Synthesis of Synchronous Elastic Architectures

Jordi Cortadella[*]
Universitat Politècnica
de Catalunya
Barcelona, Spain

Mike Kishinevsky
Strategic CAD Lab, Intel Corp.
Hillsboro, OR, USA

Bill Grundmann
Strategic CAD Lab, Intel Corp.
Hillsboro, OR, USA

ABSTRACT

A simple protocol for latency-insensitive design is presented. The main features of the protocol are the efficient implementation of elastic communication channels and the automatable design methodology. With this approach, fine-granularity elasticity can be introduced at the level of functional units (e.g. ALUs, memories). A formal specification of the protocol is defined and an efficient scheme for the implementation of elasticity that involves no datapath overhead is presented. The opportunities this protocol opens for microarchitectural design are discussed.

Categories and Subject Descriptors: B.5.2 [Register-transfer-level implementation]: Design Aids.

General Terms: Design, Theory, Verification.

Keywords: Latency-insensitive design, latency-tolerance, protocols, synthesis.

1. MOTIVATION

In current nanotechnologies, calculating the number of cycles required for transmitting an event from a sender to a receiver is a problem that often cannot be solved until the final layout has been generated.

The main motivation of this work is illustrated by Fig. 1, depicting a system with four functional units, each one delivering the result to a register (shadowed boxes). Fig. 1(a) represents a cycle accurate specification of the system (at the RTL level or level higher than RTL) in which every functional block has a 1-cycle delay.

Imagine that for scalability reasons, when migrating to a new technology, the long wires $C \rightarrow A$ and $D \rightarrow B$ must be pipelined and the unit C must be substituted by another one (C') with variable delay (e.g. 1 cycle for short operands and 2 cycles for long operands). Similar transformations can be applied due to a different motivation - an exploration of new micro-architectures, e.g. the variable delay unit can be used either for cycle time optimization or for power optimization. The resulting microarchitecture is depicted in Fig. 1(b). This type of migrations in the current design practice requires drastic *manual* changes in the control units and the data path to accommodate to the new delays of the system.

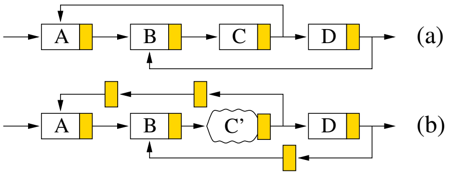


Figure 1: Elasticization of a synchronous system.

In this paper, we propose a simple control scheme to transform conventional synchronous systems into elastic, thus tolerating in a natural way any variability in the computation and communication delays. The contributions of this paper are inspired on the work by other authors in the area of *latency-insenstive* (LI) design [3] and *synchronous interlocked pipelines* [11]. They are also inspired on *de-synchronization* [2, 9, 17], an approach to transform synchronous specifications into asynchronous implementations.

1.1 Contributions

The main contribution of this work is the proposal of a simple communication protocol that makes the system totally elastic and that can be applied to any level of granularity without any overhead in the datapath of the system.

Elastic systems show notorious benefits: they can be obtained *automatically* from conventional high-level or RTL specifications and guaranteed *correct-by-construction*, they are tolerant to variations of computational and communication delays and their modularity enables a wide exploration of different architectures trading-off power and delay.

With regard to previous work on latency-insensitve design, the contributions of this work are as follows:

(1) **An abstract model of the protocol** is defined. The model is suitable for formal verification and for refinement with the goal of exploring different implementations of elastic channels and buffers. The conceptual implementation of LI systems proposed in [4] and interlock pipelines in [11] are particular solutions in this design space. To be more precise, the protocol sketched in [3, 4] although at the first glance may appear identical to ours is, in fact, significantly more complex: while a state of our protocol is determined by the *current* control values on the channel, their protocol requires state information to remember the *previous* value of the backpressure wire (stop). This (and other complicated

[*]This work has been partially supported by a grant from Intel Corp., CICYT TIN2004-07925 and a Distinction for Research by the Generalitat de Catalunya.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

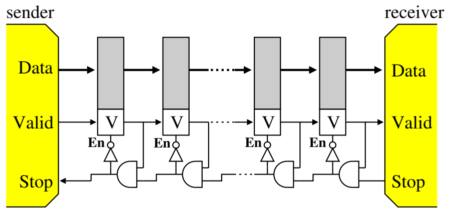


Figure 2: Unscalable elastic data transmission.

implementation choices) led to a complex control machine, an overhead of 2-3x in the sequential elements of the datapath compared to an ordinary flip-flop, and an extra cycle of latency at every point where two or more information channels merge at a single block or fork out of the block.

