

Gate Transfer Level Synthesis as an Automated Approach to Fine-Grain Pipelining

Alexandre Smirnov[†], Alexander Taubin[†], Mark Karpovsky[†], Leonid Rozenblyum[‡]

[†]*Boston University*

[‡]*Harvard University*

{alexbs, taubin, markkar}@bu.edu, leonid_rozenblyum@hms.harvard.edu

Register Transfer Level (RTL) synthesis method in clocked designs simplified circuit design and allowed design automation boosting VLSI progress for more than a decade. Shrinking technology and progressive increase in clock frequency is bringing clock to its crisis. 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. This work sketches the Gate Transfer Level (GTL) approach – it shows a general framework for automated synthesis of pipelined asynchronous circuits, presents certain aspects of GTL pipelines synthesis and informally demonstrates the equivalence of resulting GTL implementation to conventional RTL implementation of the same behavior. Experimental results show average 4.3x performance increase on MCNC benchmarks compared to synchronous RTL implementation.

Keywords: synthesis, asynchronous EDA, quasi-delay-insensitive (QDI), ASIC, HDL.

1. Introduction

Popularity of synchronous design and its support by EDA tools on one hand and the crisis of the synchronous paradigm (process variation, signal integrity problems and other physical limitations of synchronous designs) on the other hand resulted in a number of approaches to asynchronous reimplementation of synchronous design. The main idea of such reimplementation is substituting the global clocking by local control communications. It has been explored in a number of publications [1-4].

The main distinctive features of our approach is that it does not only solve the global clocking problem by substituting it with local self-timed control but also replaces the register transfer architecture by gate transfer architecture. This automatically results in very fine grain pipelined circuits regardless of the original synchronous implementation architecture.

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. There are no tools capable of establishing the correspondence between the functionality

of synchronous pipelined and synchronous non-pipelined designs. 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 [5, 6]. For asynchronous circuits there are no such issues since handshake based token propagation and synchronization are internal for a design leaving its interface behavior intact independently of implementation granularity. This is a unique capability of clockless token-based systems.

Phased logic [2, 7] is similar to our approach in the sense of replacing every combinational logic (CL) gate with its dual-rail implementation, however the phased logic design procedure is more complicated. The complexity comes from the encoding scheme ensuring that 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 fewer transitions are required per data token propagation through one stage. On the other hand to ensure safeness it requires additional feedbacks insertion in the original netlist [2] on the design stage.

De-synchronization [3] is another approach addressing designing asynchronous circuits from synchronous netlists. However the focus of the approach from [3] on low area overhead makes it use delay padding thus not providing the robustness of quasi-delay-insensitive (QDI) data driven designs. This approach results in circuits architecturally equivalent to the RTL design.

Null Convention Logic (NCL) [1, 4] EDA flow from Theseus Logic exploits the idea of synthesizing large designs using commercial synchronous synthesis engine and substituting globally clocked synchronous registers in the data path by asynchronous registers communicating asynchronously through delay insensitive handshakes. In this flow the dual-rail data encoding has also been employed to enable completion detection. The design flow from [1, 4] is the most similar to our approach from the EDA flow architecture point of view. However NCL flow produces pipeline circuits architecturally equivalent to the synchronous RTL implementation usually coarse-

grain pipelined. Heavy synchronization of completion signals at registration points slows down the design. Another NCL drawback is restrictive behavior specification. It requires the designer to manually specify some handshake signals in the VHDL specification what makes the existing RTL reuse hard.

None of the above approaches offer support for automated pipelining therefore keeping the performance on the level of the original design.

The GTL 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 a GTL implementation as it is briefly explained in section 2. By default pipelining is done on the gate level – the finest degree of pipelining resulting in extremely high-performance designs almost impossible to implement in synchronous methodology.

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 [8] while the state transitions from *spacer* (*NULL*) to proper codeword (*DATA* or *token*) in the set phase, and then back to *NULL* in the reset phase. A simple delay-insensitive scheme is obtained by encoding *DATA* codewords with one-hot codes, and the spacer *NULL* – with 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 $a.0, \dots, a.n$. 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

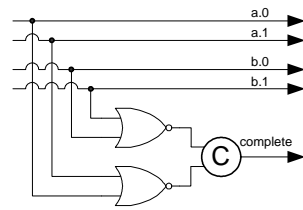


Figure 1, Completion detection for DI bus

with the function $g = x1 \times x2 + g \times (x1 + x2)$, known as a Muller’s C-element [9].

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 [10-13], where the inter-stage handshake circuit is considered as a template parameterized by the stage function.

An example of dynamic implementation of a GTL gate implementing the AND2 function is shown on the Figure 2. Illustrated is the Reduced Stack Precharge Half Buffer (RSPCHB) template from [11]. The only block specifying the gate logical function is *F*. The rest is typical for most of the stages. *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 (set/reset) control, *CD* for completion detection and *M* – for memory. ‘Staticizers’ (or *keepers*) formed by adding weak inverters as shown on the Figure 2, can store the stage output value for an unlimited time eliminating timing assumptions. At the same time keepers solve the charge sharing problem and improve the noise margin of pre-charge style implementations. 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.

