An Automated Fine-Grain Pipelining Using Domino Style Asynchronous Library

Alexander Smirnov, Alexander Taubin, Ming Su, Mark Karpovsky
Department of Electrical and Computer Engineering
{alexbs, taubin, mingsu, markkar}@bu.edu

Abstract.

Register Transfer Level (RTL) synthesis model which simplified the design of clocked circuits allowed design automation boost and VLSI progress for more than a decade. Shrinking technology and progressive increase in clock frequency are bringing clock to its physical limits. Asynchronous circuits, which are believed to replace globally clocked designs in the future, remain out of the competition due to the design complexity of some automated approaches and poor results of other techniques. Successful asynchronous designs are known but they are primarily custom. This work sketches an automated approach for automatically re-implementing conventional RTL designs as fine-grain pipelined asynchronous quasi-delay-insensitive (QDI) circuits and presents a framework for automated synthesis of such implementations from high-level behavior specifications. Experimental results are presented using our new dynamic asynchronous library.

Keywords: asynchronous EDA, synthesis, QDI, ASIC, HDL.

1. Introduction

The popularity of synchronous design and its support by EDA tools on one hand and the crisis of the synchronous paradigm due to process variation, signal integrity problems and other physical limitations on the other hand have resulted in a number of approaches to asynchronous reimplementation of synchronous designs. The main idea of such reimplementation is substituting the global clocking by local control communications. It has been explored in a number of works including [1-5]. The following three approaches to asynchronous circuits design automation are the most elaborate and closest to our approach.

Phased logic [2, 6] replaces every combinational logic (CL) gate with its dual-rail implementation. One rail carries data while the other – phase. Distinct phases correspond to distinct data tokens: the codes ‘00’ and ‘11’ can be followed by ‘01’ or ‘10’ and vice versa but ‘00’ is never followed by ‘11’ and ‘01’ by ‘10’ etc. Such an encoding scheme is beneficial for power consumption since only one transition is required per data token propagation through a stage but to ensure safeness additional feedbacks are required [2] on the design stage increasing the design complexity.

De-synchronization [3, 5] is another approach to redesigning clocking for synchronous netlists. The authors explored both delay matching and completion detection techniques. The de-synchronized circuits are architecturally equivalent to the original RTL design.

Null Convention Logic (NCL) [1, 4] EDA flow from Theseus Logic exploits the idea of synthesizing large designs using a commercial synchronous synthesis engine and substituting globally clocked synchronous registers in the data path by asynchronous registers communicating asynchronously through delay insensitive handshakes. NCL circuits are dual-rail to enable completion detection. They are architecturally equivalent to the RTL implementation. Heavy synchronization of completion detection signals at registration points slows NCL designs. The designer is required to manually specify some handshake signals in VHDL. That complicates the existing RTL reuse.

None of the above approaches offer support for automated pipelining therefore they do not improve the performance of the original design.

In synchronous designs automatic pipelining is difficult to implement because it changes the number and position of registers which finally results in a completely new specification. Synchronous design pipelining is reasonable only for more than eight levels of logic and this number is increasing because

the technology shrinking is decreasing the gate latency while clock limitations (routing and skew) stay the same. 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 [7, 8]. In asynchronous circuits the handshake implementation, latency and area are shrinking with the rest of the circuit. Since no assumptions are made about the inter-cell communication delays the circuit correctness no longer depends on the amount of process variation.

These unique features of clockless handshake-based systems have inspired the present work dedicated to automated synthesis of pipelined asynchronous implementations from HDL specification.

Our EDA flow implements the behavior specified with regular HDL as a pipelined QDI asynchronous circuit by synthesizing a synchronous implementation of the specified behavior and '*weaving*' it into an asynchronous implementation. By default the design is pipelined on the finest grain gate level. This way the highest performance is achieved.

The main distinctive feature of our approach is that it does not only solve the global clocking problem by substituting global synchronization signal with local self-timed control but also replaces the register transfer architecture by gate transfer architecture (further referred to as GTL). This automatically results in very fine grain pipelined circuits regardless of the original synchronous implementation pipelining.

Apart from the efforts to automate pipelined circuits synthesis serious efforts are dedicated to designing efficient micropipeline implementation. The latter mostly fall into two categories: matched delay [9-13] and QDI [13-17] depending on the computation completion detection implementation. Fulcrum Microsystems [18] is one of the companies known to successfully implement commercial high-speed asynchronous chips using micropipelines. However to our knowledge the design remains custom with some in-house proprietary EDA tools.