(2) **An efficient latch-based implementation** with no datapath overhead and clock-gating for all latches that guarantees minimum activity. With regard to [11], complete automation from RTL specifications is proposed and new fork/join structures without combinational cycles are presented. While [11] considers pipeline stages only, we show that any level of granularity (from single gates, to large blocks) can be selected for converting to an elastic form.

(3) The proposed scheme **can be applied to different levels of granularity**, i.e. in white-box (e.g. microprocessor design) and black-box scenarios (SoC IPs). Contrary to [3], systems can be made elastic **without any delay overhead**, i.e. preserving the same sequential latency as the original synchronous design.

2. A PROTOCOL FOR ELASTIC SYSTEMS

An elastic system is a collection of elastic modules and elastic channels. Elastic channels have two control wires implementing a handshake between the sender and the receiver. The wires are called *valid*, in the forward direction, and *stop*, in the backward direction. The role of these wires is similar to the one of the *request/acknowledge* wires in asynchronous systems. Depending on the state of the control wires, a channel can carry valid or invalid data items, that we will call *tokens* and *bubbles*, respectively.

For simplicity in the explanation, we will initially assume that the elastic modules are combinational blocks and latches. In section 4.2 we will show the generalization to modules with fixed and variable sequential latencies.

In low granularity elastic designs, all flip-flops are replaced by *Elastic Buffers* (EB). An EB can be implemented as a pair of *Elastic Half-Buffers*, (EHB), in the same fashion as flip-flops can be implemented as a pair of two transparent latches with opposite polarity (master and slave).Thus, the designer of an elastic system has the choice between using edge-triggered or transparent latches.

Fig. 2 depicts an example of a naive elastic implementation for transmitting data between two units. Each register has an associated *valid* bit (V) that keeps track of the validity of the stored data. The clock signal is not explicitly shown and the enable signal (En) indicates when new data is stored into the register. The chain of AND gates manages the *back-pressure* generated by the receiver when it is not able to accept data ($Stop = 1$). The scheme in Fig 2 is not scalable due to the long combinational path from the receiver to the sender. When the pipeline is full, i.e. all V's are at 1, the delay of the *Stop* chain becomes critical.

2.1 The SELF protocol

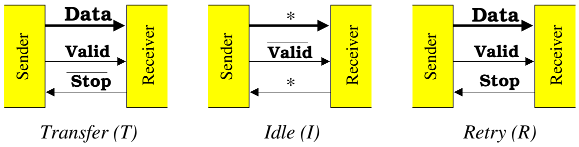


Figure 3: The SELF protocol.

Cycle	0	1	2	3	4	5	6	7	8	9
Data	$*$	A	B	B	B	C	$*$	$*$	D	D
Valid	0	1	1	1	1	1	0	0	1	1
Stop	0	0	1	1	0	0	0	1	1	0
SELF	I	T	R	R	T	T	I	I	R	T
LID	τ	T	**T**	τ	T	T	τ	τ	τ	T

Table 1: A trace committing the SELF protocol.

This section describes the protocol SELF (*Synchronous ELastic Flow*), suitable for scalable fine-grain elastic communication.

Data transfer is performed by using the control signals *Valid* (V) and *Stop* (S) that determine three possible states in the channel (see Fig. 3):

(T) Transfer, ($V \wedge \neg S$): the sender provides valid data and the receiver accepts it.

(I) Idle, ($\neg V$): the sender does not provide valid data.

(R) Retry, ($V \wedge S$): the sender provides valid data but the receiver does not accept it.

