Synchronous Elasticization: Considerations For Correct Implementation and MiniMIPS Case Study

Eliyah Kilada, Shomit Das, Kenneth Stevens
University of Utah
{Eliyah.Kilada, Shomit.Das}@utah.edu, kstevens@ece.utah.edu

***Abstract*—Latency insensitivity is a promising design paradigm in the nanometer era since it has potential benefits of increased modularity and robustness to variations. Synchronous elasticization is one approach (among others) of transforming an ordinary clocked circuit into a latency insensitive design. This paper presents practical considerations of elasticizing reconvergent fanouts. It also investigates the suitability of previously published as well as new join and fork implementations for usage in the elastic control network. We demonstrate that elasticization comes at a cost. Measurements of a MiniMIPS processor fabricated in a 0.5 μm node show that elasticization results in area and dynamic and idle power penalties of 29%, 13% and 58.3%, respectively, without any loss in performance. These measurements do not exploit the capability of pipeline bubbles that occur if one needs to have unpredictable interface latency, or to insert extra bubbles into a pipeline due to wire delays. We finally show the architectural performance advantage of eager over lazy protocols in the presence of bubbles in the MiniMIPS.**

***Index Terms*—SOC, Synchronous Elasticity, Latency Insensitive Design, MiniMIPS.**

I. Introduction

Any system on chip (SOC) consists of a set of modules that not only perform individual functions, but also need to communicate with other modules. This communication may be in the form of either data or control signals. Sometimes, the communicating modules may be quite distant on the chip. The conventional synchronous circuit reads inputs and writes outputs at every clock cycle. The basic assumption is that computation and data transfer must be completed in exactly one clock cycle [1]. This places a very rigid timing constraint on the system, especially in the case where very long interconnects are involved. It is well known that metal wires do not scale as well as semiconductor gates. In fact, the resistance capacitance (RC) delay for metal wires per unit length is degrading with every new process generation [2], [3], [4]. This, in turn, increases the percentage of the clock cycle consumed in interconnect delays. Interconnect latency is affected by its dimensions as well as metal layers used, crosstalk (mainly determined by interconnect position in the chip), and power supply drop variations, among other layout specific factors [4]. This means that accurately evaluating the actual latency between modules is an issue that can only be resolved late in the design process. Moreover, over compensating for the delay due to wire effects can lead to sub-optimal designs [4].

Latency insensitive (LI) design tackles this issue by separating the computation and communication paradigms by creating a flexible protocol for communication between modules [5]. LI designs are able to tolerate interconnect latency variations without affecting the system functionality. This facilitates IP reuse and also ensures that system functionality depends only on modular correctness and not on the timing of the communication channels.

Synchronous elasticization is one approach (among others) of transforming an ordinary clocked circuit into an LI design [6], [7], [8]. This paper focuses on a clocked elastic protocol called the Synchronous Elastic Flow (SELF) protocol [6]. Synchronous elastic circuits are comparable to asynchronous designs in that they are robust to arbitrary channel latencies. Besides, they can be easily designed with conventional design flows using static timing analysis [7], [1].

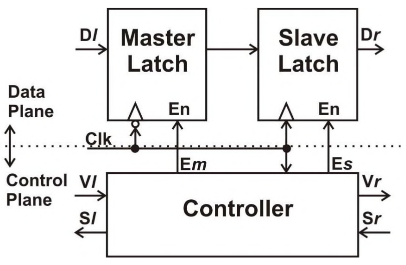


Fig. 1. An EB implementation.

A. Contribution

Practical considerations of elasticizing reconvergent fanouts are introduced. The suitability of different published, as well as new, join and fork implementations for usage in the elastic control networks are investigated. The cost of elasticizing a small microprocessor design is measured from fabricated chips, comparing area and power penalties. The performance of the elasticized MiniMIPS microprocessor is demonstrated when bubbles are inserted into the pipeline for different lazy and eager fork and join protocols.

II. Synchronous Elastic Architectures

An elastic system comprises elastic modules and elastic channels. Elastic modules are implemented using Elastic Buffers (EB) as counterparts to the flip-flops in conventional clocked systems. There are two parts to an EB implementation (Fig. 1), namely, data and control planes. EBs can be broken down into two Elastic Half Buffers (EHB) the same way a flip-flop can be replaced by a pair of master-slave latches [8].