In [19] a QDI micropipeline based standard-cell library is presented.

As it is explained in section 3 some of the above pipeline styles can be used within the proposed framework.

2. Data tokens and pipeline models

In synchronous designs distinct data portions (we shall denote them as *data tokens* or just *tokens*) can propagate through several processing stages separated with registers consisting of either flip-flops (FFs) shown in Figure 1a or pairs of alternatively clocked D-latches (DL) – in Figure 1b – pipelining. It allows subsequent data tokens to be processed concurrently allowing for performance increase at the price of some area and latency overhead. In synchronous implementation a token occupies one FF or two DLs (one stores the token data and one is transparent). A Petri Net (PN) model of a synchronous pipelined data path is shown in Figure 1c. A two-transition loop represents clock, other transitions – data latch control signals or simply data latching/propagating through the corresponding latch. Clearly the entire system is safe – no more than one token can be present in any place at the same time. In the hardware systems this means that the data tokens remain distinct – data intact.

With no clock handshake signals based on the computation completion can be used to clock the latches keeping the data distinct. One way of sensing computation completion is by inserting a delay element that would match the worst case delay of data propagation through the data path – bundled data or delay matching. Another approach is completion detection where data is encoded in such a way that either (1) data tokens in the data path are separated with so called null tokens (or spacers) or (2) consecutive data tokens are always distinct regardless of the data content [3]. First requires two transitions: spacer → data → spacer for token propagation through a stage, while the second – only one however the stage implementation is more complex in the second case.

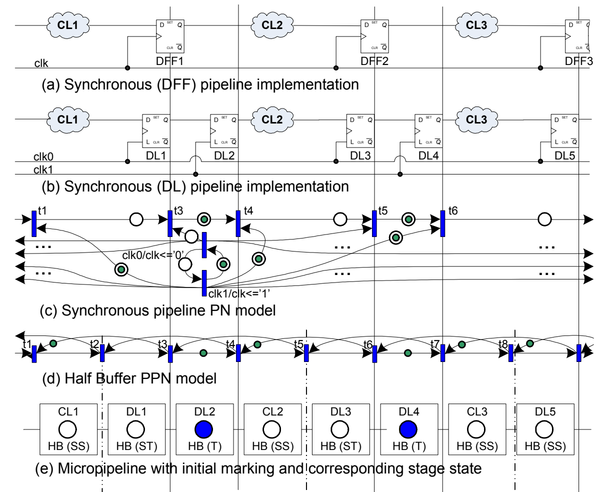

Figure 1 Pipelines and their models

In this paper we only consider the completion detection based (or QDI) return to zero (RTZ) protocols (tokens are separated with spacers). The capacity of a pipeline implementing such a protocol

is at most half the number of memory elements that can store tokens (half of them storing tokens and another half – spacers). Every pipeline stage capable of storing a token is called a half-buffer (HB) stage. Some implementations feature two memory elements per stage to keep both a token and a spacer at the same time – those are called full-buffer (FB) stages. In synchronous design latches can be considered as HB stages while FFs – as FB stages (an FF can be considered to be built out of two latches as it is shown in the Figure 1a,b). We consider the more general HB pipelines.

In Figure 1 the vertical lines separate HB stages. Notice that there are more lines in Figure 1d,e. This illustrates how the pipeline capacity changes during the move to a fine-grain pipeline.

Various templates have been developed for both HB and FB pipelines. HB pipeline Petri Net (HB PPN) – a high-level PN model for an asynchronous HB pipeline is shown in Figure 1d (places are omitted). In HB PPN each stage is represented by three places (arcs). The HB stage states are: token (T), spacer and can accept token (SS) and spacer and cannot accept token (ST). The states of the individual stages are marked in Figure 1e. Every stage goes through SS $\rightarrow$ T $\rightarrow$ ST $\rightarrow$ SS (always the same sequence). The stage source transition (on the left from the stage) moves the stage to the T state while the stage sink transition – to the ST state. The stage moves to the SS once as the sink transition(s) of the following stage(s) is fired. The HB PPN model presented in Figure 1 has been used to prove the correctness of weaving algorithms for asynchronous micropipelines composed of HB stages. The model is not used for the actual re-synthesis (weaving).

