A Lattice-Based Framework for the Classification and Design of Asynchronous Pipelines

Peggy B. McGee Steven M. Nowick*
Department of Computer Science
Columbia University
New York, NY 10027
{pmcgee,nowick}@cs.columbia.edu

ABSTRACT

This paper presents a unifying framework for the modeling of asynchronous pipeline circuits. A pipeline protocol is captured in a graph-based model which defines the partial ordering of both its control and data events. The relationship between an entire space of different protocols is then captured in a semi-lattice, which has well-defined top and bottom elements, corresponding to the most concurrent and least concurrent protocol variants, respectively. This framework also provides a set of correct-by-construction transformation rules which allows for the systematic exploration of the entire design space by their successive application. To the best of our knowledge, this is the first formal framework for asynchronous pipelines which can capture protocols from a variety of logic style families, including both dynamic and static. It is also the first to provide a formal foundation for the design-space exploration of asynchronous pipelines.

Categories and Subject Descriptors: B.6.1 [Hardware]:Logic Design - Design Styles, B.6.3 [Hardware]:Logic Design - Design Aids, C.5.4 [Computer Systems]:Implementation - VLSI
General Terms: Design, Theory
Keywords: pipeline, framework, asynchronous, digital design, protocols

1. INTRODUCTION

Pipelines are critical to the design of high-speed asynchronous systems, and their design and synthesis have been an active research area in the asynchronous community, as evident in the large amount of recent research in the area ([4, 6, 7, 8, 10, 12, 15]). This paper aims to formally capture the underlying structure and relationship of asynchronous pipeline protocols into a unifying framework. The motivations for this work are threefold. First, it is useful to have a common language for comparing and evaluating different pipeline designs. Such a unifying framework provides designers with a meaningful way to assess design trade offs, such as concurrency, latency, area, throughput, etc. during design-space exploration. Second, by capturing the fundamental properties of pipeline designs in a formal structure, a platform is provided for the development of new designs that could have been overlooked using a more intuitive design approach. Third, it can also serve as an aid in verifying existing designs. Hidden assumptions made on designs can often be exposed when put into a formal framework.

The key contributions of this paper are the following. First, it introduces a formal, graph-based model for describing the behavior of a wide variety of asynchronous pipeline protocols, including both dynamic and static logic styles. Second, it presents a taxonomy of pipeline protocols and captures their relationship formally in a semi-lattice (and in a lattice for a subset of logic families), which has clearly-defined top and bottom elements. Third, it introduces a set of correct-by-construction transformation rules (i.e., legal moves) for design-space exploration. As a demonstration for this method, we show how a number of existing protocols and some new protocols that have not appeared in published literature can be arrived at by successive applications of the rules.

The main focus of this paper is on latchless dynamic pipelines, since a complete taxonomy for these pipelines has not been previously proposed. However, in a later section, we briefly show how our model can be extended to static pipelines as well. Three dynamic logic styles are captured: non-footed, footed with unified control (eval/precharge, as in Williams' PC0 style [15]), and footed with separate control (i.e., distinct eval and precharge controls, as in *high-capacity pipelines* [12]). For each such logic style, an entire lattice or semi-lattice of the design space of legal pipeline protocols can be generated.

For simplicity, we limit ourselves to consider only linear pipelines in this paper. We believe extensions to non-linear pipelines and more complex asynchronous control circuits are possible within this framework, but is beyond the scope of the paper. Also, we confine ourselves only to robust protocols without timing assumptions. Timing assumptions are considered an implementation-level detail not dealt with at the protocol level.

There has been a small number of related papers on capturing taxonomies of asynchronous pipelines. Lines [6] presented a range of dynamic pipeline protocols by considering various handshake reshufflings (i.e., interleavings of request and acknowledge events on different communication channels). In Furber and Day [4], a similar notion of handshake reshuffling, but extended to also include data events, was used to model and compare several static pipeline protocols.[1] Blunno et al. [2] and Kondratyev et al. [5] presented innovative approaches for modeling and comparing a variety of protocols in a hierarchy. Both of these approaches model only control events, e.g. latch enable and disable. A detailed comparison with related work is presented in Section 6 of this paper.

Of these approaches, only [2] attempts to capture systematically a subspace of asynchronous pipeline protocols. In particular, Petri Net models are used to capture the behavior of a family of static

*This work was supported by NSF ITR Award No. NSF-CCR-0086036. A preliminary version of this paper was presented at [9].

[1] A similar modeling approach for individual protocols was used in Yakovlev et al. [16], but their focus was not on developing protocol taxonomies.

