Design and Verification of Lazy and Hybrid Implementations of the SELF Protocol

Eliyah Kilada, Kenneth S. Stevens

Design and Verification of Lazy and Hybrid Implementations of the SELF Protocol

Eliyah Kilada[1] and Kenneth S. Stevens[2]

[1] eliyah.kilada@utah.edu
[2] kstevens@ece.utah.edu
University of Utah, Electrical and Computer Engineering,
Salt Lake City, Utah, USA

Abstract. Synchronous Elasticization converts an ordinary clocked circuit into Latency-Insensitive (LI) design. The Synchronous Elastic Flow (SELF) is an LI protocol that can be implemented with eager or lazy evaluation in the data steering network. Compared to lazy implementations, eager SELF designs have no combinational cycles and can have a performance advantage, but consume more area and power. The design space of lazy SELF protocols is evaluated and verified. Several new designs are mapped to hybrid eager/lazy implementations that retain the performance advantage of the eager design but have power advantages of lazy implementations.

Key words: Latency-insensitive design, lazy SELF implementation, hybrid SELF implementation, elasticization, verification, MiniMIPS processor.

1 Introduction

Latency insensitivity (LI) allows designs to tolerate arbitrary latency variations in their computation units as well as communication channels [1]. This is particularly important for interfaces where the actual latency can not be accurately estimated or is required to be flexible. An Example of the former are systems with very long wire interconnects. Interconnect latency is affected by many factors that may not be accurately estimated before the final layout [2]. On the other hand, some applications require flexible interfaces that tolerate variable latencies. Examples can include interfaces to variable latency ALU's, memories or network on chip. It has been reported that applying flexible latency design to the critical block of one of Intel's SoCs (H.264 CABAC) can achieve 35% performance advantage [8].

Synchronous elasticization is a technique of converting an ordinary clocked circuit into an LI design [5, 3, 10, 7]. Unlike asynchronous circuits, synchronous elastic circuits can be designed with conventional CAD flows using STA [3, 14]. The Synchronous Elastic Flow (SELF) is a communication protocol in synchronous elastic designs [5]. Eager implementation of the SELF protocol enjoys no combinational cycles and also may have performance advantages in some

designs when compared to lazy implementations. However, eager protocols are more expensive in terms of area and power consumption. The LI control network area and power consumption may become prohibitive in some cases [3]. Measurements of a MiniMIPS processor fabricated in a 0.5 μm node show that elasticization with an eager SELF implementation results in area, dynamic and leakage power penalties of 29%, 13% and 58.3%, respectively [11]. Hence, minimizing these overheads is a primary concern. For an attempt to achieve that goal, an algorithm that minimizes the total number of control steering units (i.e., joins and forks) in the LI control network has been developed [12].

Lazy SELF implementations may be an attractive solution. Unfortunately the standard implementation suffers from combinational cycles that make it an unreliable design [5, 11]. This work defines a larger design space that can be employed to implement lazy channel protocols and to verify correctness of these protocols both independently and when combined with the standard eager protocol.

1.1 Contribution

A formal investigation of a complete set of lazy SELF protocol specifications is reported. This includes introducing new lazy join and fork structures, which are verified along with the existing designs. A novel hybrid implementation flow is then introduced that combines the advantages of both eager and lazy implementations. The hybrid SELF essentially avoids some of the redundancy of the eager implementation without any performance loss. Moreover, it is combinational cycle free. The hybrid SELF network is demonstrated with the design of a MiniMIPS processor. The hybrid implementation achieves the same runtime as an *all* eager implementation with a reduction of 31.8% and up to 32.5% and 32.1% in the control network area and dynamic and leakage power consumption, respectively.

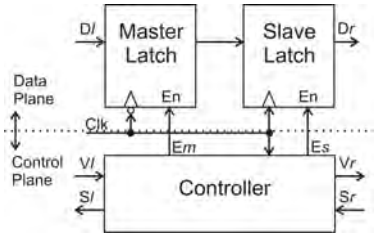


Fig. 1. An EB implementation.

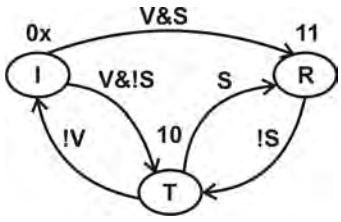


Fig. 2. SELF channel protocol.

2 SELF Overview

A LI network consists of two components: Elastic Buffers (EBs) and a control network that distributes the handshake signals to the EBs. The components of the control network (e.g., joins and forks) do not buffer the data. They, nonetheless, along with the EB controllers, schedule the data token transfers. The lazy and eager properties of a SELF system are implemented in the control network.

An elastic system replaces the flip-flops used as pipeline latches in a clocked system with EBs. EBs serve the purpose of pipelining a design as well as synchronization points that implement an LI protocol, also allowing the clocked pipeline to be stalled.

Figure 1 shows a block diagram implementation of an EB. An EB consists of a data-plane (double latches) and a controller. It can be in the Empty (bubble), Half or Full states depending on the number of data tokens its two latches are holding. Sample implementation of the EB controller can be found in [5]. EB controllers communicate through control channels. Each channel contains two control signals. 'Valid' (V) travels in the same direction as the data and indicates the validity of the data coming from the transmitter. 'Stall'(S) travels in the opposite direction and indicates that the receiver can not store the current data.

The SELF channel protocol is shown in Fig. 2 [5]. The three states of the channel protocol in Fig. 2 are (a.) *Transfer* (T): $V\&!S$. The transmitter provides valid data and the receiver can accept it. (b.) *Idle* (I): $!V$. The transmitter does not provide valid data. This paper identifies two Idle conditions: $I0$ ($!V\&!S$) where the receiver can accept data and $I1$ ($!V\&S$) where the receiver can not accept data. (c.) *Retry* (R): $V\&S$. The transmitter provides valid data, but the receiver can not accept it. In the Transfer state, the valid data must be maintained on the channel until it is stored by the receiver.

When the connection between EBs is not point-to-point, a control network is required to steer the Valid and Stall signals between the different EBs. The control network is composed of control channels connected through control steering units, namely, join and fork components. A join element combines two or more incoming control channels into one output control channel. A fork element copies one incoming control channel into two or more output control channels. Fork and join components are represented by $\odot$ and $\otimes$, respectively. The SELF protocol used over the control channels can be implemented using an eager or lazy protocol. Hereafter we use the term *control network* to aggregately refer to the joins, forks, and EB controllers in a system.

We introduce the notion of a *control buffer* in order to gain understanding of the design and verification of control network components, such as joins and forks. A linear control buffer simply breaks the control signals in a channel into left and right channels. Such a buffer will have two inputs: the Valid on the left channel and Stall on the right channel, and two outputs: the Stall on the left channel and Valid on the right.

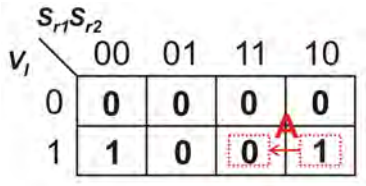


Fig. 3. V_{r1} of $LF01$.

3 SELF Channel Protocol Verification

All network components are verified to be conformant to the SELF channel protocol. The correctness requirements for the channel protocol are adapted from the general elastic component conditions consisting of persistence, freedom from deadlock, and liveness [14]. A fourth constraint is added here that disallows glitching on the control wires.

1. *Persistence.* No $R \rightarrow I$ transition may occur.
2. *Deadlock freedom.* For each component in the verification, at least two states can be reached from any other reachable state [16].
3. *Liveness.* The liveness condition is one of data preservation. Lazy control buffers must have the same number of tokens transferred on all their channels. This functional requirement is a special case of the liveness condition in [14]. This is implemented by creating token counters on all the lazy control buffer channels and verifying that they are always equivalent.
4. *Glitch Free.* No $S\uparrow$ signal transition may occur in state I. The specification of the idle protocol state I in Fig. 2 does not constrain the behavior of the Stall signal. This allows glitching on the control wires to occur. If the Stall signal is not allowed to rise in the idle state then glitching will not occur. This requirement is not explicit in the SELF specifications. However, it can be observed that this transition is not possible in published Elastic Buffer (EB) or Elastic Half Buffer (EHB) designs [5, 9]. If control wire glitching is possible, then the composition of some forks and joins may not be compliant with the channel protocol. For example, the Karnaugh map of $LF01$, one of the two lazy forks proven to be SELF compliant, is shown in Fig. 3. Transition A occurs when S_{r2} rises in the idle state. While this glitching transition is valid according to the channel specification, it results in V_{r1} falling, which produces an illegal $R \rightarrow I$ transition on channel r_1. Since this transition can never happen unless channel r_2 can make an $S\uparrow$ transition glitch, we add this condition to our verification suite.

