A METHODOLOGY FOR CORRECT-BY-CONSTRUCTION LATENCY INSENSITIVE DESIGN

Luca P. Carloni
University of California at Berkeley
Berkeley, CA 94720-1772

Kenneth L. McMillan and Alexander Saldanha[1]
Cadence Berkeley Laboratories
Berkeley, CA 94704-1103

Alberto L. Sangiovanni-Vincentelli
University of California at Berkeley
Berkeley, CA 94720-1772

Abstract
In Deep Sub-Micron (DSM) designs, performance will depend critically on the latency of long wires. We propose a new synthesis methodology for synchronous systems that makes the design functionally insensitive to the latency of long wires. Given a synchronous specification of a design, we generate a functionally equivalent synchronous implementation that can tolerate arbitrary communication latency between latches. By using latches we can break a long wire in short segments which can be traversed while meeting a single clock cycle constraint. The overall goal is to obtain a design that is robust with respect to delays of long wires, in a shorter time by reducing the multiple iterations between logical and physical design, and with performance that is optimized with respect to the speed of the single components of the design. In this paper we describe the details of the proposed methodology as well as report on the latency insensitive design of *PDLX*, an out-of-order microprocessor with speculative-execution.

1. Introduction

The advent of deep sub-micron (DSM) process technologies, 0.13μ and below, has generated a flurry of predictions on the effects of the inevitable dominance of wire delays on chip design. Although there is a certain amount of disagreement between the various studies on interconnect latencies in future design generations [9, 10], there is unanimity that the delay of a "long" wire will play a dominant role in logic synthesis and optimization. Recent ad-

[1] Author is currently with Softface, Inc.

vances on interconnect optimization techniques (such as interconnect topology optimization, optimal buffer insertion and sizing, optimal wire-sizing) can help to reduce interconnect delays significantly [8], but they are not able to reverse the trend of growing gap between device and interconnect performance [7]. In the current standard-cell design methodology, logic synthesis is performed using delay estimates for library modules that are parameterized to account for loading factors and transition (or slew) rates. As the delay of long wires become larger relative to gate delays, these estimates become increasingly sensitive to layout. Attempts have already been made to account for layout effects by performing floor-planning and wire-planning on register-transfer level (RTL) descriptions [25]. Such an approach requires extreme precaution in deriving constraints for synthesis tools, since any wire whose delay approaches a single clock may cause a failure to meet the timing constraints.

In this paper, we propose an alternative synthesis methodology that produces designs functionally insensitive to the latency of long wires. Given a synchronous design consisting of several communicating modules, automatic synthesis techniques are used to generate a functionally equivalent synchronous implementation that can tolerate arbitrary communication latency between modules. The overall goal it to achieve a robust design implementation that has as high a throughput as possible. As a preliminary assumption, each module must satisfy the *stallability* property, meaning that it can be stalled for an arbitrary amount of clock cycles without losing its internal state. In our implementation, the modules of the design communicate over channels, using a standard protocol that is insensitive to latency. This protocol allows a channel to run a number of clock cycles ahead of or behind other channels. The resulting system is guaranteed *by construction* to be functionally equivalent to it. The system maintains the appearance of a fully synchronous system despite the non uniform latencies along communication channels of the actual implementation.

The methodology is presented in Section 2 and discussed with respect to previous work in Section 3. In Section 4, we summarize the theory of latency insensitive protocols. In Section 5, we address some issues related to latency insensitive protocol implementation. In Section 6 we report on performance evaluation of the latency-insensitive design methodology for a fairly complex prototype system.

2. The Methodology

The proposed methodology is based on the automatic synthesis of a *communication architecture* implementing a *latency insensitive communication protocol*. It consists in a succession of five basic steps:

1 The designer starts with a completely synchronous specification of the system and with a collection of *modules*, which can be either acquired as

intellectual property (IP) cores from a (internal or external) third-party or can be specified as “synthesizable” code using a hardware description language such as VERILOG or VHDL.

2 Communicating modules are connected by means of *channels* as illustrated in Figure 1. Each channel operates using a latency-insensitive communication protocol and is made up of wires and logic blocks called *relay stations*. The wires of a channel are laid out together and share physical characteristics. The relay stations consist of latches together with logic gates implementing the functionality related to the latency-insensitive communication protocol.

3 Each module is encapsulated within a logic block called *shell*, playing the role of interface towards the communication architecture.