An elastic channel uses a pair of control wires, 'valid' in the forward direction and 'stall' in the backward direction. These wires implement handshaking between the source and destination entities similar to the request/acknowledge wires in asynchronous systems [9]. Valid is asserted by the source when it holds valid data. Stall is asserted by the destination when it is not able to receive the data. Hence, in the SELF protocol, the two control signals valid (V) and stall (S) determine three possible channel states [6]:

- *Transfer (V & !S)* : The source sends valid data and the destination can receive it.
- *Idle (!V)* : The source does not send valid data.
- *Retry (V & S)* : The source sends valid data, but the destination can not receive it. The source sustains the valid data until the destination is able to store the data.

Note that in the Retry state, the channel is allowed to stay in the same state or move to the Transfer state. In particular, a transition

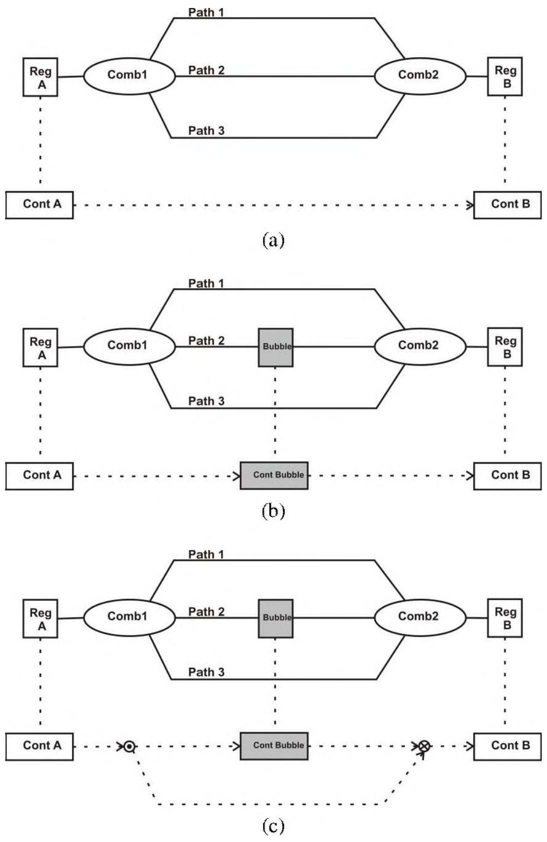


Fig. 2. Elasticization of reconvergent fanouts.

from the Retry to the Idle state is not allowed in the protocol. Note, also, the destination can stall even if the source does not send valid data.

Similarly, an EB can be in one of three states depending on the number of data tokens it holds. Following are the different states as well as the corresponding values of the EB output control signals (i.e., *Vr* and *Sl* as in Fig. 1):

- *Empty (!Vr & !Sl)* : The EB (or its two latches) holds no valid data. An EB in the Empty state is frequently referred to as a bubble.
- *Half (Vr & !Sl)* : The EB holds one data token.
- *Full (Vr & Sl)* : The EB holds two data tokens.

An EB can be initialized in any of the above states based on its reset wiring.

The first step in synchronous elasticization is to replace each flip-flop in the original design with an EB. Latches can similarly be replaced by EHBs. Following this step, communication between registers of the original design are analyzed. Each register-to-register data communication must be accompanied by a control channel consisting of the valid and stall signals. A network of control channels is then designed that reflects the data network. Join components are used to join control channels targeting the same destination register from multiple source registers. Fork components are employed when a single source register communicates with multiple destination registers. In this paper join and fork modules are designated with the $\otimes$ and $\odot$ symbols, respectively.

III. Elasticization Of Reconvergent Fanouts

Consider the reconvergent fanout between two registers A and B in Fig. 2. Datapaths are drawn in solid lines while control channels (the stall and valid signals pairs) are drawn in dotted lines. There are different ways of elasticizing such a register-to-register communication. Only one control channel is used for three reconvergent datapaths in Fig. 2a. This design is optimum in terms of the total number of used join and fork components (zero in this case).

However, modifications to the circuit of Fig. 2a are required when one (or more) bubbles are inserted into the datapaths. For example, consider the case when a bubble is required to be inserted at datapath 2. The circuit of Fig. 2a is modified in Fig. 2b by adding a bubble into datapath 2 with its associated controller placed in series without forks or joins. In this case, the control plane doesn't have sufficient information to control the data flow in all three paths. Hence, the circuit of Fig. 2b will malfunction.

