Asynchronous Pulse Logic

Thesis by

Mika Nyström

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

Caltech Computer Science Technical Report
caltechCSTR:2001.011

2001
(Defended May 14, 2001)

Acknowledgments

What am I doing in graduate school? My interest in computing goes back to when I disassembled, with my father, an Original-Odhner pinwheel calculating machine. I must have been about four or five years old at the time, and like most four-or-five–year-olds who have a passing acquaintance with addition and subtraction, arithmetic is something very mysterious. The sight of the many, many tiny gears in the insides of that calculator has dogged me ever since. I have been determined to figure out how it worked. Today I think I have a fair idea.

This thesis is first and foremost dedicated to my parents, who started it all—in more ways than one. I am sad to say that they did not live to see me through school; I should not have been here without their insistence on education before all else. I shall always be baffled by my mother's courage in sending me to a school where a foreign language—English—was the only one we young students could communicate in. Eventually it led to my leaving my native Sweden just as she once left her native Finland.

One's own insistence is not enough. I have been lucky to live in a world where willing and able teachers are plentiful. Starting with Lynn Carlisle, whom I still blame when my accent marks me a Midwesterner, I have had many: Maurice Naddermier, who first opened my eyes to the vastness of science; Simeon Leet, who in three years of high school taught me most of the mathematics that I know to this day and gave me a strong appreciation for the place of science and technology in the larger universe of human wisdom; and latest but not least, my advisor, Alain J. Martin, whose patience in the last seven years with the many things I have enjoyed myself working on has seemed infinite. I have had many other inspiring teachers in between; and if I were to list also those teachers that have taught me through their writings, the list would go on forever (some of their names appear in the Bibliography).

I came to graduate school so that I could continue learning; in this, I have been successful. Our Friday morning group meetings have always been nourishing food for thought; I remember many inspiring debates with José Tierno, Tony Lee, Marcel van der Goot, Jessie Xu, Peter Hofstee, Andrew Lines, Rajit Manohar, Paul Pénzes, Robert Southworth, Uri Cummings, Eitan Grinspun, Cathy Wong, Karl Papadantonakis, Abe Ankumah, and many occasional visitors. Andrew and Rajit, with their two very different but complementary world-views, have especially inspired me,

and much of what I write about here originally came from their minds and only made a detour through mine before landing on paper.

The members of my thesis committee, André DeHon, Alain Martin, Rajit Manohar, and Ali Hajimiri, have read the thesis carefully and suggested many changes. I have tried to control my own stubbornness and take as much of their advice as possible, but I have not always been successful in this.

Several others have read the thesis out of the goodness of their hearts and given me hundreds of helpful comments: Karl Papadantonakis, Cathy Wong, Eitan Grinspun, Paul Pénzes, Abe Ankumah, and the anonymous Institute Proofreader. Especially Chapter 3 owes much of its Greek clarity to Karl's demands. Again, these kind readers' comments have had a powerful foe in my stubbornness, and remaining errors are all mine.

This thesis is written in English (or American; but I think they are the same), a language that I learned (from Lynn) when I was four years old. Writing a Ph.D. thesis is something one tries to do carefully, so before I embarked on it, I decided to try to improve my writing by reading up on the English language. I may have succeeded, but I have also acquired some unusual habits of expression that I should perhaps caution the reader of. (Such as obstinately putting prepositions at the ends of sentences whenever I deem it appropriate.) These habits have been inspired by H. W. Fowler.[1] As I am writing computer-science English, I am permitted a great degree of leeway. I do not know why so many computer scientists write such sloppy English; we that are used to expressing ourselves with such exactitude when dealing with machines seem to often fall flat on our faces when trying to deal with human readers. I have tried to use my technical writer's leeway to write unambiguously rather than sloppily; in this I have made fewer concessions to convention than Fowler himself would have. No doubt sometimes all that I have achieved is annoyingly unconventional writing, and for all my efforts I have not managed to completely banish the sloppy mistakes; I am for instance convinced that hyphenation in English, especially noun-stacking technical English, walks a thin line between illogicality and illegibility. I cannot blame Fowler for these failings: they are due to my own pedantry and my sometimes foolish wish for consistency.

We live in a practical world, and research does not come of the mind alone. Many generous sponsors have made the work described here possible: it has been supported by the Defense Advanced Research Projects Agency (DARPA) for a long time, and I have been directly supported by the Okawa Foundation and by a two-year Graduate Research Fellowship from IBM Corporation. The computers used for the work were donated by Intel Corporation.

I am happy I came to graduate school at Caltech. It took longer than I thought possible, but

[1] H. W. Fowler and F. G. Fowler, *The King's English*, third edition (Oxford: Oxford University Press, 1931); H. W. Fowler, *A Dictionary of Modern English Usage*, first edition (Oxford: Oxford University Press, 1926). Both of these, as well as several lesser editions of *Modern English Usage*, are still in print. *The King's English* is also available on the Internet.

Caltech is a truly remarkable place filled with remarkable persons. On the practical side, my stay as a graduate student has been made less stressful by the hard work of many Computer Science staff members; especially Cindy Ferrini, Jeri Chittum, and Betta Dawson.

I thank Deidre for her patience with me.

This page intentionally left blank.

Abstract

This thesis explores a new way of computing with CMOS digital circuits, single-track–handshake asynchronous pulse-logic (STAPL). These circuits are similar to quasi delay-insensitive (QDI) circuits, but the normal four-phase QDI handshake is replaced with a simpler two-phase pulsed handshake. While a delay-insensitive two-phase handshake requires complicated decoding circuits, the pulsed handshake maintains the simpler, electrically beneficial signaling senses of four-phase handshaking by using timing assumptions that are easy to meet.

We cover many aspects of designing moderately large digital systems out of STAPL circuits, from the communicating-process level to the production-rule and transistor level.

We study the theory of operation of pulsed asynchronous circuits, starting with simple pulse repeaters; hence we progress to a general theory of operation for pulsed asynchronous circuits. This theory is a generalization of the theory of operation of synchronous digital circuits.

We then develop the family of STAPL circuits. This is a complete family of dataflow processes: the presented circuits can compute unconditionally as well as conditionally; they can also store state and arbitrate.

Next, we present some aspects of automatic design-tools for compiling from a higher-level description to STAPL circuits. Many of these aspects apply equally well to tools for QDI circuits; in particular, we study boolean-simplification operations that may be used for improving the performance of slack-elastic asynchronous systems.

Finally, a simple 32-bit microprocessor is presented as a demonstration that the circuits and design methods work as described. Comparisons are made, mainly with QDI asynchronous design-styles: SPICE simulations in 0.6-μm CMOS suggest that a system built out of automatically compiled STAPL circuits performs at about three times higher throughput (650–700 MHz in 0.6-μm CMOS) compared with a similar system built out of carefully hand-compiled QDI circuits; the STAPL system uses about twice the energy per operation and twice the area; in other words, the STAPL system improves on the QDI system by four to five times as measured by the Et^2 and At^2 metrics.

This page intentionally left blank.

Contents

This page intentionally left blank.

List of Figures

This page intentionally left blank.

Chapter 1

Introduction

For 'tis your thoughts that now must deck our kings,
Carry them here and there; jumping o'er times,
Turning th'accomplishment of many years
Into an hour-glass: for the which supply,
Admit me Chorus to this history;
Who, prologue-like, your humble patience pray,
Gently to hear, kindly to judge, our play.
— *William Shakespeare,* The History of King Henry the Fifth *(1599)*

In January of 1979, the first of a series of conferences on "Very Large Scale Integration" took place at Caltech. The two keynote speakers, Gordon Moore of Intel and Carver Mead of our Computer Science Department, both spoke of the same concern, but from two very different viewpoints. Their concern was design complexity.

Moore, the conservative industrialist, questioned whether the electronics industry was really ready for VLSI: "If the semiconductor industry had a million-transistor technology like VLSI," he wrote in the article accompanying his talk, "I'm not so sure it would know what to do with it." [62] He seemed to find it a far-fetched idea that a circuit designer should possibly know how to make use of a canvas large enough to hold a system as complex as VLSI would allow.

Mead, on the other hand, recognized that VLSI was going to be an inevitable development, whether the designers know what to do with their canvases or not, and we should probably all agree today that his article to a large extent explains why Moore was concerned. "VLSI," Mead wrote, "is a statement about system complexity, not about transistor size or circuit performance." He continued, "Many fundamental ideas [pertaining to large-system design] have yet to be discovered. The architecture and algorithms for highly concurrent systems is even less well developed." [59] This introducing of the ideas of computer science into what until then had been thought of as mere circuit design was a step that was to have far-reaching effects. Mead's article went on to predict that the large-system design problem would in time be tackled, as the fundamental problems of device physics and fabrication had been before it.

A quarter century will soon have passed since these words were written. In this time, there have been great advances, along the lines Mead predicted, in circuit design techniques and in computer-aided design and design automation. But the fact remains that most of today's VLSI systems are understood in terms of the same finite-state machines that were used for describing the mainframes of the 1960's; the fact remains that highly concurrent systems are poorly understood, especially by circuit designers.

The inevitable conclusion is that today's multi-million–transistor chips have been made possible not mainly by new fundamental ideas, but by the almost superhuman efforts made in exploiting the old ones. Though the fact may puzzle him that reads in today's newspapers that he stands in the middle of a "tech revolution," Carver Mead's VLSI revolution, 25 years in coming, is yet unfulfilled.

1.1 The VLSI design process

VLSI-system design is the process of implementing and realizing a system specification, the *architecture,* as an electronic circuit. We shall assume that the architecture is given to us and that the fabrication is not our concern. Longtime tradition, due to IBM [11], divides the design process into two stages beyond computer architecture: *implementation* of the architecture by a micro-architecture and *realization* of the micro-architecture by a physical circuit design.

The border between implementation and realization, like that between the United States and Mexico, is an artificial demarcation drawn for political purposes. The VLSI border traditionally serves to separate high-level logical reasoning from electronic-circuit design, tasks usually performed by different people, or at least by different software systems.

1.2 From physics to computer science

It has slowly been realized that, as Carver Mead suggested, VLSI system design contains aspects of both software design and electrical engineering. In VLSI, the imagination of the mathematician and enthusiasm of the programmer finally meet with the pragmaticism of the engineer. c, we are told, is the speed limit; λ is the accuracy that we can build things with. But most of us would rather ignore the problems of others. So when we imagine and program a VLSI system, we do not allow c and λ to constrain our imagination or to damp our enthusiasm. We design our systems as if c and λ did not exist, and then we tell the engineer, "Implement this." When the wafers return, we say that the poor performance is not our fault: we cannot be blamed for any failure to deal with c and λ since we left this task to our friend, the engineer.

1.3 Asynchronous digital design

Poor performance is usually unacceptable for a VLSI system. Optimists have long studied asynchronous design techniques, hoping that they have found at least a partial solution to the design problem. While it is true that proponents of asynchronous design like claiming that asynchronous circuits offer speed and power advantages, the author believes that the main advantage of asynchronous design is more subtle than these: it is the designer's ability of easily composing circuits that operate at different points in the design space (characterized by speed, power, and design effort) without destroying the beneficial properties of any of the circuits.

What makes a digital system asynchronous? A system is asynchronous if, in short, it does not use a clock for sequencing its actions. Asynchronous logic has been used for computer design since the 1950's, when several members of the ILLIAC series of computers were designed partly asynchronously [45]; somewhat later, Digital Equipment Corporation's PDP-6 computer was a modest commercial success [20].

What unites all methods of asynchronous circuit design is that they all strive for making the speed of computing dependent on the operations that are being carried out. A slow operation is allowed to take longer than a fast one; the system continues to the next operation only once the previous one is completed.

It is as if we could assemble a troika consisting of an Arabian, a Shetland pony, and a draught horse, without losing the useful qualities of the individual horses. If we should try this with real horses, the harness would act much as the clock does in a synchronous system and render the exercise pointless. But the asynchronous troika may be able to pull its load better than even a well-matched synchronous team, because the horses are not harnessed together by the clock—the draught horse does not have to keep up with the Arabian, and we do not have to feed the big horses if we only have need for the pony.

By allowing us to divide up a system into smaller, more independent pieces, the asynchronous design technique simplifies the large-system design problem: the main goal of asynchronous design is addressing Carver Mead's concern of 1979.

1.4 Asynchronous design-styles

In a synchronous system, it is easy to know when a computation is done. When the clock edge arrives, we read out the results of the computation. If it is not finished by then, we say that the system is wrong and throw it on the trash heap. (Or—less violently—adjust the clock speed.) The computation must necessarily be done by the time the clock edge arrives, or else the synchronous model would not make sense.

In contrast, the chief difficulty in asynchronous design is knowing when a specific computation is done. If we encode data in the same way as in a synchronous system, e.g., using two's-complement numbers, and start an operation $f(x)$, and the number "5" should appear on the result bus of our asynchronous system, how are we to know that it signifies the result of the present computation, and not of the previous? Worse, might it not be the bitwise combination of the results of the previous and current computations?

1.4.1 Bundled-data design

The early asynchronous computers were designed in what we shall call the *bundled-data* style. Designing in this style, the designer assumes that he can build a delay that matches whatever the delay is of the computation that he is really interested in. This matched delay is used as an "alarm clock" that is started when $f(x)$ is started and that rings when we can be sure that $f(x)$ has been completely computed. The design style is called bundled data because the data travels in a "bundle" whose timing is governed by the control signal that we called the "alarm clock." As one might guess, arranging for the matched delay is the Achilles' heel of the bundled-data style. If the delay is too short, the system will not work; if too long, then it will work slowly. Especially if computation times are data-dependent, the matched delay can easily become a designer's nightmare. The matched delay mechanism's working rests on a form of *a priori* knowledge of relative timing; we shall call making use of such knowledge a *timing assumption.*

1.4.2 Delay-insensitive design-styles

Originally conceived of at about the same time as the bundled-data design-style, *delay-insensitive* logic design attempts using the data bits themselves for sequencing. By making every input transition (change in logic level) cause, either in itself or within a cohort of input transitions, an output transition or a detectable pattern of output transitions, we can at least make interfaces between processes delay-insensitive.

Systems built using the delay-insensitive philosophy range from the *speed-independent* investigated by D. E. Muller in the 1950's [63], which work under the assumption that all wire delays are negligible compared with the operator delays (which may be of any length), to the truly delay-insensitive, in which both operator delays and wire delays may be arbitrary. Martin has shown that, using a reasonable operator model,[1] truly delay-insensitive systems are of little use [51]; the work in our research group has mainly been within the *quasi delay-insensitive* (QDI) model, which is essentially Muller's speed-independent model with information added for distinguishing between

[1]This "reasonable" operator model defines an operator as a single-output device; using the "unreasonable" model that an operator must be nothing more than a transistor, it is easy to see that building a nontrivial delay-insensitive circuit with repetitive behavior is absolutely impossible.

wires whose delays must be short compared with the operator delays and wires whose delays may be arbitrarily long.

We cannot possibly do justice to the many different design methods that have been proposed for asynchronous-circuit design;[2] bundled-data and QDI design will however serve as convenient extremes that we can compare with.

Assembling a working system out of QDI parts is almost frighteningly easy: start from a correct sequential program, decompose it into communicating processes, compile these processes into circuits, put the pieces together, and everything works. The chief advantage of this way of designing systems is that once we have decomposed, the design style is completely modular: there is no implicit use of global information (i.e., no clock), and the different parts can be designed independently.

In this sense, QDI design comes close to finally putting Carver Mead's concern to rest. But there is one difficulty with QDI design: the requirement that the circuits work properly even if *all* operator delays were to vary unboundedly is a difficult one to satisfy; our satisfying it involves inserting much circuitry whose only purpose is checking for the occurrences of transitions that we may know would in any case take place.[3] We should say that QDI systems must still be designed "within reason": it is possible to make things not work by designing them very poorly; likewise, it still takes considerable work and skill to achieve good performance. Yet, with these things in mind, the message-passing QDI design-style allows the design of large, well-performing systems with relatively little design effort [53, 18, 55].

1.5 Contributions

This thesis makes its main contribution by developing a design style that allows making use of limited amounts of timing information, i.e., limited use of timing assumptions, without destroying the most important, system-simplifying property of QDI design, namely that of the data's carrying its own timing information. We do this by replacing some of the four-phase (return-to-zero) handshakes in a QDI circuit with pulses, thus breaking the timing dependencies that are the source of the performance problems of QDI circuits. Our ultimate goal is that of improving the performance of modular asynchronous systems so much that it becomes possible to use asynchronous techniques for implementing large systems that perform well, yet are easy to design.

The organization of the thesis is as follows:

---

[2]The interested reader should see Hauck's paper for a balanced introduction [34].

[3]The reader with experience in designing CMOS circuits will realize that the situation is especially bad with regard to the checking for the occurring of downward (**true** to **false**) transitions, since such checking must be done with p-transistors. If we know that a signal x has switched from **false** to **true**, and the signal y is the output of an inverter whose input is x, then what harm is there in assuming that y has switched from **true** to **false**? In practice there may be none; but in the QDI model, there is great harm, whence the QDI designer will find it necessary to check for y's going **false** with the dreaded extra p-transistor.

I. We develop a theory that accounts for the proper functioning of pulsed asynchronous circuits (Chapter 3).

II. We develop a new target for the compilation of CHP programs, the single-track–handshake asynchronous-pulse-logic (STAPL) circuit (Chapters 5 and 6). These circuits are as easy to compose as QDI circuits, yet they operate faster: they have fewer transitions per execution cycle (10 instead of QDI's 18 for many nontrivial circuits), and they have less loading from p-transistors (no input-completion circuitry in most cases, and even when it is present, it has no p-transistors).

III. We explore some properties of Pipeline Language 1 (PL1), a simple yet expressive language for describing asynchronous bit-level processes (Chapter 7). PL1 is a convenient language for expressing the behavior of basic dataflow-style processes. It succinctly captures all the capabilities we should like to have and that are easy to implement for simple asynchronous processes. The particular capabilities that we choose for the language are inspired by the MiniMIPS work: we thus have evidence that the capabilities are enough for implementing large and well performing asynchronous digital VLSI systems. It is much easier to work with descriptions at this level than in terms of production-rule systems; compared with CHP programs, the PL1 language allows only a subset that can be straightforwardly and automatically compiled. The PL1 language is intended to be used for both STAPL and QDI design.

IV. Putting the methods developed in previous chapters to the test, we study a microprocessor design consisting of STAPL circuits, most of which were themselves designed using the PL1 language (Chapter 8). The microprocessor is a simple 32-bit one; the design shows how best to take advantage of the capabilities of the STAPL circuit family. The results of the test are good: the STAPL family is shown to be capable of significantly higher throughput than QDI circuits at a small extra cost in energy; the overall improvement using the Et^2 metric is approximately a factor of five.

Chapter 2

Preliminaries

—And why not the swift foot of Time? had not that been as proper?

—By no means, sir. Time travels in divers paces with divers persons.

— William Shakespeare, As You Like It *(1599)*

2.1 Quasi delay-insensitive design

This thesis aims at establishing a new target for hardware designers; while asynchronous, the new pulsed-logic design-style depends on timing assumptions for working properly, which quasi delay-insensitive (QDI) circuits do not. Still, many of the design issues are very similar, especially at the higher levels of the design; consequently, we shall be able to reuse much of what is known of QDI design.

We shall use much of the same terminology and notation as QDI designers do. To wit, we shall compile our circuits starting from the Communicating Hardware Processes (CHP) language, a language based on Hoare's Communicating Sequential Processes (CSP) [36]; we shall describe our communication protocols using the notation of the Handshaking Expansion (HSE) language used by QDI designers; we shall describe our transistor networks using the Production-Rule Set (PRS) notation. These languages are all explained in detail by Martin [48, 54]; some more recent extensions to CHP, whose syntax was suggested by Matthew Hanna, are described by Hanna [33] and the author [66].

2.2 High-speed CMOS-circuits

Over time, the Caltech group's way of designing of asynchronous circuits has converged with some of the ways that high-speed synchronous circuits are designed. Most of what we shall discuss in this thesis falls into the broad category of "precharge domino logic." The basic techniques used for designing these kinds of circuits are well illustrated by Glasser and Dobberpuhl [29].

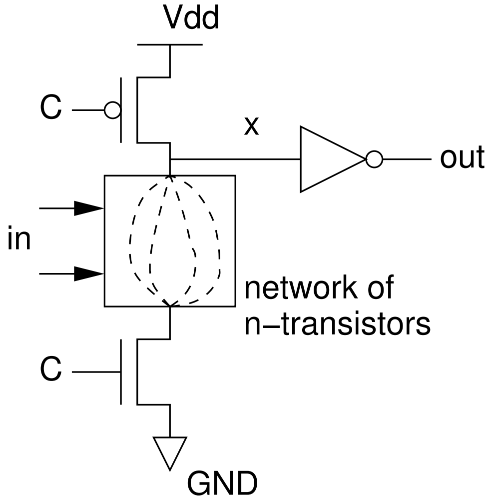

Figure 2.1: One stage of domino logic.

Here we shall only cover a few issues in nomenclature. A basic CMOS domino-logic "stage" is shown in Figure 2.1. The part on the left of the figure is the "precharged domino" part of the circuit. When the control signal C goes low, the stage precharges—the node x rises and the output *out* falls. When C next goes high, depending on the values on the *in* wires, the domino may or may not "fall" (i.e., x may or may not fall to GND). The name "domino logic" comes from these circuits' ability of being cascaded within a single clock-phase in a synchronous system. Confusingly, while Figure 2.1 depicts a single "domino stage," the same structure can also be called two "stages of logic"—the domino block plus the inverter. In the design style that we use, a block like this also implements an entire "pipeline stage"; i.e., cascaded dominos cycle independently (to an extent determined by the "reshuffling" of the handshake).

The important features of domino logic are as follows. There are few p-transistors; because of the much higher mobility in silicon of electrons compared with holes, this means that domino logic will usually be much faster than combinational logic, where pulling up the outputs has to be handled by the inputs. Furthermore, if we wish to cascade the dominos, each computation stage takes two logic transitions (one for the domino, one for the inverter)—this we call the *forward latency* of the stage; alternating "n-dominos" with "p-dominos" is possible, but the performance gain, if any, compared with standard domino logic, is small; owing to the many p-transistors in the p-dominos, this style can indeed be slower. An important drawback of domino logic is that it is more sensitive to different kinds of noise than combinational logic is.

In asynchronous circuits, the control transistor gated by C is sometimes replaced with several control transistors; this is necessary for accomplishing the more complicated synchronization that

Rail	Value			
$x.0$	**false**	**true**	**false**	**true**
$x.1$	**false**	**false**	**true**	**true**
Meaning	No data	$x = 0$	$x = 1$	***Illegal***

Figure 2.2: Dual-rail encoding of one bit of data.

can be required by asynchronous data-transfer protocols.

2.3 Asynchronous protocols and delay-insensitive codes

Asynchronous systems are based on handshaking protocols; i.e., two processes wishing to transfer data between each other synchronize the data transfers with signals that the processes themselves generate. It is most straightforward for us first to envision the handshake itself and then to add the data transfers in later. This way of designing things allows transferring data using conceptually simple protocols. One property that must be satisfied by the data is that it is encoded using a *delay-insensitive code.*

This means informally that the data encoding contains the same information that was present in the original "bare" handshake (i.e., data present or not-present) and that the data is encoded so that transitioning between the data present and not-present states is free from timing assumptions (i.e., it does not matter in which order the transitions are received). The most basic encoding that satisfies these conditions is the dual-rail encoding of a single bit (Figure 2.2); one that will also be seen often in this thesis is the 1-of-4 encoding of two bits.

Generalizing from bare handshakes to using delay-insensitive codes leads naturally to needing circuits for determining whether data is present or not-present. This we loosely refer to as "completion circuitry." For instance, a two-input OR-gate can be used for completing a dual-rail channel, as can a four-input OR-gate for a 1-of-4–coded channel.

2.4 Production rules

In this thesis, we shall not generally describe circuits at the level of transistor netlists; this would be unnecessarily much detail. Instead, we shall use *production rules.* A production rule (PR) is a statement of the form

$$G \longrightarrow x := c$$

where G is a boolean expression called the *guard* and the assignment $x := c$ is the *command.* In a production rule, c can only be **true** or **false**; nothing more complicated is allowed. We abbreviate

$x :=$ **true** as $x\uparrow$ (read as "x up") and $x :=$ **false** as $x\downarrow$ (read as "x down"). At the circuit level, the effect of such an elementary assignment is a transition on x from a low to a high or from a high to a low voltage.

In a given system, we must necessarily have rules for the setting of each node x that transitions more than once both to **true** and to **false**; the combination of the two rules is called an *operator.* In other words, an operator is a device with one or more inputs and a single output. The mapping from operators to circuit gates is fairly direct, but we do not consider it in detail in this thesis; nor do we consider layout issues in detail.

Before proceeding, it must be pointed out that although we use the same notation, our using timing assumptions means that we cannot ascribe quite the same semantics to HSE and PRS as we can in QDI designs. We shall have more to say about this later; in short, we disallow "stuttering" in HSE and we shall use a timed execution model for production rules instead of the weakly-fair–interleaving model that can be used for QDI circuits.

2.5 The MiniMIPS processor

The MiniMIPS processor, designed by the Caltech group during 1995–1998, represents the state of the art in QDI asynchronous design today [55].

The MiniMIPS processor consists of two million transistors; it has been fabricated in 0.6-μm CMOS, and in this technology, it runs approximately 170 MHz at the nominal process voltage (3.3 V).

A few notable features of the MiniMIPS processor are the following:

- Almost complete reliance on QDI circuits. (The exceptions are the low-level implementation of the cache-write mechanism and the off-chip bundled-data asynchronous interface.)

- Extensive use of 1-of-4 data-encoding to minimize completion delays and save switching power.

- Use of pipelined completion to minimize completion delays. Using pipelined completion results in a processor that can be thought of as an array of byte-slice processors, with a minimum of synchronization between the byte slices. The QDI model, which we used to verify that the design is correct, refuses to deal in delays; hence we know that the processor would work for a wide range of delays, and we simply try to pick those delays that shall result in the most efficient (i.e., the fastest) implementation. Thus the byte-slice processors nevertheless operate in synchrony most of the time, and we receive the benefits of a highly concurrent design with short, well-matched delays without paying the price of having to impose unwanted synchronization throughout our design model.

- Universal use of precharged, pseudo-static[1] domino-logic.
- A deeply pipelined design with buffering in every domino stage. The processor can execute many programs at an average speed of $18\,{}^{2}\!/_{3}$ logic transitions per fetch cycle.

As important as the MiniMIPS processor itself are the techniques used to design it:

- Initial specification as a sequential CHP program and stepwise refinement to a collection of concurrent processes.
- Use of slack elasticity [45] to allow variable latencies yet ensure deterministic behavior.
- Final, formal specification in terms of a hierarchical production-rule set (PRS), using the CAST language.
- Universal use of full-custom physical design, in terms of a `magic` cell hierarchy. Several man-years were spent on this aspect of the design.

[1] See footnote on p. 47.

2.6 Commonly used abbreviations

APL	Asynchronous pulse-logic
BDD	Binary-decision diagram
C	Consensus (in "C-element")
CAST	Caltech asynchronous synthesis tools (hardware description language)
CHP	Communicating hardware processes
CMOS	Complementary metal-oxide-semiconductor [field-effect transistor]
CSP	Communicating sequential processes
DI	Delay insensitive
DRAM	Dynamic random-access memory
ER	Event rule
GND	Ground (circuit node)
HP	Hewlett-Packard [Corp.]
HSE	Handshaking expansion
IBM	International Business Machines [Corp.]
MIPS	Microprocessor without interlocked pipeline-stages
MOS	Metal-oxide-semiconductor [field-effect transistor]
PCHB	Precharged half-buffer
PL1	Pipeline language 1
PR	Production rule
PRS	Production-rule set
QDI	Quasi delay-insensitive
RISC	Reduced–instruction-set computer
SPAM	Simple pulsed asynchronous microprocessor
SRAM	Static random-access memory
STAPL	Single-track–handshake asynchronous pulse-logic
Vdd	Positive power supply (circuit node)
VLSI	Very large-scale integration
WCHB	Weak-condition half-buffer

Chapter 3

Asynchronous–Pulse-Logic Basics

All delays are dangerous in war.

— *John Dryden,* Tyrannic Love *(1669).*

Over the years, asynchronous design techniques have gone from Muller's simple handshaking circuits and the carefully timed bundled-data circuits used in the PDP-6 to the sophisticated, yet easy to design, dataflow techniques used for the MiniMIPS design. It remains, however, that the MiniMIPS processor operates, under ideal conditions, at a fetching rate of 18⅔ CMOS transitions per instruction fetch, and designing a QDI microprocessor that fetches much faster than this seems an impossible challenge. This number compares favorably with the performance achievable by most synchronous design techniques, but it falls short of the 10–14 transitions per cycle that the most aggressive (and hard-working) synchronous designers achieve.

The barrier that prevents QDI circuits from achieving higher performance lies in the QDI handshake. By insisting on the four-phase handshake, e.g.,

$$*[\ [li];\ (\text{compute outputs});\ lo\uparrow;\ [\neg li];\ lo\downarrow\ ],$$

we demand that any process in our system shall, after it has acknowledged receipt of its inputs, wait for those inputs to reset to neutral. This is expensive because checking inputs' neutrality is done in p-transistors: hence it must be decomposed into several stages, and it also loads the inputs heavily. (Of course, switching to inverted logic does no good since then computing the outputs and checking the validity of the inputs must instead be done in p-transistors, which would be even worse than before.) The most trivial four-phase–handshaking QDI circuit takes ten transitions per cycle,[1] and anything nontrivial takes 14; inescapably, the complicated control-processes take 18. (These numbers are taken from the MiniMIPS [55] and Lines's work [43].)

Various authors have suggested that the solution to the performance problems that plague four-phase QDI circuits is that we should use two-phase signaling instead. Many variants exist; the

[1] Building a chain of buffers that take six transitions per cycle while remaining QDI is possible, but it cannot be done without inverting the signal senses from a buffer's input to its output.

simplest is:

$$*[\ [li \neq lo];\ \text{(compute outputs)};\ lo := li\ ]$$

Some things can be implemented well with this protocol, but designers struggle fruitlessly with anything but the simplest specifications when they must design logic. This is why: on each iteration of the loop, the sense of the input signal changes. At one moment, an input near Vdd means a **true** input; at another, it means a **false** input. Who can make sense of that?[2]

What we want is a design style that combines the straightforward logic of four-phase QDI with the timing of two-phase logic. Obviously, we cannot expect to have all the desirable properties at once. Accordingly, we shall no longer demand that the circuits be QDI; yet they will in many ways operate similarly to the QDI circuits we used in the MiniMIPS.

But is it even possible to use the MiniMIPS design-style for designing anything but QDI circuits; shall we not have to abandon all that we know of asynchronous design and start over?

It turns out that most QDI circuits that have been designed can be sped up considerably by introducing weak timing-assumptions, without our having to rethink the high-level design. The reason for this is simple: while using a four-phase handshake for implementing two synchronization actions is certainly possible (indeed, this technique is used in some special circuits, such as the distributed mutual-exclusion circuit designed by Martin [50]), this is not commonly done. In the dataflow-style processes used in the MiniMIPS, it is never done. Hence, out of the four phases of the four-phase handshake, only two are used: send and acknowledge. The remaining two, resetting the data and resetting the acknowledge, are not used for synchronization. These phases are entirely superfluous from the point of view of the specification.

Each phase consists of two actions: an assignment and the wait for that assignment. We can make use of the phases' being superfluous by eliminating the waits, even though we keep the assignments; by removing the waits, we get the synchronization behavior of two-phase handshaking; but by keeping the assignments, we keep the encoding properties of four-phase. What we propose doing is allowing communicating circuits to reset their interface nodes in parallel; in other words, once we acknowledge an input, we assume it will reset "quickly." This achieves the desiderata: the inputs may still always be in positive logic, yet their synchronization behavior will have many of the characteristics of two-phase signaling, since we only wait for the first phase of the inputs. Waiting for the first phase of the inputs is anyway normally required for computing the outputs, so what remains will definitely be closer to optimal.

In this chapter, we shall study a few simple pulsed circuits and then develop a theory that may be used to account for the proper functioning of a wide class of pulsed circuits and show how that

[2] The reader who finds this comment facetious is urged to contemplate the designing of a circuit that has several conditional inputs. Such a circuit will have to combine inputs of arbitrarily different senses, potentially in a different arrangement of senses for each iteration.

theory may be applied to the specific simple pulsed circuits.

3.1 Road map of this chapter

This is the most challenging chapter of this thesis as we make the needed connection between physics and computer science. So that the reader will not get lost in the chapter, let us first discuss the main points.

We shall first study the designing and simulating of a basic pulsed asynchronous circuit, viz. the pulse repeater. This will be an *ad hoc* discussion based on properties easily observable by and well-known to the electrical engineer: pulse lengths[3] (widths) and heights.

Secondly, we shall explore why these simple and readily observable properties are not enough for describing the full range of possible pulse-repeater behaviors. Simply speaking, the essential shortcoming of the pulse length and pulse height is that these two properties, while they may suffice for specifying a testing pulse that is applied to a circuit, do not suffice for completely describing the shape of the output waveform produced by that circuit.

Thirdly, we shall generalize the legal-logic-range–noise-margin argument commonly made for establishing the correctness of traditional synchronous circuits. This generalizing serves two purposes: on the one hand, it establishes a framework that we can understand the logic family of Chapters 5 and 3 within; on the other hand, it serves the wider purpose of taking a baby step towards establishing a formal model for the functioning of asynchronous circuits with the simple understandability of the synchronous model. The mathematical argument in this section may seem overly formal, but it is really a straightforward generalization of the synchronous argument.

In generalizing we consider uncountably infinite sets of waveforms instead of the simple voltage-ranges used by the synchronous argument. By specifying the sets of waveforms as "the set of all functions $f(t)$ such that each $f(t)$ is bounded below by the function $j(t)$ and above by $k(t)$," we reduce the argument to one where $j(t)$ and $k(t)$ play the leading rôles instead of the much more cumbrous infinite sets of functions. This corresponds to understanding the synchronous argument in terms of the boundaries between the voltage ranges rather than having to consider every possible intermediate voltage separately.

Fourthly, we shall recognize that, given certain desirable properties of the circuits that we study, establishing the correctness of an asynchronous–pulse-logic family can be done *entirely* in terms of functions $j(t)$ and $k(t)$; we shall determine what properties are necessary for allowing this vast simplification.

Lastly, we shall argue that the pulse repeaters we first studied in such an *ad hoc* way actually satisfy the conditions allowing the simplification. At this point, we shall have to appeal to more

[3] Also called "pulse width"; see the footnote on p. 17 for our justifying the "pulse length" nomenclature.

vaguely known things, such as transistor equations and circuit parameters; this argument is hence specific to the kind of circuits we are studying.

3.2 The pulse repeater

The first pulsed asynchronous circuit that we shall investigate is the "pulse repeater." A pulse repeater copies input pulses to its output. While this function could be performed by a wire or an open-loop amplifier, the pulse repeater has feedback; using the feedback, it restores the quality of the signals, both in the time domain and the voltage domain.

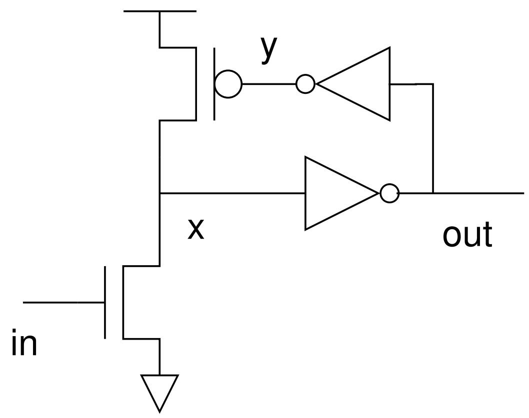

Figure 3.1: Three-stage pulse repeater.

The circuit in Figure 3.1 is a simple three-stage pulse repeater. In its idle state, both the input and the output are at a logic zero, and the internal node x is at a logic one; this is the only stable state of the circuit. When the input voltage is raised towards a logic one, the voltage on x begins to fall; which then causes *out* to rise, and finally, at least if *in* has meanwhile returned to zero, x to rise back to the idle state. The circuit can misbehave if *in* remains at a logic one for too long. Characterizing the misbehavior and finding ways of avoiding it are the main topic of the rest of this chapter.

In the three-stage pulse repeater, the node *out* (when repeaters are cascaded, *in* is a neighbor's *out*) is driven by an inverter, as is the node y. We shall see that, even as we introduce more and more dynamic nodes for handling complicated protocols, there will be nodes that shall remain driven by combinational logic. These nodes do not offer much opportunity for computing, so we shall direct our attention to the node x.

3.2.1 Timing constraints in the pulse repeater

The pulse repeater is a difficult circuit to get working reliably, owing to the timing assumptions that are necessary for verifying its correctness. If we will ensure that a pulse on *in* is noticed by the repeater, we must arrange that its length[4] exceed some minimum. On the other hand, the pulse must not be too long; if it is, then the circuit may produce multiple outputs for a single input. (Depending on device strengths, it may instead stretch the output pulse. We might endeavor to design a pulse repeater so that this stretching could be used to keep the circuit reliable even with arbitrarily long input-pulses. Owing to the difficult design problems posed by the transistor-ratioing constraints, designing a reliable pulse repeater along these lines is difficult.)

We shall not consider the possibility that two input pulses arrive so close together that they appear as a single pulse—for two reasons: first, the problem of the pulses' arriving too close together can be understood similarly to how we understand the single too-short and too-long pulses; secondly, we shall see that the issue is not of much concern in the APL circuit-family because the pulse-handshake protocols require inserting an acknowledgment of some sort between the two pulses (i.e., we ensure at a higher level of the design that we never have two pulses sent without the target's responding with an acknowledgment in between).

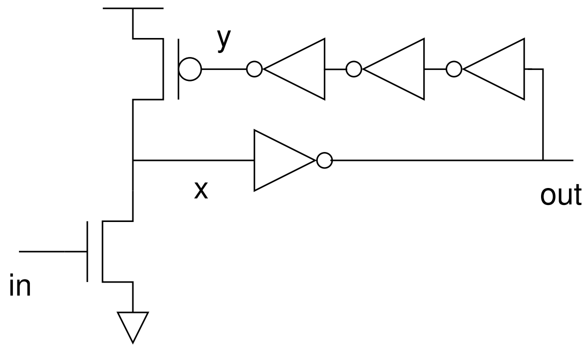

Figure 3.2: Five-stage pulse repeater.

3.2.2 Simulating the pulse repeater

The author has simulated a few variants of the pulse-repeater design described above with input pulses of varying lengths and heights applied, thus illustrating the timing margins of the pulse repeater. The repeaters that were simulated are similar to the simple three-stage version described above. The differences are that the input and output were negative-logic (i.e., the input transistor

[4]It is conventional to speak of pulse "widths" and interval "lengths"; using both concepts together, as we do, is apt to lead to confusion if this convention is adhered to. For this reason, we shall talk about pulse "lengths," thereby meaning the same as the conventional pulse "width." Similarly for "long" and "wide."

is a p-transistor) and that "keeper" resistors were used on the x nodes. We shall see the results for two separate circuit designs: a three-stage version, and a five-stage version that differs only in two extra inverters' being used in the feedback path from x to y (Figure 3.2). The author produced layout for the pulse repeaters using the `magic` layout editor and simulated them with the `aspice` circuit simulator. The assumed technology is HP's 0.6-μm CMOS via MOSIS; the supply voltage, Vdd, is 3.3 volts for all simulations presented in this thesis.[5]

In what follows, we shall mainly aim at understanding the behavior of a *single pulse traveling down an infinite chain of pulse repeaters.* Will the pulse die down? Will it lengthen until it becomes several pulses? Or will it—as we hope—travel down the chain unscathed?

Two things can go wrong with the pulse repeater. The input pulse can be too weak for the circuit to detect it, or the input pulse can be of such long duration that it is detected as multiple pulses. An example of a pulse repeater on the verge of misbehaving owing to a too-long input pulse is shown in Figure 3.3. The nodes are labeled as follows: input, $r.in$; internal node, $r.i1$; output, $r.out$; their senses are inverted compared with the pulse repeaters in the text. Here the input pulse is 1.5 ns long, beginning at t =10 ns. As we can see from the graph, the internal node $r.i1$ starts rising almost instantly, causing the output to fall about 200 ps later. At $t = 11$ ns, the internal node rises again, thus re-arming the circuit. Slightly before $t = 11.5$ ns, the re-armed circuit starts detecting the input—which has by now overstayed its welcome—as a second pulse, but the detecting transistor is cut off by the input, which falls back to GND barely in time to avoid being double-latched.

Figure 3.4 shows the results of applying pulses of varying lengths and heights to the three-stage pulse repeater. The pipe-shaped region shows when a single input pulse results in a single output pulse, as desired. The other two regions correspond to forms of misbehavior: the region to the right of the pipe shows when a single input pulse results in several output pulses, i.e., when the input pulse is too long to be properly detected as a single pulse; the region to the left of the pipe shows when the input pulse is too feeble to elicit any response at all. (The gaps in the plot are due to irrelevant practical difficulties with the simulations.)