4 The layout is obtained using standard *place & route* tools.

5 A post-layout optimization step is performed to insert the necessary number of relay stations into each “critical channel” to ensure that the cycle time is met (*channel segmentation*). Some iterations may be required, but they are limited to each channel separately, while logic and layout of all modules remain untouched.

The essential point in this methodology is the *orthogonalization of concerns* between behavior and communication. Since the communication mechanism is automatically synthesized (as described later in this paper both relay stations and shells can be built with no intervention of the designer based only on the theory of latency insensitive protocols), the designer can focus on the choice of the modules that make up the functionality of the implementation without worrying about synchronization and latency of the overall design.

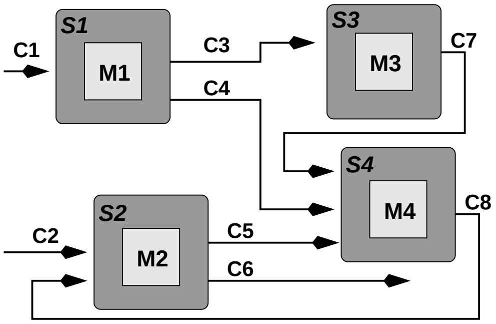

Figure 1. Shell Encapsulation and Communication Channels.

Communication design does not have any impact on the design and implementation of the modules provided that the modules and the relay stations share a fundamental property, *patience* (see Section 4). Requiring that an arbitrary module is patient at the onset is quite strong. This is the reason why we encapsulate the modules with an appropriate shell that has the task of making the module look patient. Such shells can be automatically generated for all modules if the output of the module is latched and each module is *stallable* [4]. *"Stallability"* means that a module can stall for an arbitrary amount of clock cycles without losing its internal state (and the overall state of the system) and is much weaker than patience [1].

3. Related Work

The adoption of DSM process technologies and the increasing impact of interconnect delay are destined to exacerbate the *timing-closure problem*: the designer is forced to iterate many times between synthesis and layout, because the two steps are performed independently and synthesis uses statistical delay models which badly estimate the post-layout wire load capacitance [7, 19].

In [10] Sylvester and Keutzer discuss the impact of DSM geometries on the future of design automation methodologies and envision that future integrated circuits will be implemented hierarchically with large macro-blocks of approximately $50K$ to $100K$ gates. They conclude that traditional standard-cell design flow will be still used for the design of such macro-blocks, because *"interconnect delay will be small* ($\leq 25\%$) *in block of* $50K$ *gates"*. These results are obtained from the analysis of detailed ASIC design data, such as average wirelengths and average net fan-out. However, one must observe that the timing closure problem arises when the delay of the critical path in the design is excessive, and, therefore, it is by nature a worst-case problem and not an average-case problem. Most of the solutions proposed in literature so far call for tighter interaction between synthesis and physical design. A synthesis-driven methodology that optimizes for interconnect delay rather than gate delay during logic synthesis is presented in [14]. Unfortunately, the approach produces a large amount of logic duplication, which may lead to expensive area overheads. Floorplanning, technology mapping and gate placement are combined in [27], where, after placement has been completed, the critical paths are reduced one at a time to meet the timing requirements. Since to fix one critical path may generate new ones, this approach is unable to solve *by construction* the convergence problem. A series of layout-driven approaches suggest to fix the layout by extracting accurate physical informations which are used to guide different types of logic optimization, such as gate-resizing [16], fanout optimization [18], buffer insertion [28], and logic resynthesis [22].

All these approaches represent remedies to the effects of bad estimations made during logic synthesis and do not seem able to scale well with the shrinking of process geometries. Following the old adage that *an ounce of prevention is worth a pound of cure*, we believe that the time for a radical paradigm shift is approaching.