In general, there is a need for one control channel for each datapath where a bubble may be inserted. Thus a conservative, but correct, way to elasticize reconvergent fanouts is to have a separate control channel corresponding to each datapath. This way, bubbles can be arbitrarily inserted in the design. A more efficient way – sufficient for this example – is to have only two control channels to represent the three datapaths; the first to control datapath 2 with the bubble, and the other to control datapaths 1 and 3. This is illustrated in Fig. 2c.

Fig. 3 shows an example from our case study, the MiniMIPS processor. MiniMIPS has four 8-bit instruction registers I1, I2, I3 and I4. These four registers are fed from the memory one at a time based on the enable signal IRWrite. IRWrite is generated by the MiniMIPS controller. Let ExMem and C be the names of the control channels corresponding to the memory output data and the IRWrite signal, respectively. Fig. 3b and 3c show two possible elasticization schemes of that circuit. Fig. 3c is the optimum solution in terms of the total number of join and fork components.

However, potential modifications to the circuit of Fig. 3c could be required when one (or more) bubbles are to be inserted in one (or more) datapaths. For example, consider the case when a bubble is required to be inserted just before the data input D of I3 (on the *data wire* x shown in Fig. 3a). For the circuit in Fig. 3b, that bubble (or more accurately, its *bubble controller*) will be directly inserted at the *control channel* x1. Yet, for the circuit in Fig. 3c, there is no control channel in the control plane that corresponds to data wire x. Thus inserting a bubble controller at channel x2 results in incorrect functionality because it affects the control channels corresponding to the datapaths of both MemData and IRWrite to I3. However, the requested bubble at x is only in the datapath of MemData to I3 (i.e., IRWrite-I3 datapath should not be affected). Hence, circuit of Fig. 3c must be modified so that insertion of the required bubble (and its controller) is possible. The modification is shown in Fig. 3d. The bubble controller can now be directly inserted at control channel x3. In conclusion, caution is required while trying to optimize control channels when bubbles may need to be inserted.

IV. Fork And Join Implementation Choices

In this section, we present some existing, as well as, new fork and join designs and discuss the correctness of combining them during an elasticization flow.

A. Lazy Fork

The lazy fork does not propagate a (valid) data token from its stem to its *ready* branches until *all* its branches are ready to receive it. We

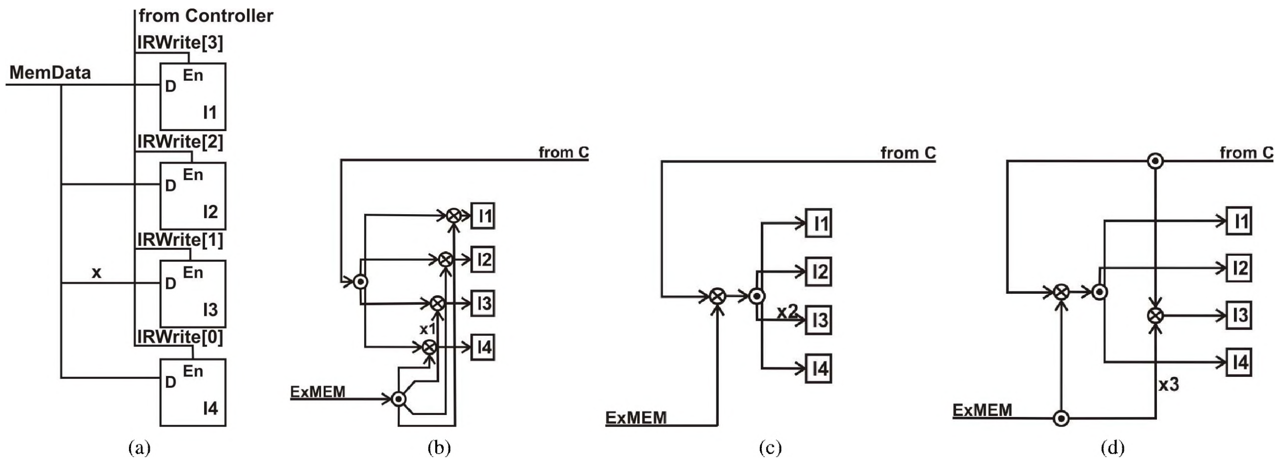
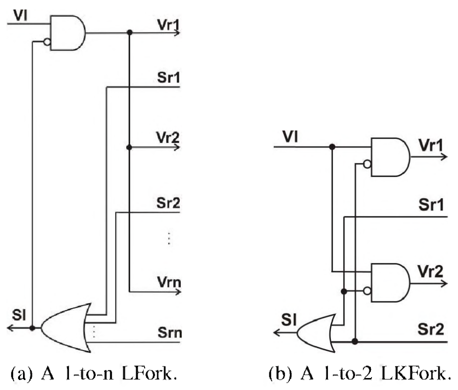


Fig. 3. Control channels optimization example.