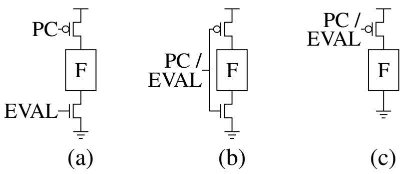

Figure 1: Dynamic function blocks: (a) Footed with separate controls (b) Footed with unified control (c) Non-footed

pipeline protocols, and the models are then placed in a partial order. However, there are several key differences between our work and theirs. First, our model uses both control and data events. It will be shown that the modeling of data events is necessary to distinguish certain dynamic pipeline protocols. In particular, we will show that some legal protocols cannot be properly captured by a control-event only model. Second, our framework places the hierarchy of pipeline protocols into a semi-lattice, with well-defined top and bottom elements. In addition, the entire space of legal protocols is captured between these bounds. Each individual protocol is mapped to a corresponding vertex in the lattice. Such an approach has not been previously proposed. Third, our graph-based model for pipeline protocols has a clear notion of "legal (i.e., safe) moves", within well-defined bounds, where a precedence arc can be moved to define a new legal pipeline protocol. Finally, we focus on dynamic pipelines, and show extensions to capture static latched pipelines; in contrast, [2] focuses on static latched pipelines only. Also, unlike previous approaches, we demonstrate that a variety of logic families can be captured, including several dynamic variants (footed *versus* non-footed, with separate *versus* unified control), as well as static styles. In addition, we show that two alternative common styles of pipeline acknowledgments can be distinguished in our model using additional constraints: "early-done" (e.g. [12]) and "conservative-done" signaling. Taken together, using a few simplifying assumptions, this work presents a unified formal framework for capturing a taxonomy of asynchronous pipeline protocols.

The rest of the paper is organized as follows. Section 2 provides basic background on the operation of asynchronous pipelines. Section 3 describes a method for formally modeling an individual pipeline protocol and deriving new protocols. Section 4 presents a taxonomy of existing and some new pipeline protocols, as points in a semi-lattice. Section 5 briefly shows how the model can be extended to capture latched static pipelines. Section 6 compares our model to some other related efforts. Finally, Section 7 presents conclusions and future work.

2. BACKGROUND

This section reviews several variants of dynamic logic and the basic operation of asynchronous pipelines.

Figure 1 shows three variants of dynamic logic function blocks. In all three variants, the function block enters its evaluate phase when the evaluate control is asserted, reads new input data and performs computation. When the precharge control is asserted, the function block enters its precharge phase and resets its output. In function blocks with separate evaluate and precharge controls (Figure 1a), called *dynamic logic with separate controls*, there is an additional "isolate" phase [12] when both the precharge and evaluate inputs are de-asserted and the output holds its previous value. The two control inputs are not allowed to be asserted at the same time to prevent fighting between the pull-up and pull-down stacks. Figure 1b shows a function block using *footed dynamic logic with unified control*, where precharge and evaluate are controlled by a single input and no isolate phase exists. Figure 1c shows a function block using *non-footed dynamic logic*, which has only a precharge control; there is also no isolate phase in this style.

Figure 2 shows the block diagram of an example three-stage

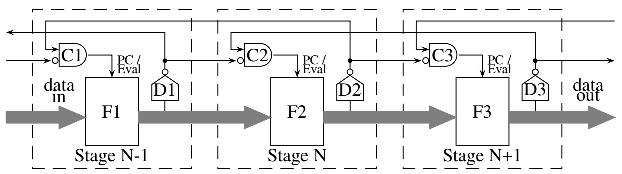

Figure 2: Block diagram of a PC0 pipeline

pipeline circuit, PC0 [15], which uses dynamic logic and four-phase signaling [11]. The datapath uses robust dual-rail logic where each bit is mapped to two wires, encoding '0' (01), '1' (10) and no valid data (00) [15]. The function blocks labeled F1, F2 and F3 are responsible for data computation and reset, and each is locally controlled by an evaluate/precharge input. Each pipeline stage must go through a data computation phase and a reset phase in each computation cycle. The completion detection units labeled D1, D2 and D3 detect the completion of data computation or reset of their preceding function blocks and generate appropriate request and acknowledgment control signals to neighboring blocks. The C-elements labeled C1, C2 and C3 [14] are asynchronous state holding elements which serve as "joins". When both inputs of a C-element are '0'('1'), the output is '0'('1'), otherwise it holds its previous value. The operation of the pipeline is as follows. Stage 2 evaluates as a result of two events from neighboring stages: when stage 1 has valid data (path through D1 and C2) and when stage 3 has completed reset of the previous data item (path through (D3 and C2). Similarly, stage 2 resets when stage 1 has completed reset and when stage 3 has valid data.

3. FORMALISM

This section introduces a formal graph-based model to capture the behavior of asynchronous pipelines and a set of transformations that can be performed on the model to derive new protocols.

3.1 Formal basics

Some basic algebraic definitions [3] that will be used in later sections is now reviewed.