3.1 Latency Insensitive vs. Asynchronous Design

The latency insensitive design methodology is clearly reminiscent of many ideas which have been proposed in the asynchronous design community during the past three decades [11]. In particular, the idea of a design methodology which is inherently modular is already present in the work on *Macromodular Computer Systems* by Clark and Molnar [5, 6]. To separate the design of these modules by the design of the system and make the entire process amenable to automation, the modules must be implemented as *delay-insensitive* circuits [24, 26]. A delay-insensitive circuit is designed to operate correctly regardless of the delays on its gates and wires (unbounded delay model) [32]. However, it has been proven that almost no useful delay-insensitive circuits can be built if one is restricted to a class of simple logic gates [2, 23]. To be able to build complex systems one must use more complex components, which are “externally” delay insensitive, while “internally” are designed by carefully verifying their timing and avoiding or tolerating metastability [13, 17, 26]. By slightly relaxing the unbounded delay model and allowing “isochronic forks” [2], practical *quasi-delay-insensitive* circuits can be built using simple logic gates [3]. A further relaxation leads to *speed independent* circuits, which operate correctly regardless of gate delays, while wire delays are assumed to be negligible [1, 12, 20]. Both quasi-delay-insensitive and speed-independent circuits assume that the designer is able to control wire delays, and, therefore, do not appear as interesting alternatives when moving to DSM implementations. Instead, a methodology based on assembling complex modules which are “externally” delay-insensitive seems the right solution, on condition that the synthesis of such modules is not too cumbersome. However, it must be noted that asynchronous approaches do not address the fundamental problem of latency, because an asynchronous design simply slows down to accommodate the slowest component, e.g. the wires.

While a delay insensitive system is based on the assumption that the delay between two subsequent events on a communication channel is completely arbitrary, in the case of a latency insensitive system this arbitrary delay is a *multiple of the clock period*. The key point is that this kind of *discretization* allows us to leverage well-accepted design methodologies for the design and validation of synchronous circuits. In fact, the basic distinction between any of the previous asynchronous design methodologies and the latency-insensitive one is essentially that a latency insensitive system is specified as a synchronous system. Notice that we say "specified" because, from an implementation point of view a latency-insensitive communication protocol can also be realized using *handshaking signaling* techniques (such as request/acknowledge protocols), which are typically asynchronous [3]. However, from a specification point of view, each module (as well as the overall system) is viewed as a synchronous system. Now, to specify a complex system as a collection of modules whose state is updated collectively in one "zero-time" step is naturally simpler than specifying the same system as the interaction of many components whose state is updated following an intricate set of interdependency relations. Furthermore, the synchronous specification allows us to slightly modify the traditional semi-custom design methodology, by simply inserting a step to encapsulate each synchronous module within a shell. Finally, the impact is very different also from a validation point of view because simulation is naturally a less complex task for a synchronous circuit than an equivalent asynchronous one. In conclusion, the proposed methodology can be implemented on top of the commonly-adopted standard-cell design flow, while all previous asynchronous approaches force the designer to use new tools and, more importantly, *to think the digital system in a completely different way*.

4. Latency Insensitive Protocols

The proposed design methodology is based on the theory of *latency insensitive protocols*, which has been recently presented in literature [4]. This theory can be summarized as follows. A latency insensitive protocol is a communication protocol governing the exchange of information in a patient system. According to the Tagged-Signal Model [21] a system is a composition of processes communicating by exchanging signals, i.e. sequences of events, on a set of channels. A behavior of a system is unambiguously described by the set of signals which are exchanged among its processes. A *patient* system is a synchronous system whose functionality only depends on the order of the events of each signals and not on their exact timing. More specifically, a patient system is a collection of patient processes communicating by means of "point-to-point" channels whose latency may be arbitrary. Normally, at every cycle t_k, a generic patient process P_i receives a new *informative event* on each of its input channels and it emits informative events, which are the result of its internal computation up to the previous cycle t_{k-1}, on its output channels. However, due to channel arbitrary latencies, it may happen that at cycle t_k a *stalling event* (denoting the absence of an informative event) arrives on one or more of its input channels. If this is the case, process P_i (being *patient*) waits an arbitrary but finite amount of extra cycles until all informative events (which were expected at t_k) have arrived on all input channels. During this wait, P_i emits stalling events. Any sequence of stalling cycles does not affect the internal state of P_i (the process is patient) as well as the overall state of the system (the protocol guarantees that all processes awaiting data from P_i receive instead a stalling event).

If all channels in the system have unit latency then no stalling events are exchanged among its processes. Let S_{ref} be a patient system with such a characteristic. Then, let S_{stall} be another patient system which is composed by exactly the same processes as S_{ref}, while having some channels with latency greater than one clock cycle. Now, assume to apply to the two systems the same external stimulus yielding two corresponding behaviors β_{ref} and β_{stall}. If all stalling events are filtered away from β_{stall}, the resulting behavior is exactly equal to β_{ref}. The two behaviors are said *latency equivalent*. Further, if every behavior of S_{ref} is latency equivalent to some behavior of S_{stall} (and *vice versa*) then the two processes are said to be latency equivalent. It has been proven that, for patient processes, latency equivalence is compositional [4].