(a) A 1-to-n LFork. (b) A 1-to-2 LKFork.

Fig. 4. Two lazy fork implementations.

say a branch is ready if its corresponding stall signal is zero. Besides, the stem stalls if any of the branches stalls.

The lazy fork can be implemented in different ways. Fig. 4a shows an implementation of *LFork* similar to the one reported in [6], [8]. We propose another implementation of the lazy fork in Fig. 4b. We refer to it as *LKFork*. The valid signal of each channel is controlled separately unlike the LFork. The valid signal on channel i, V_{ri}, of the 1-to-n LKFork is given by the following equation:

$$V_{ri} = V_l . \prod_{j=1, j \neq i}^{n} \bar{S_{rj}} \qquad (1)$$

while the stall on the source channel, S_l, is defined as follows:

$$S_l = \sum_{i=1}^{n} S_{ri} \qquad (2)$$

B. Eager Fork

The eager fork (*EFork*) will propagate valid tokens independently on each output channel, while stalling the input channel, until all output channels have accepted the data. Thus, when a (valid) data token is available at EFork stem, it will immediately pass the token to all its branches that are ready (i.e., not stalled), giving them an early start. Fig. 5 shows an n output extension of the EFork in [6]. Because of the early start provided by EFork, it may result in some performance advantage on the architecture level. However, this comes at the expense of more area and power consumption.

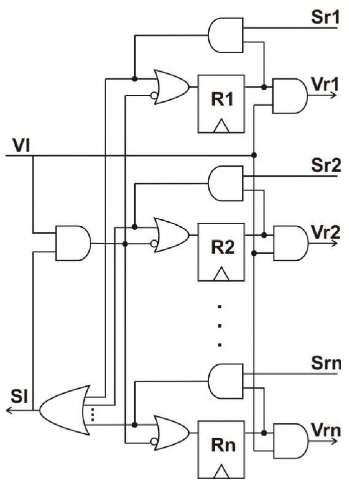
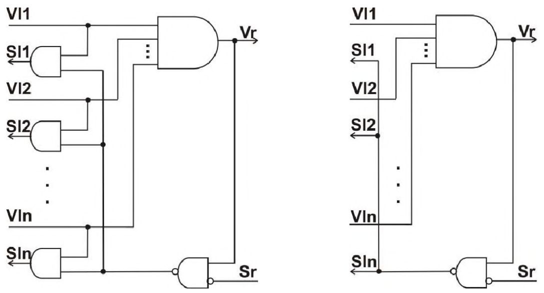


Fig. 5. A 1-to-n eager fork EFork.



(a) An n-to-1 LJoin. (b) An n-to-1 LKJoin.

Fig. 6. Two lazy join implementations.

C. Lazy Join

The lazy join must wait for all its input channels to carry valid data until it asserts its output channel valid signal. Fig. 6a shows an n-input lazy join (we refer to it as *LJoin*) proposed in [8]. Note that LJoin doesn't propagate a stall backward on any of its input channels unless that channel holds a valid control signal. This is done by using an output AND gate for each of the output stalls. We also propose another lazy join implementation in Fig. 6b. We refer to it as *LKJoin*. It avoids using the individual stall output AND gates.

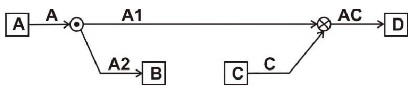


Fig. 7. An example of fork and join combination.

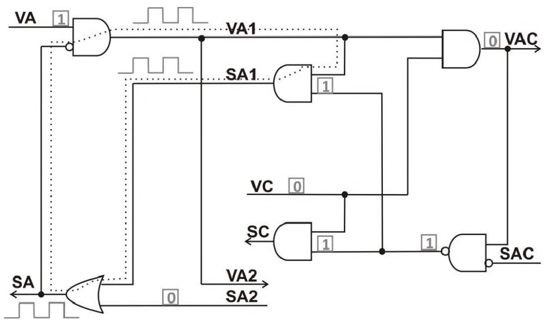


Fig. 8. LFork and LJoin combination.