The sender has a *persistent* behavior when a *Retry* cycle is produced: it maintains the valid data until the receiver is able to read it. The language observed at a SELF channel can be described by the following regular expression: $(I^*R^*T)^*$

The absence of a subtrace RI implies the persistency of the behavior. Table 1 shows an example of a trace committing the SELF protocol and transmitting values $ABCD$. When $V = 0$, the value at the data bus is irrelevant (cycles 0, 6 and 7). The receiver can issue a *Stop* even when the sender does not send valid data (cycle 7). In *Retry* the sender persistently maintains the same valid data as in the previous cycle during cycle 3, 4 and 9 [1].

2.2 Specification of elastic buffers

Figure 4 shows the interface of an elastic buffer (EB) with one input and one output channels. The extension to multi-input/output channels will be discussed in Section 4.1.

There can be different architectures to implement an EB trading-off area, delay and power consumption.

The abstract model for an EB is described in Fig. 5. Briefly, an EB is modeled as an unbounded FIFO (possibly

[1] [4], gives an incomplete description of an implementation protocol. Assuming our recovery of that protocol is accurate, its transfer condition is more complex than that of SELF and requires more complex implementation with additional sequential buffers. Line LID of Table 1 shows that the same sequence of values on the channel wires in [4] signifies transfer of a different sequence of data: $A\mathbf{B}BCD$, because a token is transferred on the LID channel when $V \wedge \neg(S \wedge S^-)$, where S^- stands for the value of S at the previous cycle. Stop pulses of length 1 are ignored here. B sent through the channel during cycle 2 is assumed to be successfully transmitted to the receiver.

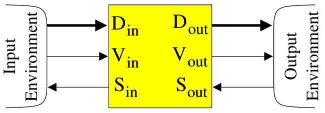
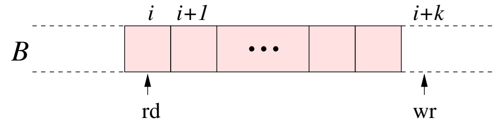


Figure 4: Interface of an EB with one input and one output channels.



State variables: B : *array*$[0\ldots\infty]$ of *data*; $\mathsf{rd},\mathsf{wr}:\mathbb{N}$; $\mathsf{retry}:\mathbb{B}$
Initial state: $\mathsf{wr}\geq\mathsf{rd}=0$; $\mathsf{retry}=\mathit{false}$;
Invariant: $\mathsf{wr}\geq\mathsf{rd}$

Combinational behavior:

$$V_{out}=\begin{cases} \mathit{true} & \text{if } \mathsf{retry}\\ \mathit{false} & \text{if } \mathsf{rd}=\mathsf{wr}\\ * & \text{otherwise}\end{cases}\qquad D_{out}=\begin{cases}\mathsf{B}[\mathsf{rd}] & \text{if } V_{out}\\ * & \text{otherwise}\end{cases}\qquad S_{in}=*$$

Sequential behavior:

$$\mathsf{rd}^{+}=\begin{cases}\mathsf{rd}+1 & \text{if } V_{out}\wedge\neg S_{out}\\ \mathsf{rd} & \text{otherwise}\end{cases}\qquad \mathsf{retry}^{+}=V_{out}\ \wedge\ S_{out}$$

$$\mathsf{wr}^{+}=\begin{cases}\mathsf{wr}+1 & \text{if } V_{in}\wedge\neg S_{in}\\ \mathsf{wr} & \text{otherwise}\end{cases}\qquad \mathsf{B}^{+}[\mathsf{wr}]=D_{in}$$

Liveness properties (finite unbounded latencies):

Forward latency: $\mathsf{G}(\mathsf{rd}\neq\mathsf{wr}\implies\mathsf{F}\ V_{out})$
Backward latency: $\mathsf{G}(\neg S_{out}\implies\mathsf{F}\ \neg S_{in})$

Figure 5: Abstract model for an EB.

with the initial content) that commits the SELF protocol at the input and output channels. The notation X^{+} is used to represent the *next-state value* of variable X. The symbol '$*$' represents a non-deterministic value (don't care).

B is an infinite array that stores the items written into the buffer, but not sent to the output yet. The variables wr and rd are the write and read indices, respectively. The value $k=\mathsf{wr}-\mathsf{rd}$ is the current number of items in the buffer.

