Control Network Generator For Latency Insensitive Designs

Eliyah Kilada, Kenneth S. Stevens
University of Utah
Eliyah.Kilada@utah.edu, kstevens@ece.utah.edu

***Abstract*—Creating latency insensitive or asynchronous designs from clocked designs has potential benefits of increased modularity and robustness to variations. Several transformations have been suggested in the literature and each of these require a handshake control network (examples include synchronous elasticization and desynchronization). Numerous implementations of the control network are possible. This paper reports on an algorithm that has been proven to generate an optimal control network consisting of the minimum number of 2-input join and 2-output fork control components. This can substantially reduce the area and power consumption of a system. The algorithm has been implemented in a CAD tool, called CNG. It has been applied to the MiniMIPS processor showing a 14% reduction in the number of control steering units over a hand optimized design in a contemporary work.**

I. Introduction

A great attention has been recently given to latency-insensitive (LI) designs. LI designs, among other benefits, allow for correct operation independently of channel latencies [1]. This, in turn, facilitates handling any channel delay variations (e.g., due to routing or technology migration), that are typically hard to estimate until the layout in the target technology is complete. This is done by increasing the channel latency without affecting the whole system functionality. This change, for non-LI designs, would require severe manual changes in the system to accommodate the new delays and, possibly, a number of iterations [2], [3]. Synchronous elasticization [2], [4], [5] and desynchronization [6], [7] have been recognized as two promising approaches of transforming an ordinary clocked system into an LI one. A typical first step in both of these approaches is to replace each flip flop in the original design with an LI synchronizing module with a controller [8]. Following this step, communications among registers of the original design are analyzed (i.e., for each register, which registers do send and receive data to and from it). For each register-to-register data communication there must be a corresponding LI control channel, to control data flow between these two registers. This forms a network of control channels. Joins and forks are used to join control channels targeting the same destination registers, and to fork control channels originating from the same source registers, respectively.

The control network can be constructed in many different ways. This paper provides an algorithm, and a CAD tool, that automatically generates a control network with minimum total number of 2-input joins and 2-output forks. This can substantially reduce power and area in the control network. The algorithm has been also applied to the MiniMIPS processor showing a 14% reduction in the number of control steering units over a hand optimized design in a contemporary work [4].

II. Problem Definition

Example 1. *Let A, B, C, D be four registers in the original ordinary clocked design. Both registers A and B pass data to both registers C and D. Find a control network implementation for the LI version of this design.*

Figures 1a and 1b are two example implementations for such a control network. The control network in Fig. 1b has one less join and one less fork component than the network of Fig. 1a. Things get more complicated when the number of registers and their corresponding communications increase. Hence, the purpose of the proposed algorithm is, given a set of required register-to-register communications, the algorithm should automatically generate a control network with minimum total number of 2-input join and 2-output fork components.

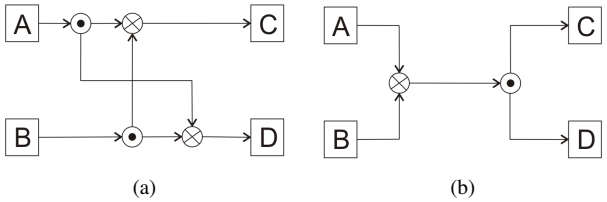

Fig. 1. Two possible implementations of Example 1. Fork and join components are represented by $\odot$ and $\otimes$, respectively.

In this section we list number of definitions required to formalize the problem. Example 2 will be used as a running example throughout the paper.

Example 2. *Let $A, B, C, D, E, F, G, X_1, X_2, X_3, X_4, X_5$ be twelve registers in the original ordinary clocked design. The following registers pass data to $X_1 : B, C, G$, and to $X_2 : A, B, C, G$, and to $X_3 : A, B, C, D, E$, and to $X_4 : A, B, D, E, F$, and to $X_5 : A, B, E, F$. Find a control network implementation for the LI version of this design, that incorporates minimum number of joins and forks components.*

In a register-to-register communication, we call the data transmitting register, a *source*, and the data receiving register, a *destination*. In Example 2, the following registers are sources: A, B, C, D, E, F, G, and the following are destinations: X_1, X_2, X_3, X_4, X_5. The vector of all sources and the vector of all destinations in the network are designated as $SourceS$ and $DestinationS$, respectively. A component joining two or more control channels into one channel is called a $join$. A component forking a control channel into two or more control channels is called a $fork$. Throughout this paper we assume that n-input (n-output) join (fork) is implemented by concatenating $(n-1)$ 2-input (2-output) join (fork) components. 2-input join and 2-output fork components will be designated as $Join2$ (or $J2$) and $Fork2$ (or $F2$), respectively.