DEFINITION 1. *Let P be a set, An* **order** *(or* **partial order***) on P is a binary relation $\preceq$ on P such that, for all $x, y, z \in P$, (i) $x \preceq x$ (reflexivity), (ii) $x \preceq y$ and $y \preceq x$ implies $x = y$ (antisymmetry), (iii) $x \preceq y$ and $y \preceq z$ implies $x \preceq z$ (transitivity).*

DEFINITION 2. *Let P be an ordered set. Then P is a* **chain** *if, for all $x, y \in P$, either $x \preceq y$ or $y \preceq x$.*

DEFINITION 3. *Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an* **upper bound** *of S if $s \preceq x$ for all $s \in S$. A* **lower bound** *is defined dually.*

DEFINITION 4. *Let P be a non-empty ordered set. If $x \vee y$ (the greatest lower bound of x and y) or $x \wedge y$ (the least upper bound of x and y) exists for all $x, y \in P$, then P is a* **semi-lattice***. If both $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$, then P is a* **lattice**.

3.2 Modeling pipeline behavior

A formal model is now introduced to describe the behavior of a single pipeline protocol.

The protocol of an asynchronous pipeline can be modeled by a directed graph representing the partial order of events in its operation. There are two types of events: control and data. *Control events* are those that change the phase of operation of circuit elements. In designs using dynamic function blocks, they correspond to the start and end of the precharge phase and the evaluate (i.e., precharge-release) phase. In static designs using latches, they correspond to the latch enable and disable events. *Data events* do not refer to circuit phases, but rather to the progress of data and spacers (i.e., null or reset tokens) through the pipeline.

In the remainder of this section, pipelines using dynamic function blocks (without explicit latches) and four-phase signaling with

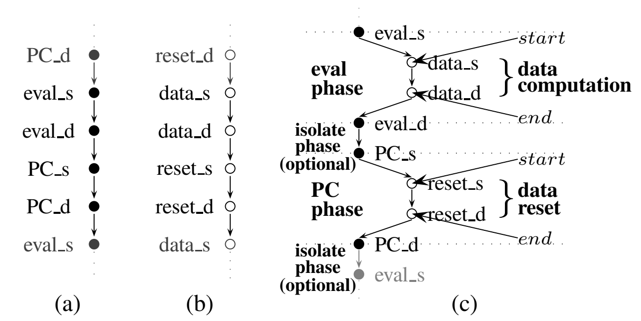

Figure 3: Constraints on ordering of events in a single pipeline stage: (a) data events (b) control events (c) interaction between data and control events

dual-rail data encoding are used as examples to illustrate the ideas. This subsection focuses only on the modeling of function blocks using dynamic logic with separate controls (Figure 1a). Discussion of the other two types of function blocks is left to Section 3.3.

Three sets of ordering constraints, two intra-stage and one inter-stage, need to be enforced among the control and data events in a pipeline using dynamic function blocks. The first two are common across all three variants of function blocks.

Intra-stage constraints. The first set of constraints, shown in Figures 3a and 3b, is imposed among the control events and among the data events *within one stage* to ensure correct circuit operation and the flow of data and reset tokens. The solid black circles and empty circles in the figures represent control and data events, respectively. An arrow between two events x and y indicates the ordering relationship $x \preceq y$. Events and arrows that belong to the same computation cycle are colored in black. Those belonging to the previous and next cycles are colored in gray.

The "eval" and "pc" events are control events, defining the physical state of the circuit. Events $eval_s$ and $eval_d$ correspond to the start and the end of the evaluate phase. Events pc_s and pc_d correspond to the start and the end of the precharge phase. The gap between $eval_d$ (end of evaluate phase) and pc_s (start of precharge phase) can be used to model a distinct isolate phase. Similarly, the gap between pc_d and $eval_s$ can model a second isolate phase.

The "data" and "reset" events are data events, defining the movement of actual data through the pipeline. Events $data_s$ and $data_d$ define the start and end of the actual evaluation of data in the pipeline stage (but not the start and end of the evaluation control phase). Events $reset_s$ and $reset_d$ define the start and end of the actual return-to-zero in the pipeline stage (but not the start and end of the precharge control phase).

The second set of constraints ensures proper interaction *between* control and data events in the same stage. A function block cannot start data computation ($data_s$) before it enters the evaluate phase ($eval_s$), and cannot leave the evaluate phase ($eval_d$) until data computation is complete ($data_d$). The dual set of these constraints is required for the precharge phase and the reset of data. Figure 3c shows the second set of constraints superimposed upon the first. Note that the arcs in Figure 3c can be classified into three categories: arcs leading from a control event to a data event, arcs leading from a data event to a data event, and arcs leading from a data event to a control event. The first two can be considered physical realities that is guaranteed by the circuit. The third is a requirement that has to be enforced by design.