The retry variable remembers whether a transfer was attempted in the previous cycle. If retry is true, the same data item as in the previous cycle is issued and $V_{out}=\mathit{true}$. If the buffer does not contain any item ($\mathsf{rd}=\mathsf{wr}$), no transfer can be performed ($V_{out}=\mathit{false}$). Finally, the value $V_{out}=*$ represents the non-deterministic behavior of a buffer with finite, but unbounded delay: the items stored in the buffer will eventually be transferred to the output after a finite unknown delay. S_{in} can non-deterministically stop any data transfer at the input channel. The behavior of the indices of the array is modeled by the rd^{+} and wr^{+} equations. When a data transfer is produced at the corresponding channel ($V\ \wedge\ \neg S$), the value of the index is incremented.

Two liveness properties expressed in *linear temporal logic* [15] ensure finite response time: (1) data from the EB will eventually be sent to the output, and (2) a non-stop at the output will eventually be propagated to the input.

2.3 Verification of EBs

The model in Fig. 5 has been used as specification to check that every implementation presented in this paper is a refinement of this model. The proofs have been performed by using the features of refinement verification and data type reductions offered by SMV [13].

We have also verified that the sequential composition of two EBs is a refinement of an EB. This guarantees that EBs with specific depths can be built by composing implementations of EBs that refine the model presented in this paper.

3. IMPLEMENTATION OF EBs

The implementation of an EB can be decomposed into two parts: data-path and control. The former deals with the data (D_{in} and D_{out}) whereas the latter manages the valid/stop handshakes and generates the enabling signals for the latches in the datapath.

Two important parameters of an EB are the *forward*, L_f, and backward, L_b, latencies. The L_f is the latency of forward propagation of the data and the Valid bit when the receiver is ready. The L_b is the backward latency for the stop signals. According to the unbounded specification of Fig. 5, L_f and L_b can be any value greater than zero (including non-deterministic values). $L_f=0$ and $L_b=0$ would reduce the EB to a channel. Having one of the latencies equal to 0 is possible in parts of the design, but does not scale, due to the long combinational delays and combinational cycles. However, for performance optimization and distributing the EBs across long communication channels and reusing them as sequential wire repeaters, the following constraint should be satisfied [3]: $L_f=L_b=1$. When L_f and L_b are not zero, combinational paths that propagate the valid/stop signals can be restricted to neighboring stages. This avoids situations like the one shown in Fig. 2, with $L_b=0$ and a long combinational path for the stop signal.

The capacity of an EB defines the maximal number of data items that can be simultaneously stored inside the buffer. The following property holds:

Property 3.1. *The capacity of an* EB*, C, must satisfy the following constraint: $C\geq L_f+L_b$*

Therefore, for the case of interest ($L_f=L_b=1$) the minimal possible capacity is $C=2$. Hence the data path of an EB can be constructed with two storage cells and different write/read policies, e.g. the different number of read and write ports. Carloni in [4] uses a structure with two read and write ports to implement his Relay Station that leads to high area and delay overhead and complex control protocol. Chelcea and Nowick [7] introduced an elastic FIFO that can be used for communication between elastic modules.

We have derived different implementations of an EB based on the number of ports and different control policies. In this paper we focus on the most efficient structure in which writes always occur to the first cell, and reads always done from the second cell. We proved that this structure cannot be implemented using single-edge flip-flops controlled by the same frequency clock as used by the environment. Intuitively, this is because such a structure would have a latency of two between the write operation through the input channel and the read operation from the output channel. A mechanism for double-pumping in one cycle is required. This can be implemented by using two transparent latches of different polarity, similar to a master-slave structure, but with the *independent* enable signals for the two latches[2]. Alternative (more expensive) structures, that are based on single-edge flops, can be used for high-level and performance modeling, formal verification and FPGA mapping of elastic designs.