Figure 3.5 shows the results for the five-stage pulse repeater. Figure 3.6 shows a plot for the five-stage pulse repeater of the length of the output pulse for different lengths of the input pulse, the input swing here being from GND to Vdd. The solid line shows the data; "0," "1," "2," and "3" indicate operating regions explained below. The diagonal dashed line in Figure 3.6 denotes the stability constraint that the output pulse is as long as the input pulse; we should expect that in an infinite chain of pulse repeaters, the pulses will eventually have the parameters implied by the intersection of the input-output curve and the stability constraint.[6]

[5] The parameters used are known to be inaccurate. The circuit speeds indicated by the simulations are 15–20 percent higher than what one can reasonably expect from fabricated parts. These parameters keep the simulations straightforwardly comparable with most of the work done in the Caltech group in the last five years.

[6] As will be clear from the rest of this chapter, this is a very naïve understanding of the situation; we are here trying to project the behavior of a many-dimensional system onto a single scalar dimension: the pulse length. The

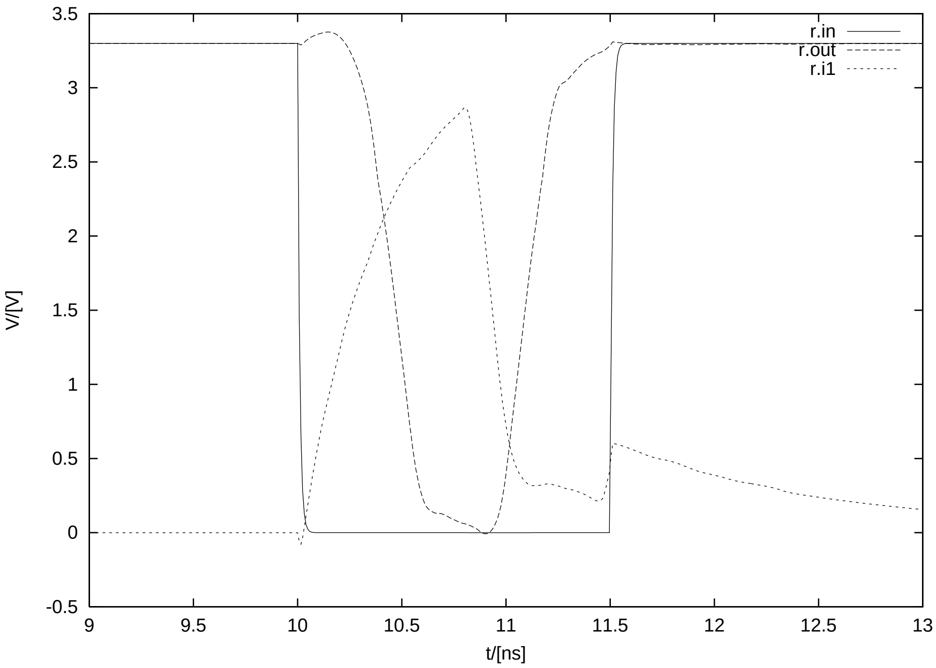

Figure 3.3: A long pulse almost triggers a pulse repeater twice.

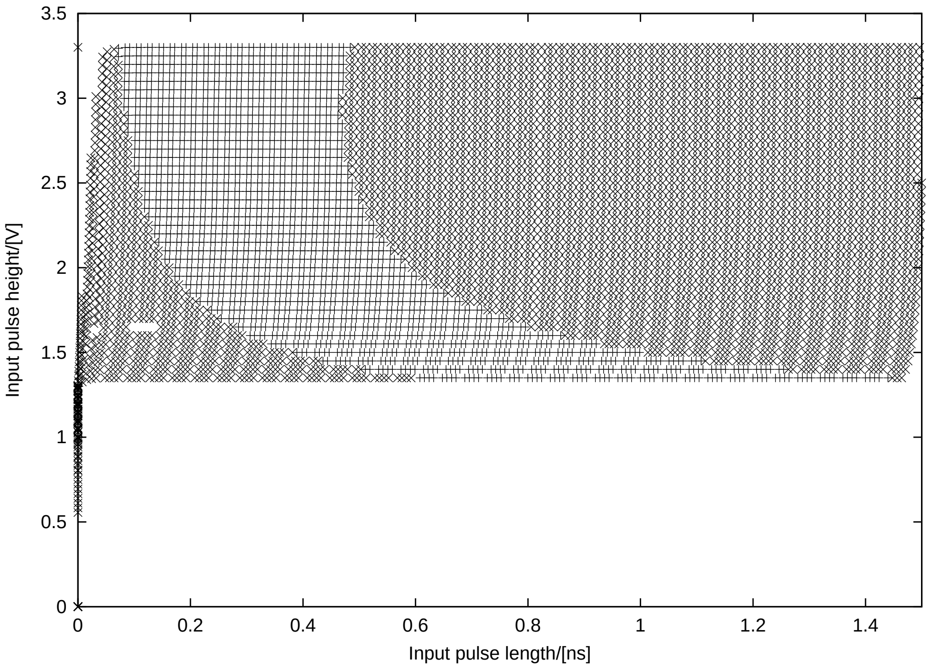

Figure 3.4: Shmoo plot for three-stage pulse repeater.

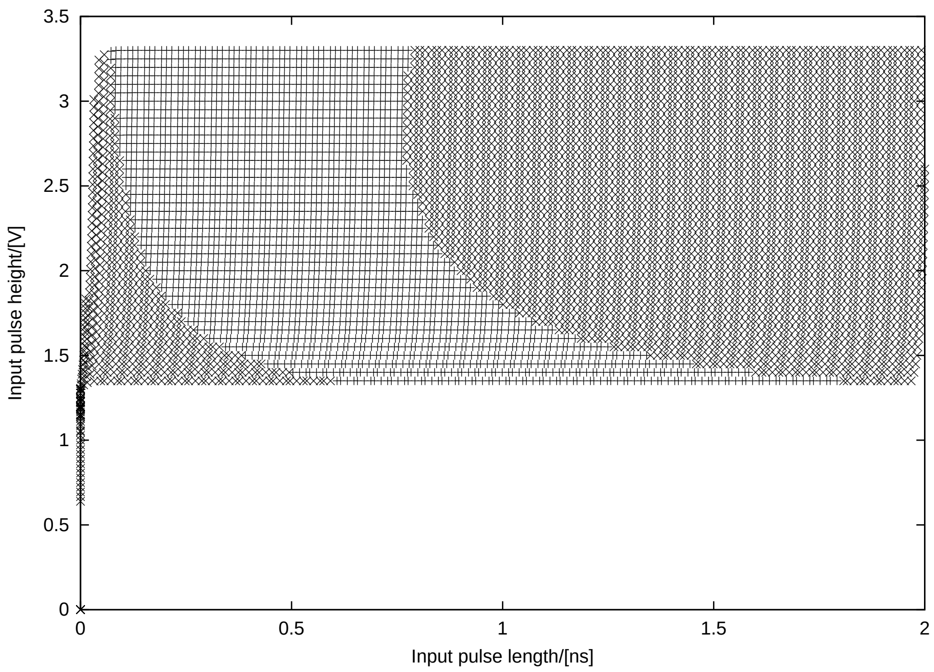

Figure 3.5: Shmoo plot for five-stage pulse repeater.

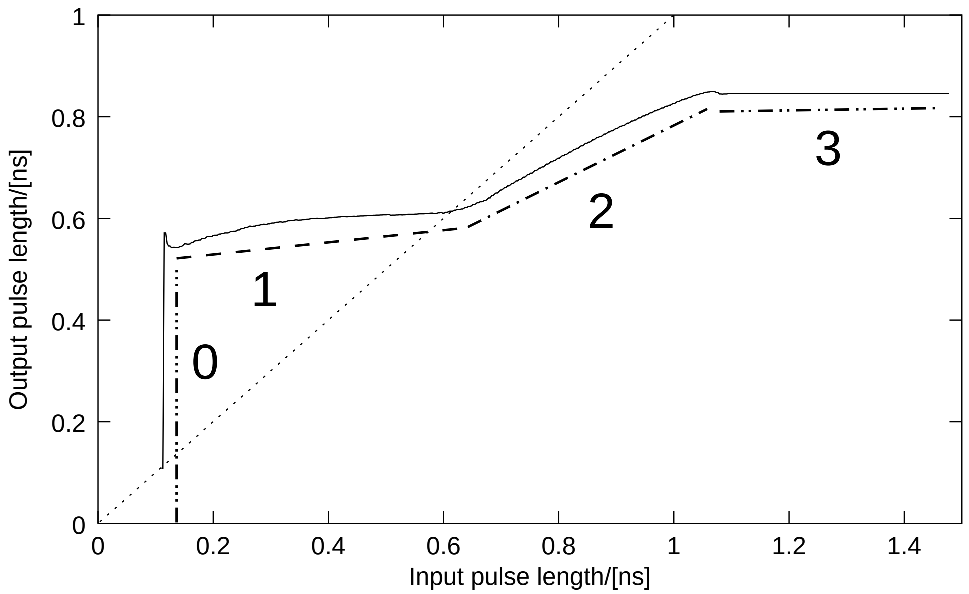

Figure 3.6: Input-output relationship of pulse lengths for five-stage pulse repeater; this particular circuit stops working for input pulses longer than 1.47 ns.

3.2.2.1 Analysis of pulse repeater data

There are two important questions we should ask when analyzing the pulse repeater data: First, can we cascade the circuits—can we connect them so that they work properly when the output of one is the input of another? Secondly, do the circuits work over a reasonable range of parameter variations?

The "shmoo" plots, Figure 3.4 and Figure 3.5, are caricatured in Figure 3.7.[7] Normally, if the input pulse is of a reasonable height and length (see below), then the gain of the pulse repeater will force the output pulse to be approximately characterized by the point marked "X" in the caricature. Furthermore, the line "A" describes the minimum pulse length that can be detected by the pulse repeater. This is set by circuit parameters, mainly by the strength of the input transistor and the load on its output. The other line, "B," marks the longest pulse length that will lead to a single output pulse.

The reason there is a maximum length that the repeater will not work properly beyond is that the repeater "double-latches" when the input pulse is so long that it is still present when the repeater has gone through the entire cycle $x{\downarrow}; \ldots y{\downarrow}; x{\uparrow}; \ldots y{\uparrow}$; furthermore, the up- and down-going behaviors of the pulse repeater are roughly similar; the same number of transitions is exercised, through roughly similar circuitry. Taken together, this means that the interval $x{\downarrow}; y{\downarrow}; x{\uparrow}$ (approximately the same length as the output pulse) is about the same length as the interval $x{\uparrow}; y{\uparrow}; x{\downarrow}$, where the final $x{\downarrow}$ is the misfiring resulting from the too-long input pulse. Hence, the pulse length along "B" will be about twice the length of the normal pulse "X."

3.2.2.2 Digital-analog correspondence

If we restrict ourselves to the digital domain, we can understand the pulse repeater's behavior for different input pulse lengths by considering the input pulse as two transitions $in{\uparrow}; in{\downarrow}$. The length of the input pulse is the length of the time interval between $in{\uparrow}$ and $in{\downarrow}$. $in{\uparrow}$ begins the operation of the pulse repeater; leaving out $in{\downarrow}$, the sequence of transitions is

$$in{\uparrow};\;\; x{\downarrow};\;\; out{\uparrow};\;\; y{\downarrow};\;\; x{\uparrow};\;\; out{\downarrow};\;\; y{\uparrow}\;\; .$$

Changing the input pulse length amounts to changing the position of $in{\downarrow}$ in this trace (we are here assuming that the sequence continues even in the absence of $in{\downarrow}$; i.e., in the presence of interference). There are five main possibilities:

0. $in{\downarrow}$ occurs so early that the pulse on in is too short to trigger the pulse repeater—then there will be no sequence $x{\downarrow}; out{\uparrow}$; etc. The repeater fails.

impatient reader is urged to take a peek at Section 3.3 and then to return here.

[7] An Internet search reveals the spelling "shmoo plot" as being five times more commonly used than the variant "schmoo plot."

1. $in\!\downarrow$ occurs long enough after $in\!\uparrow$ that the input pulse is noticed, but it occurs before $y\!\downarrow$. This is the ideal situation. There is no interference. The repeater works perfectly.

2. $in\!\downarrow$ occurs during $y\!\downarrow$. There is some interference, but because the input behavior is monotonic (the inputs tend to drive x strictly more towards Vdd as time goes by), the interference is fairly harmless—a slightly lengthened output pulse may result. The repeater still works.

3. $in\!\downarrow$ occurs after $y\!\downarrow$ but not long enough after it to trigger the repeater again. The repeater still works, but it draws a great deal of short-circuit current.

4. $in\!\downarrow$ occurs long enough after $y\!\downarrow$ that $x\!\uparrow$ has already occurred; $x\!\downarrow$ is triggered again, and the repeater generates a second output pulse. The repeater fails.

We may draw an analog connection: the possibilities 0.–3. correspond to the so labeled segments of Figure 3.6 (the part of the curve for possibility 4. is not shown). In normal operation, the repeater is thus operating at the border between possibilities 1. and 2. This is not surprising, since the input pulse is approximately the same length as the resetting pulse on y.

3.2.2.3 The cascaded repeater

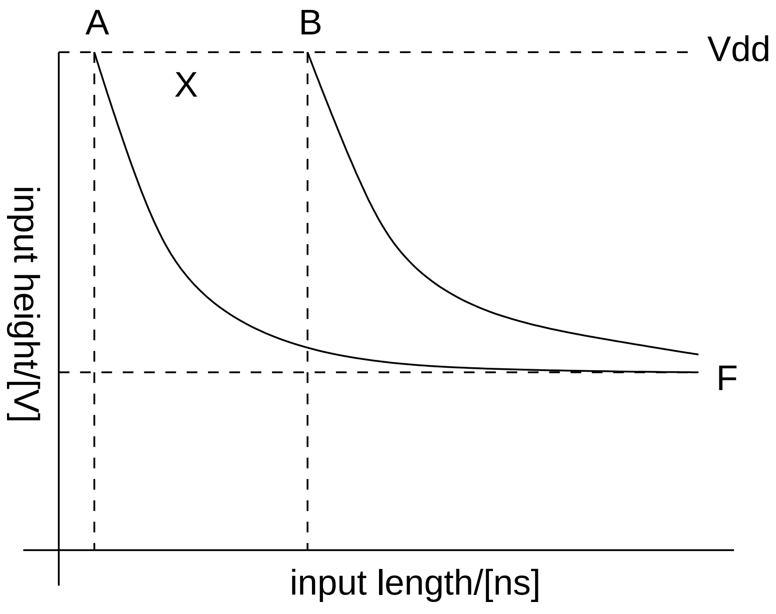

Figure 3.7: Qualitative interpretation of shmoo plots.

Now we shall justify the location of the point marked "X" in Figure 3.7. Is it really true that the output pulse will have the characteristics implied by the location of X, almost regardless of the characteristics of the input pulse? Yes, it is. We can see this[8] from Figure 3.6. This figure shows that, in this fabrication technology, for input pulse lengths in the wide range from 0.12 to 1.47 ns, the output pulse lengths range only from 0.57 to 0.85 ns. (Note that the scale along the abscissa is not the same as that along the ordinate.) Since five transitions take about 0.61 ns here, we can say that in technology-neutral terms, the input pulse lengths may vary from about 1.0 normal transition delays to about 12 delays for an output variation from 4.7 to 7.0 delays.

Since the range of input pulse lengths comfortably contains the range of output pulse lengths, we should have to add considerable load, or we should have to fall victim to extreme manufacturing variations to make the pulse either die out or double up as it travels down a pipeline of pulse repeaters of this kind. Since, further, the input-output relationship of the pulse lengths is almost entirely monotonic, we can summarize the behavior of the pulse repeater thus: an input pulse of length between about 1.0 and 12 transition-delays will generate a single legal output pulse; the length gain averages 4.8.

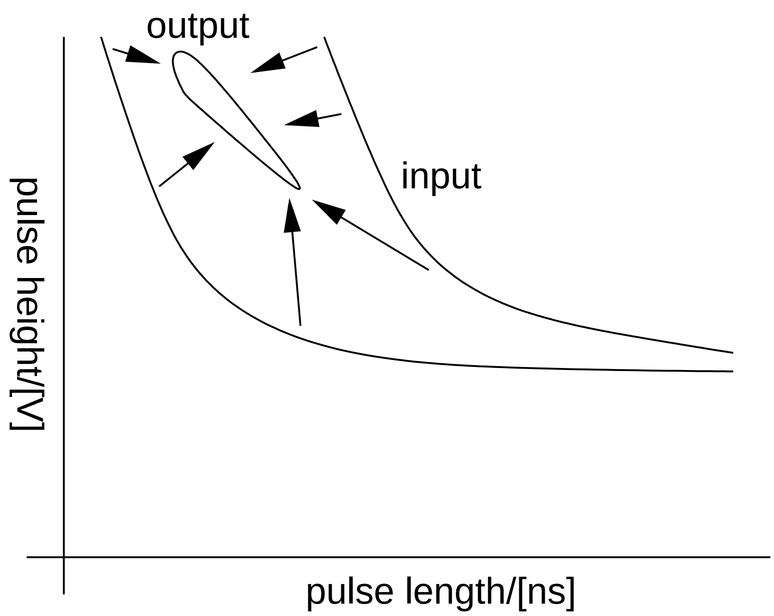

Figure 3.8: Mapping of input to output pulse parameters.

Figure 3.8 is another caricature of the operation of pulsed circuits. The input pulses within the input pipe lead to output pulses within the indicated output region.[9]

[8]Note that the various shmoo plots and the width-gain plot are drawn for several different circuits, so the numerical values are not necessarily directly comparable across them; also the criterion for a pulse's being legitimate is somewhat over-strict in the shmoo plots. We shall formalize the conditions later.

[9]The continuity of physics demands that the output region also goes to infinity where the input pipe does so. This is a nicety that we ignore because, in any case, a pulse repeater operating in this region would be unstable and fickle. Furthermore, as we shall see in Section 3.3.5, allowing the input pipe's being the largest possible for generating properly shaped output pulses will not work, since if we do that, continuity will demand that the output shape

3.2.3 The synchronous digital model

The correctness of synchronous digital logic is justified by a two-part model that is familiar to every electrical engineering undergraduate. The first part explains what it means to be a valid logic-level by dividing the possible analog voltages of a digital circuit into a few ranges with the right properties to guarantee that noise is rejected; this division we call the digital logic-level–discipline,[10] or *logic discipline* for short. The second part introduces a synchronous timing-discipline. The timing discipline can be introduced in several ways, which all rely on defining the times when circuit elements are allowed to inspect the analog voltages (i.e., when they can be proved to obey the logic discipline) and defining the times when the circuit may change the voltages (when the voltages cannot be shown to obey the logic discipline of the model). The timing discipline is normally maintained by introducing latches and a clock and specifying setup and hold times for the latches. Comparing Figure 3.8 with the synchronous logic-discipline, we can identify the big pipe with the legal input-range for a logic value; the little pipe, with the legal output range; the difference between the two is, intuitively speaking, the noise margin.

The synchronization behavior of asynchronous circuits is sufficiently different from the synchronous timing-discipline that we shall have to develop a different timing model. The synchronous logic-discipline, on the other hand, rests on a transitive-closure property of synchronous digital circuits that we may emulate for deriving sufficient conditions for the correctness of APL circuits. In the synchronous world, introducing legal-voltage ranges and noise margins establishes the correctness of the digital model; having introduced these constructs, we can show that voltages that have clear digital interpretations will be maintained throughout a circuit as long as the noise that is present is less than the noise margins [84]. We shall generalize this one-dimensional model for the asynchronous pulses.

3.2.4 Asymmetric pulse-repeaters

We noted above (Section 3.2.2.2) that the pulse repeater normally operates on the border between the "ideal" domain and the "fairly harmless" domain. The reason for this is that, in a long chain of cascaded inverters, the reset pulse on y is about the same length as the input pulse on in.

Practically speaking, there is interference in the "fairly harmless" domain; this means that the circuit generates extra noise and uses more power than necessary. Furthermore, many theoretical difficulties are caused by this interference, as we shall see below. Is there no way of avoiding this?

In fact, it is fairly easy to avoid the in pulse's interfering with the y pulse. What we need is a circuit that generates pulses of different lengths on y and out; the pulse on out needs to be shorter

shall touch the input shape. Hence, the pipe shown in Figure 3.8 is actually a little smaller than the ones shown in Figures 3.4 and 3.5 (by about $1/5$).

[10] The terms "logic discipline," "timing discipline," etc., are taken from Ward and Halstead [84].

than the one on y. An example of a circuit with this behavior is shown in Figure 3.9.

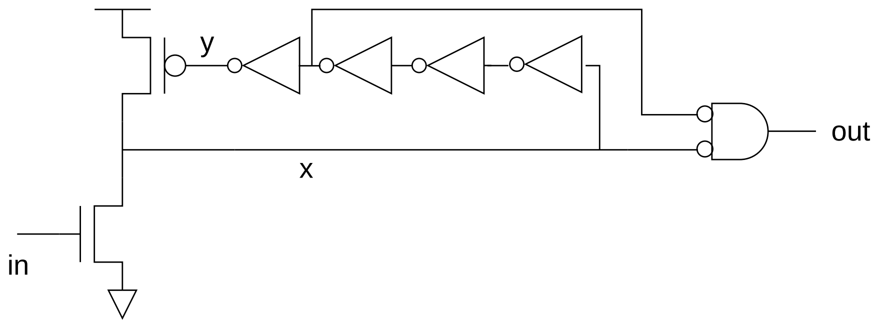

Figure 3.9: Asymmetric 3-5–stage pulse repeater.

We shall not mention these circuits further, except now and then to lament that we should be lucky to be blessed with their non-interference. The theory of these circuits may be simpler[11] and the power consumption lower; but the output pulse is shorter and driven by a more complicated gate (hence weaker), and there are two stages of logic that we can no longer use as fully as before. For these reasons, we should probably have to stretch the timing of this pulse repeater to 5-7 transitions instead of 3-5. The losses would outweigh the gains.

3.3 Formal model of pulse repeater

Formally, we may divide what we are doing with the pulse-repeater problem into three steps: our generating input excitations, the circuit's reacting to the input excitations by producing output waveforms, and our measuring of the output waveforms. The question we should like to ask is if it is possible to combine several pulse repeaters, i.e., if cascading the pulse repeaters will maintain the pulse shape.

We shall study the behavior of a pulse repeater when presented with a single input pulse; we shall not directly consider a single pulse repeater's being presented with several pulses in succession.

3.3.1 Basic definitions

The waveforms that we saw in the pulse repeater experiments are parameterized by two parameters, the voltage V and the time t; for the particular parameterization we chose, it made sense to speak of the height h and the length l. For instance, let us assume that we would use rectangular pulses

[11] But in the light of the following theory, which establishes that the symmetric pulse repeaters work, the suggested asymmetric-repeater theory might be considered dishonest. If the symmetric repeaters work despite interference, would not the asymmetric ones too? And would not the asymmetric-repeater designer be tempted to allow his circuits' operating in this domain?

as excitations for characterizing the behavior of some pulsed circuit. We could then parameterize the pulses that we use as[12]

$$p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}(t) \stackrel{\text{def}}{=} h \times \big(I(t) - I(t-l)\big), \tag{3.1}$$

where $I(x)$ is the unit step function [76]:

$$I(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } x \geq 0. \end{cases} \tag{3.2}$$

We first define the set $\mathcal{T}$ of functions over time that have their global maximum in finite time, i.e.,

$$\mathcal{T} \stackrel{\text{def}}{=} \{f : t \to V \wedge \big(\exists k, F :: (\forall x : |x| > k : f(x) < F) \wedge (\exists x :: f(x) > F)\big)\}. \tag{3.3}$$

The need for the restriction is explored in Appendix C.

We say that two functions[13] $f, g : \mathbf{R} \to \mathbf{R}$ are *equivalent under translation* if there exists a $\Delta \in \mathbf{R}$ such that $f(t) = g(t - \Delta)$ for some Δ; we write this $f \sim g$. We shall mainly deal with the partition of $\mathcal{T}$ into equivalence classes under translation; we call this partition $\mathcal{F}$. The equivalence class that contains f we write $\tau(f)$.

We can think of the p waveforms as a mapping from pairs of real numbers, i.e., members of the set

$$\mathcal{P} \stackrel{\text{def}}{=} V \times t, \tag{3.4}$$

to functions in $\mathcal{F}$. The mapping itself, $\mathbf{P} : \mathcal{P} \to \mathcal{F}$, is defined as

$$\mathbf{P}\left(\begin{bmatrix} h \\ l \end{bmatrix}\right) \stackrel{\text{def}}{=} p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}; \tag{3.5}$$

we write $\mathbf{P}(\mathcal{P})$ for the subset of $\mathcal{F}$ that represents *all* such rectangular pulses.

Secondly, the circuit's reaction to the input may again be thought of as a mapping, this time from input to output functions. We may write this $\Phi : \mathcal{T} \to \mathcal{T}$; interpreting a translation of the input as causing an identical translation of the output (the circuits themselves are of course time-invariant), we may also write

$$\Phi : \mathcal{F} \to \mathcal{F}. \tag{3.6}$$

Lastly, we consider measuring the output waveforms. Naïvely speaking, we want to characterize the output waveforms in the simplest way; since we use $\mathcal{P}$ to parameterize the inputs, it would seem convenient to extract the same parameters of the output waveforms. Perhaps we can phrase the

[12] We use the vector notation $\left[\begin{smallmatrix} j \\ k \end{smallmatrix}\right]$ mainly for making things typographically clear; the functions we are considering are generally nonlinear, whence the analogy with matrices and linear transformations is not so useful.

[13] If nothing to the contrary is stated, then we shall assume that a function f maps from real numbers to real numbers.

questions about the pulses in terms of the parameters of the pulses.

3.3.2 Handling the practical simulations

If we want to determine the mapping for a real circuit, we should run a set of simulations to check the behavior of our circuit for a variety of input pulses of the shape $p_{\left[\begin{smallmatrix} h_i \\ l_i \end{smallmatrix}\right]}$, with $i = 0, 1, 2, \ldots$. In order to determine the mapping of Figure 3.8, we measure the length and height of the output pulses. This we could do, for instance, by measuring the highest voltage recorded by SPICE and measuring the time interval between the upward crossing of $Vdd/2$ and the following downward crossing. (If we should not detect any crossings of $Vdd/2$, or if we should detect more than these two crossings, we should be able to conclude that the input pulse was outside the "pipe.") With these measurements in hand, we could then draw Figure 3.8, and if the area covered by the output pulses should be contained within that of the input pulses, then we could finally conclude that the circuit we have tested can be used as a pulse repeater because the input-output relationship is stable.

In other words, we have the commutative diagram:

$$\begin{array}{ccc} \begin{bmatrix} h \\ l \end{bmatrix} & \xrightarrow{G} & G\left(\begin{bmatrix} h \\ l \end{bmatrix}\right) \\ \big\downarrow \mathbf{P} & & \big\uparrow \mathbf{P}^{-1} \\ p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]} & \xrightarrow{\Phi} & \Phi\left(p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}\right) \end{array} \tag{3.7}$$

The upper part of the diagram defines the function $G : \mathcal{P} \to \mathcal{P}$ on the parameter space. In practice, we study G by computing $\mathbf{P}^{-1} \circ \Phi \circ \mathbf{P}$. We ultimately want to know if $\mathbf{P}^{-1} \circ \Phi^n \circ \mathbf{P}$ converges as $n \to \infty$; we can now ask the same thing of G^n.

Unfortunately, there is an important flaw in the described testing procedure. While it is true that the input, $p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}(t)$, is properly characterized by h and l, and that our measuring the height and length of the input pulses would indeed result in the values we specified in the definition of p, our measurements of the output pulses only incompletely characterizes them. This is understandable: even though a simple mathematical function describes the input pulses, the waveforms of the output are unlikely to obey some $p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}(t)$ for any values of h and l. Indeed we should be surprised if they did, given that there is an infinite variety of possible output waveforms, depending on circuit parameters, environmental conditions, noise, etc. In fact, we know that the output pulses cannot possibly be characterized by $p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}$ since the function p has discontinuities, whereas—given the nonzero capacitances that are present—the output voltage cannot have discontinuities if all the currents in the circuit are finite. Formally, we should say that $\mathbf{P}^{-1}$ only exists for members of $\mathbf{P}(\mathcal{P})$, not for arbitrary functions.

3.3.3 Expanding the model

We instead define a partial order on functions in the obvious way:

Definition 3.1 (Partial ordering of functions in $\mathcal{T}$) *Given $\theta, \phi \in \mathcal{T}$, we say that $\theta \leq \phi$ if and only if $\theta(t) \leq \phi(t)$ for all t. We say that θ equals ϕ if $\theta \leq \phi$ and $\phi \leq \theta$;*

and extend it to our translation invariant representatives in $\mathcal{F}$:

Definition 3.2 (Partial ordering of members of $\mathcal{F}$) *Given $f, g \in \mathcal{F}$, we say that $f \leq g$ if and only if there exist representatives $\theta \in f$ and $\phi \in g$ (with $\theta, \phi \in \mathcal{T}$) such that $\theta \leq \phi$.*

Appendix C shows that Definition 3.2 establishes a partial order (i.e., a relation that is transitive, reflexive, and anti-symmetric) of functions obeying (3.3) if by two functions' being equal we mean that they are members of the same equivalence class in $\mathcal{F}$.

If we consider a member $f \in \mathcal{F}$ (f is an equivalence class of functions; we can loosely speak of it as a function if we by that mean some canonical representative), we can define the mapping taking it to the subset of rectangular pulses $\mathbf{P}(\mathcal{P})$ dominated by it as

$$\mathbf{J} : \mathcal{F} \to 2^{\mathbf{P}(\mathcal{P})}; \tag{3.8}$$

a $g \in \mathbf{P}(\mathcal{P})$ is also in $\mathbf{J}(f)$ if and only if $g \leq f$.

Similarly, we define the subset of $\mathbf{P}(\mathcal{P})$ that dominates f as $\mathbf{K}(f)$. Figure 3.10 illustrates the situation as it applies to an arbitrary waveform when $\mathcal{P}$ consists of the rectangular functions $p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}(t) = h \times \big(I(t) - I(t-l)\big)$.[14]

As we know, an arbitrary f is most unlikely to be in $\mathbf{P}(\mathcal{P})$. Hence we shall not attempt to define a mapping directly from $\mathcal{F}$ to $\mathcal{P}$; instead, we define a mapping $\mathbf{M} : \mathcal{F} \to \mathbf{P}(\mathcal{P}) \times \mathbf{P}(\mathcal{P})$. ($\mathbf{M}$ stands for "measuring mapping.") Starting from $\mathbf{J}(f)$ and $\mathbf{K}(f)$, $\mathbf{M}$ picks an ordered pair $\left[\begin{smallmatrix} j \\ k \end{smallmatrix}\right]$ such that $j \in \mathbf{J}(f)$ and $k \in \mathbf{K}(f)$—in other words some pair $\left[\begin{smallmatrix} j \\ k \end{smallmatrix}\right]$ such that $j \leq f$ and $f \leq k$; we further write $\mathbf{M}_\alpha$ for j and $\mathbf{M}_\beta$ for k; we can also define $M : \mathcal{F} \to \mathcal{P} \times \mathcal{P}$ by $M \stackrel{\text{def}}{=} \left[\begin{smallmatrix} \mathbf{P}^{-1}(\mathbf{M}_\alpha) \\ \mathbf{P}^{-1}(\mathbf{M}_\beta) \end{smallmatrix}\right]$. It is clear that j is somewhat arbitrary (m satisfies the same condition as j), but there is no reason for choosing k different from the corner of the square region. Obviously, $\left[\begin{smallmatrix} j \\ k \end{smallmatrix}\right]$ is not unique, so $\mathbf{M}$ is not unique; as we shall see, picking $\mathbf{M}$ properly is important.[15]

Let us now define Φ to work on sets of functions as well as on functions. We define $\Phi : 2^{\mathcal{F}} \to 2^{\mathcal{F}}$:

$$\Phi(\mathcal{S}) \stackrel{\text{def}}{=} \{t : \big(\exists s : s \in \mathcal{S} : t = \Phi(s)\big)\} \tag{3.9}$$

[14] In order that we may keep the exposition manageable, we have simply truncated all the waveforms at the threshold voltage—any activity below the threshold we have assumed to be negligible.

[15] Figure 3.10 applies to the particular case when $\mathbf{P}(\mathcal{P})$ consists of the rectangular functions $p_{\left[\begin{smallmatrix} h \\ l \end{smallmatrix}\right]}$. Here it is clear that there is no reason for picking a k different from the vertex of the square area. Which j is best is a different story; picking the j that maximizes the product $h \times l$ is likely a good heuristic. In the general case when a more complicated $\mathbf{P}(\mathcal{P})$ is used, k can also be open to question. We might then choose the $k \in \mathbf{K}(f)$ that minimizes $\int k(t)\,dt$ and the $j \in \mathbf{J}(f)$ that maximizes $\int j(t)\,dt$.

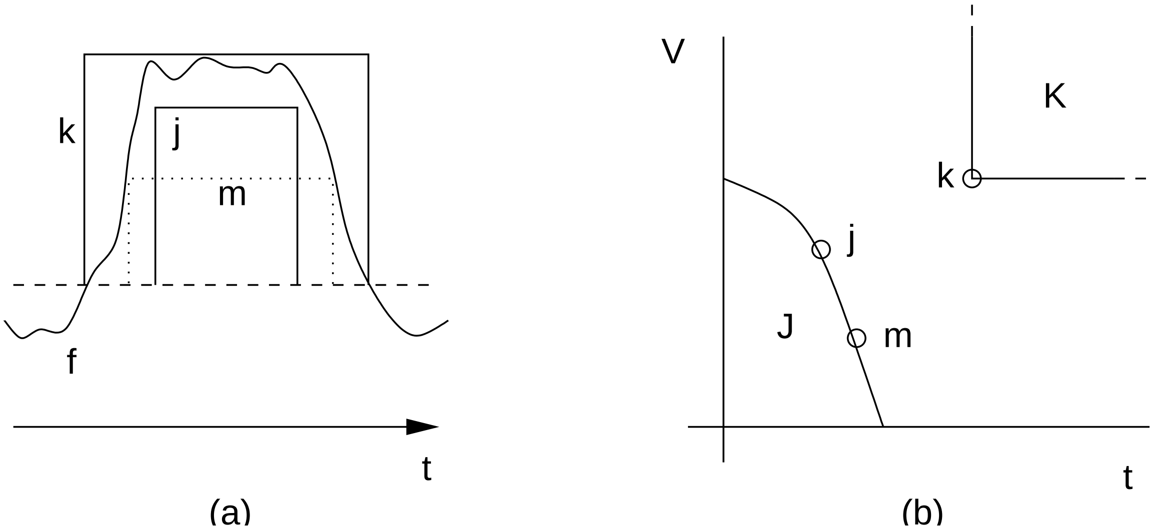

Figure 3.10: (a) the function f and two members $j, k \in \mathbf{P}(\mathcal{P})$. Here $j \leq f \leq k$. (b) parameter-space representation of sets $\mathbf{J}(f)$ and $\mathbf{K}(f)$ and the points j and k (more properly $\mathbf{P}^{-1}(j)$ and $\mathbf{P}^{-1}(k)$) picked by $\mathbf{M}$.

Similarly, we may define $\mathbf{M}$ on sets, so that $\mathbf{M} : 2^{\mathcal{F}} \to \mathbf{P}(\mathcal{P}) \times \mathbf{P}(\mathcal{P})$; thus,

$$\mathbf{M}(\mathcal{S}) \stackrel{\text{def}}{=} \begin{bmatrix} \mathbf{M}_\alpha\big(\text{minf}(\mathcal{S})\big) \\ \mathbf{M}_\beta\big(\text{maxf}(\mathcal{S})\big) \end{bmatrix}, \tag{3.10}$$

where minf : $2^{\mathcal{F}} \to \mathcal{F}$, with $[\text{minf}(\mathcal{S})](x) \stackrel{\text{def}}{=} [\inf_{s \in \mathcal{S}} s](x)$, where we use Definition 3.2 for $\leq$ and the corresponding inf; and analogously for maxf. Finally, let us define the map $\mathbf{F} : \mathbf{P}(\mathcal{P}) \times \mathbf{P}(\mathcal{P}) \to 2^{\mathcal{F}}$ that generates *all* functions between two reference pulses j, k; hence,

$$\mathbf{F}\left(\begin{bmatrix} j \\ k \end{bmatrix}\right) \stackrel{\text{def}}{=} \{f : j \leq f \leq k\}. \tag{3.11}$$

It should be clear that $\mathbf{M} \circ \mathbf{F}\left(\left[\begin{smallmatrix} j \\ k \end{smallmatrix}\right]\right) = \left(\left[\begin{smallmatrix} j \\ k \end{smallmatrix}\right]\right)$ as long as $j \leq k$; hence we could define

$$\mathbf{M}^{-1} \stackrel{\text{def}}{=} \mathbf{F}, \tag{3.12}$$

but we must then keep in mind that $\mathbf{M}^{-1} \circ \mathbf{M}(\mathcal{S})$ for a set of functions $\mathcal{S}$ does not necessarily equal $\mathcal{S}$; we do however have that

$$\mathcal{S} \subseteq \mathbf{M}^{-1} \circ \mathbf{M}(\mathcal{S}). \tag{3.13}$$

3.3.4 Using the extended model

Now we can ask questions about the behavior of the parameterized waveforms when Φ is iterated. If we consider $l_f \stackrel{\text{def}}{=} \lim_{n\to\infty} \Phi^n(f)$, we know by (3.13) that $l_f \in \lim_{n\to\infty} \mathbf{M}^{-1}(\mathbf{M}\circ\Phi\circ\mathbf{M}^{-1})^n(\mathbf{M}f)$. We hence define the mapping from input to output in terms of the reference pulses as $\mathbf{G} : \mathbf{P}(\mathcal{P})\times\mathbf{P}(\mathcal{P}) \to \mathbf{P}(\mathcal{P}) \times \mathbf{P}(\mathcal{P})$

$$\mathbf{G_M} \stackrel{\text{def}}{=} \mathbf{M}\circ\Phi\circ\mathbf{M}^{-1}; \tag{3.14}$$

if we should prefer considering the behavior in the parameter space $\mathcal{P}$, we can write $G : \mathcal{P}\times\mathcal{P} \to \mathcal{P}\times\mathcal{P}$ as

$$G_{\mathbf{M}}\left(\begin{bmatrix} p \\ q \end{bmatrix}\right) \stackrel{\text{def}}{=} \begin{bmatrix} \mathbf{P}^{-1}\circ\mathbf{M}_\alpha\circ\Phi\circ\mathbf{M}^{-1}\left(\begin{bmatrix}\mathbf{P}(p)\\ \mathbf{P}(q)\end{bmatrix}\right) \\ \mathbf{P}^{-1}\circ\mathbf{M}_\beta\circ\Phi\circ\mathbf{M}^{-1}\left(\begin{bmatrix}\mathbf{P}(p)\\ \mathbf{P}(q)\end{bmatrix}\right) \end{bmatrix}. \tag{3.15}$$

The $\mathbf{M}$ subscripts serve to remind that there is some arbitrariness in the choice of $\mathbf{M}$, which infects G and $\mathbf{G}$ too. Thus we have soundly fixed the commutative diagram, (3.7); we should write:

$$\begin{array}{ccc} \begin{bmatrix} p \\ q \end{bmatrix} & \xrightarrow{G_{\mathbf{M}}} & G_{\mathbf{M}}\left(\begin{bmatrix} p \\ q \end{bmatrix}\right) \\ \Big\downarrow \mathbf{M}^{-1}\circ\mathbf{P} & & \Big\uparrow \begin{bmatrix}\mathbf{P}^{-1}\circ\mathbf{M}_\alpha \\ \mathbf{P}^{-1}\circ\mathbf{M}_\beta\end{bmatrix} \\ \{f : \mathbf{P}(p) \le f \le \mathbf{P}(q)\} & \xrightarrow{\Phi} & \{g : \left(\exists f : \mathbf{P}(p) \le f \le \mathbf{P}(q) : g = \Phi(f)\right)\} \end{array} \tag{3.16}$$

We can think of $\begin{bmatrix} j \\ k \end{bmatrix}$ as defining a rectangle[16] in $\mathcal{P}$-space, whence we may have:

Definition 3.3 (Stable function mapping) *We say that G is* stable *under* $\mathbf{M}$ *if there exists a rectangle* $\begin{bmatrix} l \\ m \end{bmatrix}$ *such that* $G_{\mathbf{M}}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right) \subset \begin{bmatrix} l \\ m \end{bmatrix}$. *We call* $\begin{bmatrix} l \\ m \end{bmatrix}$ a region of stability *of G (and by extension of Φ).*

The connection with pulse repeaters should be clear. We find that if we can arrange that the inputs x_i to a chain of pulse repeaters will obey $\mathbf{M}(x_i) \subset \begin{bmatrix} l \\ m \end{bmatrix}$, then all nodes along the chain will also obey that relation.

In topological terms, the stable G's causing pulses to remain well-behaved is a weak application of the "contraction principle" used by Gamelin and Greene [27]; if we could guarantee that for *all* $\begin{bmatrix} l \\ m \end{bmatrix}$ in a region of the plane, we could define a metric d measuring $\begin{bmatrix} l \\ m \end{bmatrix}$ such that it decreases for each iteration of Φ, then we should have a true *contraction mapping,* in which case the pulse would converge to a single well-defined shape as it travels down the chain of pulse repeaters. This very often happens in practice, but as should be clear from our argument, so strong a property is not required for the pulses' remaining well-defined.