3. Design flow

In this section we present the Weaver EDA flow implementing a "push button" approach to micropipeline synthesis from high-level behavioral specification retargetable to different micropipeline styles through the library approach

3.1. Physical library requirements

Currently we are targeting libraries based on templates presented in [17] but many other templates can be used. The following are the requirements for the physical library to be used in the flow.

Inter-cell communication must be delay insensitive. This is the only assumption we make about the library. That assumption should hold as long as no techniques are developed which preserve isochronic forks during place and route. QDI stands for delay insensitive (no wire delay assumptions) but some forks isochronicity can be required. The DI inter-cell communication assumption is satisfied as long as all isochronic forks are placed inside library cells and guaranteed by the library cells design. Apart from guaranteeing correct functionality this assumption lowers the effect of parameters variation on the design parameters. As long as individual cells are designed to handle variation across the cell no system-wide constraints are necessary. It is also noteworthy that the variation across any given cell is generally smaller and therefore easier to handle than across a chip.

Communication protocol is uniform among the submodules of any module and in the protocol tokens are separated with spacers. (The flow cannot handle a library designed for non-RTZ protocols like phased logic since the synthesis procedure would be quite different).

Data encoding is dual-rail (DR) one-hot. Single-rail (SR) logical '1' corresponds to DR'01', SR'0' – to DR'10' while DR'11' is an invalid combination (this fact can be used for error detection and testing) and DR'00' is a ***spacer***. ***Channels*** denote sets of signals (two for dual-rail) carrying data and up to two handshake signals (***request*** propagating in the same direction as data and ***acknowledge*** – in the opposite) associated with that data. Handshake signals' ***synchronization cells*** (usually Muller C-elements) ***must be available in the library in at least two versions: resettable to both 0 and 1***.

Apart from the handshake synchronization cells the library ***must implement at least the equivalent of AND2_cell*** (or OR2 assumed to be dual to AND2) ***and an identity function stage*** (buffer stage). (Inverter is implemented as the cross-over of data wires). ***There must be at least three versions of the buffer stage implementation (resettable to data1, data0 and spacer) and at least one resettable to spacer version of cells implementing other logic functions***.

3.2. Flow architecture

We have implemented an EDA flow [20] maximizing the use of commercial design tools. It executes three steps.

Library preparation. From the physical (GTL) library a virtual library (srGTL) is created to be used on the first design compile step in such a way that it contains single rail conventional gates functionally equivalent to GTL cells. VHDL representations of GTL gates are generated to enable the GTL implementation simulation. The GTL library must be

specified in Synopsys Liberty format. We used extensibility of the latter to describe the GTL gates interface: mark pins as acknowledge, request, reset, etc and also to define channels the pins belong to, the equivalent single-rail function implemented by the GTL cell, initialization and other specific attributes.

First, conventional RTL implementation is synthesized for the high-level behavioral specification, optimized and mapped using srGTL library. As opposed to the attempts to express asynchronous formal models in HDL (Martin's CHP in case of [21] and others or Signal Transition Graphs in case of [22]) we are using DC Ultra on this step to ensure quality support for a variety of high-level specification formats including complete synthesizable HDL subset.

On the second step, *Weaver Engine* (WE) automatically expands the single-rail netlist into a dual-rail QDI fine-grain pipelined (GTL) implementation and generates all local wiring related to the expansion and handshake implementation.

Finally the GTL netlist is mapped using currently the same DC Ultra to the GTL library. The use of a commercial engine on this stage ensures support of standard formats of library specification facilitating library development and of output file formats for smooth interfacing with place and route and other tools following synthesis in the design flow.

Most of the asynchronous cells are complex sequential devices. As such they are visible as black boxes for the RTL synthesis engine. No optimization is allowed on this stage. The GTL architecture optimization algorithms are implemented in the Weaver Engine.

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

Tcl scripts implement new commands within 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.

Weaver Engine is a VHDL compiler and a synthesis engine on its own based on the Savant VHDL compiler.

VHDL packages specify architectures of particular GTL stages and some compound standard modules. These packages are also used for the design VHDL simulation. The use of packages as opposed to hard coding the architecture in the WE facilitates synthesis retargeting from one physical library to another.