Fig. 6 depicts the FSM specifications for the control of a latch-based EB and the overall structure of the design. The

[2] Similar idea was used in different domains [2,11,17]

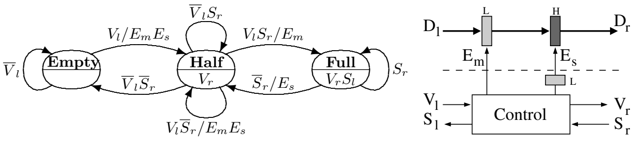


Figure 6: Specification of the latch-based EB.

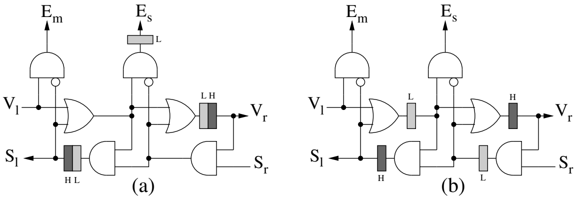


Figure 7: Two implementations of an EB control.

transparent latches are shown with single boxes, labelled with the phase of the clock, L (active low) or H (active high). The control drives latches with enable signals. To simplify the drawing the clock lines are not shown. The enable signals must be AND-ed with the corresponding clock phase for a proper operation. An enable signal for transparent latches must be emitted on the opposite phase and be stable during the active phase of the latch. Thus, the E_s signal for the slave latch is emitted on the L phase.

The FSM specification of a Fig. 6 is similar to the specification of a 2-slot FIFO: in the *Empty* state no valid data is captured in the data-path, in the *Half-full* state, an output slave latch keeps valid data, in the *Full* state - both latches keep valid data and the EB requests the sender to stop. With proper encoding of this specification one can derive an implementation of the control shown in Fig. 7(a) with flip-flops drawn as two back-to-back transparent latches. Splitting flip-flops and retiming leads to a fully symmetric implementation shown in Fig. 7(b).

4. ADVANCED STRUCTURES

4.1 Join and fork

In general, EBs can have multiple input/output channels. This can be supported by using elastic *Fork* and *Join* control structures. Figure 8(a) shows an implementation of a Join. The output valid signal is only asserted when both inputs are valid. Otherwise, the incoming valid inputs are stopped. This construction allows to compose multiple Joins together in a tree-like structure.

Figure 8(b) depicts a *lazy fork*. The controller waits for both receivers to be ready ($S = 0$) before sending the data[3]. A more efficient structure shown in Fig. 8(c), the *eager fork*, can send data to each receiver independently as soon as it is ready to accept it. The two flip-flops are required to "remember" which output channels already received the data. This structure offers performance advantages when the two output channels have different backpressure.

It is important to realize that the connection of a *lazy fork* with a *join* creates a combinational cycle in the control. This situation may arise when an arbitrary netlist of combinational blocks without latches in between is elasticized. To avoid this phenomenon, the eager fork can be used instead. The reader can verify that the connection with the eager fork does not produce combinational cycles.

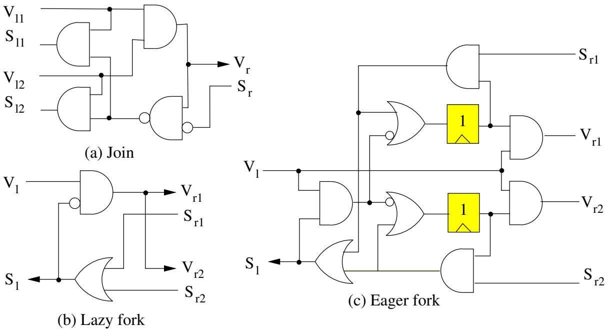


Figure 8: Controllers for elastic Join and Forks.

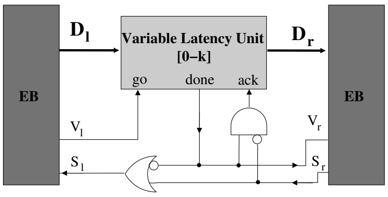


Figure 9: Control for variable-latency units.

4.2 Variable-latency units

Variable-latency units can be handled in a natural way with the SELF protocol by using the control structure shown in Fig. 9. A handshake with the datapath unit is required to keep track of the completeness of the operation. This can be done by means of three signals: *go* (start the operation), *done* (operation completed) and *ack* (the receiver has accepted the data and a new operation can start). This is a typical handshake for variable-latency units such as telescopic units [1].

