TEMPLATE BASED ASYNCHRONOUS DESIGN

By

Recep Ozgur Ozdag

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(ELECTRICAL ENGINEERING)

November 2003

Contents

List Of Tables

List Of Figures

Abstract

Asynchronous design is increasingly becoming an attractive alternative to synchronous design because of its potential for high-speed, low-power, reduced electromagnetic interference, and faster time to market. To support these design efforts, numerous design styles and supporting CAD tools have been proposed. We adopt a template-based methodology that facilitates hierarchical design using standard asynchronous channel protocols, removes the need for complicated hazard-free logic synthesis, and naturally provides fine-grain pipelines with high throughput. We propose seven different templates that provide tradeoffs between throughput and robustness to timing. The most robust templates are quasi-delay-insensitive in that they work correctly regardless of delays on individual gates. The most aggressive templates use timing assumptions that can be satisfied with additional care during transistor sizing, floorplanning, and layout.

We propose a complete design methodology for template-based designs using standard hardware description languages and the Cadence design framework. We demonstrate the advantages of the templates and methodology by designing an asynchronous sequential channel decoder based on the Fano algorithm. Spice simulations, on the extracted layout, show that the circuit runs at 450MHz and consumes 32mW at 25^{o}C. The asynchronous chip runs about 2.15 faster and consumes 1/3 the power of its synchronous counterpart.

Chapter 1

1. Introduction

Digital VLSI circuit design styles can be mainly classified as either synchronous, asynchronous or some mixture. Synchronous designs, consists of subsystems, which are controlled by one or more clocks that control synchronization and communication between blocks, have dominated the design space since the 1960's. Combinational logic is placed in between clocked registers that hold the data. The delay through the combinational logic plus relevant setup time should be smaller than the clock cycle time. In fact, the data at the inputs of the registers may exhibit glitches or hazards as long as they are guaranteed to settle before the sample clock edge arrives. Asynchronous methodologies, in contrast, use event-based handshaking to control synchronization and communication between blocks. This chapter first reviews various synchronous design methodologies and then describes some potential advantages of asynchronous design, before providing a more detailed overview of the thesis.

Synchronous design methodologies can be classified in one of two main categories; *standard cell* design and *full custom* design. Semi-custom standard-cell-based design methodologies offer good performance with typically 12-month design times [1]. They are supported by a large array of mature CAD tools that range from simulation, synthesis, verification, and test. The synthesis task is divided into architecture definition, logic/gate-level design, and physical design.

A large library of standard-cell components that have carefully been designed, verified,

and characterized supports the synthesis task. This library is generally limited to static CMOS based gates for a variety of reasons. Compared to more advanced dynamic logic families, standard CMOS static logic has higher noise margin and thus requires far less analog verification, significantly reducing design time.

Standard-cell designs also use standard clocking strategies to facilitate more automation and reduced design times. The forms of gated clocking are limited, reducing power efficiency. Standard flip-flop based designs are used to simplify timing analysis despite the incurrence of significant data to clock output overheads.

Moreover, the time-to-market advantage of standard-cell based designs is being attacked by the increasingly difficult task of estimating wire-delay. In submicron designs, the process of architecture, logic, and technology mapping design could proceed somewhat independently from placement and routing of the cells, power grid, and the clocks because wire-delays were negligible compared to gate-delays. In deep-submicron design, however, the relative delay of long-range wires are increasing and becoming harder to estimate. This is causing the traditional separation of logic synthesis and physical design tasks to break down because synthesis is not properly accounting for actual wire delays. This timing-closure problem has forced numerous shipment schedules to slip. EDA vendors have now developed a new suite of emerging CAD tools that address aspects of the physical design must occur much earlier in the design process.

In the future, predictions suggest that long-range wires may have 5 to 20 clock cycles in delay making estimation particularly critical [1]. In particular it is predicted that that high-speed clock regions communicating at perhaps reduced frequencies may become prevalent, but the semi-custom CAD support for multiple clock domains is just emerging. The

simplest approach involves adding synchronizers between clock domains that incurs a significant latency penalty.

Some manufacturers have extended the standard cell design technique to the design of datapaths and other higher-level functions such as microprocessors and their peripherals. On the other hand the design can also be implemented by optimizing every transistor of the layout. This technique is called full custom design, and is generally preferred when one or many aspects of the chip need to be optimized beyond what is readily available in a semi-custom approach. Since the designer controls the transistor size, placement of the smallest functional blocks and the main routing method, the end result in general is much better than standard cell design. In the full custom method, design time is traded in for higher performance, reduced area or power consumption, since all possible circuit techniques can be applied, where as in standard cell design, the CAD tool only has a limited number of pre-laid out cells that need to be broad enough to suit every customers need.

Full-custom design houses have found that these challenges with standard cell design can be overcome with longer design cycles of an average of 36 months. In particular, the use of advanced logic dynamic logic styles has been an area of growing interest in full-custom designs [2] [3] [4] [5]. Domino logic is estimated to be 30% faster than static logic because of the improving logical effort derived by the removal of PMOS logic. Traditional domino logic however still suffers from overhead associated with clock skew and latch delays. More advanced flip-flops and latches have been developed that somewhat improve the clock skew overhead and reduce the latch delays. At the extreme, the latch delays can be removed using multiple overlapping clocks in a widely used technique, recently named skew-tolerant domino logic [5].

In addition to the problems of clock distribution and skew is the problem of heat and power consumption. Many of the gates switch because they are connected to the clock, not because they have new input data to evaluate. The biggest gate of all is the clock driver, and it must switch all the time to provide the correct timing, even if only a small part of the chip has anything useful to do. Although gating the clock is an option to send the clock signal to only those who need it, stopping and starting a high-speed clock is not easy.

To reduce power consumption, particularly in memories and long-distance on-ship and off-chip communication, low-voltage signalling has been commonly used. These also suffer from reduced noise margins, requiring more manual design practices and extensive analog simulation.

The basic cost that achieving this higher performance and low-power presents is the reduced noise margin and the increased need for more careful, manual design practices and extensive analog verification, pre and post layout.

The increasing limitations and growing complexity of both standard-cell and full-custom synchronous design have led to a change of focus on digital circuit design. In particular, circuits that lack a global controlling clock, namely asynchronous circuits have demonstrated potential benefits in many aspects of system design (e.g. [6], [7], [8], [9], [10], [11], [12], [13],[14]). Asynchronous circuits have several advantages over their synchronous counterparts, including:

1) Elimination of clock skew: Clock skew is defined as the arrival time difference of the clock signal to different parts of the circuit. In general in standard cell design, to avoid this problem, the clock pulse is increased to assure correct operation, which yields slower running circuits. However in full custom design buffer insertion, or careful clock tree

design and analysis to improve clock routing and clock power are some of the methods synchronous designers are using to handle this problem. Although full custom design approach leads to reduction or even elimination of clock skew, for synchronous design this is still a problem that needs to be worked on. On the other hand, since asynchronous circuits have no global clock that controls the data flow, there is no clock skew problem.

2) Lower power consumption: In general, the constant activity of the clock signal causes synchronous systems to consume power even though some parts of the circuit may not be processing any data. Even though some improvements in full custom design, such as clock gating avoid sending the clock signal to the un-active parts the clock driver has to constantly provide a powerful clock to able to reach all the parts of the circuit. Although asynchronous circuits in general have more transitions due to the hardware overhead, they generally have transitions only in areas that are active in the current computation.

3) Average case performance: Synchronous circuit designers have to consider the worst-case scenario when setting the clock speed to ensure that all the data has stabilized to before being latched. However asynchronous circuits detect and react when the computation is completed, yielding average case performance rather than worst case [14].

4) Easing of global timing issues: since in synchronous circuits the slowest path dictates the clock speed, designers try to optimize all the paths to achieve the highest possible clock rate. In particular there maybe long wires, which require large buffers and consume significant power even though they may be non-critical or maybe infrequently driven. In contrast in asynchronous circuits optimizing the frequently used paths is easier [9].

5) Better technology migration potential: Since the technology which the circuit is implemented improves rapidly, for synchronous circuits better performance often can only

be achieved by migrating all the system components to the new technology where as for asynchronous design the communication between blocks only occur when the completion of the processing is detected, therefore different delays introduced with different technologies can be easily substituted into a system without altering other structures.

6) Automatic adaptation to physical properties: The delay on a path may change to the variations in the fabrication process, temperature, and power supply voltage. Synchronous system designers must consider the worst case and set the clock period accordingly. However asynchronous circuits naturally adapt to changing conditions since the slowdown on any path does not affect the functionality of the system [15].

7) Improved EMI: In a synchronous design, all activity is locked into a very precise frequency. The result is nearly all the energy is concentrated in very narrow spectral bands at the clock frequency and its harmonics. Therefore, there is substantial electrical noise at these frequencies. Activity in an asynchronous circuit is uncorrelated, resulting in a more distributed noise spectrum and a lower peak noise value [16].

1.1 Asynchronous Circuit Design Flow

The USC Asynchronous CAD and VLSI group, jointly with the Columbia Asynchronous group, is currently developing a complete asynchronous circuit design methodology that will support automated design exploration of both high-performance and low-power asynchronous circuits. The basic steps of the methodology are illustrated in Figure 1.1. First a language based, model such as CSP [17] and Verilog [18], is used as the input description. This input description describes the desired top-level functionality of the chip and maybe annotated with overall constraints on power, energy consumption, throughput, latency, chip area, etc. Note that details regarding internal structure or the

specific asynchronous protocols used are specifically not included in the description. After generating this input description and verifying its correctness, the next step in the methodology is to explore and finalize a basic architecture for the design. This basic architecture should identify the number and relative characteristics of the basic blocks in the design (register files, ALUs, multipliers, etc.) To automate this step we expect to adapt variations in classical high-level synthesis, i.e., *scheduling*, *resource sharing*, and *binding*. After architectural design is complete, the next step in the methodology is micro-architecture design. In this step the designer can choose to implement the architecture with various methods ranging from fine grain pipelines template-based using delay insensitive cells to components relying on bounded delay based with no pipelining at all. Depending on the style chosen, various optimizations can be applied, namely selection of the handshaking protocol, defining the level of pipelining, and slack optimization for pipelined designs. Once this initial mirco-architecture is created, next step is to identify critical components and perform *handshaking optimization* to achieve higher performance and lower power. Based on the final micro-architecture, a gate or transistor level design is generated. This can be done either automatically using new template-based synthesis techniques that our group is creating or manually.

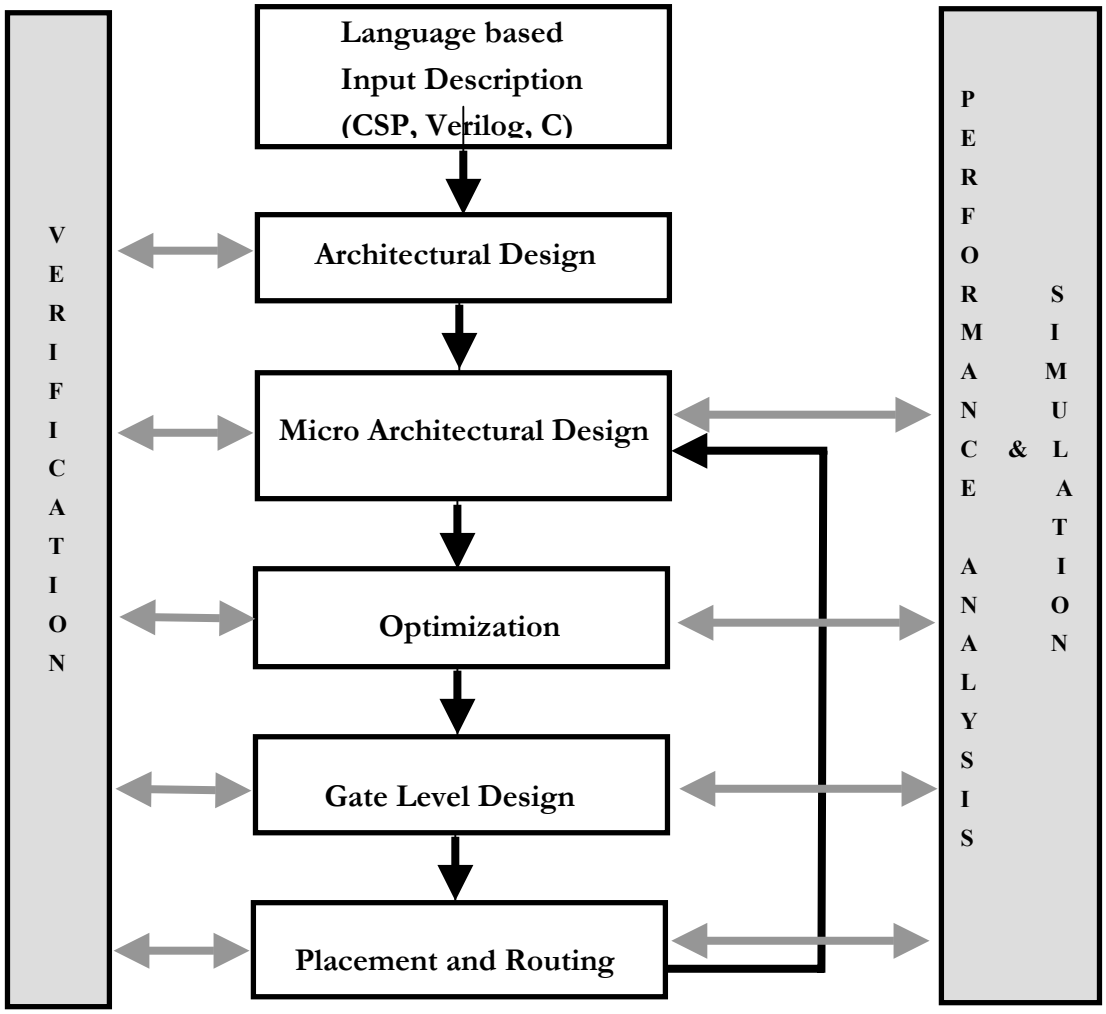


Figure 1-1: Asynchronous circuit design flow under development

Finally, placement and routing will be applied very a similarly to that required synchronous circuit design. In every step all the design process, verification and performance analysis tools are used to verify correct functionality and overall performance. The focus this proposal is the generation of new templates for template-based design, as well as to help develop the above CAD frame for the automated design of asynchronous systems.

1.2 Expected Contributions of the Thesis

Our research group's goal is to produce a complete design method for asynchronous systems, including specification, synthesis, verification, simulation, and testing and to develop a suite of CAD tools supporting the design method. And by using these CAD tools to design high-performance and energy-efficient asynchronous microprocessors, and systems-on-a-chip. As part of an ongoing research to accomplish these goals the we:

- Develop two new quasi delay insensitive, high-speed templates targeted at non-linear pipelines, which are faster and smaller than other quasi delay insensitive templates. Quasi delay insensitive templates are the most robust asynchronous building blocks for designs based on templates. By using templates we can mimic ease of design of the standard cell design methodology in synchronous design. We also show the implementation of some of the non-linear structures.
- To achieve higher speeds, we then develop five new bounded delay pipeline templates by modifying and further improving the templates developed by Columbia University, which are based on timing assumptions to shorten handshaking time and achieve higher speeds. In particular, the templates developed by Columbia University were targeted for linear pipelines such as FIFOs. Real life designs however, require more complex structures that require the template to also function correctly with non-linear pipelines. To extend the existing pipelines we modify each template to handle non-linear pipelines with little impact on performance.
- We then implement a communication algorithm as a design example in both synchronous and asynchronous methods to show the advantages of asynchronous

design over synchronous design as well as to help the development of a CAD environment, which is mainly targeted for template, based design. The asynchronous implementation of the algorithm will also be used to study the trade offs among different asynchronous templates from timed to delay insensitive.

1.3 Thesis Organization

The organization of the reminder of this proposal is as follows. Chapter 2 presents background on asynchronous circuit design styles, and linear and non-linear pipeline applications, Chapter 3 presents the new high speed QDI pipelines, Chapter 4 presents the extension to the pipelines introduced by Columbia University and the introduction of five new timed templates, Chapter 5 presents the design example in synchronous, and Chapter 6 presents in asynchronous. Finally, Chapter 7 presents our semi-custom asynchronous design flow.

Chapter 2

2. Background

This section presents the basics of asynchronous circuit design and classifies many of the existing asynchronous circuit design styles according to data encoding method, handshaking style, granularity of pipelines and circuit style. Then we describe the differences between logic synthesis-based methodologies and those that rely more on a template-based methodology. We then focus on existing templates that support the design of complex fine-grain pipelines and analyze their performance.

2.1 Data Encoding Styles

Single rail [19] communication between functional blocks consists of one request wire and one wire per data bit from the sender to the receiver and one acknowledgment wire from the receiver to the sender. *Dual rail* communication often consists of two wires per data bit from the sender to the receiver and one acknowledgment wire from the receiver to the sender. In addition, dual-rail designs can have an additional request line [20]. 1-of-N communication is a generalization of dual rail communication in which $[log_2 N]$ bits are sent using N wires.

An acknowledgment signal from the receiver to the sender is used to tell the sender that the data is no longer needed. The logic that drives this acknowledgment signal often involves *completion sensing circuitry* that helps determine when the receiver is done using the current data bits. In single rail communication, completion sensing circuits are implemented with *bundled data* lines [19] or more sophisticated speculative completion sensing circuitry [21], [22], that includes delay lines that match the critical paths of the

functional unit. On the other hand, completion sensing of dual rail designs can be done using specialized logic that actively identifies when the computation is done. This latter logic relies on the dual-rail nature of the data and can be implemented without relying on timing assumptions and thus, is more robust to variations in delay than its delay-line counterparts. Completion sensing, however, requires more circuitry than delay lines and, if not done wisely, can incur a significant performance, power and area penalty.

The functional units can be implemented using static or dynamic logic. Often functional units that communicate using dual rail or 1-of-N styles are implemented using dual rail dynamic logic [23] [24], but since static logic is also possible [23]. Functional units that communicate using single rail are more commonly implemented using static logic that is often smaller and consumes less power than dynamic counterparts. Designs implemented with dynamic logic, however, can generally achieve higher throughput than their static logic counterparts. Consequently, they can run at lower voltages to achieve a given throughput requirement and, thus may yield a lower power design than their single rail counterparts.

2.2 Handshaking Styles

Asynchronous circuits consist of functional units that communicate control and data information using various handshaking styles. The most dominant forms of handshaking styles *two-phase* [25] and *four-phase handshaking* [26] are shown in Figure 2.1. In two-phase handshake protocol, a request and an acknowledge wire is used to implement handshaking between the sender and the receiver. In two-phase handshake protocol, all transitions are functional and consequently every pair of consecutive request/acknowledge transitions forms a complete handshake. Two-phase single rail communication is usually seen with static logic functional units that use bundled-data for completion sensing. Due to some

difficulties in designing complex two-phase control circuits, a novel single-track handshaking protocol has been suggested by van Berkel and Bink [27]. This handshaking protocol is achieved by combining the request and acknowledge lines into one wire and is illustrated in Figure 2.1 (b). Where two-phase handshaking involves two events per cycle, four-phase handshaking requires four events, as shown in Figure 2.1 (c). Since four events are used to designate a complete handshaking cycle, half of these are essential for functional computation and the other half are not actively used to communicate data. Nevertheless, this reset phase is very useful for precharging dynamic units. Figure 2.1 (d) shows a four-phase handshaking protocol for dual-rail dynamic units [23] [24]. Other protocols extend the data valid region through the reset phase [19] [28], to more efficiently use four-phase handshaking with static functional units.

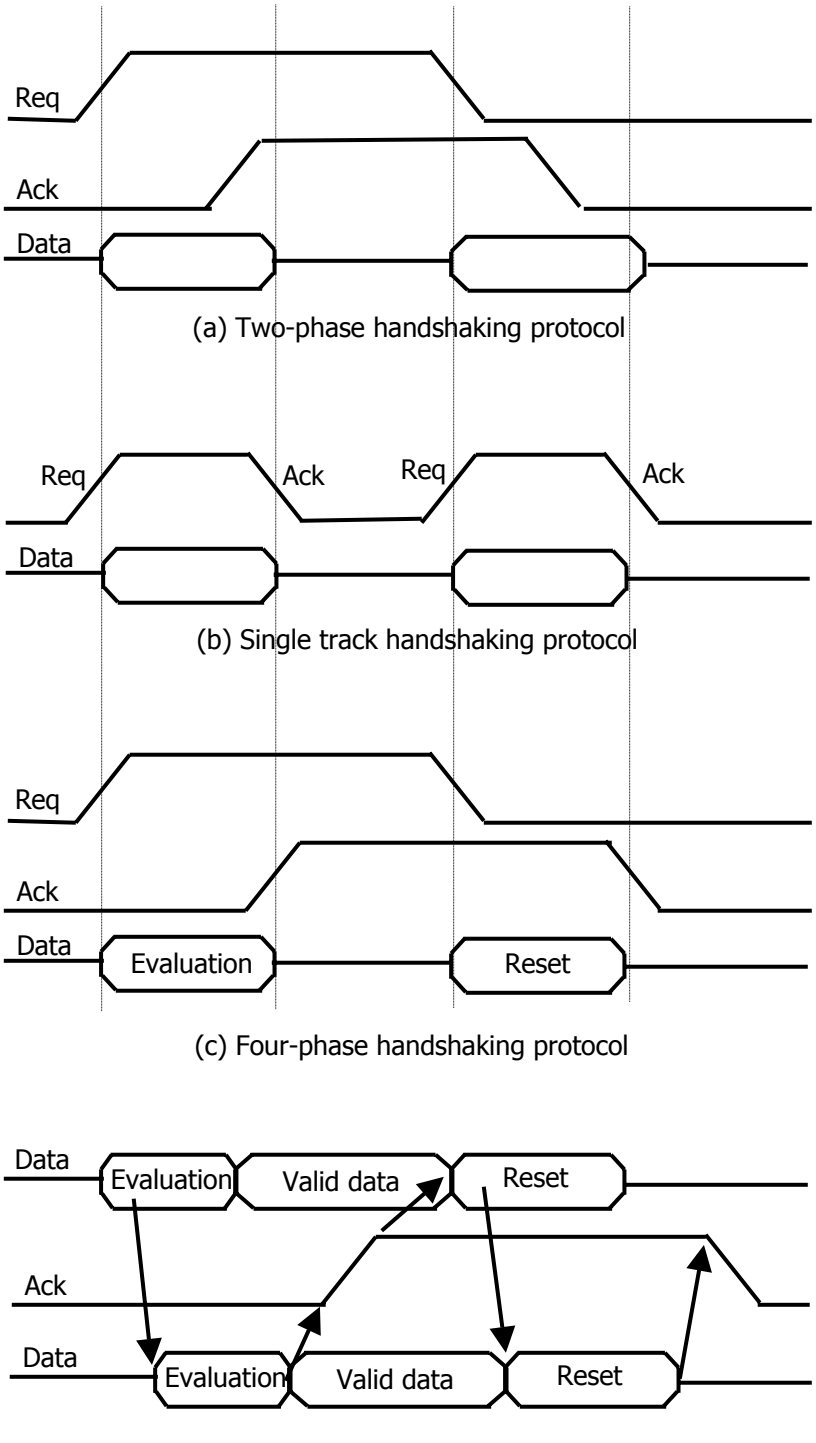


Figure 2-1: Handshaking protocols: Two-phase versus four-phase.

2.3 Delay Models

Most design techniques require some timing assumptions or constraints on the wires and/or components to ensure correct operation. For example, in synchronous circuit design, the data input to every register must satisfy all setup and hold times. The delay assumptions in asynchronous circuits widely vary based on design styles as outlined below.

- *Delay insensitive (DI)*: Delay insensitive designs [29] [30], require no timing assumptions on wither wires or gates. That is, DI circuits work correctly for any arbitrary, time-varying gate and wire delay. This is the most conservative and robust design style, but it has been shown that very few gate-level delay insensitive designs can exist [31]. That said, delay insensitivity can more easily and practically be achieved at a block level where blocks communicate only through delay insensitive channels.
- *Quasi delay insensitive (QDI)*: Quasi delay insensitive design [32] [24] is a practical approximation to delay insensitive design. QDI circuits work correctly regardless of delays in gates and all wires except in cases of wire forks designated *isochronic*. The difference in time at which the signal arrives at the ends of an isochronic fork must be less than the minimum gate delay. If these isochronic forks are guaranteed to be local to a small component, these circuits can be practically as robust as DI circuits. The QDI assumption has also been extended to include assumptions of isochronic propagation through a number of logic gates [33].
- *Speed independent (SI)*: SI design [23] [34], assumes that gate delay can be arbitrary but

that all wire delay is negligible. From a delay perspective SI design basically assumes that all forks are isochronic. For the design of small control circuits, thus timing assumption is generally satisfied.

- *Scalable delay insensitive (SDI)*: SDI approaches [36] [21], are motivated by the observation that SI design should not be used for any circuit that spans significant chip area. Consequently, in SDI design the chip area is divided into many regions, SI circuit design is used within each region, and communication between regions is done delay insensitively.
- *Bounded delay*: In bounded delay models each gate is given a minimum and maximum delay and the circuit must work if the delay of all gates are within these bounds. These timed circuits can often be faster, smaller and lower power than their QDI or SI counterparts, but require more careful timing verification during physical design [37].
- *Relative timing*: In relative timing based circuits, a list of relative orderings of events identifies sets of path pairs, where for each pair of paths, one path must be longer/shorter than each other to ensure correctness. These circuits can have the same benefits of times circuits and may be easier to validate [38] [39] [40].

2.4 Synthesis Based Design

2.4.1 Fundamental Mode Huffman Circuits

In this model, the circuit design flow is similar to that of the design of synchronous circuits[15]. The circuit is usually expressed as a *flow table* [41]. The flow table has a row for each internal state, and a column for each combination of inputs. The entries indicate the

next state entered and output generated when the column's input combination is seen while in the row's state. States where the next state is identical to the current state are called *stable states*. It is assumed that each unstable state leads directly to a stable state, with at most one transition occurring on each output variable. Similar to finite state machine synthesis in synchronous systems, state reduction and state encoding is performed on the flow table, and Karnaugh maps generated for each of the resulting signals.

There are several points that need to be considered for this design method. The system responds to input changes rather than clock ticks therefore the circuit may enter some intermediate states if multiple inputs change at the same time. Therefore it must be guaranteed that these intermediate states should still lead to the intended stable state, irrespective of the order of how inputs change.

Another concern is hazard removal. Since hazards, static or dynamic, can cause the circuit to enter an unstable state, they must be eliminated by adding a sum-of-products circuit that has functionally redundant products.

Due to the restriction of only one input changing to the combinational logic at a time, several requirement need to be forced on the implementation of sequential circuits. First, the combinational logic must settle in response to a new input before the present state entries change. The state encoding must assure a single bit transition for state transitions. The last requirement is that the next external input transition cannot occur until the entire system settles to a stable state.

While the fundamental mode assumption makes logic design easy, it also increases cycle time. There are proposed solutions, which carefully analyze an implementation to relax the fundamental mode assumption, however because of the limitations on the multiple input

changes, this design methodology has never achieved wide acceptance for complex system design. Burst-mode circuits, covered in the next section, overcome the limitations on multiple input changes.

2.4.2 Burst-Mode Circuits

The *burst-mode* design style developed by [42], [43], [44] is based on the earlier work at HP laboratories by [45], attempts to move even closer to synchronous design than the Huffman method [15]. In this method, circuits are specified via a standard state-machine, where each arc is labeled by a non-empty set of inputs (an *input burst*) and a set of outputs (an *output burst*). The assumption is that, in a given state, only the specified inputs on one of the input bursts leaving that state can occur. The inputs are allowed to occur in any order. The state reacts to the inputs only when all of the expected inputs have occurred. The state machine then fires the specified output bursts and enters the specified next state. New inputs are only allowed to occur after the system has completely reacted to the previous input burst. Therefore, the burst-mode method still requires the fundamental-mode assumption, but only between transitions in different input bursts. Another restriction is that no input burst can be a subset in another input burst leaving the same state.

Burst-mode circuits can be implemented in various ways, including similar techniques to those of Huffman circuits.

The problems with both the fundamental-mode and burst-mode circuits that restrict these circuits are the fact that circuits often are not simple single gate small state machines, but instead complex systems with multiple control state machines and datapath elements. These methods do not discuss system decomposition for complex circuits. Also, these methodologies cannot design datapath elements. This is because datapath elements tend to

have multiple input signals changing in parallel, and the fundamental-mode assumption would be easily violated. Although one solution for datapath implementation is to use synchronous components with careful add-hoc optimization, another issue is the increased delay by the additional delay elements to satisfy the fundamental-mode assumption. Not only is the delay increased but it must also be able to work under worst-case scenario.