A *relay station* is a patient process communicating with two channels c_i and c_o such that if s_i and s_o are the signals associated to the channels and $I(l,k,s_i), l \leq k$ denotes the sequence of informative events of s_i between the l-th clock cycle and the k-th one, then s_i and s_o are latency equivalent and for all k

$$I(1,(k-1),s_i) \; - \; I(1,k,s_o) \;\; \geq \;\; 0 \qquad (1)$$

$$I(1,k,s_i) \; - \; I(1,(k-1),s_o) \;\; \leq \;\; 2 \qquad (2)$$

The following is an example of relay station behavior, where τ denotes a stalling events and ι_i a generic informative event:

$$\begin{aligned} s_i &= \iota_1\,\iota_2\,\iota_3\,\tau\;\tau\;\iota_4\,\iota_5\,\iota_6\,\tau\;\tau\;\tau\;\iota_7\,\tau\;\iota_8\,\iota_9\,\iota_{10}\;\ldots \\ s_o &= \tau\;\iota_1\,\iota_2\,\iota_3\,\tau\;\tau\;\iota_4\,\tau\;\tau\;\tau\;\iota_5\,\iota_6\,\iota_7\,\tau\;\iota_8\,\iota_9\,\iota_{10}\;\ldots \end{aligned}$$

Notice, that no further specification has been given on the signals s_i and s_o, (for instance saying that s_i is the input and s_o is the output). The definition of relay station simply involves a set of relations, i.e. a protocol, between s_i and s_o without any implementation detail. Still, it is clear that each informative event received on channel c_i is later emitted on c_o, while the presence of a stalling event on c_o may induce a stalling event on c_i in a later cycle. In fact, an informative event takes at least one clock cycle to pass through a relay station (minimum forward latency = 1), at most two informative events can arrive on c_i while no informative events are emitted on c_o (internal storage capacity = 2), and, finally, one extra stalling event on c_o will “move” into c_i in at least one cycle (minimum backward latency = 1). The double storage capacity of a relay station permits, in the best case, to communicate with maximum throughput (equal to one): a practical confirmation of this fact is given in Section 5, where an RTL implementation of a relay station is discussed.

Since relay stations are patient processes, their insertion in a patient system guarantees that the system remains patient. Further, since they have minimum latencies equal to one, they can be repetitively inserted on a channel to increase its latency. Therefore, the methodology is patterned after the theory as follows: (1) we start giving an abstract specification of a digital system as collection of synchronous modules without making any assumption on the latency of the wires (which are grouped in channels), then (2) we automatically synthesize a corresponding layout, (3) we segment every wire whose latency is greater than the desired clock period by distributing on it the necessary amount of relay stations, and (4) we build the shell around the modules to obtain patient processes that interact with the appropriate relay stations. Obviously, the final result will be satisfactory only to the extent that a sufficient throughput can be maintained in the presence of increased latency of wires. However, this is a general problem that will have to be faced in the design of large chips with DSM technologies, and not specific to the latency insensitive methodology. On the other hand, the latency insensitive methodology allows an easy early exploration of latency/throughput tradeoffs as illustrated in Section 6.

5. The Implementation of the Protocol

In this Section, we present a latency insensitive communication architecture consisting of channels, relay stations, and shells built according to our methodology.

5.1 Channels

Channels are point-to-point unidirectional links between a *source* module and a *sink* module. Data are transmitted on a channel by means of *packets*: a packet consists of a variable number of fields. Here, we consider only two basic fields: *payload* contains the transmitted data and *void* is a one bit flag which, if set to 1, denotes that no data are present in the packet (*void packet*). If a packet does contain “meaningful” payload data (i.e., void is set to 0) we will call it a *true packet*. A channel is made of wires and relay stations. The number of relay stations in a channel is finite and represents the buffering capability of the channel.

At each clock cycle, the source module may either put a new true packet on the channel or, in case no output data are available to be sent, put a void packet on it; on the other side, at each clock cycle the sink module retrieves from the channel the incoming packet and, on the basis of the void field value, decides whether to discard it or to store it on its input channel queue for later use. As a source module might not be ready to send a true packet, so a sink module might not be ready to receive it, for instance because its input queue is full. However, the latency insensitive protocol demands a fully reliable communication among the modules, where no lossy communication link is allowed and all packets are properly delivered. Consequently, the sink module must have a way to interact with the channel (and ultimately with the corresponding source module) to stop momentarily the communication flow and avoid the loss of any packet. Therefore, we slightly relax our definition of a channel as unidirectional, to allow a bit of information, called the channel *stop flag*, moving in the opposite direction. By setting the stop flag equal to one during a certain clock cycle, the sink module informs the channel that the next packet can not be received and it must be held until the stop flag is reset. As the sink module also the channel has a limited amount of buffering resources: a channel dealing with a sink module that requires a long stall period may fill up all its relay stations and being forced to send a stop flag to the source module so that the latter will put its packet production on stall.