Several asynchronous pipeline styles exist [12-15]. In this work we’ve chosen the simplest data driven style presented in [16] for its minimal and local (inside the stage) delay assumptions and robust (delay insensitive) inter-stage communication.

A *dedicated library* with each cell representing an entire GTL pipeline stage makes satisfying in-stage timing assumptions a library design problem. If inter-stage communication is delay insensitive the implementation functionality no longer depends on place and route. A dedicated library is under development at this time. We used the library from [17] as a prototype for area estimations, however the library is not complete for automated synthesis (not fully characterized) and resetting dynamic gates to given values is not addressed. These and

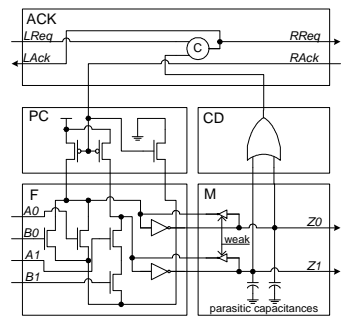


Figure 2, GTL AND2 dynamic implementation

some other issues encouraged us to start developing a new library which is out of the scope of this paper.

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 eliminated in GTL designs thanks to monotonic data transitions, completion detection and data-dependent control.

From the example of the Figure 2 it can be seen that memory and logic function implementation are of the same cost and speed as synchronous domino-like counterparts. The main sources of area overhead are the dynamic C-element for handshake implementation (*ACK*), *CD* and *ack/req* synchronization (can be seen from the Figure 4).

3. Design flow

We have implemented an EDA flow [18] maximizing the use of commercial design tools. It executes three steps.

First, a single-rail synchronous implementation is synthesized for a high-level behavioral specification, optimized and mapped into the library composed of conventional single-rail gates functionally equivalent to the target GTL library. Like in [1, 4] a commercial RTL synthesis engine (currently only Synopsys DC Ultra) is used. As opposed to the attempts to express asynchronous formal models in HDL (Martin’s CHP in case of [19] and [20] or Signal Transition Graphs in case of [21]) 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, similarly to [22], the single-rail netlist is expanded into a dual-rail QDI fine-grain pipelined (GTL) implementation. This expansion called ‘weaving’ is the main topic of the present paper.

All local wiring related to dual-rail expansion and handshake implementation is added on this stage by the *Weaver Engine* (WE) and is invisible for RTL users – no additional HDL code is necessary. Only the functionality of F (AND2 on the Figure 4) is visible for RTL designer and synthesis tool.

Finally the GTL netlist is mapped using a commercial mapping tool (currently the same Synopsys DC Ultra) into the asynchronous pipeline GTL library. Using 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 P&R and other tools following synthesis in the design flow.

The GTL cells (stages) are complex sequential devices and as such are not recognized by synchronous tools (like DC-Ultra). As a consequence the target GTL library is a set of black boxes for the RTL synthesis engine and no optimization is allowed on this stage. This makes it

necessary to implement the GTL architecture optimization algorithms in the Weaver Engine.

3.1. Flow architecture

The Weaver flow architecture (Figure 3) is an extension and generalization of that by Theseus [1, 22, 23].

Its architecture is shown on the Figure 3. The notation is self-explanatory except for the srGTL lib – a library containing single rail gates functionally equivalent to the gates from target GTL library (can be seen as data1 function implementation) and their dual counterparts.

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 – a layer unifying the physical target library cells interface. 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.

Weaver engine – the heart of the flow is a VHDL compiler and a synthesis engine on its own. It is based on

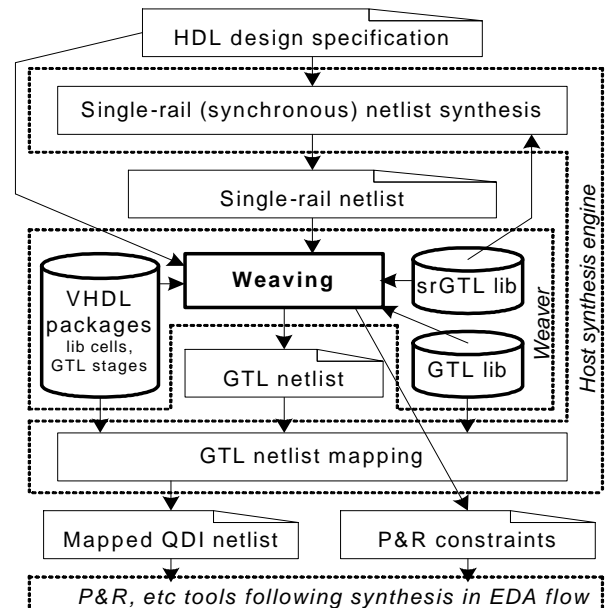


Figure 3, Weaver flow architecture

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 ‘weaving’ explained further.

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 built with the assumption that gates are converted to stages communicating with req/ack signals requiring synchronization.