4 Lazy SELF Control Network Design

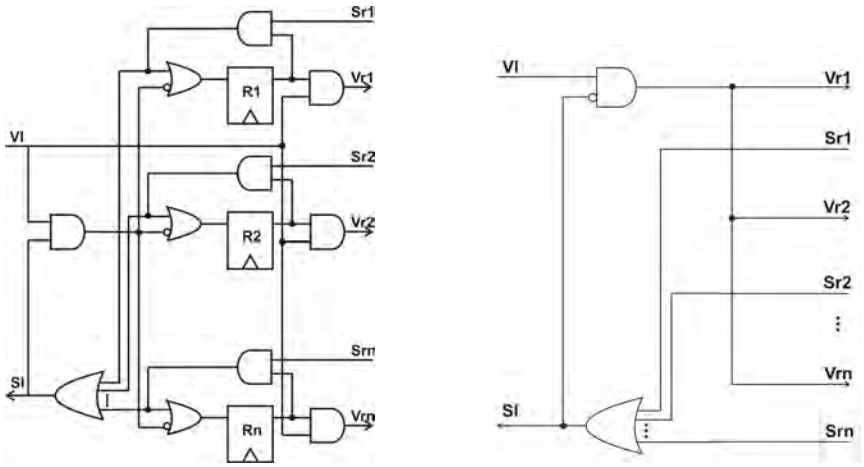
A truth table can be created to specify the permissible behaviors for the control buffer left Stall and right Valid signals that conform to the SELF channel

protocol of Sect. 2. Such a truth table shows the flexibility in design choices that can be made. The same procedure is performed for the lazy fork and join components.

5 Fork Components

5.1 Eager Fork

The eager fork ($EFork$) will immediately forward valid data tokens presented at the root to all branches that are not stalled. If any of the branches of the fork are stalled, the root of the $EFork$ will stall until all its branches receive the data. This gives the earliest possible data transfer to the branches that are ready to receive data. Hence, the $EFork$ can result in performance advantage over lazy forks in some systems. Due to the necessary pipelining that occurs in the control signals, the $EFork$ incorporates one flip-flop per branch. The control flip-flop must be updated every clock cycle to sample changes. Moreover, eager forks have higher logic complexity comparing to lazy. This makes the $EFork$ expensive in terms of both area and power consumption. Figure 4a shows an n output extension of the $EFork$ [11, 5].



(a) A 1-to-n $EFork$. (b) A 1-to-n lazy fork.

Fig. 4. Sample forks.

5.2 Lazy Fork

The lazy fork does not propagate valid data from its root to its branches until *all* branches are ready to store the data. A sample lazy fork is shown in Fig. 4b.

Lazy Fork Synthesis. The truth table for a lazy fork is shown to be purely combinational. Thus it is easily represented with the Karnaugh map (KM) shown in Fig. 5. The KM has two don't care terms m_0 and m_1 giving four possible designs. Each implementation is denoted as LFm_0m_1 (e.g., $LF00$, $LF01$,.. etc). Table 1 maps all the published lazy forks we were able to find to those of this paper.

The hand translation of the fork as a control buffer may still result in illegal channel behavior on one or more of the channels due to the interactions between branches of the fork and join. Thus we employ a rigorous verification methodology to prove correctness of the designs. Indeed, verification shows that two of the four possible designs do not fully obey the SELF channel protocol.

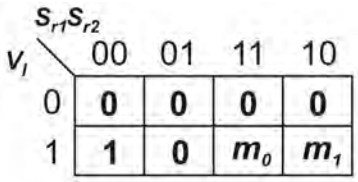


Fig. 5. Lazy fork specifications (V_{r1}).

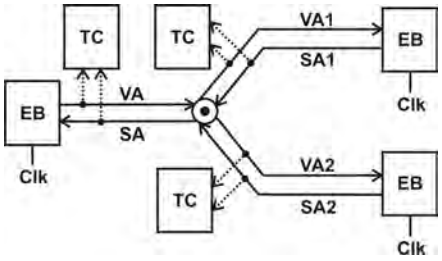


Fig. 6. Lazy fork verification setup.

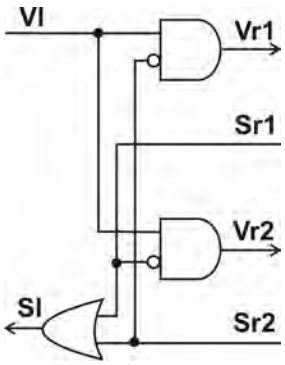


Fig. 7. A 2-output $LF01$ implementation.

Fork Verification. The setup of Fig. 6 is used to verify correctness of the fork designs. The root channel (A) as well as the branches ($A1$ and $A2$) are connected to three elastic buffers (EBs) as well as data token counters (TCs).

Table 1. Mapping between published and this paper lazy forks and joins.

Fork [10]	$LF00$	Join [10]	$LJ0000$
Fork [5]	$LF00$	Join [5]	$LJ0000$
LFork [11]	$LF00$	LJoin [11]	$LJ0000$
LKFork [11]	$LF01$	LKJoin [11]	$LJ1111$

This work employs the EB implementation published in [5]. The counters track the number of clock cycles that the channel is in the transfer state T. The structure is modeled and passed to a symbolic model checker, NuSMV [4].

All constituent blocks are connected *synchronously*. Synchronous connection guarantees that all modules advance in lock-step. Logic delays are then executed in internal cycles of the verification engine. All combinational logic are modeled to have zero delay. The clock generator is modeled to have a unit delay for each phase. For example, following is the $LF00$ model:

```
MODULE LF00(Vl,Sr1,Sr2)
DEFINE Sl := Sr1 | Sr2 ; DEFINE Vr1 := Vl & (!Sr1) & (!Sr2) ; ...
```

The four properties from Sect. 3 are applied to each design. The properties used for these checks are described below.

1. *Persistence*. For each channel (i.e., A, $A1$ and $A2$) we verify that no R $\rightarrow$ I transition occurs:
   ```
   DEFINE R_A := VA & SA ; -- Retry on channel A
   DEFINE I_A := !VA ; -- Idle on channel A
   PSLSPEC never {[*]; R_A; I_A};
   ```
   Out of the 4 lazy fork implementations only $LF00$ and $LF01$ pass this check.
2. *Deadlock freedom*. At least two states are verified as reachable from all other reachable states [16]. For example, inside the $LF00$ module the following CTL properties verify two states are reachable:
   ```
   SPEC AG EF (Vr1=1 & Vr2 =1 & Sl=0);
   SPEC AG EF (Vr1=0 & Vr2 =0 & Sl=0);
   ```
   Note that a state in $LF00$ is defined by the three variables: V_{r1}, V_{r2} and S_l. All four lazy fork implementations pass this check.
3. *Liveness* is calculated through *data token preservation*. Let the number of data tokens transferred at the fork root channel and the two branch channels be: d_l, d_{r1} and d_{r2}, respectively. (d_i is, equivalently, the number of clock cycles where channel i was in the Transfer state (T) (i.e., $V_i\&!S_i$).) The number of data tokens transferred at its root channel must always be the same as those at its branches. (i.e., the following requirement must always hold: $d_{ri} - d_l = 0$

for $i \in \{1, 2\}$.) We use the following code to model a token counter for channel i. The model counts on the negative edge of the clock.

```
MODULE TokenCounter (Clk,Vi,Si)
VAR Count: 0..31;
ASSIGN
init (Count) := 0;
next (Count) := case
(Clk=1)&(next(Clk)=0)&(Vi=1)&(Si=0)&(Count < 31): Count + 1;
1: Count;
esac;
```

Since NuSMV supports finite types only. Without loss of generality, we chose the upper limit of the *Count* variable to be a sufficiently large number (32 in this case). For each branch we define and check the following property:

```
DEFINE TokenCountError_A1 := case (dl != dr1):1; 1:0; esac;
PSLSPEC never {[*]; TokenCountError_A1};
```

All the four lazy fork implementations pass this check.