[16] When $\mathcal{P}$-space is two-dimensional; it should be clear that $\mathcal{P}$-space could have any desired dimensionality, and $\begin{bmatrix} j \\ k \end{bmatrix}$ generally defines a coordinate-aligned rectangular hyperprism. The rest of the argument is the same for any dimensionality of $\mathcal{P}$. We might for instance handle the threshold-voltage issue (see footnote on p. 28) by adding an extra parameter to the $\mathcal{P}$-space.

What have we gained—why should $\mathbf{M}\circ\Phi\circ\mathbf{M}^{-1}$ be easier to handle than Φ itself? One situation when it is easier to handle G is when Φ is locally monotonic, i.e., if it is true that $\forall f, g : j \le f \le g \le k : \Phi(f) \le \Phi(g)$. If this is true, then it is also true that $\Phi \circ \mathbf{M}^{-1}\left(\begin{bmatrix} j \\ k \end{bmatrix}\right) \subset \mathbf{M}^{-1}\left(\begin{bmatrix} \Phi(j) \\ \Phi(k) \end{bmatrix}\right)$, which means that we need only operate on *pairs* of functions j and k to determine the boundaries of the region of stability (rather than on the infinite sets of functions $\mathbf{M}^{-1}\left(\begin{bmatrix} j \\ k \end{bmatrix}\right)$); i.e., our naïve pulse-repeater experiments then carry enough information for determining whether a given pulse is a legal input pulse to the circuit.

3.3.5 Noise margins

If we consider a region of stability $\begin{bmatrix} l \\ m \end{bmatrix}$ of Φ, we know that for any input signal f that satisfies $\mathbf{M}(f) \subset \begin{bmatrix} l \\ m \end{bmatrix}$, it is true that $\mathbf{M}\circ\Phi(f) \subset \begin{bmatrix} l \\ m \end{bmatrix}$. If the region of stability is finite (which it normally is), then there exists at least one maximal region of stability $\begin{bmatrix} l_{\max} \\ m_{\max} \end{bmatrix}$, which has the property that there is no larger region of stability $\begin{bmatrix} r \\ s \end{bmatrix} \supset \begin{bmatrix} l_{\max} \\ m_{\max} \end{bmatrix}$. It can be proved that $\begin{bmatrix} l_{\max} \\ m_{\max} \end{bmatrix}$ allows at least one input function f that differs only infinitesimally from an input function g that would take the circuit out of the region of stability. Hence, if we allow as legal any pulse for which $\mathbf{M}(f) = \begin{bmatrix} l_{\max} \\ m_{\max} \end{bmatrix}$, then the circuit's noise margin will be zero. (In traditional synchronous logic, this would be equivalent to considering as a legal digital input one that is exactly at the switching threshold.)

If we instead define a norm on the noise margin, i.e., on the function-set difference $\mathbf{M}^{-1}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right) - \Phi\left(\mathbf{M}^{-1}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right)\right)$, we can say that the $\begin{bmatrix} l \\ m \end{bmatrix}$ that we should choose as our legal range of pulse inputs is the one that maximizes $\left\|\mathbf{M}^{-1}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right) - \Phi\left(\mathbf{M}^{-1}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right)\right)\right\|$. The legal range of pulse outputs is the corresponding $\Phi\left(\mathbf{M}^{-1}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right)\right)$. We define the *noise margin*

$$\mu = \left\|\mathbf{M}^{-1}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right) - \Phi\left(\mathbf{M}^{-1}\left(\begin{bmatrix} l \\ m \end{bmatrix}\right)\right)\right\|. \tag{3.17}$$

Choosing this norm properly is likely a difficult matter, although simplistic versions are not too hard to come up with. Saying more about noise here would be premature; how we treat noise depends to a large extent on what we shall do with pulsed circuits. The interested reader is referred to Section 6.8.

3.4 Differential-equations treatment of pulse repeater

How do we find out if Φ is locally monotonic, as would make the previous section's results applicable?

First, we shall reiterate (almost verbatim) the definition of a *strong upper fence* given by Hubbard and West [38].

Definition 3.4 (Strong upper fence) *For the differential equation $x' = f(t, x)$, we call a con-*

tinuous and continuously differentiable function $\beta(t)$ *a* strong upper fence *over the interval* $\mathbf{I}$ *if* $f\big(t,\beta(t)\big) < \beta'(t)$ *for all* $t \in \mathbf{I}$.

Theorem 3.1 *If* $\beta(t)$ *is a strong upper fence on* $x' = f(t,x)$, *then for any solution* $u(t)$ *with* $u(t_0) \leq \beta(t_0)$, $u(t) < \beta(t)$ *for all* $t > t_0$ *in any interval contained in* $\mathbf{I}$ *where* $u'(t) = f\big(t,u(t)\big)$.

Proof. See Hubbard and West.

Let us now consider the boundary value problem of a differential equation in $x(t)$ of the type

$$\frac{dx}{dt} = f\big(t,x,w(t)\big), \tag{3.18}$$

with boundary condition

$$x(0) = x_0. \tag{3.19}$$

To save the reader's patience, let us right away call t the *time,* w the *excitation,* and x the *response.*

Without loss of generality, define $\mathbf{I}$ so that $\mathbf{I} = \{\, t : 0 \leq t \leq T \,\}$. Furthermore, let it be the case that over the entire domain of f, an increased excitation tends to drive the response downward; i.e., formally, let everywhere

$$\frac{\partial f}{\partial w} < 0. \tag{3.20}$$

Now we should like to characterize the behavior of the solutions to the boundary value problem (3.18) for different choices of the excitation function w. Let us specifically have in mind two excitations w_ϕ and w_θ such that

$$w_\phi(t) > w_\theta(t) \text{ for all } t \in \mathbf{I}; \tag{3.21}$$

we shall for convenience refer to the responses given these excitations as x_ϕ and x_θ. Then:

Lemma 3.1 $x_\theta(t)$ *is a strong upper fence for the differential equation* $dx/dt = f\big(t,x(t),w_\phi(t)\big)$.

Proof. From Theorem 3.1, we know that we need to show that

$$f\big(t,x_\theta(t),w_\phi(t)\big) < \frac{dx_\theta}{dt}. \tag{3.22}$$

But we also know that $x_\theta(t)$ solves the boundary value problem (3.18) for the excitation $w_\theta(t)$, so that we have to prove

$$f\big(t,x_\theta(t),w_\phi(t)\big) < f\big(t,x_\theta(t),w_\theta(t)\big). \tag{3.23}$$

This relationship is obvious from (3.20) and (3.21). Therefore $x_\theta(t)$ is a strong upper fence for $dx/dt = f\big(t,x(t),w_\phi(t)\big)$. *Q.E.D.*

We can now state the desired relationship between x_ϕ and x_θ.

Theorem 3.2 *If $w_\phi(t) > w_\theta(t)$ for all $t \in \mathbf{I}$, then $x_\phi(t) \le x_\theta(t)$ for all $t \in \mathbf{I}$.*

Proof. Recall that $x_\phi(0) = x_\theta(0) = x_0$. Thus, in Theorem 3.1, $u(0) = \beta(0)$, and therefore $x_\phi(t) \le x_\theta(t)$ for all $t \in \mathbf{I}$. *Q.E.D.*

3.4.1 Input behavior of pulse repeater

Figure 3.11 illustrates the input circuitry of a pulse repeater. We shall discuss the time behavior of the node x as an input pulse arrives. The input pulse may have any shape.

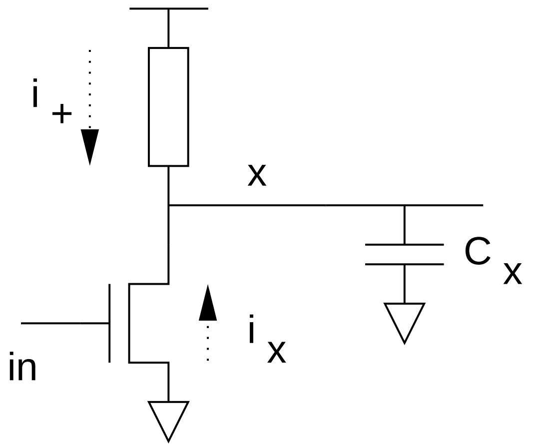

Figure 3.11: Input circuitry of a pulse repeater.

If we assume that the circuit starts out with a known voltage on the node x at time t_a, $v_x(t_a)$, we may write

$$v_x(t)' = \frac{1}{C_x}\big(i_x(t) + i_+(t)\big), \tag{3.24}$$

or in integral form, replacing the boundary condition with the term $v_x(t_a)$,

$$v_x(t) = \frac{1}{C_x}\int_{t_a}^{t} i_x(t) + i_+(t)\,dt \;+\; v_x(t_a). \tag{3.25}$$

Since the input pulse ends at some time—call it t_b—we are mainly concerned with the value of the right-hand-side expression at time t_b. The relationship between i_x and v_x follows from the transistor equations and the shape of v_{in}, but what do we know about $i_+(t)$?

3.4.1.1 Inertial approximation

Let us first assume that we can model the behavior of the circuit as "inertial"; i.e., we will consider only the behavior of the pulse repeater in situations when the input pulse has not yet acted so long that i_+ has been affected via the feedback path (y in Figure 3.1) by the change in x. Understanding exactly where this is true would involve understanding exactly when Equation 3.20 with the proper variable renamings holds, i.e., when it is true that

$$\frac{\partial(i_+ + i_x)}{\partial v_{in}} < 0. \tag{3.26}$$

We intuitively justify the approximation by noting that (3.26) holds as long as y (the input to the pullup p-transistor, see Figure 3.1) is approximately Vdd. That this is true for a while can be seen from the transistor equations. We can for instance use the familiar Sah model to model the transistors [73, 2]. Under this model, since the p-transistor is in the forward active region,[17] and the n-transistor is in saturation during most of the beginning of the input pulse, we may write

$$i_+ = k_p(V - V_{Tp})(V - v_x) - \frac{(V - v_x)^2}{2} \tag{3.27}$$

and

$$i_x = -k_n(v_{in} - V_{Tn})^2. \tag{3.28}$$

Now we verify (3.26) for this model:

$$\frac{\partial(i_+ + i_x)}{\partial v_{in}} = -2k_n v_{in} \tag{3.29}$$

The partial derivative (3.29) is certainly negative as long as v_{in} is positive, as it must be under our assumption that the transistor is in saturation. This may not be accurate for all input pulses, so we must re-check the conditions in the linear region; this work is not shown.

We may identify $\frac{1}{C_x}\big(i_x(t) + i_+(t)\big)$ with w in (3.18). Hence, as long as (3.26) holds, we can apply the theorem and conclude that if we have three pulses $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$ and $\mathbf{p} < \mathbf{q} < \mathbf{r}$ and both $\mathbf{p}$ and $\mathbf{r}$ are legal input pulses for a pulse repeater, then so is $\mathbf{q}$.

3.4.1.2 Non-inertial behavior

But is it really true that we can ignore the feedback path that could affect i_+? Let us slightly refine the model for the pulse repeater. Basing our argument on Figure 3.1, we can see that the output inverter is used for two purposes: first, it generates the output; and secondly, it provides a feedback path, which eventually resets the output, thus generating a pulse. If we break up the two functions,

[17] Also called linear region, triode region.

we have the circuit of Figure 3.12 instead. Here we have modeled the feedback path, which creates the pulse, as an inverting inertial delay called **D**. If the amount of delay introduced is Δ, we should normally write $y(t) = Vdd - x(t-\Delta)$. Writing y thus would however neglect the gain of the inverter or inverters on the delay path (two in the case of the three-stage repeater, four in the case of the five-stage repeater). Hence we shall instead write

$$y(t) = C\big(x(t-\Delta)\big), \tag{3.30}$$

where C represents the "inverting clamping function" that is Vdd if the input is less than $Vdd/2$ and is 0 if the input is more; we thus assume that the gain along the delay path is enough that we can consider y a purely digital signal. (In other words, the gain is infinite.) We further assume that the output inverter's logic threshold is at least as low as the inertial delay's (i.e., the output inverter will not trigger before the inertial delay).

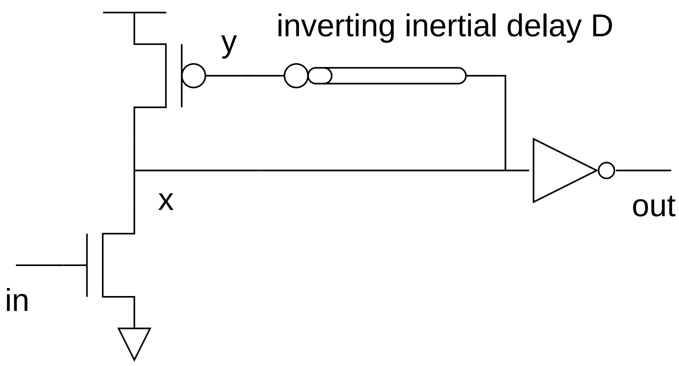

Figure 3.12: Pulse repeater modeled with inverting inertial delay.

Can we get away without the special inertial-delay element? Unfortunately not: the length of the output pulse will be approximately Δ long; if that is also the length of the input pulse, then it means that the current i_+ will begin changing approximately at the same time as the input pulse ends; we should hence be concerned that the monotonicity condition may be violated for input pulses longer than Δ; this corresponds approximately to the later half of the "pipe," at least near its top.

Consider the two input-pulses in the scenarios A and B of Figure 3.13 (as usual, we have ignored the irrelevant time-translation of the various signals). The pulse repeater whose behavior we are plotting must be avoided because it misbehaves: the weaker input-pulse causes a stronger output-pulse. Both A and B are slightly longer than Δ; hence there will be some interference on the node x during the later parts of the input pulses. The pulse in A is slightly "weaker" than in B during the first Δ, as shown by the notch at the top; we have assumed in the x waveform that the weak

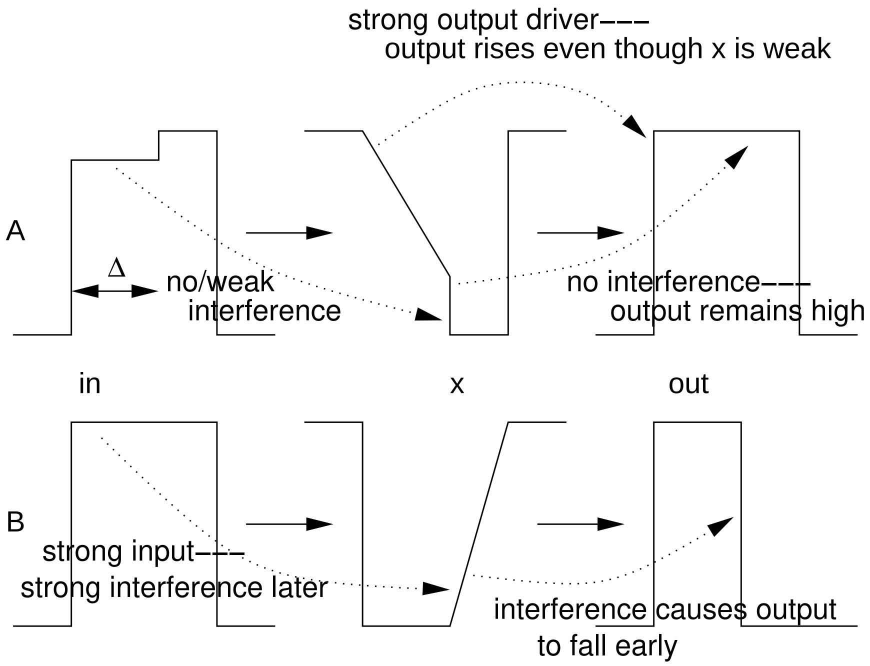

Figure 3.13: Two different input-pulse scenarios and their corresponding intermediate-node values and output values.

in_{A} merely causes a slow transition.

If we first consider, as in Figure 3.13, a circuit whose inertial delay has no gain, it is clear that the monotonicity property is in danger. This is because if $y(t)$ merely mirrors $x(t)$, then the interfering i_+ will be less for scenario A than it will be for scenario B. Hence, in scenario A, the signal on x could actually have a larger (more negative) swing than in scenario B, and the ultimate output-pulse on node *out* could be larger than in scenario B.

Now let us see how our assumptions about **D** solve the problem. There are two possibilities we need to consider:

I. First, the pulse in_{A} may be so weak that it does not trigger **D** until much later than the pulse in scenario B. But we said that if **D** is not triggered, then the output inverter is not either, and we know that the output pulse will then *start later* than in scenario B.

II. Secondly, the pulse in_{A} may be strong enough that it triggers **D** immediately. But since **D** has infinite gain, i_+ will now be the same in the two scenarios.

This argument shows that monotonically stronger input-pulses will still cause monotonically stronger output-pulses, even in the presence of interference when the input pulses are more than Δ long. (The reader may wish to verify that our argument with respect to possibility I. holds as long as the input pulses are no longer than approximately 2Δ.)

3.4.2 Generalizations and restrictions

Because of the simple requirements of the Fence Theorem, Theorem 3.1, we should find it easy to verify the fence conditions for (3.24) even for a complicated transistor model, because all that is required is that the circuit's response—the instantaneous current flowing to x—is strictly antimonotonic in the input voltage.

Of course, physics is never quite as simple as we have made it seem. First, we have used a very simplistic model; the behavior of the cascaded inverters is only imperfectly captured by an infinite-gain inertial delay. Secondly, the data in the pipe figures was captured with a rudimentary technique that does not directly correspond to the theory we have developed. Both of these are reasons for mistrusting some aspects of the pipe figures; especially the data around the edges of the pipes are suspect. Lastly, we may expect that the pulse-logic designer loses sleep over not being able to verify that his circuit satisfies the fence condition (3.20) at all times. If $\partial f/\partial w$ can sometimes be positive (e.g., owing to second-order effects in the transistors; the MOS Miller effect comes to mind), but this deviation from the fence condition is small, then how does the designer prove that the required monotonicity-property for the pulses still holds? We could perhaps save the day with more detailed analysis. In practice, we could also use differently shaped test signals to eliminate or minimize the unwanted behavior of $\partial f/\partial w$.[18]

[18] Note that the partial derivative is with respect to changes in w from one waveform to another, *not* with respect to changes in w over time in a single waveform; this means that although we cannot avoid the Miller effect's suggesting that dx/dt can sometimes be positive in response to a positive dw/dt, we may conjecture that we shall still be able to shape the input pulses so as to keep $\partial f/\partial w < 0$.

This page intentionally left blank.

Chapter 4

Computing With Pulses

My pulse, as yours, doth temperately keep time

— *William Shakespeare,* Hamlet, Prince of Denmark *(1603)*

As undergraduates, we learned that while important, the logic discipline alone does not suffice for building digital systems that shall be able to compute arbitrary functions repeatedly. Two further attributes are required before a circuit-design method can be used for implementing a wide range of digital systems: *timing discipline*—in essence, design properties allowing the reuse of circuitry (i.e., as repetitive systems); and *logic*—our circuits' ability of computing arbitrary boolean functions.

In traditional synchronous design, simplicity requires treating timing discipline and logic separately; this approach leads to design styles that alternate "latching elements" with "combinational logic." This is taken to an extreme in automated design systems: the latching elements are special library cells and no feedback paths whatsoever are allowed in the logic—a formally complete separation of logic and timing.

Over the years, however, designers of high-performance synchronous systems have begun investigating techniques that more and more mix logic and timing. We see a familiar example of this in "precharge logic" or "domino logic." Precharge logic is more difficult to design because of the mixture of timing and logic; the precharge-logic designer must renounce attempting to implement logical specifications that do not fit in the precharge timing-discipline. The mixing trend has not abated over the years; currently, "self-resetting" (or "post-charged") logic [71] is an active area of research in synchronous systems—this type of logic is in many ways the synchronous parallel to the asynchronous pulse-logic that is the subject of this thesis. Self-resetting synchronous circuits are indeed asynchronous to the extent that some of their timing is data driven. Unfortunately, using the clock for synchronizing these circuits globally, albeit less frequently, still spoils the broth. Designers of self-resetting synchronous circuits hence face the worst of both worlds: the system-design difficulties of synchronous systems and the circuit-design difficulties of asynchronous circuits.

Asynchronous-circuit design-styles inherently require that the logic computations carry timing information. This can mean either that the computations explicitly carry timing information (as in QDI or speed-independent design) or that implicit timing-assumptions are made (as in bundled-data design); from the historical trends in synchronous design, we may infer that we need not apologize for the mixing of timing and logic—everyone that builds fast circuits mixes. Neither should we be surprised to find out that the subtleties of asynchronous pulse-logic are due mainly to dependencies between logic and timing.

4.1 A simple logic example

The next pulsed circuit we shall examine is what we call a "pulse merge." We can think of this as the implementation of the CHP program

$$*[\ L0,\ L1;\ R\ ]\ ,$$

where $L0$, $L1$, and R are ports (i.e., the mention of $L0$ etc. in the program signifies a communication on that channel). The synchronization of $L0$ and $L1$ is here explicit in the CHP program, but the kinds of circuit structures required for this explicit synchronization are also used for implementing data computations, where implicit synchronizations capture the fact that output values of a logical function cannot be computed until inputs are available. For instance, the program $*[A?a, B?b; C!(a+b)]$ explicitly synchronizes at the explicit semicolon and at the "loop semicolon" between loop iterations, and it also implicitly synchronizes the data because producing a value on C is impossible before inputs have arrived on A and B. The explicit synchronizations are mainly for the convenience of human understanding—they could, and should, eventually be removed; the data dependencies cannot be—the real task of the system designer lies in minimizing the need for data synchronization.

For the time being, let us keep pretending that the circuits we design will only have to be used once. It should right away be obvious that a QDI-inspired structure for the merge such as the one seen in Figure 4.1 (where $l0$, $l1$, etc. signify the circuit implementations of the channels—i.e., the corresponding electrical nodes) and described by the PRS

$$\begin{array}{lcl} \ldots \wedge l0 \wedge l1 & \rightarrow & r\_\!\downarrow \\ r\_ & \rightarrow & r\!\downarrow \\ \neg r\_ & \rightarrow & r\!\uparrow, \end{array}$$

will not do.[1] The QDI merge behaves like a C-element; in other words, it waits until both inputs have become asserted and then asserts its output. This will not work for pulses because the pulses are ephemeral: the high voltage-level signifying a positive pulse is only present for a short time.

[1] The ellipsis indicates that some details about the internal synchronization of the circuits have been left out. In QDI circuits, we should have various "state variables" in this place for sequencing the actions of the circuit properly [43]. Synchronous designers will see the analogy with the "foot" transistor of clocked domino-stages.

Unless we are willing—and we are not—to insist that outside agents shall synchronize the arrivals of $l0$ and $l1$, attempting to work with coincident pulses will get us nowhere.

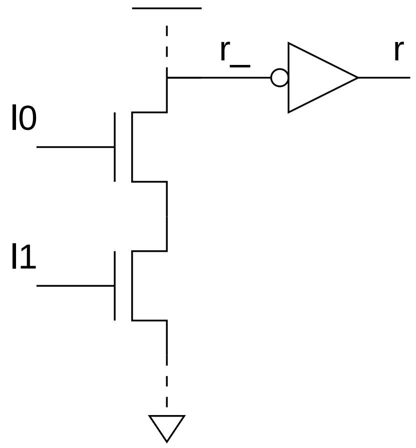

Figure 4.1: Input transistors in QDI merge.

If we are to design circuits that generate outputs computed from pulse inputs that can arrive at different times, we shall have to capture the pulses somehow and "remember" them. The circuit in Figure 4.2 is a conceptual solution to the problem. Incoming pulses on $l0$ and $l1$ are captured by the diodes; the maximum voltage during the pulse is stored on the gate capacitance of each transistor.

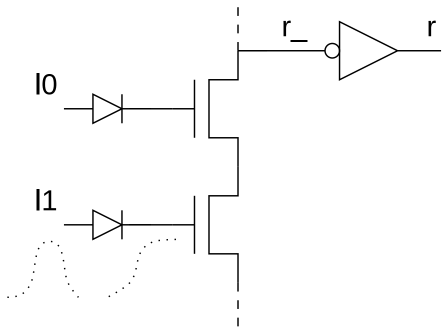

Figure 4.2: APL circuit, version with diodes.

The diode-transistor combination captures pulses by turning them into voltage levels. This is enough for using the circuit just once, but we seem to have pushed part of the problem ahead of us; if we capture a pulse by converting it into a voltage level with a one-way device, how do we reset the circuit so that we can use it again? A straightforward way of doing this is shown in Figure 4.3; here we have added reset transistors that are exercised by a separate reset pulse—the gates of the reset transistors are marked R in the figure; presumably, the generation of the reset pulse is contingent on the circuit's having produced its outputs, thus ensuring that the inputs are no longer required. (Of course, we shall have to explore this presumption later.) In simple circuits, the reset signal can often be the same for all inputs, as shown. In the case of data, the reset signal can also fan out to all the data rails, i.e., to inputs that have not necessarily been asserted, since vacuously[2] resetting a data rail that was not previously set by an incoming pulse is harmless. When we reuse the circuit, we shall have to arrange things so that the pulse on R avoids interfering with the input pulses on $l0$ and $l1$, or chaos will ensue.

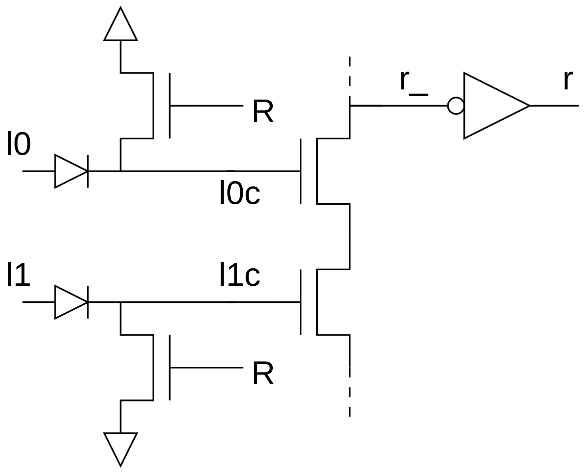

Figure 4.3: APL circuit, version with diodes and reset transistors.

Now we must get our signal senses in order. For simplicity, we have so far preferred discussing positive (i.e., from a stable low value: up, then down back to low) pulses. This turns out to be the wrong choice. We shall not discuss all the reasons why this is so; instead, we justify our choice of signaling senses by observing that in our example circuit, $r\_$ will certainly be in the negative sense; i.e., it goes to **false** rather than to **true** when it produces an output. The obvious choice is to make the logic-gate signals, $l0c$ and $l1c$ in Figure 4.3, positive logic; and $r\_$ negative logic, as indicated by the figure. Furthermore, we can see that the minimum number of stages of logic that we can use is two—one for implementing $\ldots \wedge l0c \wedge l1c \to r\_\!\downarrow$ and one for the diode; this means that we shall have to design the circuit so that $r\_$ is pulsed. While we might entertain the idea of using an actual diode (a substrate diode or a transistor with the drain and gate tied), normally using a transistor will be better; the transistor has gain, and foundries optimize CMOS processes for producing good transistors, not diodes. Figure 4.4 shows the modified circuit; the kinship with

[2]An assignment $x := a$ is called *vacuous* if x already has the value a before the assignment; else it is *effective*.

synchronous domino-logic and QDI-asynchronous precharge-logic (see Lines [43]) is obvious.

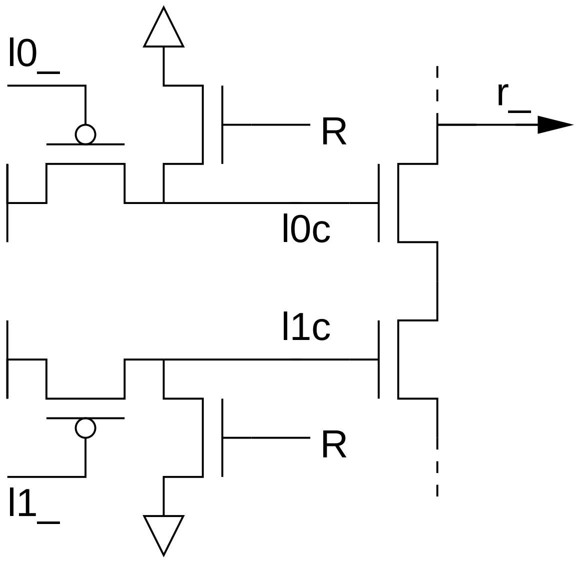

Figure 4.4: APL circuit; diodes implemented with transistors.

In the form of a PRS, then, we may write the asynchronous–pulse-logic implementation of $*[L0, L1; R]$ as

$$\begin{array}{lcl}
\neg l0_ & \to & l0c\uparrow \\
\neg l1_ & \to & l1c\uparrow \\
\ldots \wedge l0c \wedge l1c & \to & r_\downarrow \\
\ldots & \to & r_\uparrow \\
R & \to & l0c\downarrow \\
R & \to & l1c\downarrow \; .
\end{array}$$

Here, the p-transistors denoted by $\neg l0\_ \to l0c\uparrow$ and $\neg l1\_ \to l1c\uparrow$ are implementations of inverting diodes. The inversion at the "diode" means that we may (or must) merge the diode with the inverter of Figure 4.3; removing the inverter completely is usually preferable to adding another. There remains filling in the implementation of the ellipses, but the general form of this circuit is what we shall see throughout the rest of this thesis: a negative pulse appears; a single p-transistor captures the pulse; the captured pulse, now a high voltage-level, gates a transistor in a logic network, which computes the desired function and produces an output pulse, negative like the original pulse; the output pulse having been produced, a suitably generated reset pulse removes the original, captured pulse.

4.2 Pulse-handshake duty-cycle

Let us now consider two repetitive processes, named P and Q, that synchronize once per iteration by means of a channel. We refer to the nodes used for implementing the channel as *interface nodes* when necessary for distinguishing them from other nodes in the system (e.g., nodes that are private to the implementations of P and Q).

The standard QDI handshake[3] is

$$P \;:\; *[\; [re];\; rd\uparrow;\; [\neg re];\; rd\downarrow \;] \;\|\; Q \;:\; *[\; re\uparrow;\; [rd];\; re\downarrow;\; [\neg rd] \;] \;.$$

This handshake in theory has two synchronization points (e.g., the up- and down-going phases on rd), but the difficulties attached to designing circuits that compute on a downgoing transition are enough to convince us that using the handshake for more than a single synchronization is usually a bad idea.

We first consider a simple adaptation of the QDI handshake to the pulsed world by writing the pulse generation on a node x as $x\wr\!\downarrow$ (read "x pulse down") and $x\wr\!\uparrow$ (read "x pulse up"). If a process executes $x\wr\!\uparrow$, x will from then on evaluate to **true**, until some process executes $x\wr\!\downarrow$. This corresponds directly to the "diode-capacitor-transistor" model we used in the previous section.

Now we can write a pulse handshake corresponding to the QDI handshake thus:

$$P \;:\; *[\; [re];\; rd\wr\!\uparrow,\; re\wr\!\downarrow \;] \;\|\; Q \;:\; re\wr\!\uparrow;\; *[\; [rd];\; rd\wr\!\downarrow,\; re\wr\!\uparrow] \;.$$

(Note that already this handshake is not delay-insensitive; unless we add timing constraints, there may be interference.) But, secondly, we should realize that if we allow P's directly detecting that Q has executed $rd\wr\!\downarrow$, then the pulse $re\wr\!\uparrow$ is unnecessary, since simply by knowing that $rd\wr\!\downarrow$ has completed, P may know it can safely send another $rd\wr\!\uparrow$. P can glean the necessary knowledge by monitoring rd (at the perhaps significant cost of using p-transistors in series).

By thus eliminating the communicating on re, we go further than the QDI designer; he had the freedom to release and leave floating his nodes early but could choose not to use that freedom—we shall *require* the nodes' early release. We call the resulting protocol the *single-track handshake.*[4,5]

The removal of the acknowledgment wire and pulse does somewhat reduce the flexibility of the allowable family of pulsed circuits, because one extra means for flow control has been removed: in the example, delaying P further is not possible once $rd\wr\!\downarrow$ has been executed, because no further

[3]Here we have written the handshake with an inverted acknowledge, called the *enable*. This choice of senses improves circuit implementations; while this is admittedly a weak reason for choosing the sense at the HSE level, there is really no reason at all except convention itself for choosing the conventional sense for the acknowledge. Perhaps also the term "enable" is more suggestive of a pipeline with "flow control" than of an orderly handshake between two otherwise independent processes.

[4]The name *single-track handshake* was coined by van Berkel and Bink [8].

[5]Let the reader beware: the terms "single-track" and "single-rail" mean quite different things. "Single-track" refers to a handshake whose transmit and receive phases occur on the same wire or wires; whereas "single-rail" refers to the practice of encoding data one bit to a wire, conventionally used in synchronous and "bundled-data" systems. This thesis speaks only of "single-track" circuits; the author feels that the problems involved in using "single-rail" data encoding are doomfully severe, whence we shall not explore it.

notice will be given that it is all right for P to send another datum; $rd\wr$ is now the only signal to notify P, whereas we previously both removed the old datum and acknowledged it, either of which could have been used by P for determining that Q is ready for more input. But the flexibility that we thus remove is actually an unfamiliar one: for instance, it does not even exist in QDI systems, since these usually also have only one mechanism for flow control (the acknowledge).

Hence we shall in what follows restrict our attention to circuits that use the same wires for sending data as they do for receiving acknowledges and the same wires for receiving data as they do for sending acknowledges. The single-track handshake may be written in terms of HSE:

$$P \;:\; *[\;[\neg rd];\; rd\uparrow\;] \;\|\; Q \;:\; *[\;[rd];\; rd\downarrow\;]\;.$$

Note, however, that even though the syntax for the pulsed program looks similar to that used in the HSE describing ordinary QDI circuits, the semantics may be quite different. In QDI-HSE, it does not matter whether the driver for rd has three states (driving up, not driving, driving down) or two (driving down, driving up) in the implementation of the program $*[[re];rd\uparrow;[\neg re];rd\downarrow]$. The driver could, e.g., be implemented with an inverter, in which case rd is always driven (a combinational node). In general, what this means is that in QDI-HSE, a process that executes a sequence of commands $S;rd\uparrow;T;rd\downarrow$, where S and T are arbitrary program parts, may choose to stop driving up rd and leave it floating at any time after $rd\uparrow$ has completed, i.e., before T, during T, or after T, as long as the $rd\downarrow$ action has not yet begun.[6] This is no longer allowed.

We shall take the following complementary specifications as given:

Definition 4.1 (Maximum single-track hold time (maximum impulse)) *If a process P begins driving an interface node to a new value v at time t, then P must have stopped driving the node at time $t+\sigma_v$, where σ_v is a (system-wide) global constant; P may not again drive the node to v until it has detected that the node has left v (see Definition 4.2).*

Definition 4.2 (Minimum single-track setup time (minimum inertia)) *If a process P detects that an interface node has switched to a new value v at time t, then P must not drive that node away from v until the time $t+\xi_v$, where ξ_v is a (system-wide) global constant.*

We should like to design our circuits so that they satisfy:

Definition 4.3 (single-track–handshake constraint) *A set of processes $\mathcal{S}$ satisfies the* single-track–handshake constraint *if $\xi_v \geq \sigma_v$, for all v and all processes in $\mathcal{S}$.*

[6]This is a big freedom. The conservative would say that this freedom should be approached with respect, because he thinks that it is difficult to design the state-holding circuits that must be used if any appreciable amount of time is allowed between the abandonment of $rd\uparrow$ and the start of $rd\downarrow$. (The electrical engineer refers to the state of the circuit during this period of time as "high-impedance" or "high-Z"; we shall call it *floating*.) On the other hand, the designer of traditional QDI circuits is apt to use this freedom to great effect for simplifying many aspects of his design; the freedom might perhaps allow his inserting inverters in convenient places without violating the rules of the QDI game. What this means is explained in detail in the QDI literature, e.g., by Martin, who explains the need for having such freedoms under the heading "bubble reshuffling" [54]. Martin also uses a technique called "symmetrization," which involves replacing $x\uparrow;\ldots;x\downarrow$ with the "stuttering" $x\uparrow;\ldots;x\uparrow;\ldots;x\downarrow$ when this simplifies the circuit realization.

This property will guarantee that there shall never be interference between the two actions $x\uparrow$ and $x\downarrow$. We may in practice choose to be lax about this, allowing some interference (see Section 3.2.2.2). The rationale for allowing a different σ and ξ for each possible value of v (usually only **true** and **false** are allowable values for v) is that this allows implementing $x\downarrow$ and $x\uparrow$ differently. But because the constants are global, we must still implement $x\downarrow$ similarly throughout a system, and likewise for $x\uparrow$.[7]

4.3 Single-track–handshake interfaces

We must remember that the adoption of the single-track handshake, while it appears to follow naturally from the pulsed version of the four-phase handshake, does not in itself involve exchanging pulses between communicating processes. One process sets x and another, having seen the activity, resets x as an acknowledgment. At this level, no pulses are visible, and the processes may defer the actions on x indefinitely, if they should prefer doing so. In this design style, we oblige no process to respond immediately to an input any more than we do a QDI process. What a single-track–handshake process may never do, on the other hand, is to drive one of its interface nodes for a long period of time (to either rail); it may also not drive one of its inputs too soon after it has changed.

Where did the pulses go? The single-track processes use pulses internally for guaranteeing that the single-track–handshake constraint is satisfied. If we compare the single-track processes with the straightforward translation of QDI handshakes into pulsed handshakes, the main change is that we have moved the "diode" transistors at the inputs of the "diode-capacitor-transistor" circuits to the transmitting process.

We should note that the requirement that single-track processes use pulses internally is fundamental. There is simply not enough information available to a process for it to implement $*[[\neg rd];rd\uparrow]$ quasi delay-insensitively.[8]

[7]The careful reader will notice that we really only need to satisfy the constraints on each channel instance separately; there is in theory no need for making the constraints global. We shall not discuss such refinements in this thesis. On the one hand, our making the constraints local would break the modularity of the design style, and this is reason enough for saying no.

On the other hand, we could profitably take the view that: first, we should design systems as if the constraints were to be globally satisfied—thus ensuring that the digital design could be sized to operate properly; secondly, the final sizing should be done with only local constraints, local values of ξ_v and σ_v—thus making best use of the silicon: this compromise should allow the design of formally modular systems without paying the practical price of complete modularity. The required automatic design-tools have yet to be written.

[8]Seitz [75] rejected pulsed asynchronous circuits for this reason; but on closer inspection, the reason is weak: he replaces the timing assumption required for proper pulsed operation with others that appear to be just as difficult to satisfy. Admittedly, this comparison is a bit unfair: if we were transported back in time to 1980, the reason we should reject the pulsed circuits developed in this thesis would be the high transistor-count; the dataflow-influenced QDI style used in the MiniMIPS would likely have to be rejected for the same reason.

4.4 Timing constraints and timing "assumptions"

One of the great strengths of QDI circuits is their reliability when faced with uncertainties or variabilities in the timing characteristics of their constituent circuit elements. Unfortunately, the requirement that the circuits must have internal equipotential regions (isochronic forks) means that even QDI circuits are not immune to reliability problems that result from timing mismatches. We should also remember that, while we should not normally consider a single operator to have internal isochronic forks, this is merely a convenient fiction resulting from a simplistic model for QDI circuits.

We consider implementing the inverting C-element:

$$\begin{aligned} a \wedge b &\rightarrow c\downarrow \\ \neg a \wedge \neg b &\rightarrow c\uparrow \end{aligned}$$

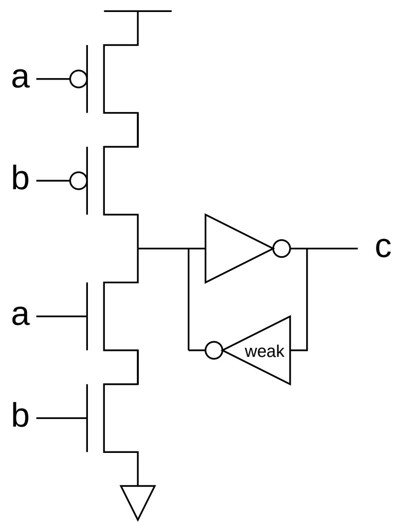

Figure 4.5: Pseudo-static C-element.

The simplest implementation of the C-element is the pseudo-static[9] version shown in Figure 4.5. The wire that connects the gate of an n-transistor with the corresponding p-transistor in the C-element can behave like an isochronic fork. For instance, we might have that the input a changes very slowly from a valid logic-zero to a valid logic-one. During the time when a is between the two legal logic levels, both the n-transistor and the p-transistor connected to that a will be turned on. As long as this situation persists, the C-element behaves like a slow inverter with respect to b; this behavior is not at all what we should like. If the transition on a is extremely slow with respect to the circuitry that acknowledges b, the circuit may misfire, which would likely result in a system failure.

[9] An operator $U \rightarrow x\uparrow$, $D \rightarrow x\downarrow$ is called *static* or *combinational* if $U = \neg D$; if not, it is *dynamic*, but if a dynamic operator is realized with a staticizer (keeper), we call it *pseudo-static*.

The possibility that the circuit should misfire is especially menacing for the pseudo-static implementation that we have shown (and even more so for fully dynamic versions). The reason is that the switching thresholds of a pseudo-static operator are moved unsafely towards the power rails because there is no "fight" between the transistor that is just turning on and the one that in combinational logic would be about to turn off; i.e., the noise margins are reduced. We can remedy the situation either by making the staticizer larger or by changing the circuit so that it is fully or partially static, thus re-introducing the fight [7]. In an extreme case, we can even make the C-element hysteretic (like a Schmitt trigger; see Glasser & Dobberpuhl [29]). The reason we shall not do these things is that we should lose much of the performance advantage of the dynamic implementation. The energy dissipation would also increase, compounding the undesirability of the design.