Since the bubble insertion preserves correctness of the behavior it is also possible to convert some units of the system from fixed to variable latency. For instance, one could replace a 1-cycle ALU by a variable-latency ALU optimized for the typical data case (e.g. short carry propagation). This ALU calculating with latency 1 for the typical data mix, and with latency 2 for the rare data mix, can lead to performance improvement by designing the whole pipeline for a faster clock cycle, and to area reduction by reducing the number of logic gates per pipeline stage.

Under the control of some supervisor algorithm, variable latency can also be used for temporal and gradual shutdown/wake-up of computation units, trading-off power vs. performance at different levels of granularity.

5. SYNTHESIS FLOW

The elasticization of a synchronous netlist can be totally automated. We next present the steps to synthesize an elastic netlist from a conventional synchronous netlist:

(1) The flip-flop-based registers in the data-path are transformed into latch-based registers (master and slave) with independent enable signals. This step is not required if the netlist is already designed with latches.

(2) A control layer that generates the enable signals for the latches of the datapath is built as follows:

- For every flip-flop (latch) in the original datapath, an EB (EHB) controller is included in the control layer. The implementation of the EB is shown in Fig. 7.
- For every block, a controller with a *join* (J) at the

[3] This implementation is identical to the one in [11].

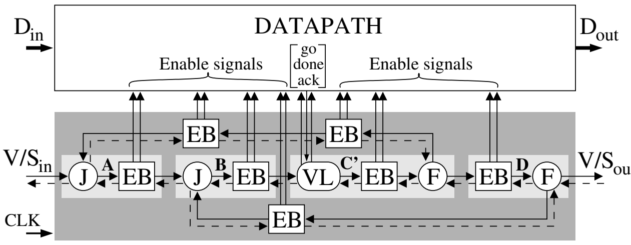


Figure 10: Control layer for the netlist in Fig. 1(b).

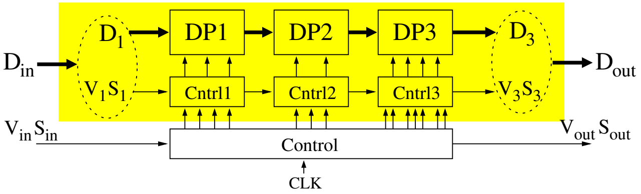


Figure 11: Hierarchical elasticization.

inputs and a *fork* (F) at the outputs is built. The join or fork components can be omitted if the block has only one input or one output, respectively. The implementations of J and F are shown in Fig. 8.

- A variable latency controller (VL) must be inserted for those components having variable latency (see Fig. 9).
- The connection of the controllers with the *valid/stop* interfaces is done according to the connectivity of the corresponding blocks in the datapath.

(3) Micro-architectural exploration can be performed by inserting/removing elastic buffers and using variable-latency units. Thus, design points with different cycle times, area, performance and power consumption can be explored.

An example of the resulting control layer for the netlist in Fig. 1(b) is depicted in Fig. 10. The pairs of solid/dotted wires represent the *valid/stop* bits of the channels. The signals *go/done/ack* in the datapath implement the handshake protocol for the variable latency unit C' using the controller shown in Fig. 9. The pairs of signals going from the control layer to the data-path are the *enable* signals for the master/slaves latches.

5.1 Hierarchical elasticization

Elasticization can be applied to any level of granularity by considering that each arbitrary block in a netlist is an FSM with variable delay. This can be done in a straighforward manner and the details are omitted here for the lack of space.

Elasticization can also be applied hierarchically at multiple levels. An elastic circuit can be considered as a conventional synchronous circuit in which the control layer and the datapath can be considered to be embedded in a higher-level datapath. This is illustrated in Fig. 11, in which several elasticized components (DP1, DP2 and DP3) are merged together with their controllers. The V and S signals of the controllers are now considered to be 1-bit data items in a datapath. This datapath is further elasticized by adding the bottom control layer. In this way, elasticization between different IP components of a system can be performed.