4. *No glitching.* This verifies that the Stall signal does not rise in the idle state:

```
DEFINE I0_A := !VA & !SA ; -- Idle0 on A
DEFINE I1_A := !VA & SA ; -- Idle1 on A
PSLSPEC never {[*]; I0_A; I1_A};
```

All lazy fork implementations pass this check.

Hence, among the four possible lazy fork implementations, only $LF00$ and $LF01$ conform to our channel specification. Similarly, the $EFork$ of Sect. 5.1 is also verified. Since the $EFork$ allows its ready branches to transfer tokens while stalled waiting for the other branches to be ready, the data token preservation requirement is: $0 \leq d_{ri} - d_l \leq 1$ for $i \in \{1, 2\}$. Indeed, $EFork$ passes these checks and, hence, is also compliant with the SELF protocol.

Lazy Fork Characterization. To help characterize the different fork implementations as well as their combinations with lazy joins in a network, we introduce the following definitions:

Definition 1. C_{Fr}, Fork Reflexive Characterization Set C_{Fr} *is a set of characterization elements* (c_{Fr})*, where:* $c_{Fr} \in \{I, N, 0, 1\}$

where

1. $c_{Fr} = I$ (or *inverting*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = !S_{ri}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Fr} = N$ (or *non-inverting*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = S_{ri}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
3. $c_{Fr} = 0$ (or *constant zero*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.
4. $c_{Fr} = 1$ (or *constant one*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Table 2. C_{Fr} Computation of $LF00$

V_l	S_{r2}	$S_{r1} \rightarrow V_{r1}$	c_{Fr}
0	0	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
0	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
1	0	$0 \rightarrow 1$ $1 \rightarrow 0$	**I**
1	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**

Table 3. C_{Ft} Computation of $LF00$

V_l	S_{r1}	$S_{r2} \rightarrow V_{r1}$	c_{Ft}
0	0	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
0	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
1	0	$0 \rightarrow 1$ $1 \rightarrow 0$	**I**
1	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**

Definition 1 can be easily extended to n-output forks with $n > 2$.

Table 2 illustrates C_{Fr} computation of $LF00$. From the table, C_{Fr} of $LF00$ is $\{0, I\}$. Similarly C_{Fr} of $LF01$ is $\emptyset$. This is because in $LF01$ (see Fig. 7), V_{ri} is not a function of S_{ri}. As we will show in Sect. 8.1, this property gives an advantage to $LF01$ since it can reduce the number of combinational cycles in the control network substantially.

Definition 2. C_{Ft}, Fork Transitive Characterization Set C_{Ft} *is a set of characterization elements* (c_{Ft})*, where:* $c_{Ft} \in \{I, N, 0, 1\}$

where

1. $c_{Ft} = I$ (or *inverting*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = !S_{rj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Ft} = N$ (or *non-inverting*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = S_{rj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
3. $c_{Ft} = 0$ (or *constant zero*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.
4. $c_{Ft} = 1$ (or *constant one*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Definition 2 can be easily extended to n-output forks with $n > 2$.

Table 3 illustrates C_{Ft} computation of $LF00$. From the table, C_{Ft} of $LF00$ is $\{0, I\}$. Similarly C_{Ft} of $LF01$ is also $\{0, I\}$.

6 Lazy Join

The lazy join has to wait for all its input branch channels to carry valid data until data is transferred on the output channel. A sample lazy join is shown in Fig. 8.

6.1 Lazy Join Synthesis

The synthesis of a lazy join as a control buffer is performed similar to the lazy fork. The KM is shown in Fig. 10. There are 16 possible implementations.

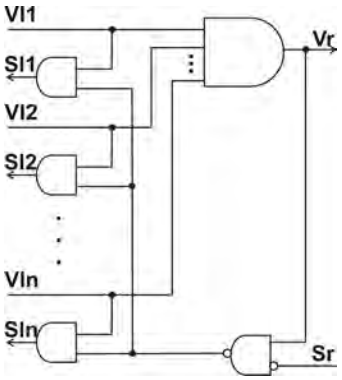


Fig. 8. An n-to-1 lazy join.

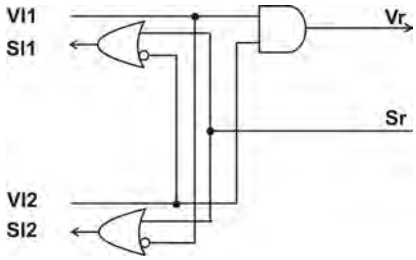


Fig. 9. A 2-input $LJ1011$ implementation.

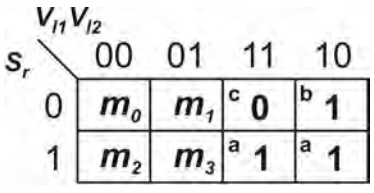


Fig. 10. Lazy join specification (S_{l1}).

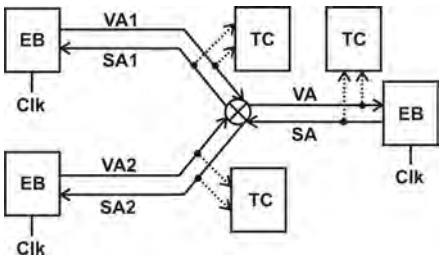


Fig. 11. Lazy join verification setup.

6.2 Lazy Join Verification

Similar to the lazy fork verification in Sect. 5.2, we use the structure of Fig. 11 to verify the different lazy join implementations. We check the following properties: 1. *Persistence*: All the 16 lazy joins pass this check. 2. *Deadlock freedom*: All the 16 joins pass. 3. *Data token preservation*: All the 16 joins pass. 4. *Glitch Free*: Out of the 16 lazy joins, only 6 pass. Only the following lazy join designs pass verification: $LJ0000$, $LJ0010$, $LJ0011$, $LJ1010$, $LJ1011$, $LJ1111$.

6.3 Lazy Join Characterization

To help characterize the different join implementations as well as their combinations with lazy forks in a network, we introduce the following definitions:

Definition 3. C_{Jr}, Join Reflexive Characterization Set C_{Jr} *is a set of characterization elements* (c_{Jr})*, where:* $c_{Jr} \in \{I, N, 0, 1\}$

where

1. $c_{Jr} = I$ (or *inverting*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} = !V_{li}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Jr} = N$ (or *non-inverting*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} = V_{li}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
3. $c_{Jr} = 0$ (or *constant zero*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.
4. $c_{Jr} = 1$ (or *constant one*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Definition 3 can be easily extended to n-input joins with $n > 2$.

Similar to Table 2, we can also find that C_{Jr} of $LJ0000$, for example, is $\{N, 0\}$. $LJ1011$ has a C_{Jr} of $\emptyset$. This is because in $LJ1011$ (see Fig. 9), S_{li} is not a function of V_{li}. As we will show in Sect. 8.1, this property gives an advantage to $LJ1011$ since it can reduce the number of combinational cycles in the control network substantially.

Definition 4. C_{Jt}, Join Transitive Characterization Set C_{Jt} *is a set of characterization elements* (c_{Jt})*, where:* $c_{Jt} \in \{I, N, 0, 1\}$

where

1. $c_{Jt} = I$ (or *inverting*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} = !V_{lj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Jt} = N$ (or *non-inverting*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} = V_{lj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
3. $c_{Jt} = 0$ (or *constant zero*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.
4. $c_{Jt} = 1$ (or *constant one*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Definition 4 can be easily extended to n-input joins with $n > 2$.

Similar to Table 3, we can also find that C_{Jt} of $LJ0000$, for example, is $\{I, 0, 1\}$.

