Latency and Throughput Tradeoffs in Self-Timed Speed-Independent Pipelines and Rings

Ted Williams

Technical Report No. CSL-TR-90-431

August 1990

Latency and Throughput Tradeoffs in Self-Timed Speed-Independent Pipelines and Rings

Ted Williams

Technical Report: CSL-TR-90-431

August 1990

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 943054055

Abstract

Asynchronous pipelines control the flow of tokens through a sequence of logical stages based on the status of local completion detectors. As in a synchronously clocked circuit, the design of self-timed pipelines can trade off between achieving low latency and high throughput. However, there are more degrees of freedom because of the variances in specific latch and function block styles, and the possibility of varying both the number of latches between function blocks and their connections to the completion detectors. This report demonstrates the utility of a graph-based methodology for analyzing the timing dependencies and uses it to make comparisons of different configurations. It is shown that the extremes for high throughput and low latency differ significantly, the placement of the completion detectors influences timing as much as adding an additional latch, and the choice as to whether precharged or static logic is best is dependent on the cost in complexity of the completion detectors.

Key Words and Phrases: asynchronous, self-timed, pipelines, latency, throughput

I. Introduction

Self-timed systems [SEIT80] avoid the need for distributing global clocks and eliminate the performance loss from added margins necessary to account for clock-skew and worst-case conditions. Self-timing can refer both to the use of an on-chip clock generator [SANT89] providing an internal clock for synchronous blocks, or to systems such as those this report will consider which use local control communication between fully asynchronous blocks [MULL63]. A spectrum of possibilities exists for asynchronous design, ranging from utilizing carefully crafted matched delays [SUTH89] to completely delay-insensitive circuits [UDDI86],[EBER88]. Whereas delay-insensitive circuits have the same logical functionality for arbitrary delays in both gates and their interconnecting wires, circuits which satisfy the weaker property of correct operation for arbitrary gate delays but allow isochronic forks [MART86] in the interconnecting wires are called speed-independent. While actual delays may vary due to fabrication variations, voltage, temperature, or data dependencies, the philosophy behind delay-insensitive or speed-independent design is that by constructing a circuit which will be logically correct for any delay values, the resultant design will be more robust and the designer need not acquire all the information and specifications affecting the actual delays. However, to the extent that delay information is available, the designer can use it to optimize nominal performance in choosing between design alternatives, sizing transistors, and making local exceptions to a purely speed-independent design approach. This report will therefore analyze performance based on symbolic component delays, even though the delays would not affect the logical correctness of the circuit operation.

This report specifically addresses building deterministic pipelines which pass a sequence of data tokens through a succession of stages as shown in Figure 1. In contrast to a synchronous pipeline where the stages are all controlled by a global clock, a self-timed pipeline uses completion detectors along the datapath to generate local signals controlling the flow of tokens. Pipelines are useful where a sequence of data must pass through the same series of functional operations, as for example, in digital signal processing [MENG88]. Rings are a particular subset of pipelines where tokens are circulated from the output back to the input by self-timing [GREE87] without needing to wait for more external inputs as suggested in Figure 2. Such iterating rings are useful in implementing recursive operations such as for the evaluation of arithmetic functions [WILL87].

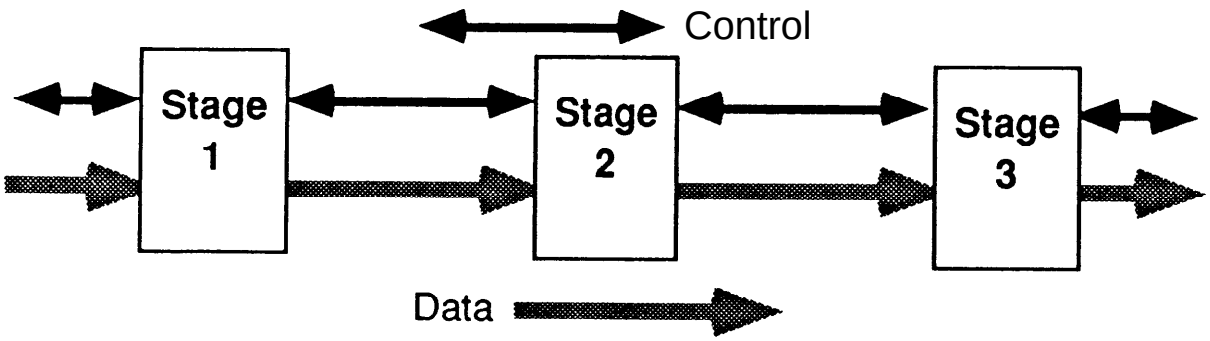

Figure 1: Overall structure of a pipeline is a linear sequence of stages

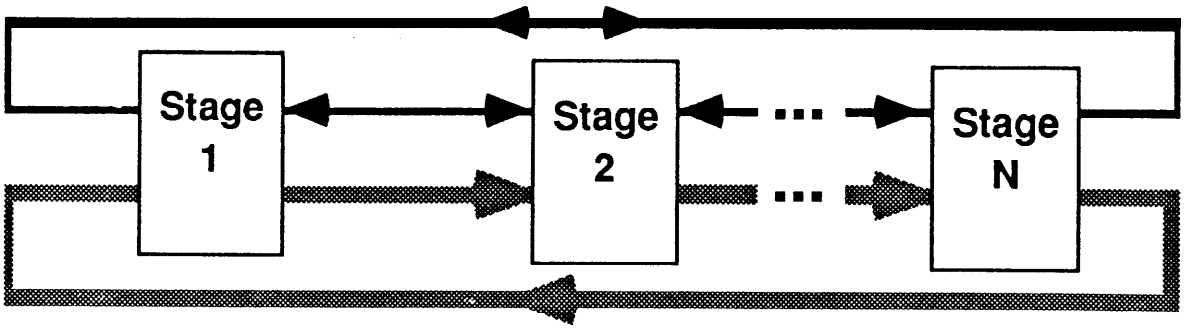

Figure 2: A self-timed ring is a loop of stages containing one or multiple tokens flowing around without intervention from external control except during initialization.

Both synchronous and asynchronous pipelines compose each stage by a function block with some number of latches. The performance depends on the relative timings and ordering of the components, as has been analyzed in [RAO85] for the synchronous case with registers. Good re-timing algorithms have been developed for increasing performance in synchronous systems by changing the number and location of registers [LEIS86]. The present report will characterize the performance of a range of possible configurations for self-timed pipelines with varying styles of function blocks and latches, varying numbers of latches per stage, and varying connection ordering to the completion detectors. Unlike the work in [MENG89] which synthesizes particular control arrangements based on assuming function block evaluation dominates all other delays, this report analyzes various configurations in terms of variables for the delays of each component.

Section II in this report describes the datapath signaling convention and the specific types of function blocks and latches to be considered. After defining the variables used to represent the component delays and measures of performance, Section III introduces the analysis method and the construction of two types of marked directed graphs [COMM71],[MURA77] used to determine the pipeline cycle time. Section IV explores the family of possible configurations of function blocks, latches, and completion detectors and analyzes the performance of each configuration. The comparisons are simplified by a set of standard assumptions, and Section V summarizes the resultant equations by a table displaying their coefficients. This section also constructs tables giving the latency, throughput, and occupancy of the different configurations for cases with specific bounds on the component delays. Section VI makes observations about the results and discusses the conclusions of this work for applications in self-timed pipeline and iterative ring design.

II. Datapath, Function Block, and Latch styles

Each stage in the pipelines to be considered in this report will consist of one function block and zero to several latches. Between the stages, a datapath conveys information on a unidirectional bus, and control wires may traverse in both directions. Every data token which is transmitted on a datapath must convey its presence with some completion indicating method, either encoded within the datapath itself, or bundled alongside of it on a separate wire.

Encoding completion into the data itself is usually implemented by passing each bit on a dual-monotonic pair [SEIT80] with the convention shown in Table 1, but higher order group-encodings are also possible [WILL87]. Sometimes, for small ***n,*** a simple l-of-n unary encoding on n wires is appropriate.

Wire A^T	Wire A^F	Signal A
0	0	Reset = Not Ready
0	1	Evaluated FALSE
1	0	Evaluated TRUE
1	1	Not used = Never occurs

Table 1: A dual-monotonic pair uses a simple encoding to convey both the value and completion-indication for a signal A on two wires, A^T and A^F

A disadvantage of embedded completion is the greater number of wires necessary. However, passing a complete datapath using embedded completion on dual-monotonic pairs has the advantage of being delay-insensitive; whereas, the use of a bundled completion signal must assume congruence in the delays of the data bus wires. Furthermore, the silicon area penalty does not necessarily imply a speed penalty since logic blocks can obtain either polarity of the input signal by using the appropriate wire from dual-monotonic pair; hence, signal inversions are free.

Completion detectors will tap off of the data bus at different points in different pipeline configurations. Since they need to detect both when all of the signals in a datapath are finished evaluating and when they are all finished resetting, completion detectors are usually formed by a tree of standard Muller C-elements [SEIT80] whose output is the state of the inputs when they were last the same. Though the data bus may be any number of bits wide and may contain more than one field, this report is not concerned with the details of combining the completion signals to form a single completion signal for the whole datapath and this task is abstracted into the "completion detector." However, particular applications where the datapath varies in width at different points within a pipeline stage may find significant interplay between the structure of the completion detector and the choice of where to place it along the pipeline. For example, the structure of the problem in [WILL87] allowed choosing to place a completion detector on a three wire unary encoded datapath instead of across a 48-bit dual-montonic pair datapath, saving significant delay and complexity.

In order to provide inputs to a datapath completion detector, each bit slice of the function blocks must embed its individual completion status on its output wires, even if only single-ended data is passed on to the next function block. For that reason, this report requires all function blocks to generate dual-monotonic outputs. Four styles of function blocks taking dual-monotonic inputs and generating outputs as dual-monotonic pairs with embedded completion are contrasted in Figure 3 for the implementation of a simple AND gate. The four styles are static logic, direct logic, semi-controlled precharge logic, and full-controlled precharge logic. Static logic and direct logic have the same pull-down tree and neither requires a precharge control input. However, the direct logic pull-up tree requires all its inputs to reset before resetting its output; whereas, the static logic pull-up tree is always the dual of its pull-down tree. By holding its outputs until all its inputs have reset, the timing of direct logic is more similar to controlled-precharge logic. Furthermore, by verifying that all of its inputs have reset before resetting its outputs, direct logic has the same abstracted dependencies as a C-element. Thus, configurations using direct logic function blocks can be delay-insensitive whereas static logic will, in general, not verify the arrival of both rising and falling transitions on its input signals.

Both semi-controlled and full-controlled precharge logic styles take a precharge control input. This input is also the logical inverse of enable, because the blocks must have precharge removed before the outputs can be enabled to transition to an evaluate state. The only difference between the semi-controlled and full-controlled precharge styles is the presence of the bottom transistor in the full-controlled style which prevents fighting if precharge is ever active concurrently with valid data inputs.