4.5 Minimum cycle–transition-counts

We can say that a QDI circuit is correct only if each signal within it satisfies certain timing constraints. The rise time of the signal must be small compared with the delay of the cycle of transitions that invalidates that signal. As we have seen above, these constraints must be satisfied also by signals that are not on isochronic forks in the normal sense, i.e., those that are checked in both the up- and down-going directions. Since the delay of a cycle of transitions can be thought of as a linear combination of rise (and fall) times, the constraint is two-sided as stated: we cannot allow a transition to be slow compared with its environment, and we cannot allow a sequence of transitions to be fast compared with their environment. Because we are trying to keep a single transition faster than the sum of delays of a sequence of transitions, the difficulty in maintaining reasonable rise times becomes lesser if we design circuits so that every cycle has a minimum number of transitions before invalidating the input, e.g., five.

On the other hand, the number of transitions on a cycle acts as a minimum constraint on the cycle time of a system; in other words, the fewer the transitions on the cycles, the faster the circuits run. This suggests that we should decrease the number of transitions on the cycles to the minimum possible.

Summing up, we see that part of the reliability of a QDI circuit is determined by the minimum number of transitions on any cycle, and at the same time, the maximum speed of the circuit is determined by the maximum number of transitions on any cycle that is exercised frequently. A reasonable design approach in the face of this dichotomy is to aim at a roughly constant number of transitions in each cycle of the system. If the target number of transitions is small, then the circuits designed will be fast and unsafe (i.e., difficult to verify the correctness of); conversely, if the number is large, the circuits will be slow and safe.

4.6 Solutions to transition-count problem

Obviously, we must be careful when dealing with pseudo-static non-combinational circuits. In simple cases, we could follow van Berkel's advice [7] and use techniques that add hysteresis to the circuits; but these techniques are often inconvenient, and they always complicate the circuits, as well as raise their power consumption and increase their latency. In any case, we did not use these techniques in the circuits used in the MiniMIPS processor; instead, we used pseudo-static circuits and hoped for the best. In honesty, it should be mentioned that our hope was supported by extensive `aspice` simulations and mechanical circuit-verifications that verified that the worst-case ratio of the rise time T_a to the delay $T_{c \to b}$ was small.

The fact that the MiniMIPS processor and other chips using the same circuit techniques were functional is evidence that timing assumptions involving isochronic forks in QDI circuits are manageable, even when the circuits involved are complex. In APL circuits, we shall take a different approach: the timing assumptions used in APL circuits depend on internal operator delays; thus, they are formally more restrictive. On the other hand, as we shall see, the timing assumptions appear under much more controlled circumstances than in QDI circuits; under certain circumstances APL circuits may be more reliable than QDI circuits because of the APL circuits' simpler internal timing relationships.

4.7 The APL design-style in short

The APL design method aims at describing how to compile CHP programs into circuits; it is thus similar to the QDI design method. But whereas we strive for designing circuits with a minimum of timing assumptions when designing QDI circuits, we use internal pulses for implementing the single-track handshake when designing APL circuits.

We could introduce timing assumptions in many different ways for the purpose of simplifying or increasing the performance of QDI circuits; several schemes have been mentioned already. Our APL scheme takes a simple approach: we use a single-track external handshake, and we minimize the number of timing assumptions at the interfaces between processes; internally, in contrast, we design the circuits so that they generate predictably timed internal pulses. This is a separation of concerns: most of the variable parts of an APL circuit (i.e., those parts that vary depending on what CHP is being implemented) are arranged so that their delays do not matter much for the correct operation of the circuit; conversely, the pulse generator, whose internal delays *do* matter for the correct operation of the circuit, does on the other hand not vary much.

We consider the implementation of some CHP as an APL circuit in 0.6-μm CMOS (the same technology that we used for the pulse repeater demonstrations). We should not expect to be able to say much about the delay from the inputs' arriving to the outputs' being defined; in contrast, we should expect that the internal pulses always are approximately 0.7 ns long. Before we fabricate a chip, we want to verify that there is a good chance that it will work as designed. This is when we benefit from the invariability of the pulse length: since the pulse length varies so little (this is a different way of saying that the pulse repeater has a high length-gain), we commit only a minor infraction if we assume that the length is constant.

The simplifying power of this assumption can hardly be overstated: once we have assumed that the pulse length is given, we need only verify that the circuitry generating the pulse and the circuitry latching the pulse work properly given that pulse length, and—this is the important part—we need not consider the effects of the inputs and outputs on the pulse length. This means that we can verify our timing properties locally. In effect, we have reduced a problem consisting of verifying the properties of the solution to a system of N coupled nonlinear equations into one involving N uncoupled nonlinear equations: we have gone from a task that seems insurmountable to one that is (in theory at least) easy.

Chapter 5

A Single-Track Asynchronous–Pulse-Logic Family: I. Basic Circuits

Remember that the slowest link in the APL system is you, the user. You are limited by the speed with which you can enter information via the keyboard.

— *L. Gilman and A. J. Rose,* APL, An Interactive Approach *(1970)*

5.1 Introduction

In Chapters 3 and 4, we developed a theory that accounts for the proper operation of pulsed circuits, and we described some experiments bearing out the theory in practice. In this chapter, we apply the theory to the design of a family of circuits that can be used for implementing a wide variety of logic functions. The particular features that we choose to implement directly in our logic family are strongly influenced by the MiniMIPS work; the QDI circuits developed for the MiniMIPS will here have their single-track–handshake APL counterparts. For brevity, we shall abbreviate "single-track–handshake asynchronous–pulse-logic" as *STAPL*.

5.2 Preliminaries

Chapter 4 has established a number of constraints that STAPL circuits must obey. These constraints are inequalities, however, and there remains a great deal of freedom in our choosing the design parameters (e.g., ξ_v and σ_v in Definition 4.3). These parameters are somewhat arbitrary. Some of the things that influence them are outside the scope of the thesis; for instance, we should like a simple software implementation of the design tools (leading to uniform choices of ξ_v and σ_v); also, the author finds it easier to make a convincing argument about the quality of the STAPL design style in terms of high-speed circuits rather than in terms of low-power circuits, because the speed advantage of STAPL is obvious compared with QDI, whereas the power advantage—if any—is harder to quantify. We shall compare the QDI and STAPL design-styles for speed and energy later; see Section 8.4.5.

5.2.1 Transition counting in pipelined asynchronous circuits

We may get a rough estimate of timing by "counting transitions," i.e., the number of stages of logic that a signal must pass through. At first thought, this may seem an inaccurate way of estimating timing information; is it not the case that only the naïve would attach any weight to a timing estimate that reckons an inverter delay to be the same as the delay of a complicated precharge-logic gate, and that the sophisticated man must deal in more sophisticated delay models?

Our experience in the MiniMIPS project has first of all shown that when dealing with pipelined asynchronous circuits, transition counts are a useful delay measure, at least when the circuits are designed for maximum reasonable speed. We encountered many instances of circuits for computing some useful eight-bit operations, designed by someone reasonable and intended by him to operate at around 280–300 MHz according to our 0.6-μm parameter set. We saw almost universally that when such a circuit was implemented so that it cycled in 14 transitions (i.e., could accept a new input datum every 14 transitions), we had to size the transistors far larger than reasonable, compared with the transistors in an 18-transition–per-cycle implementation capable of running at the same speed. Increasing the transition count in the circuit to 22, we found that achieving the desired throughput becomes impossible. Only very carefully designed circuits (e.g., in the MiniMIPS, the register file) operating at 20 transitions per cycle could compete with the 18-transition-per-cycle implementations.

Secondly, to some extent also in explanation of the MiniMIPS results, asynchronous circuits by their nature mix computation and communication. While the logic delays of computation may vary greatly—even dynamically, depending on the data—the communication delays are often much more predictable. For example, an eight-bit QDI unit is difficult to implement in less than 18 transitions per cycle. Of these 18, only two are transitions of the logic, and one of these may even be masked by communication transitions (both transitions' being so masked would indicate a poor design). As a result, only a small part of the cycle time of a QDI unit will be affected by the delays of computation. One of our design objectives for pulsed circuits is to increase the proportion of the cycle that is the logic delay. As we shall see, however, we do not attempt bringing the cycle time for a complex system below ten transitions per cycle. Also, owing partly to our way of automating their compilation, the pulsed circuits will have more uniform completion-delays than the QDI circuits we are familiar with. We should keep in mind that the logic delay that is the largest and the most variable is the "falling domino" transition. Increasing the delay of this transition can only improve things so far as the circuit's satisfying the single-track–handshake constraint goes; in other words, if the uniform-delay model suggests that a pulsed circuit is correct, then the real circuit delays will only improve the operating margins if the logic-computation delay is increased compared with the other delays.

Thirdly but not least importantly, the equal transition-counts assumption can for several reasons be self-fulfilling. For instance, let us assume that we have decided to use a uniform implementation for pipeline stages, i.e., an implementation whose every stage runs at the same speed, counted in transitions per cycle. As we have seen, the completion circuitry will be similar from stage to stage. Thus, if a process is part of a pulsed pipeline, and the completion delays are uniform throughout the pipeline, then the designer will feel an urge for making the logic delays equal also, since the cycle time of the pipeline as a whole will be determined by the slowest stage.

In summary: the MiniMIPS experience shows that our assuming the transition delays to be equal can be a useful model; the general nature of pipelined asynchronous circuits suggests that equal transition-counts are not, as we might fear, an unnatural design corner for the circuits to be shoehorned into; on the contrary, a good designer's tending to equalize delays throughout an asynchronous pipeline will lead to a circuit with roughly equal transition delays. Hence equal transition delays are likely a natural endpoint in the design space.

If we consider circuits that are designed more with reducing energy dissipation in mind, not for maximum speed as we assumed above, the situation can become more difficult; we might not want to match equal transition counts. But this is not an essential difference: in either case, we shall eventually have to verify that the transistor implementations of the pulsed circuits have delays that satisfy the single-track–handshake constraint to an acceptable degree.

5.2.2 Transition-count choices in pulsed circuits

The preceding section has made the case that transition counting can be an effective way of estimating delays in asynchronous circuits, with the caveat that the prophecy of equal delays for equal transition counts partly needs to be self-fulfilling. A corollary of our using transition counts for estimating delays is that when we should like particular delays to be equalized or ordered (i.e., in amount: we might always want the delay from transition a to transition c to be larger than that from transition b to transition d) in a circuit, a first-order approximation of the required delay-ordering is arrived at simply by requiring the corresponding ordering of the transition counts.

The inverting property of restoring CMOS logic implies that the number of transitions per execution cycle must be even if we will ensure the possibility of a circuit's returning to its initial state (e.g., as in an execution cycle of a CHP program); furthermore, any circuit that is symmetric in the sense that up- and down-going transitions take similar paths—e.g., through the same circuit elements—must have a transition count per cycle of the form $4n + 2$.[1] While the simplest QDI circuits are symmetric in this sense, this is only one way to design things. And just as for the QDI circuits, it is not necessary for the pulsed circuits to have this kind of symmetry: on the contrary, one of the purposes of our developing pulsed circuits is that these circuits can be asymmetric; the asymmetry allows our avoiding a good deal of work due to the circuits' checking for the occurrence of transitions that we know must anyway occur. The asymmetry is illustrated by Figures 5.1 and 5.2; the fact that the up- and down-going transitions follow the same path in the QDI implementation in Figure 5.1 is illustrated by the forward-path (for the QDI circuit, $[ri]; re\!\downarrow$, for the STAPL circuit, $[ri]; \ldots$) and backward-path arrows (QDI, $[\neg ri]; re\!\uparrow$, STAPL, $\ldots; ri\!\downarrow$) in the middle process and $[\neg ro]; re\!\uparrow$ in the one on the left), which both go through the completion circuitry, whereas they do not in the STAPL implementation in Figure 5.2.

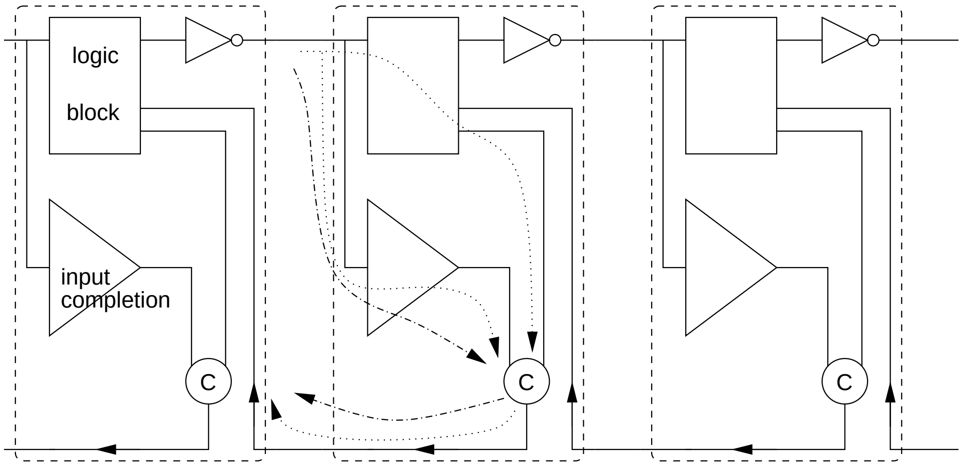

Figure 5.1: Path from input's arriving to acknowledge in QDI circuit: dotted, forward path; dash-dotted, backward path.

If we want the single-track–handshake timing-constraint to be satisfied in terms of transition counts, we have the following choices: each σ_v must be equal to the delay of an odd number of transitions, at least three (because a self-invalidating CMOS-gate does not work), and each ξ_v must be given an odd number of transitions larger than the corresponding σ_v (but see below).

As we earlier alluded to, a happy-go-lucky design-style might allow $\xi_v \approx \sigma_v$; in this case, we could allow the transition counts to be equal. SPICE simulations show that this may be reasonable. In practice, a choice of $\xi_v \approx \sigma_v$ may lead to the single-track–handshake constraint's being violated. The result of this need not be disastrous, however. Because the violation persists for only a short

[1] To see why this is so, consider the handshake $[re]; rd\!\uparrow; [\neg re]; rd\!\downarrow$. Since the total effect of executing the path from $rd\!\uparrow$ to $rd\!\downarrow$ amounts to an inversion of rd (in addition to possibly many other activities), this path must consist of an odd number of stages of logic, e.g., $2n + 1$. If now the circuit is symmetric in the way described, then the path from $rd\!\downarrow$ to $rd\!\uparrow$ is the same length, so that the total cycle time $t_c = t_{rd\uparrow\to rd\downarrow} + t_{rd\downarrow\to rd\uparrow} = 4n + 2$.

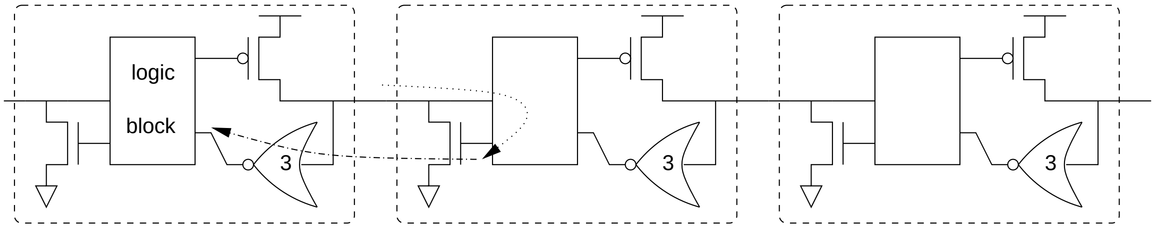

Figure 5.2: Path from input's arriving to its being removed in STAPL circuit: dotted, forward path; dash-dotted, backward path.

period of time, and because the violation occurs during a "handoff" (the driving of the node is passed from the sender to the receiver; the sender becomes the receiver, and the receiver, the sender) that guarantees the inputs' being monotonic, the effect is merely some extra power consumption because a node is briefly being driven both to Vdd and to GND. If the timing mismatches are not too large, then this situation is no different from what occurs in normal combinational logic when the input switches through the forbidden region, during which time both the pull-up and pull-down transistors are (weakly & briefly) tied. We must also remember that this particular problem is present in many modern synchronous clocking schemes for the very same reason that we see it in pulsed asynchronous circuits.[2] Finally, it is also present in the pulse repeaters we studied in Chapter 3.

To determine the cycle time of a STAPL handshake, let us refer back to

$$P \ : \ *[\ [\neg rd];\ rd\uparrow\ ] \ \| \ Q \ : \ *[\ [rd];\ rd\downarrow\ ] \ .$$

The trace of executing this handshake is $rd\uparrow; rd\downarrow; rd\uparrow; rd\downarrow; \ldots$. We shall compute the time taken from an $rd\uparrow$ to the next $rd\uparrow$; this is the cycle time.

After the rising edge of $rd\uparrow$, the driving process must not hold rd high for more than $\sigma_{\mathbf{true}}$ time units. Likewise, the receiving process must not begin executing $rd\downarrow$ until $\xi_{\mathbf{true}}$ time units have passed. Since we have $\xi_{\mathbf{true}} \geq \sigma_{\mathbf{true}}$, we know that $rd\downarrow$ can begin at the earliest after $\xi_{\mathbf{true}}$ time units have passed. Repeating the argument for the down-going part of the handshake, we should find that the cycle time for a STAPL circuit is constrained so that

$$t_c \geq \xi_{\mathbf{true}} + \xi_{\mathbf{false}}. \tag{5.1}$$

We have previously experienced problems when building asynchronous circuits with very fast feedback paths; in Section 4.4, for instance, we saw what could go wrong if the delays on a three-transition feedback path were not carefully adjusted. This is a strong reason for avoiding three-transition feedback paths and hence for requiring σ_v's being at least five transitions' worth of delay.

[2]High-performance clocking schemes using a "delayed reset" suffer from the same problem; the designers' response has been to make an effort to match the delays to minimize the crowbar currents [83].

The author believes that a design with $\xi_v \approx \sigma_v$ with all these equal to five transitions is safer than one where ξ_v is five and σ_v is three transitions. Whether this justifies the inevitable performance loss that results from our going from an eight-transition cycle time to a ten-transition cycle time is unclear; but we might also find it difficult to implement the amount of logic we should like in a single process in as little as eight transitions per cycle (see Section 6.3), and we should remember that a circuit with different numbers of transitions on its set and reset phases will necessarily have to be implemented asymmetrically, which makes designing it more difficult (see Section 3.2.4).

For all these reasons, we shall mainly study STAPL circuits with $\xi_v \approx \sigma_v$ and all equal to five transitions' delay. These circuits will also have the minimum reasonable input-to-output latency, which is two transitions' delay.

5.2.3 Execution model

In the next few chapters, we shall see the STAPL circuit family described in terms of production-rule sets (and the corresponding transistor networks according to the usual transformation rules developed for CMOS QDI circuits). Because the PRS of a STAPL system is not QDI (or speed independent), we cannot use a model where every PR can take an arbitrary time before it fires. We shall instead assume that all PRs take a single time unit from becoming enabled to firing, except when we say otherwise. PRs that take more time to fire will be labeled thus: $(n) a \to b\uparrow$ will take n time units from becoming enabled to firing.[3]

5.2.4 Capabilities of the STAPL family

Lines's work, which establishes a straightforward way of compiling QDI circuits with reasonable capabilities, has inspired the capabilities that we endow our STAPL circuit family with. His methods deviate from earlier practice: the more traditional asynchronous design procedure starts from a high-level CHP program, decomposes it into many small CHP programs, picks reshufflings (HSE), and thence compiles into production rules. Lines's work, on the other hand, suggests that a large class of QDI circuits can be built efficiently by translating more or less directly from decomposed CHP processes to production rules, thus avoiding the frequent explicit use of the HSE level of description (naturally, the compilation procedure itself implicitly represents the HSE).[4]

For pulsed circuits, the reshufflings are necessarily simpler than the allowable four-phase QDI reshufflings; consequently, the HSE is even less important for the working designer (the use of HSE

[3] What happens if a PR's delay is different in reality from what we have assumed? See Section 5.2.5.

[4] This is not to say that Lines's work was the first to be able to compile directly from CHP to PRS. In early work, Burns and Martin [13, 15] translated directly from CHP syntax to PRS; later on, the Philips group's work on Tangram has done the same [9]. Indeed, Lines does not present a completely automatic way of doing the translation: the difference between Lines's work and the other direct CHP-to-PRS work is his (implicit) use of a slack-elastic dataflow model and process templates at the level of a QDI buffer rather than at the much lower level of the various syntactic constructs of CHP.

is crucial in this thesis, where we are examining the reshufflings themselves; but once a reshuffling has been picked, there is much less leeway for the designer to affect the protocols that processes use to communicate). Also, the meaning of production rules is less clear for pulsed circuits; in this thesis, the author has chosen to use production rules as a convenient representation for transistor networks; the reader should not infer from the syntactic similarity to the PRS used in QDI circuits that the properties that are true of production-rule sets in QDI circuits—viz., stability and noninterference [54]—also are true of pulsed circuits.

The differences between QDI and APL circuits at the HSE and PRS levels are a good reason for our taking a higher-level view in describing the family of STAPL circuits. The higher-level approach will require our compiling directly from a level of description corresponding to small CHP programs into circuits. We shall see how to formalize the compilation in Chapter 7; for the time being, we refer to the capabilities informally.

STAPL circuits should be capable of basic dataflow operations:

- Computing an arbitrary logical function
- Computing results conditionally
- Receiving operands conditionally
- Storing state
- Making non-deterministic decisions
- Communicating with four-phase QDI circuits

In the rest of this chapter, we shall explore how to provide each one of these capabilities in STAPL circuits. Our final goal will be a circuit template that simultaneously admits of as many of these capabilities as possible, because such a template will allow the direct compilation of as wide a class of CHP programs as possible.

Most of the work we have to do in implementing the STAPL family consists of reconciling the handshake specification of $\sigma_v \approx \xi_v \approx 5$ transitions with the CHP specification of each circuit. We build up the circuits gradually, showing at each stage how the mechanisms required for building any desired STAPL circuit may be combined.

5.2.5 Design philosophy

In this chapter, the various techniques that we need for implementing the building blocks that we shall ultimately want are presented one at a time. Therefore, the reader is here cautioned that, especially in the earlier sections of the chapter, many of the circuits will be presented in an incomplete or at least not generalizable way so that they shall remain understandable. The sum of the techniques is what we should properly call the STAPL design-style.

Since our circuits depend on satisfying certain timing assumptions to work, we shall have to proceed carefully. We shall use a few simple techniques for guaranteeing that the circuits will stand a good chance of working properly. First, we shall always use the same circuit for generating the σ delays: a five-stage pulse generator. Secondly, we shall insist that the delay margins shall always be (in the limit of instantaneous transitions) half a cycle, or five transitions: this is the best that can be done uniformly. (This insistence will lead to our rejecting a "naïve" design in Section 6.4.3.1. We will still sometimes make an exception from this rule when we can thereby remove the foot transistor in simple circuits.) Lastly, the complicated logic will always be responsible for implementing the ξ delays; hence, if the logic gets slower (e.g., more heavily loaded), satisfying the timing assumptions becomes easier, not harder.

The basic template that we shall implement will be of the following form: produce each output as soon as possible (when the necessary inputs are available and when there is space in the output channels), then wait for the other inputs that are to be received; when all inputs that are to be received on a given cycle have arrived and all outputs that are to be produced on that cycle have been produced, reset all the inputs; and repeat.

5.3 The basic template

We start with a few simple STAPL circuits; thus we illustrate the basic properties of the circuit family. The simplest useful circuits in the family are the *bit generator,* the *bit bucket,* and the *left-right buffer.*[5]

5.3.1 Bit generator

The STAPL bit generator implements the CHP program

$$P \equiv *[\ R!0\ ]\ ,$$

or in terms of HSE,

$$P \equiv *[\ [\neg r0 \wedge \neg r1 \wedge \cdots];\ \ r0\uparrow\ ]\ .$$

[5] The reader should be cautioned that some of the mechanisms that we develop may appear to be very *ad hoc.* In some cases, it may even seem that there are much simpler ways of implementing the specifications. This is true: the bit bucket and left-right buffer could be implemented more simply without violating the single-rail–handshake constraint. The reason for the more complicated implementations we give here is that they generalize; i.e., they lead naturally to the implementations in Sections 6.3.3.2 and 6.3.3.3. The author feels that this method of exposition, while slightly illogical, is more understandable and pedagogical than the alternative of first presenting the full template and then presenting as examples the special cases that we now start with.

The bit generator will illustrate how to transmit data in the STAPL family. The basics are simple: we wish to send a zero value on R repeatedly. For each value we send: P must first wait until its communication partner—call it Q—signals that it is ready to receive, which Q does by lowering any asserted data wire; secondly, after the imposed setup-time $\xi_{\mathbf{false}}$, P asserts the zero rail of the channel; thirdly, we must ensure that P no longer drives the rail after the hold time $\sigma_{\mathbf{true}}$ has passed after the rail's being asserted.

5.3.1.1 Output completion

Using the precharged implementation suggested previously for the circuits, we find that the minimum delay in a STAPL stage from an input's being asserted to an output's being asserted is two transitions. Furthermore, a STAPL stage is not allowed to produce outputs until five transitions after its partner has removed the previous output from the channel. This means that the logic path bringing flow control from the output rails to the domino block must be three $(5 - 2)$ transitions long.

5.3.1.2 Forward path

Since the bit generator does nothing besides generating outputs, we can now start deriving a STAPL implementation. The only thing that remains to sort out is what to do with the remaining rails of the channel: since we know that only P will send on R, we can tie the other rails to GND and ignore them in P. This results in the HSE

$$P \equiv *[\ [\neg r0];\ \ r0\uparrow\ ]\ \ ;$$

straightforward compilation of the "forward path" $[\neg r0]; r0\uparrow$ results in the partial PRS

$$(3)\neg r0 \ \rightarrow\ re\uparrow$$

$$\begin{aligned} re \quad &\rightarrow\ r0\_\downarrow \\ \neg r0\_ \ &\rightarrow\ r0\uparrow, \end{aligned}$$

where the annotation "(3)" means that the production rule in question shall take three transitions to execute. The corresponding circuit is shown in Figure 5.3; the use of the NOR gate in the figure in place of the inverter of the PRS suggests how the R channel could be generalized to multiple rails.

5.3.1.3 Pulse generation

We shall finally see how $r0\_$ is precharged. The bit generator is particularly simple, and all that is required for precharging is a sequence of inverters. The final PRS, with the PRs listed in order of execution, is as follows:

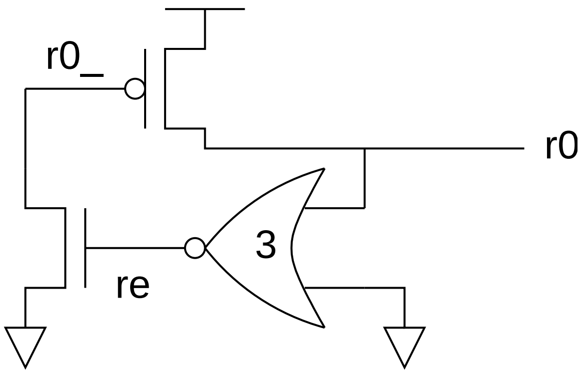

Figure 5.3: Forward (compute) path of STAPL bit generator.

$$
\begin{aligned}
(3)\neg r0 &\rightarrow re\uparrow \\
re &\rightarrow r0\_\downarrow \\
\neg r0\_ &\rightarrow r0\uparrow \\
(4)\neg r0\_ &\rightarrow rf\downarrow \\
(3)r0 &\rightarrow re\downarrow \\
\neg rf &\rightarrow r0\_\uparrow \\
(4)r0\_ &\rightarrow rf\uparrow
\end{aligned}
$$

The final circuit is shown in Figure 5.4, where we see the expanded version of each PR. Although it is not shown in the diagram, all nodes that are dynamic must be staticized; nodes that are pulsed may be staticized with a resistor to Vdd, whereas those that hold state must be staticized with a cross-coupled inverter-pair with weak feedback or an equivalent latching circuit. (The bit generator does not have any nodes that hold state.)

5.3.1.4 Execution

It will be instructive to make a timeline for P's execution, labeling each transition with the time when it occurs. Starting with the fall of $r0$ at $t = -3$ (the rationale for this choice is that we shall

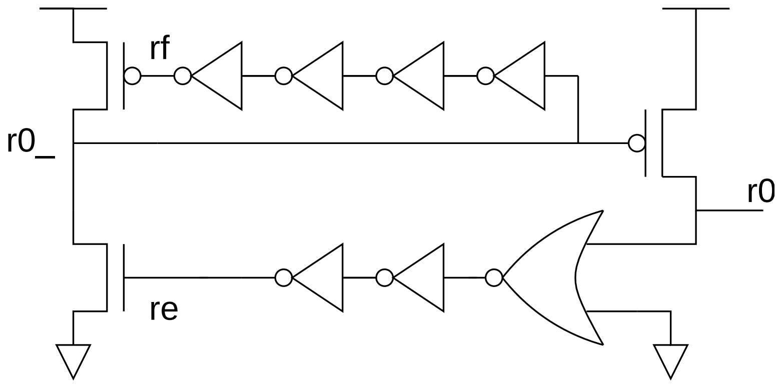

Figure 5.4: Complete STAPL bit generator.

consistently have the inputs to the domino block become active at $t = 0$), we get the following:

action	**time**				
$r0{\downarrow}$	-3				
$re{\uparrow}$	0	$=$	$(-3) + 3$		
$r0\_{\downarrow}$	1				
$r0{\uparrow}$	2				
$rf{\downarrow}, re{\downarrow}$	5	$=$	$1 + 4$	$=$	$2 + 3$
$r0\_{\uparrow}$	6				
$r0{\downarrow}$	$7 + \delta$	$=$	$2 + (5 + \delta)$		
$rf{\uparrow}$	10	$=$	$6 + 4$		
$re{\uparrow}$	$10 + \delta$	$=$	$(7 + \delta) + 3$		

An arbitrary delay, δ, has been added to the response time of process Q. This δ accounts for the pulse-signaling constraint's being single-sided on the response time of Q: Q *may* respond to $r0{\downarrow}$ after $\xi_{\mathbf{false}}$ has elapsed, but it *need not;* the arbitrary extra time that Q lingers in a particular execution is captured by δ. This allowed, arbitrary δ is what makes the design style asynchronous and composable.

5.3.1.5 Constraint satisfaction

Let us verify that this circuit satisfies the single-track–handshake constraint. Calling the pulse generator P and its neighbor Q, we must check that P obeys $\sigma_{\mathbf{true}}$ and $\xi_{\mathbf{false}}$ on $r0$ and Q obeys $\sigma_{\mathbf{false}}$ and $\xi_{\mathbf{true}}$. We assume all the σs and ξs are five time units (transition times). Since we are here describing the bit generator, we shall postpone the verification for Q to the section on the bit bucket; we shall assume that Q does its part and resets $r0$ at the earliest five time units after P has set it to **true** and that Q holds it **false** for no more than five time units.

Process P drives $r0$ from the time $r0_{\_}$ goes down until it goes back up; this is $6 - 1 = 5$ time units, as desired. Likewise, after $r0$ goes down at -3, P does not attempt driving it until at 2, again five time units. These things are clear from the production rules.

We should note that the pull-up transistor that causes $r0\uparrow$ must be sized large enough to drive the actual output most of the way to Vdd during the pulse. This means, for instance, that the only way of handling intrinsically long-delay outputs (e.g., outputs that themselves behave like RC loads rather than, as we have assumed, mere capacitive loads) is to slow down the circuits. We should hence not expect to use STAPL signaling off-chip or even on very long on-chip wires (see Section 8.2.3).

5.3.1.6 Remarks

A few things are noteworthy in the trace of P: $rf\downarrow$ and $re\downarrow$ occur after the same number of transitions; falling transitions occur at odd time indices, rising transitions at even indices; there are two "extra" transitions in $\neg r0 \to re\uparrow$, and three in $\neg r0_{\_} \to rf\downarrow$, that we have not made use of (in the sense that they are realized with inverters, but we could conceivably introduce logic instead; we cannot remove them completely since then the circuit would no longer satisfy $\xi_{\mathbf{false}}$).

First, the fact that $rf\downarrow$ and $re\downarrow$ occur at the same time is evidence that an important general design principle has not been ignored: the inputs to a logic gate's arriving simultaneously ensures that the gate spends the least possible time in an idle state. We shall see later that we cannot always trust re as sufficient flow control, and we shall sometimes have to use $re \wedge rf$ in the pulldown network; re and rf's being synchronized will then be of even more value.

Secondly, the strict alternation of falling and rising transitions suggests that a direct CMOS implementation is possible. We shall see that we can maintain this property while generalizing the design of the bit generator.

Lastly, we shall also find the "extra" transitions useful in generalizing the circuits; it is for instance obvious that the three transitions allotted to the path $\neg r0 \to re\uparrow$ could be used for implementing the neutrality and validity checks of wide channels, e.g., 1-of-8 codes.

5.3.2 Bit bucket

The bit bucket is the counterpart to the bit generator; its CHP is

$$*[\ L?_{\_}\ ]\ ,$$

where the use of the underscore variable _ signifies that the process should discard the incoming values, which it has read on L. The corresponding HSE is

$$*[\ [l0 \vee l1];\ l0{\downarrow}, l1{\downarrow}\ ]\ ,$$

where every $l0{\downarrow}, l1{\downarrow}$ except one is vacuous. The bit bucket's specification is similar to the bit generator's; and this suggests that we might be able to reuse the bit generator design, with appropriate modifications. However, this approach would not be suitable for generalization to the more complex units that we shall study later, because of our asymmetric choice of delays in the STAPL family: two transitions for the path input↑-to-output↑ (the forward latency), but five transitions for input↑-to-input↓ ($= \xi_{\mathbf{true}}$). Therefore, we develop the bit bucket quite differently from how we developed the bit generator; this will serve the purpose of providing an example of the input circuitry required in a generic STAPL unit.

5.3.2.1 PRS implementation

Although the bit bucket does not require the input values on L for any computation, it obviously cannot work without detecting the presence of inputs. We introduce a "dummy output" for this purpose; it is convenient to choose this output to be a single-rail channel—we call it X—, which cycles for every L received. The node implementing X, $x\_$, is precharged in the same way that the domino output is precharged in the bit generator. The corresponding PRS is

$$\begin{aligned}
&l0 \vee l1 &&\rightarrow x_{\downarrow}\\
&(4)\neg x_ &&\rightarrow xf{\downarrow}\\
&\neg xf &&\rightarrow x_{\uparrow}\\
&(4)x_ &&\rightarrow xf{\uparrow}.
\end{aligned}$$

If we consider the case when $l0$ is the asserted input at $t = 0$, the execution trace is as follows:

action	**time**
$l0{\uparrow}$	0
$x\_{\downarrow}$	1
$xf{\downarrow}$	$5 = 1+4$
$x\_{\uparrow}$	6
$xf{\uparrow}$	$10 = 6+4$

What remains is for us to ensure that the input is removed at time index 5. We do this by, first, adding output-validity circuitry (even though all we have is a dummy output); we call the node that checks the output validity xv. Since $x\_$ is here a single-rail signal, this amounts to an inverter.

Secondly, we add a pulse generator for generating the pulse that resets the inputs. To minimize the number of different circuits that shall need to be verified for timing and other properties, we arrange that this pulse generator is as similar as possible to the x_{-} pulse generator; i.e., it will generate negative pulses; this being the case, the actual reset pulse will have to be generated by an inverter.

We introduce the names ρ for the internal, negative pulse; ρf for the precharge of ρ; and $R4$ for the positive reset pulse, where the "4" conveniently denotes that the pulse becomes active four transitions after the input arrives (recall our convention of choosing for $t = 0$ the time of the input's becoming defined). Hence the PRS:

$$\begin{array}{lcl}
l0 \lor l1 & \rightarrow & x_{-}\downarrow \\
\neg x_{-} & \rightarrow & xv\uparrow \\
(4)\neg x_{-} & \rightarrow & xf\downarrow \\
xv & \rightarrow & \rho\downarrow \\
\neg\rho & \rightarrow & R4\uparrow \\
(4)\neg\rho & \rightarrow & \rho f\downarrow \\
R4 & \rightarrow & l0\downarrow, l1\downarrow
\end{array}$$

$$\begin{array}{lcl}
\neg\rho f & \rightarrow & \rho\uparrow \\
\rho & \rightarrow & R4\downarrow \\
(4)\rho & \rightarrow & \rho f\uparrow \\
\neg xf & \rightarrow & x_{-}\uparrow \\
\neg x_{-} & \rightarrow & xv\downarrow \\
(4)x_{-} & \rightarrow & xf\uparrow
\end{array}$$

The circuit diagram is shown in Figure 5.5. Note that we have shown xv as being within the first-stage pulse-generator because we should consider it part of the output completion of the pulse generator, not as an output. In other words, a pulse generator with a wide output channel would still have only a single xv output.

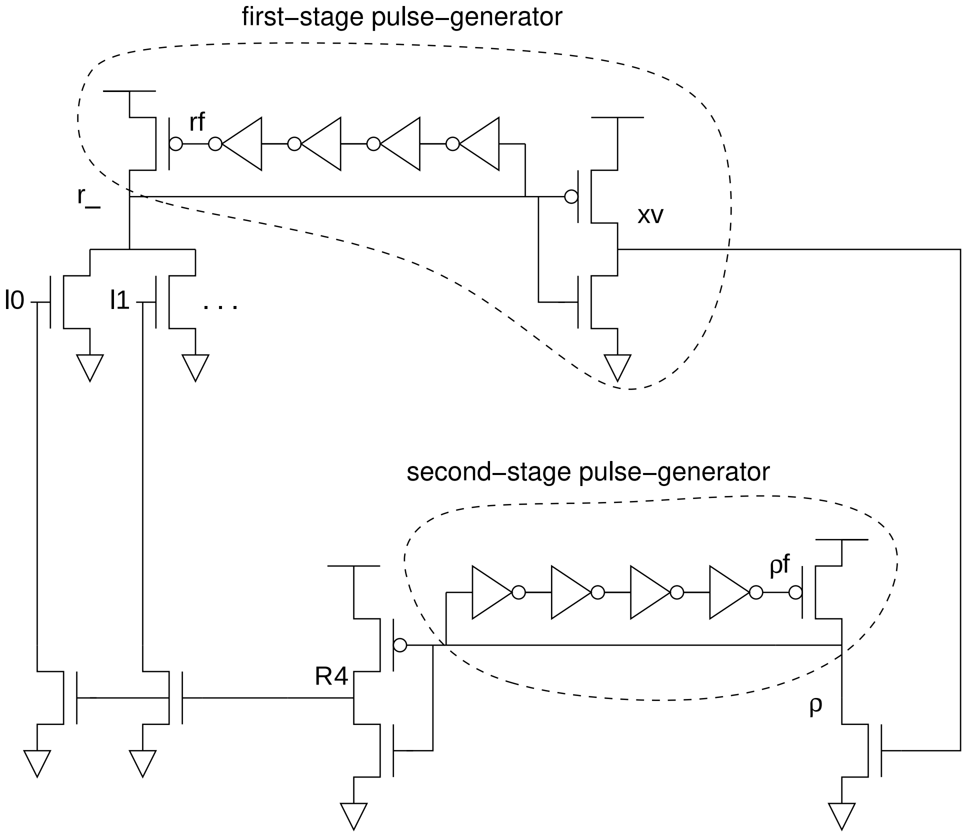

Figure 5.5: STAPL bit bucket.

5.3.2.2 Execution

The execution trace becomes as follows:

action	**time**
$l0\uparrow$	0
$x\_\downarrow$	1
$\rho\downarrow$	3
$R4\uparrow$	4
$l0\downarrow, xf\downarrow$	$5 = 4+1 = 1+4$
$x\_\uparrow$	6
$\rho f\downarrow$	$7 = 3+4$
$\rho\uparrow$	8
$R4\downarrow$	9
$xf\uparrow$	10
$l0\uparrow$	$10+\delta = 5+(5+\delta)$
$x\_\downarrow$	$11+\delta$
$\rho f\uparrow$	$12 = 8+4$
$\ldots$	

We should now verify that the bit bucket satisfies the timing constraints we claimed for it when we verified the bit generator, in Section 5.3.1.5. There we claimed that the bit bucket does its part and resets $l0$ (the bit generator's $r0$) at the earliest five time units after the bit generator has set it to **true** and that the bit bucket holds it **false** for no more than five time units.

We see from the production rules and the execution trace that the bit bucket indeed takes five transitions to respond, satisfying $\xi_{\mathbf{true}}$. Furthermore, the resetting of the input is handled by the signal $R4$; this signal is active (high) for five transitions too, which satisfies $\sigma_{\mathbf{false}}$.

We observe that the remarks of Section 5.3.1.6 hold for the bit bucket also.