7 Lazy SELF Networks

Unlike eager forks, lazy forks have no state holding elements (e.g., flip-flops). Hence, arbitrary connections of lazy joins and forks in a control network typically result in combinational cycles. These cycles can cause deadlock or oscillation due to logical or transient instability:

7.1 Deadlock - D

A combinational cycle can cause a deadlock if under some input sequence its internal signals can get stuck at certain values. For example, consider a structure in which a fork output channel is feeding a join (Fig. 12a). This structure is a basic building block of typical elastic control networks. Figure 13 shows a circuit implementation of Fig. 12a using $LF00$ and $LJ1111$.

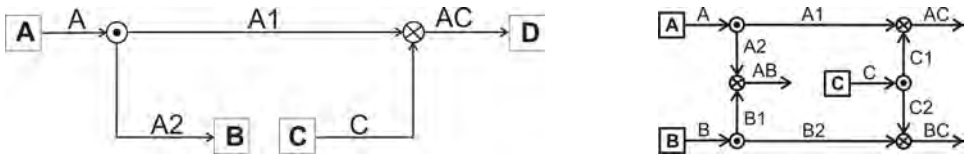


Fig. 12. Sample fork join combinations.

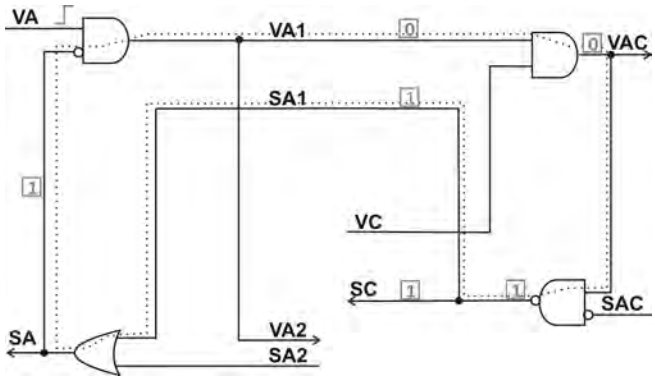


Fig. 13. LF00 and LJ1111 combination.

It can be easily shown that if VA is zero, $VA1$ and VAC must also 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 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.

In general, for the common structure of Fig. 12a, the following can be readily proved. Let C_{Jr1} (C_{Fr1}) and C_{Jt1} (C_{Ft1}) be the join (fork) reflexive and transitive characteristic sets of the lazy join (fork) used, $LJ1$ ($LF1$), respectively. Then, the connection of Fig. 12a will result in deadlock if the following condition holds: $C_{Jr1} = \{1, I\}$ and $C_{Fr1} = \{0, I\}$. To illustrate, since $C_{Fr1} = \{0, I\}$, therefore, for all the possible values of $LF1$ inputs, $VA1$ is either 0 or the inverse of $SA1$. Similarly, since $C_{Jr1} = \{1, I\}$, therefore, for all the possible values of $LJ1$ inputs, $SA1$ is either 1 or the inverse of $VA1$. Hence, once $VA1$ is 0 or $SA1$ is 1, the loop formed by $VA1$ and $SA1$ will stuck at these values.

Similarly, a deadlock will occur in the connection of Fig. 12b if the following condition holds: $C_{Jt1} = \{1, I\}$ and $C_{Ft1} = \{0, I\}$.

7.2 Oscillation due to Logical Instability - LI

A loop is logically unstable if it has an odd number of inverting elements. Under some input sequence, it can behave as a ring oscillator.

For example, consider the structure of Fig. 12a. Figure 14 shows a circuit implementation of that structure using $LF00$ and $LJ0000$.

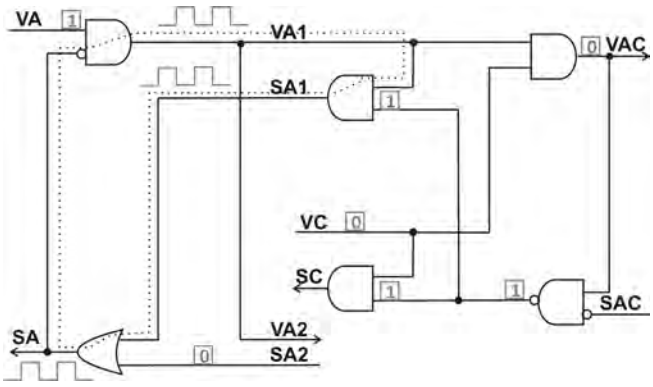


Fig. 14. LF00 and LJ0000 combination.

Assume the elastic buffer C in Fig. 14 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. 14). 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.

In general, for the common structure of Fig. 12a, the following can be readily proved. Let C_{Jr1} (C_{Fr1}) and C_{Jt1} (C_{Ft1}) be the join (fork) reflexive and transitive characteristic sets of the lazy join (fork) used, $LJ1$ ($LF1$), respectively.

Table 4. Lazy fork-join combination characterization. All other combinations (14 Forks × 10 Joins) are non-compliant with the SELF protocol.

	Join	0000	0010	0011	1010	1011	1111
Fork	$\frac{C_r}{C_t}$	$\frac{N,0}{I,0,1}$	$\frac{N,0,1}{I,0,1}$	$\frac{N,0,1}{I,0,1}$	$\frac{N,0,1}{I,1}$	$\frac{\emptyset}{I,1}$	$\frac{I,1}{I,1}$
0000	$\frac{I,0}{I,0}$	**LI**	**LI**	**LI**	**LI**	**D**	**D**
0001	$\frac{\emptyset}{I,0}$	**TI**	**TI**	**TI**	**D**	**D**	**D**

Then, the connection of Fig. 12a will result in logical instability if any of the following condition holds:

- $I \in C_{Jr1}$ and $N \in C_{Fr1}$.
- $N \in C_{Jr1}$ and $I \in C_{Fr1}$.

7.3 Oscillation due to Transient Instability - TI

Even if a combinational loop does have even number of inverting elements it can still cause oscillation in the elastic control network. Since the loop has more than one input, and in some input sequences, both logic one and zero values may be injected in the loop simultaneously. This can result in both values oscillating around the loop.

Table 4 shows the different lazy fork-join combinations characteristics. The table refers to the network structures of Fig. 12.

Research is still in progress to investigate whether the oscillation due to transient instability can be avoided by forcing network-specific timing constraints on the control network. However, a simpler solution, not only for transient instability, but also for deadlock and logical instability, is to use eager forks when needed to cut such combinational cycles. This will be discussed in Sect. 8.

Finally, the following logic was used for the root's Stall signal in all of the lazy forks investigated: $S_l = S_{r1}|S_{r2}$. Similarly, the lazy join elements used $V_r = V_{l1}\&V_{l2}$. Other implementations for these signals that consider flexibility allowed by lazy control buffers is not presented here. However, note that designs with additional logic will increase the probability of combinational loops in component composition.

8 Hybrid SELF Protocol

Two lazy forks and six lazy joins, as well as the traditional eager fork, have been proven to be compliant with our strict SELF channel protocol. Therefore, eager and lazy forks (and joins) can be *correctly* connected together as long as no combinational cycles are formed [14]. Eager forks exhibit no cycles and can achieve better runtime in some systems. However, they consume more power and

area than lazy forks. Hence, we propose to use a hybrid SELF implementation, that uses both eager and lazy forks, has no cycles, and achieves the same runtime as an *all* eager implementation. Hybrid implementation should keep minimal number of eager forks in the control network that are necessary for the following reasons:

8.1 Cycle Cutting

Lazy fork-join combinations can result in combinational cycles that cause oscillation or deadlock. These cycles can be avoided by replacing lazy forks with eager in places where cycles exist. Cycles can be easily identified either by hand analysis of the control network or through synthesis tools (e.g., `report_timing -loops` command in Design Compiler).