5.2 Correctness by construction

In Sect. 2, a formal model for the protocol and the EBs has been presented. This model is an abstraction that can be used for compositional verification of complex systems to guarantee correct system behavior. In particular, Fig. 1(b) illustrated the insertion of empty EBs in a synchronous netlist. The following property that guarantees soundness of this transformation holds:

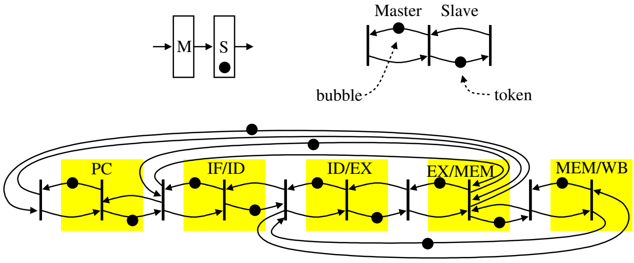


Figure 12: A concurrent model for elastic designs.

PROPERTY 5.1. *The insertion of empty EBs in an elastic design preserves* flow equivalence.

Informally, *flow equivalence* [8] between two designs guarantees that for every output the order of *valid* data items is the same[4]. An example on what flow equivalence means is illustrated below.

Synchronous behavior:	$a_1\ a_2\ a_3\ a_4\ \cdots\ a_i\ \cdots$ $b_1\ b_2\ b_3\ b_4\ \cdots\ b_i\ \cdots$
Elastic behavior:	$a_1\ *\ a_2\ *\ *\ *\ a_3\ *\ a_4\ \cdots\ a_i\ \cdots$ $b_1\ *\ *\ b_2\ *\ b_3\ b_4\ \cdots\ b_i\ \cdots$

The synchronous behavior shows the trace of values observed at two registers, a and b, at every cycle. After making the design elastic, some don't care values marked as invalid may appear (denoted as $*$). However, the order of valid data is preserved. It is important that the values at different registers may be shifted in time with respect to the pure synchronous behavior without affecting the functional correctness of the system, since any observer of both values would synchronize the corresponding valid tokens.

While bubble insertion (empty EBs) is correct-by-construction, token insertion (non-empty EBs) requires care and partial re-design similar to the standard pipelining.

5.3 Performance evaluation

Performance evaluation is another important aspect in system design. The behavior of elastic systems can be modeled by using a subclass of Petri nets called *marked graphs* [14], often used to model asynchronous systems. As an example, Fig. 12 depicts the marked graph model corresponding to the DLX abstraction shown in Fig. 14(a). Each latch is modeled as a pair of complementary arcs in which the location of the token indicates whether the latch contains valid (token) or invalid (bubble) data. The transitions of the marked graph represent the computations between latches (a simple data transfer if the transition is between a master and slave latch). Modeling systems with marked graphs enables the use of an extensive set of tools for the analytical performance analysis that can be effectively used at the earliest stages of the design. The authors have implemented computation of the effective cycle time for the SELF systems based on the separation analysis method from [6].

[4] Other authors call this property *latency equivalence* [16].

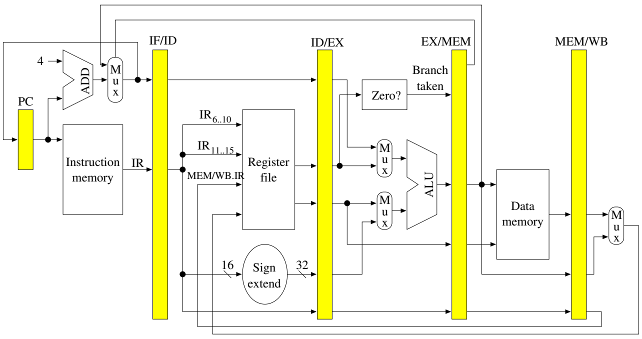


Figure 13: The DLX pipeline.