5.3.3 Left-right buffer

We have seen enough detail in the description of the bit bucket and bit generator that we can combine the two to build a left-right buffer. The CHP specification for the buffer is

$$BUF \equiv *[\ L?x;\ \ R!x\ ]\ .$$

We shall give the implementation of BUF for the case when L and R are one-bit (1-of-2) channels. The HSE for BUF is then

$$*[[\ l0 \longrightarrow r0\uparrow\ [\!]\ l1 \longrightarrow r1\uparrow\ ];\ l0\downarrow,\ l1\downarrow,\ [\neg r0 \wedge \neg r1]\ ]\ .$$

5.3.3.1 PRS implementation

Except for the fact that both the output rails are used (and thus must be checked for validity), the output looks like that of the bit generator, in other words,

$$\begin{array}{ll}
(3)\neg r0 \wedge \neg r1 & \rightarrow re\uparrow \\
re \wedge l0 & \rightarrow r0_\downarrow \\
re \wedge l1 & \rightarrow r1_\downarrow \\
\neg r0_ & \rightarrow r0\uparrow \\
\neg r1_ & \rightarrow r1\uparrow \\
(4)\neg r0_ \vee \neg r1_ & \rightarrow rf\downarrow \\
(3) r0 \vee r1 & \rightarrow re\downarrow \\
\neg rf & \rightarrow r0_\uparrow, r1_\uparrow \\
(4) r0_ \wedge r1_ & \rightarrow rf\uparrow;
\end{array}$$

the inputs are handled as in the bit bucket:

$$\begin{array}{ll}
\neg r0_ \vee \neg r1_ & \rightarrow rv\uparrow \\
r0_ \wedge r1_ & \rightarrow rv\downarrow \\
rv & \rightarrow \rho\downarrow \\
\neg \rho & \rightarrow R4\uparrow \\
(4)\neg \rho & \rightarrow \rho f\downarrow \\
R4 & \rightarrow l0\downarrow, l1\downarrow \\
\neg \rho f & \rightarrow \rho\uparrow \\
\rho & \rightarrow R4\downarrow \\
(4)\rho & \rightarrow \rho f\uparrow
\end{array}$$

The only thing that is different in BUF compared with the program that should result from merging the PRs for the bit bucket and the bit generator is the two rules $re \wedge l0 \rightarrow r0\_\downarrow$ and $re \wedge l1 \rightarrow r1\_\downarrow$. These rules are responsible for the computation of the output, when a value arrives on L, as well as for flow control, when the process that receives R is slow to respond. The complete circuit is shown in Figure 5.6. An n-input pulse generator is made by replacing the first inverter in the 1-input pulse generator with an n-input NAND-gate, as suggested by the PRS.

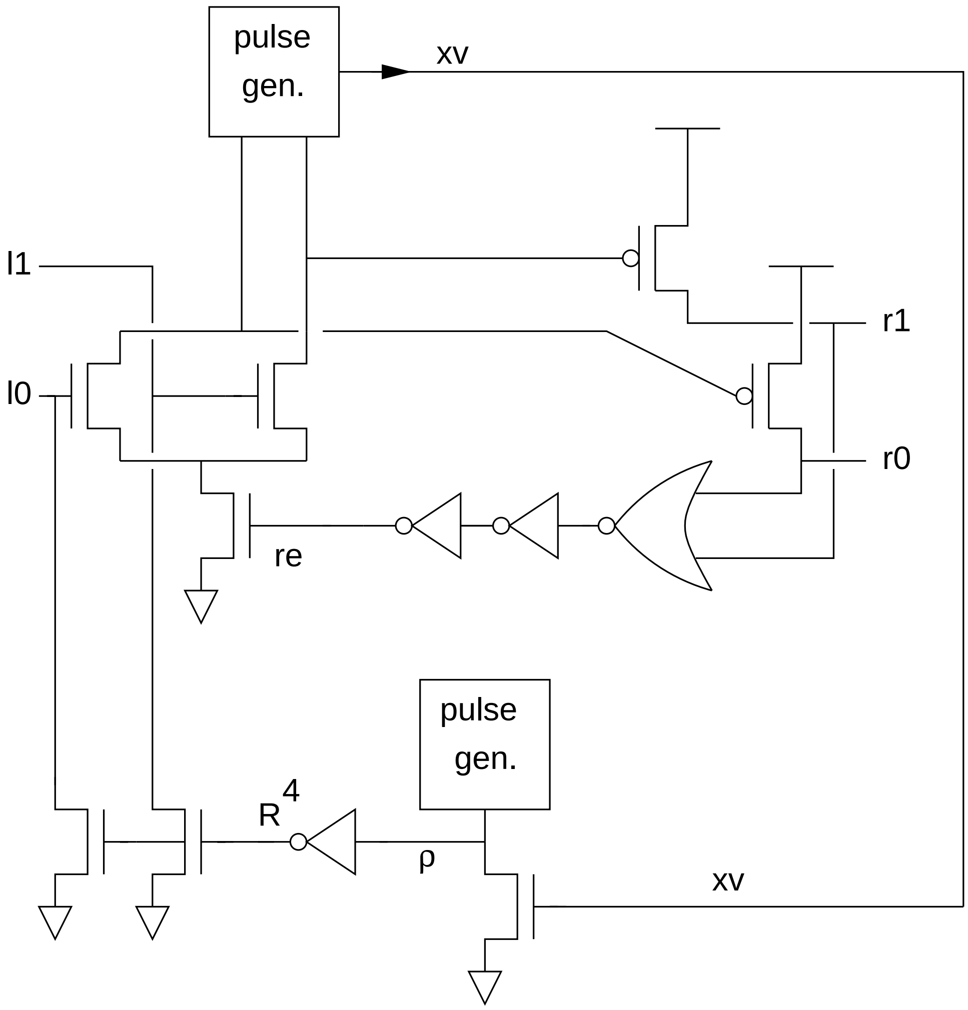

Figure 5.6: STAPL left-right buffer.

5.3.3.2 Execution

The execution trace for this process, assuming that the first L and the first $re\!\uparrow$ both arrive at $t = 0$ and that the counterpart on L sends 0, 1, . . . , is as follows:

action	time		
$l0\!\uparrow, re\!\uparrow$	0		
$r0\_\!\downarrow$	1		
$r0\!\uparrow$	2		
$\rho\!\downarrow$	3		
$R4\!\uparrow$	4		
$l0\!\downarrow, rf\!\downarrow, re\!\downarrow$	5		
$r0\_\!\uparrow$	6		
$\rho f\!\downarrow$	7		
$r0\!\downarrow$	$7+\epsilon$	$=$	$2 + (5 + \epsilon)$
$\rho\!\uparrow$	8		
$R4\!\downarrow$	9		
$rf\!\uparrow$	10		
$l1\!\uparrow$	$10+\delta$	$=$	$5 + (5 + \delta)$
$re\!\uparrow$	$10+\epsilon$	$=$	$(7 + \epsilon) + 3$
$r1\_\!\downarrow$	$11+\max(\delta, \epsilon)$	$=$	$\max\big((10 + \delta) + 1, (10 + \epsilon) + 1\big)$
$\rho f\!\uparrow$	12	$=$	$8 + 4$
. . .			

Arbitrary delays δ and ϵ have been inserted where the neighbor processes are allowed to linger; these delays have the same meaning as the δ of Section 5.3.1.4. Again, these allowable extra delays are what make this design style asynchronous, i.e., composable and modular.

5.3.3.3 Timing assumptions

Figure 5.7 shows how the different parts of the circuit satisfy the timing constraints: as promised, the σ pulse-lengths are controlled with pulse generators, and the ξ response-delays are delays through the logic. We call the actual delays of the circuit, as opposed to the timing constraints, $s_{\mathbf{true}}$, $s_{\mathbf{false}}$, $x_{\mathbf{true}}$, and $x_{\mathbf{false}}$.

While the single-track–handshake constraint only requires the conditions $\xi_{\mathbf{true}} \geq \sigma_{\mathbf{true}}$ and $\xi_{\mathbf{false}} \geq \sigma_{\mathbf{false}}$, our circuit implementations depend on more than that. The handshake constraint gives us the following constraints on $s_{\mathbf{true}}$, $s_{\mathbf{false}}$, $x_{\mathbf{true}}$, and $x_{\mathbf{false}}$:

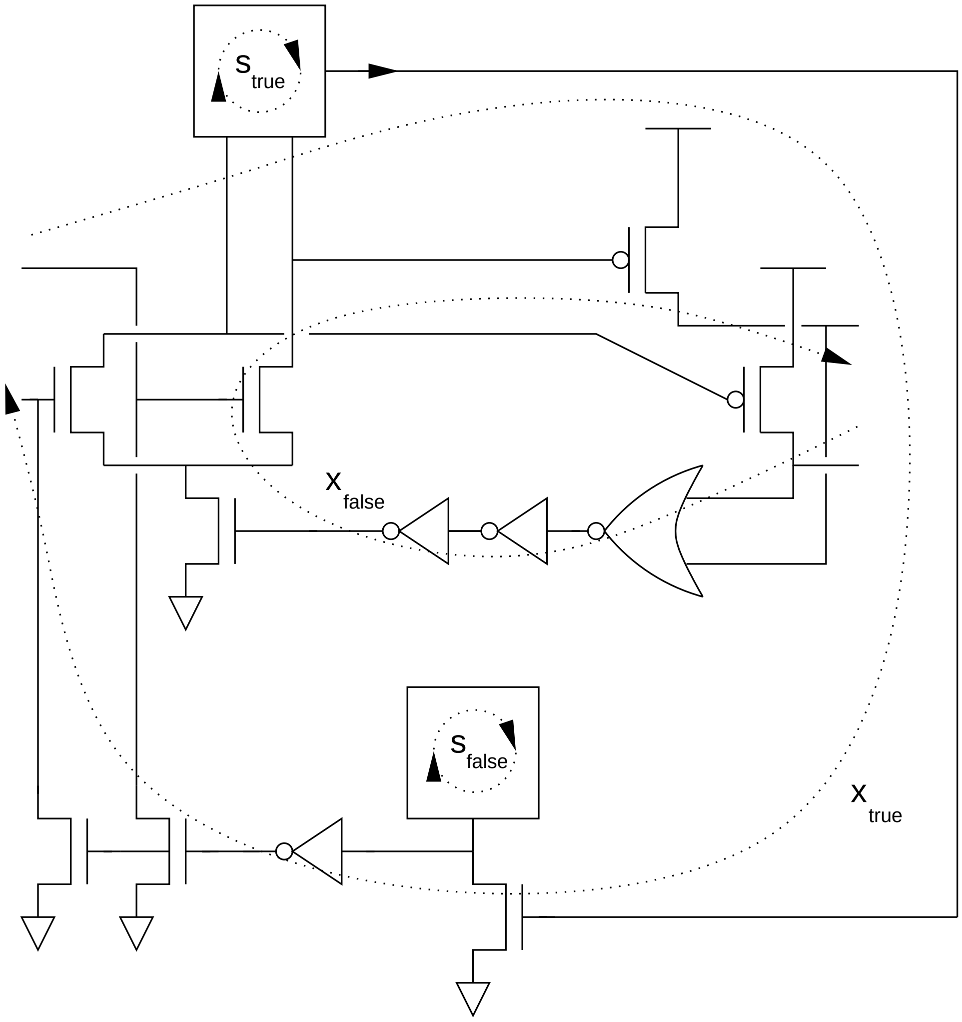

Figure 5.7: Paths implementing the delays $s_{\mathbf{true}}$, $s_{\mathbf{false}}$, $x_{\mathbf{true}}$, and $x_{\mathbf{false}}$.

$$\begin{aligned} s_{\mathbf{true}} &\leq \sigma_{\mathbf{true}} && (5.2) \\ s_{\mathbf{false}} &\leq \sigma_{\mathbf{false}} && (5.3) \\ x_{\mathbf{true}} &\geq \xi_{\mathbf{true}} && (5.4) \\ x_{\mathbf{false}} &\geq \xi_{\mathbf{false}} && (5.5) \end{aligned}$$

As we have remarked earlier, we always handle the σ constraints with pulse generators; hence we should expect $s_{\mathbf{true}} \approx s_{\mathbf{false}}$; $\sigma_{\mathbf{true}}$ and $\sigma_{\mathbf{false}}$ are also of course approximately equal to the s's. Since we have the difficult task of making sure that s is long enough for latching the output transistor yet not too long to violate (5.2) or (5.3), choosing to generate s with a single, well-characterized circuit is the right thing to do.

The only part of the circuit that can be slowed down arbitrarily is the domino pull-down that computes the logic; if the reset pulse is delayed, then the circuit may double-latch the inputs, and if the flow control through the NOR gate and inverters to re is slowed down, then the circuit may produce another output before it is allowed to do so.

5.3.3.4 Remarks

The left-right buffer consists mainly of a domino block and two pulse generators. One pulse generator is used for generating the outputs of the circuit; the other is used for clearing the inputs.

Again, the remarks of Section 5.3.1.6 hold. We note that $r1_{-}\downarrow$ happens only after both the arbitrary extra delays before $l1\uparrow$ and $r0\downarrow$ have been accounted for; this means that, as required, the circuit will not produce a new R until $\xi_{\mathbf{false}}$ time units after the old R has been consumed, nor will it attempt removing the new L until $\xi_{\mathbf{true}}$ time units after the new L has arrived. Furthermore, we should note with satisfaction that $l0\downarrow$, $rf\downarrow$, and $re\downarrow$ are perfectly synchronized and will stay thus as long as δ and ϵ are both zero; connecting buffers in a long chain with a bit generator at one end and a bit bucket at the other end will keep them at zero. But it is a bad sign for efficiency that several transistors in series are required in some places, viz. in the gates that compute rv and rf; we should like to avoid this kind of variance from the template because it introduces delays that are more difficult to keep uniform across a large system with diverse circuits.

5.4 Summary of properties of the simple circuits

So far, we have seen three STAPL circuits: the bit generator, the bit bucket, and the STAPL dual-rail left-right buffer. The following properties hold.

1. Each circuit takes at minimum 10 transitions for a cycle (the time it takes to return to a state once it has been departed).

2. If the environment of the circuit does not impose further timing constraints (in our discussion, by setting $\delta > 0$ or $\epsilon > 0$), then the circuit takes exactly 10 transitions per cycle.

3. The forward latency of the left-right buffer is two transitions.

4. If the environment imposes $\delta > 0$ or $\epsilon > 0$, then the circuit slows down accordingly; i.e., flow control is automatic.

5. The circuits can be implemented in CMOS; i.e., all the PRs are antimonotonic.

6. If the environment does not impose $\delta > 0$ or $\epsilon > 0$, then every input to every conjunctive gate arrives simultaneously.

7. The static slack of the left-right buffer is one; its dynamic slack is, to first order, 1/5.[6]

8. Foot transistors, except for the flow-control transistor, are unnecessary.

9. The inputs of the left-right buffer lead only to the domino block; no extra completion is necessary.

10. One or several NAND-gates with fanin equal to the width of the output channel are required in the circuit.

Each one of these except the last is a desirable property. Unfortunately, not all of the desirable properties can be maintained in more complex situations: specifically, we shall need foot transistors and extra completion-circuitry in some cases. We shall be able to remove the NAND-gates' series transistors, however.

[6]The *static slack* is the maximum number of tokens (data items) that a chain of buffers can hold when deadlocked; the *dynamic slack* is the number of tokens held at maximum throughput. See also Lines [43] and Williams [85].

Chapter 6

A Single-Track Asynchronous–Pulse-Logic Family: II. Advanced Circuits

Beware of the man who won't be bothered with details.

— *William Feather*

In Chapter 5, we saw how to design simple circuits in a pulsed asynchronous style: bit buckets, bit generators, and a simple left-right buffer. This is not enough to do much of interest: even simple processes like an adder or a controlled merge have behaviors that are not captured by these trivial circuits.

We shall develop the modifications to the basic circuits by considering how to add the necessary functions without breaking the mechanisms we have added before.

6.1 Multiple input and output channels

Let us consider the program

$$DBUF \equiv *[\ L?x,\ M?y;\ R!x,\ S!y\ ]\ ,$$

which is a simple example of synchronized input and output channels. In this form, this is not a very useful program; as we mentioned, the computation model that we are working in assumes that only the sequence of values sent on every channel—not the relative timing of the communications—has meaning, so $DBUF$ could equally well be written $*[L?x; R!x] \parallel *[M?y; S!y]$. In a slack-elastic program, the syntactic semicolon, like the one in $DBUF$, is not what demands synchronization; dependencies between different data computations are what demand it. But still we study $DBUF$ so that we shall see synchronizations in their simplest form; we do not yet want to think about the reasons for and extents of data dependencies.[1]

[1] We discuss these issues more fully in Sections 6.2 and 7.7.2.

We would never introduce unnecessary synchronization on the compute path of a STAPL process; accordingly, we shall not invent an artificial scheme for synchronizing $DBUF$ exactly as the CHP has been written. Instead, we shall implement the program

$$DBUF2 \;\equiv\; *[\ (L?x; R!x)\ ,\ (M?y; S!y)\ ]\ .$$

The synchronization between L, R and M, S in $DBUF2$ lies in the implied semicolon at the end of the loop, which keeps the channels loosely synchronized (i.e., $\mathbf{c}L - \mathbf{c}M \leq 1$, etc. at all times).

We shall not burden our circuits with synchronization on the forward path (except for the needed data synchronization); hence, at the HSE level, $DBUF$'s synchronizes the channels on the reset phase. In other words, assuming single-rail data,

$$DBUF \;\equiv\; *[\ ([l \ \wedge\ \neg r]; r\uparrow), ([m \ \wedge\ \neg s]; s\uparrow)\ ;\ l\downarrow, m\downarrow\ ]\ .$$

6.1.1 Naïve implementation

Most of the PRS implementation of *DBUF* is a straightforward composition of the two left-right buffers we saw above; one may surmise that certain parts will have to be shared in order to accomplish the synchronization, and that other parts cannot easily be shared. Examining the structure of the left-right buffer (see Section 5.3.3.4), we see that we may attempt generalizing it—at first incorrectly, it will turn out—into a new circuit that can handle several channels by using one pulsed domino block for each output channel and one extra pulsed block for generating the clear signal for the inputs. (The general scheme we use for generalizing the buffer's circuit structure is shown in Figure 6.1.) The block generating the clear signal will accomplish the simultaneous reset of l and m specified by the HSE. We should thus get the following PRS:

$$
\begin{array}{lll@{\qquad\qquad}lll}
(3)\neg r & \rightarrow & re\uparrow & rv \wedge sv & \rightarrow & \rho\downarrow \\
re \wedge l & \rightarrow & r_{-}\downarrow & \neg\rho & \rightarrow & R4\uparrow \\
\neg r_{-} & \rightarrow & r\uparrow & (4)\neg\rho & \rightarrow & \rho f\downarrow \\
(4)\neg r_{-} & \rightarrow & rf\downarrow & R4 & \rightarrow & l\downarrow, m\downarrow \\
(3)r & \rightarrow & re\downarrow & \neg\rho f & \rightarrow & \rho\uparrow \\
\neg rf & \rightarrow & r_{-}\uparrow & \rho & \rightarrow & R4\downarrow \\
(4)r_{-} & \rightarrow & rf\uparrow & (4)\rho & \rightarrow & \rho f\uparrow; \\
\neg r_{-} & \rightarrow & rv\uparrow & & & \\
r_{-} & \rightarrow & rv\downarrow & & &
\end{array}
$$

$$
\begin{array}{lll}
(3)\neg s & \rightarrow & se\uparrow \\
se \wedge m & \rightarrow & s_{-}\downarrow \\
\neg s_{-} & \rightarrow & s\uparrow \\
(4)\neg s_{-} & \rightarrow & sf\downarrow \\
(3)s & \rightarrow & se\downarrow \\
\neg sf & \rightarrow & s_{-}\uparrow \\
(4)s_{-} & \rightarrow & sf\uparrow \\
\neg s_{-} & \rightarrow & sv\uparrow \\
s_{-} & \rightarrow & sv\downarrow
\end{array}
$$

note that the only PR that synchronizes the activities of the $L-R$ block with those of the $M-S$ block is $rv \wedge sv \rightarrow \rho\downarrow$.

6.1.2 Double triggering of logic block in the naïve design

From the experiments with the pulse repeaters of Section 3.2 we remember that an input staying active for too long could cause a pulse repeater's consecutively interpreting that input as several;

the same problem could occur in $DBUF$. If for instance an input arrives by way of $l\uparrow$ but some delay should intervene before the arrival of $m\uparrow$, then $DBUF$ would not quickly reach the action $l\downarrow$; as a result, l could stay **true** for an arbitrarily long time. The trouble this would cause is apparent from an examination of the relevant PRs, those of the $L-R$ block, viz.

$$\begin{array}{llll}
(3)\neg r & \rightarrow re\uparrow & \neg rf & \rightarrow r_{-}\uparrow \\
re \wedge l & \rightarrow r_{-}\downarrow & (4) r_{-} & \rightarrow rf\uparrow \\
\neg r_{-} & \rightarrow r\uparrow & \neg r_{-} & \rightarrow rv\uparrow \\
(4)\neg r_{-} & \rightarrow rf\downarrow & r_{-} & \rightarrow rv\downarrow; \\
(3) r & \rightarrow re\downarrow & &
\end{array}$$

there is here no mention of $R4$ nor of ρ; since it does not wait for these signals, this circuit would read l as being **true** repeatedly; this would continue until $m\uparrow$ occurs, when ρ and $R4$ will at last pulse, removing l from the input and finally—alas, too late!—putting an end to the nonsense.

The way to eliminate the repeated triggering of the $L-R$ block is obvious. The issue is simply that we did not properly implement the final semicolon in $*[([l \wedge \neg r]; r\uparrow), ([m \wedge \neg s]; s\uparrow); l\downarrow, m\downarrow]$. We cannot allow the $L-R$ block's cycling twice before the $M-S$ block has had its say. This is not hard to do; we change the pulse generator so that it will have to be "armed" before it will cycle. We do this by making the pullup of rf conditional on the arming signal, which we call Rx. But is then the PR $r_{-} \rightarrow rf\uparrow$ necessary? Not if we guarantee that $Rx\uparrow$ can cause $rf\uparrow$ only after r_{-} would have caused it in the naïve design; and this is easy to do, because we know exactly when r_{-} will go back up, viz. at transition 6.

6.1.3 Solution

The naïve design works properly and efficiently when l and m are synchronized; as long as they are synchronized, Rx must re-arm the pulse generator at the same as in the naïve design. When l and m are not synchronized, Rx needs to re-arm both pulse generators exactly when the later of l and m should have re-armed it. From this discussion, it is obvious that Rx can be a delayed version of $R4$ since $R4$ already waits for the later of l and m.

Thus we arrive at a satisfying design that not only can be generalized to multiple outputs but also does away with the pesky series n-transistors that were required by the static-logic design when $r\_$ carries data. The production rules for rf become

$$
\begin{aligned}
&(4)\neg r\_ &\rightarrow\ & rf\!\downarrow\\
&(?)Rx &\rightarrow\ & rf\!\uparrow.
\end{aligned}
$$

By comparing the transition time-indices of Rx, $R4$, and the inputs, we find that $rf\!\uparrow$ should be enabled at $t = 10$, whence we deduce that

$$
\begin{aligned}
&(2)R4 &\rightarrow\ & R6\!\uparrow\\
&(2)\neg R4 &\rightarrow\ & R6\!\downarrow\\
\\
&(4)\neg r\_ &\rightarrow\ & rf\!\downarrow\\
&(4)R6 &\rightarrow\ & rf\!\uparrow
\end{aligned}
$$

will do the job.

For the scheme to work completely, we must eliminate the possibility that the inputs cause the path $re \wedge l \rightarrow r\_\!\downarrow$ to turn on at the same time that rf is pulling $r\_$ up. This is our first encounter with the problem because of the fortuitous (and fortunate) way that the timing of rf always aligned with that of re in the bit generator and in the left-right buffer. The solution lies in adding a foot transistor to the PR for $r\_\!\downarrow$; the foot's being gated by rf will prevent the unwanted interference (in the simple circuits of Chapter 5, the foot transistor is not required because the re node cuts off the pulldown path; since no other outputs are being generated, the pulse generator always re-arms immediately—eight transitions—after the output is produced). Observe that the foot transistor is required in an input-output domino-block if and only if there is in the process more than one input-output block; i.e., it is required for *all* output channels if and only if there are in total *two or more* output channels. No extra foot transistor is required in the ρ block.

We should of course make the corresponding changes for the $S - M$ block; we can also introduce a signal $R8$, defined in analogy with $R6$, thus removing the combinational pullup for ρ; as far as we know at present, the only reason we should do this is to maintain the similarity between the $L - R$ and $M - S$ pulse generators on the one hand and the $\rho - Rx$ pulse generator on the other; but see Section 6.3 for a better reason.

Summing up, we have the PRS for $DBUF$:

$$
\begin{array}{llll}
(3)\neg r \rightarrow re\uparrow & \neg r_- \rightarrow rv\uparrow \\
rf \wedge re \wedge l \rightarrow r_-\downarrow & r_- \rightarrow rv\downarrow \\
\neg r_- \rightarrow r\uparrow & (3)\neg s \rightarrow se\uparrow \\
(4)\neg r_- \rightarrow rf\downarrow & sf \wedge se \wedge m \rightarrow s_-\downarrow \\
(3)r \rightarrow re\downarrow & \neg s_- \rightarrow s\uparrow \\
\neg rf \rightarrow r_-\uparrow & (4)\neg s_- \rightarrow sf\downarrow \\
(4)R6 \rightarrow rf\uparrow &
\end{array}
$$

$$
\begin{array}{ll}
(3)s \rightarrow se\downarrow & (4)\neg\rho \rightarrow \rho f\downarrow \\
\neg sf \rightarrow s_-\uparrow & R4 \rightarrow l\downarrow, m\downarrow \\
(4)R6 \rightarrow sf\uparrow & \neg\rho f \rightarrow \rho\uparrow \\
\neg s_- \rightarrow sv\uparrow & \rho \rightarrow R4\downarrow \\
s_- \rightarrow sv\downarrow & (4)\rho \rightarrow \rho f\uparrow \\
rv \wedge sv \rightarrow \rho\downarrow & (2)R4 \rightarrow R6\uparrow \\
\neg\rho \rightarrow R4\uparrow & (2)\neg R4 \rightarrow R6\downarrow\ .
\end{array}
$$

6.1.4 Timing assumptions

With the addition of the $R6$ circuitry, some of the timing constraints that were present in the simple circuits have become *easier* to satisfy: this is good because it may be more difficult to predict the delays in these more complicated circuits. Specifically, the two constraints mentioned in Section 5.3.3.3 are now easier to satisfy: the reset pulse's being delayed now cannot cause the circuit to double-latch the inputs, because the reset pulse is used for re-arming the pulse generators, which must happen before another output can be produced; similarly, adding the foot transistor removes the need for re's switching early enough to keep the circuit from producing another output during the same cycle.

6.2 General logic computations

We need to change very little in the buffer template that we have been studying for it to be used for more general computations. If we consider $DBUF$ from the previous section and compare it with a half-adder, $HADD$, the kinship is obvious because

$$DBUF \;\equiv\; *[\ L?x,\ M?y;\ R!x,\ S!y\ ]\ ,$$

and

$$HADD \equiv *[\ A?a,\ \ B?b;\ \ S!(a+b)_0,\ \ D!(a+b)_1\ ]\ ,$$

where the subscripts denote bit indexing. If we implement the computation of $a + b$ directly on the input rails, the calculation is self-synchronizing; i.e., no result will be produced till the inputs have arrived. The only PRs for *HADD* that we need state are

$$se \wedge sf \wedge (a.0 \wedge b.0 \vee a.1 \wedge b.1) \ \rightarrow\ s\_.0\downarrow$$
$$se \wedge sf \wedge (a.1 \wedge b.0 \vee a.0 \wedge b.1) \ \rightarrow\ s\_.1\downarrow$$

$$de \wedge df \wedge (a.0 \vee b.0) \ \rightarrow\ d\_.0\downarrow$$
$$de \wedge df \wedge (a.1 \wedge b.1) \ \rightarrow\ d\_.1\downarrow;$$

the reader can easily infer the rest from the descriptions of the left-right buffer and *DBUF*.

6.2.1 Inputs whose values are not used

We can only trust the simple compilation, which appears to work in the case of *HADD* and *DBUF*, when the logic computation is itself enough to implement the required handshaking behavior. When the logic computation does not suffice for this, we shall have to add further circuitry.

An example will clarify. Let us consider a circuit that generates only the carry output of a full-adder,

$$CADD \equiv *[\ A?a,\ \ B?b,\ \ C?c;\ \ D!(a+b+c)_1\ ]\ ,$$

where the subscript "1" denotes bit indexing. By following the procedure we used for *HADD*, we should arrive at the following:

$$de \wedge df \wedge (a.0 \wedge b.0 \vee a.0 \wedge c.0 \vee b.0 \wedge c.0) \ \rightarrow\ d\_.0\downarrow$$
$$de \wedge df \wedge (a.1 \wedge b.1 \vee a.1 \wedge c.1 \vee b.1 \wedge c.1) \ \rightarrow\ d\_.1\downarrow$$

What is the HSE that we must implement? Omitting the details of the data computations, we can say that it is at least

$$\begin{array}{l} *[\ [a.0\ \vee\ a.1]\ ,\ [b.0\ \vee\ b.1]\ ,\ [c.0\ \vee\ c.1]\ ; \\ \quad [\ \ldots \longrightarrow d.0\uparrow \| \ldots \longrightarrow d.1\uparrow]\ ,\ \ a.0\downarrow, a.1\downarrow, b.0\downarrow, b.1\downarrow, c.0\downarrow, c.1\downarrow\ ]\ . \end{array}$$

But neither $a.0 \wedge b.0 \vee a.0 \wedge c.0 \vee b.0 \wedge c.0$ nor $a.1 \wedge b.1 \vee a.1 \wedge c.1 \vee b.1 \wedge c.1$ actually implements $[a.0 \vee a.1], [b.0 \vee b.1], [c.0 \vee c.1]$, whence we must believe that something is amiss with this implementation of *CADD*.

6.2.1.1 Aside: Comparison with QDI precharge half-buffer

The following discussion has been prepared for those familiar with the implementation of the QDI precharge half-buffer (PCHB) and the QDI weak-condition half-buffer [43].

We may write the simple (dataless) QDI process[2] $*[A?\_, B?\_, C!\_]$ in terms of HSE as

$$*[\ ([ai];\ ao\uparrow;\ [\neg ai];\ ao\downarrow)\ ,\ ([bi];\ bo\uparrow;\ [\neg bi];\ bo\downarrow)\ ,\ (co\uparrow;\ [ci];\ co\downarrow;\ [\neg ci])\ ]\ ;$$

at present, we shall only be concerned with the inputs ai and bi. The handshake on an input, $[ai]; ao\uparrow; [\neg ai]; ao\downarrow$, may be broken down into the rising-edge input completion (also called *input validity*) $[ai]$, the acknowledgment $ao\uparrow$, the falling-edge input completion (also called *input neutrality*) $[\neg ai]$, and the reset phase of acknowledgment $ao\downarrow$. If data is used instead of merely bare handshakes, then $[ai]$ becomes instead $[a0 \vee a1 \vee \cdots]$, and $[\neg ai]$ becomes $[\neg a0 \wedge \neg a1 \wedge \cdots]$.

The essential insight that establishes that the PCHB compilation is often superior to the weak-condition half-buffer (WCHB) compilation is that it is unnecessary and usually unwanted to perform, as the WCHB does, the waits required by a process's handshaking specification in the same operators as the logic computation is performed. Performing the waits with these operators often means strengthening the operators, which reduces their performance for two reasons: it adds extra transistors in series, and it means that inputs that are not required for the computation of the outputs are still waited for; and while our having to insert these waits is troubling enough, in the WCHB we also have to insert neutrality waits for the down-going phase of the handshake; the terror when we realize that this can compile to one additional p-transistor in series for every rail in every input channel!

The PCHB avoids the completion difficulties of the WCHB by our compiling the two functions of handshaking and completion into separate circuitry; the two parts are joined on the input with isochronic forks, and on the output synchronized by a C-element. The very simple requirements on the handshaking part allow an implementation that can be decomposed into OR-gates and C-elements; these operators can be further decomposed, if that should be deemed necessary.

If we observe a PCHB circuit, e.g., an implementation of our offending *CADD*, in operation, we shall see the following. Sometimes, the logic part of the circuit uses enough information about the inputs in computing its outputs that the input validity may thence be inferred; since, however, the logic part has no p-transistors, it cannot possibly compute the input neutrality. In contrast, the completion part of the circuit always checks both the input validity and the input neutrality.

In a STAPL circuit, input neutrality need not be checked. In effect, our timing constraints guarantee that the inputs are neutral when required—no p-transistors are required for this, and this is of course one of the reasons that STAPL circuits are faster than QDI circuits.

In summary: the QDI circuit often needs only the p-transistors in the completion network; the n-transistors are added so that the completion network itself shall be delay-insensitive. The STAPL

[2]Traditionally, authors—Martin among others—have used the notation A to signify a dataless synchronization, thus emphasizing that a dataless synchronization is symmetric. This is sensible when there is no "direction" in the synchronization. But communications almost always have a definite send-receive direction in the design style that we are exploring in this thesis—the only reason for introducing dataless synchronizations is, with few exceptions, pedagogy. This is why we shall still normally identify the direction of the synchronization, i.e., the party that sends ($A!\_$) and the one that receives ($A?\_$).

circuit does not need the p-transistors; hence we can also omit the n-transistors, except in those unfortunate circumstances (as in *CADD*) where the logic computation is insufficient for always checking for the arrival of inputs. In other words, in STAPL circuits the need for the completion circuitry is much diminished, but not eliminated.

6.2.1.2 Solving the problem

Obviously we shall need to add completion circuitry to solve the problem posed by the *CADD* compilation. One way of doing this is simply to make the circuit's producing the outputs always depend on its receiving inputs on *all* channels: this amounts to checking for the arriving of unneeded inputs in the compute logic. Sometimes this is the preferred way of doing things; but it reduces the slack and increases the number of transistors in series, so it often is not.

Indeed, the bit-bucket compilation suggests what needs to be done if we want to complete the inputs without complicating the computing logic. We add to the circuit a *dummy output* δ whose task is checking that all the inputs have arrived before the Rx reset pulses are generated. While it may sometimes be possible to do tricky things with the dummy output, it seems simplest to make it an unconditional output that indicates only that all the inputs have arrived and nothing else; if the normal outputs always check the arrival of certain inputs but not others, the dummy output needs only check the others. We can specify it thus at the HSE level:

$$\begin{array}{l} *[\ \ ([a.0 \lor a.1], [b.0 \lor b.1], [c.0 \lor c.1]; \delta{\downarrow}), [\ \ldots \longrightarrow d.0{\uparrow} [\!] \ldots \longrightarrow d.1{\uparrow}]\ \ ; \\ \quad \delta{\uparrow}, a.0{\downarrow}, a.1{\downarrow}, b.0{\downarrow}, b.1{\downarrow}, c.0{\downarrow}, c.1{\downarrow}\ ]\ \ ; \end{array}$$

the negated sense of δ allows implementing it as we implement the $a\_.x$ operators. The implied PRS is

$$\begin{array}{l} se \land sf \land (a.0 \land b.0 \lor a.1 \land b.1) \ \rightarrow\ s\_.0{\downarrow} \\ se \land sf \land (a.1 \land b.0 \lor a.0 \land b.1) \ \rightarrow\ s\_.1{\downarrow} \\ \\ de \land df \land (a.0 \lor b.0) \ \rightarrow\ d\_.0{\downarrow} \\ de \land df \land (a.1 \land b.1) \ \rightarrow\ d\_.1{\downarrow} \\ \\ \delta f \land (a.0 \lor a.1) \land (b.0 \lor b.1) \land (d.0 \lor d.1) \ \rightarrow\ \delta{\downarrow}\ ; \end{array}$$

since δ does not leave the process, no δe node need exist. Note that we get to keep the desirable property that no p-transistors are required for the completion of the unused inputs.

6.2.1.3 Unconditional process template

We now know how to implement as a STAPL circuit any process of the form[3]

[3] We may deplore the outward similarity of the parallel "," in $<, i :: L_i?x_i>$ to the merely syntactic function-call "," in $f_j(<, i :: x_i>)$. Making matters worse, the last comma on the program line is part of the English text; written

$$*[\ \langle ,i :: L_i?x_i \rangle\ ;\ \langle ,j :: R_j!f_j(\langle ,i :: x_i \rangle) \rangle\ ]\ ,$$

as long as the process is reasonably simple; what is reasonable varies with technology and application.

In terms of circuits, the unconditional process template is schematically illustrated by Figure 6.1. Any unconditional STAPL process can be built out of these pieces: one output block for generating the data on each output channel, an input-clearing block for clearing all the inputs, and the acknowledgment block for implementing the necessary sequencing in the process. Dummy channels as mentioned in the previous section are simply implemented as output channels without the output p-transistors or flow-control NOR-gates. The areas marked "LD" in the figure will contain the circuitry for evaluating the output functions.[4]

6.3 Conditional communications

As is obvious from synchronous systems, in which we may consider the value that each node assumes on every clock cycle as the result of a communication, the unconditional template we have so far developed is enough for building a system that implements any arbitrary high-level specification. As is equally obvious from the CHP programming-model, however, our using only unconditional processes is not the only—or even the obvious—way to build message-passing systems: lately, even synchronous designers are investigating conditionally message-passing systems; they do this in the hope that they will thus be able to avoid the costs involved in communicating unnecessarily often.

It is not always clear when introducing conditionality into a system is a good idea, and determining whether it is a good idea in a given situation would take us too far afield from the main subject of this thesis; we shall simply assume that conditionality may be specified in the CHP, and that when it is specified, it must be implemented.

6.3.1 The same program can be expressed in several ways

An elementary example of conditional communication is the *split,* viz.,

$$SPLIT \ \equiv\ *[C?c,\ L?x;\ [c=0 \longrightarrow R0!x\ [\!]\, c=1 \longrightarrow R1!x\ ]]\ ;$$

another is the *merge,*

$$MERGE \ \equiv\ *[C?c;\ [c=0 \longrightarrow L0?x\ [\!]\, c=1 \longrightarrow L1?x\ ];\ R!x\ ].$$

The asymmetry (most noticeably, the differing number of semicolons) between *SPLIT* and *MERGE* is somewhat illusory, due in part to syntactic issues with the CHP language. We might, e.g., write

thus, the program satisfies the author's sense of order but admittedly leaves him answerable to the charge of pedantry.

[4] The reason the circuit in Figure 6.1 does not generate $R6$ directly from $R4$ is to avoid that the designer's imprudently overloading $R4$ should affect the pulse shape on $R6$.

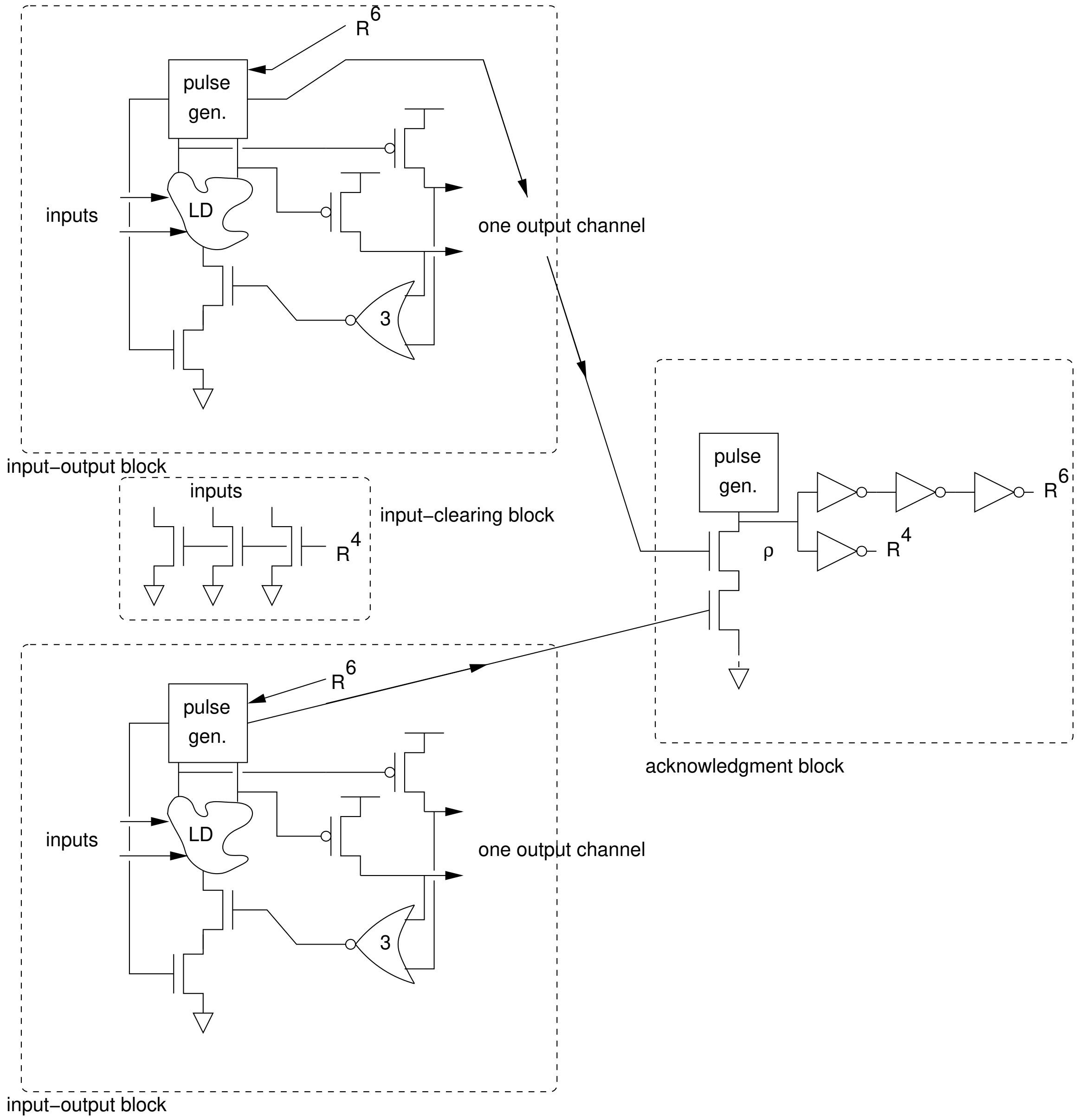

Figure 6.1: Schematic version of unconditional STAPL template.

$$SPLIT \;\equiv\; *[C?c;\;\; [c = 0 \longrightarrow R0!(L?) \;[]\, c = 1 \longrightarrow R1!(L?)\;]\;],$$
$$MERGE \;\equiv\; *[C?c;\;\; [c = 0 \longrightarrow R!(L0?) \;[]\, c = 1 \longrightarrow R!(L1?)\;]\;];$$

in the slack-elastic model, rewriting like this in no way changes the meanings. Inventive persons have carried this argument further, noticing that the receiving of c into an internal variable is needless; the output or outputs depend on c as much as they depend on l, only in a different way. If we insisted on our code's reflecting the symmetry between c and l, we should for example have that, written with a single, implied semicolon,

$$\begin{aligned}
&SPLIT \;\equiv\\
&\quad *[[\;\overline{C=0} \;\longrightarrow\; C?_, R0!(L?)\\
&\qquad []\;\overline{C=1} \;\longrightarrow\; C?_, R1!(L?)\\
&\quad]]\;.
\end{aligned}$$

While *SPLIT* seems at home with this transformation, the same could not be said for many more complex processes; the synchronization behavior implied by our writing the processes in this way may be closer to what we aim at in our HSE-PRS compilation, but the semantic advantage is outweighed by the degree that the code is obscured to, and we hence shall usually take the position that slack elasticity allows us: all the programs for *SPLIT* we have given in this section are equivalent and should compile the same.