Definition 1. Term *A set of one or more source registers.*

Constructing a term typically means joining the control channels coming from these source registers into one control channel. Each term has a unique identifier, $TermID$. As an example, a term that joins the control channels coming from: B, D, E, is $\{B, D, E\}$ and, for simplicity, will be referred to as BDE. $|Term_1|$ designates the cardinality of $Term_1$. A term that is composed of only one source register, is called *single letter term* or *SLTerm*. The vector of all single

letter terms (or all source registers) is designated as *SLTermS*.

Definition 2. Target *A term that is associated with a destination register. A $target$ of a certain destination register is a term composed of all source registers that send data to that destination.*

In Example 2, BCG is the target term associated with X_1. The vector of all target terms is designated as *TargetS*.

Definition 3. Common term or CTerm *A term that is the intersection of two or more targets. Formally, a term, $CTerm_1$ is a CTerm iff $CTerm_1 = \bigcap_{i=1}^{n} Target_i$ for any $1 < n \leq |TargetS|$.*

Cardinality of a $CTerm$ must be greater than one. In Example 2, $Term_1 = BCG$ would be a $CTerm$ since it is the intersection of the target terms associated with X_1 (i.e., BCG) and X_2 (i.e., $ABCG$). We define $CTermS$ to be the vector of all common terms, ordered by their cardinalities from the largest to the smallest.

Definition 4. Potential terms or PTermS *A vector composed of the concatenation of CTermS and SLTermS, respectively.*

Finally, *TermS* is the concatenation of *TargetS* and *PTermS*, respectively. *TermS* of Example 2 is listed in Table I.

Definition 5. Partial Solution or PS *A set of terms that could be used to implement a certain term. Formally, PS_t is a partial solution of a term, $Term_t$, iff $\forall\ Term_i \in PS_t$ where $i = 1, 2, .., |PS_t|$; $Term_t = \bigcup_{i=1}^{|PS_t|} Term_i$*

PS_t represents one way of constructing $Term_t$. One term could be constructed in multiple ways, and thus has more than one PS. In Example 2, to construct $Term_t = ABCDE$, one possible PS, is $\{ABC, D, E\}$. Another, is $\{ABDE, C\}$.

Definition 6. Solution or Soln *A vector of PS's, where $TermID$'s are used as indices. If $Soln_1$ is a solution, and $Term_tID$ is the $TermID$ of $Term_t$, then $Soln_1[Term_tID]$ is the chosen PS to construct $Term_t$ in $Soln_1$.*

For brevity, $Soln_1[Term_tID]$ and $Soln_1[Term_t]$ will be used interchangeably. In Example 2, the following is a possible solution (Terms are sorted by their $TermID$'s of Table I, and $SLTerms$ PS's are ignored):

$$\begin{aligned} Soln_1 = < &\{BCG\}, \{BCG, A\}, \{ABDE, C\}, \{ABDE, F\}, \\ &\{ABEF\}, \{ABE, D\}, \{ABE, F\}, \{BC, G\}, \\ &\{AB, C\}, \{AB, E\}, \{B, C\}, \{A, B\} > \end{aligned} \quad (1)$$

Hence, solution, $Soln_1$, can be seen as a vector of PS choices of the different terms used in $Soln_1$. For example, $Soln_1[2] = \{BCG, A\}$. This means the $PS = \{BCG, A\}$ is used in $Soln_1$ to construct term $ABCG$ (whose $TermID$ is 2). $Soln_1$ is depicted in Fig. 2.

Definition 7. nUsed $nUsed[Term_i]$ *defines how many times $Term_i$ is used to construct other useful terms in a certain solution. Formally, $nUsed[Term_i]$ in a solution, $Soln_1$, is defined recursively to be the number of terms, $Term_t$, that satisfy the following two conditions:*

1) *$Term_i \in PS_t$ where $Soln_1[Term_t] = PS_t$.*
2) *$nUsed[Term_t] > 0$ in $Soln_1$.*

$nUsed$ of all targets are 1.

Definition 8. Useful Term *$Term_i$ is said to be useful in $Soln_1$ (or $Soln_1$ uses $Term_i$), iff $nUsed[Term_i] > 0$ in $Soln_1$.*

For Example 2 and $Soln_1$ of Eq. 1: term ABE (with $TermID$ of 10) is used to construct both terms $ABDE$ (with $TermID$ of 6) and

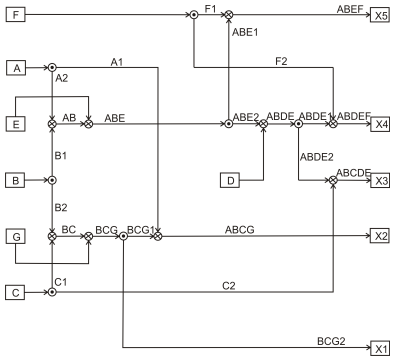

Fig. 2. A sample control network of Example 2.