2.4.3 Event-Based Design

Petri nets and other graphical notations are a widely used alternative to specify and synthesize asynchronous circuits. In this model, an asynchronous system is viewed not as state-based, but rather as a partially ordered sequence of events. A Petri net [46] is a directed bipartite graph, which can describe both concurrency and choice. The net consists of two kinds of vertices: *places* and *transitions*. Tokens are assigned to the various places in the net. An assignment of tokens is called a *marking*, which captures the state of the concurrent system. When all the conditions preceding a transition are true the action may *fire* which removes the tokens from the preceding places and marks the successor places. Hence, starting from an initial marking, tokens flow through the net, transforming the system from one marking to another. As tokens flow, they fire transitions in their path according to certain *firing rules*.

Patil proposed the synthesis of Petri nets into *asynchronous logic arrays*. In this approach, the structure of the Petri net is mapped directly into hardware. Many modern synthesis methods use a Petri net as a behavioral specification only, not as a structural specification. Using reachability analysis, the Petri net is typically transformed into a *state graph*, which describes the explicit sequencing behavior of the net. An asynchronous circuit is then derived from the state graph.

More general glasses of Petri nets include Molnar *et al.'s I-Nets* [47], and Chu's *Signal Transition Graphs* or *STGs* [48]. These nets allow both concurrency and a limited form of choice. Chu developed a synthesis method, which transforms an STG into a speed-independent circuit, and applied the method to a number of examples.

Petrify is a tool for manipulating concurrent specifications and synthesis and optimization of asynchronous control circuits[49]. Given a Petri net, or a STG it generates another Petri net or STG, which is simpler than the original description and produces an optimized net-list of an asynchronous controller in the target gate library while preserving the specified input-output behavior. An ability of back annotating to the specification level helps the designer to control the design process.

For transforming a specification petrify performs a token flow analysis of the initial Petri net and produces a transition system. In the initial transition system, all transitions with the same label are considered as one event. The transition system is then transformed and transitions relabeled to fulfill the conditions required to obtain a safe irredundant Petri net. For synthesis of an asynchronous circuit petrify performs state assignment by solving the *Complete State Coding* problem. State assignment is coupled with logic minimization and speed-independent technology mapping to a target library. The final netlist is guaranteed to be speed-independent, i.e., hazard-free under any distribution of gate delays and multiple input changes satisfying the initial specification. The tool has been used for synthesis of Petri nets and Petri nets composition, synthesis and re-synthesis of asynchronous controllers and can be also applied

2.5 Template-Based Design

A different approach is for asynchronous design is to view the system as

communication blocks or processes, called templates that encapsulate all the design constraints inside the modules. These templates will have requirements of their environment that must be met, and which will restrict how these templates are used. However, such restrictions or internal timing constraints are much simpler than those of most other methodologies, and the proper template will usually be obvious from the functionality required.

Template-based design is somewhat similar to standard cell design in synchronous logic. Templates can be either pre-designed to implement simple logic functions, with handshaking, or can synthesized to create more complex ones.

The advantage of template-based design is the ease of manual design. In general a datapath is created, and the control unit is designed around the datapath. Once a general architecture is created the rest of the task is to implement the blocks of the architecture using templates. Also template-based design has the potential advantage, which is currently being investigated, of being able to be used as a backend to a synchronous CAD tool. The highly optimized synchronous design can be converted to an asynchronous one by replacing every gate with its asynchronous handshaking counterpart template. However additional optimization might be required to improve the performance of the system.

2.5.1 Template-Based Compilation Systems

Although template-based system can ease manual design, their main power is seen when they are coupled with a high-level language and automatic translation software. The following section presents some well-known methodologies, which have their own language for easy compilation of asynchronous systems.

2.5.1.1 Caltech's Design Methodology

Caltech's communicating processes compilation technique [50], translates programs written in a language similar Communicating Sequential Processes into asynchronous circuits, which communicate on channels. The source language describes circuits by specifying the required sequences of communications in the circuit.

Caltech's translation process is accomplished in several steps: (1) in *process decomposition*, a process is refined into an equivalent collection of interacting simpler processes; (2) in *handshaking expansion*, each "communication channel" between processes is replaced by a pair of wires, and each atomic "communication action" is replaced by a handshaking protocol on the wires; (3) in *production-rule expansion*, each handshaking expansion is replaced by a set of "production rules (PRs)", where each rule has a "guard" that insures it is activated (*i.e.*, "fires") under the same semantics as specified by the earlier handshaking expansion; and finally, (4) in operator reduction, PRs are grouped into clusters, and each cluster is than mapped to a basic hardware component. It is important to realize that many of these steps require subtle choices that may have significant impact on circuit area and delay. Although heuristics are provided for many of the choices, much of the effort is directed towards aiding a skilled designer instead of creating autonomous tools. This has the benefit in that the designer can usually make better decisions, provided that the designer is skilled enough.

Caltech has later moved to using more standardized, pre-designed, less complex building blocks, which simplify the design method, explained above. Caltech's template-based design methodology has moved from the synthesis of complex templates to chip implementation using smaller, and simpler templates, which have very standard design

guidelines. These templates are in general targeted for implementing fine grain pipelined chips.

2.5.1.2 Tangram and Balsa

Another compiler-based approach developed by van Berkel, Rem and others [51], at Philips Research Laboratories and Eindhoven University of Technology uses the *Tangram* language. Tangram, which is based on CSP, is a specification language for concurrent systems. A system is specified by Tangram program, which is then compiled by syntax-directed translation into an intermediate representation called a *handshake circuit*. A handshake circuit consists of a network of *handshake processes*, or *components*, which communicate asynchronously using handshaking protocols. The circuit is then improved using peephole optimization and, finally components are mapped to VLSI implementations.

Although Tangram is also syntax derived like Caltech's design methodology, it also targets non-pipelined designs, which can support non-linear sequential processing as well as pipeline processing.

The Tangram compiler has been successfully used at Philips for several experimental DSP designs and electronics; including counter, decoders, image generators, and an error corrector for a digital compact cassette player.

Balsa [52], developed at University of Manchester, adopts syntax-directed compilation into handshaking components and closely follows Tangram. A circuit described in Balsa is compiled into a communicating network composed from a small (~35) set of handshake components. Balsa can be thought as of an public extension to Tangram. In particular the support for separate compilation and the use of a flexible communication enclosed input choice mechanism are claimed as useful additions to the expressiveness of Tangram. New

handshake components (which are the constituent parts of handshake circuits) are proposed which are used to implement this choice mechanism as well as more generalized forms of the existing Tangram system components.

2.5.2 Micropipelines

Micropipelines, introduced by Ivan Sutherland, use standard synchronous datapath logic to build asynchronous pipelines [25]. A micropipeline has altering computation stages separated by storage elements and control circuitry. This approach uses transition signaling for control along with bundled data. Sutherland describes several designs for the storage elements, called "event-controlled registers", which respond symmetrically to rising and falling transitions on inputs.

Computation on data in a micropipeline is accomplished by adding logic computation blocks between register stages. Since these blocks will slow down the data moving through them, the accompanying transition is delayed as well by the explicit delay elements, which must have at least as much delay in them as the worst-case logic block delay. The major benefit of the micropipeline design style is that the registers or latches at the boundaries of pipeline stages filter out logic hazards within the combinational logic. Thus, standard synchronous combinational logic design styles and supporting CAD tools can be used.

Although micropipelines is a powerful design style, which elegantly implements elastic pipelines, there are some problems with them as well. It delivers worst-case performance by adding delay elements to the control path to match worst-case computation times. Also there are delay assumptions that must be carefully verified. Finally, there is little guidance currently on how to use micropipelines for more complex (add speculative completion pros and cons) systems.

2.5.3 Ad Hoc Design

Our final design methodology is ad hoc design. Although it may not seem like a design methodology, the ad hoc design approach implemented buy a skilful designer can lead to very competitive results. A design can be completely implemented in an ad hoc fashion, or can be initially developed using one of the methods above and then be optimized in an ad hoc sense.

An asynchronous design can be implemented the same way a synchronous design would, using synchronous components for the datapath. A matched delay can be used to indicate the completion of the computation. The control circuit can be implemented by modifying a synchronous FSM to work with input transitions rather than a global clock.

Another approach is the use self-resetting logic. Although self-resetting logic has a number of difficult to satisfy timing assumptions careful ad hoc design can achieve high throughput with self-resetting asynchronous circuits. The synchronous parts of the circuit can be replaced with self-resetting logic. Important aspects of self-resetting design such as data insertion and pulse generation would require an ad hoc approach. Or alternatively, an asynchronous circuit can be implemented using any of the approaches presented above and can be later optimized for speed, area or power using verifiable ad hoc optimizations.

2.6 Linear and Non-Linear Asynchronous Pipelines

This section presents the basics of linear and non-linear fine-grain asynchronous pipelines where each pipeline stage is derived through one of several basic templates.

2.6.1 Linear Pipelines

A pipeline is a linear sequence of functional stages where the output of one stage is connected to the input of the next stage. Data signals, which flow from the inputs to the outputs of the pipeline, are also called as data tokens. A linear pipeline has no forking or joining stages. The tokens in the pipelines remain in a first in first out order (FIFO). In synchronous design the sequential functional stages are registers. These registers hold the data tokens and are controlled by a global clock signal. Depending on the implementation, on rising or falling edge of the clock, all the registers sample new data values which wait at their inputs. Since all the registers "see" the clock signal at the same time, the movement of one data token to the next register is synchronized to all other data tokens, and they all move at the same time. However there is no central global clock in asynchronous design therefore a data token in one stage only moves to the next stage if it is empty. The handshaking protocol between the two stages (the sender and the receiver) determines how the two stages inform each other when there is an empty space, when the data has been sent, if the data has been received by the next stage (receiver) and when the previous data holding stage (sender) can reset its data. The handshaking protocol is accomplished through a *communication channel* between the sender and the receiver. Although in this section we explain a communication channel under the context of pipelines, a communication channel can exist between any two asynchronous units. An asynchronous communication channel shown in Figure 3.1 is a bundle of wires and a protocol to communicate data between a sender and a receiver. For single rail encoding one wire per bit is used to transmit the data and an associated request line is sent to identify when data is valid. The associated channel is called a bundled-data channel. Alternatively for dual rail encoding the

data is sent using two wires for each bit of information. Extensions to 1-of-N encoding also exist.

Both single-rail and dual-rail encoding schemes are commonly used, and there are tradeoffs between each. Dual-rail and 1-of-N encoding allow for data validity to be indicated by the data itself and are often used in QDI designs. Single-rail, in contrast, requires the associated request line, driven by a matched delay line, to always be longer than the computation, as we described in section 2.1.

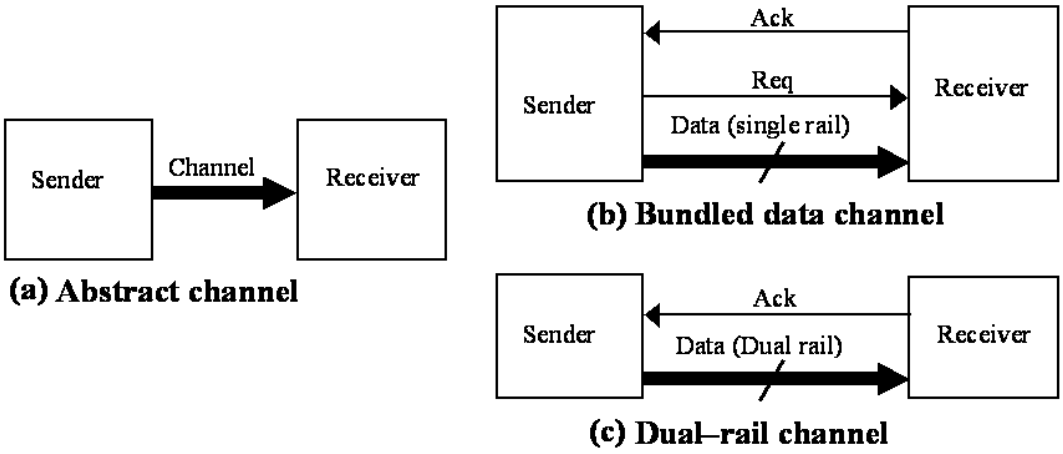
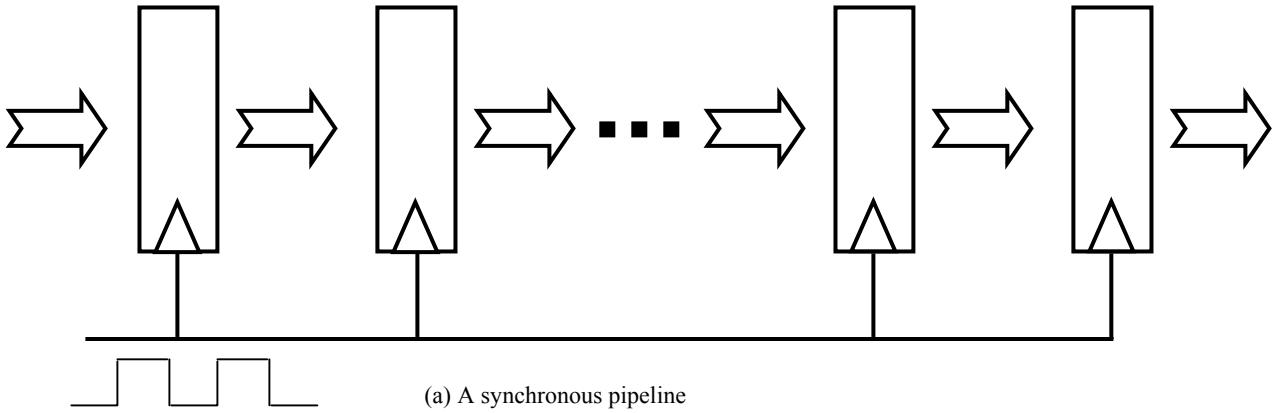
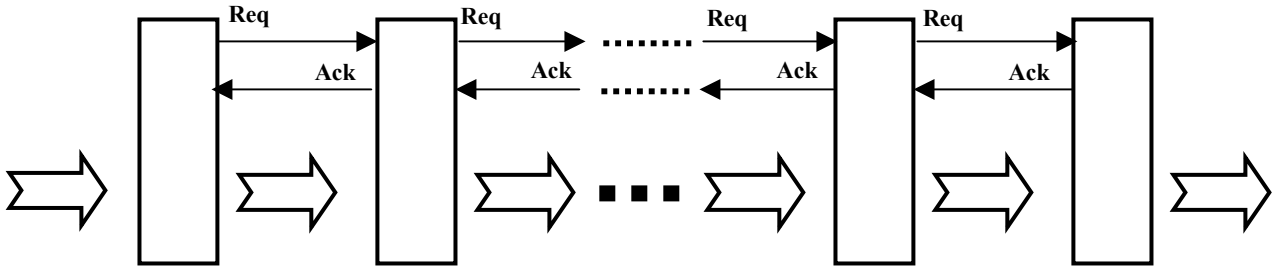


Figure 2-2: Pipeline channels



(a) A synchronous pipeline



(b) An asynchronous pipeline

Figure 2-3: Synchronous vs. asynchronous pipelines

Figure 2-3 illustrates the difference between typical synchronous and asynchronous linear pipelines

Abstractly the operation of a general asynchronous pipeline with four-phase handshaking can be described as follows. Initially the pipeline is empty, and all the data lines as well as the handshaking signals *req* (the request signal) and *ack* (the acknowledgment signal) are de-asserted. The request signal req can be used if the data lines are single rail, to inform the next stage the arrival of data. On the other hand if the data lines are implemented with dual rail, conventionally, there is no need for the req signal. When the

first stage evaluates and generates an output the req signal is also assert. When the second stage evaluates it asserts its req signal as well as the ack signal to acknowledge the first stage that it has consumed the data. The first stage responds to this acknowledge signal by resetting its outputs. The first stage can only generate new data when the acknowledge signal is de-asserted, indicating that the second stage is ready to consume the second data token. When the third stage evaluates it will generate an ack signal to the second stage, which will cause it to reset its outputs as well as lower its ack and req signal. Since the second stage has lowered its ack signal it can now consume a second data token.

2.6.2 Fine Grain Pipelining

The design methodology in this thesis is targets fine grain pipelining and small cells, where the forward latency is two gate delays. Fine grain pipelining is achieved by dividing the processing blocks to even smaller cells where each cell has its own input and output completion detector. For example a 32 bit multiplier can be implemented by using a 32 bit input completion detector at the inputs and a 32 bit output completion detector at the outputs. When the multiplier completes it processing and generates a 32 bit output, the output completion detector detects it and combined with the input completion detector generates and acknowledge. However the multiplier can only accept a new input only when the whole multiplier has finished processing. Therefore the throughput is limited to how fast the multiplier can multiply two numbers, generate and acknowledge and then reset. As in the synchronous case the throughput of the multiplier can be increased by further pipelining the multiplier. In asynchronous design, this can be done by constructing the multiplier using small number of cells such as adders and other logic gates which have their own input and output completion detectors. Not only now can the multiplier accept new

input as soon as the first row of logic in the multiplier has evaluated and reset but also simplifies the 32 bit completion detectors into 1 bit input and output completion detectors. For a 2 dimensional structure such as a multiplier this is called *2D Fine Grain Pipelining*. Also since fine grain pipelining uses pre-designed templates it has an added benefit of cell reuse and faster design time.

2.6.3 Performance Analysis of Linear Pipelines

Determining the performance of an asynchronous pipeline can be more complex than determining the performance of a synchronous pipeline. In an asynchronous pipeline, control signals govern token flow with local handshaking. Each four phase token is composed of a data element and a reset spacer. At any instant, the pipeline stages not occupied by data elements or reset spacers can be described as containing a *hole* or *bubble*. Control logic only allows an element to flow forward when the stage it will occupy is empty. When an element does flow forward, it leaves behind an empty slot. Thus, bubbles flow backward as they displace forward-flowing data elements and reset spacers. The performance can be limited by the supply of tokens, the supply of bubbles or the local control handshaking between two pipeline stages. In a pipeline, the left or input environment supplies data tokens and the right or output environment supplies bubbles.

In an asynchronous pipeline the time it takes for a data token to flow from the inputs to the outputs of one pipeline stage is defined as *forward latency*. The *reverse or backward latency* specifies the delay from the acknowledgment of a stage's output to the acknowledgment of the predecessor's output. The time difference two tokens passing through the same pipeline stage is called *cycle time*. The cycle time is the total of the forward and backward latency.

In an asynchronous pipeline, the per-stage forward or backward latency depends on the implementation of the circuit and the handshaking protocol. Pipeline stages, which can hold one data token using only one stage, are called *full buffers (*also known as *high capacity* or *slack)*. Pipeline stages, which need two stages two hold one data token are called *half buffers*. Assuming that the right environment is not operating, or has stalled handshaking with the last stage of an asynchronous pipeline, and the left environment keeps inserting as much data tokens as it can, the maximum possible tokens that the pipeline can hold is defined as the *static slack* of the pipeline. Assuming that the left environment is asserting and the right environment is consuming data tokens as fast as the pipeline can operate, the number of tokens needed for the pipeline to operate at the highest throughput is called the *dynamic slack* of the pipeline.

For a pipeline where the forward latency is less than the backward latency, the cycle time is dominated by the backward latency. For the opposite case the cycle time will be dominated by the forward latency. The following figure illustrates the throughput vs. number of tokens for a linear asynchronous pipeline. The left side of the triangle shows the characteristic of an asynchronous pipeline operating in a data-limited region. In this region, as the data tokens are inserted more frequently the pipeline operates at a higher throughput. The speed of the pipeline is limited by how fast data can be inserted into the pipeline. The right side of the triangle shows the characteristic of an asynchronous pipeline operating in a bubble-limited region. In this region the right environment cannot consume the data provided by the asynchronous pipeline and therefore the data tokens start to accumulate in the pipeline. Another way to view this region is to say that the handshaking between pipeline stages is limiting the throughput at which tokens can be

processed and therefore the overall pipeline performance starts to degrade. The figure has two throughput vs. tokens triangles. The left one is for a forward-latency limited pipeline and the right one is for a backward-latency limited pipeline.

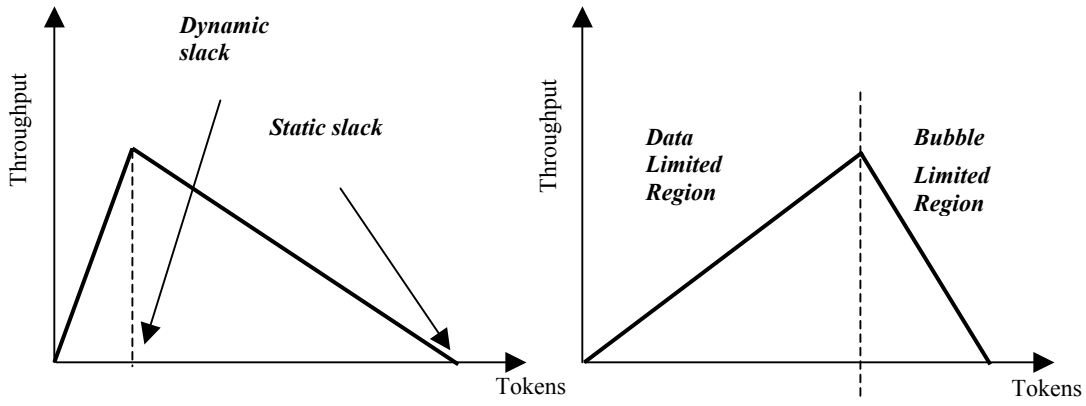


Figure 2-4: Throughput vs. tokens graphs

In order to determine the latencies and cycle time of a pipeline built out of a particular configuration of components in each stage, it is necessary to analyze the dependencies of the required sequences of transitions. These dependencies can be drawn in a marked graph [53], in which the nodes of the graph correspond to specific rising and falling transitions of circuit components, and the edges depict the dependencies of each transition on the output of other components. Unfolded dependency graphs are functionally equivalent to Signal Transition Graphs. STG's can be used to determine both the forward latency and the cycle time. The local cycle time is determined by *cyclic* paths in the STG. These cycles occur because a pipeline processes successive data tokens and the components in each stage go through a series of transitions. The transitions eventually

return a stage to the same *state*, where the state is defined by the output values of each component. Each transition in a STG can fire only when all of its predecessors have executed their specified transitions, and cannot fire again until all of its predecessors have fired again.

2.6.4 Non-Linear Pipelines

Recently many new asynchronous pipelines have been introduced. However most of them have been targeted for linear pipeline applications such as FIFOs. Real designs, however, require more complicated non-linear pipeline structures. In particular, linear pipeline stages have only a single input and a single output channel, where as non-linear pipelines stages can have multiple input and output channels. This section presents an overview of the challenges involved in designing non-linear pipelines. In particular we address issues with (i) synchronization with multiple destinations (for *forks*), and (ii) synchronization with multiple sources (for *joins*).

To introduce these issues we focus on *forks* and *joins*. A join is a pipeline stage with multiple input channels whose data is merged into a single output channel. A fork is a pipeline stage with one input channel and multiple output channels. Complex forks and joins can involve conditionally reading from or writing to channels based on the value of a control channel that is unconditionally read, as in a merge or split channel. Abstract illustrations of these channels are shown in Figure 3.4.

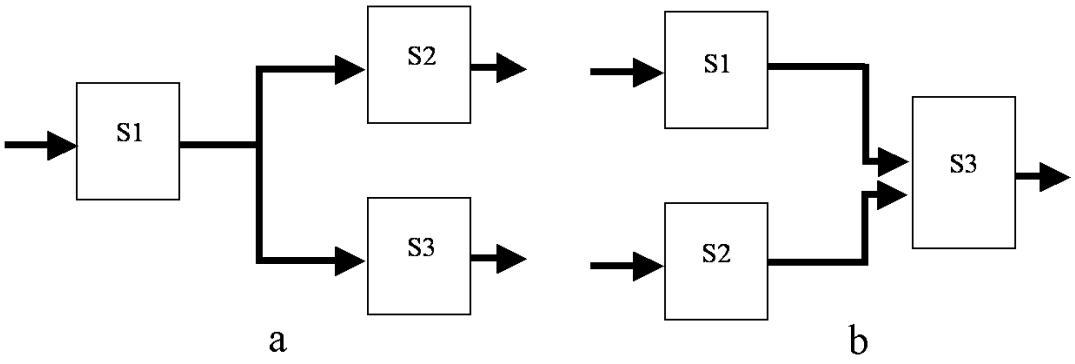


Figure 2-5: a) a fork and b) a join

Since a fork has multiple output channels, it must receive an acknowledgment signal from all of them before it precharges. A join, on the other hand, receives inputs from multiple channels and must broadcast its acknowledgment signal to all its input stages.

A join acts as a synchronization point for data tokens. The acknowledgment from the join should only be generated when all the input data has arrived. Otherwise a stage feeding a join, referred to as A, that is particularly slow in generating its data token may receive an acknowledgment signal when it should not, violating the 4-phase protocol. If the acknowledgment signal is de-asserted before the slow stage A generates its token, the token is not consumed by the join, as it should be. In fact, this token may cause the join to generate an extra token at its output, thereby corrupting the intended synchronization.

A conditional split is a combined fork and join where a control channel is used to determine which output is generated. The control may indicate to send the input data to any of the output channels, any combination of the output channels, or none of them. The third option is also known as a *skip*.

A conditional join is a join where the control signal, select, comes from another pipeline stage. The select signal controls which incoming channel should be read.

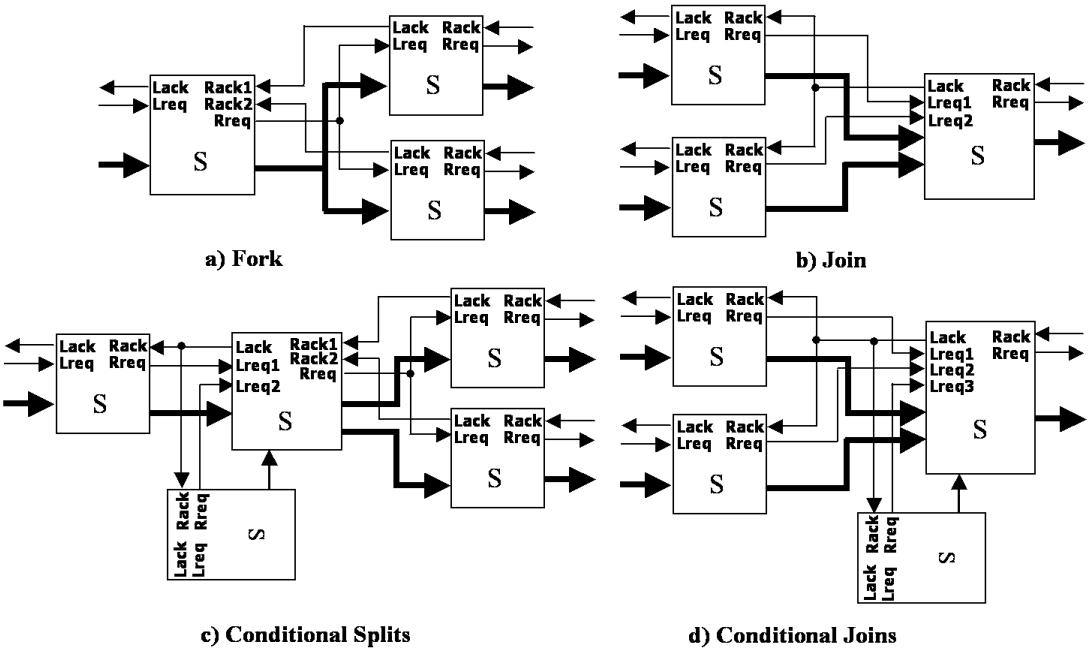


Figure 2-6: Fundamental non-linear pipeline structures

Chapter 3

3. New High Speed QDI Asynchronous Pipelines

In this chapter we introduce two new QDI templates that provide significant performance improvements over those proposed by Caltech without sacrificing quasi delay insensitivity. The key idea is to reduce the complexity of internal circuitry by intelligently reducing concurrency and using an additional wire for communication between pipeline stages. We present two templates: one that is a half-buffer which requires two pipeline stages to hold one data token and one full-buffer template that can itself hold one data token.

We first give background on Caltech's commonly used QDI templates, the Weak-Conditioned Half Buffer (*WCHB*), the Precharged Half Buffer (*PCHB*), and the Precharged Full Buffer (*PCFB*) templates [24].

3.1 Caltech's QDI templates

3.1.1 WCHB

Figure 3-1 shows a WCHB template for a linear pipeline with a left (L) and right (R) channel and an optimized WCHB dual-rail buffer. L0 and L1, R0 and R1 identify the false and true dual rail inputs and outputs, respectively. *Lack* and *Rack* are active-low acknowledgment signals. Note that we do not show staticizers that are required to hold state at the output of all C-elements.