Inter-stage constraints. A third set of constraints is required to ensure proper communication *between stages*. This set of constraints is different for each of the three dynamic logic function block styles. Figure 4 shows the weakest constraints for function blocks using dynamic logic with separate controls between three pipeline stages $N-1$, N and $N+1$. Each stage follows the pro-

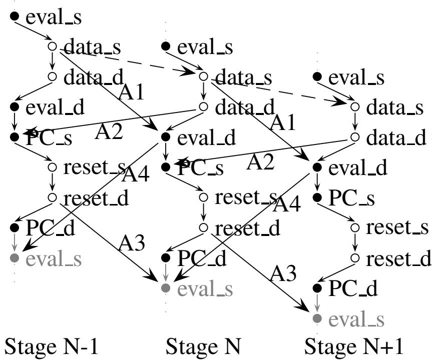

Figure 4: Constraints for footed dynamic logic with separate controls (most concurrent protocol)

tocol outlined in Figure 3c. Several inter-stage arcs are added to synchronize behavior of adjacent stages.

The most basic inter-stage synchronization is a *data-dependency arc*, shown as the dotted arc from the $data_s$ event of stage $N-1$ to the $data_s$ event of stage N in Figure 4. It ensures that the start of computation of stage $N-1$ precedes that of stage N.[2]

The remaining four inter-stage arcs control sequencing between data to control events or control to control events. Arc A1 ensures that stage $N-1$ has started computation before stage N ends its evaluate phase. This ensures that stage N has time to read the new data token before closing itself to inputs. Arc A2 ensures that stage $N-1$ does not start its precharge phase before stage N has completed data computation. This prevents stage $N-1$ from removing the data token too early while stage N still needs it. Arc A3 ensures that stage $N-1$ has reset its previous data token before stage N starts the evaluate phase for the next data cycle. This prevents stage N from reading stale data. Arc A4 ensures that stage $N-1$ does not start its evaluate phase for the next data cycle before stage N has finished with the evaluate phase of the current data cycle. This prevents new data from stage $N-1$ from overwriting the data in stage N if stage N is slow.

These four control arcs, two in the forward direction and two in the backward direction, together form a minimal set of constraints for inter-stage communication between two pipeline stages.

The same sequence of events is repeated for every iteration during the execution of the pipeline for each stage, and each stage in the pipeline has the same sequence of events.

3.3 Modeling extensions

In this section, the basic model developed in Section 3.2 is extended to handle designs using other variants of dynamic logic families, as well as designs using more aggressive signaling.

Other variants of dynamic logic. The basic model can be extended to handle two alternative logic families: (i) footed dynamic logic with unified control (Figure 1b), and (ii) non-footed dynamic logic (Figure 1c). Both styles share the same design invariant constraints shown in Figure 3; additional constraints are simply superimposed to capture the different circuit behaviors.

In footed dynamic logic with unified control, the function block can only be in either precharge or evaluate phase, meaning there is no isolate phase for extra latching capability. This translates to an extra constraint in the specification: stage $N+1$ must have started its precharge phase before stage N starts changing its outputs from its reset values, otherwise the new data from stage N would overwrite the data in stage in stage $N+1$.

In non-footed dynamic logic, further restrictions must be ob-

[2]In some designs, stage N cannot start computation until stage $N-1$ has completed all its computation. In this case, the dotted arc would go from the $data_d$ event of stage $N-1$ to the $data_s$ event of stage N. This paper considers the more general case where there is no such restriction (i.e., allowing "eager evaluation").

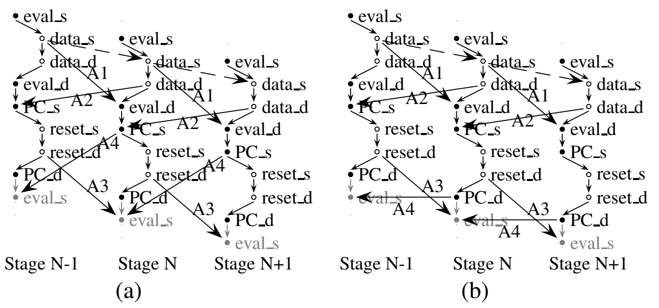

Figure 5: Constraints for alternative dynamic logic families (most concurrent protocol): (a) Footed with unified control (b) Non-footed

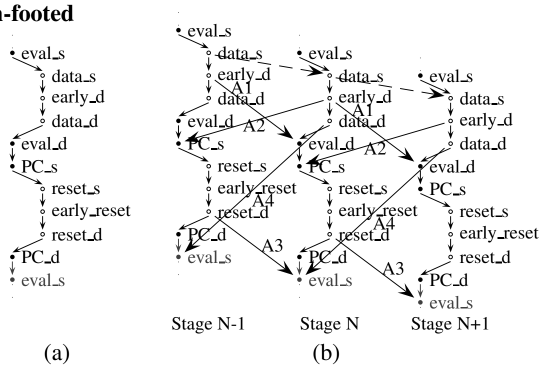

Figure 6: Constraints for dynamic logic using "early done" signaling: (a) Single stage constraints (b) Inter-stage constraints

served. When it is in its precharge phase, its input data must be at reset, or there will be a direct path from power to ground.