6.3.2 Simple techniques for sends

There is a wide variety of *ad hoc* techniques available for adding conditional sends to QDI process templates; we shall briefly study the simplest one before proceeding to general methods.

Consider the "filter" process

$$FILTER \;\equiv\; *[C?c,\;\; L?x;\;\; [c = 0 \longrightarrow R!x \;\,[]\, c = 1 \longrightarrow \mathbf{skip}\;]\;]\;\;.$$

The simplest QDI implementation of this process is arrived at by starting with a QDI left-right buffer and to it adding an extra, dummy output-rail of $r_{\_}$; this appears not to work so well in the STAPL family, because it appears that we shall need to add a dummy block for completing L when $c = 1$. However, that the dummy block is required is a property of the output function—not of its implementation; the dummy block would in any case be required because the outputs do not always complete the inputs. For other circuits with conditional outputs, the dummy block may not be required, because the input could be completed by some other, unconditional, output; or conversely the dummy block may be needed because the output functions do not compute the input completion even when the outputs are generated, as happened with *CADD*.

We give the PRS for $r_{\_}$:

$$re \wedge c.0 \wedge l.0 \ \rightarrow \ r\_.0\downarrow$$
$$re \wedge c.0 \wedge l.1 \ \rightarrow \ r\_.1\downarrow$$
$$c.1 \qquad\qquad \rightarrow \ r\_.\infty\downarrow;$$

using re in the PR for $r\_.\infty\downarrow$ is quite legal, but unnecessary. (Sharp eyes and minds will notice that replacing the PR for $r\_.\infty\downarrow$ with $c.1 \wedge (l.0 \vee l.1) \rightarrow r\_.\infty\downarrow$ will, in this special case, obviate the dummy block.) Being only an internal signal in *FILTER*, $r\_.\infty$ has no output p-transistor.

6.3.3 General techniques for conditional communication-actions

We may solve the problem of conditional outputs by adding a dummy rail, but this is not always the most efficient way of doing it; thinking about the behavior of the circuit, we should realize that while no outputs are generated when $r\_.\infty$ cycles, there is still much internal activity going on. The situation becomes especially bad if most output circuits are not generating outputs on a given cycle—they shall still have to cycle their dummy rails.

More seriously, the dummy-rail technique helps not at all if what we want is a conditional *input.* That satisfying this desire is more difficult is clear if we consider that a conditional input implies, among other things, conditional reset pulses. We cannot sidestep this difficulty by resetting inputs that are not being used because we are required to keep our circuits slack-elastic: inputs must be allowed to arrive arbitrarily early; hence resetting inputs that are not being used would violate the handshaking specification.

6.3.3.1 A general method

We shall solve the problem of general conditional communications by introducing another domino-logic block. This domino-logic block will compute which inputs and outputs the process uses on a given cycle. We shall avoid introducing new fundamental constructs by using the same kind of domino block for the conditions as for the logic computations; this means that this *conditions block* shall have a one-hot (i.e., 1-of-n) output. In other words, depending on the input, the conditions block computes which of several possible *communication patterns* is being followed.

We shall illustrate the method by implementing *SPLIT* and *MERGE*.

6.3.3.2 *SPLIT* implementation

The *SPLIT* process has only conditional outputs; this will make the compilation simpler than for *MERGE*. We first introduce an internal channel $p\_$ that denotes the communication pattern. There are two mutually exclusive communication patterns followed by *SPLIT*: receive on C, receive on L, send on $R0$; and receive on C, receive on L, send on $R1$: we call the former $p\_.0$, and the latter

$p_{-}.1$. Furthermore, we may consider p_{-} as the manifestation of an internal, unconditional channel P, whence we may summarize the communication patterns in the table:

Condition	**When true**	**Channels exercised**
$p.0$	$c.0$	$C\ L\ R0\ P$
$p.1$	$c.1$	$C\ L\ R1\ P$

Why can we not use c directly instead of generating the middleman p_{-}? Admittedly, *SPLIT* is a special case where we could implement the conditional communications more simply; but one of the main problems is that if we try to use c directly, it becomes defined and needs to be reset at the wrong times, viz. in each case two transitions after the other inputs. (Recall that the logic block synchronizes c and l because they are both used in the same production rules; hence we cannot simply require that c be presented two stages later by the environment.) This is an unwanted asymmetry in the circuit; furthermore, the additional two stages of delay introduced by the conditions block also allow our using much more complex conditions.

We make all the activity in the acknowledge block conditional on the communication pattern; thus, ρ, $R4$, and $R6$ become one-hot codes (one-“cold” for ρ).

In any case, the PRS consists of the usual compilation for the outputs and additionally of conditional-communication circuitry. First, the PRs for p_{-} and p are

$$
\begin{aligned}
pf \wedge c.0 &\rightarrow p_{-}.0\downarrow \\
pf \wedge c.1 &\rightarrow p_{-}.1\downarrow \\
\neg p_{-}.0 &\rightarrow p.0\uparrow \\
\neg p_{-}.1 &\rightarrow p.1\uparrow .
\end{aligned}
$$

Secondly, since we need separate resets for the output channels, we can re-use them for resetting the p's, so we have

$$
\begin{aligned}
R6.0 &\rightarrow p.0\downarrow \\
R6.1 &\rightarrow p.1\downarrow .
\end{aligned}
$$

Thirdly, the PRs for re-arming the pulse generators are now different for the different outputs (strictly speaking, this is not required for the *SPLIT*, but in the general case, slack-elasticity requires it; also, if a single output channel participates in several communication patterns, each one of the corresponding $R6$'s must be able to reset it), so that the pulse generators now become

$$\begin{array}{lcl}
(4)\neg r0.0_ \vee \neg r0.1_ & \rightarrow & r0f{\downarrow} \\
\neg r0.0_ \vee \neg r0.1_ & \rightarrow & r0v{\uparrow} \\
(4)R6.0 & \rightarrow & r0f{\uparrow} \\
R6.0 & \rightarrow & r0v{\downarrow}
\end{array}$$

$$\begin{array}{lcl}
(4)\neg r1.0_ \vee \neg r1.1_ & \rightarrow & r1f{\downarrow} \\
\neg r1.0_ \vee \neg r1.1_ & \rightarrow & r1v{\uparrow} \\
(4)R6.1 & \rightarrow & r1f{\uparrow} \\
R6.1 & \rightarrow & r1v{\downarrow}\,.
\end{array}$$

Lastly, the PRs for ρ, $R4$, and $R6$ are

$$\begin{array}{lcl}
p.0 \wedge r0v & \rightarrow & \rho.0{\downarrow} \\
p.1 \wedge r1v & \rightarrow & \rho.1{\downarrow} \\
\rho.0 & \rightarrow & R4.0{\downarrow} \\
\rho.1 & \rightarrow & R4.1{\downarrow} \\
(2)R4.0 & \rightarrow & R6.0{\uparrow} \\
(2)\neg R4.0 & \rightarrow & R6.0{\downarrow} \\
(2)R4.1 & \rightarrow & R6.1{\uparrow} \\
(2)\neg R4.1 & \rightarrow & R6.1{\downarrow}\,.
\end{array}$$

Happily, all the input channels are completed by the outputs. In this compilation, all the logic blocks (including the conditions block) require the extra foot transistor; as always, the ρ block does not. Schematically, the arrangement is shown in Figure 6.2.

We may generalize the conditional communications of the $SPLIT$, thus implementing any process of the type

$$*[\ \langle, i :: L_i?x_i \rangle\ ;\ \langle, j :: [G_j(\mathbf{x}) \longrightarrow R_j!f_j(\mathbf{x}) [] \neg G_j(\mathbf{x}) \longrightarrow \mathbf{skip}] \rangle\ ]\ ,$$

where we for simplicity write $\mathbf{x}$ for $\langle, i :: x_i \rangle$.

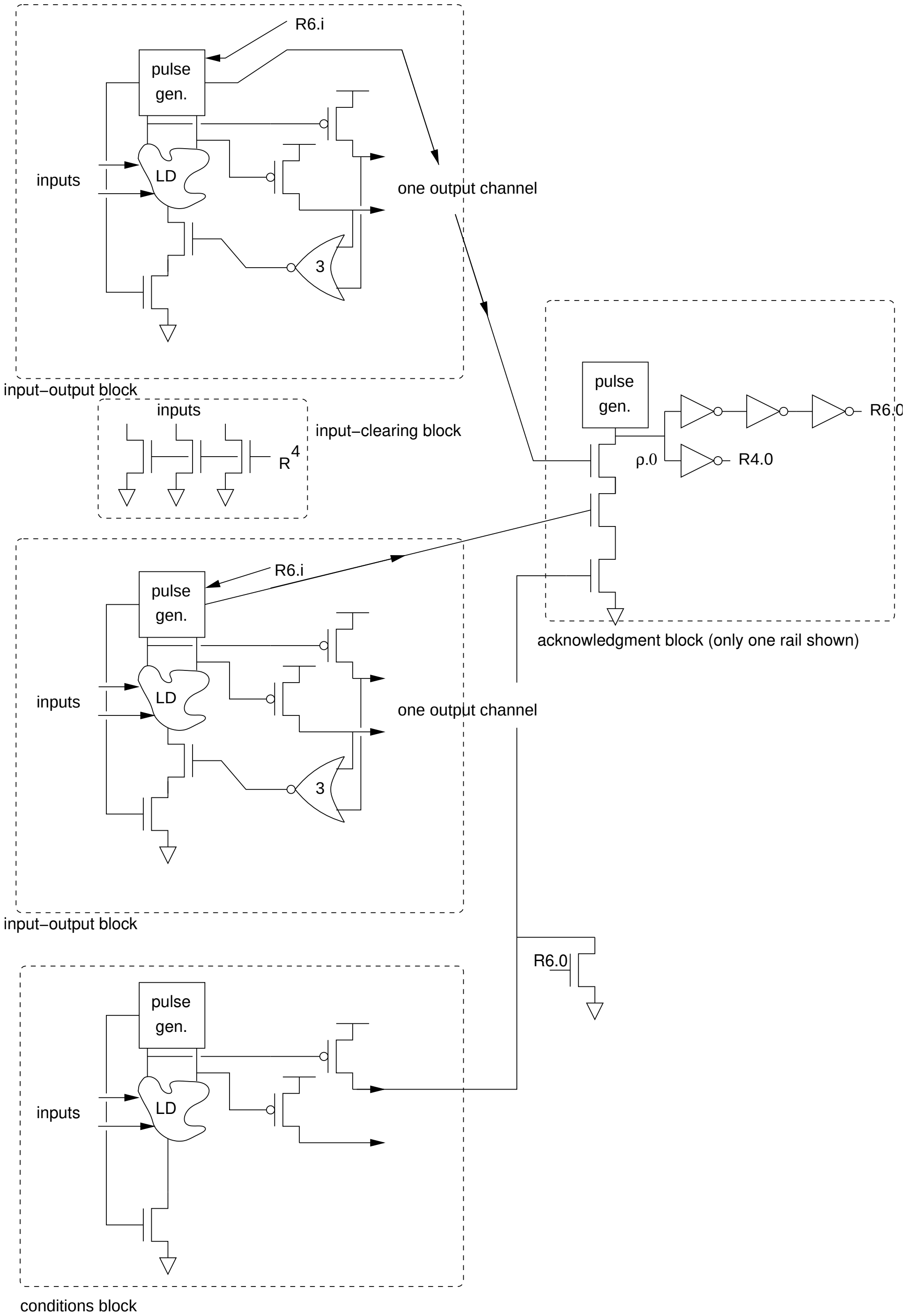

Figure 6.2: Schematic version of conditional STAPL template.

6.3.3.3 *MERGE* implementation

The *MERGE* is similar to the *SPLIT* except for its conditional inputs, to wit

$$\mathit{MERGE} \equiv *[C?c;\ [c = 0 \longrightarrow R!(L0?)\ [\!]\ c = 1 \longrightarrow R!(L1?)\ ]\ ],$$

which compiles (in the dual-rail case) to

$$\begin{array}{l} *[[\ c.0 \wedge \neg(r.0 \vee r.1) \wedge l0.0 \longrightarrow r.0\uparrow,\ l0.0\downarrow,\ c.0\downarrow \\ \ [\!]\ c.0 \wedge \neg(r.0 \vee r.1) \wedge l0.1 \longrightarrow r.1\uparrow,\ l0.1\downarrow,\ c.0\downarrow \\ \ [\!]\ c.1 \wedge \neg(r.0 \vee r.1) \wedge l1.0 \longrightarrow r.0\uparrow,\ l1.0\downarrow,\ c.1\downarrow \\ \ [\!]\ c.1 \wedge \neg(r.0 \vee r.1) \wedge l1.1 \longrightarrow r.1\uparrow,\ l1.1\downarrow,\ c.1\downarrow \\ \ ]]\ . \end{array}$$

We shall need separate reset signals for the channels $l0$ and $l1$; since we must in any case have these separate reset signals, we can take advantage of them and give $c.0$ and $c.1$ separate resets; however, introducing separate reset signals for each of the values of the channels $l0$ and $l1$ is needless and would lead to a more complex circuit. Hence the program we implement is better described as

$$\begin{array}{l} *[[\ c.0 \wedge \neg(r.0 \vee r.1) \wedge l0.0 \longrightarrow r.0\uparrow,\ l0.0\downarrow, l0.1\downarrow,\ c.0\downarrow \\ \ [\!]\ c.0 \wedge \neg(r.0 \vee r.1) \wedge l0.1 \longrightarrow r.1\uparrow,\ l0.0\downarrow, l0.1\downarrow,\ c.0\downarrow \\ \ [\!]\ c.1 \wedge \neg(r.0 \vee r.1) \wedge l1.0 \longrightarrow r.0\uparrow,\ l1.0\downarrow, l1.1\downarrow,\ c.1\downarrow \\ \ [\!]\ c.1 \wedge \neg(r.0 \vee r.1) \wedge l1.1 \longrightarrow r.1\uparrow,\ l1.0\downarrow, l1.1\downarrow,\ c.1\downarrow \\ \ ]]\ . \end{array}$$

We compile *MERGE* in much the same way as *SPLIT*. If we introduce P the same way as before, the condition table for *MERGE* becomes

Condition	When true	Channels exercised
$p.0$	$c.0$	$C\ L0\ R\ P$
$p.1$	$c.1$	$C\ L1\ R\ P$

The condition computation is identical to *SPLIT*'s, but we shall have to generate the two reset signals from two separate ρ signals; the PRS becomes

$$
\begin{aligned}
pf \wedge c.0 &\rightarrow p\_.0\downarrow \\
pf \wedge c.1 &\rightarrow p\_.1\downarrow \\
\neg p\_.0 &\rightarrow p.0\uparrow \\
\neg p\_.1 &\rightarrow p.1\uparrow \\
\\
p.0 \wedge rv &\rightarrow \rho.0\downarrow \\
p.1 \wedge rv &\rightarrow \rho.1\downarrow \\
\\
\neg \rho.0 &\rightarrow R4.0\uparrow \\
\neg \rho.1 &\rightarrow R4.1\uparrow \\
\\
(2) R4.0 \vee R4.1 &\rightarrow R6\uparrow \\
\\
R6 &\rightarrow p.0\downarrow \\
R6 &\rightarrow p.1\downarrow \\
(2) R6 &\rightarrow R8\uparrow \\
(2) \neg R6 &\rightarrow R8\downarrow \\
\\
(4) R8 &\rightarrow \rho.0\uparrow \\
(4) R8 &\rightarrow \rho.1\uparrow ,
\end{aligned}
$$

where we see that introducing $R8$ becomes necessary (see Section 6.1.3) if we insist on avoiding the long pulldowns of a static implementation of ρf. The only drawback to our introducing $R8$ in this way is that part of the path to the ρ pullups becomes dynamic, but this is a drawback that we have long ago accepted for the first-stage logic blocks, so why not here too?

Conceptually, we should not find difficult generalizing the template of Section 6.2.1.3 so that it covers conditional receives; however, the CHP language does not have the necessary constructs for easily and unambiguously describing the semantics that we can implement, whence we defer this issue to the next chapter and to Appendix A (see p. 182).

6.4 Storing state

Any method of digital design that aims at components' being used repeatedly must allow state to be stored across circuit iterations. A simple circuit that requires storing state is given by the *alternator,* which is specified by the CHP program

$$ALT \equiv *[\ R!0;\ R!1\ ]\ .$$

During an execution of P, it may be that ALT has lately executed $R!0$ and will presently execute $R!1$; that this is so and not the other way around (i.e., that ALT has lately executed $R!1$, *et seq.*) need not be a fact inferable from any outside information. Therefore P must store state within itself.

6.4.1 The general state-storing problem

We have expressed all our programming problems in terms of repetitive programs with two parts: receiving inputs, then producing results. In these general terms, a circuit's storing state becomes necessary when actions in a later iteration depend on events in an earlier iteration. We shall extend the template given in Section 6.2.1.3 to[5]

$$
\begin{aligned}
&*[\ \langle ,i :: L_i?y_i \rangle\ ,\ \langle ,k :: x_k := x'_k \rangle\ ;\\
&\quad \langle ,j :: R_j!f_j(\mathbf{y},\mathbf{x}) \rangle\ ,\ \langle ,k :: x'_k := g_k(\mathbf{y},\mathbf{x}) \rangle\\
&\ ]\ ,
\end{aligned}
$$

whence it is already clear that the updating of a state variable is similar to receiving and sending values on a channel—unsurprisingly so, since sending and receiving together implement a distributed assignment. We may note in passing that complicating matters with conditional actions is unnecessary since

$$\ldots;\ [\ G_0 \longrightarrow x := g_0\ \talloblong\ \neg G_0 \longrightarrow \mathbf{skip}\ ]\ ;\ldots$$

is identical in effect to

$$\ldots;\ [\ G_0 \longrightarrow x := g_0\ \talloblong\ \neg G_0 \longrightarrow x := x\ ]\ ;\ldots;$$

hence all state variable updates may, without loss of generality, be made unconditional.

How would one use the template? ALT will serve as an example. We need to rewrite ALT so that it contains only a single send action per iteration; we replace the sequencing implied by the semicolon with loop iterations and introduce a state variable for keeping track of the program's progress with respect to the semicolon. The similarity to software compilation into assembly language statements is clear: one could introduce a program counter to fold an arbitrarily complex sequential program into a single statement. We have

$$
\begin{aligned}
&ALT \equiv x := 0;\\
&*[[\ x = 0 \longrightarrow R!0;\ x := 1\\
&\quad \talloblong\ x = 1 \longrightarrow R!1;\ x := 0\\
&\ ]]\ .
\end{aligned}
$$

A final rewriting will merely introduce the special intermediate variable x'; hence

$$
\begin{aligned}
&ALT \equiv\\
&x' := 0;\ *[\ x := x'\ ;\ R!x,\ [\ x = 0 \longrightarrow x' := 1\ \talloblong\ x = 1 \longrightarrow x' := 0\ ]\ ]\ .
\end{aligned}
$$

[5]If the program seems puzzling, please see Section 6.3.3.2 and also the footnote to Section 6.2.1.3.

6.4.2 Implementing state variables

Observing that state-variable updates are similar to channel sends and receives, we can see that state variables could be implemented with feedback loops—channels that lead from a process P, possibly through buffering, back to P. P would send the updated state variable on, e.g., X' and receive the new value, some time later, on X. This approach works, is easy to implement, and can be reasonably efficient if there are not too many state variables in a system; it also has the advantage that we can in some situations easily add pipelining, since it is now straightforward to pipeline the state variable—the procedure we are alluding to is similar to loop unrolling and software pipelining, and it was extensively used in the MiniMIPS processor.

The high costs of implementing state variables with feedback loops are obvious in the MiniMIPS processor: they are due to the channel mechanism's being more powerful than is required for state variables; in other words, channels carry more information—specifically, synchronization information—than variables that may be read and assigned at any time; they hence work in many situations where (shared) variables are insufficiently powerful; but using channels where variables are sufficiently powerful is wasteful: taking the dynamic slack properties of Williams [85] and Lines [43] into account, we can deduce that implementing a single-bit state variable may require as many as four or five left-right buffers, which adds up to an exorbitant cost of several hundred transistors. In short, the approach may be acceptable for infrequent use in control circuitry, but we should avoid using it in datapath circuitry.

6.4.2.1 Issues with direct implementation

The better way of implementing state variables is implementing them directly. The main obstacle to this is that, till now, all nodes in our STAPL circuits have had the same switching behavior; namely, if we consider an iteration of such a circuit, each node either remains idle or switches with (ideally) a 50 percent duty cycle, which means that in the circuits studied so far, the node goes active for five transitions, then returns to its neutral value for another five transitions, and may go active again at that time;[6] as we have explored the handshake phases may be longer, but not shorter, than five transitions in the presence of external synchronizations.

The symmetric and long pulses that appear in normal asynchronous circuitry are due to the handshake protocols that are used for moving data around. These protocols come at a high cost, but this is often unobjectionable when the computation that is implemented is logically complex, as is often the case in normal control or datapath circuitry. In the case of state variables, however, the mass of extra circuitry that is needed for generating the desired symmetric pulses does in fact give rise to the feedback loops that we have already deprecated.

[6] Our using the terms "active" and "neutral" in place of **true** and **false** indicates that the argument applies to logic that can be either positive or negative.

6.4.3 Compiling the state bit

To find our way out of the timing conundrum, we must abandon the similarity between handshake variables and state variables. We shall instead explore a STAPL circuit with characteristics similar to the state-variable compilation given by Lines for QDI circuits [43]. Our new compilation uses the fact that the state variable is embedded inside a circuit that we already have timing information about; in this way, we can achieve a very simple state-variable implementation that allows of using a simple dual-rail, non-handshaking state-bit, but which at the same time can safely be accessed by its parent circuit according to the parent's handshake timing. We shall see that the timing signals that are necessary for a simple state-variable compilation are already available in the STAPL template.

6.4.3.1 Circuits

The simplest state variable (that does not need to be refreshed) is the SRAM cell, which consists of two cross-coupled inverters and read-write circuitry. Whereas a standard SRAM has pass-gates for reading out and writing the stored bit, we shall use different circuits, which will implement the conversion between handshaking protocols and the state variable.

The state bit is shown conceptually in Figure 6.3.

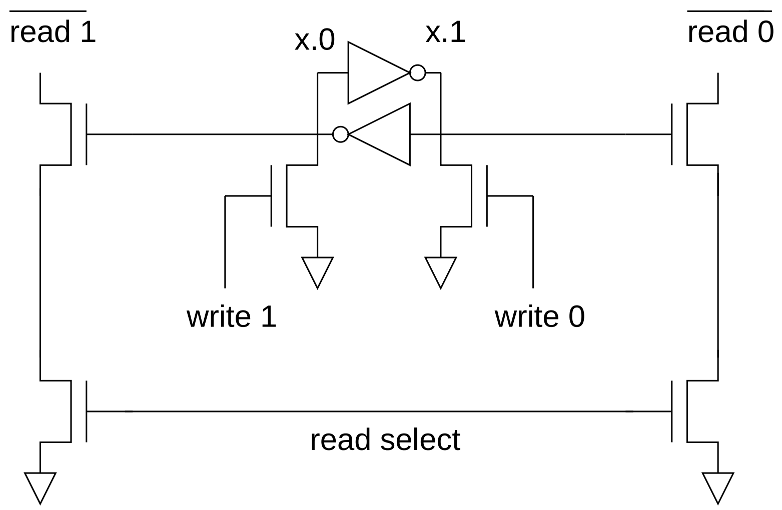

Figure 6.3: Basic state bit.

How can we generate the appropriate control signals for the state bit? Recalling the timing of a STAPL process, we know that once we have generated the outputs (on transition 1 internally, transition 2 for the outputs), we shall have eight transitions for getting the circuit ready for the next set of inputs. We shall find it convenient to synchronize the arrival of the new state-bit value with the next set of data inputs; this allows us the maximum time for computing the new state bit without incurring a delay penalty on the next iteration.

Our first implementation of the STAPL state bit works as follows: on transition 1, the internal nodes begin their active-low pulse; we use p-transistors to write the state bit immediately on transition 2; this being done, the last reset pulse $R8$ copies the new state bit to the input side of the process. This circuit is shown in Figure 6.4.

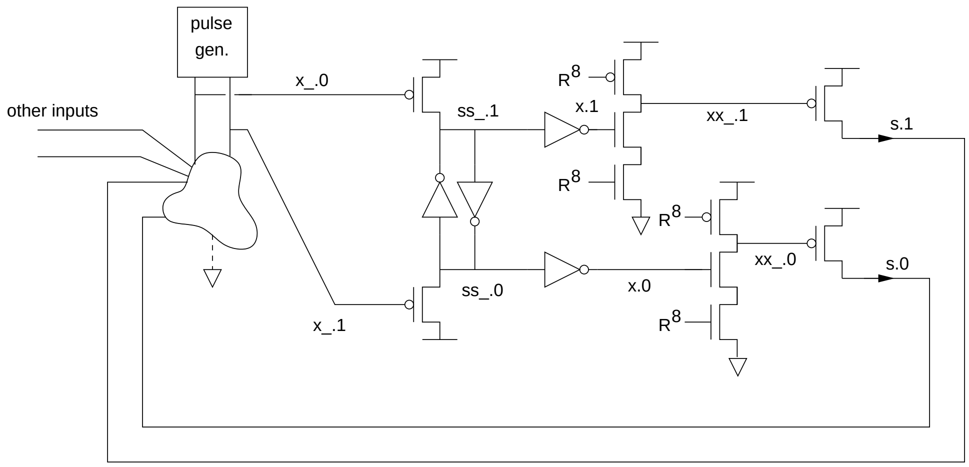

Figure 6.4: Naïve state-variable compilation.

The corresponding PRS is

$$
\begin{array}{llll}
\neg x\_.0 & \rightarrow ss\_.0\uparrow & R8 \wedge x.0 & \rightarrow xx\_.0\downarrow \\
\neg x\_.1 & \rightarrow ss\_.1\uparrow & R8 \wedge x.1 & \rightarrow xx\_.1\downarrow \\
ss\_.0 \wedge x\_.1 & \rightarrow ss\_.1\downarrow(\dagger) & & \\
ss\_.1 \wedge x\_.0 & \rightarrow ss\_.0\downarrow(\dagger) & \neg xx\_.0 & \rightarrow s.0\uparrow \\
 & & \neg xx\_.1 & \rightarrow s.1\uparrow \\
ss\_.0 & \rightarrow x.0\downarrow & & \\
ss\_.1 & \rightarrow x.1\downarrow & \neg R8 & \rightarrow xx\_.0\uparrow \\
\neg ss\_.0 & \rightarrow x.0\uparrow & \neg R8 & \rightarrow xx\_.1\uparrow. \\
\neg ss\_.1 & \rightarrow x.1\uparrow & &
\end{array}
$$

Here, we call the input to the circuit $x\_$; this is the "output" of the parent. Conversely, the output of the circuit, also the "input" of the parent, is called s. The production rules marked (†) are implemented with interference (weak feedback).

6.4.3.2 Problems

What are the problems that introducing this circuit into the STAPL family can give rise to?

We first note that we may have a problem when an input is asserted and it attempts setting the state bit to a new value. When this happens, the n-transistor pulldown chain that is responsible for setting the new value turns on at the same time that the opposing p-transistor is turned on. Hence, we must make arrangements that ensure this situation's happy resolution; this is an instance of a *ratioing assumption.* The only way we can solve this problem (short of using a much more complex implementation) is by careful sizing and verification. This being a familiar problem, it should not concern us too deeply.

Secondly, the worries that we had regarding the unusual timing relationships of the state-variable nodes must be considered. When one side of the cross-coupled inverter pair is pulled down to GND, the other side is pulled up one transition later; it may be a slow transition, but in either case, there is no reason to believe that the timing relationship between this transition and the transitions otherwise seen in STAPL circuits should be predictable. If we compare the behavior of a dual-rail STAPL channel with the state bit, we see that the dual-rail channel ideally is defined five transitions out of ten; the state bit ideally nine or—when it does not change—ten out of ten. The main issue that we need to worry about is the new value's being generated too early and hence its triggering an illegal 1-1 state (because it may be read one cycle early, when the old value is still available).

This is a real problem. $R8$ is, as we know, active (high) for five transitions, going high at transition 8 and low at 13. If the state variable changes, however, the new value of x goes high at transition 4, viz. transition 14 of the previous cycle. Terror strikes! We have only a single transition of delay margin; should x go high a little early or $R8$ be a little slow to reset, then the circuit may enter the illegal 1-1 state, and all is lost.

Since solving the state variable's timing problem by delaying the variable's update would defeat the purpose (this solution would turn the state variable back into some sort of feedback loop with the same kind of timing as a channel), we must use something more unconventional. The problem we wish to avoid can be seen from the following partial trace:

action	**time**
$x.1\uparrow$	4
$R8\uparrow$	8
$xx\_.1\downarrow$	9
$R8\downarrow$	13
$x.0\uparrow$	14

whence we see that if the reset of $R8$ is delayed slightly, then $xx\_.0\downarrow$ may become enabled at time index 14. We introduce an *interlock;* this is a mechanism that keeps the circuit from getting into the 1-1 state. Since $xx\_.1$ has fallen at time index 9, it is the obvious choice for the interlock; the

PRS for the state variable becomes

$$\begin{aligned}
\neg x_.0 &\rightarrow ss_.0\uparrow \\
\neg x_.1 &\rightarrow ss_.1\uparrow \\
ss_.0 \wedge x_.1 &\rightarrow ss_.1\downarrow \\
ss_.1 \wedge x_.0 &\rightarrow ss_.0\downarrow
\end{aligned}$$

$$\begin{aligned}
ss_.0 &\rightarrow x.0\downarrow \\
ss_.1 &\rightarrow x.1\downarrow \\
\neg ss_.0 &\rightarrow x.0\uparrow \\
\neg ss_.1 &\rightarrow x.1\uparrow
\end{aligned}$$

$$\begin{aligned}
R8 \wedge x.0 \wedge xx_.1 &\rightarrow xx_.0\downarrow \\
R8 \wedge x.1 \wedge xx_.0 &\rightarrow xx_.1\downarrow
\end{aligned}$$

$$\begin{aligned}
\neg xx_.0 &\rightarrow s.0\uparrow \\
\neg xx_.1 &\rightarrow s.1\uparrow
\end{aligned}$$

$$\begin{aligned}
\neg R8 &\rightarrow xx_.0\uparrow \\
\neg R8 &\rightarrow xx_.1\uparrow .
\end{aligned}$$

The circuit is shown in Figure 6.5. Compiled thus, the state variable is again resistant to minor timing variations; in fact, the interlock makes the production rules for $xx\_$ more resistant to timing variations than the other parts of the STAPL circuit—the margin on $xx\_$ is nine transitions, rather than the usual five.

6.5 Special circuits

So far, we have seen circuits for buffering data, computing functions, performing conditional communications, and remembering state. These are enough for implementing deterministic computations that take place entirely within the STAPL model. The reader should see—if he does not, he might want to look at the next chapter—that any deterministic logical specification could be implemented straightforwardly in these terms.

The things that most clearly are missing from our model are nondeterministic devices (i.e., arbiters) and devices that allow us to communicate with systems built in different design-styles, e.g., QDI systems. The devices that we shall present as solutions to these problems are different from the ones presented so far in that they are not necessarily intended to be generalized.

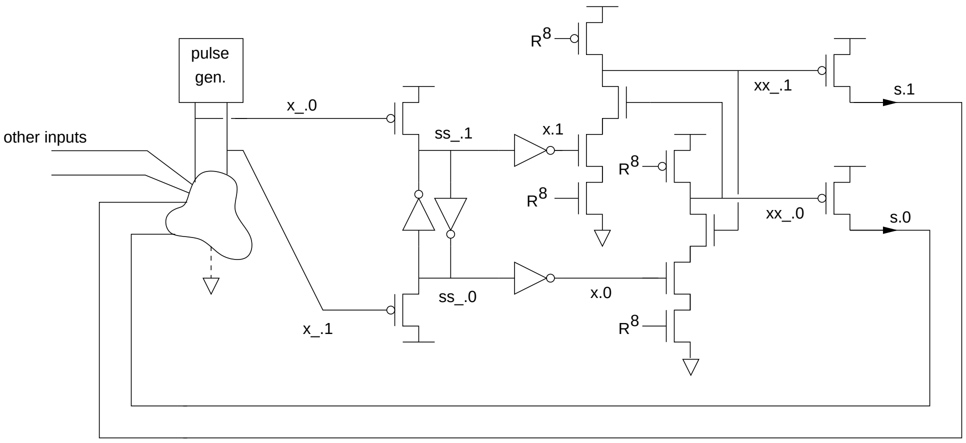

Figure 6.5: Sophisticated state-variable compilation.

6.5.1 Arbitration

Whereas the prudent asynchronous VLSI designer avoids nondeterministic choice when possible, he will find, prejudices notwithstanding, that there are situations in which nondeterministic choice simplifies or allows a more "asynchronous" implementation. In the MiniMIPS for instance, we introduced nondeterminism in two places where it naïvely appeared unnecessary: in merging the cache operations in the off-chip memory-interface, and in the exception mechanism. In the former case, the nondeterminism simplified; in the latter, it allowed a more asynchronous implementation because it allowed the system's being designed without any prior knowledge of actual pipeline depths.

We shall implement arbitration in one way only; this we do with the program

$$
\begin{array}{l}
ARB \;\equiv \\
*[[\;\overline{A} \;\longrightarrow\; A, R!0 \\
\quad |\;\overline{B} \;\longrightarrow\; B, R!1 \\
\;]]\;.
\end{array}
$$

The reader will recognize that this program is at least sufficient, although perhaps not always the most convenient, for implementing most interesting kinds of nondeterministic choice (but see also Section 6.5.2).

We shall use the standard QDI arbiter (Figure 6.6) as the central building block of the STAPL ARB. The only difficulty this causes is that the QDI arbiter takes more than one stage of logic; hence we cannot simply insert it into a normal STAPL circuit. Instead, we provide for the extra delay by omitting the second stage (the completion stage) from the circuit and instead using the grant lines to reset the request lines directly. The resulting ARB is shown in Figure 6.7. In terms of PRS, the circuit consists of the usual arbiter-filter combination [54], the usual STAPL pulse generator, and

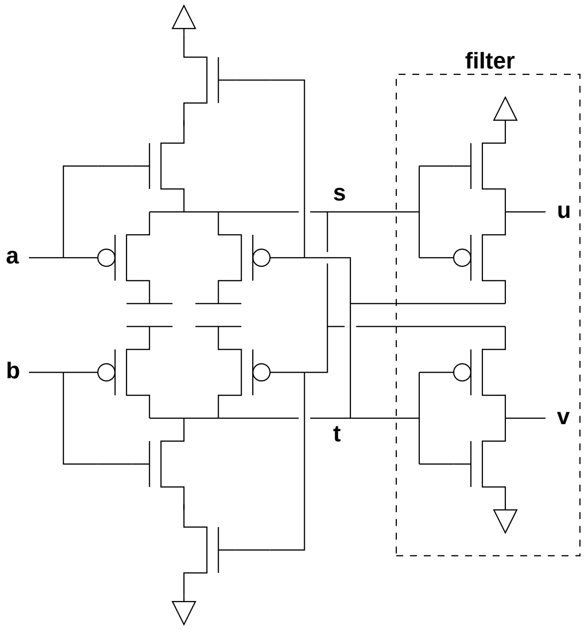

Figure 6.6: “Mead & Conway” CMOS arbiter.

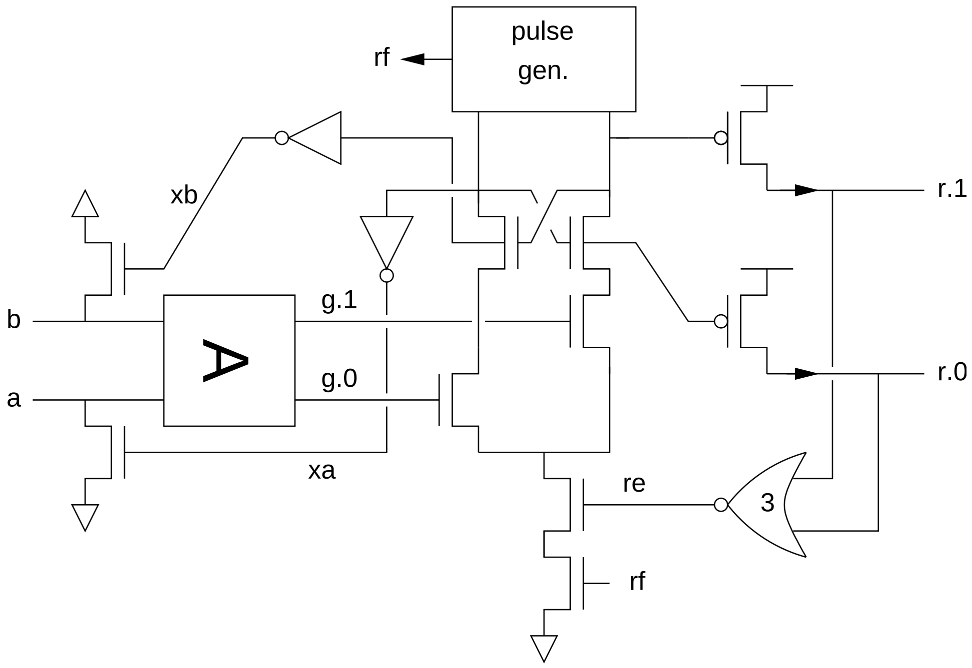

Figure 6.7: Complete STAPL ARB process.

$$re \wedge rf \wedge g.0 \wedge r\_.1 \;\rightarrow\; r\_.0\downarrow$$
$$re \wedge rf \wedge g.1 \wedge r\_.0 \;\rightarrow\; r\_.1\downarrow$$

$$r\_.0 \;\rightarrow\; xa\downarrow$$
$$\neg r\_.0 \;\rightarrow\; xa\uparrow$$
$$r\_.1 \;\rightarrow\; xb\downarrow$$
$$\neg r\_.1 \;\rightarrow\; xb\uparrow$$

$$xa \;\rightarrow\; a\downarrow$$
$$xb \;\rightarrow\; b\downarrow$$

$$\neg r\_.0 \;\rightarrow\; r.0\uparrow$$
$$\neg r\_.1 \;\rightarrow\; r.1\uparrow\,.$$

Since the circuit used here is slightly different from what we have used in STAPL circuits so far, some care may be necessary to ensure that the circuit verifiably works.

Attentive readers will have noticed that the interlock we introduced for the state variable has appeared again in ARB. We see why: the S-R latch used for arbitration is indeed a state-holding element (albeit one that in these enlightened times is considered somewhat archaic); it has timing characteristics similar to those of the cross-coupled inverters used in the state-variable compilation. Consider a scenario that both inputs to the arbiter are asserted in. The arbiter chooses one, setting in motion a train of events that ends in the chosen input's being deasserted. At this time, the output of the arbiter will quickly change from a 1-0 state to a 0-1 state, in exactly the same troublesome way that a state variable can change. This shows that if the interlock were not added, one output's rising a mere single transition later than designed could cause fatal confusion.