5.2 Relay Stations

Figure 2 illustrates a possible relay station implementation based on the following specification, which refines the abstract notion given in Section 4:

"At each clock cycle t it takes a packet $packetIn^{t+1}$ and a stop signal $stopIn^{t+1}$ as inputs and it emits a packet $packetOut^{t+1}$ and a stop signal $stopOut^{t+1}$ as outputs: $stopOut^{t+1}$ is always equal to $stopIn^{t}$, while, according to the value of the internal variable $stalling^{t} = stopIn^{t} \wedge stopIn^{t-1}$ the relay station decides whether to set $packetOut^{t+1}$ equal to $packetIn^{t}$ (case: $stalling^{t} = 0$) or to stall by keeping $packetOut^{t+1}$ equal to $packetOut^{t}$ and saving $packetIn^{t}$ value into an auxiliary register (case: $stalling^{t} = 1$)".

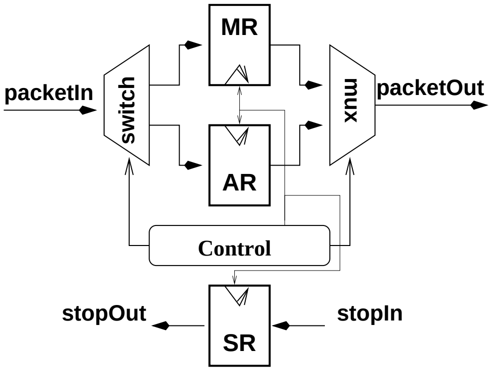

Figure 2. Relay Station Implementation

Figure 3 illustrates two modules, *Fetch Unit* and *Instruction Cache*, communicating using two channels *Address Channel* and *Data Channel*. Both channels have been partitioned in 4 segments by the insertion of 3 relay stations and, as

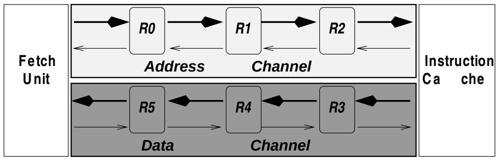

Figure 3. Channels between *Fetch Unit* and *Instruction Cache*

a consequence, the lower bound on the latency of each channel has become 4 clock cycles. Figure 4 reports a snapshot of the waveforms obtained by simulating a VERILOG RTL description of the *Address Channel*: here, the source module is the *Fetch Unit* producing a sequence of addresses for a *Memory Block* which represents the sink module. The addresses are reported as hexadecimal numbers.

Beside the system clock having period T_{CLK} equal to $10ns$, one can see 8 waveforms which, going from top to bottom, correspond respectively to the following signals of Figure 3: $R2.packetOut$, $R2.stopIn$, $R1.packetOut$, $R1.stopIn$, $R0.packetOut$, $R0.stopIn$, $FU.packetOut$, $FU.stopIn$.

At time $t = 75ns$ the sink module sets $R2.stopIn$ equal to one and keeps it equal to one for three clock cycles. As a consequence, $R2$ stalls two cycles as it maintains $R2.packetOut = h'44$ for the next three cycles while storing $R1.packetOut = h'45$ on a auxiliary set of registers. In the meantime, the stop signal is propagated to $R1.stopIn$. When, after three clock cycles, at time $t = 105ns$, the sink module can finally receive $R2.packetOut = h'44$, it resets $R2.stopIn$ such that at the following clock cycle $R2$ may set $R2.packetOut = h'45$. In the meantime, the three consecutive high values of the stop signal propagate back through the channel, provoking a stall of two cycles for each station while guaranteeing that no packets are lost. Notice that a characteristic of this implementation of the protocol is that when a $stopIn$ signal is kept high for only one cycle, the relay station does not really stall: in Figure 4 this can be observed for the sequence of clock cycles starting at $t = 165ns$. This fact is simply a positive bi-product of the fact that the storing capacity of a relay station is double [4].

5.3 Shells

As introduced in Section 2, given a particular module M, an instance of a shell can be automatically synthesized as a wrapper to encapsulate M and interface it with the channels so that M becomes a patient process. To do so the only necessary condition is that M be stallable.