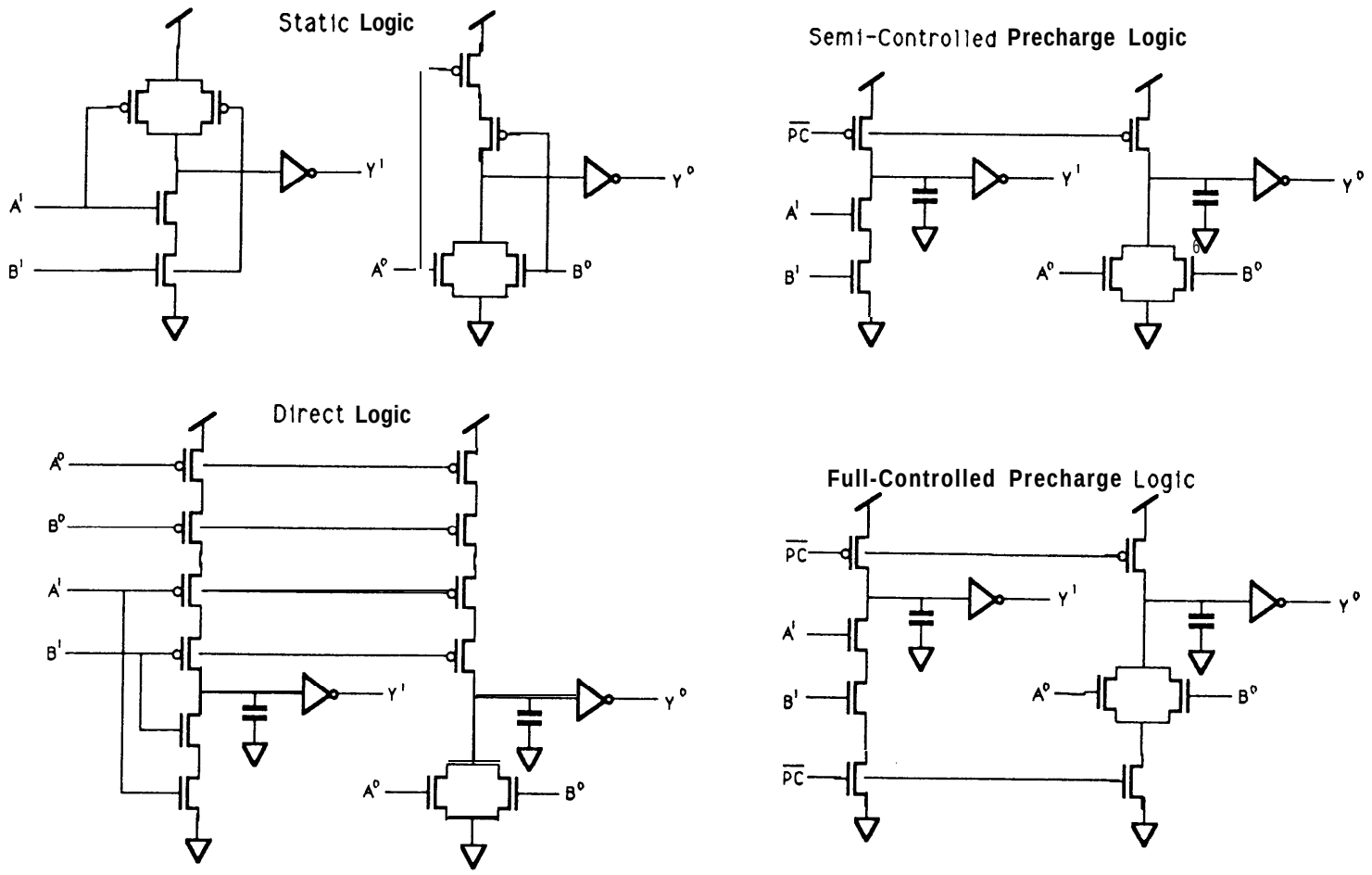

Figure 3: Four possible Function block styles to generate dual-monotonic outputs:
Static logic, Direct Logic, Semi-controlled Precharge Logic, Full-Controlled Precharge Logic

Fighting can occur, for example, in the later gates of a precharged stage which is internally composed of several serial domino gates with their precharge controls tied together. Because the later gates in the chain would not have their data inputs reset until the earlier gates had finished resetting, semi-controlled precharge logic would ripple reset data serially through the gates instead of allowing the gates to reset in parallel as they do with full-controlled precharge logic. Ratioing of semi-controlled precharge logic could also achieve parallel resetting of internal domino chains, but at the expense of large precharge transistors. The advantages of semi-controlled precharge logic are that it has fewer transistors, faster evaluate transitions due to shorter pull-down stacks, and lower loading of the precharge control input.

Both controlled precharged function blocks have a significant property not possessed by direct or static logic: after valid input data has returned to reset but before the precharge control is asserted, the block will "hold" valid outputs and this can provide the function of a latch without adding any additional transistors.

Explicit latches to be used in a self-timed pipeline can be constructed either as a traditional flow-latch or by using a C-element for each bit to make a "C-latch" [GREE88] as illustrated in Figure 4. The flow-latch passes the value of its data signal, D, to its output, Y, when its enable signal, E, is high, and keeps the output unchanged when the enable is low. The examples in this report will predominantly use the symmetric C-latch which passes a high data signal to its output when the enable is high, and a passes a low data signal when the enable is low, and otherwise leaves the output is unchanged. The C-latch is therefore delay-insensitive because it verifies that it has received a new transition on both data and control before changing its output. Conversely, the ordinary flow-latch is not delay-insensitive because there is no way to detect that the latch has indeed transitioned to the "holding" state when the enable is low. An additional advantage of the C-latch is its ability to function simultaneously as part of the control handshaking and as the datapath, whereas a flow-latch will always require control from a separate C-element in the handshaking logic.

Because of the asymmetry introduced by the "hold or pass" functionality of flow-latches or the edge triggering of registers (pairs of flow-latches), the proper control for self-timed pipelines using these structures may require "generalized" or asymmetric C-elements such as in Figure 5 which will be denoted by the symbol G. A generalized C-element [BURN87] has separate inputs triggering the rising and falling output transitions rather than using the same inputs for both transitions as in an ordinary C-element,

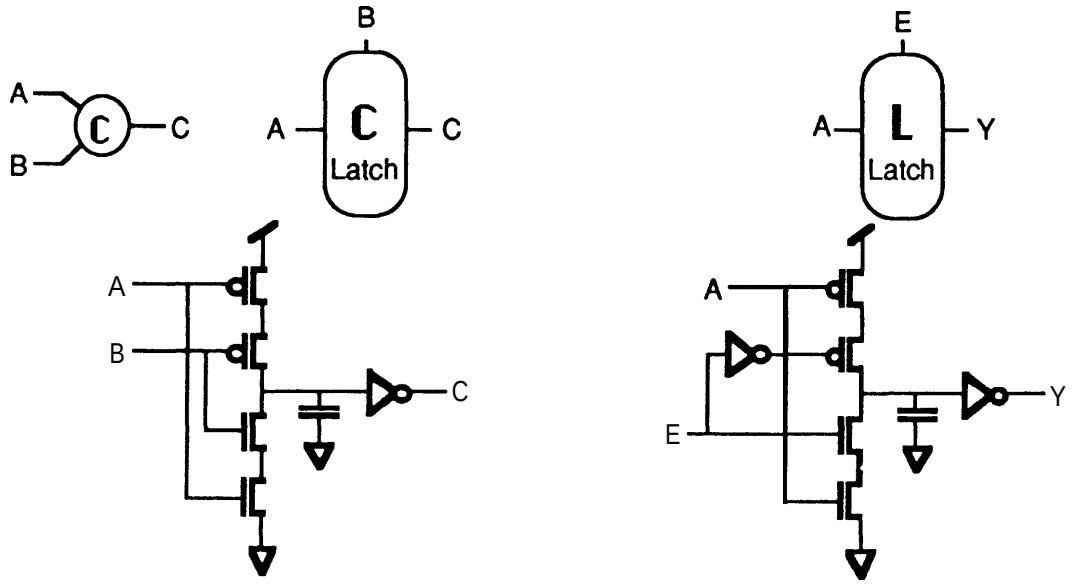

Figure 4: CMOS implementations of a C-element (or C-latch) and a standard flow latch.

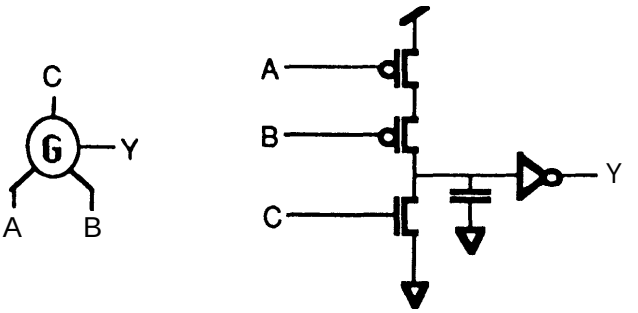

Figure 5: A "Generalized C-element" may have different signals triggering the rise and fall of the output. This figure shows a Generalized C-element with one input on the top to trigger the rising output transition, and two inputs on the bottom required for the falling transition.

III. Analysis technique using Dependency Graphs

Having defined the components composing each stage, it is now necessary to define the variables that describe the performance of the pipeline. Like the minimum clock period for a synchronous pipeline, let *P* denote the minimum cycle time for an asynchronous pipeline. *P* is determined, just as in the synchronous case, by the slowest stage. The throughput, T, is the reciprocal of *P*, and is the rate at which the input and output must respectively deliver and consume tokens to keep the pipeline flowing at its maximum capacity. In a synchronous pipeline, the delay from the output of one stage to the output of the next, or the per-stage latency, is equal to the clock period; whereas, in an asynchronous pipeline, it is an independent quantity which will be denoted by L. The total function latency is the sum of L through. all of the stages necessary to accomplish the desired function. Both synchronous and asynchronous pipelines can, of course, trade off total latency and throughput by changing the number of stages, N, into which the overall function is broken, but this report is concerned with the further choices available in pipeline configuration after the choice of N has been fixed.

In order to determine the cycle time, *P* , of a pipeline built out of a particular configuration of components in each stage, it is necessary to analyze the dependencies of the required sequence of transitions. These dependencies can be drawn in a marked directed graph where the nodes of the graph correspond to specific rising or falling transitions of circuit components, and the edges represent the dependencies of each transition on the outputs of other components. The delay of each transition is represented by a value attached to the corresponding node in the graph. These graphs will be called "Dependency Graphs" because of the similarly named graphs used in analyzing the cycle time of synchronous systems [RAO85]. Dependency Graphs are also similar to the signal transition graphs of [MENG89] and [CHU86], but differ by representing transitions on nodes instead of edges.

For an example to illustrate the construction of a Dependency Graph, Figure 6 shows the schematic of a simple stage style, both singly and arranged in a pipeline. The components of each stage instance are subscripted with an instance index. This stage style, denoted by the name PCO, uses a precharged function block controlled by a C-element which merges the "request" signal from the preceding stage with the "finished" signal from the following stage. The completion detector labelled D has a bubble on its output meaning that it will go low when there is evaluated data on its input bus, and high when the input bus is reset. A full completion detector uses a tree of C-elements to combine the outputs of a NOR gate on each dual montonoic pair in the databus.

The operation of the pipeline is intuitively straightforward from the control wire connections; the precharged block in each stage will reset when the data token it was holding has been used by the following stage, and when the preceding stage has finished resetting. Likewise, the precharged block will evaluate when there is valid data available from the preceding stage, and the following stage has completed resetting providing a new destination for the data token. Because the precharged function block and C-elements are symmetric for rising and falling transitions, the completion of every transition

is verified and the configuration is speed-independent for any value of component delays. Thus, the pipeline will operate correctly even if the stages along the pipeline do not have the same delays.

Dependency graphs to determine the cycle time are constructed by examining the schematic for both the rising and falling transitions of each component. For a component like a C-element which has symmetric dependencies for rising and falling transitions, its portion of the Dependency Graph will likewise be symmetric. A segment of the Dependency Graph for the **PC0** pipeline configuration is shown in Figure 7. Only a segment large enough to show the repeating pattern of the graph is drawn. The up-arrows (↑) and down-arrows (↓) in each name distinguish the rising and falling transitions of the components. The Dependency Graph, as thus constructed, is a simplification of the more general timed Petri-net description of asynchronous components [RAMC74]. Since the pipelines under consideration are deterministic, the Petri-net is decision-free and can therefore be equally well represented by such a marked graph.