These extra constraints can be layered on to the basic model for footed dynamic logic with separate control shown in Figure 4. The resulting protocol graphs are shown in Figures 5a and 5b.

"Early-done" signaling. The basic model can also be augmented to capture more aggressive signaling in a design. For example, in some dynamic pipelines [12], an "early tap-off" of data completion is allowed: as soon as data passes through the first stage of logic (before the end of computation), the function block can signal the previous stage to precharge. In dynamic logic, early done is typically a safe communication: once data passes through the first level of dynamic logic in a stage, it is safe to reset the input to the stage. The use of early done signals can be captured simply by inserting two extra events into the basic model: $early\_d$ and $early\_reset$, as shown in Figure 6a. Figure 6b shows the inter-stage constraints for designs using footed dynamic logic with separate controls with early done signaling.

3.4 Transformations

The constraints derived in Sections 3.2 and 3.3 represent the weakest constraints that need to be imposed between the control and data events in a protocol to ensure safe operation. These constraints are represented as a partial order of events, which can be satisfied by a variety of distinct protocols. In this section, a set of transformations that allows for the automatic generation of all such protocols is introduced.

By definition, a partial order is closed under transitivity (see Section 3.1). Hence, starting from an initial object respecting certain constraints, successive applications of the transitive rule generates designs respecting the same constraints. Correct-by-construction transformation of the model thus simply amounts to the application of such rule, which can be translated to *moves of arcs* in the graph.

Figure 7 illustrates three allowable moves in a partial order graph that guarantees safe transformation. Moves M1 and M2 are *concurrency reduction* moves, which impose more ordering in a graph

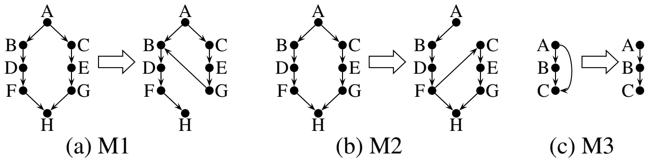

Figure 7: Concurrency reduction transforms

while still enforcing the original constraints. Move M3 removes redundant arcs in a partial order graph without affecting the underlying concurrency. Note these transformations apply only to the solid control arcs in the graph models. The dotted data arcs represent physical realities that cannot be changed.

In more detail, move M1 states that the target of an arc between two chains can be moved up, as long as it does not go above the upper bound of the two chains. Algebraically, the move in Figure 7a transforms the original ordering

$$A \preceq B \preceq D \preceq F \preceq H \; and \; A \preceq C \preceq E \preceq G \preceq H \tag{1}$$

to

$$A \prec C \preceq E \preceq G \preceq B \preceq D \preceq F \preceq H \; and \; A \preceq B \preceq D \preceq F \preceq H. \tag{2}$$

Note that the ordering imposed by Relation (2) still respects that by Relation (1), though it is more restricted. If the target of the arc moves to or above the upper bound of the two chains at point A, it would result in the ordering $A \preceq C \preceq E \preceq G \preceq A$, which violates the original constraints, and introduces a "circular wait" condition which can result in *deadlocks* in a protocol. Move M2 states that the origin of an arc between two chains can be moved down, as long as it does not go below the greatest upper bound of the two chains. The same mathematical reasoning as in M1 applies.

Successive applications of moves M1 through M3 can thus generate a set of orderings that are guaranteed to respect the safety and liveness properties of the original constraints. However, starting from an initial ordering, there exists a limit to the number of transformation moves that can be made, after which no more legal moves are possible, and a most constrained protocol is arrived. When this bound is reached, one cannot move the origin of an arrow further down or its head further up without violating moves M1 or M2, i.e., introducing deadlocks.

For simplicity, not all legal configurations will be considered in this paper. An arc in the protocol graph can originate from a data or control event, and end in a data or control event. All arcs originating from and ending in a combination of these events can form legal protocols as long as they respect the minimal set of constraints for the chosen logic style. However, some of these arcs have more natural translations to circuit implementations than others. In the remainder of this paper, only control arcs with targets at control start events, and origins at the start or end of data or control events will be considered. Investigations into designs with other types of arcs will be included in future work.

As an example, Figure 8a shows a protocol after a series of concurrency reduction transforms from the initial weakest constraints of footed dynamic logic with separate control, as shown in Figure 4. No further legal moves can be made in the graph. In this protocol, stage N is allowed to enter the evaluate phase when stage N-1 has completed data computation, and is not allowed to enter the precharge phase until stage N+1 has completed reset of its data. In a linear pipeline, the leftmost stage would have to wait for the data token to propagate all the way to the rightmost stage, wait for a response from the environment, and propagate the reset token all the way from the rightmost stage backward before it could start evaluating the next data token.