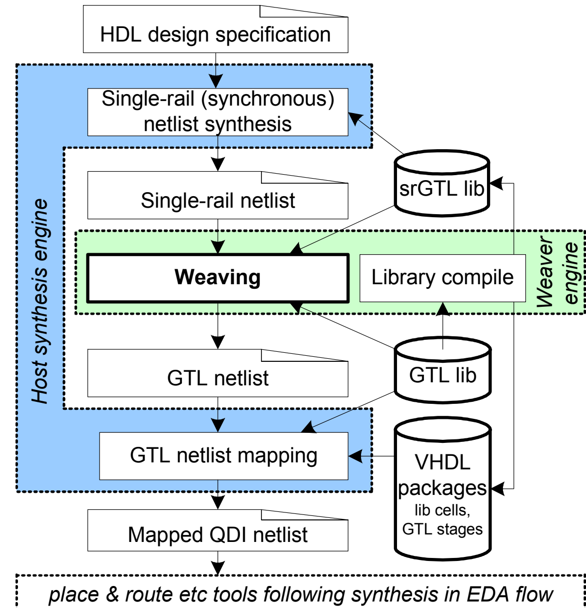

Figure 2 Design flow using Weaver

3.3. Weaving: general approach

Weaving is a procedure of re-synthesis of a synchronous single-rail implementation into a GTL implementation.

During the mapping RTL implementation data wires are substituted by *channels* and

(i) *no additional data dependencies are added and no existing data dependencies are removed;*

Channels depicted in Figure 3 reflect the general case (both *req* and *ack* are used and their synchronization for multiple-input gates and for multi-fan-out cases respectively are shown). The *req* and *ack* synchronizers are shown as they are introduced by the Weaver Engine. Some templates do not use all four (handshake *req*, *ack* and data *d0, d1*) communication lines (e.g. PCHB from [17] does not use *req*). Those still fit in the framework as long as they satisfy the requirements given in section 3.1.

Portions of combinational logic are substituted by functionally equivalent pipeline stages. Without loss of generality let every such portion be a single logic gate in synchronous implementation. In general, the portions can be of arbitrary size but currently every synchronous gate is replaced by a pipeline stage (very fine-grain pipelining) for a number of reasons:

- fine-grain gate level pipelining results in the highest performance implementations;
- smaller cells/macros have smaller routing overhead;
- smaller number of stage inputs results in smaller synchronization overhead;

- only cell/macro wide variation matters for QDI implementations delay insensitive outside the cell boundaries – so the smaller the cells the more optimized they can be;
- large cells are very inefficient in terms of place and route so if the stage is large it must be composed of basic cells; this brings up the issue of isochronic forks outside the cell boundaries; no solution for this problem is currently implemented in the flow.

Fine-grain pipelining has a large area overhead due to a very large number (one per each gate) of handshake control circuits. An alternative solution is necessary to allow for better area/performance trade-off.

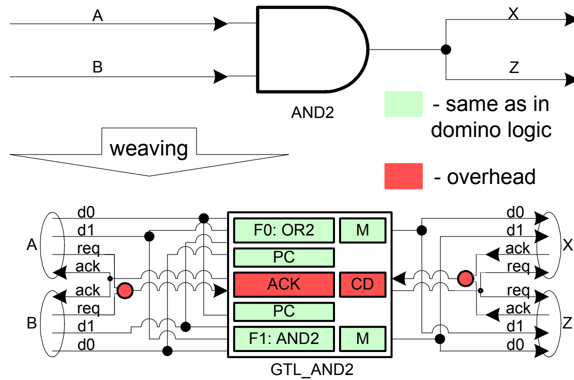

Figure 3 Weaving a combinational gate

Note that for instance inverters do not have to be implemented as stages using dual-rail encoding – their implementation is data wires cross-over (zero cost).

(ii) *every gate implementing a logical function is mapped to a GTL gate (stage) implementing equivalent function for dual-rail encoded data and initialized to spacer;*

Replacing a combinational gate implementing the AND2 function with a GTL gate (or stage) and the wires with channels is illustrated on the Figure 3.

3.4. Weaving: mapping latches and flip-flops

The section 3.3 addresses basic weaving – synthesizing a fine-grain pipelined GTL implementation for single-rail combinational netlists. However, RTL designs are not always combinational. Suppose that a synchronous RTL netlist produced by a commercial synthesis engine consists of combinational logic (CL) gates, D-latches (DL) and D-flip-flops (FF). Such an assumption is safe since we can limit the RTL target synthesis library to any set of gates as long as it satisfies the synthesis engine requirements. DFF and DL are sufficient for the DC-Ultra to implement any design.