3.2. Weaving: general approach

Weaving is a procedure of synthesis of a GTL implementation for a certain behavior from a synchronous implementation of the same behavior.

To transform an RTL implementation into a GTL the data wires are substituted by *channels* generally comprising *req*, *ack*, *data0* and *data1* lines.

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

Channels depicted in Figure 4 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 pipelining styles do not use all four (*req*, *ack*, *data0*, *data1*) communication lines (e.g. PCHB from [11] does not use *req*). Those still fit in the framework.

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 replacing every gate a pipeline stage provides the finest grain gate level pipelining resulting in the highest performance implementations. The only gates not mapped to stages are inverters – these are implemented as data wires cross-over when dual-rail one-hot encoding is used and as such require no additional hardware.

(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 NULL (spacer);*

Replacing a single-rail combinational gate implementing the AND2 function with a GTL gate (or stage) and the wires with channels is illustrated on the Figure 4. On this figure *ACK* stands for handshake circuitry, *PC* – for phase control, *CD* – for completion detection, *M* – for memory and *F* – for dual-rail implementation of the AND2 function.

Many pipeline styles suit GTL framework. Most efficient ones are dynamic cells designed as whole stages like in [11].

Dual-rail (DR) data encoding is used to simplify completion

detection. 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 *NULL* or *spacer* representing ‘no data’ state.

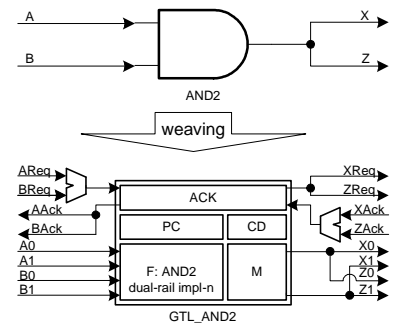


Figure 4 Weaving

3.3. Weaving: mapping latches and flip-flops

The section 2 addresses general 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 CL gates, D-latches (DL) and D-flip-flops (DFF).

The pipeline cells can be divided into two categories *full-buffer (FB)* and *half-buffer (HB)* [11]. The two are distinct by the *token capacity* – the number of data portions that can fit in a pipeline of a given length where the length is measured in the number of stages *S*.

(iii) *closed asynchronous HB pipeline maximum token capacity is $\lfloor S/2 \rfloor - 1$ (where *S* is the number of HB stages);*

(iv) *closed asynchronous FB pipeline maximum token capacity is $S - 1$ (*S* is the number of HB stages);*

therefore

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

“Distinct” in (v) is important since in real pipeline the token propagation does not occur immediately letting the same token occupy more than one stage for some time.

HB pipeline stage implementation requires smaller area than a functionally equivalent FB stage so in search for smaller area overhead in this work we mostly consider HB GTL implementations.

Combinational logic (CL) in RTL implementations is often pipelined to increase the implementation performance. Figure 5 (a and b) illustrate that. As it is shown on the Figure 5 the CL is broken into two stages by inserting a DFF. This way the performance is increased approximately by the factor of two (Figure 5b) since the

intermediate result is stored in the DFF. Empty and shaded circles in the latches (consider a DFF as a pair of D-latches) reflect alternative clock phases of ‘master’ and ‘slave’ latches (one of them is in storage mode when clock is low and the other – when it is high). As with

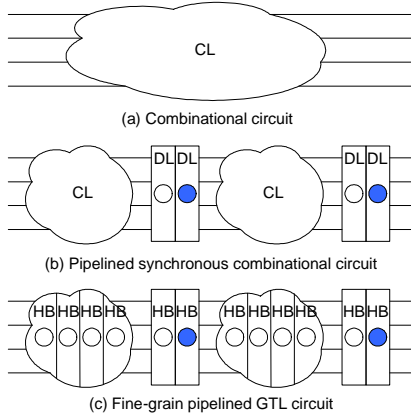


Figure 5 Latches and flip-flops mapping: deep CL

CL gates (basic pipelining) we can substitute every DL with one HB pipeline stage. At the same time basic weaving is applied to the CL portions and the result is shown on the Figure 5c. At this point there is no clock so the circles represent alternating *NULL* (empty circles) and *DATA* states (shaded circles) as explained in section 4.

Now more data portions can simultaneously fit in the pipeline increasing its performance relative to the synchronous implementation. This way the original implementation is fine-grain pipelined. Let n denote the number of DFFs and m – of CL levels in the synchronous implementation (RTL implementation token capacity is n) the resulting GTL implementation token capacity is $n+m/2$. Therefore by weaving:

- (vi) for each DFF in RTL implementation there exist in GTL implementation two HB stages one initialized to a spacer and another – to a token;
- (vii) the number of HB pipeline stages in any cycle of GTL implementation is greater than the number of DLs (or half-DFFs) in the corresponding synchronous RTL implementation;

Note that the condition (vii) is strict (the number of stages is strictly greater). This is because the closed pipeline token capacity for synchronous pipelines is one greater than that of asynchronous pipelines where the tokens propagation is not synchronized and an additional spacer (vacant position) is required. This condition is usually satisfied except for rare cases like a circular shift register with no logic between DFFs (e.g. Figure 6 and Figure 12). Therefore:

- (viii) GTL pipeline token capacity is greater or equal to that of the synchronous implementation;

4. Modeling behavior with Petri nets

We use Petri nets (PN) [25] to model the behavior of the original synchronous and resulting GTL circuits. In

the subsection 5 we demonstrate the correctness of the resulting GTL implementation.