$LF01$ enjoys the property that there is no internal path in the fork that connects any of its branch Stalls to its corresponding Valid. This reduces the cycles substantially. Similarly, $LJ1011$ enjoys the property that there is no internal path in the join that connects any of its input channel Valid signals to its corresponding Stall. This also reduces the cycles substantially. Hence, the fork-join combination of $LF01-LJ1011$ results in the minimum number of combinational cycles among all the other fork-join combinations. This, in turn, minimizes the need to use eager forks to cut the cycles, resulting in minimizing the total area and power consumption of the hybrid control network.

8.2 Runtime Boosting

Eager forks can enjoy better performance than lazy due to the early start they provide for ready branches (Sect. 5.1). However, we show in this section that under some constrained input behavior, a lazy fork can replace an eager fork without any performance loss. In that context, we will use the term $LFork$ to refer to the lazy forks $LF00$ and/or $LF01$.

A 2-output $EFork$ operation will reduce to the KM of Fig. 15a if the $EFork$ registers are initialized to logic one and if the following input combinations are avoided [13]:

1. $(V_l = 1)\&(S_{r1} = 0)\&(S_{r2} = 1)$
2. $(V_l = 1)\&(S_{r1} = 1)\&(S_{r2} = 0)$

The KM of the lazy forks $LF00$ and $LF01$, with the above input combinations avoided, is shown in Fig. 15b. Comparing Fig. 15a and Fig. 15b, it is apparent that, under these conditions, the $EFork$ will behave exactly the same as the lazy forks, except in the case when both branches are stalled simultaneously. One might add a conservative constraint by avoiding such an input as well. However, as the following verification will confirm, when both branches are stalled, the lazy forks will have both branches in the Idle state, while the $EFork$ will keep them in the Retry state. Since there is no data transfer occurring in either states (i.e., I or R), there is no performance advantage of the $EFork$ comparing to the

V_l \ $S_{r1}S_{r2}$	00	01	11	10
0	0	0	0	0
1	1	-	1	-

(a) $EFork$

V_l \ $S_{r1}S_{r2}$	00	01	11	10
0	0	0	0	0
1	1	-	0	-

(b) $LFork$

Fig. 15. V_{r1} (or V_{r2}) of the $EFork$ and $LFork$ under some constrained input behavior, respectively.

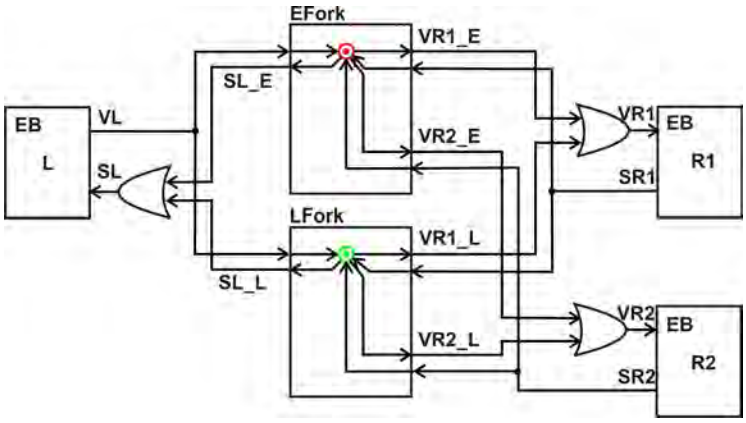


Fig. 16. Performance equivalence verification setup.

$LFork$ in such a case. Hence, we conclude that the above stated conditions are sufficient to replace an $EFork$ with $LF00$ or $LF01$ without any performance loss. We, therefore, refer to the above conditions as the *performance equivalence conditions*, or, for short, the *equivalence conditions*.

To verify this argument, the verification setup of Fig. 16 is employed. The whole structure is modeled in the symbolic model checker, NuSMV [4]. The input and output channels of both the $EFork$ and $LFork$ are connected to terminal Elastic Buffers (EBs). The EBs are initialized in random states. The $EFork$ input and two output channels are named: L_E (read $Left\_Eager$), $R1\_E$ (read $Right1\_Eager$), $R2\_E$ (read $Right2\_Eager$), respectively. Similarly, the $LFork$ input and 2 output channels are named: L_L, $R1\_L$, $R2\_L$, respectively. V and S are prepended to the channel names to indicate the Valid and Stall signals of these channels, respectively.

All the blocks as well as the clock generator have been connected synchronously inside NuSMV. The clock changes phase with each unit verification cycle. The Transfer state on the $EFork$ input and output channels are defined as follows:

```
DEFINE L_E_T := VL_E & !SL_E;
DEFINE R1_E_T := VR1_E & !SR1_E;
DEFINE R2_E_T := VR2_E & !SR2_E;
```

Similarly, for the LFork:

```
DEFINE L_L_T := VL_L & !SL_L;
DEFINE R1_L_T := VR1_L & !SR1_L;
DEFINE R2_L_T := VR2_L & !SR2_L;
```

A performance mismatch can occur if any of the channels in the $EFork$ transfers data while the corresponding channel in the $LFork$ does not. Hence, we define TOKEN_MISMATCH on the different channels as follows:

```
DEFINE L_TOKEN_MISMATCH := (L_E_T xor L_L_T);
DEFINE R1_TOKEN_MISMATCH := (R1_E_T xor R1_L_T);
DEFINE R2_TOKEN_MISMATCH := (R2_E_T xor R2_L_T);
```

A TOKEN_MISMATCH is defined to be the ORing of any channel mismatch:

```
DEFINE TOKEN_MISMATCH := L_TOKEN_MISMATCH | R1_TOKEN_MISMATCH |
R2_TOKEN_MISMATCH;
```

Finally, to force the performance equivalence conditions, we define the following constraint:

```
DEFINE C_1 := VL & (SR1 xor SR2);
```

Constraint C_1 is forced by using the NuSMV reserved word INVAR which semantically defines an invariant:

```
INVAR C_1;
```

The performance equivalence property is then verified using PSLSPEC:
`PSLSPEC never TOKEN_MISMATCH;`

The property is proven true by the model checker. There is no clock cycle in which any of the $EFork$ channels is in the Transfer state while the corresponding channel in the $LFork$ is not transferring data as well. Hence, under the stated performance equivalence conditions, the $EFork$ and $LFork$ will transfer exactly the same number of tokens, thus, achieving the same performance.

The results can be easily extended to n-output forks with $n > 2$, based on the fact that an n-output fork is logically equivalent to concatenated $(n-1)$ 2-output forks. Hence, all the eager forks in the control network that meet the performance equivalence conditions can be safely replaced by lazy forks. The result will be a hybrid control network (incorporating both eager and lazy forks) that has the same runtime of an *all* eager network with substantially smaller area and power.

8.3 Eager to Hybrid Conversion Flow

Algorithms to automatically identify which eager forks can be replaced by lazy in a network are currently being developed [13]. For the time being, simulation-based analysis is used. In this approach, a closed *eager* control network is simulated and all the fork Valid and Stall patterns are collected and analyzed. An example will be shown in the MiniMIPS case study in Sect. 9. Starting with an elastic control network (generated manually or through automatic tools like CNG [12]), the following flow generates a hybrid SELF implementation (H) of that network:

1. Define the set of all forks in the control network, Φ.
2. Construct a pure eager implementation of the control network, E_1, such that each fork $F \in \Phi$ is an eager fork. Define the set of forks, Φ_s, that do not meet the performance equivalence conditions. Φ_s are the forks that must be implemented as eager to achieve the same runtime as a purely eager implementation of the control network.
3. Construct an intermediate hybrid network, H_1, such that: each fork $F \in \Phi - \Phi_s$ is a lazy fork, and each fork $F \in \Phi_s$ is an eager fork.
4. In H_1, identify the set of forks, Φ_c, that need to be replaced by eager forks to cut the combinational cycles.
5. Build a final hybrid network,H, such that: each fork $F \in \Phi - \Phi_s - \Phi_c$ is lazy, and each $F \in \Phi_s \cup \Phi_c$ is eager.

9 MiniMIPS Case Study and Results

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture with 32 registers, first designed by Hennessey [6]. The MiniMIPS is an 8-bit subset of MIPS, fully described in [17].