The operation of the buffer is as follows. After the buffer has been reset, all data lines are low and acknowledgment lines, *Lack* and *Rack,* are high. When data arrives by one of the input rails going high, the corresponding C-element output will go low, lowering the left-side acknowledgment *Lack*. After the data is propagated to the outputs through one of the inverters, the right environment will assert *Rack* low, acknowledging that the data has been received. Once the input data resets, the template raises *Lack* and resets the output.

Since the L and R channels cannot simultaneously hold two distinct data tokens, this circuit is said to be a *half buffer* or has *slack* ½ [24]. This WCHB buffer has a cycle time of 10 transitions, which is significantly faster than buffers based on other QDI pipeline templates.

Another feature of the WCHB template is that the validity and neutrality of the output data R implies the validity and neutrality of the corresponding input data L. This is called *weak-conditioned* logic [20] and we will discuss its advantages and disadvantages after we discuss non-linear pipeline templates.

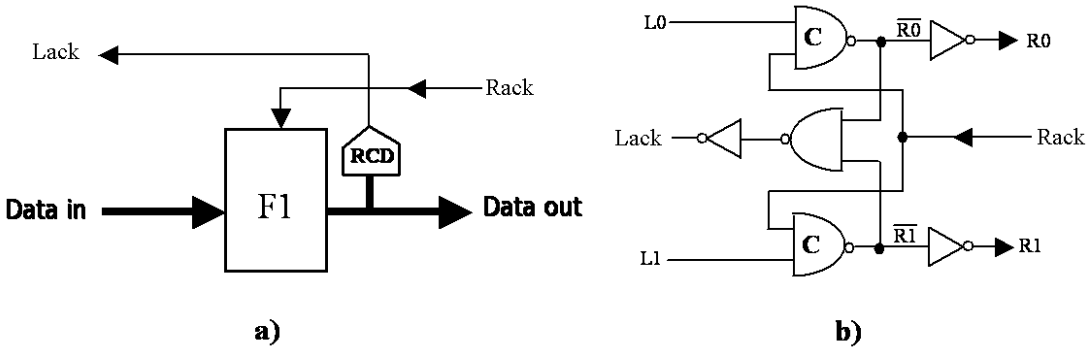


Figure 3-1: WCHB

3.1.2 PCHB and PCFB

Figure 3-2 shows the template for a pre-charged half-buffer (PCHB). Unlike the WCHB, the test for validity and neutrality is checked using an input completion detector. The input completion detector is denoted as LCD and the output completion detector as RCD.

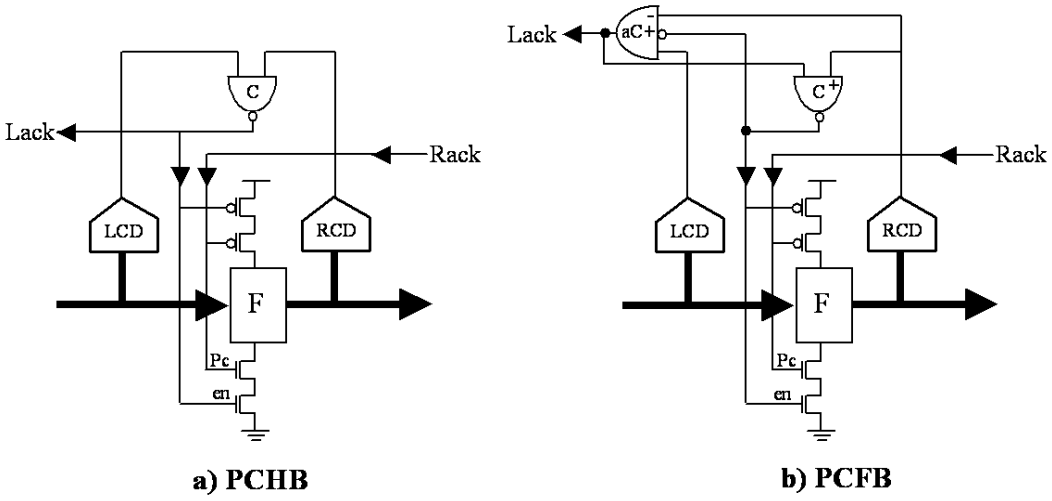


Figure 3-2: a) PCHB and b) PCFB templates

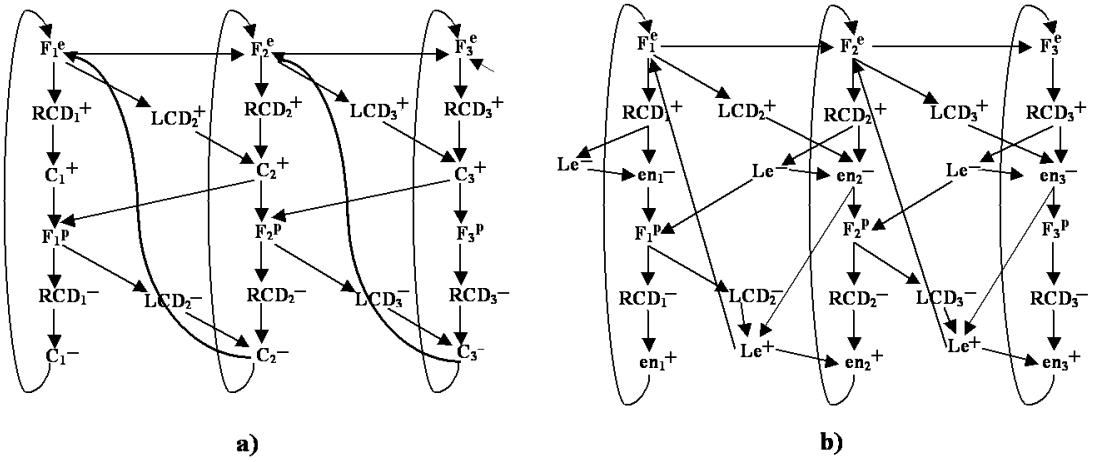


Figure 3-3: a) PCHB and b) PCFB STG

The function block need not be weak-conditioned logic and thus can evaluate before all the inputs have arrived (if the logic allows). However, the template only generates an acknowledgment signal *Lack* after all the inputs have arrived *and* the output has evaluated. In particular, the LCD and the RCD are combined using a C-element to generate the acknowledgment signal.

A few minor aspects of this template should also be pointed out. First, because the C-element is inverting the acknowledgment signal is an active-low signal. Second, the *Lack* signal is often buffered using two inverters before being sent out. Another two inverters are also often added to buffer the internal signal *en* that controls the function block. For simplicity, these buffering inverters will not be shown in the figures in this paper.

The protocol for a PCHB pipeline stage is captured by the STG for a three-stage pipeline illustrated in Figure 3-3. From the STG, it is possible to derive the pipeline's analytical cycle time:

$$T_{PCHB} = 3.\ t_{Eval} + 2.\ t_{CD} + 2.\ t_c + t_{prech}$$

Due to the extra buffering and bubble shuffling, the cycle time generally amounts to 14 gate delays or *transitions*.

The PCFB template and its STG are shown in Figure 3-2(b) and Figure 3-3(b). The PCFB is more concurrent than the PCHB because its L and R handshakes reset in parallel at the cost of requiring an additional state variable. The PCFB analytical cycle time is:

$$T_{PCFB} = 2.\ t_{Eval} + 2.\ t_{CD} + 2.\ t_c + t_{prech}$$

which generally amounts to 12 transitions. Here t_{CD} takes two transitions, one of the C-elements takes one transition, and the other takes two transitions.

3.1.3 Why Input Completion Sensing?

A join is a pipeline stage with multiple input channels whose data is merged into a single output channel. A fork is a pipeline stage with one input channel and multiple output channels. Complex forks and joins can involve conditionally reading from or writing to channels based on the value of a control channel that is unconditionally read, as in a merge or split channel.

Since a fork has multiple output channels, it must receive an acknowledgment signal from all of them before it precharges. A join, on the other hand, receives inputs from multiple channels and must broadcast its acknowledgment signal to all its input stages.

A join acts as a synchronization point for data tokens. The acknowledgment from the join should only be generated when all the input data has arrived. Otherwise a stage feeding a join, referred to as A, that is particularly slow in generating its data token may receive an acknowledgment signal when it should not, violating the 4-phase protocol. If the acknowledgment signal is deasserted before the slow stage A generates its token, the token is not consumed by the join, as it should be. In fact, this token may cause the join to generate an extra token at its output, thereby corrupting the intended synchronization.

Validity of data should be checked on all input channels before the acknowledgment signal is asserted to prevent the incorrect insertion of a token caused by a slow/late input channel. Neutrality should be checked to guarantee that the previous stages have been precharged, so that the acknowledgment signal is not deasserted too early, thereby violating the four-phase protocol on any stage slow to precharge.

The templates presented in this section check validity and neutrality in different ways. Because the function block in WCHB template is weak-conditioned, the output completion detector implicitly checks validity and neutrality of the input data token. In the WCHB buffer the weak conditioned function block is a simple C-element. However, for more complex non-linear pipelines, weak-conditioned function blocks unfortunately require complex nmos and pmos networks. This results in slower forward latency and bigger transistor sizes. As an example, a weak-conditioned dual-rail OR is shown in Figure 3-4.

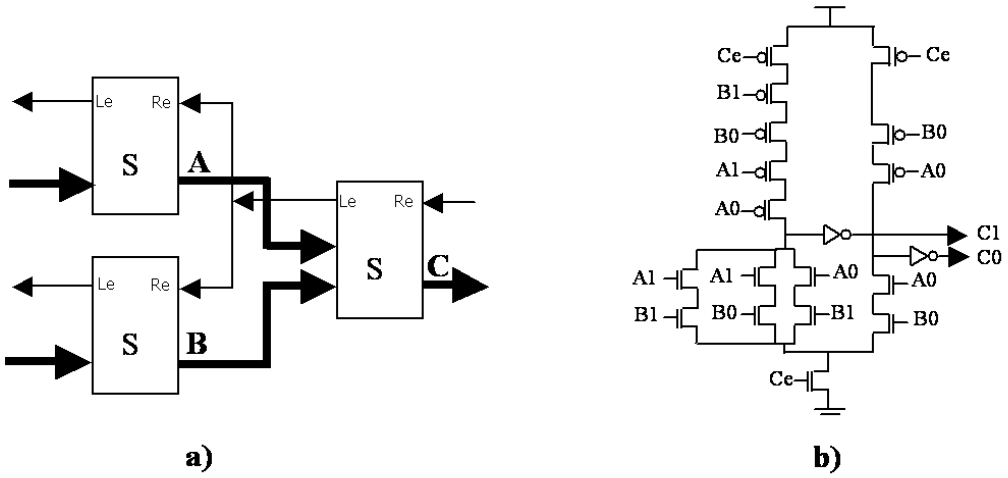


Figure 3-4: An OR gate implementation using weak conditioned logic

3.2 New QDI Templates

One optimization that can be applied to the PCHB and PCFB templates is to merge the LCD of one stage with the RCD of the other by adding an additional request line to the channel. This is shown in Figure 3-5 for a PCHB template.

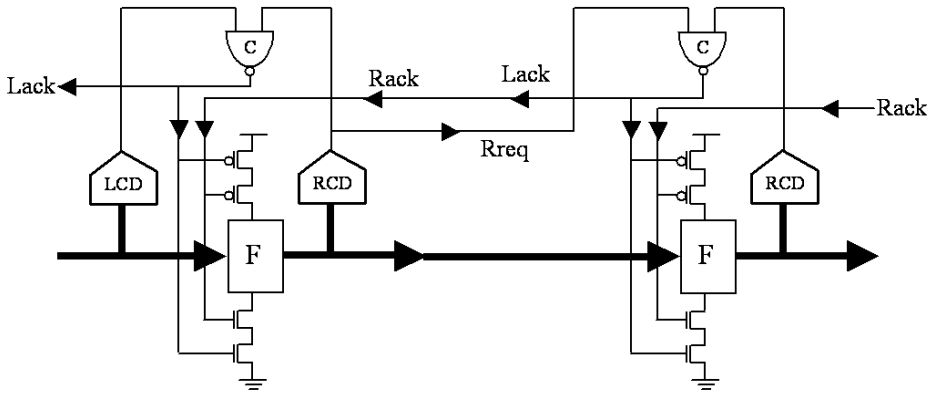


Figure 3-5: Optimized PCHB for a 1-of-N+1 channel

The request line indicates the assertion/de-assertion of the input data, as in the bundled-data channel. However in contrast to a bundled-data channel, the data is sent using 1-of-N encoding, yielding what we call a *1-of-N+1 channel*. The request line, at least from the channel point of view, may appear redundant. However, the request line enables the removal of the input completion detector thereby saving area and reducing capacitance on the data lines. Moreover, the request line does not significantly impact performance, the template is still QDI, and the communication between stages remains delay-insensitive.

In this section we propose two new 1-of-N+1 QDI templates that intelligently reduce concurrency to reduce the stack size of the function blocks and thereby improve performance.

3.2.1 RSPCHB

The key goal of the RSPCHB compared to the PCHB is to eliminate the need of the enable signal *en* from the control of the function block. We now explain that the need for this enable signal is only to support concurrency in the system that effectively does not improve performance.

More specifically, in the PCHB template the output of the LCD and RCD are combined using a C-element to generate the acknowledgment signal *Lack*. This supports the integration of the handshaking protocol with the validity and neutrality of both input and output data, which removes the need for the function block to be weak-conditioned, but also requires the use of the *en* signal. It is this replacement however that introduces more concurrency than is necessary.

In particular, in the case of a join, the non-weak-conditioned function block may generate an output as soon as one the input channels provide data. In response, the RCD of the join will assert its output. Meanwhile, any subsequent stage can receive this data, evaluate, assert both its LCD and RCD outputs, and assert its acknowledgment signal. Although the join can receive this acknowledgment, it will not precharge until after *en* is asserted. The *en* signal delays the precharge of the circuit until after the acknowledgement to the input stages has been asserted. This delay is critical to prevent the precharge from triggering the RCD to deassert which would prevent the C-element from ever generating the acknowledgment.

If only the generation of the acknowledgment signal from any stage subsequent to the join was delayed until all input data to the join has arrived and been acknowledged, then the *en* signal could be safely removed. In fact, such a delay of the acknowledgement would not generally impact performance because the join is the performance bottleneck for the subsequent stages. Therefore, this added concurrency is essentially unnecessary.

We propose a different pipeline template, which reduces this unnecessary concurrency to eliminate the internal *en* signal, thereby reducing the transistor stack sizes in the function block. We refer to this new QDI pipeline template, illustrated in Figure 3.6(a), as a *Reduced*

Stack Precharged Half Buffer (RSPCHB). A specific form of this template for dual-rail data is shown in Figure 3-6(b). Notice that we optimized the RCD block by tapping its inputs before the output inverter and using a NAND gate instead of an OR gate.

The unique feature of the RSPCHB is that it derives the request line from the output of the C-element instead of the RCD. (In particular, since the output of the C-element is active low and the request line is active high, the output of the C-element is sent through an inverter before driving *Rreq*.) The impact of this change is that the assertion/de-assertion of *Rreq* is delayed until after all *Lreq*'s are asserted/de-asserted.

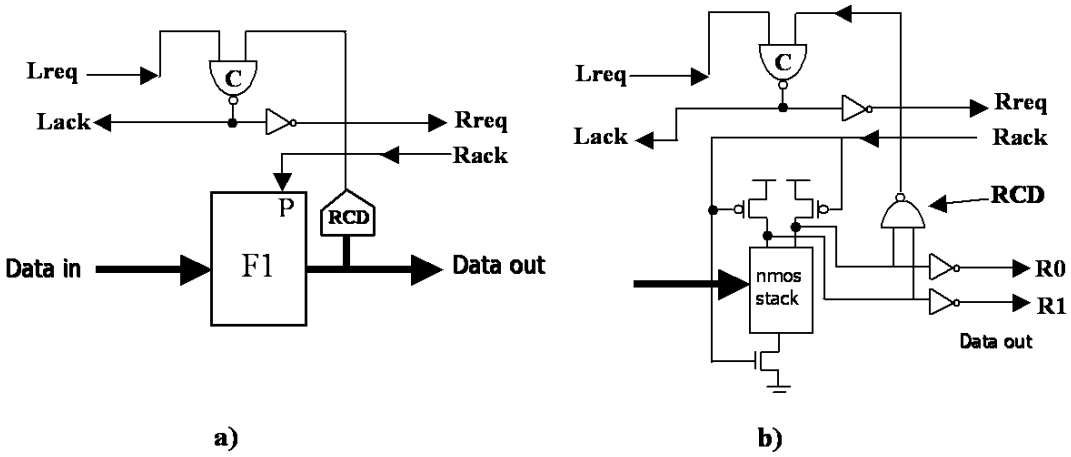


Figure 3-6: a) Abstract and b) detailed QDI RSPCHB pipeline template

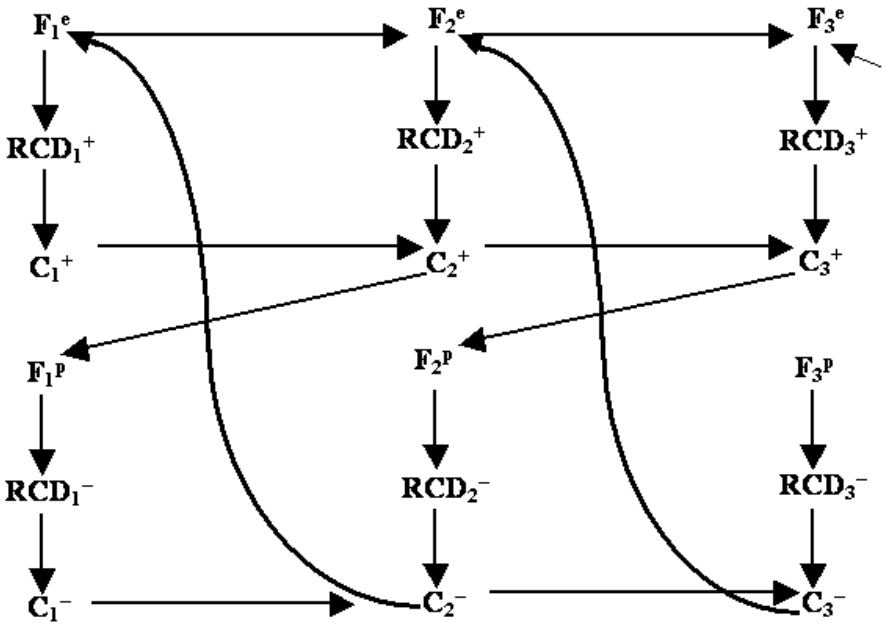


Figure 3-7: The STG of the RSPCHB

As a consequence, the acknowledgment from a subsequent stage of the join may be delayed until well after its data inputs and outputs are valid. More specifically, the stage will delay the assertion of its acknowledgment signal until all *Lreq*'s are asserted which can occur arbitrarily later than the associated data lines becoming valid. This extra delay, however, has no impact on steady-state system performance because the join stage is the bottleneck, waiting for all its inputs to arrive before generating its acknowledgement. In fact, this change yields a template with no less concurrency than WCHB.

The advantage of this generation of the request line is that the function block does not need to be guarded by the enable signal. In particular, it is now sufficient to guard the function block solely by the *Pc* signal because the *Pc* signal now properly identifies when inputs and outputs are valid. Namely, the function block is allowed to evaluate when *Pc* is deasserted which occurs only after all inputs and outputs data lines are reset. Similarly, it is

allowed to precharge when *Pc* is asserted which occurs only after all input and output data lines are valid.

The RSPCHB is still QDI, however, the communications along the input channels to joins become QDI instead of delay-insensitive (other channels remain delay-insensitive). In particular, the assumption that must be satisfied is that the data should reset before the join stage enters a subsequent evaluation cycle. If we assert that the fork between the function block, the RCD, and the next stage is *isochronic* [33], this assumption is satisfied. In particular, the data line at the receiver side is then guaranteed to reset before the request line *Rreq* resets because only after the data lines reset can the RCD trigger the C-element, subsequently triggering *Rreq*. The analytical expression for the timing margin associated with this isochronic fork assumption can be derived from the abstract STG of the RSPCHB shown in Figure 3-7. In particular, the delay difference between the resetting of the data and the associated request line should be less than:

$$T_{Margin} = 2.\ t_{Inv} + 1.\ t_{CD} + 3.\ t_c$$

This margin is between 6 and 8 gate delays depending on buffering and is easily satisfied with modern routers.

Notice that this timing assumption only applies to input channels of join stages because non-join stages must receive both valid data and a valid *Lreq* before generating valid output data or valid *Rreq*.

The analytical cycle time of the RSPCHB can be derived from the STG shown in Figure 3.7 as:

$$T_{RSPCHB} = Max(\ 3.\ t_{Eval} + 2.\ t_{CD} + 2.\ t_c + t_{prech},\ t_{Eval} + 2.\ t_{CD} + 4.\ t_c + t_{prech})$$

With bubble shuffling, RSPCHB and PCHB have equal numbers of transitions per

cycle. The advantage of RSPCHB is that the lack of an LCD and reduced stack size of the function block, which reduces capacitive load, and yields significantly faster overall performance. The cost of this increase in performance is that it requires one extra communicating wire between stages.

A fork can be implemented easily by either using a C-element to combine the acknowledgment signals from the forking stages or by combining them by increasing the stack size of the function block. Similarly a join can be implemented, by combing the request lines in the C-element and forking back the acknowledgment signal.

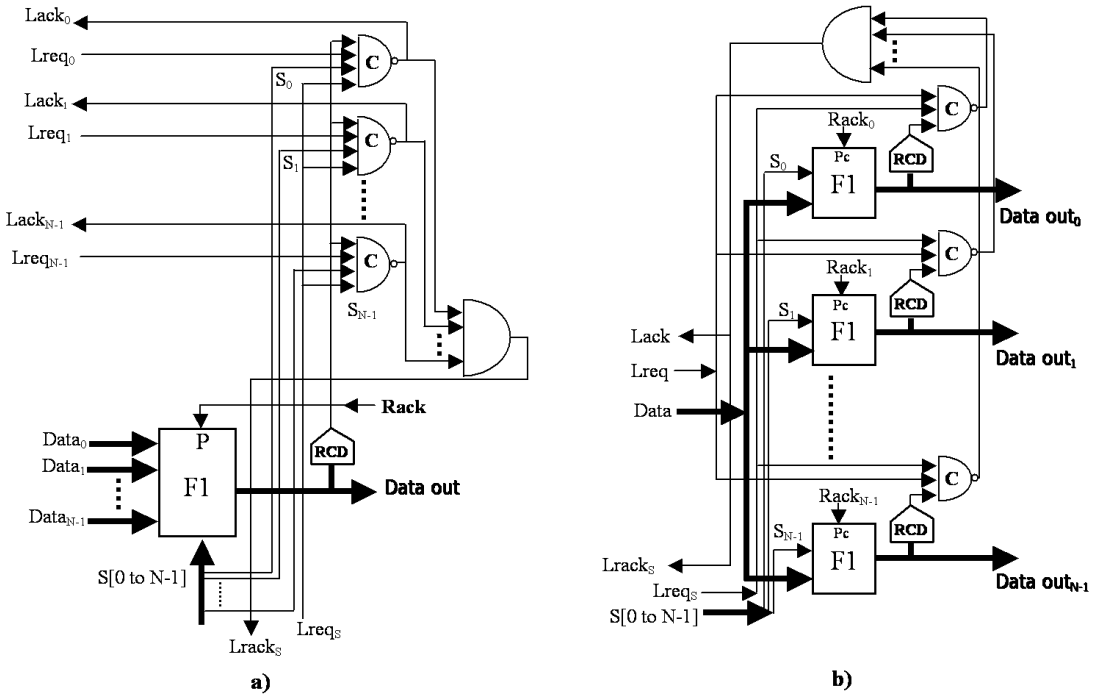


Figure 3-8: Conditional a) join and b) split using RSPCHB

Consider the slightly more complicated template for a conditional join in which a control channel S is used to select which input channel to read and write the read data

token to the single output channel illustrated in Figure 3-8(a). The template has one C-element per input channel, each responsible for generating the associated acknowledgement signal. Each C-element is triggered by not only the RCD output, but also the corresponding control channel bit. The collection of C-elements are simply ORed to generate the $Lreq_s$ because the C-elements are mutually exclusive. This template can be easily extended to handle more complex conditionals in which multiple inputs can be read for some values of the control.

The template for the conditional fork is shown in Figure 3-8(b). Here, the functional block, the RCD and the C-element are repeated for each output channel. The select data lines ensure only one function block evaluates. All C-elements are combined using an AND gate to generate the acknowledgement for the select channel. (This is because both the C-element outputs and the acknowledgement signal are active low.) This template can easily be extended to handle the generation of multiple outputs in response to some values of the control.

A common example of a conditional fork is a *skip* in which depending on the control value the input is consumed but no output is generated. The implementation has a skip output acting as an internal N+1 output rail that is not externally routed and is triggered upon the skip control value. A skip in which all control values generate no output is called a *bit bucket* [54].

Figure 3-9 shows a one-bit memory implemented using a RSPCHB template. A and C represent the input and output channels. B is the internal storage. S is an input control channel that selects the write or read operation. When S0 is high, the memory stores the value at the input channel A to the internal storage B. When S1 is high, on the other hand,

the memory is read, that is, the stored memory value is written to the output channel C. For a write, both input data and control channels are acknowledged, while for a read, only the control channel is acknowledged.

The write and read operations are as follows. After reset, the memory, stored in the dual-rail *Memory Unit*, MU (similar to [24]) is initialized to some value and one of the rails of the internal signal B is high. When an input A is applied and S0 is high, one of rails of B is asserted high, thereby storing the data. The *Memory Completion Detector, MCD*, detects that the value in the memory is updated, and asserts its output. The output of the MCD as well as the request lines from the data and control channel drive a C-element, which generates the acknowledgment signal $Lack_A$. When S1 is high, on the other hand, the internal data stored in B is sent to the output channel C. When an acknowledgment is received from the output channel C, the outputs are reset but the data stored remains unchanged. The control channel S is acknowledged for both write and read operations using an AND gate driven by the two C-element outputs.

Notice that the memory is actually implemented by merging two RSPCHB units. The first one is used to store data (write), and the second one to send it to the outputs (read). The MCD detects the completion of the write operation and resets when all inputs are lowered.

The MCD can be simplified by replacing the pmos transistors driven by A0 and A1 with a pmos transistor driven by $Lack_A$. However this requires that the delay difference between the data lines of channel A and its associated request line is not long enough to cause short circuit current. This restriction can be removed by also controlling the nmos stack by also adding one more nmos transistor driven by the $Lack_A$ signal. The overall

benefit however is not clear.

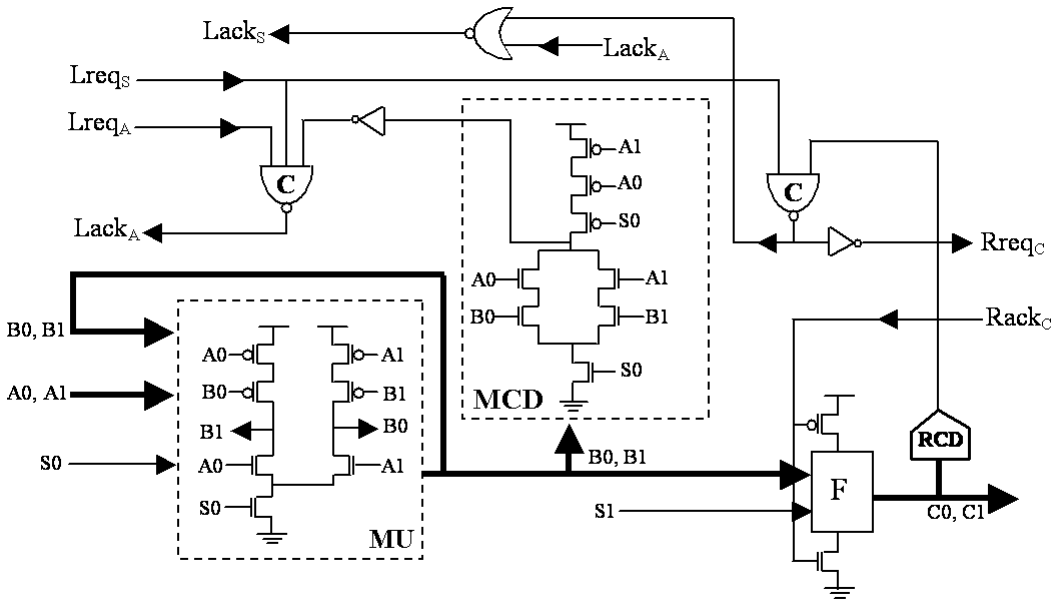


Figure 3-9: A RSPCHB 1-bit memory

3.2.2 RSPCFB

Our second new *1-of-N+1* QDI pipeline template is a full buffer constructed by merging our RSPCHB with a modified WCHB. An abstract illustration of this *reduced stack pre-charged full buffer* (*RSPCFB*) is shown in Figure 3-10(a) and a more detailed implementation for dual-rail data is shown in Figure 3-10(b).