We use high level abstraction where PN markings represent position of tokens (data portions) in the pipeline and not the states of signals (as in Signal Transition Graphs).

With the low computational complexity requirement in mind in contrast to [2, 7] our flow does not utilize PN based model for synthesis. We are only using the Petri nets to proof the correctness of weaving algorithms.

4.1. Linear case.

In synchronous implementations each pipeline stage is implemented with a D-flip-flop (DFF) or two D-latches (DL) with alternating clocks to store the result of data processing in the CL. It is almost the same since a DFF comprises two D-latches and an inverter to provide alternating clock for them as shown on the Figure 6a, b. Synchronous pipeline model is shown on the Figure 6c where $t1$ represents the $DL1$ state change, $t3$ – $DL2$ state change etc while the transitions $clk0$, $clk1$ represent clock edges.

For asynchronous pipeline implementations the model from [8] (Figure 6d) can be used. It restricts PN in such a way that for every two transitions t_i , t_j for which there exists a place p_k such that p_k is a postcondition for t_i and a precondition for t_j , there exists p_l such that it is a postcondition for t_j and a precondition for t_i . We refer to such PNs as **Pipeline Petri nets (PPNs)** [8]. Since the pairs of conditions like p_k , p_l model a pipeline stage we denote the pair as a **stage state** and the stage state with adjacent transitions as a **stage**. A PPN consists of *stages* in such a way that

- (ix) no stage state is shared between any two stages.

To ensure liveness and safeness initial PPN markings are restricted to those where

- (x) exactly one place is marked in every stage state.

Depending on whether the post- or precondition is marked for a stage the latter is said to contain a *token* or a *spacer*. In the course of PPN execution adjacent stages having opposite markings can exchange their states by firing transitions. Tokens propagate in one direction (*the direction of data propagation*) while spacer – in the opposite. The PPN feedback arcs along with proper initial marking M_0 preserve PPN 1-safeness. Non-safe model is useless since with current interpretation every PPN token represents a data portion held by the pipeline stage. Multiple tokens in one stage would mean that more than one data portion is stored in a given stage what is impossible.

On the PPN firing the transition $t1$ corresponds to processing data in the $CL1$ and storing it in $DL1$, $t3$ – storing data in $DL2$, $t4$ – processing at $CL2$ and storing in $DL3$. Clearly PPN stages represent pipeline stages and the

PPN tokens –data portions. Even if the data does not change on the input from one clock cycle to another the new token is still introduced in the pipeline. The nature of synchronization makes the tokens always alternate with spacers. Every edge of the clock cycle causes half of latches to be transparent propagating all data one step forward, the next edge makes the other half of them transparent propagating data one more step. This way, unless we distinguish tokens every two steps the pipeline state is the same.

One can observe that transitions are shared by adjacent stages. This is because transitions are interpreted as the events of exchanging a token with a spacer between adjacent stages.

On the Figure 6 stages are delimited by vertical lines.

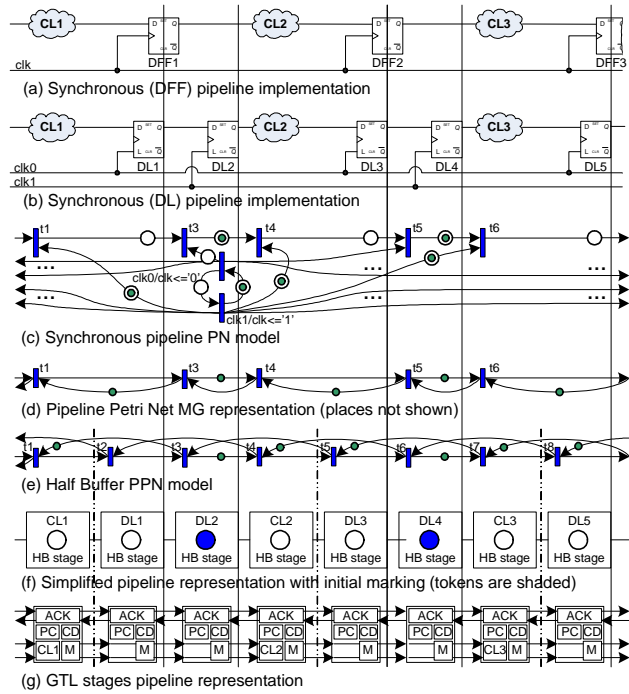


Figure 6 Modeling pipeline with Petri Nets

Linear PPNs and HB PPNs are *marked graphs* (MG) [26]. Thus starting from the Figure 6d we imply PN places on arcs without depicting them and put markers on the arcs themselves where necessary.