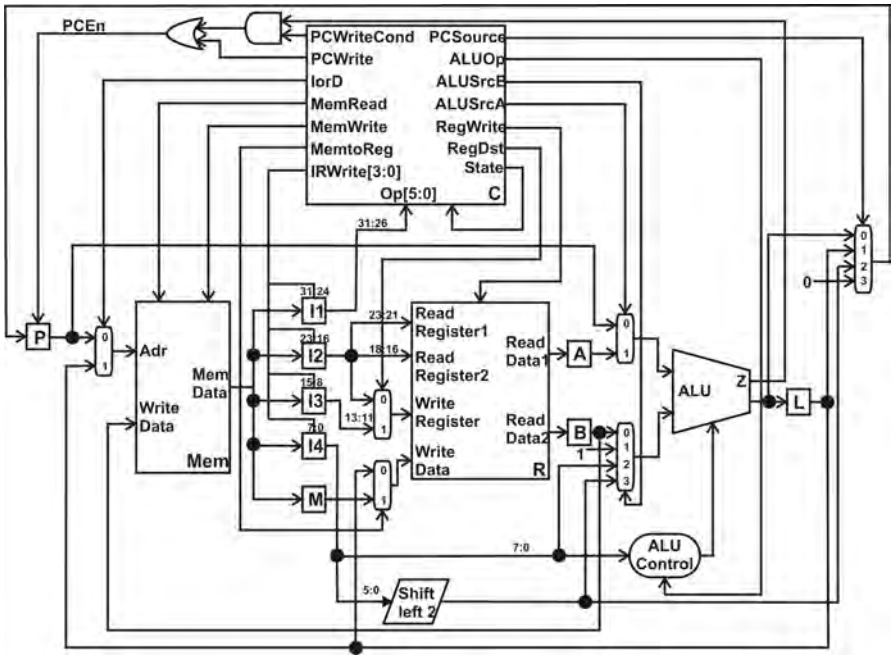


Fig. 17. Block diagram view of the ordinary clocked MiniMIPS. Adapted from [15, 17].

9.1 Elasticizing The MiniMIPS

The MiniMIPS is used as a case study of elasticization. Figure 17 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. 17: P (program counter), C (controller), $I1, I2, I3, I4$ (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 elastic buffer (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. For each register to register data communication there must be a corresponding control channel. The resultant control network can be implemented in different ways. Figure 18 shows a control network that has been hand-optimized to minimize the number of joins and forks used in the control network (to reduce area and power consumption) [11]. From the control point of view, the register file (R) and memory (Mem) in a microprocessor can be treated as combinational units [5]. Hence, we did not incorporate a separate EB for the register file (R) in Fig. 18. For the purpose of this case study, the memory (Mem) is off-chip.

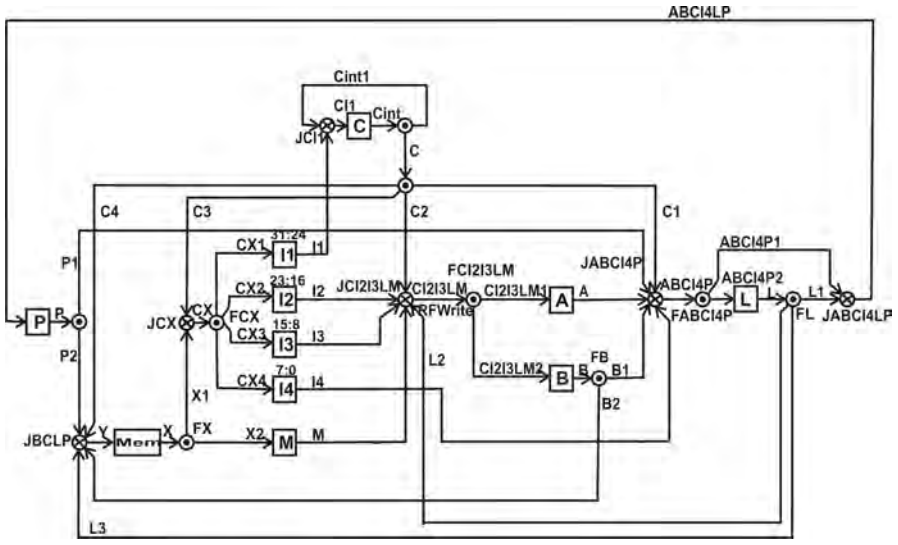


Fig. 18. Hand-optimized control network of the elastic clocked MiniMIPS. Adapted from [11].

From the elastic control point of view, the MiniMIPS control 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. 17, 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.

Both the clocked and purely eager elastic 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 [17]. The eager implementation of the SELF protocol has been used (EFork and LJ0000 have been used to implement all the forks and joins in the control network, respectively). Table 5 summarizes chip 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.

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

Table 5. Clocked and eager elastic MiniMIPS chip results. Measurements are done at 5V and 30°

	Clocked MiniMIPS	Eager 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%

both designs. Schmoo plots for both clocked and elastic MiniMIPS are shown in Fig. 19.

It should be noted that these measurements do not take advantage of bubble problems that occur if one needs to have flexible interface latencies or extra pipeline stages inserted.

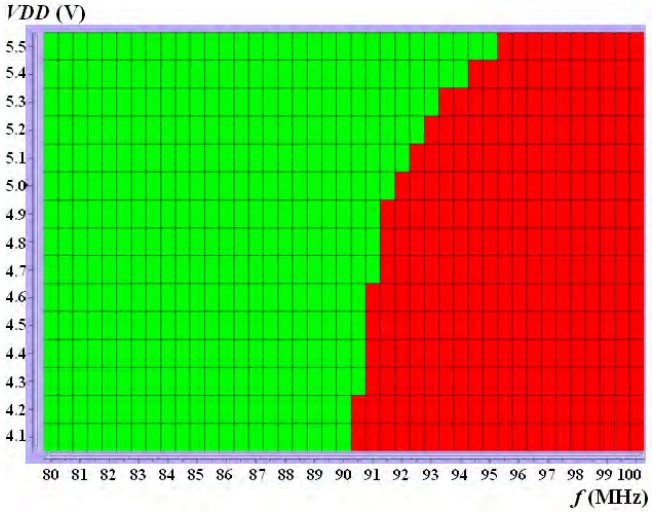
There is no performance loss due to elasticization. Part of the reason for the noticeable area and power overheads is that the MiniMIPS is, relatively, a small design (8-bit datapath). However, part of it too is the usage of eager forks. The $EFork$ has one flip-flop per each branch that consumes power every cycle. Add to this, its gate complexity. Next subsections will show the area and power savings when switching from eager SELF implementation to hybrid SELF.

9.2 Eager Versus Lazy SELF Implementations

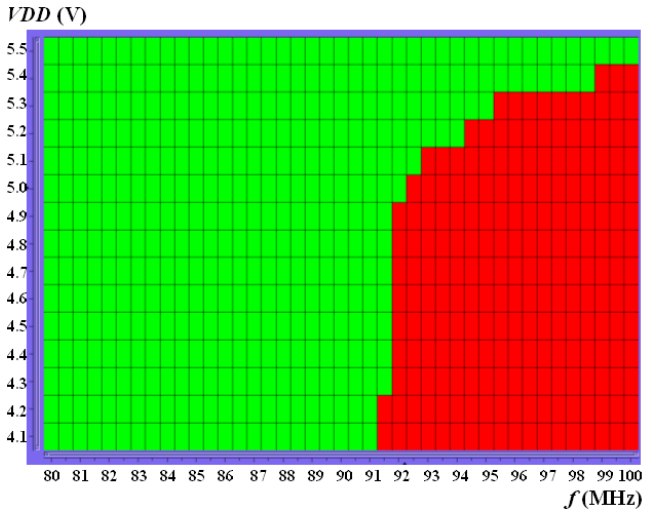
Lazy forks can substantially reduce the area and power of the elastic control network. However, when combined with lazy joins, the combinational cycles are typically prohibitive causing deadlocks or oscillations (Sect. 7).