Clocked registers (either composed of DFFs or DLs) are used in RTL implementations to separate concurrent processing of distinct consecutive data tokens which advance through the registers at the pace of the clock signal (shown on the Figure 4a with empty and shaded circles inside latches representing latching at two possible clock phases of 'master' and 'slave' latches one of them being transparent when clock is low and the other – when it is high).

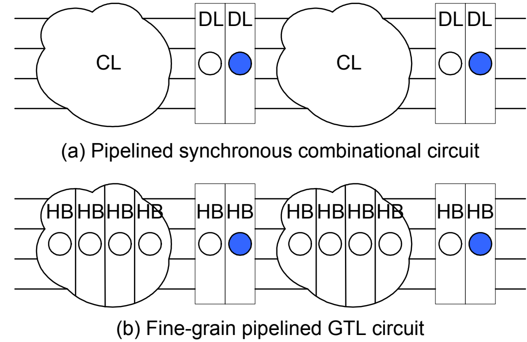

Figure 4 Weaving with clocked registers

The handshake protocols we are considering already support consecutive data tokens separation (with spacer) what makes it possible to substitute every DL with one HB stage (DFF with two HB or one FB stages) at the same time applying basic weaving to the CL portions as with CL gates (basic pipelining). The result is shown in Figure 4b where in the absence of clock the circles represent initial states of the stages (spacers – empty circles and tokens – shaded ones).

If S denotes the number of stages in a pipeline:

(iii) *in HB pipelines distinct tokens are always separated with spacers (no two distinct tokens in any two adjacent stages);*

therefore

(iv) *closed asynchronous HB pipeline maximum token capacity is $\lceil S/2 \rceil - 1$ (S – the number of HB stages);*

(v) *closed asynchronous FB pipeline maximum token capacity is S - 1 (S – the number of FB stages);*

and following from the above as it is also shown in [23]

(vi) *a live closed HB PPN has at least three transitions (HB stages);*

(vii) *a live closed HB PPN has at least one token and at most $\lceil S/2 \rceil - 1$ tokens;*

Now more tokens can simultaneously fit in the pipeline increasing its performance relative to the

synchronous implementation. Let *n* denote the number of FFs and *m* denoting the number of CL levels in the synchronous implementation (RTL implementation token capacity is *n*) the resulting GTL implementation token capacity is $n+m/2$. By weaving:

(viii) *for each FF in RTL implementation in GTL implementation there exist two HB stages one initialized to a spacer and another – to a token;*

(ix) *the number of HB pipeline stages in any cycle of GTL implementation is greater than the number of DLs (or half-FFs) in the corresponding synchronous RTL implementation;*

The initialization is unimportant for linear and loop free data paths since the tokens can enter until the pipeline is full and exit as long as it is not empty. In the presence of loops and/or nonlinearities initialization is crucial for liveness as shown in [23].

The condition (ix) is strict – the additional spacer (vacant position) is required for liveness. This condition is usually satisfied except for the cases like a circular shift register with no logic between FFs.

As long as all necessary conditions are satisfied (the tokens moved to other stages) the identity function stages corresponding to FFs and DLs can be optimized out.

Latches controlled by signals other than clock are often used to implement a selector between the old (stored) and new (input) values rather than separating consecutive data tokens. These obviously cannot be substituted by just HB stages. One of the possible mappings is shown on the Figure 5 (squares represent HB buffer stages while the MUX comprises one more HB stage). This way the loop supplies the previous value that can be directed to the output by the MUX. The value fed to the output is always stored (fed in the loop).

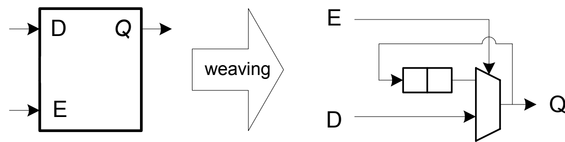

Figure 5 Weaving non-clocked latches

The minimum of three HB stages in a loop is required by (vi).

3.5. Weaving: mapping circuits with loops

In this section let us consider a combination of loop(s) and combinational data paths, for example a design with a data path controlled by a state machine.