To model HB GTL pipelines we modify the PPN model to preserve the property (iii) of HB pipelines. Indeed in the PPN one token can immediately approach the previous one thus allowing for the token capacity of $S-1$ natural for FB pipelines.

By making the PPN feedbacks twice as long (to span over one stage) we reduce the modeled pipeline token capacity by the same factor of two. We call this model HB PPN and use it to model HB pipelines (Figure 6e).

In HB PPN every two adjacent stages can be in one of the three stable states: token-spacer (TS), spacer-token (ST) and spacer-spacer (SS). TT would violate (v). Every such a pair is modeled by tree HB PPN transitions, two forward (relative to the data propagation direction) and one feedback arcs. ($CL1, DL1$) on the Figure 6f are in SS state stages are modeled by the feedback arc ($t3, t1$) marked (forward arcs are unmarked) on the Figure 6e. Observe the markings corresponding to the pairs ($DL1, DL2$) and ($DL2, CL2$) – in both cases the feedback arc is unmarked while ($t3, t4$) being the second forward arc for ($DL1, DL2$) and the first for ($DL2, CL2$) is marked.

Similarly to the stage notation in PPN we denote every t_i, t_j, t_k such that (t_i, t_j) and (t_j, t_k) are forward and (t_k, t_i) – feedback arcs along with the arcs connecting them as *FB-stages*. A valid marking would assigned to a *FB-stage* defines one of its three states (TS, ST or SS) i.e.

(xi) a HB PPN marking is valid iff every FB-stage in the HB PPN has exactly one marker;

Based on the token capacity equivalence of HB PPN model to the HB pipelines as well as on the intuition provided above we assume that:

(xii) GTL style pipeline is properly modeled by HB PPN.

Note that during weaving both DL and portions of CL (by default individual gates) are mapped to HB GTL stages (increasing the pipeline token capacity) hence starting from the pipeline on Figure 6e there are 8 stages (assuming CL1, CL2, CL3 are one gate each).

4.2. HB PPN liveness

From (iii) it follows that

(xiii) a live closed HB PPN is at least 3 HB stages long;

It is easy to see that a two stage closed HB PPN cannot be live. Similarly from (iv) it follows that the shortest live PPN must be at least two stages long ($S=2$). On the other hand a FB stage can be considered to comprise two HB stages, thus the liveness condition for HB PPN (xiii) is stronger.

The Figure 7 illustrates the only valid (satisfying the properties (iii), (v), (ix) and (xi)) three-stage HB PPNs initial markings possible. Other (than reachable from those shown) initial markings are not valid due to the token separation condition (v).

Only the HB PPN on the Figure 7a is live for there are no tokens to exchange on the Figure 7b. Clearly

(xiv) a live closed HB PPN has at least one token and at most $\epsilon S/2 \hat{u} - 1$ tokens;

Thus for liveness a closed linear HB PPN requires the following conditions: (xi), (xiii) and (xiv).

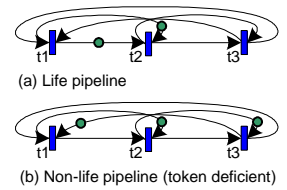


Figure 7 Closed HB PPNs

4.3. Nonlinear pipelines and conditional behavior

We will distinguish conditional and unconditional pipeline branching.

Consider only unconditional branching first. It corresponds to deterministic behavior where:

(xv) *the token flow is deterministic and does not depend on data itself;*

Unconditional branching is shown for HB PPN, PPN and PN on the Figure 8a, b and c respectively. Empty pipeline (no tokens) is shown in all three cases. Deterministic HB PPN contains no choice points therefore it is a marked graph. It is proven in [26] that

(xvi) *a marked graph is live iff M_0 assigns at least one token on each directed loop (or circuit);*

Consider directed circuits in HB PPN. Every FB-stage is a directed circuit by itself but it is always marked by definition (xi). The only circuits remaining are the loops consisting of only feedback or only forward arcs – design loops (circuits). A situation when no

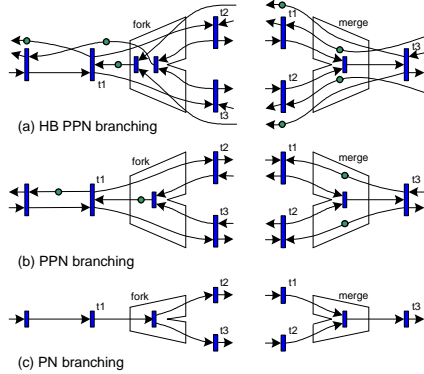


Figure 8 Modeling nonlinear pipelines

feedback arcs are marked corresponds to overloaded pipeline with $S/2$ tokens which is not valid as it violates (xiv). Another case where none of the forward arcs are marked corresponds to a token deficient pipeline also violating (xiv). The number of tokens in a pipeline can be counted as the number of markers on the forward arcs (shown on the Figure 8c separate from any other arcs). Thus:

(xvii) *for a HB PPN to be live each of its directed circuits composed of forward arcs as a closed HB PPN must satisfy the conditions (xi), (xiii) and (xiv);*

Data flow steering is performed by means of multiplexers (MUX) and demultiplexers (DEMUX). The data value of the MUX/DEMUX control channel defines which input/output data channel is chosen. What the “chosen” means here depends on the MUX (DEMUX) implementation. There are two ways of implementing such a device. Consider an example of a 2-to-1 multiplexer (MUX12).