$ABEF$ (with $TermID$ of 7). Hence, $nUsed[ABE]$ in $Soln_1$ is 2. Also, term ABC (with $TermID$ of 9) is not useful in $Soln_1$. Term AB (with $TermID$ of 12) is used to construct both terms ABC (with $TermID$ of 9) and ABE (with $TermID$ of 10). However, since term ABC is not useful in $Soln_1$, therefore, $nUsed[AB]$ in $Soln_1$ is only 1.

Definition 9. Usable Term *Usability is defined recursively as follows: $Term_i$ is usable iff it belongs to, at least, one PS of one usable term. All targets are usable.*

In other words, a term is usable iff it is useful in at least one solution. A term can be usable, but not necessarily useful in some solutions.

Definition 10. Cost *A function that returns the number of 2-input joins ($J2$'s) required to implement a term or a $Soln$.*

Formally, let PS_t be the PS of term, $Term_t$, in $Soln$, $Soln_1$, then $Cost(Term_t)$ in $Soln_1$ is defined as follows:

$$Cost(Term_t) = |PS_t| - 1 + \sum_{i=1}^{|PS_t|} \frac{Cost(Term_i)}{nUsed[Term_i]} \quad (2)$$

where $\forall i = 1, 2, ..|PS_t|, Term_i \in PS_t$. $Cost(Term_t)$ in $Soln_1$ and $Cost(PS_t)$ in $Soln_1$ will be used interchangeably (since $Soln_1[Term_t] = PS_t$). Two factors contribute to $Cost(Term_t)$. First, the number of $J2$'s used to join PS_t's constituent terms. We assume, in Eq. 2, that, to implement an n-input join, $(n-1)J2$'s are required. The other factor is the cost of the constituent terms themselves. $Cost(Term_t)$, in a solution, takes into account how much the constituent terms are shared among other terms in that solution. This information is provided by $nUsed$ vector. Cost of all $SLTerms$ are zero.

For Example 2 and $Soln_1$ of Eq. 1: the chosen PS to construct term ABE (with $TermID$ of 10) is $\{AB, E\}$, $nUsed[AB] = 1$. Hence, $Cost(ABE) = 1 + Cost(AB)$. The chosen PS to construct term AB (with $TermID$ of 12) is $\{A, B\}$, and, hence, $Cost(AB) = 1$. Therefore, $Cost(ABE)$ in $Soln_1$ is 2. Similarly, $Cost(ABDE) = 2$.

Similarly, we also define $Cost(Soln_1)$ to be the total number of $J2$'s used to construct all the targets in $Soln_1$. Formally,

$$Cost(Soln_1) = \sum_{i=1}^{|TargetS|} Cost(Target_i) \quad (3)$$

where $\forall i = 1, 2, ..|TargetS|, Target_i \in TargetS$.

For Example 2 and $Soln_1$ of Eq. 1: we have five targets (those that are associated with the five destinations), namely, BCG, $ABCG$,

$ABCDE$, $ABDEF$, $ABEF$. The summation of the costs of these targets in $Soln_1$ (i.e., $Cost(Soln_1)$ is 9.

Definition 11. Optimum Solution or OptSoln *A solution with minimum Cost among all other solutions.*

Definition 12. nAJ(Term) *If PS_t is the chosen PS of $Term_t$ in $Soln_1$, then, $nAJ(Term_t)$ is the number of $J2$'s added to the network just to construct $Term_t$ (i.e., $J2$'s that would not have been in the network, if $Term_t$ has not been used in $Soln_1$). Formally,*

$$nAJ(Term_t) = |PS_t| - 1 + \sum_{i=1}^{|PS_t|} nAJ(Term_i) \times s_i \quad (4)$$

where $\forall i = 1, 2, ..|PS_t|$, $Term_i \in PS_t$ and $s_i = 1$ if $nUsed[Term_i] = 1$ and $s_i = 0$ if $nUsed[Term_i] > 1$

$nAJ(Term_t)$ in $Soln_1$ and $nAJ(PS_t)$ in $Soln_1$ will be used interchangeably (since $Soln_1[Term_t] = PS_t$). As an example, let all the terms used by PS_t be already shared by other terms in $Soln_1$. In this case, all that is added to the network to construct PS_t is the $J2$'s required to join its constituent terms (i.e., $|PS_t| - 1$).

For Example 2 and $Soln_1$ of Eq. 1: $nAJ(AB) = 1$, $nAJ(ABE) = 2$, $nUsed[ABE] = 2$. Although the $Cost$ of $ABDE$ is 2, its nAJ is only 1. The reason is, term ABE which is used to construct $ABDE$ in $Soln_1$ is also used in the solution to construct another term (i.e., term $ABEF$ with $TermID$ of 7). Hence, to construct term $ABDE$, the only added $J2$ to $Soln_1$ is the join required to join ABE with D.