Furthermore, lazy forks can suffer inferior performance comparing to eager, in the presence of bubbles. To measure this advantage, a different number of bubbles are inserted at the register file outputs (i.e., before registers A and B of Fig. 18, simultaneously). Table 6 compares the number of clock cycles required by each of purely lazy and eager implementations of the MiniMIPS control network to complete the testbench program of [17]. For the lazy protocol, the LF01-LJ0000 combination is used. The behavioral simulations used some timing constraints to avoid possible oscillations. Table 6 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 penalty due to elasticization in the absence of bubbles.

The runtime advantage of the eager versus lazy designs is illustrated in the following example (taken from the MiniMIPS control network of Fig. 18). Fig-



(a) Schmoo plot for clocked MiniMIPS.



(b) Schmoo plot for elastic MiniMIPS.

Fig. 19. Fabricated chips schmoo plots. Red boxes are for failed tests, while green are for passed ones.

ure 20 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 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

Table 6. Simulation runtime (in terms of #cycles) of the testbench in [17] 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: LF01-LJ0000	98	195	389
Eager Protocol: EFork-LJ0000	98	147	245
Clocked MiniMIPS	98	-	-

$b2$ 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.

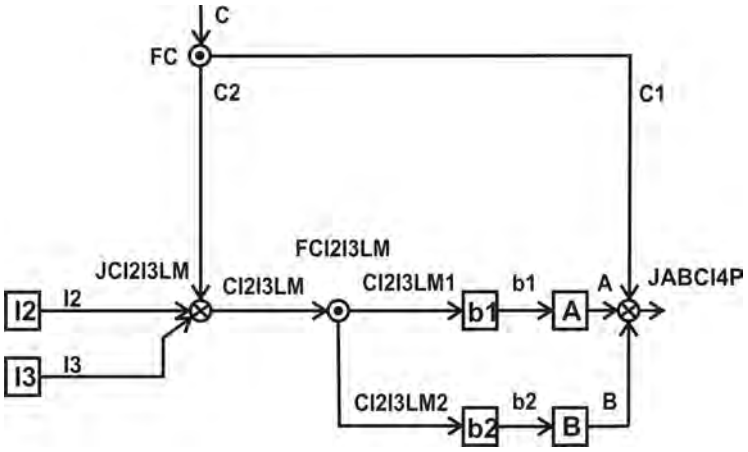


Fig. 20. A sample structure where eager protocol will have runtime advantage over lazy.

9.3 Eager Versus Hybrid SELF Implementations

The hybrid SELF implementation attempts to achieve the same performance of the eager SELF with less area and power consumption, by using as many lazy forks as possible. Without a loss of generality, we will apply both eager and hybrid implementations to the elastic MiniMIPS control network of Fig. 21. This control network achieves the same register-to-register communications as the one

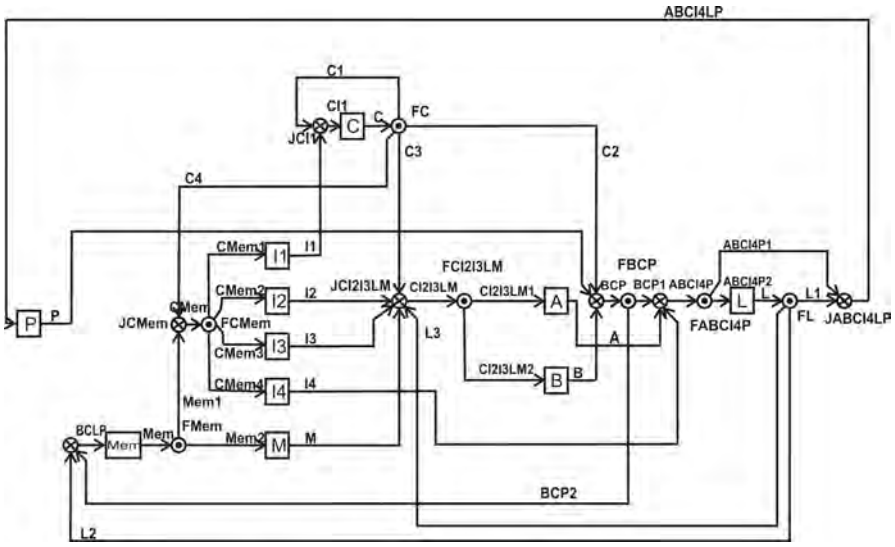


Fig. 21. *CNG*-optimized control network of the elastic clocked MiniMIPS [12].

in Fig. 18 but with two fewer joins and two fewer forks. It is auto-generated by the *CNG* tool, a tool that given the required register-to-register communications will automatically generate a control network with the minimum total number of joins and forks [12]. Furthermore, we insert zero to three bubbles (i.e., EBs that hold no valid data) at the register file output (i.e., at the inputs of A and B registers simultaneously). In practice, this might be done, for example, to accommodate a high latency register file without affecting the functionality of the whole system.

The flow of Sect. 8.3 will be followed to construct the hybrid implementation. Starting with an all eager implementation of the closed control network of Fig. 21 (call it E_1), we run the sample testbench program of [17]. The simulation waveform of each eager fork in the network is analyzed. EForks whose input behavior does not meet the performance equivalence conditions (of Sect. 8.2) are then identified. These are the forks that must be implemented as eager in the (to-be) hybrid control network in order to maintain the same performance as the all eager network. The set of these forks will be called ϕ_s.

Analysis of the simulation waveforms of the MiniMIPS case (with 0 to 3 bubbles at the register file output) shows that all forks except FC and FL receive Valid and Stall patterns that meet the performance equivalence conditions. Hence, all the forks except FC and FL can be safely implemented as lazy forks without any performance loss. For FC, we observe repetitive Stall patterns similar to those shown in Fig. 22. The numbered columns in Fig. 22 represent the clock cycles. The red 0s and 1s are the branch Stall signal values at the corresponding clock cycles. It is obvious that the Stall patterns at $C1$ and $C3$ meet the conditions of Sect. 8.2 (they do not stall at all). Hence, branches $C1$ and

$C3$ can be safely connected through a lazy fork (call it FC_1_3). Similarly, the Stall patterns at branches $C2$ and $C4$ meet the replacement conditions (their Stall patterns always match). Hence, branches $C2$ and $C4$ can also be connected through another lazy fork (call it FC_2_4). FC_1_3 and FC_2_4 should be connected through an eager fork since their corresponding Stall patterns do not match. The resultant hybrid FC implementation is shown in Fig. 23. EF and LF in the Figure refer to eager and lazy forks, respectively. Similarly, based on the simulation waveform analysis, branches 1 and 2 of FL could be connected through a lazy fork (FL_1_2). FL_1_2 must be connected eagerly to the third branch of FL to maintain the runtime of an *all* eager implementation.

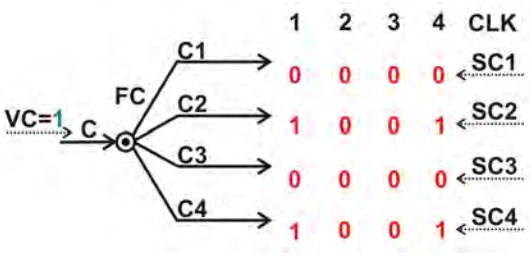


Fig. 22. Stall patterns at the branches of FC in the presence of bubbles.

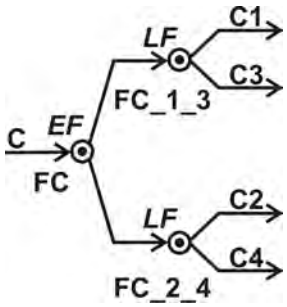


Fig. 23. Hybrid implementation of FC

As stated in Sect. 8.3, a hybrid network (call it H_1) is now constructed. All forks of H_1 are implemented as lazy except those in set (ϕ_s) (i.e., that do not meet the equivalence conditions). H_1 typically involves combinational cycles formed by the connection of lazy forks and joins. To cut the cycles in H_1, more forks have to be implemented as eager (call this set of forks ϕ_c). The number of forks in ϕ_c depend on the lazy fork and join combination used. Some lazy fork-join combinations exhibit more cycles than others and, hence, require more eager fork replacements. The MiniMIPS control network is implemented using all the correct 12 lazy fork-join combinations (with some eager fork replacements). The network is also implemented with an all eager control network.