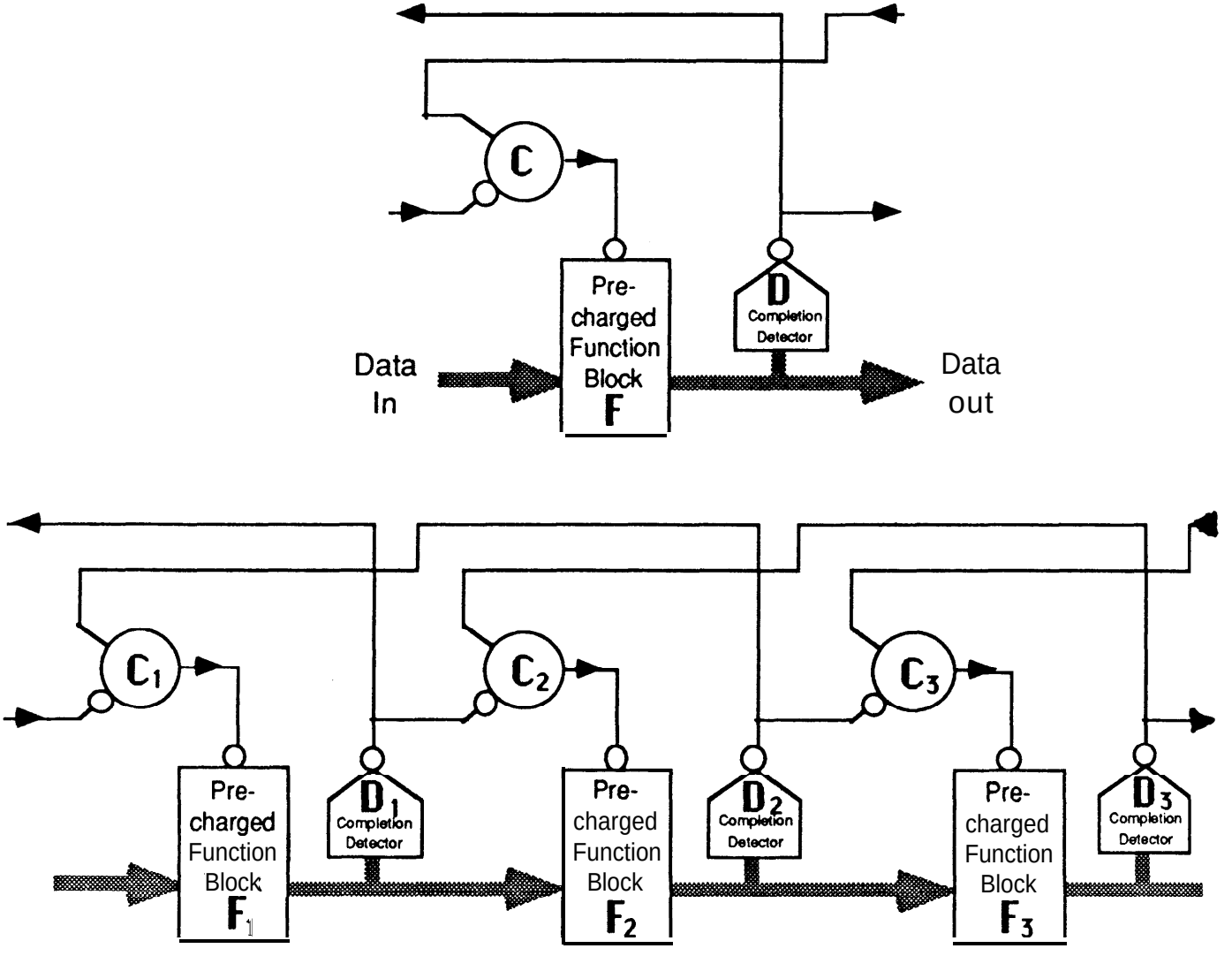

Figure 6: Schematic for stage style **PC0** and a short pipeline composed using that configuration.

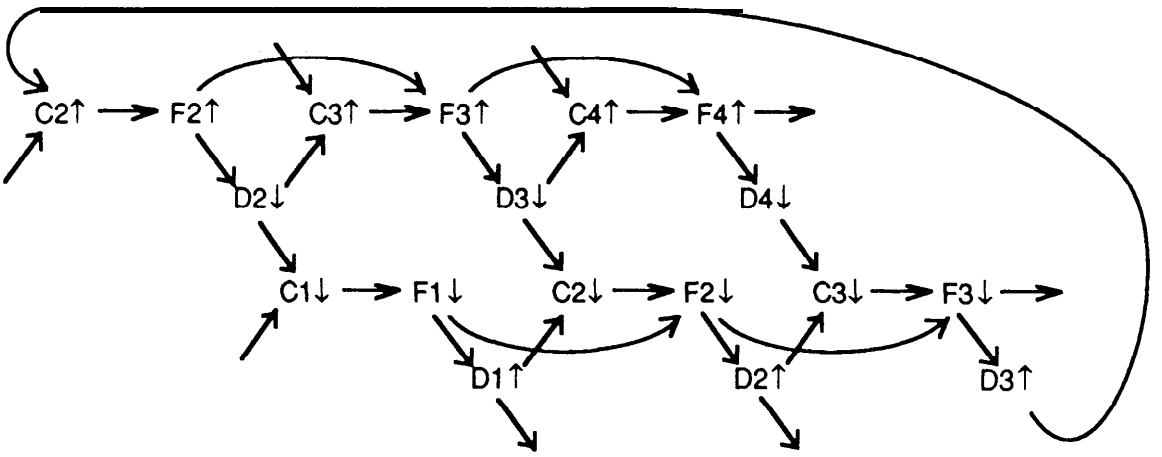

Figure 7: A portion of the Dependency Graph for the **PC0** configuration style pipeline.

As the pipeline processes successive data tokens, the components in each stage will go through a series of transitions and return to the same "state" as defined by the output values of each component. Tracing this sequence of transitions in the Dependency Graph shows cycles. Because the graph is "timed" by the delays at each node, positive-length cycles do not indicate impossible situations, but rather the sum of the node delay values around a cycle is a lower bound on the period required for the components to go through the sequence of transitions to process a successive token. Only if the sum were zero would there be a problemetic "dependency loop" akin to having a loop with no registers in the synchronous case.

Since each transition can fire only when all of its predecessors in the Dependency Graph have executed their specified transitions, all of the cycles through a node are lower bounds on the cycle time before this node can fire again. The actual cycle time will therefore be the sum of the delays of the longest cycle. The correct construction of each stage guarantees that all of the components in a stage will cycle at the same rate since every transition is part of some cycle in the Dependency graph. Thus, the longest simple cycle in the complete Dependency Graph gives the minimum cycle time of the pipeline as a whole. These results were proved in [RAMA80] for decision-free Petri-nets and the proofs for the Dependency Graph formulation would be analogous. As long as the input to the pipeline can supply tokens at that rate, the self-timed pipeline operates with cycle time, *P*, *equal* to the minimum cycle time determined by the Dependency graph.

For any specific pipeline configuration, the structure of the Dependency Graph repeats after each stage and this can be used to make the representation more compact. When the stages are identical and thus the delay values also repeat, the Dependency graph can be folded together to make a "Folded Dependency Graph," or simply "Folded Graph." An example of a Folded Graph is shown in Figure 8 for pipeline conguration type PCO. The nodes in the Folded Graph represent the transition delays as before but it is not necessary to subscript them with a particular stage index since the node represents that transition in all stages. Rather, each edge in the Folded Graph is annotated with an integer weight giving the offset in stage indices to which that dependency refers. Dependencies between components in the same stage thus have a weight of zero. Cycles in the Folded Graph whose edge weights sum to zero correspond to the cycles in the original Dependency Graph and thus the zero-weight cycle with the largest sum of node delay values gives the cycle time *P*. Like it was only necessary to examine simple cycles in the Dependency graph, it is not necessary to examine cycles in the Folded graph which pass through the same node more than once with the same cumulative edge weight.

For the particular example drawn in Figure 6, any zero-weight cycle must be the concatenation of the sequence $F\uparrow.D\downarrow.C\downarrow.F\downarrow.D\uparrow.C\uparrow$ which has edge weight -2 with two more trips through adjoining loops with edge weight +1. Since it is required to find the longest cycles, the self-cycles on the $F\uparrow$ and $F\downarrow$ nodes can be ignored because they are always shorter than the $D\downarrow.C\uparrow.F\uparrow$ and

$D\uparrow.C\downarrow.F\downarrow$ cycles. Therefore the following cycles are the possibilities for the longest zero-weight cycles:

$$F\uparrow.D\downarrow.C\downarrow.F\downarrow.D\uparrow.C\uparrow.F\uparrow.D\downarrow.C\uparrow.F\uparrow.D\downarrow.C\uparrow$$
$$F\uparrow.D\downarrow.C\downarrow.F\downarrow.D\uparrow.C\downarrow.F\downarrow.D\uparrow.C\uparrow.F\uparrow.D\downarrow.C\uparrow$$
$$F\uparrow.D\downarrow.C\downarrow.F\downarrow.D\uparrow.C\downarrow.F\downarrow.D\uparrow.C\downarrow.F\downarrow.D\uparrow.C\uparrow$$

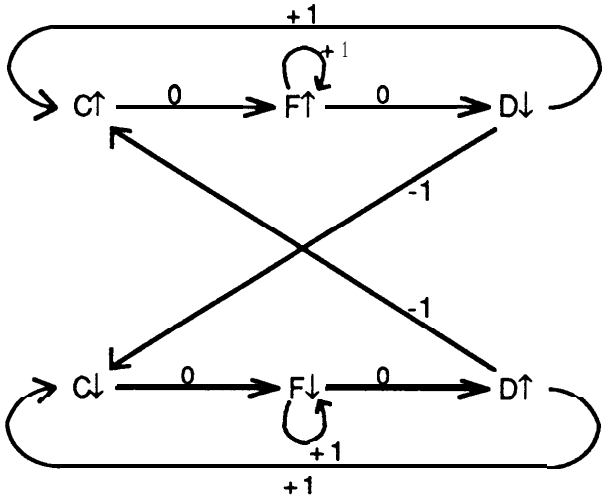

Figure 8: The Dependency Graph for the **PC0** configuration style pipeline can be folded together if the stages are identical to make the Folded Graph shown here.

The Folded Graph gives the same information as the original Dependency Graph but in a more compact form where symmetry is easier to see graphically. Either graph can be used for the analysis of a pipeline with identical stages. If the stages are not identical, then it should be emphasized that the full Dependency Graph needs to be drawn because the delay terms from the different stages need to be distinguished. The longest cycles in the full Depenency Graph will be the ones through the slowest stage and will hence correctly determine the limiting cycle time for the entire pipeline.

IV. Analysis of specific configurations

Though an actual implementation may have different function blocks in the stages composed together into a pipeline, this report will, for simplicity, make comparisions of pipelines composed of identical stages. After defining the notation, this section will write the exact equation for cycle time for each pipeline configuration to be considered, and then simplify the equation by applying a set of standard assumptions likely to be true, at least approximately, in real implementations. The configurations will be grouped in "families" based on their basic control format, where the family is composed of members with zero, one, two, or three additional latches per stage. The starting configuration for this section will be the speed-independent configuration family PC, a precharged function block controlled by a C-element which was used as the example in the Section III. A specific delay assumption will enable the formation of configuration family PS which does not require a C-element in the control. Next considered will be a transformation of the precharged function block into either the delay-insensitive CF or FC configurations which use a direct function block either followed or preceded by a latch, respectively. Finally, an alteration of the latch type to flow latches and registers will be examined in the PL family.

The nodes in a Dependency Graph represent the delays of particular transitions. These delays will be written in the equations by subscripting a lowercase t which signifies propagation time with a capital letter abbreviating the block type as follows:

F	Function blocks
D	completion Detectors
C	C-elements or C-latches
L	flow Latches
R	Register (pair of flow latches)
G	Generalized C-elements

If an up-arrow ($\uparrow$) or down-arrow ($\downarrow$) is specified after the block type, then the term refers specifically to the delay of the rising or falling transition. If no arrow is specified, then the delay refers to both transition directions.