The constraints shown in Figure 8a present an absolute bound on how far the arcs in the specification can move during the transformation. This protocol represents an *extreme linearization* of dynamic pipeline behavior: at most one data token can propagate

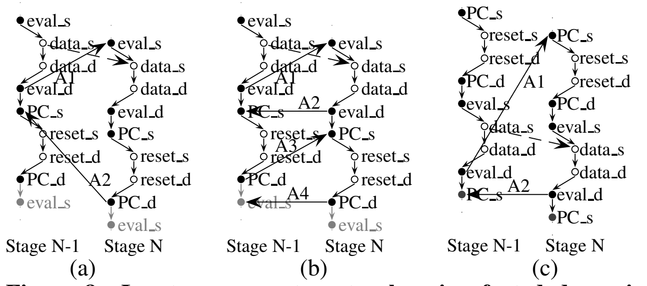

Figure 8: Least concurrent protocols using footed dynamic logic with separate control

through the pipeline at any time, followed by a reverse propagation of a "spacer" wave (i.e., precharge). Such a pipeline is legal, but cannot be formed into a ring without deadlock.

If this "ring-based deadlock" situation is not desired, an extra restriction must be imposed upon the movement of the arcs during the transformation step: the enabling condition for the precharge of stage N cannot be any event in the downward chain from the start of precharge of stage $N+1$.[3] Likewise, any other event from stage N cannot be the enabling event for the corresponding event in stage N-1. With this extra restriction imposed, two distinct least concurrent designs can be derived from the specification in Figure 4, and are shown in Figures 8b and 8c. Note that there are only two inter-stage arcs in Figures 8a and 8c, as the rest become redundant and are removed by move M3 during the graph transformation. The protocol in Figure 8c is novel: to the authors' knowledge it has not appeared in the literature.

4. A TAXONOMY OF PROTOCOLS

This section presents a taxonomy of pipeline protocols. Each distinct protocol is mapped to a corresponding vertex in a semi-lattice, and a concurrency relation defines their ordering.

It is shown in Sections 3 how the least constrained (i.e., most concurrent) and the most constrained (i.e., least concurrent) protocols for a particular logic family are derived. In between these two extrema, a continuum of protocols exists. These protocols together can be represented in a semi-lattice or a lattice, with the least constrained and most constrained protocols as its top and bottom elements, respectively. In particular, the unique top element corresponds to the most concurrent protocol for which safe data computation is guaranteed (i.e., no data overrun or sampling of stale data), as derived in Sections 3.2 and 3.3. The bottom elements correspond to the least concurrent protocols which still allow for useful data computation (i.e., no deadlock), as derived in Section 3.4 based on the structural constraints in graph transformation. The semi-lattice for protocols using dynamic logic is shown in Figure 9.

The semi-lattice bounded by the solid line represents the design space of footed dynamic logic with separate controls. The sub-lattices bounded by the dotted line and the dashed line represent the design space of footed dynamic logic with unified control and that of non-footed dynamic logic, respectively. Note that for the latter two logic families, the design space form *lattices* (i.e not simply semi-lattices) with unique top and bottom elements.

The top element in the lattice or semi-lattice for each logic family corresponds to the most concurrent protocol of the logic family as shown in Figures 4 and 5. The bottoms of the semi-lattice correspond to the protocols shown in Figure 8. Note that for simplicity, the semi-lattice is restricted to consider only designs with more natural translations to circuits (as defined in Section 3.4).

Each vertex in the semi-lattice indicates a valid pipeline protocol. Vertices marked with black dots indicate protocols used by

[3]This condition, together with the earlier bounds imposed on moves M1 and M2, form the necessary and sufficient conditions for ring-based deadlock avoidance in deriving new protocols. The proof is omitted from this paper due to space limitations.

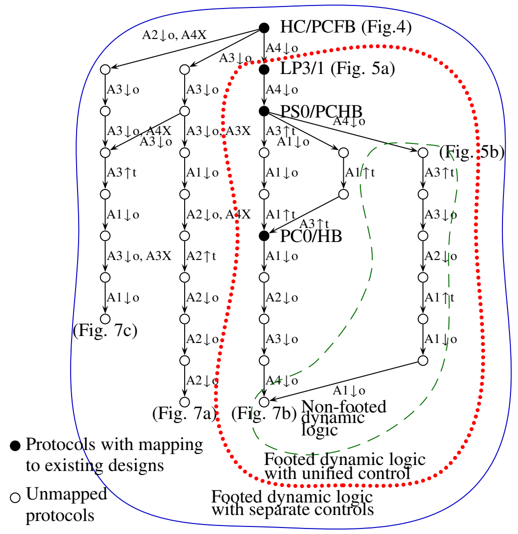
HC/PCFB (Fig.4)
LP3/1 (Fig. 5a)
PS0/PCHB
PC0/HB
(Fig. 5b)
(Fig. 7c)
(Fig. 7a)
(Fig. 7b)
Non-footed dynamic logic
● Protocols with mapping to existing designs
○ Unmapped protocols
Footed dynamic logic with unified control
Footed dynamic logic with separate controls