The RSPCFB has two new features. First, the inverters from both of the half buffers have been removed to keep the forward latency of the new template at two gate delays. We assert that the inverters between the two half buffers can safely be removed because the RSPCHB has little gate load and wire load can be minimized by placing/routing this template as a single unit. The output inverters are only necessary if this unit is driving a significant load and can be added as necessary. (However a staticizer, not shown, is still

necessary.) Second, the WCHB has to be modified to accept an input request signal and generate an output request signal. This input request signal drives a C-element whose other input is the RCD output. This C-element then triggers the internal acknowledgement to the RSPCHB part instead of the RCD alone. In addition, the output request signal is implemented by simply tapping of a signal from the RCD output. One other difference is that the request signal is now active low because the inverters have changed locations (i.e., *bubble shuffling* [50]).

The circuit operates as follows. The RCD of the RSPCHB part detects the evaluation of the function block and asserts its output. The output of the RCD drives the C-element, which generates the acknowledgment signal *Lack* to the previous stage after all the request lines associated with the data also arrive. If the next stage is ready to accept new data, the acknowledgment signal *Rack* should already be de-asserted, allowing the C-elements in the forward path to pass the data to the next stage. Subsequently, the WCHB's RCD will assert its output asserting the request signal to the next stage. The output of the RCD also drives the C-element C_b, which asserts the internal acknowledgement back to the RSPCHB part, allowing the function block to precharge. When the acknowledgment signal *Rack* is de-asserted, the C-element in the forward path will de-assert its outputs. This will trigger the WCHB's RCD to de-assert *Rreq*, the C-element C_b to de-assert the internal acknowledgement back to the RSPCHB, and thereby enable the function block to re-evaluate.

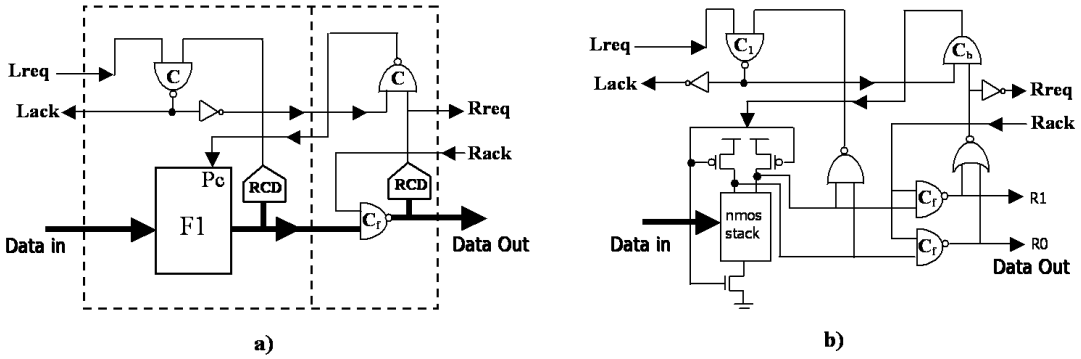


Figure 3-10: a) Abstract and b) detailed RSPCFB

Notice that the *Rreq* of the RSPCFB is taken from the output of the RCD instead of the C-element, unlike the RSPCHB. This is because the WCHB part has weak-conditioned logic, which will not reset until all inputs, including inputs from the RSPCHB part, have reset. This implicitly avoids the problem of preventing the assertion of the acknowledgement back to the RSPCHB part that delaying *Rreq* solved. The advantage of this is that the *Rreq* can be generated earlier. The disadvantage is that this reduces are timing margin on input channels to joins to 5 to 7 gate delays, depending on buffering.

The RSPCFB has 10 transitions per cycle, less than Caltech's PCFB, which has 12 transitions. The analytical cycle time, using the STG in Figure 3-11, can be expressed as:

$$T_{RSPCFB} = Max(\, 3.\, t_{Eval} + 2.\, t_{CD} + 2.\, t_c + t_{prech},$$

$$2.\, t_{Eval} + t_{CD} + 3.\, t_c + t_{NAND})$$

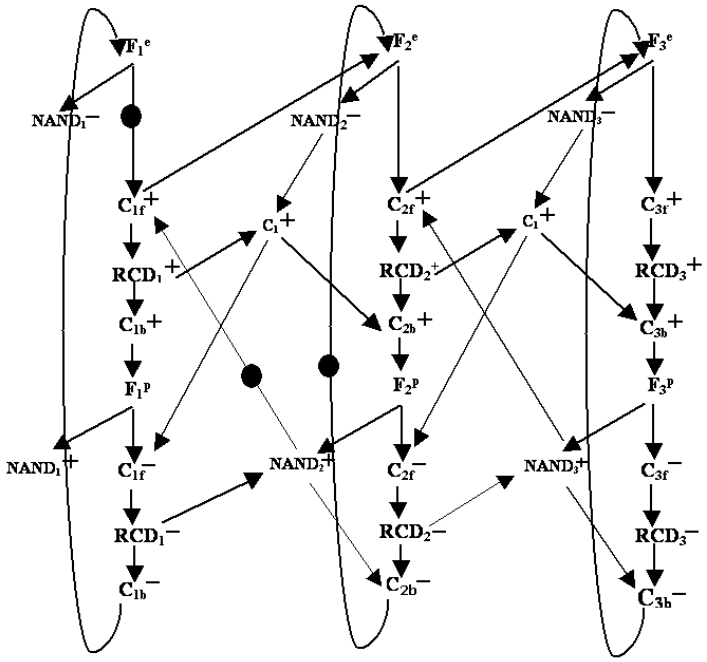


Figure 3-11: a) Abstract and b) detailed RSPCFB

The RSPCFB can be extended to handle non-linear pipeline structures in the same way as the RSPCHB without any additional timing assumptions.

3.2.3 FSM Design

One of the most important aspects of a complete system design is the implementation of the controller. An FSM is actually a state holding circuit, which only changes its state when the expected inputs for that state are available. One way to build an asynchronous FSM is to feed the outputs of the pipeline stage back to its inputs using buffers to hold the data (also proposed in [24]). This technique is similar to the synchronous case. In addition it requires no new circuits and can be easily applied to template-based design. Figure 3-12 shows an abstract FSM.

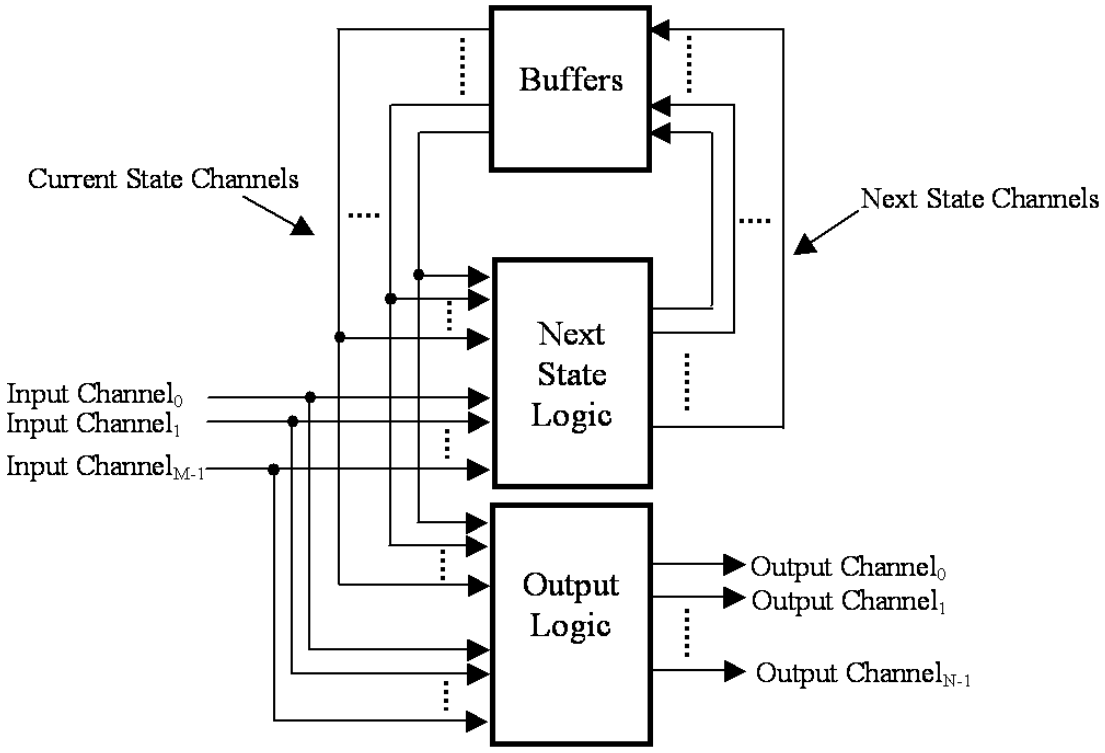


Figure 3-12: An abstract asynchronous FSM

Each channel either is an input, an output, or holds state. The next and current state channels can be implemented with either 1-of-N+1 channels, ideally suitable for one-hot state encoding of the FSM. The next state and the output logic blocks are complex QDI pipeline stages, which can have multiple function blocks inside. These multi-input multi-output conditional blocks are implemented the same way as the conditional read and write blocks shown previously.

The simplicity of this method for designing FSMs allows all known synchronous design techniques for generating Boolean next state and output expressions directly to be applied. Also the next state logic can be implemented as several stages of pipelined logic, reducing the number of necessary feedback buffers. Aside from using feedback buffers,

which for a high number of states can yield a large circuit there are also other ways to design circuits that hold state.

Another way to implement state holding is not to generate an acknowledgment signal. This avoids the reset of the input data. Although this technique can be used for specific problems like loop control [24], it is very limited. A more general way is to use the memory block for state holding presented in the previous section. This memory can also be further modified by adding one more internal state to allow read and write operations at the same cycle making it more suitable to be used as a register in FSMs.

3.2.4 Simulation Results

Both Verilog and HSPICE simulations were performed to check the correctness of functionality and to measure performance of all the proposed linear and non-linear pipelines.

A structural Verilog netlist has been generated with both random and unit delays. The Verilog code is written such that in the case of any hazard on any of the signals the simulator asserts a warning or error. The Verilog simulations with unit delay were performed for cycle time analysis, and the simulations with random delay were performed to intuitively verify that the circuits are QDI. No asserts have been found for random delays and the unit delay simulations confirm the transition counts.

HSPICE simulations were performed using a 0.25 TSMC process with a 2.5V power supply at 25^{o}C. The purpose of these simulations was to confirm the results obtained by the Verilog simulations, and to compare the throughputs of the proposed pipelines with the pipelines presented in the background section. Since the goal was comparison, no

attempt was made to fine-tune the transistor sizing to achieve optimum performance. In particular, all transistors were sized in order to roughly achieve a gate delay equal to a small inverter (Wnmos=0.8um, Wpmos=2um, and L=0.24um) driving a same-sized inverter. For the purposes of this comparison, wire delay also has been ignored.

For the half buffers, the PCHB and the RSPCHB, a linear dual-rail pipeline of buffers with 60 stages has been constructed to achieve a static slack of 30, which means that it can hold 30 distinct data tokens. For the full buffers, the PCFB and the RSPCFB, 30 stages have been used to achieve the same static slack. All pipelines can hold 30 distinct tokens. Figure 3-13(a) shows throughput versus tokens triangles for the half buffers and Figure 3-13(b) shows them for the full buffers. The triangles for the PCHB and PCFB are indicated with the dotted lines. Approximately 15 distinct points have been obtained per pipeline for the triangle graphs using HSPICE simulation. One key result obtained from this simulation is the *dynamic slack* of each pipeline, which is the number of tokens required to achieve maximum throughput [23], [24].

The PCHB achieves a maximum throughput of 772MHz with a dynamic slack of 7.3. The RSPCHB is faster with a maximum throughput of 920MHz and a dynamic slack of 8.25. The throughput improvement is approximately 20%. For the full buffers, the PCFB achieves a maximum throughput of 707MHz and a dynamic slack of 3.7. The RSPCHB is faster with a maximum throughput of 1000MHz and a dynamic slack of 5.9. The speed improvement is approximately 40%, however due to the C-elements in the forward path of the RSPCFB, the forward latency is about 15% slower. In both the half and full buffer, we achieved higher dynamic slack. This means that our templates support more system-level concurrency and higher stage utilization.

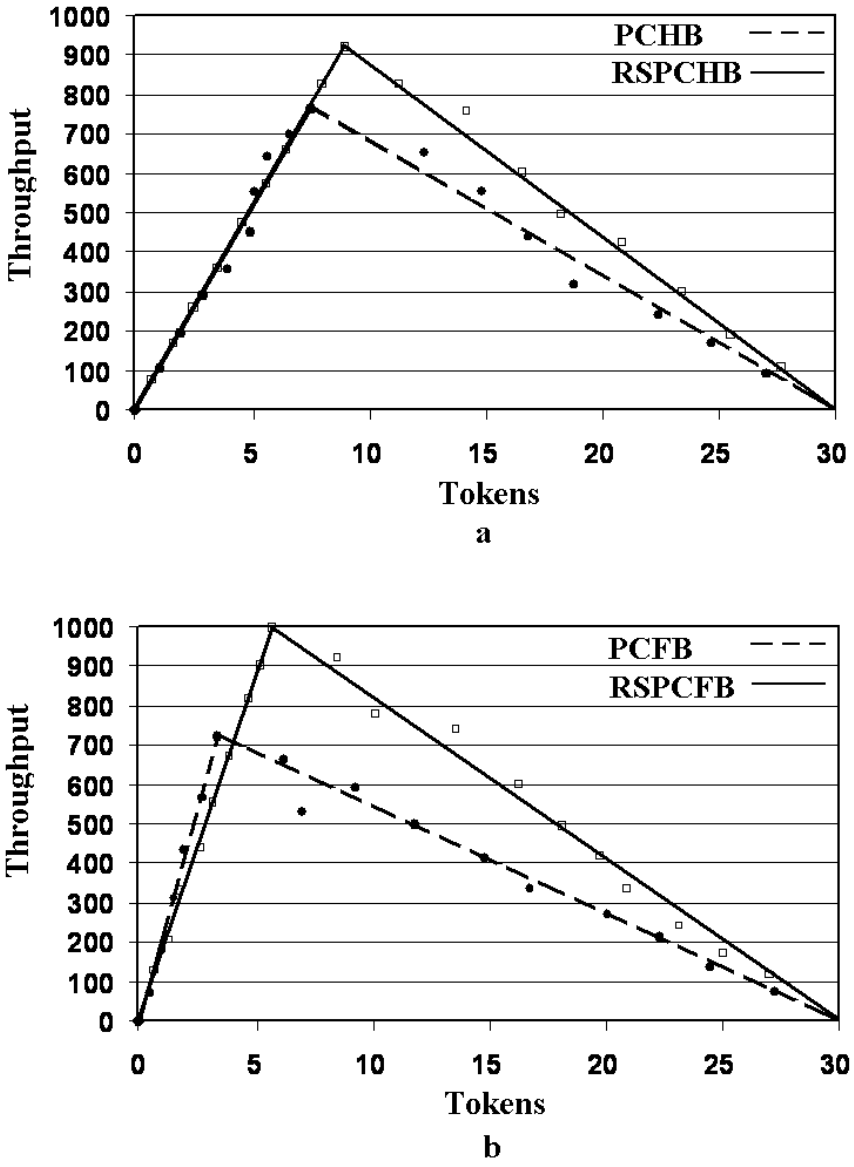


Figure 3-13: Throughput versus tokens for a) the PCHB and RSPCHB and b) the PCFB and RSPCFB linear pipelines

Notice that although the PCFB has 12 and the PCHB has 14 transitions per cycle, the PCFB was slower. This is partially due to the heavier load on the internal wiring in the

PCFB compared to the PCHB. Clearly, careful transistor sizing and buffering can improve the performance of all pipeline templates, however, we expect the relative performances to remain approximately the same.

3.2.5 Conclusions

This chapter has introduced new high-speed QDI asynchronous pipeline templates for non-linear dynamic pipelines, including forks, joins, and more complex configurations in which channels are conditionally read and/or written. Timing analysis and HSPICE simulation results demonstrate that our new RSPCHB achieves ~20% throughput over its PCHB counterpart and our new RSPCFB achieves ~40% throughput improvement over the PCFB counterpart.

Chapter 4

4. Timed Pipelines

A number of fast asynchronous fine-grain pipeline templates have been proposed for high-speed design, including IPCMOS [2] and GasP [55], [56]. These ultra-high speed designs have very aggressive timing assumptions that introduce stringent transistor sizing requirements and high demands on post-layout verification.

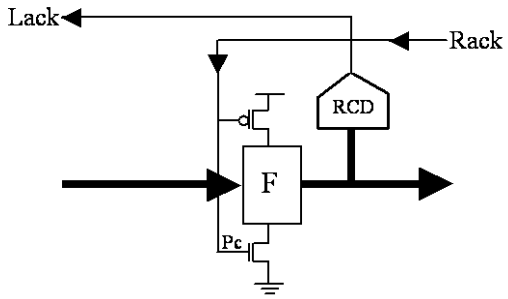
Researchers from Columbia University have recently proposed several high-speed dynamic-logic pipeline templates that achieve comparable performance with much less stringent timing assumptions[57], [58]. These pipelines are based on Williams' well known PS0 pipelines which is an optimized version of Caltech's PCHB, where the optimization takes place by removing the input completion detector and adding a timing assumption to assure correct operation. The Columbia pipelines, which also have PS0's timing assumption, were introduced for linear datapaths (i.e. without forks and joins), although preliminary solutions for handling joins were proposed in [58]. In addition, an initial approach to handling slow or stalled environments for the limited case of linear pipelines was also proposed in [57]. However, the synchronization problems that arise when using arbitrary forks and joins are much more complex and challenging, and the approaches of [57],[58] do not address these issues. This chapter attempts to fill this void.

The contribution of this chapter is a set of five new non-linear pipeline templates that extend the Columbia pipelines to handle non-linear datapaths. Both of Columbia's dynamic-logic pipeline styles are targeted: lookahead pipelines (LP) [57] and high-capacity

pipelines (HC) [58]. Several distinct lookahead pipeline styles were proposed in [57], both single-rail and dual-rail. This chapter builds upon one representative each of single-rail ($LP_{SR}2/2$) and dual-rail (LP3/1) lookahead pipelines, and also upon the single-rail high capacity pipeline (HC). The ideas presented here, however, can be easily adapted to the remaining styles. First we present Williams' PS0 pipelines. Then we review Columbia's three asynchronous pipelining styles: (i) $LP_{SR}2/2$, a single-rail lookahead pipeline, (ii) LP3/1, a dual-rail lookahead pipeline, and (iii) HC, the high-capacity pipeline. Finally we present solutions to extend these pipelines for non-linear applications.

4.1 Williams' PS0 Pipeline

Figure 4-1 shows one stage of Williams' PS0 pipeline [23]. The pipeline stage consists of a dual rail function block and a completion detector. The output of the completion detector is fed back to the previous stage as the acknowledgment signal. The completion detector checks the validity or absence of data at the outputs. There is no input completion detector.



a) Williams' PS0

Figure 4-1: Williams' PS0 pipeline stage

The function block is implemented using dynamic logic. The precharge/evaluation control input Pc, of each stage comes from the output of the next stage's completion detector. The precharge logic can hold its data outputs even when its inputs are reset, therefore it also provides the functionality of an implicit latch. Each completion detector verifies the completion of every computation and precharge of its associated function block.

The operation of the PS0 pipeline is quite simple. Stage N is precharged when stage N+1 finishes evaluation. Stage N evaluates when stage N+1 finishes reset. This protocol ensures that consecutive data tokens are always separated by reset tokens, holes.

The complete cycle of events for a pipeline stage is derived by observing how a single data token flows through an initially empty pipeline. The sequence of events from one evaluation by stage 1, to the next is: (1) Stage 1 evaluates, then (2) stage 2 completes, then (3) stage 2's completion detector detects completion of evaluation, and then (4) stage 1 precharges. At the same time, after completing step (2), (3)' stage 3 evaluates, then (4) stage 3's completion detector detects completion of evaluation and initiates the precharge of stage 2, then (5) stage 2 precharges, and finally, (6) stage 2's completion detector detects the completion of precharge, thereby releasing the precharge of stage 1 and enabling 1 to evaluate once again. Thus there are six events in the complete cycle for a stage from one evaluation to the next.

The protocol for a PS0 pipeline stage is captured by the STG for a four-stage pipeline illustrated in Figure 4-2. From the STG, it is possible to derive the pipeline's analytical cycle time:

$$T_{PS0} = 3.\ t_{Eval} + 2.\ t_{CD} + t_{prech}$$

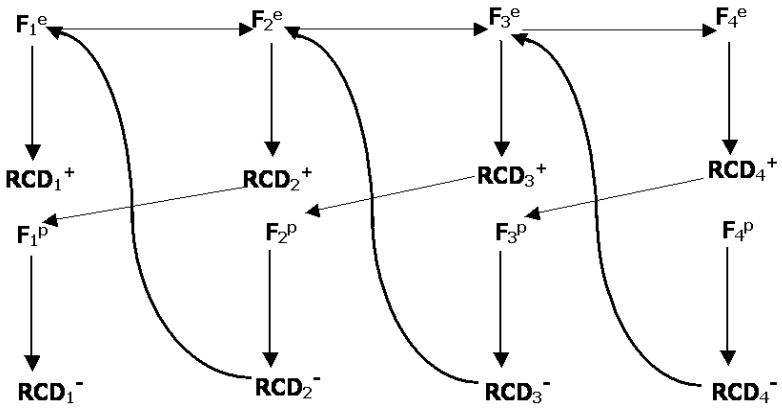


Figure 4-2: The STG of the PS0 Pipeline

Williams has simplified the pipeline stage at the expense of sacrificing delay insensitivity. Williams' PS0 pipeline has the following timing assumption:

$$T_{Prech\_1} + t_{CD\_1} <= t_{Eval\_3} + t_{CD\_3} + t_{Prech\_2} + t_{CD\_2}$$

which must be verified during physical design.

4.2 Lookahead Pipelines (Single Rail)

Figure 4-3(a) shows the structure of one stage of the $LP_{SR}2/2^1$ lookahead single-rail pipeline [57]. Each stage has a dynamic function block and a control block. The function block alternatively evaluates and precharges. The control block generates the bundling signal, *done*, to indicate completion of evaluation (or precharge). The bundling signal is passed through a suitable delay line, allowing time for the dynamic function block to complete its evaluation (or precharge). Note that there is one function block (F) for each individual output rail of the stage, and different function blocks can sometimes share precharge and evaluate (foot) transistors.

This pipeline style has two important features. First, the completion signal, *done*, is sent to the previous stage as an acknowledgment (Lack) by tapping off from before the matched delay. This early tap-off is safe because a dynamic function block typically is immune to a reset of its inputs as soon as the input data has been absorbed by the first level of dynamic logic. The second feature is that the control signal, Pc, is applied to both the control block and the function block in parallel. Therefore, the function block can be precharge-released even before the arrival of new input data. This early precharge-release is safe because the dynamic logic will compute only upon the receipt of actual data. Both of these features eliminate critical delays from the cycle time, resulting in very high throughput.

The analytical cycle time can be expressed using the following components:

t_{Eval} = delay of function block evaluation

t_{gc} = delay of control (generalized C-element)

[1] The 2/2 label characterizes the operation of the stage of a pipeline: 2 componnents in the evaluation phase and 2 component delays in the precharge phase, forming a complete cycle.

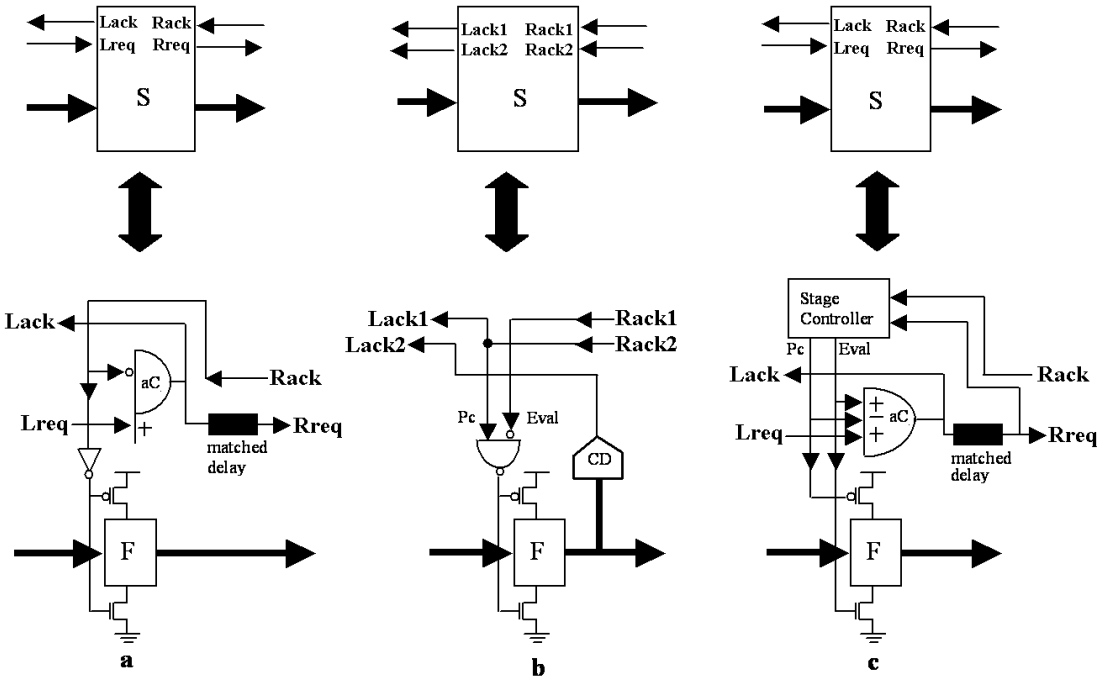


Figure 4-3: a) $LP_{SR}2/2$ b) LP3/1 and c) HC pipelines

For correct operation, the matched delay t_{delay} must satisfy, $t_{delay} \geq t_{Eval} - t_{gc}$. For ideal operation, we will assume that t_{delay} is no larger than necessary, $t_{delay} = t_{Eval} - t_{gc}$. Note that to simplify the analytical expressions we assume that the completion delay is longer than the evaluation delay, which is generally true for fine-grain pipelines.

Using the above notation and assumption, the pipeline's analytical cycle time is:

$T_{LPSR2/2} = 2.\ t_{Eval} + 2.\ t_{gc}$

The per-stage latency of the pipeline is:

$L_{LPSR2/2} = t_{Eval}$

4.3 Lookahead Pipelines (Dual Rail)

Figure 4-3(b) shows the structure of one stage of the dual-rail LP3/1[2] pipeline [57]. In this pipeline, there are no matched delays. Instead, each stage has an additional logic unit, called a *completion detector,* to detect the completion of evaluation and precharge of that stage.

Unlike most existing approaches, such as Williams and Horowitz's pipelines [23], [59]each stage of the LP3/1 pipeline synchronizes with two subsequent stages, *i.e.,* not only with the next stage, but also its successor. Consequently, each stage has two control inputs. The first input, *Pc*, comes from the completion detector *(CD)* of the next stage, and the second control input, *Eval*, comes from the completion detector two stages ahead.

The benefit of this extra control input is to allow a significantly shorter cycle time. This *Eval* input allows the current stage to evaluate as soon as the subsequent stage has *started* precharging, instead of waiting until the subsequent stage has completed precharging.

The analytical cycle time can be expressed as:

$$T_{LP3/1} = 3.\ t_{Eval} + t_{CD} + t_{NAND}$$

The per-stage latency of the pipeline is:

$$L_{LP3/1} = t_{Eval}$$

4.4 High Capacity Pipelines (Single Rail)

Finally, the structure of one stage of the HC pipeline [58] is shown in Figure 4-3 (c). A key feature of this pipeline style is that is uses decoupled control of evaluation and precharge: separate Eval and Pc signals are generated by each stage's control. Precharge occurs when *Pc* is asserted and *Eval* is de-asserted. Evaluation occurs when *Pc* is de-asserted

and *Eval* is asserted. When both signals are de-asserted, the gate output is effectively isolated from the gate inputs; this is the isolate phase. To avoid short circuit, *Pc* and *Eval* are never simultaneously asserted.

An asymmetric C-element, *aC*, is used as a completion detector. The aC element output is fed through a matched delay, which (combined with the completion detector) matches the worst-case path through the function block.

Unlike most existing pipelines, the HC pipeline stage cycles through three phases. After it completes the evaluate phase, it enters the isolate phase (where both Eval and Pc are de-asserted) and subsequently the precharge phase, after which it re-enters the evaluate phase, completing the cycle.