For simplifying the equations, the following relationships are defined as the standard assumptions:

$t_D = t_{D\uparrow} = t_{D\downarrow}$	(Completion detectors delays are equal and symmetric)
$t_C = t_{C\uparrow} = t_{C\downarrow}$	(C-element delays are equal and symmetric)
$t_L = t_{L\uparrow} = t_{L\downarrow}$	(Latch delays are equal and symmetric)
$t_L = t_{R\uparrow} = t_{R\downarrow}$	(Registers have same delay from enable to output as latches)
$t_C = t_L = t_R$	(Registers and latches have same delay as C-elements)
$t_C = t_{G\uparrow} = t_{G\downarrow}$	(Generalized C-element delays are equal to C-elements)
$t_{F\downarrow} <= t_{F\uparrow}$	(Function resetting is no worse than evaluation)
$t_C <= t_{F\downarrow}$	(C-element delays are no worse than function resetting)

Section III described the analysis methodology using the **PC0** pipeline configuration shown in Figure 6 as an example. The control in this configuration enforces that it functions correctly as a pipeline for processing a stream of data tokens separated by reset "spacer" tokens and keeping the tokens separated. The C-element in each stage will enable the function block for evaluation when the inputs are valid and the following stage's outputs are finished resetting. After the stage has evaluated, the function block holds its outputs, until finally when its inputs have reset and when the outputs of the following function block have finished evaluating, the C-element in the control will precharge the function block. Thus, the **PC0** pipeline works correctly without any explicit latches because each of the precharged logic blocks is used as a latch "for free" to hold its output data during the interval after it has evaluated but before it is precharged again.

Unfortunately, the per stage latency, $L = t_{F\uparrow} + t_{C\uparrow} + t_{D\uparrow}$, of the **PC0** configuration includes the completion detector delay in the forward serial path because it must detect valid inputs before enabling the precharge block. The longest paths in the Folded Graph for the **PC0** configuration, shown in Figure 8, were enumerated in Section III and their maximum indicates that the configuration has a cycle time of:

$$P = t_{F\uparrow} + t_{F\downarrow} + t_{D\uparrow} + t_{D\downarrow} + t_{C\uparrow} + t_{C\downarrow} + 2 \max (t_{F\uparrow} + t_{C\uparrow} + t_{D\uparrow}, t_{F\downarrow} + t_{C\downarrow} + t_{D\downarrow})$$

which under the standard assumptions reduces to

$$P = 3t_{F\uparrow} + t_{F\downarrow} + 4t_C + 4t_D.$$

Extending the precharged function block configurations by adding C-latches can reduce the cycle time. Configuration **PC1** imposes a latch between the precharged function blocks as shown in the stage schematic in Figure 9. A second completion detector is also added of each stage to detect the status at the output of the C-latch and use this as the input to the control C-element. Configuration **PC1** has a per-stage latency $L = t_{F\uparrow} + 2\,t_{C\uparrow} + t_{D\uparrow}$ because of the added propagation time through the C-latch.

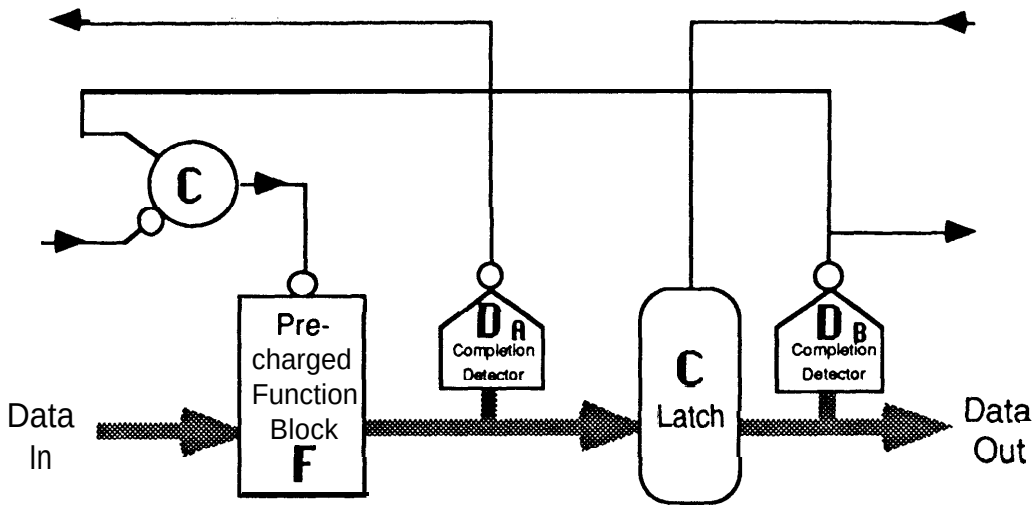

Figure 9: Schematic for stage style **PC1**

Analyzing the cycles in its Folded Graph shown in Figure 10 determine that a pipeline composed of identical stages in this configuration has a cycle time of:

$$P = t_{C\uparrow} + t_{C\downarrow} + t_{D\uparrow} + t_{D\downarrow} + \max\,[\, t_{F\uparrow} + t_{F\downarrow} + \max(\, t_{D\uparrow} + 2t_{C\uparrow}\,, t_{D\downarrow} + 2t_{C\downarrow},\; t_{D\uparrow} + t_{D\downarrow} + t_{C\uparrow} + t_{C\downarrow}\,),\; 2tC_{\uparrow} + t_{D\downarrow} + 2t_F\,, 2tC_{\downarrow} + t_{D\uparrow} + 2t_{F\downarrow}\,]$$

which under the standard assumptions reduces to $P = 4t_C + 3t_D + \max\,(\,2t_{F\uparrow}\,, t_{F\uparrow} + t_{F\downarrow} + t_D\,)$.

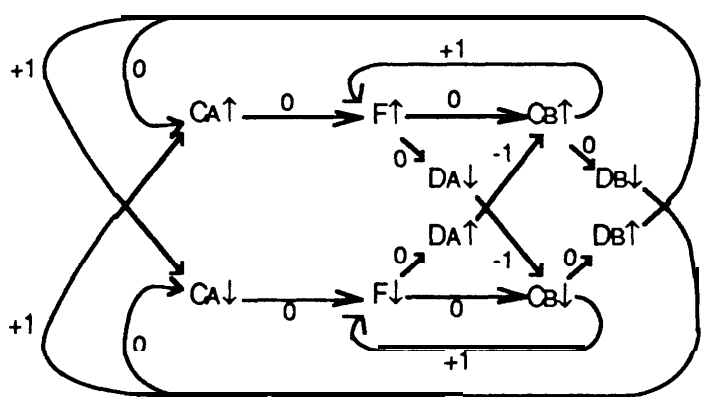

Figure 10: The Folded Dependency Graph for the **PC1** configuration pipeline

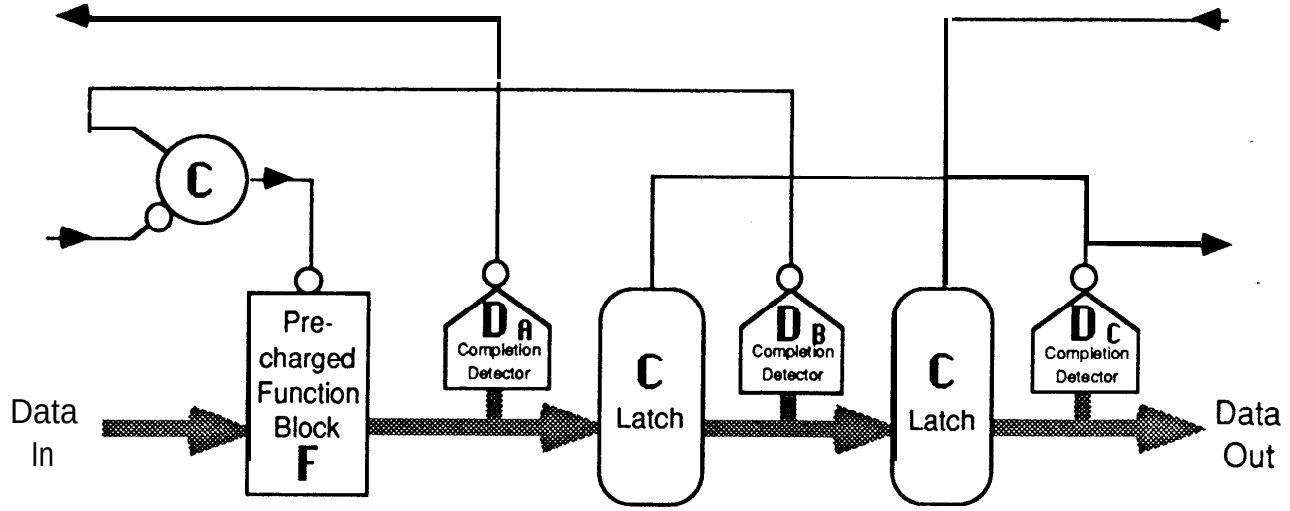

Figure 11: Schematic for stage style PC2

Configuration PC2 contains one more additional latch between the precharged function blocks as illustrated in Figure 11. Its per-stage latency is $L = t_{F\uparrow} + 3t_{C\uparrow} + t_{D\downarrow}$, and it has an even better cycle time of:

$$P = t_{F\uparrow} + t_{F\downarrow} + t_{C\uparrow} + t_{C\downarrow} + t_{D\uparrow} + t_{D\downarrow} + \max [\, 2t_{C\uparrow} + t_{D\downarrow}, 2t_{C\downarrow} + t_{D\uparrow}, t_{C\uparrow} + t_{C\downarrow} + t_{D\uparrow} + t_{D\downarrow} \,]$$

which reduces under the standard assumptions to $P = t_{F\uparrow} + t_{F\downarrow} + 4t_C + 4t_D$.

Because the C-element in the control of each stage in the PC pipeline configuration family explicitly enables the function block evaluation only after valid data is present on the inputs, this configuration works either with datapaths having embedded completion or with those having only a bundled completion signal. The C-element in the control will assure that the pipeline is speed-independent for both types of datapath. Further, the precharged function blocks can have either full-controlled precharge or a semi-controlled precharge because the control C-element will not apply the precharge signal until the inputs have reset. This means, of course, that even though the analysis in the previous paragraphs specified the timing which would result from a pipeline of identical stages, the pipeline would work correctly for any composition of individual stage reset or evaluation delays.

If bounds are available on some of the relative stage delays, then another stage style is possible which overcomes the poor latency characteristic of the PC family. Specifically, if it can be assumed that the resetting of each stage's neighboring stages are no slower than their evaluation, then the inputs to the C-element in the control will always come in a known order. Since the output of a C-element is always equal to the value of the input which last changed to be the same as the other input, an assumed ordering of the inputs makes the C-element redundant, and it can be just replaced with a wire from the input which is assumed to come last for both rising and falling transitions. The style formed by this transformation is called PS. Figure 12 shows the simplest stage in this family, PSO, which directly concatenates precharged function blocks, and has no C-elements at all. Its per-stage latency is $L = tF_{\uparrow}$, and the maximum of the delays of all the zero-weight cyclic paths in the Folded graph in Figure 13 determine the cycle time:

$$P = t_{F\uparrow} + t_{F\downarrow} + t_{D\uparrow} + t_{D\downarrow} + 2 \max (t_{F\uparrow}, t_{F\downarrow})$$

which under the standard assumptions reduces to $P = 3t_{F\uparrow} + t_{F\downarrow} + 2t_D$.