Combinational implementation of MUX12 would be an implementation of the function $Z=AC+BC'$ where X is the output, A, B – input data channels and C is a control

channel. The channels A, B and C will be acknowledged once the data tokens are present at all of them. Thus such a MUX behaves as a 3-input gate and does not affect the token flow. The choice of the channel only affects the data in the token generated at the output channel X.

On the other hand **selective** implementation would acknowledge only the “selected” channel (thus either A and C or B and C are acknowledged every time a token is produced on the channel X). The latter makes the token flow dependent on the data itself or in other words non-deterministic relative to the design architecture what makes it impossible to guarantee the overall implementation liveness. Selective multiplexing however is rarely needed and currently is left as a tuning option for experienced designers.

Combinational implementation of multiplexing token flow-wise is equivalent to the case of unconditional branching and as such creates no liveness problems. It is used by default in or flow.

5. Correctness

Similarly to [3] we define the correctness of weaving to be three-fold. The GTL implementation must be:

1. *safe* as it follows immediately from (x);
2. *live* to ensure continuous operation (pipeline never halts);
3. *flow equivalence* ensures that in both RTL and GTL implementations the order in which corresponding data portions appear in the corresponding storage elements (latches in RTL and HB stages in GTL) is the same.

To show the weaving correctness we need the following assumption which to the best of our knowledge holds for the circuits synthesized by contemporary RTL synthesis engines we may use as host synthesis engines:

(xviii) *every feedback loop in synchronous implementation contains at least one DFF (or a pair of DLs);*

Liveness. We are using HB PPN model for the proof of liveness as we primarily target HB pipelines for which this model is adequate according to (xii).

Finally as follows from (vi), (vii), (xvii), (xviii) the GTL implementation is live as long as all the mentioned conditions are satisfied. Thus, weaving guarantees the implementation liveness.

Flow equivalence can be proven along the ways of [3] since the communication protocols are compatible. The difference comes from the fact that in the de-synchronized implementation (asynchronous) has the same set of latches as the original (synchronous). This is in general not true for the GTL circuits exhibiting the property (vi). However in a closed linear pipeline the number of tokens stays the same regardless of the number of stages in accordance with (v).

We omit the rigorous proof here as it would repeat that of [3] but rather include the Figure 9 illustrating the data portions propagation through the pipeline shown on the Figure 5. In each case the first line shows the state of storage elements after initialization (numbers are random and represent data tokens). In RTL implementation (a) the tokens propagate simultaneously – one DL at a time.

In GTL implementation (b, c) the latency of different stages can be different so the tokens can propagate at different paces. The speed depends on the latency of the stages being passed. However functional dependencies do not allow the result to be computed before both operands are present ensuring the correctness. Thus, as long as two channels in a bus are not functionally related in the module their outputs arrive independently at different times but the order of the tokens is always preserved i.e. the i^{th} token in certain HB in GTL implementation carries the same data as the corresponding DL of synchronous implementation on the i^{th} clock cycle thus weaving preserves the flow equivalence.

IN		DL	DL		DL	DL
4		4	5		5	3
1		1	4		4	5

(a) Pipelined synchronous combinational circuit

IN	HB	HB	HB	HB	HB	HB	HB	HB	HB	HB	HB	HB
4	N	N	N	N	5	N	N	N	N	N	N	3
4	4	N	N	N	5	5	N	N	N	N	N	3
9	N	7	N	2	N	1	N	4	N	5	N	3

(b) Fine-grain pipelined GTL circuit

IN	HB	HB	HB	HB	HB	HB	HB	HB	HB	HB	HB
4	N	N	N	5	N	N	N	N	N	N	3
4	4	N	N	5	5	N	N	N	N	N	3
7	N	2	N	1	N	4	N	5	N	3	

(c) Fine-grain pipelined GTL circuit (no redundant stages)

Figure 9 Data portions propagation

It follows from the flow equivalence and the property (i) that the corresponding data portions pass through the same set of CL blocks what in turn would guarantee that the same output data sequence is obtained from both implementations for a given input data.

The Figure 9 demonstrates pipeline filling i.e. tokens are not consumed (acknowledged) by the receiver on the output.

6. Optimizing the number of stages

Let us examine the resulting fine-grain pipeline for the area overhead. The stages corresponding to DFFs and/or DLs carry no functionality (*identity function stages*) and can be optimized out as redundant to reduce the area and latency overhead as long as the initial marking can be transferred to other stages, the data dependencies are not

affected (since the stages to be removed implement identity function) and all the conditions necessary for the implementation liveness are satisfied.