Furthermore, unlike the other pipelines covered in this paper as well as the PS0 style in [59] the HC pipeline has only one explicit synchronization point between stages. Once the subsequent stage has completed its evaluate phase, it enables the current stage to perform its entire next cycle. The analytical cycle time can be expressed as:

$$T_{HC} = t_{Eval} + t_{Prech} + t_{aC} + t_{NAND3} + t_{INV}$$

The per-stage latency of the pipeline is:

$$L_{HC} = t_{Eval}$$

4.5 Designing Non-linear Pipeline Structures

The basic assumption in linear pipelines is that each pipeline stage has a single input and a single output channel. Non-linear pipelines stages, however, may have multiple input and output channels. This section presents an overview of the challenges involved in designing non-linear pipelines using timed templates. In particular we address issues with (i)

[2] As with the previous pipeline style, the 3/1 label characterizes the operation of a stage of the pipeline: 3 component delays in the

synchronization with multiple destinations (for forks), and (ii) synchronization with multiple sources (for joins). Subsequent sections provide our detailed solutions for each of the three pipeline styles reviewed above and then briefly describe how these solutions are extended to channels that are *conditionally read* or *written*.

4.5.1 Slow and Stalled Right Environments in Forks

Figure 2-5(b) shows an abstract two-way fork in which the forking stage S1 drives stages S2 and S3. For correct operation, S1 must receive (and recognize) acknowledgments from both S2 and S3. A problem is that S2 and S3, and the subsequent stages of each, may be operating largely independently of each other. One of these stages may get arbitrarily stalled, thus potentially stalling its acknowledgment from either S2 or S3.

If the pipeline templates designed for linear pipelines were naively extended to a datapath with a fork, by expecting S1 to synchronize on all of the acknowledges from the forked stages using a C-element to combine them, then the resulting pipeline may malfunction.

In particular, the acknowledgments generated in most linear pipeline structures are *non-persistent*. That is, after a stage asserts its acknowledgment, it assumes that the precharge of the previous stage is fast. Therefore, it does not explicitly check for the completion of that precharge before de-asserting the acknowledgment. We call this restriction/assumption the *fast precharge constraint*. In the case of a non-linear pipeline, however, if exactly one of S2 or S3 is slow or stalled, the acknowledgment signal of the fast stage may be de-asserted before S1 has a chance to precharge, causing deadlock. In other words, in this situation, S1 violates the fast precharging constraint. We call this problem the *slow* or *stalled right environment (SRE)*

---

evaluation phase and 1 component delay in the precharge phase, forming a complete cycle.

problem. In particular, Williams' classic PS0 pipelines [23] along with the recent lookahead and high-capacity pipelines all have this problem.

We propose two general solutions. The first solution is to modify only the immediate stages after a fork, such that, even after precharging, they maintain the assertion of their acknowledgment signal and are explicitly prevented from re-evaluating until after the forking stage is guaranteed to have precharged. The key is to modify the stages after a fork to guarantee their acknowledgments are properly received while still guaranteeing that these stages satisfy the fast precharge constraint.

The second solution is to modify every pipeline stage such that they maintain the assertion of their acknowledgment signal until after its predecessor stages are guaranteed to have precharged. In other words, this solution is to modify the entire pipeline to remove the fast precharge constraint, implicitly solving the SRE problem. This solution must be applied to all stages because an unmodified stage may otherwise assume its predecessors satisfy the fast precharge constraint, which may not be the case.

4.5.2 Slow and Stalled Left Environments in Joins

The second challenge is one of synchronization with multiple input channels, as needed in a join. Figure 2-5(a) shows a two-way join structure for an abstract pipeline where the data from each input stage, S1 and S2, must be consumed by the join stage S3. The data outputs of S1 and S2 are gathered together and presented to S3 as its inputs. Subsequently, S3 sends an acknowledgment to both S1 and S2 once it has consumed the input data. Thus, a two-way join represents a synchronization point between the outputs of two senders.

A problem can arise if the logic implementation of stage S3 is "eager", i.e. S3 may

produce output after consuming one but not both of its data inputs (see [59]). For example, if S3 contains a dual-rail OR function that evaluates eagerly (i.e., as soon as one high input bit arrives), then, after evaluation it will send an acknowledgment to both S1 and S2, even though one of them may not have produced data at all. As a result, if one of the input stages is particularly slow or stalled, it may receive an acknowledgment from S3 too soon. This can cause the insertion of a new unwanted data token at the output of the slow stage and thus corrupt the synchronization between the stages. We call this the *stalled left environment (SLE) problem*.

One solution is to allow join stages to have eager function blocks but still ensure that the generation of the acknowledge signal occurs only after consuming data from all of the input stages. This solution has been used extensively in quasi-delay insensitive templates [24].

4.6 Lookahead Pipelines (Single Rail)

Handling joins in single-rail lookahead pipelines is straightforward, and was initially proposed in [58]. The join stage receives multiple request inputs (*Lreq's*), all of which are merged together in the asymmetric C-element (aC) that generates the completion signal. In particular, each additional request is accommodated by adding an extra series transistor in the pull-down stack of the aC element. The aC will only acknowledge the input sources after all of the *Lreq's* are asserted and the stage evaluates.

To handle forks, on the other hand, a C-element must be added to the forking stage to combine the acknowledgments from its immediate successors. In addition, the other stages of the pipeline must also be modified to overcome the SRE problem of Section 4.5.1. As indicated, the problem is that the acknowledge signal from an immediate successor of a

fork stage can be regarded as a pulse, which may be de-asserted before its predecessor forking stage has precharged, causing deadlock. This section gives two distinct solutions for handling such forks in $LP_{SR}2/2$.

4.6.1 Solution 1 for $LP_{SR}2/2$

The first solution is to modify the immediate successor stages of forking stages to latch their *Lack* acknowledgment signals and delay their re-evaluation until after all predecessors have precharged. For $LP_{SR}2/2$, this is solution achieved by modifying *Lack* logic and the control of the foot transistor, as shown in Figure 4-4.

Assume the forked stage has just evaluated and the acknowledgment signal *Lack* signal has just been asserted. At this time, the right environment will assert *Rack* causing the output of the latch, *X*, to be asserted (*X=0, i.e., active low*), effectively latching the non-persistent acknowledgment signal. The *X* output is held low even when *Rack* is de-asserted. In particular, *X* is de-asserted (*X=1*) only after *Done* goes low caused by *Lreq* going low, implying that the input forking input stage has precharged. Effectively, the foot transistor now prevents re-evaluation until after X goes low, delaying re-evaluation until all inputs (including any slow input) are guaranteed to have precharged.

These modifications ensure that even late acknowledgments from a stage S3 immediately after a fork are guaranteed to be properly received while still ensuring that S3 satisfies the fast precharge constraint, thereby solving the SRE problem.

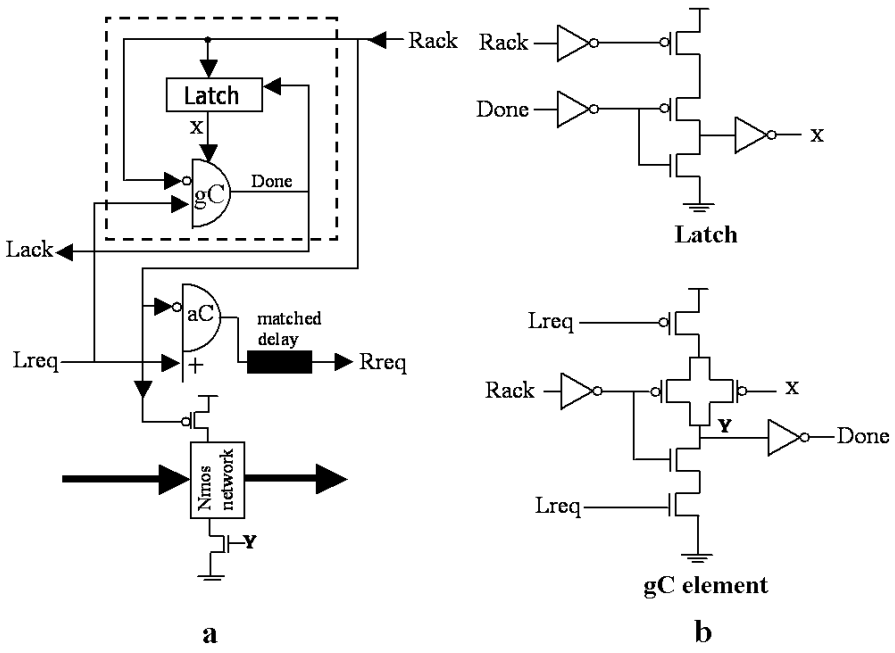


Figure 4-4: a) Modified first stage after the fork. b) Detailed implementation of the gates in the dotted box

The only new timing assumption that this template introduces compared to $LP_{SR}2/2$ is that the *Rack* pulse width must be long enough to properly latch it. This pulse width assumption, however, is looser than the original timing assumption that remains: the pulse width must be longer than the stage's precharge time.

4.6.2 Solution 2 for $LP_{SR}2/2$

The second solution is to modify each stage so that it does not de-assert its acknowledgments until after all input stages are guaranteed to have precharged. This solution can be implemented using the modified $LP_{SR}2/2$ template shown in Figure 4-5 in which the asymmetric C-element is converted to a symmetric C-element. As suggested earlier, this modification removes the fast precharge constraint, implicitly solving the SRE problem.

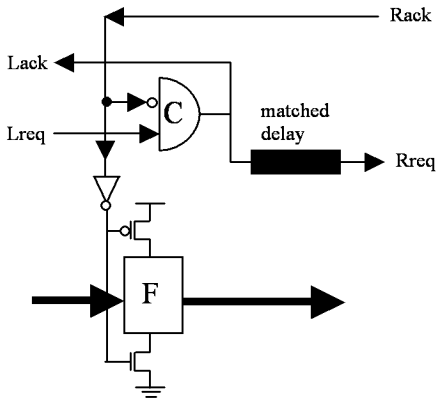


Figure 4-5: The $LP_{SR}2/2$ pipeline stage with a symmetric c-element

4.6.3 Pipeline Cycle Time

For the first solution, the cycle time expressions do not change if the additional acknowledgment signals simply increase stack height and do not add additional gates. For multi-way forks and joins, however, the cycle time will increase by the additional C-elements needed to combine them. For the second solution, the cycle time becomes:

$$T_{LPSR2/2} = max(2.\ t_{Eval} + 2.\ t_{gc},\ t_{Eval} + t_{prech} + 2.\ t_{go})$$

4.7 Lookahead Pipelines (Dual Rail)

This section extends a dual-rail lookahead pipeline, LP3/1, to handle forks and joins. Since both the stalled left environment (SLE) and the stalled right environment (SRE) problems of Section 4.5 can arise in dual-rail pipelines, detailed solutions are presented for both forks and joins.

4.7.1 Joins

Unlike $LP_{SR}2/2$, the LP3/1 pipeline has no explicit request line and thus may not function correctly unless it is modified to handle the SLE problem in joins. Our proposed solution still allows the use of eager function blocks; however it ensures that no

acknowledgment is generated from a stage until after all it's input stages have evaluated.

In particular, our solution is to add request signals to the input channels of the joins and feed them into the join stage's completion detector, as illustrated in Figure 4-6. The join's completion detector now delays asserting its acknowledgment until not only the function block is done computing, but also until after all the input stages have completed evaluation, thereby solving the left environment problem. Note that the additional request signals are taken from the outputs of the preceding stages' completion detectors. While this modification does not affect the latency of the pipeline, the analytical cycle time changes to:

$$T_{LP3/1} = 2.\ t_{Eval} + 2.\ t_{CD} + t_{NAND}$$

4.7.2 Forks

As in the single-rail lookahead pipeline, $LP_{SR}2/2$, we propose two solutions for the slow or stalled right environments. These solutions are similar in essence to the solutions for the single-rail case, but adapted to dual-rail.

The implementation of solution 1 is very similar to $LP_{SR}2/2$ as shown in Figure 4-7. First, the completion detector (CD) has been modified such that the acknowledgment signal is de-asserted only after the forking stage has precharged. In addition, we delay the re-evaluation of the function block until after the forking stage has precharged using a decoupled foot transistor controlled by the Y signal.

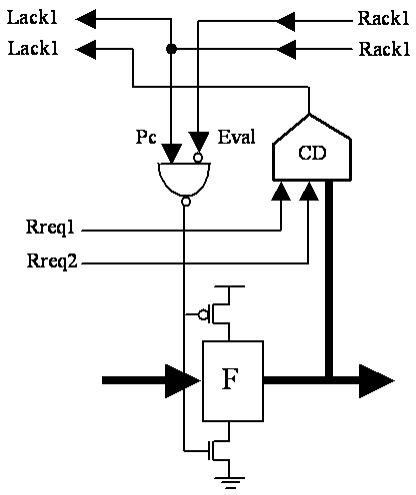


Figure 4-6: The LP3/1 pipeline with a modified CD to handle joins

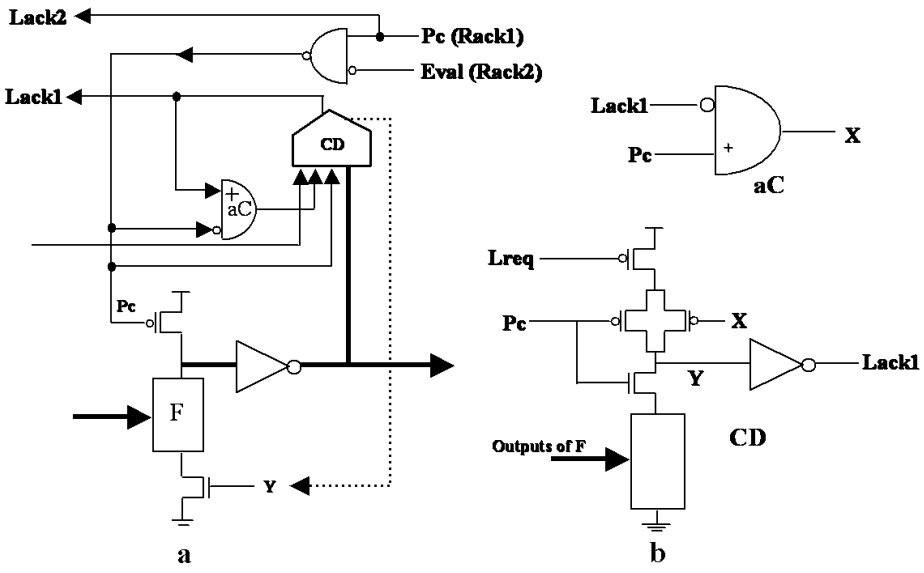


Figure 4-7: a) Modified first stage after the fork. b) Detailed implementation of the additional gates

The second solution is to add a request line to all LP3/1 channels and delay de-assertion of the acknowledgment (*Lack1* in this case) until after all immediate predecessors

have precharged, as shown in Figure 4-8. The request line is generated via a C-element that combines the incoming request line(s) and the output of the completion detection. The output of this C-element becomes the new *Lack1*. Because the C-element de-asserts its acknowledgment only after *Lreq* is de-asserted, the fast precharge constraint is removed, solving the SRE problem.

For solution 1, compared to the original LP3/1 template, the cycle time is slightly increased to:

$T_{LP3/1} = 2.\ t_{Eval} + \ 3.t_{CD} + \ t_{Prech}$

For solution 2, the cycle time increases to:

$T_{LP3/1} = \ t_{Eval} + \ 3.t_{CD} + \ t_{NAND}$

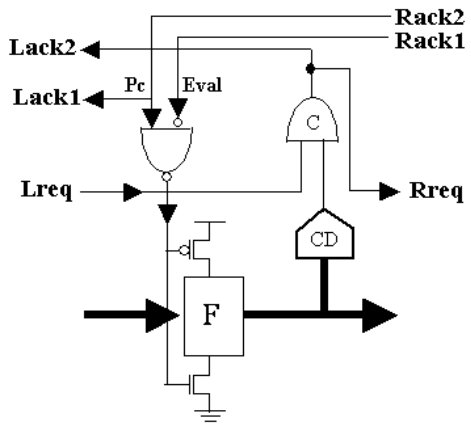


Figure 4-8: The LP3/1 stage with a C-element

4.8 High Capacity Pipelines (Single Rail)

Since the high capacity pipeline template uses single-rail encoding, it has a request line associated with the data and thus does not have the slow or stalled left environment problem in joins. However, because the acknowledgment signals in the high capacity pipelines are also non-persistent (effectively, timed pulses), they do have problems with a

slow or stalled right environment in forks.

The simple modification to the original stage controller of the high capacity pipeline illustrated in Figure 4-9 delays de-asserting the acknowledgment until after the request line goes low, thus removing the fast precharge constraint and solving the SRE problem using solution 2.

In particular, by replacing the *NAND3* gate by the state holding generalized C-element, the acknowledgment signal *Rack* only triggers the assertion of the precharge control signal, *Pc*. The de-assertion of *Pc* is caused by the input request signal *Rreq* going low. Thus, *Pc* remains asserted until after precharge is completed, and is unaffected by the acknowledge signal from the next stage getting de-asserted. Furthermore, the inverter is replaced by a *NOR2* gate with an additional input to delay the stage's re-evaluation until after the stale input data is reset.

In the new version of the HC pipeline stage the state variable, *ok2pc,* belongs to the channel between stage *N-1* and *N*. The reasoning is as follows. The function of the state variable is to keep track of whether stages *N-1* and *N* are computing the same token, or distinct (consecutive) tokens; precharge of *N-1* is inhibited if the tokens are different. If there are two stages, $N\text{-}1^{(A)}$ and $N\text{-}1^{(B)}$, supplying data for stage *N,* we propose to have two separate state variables, one to keep track of whether stages $N\text{-}1^{(A)}$ and *N* have the same token, and the second to keep track of whether stages $N\text{-}1^{(B)}$ and *N* have the same token. Similarly, if stage *N* had two *successors,* $N+1^{(A)}$ and $N+1^{(B)}$, we propose to have two distinct state variables, one each for the pair $(N, N+1^{(A)})$ and the pair $(N, N+1^{(B)})$.

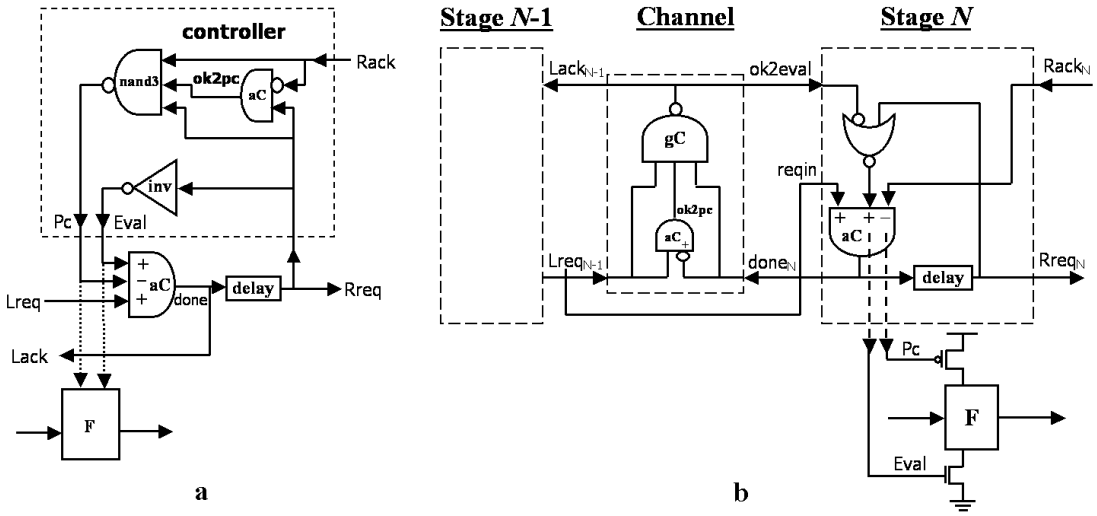


Figure 4-9: a) Original and b) New HC stage

The *aC* element, which implements the state variable *ok2pc*, is pulled out of the stage controller and placed in-between stages *N-1* and *N* (i.e., moved into the channel). In addition, the *gC* element is also moved into the channel to avoid extra wiring.

4.8.1 Handling Forks and Joins

Figure 4-10 shows the implementation of a template for stage, *N*, for the case where stage *N* is both a fork as well as join. The multiple *reqin's*, *ok2eval's* and *ack's* are handled by simple modifications to the linear pipeline of Figure 4-9(b), as shown in Figure 4-10.

Multiple *reqin's*: Each additional *reqin* is handled by adding a single series transistor to the aC element that makes up the completion generator, much like it was done for $LP_{SR}2/2$ in Section 4.6. Hence, *done* is generated only after all the input data streams have been received.

Multiple *ok2eval's*: Each additional *ok2eval* is handled by adding it as an extra input to the NOR gate that produces the *eval* signal. Consequently, the stage is enabled to evaluate (*eval* asserted) only after all of the *ok2eval* signals are asserted, i.e. after all of the senders

have precharged.

Multiple *ack's*: Multiple *ack's* are handled by *OR*'ing them together. Since the *ack's* are all asserted low, the *OR* gate output goes low only when all the *ack's* are asserted, thus ensuring that precharge occurs only after the stage's data outputs have been absorbed by all of the receivers. The *OR* gate is actually implemented as a *NAND* with bubbles (inverters) on the *ack* inputs. This *NAND* has an additional input --- the stage's completion signal --- whose purpose is to ensure that, once precharge is complete, *Pc* is quickly cut off. Otherwise, *Pc* may get de-asserted slightly after *Eval* is asserted, causing momentary short-circuit between supply and ground inside the dynamic gates.

4.8.2 Pipeline Cycle Time

If only joins are present, the cycle time is only slightly increased. Compared with the cycle time obtained in [58], the new cycle time equation has a NOR delay instead of an inverter delay, and a gC delay instead of a NAND3 delay:

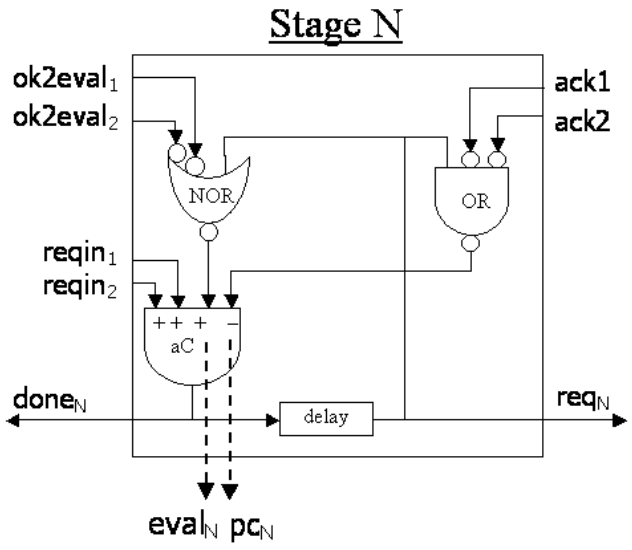


Figure 4-10: A 2-way join 2-way fork HC stage

$$T_{HC} = t_{Eval} + t_{Prech} + t_{aC} + t_{gC} + t_{NOR}$$

If forks are also present, then the cycle time increases by the delay of the OR gate which is needed to combine the multiple acknowledgments:

$$T_{HC} = t_{Eval} + t_{Prech} + t_{aC} + t_{gC} + t_{NOR} + t_{OR}$$

4.9 Conditionals

Other complex pipeline stages allow conditionally reading and writing data and can have internal state. This section briefly covers the implementation of these cells for the $LP_{SR}2/2$ template; however, a similar approach can also be applied to the other pipeline styles.

Figure 4-11(a) shows a conditional read, where the stage reads only one of the input channels depending on the value of the select channel. Only the channels read are acknowledged. Figure 4-11(b) shows a conditional write, where the stage reads the input channel and outputs the data (writes) to only one of the output channels depending on the value of the select channel. It receives an acknowledgment only from the output channel where the data is written. Note that the C-elements are only symmetric for the *Rack* input and asymmetric for all others.

Figure 4-12 shows a one-bit memory implemented using a $LP_{SR}2/2$ template. A and C represent the input and output channels. B is the internal storage. S is an input control channel that selects the write or read operation. When S0 is high, the memory stores the value at the input channel A to the internal storage B. Both the input A and the select channels are acknowledged. The implementation of how data is stored is shown in the dotted box (similar to [24]). Assuming that there is already data stored, one of the dual rail

bits of B is high and the other is low. When an input A is applied and S0 is high, first both rails are lowered and then one of them is asserted high, thereby storing the data. The C-element, which generates the acknowledgment of the input channel *LackA* through a matched delay line, is reset using its own output, since it doesn't receive an acknowledgment from an output. The delay of the delay line is matched to the delay of writing the internal node B.

When S1 is high, on the other hand, the internal data stored in B is sent to the output channel C. When an acknowledgment is received from the output channel C, the outputs are reset however the data stored remains unchanged.

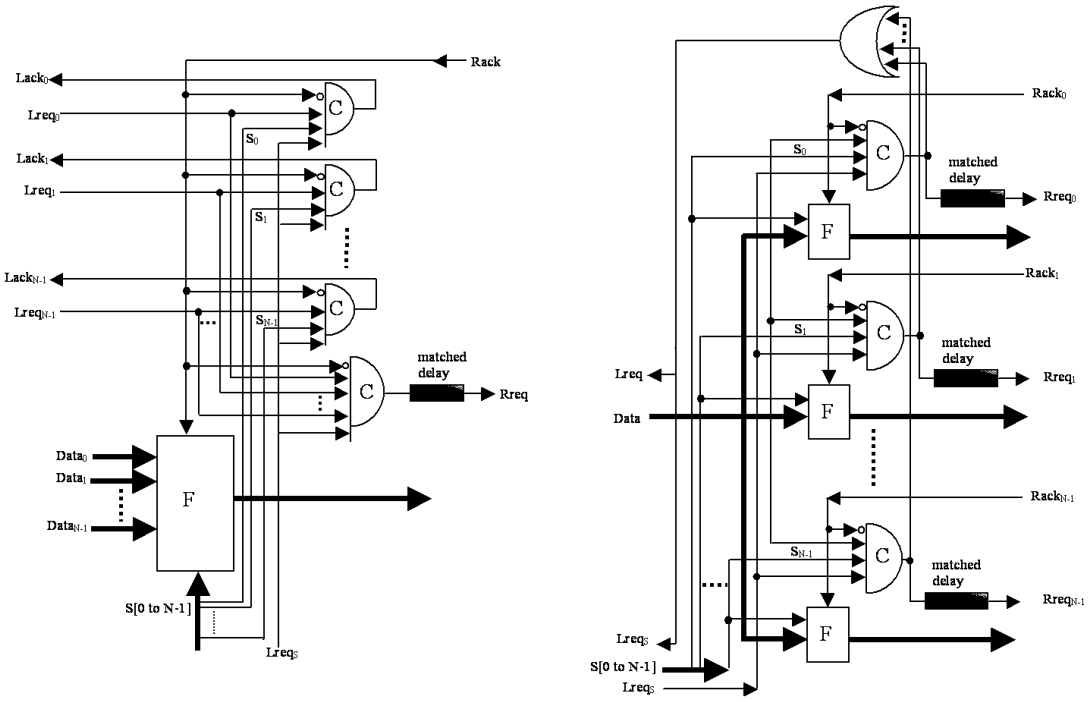


Figure 4-11: Conditional read and b) write.