At each clock cycle the module internal computation must be *fired* only if all inputs have arrived. Guaranteeing this *input synchronization* is the first task of the shell of a module. The second task is called *output propagation*: at each

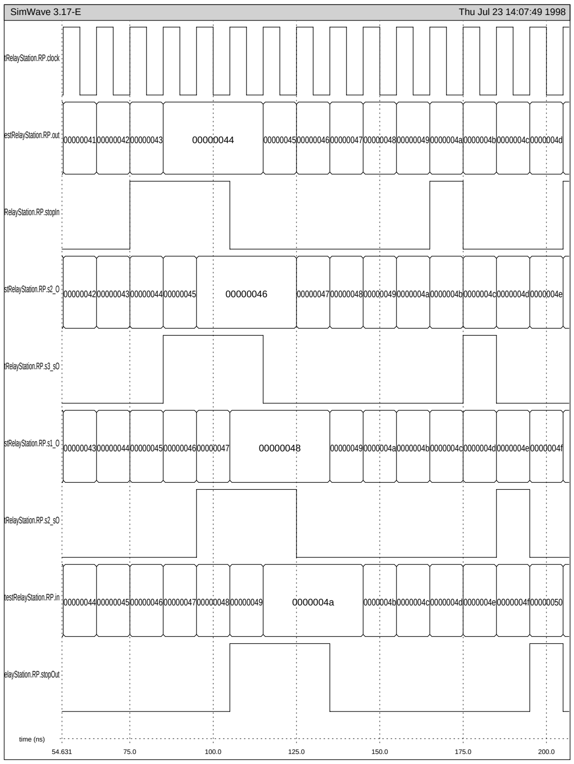

Figure 4. Waveforms on *Address Channel*

clock cycle, if module *M* has produced new output values and no output channel has previously raised a stop flag, then these output values can be transmitted generating new true packets; if any of these two conditions is not verified, then the packet transmitted in the previous cycle is re-transmitted as a void packet.

In summary a shell for module *M* performs the following actions cyclically:

1 it gets the incoming packets from the input channels, filters away the void packets and extracts the input values for *M* from the payload fields of the true packets;

2 when all input values are available for the next computation, it passes them to *M* and fires the computation;

3 it gets the results of the computation from *M*;

4 if no output channel has previously raised a stop flag, it routes the result into the output channels.

6. Case Study: The PDLX Microprocessor

To test our methodology, we performed a "latency insensitive design" of an out-of-order microprocessor (PDLX) with speculative execution. Its instruction set is the same of the DLX microprocessor, described in [15]. Its architecture is based on an extended version of the *Tomasulo's Algorithm* [31], which combines traditional dynamic scheduling with hardware-based speculative execution. The data-path of PDLX is similar to the one of some of the most advanced microprocessor available on the market today.

Figure 5 illustrates a simplified block diagram of the PDLX architecture: the *PC Unit* sends the current value of the *Program Counter (PC)* to the *Instruction Cache* and the *Fetch Unit*. After receiving the corresponding instruction, the *Fetch Unit* couples it with the PC value and sends it to the *Decode Unit*. Once instruction decoding is completed, the result arrives to the *Execution Unit* which performs the execution phase working with the *Data Cache* and the *Register File*. If the result of the execution is a "*branch taken*", then the branch target address is sent to the *PC Unit*.

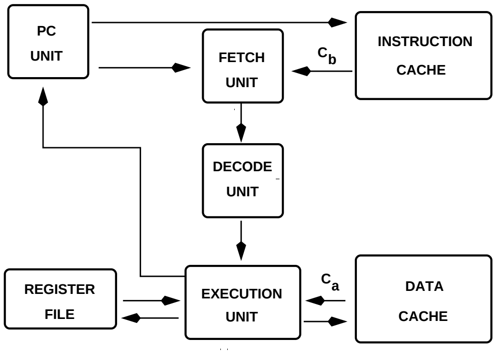

Figure 5. PDLX Microprocessor Block Diagram : top level view.

In our implementation, the 7 units correspond to 7 modules made patient by adding an appropriate shell. Obviously, this decomposition of the hardware implementing the PDLX, is not the only possible, let alone the best one. Still, while reasonably simple, it presents interesting challenges to the realization of the proposed latency insensitive communication architecture. In particular, the *Fetch Unit* shell merges two separate channels (likely they have different latencies), and each time a "*branch taken*" is executed a "feedback path" is activated between the *Execution Unit* and the *PC Unit*.