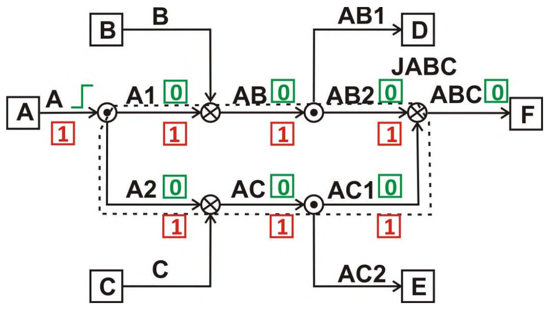


Fig. 10. LKFork and LKJoin combination.

D. Fork And Join Combinations

1) LFork-LJoin: If an LFork branch feeds an LJoin input, a logically unstable loop can be formed. This occurs when a logical cycle contains an odd number of inverting elements. Hence, this combination is not generally suitable for use in the SELF protocol.

To illustrate this point, assume four elastic buffers, *A*, *B*, *C* and *D* are connected as in Fig. 7. For simplicity, we draw only one line to represent the two wires of each control channel (valid and stall). The direction of a control channel is taken to be the same as the direction of the valid signal.

The circuit implementation of this connection using LFork and LJoin is shown in Fig. 8. Assume that elastic buffer *C* holds a bubble (i.e., its output valid signal is zero), while *A* holds data. Assume also that *SA2* is zero (*B* is not stalled). This connection will form a loop (shown in dotted lines in Fig. 8). The loop is logically unstable since it has an odd number of inverting elements. This results in an oscillation inside the loop as well as on the *SA* wire.

2) LFork-LKJoin: If an LFork branch feeds an LKJoin input, the combination can cause a deadlock in the control network of a synchronous elastic design. Hence, this combination is not generally suitable for use in the SELF protocol.

We use the same block diagram of Fig. 7 to illustrate the point. The circuit implementation of this connection using LFork and LKJoin is shown in Fig. 9. It can be easily shown that if *VA* is zero, *VA1* must also be zero and *VAC* be zero. This will force *SA1* to be one, *SA* to be one and *VA1* to be zero. Apparently, the loop shown in dotted lines forms a latch, since all its wires can simultaneously carry controlling

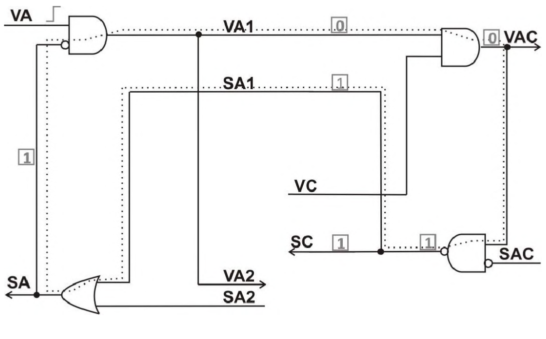


Fig. 9. LFork and LKJoin combination.

values to all the gates in the loop. Hence, after a zero on *VA*, the system will deadlock. *VA2*, *VAC*, *SC* and *SA* will be stuck at zero, zero, one and one, respectively.

3) LKFork-LKJoin: Through similar argument to Section IV-D2, we can show that LKFork-LKJoin combination can result in a deadlock. Consider, for example, the network structure of Fig. 10. Assume that we use LKFork and LKJoin to implement all the forks and joins in the figure. The boxes above the channel line represent valid signal values, and below are stall values. Once VA is zero, VA1, VA2, VAB, VAC, VAB2, VAC1 and VABC will also be zero. This, in turn, will cause SAB2 and SAC1 to be one (through join JABC) and SAB, SAC, SA1, SA2 and SA to be asserted. Again, the dotted loop will stuck at these values whatever the control network inputs are. Hence, this combination is not generally suitable for use in the SELF protocol.

4) LKFork-LJoin: In all the network structures we examined, the loops formed by LKFork-LJoin combination are logically stable (i.e., have even number of inverting elements). However, since there are no state holding elements (e.g., flip-flops) in these loops, any glitches can dangerously oscillate back and forth in a complex control network. Consider for example the LJoin in Fig. 6a. If *VI1* does a zero to one transition while all the other input valids are one. Without appropriate delay matching inside the join, a short positive glitch will propagate on *SI1*. Besides, if zero and one values are *simultaneously* injected at *different* points in a loop, the two values can race around that loop causing oscillation. We also noticed that deadlock can occur under certain values of the input arrival times and gate delays in some structures. Research is in progress to determine the sufficient timing constraints and device sizing that can eliminate such oscillations and possible deadlocks.