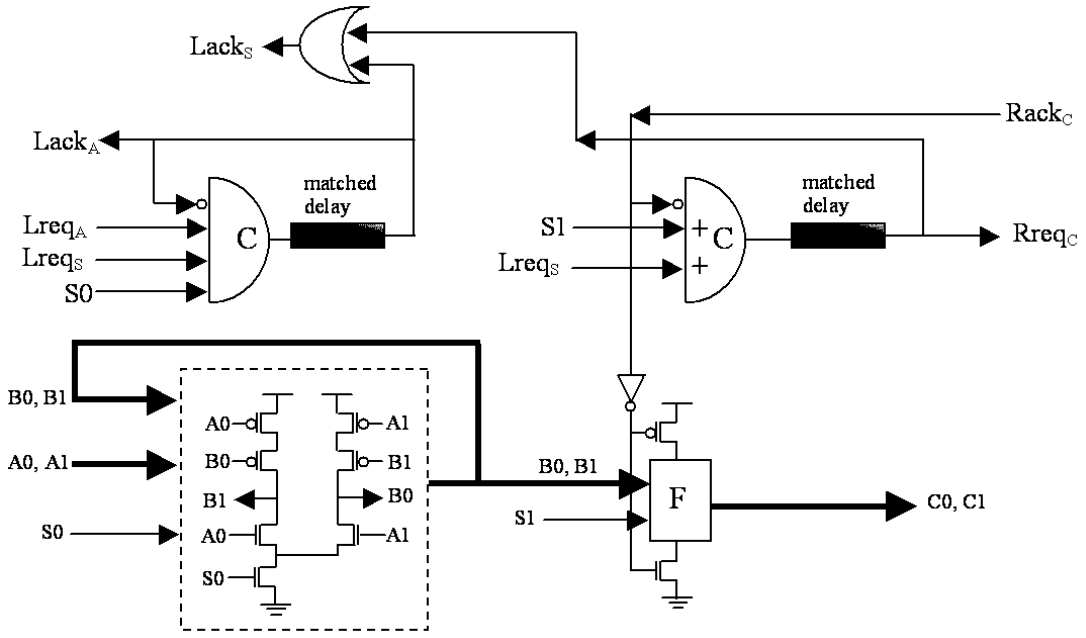


Figure 4-12:A one-bit LP$_{SR}$2/2 memory

4.10 Simulation Results

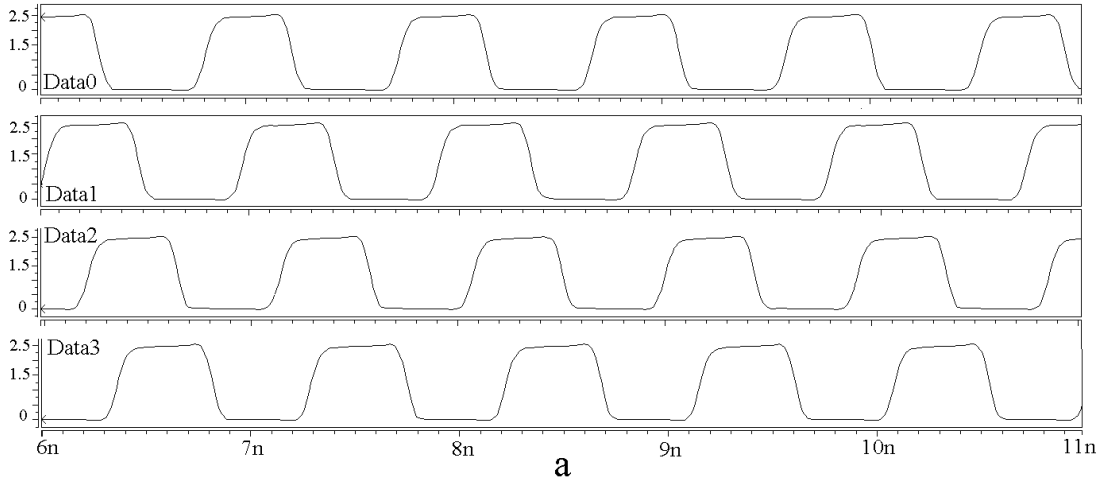
HSPICE simulations were performed using a 0.25 TSMC process with a 2.5V power supply at 25°C. The purpose of these simulations was only to quantify the performance overhead of using the fork-join structures of this paper, compared with linear pipelines. Hence, no attempt was made to fine-tune the transistor sizing to achieve optimum performance. In particular, all transistors were sized in order to roughly achieve a gate delay equal to a small inverter (Wnmos=0.8um, Wpmos=2um, and L=0.24um) driving a same-sized inverter. For the purposes of this comparison, wire delay also has been ignored.

The simulation results for all linear and non-linear pipelines discussed in this paper are presented in Table 4.1. The original linear pipelines appear under the **Sol1** columns and the **linear1** row because solution 1 involves only modifying the first stages after a fork and forks do not exist in linear pipelines. The **linear2** row and **Sol2** column has the cycle times

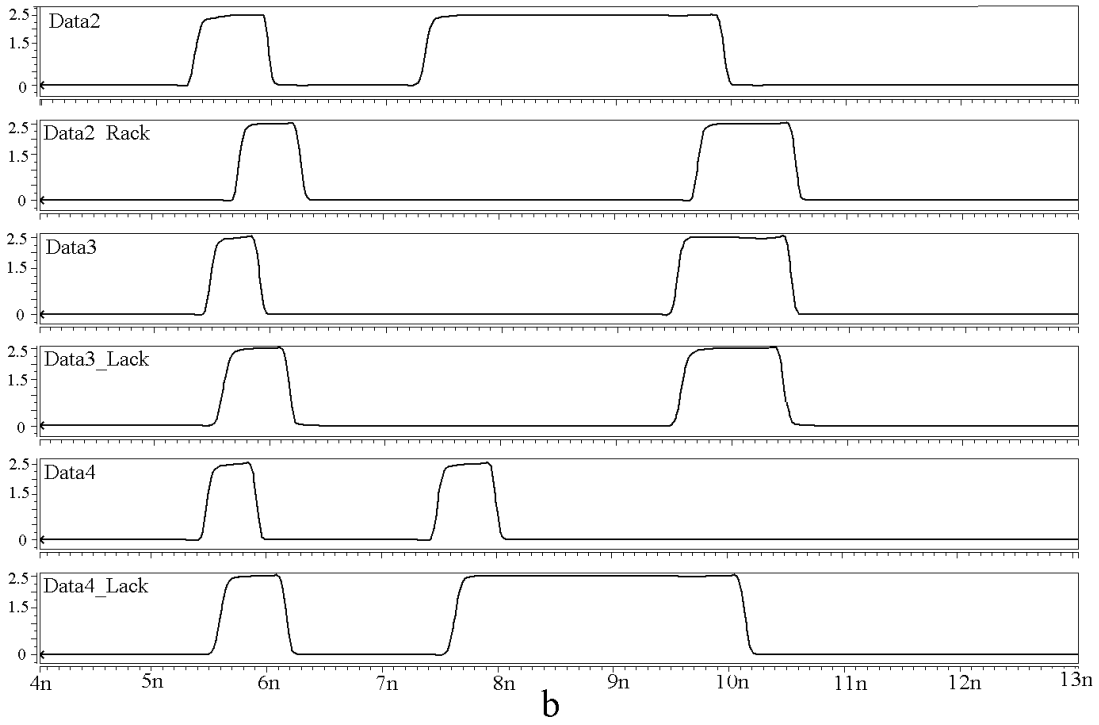
for linear pipelines, where each stage has been modified according to solution 2. Note that while the joins add only ~5% to the cycle time, the forks increase the cycle time by ~20% because of the additional C-element needed. The waveforms in Figure 4-13(a) show the data signal of a $LP_{SR}2/2$ one-bit linear pipeline. Note also that the cost of the more robust solution 2 compared to solution 1 is generally less than 5%. Figure 4-13(b) shows waveforms for a fork with a slow right environment channel called Data4 and Figure 4-13(c) shows a join with a slow left environment channel called DataB.

	$LP_{SR}2/2$		**LP3/1**		**HC**
	Sol1	*Sol2*	*Sol1*	*Sol2*	*Sol2*
Linear1	0.99	N/A	1.20	N/A	N/A
Linear2	N/A	1.06	N/A	1.28	0.93
Fork	1.23	1.29	1.41	1.45	1.20
Join	1.05	1.10	1.27	1.34	1.01

Table 4.1: Cycle time (ns) of original linear pipelines vs. proposed non-linear pipelines



a



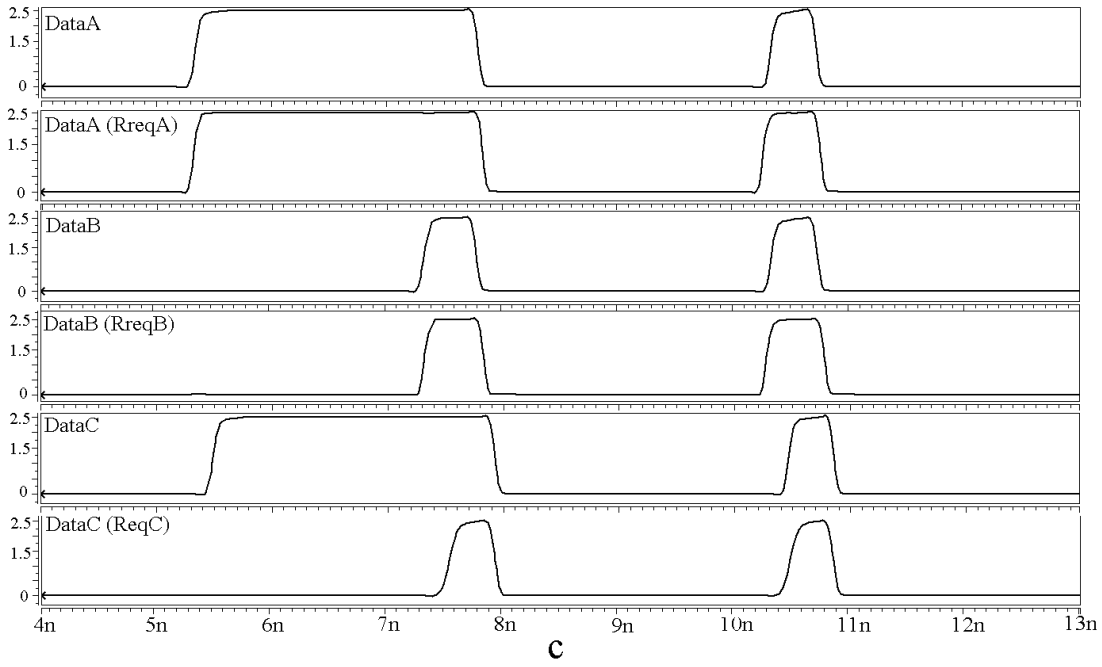


Figure 4-13: HSPICE Waveforms. a) Linear pipeline, b) Two-way fork and c) Two-way join

4.11 Conclusions

In this chapter we introduced new high-speed asynchronous circuit templates for non-linear dynamic pipelines, including forks, joins, and more complex configurations in which channels are conditionally read and/or written. Two sets of templates arise from adapting the $LP_{SR}2/2$ and LP3/1 pipelines and one set of templates arises from adapting the HC pipelines. Timing analysis and HSPICE simulation results demonstrate that forks and joins can be implemented with a ~5%–20% performance penalty over linear pipelines. All pipeline configurations have timing margins of at least two gate delays, making them a good compromise between speed and ease of design.

Chapter 5

5. A Design Example: The Fano Algorithm

In this chapter we present The Fano algorithm, a convolutional code decoder, and its efficient semi-custom synchronous implementation. The algorithm is used in communication systems to decode the symbols received over a noisy communication channel. Our goal is to later develop an efficient asynchronous counterpart, which we try to explore the challenges in designing asynchronous chips. In this chapter first we will present the Fano Algorithm. Then we will present the synchronous implementation of the algorithm.

5.1 The Fano Algorithm

5.1.1 Background on the Algorithm

The Fano algorithm [60] [61] [62], is a tree search algorithm that achieves good performance with a low average complexity at a sufficiently high signal-to-noise (SNR) ratio. A tree comprises nodes and branches, associated with each branch is a *branch metric* (or weight, or cost). A path is a sequence of nodes connected by branches with the path metric obtained as the sum of the corresponding branch metrics. An optimal tree-search algorithm determines the complete path (i.e., from the root to leaf) with minimum path metric, while a good (suboptimal) tree search algorithm finds a path with metric close to this minimum.

The Fano algorithm searches through the tree sequentially, always moving from one node to a neighboring node until a leaf node is reached. The Fano algorithm is a depth first tree-search algorithm [60], meaning that it attempts to search as few paths as possible

to obtain a good path. Thus, the metric of a path being considered is compared against a threshold T. The relation between T and the metric is determined by the statistics of the branch metrics (i.e., underlying model) and the results of partial path exploration. The latter is reflected by dynamically adjusting the threshold to minimize the number of paths explored.

The key steps of the algorithm involve deciding which way to move (i.e., forward, or deeper, into the tree or backward) and threshold adjustment. Intuitively, it moves forward only when the partial path to that node has a path weight that is greater than T. If no forward branches satisfy this threshold condition, the algorithm backtracks and searches for other partial paths that satisfy the threshold test. If all such partial paths are exhausted, it will *loosen* the threshold and continue. In addition if the current partial path metric is significantly above the threshold, it may tighten the threshold. Threshold tightening prevents always backtracking to the root node at the cost of potentially missing the optimal path. Moreover, a maximum *traceback depth limit* is often imposed to limit worst-case complexity. The details of the Fano algorithm are illustrated in the flow chart depicted in Figure 5-1 and a more detailed explanation can be found in [62] [61].

The decoding of a convolutional code with known channel parameters can be viewed as a tree-search problem with the optimal solution provided by the Viterbi algorithm [61], a breadth-first, fixed complexity algorithm. The Fano algorithm is known to perform near-optimal decoding of convolutional codes with significantly lower average complexity than the Viterbi algorithm.

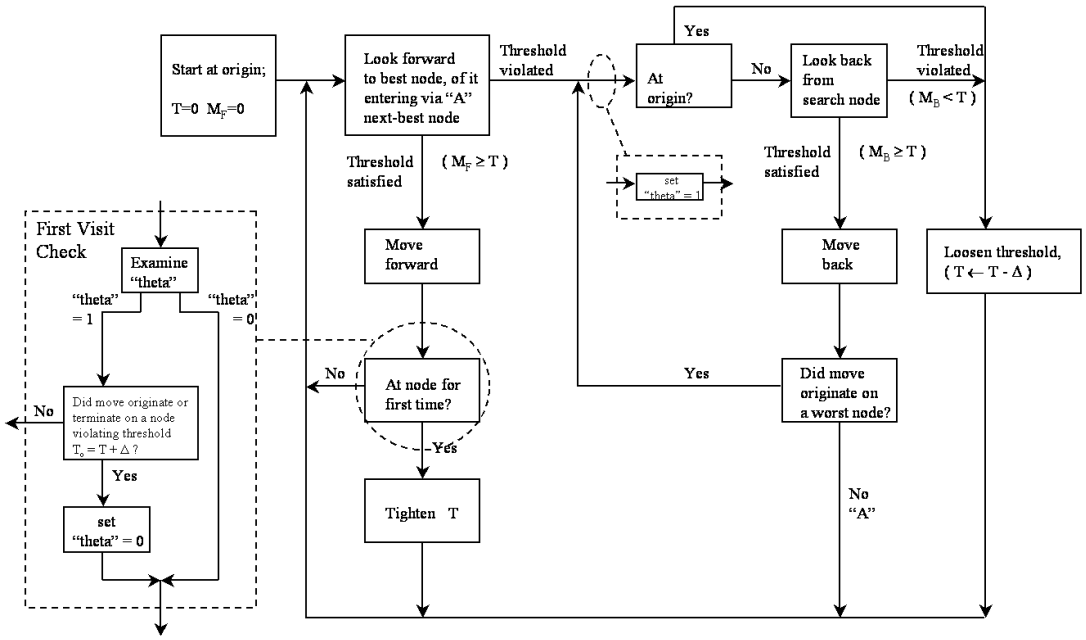


Figure 5-1: Flow-chart of Fano Algorithm

5.2 The Synchronous Design

This section describes the efficient normalization scheme used to optimize the algorithm, our architecture at the register transfer level, and statistics of the chip.

5.2.1 Normalization and its benefits

The basic idea behind normalization is to change the point of reference (e.g., from the origin of the tree to a current node under consideration). Normalization is often necessary to prevent hardware overflow/underflow. Interestingly, in traditional communication algorithms, such as the Viterbi algorithm, normalization often yields significant performance and area overhead that hardware designers generally avoid by using slightly larger bit-widths and modulo arithmetic [63]. In contrast, we show that using normalization in the Fano algorithm can yield a smaller, faster and more energy efficient design.

In particular, we normalize our variables in such a way as to make to current node's

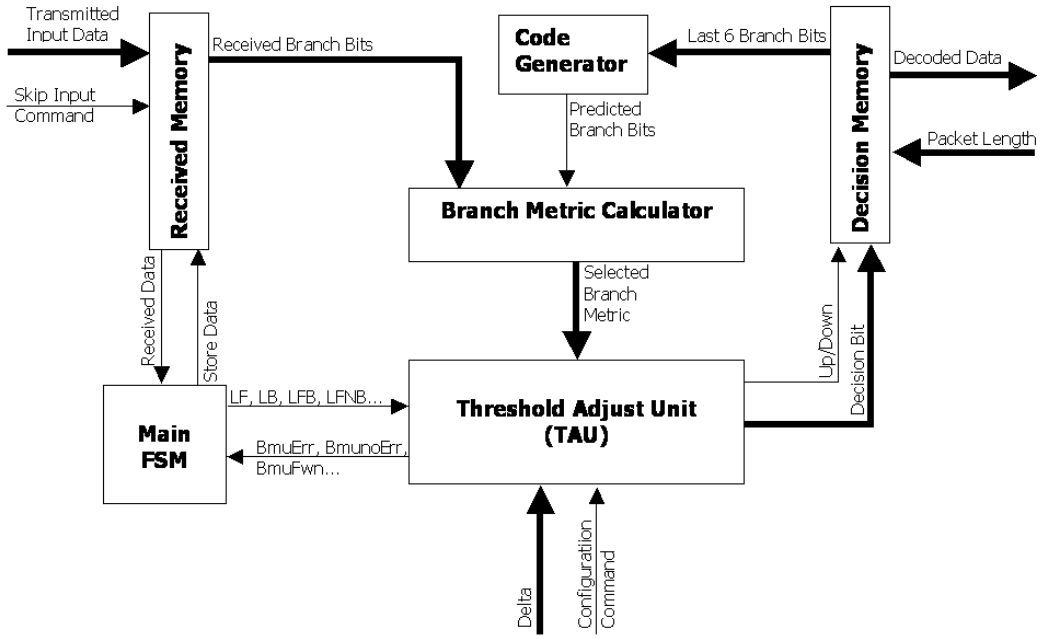
metric always equal to zero. This is equivalent to subtracting the current node's metric from every variable in the algorithm, which does not change the overall behavior of the algorithm. The advantages of this type of normalization in the Fano algorithm is as follows. 1) Additions involving the current metric (i.e., during the threshold check) are removed and comparisons with the current metric (i.e., during the first visit check and threshold tightening steps) reduce to a 1-bit sign check. 2) The normalization of the next threshold (subtracting the current node's metric from it) can be done by the ALU that compares the threshold with the next metric, and thus consumes negligible additional energy. 3) Lastly, the normalization enables us to work with numbers with smaller magnitudes that can be represented with fewer bits.

5.2.2 Register-Transfer Level Design

The register-transfer level architecture is illustrated in Figure 5-2. The Threshold Adjust Unit (TAU) is shown in more detail, but still with some of the details omitted to simplify the schematic. At each clock cycle, the best and next best branch metrics are both calculated using data that is stored in memory. (See [62] for more details regarding the branch metric computation.) The threshold check unit compares the error metric with the current threshold to determine if a forward move can be performed and simultaneously speculatively calculates two normalized next thresholds, the first assuming a forward move will be taken and the second assuming the threshold must be loosened (by subtracting Δ from T).

Based on the above results, either the move will be made and the pre-computed threshold will be stored or the threshold T will be loosened, all in one clock cycle. Additional clock cycles are needed to compute tightening the threshold if (i) a forward

move is made, (ii) the first visit check is passed, and (iii) the pre-computed tightened threshold is not in the range of Δ. Fortunately, with reasonable choices of Δ, computer simulations suggest that these additional cycles of tightening are rarely needed. Similar speculative execution allows us to perform a look/move back in one clock cycle.



General Synchronous Arcitecture

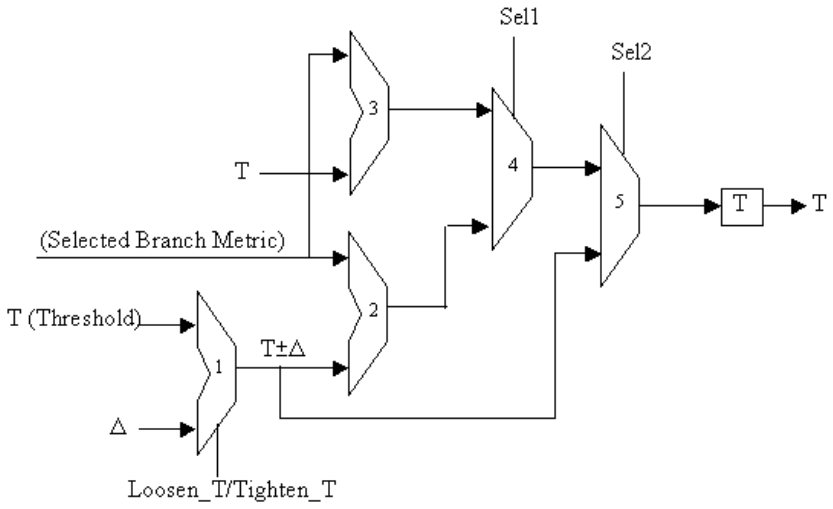


Figure 5-2: RTL architecture of the synchronous Fano Algorithm

The register-transfer-level architecture shown in Figure 5-2, is controlled by the finite state machine (FSM) illustrated in Figure 5-3. Three states, state 2-4, make up the main algorithm. In each of these states, the branch metric unit computes the needed selected branch metric using data that is stored in the sequence memory. Depending on control bits from the FSM (not shown) the selected branch metric that is associated with the best or worst branch. In either case, the corresponding input bit is sent to the decision memory where, in the case the branch is taken, it is used to update the selected path.

In state 2, the machine looks forward, moves forward if possible, and, if necessary, performs one step of threshold tightening. More specifically, after the selected branch metric is computed, the FSM performs a threshold check to see if the machine can move forward. That is, ALU3 computes T minus the selected branch metric and the FSM examines the most significant bit. If the sign bit is a 1, the branch metric is no smaller than T and the threshold check passes. Otherwise, the threshold check fails. Meanwhile, ALU1 and ALU2 speculatively compute T+Δ and T+Δ minus selected branch metric respectively.

These values, along with θ, a state variable shown in Figure 5-1, allow the FSM to determine whether the first visit check passes. That is, the first visit check passes if and only if $\theta = 0$ or if $T+\Delta$ is positive or $T+\Delta$ minus the selected branch metric is positive.

Based on the above results, the FSM acts in one of three ways. 1)The threshold check passes and a forward move is performed, but the first visit check fails so that the NextState is set state S2, in preparation of another look forward. 2) Both the threshold check and the first visit check pass in which case the FSM moves to state S3. 3) The threshold check fails and the FSM moves to state S4 in preparation of look/move backward. In the case of 1) the threshold register is updated with T minus the selected branch metric, computed by ALU3. In the case 2), on the other hand, the threshold is updated with the tighter threshold $T+\Delta$, computed by ALU1, whereas in the case of 3) the threshold register remains unchanged.

In state S3, the FSM checks whether a subsequent tightening is needed (by computing and checking the sign of $\Delta+T$). Simultaneously, it speculatively performs a

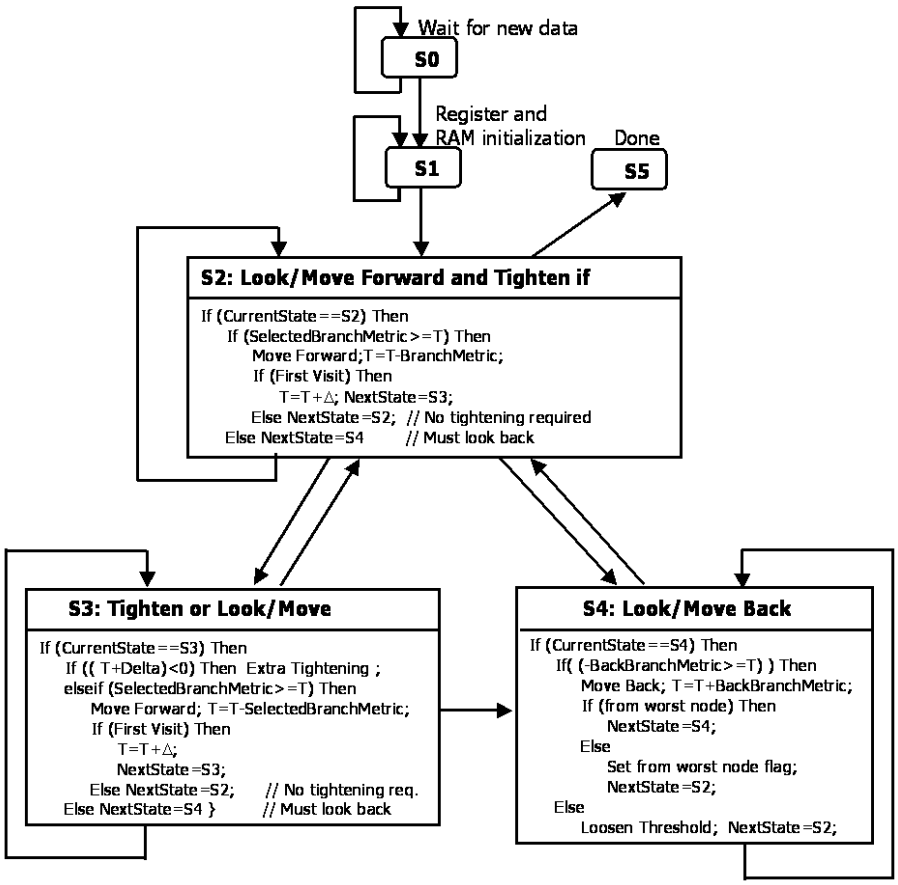


Figure 5-3: Finite State Machine describing the RTL

threshold check (by checking whether the Branch Metric is no smaller than T) which is needed in the event that the threshold need not be immediately tightened (i.e., in the event that tightening of the threshold requires only the one addition of Δ performed in state S2). If tightening is required, the NextState is set to state S3. For the case where no immediate tightening is needed, the FSM performs the same move/look forward/tightening/next-state operations as in state S2.

State S3 is entered when the threshold check fails in either state S2 ot state S3. In state S4, a look backward is performed and, if possible, a backward move is made and the threshold is updated with the re-normalized threshold. Both the look backward and re-

normalization are performed through ALU3 by adding T and the selected (backward) branch metric. Specifically, the look backward check is satisfied if and only if the negative selected branch metric is greater or equal to the threshold, i.e., the result of the ALU3 operation is negative and the re-normalized threshold is precisely the output of ALU3. If a backward move is performed and it is originated from a worst node, via an additional FSM flag, NextState is set to state S4, in preparation of another look backward. Alternatively, NextState is set to state S2 in preparation of a look forward to the next best node, controlled by a LookNextBest flag that is not shown to simplify exposition. If the backward look fails, on the other hand, the threshold is updated with a loosened threshold, speculatively computed by ALU1, and NextState is set to state S2.

The key feature of the speculative control strategy is that each forward move typically takes only once clock cycle with negligible performance overhead associated with the first visit check or tightening. In particular, with reasonable choices of Δ, computer simulations suggest that additional cycles of tightening are rarely needed.

5.2.3 Chip Implementation

The chip supports a packet length of N=128. The depth of the search tree, which also including 7 tail bits, is thus 135. It supports a rate ½ convolutional code, (i.e., n=2) with generator polynomials 1+D+D2+D5+D7 and 1+D3+D4+D5+d6+D7. For this prototype, we assumed the chip would have fixed branch metrics B(0)=2, B(1)=-7, and B(2)=-16, requiring 5 bits to represent. These metrics are ideal for the SNR range of $1<E_b/N_o(dB)<3$. In practice, they would be dynamically adjusted when the estimated channel SNR is outside this region, which may require an extra bit.

We used automatic placement and routing tools with a combination of synthesized and

manually laid-out components in the 0.5u HP14B CMOS process. The layout has an area of 1.2mm by 1.8mm. Powermill was used to estimate the performance of the design. At 1.5V power supply the design successfully operated at 15MHz and at 3.3V it successfully operated at 100MHz.

Chapter 6

6. The Asynchronous Fano

Deeper analysis of the Fano algorithm shows that the operation of the algorithm can be divided into two: The *Error Free Region* and the *Error Region*. In the Error Free Region, the algorithm moves forward while the received bits from the sender are error free and match the expected bits. In this region of operation the un-normalized threshold is incremented with a constant value, namely the value given for an error free branch of the tree. If the threshold value is known at the time the algorithm enters the Error Free Region then the next value of the threshold can be calculated. The normalized threshold, however, stays in the range of $-\Delta \leq T \leq 0$ and rotates through a finite number of values in a pre-determined order.