The set of all forks that had to be implemented as eager (to both maintain the performance and cut the cycles) is listed in Column 2 for each combination in Table 7.

Table 7 shows the synthesis results. The Artisan academic library for IBM's 65nm library was used for physical design. The MiniMIPS control network has been synthesized separately from the data path. All area and power numbers in Table 7 are for *the control network only*. All combinations have passed post synthesis simulation (with 0 to 3 bubbles). The MiniMIPS testbench program in [17] was used to validate correctness. Column 1 in Table 7 lists the different combinations (sorted by their area). Column 2 lists the eager fork replacements

in each implementation. Unsurprisingly, $LF01 - LJ1011$ needs the least number of eager fork replacements (See Sect. 8.1), tying with $LF00 - LJ1011$ in this specific network. Column 3 lists the number of combination cycles in the control network (after eager fork replacements), which is zero for all of them. Column 4 lists the synthesis area. $LF00 - LJ1011$ requires minimum area among all with 31.8% reduction comparing to an all eager implementation. $LF01 - LJ1111$ comes second.

Column 5 lists the dynamic and leakage power consumption reported by the synthesis tool. Power is calculated with different number of bubbles inserted at the output of the register file. To accurately estimate the power, we simulated the synthesized netlist and generated an saif file that was read by the synthesis tool to calculate the power. Synthesis and simulation was done at 4 ns clock period for all the implementations. $LF00 - LJ1011$ consumes the least power among all with up to 32.5% and 32.1% dynamic and leakage power reduction comparing to an eager implementation. $LF01 - LJ1011$ comes second.

Finally, column 6 lists the required runtime (in terms of number of clock cycles) to finish the testbench program in [17]. The hybrid networks all achieve the same runtime as the eager implementation.

Table 7. Area, power, and runtime of the MiniMIPS control network using Different fork-join combinations.

Combination	Eager Forks Used	nCycles	Area	Power @ 4ns $\frac{P_{dyn}}{P_{leakage}}$ (μW)			Runtime (nCycles)		
			(μ^2)	0 B	1 B	3 B	0 B	1 B	3 B
$F00 - J1011$	*Some branches of FC, FL*	0	513.0	$\frac{58.187}{1.980}$	$\frac{164.284}{1.990}$	$\frac{122.720}{1.992}$	98	147	245
$F01 - J1111$	*FC, FL, FBCP*	0	575.4	$\frac{65.626}{2.339}$	$\frac{188.094}{2.307}$	$\frac{140.389}{2.278}$	98	147	245
$F01 - J1011$	*Some branches of FC, FL*	0	588.0	$\frac{58.187}{2.640}$	$\frac{183.991}{2.536}$	$\frac{134.636}{2.542}$	98	147	245
$F01 - J0000$	*FC, FL, FBCP*	0	634.2	$\frac{65.626}{2.739}$	$\frac{194.001}{2.663}$	$\frac{143.822}{2.599}$	98	147	245
$F00 - J1111$	*FC, FL, FBCP, FMem, FABCI4P*	0	639.0	$\frac{74.475}{2.525}$	$\frac{206.882}{2.514}$	$\frac{155.145}{2.499}$	98	147	245
$F01 - J0011$	*FC, FL, FBCP*	0	646.8	$\frac{65.626}{2.738}$	$\frac{192.545}{2.672}$	$\frac{143.065}{2.617}$	98	147	245
$F01 - J1010$	*FC, FL, FBCP*	0	649.8	$\frac{64.710}{2.761}$	$\frac{197.261}{2.691}$	$\frac{145.481}{2.631}$	98	147	245
$F01 - J0010$	*FC, FL, FBCP*	0	653.4	$\frac{65.635}{2.685}$	$\frac{191.208}{2.642}$	$\frac{142.149}{2.598}$	98	147	245
$F00 - J000$	*FC, FL, FBCP, FMem, FABCI4P*	0	683.4	$\frac{74.933}{2.825}$	$\frac{196.338}{2.762}$	$\frac{148.919}{2.713}$	98	147	245
$F00 - J0011$	*FC, FL, FBCP, FMem, FABCI4P*	0	695.4	$\frac{74.933}{2.790}$	$\frac{198.957}{2.742}$	$\frac{150.580}{2.699}$	98	147	245
$F00 - J0010$	*FC, FL, FBCP, FMem, FABCI4P*	0	698.4	$\frac{74.475}{2.853}$	$\frac{202.539}{2.838}$	$\frac{152.374}{2.811}$	98	147	245
$F00 - J1010$	*FC, FL, FBCP, FMem, FABCI4P*	0	704.4	$\frac{73.101}{2.887}$	$\frac{205.521}{2.867}$	$\frac{153.914}{2.844}$	98	147	245
$EFork - LJ0000$	*ALL*	0	752.4	$\frac{86.158}{2.914}$	$\frac{221.921}{2.875}$	$\frac{168.807}{2.842}$	98	147	245

10 Conclusion

Lazy implementations of fork and join control buffers of SELF latency insensitive protocol are implemented and formally verified. A novel hybrid SELF protocol network is introduced that combines the advantages of both eager and lazy elements. It is cycle free and has the same performance as an all eager implementation. A MiniMIPS case study showed that hybrid implementations achieve the same runtime as the *all* eager implementation with a reduction of 31.8% and up to 32.5% and 32.1% in area and dynamic and leakage power consumption, respectively.

Acknowledgments

The authors like to thank Shomit Das for his help in the place and route and layout of the 0.5 μm chips. This material is based upon work supported by the National Science Foundation under Grant No. 0810408.

References

[1] L. Carloni, K. Mcmillan, and A. L. Sangiovanni-VincentelliR. Theory of latency insensitive design. In *IEEE Transactions on CAD of Integrated Circuits and Systems*, volume 20, pages 1059–1076, Sept. 2001.

[2] L. Carloni and A. Sangiovanni-Vincentelli. Coping with latency in SOC design. *Micro, IEEE*, 22(5):24–35, Sept./Oct. 2002.

[3] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1437–1455, Oct. 2009.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proc. of 14th Conf. on Computer Aided Verification (CAV 2002)*, volume 2404, July 2002.

[5] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *ACM/IEEE Design Automation Conference*, pages 657–662, July 2006.

[6] J. H. et al. The MIPS Machine. In *COMPCON*, pages 2–7, 1982.

[7] D. Gebhardt and K. S. Stevens. Elastic flow in an application specific network-on-chip. *Elsevier Electronic Notes in Theoretical Computer Science*, 200(1):3–15, Feb. 2008.

[8] A. Gotmanov, M. Kishinevsky, and M. Galceran-Oms. Evaluation of flexible latencies: designing synchronous elastic H.264 CABAC decoder. In *The Problems in design of micro- and nano-electronic systems*, 2010.

[9] G. Hoover and F. Brewer. Synthesizing synchronous elastic flow networks. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 306 –311, 10-14 2008.

[10] 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*, pages 3–12, Apr. 2002.

[11] E. Kilada, S. Das, and K. Stevens. Synchronous elasticization: considerations for correct implementation and MiniMIPS case study. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, pages 7 –12, Sept. 2010.

[12] E. Kilada and K. Stevens. Control network generator for latency insensitive designs. In *Design, Automation & Test in Europe Conference Exhibition (DATE), 2010*, pages 1773 –1778, March 2010.

[13] E. Kilada and K. S. Stevens. Synchronous elasticization at a reduced cost: utilizing the ultra simple fork and controller merging. In *International Conference on Computer-Aided Design (ICCAD-11)*, Nov. 2011.

[14] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, pages 19–30, Nov. 2006.

[15] D. Patterson and J. Hennessy. *Computer Organization and Design*. 2004.

[16] V. Vakilotojar and P. Beerel. RTL verification of timed asynchronous and heterogeneous systems using symbolic model checking. In *Design Automation Conference 1997. Proceedings of the ASP-DAC '97. Asia and South Pacific*, pages 181 –188, 28-31 1997.

[17] N. Weste and D. Harris. *CMOS VLSI design: a circuit and systems perspective*. 2004.