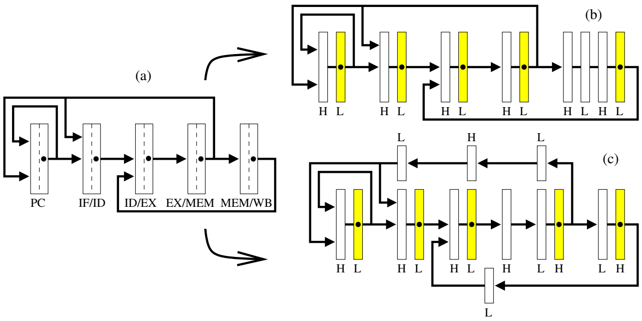


Figure 14: The control layer for the DLX pipelines.

6. A DESIGN EXAMPLE: THE DLX

We use the DLX pipeline [10] to illustrate how elasticity can be used in microarchitectural design and to briefly review the design flow (Figure 13). We have deliberately chosen roughly the same example as used by [3] to better illustrate the efficiency of our approach. It is assumed, for simplicity, that delay slots are used for handling data hazards and no forwarding is used. The shadowed boxes represent the registers that store the state of the microprocessor. From the control point of view, the register file and the memories can be considered as combinational units.

Figure 14(a) shows an abstraction of the pipeline in which only the channels among units are shown. The boxes represent the registers composed of two latches. The diagram is equivalent for the datapath and control layers. The join and fork controllers must be used when one unit has more than one input or output channel, respectively.

Figures 14(b) and 14(c) show a latch-based diagram of the same pipeline after the insertion of "bubbles". The shadowed latches are the ones initialized with valid data. This example illustrates that the insertion of bubbles does not affect the functionality of the system. Thus, the elastic architecture is *correct-by-construction* with respect to inserting empty EBs. This feature is even more interesting when the bubbles are inserted dynamically - another option in the micro-architectural optimization.

The criteria for bubble insertion can be different: break long wires, cycle time reduction, power reduction, etc. Some of these criteria have been studied in [5, 12].

7. CONCLUSIONS

A novel scheme for latency-insensitive design has been presented. It combines the modularity of asynchronous design with the efficiency of synchronous implementations.

The little overhead introduced by the implementation of the elastic buffers makes this scheme attractive for different levels of granularity and for exploring novel micro-architectural solutions focusing on the *typical* instead of the *worst* case. Additionally, the correct-by-construction paradigm of this method enables its applicability at the latest stages of the design, when accurate delay estimations of data transfers have been performed, without any impact on the functionality of the system.

8. REFERENCES

[1] L. Benini, G. D. Micheli, A. Lioy, E. Macii, G. Odasso, and M. Poncino. Automatic synthesis of large telescopic units based on near-minimum timed supersetting. *IEEE Transactions on Computers*, 48(8):769–779, 1999.

[2] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 149–158. IEEE Computer Society Press, Apr. 2004.

[3] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design*, 20(9):1059–1076, Sept. 2001.

[4] L. Carloni and A. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro, Special Issue on Systems on Chip*, 22(5):12, October 2002.

[5] L. Carloni and A. Sangiovanni-Vincentelli. Combining retiming and recycling to optimize the performance of synchronous circuits. In *16th Symp. on Integrated Circuits and System Design (SBCCI)*, pages 47–52, Sept. 2003.

[6] S. Chakraborty. *Polynomial-Time Techniques for Approximate Timing Analysis of Asynchronous Systems*. PhD thesis, Stanford University, Aug. 1998.

[7] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(8):857–873, 2004.

[8] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–304, Apr. 2003.

[9] G. Hazari, M. Desai, A. Gupta, and S. Chakraborty. A novel technique towards eliminating the global clock in VLSI circuits. In *Int. Conf. on VLSI Design*, pages 565–570, Jan. 2004.

[10] J. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[11] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, Apr. 2002.

[12] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 227–231, Nov. 2003.

[13] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.

[14] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, Apr. 1989.

[15] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[16] S. Suhaib, D. Berner, D. Mathaikutty, J.-P. Talpin, and S. Shukla. Presentation and formal verification of a family of protocols for latency insensitive design. Technical Report 2005-02, FERMAT, Virginia Tech, 2005.

[17] V. Varshavsky and V. Marakhovsky. GALA (globally asynchronous - locally arbitrary) design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 61–107. Springer-Verlag, 2002.