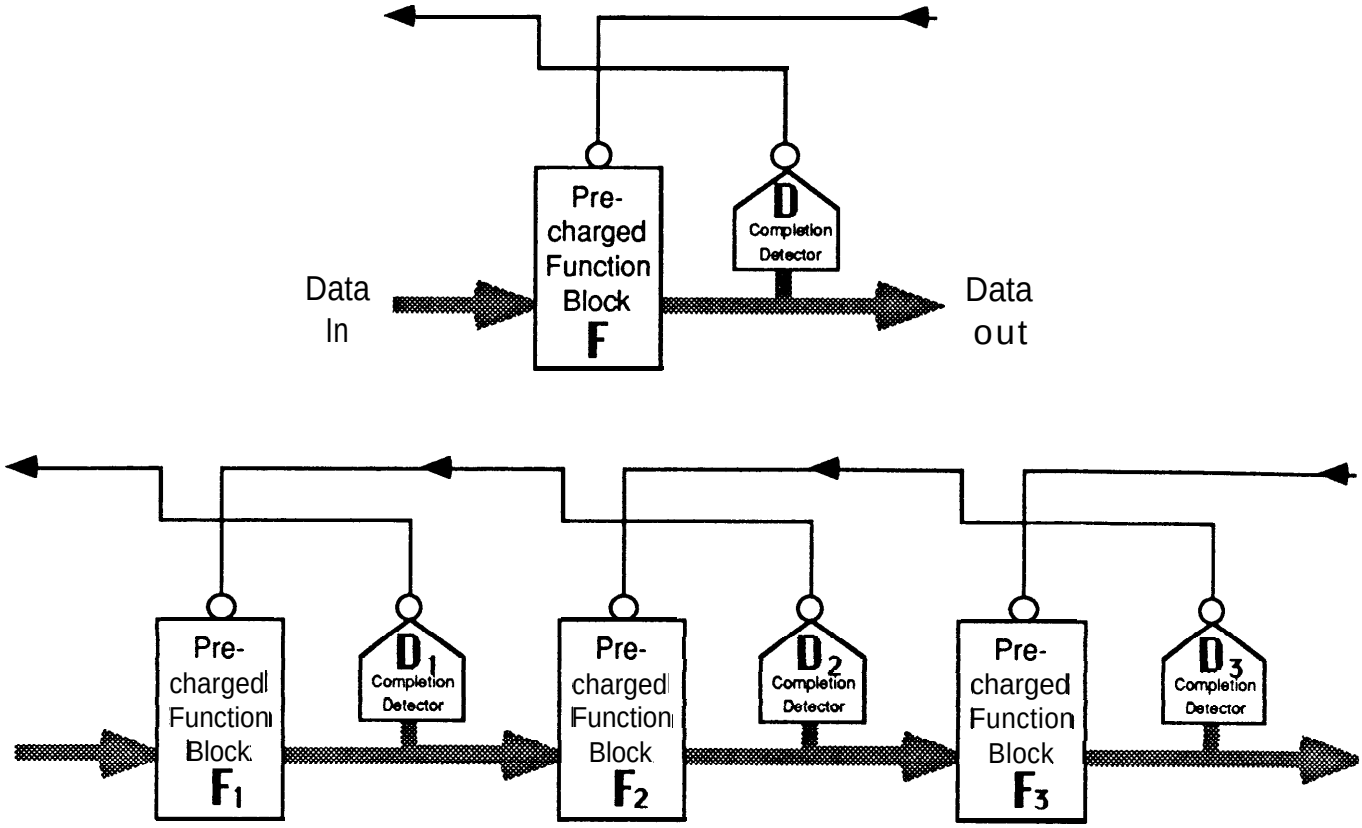

Figure 12: Schematic for stage style **PS0** and a short pipeline composed using that configuration.

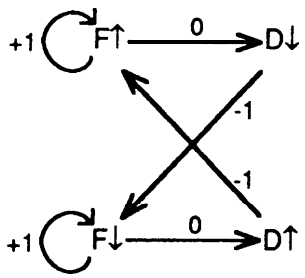

Figure 13: The Folded Dependency Graph for the **PS0** configuration pipeline

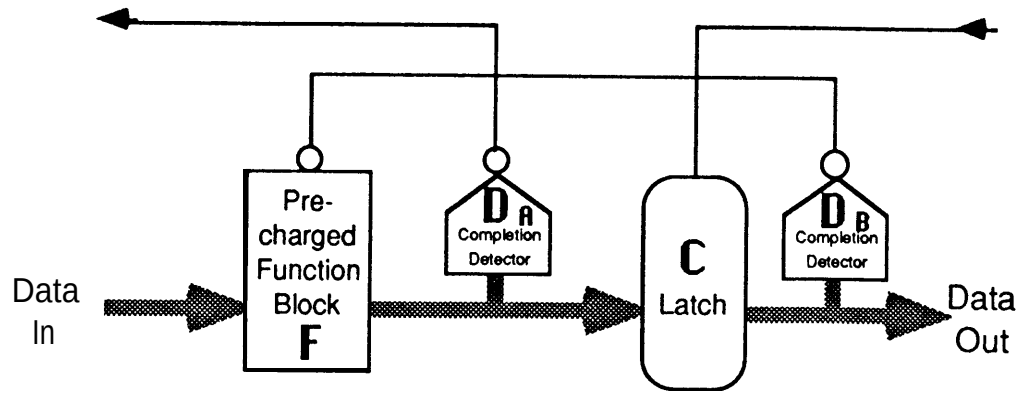

Figure 14: Schematic for stage style **PS1**

Configuration **PS1** adds one latch between the precharged function blocks and is shown in Figure 14. Its per-stage latency is $L = t_{F\uparrow} + t_{C\uparrow}$, and analyzing the cycles in its Folded Graph shown in Figure 15 determine it has a cycle time of:

$$P = t_{D\uparrow} + t_{D\downarrow} + \max [\, t_{F\uparrow} + t_{F\downarrow} + \max(\, 2t_{C\uparrow}\,,\, 2t_{C\downarrow}\,,\, t_{C\uparrow} + t_{C\downarrow}),$$
$$t_{C\uparrow} + t_{C\downarrow} + 2 \max (t_{F\uparrow}\,,\, t_{F\downarrow}\,) \,1$$

which under the standard assumptions reduces to $P = 2t_{F\uparrow} + 2t_C + 2t_D$.

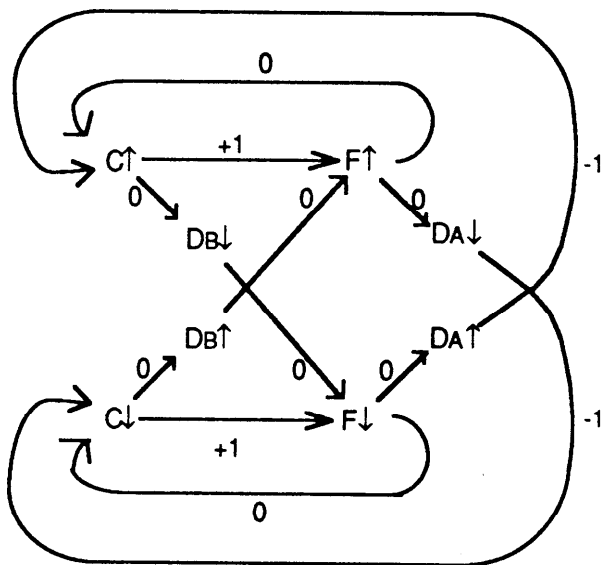

Figure 15: The Folded Dependency Graph for the **PS1** configuration pipeline

Configuration PS2 contains two latches between the precharged function blocks and is shown in Figure 15. Its per-stage latency is $L = t_{F\uparrow} + 2t_{C\uparrow}$, and it has an even better cycle time of:

$$P = t_{D\uparrow} + t_{D\downarrow} + \max [\, t_{F\uparrow} + 2t_{C\uparrow} + t_{C\downarrow}\,,\, t_{F\downarrow} + 2t_{C\downarrow} + t_{C\uparrow},$$
$$t_{F\uparrow} + t_{F\downarrow} + 2 \max (t_{C\uparrow},\, t_{C\downarrow}) \,]$$

which under the standard assumptions reduces to $P = t_{F\uparrow} + t_{F\downarrow} + 2t_C + 2t_D$.

Configuration PS3 contains three latches between the precharged function blocks and is shown

in Figure 16. Its per-stage latency is $L = t_{F\uparrow} + 3t_{F\uparrow}$, and it has cycle time:

$$P = t_{D\uparrow} + t_{D\downarrow} + \max [\ \max(t_{F\uparrow}, t_{F\downarrow}) + t_{C\uparrow} + t_{C\downarrow} + \max(t_{C\uparrow}, t_{C\downarrow}),$$
$$3 \max(t_{C\uparrow}, t_{C\downarrow}) + \min(t_{C\uparrow}, t_{C\downarrow}),$$
$$t_{F\uparrow} + t_{F\downarrow} + 2 \max(t_{C\uparrow}, t_{C\downarrow})\ ]$$

which under the standard assumptions reduces to the same cycle time as the previous case, $P = t_{F\uparrow} + t_{F\downarrow} + 2t_C + 2t_D$.

It should be emphasized that while the PC configuration family worked correctly for either bundled or embedded completion datapaths, the PS family requires embedded completion because the function blocks may be enabled for evaluation before valid data actually arrives. But in addtion to the improvement of not adding the C-element delays for the bundled signal, using embedded completion signals has the additional advantage of allowing the individual bits of a bus to begin evaluation individually as soon as their own inputs have arrived without having to wait for all of the bits in the bus.

Both the PC and PS families illustrate the clear tradeoff between latency and throughput, but they also illustrate an additional feature not relevant in synchronous pipelines: the dependence on the relative sizes of $t_{F\uparrow}$ and $t_{F\downarrow}$. If the precharge time is about equal to the evaluation time of the function blocks, then adding a second latch does not help, but if $t_{F\downarrow} << t_{F\uparrow}$ then a second latch helps significantly. Adding a third latch helps neither the throughput nor the latency and therefore has no advantage.

There are two different ways to substitue a direct function block for a precharged function block, and they differ significantly in perfromance. For an analogous analysis, both substitutions will replace the precharged function block with a direct function block together with a C-latch, but the difference is as to whether the latch comes before or after the direct function block as illustrated in Figure 16. If the precharged function block of the PS family is replaced by a direct function block preceded by a latch, then the resultant configuration is called CF. The simplest member of this family, CFO, is shown in Figure 17. The operation of the pipeline is straightforward from the control wire connections; the latch in each stage will reset when the data token it was holding has passed down the pipe and is no longer needed. A token is known to be no longer needed when it has passed through the second function block following a latch. Observe that only waiting for the token to pass through the function block immediately after a latch would not verify that the token had passed through the latch following that function block. Because the direct function blocks and C-elements are symmetric for rising and falling transitions, this control connection also correctly enables a latch to accept new valid data only when the successor is verified to have reset. The CF family will therefore be delay-insensitive for any delay values of the components without requiring the stages along the pipeline to be identical. The CFO configuration has a per-stage latency of $L = t_F + tC_{\uparrow}$, and the maximum of the delays of all the zero-weight cyclic paths in the Folded graph shown in Figure 18 determines the cycle time:

$$P = t_{F\uparrow} + t_{F\downarrow} + t_{D\uparrow} + t_{D\downarrow} + t_{C\uparrow} + t_{C\downarrow} + 2 \max(t_{F\uparrow} + t_{C\uparrow}, t_{F\downarrow} + t_{C\downarrow})$$

which under the standard assumptions reduces to $P = 3t_{F\uparrow} + t_{F\downarrow} + 4t_C + 2t_D$.