Section 3.4 summarized the token capacity aspects of the loops mapping. From (ix), (vi), (vii) for HB implementation ensuring that every loop length is at least greater than twice the number of tokens in the loop is sufficient for correct operation.

Consider the GTL implementation performance. In the presence of loops the entire circuit performance is limited by the latency through the loop divided by the number of tokens following each other in that loop. FSM usually has only one token per loop. Thus, the performance of our design is limited by the FSM cycle time (the time between acknowledging the ith output vector and arrival of the $i+1^{th}$ output vector). Thus the liveness requirement of three stages per loop should be approached as closely as possible. This can be solved automatically in our flow by identifying the FSM and re-synthesizing it with one-hot encoding style (WE is responsible for the FSM identification and the host synthesis engine is called to perform re-synthesis).

Another problem is illustrated on the Figure 6 (each square represent an HB stage). The design has two primary inputs (*a*, *b*), a 14-stage long data path and an FSM – a loop of some width and the length of three stages. In general in the FSMs particular bits of the state vector may depend on the subset (not the entire set) of bits of the state and input vectors. However for simplicity in this example we assume that the every bit of the state vector depends on the entire input and state vector.

The design performance is limited by the maximum of the slowest stage and the FSM cycle time. However at the point *f* data depends on the result of computation at *d*. Both depend on the FSM outputs and for a data token computed at *d* it takes 8 HB stages to arrive at the input of *f*. Assuming that the FSM cannot proceed until all its outputs are acknowledged, the delay of propagating through 9 HB stages is added to the FSM cycle time drastically reducing the system performance. The problem is solved by slack matching first substantially explored in [24]. Slack matching consists in inserting additional identity function (buffer) stages represented by slim rectangles in Figure 6. These stages balance the length of paths synchronized later in the system. In our example in Figure 6b the buffer stages acknowledge the data token traveling to the point *f* so that the FSM can proceed. Similar procedure is applied to the data path to ensure its maximal token capacity. With proper slack matching and no loops the pipeline performance is limited by the total latency of the two slowest adjacent stages.

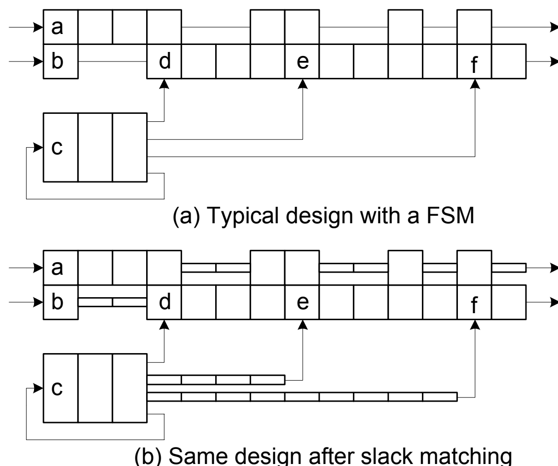

Figure 6 Slack matching

In general the slack matching algorithm balances the number of stages in all converging paths starting from the token sources: primary inputs initialized flip-flops, and latches (as in the case with FSM).

4. Experimental results

To estimate the efficiency of our flow we compare the performance of synchronous and GTL implementations.

It is important to show the speed-up achieved from pipelining a multilevel combinational circuit and the area penalty paid for the pipeline implementation. As it was explained above the GTL cells are specifically designed to meet the isochronic forks requirement satisfied inside the cells so that the communication is delay insensitive. Dynamic GTL cell implementation is advantageous comparing to static one for better area and performance. With that in mind we implemented a simple dynamic logic based library in TSMC 0.18 µm technology obtained through MOSIS. The library was optimized for speed and at this point has only an identity function stage (to be used for slack matching) and as a mapping for clocked DLs and DFFs, and an AND2 gate. With this library we ran the set of MCNC multilevel combinational benchmarks [25]. Timing results for relatively deep and for some shallow benchmarks are summarized in the Table 1. In the Table 1 *d* stands for the depth in gates of the combinational implementation, *cl* – for combinational (conventional gates) implementation latency, *gc* – for the cycle time of the GTL implementation and *gl* – for its forward latency. The area is shown in Table 2 where *comb* stands for the combinational (conventional gates) implementation.