6.5.2 Four-phase converters

Why would we ever convert to four-phase logic? Have we not covered all the kinds of circuits necessary for building any system we might desire? The practical man will know that sometimes he will be called upon to interface his circuits with the outside world, from time to time even to interface with inferior things he has little or no control over. In these cases, he may find the synchronizer useful; and it is otherwise also one of the most practical circuits that we have not given a STAPL implementation for. Our introduction of means for converting between the STAPL family and four-phase QDI circuits ensures that the existing QDI synchronizer implementations can be used; it also obviously allows our carrying over other convenient QDI circuits. For instance, the extra design freedoms of four-phase logic appear to allow designing circuits that are very efficient in terms of energy consumption or number of devices, by using creative and non-obvious reshufflings that efficiently encode complex control [54].

There are enough similarities between the STAPL handshake and the QDI handshake that converting between the two is not too difficult. The only thing that needs to be done is to make the STAPL circuit respect the falling edge of the QDI handshake; i.e., it now has to wait for $[\neg li]$. Because of the timing assumptions in the interface of STAPL circuits (σ and ξ), the QDI circuit that is to be interfaced with has to obey certain timing constraints. The easiest way of solving the problem is probably to make a standard cell with a QDI interface on one side and a STAPL interface on the other. Using the cells that we have already built, we can put together a QDI "weak-condition half buffer" [43] and a STAPL half-buffer. We force the STAPL buffer to respect the QDI handshake by not re-arming the pulse generator until $[li]$ has occurred; for this purpose, we generate $R6$ by completing the input channel rather than in the usual way. Figure 6.8 shows the resulting circuit; notice how little extra circuitry is required.

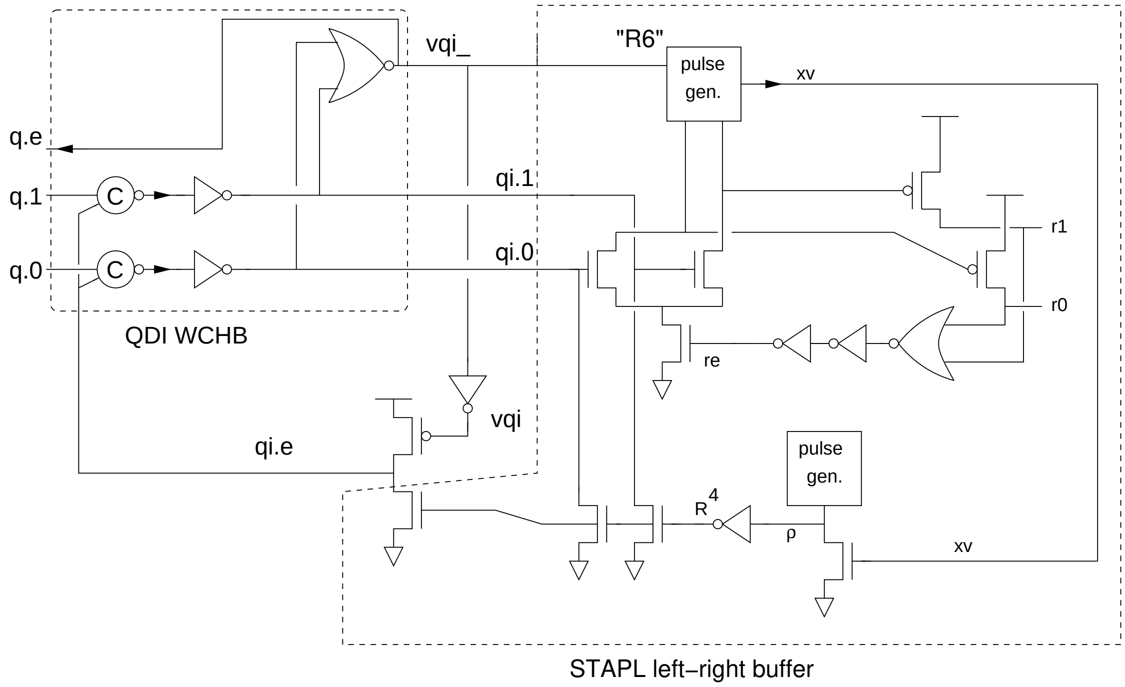

Figure 6.8: QDI-to-STAPL interfacing cell built from a QDI and a STAPL buffer.

Converting from STAPL to QDI is about as easy. A circuit for doing it is shown in Figure 6.9; the main changes from standard QDI and STAPL buffers are that the QDI buffer needs to reset its inputs with a pulse, easily generated from its acknowledge; and the STAPL buffer waits for the QDI acknowledge as well.

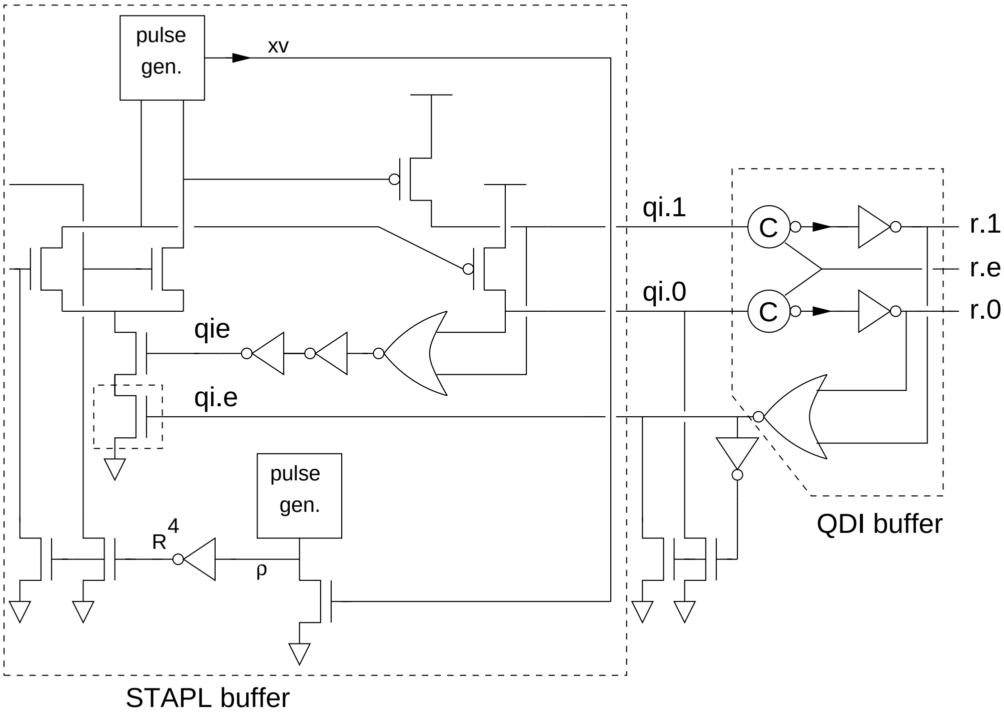

Figure 6.9: STAPL-to-QDI interfacing cell built from a STAPL and a QDI buffer.

Simpler circuits could probably be designed for converting between the STAPL and QDI protocols. The ones we have seen however have the advantage that, since the conversion is implemented with a pair of buffers and a few extra transistors, they easily generalize much the same as other STAPL and QDI circuits do.

6.6 Resetting STAPL circuits

It should be understood that out of the 2^N states that a digital STAPL circuit of N nodes can be in, only a small minority are meaningful; for instance, any circuit that has more than one node out of a one-hot channel **true** is in an illegal nonsense-state. Until now, we have been concerned with describing the repetitive operation of STAPL circuits; we have described mechanisms whose incorporation into circuits will allow these circuits' repetitively computing useful things, all the while remaining in the legal parts of their state spaces; but how do we initially coax them thither?

Since an asynchronous circuit has no clock that governs its timing, it is in general difficult to predict when it shall respond to a presented input. What is more frightening is that an asynchronous circuit is by its nature self-invalidating; i.e., there is no legal stable state for an asynchronous circuit to be in, because if there were, then the circuit would never get out of that state. Hence it is clear that what we seek to do in resetting an asynchronous system is to put it in an unstable state, whence it may proceed to execute normally. On the other hand, there is no requirement that the reset state itself be a legal state that could be encountered later during execution, as long as we can show that from the reset state we shall eventually encounter only legal states.

6.6.1 Previously used resetting schemes

The most commonly used resetting scheme in QDI circuits consists of introducing a single active-low reset node called *Reset_* together with its active-high inverse *Reset*; this has been used in most QDI projects undertaken at Caltech. The scheme works by resetting the circuit nodes to a known state, call it $\mathcal{R}$, when the reset nodes are asserted. Interference between production rules is allowed during the beginning of the reset period, as long as it resolves itself within a limited time; no interference is allowed afterwards. The time allotted to resetting is "long"; i.e., the timing of the reset pulse is handled through a comfortably safe timing assumption. When the reset nodes are finally deasserted, the circuit begins executing.

From the preceding, we understand that the actual initial state of the circuit from the point of view of the CHP specification is not the reset state that is the fixed point achieved by asserting the reset nodes, but rather it is the special state $\mathcal{R}\big|_{Reset\to\mathbf{false},Reset\_\to\mathbf{true}}$, achieved just after our deasserting the reset nodes. Designing the MiniMIPS, we realized that—while the timing assumption governing the length of the reset pulse is unavoidable—it would be inadvisable to trust that the two transitions $Reset\_\uparrow$ and $Reset\downarrow$ occur simultaneously; we avoided trusting this by using *Reset* only in downgoing production rules, i.e., by making the transition $Reset\downarrow$ undetected by the circuit. To complete the reset protocol, we arranged things so that the $Reset\downarrow$ transition always occurs before the $Reset\_\uparrow$ transition. Hence, the MiniMIPS reset procedure consists of the following succession of states (where _ denotes an arbitrary state, which need not be legal):

$$\_ \Rightarrow \mathcal{R} \Rightarrow \mathcal{R}\big|_{Reset\to\mathbf{false}} \Rightarrow \mathcal{R}\big|_{Reset\to\mathbf{false},Reset\_\to\mathbf{true}} \tag{6.1}$$

Since $Reset\downarrow$ is unchecked, no circuit activity can take place between the second and third states in the sequence, and we may again use a comfortably safe timing assumption. In practice, whereas we handled the reset-length timing-assumption by generating a pulse of the appropriate length off-chip, the second timing assumption is most easily handled by a few on-chip inverter delays.

6.6.1.1 *Go* signal

There are obvious problems with the described resetting scheme. The first and most obvious is the timing assumption used for sequencing $Reset\downarrow$ and $Reset\_\uparrow$. Secondly, the transition $Reset\_\uparrow$ causes other transitions to occur immediately; hence, making this transition too slow could conceivably lead to problems of the kind described in Section 4.4. The first of these problems can be eliminated and the second alleviated by introducing a third reset signal, which we call Go. In the resulting reset scheme, we cause the transitions to occur in the following order: first, reset in the state where $Reset = \mathbf{true}, Reset\_ = \mathbf{false}, Go = \mathbf{false}$; secondly, $Reset\!\downarrow$ and $Reset\_\!\uparrow$ occur in any order; lastly, $Go\!\uparrow$ shall occur after a comfortably long delay. The reason that this scheme is better is that $Reset\_\!\uparrow$ no longer needs to do the double duty of on the one hand establishing $\mathcal{R}$ and on the other holding back the execution; in other words, most of the load that was on $Reset\_$ can be kept on it and only a small part shifted to Go, which is the signal that must switch reasonably quickly. The progression of states is now:

$$\_ \Rightarrow \mathcal{R} \Rightarrow \mathcal{R}\big|_{Reset \to \mathbf{false}} \Rightarrow \mathcal{R}\big|_{Reset \to \mathbf{false}, Reset\_ \to \mathbf{true}} \equiv \mathcal{R}' \Rightarrow \mathcal{R}'\big|_{Go \to \mathbf{true}}; \tag{6.2}$$

we have here labeled the two stable reset states, $\mathcal{R}$ and $\mathcal{R}'$, separately.

On the system-design level, a third problem occurs with the QDI reset scheme. Consider a "token ring," i.e., a chain of buffers connected in a ring. We should like to think of these buffers as being all of the same type, viz.,

$$BUF \;\equiv\; *[\ L?x;\ \ R!x\ ]\ .$$

Such a ring cannot be useful. It will do nothing, since all processes are attempting to receive but no one is sending—sadly reminding us of dining philosophers that starve.

We can simply solve the ring-resetting problem by introducing an asymmetry; we shall then have two types of buffer process, viz.,

$$\begin{aligned} BUF0 \;&\equiv\; *[\ L?x;\ \ R!x\ ]\ , \quad \text{and} \\ BUF1 \;&\equiv\; *[\ R!x;\ \ L?x\ ]\ . \end{aligned}$$

The number of buffers of type 1 used determines the number of "initial tokens" in the system. While correct and conventional, the solution leaves us unsatisfied. More precisely: when we are dealing with a system that is described in slack-elastic terms, we are allowed to add slack *after* the CHP description has been completed, during its compilation to PRS and circuits; we might want to put the initial tokens in that extra slack when possible, but the form of description we have chosen does not allow that; furthermore, once the decision has been made at the CHP level to call one buffer $BUF0$ and another $BUF1$, should we not expect that the top-down compilation procedure will yield different implementations? But the same specification (and hence implementation) could be used for both if the initial tokens could be shifted into the extra slack.

We thus get the impression that the placing of initial tokens should properly occur in a separate phase of the design; it would then be possible that our ring of buffers should have processes only of the type BUF. At this point, the possibility of using a single implementation of BUF may seem like an insignificant advantage, but we shall see later (or may imagine now) that it would allow our using a vastly simpler template for describing the CHP processes, which need not make reference to the initial state: we should realize that the simplification, small for BUF, will be much more significant for complicated processes, since a process with N channels may at reset have a token (or not) on each of them.

Let us not deal further in hypotheticals: there is a catch. Normally, QDI processes have combinational logic at their outputs. Hence, any scheme that should attempt resetting QDI processes in two phases as suggested must have knowledge about their internals (it must reset the inputs of the combinational logic, not the outputs).

Here the STAPL circuits have a definite advantage: they have a single transistor on their outputs, so it is possible to create the initial tokens directly on the channels between processes. The reset protocol that we use is the same as the modified QDI protocol: $Reset = \mathbf{true}, Reset\_ = \mathbf{false}$, $Go =$ **false**. Now we can choose to identify the two reset states $\mathcal{R}$ and $\mathcal{R}'$ thus: $\mathcal{R}$ will be the starving philosophers' state, i.e., when all processes are ready to receive and none is sending; $\mathcal{R}'$ will be the state when all tokens that shall be created on reset have been created. The happy conclusion is that we can design all processes so that they themselves attain $\mathcal{R}$ (when $Reset \wedge \neg Reset\_$ holds); we shall separately add circuitry for getting from $\mathcal{R}$ to $\mathcal{R}'$ (when $\neg Reset \wedge \neg Go$ holds).

6.6.2 An example

Let us now turn to a simple example. How should we reset the STAPL left-right buffer? From above, we know that we need only concern ourselves with the empty-pipeline case. The goal will be resetting the circuit with as few transistors as possible.

The most obvious places for inserting reset circuitry are the pulse generator and sequencing circuits that must always be present in the same way. Considering the HSE for the (dual-rail) left-right buffer,

$$*[(\dagger)[\ l0 \longrightarrow r0\uparrow\ [\!]\ l1 \longrightarrow r1\uparrow\ ];\ l0\downarrow,\ l1\downarrow,\ [\neg r0\ \wedge\ \neg r1]\ ]\ ,$$

we should like to reset into the state marked (†). Given a set of processes to reset, we choose the following division of labor: each process shall be responsible for resetting its own internal nodes and its own *inputs*. Hence, we shall have the various $R4$ nodes resetting in the **true** state (which incidentally violates the otherwise sound property of their being mutually exclusively **true**): this will clear the inputs. As for the internal nodes, we make the pulse generators reset into the state where the precharge signal is active.

6.6.3 Generating initial tokens

So far we know how to reset a system so that all processes begin by receiving. We earlier mentioned that we should like to create the initial tokens during the period when $\neg Reset \wedge \neg Go$ holds. Doing this is straightforward: tokens between processes are signified by **true** nodes; hence, all we shall need to do is pull the nodes that we wish to reset up with two p-transistors implementing $\neg Reset \wedge \neg Go$. Of course, we must check that *Go* switches fast enough that the isochronic fork is unimportant. If we feel that we cannot arrange this, then we might have to add yet another reset node, e.g., *ReallyGo*, since strictly speaking, using *Go* for generating initial tokens violates a property implied by Section 6.6.1.1, namely, the property that *Go* should be used only for holding tokens back from execution, not for resetting things.

6.7 How our circuits relate to the design philosophy

In Section 5.2.5, we outlined a few guidelines that our circuits should obey. We have since developed the bit generator, bit bucket, left-right buffer, multiple-input and multiple-output circuits, circuits that compute arbitrary functions, circuits that do not use certain inputs, as well as a number of specialized circuits.

For each one of the designs, it may have seemed that a new mechanism was invented out of thin air. Of course, if we take a different view of things—if we consider the simpler circuits as special cases—then these mechanisms may not seem so *ad hoc.*

Let us therefore examine Figure 6.2. How much of the design presented there is an inevitable consequence of our design philosophy, and how much of it simply the result of arbitrary design decisions?

First, while it is not exactly part of the design philosophy, the choice of two transitions' delay for the forward path is a natural one: this choice minimizes the delay, given that we want to maintain the same signal senses at the outputs as at the inputs. Using a single p-transistor per output rail is clearly the simplest way of implementing the output part of the handshake, as is using a single n-transistor per input rail for implementing the resetting of the inputs.

Secondly, we stated that we were going to implement the σ delays everywhere with pulse generators: this gives us the pullups for the internal nodes. But why can we sometimes get away with combinational pulse-generators and why do we sometimes need to use ones that are one-shot, i.e., explicitly re-armed? Similarly, why do we sometimes need the foot transistor and why do we sometimes not need it? The answer to these questions is the same: in general, the one-shot pulse generators and foot transistors are required. It is only in the very special case of a single output that they are overkill, because with only a single output we can make an additional timing assumption; namely, we can assume that once we have produced the output, all the necessary inputs have arrived; this is true even if some inputs are being ignored since we are counting dummy outputs used for completion as separate outputs. We should note that the timing assumptions are slightly more difficult to meet when we omit the foot transistor: if the foot transistor is used, we know that the pulse generator cannot re-arm until the second stage has reset it; if it is not used, then a slow falling transition on the output can cause the pulse generator to fire twice.

Thirdly, why do we use a second stage for computing what to acknowledge, and why does the second stage have the form we have seen? Here we have mainly tried to keep things simple: the same implementation is used for the second stage as for the first. Note that the reason that we can omit the foot transistor in the second stage is that we can consider the second stage as having a single output; namely, the internal channel $R4$, which always produces a result. As we have seen in Section 6.3.2, there are other ways of implementing the functionality provided by this second stage.

Lastly, what of the special circuits: the arbiter, state bit, and four-phase converters? These circuits are harder to understand directly within the stated design philosophy. The reason this is so is not hard to see: the state bit is essentially a transistor-saving trick that eliminates much of a feedback loop that could as well have been implemented with buffers, and the arbiter has the same timing characteristics. In the case of the four-phase converters, we are dealing with circuits that in any case do not obey our design philosophy, so it is not surprising that these circuits should look a little odd.

6.8 Noise

When we speak of "noise," the implicit assumption is often that noise is due to some external mechanism, or at least to some mechanism that we are not modeling properly, e.g., shot noise, thermal noise, noise from outside electronic systems. But in more performance-oriented design-styles, the digital model is often just an approximation; the difference between the pure digital model and the physical behavior of the system we can also loosely call "noise." Hence, we shall use the term "noise" to denote any deviation from ideal conditions.

6.8.1 External noise-sources

External noise-sources are the easiest to deal with. In Section 3.3.5, we defined what noise margins mean in pulsed circuits and left it up to the reader to come up with a metric suitable for making sense of the multi-dimensional noise that we should find in such circuits. The noise has as many dimensions as the test pulses in $\mathcal{P}$ have parameters, to use the terminology of Section 3.3; but apart from this mathematical annoyance, noise margins in STAPL circuits are really of the same form as in synchronous circuits, and may be treated similarly.

Are the noise margins wide enough in the circuits that we have studied? This is a quantitative question, and there are several ways of answering it. First, we can flip back to Section 3.2.2.3, where we should see that the input pulse lengths can vary between 1.0 and 12 normal transition delays, and the circuit will still work as intended; this will probably satisfy most readers. Secondly, we note that we can build STAPL circuits with any desired noise margins by manipulating the delay in the feedback loops and the thresholds of the input logic; as long as the noise is smaller than the signals we are looking for, we can build a STAPL circuit that works. Lastly, we can rephrase the question thus: do STAPL circuits give higher performance for the same degree of noise immunity than implementation technology X? This question is harder to answer; it does seem that the STAPL circuits can be made almost as noise-immune as QDI circuits at much higher performance levels, and compared with the highest-performance synchronous logic-styles, STAPL circuits achieve the same or better performance. But the question will probably not be answered to everyone's satisfaction until STAPL chips have been fabricated and are found to work reliably.

6.8.2 Charge sharing

The STAPL circuit family makes great use of dynamic (or at least pseudo-static) logic. Charge sharing (between internal parasitic capacitances in a domino block and the output node) is the bane of dynamic logic styles. The situation could be particularly bad in STAPL circuits because we cascade dynamic-logic stages. (In the MiniMIPS, for instance, an effort was made to avoid cascading dynamic logic by attempting to alternate dynamic and static logic as much as possible.)

The good news is, first, that the STAPL circuit family never makes use of p-transistors in series driving dynamic nodes (the only p-transistors in series are in the circuitry that generates the *re* signals), and secondly, that the timing behavior of STAPL circuits is much simpler than it is for QDI circuits. For these reasons, we should not generalize all the bad experiences from high-speed QDI design and think that things are only going to be worse in STAPL. The simpler timing behavior, especially, allows using much simpler circuitry for avoiding problems with charge sharing. Since the domino block never has its outputs "floating low" except potentially for a very short period of time (because they are pulsed), we need not worry about sneak paths from *Vdd* to the output rails, as long as the transistors along these paths are weak. Concretely speaking, we can systematically use the circuit in the dashed box of Figure 6.10 for reducing charge-sharing problems. (This circuit will not work in QDI circuits, because sneak paths from the resistor to the output could pull up the outputs out of sequence when they are left floating.) The costs of using such charge-sharing avoiders are that the circuits are a little slower and that static power dissipation is possible if some of the inputs arrive but not the others (so one has to be a bit careful when introducing these circuits if low power should be an important design objective).

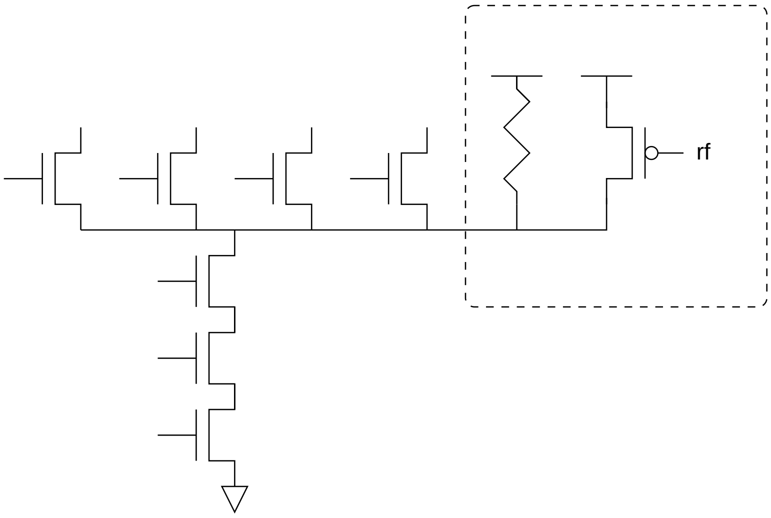

Figure 6.10: Circuit alleviating charge-sharing problems. Resistor implemented with weak transistor.

6.8.3 Crosstalk

Aside from charge sharing (more properly "static" charge sharing), something that causes reliability problems in modern VLSI circuits is crosstalk (also variously called coupling noise or "dynamic" charge sharing). Some authors (e.g., Balamurugan and Shanbhag [6]) have even suggested that crosstalk noise gets worse when device sizes are scaled down. While arguments suggesting that crosstalk noise gets worse because of *Vdd* scaling should be eyed with suspicion, it is on the other hand true that the aspect ratio of minimum-size wiring on modern chips has made crosstalk worse: using wires that are tall and narrow means that most of a wire's capacitance is to its horizontal neighbors, not to the substrate.

The STAPL circuit family offers no special remedies for crosstalk noise; the dynamic nature of STAPL circuits indeed suggests that they are susceptible to it. At the same time, we should not exaggerate the problem: 1-of-4 encodings for instance allow signals to be routed so that a wire is never routed adjacent to two "aggressors" (i.e., circuit nodes that couple strongly to the wire in question). Furthermore, as we have stated elsewhere, a well-designed asynchronous circuit will have most of its capacitance in the transistor gates (see footnote on p. 147). Finally, we can use our circuits' being asynchronous by inserting extra buffering: this is easier than in synchronous systems, since our timing constraints are less rigid.

In practice, the avoiding of destructive crosstalk noise will have to be done with design tools: we shall have to map the noise margins that we have defined for STAPL circuits to specific circuit-design guidelines.

6.8.4 Design inaccuracies

The most serious issue with STAPL circuits—the most serious way that the real world deviates from ideal conditions—is probably design errors or design uncertainties. The reader has probably guessed, for instance, that mis-sizing transistors in a STAPL circuit can cause the circuit to fail. Many designers will be reluctant to give up their "ratioless" CMOS logic. It would also be unsatisfactory if every *instance* of every circuit in a large STAPL system had to be sized specially just to keep things working.

We can phrase it thus: assume that we have a STAPL circuit designed to work well in a particular environment, and now it turns out that, e.g., the capacitance on its outputs is much higher than anticipated—this can happen because our extractor is inaccurate or because we are lazily trying to reuse a piece of layout that was designed for something else—what happens?

If we overload one of the STAPL circuits presented in this chapter, then its internal pulse may not be quite enough for setting the output signal to Vdd; equivalently, we may consider the situation when the output p-transistor is sized very large and the internal pulse is too feeble to set the output to Vdd. We can think of this as moving the normal operating point of the circuit to one where the output pulse has a smaller height; in terms of the pipe diagrams, the operating point is moving more or less along the arrow marked "1" in Figure 6.11. We can see that we shall have to overload the circuit considerably before it fails (until the pulse height is about one half of normal—this is more than double capacitance because the pulses have flat tops/bottoms; they are not normally triangle waves).

The ideal situation would be if the circuit could move along the arrow marked "2"; if it did that, then we should have the largest possible noise margin. The STAPL circuits naturally move their operating points somewhat to the right in the figure when they are overloaded by a too-large output transistor because the pulsed node drives that transistor directly; the transistor's being larger than expected causes the internal pulse to be wider than designed because it delays the rise and also the fall of the pulsed node. It is possible to add further feedback from the output node (i.e., we could add a transistor that senses that the output is being pulled up too slowly and then adds extra drive). By using these kinds of mechanisms, we could aim the load line down the pipe and thus achieve very good noise margins.

Single-track circuit-families have been studied in the past [8, 83]. These have tended to use much stronger output-feedback than our STAPL family—recall that the STAPL family's circuits wait until the inputs have arrived, but once they have been detected, the timing of the pulse is almost entirely locally determined. As we have seen, the feedback could be a good thing, if it aims the load line in the proper direction.

Why have we not studied these other kinds of circuits in more detail? The reason is that the output feedback, via the internal-pulse length and height, affects the pulse widths that can be

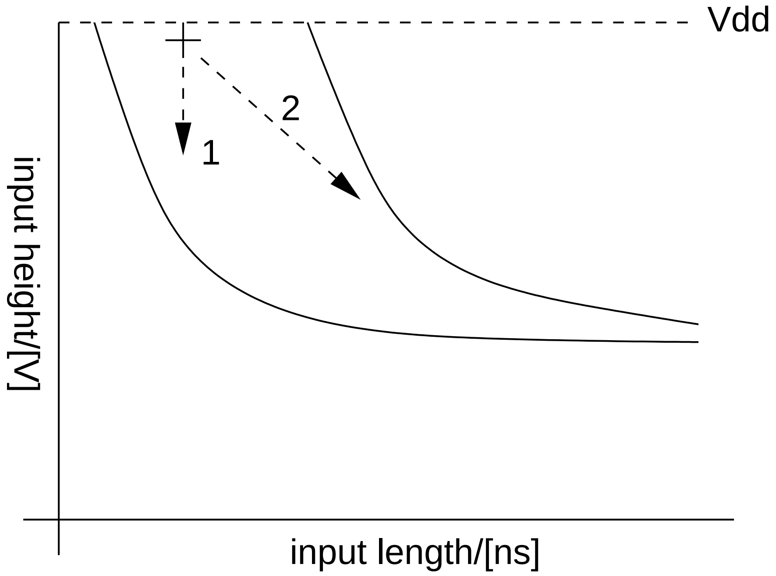

Figure 6.11: "Load lines" of pulsed circuit. 1: pulse becomes lower when the circuit is overloaded; 2: pulse becomes lower and longer.

tolerated on the inputs and it also affects—in the more complex STAPL circuits—the behavior of the second stage of the circuit (the acknowledgment stage). In other words, the theory required for explaining the circuit behavior becomes more difficult because changes at the outputs now cause changed input behavior (i.e., we have to solve equations rather than just propagate functions; the simple conditions on σ and ξ will no longer be so simple). This will undoubtedly be an interesting area for further research.

Chapter 7

Automatic Generation of Asynchronous–Pulse-Logic Circuits

You cannot fight against the future. Time is on our side.

— *Gladstone (1866)*

We have so far developed a theory accounting for the proper functioning of asynchronous-pulse circuits, and we have developed some example circuits. While he who is skilled in slack-elastic QDI design will see how, following the guidelines that we have laid down, he could realize any desired computation in STAPL circuits, those not so skilled may not see as clearly how this should be done. Furthermore, we should like to automate the design procedure from CHP-level specification to STAPL circuit, so that we may avoid both needless labor and the making of careless mistakes. For these reasons, we shall now take a step back from the STAPL circuits and explain how we can succinctly describe processes of the kinds that we discussed in Chapters 5 and 6.

We will design circuits at a high level of abstraction, i.e., in terms of digital abstractions. For this, the abstract CHP-level of description is ideal. Eventually, of course, we will compile from the CHP to transistor networks, and thence finally to concrete geometric layout. In most situations, we should like human designers to have to do as little work as possible: they are best at the abstract work needed for generating the original CHP; they are not so good at intricate but mindless tasks like logic minimization; and they are lazy. Hence, we should aim at designing processes or circuits as much as possible at the CHP level, and we should permit describing the processes in as abstract terms as necessary.

7.1 Straightforwardly compiling from a higher-level specification

In software systems, we usually compile a program as follows. First, we convert the high-level program into an intermediate-language representation; this is mainly a syntactic transformation for streamlining the syntax of the program to simplify automatic translation tools' analysis of the statements of the program. Secondly, we convert the intermediate-level representation into a dataflow graph, which is an abstract representation of how each value computed by the program depends on previous operations and of how later operations depend on the value. Thirdly, we manipulate the dataflow graph, aiming at lowering the cost of evaluating the statements it implies, but maintaining its meaning. Lastly, we convert the optimized dataflow graph into a machine language program, which can be loaded and executed by a processor when desired.

The technique that has been evolved for compiling software programs into machine language is attractive because it cleanly separates the question of *what* is computed from *how* it is computed. Specifically, given a simple program that performs actions that are independent, the dataflow graph can be used to deduce this property. Having determined that the actions are independent, the compiler can convert them separately into the target language. The dataflow graph also represents the constraints on the reordering of actions in the program.

The dataflow technique can be applied to the compiling of HSE into PRS, but because the necessary properties (stability and noninterference [54]) are global system properties, this is not simple. The only known algorithms that work on general HSE programs conduct exhaustive state-space exploration. As far as is known, these algorithms all take exponential time in the worst case, and they do not in practice work on large systems.

The difficulties of directly compiling from a higher-level description to PRS suggest that this is the wrong way of going about things. A description of an algorithm at the level of the sequence of actions on each bit (or electrical node) of a system is simply at too fine a level for most purposes. Once an algorithm has been described in this much detail, it has been over-sequenced; and removing the extra sequencing is too difficult. The bad level of specification that we speak of is exactly the HSE level.

That the compilation from HSE to PRS is hard is not the only problem with this approach. Another is that we have no trustworthy metrics for determining when one compilation is better than another. While we could possibly develop such metrics for determining when a given compilation result will run faster than another in a known environment, we may not know *a priori* all the parameters of the environment where a circuit will operate; if we had to know these parameters before compiling the circuit, we should certainly not be able to claim that the compilation procedure is modular. And modularity is the principle, above all others, that we strive for in asynchronous design. Better then to abandon the HSE level in our day-to-day design work and use PRS templates for compiling directly from CHP to PRS; the resulting PRS could be trusted to work efficiently in most environments.[1]

7.2 An alternative compilation method

Because HSE is difficult to compile, we shall in this thesis—as we did in the MiniMIPS project—take the position that we should like to compile *ab initio* as seldom as possible. (In the MiniMIPS project, we used only a few templates to compile almost all the processes into QDI circuits. Lines describes most of these templates in detail [43].)

Compiling arbitrary CHP programs directly (i.e., syntactically) into circuits is possible [13, 9]; doing it efficiently is difficult. This is why we have chosen to compile only a restricted subset of CHP programs into circuits; the particular subset we have chosen is approximately those processes that are described by the capabilities mentioned in Section 5.2.4, namely that the circuits should be capable of the following:

- Computing an arbitrary logical function
- Computing results conditionally
- Receiving operands conditionally
- Storing state
- Making non-deterministic decisions
- Communicating with four-phase QDI circuits

Since we explained how to implement each of these capabilities in terms of STAPL circuits in Chapter 5, the methods we saw will form the basis for the discussion. In short, we shall bridge the gap between those circuits and CHP programs.

7.3 What we compile

We should realize that the last two capabilities in the list of Section 5.2.4 are used infrequently, and as we pointed out in Section 6.5, these circuits are not easily generalizable; hence we shall drop these

[1]We should make it clear that we are not condemning the HSE language itself. The HSE notation is, as we have seen, extremely useful for designing the templates used for compiling from CHP to PRS; the HSE language is indeed the most convenient of the languages we use for describing handshaking behaviors (as it should be). What we are suggesting is however that we should probably not manipulate the HSE descriptions of processes too frequently; we should do it only when we are developing the compilation templates or when we have to design some special circuit that we do not know how to design well using the day-to-day templates.

capabilities from the list of what an automatically-compilable process needs to be able to do, and we are left with the following four capabilities:

- Computing an arbitrary logical function
- Computing results conditionally
- Receiving operands conditionally
- Storing state

We should realize that there is nothing in these capabilities that is specifically tied to STAPL implementations, or even anything that is necessarily tied to hardware implementations: we could from these abstractions equally well build up a software-programming methodology. What is however clear is that these capabilities are fundamentally "asynchronous"; it is possible to introduce a clock to sequence the actions further—if we should for some reason be frightened of the prospect of asynchronous-circuit design—but as the send and receive actions already in themselves supply the necessary synchronization, this would seem otiose.[2]

The STAPL circuits that we have developed have the capabilities we desire, but they have no further means of control. Hence, the only programs that they can implement have the structure of the templates described in the previous chapter, viz.,

$$*[\ \langle ,i :: L_i?x_i \rangle\ ;\ \langle ,j :: R_j!f_j(\mathbf{x}) \rangle\ ]\ ,$$
$$*[\ \langle ,i :: L_i?x_i \rangle\ ;\ \langle ,j :: [G_j(\mathbf{x}) \longrightarrow R_j!f_j(\mathbf{x}) [\!]\, \neg G_j(\mathbf{x}) \longrightarrow \mathbf{skip}] \rangle\ ]\ ,$$
$$*[\ \langle ,i :: L_i?y_i \rangle\ ,\ \langle ,k :: x_k := x_k' \rangle\ ;\ \langle ,j :: R_j!f_j(\mathbf{y},\mathbf{x}) \rangle\ ,\ \langle ,k :: x_k' := g_k(\mathbf{y},\mathbf{x}) \rangle\ ]\ ,$$

as well as the conditional-inputs template (see Section 6.3.3.3) that we have not made explicit, and combinations of any of these templates.

The conditional-inputs template is not easy to describe in terms of CHP; let us merely say here that any of the inputs can be conditional. A more accurate definition of what we can and cannot do is given in Appendix A, p. 182.

7.4 The PL1 language

A CHP program fitting the templates described in Section 7.3 is easy for us to compile because it uses only a small, carefully chosen part of the CHP language. For the purposes of making clear the compilation procedure and simplifying the compiler as well as carefully delineating the kinds of conditional programs we can compile, we shall describe the programs in terms of a language that

[2]For various reasons, synchronous "asynchronous" systems have been investigated by Philips [67]. The dataflow models used in the early 80's and in current work in reconfigurable computing are also related [19].

compactly captures exactly those behaviors that we know how to compile; this language we call *Pipeline Language, version 1:* abbreviate as *PL1.*

The precise scope and syntax of the PL1 language are given in Appendix A; here we shall mainly be concerned with justifying the design decisions of the language and showing how one may compile PL1 programs into STAPL circuits.

7.4.1 Channels or shared variables?

Although CHP processes communicate with each other on channels, once the processes are implemented as circuits, the channels are implemented as shared variables. The shared variables' being products of such a compilation implies certain properties about them: for instance, a (slack-zero) channel is never "written" (i.e., sent on) twice without an intervening read. These properties may be useful for further compilation or for verification, but a naïve outside observer would not be able to tell that the shared variables resulted from a channel compilation. A single semantic construct hence can be described as either a channel, at the CHP level; or as a shared variable, at the circuit level.

It is almost certain that some useful operations are difficult to do with only the CHP constructs; it is even more certain that shared-variable hardware design is far more difficult, far less modular, and far more time-consuming than CHP hardware design. The PL1 language aims at combining the channel and shared-variable constructs in a way that, for the present circuit-design purposes, improves upon both the CHP and shared-variable (HSE or PRS) descriptions. The innovation is straightforward: in the PL1 language, we read and write channels as if they were shared variables, but the implementation—not the programmer—ensures that all channel actions are properly sequenced. The language forbids interfering constructs.

The PL1 language also only allows "safe" constructs. As we shall see, writing many simple CHP processes in terms that are close to our desired implementations involves the frequent use of constructs like the value probe or the peek operation. While the responsible use of these constructs is unobjectionable, the untrained eye cannot easily determine if the use has been responsible or not. Irresponsible uses quickly lead to nondeterministic programs, non–slack-elastic programs, and other abominations.

7.4.2 Simple description of the PL1 language

The PL1 language is a simple language for describing the small processes that we should like to build hardware systems out of. The semantics of the PL1 language allow the implementation to add more slack than exists in the specification; hence the language is appropriate for the design of slack-elastic systems.

In most message-passing programming languages (CHP in particular), using a data value that arrives on a channel first requires receiving it. In the hardware implementation, however, we can use and receive the value at the same time, or even delay the acknowledging of the value so that it remains pending. This functionality we have added to CHP with the value probe and peek operations. In the PL1 language the value probe and peek are the most basic operations: receiving a value is done by first using it (the peek), and then acknowledging it as a separate action.

PL1 programs consist of sets of guarded commands. The guards are not necessarily mutually exclusive. The semantics are that the process waits until it can determine, for each guard, whether or not it will be true for the next set of values that shall arrive. For instance, determining whether the guard `a==1` is true requires knowing the value of `a`. It is not enough that no value of `a` be present, since this would not be slack-elastic: the value `1` could have been delayed en route; hence if there is no value of `a` yet present and `a==1` is evaluated, the process will suspend. Of course, a value of `0` does establish that `a` will not next be `1`. Thus we can evaluate expressions while banishing from our language the "undefined" value of a channel: there is in PL1 no way of writing the true negated probe.

Let us examine at a simple example PL1 program:

```
define filter (e1of2 c, l, r)
{
  communicate {
    c==1 -> r!l;
    true -> l?,c?;
  }
}
```

The expression syntax is the same as in the C language [41]. The first line is the prototype for the process. The declaration of the parameters as `e1of2` means that these are channels that can hold the values `0` and `1`. Hence, evaluating `c==1` requires receiving a value on `c`.

If `c==1` evaluates to **false** (i.e., if `c` should get the value `0`), then only the second guarded command is executed, and the values on `l` and `c` are received and acknowledged; the process suspends until values are present on both the channels.

If `c==1` evaluates to **true** (i.e., if `c` should get the value `1`), then *both* the guarded commands will execute; the value received on `l` will be sent on `r` as well.

The PL1 language is defined so that programs like this one are meaningful, even though `l` and `c` are each used in two places at the same time. In essence, all the uses that require the value are performed first, then it is acknowledged. Only strictly contradictory programs are disallowed (see below). Appendix A has more details.

7.4.3 An example: the replicator

It is often useful to be able to replicate data sequentially; let us therefore consider the process

$$REP1 \equiv \ *[\ L?x;\ \ c := \mathbf{true};\ \ *[\ c \ \longrightarrow\ R!x\ ;\ C?c\ ]\ ]\ .$$

If we are to implement REP with the methods of Chapter 5, we shall have to remove the nested loop from this program and rewrite it using value probes. The result of this is[3]

$$\begin{aligned}
&REP2 \equiv \\
&*[[\ \overline{C = \mathbf{true}} \ \longrightarrow\ R!(L¿),\ \ C? \\
&\ \ [\!]\ \overline{C = \mathbf{false}} \ \longrightarrow\ R!(L?),\ \ C? \\
&\ ]]\ .
\end{aligned}$$

The $REP2$ program is not, strictly speaking (given the usual semantics of CHP), equivalent to $REP1$; but it is equivalent under the assumptions of slack-elasticity. The transformation from $REP1$ to $REP2$ is anything but obvious; it is difficult to explain what it is that makes $REP2$ a reasonable program for an implementor to compile into a circuit and what it is that makes $REP1$ unreasonable.