III. Algorithm

Lemma 1. *If $nJ2$ and $nF2$ are the total number of $J2$'s and $F2$'s in the network, respectively. Then, whatever the PS choices of different terms used in a solution, the following equality holds: $nJ2 - nF2 = |SourceS| - |DestinationS|$*

Proof: Proof is omitted due to space limitation. ∎

Theorem 2. *An algorithm that minimizes $nJ2$ will also minimize $nF2$ and also $nJ2 + nF2$.*

In other words, $OptSoln$ defined in Def. 11 will incorporate minimum total number of $J2$'s and $F2$'s in the network.

Proof: The theorem follows directly from Lemma 1. ∎

Theorem 2 narrows down the problem to: Given a set of source registers and a set of targets, the algorithm aims to construct the targets by using minimum total number of $J2$'s (i.e., to find $OptSoln$). The proposed algorithm consists of four main steps, covered in the following four subsections. The first step (covered in Subsection III-A) is to find the candidate terms that can be used in $OptSoln$. Then, for each of the candidate terms, the algorithm finds the candidate PS's that may be used by $OptSoln$. This step is explained in Subsection III-B. At this point the search space of the problem consists of all the possible PS choices of all the candidate terms. In step 3 (covered in Subsection III-C), the algorithm does a number of iterations. At the end of each iteration, it eliminates part of the search space. When the algorithm can do no more elimination, it goes to step 4. Step 4 (covered in Subsection III-D) does a final possible reduction in the remaining search space. It then calculates the $Cost$ of the remaining solutions and returns $OptSoln$.

A. Construct Potential Terms

Theorem 3. (Potential Terms) *An optimum solution can be found by using only potential terms (or $PTermS$, Def. 4) .*

Proof: Proof is omitted due to space limitation. ∎

The first step in the algorithm is to determine which terms could be used to construct the targets. It tries to exclude terms that are guaranteed not to be used in $OptSoln$. Theorem 3 narrows down the search space, by confining the candidate terms to $PTermS$. $PTermS$ are constructed following their definition in Def. 4.

B. Construct Partial Solutions

The search space (i.e, the possible solutions), at this point, consists of all combinations of all possible PS choices of all $PtermS$. This step aims at excluding PS's that will not be needed in $OptSoln$. A cost metric must be used that can differentiate between two PS's of a certain term without actual search space exploration. Following are a set of theorems that guide this step.

Theorem 4. (nAJ(PS)) *Let $Soln_1$ and $Soln_2$ be two solutions. Let also, $\forall(i = 1, 2, ..|TermS| \wedge i \neq t)$: $Soln_1[Term_i] = Soln_2[Term_i]$, $Soln_1[Term_t] = PS_{t1}$ and $Soln_2[Term_t] = PS_{t2}$. Then, if ($(nAJ(PS_{t1})$ in $Soln_1) \geq (nAJ(PS_{t2})$ in $Soln_2)$), then $Cost(Soln_1) \geq Cost(Soln_2)$. Greater and equal operators are ordered respectively.*

Proof: Proof is omitted due to space limitation. ∎

Corollary 5. *Let PS_1 and PS_2 be two PS's of $Term_t$. Then, if, for all possible combinations of other terms' PS's choices, $nAJ(PS_1) > nAJ(PS_2)$ then $OptSoln$ will not use PS_1.*

Corollary 6. *Let PS_1 and PS_2 be two PS's of $Term_t$. Then, if, for all possible combinations of other terms' PS's choices, $nAJ(PS_1) \geq nAJ(PS_2)$ then $OptSoln$ can be found that doesn't use PS_1.*

Proof of both Corollaries 5 and 6 follows from Theorem 4 as well as Definitions 10 and 11.

Theorem 7. (Cost(PS)) *Let $Soln_1$ and $Soln_2$ be two solutions. Let also, $\forall(i = 1, 2, ..|TermS| \wedge i \neq t)$: $Soln_1[Term_i] = Soln_2[Term_i]$, $Soln_1[Term_t] = PS_{t1}$ and $Soln_2[Term_t] = PS_{t2}$. Then, if ($(Cost(PS_{t1})$ in $Soln_1) \geq (Cost(PS_{t2})$ in $Soln_2)$), then the following inequality does not necessarily hold: $Cost(Soln_1) \geq Cost(Soln_2)$.*

Proof: Proof and counter example are omitted due to space limitation. ∎

Based on theorems 4 through 7, nAJ (rather than $Cost$) could be used to differentiate between two PS's of a certain term without actual search space exploration. The following three theorems, set a list of rules that should be considered while constructing the PS's of the terms in $PTermS$.

Theorem 8. (Rule 1) *Adding a whole redundant term to a PS always causes it to be more expensive (in terms of nAJ). Formally, let $Term_t, Term_1, Term_2 \in TermS$, $Term_2 \subseteq Term_1$, and $Term_1 \subseteq Term_t$. Let PS_1 and PS_2 be two PS's of $Term_t$. Let both PS_1 and PS_2 be the same except that PS_1 contains one $Term_1$, while PS_2 contains one $Term_1$ and one $Term_2$. Then, an optimum solution will not use PS_2.*

Proof: Proof is omitted due to space limitation. ∎

Consider term $ABCG$ in Example 2. $PS_1 = \{A, BCG\}$ is always cheaper than $PS_2 = \{A, BCG, BC\}$. Hence, PS_2 should be excluded from the search space.