Figure 9: A lattice of pipeline protocols using dynamic logic

existing designs. These include, from top to bottom, HC [12] and PCFB [6], LP3/1 [12], PS0 [15] and PCHB [6], and PC0 [15] and HB [6]. Three of these protocols, PCFB, PCHB and HB, have been used in the asynchronous MIPS R3000 Microprocessor from Caltech [8] and at Fulcrum Microsystems Inc. [1]. Vertices marked by hollow dots indicate protocols that have not yet been mapped to any existing design.

Any two protocols in the semi-lattice joined by an edge has the following relationship: The protocol at the target of the edge is obtained from that at the origin by one application of one of the concurrency reduction moves introduced in Section 3.4. Upward arrows marked with 'o's, downward arrows marked with't's, and the 'X' symbols represent moves M1 (moving origin of a arc down), M2 (moving target of an arc up) and M3 (redundancy removal), respectively. The arc names next to the transformation symbols correspond to the labels in Figures 4 and 5.

Several otherwise not obvious observations can be made by inspection of the semi-lattice. First, the top and the bottoms are well defined. This observation implies that, starting from any point in the semi-lattice, one can arrive at any other point via a finite number of concurrency-reduction (i.e., downward), or its dual, concurrency-augmentation (i.e., upward) moves. In terms of design space exploration, this means one can traverse the entire space by successive and systematic application of these moves.

Second, the diagram exposes interesting relationships between some designs. For example, PCHB [6] and HC [12] share the same underlying pipeline protocol. The former uses dual-rail signaling convention and is timing-robust (except for quasi-delay insensitive assumptions). The latter uses bundled-data signaling with one-sided timing assumptions, which uses less area and has better performance. A designer wishing to use the protocol can assess the trade offs and decide between the two.

Third, protocols that have not yet been mapped to any existing design, indicated by hollow dots in the semi-lattice, represent new design possibilities. Some can be seen as variants of existing protocols in the semi-lattice, while others might suggest entirely new synchronization schemes.

5. DESIGNS USING STATIC LOGIC

In this section, it is briefly shown how the graph-based model introduced in Section 3.2 can be extended to handle pipelines using static logic with latches. Figure 10 shows the protocol for a linear pipeline using static logic. The constraints require that the latch at stage N to be open before the stage receives a new data token, and that it cannot be closed until the data token is safely received. The

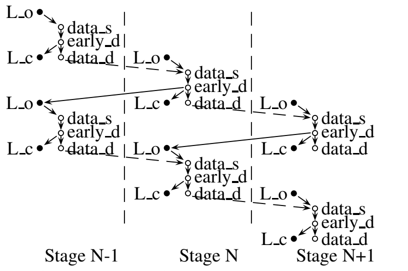

Figure 10: Constraints for static logic with 2-phase signaling

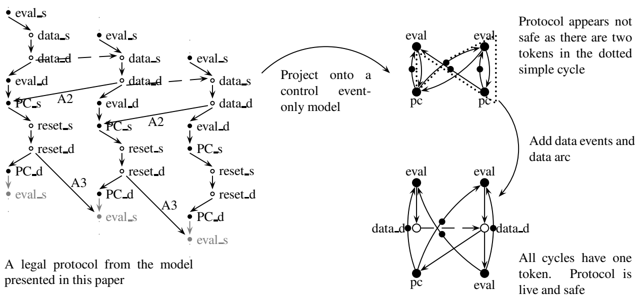
eval_s
data_s
data_d
eval_d
PC_s
A2
reset_s
reset_d
PC_d
A3
A legal protocol from the model presented in this paper
Project onto a control event-only model
eval
pc
Protocol appears not safe as there are two tokens in the dotted simple cycle
Add data events and data arc
data_d
All cycles have one token. Protocol is live and safe

Figure 11: Comparison of models: using only control events vs. using data and control events

latch cannot be open again until the data token has been safely received at stage $N+1$. The protocol shown in the figure corresponds to that used by the MOUSETRAP pipeline proposed in [13].

6. COMPARISON WITH RELATED WORK

In this section, we compare our modeling approach to other related work, and highlight some advantages.