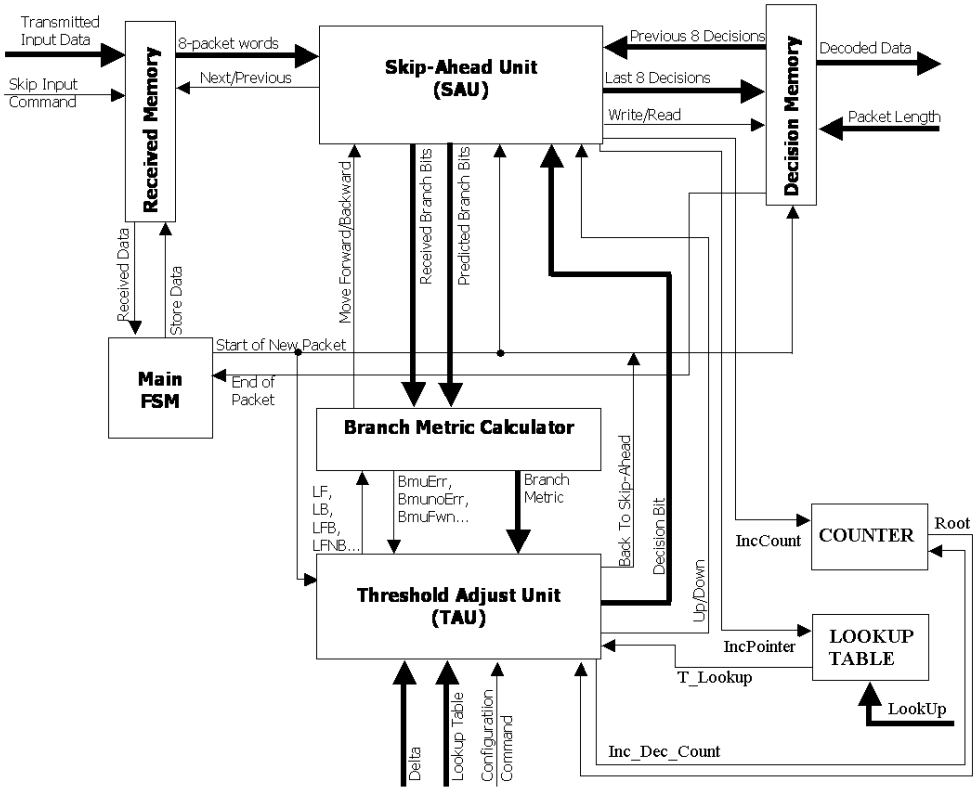
Consequently, instead of calculating the threshold values explicitly, a pointer to a lookup table containing these pre-determined values is incremented. When an error is encountered, the design enters the error region where the current value of the threshold is accessed from the lookup table and the full algorithm is applied in order to determine whether to move forward, move backward, or loosen the threshold. The algorithm stays in the error region until a node in the search tree is reached for the first time and the move was a forward move, at which point the algorithm moves back into the error-free region. The algorithm continues until the end of the tree by alternating between the error free and the error regions.

For high SNR applications most of the received packets have little to no errors therefore most of the decoding process consists of reading the data from the memory, comparing to the predicted data, and the writing the decision to the memory and involves little to no multi-bit additions/subtractions/comparisons due to loosening or tightening the threshold. This fact motivates a two-block architecture that are specifically designed to handle the two different operating regions of the algorithm efficiently.

6.1 The Asynchronous Fano Architecture

The proposed asynchronous architecture, shown in Figure 6-1 localizes the Error-Free region in a small block that is highly optimized. In particular, the Branch Metric Unit (BMU) is partitioned into a Skip Ahead Unit optimised for the Error Free Region and a Threshold Adjust Unit and the Branch Metric Calculator that are active only in the Error Region and have implementations analogous to the synchronous version.

The data received to the decoder via the Transmitted Input Data channel are stored in the Received Memory. The fast Skip Ahead Unit requests data from the Received Memory in 8 word chunks via the Previous/Next channel, where each data word is for the (7,1,2) code two bits wide. As the Skip Ahead Unit decodes the code and moves forward in the tree, it locally stores its decisions. Every 8 decision is sent to the Decision Memory via the Last 8 Decisions channel. When an error is encountered, the Skip Ahead Unit may need to go back in the tree to explore different branches by requesting previous decisions from the Decision Memory that arrive on the Previous 8 Decisions channel. The data flow between the Decision Memory and the Skip Ahead Unit is controlled via the Write/Read channel.



General Asynchronous Architecture

Figure 6-1: RTL architecture of the asynchronous implementation

In the Error Free Region, the received bits are read from the Received Memory and decoded in the Skip Ahead Unit. The resulting decisions are then sent to the Decision Memory and the SAU unit increments the look-up table pointer via the IncPointer channel. In this region, the Main FSM, Branch Metric Calculator, and the TAU are inactive.

When an error is encountered the SAU informs the Branch Metric Calculator via the Error channel and also sends it the received branch bits and the predicted branch bits calculated using the previous decisions and the convolutional code. Depending on the move commanded by the Threshold Adjust Unit via the LFB (look forward best), LB (look backward), LFNB (look forward next best), and LFBTE (look forward best until error) channels, the Branch Metric Calculator calculates and compares the branches, selects the

appropriate one, and sends it to the TAU with additional information notifying if the move originated from a worse branch and if the branch had any errors (via the additional BmuErr and BmuFwn channels). Every time the TAU is accessed for the first time when an error has occurred, the TAU reads the normalized threshold from the look-up table and updates the threshold value. The TAU is implemented analogously to the synchronous version and is responsible for deciding to move forward, move backward, or adjust the T threshold. Upon deciding a move, the relevant information is sent to the SAU and a new command is issued to the Branch Metric Calculator. Finally when a new error free node is reached for the first time, the TAU issues the LFBTE command, stores the normalized threshold, updates the pointer to the look-up table and resume operation in the fast SAU via the Back To Skip-Ahead channel. The operation switches back and forth between the SAU and the TAU until all the data is encoded. Upon reaching the end of the tree, the Decision Memory sends out the decoded data.

The fact that the asynchronous circuit has no global clock allows the asynchronous architecture to be naturally divided into two blocks, each operating at its ideal speed that communicate only when and where needed via the inter-block asynchronous channels.

6.2 The Skip-Ahead Unit

A high level implementation of the SAU is shown in Figure 6-2. The core of the SAU is the Error Detector, which compares the predicted branch bits with the received branch bits and stores the decision. To operate at full rate, the memories must keep up with writing/reading one data word per decoding cycle. As the memory capacity increases, this becomes a difficult task and for this reason we have opted to use shift registers that act as caches for the bigger memories. In particular, the Fast Data Register stores 8 words from

the Received Memory and the Fast Decision Register acts as an 8-word read/write cache for the Decision Memory. When the Received Memory sends an 8-word packet to the Fast Data Register, the Received Memory speculates that the SAU will not encounter any errors and moves forward thus prepares to send a new set of data. This cache structure allows the larger memory to run at 1/8 the speed of the SAU. The same motivation applies to the use of the Fast Decision Register with the exception that it is a read/write register. Both of the registers have an associated controller to request and send data to their respective memories.

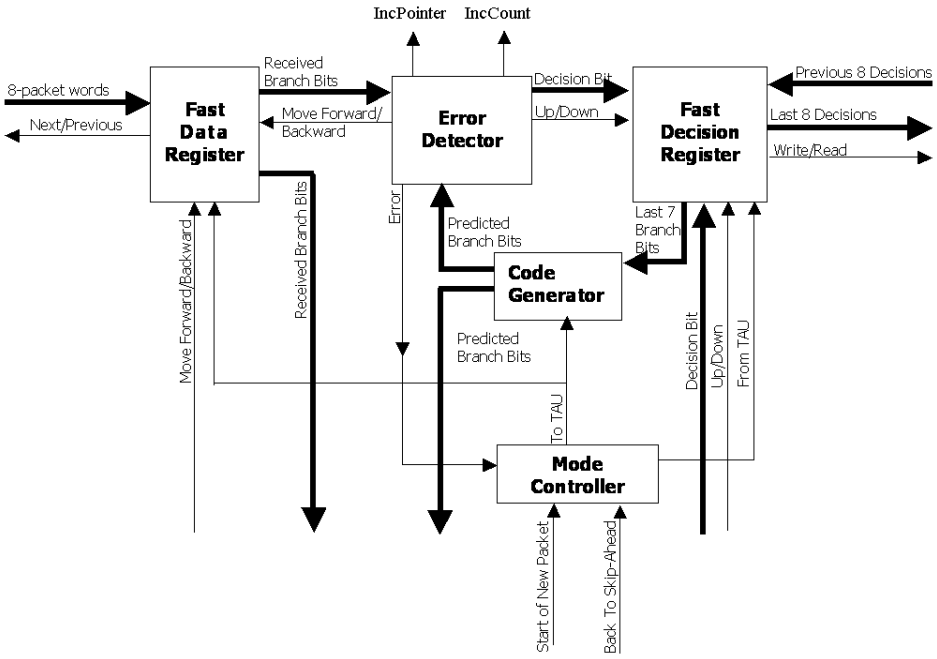


Figure 6-2: Detailed implementation of the Skip-Ahead Unit

The most recent decisions in the search tree, which always reside in the Fast Decision Register, are sent to the Code Generator, which predicts the values of the new branch bits. The predicted branch bits are compared to the received one in the Error Detector. If there

is a match, indicating that there is no error, the decision is stored in the Fast Decision Register, an internal counter and the pointer in the look-up table are incremented, and new data are requested from the shift registers via the Move Forward/Backward, Up/Down, IncCount and IncPointer channels. If there is no match, then an error is encountered. The predicted and received branch bits are sent to the Branch Metric Calculator and the controls of the shift registers are transferred to the TAU.

The critical loop in the Error Free Region consists of the Fast Shift Register, the Error Detector, the Fast Decision Register, and the Code Generator. For high SNR operation, most of the time the decoder operates in the Error Free Region, therefore our goal is to achieve high speed in this region by optimizing the circuit. However if and when the circuit encounters an error, it enters the Error Region and the critical path consists of the Fast Shift Register and the Convolutional Code Generator serving data to the Slow BMU. In the Error Region, the operation is the same as in the synchronous version consisting of a number of sequential operations. In this region the speed is expected to be comparable to the synchronous case.

6.3 The Memory Design

Since the chip supports a packet length of only 135 bits (128 data and 7 tail bits), we have opted to design the main data memory blocks of the Received and Decision memories using standard PCHB templates. However, we introduced unacknowledged tri-state buffers on the data bus to efficiently allow multiple drivers of the bus. This is typical in synchronous design, but does introduce some minor timing assumptions not typical of PCHB-based designs. We also used standard place and route tools for the physical design

of the memories for faster design time at the expense of more area and power consumption.

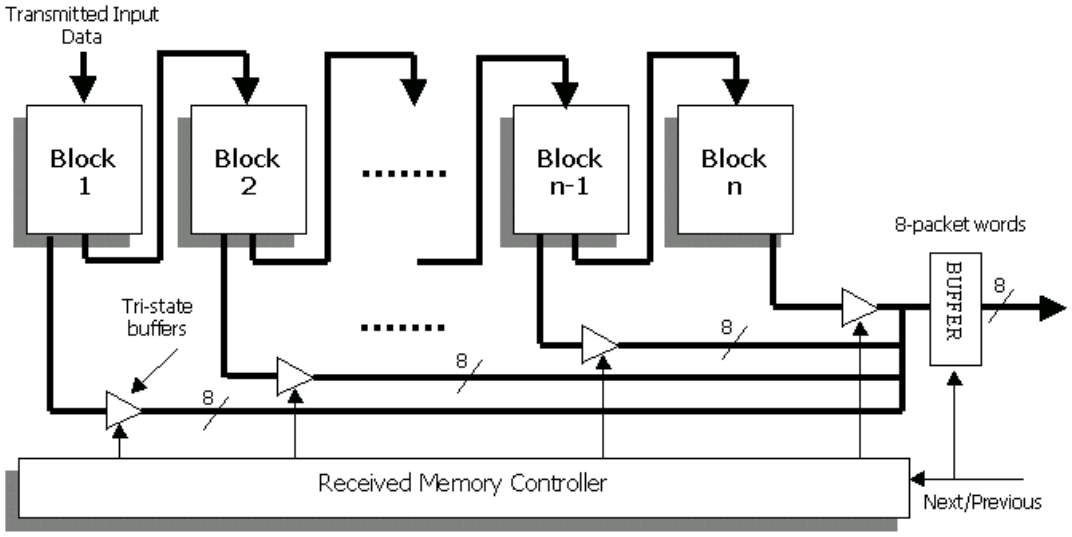


Figure 6-3 Implementation of the Received Memory

In particular, as depicted in Figure 6-3 the received memory consists of n blocks where each block can hold 8 words. For the (7,1,2) convolutional code each word is 2 bits. The blocks are FIFO's implemented with PCHB's. At any time only one of the tri-state buffers is enabled allowing only one of the blocks to send their data. The Fano algorithm is a sequential tree search algorithm, therefore SAU accesses the memory sequentially via the Next/Previous channel. The Received Memory Controller responds to the request by enabling a preceding or proceeding tri-state buffer and sending new data. The buffer captures the new data and sends it to the requesting unit. The timing assumption for correct operation is that the delay from the Next/Previous channel through the Received Memory Controller and the selected tri-state buffer should be less than the delay from the

Next/Previous channel to the output buffer. Moreover, the output buffer should only latch its input when the enabled tri-state buffers outputs have changed and stabilized.

The decision memory has a similar structure, however since it is a read/write memory each of the blocks can be accessed individually to read from or to write to.

6.4 The Fast Data and Decision Registers

The fast data register is implemented using two 8-word, 1-bit shift registers, as shown in Figure 6-4. The register consists of 8 conditional input, conditional output 1-bit memory pipeline stages. Depending on the command, cmd, it either shifts forward by receiving new data from InF and sending the old to OutF, shifts backward by receiving data from InB and sending the old to OutB, or loads 8-words in parallel from the main memory. The parallel load command overwrites the old data tokens inside each stage. The command channel Cmd should go to all of the stages, however to prevent the use of a big c-tree to generate the Cmd acknowledgement signal the Cmd signal is broadcasted with a tree of copy buffers. Altough this solution reduces the load on the Cmd channel if it were to be copied to all stages directly, this solution increases critical loop delay of the algorithm.

The fast data register is implemented similarly.

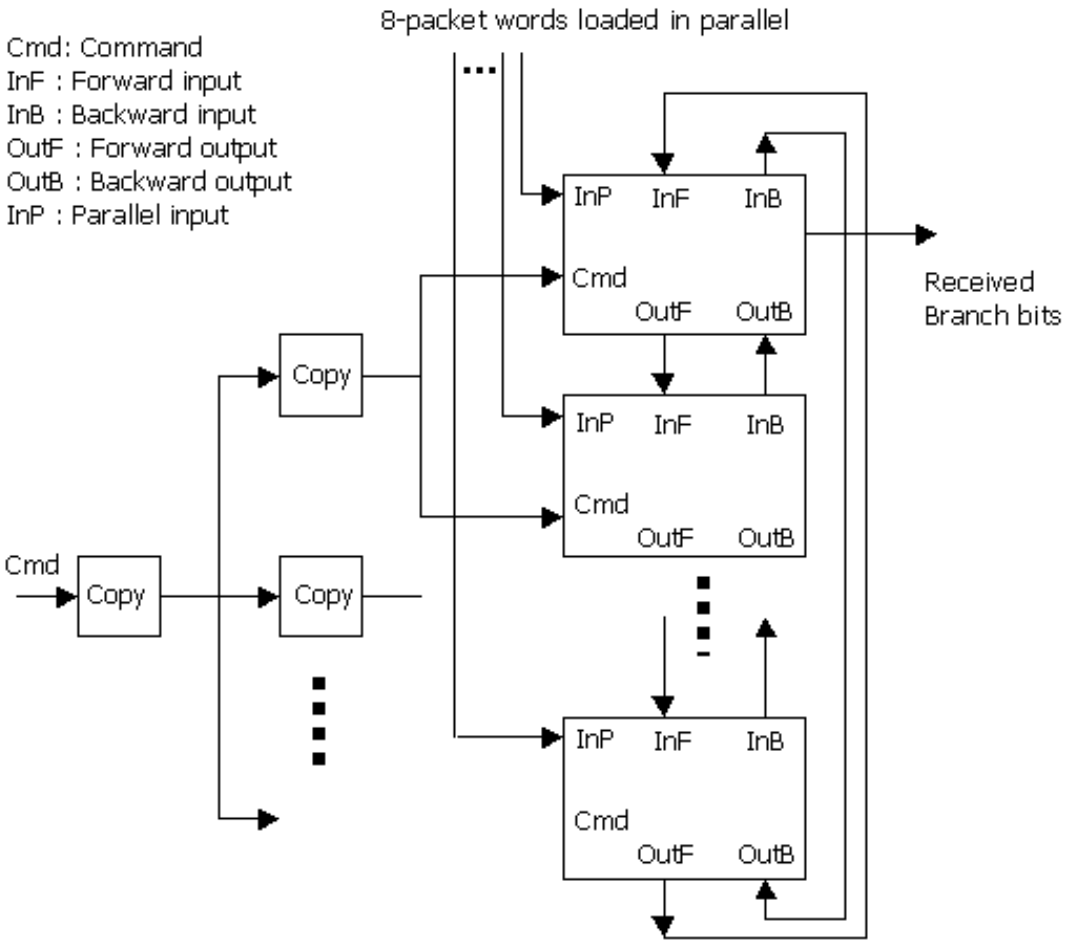


Figure 6-4 Implementation of a 1-bit fast shift register

6.5 Simulation Results and Comparison

The core layout of the chip designed in TSMC 0.25μ CMOS technology is illustrated in Figure 6-5. Nanosim simulations, on the extracted layout, show that the circuit runs at 450MHz and consumes 32mW at 25°C and has an area of 2600μm x 2600μm = 6.76mm^2. The asynchronous chip runs about 2.15 faster than its synchronous counterpart. However it occupies 5X the area. This is partially due to the fact that both of the memories which

occupy half the chip area in the asynchronous chip are implemented with PCHB's. Lastly the design consumes 1/3 the power of its synchronous counterpart.

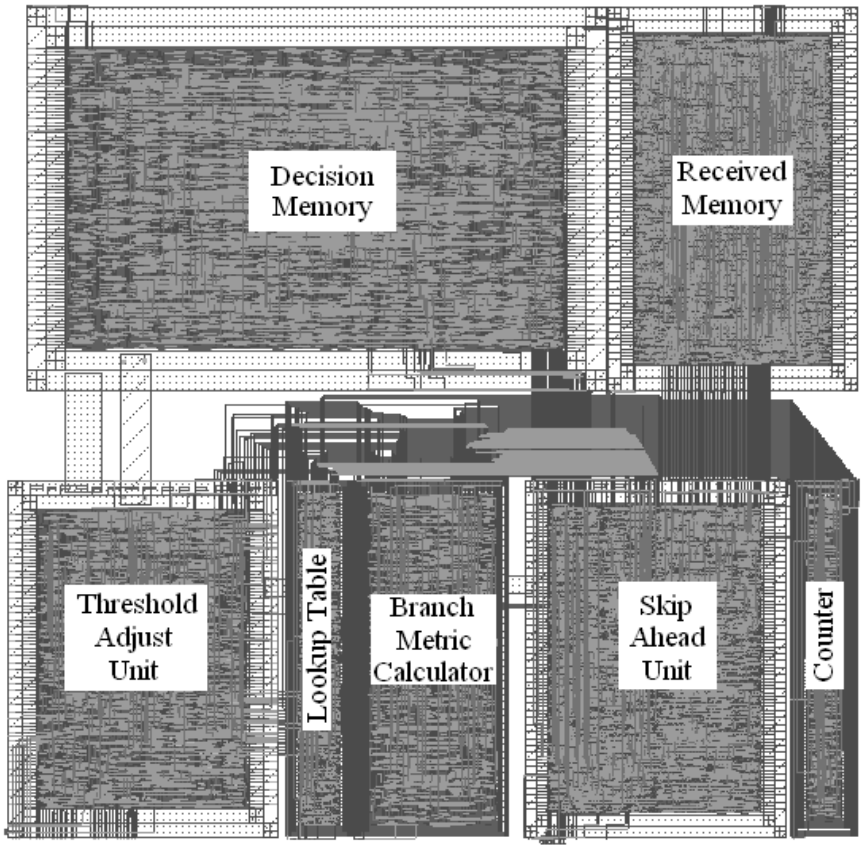
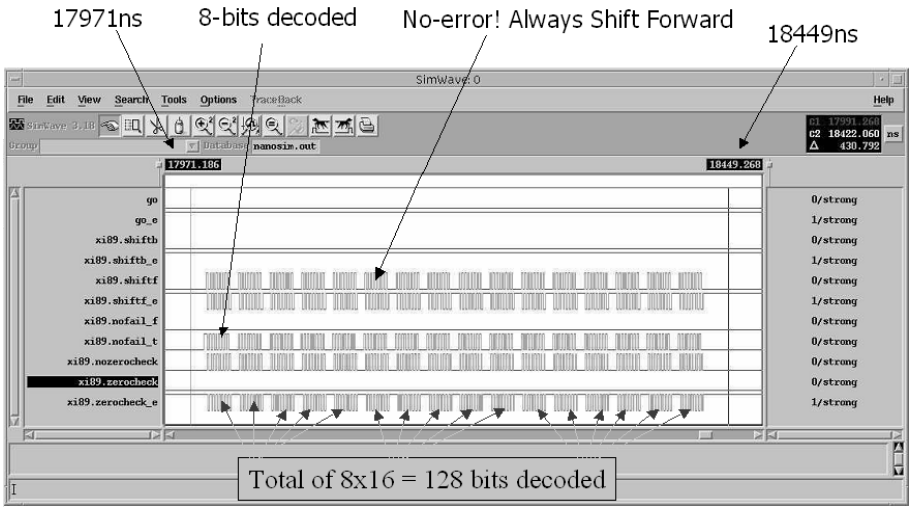


Figure 6-5: Layout of the asynchronous Fano

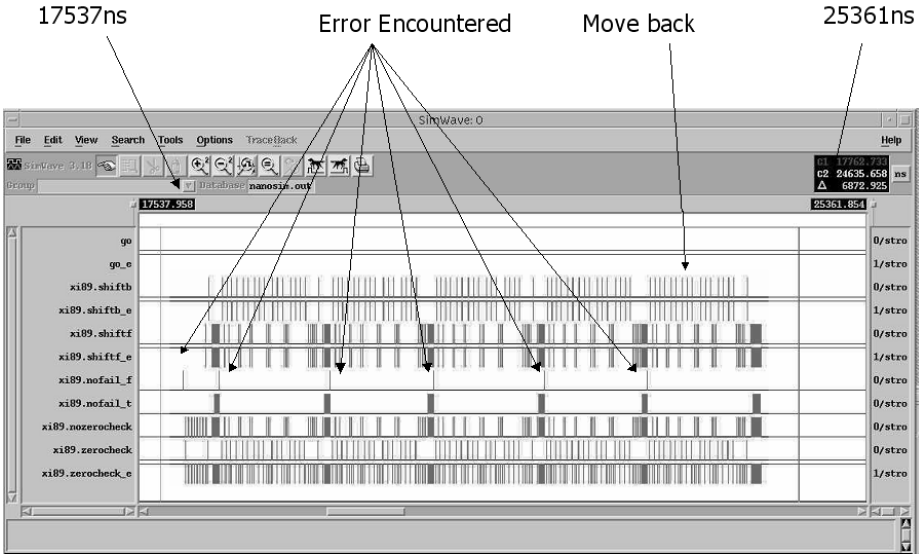
Figure 6-6 (a) below shows the post-layout simulation results for the circuit operating under the Error Free Region. Since the Fast Data and Decision Registers can only hold 8 words, once the data held by the Fast Data Register is consumed a new set of data is requested from the main Received Memory. This request and data transfer causes a slight delay, which can be observed in the waveforms as a slight gap every 8 pulses. Since there are no errors in the Error Free Region the *nofail_f* signal used to indicate the encounter of

an error is never asserted but instead the *nofail_t* signal, which indicates that there are no errors is asserted by the detection logic.

On the other hand, as shown in Figure 6-6 (b) in the Error Region, as errors are encountered the decoder moves back and forth to find the correct path. This can be observed with the assertion of the *shiftb* (shiftback) and *nofail_f* signals.



(a)



(b)

Figure 6-6: a) Error-Free and b) Error Region operation waveforms

Chapter 7

7. An Asynchronous Semi-Custom Physical Design Flow

The general design flow that the USC Asynchronous Design Group has refined was already covered in the introduction of this thesis. In this chapter we will specifically focus on the last parts of the flow mainly the gate level and physical design.

7.1 Physical Design Flow Using Standard CAD Tools

One of the biggest obstacles today of designing asynchronous circuits is the lack of CAD tools specifically targeted for the design of such chips. However it is still possible to complete a fairly complex chip in a reasonable amount of time using standard CAD tools used for synchronous design. Figure 7-1 below illustrates the flow.

There is no difference for the initial specification step of the design for synchronous or asynchronous design, since a spec typically describes the expected functionality (Boolean operations) of the designed block, as well as the delay times, the silicon area and other properties such as power dissipation. Usually, the design specifications allow considerable freedom to the circuit designer on issues concerning the choice of a specific circuit topology, individual placement of the devices, the locations of input and output pins, and the overall aspect ratio (width-to-height ratio) of the final design. The actual implementation of the asynchronous circuit starts at the schematic level. The top-level circuit or design is hierarchically decomposed until the design consists of a netlist of leaf

cells. If a leaf cell library exists then the automatic place and route tool can generate the layout using this library. Otherwise the leaf cells can be further decomposed into gates where the gate level netlist can be mapped to a gate library.

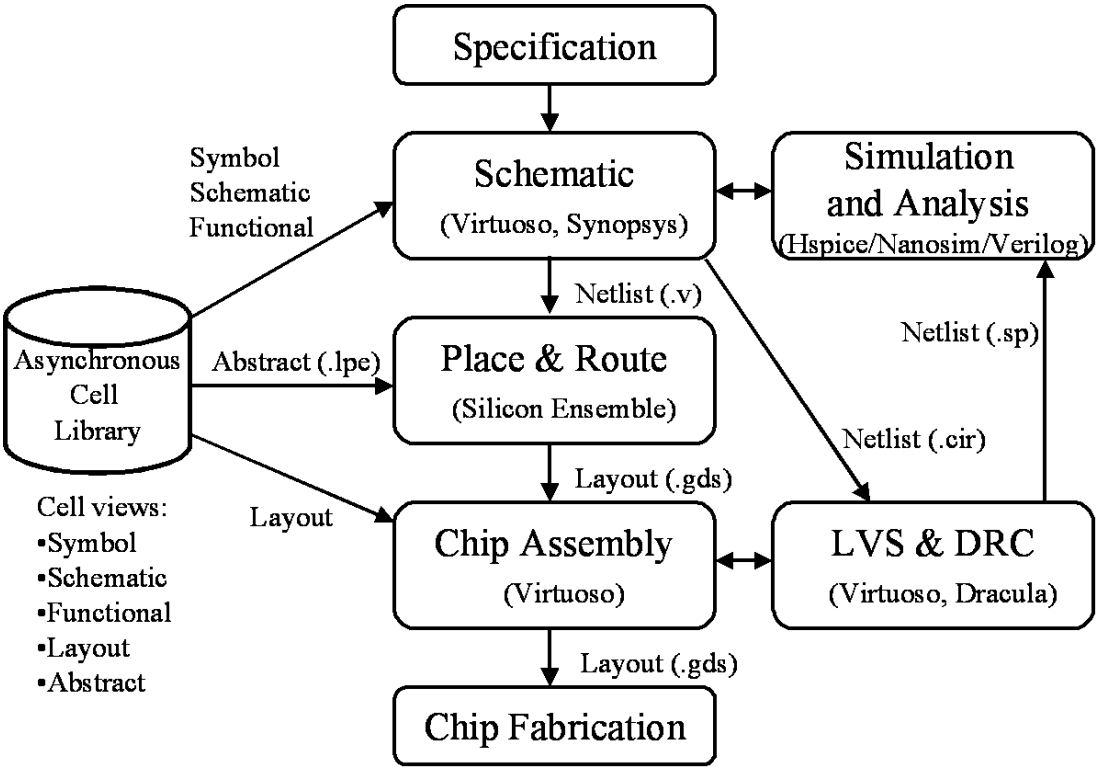


Figure 7-1: Physical design flow using standard CAD tools

Depending on the final design size either the whole design can automatically be placed and routed using the P&R tool or the design can be partitioned into smaller blocks and each block can be placed and routed separately. This allows for better control over the layout for performance. Once the whole design is laid-out and Design Rule Check (DRC) is completed a Layout-vs.-Schematic (LVS) must be performed to ensure that the layout is the same as the schematic. This step is followed by extraction of the layout for post-layout spice simulation. We have used the Dracula tool from Cadence for this step. The extracted

netlist accurately represents the laid-out transistor dimensions as well as the wiring resistance and capacitance. Depending on the post-layout simulation to achieve the desired performance and power requirements the top-level design might have to be changed and the whole step repeated.

The architectural and leaf cell design steps of the physical design flow followed in this thesis are illustrated below in Figure 7-2.