Suppose the Figure 10a depicts the initial state – two identity function stages are initialized to tokens. If the CL portions are deep enough (greater than 2 levels) redundant stages can be optimized out with the initialization moved to the previous stages implementing logic functions (Figure 10). We demonstrated how pipelined RTL code can be reused. The synthesis result (in terms of pipelining) is the same for both initially pipelined and not pipelined specification. Thus the designer’s pipelining effort can be saved and existing pipelined design specifications reused.

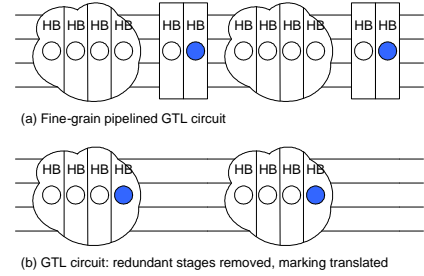


Figure 10 Removing redundant stages

In the example from Figure 10 for simplicity it is assumed that the CL portions have four layers of combinational gates each. Such an assumption is barely practical. A more realistic example is shown on the Figure 11. The combinational gates can be topologically sorted with respect to their mutual dependency in such a way that the gates that only depend on the primary inputs (relative to the CL block) are placed on the leftmost level (marked as 1), those that depend on the gates from the level 1 and/or primary inputs – on the level 2 etc (Figure 11a). The HB stages corresponding to DFFs in registers are no longer synchronized so (Figure 11b). Data dependencies that span over layers of stages will introduce token propagation delays (distinct paths will have different lengths). Such dependencies are broken by introducing identity function stages (lightly shaded rectangles on the Figure 11b). This procedure is known as slack matching [10]. It consists in balancing distinct independent paths along the pipeline to balance their token capacities and thus increase the performance. In this example the outputs of the first CL are fed only to the second CL. This makes it possible to merge the two blocks in such a way that (vi) is satisfied.

Generally the best performance is achieved if the ‘shape’ of a module is rectangular i.e. all paths along the pipeline have the same slack. However the area overhead induced by slack matching can be significantly reduced if the advantage can be taken of the shape of the modules to be connected as it is shown on the Figure 11c. The resulting module is made rectangular (Figure 11d).

Current implementation of the slack matching in our flow assumes that HB stages found in the library have

approximately the same delay so the path lengths are measured in the number of stages.

The slack matching is currently only performed on non-cyclic paths.

The procedure always preserves the shape of submodules in hierarchical designs and only the top module is forced to have rectangular ‘shape’ (white rectangles represent the stages added on this stage). Note that two tokens are added to the initial state of two stages in each path to correspond to the number of DFFs in each path in the original RTL implementation.

The complexity of such analysis is roughly $O(|X||C|/2)$ where $|X|$ is the number of primary inputs and $|C|$ is the number of connection points in the netlist (not the number of gates to support subdesigns of non-trivial ‘shape’ and hence different distances between distinct input-output pairs of a module).

Another example (Figure 12) presents a shift register/counter style design where the combinational logic between flip-flops is shallow (one level to none) in the synchronous implementation (Figure 12a). Here the stages corresponding to synchronous flip-flops are no longer redundant and cannot be optimized out as in the previous examples.

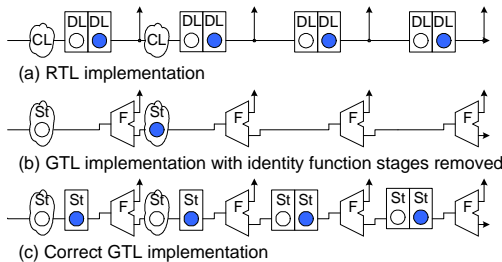


Figure 12 Latches and flip-flops mapping: shallow CL

Indeed since the fork modules are only *ack* synchronizers and represent no data buffering (not stages)

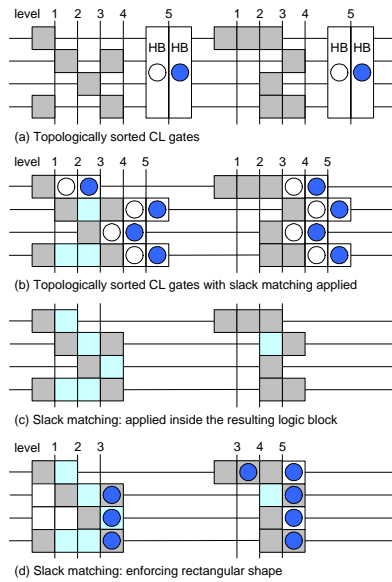


Figure 11 Slack matching

the implementation achieved by removing the flip-flops Figure 12b violates the condition (vi) and has the functionality distinct from the specification and the synchronous implementation: the last three parallel output bits represent the same bit branched to three while the synchronous implementation is a shift register outputting three consecutive distinct data portions.