We compared the results with a synchronous implementation using 0.18µm library from Illinois Institute of Technology (sync). The GTL performance was measured using simulation. For synchronous circuits we used the data propagation time from the Synopsys DC report.
As it can be seen from the table the GTL implementations performance is approximately the same for all benchmarks without loops that confirms our expectations – the performance is limited by the cycle time of the slowest stage.

Table 1 Cycle time and latency figures for multilevel MCNC benchmarks

Bench-mark	d	time, ns			Ratio	
		cl	gc	gl	cl/gc	gl/cl
C17	2	0.31	1.25	0.15	0.25	2.0
C1355	12	2.72	1.35	1.03	2.01	2.6
C1908	17	3.71	1.56	1.10	2.38	3.4
C432	14	3.23	1.43	1.26	2.26	2.6
C499	12	2.67	1.32	0.77	2.02	3.5
C880	15	2.58	1.57	0.46	1.64	5.7
cm162a	5	0.93	1.31	0.47	0.71	2.0
cm163a	6	0.91	1.27	0.39	0.72	2.3
cordic	6	1.01	1.25	0.39	0.81	2.7
dalu	13	3.26	1.53	0.48	2.13	6.8
sct	7	0.84	1.38	0.39	0.61	2.2

Table 2 Area comparison for combinational MCNC benchmarks

Benchmark	# of gates	area, um^3 x 10^3		gtl/ comb
		comb	gtl	
C17	6	0.2	1.4	6.97
C1355	546	17.7	123	10.6
C1908	880	14.6	109	7.46
C432	160	6.0	73.6	12.3
C499	202	14.9	92	6.17
C880	383	11.4	111	9.79
cm162a	19	1.2	6.9	5.64
cm163a	16	1.2	7.8	6.52
cordic	102	2.8	13.7	4.96
dalu	1131	27.2	214	7.85
sct	40	2.0	15.4	7.64

Another comparison compares the automated pipelining in synchronous framework and a manually pipelined implementation of a deep combinational circuit with the results obtained with the Weaver flow. In this experiment we synthesized a 10-rounds implementation of the Advanced Encryption Standard. The charts in Figure 7 show the area and performance characteristics of several of its implementations.

The first implementation is a purely combinational non-pipelined implementation by Synopsys DC-Ultra

synthesized from a hierarchical behavioral specification.

Retiming is a technique used to balance stages in synchronous pipeline. Synopsys DC-Ultra includes the `pipeline_design -period 0` command to automatically insert in the initially combinational design as many registers as possible. This command inserts a register at the output of a circuit and tries to move it backwards reducing the clock cycle. Then another register is inserted and moved until the constraints are satisfied. The results of this command were poor for the hierarchical design so we flattened it to get the most of the tool performance. The next (second) columns on the chart present the area and performance of the automatically pipelined flattened design. Here pipelining improved the performance drastically – about 25 times. The area was doubled.

The next implementation running at approximately 578 MHz is manually pipelined [26] and synthesized by the DC-Ultra for the same library.

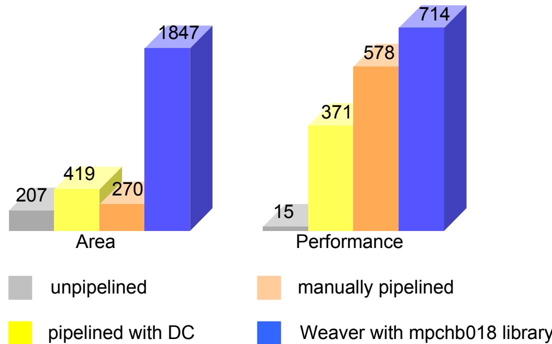

Figure 7 10-round AES implementations comparison

Finally the last bar corresponds to the GTL implementation area and performance. The implementation was synthesized by the Weaver flow using (the same as in MCNC benchmarks) *mpchb* library developed in our lab [20].

5. Conclusion

In this paper we've briefly presented a framework for synthesis of fine-grain pipelined QDI circuits from high-level specification. We have implemented techniques for efficiently re-synthesizing architectures with clocked registers and no loops or where the only loops are inside the FSM. The efficiency of such architectures after re-synthesis is defined by the library.

The behavior specification can be written in VHDL, Verilog HDL or other languages supported as input by Synopsys DC-Ultra. Theseus Logic NCL flow used the same approach utilizing industrial synthesis engine to perform logic synthesis. However in the NCL flow some specific programming is required on the level of input specification e.g. handshake signals for non-linear data paths had to be explicitly specified by the designer. In our flow the initial specification does not need to be tailored for asynchronous implementation allowing for the HDL code reuse.