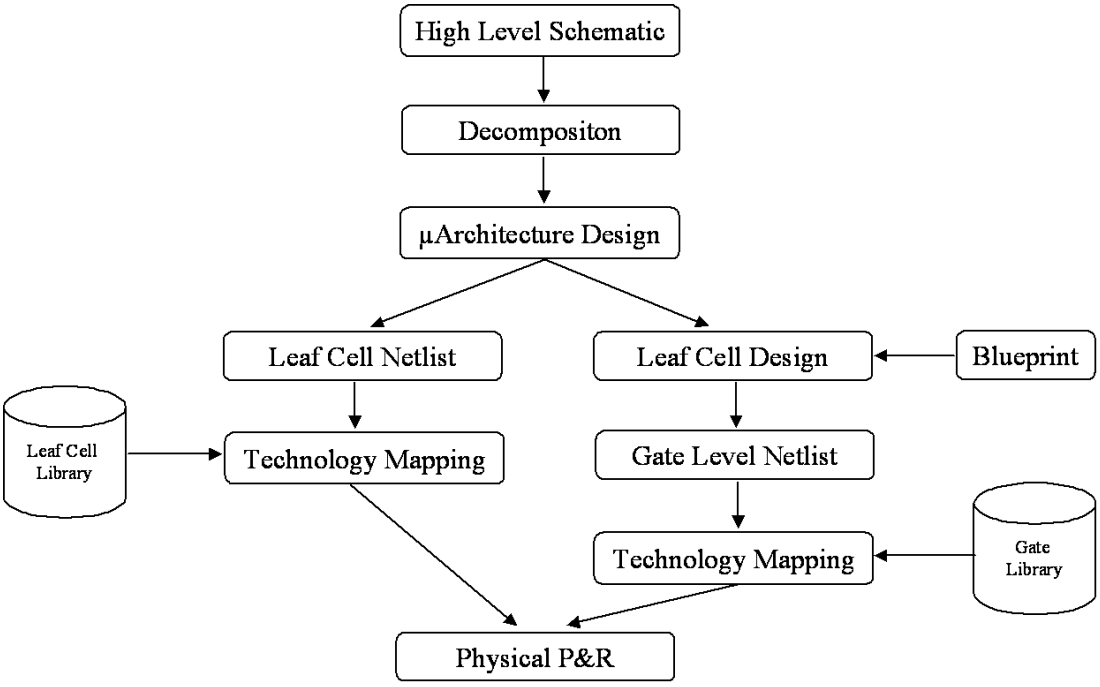


Figure 7-2: Asynchronous circuit design flow followed

The high level schematic is developed in C and Verilog codes and used to describe the specification of design. The high level schematic is hierarchically implemented by decomposing the design to the lowest level communicating blocks, namely the PCHB leaf cells. In the micro-architecture step the designer can choose to implement the architecture

with various methods ranging from fine grain pipelines template-based using delay insensitive cells to components relying on bounded delay based with no pipelining at all. The asynchronous Fano has been implemented with fine grain pipelining using PCHB templates. Slack optimization in consideration of performance is also completed in this step. At the end of the micro-architecture design there are two possible options.

One option is to keep going in the decomposition and generate a leaf cell design. The leaf cell design will depend on the template used (PCHB, RSPCHB, LP3/1, HC…). The next step is to generate a gate level netlist of the whole circuit just like in synchronous design. The gate library consisting of static and dynamic gates will be mapped to the netlist and the design can be laid out using standard place and route tools.

The other option is to generate a leaf cell netlist rather than going any further and use a leaf cell library. The leaf cell library would be mapped to the netlist and the automatic place and route would be done at the leaf cell level rather than the lower gate level. This option would probably yield denser circuits with better performance since the leaf cells would be optimized and laid out using more of a full custom approach, although even automatic place and route can be applied to generate the leaf cells. Choosing the first option and applying place and route directly on a gate netlist can lead to a number of undesired effects. One of them is a less dense circuit since rather than sharing area and optimizing leaf cells, the leaf cells will be implemented with discrete gates. Another issue is that the handshaking circuits might not be as close to the dynamic functional evaluation circuit when the place and route is applied to the gate netlist rather than the leaf cell netlist, therefore effecting performance.

We have used the Virtuosa Schematic Editor from Cadence as a schematic entry tool to design the PCHB based leaf cell. All of the decomposition was also done using this tool. Initially only the functional and symbol views of the dynamic and static gates needed in the design are created and added to the asynchronous cell library. The functional description of a dynamic circuit used as a buffer is shown below in Figure 7-3.

```
module Dynamic_BUFFER_Function (nBUF0, nBUF1, A0, A1, BUFe, en, BUF1, BUF0);
    output nBUF0;
    output nBUF1;
    output BUF0;
    output BUF1;

    input A0;
    input A1;
    input BUFe;
    input en;

    reg nBUF1, nBUF0, BUF1, BUF0, temp;

    initial begin temp=0; end

    parameter  D1=10; //unit delay 1
    parameter  D2=20;

    always @(BUFe or A0 or A1 or en)
    begin
        if(BUFe==1 && en==1 && temp==0)
        begin
            if(A1==1 && A0==0) begin nBUF1 <= #D1 0; nBUF0 <= #D1 1;
                                     BUF1 <= #D2 1; BUF0 <= #D2 0; temp=1; end
            else if(A1==0 && A0==1) begin nBUF1 <= #D1 1; nBUF0 <= #D1 0;
                                          BUF1 <= #D2 0; BUF0 <= #D2 1; temp=1;end
        end

        else if(BUFe==0 && en==0)
        begin
             nBUF1 <= #D1 1; nBUF0 <= #D1 1; BUF1 <= #D2 0; BUF0 <= #D2 0;
             temp=0;
        end
    end
```

Figure 7-3: The functional description of a dynamic buffer

Once the design is completed and the correctness has been verified at the behavioral level, the schematic (transistor) views of the cells are implemented for spice simulation. The transistor level view of the dynamic buffer is shown below in Figure 7-4.

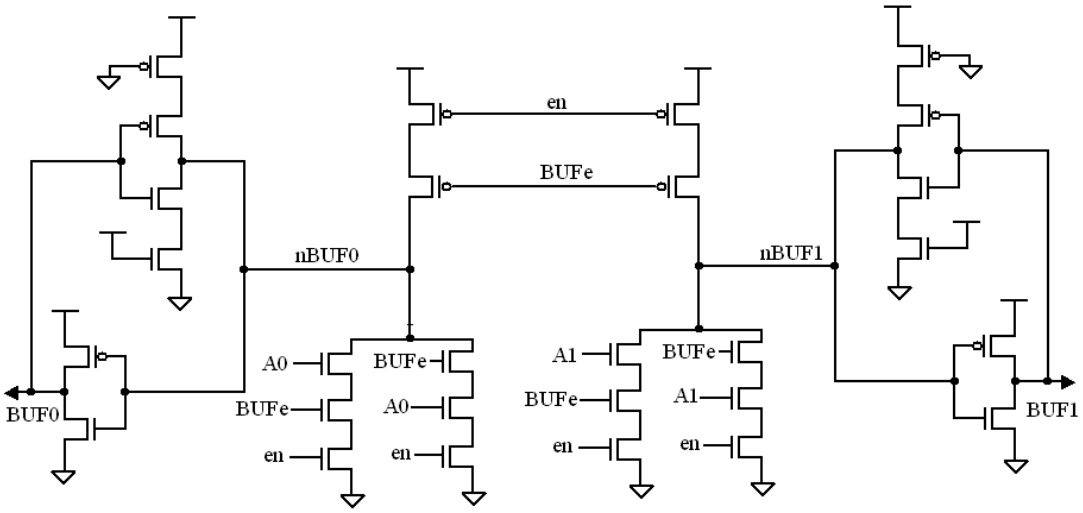


Figure 7-4: The transistor view of a dynamic buffer

For spice simulation we have used Nanosim from Synopsys. The layout views were created once we were confident that the design worked as expected at the transistor level. The layout view for the dynamic buffer is shown below in Figure 7-5.

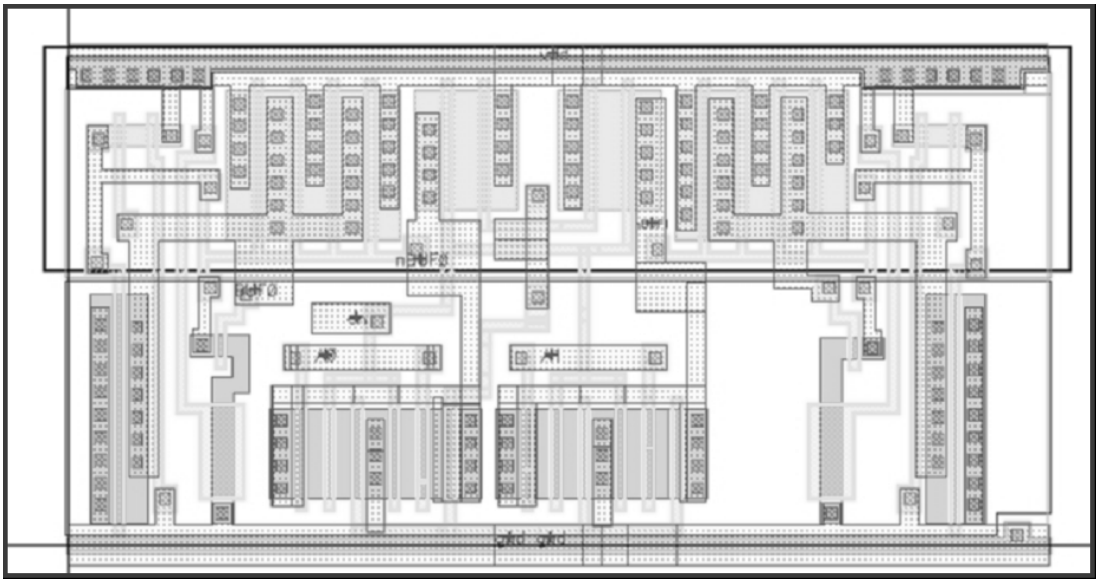


Figure 7-5: The layout view of a dynamic buffer

One important aspect of designing cells for dynamic logic is charge sharing and transistor sizing. After a number of test simulations on individual cells we have decided to use 8X for the size of the output transistors, 2X for the pull-down transistors. The staticizer inverters were set to approximately 1/10 the strength of the pull-down transistors to balance reliability of operation against speed. The other aspect for reliable operation is charge sharing. Unlike the schematic in Figure 7-4, if the *nBUF1* and *nBUF0* signals were generated using the *A*, *en* and *BUFe* signals as a stack of three transistors in series, there would the possibility of the internal dynamic nodes nBUF1 and nBUF0 loosing their value due to the charge sharing. This scenario could occur if *A* and *en* were asserted high turning on their respective transistors and *BUFe* was still asserted low. To prevent this problem, we have opted to use a widely known solution of doubling the pull-down logic and cross-coupling it as illustrated in Figure 7-4.

To reduce to load on the automatic place and route tool and to meet the performance of the circuit we partitioned the top-level design into a number of blocks as shown in Figure 6-5. The place and route, which was performed using Silicon Ensemble from Cadence, was not timing based, to show that a QDI based asynchronous circuit will work no matter what the delays are as long as the isochronic fork assumption is met. The Figure 7-6 below is a snapshot illustrating the cell placement of the counter block. The picture is zoomed in to the lower left corner of the design for clarity.

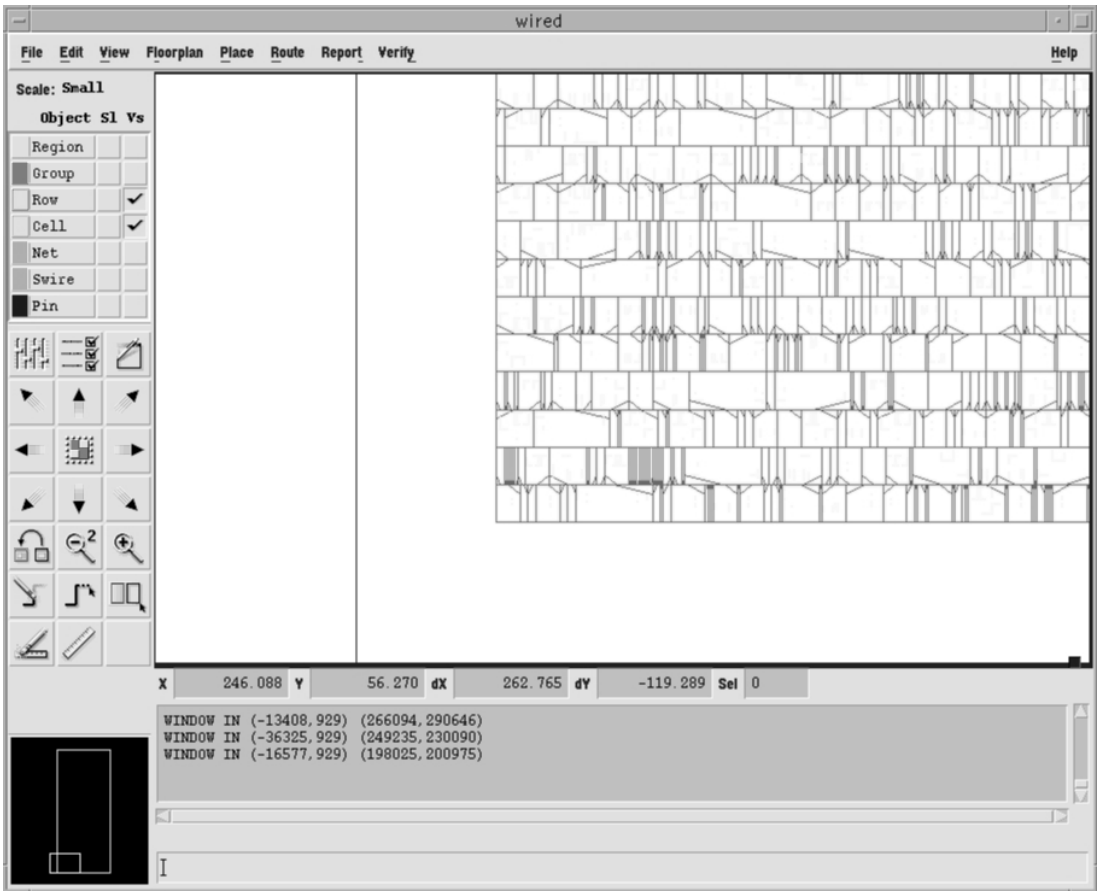


Figure 7-6: Cell placement in Silicon Ensemble

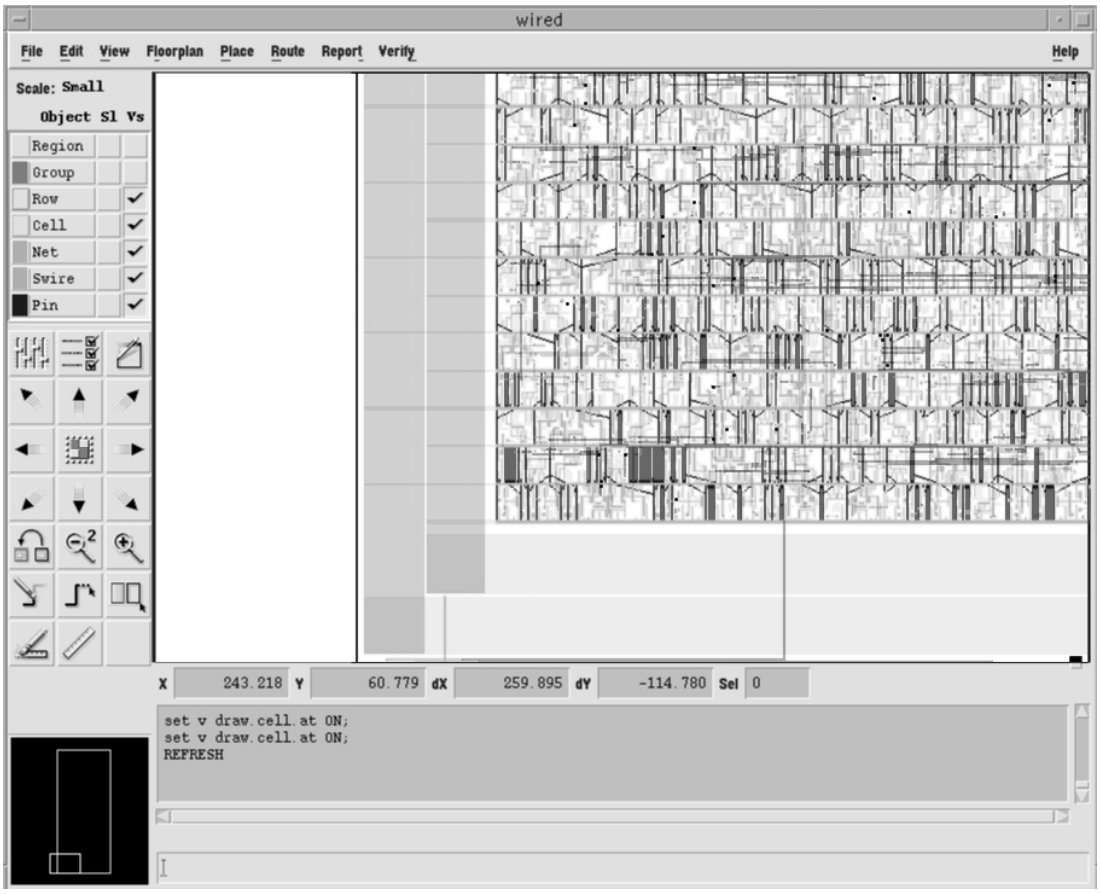


Figure 7-7: Routed Counter block with Silicon Ensemble

Each block was streamed back into the Virtuosa Layout Editor for DRC and LVS check against its transistor level netlist. The LVS check also generates an extracted netlist of the design for spice simulation. A short sample of the extracted netlist is shown below. The flattened netlist consists of two parts, the transistor connections and the extracted capacitances.

```
*
*  CADENCE/LPE SPICE FILE : SPICE
*                 DATE :  5-JUN-2003
*
******   MOS XTOR    PARAMETERS FROM : 7MOSXREF
*.GLOBAL   VDD!  GND!
.SUBCKT INC2   DATA  REQ  ACK  NRST4  L0  L1
*
******     CORNER ADJUSTMENT FACTOR =   0.0000000
******
MM2-XI60-XI36  XI36-A  NET0432  VDD!  VDD!  PCH  L=0.24U  W=2.80U  AD=1.04P
+         PD=3.54U  AS=1.88P  PS=6.94U  NRS=0.079  NRD=0.079
MM3-XI60-XI36  XI36-A  NR<6>  VDD!  VDD!  PCH  L=0.24U  W=2.80U  AD=1.04P
+         PD=3.54U  AS=1.88P  PS=6.94U  NRS=0.079  NRD=0.079
MM7-XI60-XI36  XI36-XI60-NET029  NET0432  XI36-A  GND!  NCH  L=0.24U  W=1.20U
+         AD=0.24P  PD=1.60U  AS=0.44P  PS=1.94U  NRS=0.183  NRD=0.167
MM7-XI60-XI36-1  685  NET0432  GND!  GND!  NCH  L=0.24U  W=1.20U  AD=0.24P
+         PD=1.60U  AS=0.80P  PS=3.74U  NRS=0.183  NRD=0.167
...
MM1-XI59-3  NET72  XI59-NET35  VDD!  VDD!  PCH  L=0.24U  W=2.50U  AD=0.93P
+         PD=3.24U  AS=1.65P  PS=6.32U  NRS=0.088  NRD=0.088
*
*----- TOTAL # OF MOS TRANSISTORS FOUND :       2018
*
*
C1    NET77  GND!  8.00421E-15
C2    NET209  GND!  1.06917E-14
C3    NET188  GND!  1.16892E-14
C4    NET121  GND!  1.34065E-14
C5    NET215  GND!  1.02445E-14
...
C583    XI36-XI56-NET016  GND!  6.91710E-17
C584    XI29-XI50-XI50-NET016  GND!  1.85150E-17
C585    XI30-XI50-XI50-NET016  GND!  4.84647E-17
*
*----- TOTAL # OF CAPS FOUND :        585
*-----              COMMENTED :          2
*
.ENDS
```

Figure 7-8: Extracted netlist of a block

The layout of the whole design is show in Figure 6-5. All of the blocks have been individually placed and routed. However the routing between the blocks have been done manually.

8. References

[1] International Technology Roadmap for Semiconductors – 1999 Edition. http://public.itrs.net/files/1999_SIA_Roadmap/Home.htm

[2] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz. In *IEEE ISSCC Digest of Technical Papers,* pp. 292–293.

[3] W Belluomini, C.J. Myers, H.P. Hofstee. Verification of delayed-reset domino circuits using ATACS. In *Proc. of Advanced Research in Asynchronous Circuits and Systems*, 1999 pp. 3 –12.

[4] H.P Hofstee, Sang H. Dhong; D. Meltzer, K.J Nowka, J.A. Silberman, J.I. Burns, S.D Posluszny, O. Takahashi. Designing for a gigahertz [guTS integer processor]. In *IEEE Micro,* vol. 18, no.3, pp. 66 –74, May-June 1998.

[5] D. Harris, M.A. Horowitz. Skew-tolerant domino circuits. In *IEEE Journal of Solid-State Circuits*, vol. 32, no.11, pp. 1702 –1711, Nov. 1997.

[6] Joep Kessels and Paul Marston. Designing asynchronous standby circuits for a low-power pager. In *Proceedings of the IEEE,* vol. 87, no. 2, pp. 257–267, February 1999.

[7] J.D. Garside S.B. Furber, and S.H. Chung. AMULET3 revealed. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems,* pp. 51–59, April 1999.

[8] M. Benes, S. M. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and System*s, pp. 43–56, 1998.

[9] Shai Rotem, Ken Stevens, Ran Ginosar, Peter Beerel, Chris Myers, Kenneth Yun,Rakefet Kol, Charles Dike, Marly Roncken, and Boris Agapiev. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 60–70, April 1999.

[10] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLS*I, pp. 164–181, September 1997.

[11] Hiroaki Terada, Souichi Miyata, and Makoto Iwata. DDMP's: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEE*E, Vol. 87, No. 2, pp. 282–296, February 1999.

[12] M. Benes, S. M. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and System*s, pp. 43–56, 1998.

[13] S. B. Furber, D. A. Edwards, and J. D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proc. International Conf. Computer Design (ICCD*), September 2000.

[14] Kenneth Y. Yun, Peter A. Beerel, Vida Vakilotojar, Ayoob E. Dooply, and Julio Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Transactions on VLSI System*s, vol. 6, no.4, pp. 643–655, December 1998.

[15] S. Hauck. Asynchronous Design Methodologies, An Overview. *Proceedings of the IEEE*, vol. 83, no.1, pp. 69-93, January 1995.

[16] C. J. Myers, Asynchronous Circuit Design, John Wiley and Sons, July 2001.

[17] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Desig*n, chapter 6, pp. 237–283. North-Holland, 1990.

[18] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesi*s. Prentice Hall, 1995.

[19] Ad M. G. Peeters. *Single-Rail Handshake Circuit*s. PhD thesis, Eindhoven University of Technology, June 1996.

[20] Charles L. Seitz, System timing, In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI System*s, chapter 7. Addison-Wesley, 1980.

[21] Steven M. Nowick, Kenneth Y. Yun, and Peter A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. Inter-national Symposium on Advanced Research in Asynchronous Circuits and System*s, pp. 210–223. IEEE Computer Society Press, April 1997.

[22] Peter A. Beerel, Sangyun Kim, Pei-Chuan Yeh, and Kyeounsoo Kim. Statistically optimized asynchronous barrel shifters for variable length codecs. In *International Symposium on Low Power Electronics and Desig*n, pp. 261–263, August 1999.

[23] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

[24] Andrew M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1996.

[25] Ivan E. Sutherland. Micropipelines. *Communications of the AC*M, vol.32, no.6, pp. 720–738, June 1989.

[26] S. B. Furber and J. Liu. Dynamic logic in four-phase Micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and System*s. IEEE Computer Society Press, March 1996.

[27] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with applica-tion to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and System*s, pp. 122–133. IEEE Computer Society Press, March 1996.

[28] Erik Brunvand. Parts-R-Us: A Chip Aparts. Technical Report CMU-CS-87-119, Carnegie Mellon University, May 1987.

[29] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuit*s. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[30] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computin*g, vol.1, no.4, pp. 197–204, 1986.

[31] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLS*I, pp. 263–278. MIT Press, 1990.

[32] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computin*g, vol.1, no.4, pp. 226–234, 1986.

[33] Kees van Berkel, Ferry Huberts, and Ad Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologie*s, pp. 99–106. IEEE Computer Society Press, May 1995.

[34] Peter A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuit*s. PhD thesis, Stanford University, 1994.

[35] T. Nanya, A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, F. Okamoto, H. Fujimoto, O. Fujita, M. Yamashina, and M. Fukuma. TITAC-2: A 32-bit scalable-delay-insensitive microprocessor. In *Symposium Record of HOT Chips I*X, pp. 19–32, August 1997

[36] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computer*s, vol.11, no.2, pp. 50–63, 1994.

[37] C. Myers. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conferenc*e, February 2001.

[38] Ken Stevens, Ran Ginosar, and Shai Rotem. Relative timing. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and System*s, pp. 208–218, April 1999.

[39] Hoshik Kim and Peter A. Beerel. Relative timing based verification of timed circuits and systems. In *Proc. International Workshop on Logic Synthesi*s, June 1999.

[40] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. *IEEE Transactions on Computer-Aided Desig*n, vol.18, no.9, September 1999.

[41] S. H. Unger, *Asynchronous Sequential Switching Circuit*s. New York NY: Wiley-Interscience, 1969.

[42] S. M. Nowick, D. L. Dill, Automatic Synthesis of Locally-Clocked Asynchronous State Machines. In *Proceedings of ICCAD,* pp. 318-321, 1991.

[43] S. M. Nowick, D. L. Dill, Synthesis of Asynchronous State Machines Using a Local Clock. In *Proceedings of ICCD,* pp. 192-197, 1991.

[44] K. Yun, D. Dill, Automatic Synthesis of 3D Asynchronous State Machines, In *Proceedings of ICCAD,* pp. 576-580, 1992.

[45] A. Davis, B. Coates, K. Stevens, The Post Office Experience: Designing a Large Asynchronous Chip. In *Proceedings of the 26th Annual Hawaii International Conference on Systems Science*s, vol. I, pp. 409-418, 1993.

[46] T. Murata, Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE,* vol. 77, no. 4, pp. 541-580, 1989.

[47] C. E. Molnar, T. P. Fang, F. U. Rosenberger, Synthesis of Delay-Insensitive Modules. In *Proceedings of the 1985 Chapel Hill Conference on Advanced Research in VLS*I, pp. 67-86, 1985.

[48] T. A. Chu, Synthesis of Self-timed VLSI Circuits from Graph-Theoretic Specifications. M.I.T. Tech. Rep. MIT/LCS/TR-393, June 1987.

[49] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, March 1997, pp. 315-325.

[50] A. J. Martin, "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits", in *UT Year of Programming Institute on Concurrent Programming,* C. A. R. Hoare, Ed. MA: Addison-Wesley, 1989, pp. 1-64.

[51] J. Kessels, A. Peeters The Tangram framework: asynchronous circuits for low power. *Proceedings of the ASP-DAC* 2001, pp. 255 –260, 2001

[52] A. Bardsley, D. A. Edwards. The Balsa Asynchronous Circuit Synthesis System. *FDL 2000.* 4-8th September 2000

[53] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Science*s, 5:511–523, 1971.

[54] Private communications with Andrew W. Lines, 2001.

[55] I. Sutherland, and S. Fairbanks. GasP: a minimal FIFO control. In *Proc. of ASYNC, 2001,* pp. 46–53.

[56] W.S. Coates, J.K. Lexau, I.W. Jones, S.M. Fairbanks, and I.E. Sutherland. FLEETzero: an asynchronous switching experiment. In *Proc. of ASYNC,* 2001, pp. 173–182.

[57] M. Singh, and S.M. Nowick. High-throughput asynchronous pipelines for fine grain dynamic datapaths. In *Proc. of ASYNC, 2000,* pp. 198–209.

[58] M. Singh, and S.M. Nowick. Fine-grain pipelined asynchronous adders for high-speed DSP applications In *Proc. of IEEE Computer Society Annual Workshop on VLSI, Orlando, FL,* April 2000, pp. 111–118.

[59] T.E. Williams, and M.A. Horowitz. A Zero-overhead self-timed 160ns 54b CMOS divider. In *ISSCC Digest of Technical Papers,* 1991, pp. 98-296.

[60] J. B. Anderson and S. Mohan. Sequential coding algorithms: A survey cost analysis. *IEEE Trans. on Communications*, COM-32, pp. 169-176, Feb. 1984

[61] S. Lin and Jr. D. J. Costello. *Error Control Coding: Fundamentals and Applications.* Prentice Hall, Englewood Cliffs, N.J. 1983

[62] J. M. Wozencraft and I. M. Jacobs. *Principles of Communication Engineering.* John Wiley and Sons, 1965

[63] P. J. Black. Algorithms and Architectures for High-Speed Viterbi Decoding. PhD thesis, Standford University, 1993