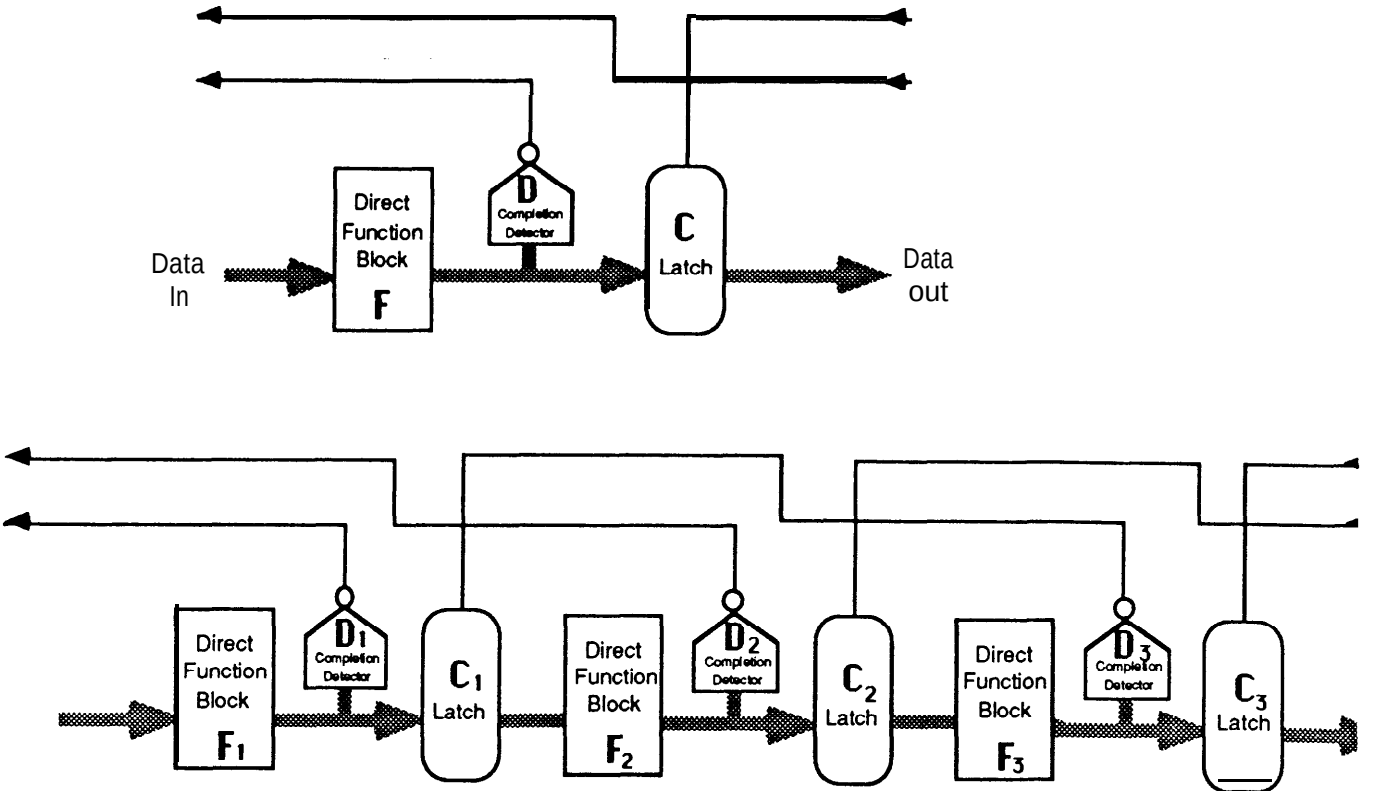

Figure 17: Schematic for stage style **CFO** and a short pipeline composed using that configuration.

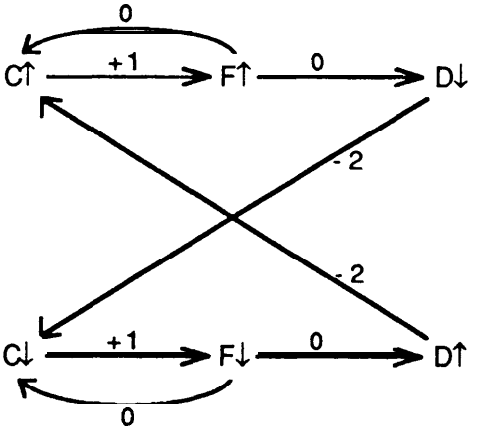

Figure 18: The Folded Dependency Graph for the **CFO** configuration pipeline

Configuration **CF1** imposes an additional latch between the direct function blocks. Its per-stage latency is $L = t_{F\uparrow} + 2\,tC_{\uparrow}$, and analyzing the cycles in its Folded Graph show that this configuration has a cycle time of:

$$P = t_{D\uparrow} + t_{D\downarrow} + \max[\, t_{F\uparrow} + t_{F\downarrow} + t_{C\uparrow} + t_{C\downarrow} + \max(2t_{C\uparrow}, 2t_{C\downarrow}, t_{C\uparrow} + t_{C\downarrow}),$$
$$t_{C\uparrow} + t_{C\downarrow} + 2\max(t_{F\uparrow} + t_{C\uparrow}, t_{F\downarrow} + t_{C\downarrow})\,1$$

which under the standard assumptions reduces to $P = 2t_{F\uparrow} + 4t_C + 2t_D$.

Configuration CF2 contains three latches between the direct function blocks. Its per-stage

latency is $L = t_{F\uparrow} + 3t_{C\uparrow}$, and it has an even better cycle time Of:

$$P = t_{D\uparrow} + t_{D\downarrow} + \max [\ t_{F\uparrow} + 3t_{C\uparrow} + t_{C\downarrow}\ ,\ t_{F\downarrow} + 3t_{C\downarrow} + t_{C\uparrow},\ t_{F\uparrow} + t_{F\downarrow} + t_{C\uparrow} + t_{C\downarrow} + 2 \max (t_{C\uparrow}, t_{C\downarrow})\ ]$$

which under the standard assumptions reduces to $P = t_{F\uparrow} + t_{F\downarrow} + 4t_C + 2t_D$.

Configuration CF3 contains four latches between the direct function blocks. Its per-stage latency is $L = t_{F\uparrow} + 4t_{C\uparrow}$, and it has cycle time:

$$P = t_{D\uparrow} + t_{D\downarrow} + \max [\ \max (\ t_{F\uparrow}, t_{F\downarrow}\ ) + t_{C\uparrow} + t_{C\downarrow} + 2 \max (t_{C\uparrow}, t_{C\downarrow}),\ 3\ \max (t_{C\uparrow}, t_{C\downarrow}) + \min (t_{C\uparrow}, t_{C\downarrow}\ ),\ t_{F\uparrow} + t_{F\downarrow} + t_{C\uparrow} + t_{C\downarrow} + 2 \max (t_{C\uparrow}, t_{C\downarrow})\ ]$$

which under the standard assumptions reduces to the same cycle time as the previous case,

$$P = t_{F\uparrow} + t_{F\downarrow} + 4t_C + 2t_D.$$

Comparing the latency and throughput for the CFO, **CF1**, and CF2 configurations to those determined previously for the PSO, **PS1**, and PS2 configurations, it is seen that they differ only in the obvious way that t_F is replaced by $t_C + t_F$.

One of the basic operations in retiming synchronous circuits is to move registers from one side of a function block to the other which can change the timing and performance without changing the total number of registers. In the CF configuration family each direct function block was preceded by a latch and followed by a completion detector. If the latches are "pushed" to the other side of the function blocks then the pipeline will be resequenced so that each direct function block is preceded by a completion detector and followed by a latch. The family thus created is called FC and maintains the property of being delay-insensitive for correct logical operation. The formation of the FC family could equivalently be described as the other transformation of the PS family, where a direct function block followed by a C-latch is substituted for each precharged function block

The FCO configuration shown in Figure 19 has a per-stage latency of $L = t_{F\uparrow} + t_{C\uparrow}$, and the maximum of the delays of all the zero-weight cyclic paths in the Folded graph shown in Figure 20 determines the cycle time:

$$P = t_{D\uparrow} + t_{D\downarrow} + \max [\ t_{F\uparrow} + t_{F\downarrow} + t_{C\uparrow} + t_{C\downarrow} + \max(\ 2t_{C\uparrow}\ ,\ 2t_{C\downarrow}\ ,\ t_{C\uparrow} + t_{C\downarrow}),\ t_{C\uparrow} + t_{C\downarrow} + 2 \max (t_{F\uparrow} + t_{C\uparrow}\ ,\ t_{F\downarrow} + t_{C\downarrow})\ ]$$

which under the standard assumptions reduces to $P = 2t_{F\uparrow} + 4t_C + 2t_D$.

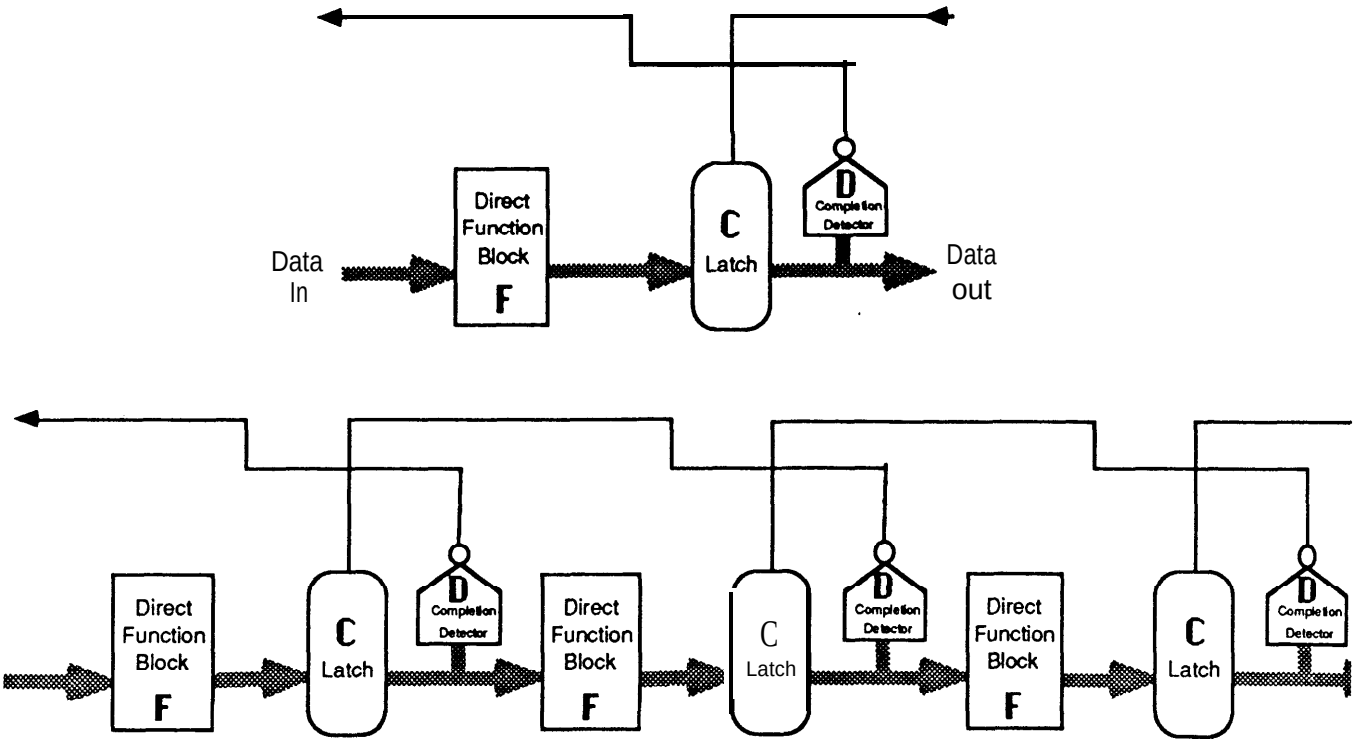

Figure 19: Schematic for stage style **FCO** and a short pipeline composed using that configuration.

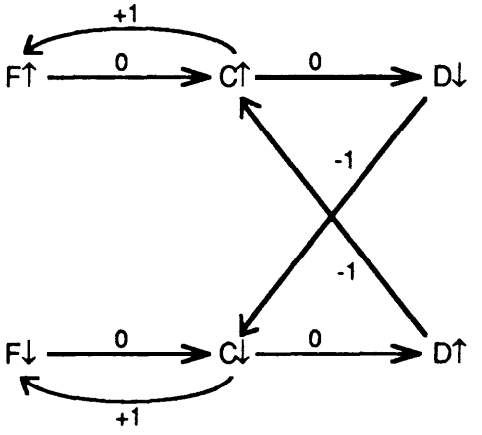

Figure 20: The Folded Dependency Graph for the **FCO** configuration pipeline

The **FCO** example shows the per-stage latency, *L*, of the members of the **FC** family will be the same as for the **CF** family; however, the cycle time, *P*, for any specific delay values will be improved. The **FCO** cycle time has, in fact, the same coefficients as obtained for the **CF1** configuration even though **FCO** has one fewer latch per stage. Likewise, **FC1** and FC2 will have cycle times corresponding to those found for **CF2** and **CF3. So,** in the **FC** family where the latches follow the function blocks, if the precharge time of the function blocks is about equal to the reset time, then using **FC1** is no better than **FCO,** but if $t_{F\downarrow} << t_{F\uparrow}$ then **FC1** is an improvement. The **FC2** and **FC3** configurations provide no further improvement.

The PC configuration family will be the base for transforming the latch used from the C-latch to the flow-latch or register structure more commonly used in synchronous circuits. Since flow latches and registers do not actively reset their outputs but rely on the function block to reset, only the full-controlled precharged logic block makes sense and substitution of these types of latches with the other function block types will not be considered.