The architecture of NCL, phased logic and in the de-synchronization implementations mimics the architecture of the synchronous design usually coarse-grain thus not improving the performance of the original design. NCL also adds significant synchronization overhead for wide data paths. GTL circuits are fine-grain pipelined regardless of whether the synchronous implementation was pipelined or not. In addition GTL avoids unnecessary synchronization which even more improves the implementation performance.

Lastly, NCL flow only used proprietary NCL cells while the Weaver flow can be retargeted to any QDI pipeline library.

The experimental results show that GTL synthesis contributes high performance with robustness to process, temperature etc variations and the design automation allowing low-latency domino-like logic at the expense of area overhead and automatic fine-grain pipelining (Table 1, Table 2 and Figure 7).

The performance of synthesized implementation is limited by (1) the cycle time of the slowest stage for the circuits with no internal loops or (2) the worst loop cycle time over the number of tokens in the loop. GTL cell cycle time is slower than the latency of a conventional static gate. This fact makes GTL performance higher only compared if a 6-10 gate deep circuit is pipelined (Table 1). On the other hand the GTL cell latency is lower than that of static gates since our dynamic cells are domino style. This suggests two main applications of GTL circuits – those that take advantage of deep pipelining and those where the forward latency is important.

We did not include any tables on the FSMs in this paper but the performance of a one-hot GTL FSM is usually on the level of 500MHz. The area overhead is about the same as with combinational benchmarks.

The area and performance figures heavily depend on the quality of the library so development of libraries for the flow remains crucial for the methodology to succeed in industry.

6. References

1. Ligthart, 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.
2. 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.
3. Blunno, I., et al. *Handshake protocols for de-synchronization*. in *International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2004.
4. Kondratyev, A. and K. Lwin, *Design of Asynchronous Circuits using Synchronous CAD Tools.* IEEE Design & Test of Computers, 2002. **19**(4): p. 107--117.
5. Cortadella, J., et al. *Coping with the variability of combinational logic delays*. in *Int. Conf. on Computer Design (ICCD)*. 2004. San Jose.
6. Reese, R.B., M.A. Thornton, and C. Traver. *A Fine-Grain Phased Logic CPU*. in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2003)*. 2003. Tampa, Florida.
7. 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.
8. Hartstein, A. and T.R. Puzak. *Optimum Power/Performance Pipeline Depth*. in *MICRO-36 International Symposium on Microarchitecture*. 2003.
9. Singh, M. and S.M. Nowick, *MOUSETRAP: Ultra-High-Speed Transition-Signaling Asynchronous Pipelines*, in *Proc. International Conf. Computer Design (ICCD)*. 2001. p. 9--17.
10. 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.
11. Schuster, S., et al. *Asynchronous Interlocked Pipelined CMOS Circuits Operating at 3.3-4.5 GHz*. in *International Solid-State Circuits Conference*. 2000.
12. 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.
13. 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.
14. 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.
15. Choy, C.-s., et al. *A fine-grain asynchronous pipeline reaching the synchronous speed*. in *ASIC*. 2001. Shanghai, China.
16. Nowick, M.S.a.S.M., *High-Throughput Asynchronous Pipelines for Fine-Grain Dynamic Datapaths.* Proceedings of the 6th IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems ("Async-2000"), Eilat, Israel, 2000.
17. 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.
18. *Fulcrum Microsystems Inc. Web site:* http://www.fulcrummicro.com/technology.htm.
19. Ozdag, R.O. and P.A. Beerel. *A Channel Based Asynchronous Low Power High Performance Standard-Cell Based Sequential Decoder Implementated with QDI Templates*. in *International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 2004.
20. *Weaver: GTL synthesis flow.* http://async.bu.edu/weaver/. 2004.
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. 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.
23. 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.
24. 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.
25. Yang, S., *Logic Synthesis and Optimization Benchmarks Version 3.0*. 1991, Microelectronics center of North Carolina.
26. Verbauwhede, I., P. Schaumont, and H. Kuo, *Design and Performance Testing of a 2.29-GB/s Rijndael Processor.* IEEE Journal of Solid-State Circuits, 2003. **38**(3): p. 569-572.