 1997. p. 164--181.
6. Smith, R. and M. Lighthart, *High-Level Design for Asynchronous Logic*, in *Proc. of Asia and South Pacific Design Automation Conference*. 2001. p. 431--436.
7. Kondratyev, A. and K. Lwin, *Design of Asynchronous Circuits by Synchronous CAD Tools*, in *Proc. ACM/IEEE Design Automation Conference*. 2002. p. 411-414.
8. Lighthart, M., et al., *Asynchronous Design Using Commercial HDL Synthesis Tools*, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2000, IEEE Computer Society Press. p. 114--125.
9. Lines, A. *An Asynchronous SoC Interconnect*. in *HOT Chips. A Symposium of High Performance Chips*. 2002.

10. Hrishikesh, M.S., et al. *The Optimal Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays*. in *29th Int'l Symp. Computer Architecture*. 2002: IEEE CS Press.
11. Hartstein, A. and T.R. Puzak. *Optimum Power/Performance Pipeline Depth*. in *MICRO-36 International Symposium on Microarchitecture*. 2003.
12. Varshavsky, V.I., et al., *Self-timed Control of Concurrent Processes*. 1990: Kluwer Academic Publishers.
13. Muller, D.E. and W.S. Bartky, *A Theory of Asynchronous Circuits*, in *Proceedings of an International Symposium on the Theory of Switching*. 1959, Harvard University Press. p. 204--243.
14. Ozdag, R.O. and P.A. Beerel, *High-Speed QDI Asynchronous Pipelines*, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2002. p. 13--22.
15. Singh, M. and S.M. Nowick, *Fine-grain pipelined asynchronous adders for high-speed DSP applications*, in *Proceedings of the IEEE Computer Society Workshop on VLSI*. 2000, IEEE Computer Society Press. p. 111--118.
16. Singh, M. and S.M. Nowick, *High-Throughput Asynchronous Pipelines for Fine-Grain Dynamic Datapaths*, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2000, IEEE Computer Society Press. p. 198--209.
17. Ferretti, M. and P.A. Beerel, *Single-Track Asynchronous Pipeline Templates Using 1-of-N Encoding*, in *Proc. Design, Automation and Test in Europe (DATE)*. 2002. p. 1008--1015.
18. Sutherland, I. and S. Fairbanks, *GasP: A Minimal FIFO Control*, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2001, IEEE Computer Society Press. p. 46--53.
19. Williams, T.E. and M.A. Horowitz, *A Zero-Overhead Self-Timed 160ns 54b CMOS Divider*. *IEEE Journal of Solid-State Circuits*, 1991. **26**(11): p. 1651--1661.
20. Cummings, U., A. Lines, and A. Martin, *An Asynchronous Pipelined Lattice Structure Filter*, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 1994. p. 126--133.
21. Renaudin, M., P. Vivet, and F. Robin, *A Design Framework for Asynchronous/Synchronous Circuits Based on CHP to HDL Translation*, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 1999. p. 135--144.
22. Saifhashemi, A. and H. Pedram. *Verilog HDL, powered by PLI: a suitable framework for describing and modeling asynchronous circuits at all levels of abstraction*. in *Design Automation Conference*. 2003.
23. Blunno, I. and L. Lavagno, *Automated synthesis of micro-pipelines from behavioral Verilog HDL*, in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2000, IEEE Computer Society Press. p. 84--92.
24. Martin, D.E., et al., *Analysis and Simulation of Mixed-Technology VLSI Systems*. *Journal of Parallel and Distributed Computing*, 2002. **62**(3): p. 468-493.
25. Yang, S., *Logic Synthesis and Optimization Benchmarks Version 3.0*. 1991, Microelectronics center of North Carolina.
26. Kim, S. and P.A. Beerel, *Pipeline Optimization for Asynchronous Circuits: Complexity Analysis and an Efficient Optimal Algorithm*, in *Proc. International Conf. Computer-Aided Design (ICCAD)*. 2000.
27. *FIPS PUB 197: Advanced Encryption Standard*, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
28. Hess, E., et al. *Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures - A Survey*. in *EUROSMART Security Conference*. 2000.
29. Sulistyo, J.B. and D.S. Ha, *Developing Standard Cells for TSMC 0.25um Technology under MOSIS DEEP Rules*. 2002, Department of Electrical and Computer Engineering, Virginia Tech.
30. Smirnov, A., et al. *Gate Transfer Level Synthesis as an Automated Approach to Fine-Grain Pipelining*. in *Workshop on Token Based Computing (ToBaCo)*. 2004. Bologna, Italy.
31. Commoner, F., et al., *Marked directed graphs*. *Journal of Computer and System Sciences*, 1971. **5**: p. 511-523.
32. Blunno, I., et al. *Handshake protocols for desynchronization*. in *International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2004.
33. Linder, D.H. and J.C. Harden, *Phased Logic: Supporting the Synchronous Design Paradigm with Delay-Insensitive Circuitry*. *IEEE Transactions on Computers*, 1996. **45**(9): p. 1031--1044.