5) EFork-LJoin/LKJoin: Eager forks inherently cut the cycles through the flip-flops used at each of its outputs. Hence, they do not suffer from any of the loop problems mentioned above. An EFork-LKJoin can show slight area and power advantage over an EFork-LJoin (due to the removal of one AND gate from each join input channel).

V. Case Study: MiniMIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture with 32 registers, first designed by Hennessy [10]. The MiniMIPS is an 8-bit subset of MIPS, fully described in [11]. The MiniMIPS uses an 8-bit datapath. Only 8 registers are implemented and the program counter is also 8-bit long.

The MiniMIPS is used for a case study of elasticization. Fig. 11 shows a block diagram of the ordinary clocked MiniMIPS. The MiniMIPS has a total of 12 synchronization points (i.e., registers), shown as rectangles in Fig. 11: *P* (program counter), *C* (controller),

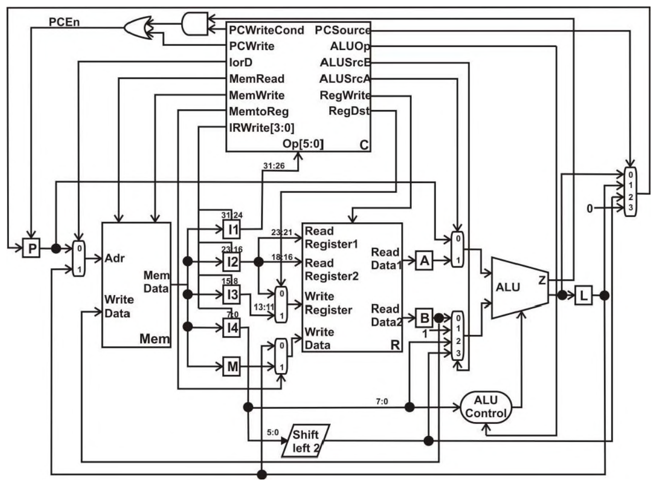


Fig. 11. Block diagram view of the ordinary clocked MiniMIPS. Adapted from [12], [11] with permission from Elsevier and Pearson Education.

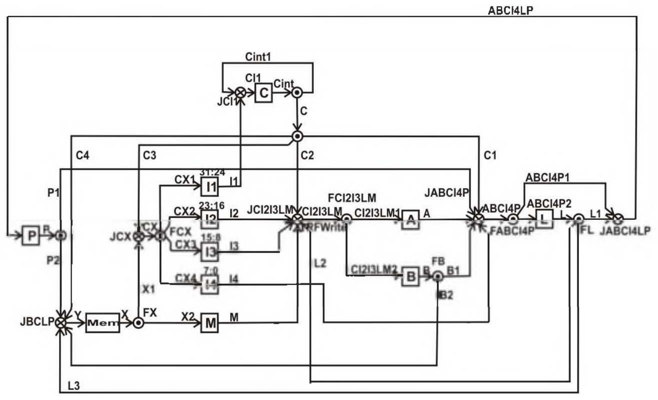


Fig. 12. Control network of the elastic clocked MiniMIPS.

$I1, I2, I3, I4$ (four instruction registers), A, B and L (ALU two input and one output registers, respectively), M (memory data register), R (register file) and Mem (memory). To perform elasticization, each register is replaced by an EB. Then, the register to register data communications in the MiniMIPS are analyzed. The following registers pass data to both A, B : R, and to R : $C, I2, I3, L, M$, and to C : $C, I1$, and to $I1, I2, I3, I4$: C, Mem, and to L : $A, B, C, I4, P$, and to M : Mem, and to Mem : B, C, L, P, and to P : $A, B, C, I4, L, P$. Then, for each register to register data communication there must be a corresponding control channel. The resultant control network can be implemented in different ways. We tried to manually construct such a network with the minimum total number of joins and forks (to reduce area and power consumption). The resultant control network of the elastic MiniMIPS is shown in Fig. 12. From the control point of view, the register file (R) in a microprocessor could be considered as combinational units [6]. Hence, we did not incorporate a separate EB for the register file (R) in Fig. 12. The chip was fabricated in a 0.5 μm process without the memory module.