Theorem 9. (Rule 2) *While searching for $OptSoln$, using a term in a PS is always the same or cheaper (in terms of nAJ) than using all its*

TABLE I
TERMS AND PS'S OF EXAMPLE 2

TermID	Term	Type [1]	PSID	PS	Initial nUsed Max	Initial nUsed Min
1	BCG	T	1	$\{BCG\}$	1	1
2	$ABCG$	T	1 2	$\{BCG, A\}$ $\{ABC, G\}$	1	1
3	$ABCDE$	T	1 2	$\{ABDE, C\}$ $\{ABC, D, E\}$	1	1
4	$ABDEF$	T	1 2	$\{ABDE, F\}$ $\{ABEF, D\}$	1	1
5	$ABEF$	T	1	$\{ABEF\}$	1	1
6	$ABDE$	C, P	1	$\{ABE, D\}$	2	0
7	$ABEF$	C, P	1	$\{ABE, F\}$	2	1
8	BCG	C, P	1	$\{BC, G\}$	2	1
9	ABC	C, P	1 2	$\{BC, A\}$ $\{AB, C\}$	2	0
10	ABE	C, P	1	$\{AB, E\}$	2	1
11	BC	C, P	1	$\{B, C\}$	2	1
12	AB	C, P	1	$\{A, B\}$	2	1
13-19	$A-G$	S, P	1			

constituent terms. Formally, let $Term_t, Term_1, Term_2, Term_3 \in TermS$, $Term1 \subseteq Term_t$, *and* $Term_1 = Term_2 \cup Term_3$. *Let* PS_1 *and* PS_2 *be two* PS*'s of* $Term_t$. *Let both* PS_1 *and* PS_2 *be the same except that* PS_1 *contains one* $Term_1$, *while* PS_2 *contains one* $Term_2$ *and one* $Term_3$. *Then, an optimum solution can be found that doesn't use* PS_2.

Proof: Proof is omitted due to space limitation. ∎

Consider term $ABCG$ in Example 2. While searching for $OptSoln$, $PS_1 = \{A, BCG\}$ is always the same or cheaper than $PS_2 = \{A, BC, G\}$. Hence, PS_2 could be excluded from the search space.

Theorem 10. (Rule 3) *An* $SLTerm$ *is always the same or cheaper (in terms of* nAJ*) than any other* $non-SLTerm$. *Let* $Term_t, Term_1, Term_3 \in TermS$ *and* $\notin SLTermS$, *and* $SLTerm_2 \in SLTermS$. *Let also* $Term_1, SLTerm_2, Term_3 \subseteq Term_t$. *Let* PS_1 *and* PS_2 *be two* PS*'s of* $Term_t$. *Let both* PS_1 *and* PS_2 *be the same except that* PS_1 *contains one* $SLTerm_2$, *while* PS_2 *contains one* $Term_3$. *Then,* $OptSoln$, *can be found that doesn't use* PS_2.

Proof: Proof is omitted due to space limitation. ∎

Consider term $ABCG$ in Example 2. $PS_1 = \{BCG, A\}$ is always the same or cheaper than $PS_2 = \{BCG, AB\}$. Hence PS_2 should be excluded from the search space.

An algorithm has been developed, that takes into account all the three rules while constructing PS's of the terms. The algorithm has been omitted due to space limitation. Table I shows the PS's of Example 2 after applying the three rules.

C. Update Cost Structures And Remove Higher nAJ Partial Solutions

Theorem 3 narrowed down the search space by confining the number of candidate terms. Furthermore, Theorems 4 through 10 reduced their possible corresponding PS's. At this point the search space of the problem consists of all the remaining possible PS choices of all the candidate terms. This step aims at pruning out the search space, even more, through a number of iterations. At the end of each iteration, more information about the cost of the different possible solutions, as well as $OptSoln$, is revealed and, hence, more areas of the search space can be eliminated. When the algorithm can do no more eliminations, it goes to the next step.

Although, the value of $nAJ(Term_t)$ in a certain solution, $Soln_1$, (or, equivalently, the value of $nAJ(PS_t)$ in $Soln_1$, where $Soln_1[Term_t] = PS_t$), depends on the PS's choices of the other terms (which we call, hereafter, the environment). And, the search space, at this point, allows for all PS's choices of the environment - which allows $nAJ(PS_t)$ to take numerous values. However, we can still utilize some information we know about the environment to deterministically differentiate between two PS's of a certain term based on their nAJ. The type of information needed is supplied by $nUsedMax$ and $nUsedMin$ which are defined, along with other related concepts, as follows:

Definition 13. nUsedMax *A vector of numbers, where* $TermID$*'s are used as indices.* $nUsedMax[Term_i]$ *is the number of usable terms that may use* $Term_i$. *Formally,* $nUsedMax[Term_i]$ *is the number of* $Term_t$*'s that satisfy the following two conditions:*

1) $Term_i \in PS_t$ *where* PS_t *is a* PS *of* $Term_t$.
2) $Term_t$ *is usable.*

$nUsedMax[Target_i] = 1 \ \forall \ Target_i \in TargetS$.