Configuration **PL1** is illustrated in Figure 21 and corresponds to replacing the C-latch in PC1 with a flow latch and modifying the control appropriately with generalized C-elements to provide the asymmetric control to the flow-latch. The control is asymmetric because it must pass when its inputs are reset and its successor is evaluated, but can block as soon as its outputs are evaluated. The function block itself can enable evaluation when its latch outputs are reset, but must wait before resetting until its inputs are reset and the latch is blocked. Configuration **PL1** has a per-stage latency $L = t_{F\uparrow} + t_{L\uparrow}$, and a cycle time of:

$$P = t_{F\uparrow} + t_{L\uparrow} + t_{L\downarrow} + t_{D\uparrow} + t_{D\downarrow} + t_{G_A\uparrow} + \max\left[t_{F\uparrow} + t_{G_B\uparrow}, t_{F\downarrow} + t_{G_A\downarrow} + t_{G_B\downarrow}, t_{F\downarrow} + t_{L\downarrow} + t_{G_B\uparrow} \right]$$

which under the standard assumptions reduces to $P = 2t_{F\uparrow} + 2tL + 2t_C + 2t_D$. This cycle time is similar to that for **PC1**, but with the deletion of the extra $2t_D$ for requiring the detection of completion within the forward path.

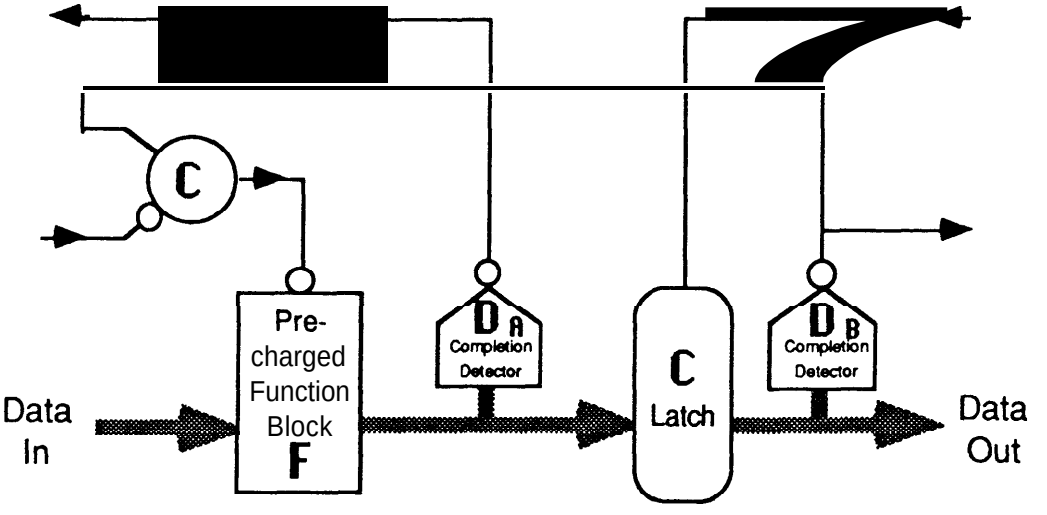

Figure 21: Schematic for stage style **PL1**.

Replacing the C-latches in configuration PC2 with flow latches paired together to make a register yields configuration **PL2** which is illustrated in Figure 22. It has per-stage latency $L = t_F + tR_\uparrow$, and a cycle time under the standard assumptions of $P = t_{F\uparrow} + t_{F\downarrow} + 4t_C + 2t_D$ which is similar to the P for **PC2**, but with the deletion of the extra $2t_D$ for requiring the detection of completion within the forward path. So, the timing dependencies of the other latch types as used in the PL family are similar to those found for the PC family, but without being speed-independent.

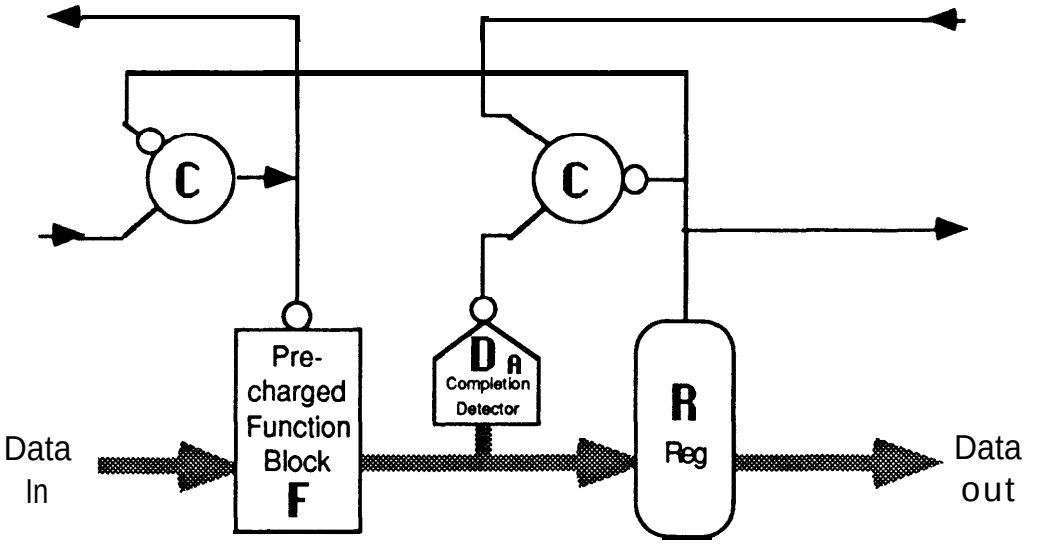

Figure 22: Schematic for stage style **PL2.**

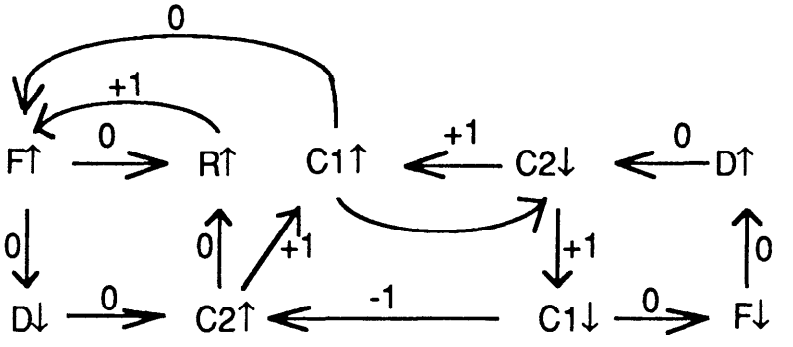

Figure 23: The Folded Dependency Graph for the **PL2** configuration pipeline

V. Summary of Latency and Throughput Comparisons

The equations derived in the previous sections giving latency and throughput under the standard assumptions were all in terms of four variables: $t_{F\uparrow}$, $t_{F\downarrow}$, t_C, and t_D. To clearly compare the different configurations, Table 2 summarizes the equations for the cycle time and per-stage latency by showing the coefficients of these four variables.

Config	Class	Cycle Time Coefficients				Latency Coefficients		
		$t_{F\uparrow}$	$t_{F\downarrow}$	t_C	t_D	$t_{F\uparrow}$	$t_{C\uparrow}$	$t_{D\uparrow}$
PC0	SI	3	1	4	4	1	1	1
PC1	SI	2	0	4	4	1	2	1
PC2	SI	1	1	4	4	1	3	1
PC3	SI	1	1	4	4	1	4	1
PS0		3	1	0	2	1	0	0
PS1		2	0	2	2	1	1	0
PS2		1	1	2	2	1	2	0
PS3		1	1	2	2	1	3	0
CFO	DI	3	1	4	2	1	1	0
CF1	DI	2	0	4	2	1	2	0
CF2	DI	1	1	4	2	1	3	0
CF3	DI	1	1	4	2	1	4	0
FCO	DI	2	0	4	2	1	1	0
FC1	DI	1	1	4	2	1	2	0
FC2	DI	1	1	4	2	1	3	0
PL1		2	0	4	2	1	1	0
PL2		1	1	4	2	1	2	0

Table 2 : Coefficients of delay equations under standard assumptions

Under many cases, it may be possible to make further simplifying assumptions to compare the different configurations. Table 3 shows the results of applying the simplifications at the tops of its columns to the coefficients previously summarized in Table 2. The numbers for both latency and throughput are normalized by $t_{F\uparrow}$, the function block evaluation time. Since all the numbers represent delays, the smaller numbers are always better. The first two columns of Table 3 give the latency and throughput assuming that $t_{F\downarrow}=t_C=t_D = 0$ which is nearly the case in the extreme of large function blocks which are composed internally of several precharged domino stages. Since the stages evaluate in series but reset in parallel, the reset time will be much less than the evaluate time, and could justifiably be approximated as being zero. The table is not filled in for the static logic cases in these columns because these assumptions would not apply in those cases. The middle two columns in Table 2 are for the case of $t_C = t_D = 0$ while $t_{F\downarrow} = t_{F\uparrow}$ which would be appropriate for large function blocks where the reset time is comparable to the evaluate time. The last two columns are for the case of $t_{F\downarrow}=t_C=t_D=t_{F\uparrow}$ which would be an appropriate assumption for very small function blocks where both the function blocks and the latches were all just a gate delay. Real applications would, of course, have throughput and latency somewhere between the extremes listed in this table.

Pipeline Config Style	Big Composite F Block		Big Serial F Block		Small F Block	
	$tF_{\downarrow}=0$		$tF_{\downarrow}= tF_{\uparrow}$			
	$t_C = t_D = 0$				$t_C = t_D = tF_{\uparrow}$	
	P	*L*	*P*	*L*	*P*	*L*
PC0	3	1	4	1	12	3
PC1	2	1	2	1	10	4
PC2	1	1	2	1	10	5
PC3	1	1	2	1	10	6
PS0	3	1	4	1	6	1
PS1	2	1	2	1	6	2
PS2	1	1	2	1	6	3
PS3	1	1	2	1	6	4
CFO			4	1	10	2
CF1			2	1	8	3
CF2			2	1	8	4
CF3			2	1	8	5
FCO			2	1	8	2
FC1			2	1	8	3
FC2			2	1	8	4
PL1	2	1	2	1	8	2
PL2	1	1	2	1	8	3

Table 3 : Cycle Time and Latency under further simplifications
(Normalized to $t_{F\uparrow}$, the function block evaluate delay)

VI. Dynamic and Static Spreads, and Flow Rates

Together the latency and throughput determine the number of stages over which each token "spreads" in an asynchronous pipeline. If the input supplies tokens as fast as the pipeline consumes them and the output takes tokens as fast as the pipeline supplies them, then in the steady state the input and output rates will match and the number of tokens in an N stage pipeline will be $NLT = NL/P$. *The* dynamic spread in stages between tokens in the flowing pipeline can therefore be defined as $D=P/L$. The reciprocal of the dynamic spread, D, is the dynamic occupancy or "utilization" of the pipeline. This tells how effectively the stages are being used in parallel. For example, if D = 2, then the utilization of one-half means that only every other stage in the pipeline can be simultaneously evaluating.

If the output of a pipeline is blocked so that the pipeline fills up with tokens and stops, then another important quantity is the static spread in stages between tokens. The static spread, denoted by S is the reciprocal of the "packing density" of the pipeline, and is, of course, determined by the connectivity of the components but not their delays. The static spread is important if the application requires the pipeline to also provide a buffer queue for a specified number of tokens during brief periods of I/O mismatch. Both the dynamic and static spreads for the various pipeline configurations are shown in Table 4, where, of course, smaller numbers are better. The numbers are in units of stages/token; however, the stages are not necessarily of constant area since the stages which include more latches will, of course, be larger. The numbers provide a fair indication of the relative areas of the different configurations if the function block area is large compared to the latch area.