In synchronous implementation a DFF (or a pair of DLs) holds one token. Due to the nature of synchronous implementation it is initialized with one token per DFF. To ensure liveness a GTL implementation with loops (circuits) must be initialized so that every pair of HB stages (or an FB stage) corresponding to a DFF in synchronous implementation is initialized to a token (DATA). For performance reasons a GTL implementations can be initialized with more tokens than the original RTL implementation but this topic is left out of the scope of the paper.

The condition (vi) is clearly met for the implementation on the Figure 12c. For the stages with no combinational logic every flip-flop has been mapped to two HB stages initialized with one token and one spacer. Note that in the first bits combinational gates mapped to GTL HB stages already provide one stage each. Hence one HB corresponding to a DL (half DFF) is optimized out. The stage number optimization is performed automatically in our GTL flow.

The *clock* signal is optimized out in the final design. Such an optimization 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).

7. 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 [27]. The examples in the Table 1 are combinational multilevel circuits. These are used to optimize pipeline balancing technique. Similarly to slack matching [28] 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		g/r	Depth
		rtl	gtl		
C17	6	2857	667.6	0.23	2
C1355	546	37	120.6	3.26	12
C1908	880	21.7	105.2	4.84	17
C432	160	189.9	351.0	1.85	14
C499	202	43.0	122.0	2.84	12
C5315	2307	44.8	181.3	4.04	16
C880	383	66.5	246.4	3.71	15
apex6	238	65.8	158.9	2.80	11
cm162a	19	63.0	388.0	6.16	5
cm163a	16	63.2	491.2	7.78	6
Cordic	102	47.8	289.1	6.04	6
Dalu	1131	38.2	268.0	7.02	13
frg2	526	44.7	223.3	5.0	10
Lal	71	40.2	215.6	5.36	8
Sct	40	55.0	242.6	4.41	7
X4	136	60.9	283.0	4.65	7

Avg

4.37

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).

Table 2 Performance comparison

Example	Performance, MHz			Area, $\mu\text{m}^2 \times \text{E}+06$		
	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

We also compared the synthesis results for an Advanced Encryption Standard (AES) [30] 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 [31] because it has a balanced power dissipation independent from the data patterns at bit lines.

The synchronous implementation numbers were obtained with the same library ($1/\text{delay} \times 10^6$). 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 far from final we could observe a promising area overhead versus performance increase trade-off that may be explored using different timing assumptions and different pipelining granularity. In our AES10 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. However we know from previous experience and [1, 22, 23] 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.

8. Conclusion

In this paper we've briefly presented a GTL framework and weaving technique to compile fine-grain pipelined QDI circuit from a synchronous implementation. We've shown how the basic synchronous constructs are mapped into asynchronous pipeline. By modeling pipeline behavior with Petri nets we've shown that such a translation leads to a functionally equivalent QDI implementation with the same or greater degree of pipelining.

Increasing the pipelining degree leads to performance improvement as it can be seen from experimental results. Obviously the performance increase is due to fine-grain pipelining and slack matching and as such is proportional to the depth of the RTL implementation. AES implementation is very deep, what explains the drastic performance improvement.

The automatically synthesized asynchronous Advanced Encryption Standard implementation with static standard cell library demonstrated a performance increase of up to 36.4x (Table 2) 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.

9. References

1. 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.
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. 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.
6. Hartstein, A. and T.R. Puzak. *Optimum Power/Performance Pipeline Depth*. in *MICRO-36 International Symposium on Microarchitecture*. 2003.
7. 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.
8. Varshavsky, V.I., et al., *Self-timed Control of Concurrent Processes*. 1990: Kluwer Academic Publishers.
9. 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.
10. Martin, A.J., et al., *The Design of an Asynchronous MIPS R3000 Microprocessor*, in *Advanced Research in VLSI*. 1997. p. 164--181.
11. 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.
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. 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.
15. 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.
16. 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.
17. *USC's PCHB Based Asynchronous Gate Library*. <http://jungfrau.usc.edu/AsyncLib.html>.
18. Weaver: *GTL synthesis flow*. <http://async.bu.edu/weaver/>. 2004.
19. 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.
20. 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.
21. 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.
22. Kondratyev, A. and K. Lwin, *Design of Asynchronous Circuits by Synchronous CAD Tools*, in *Proc. ACM/IEEE Design Automation Conference*. 2002. p. 411-414.
23. 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.
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. Murata, T., *Petri nets: Properties, analysis and applications*. Proceedings of the IEEE, 1989: p. 541-580.
26. Commoner, F., et al., *Marked directed graphs*. Journal of Computer and System Sciences, 1971. **5**: p. 511-523.
27. Yang, S., *Logic Synthesis and Optimization Benchmarks Version 3.0*. 1991, Microelectronics center of North Carolina.
28. 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.
29. Sulisty, 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. *FIPS PUB 197: Advanced Encryption Standard*, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
31. Hess, E., et al. *Information Leakage Attacks Against Smart Card Implementations of Cryptographic Algorithms and Countermeasures - A Survey*. in *EUROSMART Security Conference*. 2000.