Table I shows the initial values of $nUsedMax$ of different terms in Example 2. At the end of each iteration, some PS's are omitted from the search space, and, hence, the value of $nUsedMax$ of some terms will be decreased.

Definition 14. Essential Term or ETerm $Term_t$ *is essential term (or* $ETerm$*) iff it is useful in* $OptSoln$.

All targets are $ETerms$.

Definition 15. Essential Child or EChild $Term_i$ *is said to be an essential child of* $Term_t$ *iff all the following conditions are satisfied:*

1) $Term_i$ *belongs to all* PS*'s of* $Term_t$.
2) $Term_t$ *is usable.*

We also define $EChildren[Term_t]$ to be all $EChild$ terms of $Term_t$.

Definition 16. nUsedMin *A vector of numbers, where* $TermID$*'s are used as indices.* $nUsedMin[Term_i]$ *is the number of useful terms in* $OptSoln$ *that are guaranteed to use* $Term_i$. *Formally,* $nUsedMin[Term_i]$ *is the number of* $Term_t$*'s that satisfy the following two conditions:*

1) $Term_i \in PS_t$ *where* $OptSoln[Term_t] = PS_t$.
2) $Term_t$ *is useful in* $OptSoln$.

$nUsedMin[Target_i] = 1 \ \forall \ Target_i \in TargetS$.

The calculation of $nUsedMin$ is initialized by having $nUsedMin[Target_i] = 1 \ \forall \ Target_i \in TargetS$. Then, propagation of *usefulness* in $OptSoln$ can take place. If $Term_t$ is useful in $OptSoln$, then all its $EChildren$ will also be useful in $OptSoln$. An algorithm that automates this process has been omitted due to space limitation.

Table I shows the initial values of $nUsedMin$ of different terms in Example 2. At the end of each iteration, more information about $OptSoln$ are revealed, and, hence, the value of $nUsedMin$ of some terms will be increased.

Definition 17. nAJMax(PS) $nAJMax(PS_t)$ *is the value of* $nAJ(PS_t)$ *when the environment provides minimum sharing to the terms used by* PS_t. *Formally, let* PS_t *be a* PS *of* $Term_t$, *then*

[1] T is for $Target$, C for $CTerm$, P for $PTerm$ and S for $SLTerm$.

$nAJMax(PS_t)$ *is defined as follows:*

$$nAJMax(PS_t) = |PS_t| - 1 + \sum_{i=1}^{|PS_t|} nAJMax(Term_i) \times s_i \quad (5)$$

where $\forall i = 1, 2, ..|PS_t|$, $Term_i \in PS_t$ *and* $s_i = 1$ *if* $x_i = 1$ *and* $s_i = 0$ *if* $x_i > 1$, *where* $x_i = 1 +$ *the number of* $Term_{t1}$*'s (where* $Term_{t1} \neq Term_t$*) that satisfy the following two conditions: First,* $Term_i \in PS_{t1}$ *where* $OptSoln[Term_{t1}] = PS_{t1}$. *Second,* $Term_{t1}$ *is useful in OptSoln.*

It is a trivial task to compute x_i in Def. 17 from $nUsedMin$.

Definition 18. nAJMin(PS) $nAJMin(PS_t)$ *is the value of* $nAJ(PS_t)$ *when the environment provides maximum sharing to the terms used by* PS_t. *Formally, let* PS_t *be a PS of* $Term_t$, *then* $nAJMin(PS_t)$ *is defined as follows:*

$$nAJMin(PS_t) = |PS_t| - 1 + \sum_{i=1}^{|PS_t|} nAJMin(Term_i) \times s_i \quad (6)$$

where $s_i = 1$ *if* $nUsedMax[Term_i] = 1$ *and* $s_i = 0$ *if* $nUsedMax[Term_i] > 1$

Definition 19. nAJMax/Min(Term) *Let* $PSs[Term_t]$ *be a vector of the* PS*'s of* $Term_t$. *Then,*

$$nAJMax(Term_t) = \max_{i=1}^{|PSs[Term_t]|} nAJMax(PS_{ti})$$
$$nAJMin(Term_t) = \min_{i=1}^{|PSs[Term_t]|} nAJMin(PS_{ti}) \quad (7)$$

A more restricted condition, yet easier to check, of Corollaries 5 and 6 is stated in the following corollaries:

Corollary 11. *Let* PS_1 *and* PS_2 *be two* PS*'s of* $Term_t$. *Then, if,* $nAJMin(PS_1) > nAJMax(PS_2)$ *then OptSoln will not use* PS_1.