Config	$t_{F\downarrow}= 0$	$t_{F\downarrow}=t_{F\uparrow}$		Static Spread
	$t_C = t_D = 0$		$t_C = t_D = t_{F\uparrow}$	
	D	D	D	S
PC0	3.00	4.00	4.00	2.00
PC1	2.00	2.00	2.50	1.00
PC2	1.00	2.00	2.00	0.67
PC3	1.00	2.00	1.67	0.50
PS0	3.00	4.00	6.00	2.00
PS1	2.00	2.00	3.00	1.00
PS2	1.00	2.00	2.00	0.67
PS3	1.00	2.00	1.50	0.50
CF0		4.00	5.00	2.00
CF1		2.00	2.67	1.00
CF2		2.00	2.00	0.67
CF3		2.00	1.60	0.50
FC0		2.00	4.00	2.00
FC1		2.00	2.67	1.00
FC2		2.00	2.00	0.67
PL1	2.00	2.00	4.00	
PL2	1.00	2.00	2.67	1.00

Table 4 : Dynamic and Static spreads in units of stages/token

At any instant in a pipeline, the stages not occupied by the data tokens or their intervening reset spacers can be said to contain a "bubble." Bubbles are introduced at the output by the consumption of data tokens. Like holes in a semiconductor, the bubbles flow backwards as the data and reset spacers flow forwards. But unlike a synchronous pipeline, which can flow when fully packed, an asynchronous pipeline can be limited by the supply of bubbles. In fact, there must be bubbles for an asynchronous pipeline to flow at all. In a pipeline of N stages with static occupancy S from Table 4, there can be up to N/S tokens; however, if the pipeline has nearly that many tokens then it will flow at a rate wholly limited by the "backwards latency" of bubbles introduced at the output rather than the "forward latency" of data tokens introduced at the input.

An asynchronous pipeline will be able to achieve its maximum flow rate, given by the reciprocal of the cycle time P, when it flows uniformly throughout its length. For this flow rate to be supported, there must be enough bubbles. This required number of bubbles for maximum flow is given by

$$N\ (\frac{1}{S}\ -\ \frac{1}{D}\ ),$$

the difference of the reciprocals of the static and dynamic spreads. If there are fewer bubbles than this, then the pipeline will not be able to flow at the maximum rate, and while momentary input or output rates in excess of the overall flow rate can occur, they will result in a non-uniform distribution of tokens along the pipeline.

VI. Application to Rings

A pipeline which recirculates its output back around to its input can form a loop, or ring, which cycles wholly under self-timed control. The rate at which the ring cycles, or iterates, will be determined by the configuration and delays of the stages, independent of any external control signals. Rings can

circulate one or several tokens. Although in general, the rings may go through merge or join stages which introduce or consume additional tokens as the tokens flow around the ring, this report will examine the simpler case of a ring where the number of tokens is fixed once the ring has been initialized. So although a ring certainly requires a means for initialization and output, they can be ignored in discussing the fundamental iteration time of the ring. The dominant consideration in evaluating the ring's performance is therefore the overall function latency which will be determined by the rate at which the tokens are able to flow around the ring.

The rate at which tokens flow around the ring is limited by two factors. The first is the fundamental per-stage latency of the particular stage configuration chosen, as given in Tables 2 and 3. A lower bound on the overall function latency is ***NLR***, the number of stages in the ring times the per-stage latency times R, the number of times around the ring necessary to accomplish the desired function.

The other consideration giving a lower bound on the overall latency is the possibility of being limited by waiting for the control handshakes or the propagation of bubbles. Such waiting can occur if there are so few stages in the ring that the local handshakes between adjacent stages have dependencies which affect each other. To keep the dependencies from introducing additional constraints, there needs to be enough stages in the ring to support the desired number of data tokens and spacers, as well as additional extra space which forms the "bubbles" around the tokens. If there is not enough bubble space around the tokens then the ring will cycle at a reduced rate because the extra dependencies cause reduced parallelism between the stages. Optimally, data tokens will circulate through the ring at the same rate they could flow down a pipeline using the same stage configuration. The minimum number of stages needed in the ring for this optimal flow is given by ***TD,*** where ***T*** is the number of data tokens desired and ***D*** is the dynamic spread from Table 4 for the particular pipeline configuration. The dynamic spread thus specifies the average number of stages occupied by a data token and its accompanying reset spacer along with enough bubble space so that it can flow unimpeded by the other tokens. If there are ***TD*** or more stages then the rate at which the tokens flow around the ring will be limited only by the fundamental per-stage latency.

An important application of self-timed rings is their usage in solving a single function which requires the recursive evaluation of an operation many times. In this case, the stages in the ring can form the steps of the operation or can repeat the operation several times as computation progresses around the ring. For this application, it is usually desired to concentrate on solving just one problem at a time and therefore there need only be one data token circulating around the ring. For just a single token, the minimum number of stages in the ring is simply ***D, the*** dynamic spread.

Actual implementions of single-token self-timed rings have been applied to evaluating the arithmetic function of division of normalized fractions in [WILL87] and [WILL91]. Both of these designs chose the PSO configuration because of its minimal latency and simple control. By directly concatenating the functional blocks, the per-stage latencies in this configuration come solely from the raw combinational logic. The design in [WILL87] used only three columns and was unfortunately limited by the cycling of the control circuitry because the particular value of dynamic spread incurred by the control logic was nearly 4.5 stages. The second design in [WILL91] contained several improvements which lessened the absolute value of ***L,*** the per-stage latency, and also chose to use five stages. Experimental measurements found the actual value of the dynamic spread was about 4.2 stages. Therefore by using 5 stages, the rate at which data flows around the ring in this circuit is limited solely by the fundamental per-stage latency ***L*** since the control logic never enters into the critical path of the data tokens.

VII. Conclusions and Further Work

Using Dependency graphs, a methodology has been prescribed to quickly determine the exact throughput and latency for deterministic self-timed pipelines. Applying this method to examples has led to useful tables for comparison of self-timed pipeline configurations. These comparisons could be used by synthesis tools [CHU86][MENG89] to choose from a wider range of possible circuits depending on

specific delay considerations.

For ordinary pipelines, throughput is usually the dominant consideration. From Table 1 it can be seen that the best choice for a high-throughput pipeline may indeed vary depending on the actual ratios of t_C and $t_{F\downarrow}$ to $t_{F\uparrow}$. Likely good choices for pipeline configurations are **PS2**, **PL2**, and **FC1**. The latter uses a direct function block style rather than a precharged function block and can save the cost of a completion detector per stage while achieving similar throughput, but it is important that the remaining completion detectors following the latches and not the function blocks. Using direct function blocks is most significant if a completion detector has a high cost in area, because for example of a large bus width.

Latency is the most important concern for self-timed rings. The direct concatenation of precharged function blocks, configuration **PSO,** is the best because it adds no additional latency over the functional blocks themselves, and it will, moreover, be compact because of the absence of control logic. However, this configuration has a higher dynamic spread than others which will increase the number of stages necessary to keep the ring from being slowed by additional dependencies between the stages. If area limitations are important than choosing configuration **PS1** might be a good compromise between latency and area for a ring. Evaluating the actual performance for a particular ring implementation requires specific information about the relative delays of the control elements and the function blocks. The Dependency Graph analysis can used to evaluate the possible problem of extra dependencies in the ring by drawing the graph for the whole ring rather than just a pipeline segment. Examining the design in this way will allow the comparison of possible alternatives in defining the stage boundaries and control logic connections.

Although this report is concerned primarily with four-phase value-encoded systems, many of the results will also apply to two-phase transition encoded systems [SUTH89] with appropriate modifications to the function blocks [DEAN90]. In particular, the last four columns of Tables 2 and 3 have the assumption $t_{F\downarrow}=t_{F\uparrow}$ which would be appropriate for two-phase pipelines where both transistions convey a useful data token. In the two-phase case, the latency remains the same but the throughput is doubled. Likewise, the static and dynamic occupancies would both be doubled, and thus the static and dynamic spreads shown in Table 4 would both be halved. More work will be needed for alternate control or function block styles designed specifically for two-phase pipelines.

The performance analysis in this report was based on fixed component delays. An extension would be to consider stochastic delays with specified probabilistic distributions. Such an analysis has been performed in [GREE88] for a very abstract model, closest to **PSO** in this report, but could be extended to the other pipeline configurations suggested here. Such stochastic models would more accurately reflect situations where delays are more variable and unknown. Unfortunately, the results of a stochastic delay model will always be worse than the case of fixed delays because the former can only introduce additional waiting and pipeline stalling in a speed-independent circuit.

Acknowledgements

The author wishes to thank David Dill, Mark Horowitz, Theresa Meng, and especially Steve Nowick for their useful insights and comments in reviewing drafts of this report.

References

[BURN87] Bums, S., "Automatic Compilation of Concurrent Programs into Self-timed Circuits," M.S. Thesis, Caltech, Dec. 1987.

[CHU86] Chu T.A., "Synthesis of Self-timed Control circuits from Graphs: An example," ***Proceedings of ICCD,*** pp. 565-57 1, October 1986.

[DEAN90] Dean M., Williams T.E., Dill D., "Transition Dual Rail, An Alternate Dual-Rail Encoding Scheme for Self-Timed Logic," work in progress, Stanford University, 1990.

[EBER88] Ebergen J.C., "Transforming Programs into Delay-Insensitive Circuits," Ph.D. Thesis, Eindhoven Tech. Univ., 1988.

[GREE87] Greenstreet M., Williams T.E., Staunstrup J., "Self-Timed Iteration," ***Proceedings of VLSI-87,*** Vancouver Canada, Aug. 1987.

[GREE88] Greenstreet M., Steiglitz K., "Throughput of Long Self-Timed Pipelines," CS-TR-190-88, Princeton U., Nov. 1988.

[LEIS86] Leiserson C., Saxe J., "Retiming Synchronous Circuitry," DEC Systems Research Center TR-13, Palo Alto, 1986.

[MART86] Martin, A., "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, ***Distributed Computing,*** vol. 1, no. 4., pp. 226-234, 1986.

[MART89] Martin A., "On the Existence of Delay-Insensitive Circuits," ***MIT Conference on Advanced Research in VLSI***, March 1989.

[MENG88] Meng T., "Asynchronous Design for Programmable Digital Signal Processors," Ph.D. Thesis, UC Berkeley, 1988.

[MENG89] Meng T., Brodersen R., Messerschmitt D., "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications," ***IEEE Tran. on CAD,*** vol. 8, no. 11, November 1989.

[MILL65] Miller R.E., ***Switching Theory,*** Wiley, 1965.

[MULL63] Muller D.E., "Asynchronous logics and applications to information processing," ***Proceedings of Symposium on Applications Switching Theory Space Technology,*** pp. 289-297, 1963.

[MURA77] Murata T., "Petri Nets, Marked Graphs, and Circuit-system ***Theory," Circuits and Systems,*** vol 11 no. 3, June 1977.

[RAMA80] Ramamoorthy C., Ho G., "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," ***IEEE Tran. on Software Engineering, vol.*** SE-6 no. 5, Sept. 1980.

[RAMC74] Ramchandani C., "Analysis of Asynchronous Concurrent Systems by Petri nets," Project MAC, TR-120, MIT, Cambridge MA, Jan. 1974.

[RAO85] Rao S.K., "Analysis and Construction of Synchronous Regular Iterative Arrays," Ph.D. Thesis, Stanford Univ., 1985.

[SEIT80] Seitz C., "System Timing," Chapter *7* in ***Introduction to VLSI Systems,*** eds. Mead C. & Conway L., Addison-Wesley, 1980.

[SUTH89] Sutherland I., "Micropipelines," ***Communications of the ACM, vol. 32*** no. 6, July 1989.

[UDDI86] Udding, J.T., "Classification and Composition of Delay-Insensitive Circuits," Ph.D. Thesis, Eindhoven Tech. Univ., 1986.

[WILL87] Williams T., Horowitz M., et.al., "A Self-Timed Chip for Division," ***Advanced Research in VLSI, Proceedings of the Stanford Conference,*** pp. 75-96, March 1987.

[WILL91] Williams T., Horowitz M., "A Zero-Overhead Self-Timed 54b 160nS CMOS Divider," submitted to ***IEEE Conference on Solid-State Circuits,*** Feb. 1991.