From the elastic control point of view, even the MiniMIPS controller signals (e.g., RegWrite, IRWrite, ..etc) are considered part of the data plane and they need their own control channels. Mapping between datapath signals in the clocked MiniMIPS and the control channels in the elastic MiniMIPS should be self explanatory for most signals. RFWrite, in Fig. 12, is the register-file-write control channel. RFWrite_valid must be active if data is going to be written in the register file. Therefore, RFWrite_valid has been ANDed with RegWrite inside the register file.

TABLE I
CLOCKED AND ELASTIC MINIMIPS CHIPS RESULTS. MEASUREMENTS ARE DONE AT 5V AND 30°

	Clocked MiniMIPS	Elastic MiniMIPS	Penalty
Area (μm X μm)	1246.765 X 615.91	1284.1 X 771.54	29%
P_{dyn} @80 MHz (mW)	330	373	13%
P_{idle} (μW)	16.3	25.8	58.3%
f_{max} (MHz)	91.7	92.2	-0.5%

VI. Verification, Fabrication And Performance Evaluation

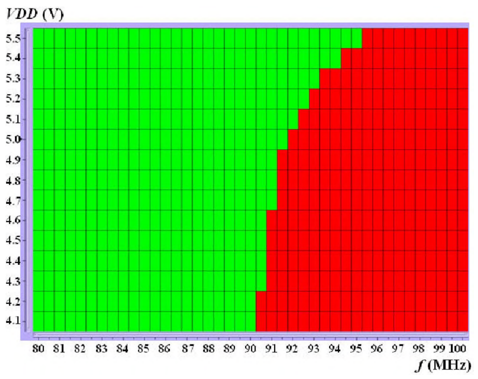
Both the clocked and elastic (without bubbles) MiniMIPS have been synthesized, placed, routed and fabricated in a 0.5 μm technology. The functionality of the fabricated processors have been verified on Verigy's V93000 SOC tester using the testbench in [11]. The EFork and LJoin circuits (of Fig. 5 and 6a) have been used in the elastic MiniMIPS. Table I summarizes chips measurements. It shows that elasticizing the MiniMIPS has area and dynamic and idle power penalties of 29%, 13% and 58.3%, respectively. For accurate idle power comparison, both designs have been set to the same state (through a test vector) before measuring the average idle supply current.

On the other hand, there is no performance loss due to elasticization. Both MiniMIPS have been fabricated without the memory block. Memory values have been programmed inside the tester. Hence, we had to make an assumption about the memory access time, and the assumptions affect the maximum operating frequency of both MiniMIPS in the same way. Therefore, the actual value of memory access time would minimally affect the performance comparison. Hence, we chose an arbitrary value of zero for memory access time for both designs. Shmoo plots for both clocked and elastic MiniMIPS are shown in Fig. 13.

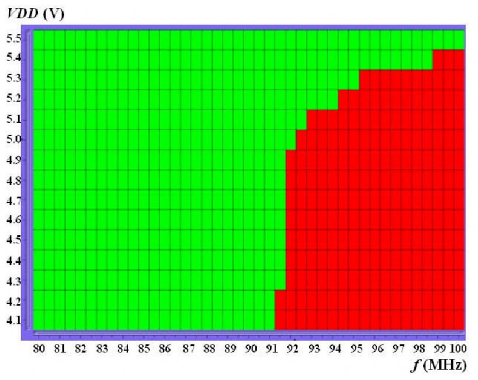
The noticeable area and power overheads of the elastic MiniMIPS are mainly due to the usage of eager forks. EFork has one flip-flop per each branch that consumes power every cycle. Add to this, its gate complexity. This motivated the research toward less complex implementations of eager forks or switching to lazy protocols.

Note that these measurements do not take advantage of bubbles problems that occur if one needs to have flexible interface latencies or extra pipeline stages inserted. In the presence of bubbles, eager forks can enjoy performance advantage over lazy protocols. To measure this advantage, we inserted a different number of bubbles at the register file outputs (i.e., before registers A and B, simultaneously). Table II compares the number of cycles required to run simulations of lazy and eager protocol implementations using the testbench in [11]. For the lazy protocol, we used LKFork-LJoin combination. The behavioral simulations used some timing constraints enforced to avoid possible oscillations, see Sec. IV-D4. Table II shows that, in this case, there is an advantage for using eager forks, specially with a large number of bubbles in the system. The table also shows that there is no runtime penalties due to elasticization in the absence of bubbles.