Corollary 12. *Let* PS_1 *and* PS_2 *be two* PS*'s of* $Term_t$. *Then, if,* $nAJMin(PS_1) \geq nAJMax(PS_2)$ *then OptSoln can be found that doesn't use* PS_1.

Algorithm 1 makes use of Corollaries 11 and 12, along with $nUsedMax$ and $nUsedMin$ structures to, iteratively, refine the search space. It incorporates the following data structures:

- $nAJMin/Max[Term_t]$: A vector that stores $nAJMin/Max$ of all $Term_t$, respectively.
- $PSnAJMin/Max[Term_t][PS_{ti}]$: A two dimensional structure that stores $nAJMin/Max$ of all PS_{ti} of all $Term_t$, respectively.
- *UT*: a set of terms whose $nAJMax$ and/or $nAJMin$ need to be updated. The terms are ordered within the set by their cardinalities starting by the largest to the smallest. UT is initialized with all targets and common terms.
- $UPSMin/Max[Term_t]$: a set of PS's of $Term_t$ whose $nAJMin/Max$ need to be updated, respectively. They are initialized with all PS's of $Term_t$.
- *PSR*: a set of PS's that are scheduled to be removed from the search space.

Algorithm 1 starts with UT initialized with all targets and common terms. Line 2 picks the smallest term in UT, $Term_t$. Lines 4 and 5 store the old values of $Term_t$'s $nAJMax, nAJMin, EChildren$

Algorithm 1 Update nAJ Structures And Remove PS's of higher nAJ's

```
1: while |UT| ≥ 1 do
2:   Get and remove the last element in UT, Term_t
3:   if nUsedMax[Term_t] ≥ 1 then // Term_t is usable
4:     OldnAJMin/Max = nAJMin/Max[Term_t]
5:     OEChildren = EChildren[Term_t]
6:     for each PS_ti in UPSMin[Term_t] do
7:       Update PSnAJMin[Term_t][PS_ti]
8:       Remove PS_ti from UPSMin[Term_t]
9:     end for
10:    for each PS_ti in UPSMax[Term_t] do
11:      Update PSnAJMax[Term_t][PS_ti]
12:      Remove PS_ti from UPSMax[Term_t]
13:    end for
14:    for all PS_ti and PS_tj of Term_t do
15:      if PSnAJMin[Term_t][PS_ti] ≥ PSnAJMax[Term_t][PS_tj] then
16:        PSR.insert(PS_ti)
17:      end if
18:    end for
19:    if |PSR| ≥ 1 then // Some PS's are to be removed
20:      Remove PS's And Update nUsedMax
21:      if nUsedMin[Term_t] ≥ 1 then // ETerm
22:        NEChildren = EChildren[Term_t] − OEChildren
23:        Update nUsedMin Because Of NEChildren
24:      end if
25:    end if
26:    Calculate and store NewnAJMin/Max of Term_t
27:    Compare them with OldnAJMin/Max respectively
28:    if NewnAJMax ≠ OldnAJMax then
29:      Determine which PS's (of other terms) whose nAJMax need to be updated.
30:      Update UPSMax accordingly
31:    end if
32:    if NewnAJMin ≠ OldnAJMin then
33:      Determine which PS's (of other terms) whose nAJMin need to be updated.
34:      Update UPSMin accordingly
35:    end if
36:  end if
37: end while
38: return
```

before doing any update. Lines 6 through 9 (Lines 10 through 13) update $nAJMin(nAJMax)$ of the PS's of $Term_t$ specified in $UPSMin[Term_t](UPSMax[Term_t])$, respectively. Lines 14 through 18 apply Corollary 12 to prune out expensive PS's. PS's to be removed are stored in PSR. Details of procedure in Line 20 is omitted due to space limitation. That procedure, essentially, propagates the effect of removing a PS_t of $Term_t$ to $nUsedMax$ of some (or all) of its constituent terms. This, in turn, can affect $nAJMin$ of some other terms PS's. The affected terms and PS's are added to $UT, UPSMin$, respectively, so that they are updated in the following iterations. Removing PS's from $Term_t$ may not only affect $nUsedMax$ of the constituting terms, but also, may add to $EChildren[Term_t]$. Now, if $Term_t$ is an $ETerm$, and it gained new $EChildren$ in this iteration, then its new $EChildren$ will also become $ETerms$. This is handled in Lines 21 through 24