We performed a high-level cycle-accurate design of PDLX by using `BONeS DESIGNER` [29]. We first designed the PDLX modules illustrated in Figure 5, keeping in mind only the following informal rule to make the process stallable:

At each clock cycle the execution process of a module can always be frozen without affecting its internal state. Then, we designed the latency insensitive protocol library, containing as building blocks relay stations and shells. Finally, we encapsulated each module in a shell and we obtained the final system. To test our design, we took some simple numerical *C* programs (permutations, binary search,...) and we generated the corresponding DLX assembler code by using DLXCC, a publicly available DLX compiler [30]. Then, we loaded the assembler into the PDLX *Instruction Cache* and we executed it, while logging every read/write access to the *Data Cache*. Finally, we compared the "log file" with the one obtained executing the same assembler code on the DLX simulator DLXSIM to verify that the functional behavior was indeed the same.

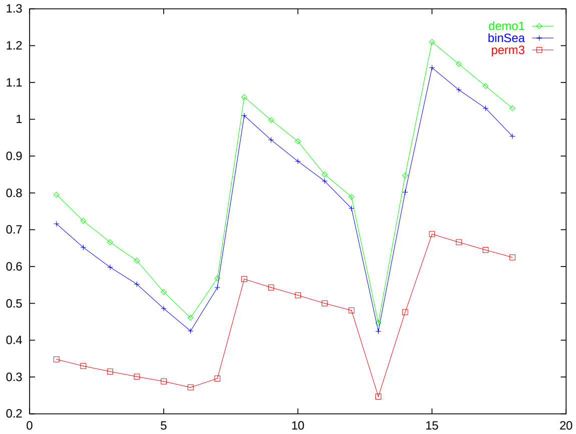

Figure 6. Effective Throughput vs. PDLX implementations

For each program execution, we computed the total number of clock cycles T necessary to complete the execution of the assembler code: this number is equal to $I+S+P$, where I is the number of instruction which have been committed, S is the number of cycles lost due to a stall within the execution unit, and P is the number of cycles lost due to pipeline latency. Since the PDLX is a single-issue multiprocessor, the instruction throughput $T = I/T$ is a quantity less than or equal to one. This quantity can be multiplied by the system clock frequency to obtain the *effective instruction throughput* $ET = (I/T) * f_{CLK}$, which allows us to compare the execution of the same assembler code on different PDLX implementations running at different speeds. Figure 6 illustrates the results obtained running three different assembler programs: the effective instruction throughput is reported on the *y*-axis, while each discrete point on the *x*-axis corresponds to a different PDLX implementation with a different fixed amount of latency on some channels. We focused on two specific channels on Figure 5: channel C_a

between the *Execution Unit* and the *Data Cache* and channel C_b between the *Fetch Unit* and the *Instruction Cache*. We assumed that the wires grouped in these two channels represent the critical path of the PDLX design and that, after segmenting them (by inserting relay stations), we could afford to raise the clock frequency appropriately. We varied the latency on the two channels as follows: going from left to right on the *x*-axis, the 18 data-points represent 18 implementation cases which can be grouped in three subsets in correspondence to latency values L_a for C_a equal respectively to $0, 1, 2$ clock cycles. Each of these subsets contains 6 data-points corresponding to latency values L_b for C_b going from 0 to 5 clock cycles. Finally, for each implementation case, we set the system clock frequency as $f_{CLK} = \min\{L_a, L_b\} + 1$. The plot illustrates how different PDLX implementations perform under the same data stimulus, showing that the throughput values are consistent across different benchmarks. All implementations are functionally equivalent by construction, being obtained simply by changing the number of relay stations on the channels and with no need of redesigning any PDLX module. The insertion of relay stations can be made at late stages in the design process, after detailed information can be extracted from the physical layout, to “fix” those channels whose latency is longer than the desired clock cycle.

7. Conclusions and Future Work

We proposed a new *“correct-by-construction”* synthesis methodology for designing very large digital systems by assembling IP functional modules. The modules communicate by exchanging data on communication channels according to an appropriate protocol, which guarantees a correct system behavior independently from channel latencies. As a consequence, a robust implementation is achieved in a shorter time by reducing the multiple iterations between logical and physical design. We developed a set of RTL libraries for a specific latency insensitive protocol and we used them to design a latency insensitive implementation of PDLX, an out-of-order microprocessor with speculative execution. There are several avenues for further investigations: (1) application to other designs, particularly in the multimedia domain, (2) study of the impact of our approach on other design metrics such as area and, especially, power, (3) extension of the theory to *speculation insensitive* protocols.