The runtime advantage of the eager versus lazy designs (see Sec. IV-B) is illustrated in the following example. Fig. 14 shows a simplified part of the MiniMIPS control network. We added one bubble before the A register, and another one before the B register, labeled $b1$ and $b2$ respectively. Consider the clock cycle when VA



(a) Shmoo plot for clocked MiniMIPS.



(b) Shmoo plot for elastic MiniMIPS.

Fig. 13. Fabricated chips shmoo plots. Red boxes are for failed tests, while green are for passed ones.

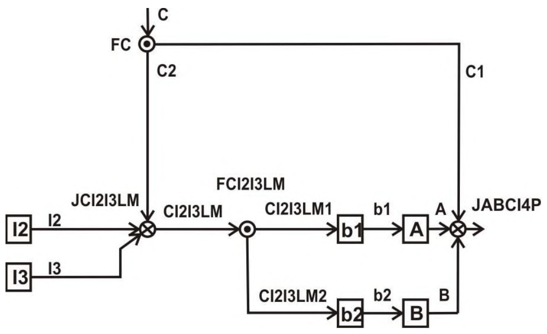


Fig. 14. A sample structure where eager protocol will have runtime advantage over lazy.

and VB go low. $SC1$ will go high through join $JABCI4P$. In FC (assuming $SC2$ is low), VC is high, $SC1$ is high. A lazy FC will invalidate the data at $C2$ (i.e., deasserts $VC2$) until $SC1$ goes low again. Hence, no new data token can be written at register $b1$ or $b2$ until the stall condition on $C1$ is removed (i.e., $SC1$ goes low again). On the other hand, an eager FC will validate the data on $C2$ (i.e., asserts $VC2$) for the first clock cycle giving $C2$ branch an early start. Hence, new data tokens can be written immediately in registers $b1$ and $b2$ in the following cycle.

VII. Conclusion

Synchronous elasticization is an approach of converting ordinary clocked designs into latency insensitive implementations. Being latency insensitive, the synchronous elastic designs are able to tolerate interconnect, interface, and internal latency variations without affecting the system correctness. This allows for increased modularity and facilitates IP reuse. We presented some practical considerations of elasticizing reconvergent fanouts. We also investigated the suitability of different published, as well as new, join and fork implementations for usage in the elastic control network. We also demonstrated that elasticization can be expensive. Measurements of two MiniMIPS processor chips in a 0.5 μm node showed that elasticization results in area and dynamic and idle power penalties of 29%, 13% and 58.3%, respectively, without any performance loss. These measurements, nonetheless, do not take advantage of bubbles that occur if one needs to have unpredictable interface latencies or extra pipeline stages inserted. We finally demonstrated the possible architectural performance advantage of eager over lazy protocols in the presence of bubbles.

TABLE II
Simulation runtime (in terms of #cycles) of the testbench in [11] in case of lazy and eager protocols. Bubbles are inserted at the register file outputs.

Fork/Join Combination	0 Bubbles	1 Bubble	3 Bubbles
Lazy Protocol: LKFork-LJoin	98	195	389
Eager Protocol: EFork-LJoin	98	147	245
Clocked MiniMIPS	98	-	-

Acknowledgments

This work was supported by the National Science Foundation under grant CCF-0810408.

References

[1] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary, "Synchronous elastic networks," in *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, Nov. 2006, pp. 19–30.
[2] M. Bohr, "Interconnect scaling-the real limiter to high performance ulsi," in *Electron Devices Meeting, 1995., International*, Dec 1995, pp. 241–244.
[3] R. Ho, K. Mai, H. Kapadia, and M. Horowitz, "Interconnect scaling implications for cad," in *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, 1999, pp. 425–429.
[4] L. Carloni and A. Sangiovanni-Vincentelli, "Coping with latency in soc design," *Micro, IEEE*, vol. 22, no. 5, pp. 24–35, Sep/Oct 2002.
[5] L. Carloni, K. Mcmillan, and A. L. Sangiovanni-VincentelliR, "Theory of latency insensitive design," in *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 20, no. 9, Sep 2001, pp. 1059–1076.
[6] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *ACM/IEEE Design Automation Conference*, July 2006, pp. 657–662.
[7] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1437–1455, Oct. 2009.
[8] 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 *8th International Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp. 3–12.
[9] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006.
[10] J. H. et al., "The MIPS Machine," in *COMPCON*, 1982, pp. 2–7.
[11] N. Weste and D. Harris, *CMOS VLSI design: a circuit and systems perspective*, 2004.
[12] D. Patterson and J. Hennessy, *Computer Organization and Design*, 2004.