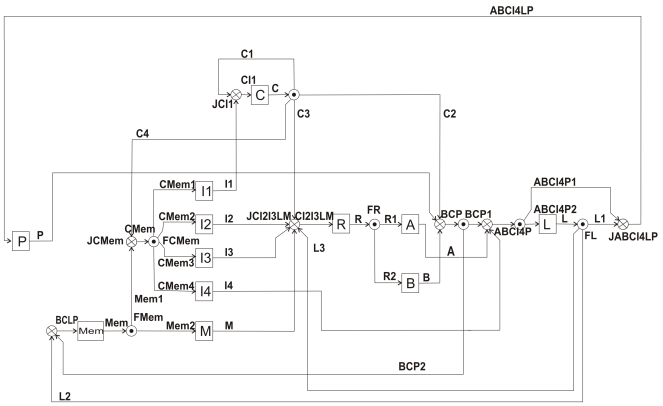

Fig. 3. Control network of the LI version of MiniMIPS processor.

of Algorithm 1. Details of procedure in Line 23 has been omitted due to space limitation. This procedure propagates the effect of essentiality (i.e., $nUsedMin$) to the $nAJMax$ values of other terms PS's in the network. Again, the affected terms and PS's are added to $UT, UPSMax$, respectively, so that they are updated in the following iterations. Final part of Algorithm 1 (i.e., Lines 26 through 35) checks if there is any change occurred to the values of $nAJMax$ and $nAJMin$ of $Term_t$. If so, it determines which terms and PS's are affected by these changes. It changes UT, $UPSMax$ and $UPSMin$, accordingly. Algorithm 1 will continue to iterate until UT is empty (i.e., no more terms need to be updated).

D. Find Optimum Solution

In all the examples included in this paper, the search space, at this point, contains only one solution (i.e., $OptSoln$). However, in general, we have to explore the remaining search space to find $OptSoln$. Solutions costs are evaluated using Definitions 7 and 10. Two more lemmas, a theorem, and an algorithm have been developed to expedite finding $OptSoln$ in the remaining search space. However, they have been omitted due to space limitation. In Example 2, $OptSoln$ is the same as $Soln_1$ of Eq. 1, and its $Cost$ is 9.

IV. Results

A. Case Study: MiniMIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture, first designed by Hennessy [9]. MiniMIPS is an 8-bit subset of MIPS. It is fully described in [10].

MiniMIPS has total of 12 synchronization points (i.e., registers): 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). 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 $P : A, B, C, I4, L, P$.

A hand optimized version of an LI control network of MiniMIPS, is reported in [4]. It incorporates 14 $J2$'s and 14 $F2$'s. On the other hand, applying this algorithm to the same problem, results in the control network of Fig. 3. $OptSoln$ uses 12 $J2$'s and 12 $F2$'s, for a 14% reduction.

B. CNG Tool

The algorithm has been coded in C++ within a tool called CNG. CNG accepts an input file with the required register-to-register communications. It returns $OptSoln$ and the minimum cost. Table II shows CNG runtime of the same examples encountered in this paper, plus two design examples. S298 is an ISCAS-89 benchmark. The machine used has Intel(R) Xeon(TM) CPU 3.20GHz processor and a 2GB memory.

TABLE II
CNG Runtime of different problems

Problem	$\|SourceS\|$	$\|DestinationS\|$	$\|TermS\|$	$Cost$	Runtime
Example 1	2	2	5	1	5 ms
Example 2	7	5	19	9	6 ms
MiniMIPS	12	12	29	12	6 ms
S298	17	20	53	22	15 ms

V. Conclusion

Transforming clocked designs into latency insensitive designs require creation of a handshake control network. We presented an algorithm that has been proven to automatically generate an optimal control network consisting of the minimum number of 2-input joins and 2-output forks. We also presented a CAD tool, CNG, that implemented the algorithm. It was applied to the MiniMIPS processor resulting in 14% reduction in the number of control steering units over a hand optimized design in a contemporary work. The algorithm does not guarantee minimum control network path delay. In some cases, a control path delay can be reduced by reconstructing some common terms rather than sharing them. Additional delays may exist due to the sequential ordering required by sharing. Normally, this is not a problem since the critical data path delay is usually larger than the critical delay in the control network. Adding delay metrics to the algorithm is kept for future work.

Acknowledgment

This work was supported by the National Science Foundation under grant CCF-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*, vol. 20, no. 9, Sep 2001, pp. 1059–1076.

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

[3] L. Carloni and A. Sangiovanni-Vincentelli, "Coping with latency in soc design," *Micro, IEEE*, vol. 22, no. 5, pp. 24–35, Sep/Oct 2002.

[4] E. Kilada, S. Das, and K. S. Stevens, "Synchronous Elasticization: Practical Considerations And MiniMIPS Case Study," *Submitted to The IEEE International Symposium on Circuits and Systems 2010. ISCAS 2010.*

[5] 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.

[6] 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.

[7] N. Andrikos, L. Lavagno, D. Pandini, and C. Sotiriou, "A fully-automated desynchronization flow for synchronous circuits," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, June 2007, pp. 982–985.

[8] 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.

[9] J. H. et al., "The MIPS Machine," in *COMPCON*, 1982, pp. 2–7.

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