In [4], [6], and [16], *handshake protocol* events such as request and acknowledgment signal transitions are used to model the behavior of asynchronous pipelines. In contrast, we model *data* and *control* events at the *circuit* level, such as the start and the end of the evaluate or precharge phase. We believe that by directly dealing with control events at the circuit level, as well as data events, our modeling is at the appropriate level of granularity which allows us to address the safety and liveness issues of the resulting circuit, and to precisely define the upper and lower bounds of legal protocols, which previous approaches have not been able to provide.

In [2], circuit *control* events only (but not data events, which we include) are used for modeling. We believe that some legal protocols cannot be properly captured by a control event-only model. Figure 11 shows a legal protocol within our framework, using both control and data events. When projected onto a marked graph representation of a control-event only model, the protocol appears unsafe as it violates the safety property of a marked graph, which requires that all simple cycles contain one token only (according to [2]). As shown, by adding a data event and a data arc to the graph, the protocol satisfies the safety requirement. In sum, a control-only modeling style does not allow expressing some legal behaviors which can be captured in our more expressive control-and-data modeling approach.

7. CONCLUSIONS

In this paper, we have presented a new framework to capture a taxonomy of pipeline protocols. First, each protocol is modeled by a graph, containing both control and data events, which are partially-ordered. It is shown that the expressiveness of including both event types allows the modeling of behaviors which are not easily captured in previous models (i.e., using only control events), as demonstrated by an example in Section 6. In addition, we have demonstrated that a variety of targeted logic families can be captured, including several dynamic styles (footed *versus* non-footed, with separate *versus* unified control), as well as static styles.

Second, we introduce a formal means of systematic design-space exploration, through the use of correct-by-construction transformations on the graph-based model of pipeline protocols.

Finally, the pipeline protocols are arranged in a semi-lattice, forming a complete taxonomy. Each pipeline protocol is mapped to a single point in the semi-lattice, and several existing pipeline styles are identified. The semi-lattice has well-defined top and bottom elements, which correspond to the least constrained and most constrained protocol variants of a particular logic family, respectively. In particular, our approach provides, for the first time, precise, well-defined upper and lower bounds on the design space of concurrent protocols for asynchronous pipelines using dynamic logic, as well as the complete set of legal, intermediate protocols, which together form the complete design space. Protocols above the upper bound are not safe, and those below the lower bound are not live. Extensions to capture the static pipeline design space are also proposed.

Taken together, the paper provides a new approach to defining the design space of asynchronous pipeline protocols.

In future work, we plan to introduce objective functions as guides for design-space exploration in the lattice. We plan to explore an "instantiation" step, where the actual abstract protocols are synthesized into hardware. We plan to generalize the models, where timing arcs can be added, and more aggressive relative timing-oriented protocols can be captured. We also plan to consider a unified way of presenting the design space of pipelines using dynamic logic and static logic into a single lattice.

Acknowledgments

The authors would like to thank Profs. Stephen Edwards and Luca Carloni of Columbia University for helpful comments.

8. REFERENCES

[1] P. Beerel and A. Lines. Personal communication.

[2] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. In *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems*, 2004.

[3] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[4] S. B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(2), June 1996.

[5] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Automatic synthesis and optimization of partially specified asynchronous systems. In *Proceedings, 36th Design Automation Conference*, 1999.

[6] A. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, June 1996, revised 1998.

[7] R. Manohar, T.-K. Lee, and A. J. Martin. Projection: A synthesis technique for concurrent systems. In *Proceedings of 5th International Symposium on Asynchronous Circuits and Systems*, Apr. 1999.

[8] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Proceedings of Conference on Advanced Research in VLSI*, 1997.

[9] P. McGee and S. Nowick. A unifying lattice-based framework for the classification and design of asynchronous pipelines. *ACM/IEEE International Workshop on Timing Issues*, 2005.

[10] R. Ozdag and P. Beerel. High-speed QDI asynchronous pipelines. In *Proceedings of 8th International Symposium on Asynchronous Circuits and Systems*, 2002.

[11] C. Seitz. *"System Timing," Chapter 7 in Introduction to VLSI Systems, eds. C. Mead and L. Conway*. Addision-Wesley, 1980.

[12] M. Singh and S. M. Nowick. Fine-grain pipelined asynchronous adders for high-speed DSP applications. In *IEEE Computer Society Annual Workshop on VLSI*, 2000.

[13] M. Singh and S. M. Nowick. Mousetrap: Ultra-high-speed transition-signaling asynchronous pipelines. In *International Conf. Computer Design (ICCD)*, 2001.

[14] I. Sutherland. Micropipelines. In *Communications of the ACM*, 1989.

[15] T. Williams. *Self-Timed Rings and Their Applications to Division*. PhD thesis, Stanford University, Jun 1991.

[16] A. Yakovlev, A. Koelmans, and L. Lavagno. High-level modeling and design of asynchronous interface logic. *IEEE Design and Test of Computers*, 12, 1995.