Acknowledgments

The authors would like to thank Luciano Lavagno and Patrick Scaglia for their support and useful discussions.

Notes

1. Observe that most hardware systems can be easily made stallable: for instance, consider any sequential logic block together with a *gated clock* mechanism, or a Moore finite state machine with an extra input that can force it to stay in the current state while emitting a "flag signal".

2. A bounded skew is allowed between the different branches of a net.

3. But the communication bandwidth would be limited by the inverse of the longest of the round trip times between pairs of communicating relay stations.

4. Recall that the primary reason for this double capacity is the need of avoiding losing data while spending one cycle to propagate the stop signal.

References

[1] P. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.

[2] Janusz A. Brzozowski and Jo C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992.

[3] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[4] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency Insensitive Protocols. In *Proc. of the 11th Intl. Conf. on Computer-Aided Verification (N. Halbwachs and D. Peled editors)*, pages 123–133. LNCS 1633, Springer, July 1999.

[5] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press.

[6] Wesley A. Clark and Charles E. Molnar. The promise of macromodular systems. In *Digest of Papers of the Six Annual IEEE Computer Society International Conference*, pages 309–312, San Francisco, CA, 1972. IEEE Press.

[7] J. Cong. Challenges and Opportunities for Design Innovations in Nanometer Technologies. In *SRC Design Sciences Concept Paper*, December 1997.

[8] J. Cong, L. He, K.Y. Khoo, C.K. Koh, and Z. Pan. Interconnect Design for Deep Submicron ICs. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 478–585. IEEE, November 1997.

[9] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 8(9):37–39, September 1997.

[10] D. Sylvester and K. Keutzer. Getting to the Bottom of Deep Submicron. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1998.

[11] Al Davis and Steven M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[12] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[13] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[14] W. Gosti, A. Narayan, R.K. Brayton, and A. Sangiovanni-Vincentelli. Wireplanning in Logic Synthesis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 26–33. IEEE, November 1998.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1996.

[16] S. Hojat and P. Villarrubia. An Integrated Placement and Synthesis Approach for Timing Closure of Power PC Microprocessors. In *Proc. Intl. Conf. on Computer Design. VLSI in Computers and Processors*, pages 206–210. IEEE, October 1997.

[17] M. B. Josephs and J. T. Udding. An overview of DI algebra. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, January 1993.

[18] L.N. Kannan, P.R. Suaris, and H. Fang. A Methodology and Algorithms for Post-Placement Delay Optimization. In *Proc. of the Design Automation Conf.*, pages 327–332, June 1994.

[19] H. Kapadia and M. Horowitz. Using Partitioning to Help Convergence in the Standard-Cell Design Automation Method. In *Proc. of the Design Automation Conf.*, pages 592–597, June 1999.

[20] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 56–62, June 1994.

[21] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design*, 17(12):1217–1229, December 1998.

[22] A. Lu, H. Eisenmann, G. Stenz G., and F.M. Johannes. Combining Technology Mapping with Post-Placement Resynthesis for Performance Optimization. In *Proc. Intl. Conf. on Computer Design. VLSI in Computers and Processors*, pages 616–621. IEEE, October 1998.

[23] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[24] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[25] R. H. J. M. Otten and R. K. Brayton. Planning for Performance. In *Proc. of the Design Automation Conf.*, pages 122–127, June 1998.

[26] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988.

[27] A. Salek, J. Lou, and M. Pedram. A DSM design flow: Putting Floorplanning, Technology Mapping and Gate Placemente Together. In *Proc. of the Design Automation Conf.*, pages 287–290, June 1998.

[28] K. Sato, M. Kawarabayashi, H. Emura, and N. Maeda. Post-Layout Optimization for Deep Submicron Design. In *Proc. of the Design Automation Conf.*, pages 740–745, June 1996.

[29] S.J. Schaffer and W.W. LaRue. BONeS DESIGNER: a Graphical Environment for Discrete-Event Modeling and Simulation. In *MASCOTS '94. Proc. of the 2nd. Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 371–374, Los Alamitos - CA, February 1994. IEEE.

[30] The DLX Software. *ftp://max.stanford.edu/pub/hennessy-patterson.software*.

[31] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal Research and Development*, 11:25–33, January 1967.

[32] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.