In the PL1 language, we must declare the variables; this is no drawback, since declarations would anyhow be necessary for specifying the variable types at some point before compilation into the target PRS/circuit; we thus arrive at, e.g.,[4]

```
define rep3(e1of2 c,l,r)
{
  communicate {
    true -> c?,r!l;
    c==0 -> l?;
  }
} .
```

There are two executable statements in this PL1 program:

```
true -> c?,r!l;
c==0 -> l?;
```

[3]The Spanish inverted question mark, ¿, is the notation used for the recently introduced *channel peek* operation in CHP [33, 66].

[4]We should note that the semicolons at the end of each line are syntactic, separating the two possibly concurrent statements `true -> c?,r!l` and `c == 0 -> l?`: in this regard, these semicolons have the same rôle as the commas in the interface declaration `e1of2 c,l,r`; on the other hand, the comma in `c?,r!l` is semantically meaningful, signifying parallel execution. There should be no confusion since there is no way of specifying sequential execution in the PL1 language beyond the global sequencing that is implied by the process structure itself.

We call the construct `c==0 -> l?` a *guarded command* (the guarded-command idea is due to Dijkstra [21]), where `c==0` is called the *guard*, and `l?` the *command* or *action*; `l` we occasionally shall refer to as an *action variable*.

It is worthwhile stating here—the appendix explains in more detail—that the semantics of PL1 are such that a process's concurrently executing `r!l` and `l?` presents no trouble: the actions are sequenced by the implementation so that they shall be non-interfering. Likewise, the implementation will see to it that the action `c?` is delayed enough that the condition `c==0` may be safely evaluated.

Why should we use the PL1 language, and in what sense is it preferable to CHP? The answer to these questions follows from the PL1 language's being capable of expressing only a small fraction of what the CHP language can express; however, it is a fraction that we know how to compile into efficient APL and QDI circuits. To some extent, we use the PL1 language so that the compiler may avoid the difficult problem of determining that $REP2$ is a reasonable implementation of $REP1$ or that `rep3` is a reasonable implementation of either; we should also like to avoid stating exactly what subset of CHP shall be a legal input to our circuit compilation method.

The PL1 language is not intended to replace the CHP language; on the contrary, the author expects that software tools will be written that shall be able to automatically convert CHP into PL1, or better yet, into PL1-equivalent data structures.[5]

7.5 Compiling PL1

Compiling PL1 programs, regardless of the target language, is a task best carried out in phases. The first phase, parsing and analysis of the input program, is common to each target; the second, generating the output, varies with the target. Figure 7.1 shows the resulting structure of the compiler: here we have shown the three back-end modules for generating different kinds of target representations, viz., one module that generates Modula-3 [65] code that shall compile and link as part of a Modula-3 simulation system, another for generating QDI circuit descriptions, and a third for generating STAPL circuit descriptions; the last QDI and STAPL generators generate CAST that describes a single process and that shall have to be combined with some infrastructure ("header files") to be useful.[6]

The intermediate form of the compiler is a Modula-3 data structure, not a file. This data structure

[5] Unfortunately, we do not have such tools today, and he hopes that by making explicit the kinds of information that can easily be compiled from, minds will be stimulated into solving the CHP–PL1 compilation problem.

[6] At the time of writing, there are two PL1 compilers. The first, written in C, generates C code (not Modula-3) that will work within a fast but rudimentary simulation environment: in this environment, function calls are used for context switches; hence, execution speeds are approximately 3× higher than for the same code using even very light-weight threads. This compiler can also generate rudimentary QDI production rules, which are correct but inefficient.

The second, written in Modula-3, generates efficient & optimized STAPL circuits; it can also (this work, still in progress, is due to Abe Ankumah [4]) generate QDI circuits; its abilities of generating high-level language code (Modula-3) for simulation purposes are yet unfinished work.

We shall mainly be concerned with the structure of the second compiler and especially with its STAPL back-end.

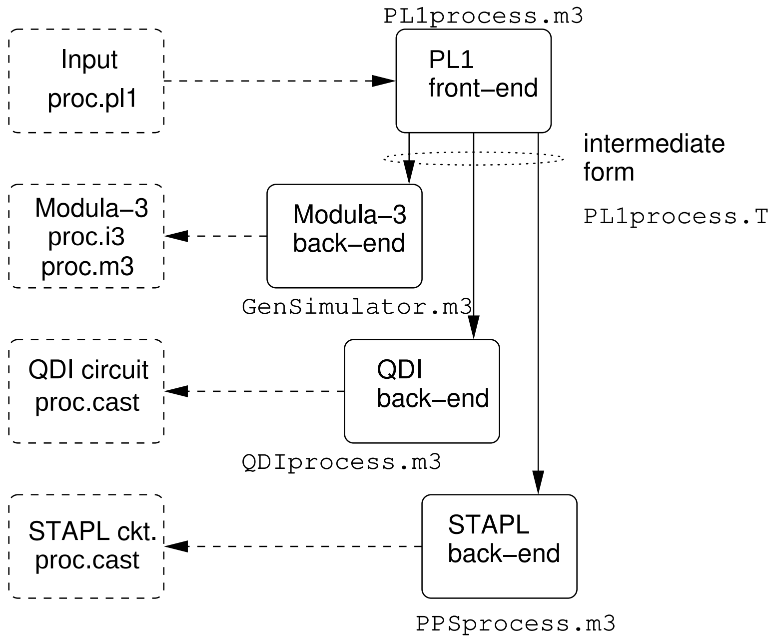

Figure 7.1: Structure of the PL1 compiler. Files are shown in dashed boxes; program modules in solid.

is an object, called a `PL1process.T`. The back-end modules are implemented similarly: each back-end module specifies an object type that inherits many of its properties from a generic "process object-type," `ProcessSpec.T`, which incidentally is also the supertype of the `PL1process.T`.

7.6 PL1-compiler front-end

The PL1 compiler represents the program being compiled as a Modula-3 object; all guards and actions of the program are represented in terms of binary decision diagrams (BDDs) [12].[7] Three levels of representation are used for data: first, the compiler front-end evaluates all PL1-language expressions as two's-complement numbers (a BDD expression is used for each bit of the numbers, the unspecified bits being treated as **false**); secondly, when the front end passes the results of its analysis to the back end, it uses a unary encoding, which makes the results suitable for the back end's compiling into circuits that use delay-insensitive codes; lastly, the back end generates the PRS for the circuit using a sum-of-products representation.

7.6.1 Determinism conditions

After parsing the input program and generating the appropriate BDD representations for the expressions used in it, the first task of the compiler front-end is the checking of some remaining determinism[8] conditions: while the slack-elastic design-style handles determinism issues well, the syntax of PL1 allows the programmer to specify meaningless, nondeterministic programs, e.g.,

```
define mistake(e1of2 r)
{
  communicate {
    true -> r!0;
    true -> r!1;
  }
}
```

We cannot tell if the programmer intended that `mistake` should send a `0` or a `1` repeatedly on `r`; or perhaps he intended some interleaving? The two benefits of ruling out mistakes like this one are: first, that programming mistakes are caught and programs will have well-defined meanings; and secondly, that the back-end of the compiler can now use the expressions of the PL1 program directly in generating its target PRS: the logical expressions of the PL1 program can be converted into production rules without further ado.

While at first glance it may seem easy to banish programs like our `mistake`, a moment's reflection will show that this is not so. The first-glance solution is to require that actions on a particular interface variable or state variable appear syntactically in a single place in the program for each

[7] The BDD package used in the compiler was written in C by Rajit Manohar.

[8] In light of the fact that the only non-deterministic programs that can be specified in PL1 have the kinds of nonsensical behaviors discussed here, these might as well be called "meaningfulness conditions."

such variable. This is unsatisfactory because it is often desirable to use a particular variable in two completely different ways, the choice being based on some arbitrary piece of information: consider a process that could either add or multiply its two data inputs based on a control signal; writing `c==0 -> r!(a+b); c==1 -> r!(a*b);` is easy, but if we had to combine the `+` and `*` operators in a single boolean expression?—at best, a confusing program; much more likely, an incorrect one.

At second glance, we could simply forbid that the actions on interface variables or state variables "appear dynamically" in several places on the same execution cycle of a program. With this view,

```
define buffer(e1of2 c,r)
{
  communicate {
    true -> c?;
    c==1 -> r!1;
    c==0 -> r!0;
  }
}
```

would be right, but `mistake` would be wrong. But what should we make of

```
define dubious_buffer(e1of2 c,r)
{
  communicate {
    true -> c?;
    c==1 -> r!1;
    true -> r!c;
  }
} ?
```

He who would take the position that `dubious_buffer` is another mistake would not injure reason, but the PL1 language described in the appendix allows `dubious_buffer` as having the same meaning as `buffer`. The grounds for allowing it are a very simple execution model for PL1 programs: on a given execution of a PL1 program, all enabled actions are collected and then executed concurrently, at most one action to an interface variable or state variable; on the next execution of the program, no memory of these events is retained except as specified in explicit state variables. We hence must forbid `x!0,x!1`; but of the two interpretations of `x!0,x!0`, viz. forbidden and idempotent, we choose the latter (i.e. `x!0`).

To check the determinism conditions, the compiler must make certain that whenever two commands are enabled at the same time, any overlapping state variables or send channels are treated the same way. In practice, we can also consider any invariant I that the programmer should care to mention; to check the conditions for the PL1 program G_0 `->` $\mathtt{S}_0(C_{S0})$ `[]` G_1 `->` $\mathtt{S}_1(C_{S1})\ldots$, the compiler hence has to run the following:

forall $P = G_\mathrm{i} \wedge G_\mathrm{j} \wedge I$
    if $P \not\equiv$ **false then**
        forall $a \in \mathbf{vars}(C_{S\mathrm{i}}) \cap \mathbf{vars}(C_{S\mathrm{j}})$
            assert $\mathbf{val}(b_\mathrm{a}, C_{S\mathrm{i}} \mid P) = \mathbf{val}(b_\mathrm{a}, C_{S\mathrm{j}} \mid P)$
        end
    end
end

What this program does is the following: for each pair of guards G_i and G_j, we check for a non-zero overlap of the pair of guards given the invariant I. If a pair of guards does overlap (i.e., if it is conceivable that they should both be enabled at the same time), we must check the equivalence of expressions sent on the same channels; the channels that are mentioned for both the guarded commands are given by the expression $\mathbf{vars}(C_{S\mathrm{i}}) \cap \mathbf{vars}(C_{S\mathrm{j}})$. The last step is checking that the values that are sent do match; we check this conditionally on every bit of the two's-complement vector of BDDs, given the overlap condition P—this is denoted by **assert** $\mathbf{val}(b_\mathrm{a}, C_{S\mathrm{i}} \mid P) = \mathbf{val}(b_\mathrm{a}, C_{S\mathrm{j}} \mid P)$.

He that yet insists that we should forbid `x!0,x!0` cannot cite the complexity of determinism-checking in his favor: it would be quite as difficult to figure out, as we anyway must, which guards overlap as it is to carry out the described determinism checks; the programmer's extra freedom coming at so little cost, we should be miserly to deny him it.

7.6.2 Data encoding

Once the compiler front-end has checked the determinism conditions, it generates the expressions for assignments and sends in terms of two's-complement BDD expressions. Expressions that can take on more than two values, e.g., for those variables declared `1of4`, `1of8`, etc., are thus represented by a vector of BDDs.[9] The expressions are first generated in terms of sets of guard-value pairs (G, E) for sends and assignments and simply guards G for receives; the list entries are collectively called actions.

[9]It is a current implementation-restriction that variables must be of the form `1of`n, where n is a power of two. The `e` in `e1of2` is present for historical reasons only; being logical, we should write `1of2` in the declaration and leave the `e` to be filled in by the QDI-compiler back-end.

Let us use as an example the two-way, 1-of-4 merge:

```
define merge2_4(e1of2 c; e1of4 la, lb , s)
{
  communicate {
    c==0 -> s!la,la?,c?;
    c==1 -> s!lb,lb?,c?;
  }
}
```

The BDD representation will be: for `c`, b_c; for `la`, the vector $[b_{la,0}, b_{la,1}]$; for `lb`, the vector $[b_{lb,0}, b_{lb,1}]$; and for `s`, the vector $[b_{s,0}, b_{s,1}]$. The guard-value set for `s` is $\{(b_c, [b_{lb,0}, b_{lb,1}]), (\neg b_c, [b_{la,1}, b_{la,0}])\}$.

The compiler's first step towards generating the logic expressions for sends and assignments from the BDD representations is to loop through all the possible values v for the outputs. If we consider an action variable x, then we may state the condition c that v is sent on or assigned to x by a single action (G, E) thus: $c = (E = v) \wedge G$. Since we have ensured that actions are non-interfering, we can aggregate the conditions for v on x for each of the actions in the action set for x, A_x; we now introduce x_v as being the (unqualified) condition upon which the value v is sent on or assigned to x:

$$x_v = \bigvee_{i \in A_x} (E_i = v) \wedge G_i \tag{7.1}$$

In terms of our example, we may illustrate by considering $\mathtt{s}_2$: $v = 2$ is equivalent to $[\mathbf{false}, \mathbf{true}]$. Considering the first element of the guard-value set, we may compute $c = (E = v) \wedge G$: $\neg b_{lb,0} \wedge b_{lb,1} \wedge b_c$; considering the second, we compute $\neg b_{la,0} \wedge b_{la,1} \wedge \neg b_c$. Hence

$$\mathtt{s}_2 = \neg b_{lb,0} \wedge b_{lb,1} \wedge b_c \vee \neg b_{la,0} \wedge b_{la,1} \wedge \neg b_c. \tag{7.2}$$

The next issue that we need to handle is that the expression that we have computed for determining whether we shall send a value x_v is in terms of a BDD on the two's-complement representation of PL1 variables, whereas x_v itself is already suitable for a unary encoding. Hence we shall have to convert the representation of the BDD for x_v to a unary representation. Substituting unary expressions for the two's-complement expressions is the most straightforward way of doing this. We introduce the unary "rails expression" $r_{x,i}$ as the BDD describing the condition when action variable x takes on value i; we now have that we should in our example replace $b_c \mapsto r_{c,1}$ and $b_{la,0} \mapsto r_{la,1} \vee r_{la,3}$.

We also have the useful invariants, due to the 1-of-n encodings, that

$$\forall x :: \forall i :: \forall j : j \neq i : r_{x,i} \Rightarrow \neg r_{x,j}. \tag{7.3}$$

Returning to our example, we see that we may write $\mathbf{s}_2$ in terms of the r's as

$$\mathbf{s}_2 = \neg(r_{lb,1} \vee r_{lb,3}) \wedge (r_{lb,2} \vee r_{lb,3}) \wedge r_{c,1} \vee \neg(r_{la,1} \vee r_{la,3}) \wedge (r_{la,2} \vee r_{la,3}) \wedge \neg r_{c,1}. \tag{7.4}$$

It is immediately obvious that some simplifications can be made; e.g., we observe that $r_{lb,3}$ is unnecessary in $\neg(r_{lb,1} \vee r_{lb,3}) \wedge (r_{lb,2} \vee r_{lb,3}) \wedge r_{c,1}$ since it appears in the form $\neg x \wedge \neg r_{lb,3} \wedge (r_{lb,2} \vee r_{lb,3})$, and $r_{lb,2} \Rightarrow \neg r_{lb,3}$. Following this hint, we simplify using Equation 7.3 and get that

$$\mathbf{s}_2 = r_{lb,2} \wedge r_{c,1} \vee r_{la,2} \wedge \neg r_{c,1}. \tag{7.5}$$

This is almost what we should like to see, but $\neg r_{c,1}$ is cause for concern. The reader will recall that our final objective is the generating of domino logic.[10] The evaluation part of domino logic consists of n-transistors only, and with the data encodings that we use, we cannot directly test a negative expression like $\neg r_{c,1}$. What we should realize is that $\neg r_{c,1}$ is not to be understood as testing that "the voltage on the circuit node $c.1$ is close to *GND*"—after all, we have not brought up anything at all about circuits in our discussion of PL1, so why should we think this?—instead, it is to be understood as meaning "the value of $c.1$ will not become close to *Vdd* on this execution cycle": the very same statement that can in a slack-elastic system only be tested by "the value of $c.0$ has become close to *Vdd*," i.e., we must replace $\neg r_{c,1} \mapsto r_{c,0}$, and we should similarly treat any other negated literals that remain after simplification. Once we have done this, we may directly identify the $r_{x,i}$ BDD literals with the actual circuit nodes $x.i$.

7.7 PL1-compiler back-end

The PL1-compiler back-end is implementation-technology dependent, and therefore what we learn here need not apply to all back ends; broadly speaking, the back ends that the author has implemented have fallen into two categories: circuit generators and software generators.

The software generators are useful for fast high-level simulation that captures enough of the synchronization behavior of processes to ensure that the system being designed does compute the right thing and does not deadlock or exhibit other unwanted characteristics; simulation at this level

[10]While this discussion is phrased in terms of circuits, the reader should bear in mind that it applies equally well to software implementations that use shared variables: the naïve implementation of $\neg r_{c,1}$ that we avoid in the hardware would in the software involve the testing of a channel's being empty, i.e., a negated probe. Either naïve implementation destroys slack-elasticity, whence they must be avoided and the semantic interpretation that we take in the text must be substituted.

is even useful for capturing reasonably accurate performance estimates. Simulation at this level is much faster than what is possible with PRS-level simulators (on the order of two to three orders of magnitude).

7.7.1 Slack

Because predicting the exact amount of slack that shall be present in a circuit implementation of a PL1 program can be difficult (we shall see reasons why this may be so), we desire that the software-simulator implementation of the program should have an amount of slack that helps in finding bugs. Manohar has proved [45] that adding slack to certain kinds of deterministic and non-deterministic systems (which he calls slack-elastic) cannot change the degree of nondeterminism or cause deadlock, whereas it is obvious that removing slack may cause deadlock; hence the software simulator should provide, as far as possible, at most as much slack as the hardware implementation. Things having been thus arranged, we should know that if the software implementation runs correctly, then the hardware implementation, which has at least as much slack everywhere, must also run correctly.

Why should it not be entirely obvious how much slack a hardware implementation of a PL1 program shall have? The answer is that we should prefer allowing the compiler back-end to adjust the amount of slack, if it can thereby improve the circuit implementation.

Let us consider two examples. First, the full-adder:[11]

```
define fa(e1of2 a,b,c, s,d)
{
  communicate {
    true -> a?,b?,c?,s!(a+b+c)&0x1,d!!!((a+b+c)&0x2);
  }
}
```

If we compile `fa` into a circuit (either STAPL or QDI), we find that the obvious production rules for the carry-out `d` have the form

$$\cdots \wedge ((a.0 \wedge b.0 \wedge c.0) \vee (a.0 \wedge b.0 \wedge c.1) \vee (a.0 \wedge b.1 \wedge c.0) \vee (a.1 \wedge b.0 \wedge c.0)) \;\rightarrow\; d\_.0{\downarrow}$$
$$\cdots \wedge ((a.1 \wedge b.1 \wedge c.1) \vee (a.1 \wedge b.1 \wedge c.0) \vee (a.1 \wedge b.0 \wedge c.1) \vee (a.0 \wedge b.1 \wedge c.1)) \;\rightarrow\; d\_.1{\downarrow}\,,$$

where $\cdots$ stands for technology-dependent control signals. Because a slack-elastic system's correctness depends only on the sequence of values sent on its channels and not on the timing of those

[11] The syntax of `d!!!((a+b+c)&0x2)` is something of a puzzle to the uninitiated, but should not frighten him that has experience with both C and CHP programming: the first exclamation represents the channel send and the next two represent logical inversions. In Modula-3, the same statement would be written far more clearly, e.g.: `d.send(Word.And(a+b+c,2)#0)`.

values, and because we may assume that a correct system does not deadlock, we may infer that the expression $a.0 \wedge b.0 \wedge c.0 \vee a.0 \wedge b.0 \wedge c.1$ may be "simplified" to $a.0 \wedge b.0$. This is especially desirable for the full-adder, because cascading full-adders into an n-bit adder will lead to a design whose latency is limited by the length of the carry chain; if we do not make the "simplification," then n will always determine the input-to-output latency of the circuit, since the carry information must always propagate from the least significant bit to the most significant bit, regardless of the disposition of the data; if on the other hand we do "simplify," then what matters is only the length of the longest string of carry-propagates for the particular pair of n-bit numbers being added. (Asynchronous designers are familiar with this result; but see Section 8.4.2 for some disturbing & relevant observations.) There is really no reason for avoiding the replacement: the circuit will be simpler and faster, and it will have more slack, slack that may allow the system to run faster because there are fewer data dependencies.

Secondly, let us consider the two-way merge:

```
define merge(e1of2 c,la,lb,r)
{
  communicate {
    true -> c?;
    c==0 -> r!la,la?;
    c==1 -> r!lb,lb?;
  }
}
```

For this program, the obvious production rules would be as follows:

$$\cdots \wedge (c.0 \wedge la.0 \vee c.1 \wedge lb.0) \;\rightarrow\; r\_.0{\downarrow}$$
$$\cdots \wedge (c.0 \wedge la.1 \vee c.1 \wedge lb.1) \;\rightarrow\; r\_.1{\downarrow}$$

Can we make the same sort of "simplification" as we did for the full-adder? Yes and no. In logical terms, the "simplification" can be stated as a weakening of the production rules that respects the determinism conditions; any such weakening is permitted. In `merge`, we are permitted to weaken the above thus:

$$\cdots \wedge (c.0 \wedge la.0 \vee c.1 \wedge lb.0 \vee la.0 \wedge lb.0) \;\rightarrow\; r\_.0{\downarrow}$$
$$\cdots \wedge (c.0 \wedge la.1 \vee c.1 \wedge lb.1 \vee la.1 \wedge lb.1) \;\rightarrow\; r\_.1{\downarrow}$$

But how is this a "simplification"? We have added slack by logical weakening, as before, but the circuit has now become more complicated—it has more transistors than before the transformation. Except under special circumstances, we should probably avoid this kind of transformation. And he that would say that the extra transistors are a small price well worth our paying would be wise to refer to Section 6.2.1: when the $la.0 \wedge lb.0$ disjunct is enabled (**true**), the input value on c is completely ignored, and we shall have to add completion circuitry; the price was not so small after all!

7.7.2 Logic simplification

The previous section makes it clear that there are choices to be made at the circuit level. Given the BDD representation of $c.0 \wedge la.0 \vee c.1 \wedge lb.0$, which need indeed not look at all similar to the straightforward sum-of-products form $c.0 \wedge la.0 \vee c.1 \wedge lb.0$, what production rule[12] should we generate? Should we weaken maximally? Not at all?

The answers to these questions depend, of necessity, on things such as the implementation technology, and in general we should not be surprised to learn that the optimal answers vary from situation to situation, even within a single system implemented in a single technology. Instead of examining all the possible cases, we shall develop a heuristic procedure for going from the BDDs to production rules that are reasonably efficient. In particular, this procedure makes the "right" choices for both `fa` and `merge` of the previous section.

Let us take as an example the merge logic described above. The details of the structure of the BDD representing $c.0 \wedge la.0 \vee c.1 \wedge lb.0$ need not concern us overly here, because we are not going to make any more use of the special properties of the BDD data structure; the main thing for us to remember about it is that it looks nothing like what we want for our circuit implementation: in fact, it happens to have the form (the particulars depend on an arbitrary variable ordering, so this is only an example):

$$\begin{aligned} r_{c,0} \wedge (r_{la,0} \wedge \mathbf{true} \vee (r_{c,1} \wedge (r_{lb,0} \wedge \mathbf{true} \vee \mathbf{false} \wedge \neg r_{lb,0}) \vee \mathbf{false} \wedge \neg r_{c,1}) \wedge \neg r_{la,0}) \\ \vee (r_{c,1} \wedge r_{lb,0} \wedge (\mathbf{true} \vee \mathbf{false} \wedge \neg r_{lb,0}) \vee \mathbf{false} \wedge \neg r_{c,1}) \wedge \neg r_{c,0} \end{aligned} \tag{7.6}$$

Following the procedure for negated literals we mentioned above, we should arrive at—now represented as a sum-of-products expression—

$$c.0 \wedge la.0 \vee c.1 \wedge lb.0 \wedge c.1, \tag{7.7}$$

where the extra $c.1$ is obviously superfluous.

How do we know that the $c.1$ is superfluous? Quite simply because

$$c.0 \wedge la.0 \vee c.1 \wedge lb.0 \wedge c.1 \equiv c.0 \wedge la.0 \vee c.1 \wedge lb.0 \tag{7.8}$$

[12] Here we are not even considering the important question of how we should convert production rules into transistor networks; in this process, there are also choices to be made.

for all values of the literals. More generally, we are interested in not the unqualified equivalence of boolean expressions, but rather in their equivalence under known invariants. Hence if two boolean expressions B and C satisfy

$$B \wedge I \equiv C \wedge I \tag{7.9}$$

for all values of their literals, where I is some invariant known to be true of the system, then we should pick between B and C the one that we should prefer to implement; this choice would commonly be guided by which of B or C has better performance or lower cost. The weakest invariant is **true**, which was yet enough for the trivial example of removing $c.1$. More commonly, we shall use the invariant of Equation 7.3; this way, we should for instance see that we could simplify $c.0 \wedge c.1$ as **false**.

In fact, several transformations that we do can be treated as boolean simplifications under various "invariants." Taking advantage of this, we introduce three separate boolean expressions, as follows:

- The *invariant,* $\mathcal{I}$: this is the invariant of Equation 7.3 strengthened with any invariants that the user should care to specify in the PL1 source code. For `merge`, $\mathcal{I}$ is $\neg(c.0 \wedge c.1) \wedge \neg(la.0 \wedge la.1) \wedge \neg(lb.0 \wedge lb.1)$.[13]

- The *slack-elastic invariant,* $\mathcal{S}$: this is what can always be true in a slack-elastic system, namely the statement that *some* value has arrived on each of the input channels. For `merge`, $\mathcal{S}$ is $(c.0 \vee c.1) \wedge (la.0 \vee la.1) \wedge (lb.0 \vee lb.1)$. (This is not really an invariant at all, but we call it that anyhow because we use it in the same way as the real invariant.)

- The *eventual invariant,* $\mathcal{E}$: this is what eventually must be true of the inputs to a process if the system is not to deadlock; in other words, $\mathcal{E}$ is exactly the progress condition under which a process shall finish its current execution cycle. For `merge`, $\mathcal{E}$ is $(c.0 \wedge (la.0 \vee la.1)) \vee (c.1 \wedge (lb.0 \vee lb.1))$. (This is a bit more like the usual notion of an invariant than $\mathcal{S}$.)

We use a simple, greedy algorithm for simplification of the sum-of-products expressions; the Modula-3 code for it is given in Figure 7.3. In English, these are the steps:

- First, clean out any disjuncts that must be **false** under $\mathcal{I}$.

- Secondly, try removing literals from the disjuncts, starting with the *longest disjunct first*—the disjuncts are sorted so that this should be easy. The simplifying invariant that is appropriate for this operation is $\mathcal{I} \wedge \mathcal{S}$. The reason we can remove literals under $\mathcal{S}$ is that removing literals is a logical weakening operation, which hence increases the slack of the process; any behaviors that we thus introduce are allowable under the assumptions of slack-elasticity (this is the weakening that we spoke of in Section 7.7.1).

[13] Whether we choose to include output variables in the invariants has no effect on the simplification procedure; in any case, we leave them out here to keep down the typographical clutter.

```
TYPE
  Disjunct = REF ARRAY OF SopLiteral.T;
  Rep = REF ARRAY OF Disjunct;

  Public = OBJECT METHODS
    init( from : Bool.T ) : T; (* initialize from a Bool.T literal *)
    toBool() : Bool.T;
    invariantSimplify( invariant,
                       disjunctiveInvariant,
                       eventualInvariant : Bool.T) : T;
  END;

  T = Public BRANDED "Sop Expression" OBJECT
    rep : Rep;
    bool : Bool.T;
  OVERRIDES
    init := Init;
    format := Format;
    toBool := ToBool;
    invariantSimplify := InvariantSimplify;
    map := Map;
  END;
```

Figure 7.2: Relevant parts of declaration of sum-of-products data structure in `Sop.i3`.

- Thirdly, try removing whole disjuncts, again starting with the longest disjunct first. We cannot use $\mathcal{S}$ now because while $\mathcal{S}$ *may* always be true, there is no guarantee that it *will*: we cannot, in the hope that we may simplify the transistor networks, force a process to wait for an input on a channel that is not going to be used on the present cycle: that input may never arrive, whence the system may deadlock. On the other hand, all that we need to do is to avoid deadlock; since $\mathcal{E}$ specifies the conditions that must anyway obtain for avoiding deadlock, we can use $\mathcal{I} \wedge \mathcal{E}$ for the simplifying invariant here.

- Lastly, if any one of the simplifying steps should succeed, then recurse.

Referring to the Modula-3 code, we should make it clear that `Bool` represents the BDD library: even though we are here simplifying sum-of-products expressions, we convert the expressions back into BDDs so that we may conveniently check the logical equivalences that we must check. `Bool.And`, etc., are the BDD library routines for performing logical operations on BDDs.

The `InvSimplify` routine is normally called as a method on a sum-of-products–expression object; this expression is referred to as `self` in the code. Normally, the routine would be called with $\mathcal{I}$ in `inv`, $\mathcal{S}$ in `weakeningInv`, and $\mathcal{E}$ in `eventualInv`. The sum-of-products expression itself is declared as shown in Figure 7.2; i.e., the data structure is an array of arrays of literals, each of which may be negated.

So far, we have phrased the boolean-simplification problem in terms of simplifying the logic used

```
PROCEDURE InvSimplify(self : T; inv, weakeningInv, eventualInv : Bool.T) : T =
  VAR
    res := Copy(Simplify(self)); (* pre-process *)
    fullInv := Bool.And(inv,weakeningInv);
  BEGIN
    SortSopDisjunct.Sort(res.rep^);
    (* first remove all disjuncts that are false under the inv *)
    FOR i := LAST(res.rep^) TO FIRST(res.rep^) BY -1 DO
      IF Bool.And(FromDisjunct(res.rep[i]).toBool(), inv) = Bool.False() THEN
        res.rep := DeleteDisjunct(res.rep,i)
      END
    END;
    VAR simplify := FALSE; BEGIN
      FOR i := LAST(res.rep^) TO FIRST(res.rep^) BY -1 DO
        (* for each disjunct, try removing literals, one by one *)
        WITH c = res.rep[i] DO VAR oldc : Disjunct; BEGIN
            FOR j := LAST(c^) TO FIRST(c^) BY -1 DO
              oldc := c;
              c := DeleteLiteral(c,j);
              IF Bool.And(res.toBool(),fullInv) =
                 Bool.And(self.toBool(),fullInv) THEN
                simplify := TRUE;
              ELSE c := oldc END
            END
        END END
      END;
      IF simplify THEN
        RETURN res.invSimplify(inv,weakeningInv,eventualInv)
      END
    END;
    VAR oldRep := res.rep; BEGIN
      (* try removing disjuncts *)
      FOR i := 0 TO LAST(res.rep^) DO
        res.rep := DeleteDisjunct(res.rep,i);
        IF Bool.And(res.toBool(),eventualInv) =
           Bool.And(self.toBool(),eventualInv) THEN
          RETURN res.invSimplify(inv,weakeningInv,eventualInv)
        ELSE res.rep := oldRep END
      END
    END;
    RETURN res
  END InvSimplify;
```

Figure 7.3: Modula-3 code for boolean simplification.

for computing the output values. We can use the same simplification methods for simplifying the control signals introduced in Section 6.3.3.1 for the handling of general conditional communications.

7.7.3 Code generation

At this point, we have seen how we should generate the logic production-rules. Our compilation job is now mostly done. What we have left is the code-generation phase.

For the most part, code generation for STAPL circuits is straightforward; it consists chiefly of adding control signals in the manner described in detail in Chapter 5 and then formatting and printing the resulting logical equations. There is little flexibility in this and few real traps that we could fall into. Mainly, we need to be concerned with whether we need the "pattern" logic block, owing to the presence of conditional communications (Section 6.3.3.1); whether we need to use a pulse generator that can be re-armed in several ways, owing to the process's having conditional sends (Section 6.3.3.3); whether we need to add foot transistors, owing to the presence of multiple outputs (Section 6.1.3); and whether we need to add extra completion circuitry, owing to the presence of inputs that go unchecked by the logic computations (Section 6.2.1).

We having already decided on the compilation method to the extent described in Chapter 5, the only real difficulty that remains is the detecting of inputs that go unchecked by the logic computations. As should be clear from what we have said above of boolean-logic manipulations, an input's being acknowledged by an output can be affected by these manipulations. Whereas we could design the compiler to take these manipulations into account (in the best of worlds, the compiler should treat the boolean-logic manipulations and the completion-circuitry generation as a single problem); this has not yet been done, and may even be an unreasonable thing to ask for. The current compiler uses a simple (and safe) heuristic for determining whether an input will always be acknowledged by the generated outputs. The heuristic works well for all cases that have so far been tested, and it allows a more modular compiler-design than would be possible with a more complicated and accurate method.

The heuristic we use for checking whether a value on an input channel L is acknowledged by the outputs has two parts, both of which err on the safe side:

- First, check if the logic of the computed outputs is such that the outputs *must* acknowledge the input in question. This is the case if, regardless of the disposition of the other inputs, it is *always* the case that the value on L will affect the computed output—i.e., if regardless of the other inputs, the value arriving on L can always force a choice between at least two alternative outputs on some output channel. If this is so, then no amount of boolean manipulation can remove the outputs' checking of the L-value.

- Secondly, check the generated logic for each output channel: does it require the presence of one literal of L before it will produce an output? The conditions that we can thus determine that L will be acknowledged under are the union of the communication conditions for the output channels that contain a literal of L in every disjunct of their output logic.

If either one of the two checks should always succeed, then L is known to be acknowledged by the outputs, and no further checking is necessary. Strictly speaking, we should prefer using only the first condition (since this is the modular one—the second condition is implementation-dependent), but the author has unfortunately found that processes requiring the second condition's being checked before they would compile without unnecessary input-completion circuitry are fairly common.

Chapter 8

A Design Example: The SPAM Microprocessor

NONSENSE, *n. The objections that are urged against this excellent dictionary.*
— *Ambrose Bierce,* The Devil's Dictionary *(1881–1906)*

In the Introduction (Chapter 1), we said—following Carver Mead—that VLSI is a statement about system complexity, not about circuit performance or sheer circuit size. Consequently, the touchstone that shall determine whether a way of building VLSI is worth pursuing must involve the designing of complex systems. Unfortunately "system complexity" is impossible to define directly and objectively. The best we can do is to design a real system of at least moderate complexity. If we should find that the system so designed performs well or was particularly easy to design or had some other attractive feature, then we should know that we are on the right track.

And this is why this chapter has been written: the only way we could possibly tell whether STAPL circuits are any good or whether the PL1 language is at all useful is by designing, with them, a complex concurrent system.

8.1 The SPAM architecture

The SPAM (Simple Pulsed Asynchronous Microprocessor) architecture is defined in detail in Appendix B. The SPAM architecture defines a simple 32-bit RISC instruction set. It defines eight registers and a number of integer operations, and it is generally intended to be easy to implement without making any real sacrifices of functionality. The instruction set is completely orthogonal; i.e., all instructions have the same addressing modes, always specified by the same bit fields in the instruction word.

Given what we are trying to accomplish and the resources that have been available, the SPAM processor is somewhat more complicated than would have been attempted with the same resources, had for instance the PL1 language not existed. We make no comments about the designer's rela-

$$
\begin{array}{l}
\mathit{SEQSPAM} \equiv \\
*[\ i\ :=\ \mathit{imem}\texttt{[}\mathit{pc}\texttt{]};\\
\quad \mathit{opx}\ :=\ \mathit{gpr}\texttt{[}i.\mathit{rx}\texttt{]},\ \ \mathit{opy}\ :=\ \mathit{YMODE}(i.\mathit{ymode})(\mathit{gpr}\texttt{[}i.\mathit{ry}\texttt{]}, i.\mathit{imm});\\
\quad \mathit{opz}\ :=\ \mathit{OP}(i.\mathit{opcode})(\mathit{opx}, \mathit{opy}),\ \ \mathit{pc}\ :=\ \mathit{PCOP}(i.\mathit{opcode})(\mathit{pc}, \mathit{opx}, \mathit{opy});\\
\quad \mathit{gpr}\texttt{[}i.\mathit{rz}\texttt{]}\ :=\ \mathit{opz}\\
\ ]
\end{array}
$$

Figure 8.1: Sequential CHP for SPAM processor.

tive abilities, and appeal only to general notions of honesty when it comes to the accuracy of the simulations.

8.2 SPAM implementation

The sequential SPAM is specified by the remarkably simple program of Figure 8.1; this program is a restatement of the English description in the appendix of how the processor executes instructions.

8.2.1 Decomposition

We shall study the decomposition of the SPAM processor into the processes shown in Figure 8.2; the decomposition is similar to but not identical to the one chosen for the MiniMIPS.

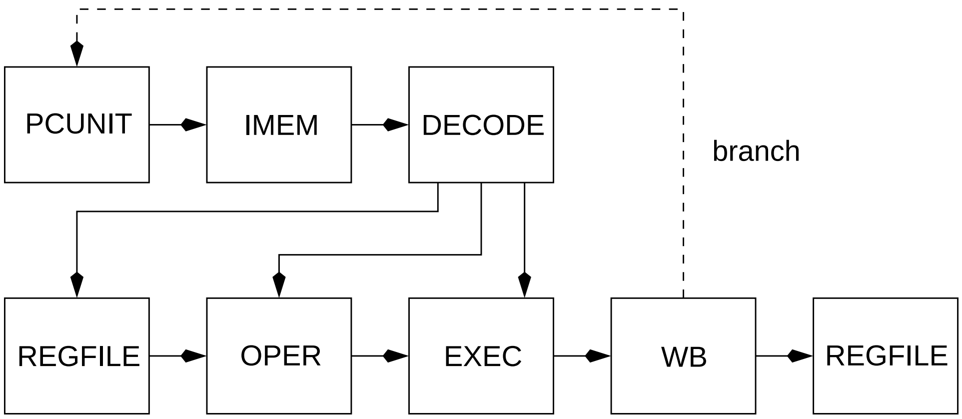

Figure 8.2: Overview of SPAM decomposition.

Seven units are identified in the figure:

- *PCUNIT*, responsible for generating the program-counter values representing the addresses in the instruction memory of the instructions that are to be executed. *PCUNIT* corresponds to the operation $pc := PCOP(i.opcode)(pc, opx, opy)$ of *SEQSPAM*.

- *IMEM*, the instruction memory. In the simple test-processor we are speaking of here, there is no off-chip memory; i.e., *IMEM* is a memory, not a cache. *IMEM* corresponds to $i := imem\texttt{[}pc\texttt{]}$.

- *DECODE*, the instruction-decode unit. This unit generates the control signals for the units that are to execute the fetched instruction. *DECODE* corresponds to computing $i.ymode$, $i.rx$, $i.ry$, $i.imm$, $i.opcode$, and $i.rz$.

- *REGFILE*, the register file. It contains eight registers. It appears twice in the figure, which signifies that it conceptually acts twice for each instruction that is executed: once to fetch the operands and once to write back the result. *REGFILE* corresponds to computing $gpr\texttt{[}i.ry\texttt{]}$, $opx := gpr\texttt{[}i.rx\texttt{]}$ and performing $gpr\texttt{[}i.rz\texttt{]} := opz$.

- *OPER*, the operands-generation unit. This unit is responsible for computing opy in Figure 8.1; hence it contains a conditional shift-add combination (see table on p. 190). *OPER* corresponds to $YMODE(i.ymode)(gpr\texttt{[}i.ry\texttt{]}, i.imm)$.

- *EXEC*, the instruction-execution unit. This unit internally consists of several sub-units: an arithmetic-logic unit (ALU), a shifter, and a data-memory unit. *EXEC* corresponds to $OP(i.opcode)(opx, opy)$. In this decomposition, it also contains the part of $PCOP(i.opcode)(pc, opx, opy)$ that uses the registers, i.e., the branch comparator.

- *WB*, the writeback unit. This unit is responsible for canceling instructions whose results should not be written back (see Section 8.2.2); it also notifies the *PCUNIT* of taken branches. *WB* is not present in *SEQSPAM*, because it is used only for providing sequencing in the decomposed, concurrent version.

8.2.2 Arbitrated branch-delay

Most programs that are run on general-purpose processors have unpredictable control-flow; they are not simple, straightline programs. The straightline program-flow is interrupted by branches or exceptions; it is well-known that programs for these processors execute on average only five to ten instructions for every branch that they execute. If we treat exceptions similarly to how we treat branches, the rate increases further: on some architectures nearly every instruction may raise an exception.

Especially if we treat exceptions and branches together, it is clear that processor performance can be improved by adding some sort of *branch prediction* mechanism. Such a mechanism has two fundamentally distinct parts: predicting whether a given instruction will branch, raise an exception, or do neither; and dealing with mispredictions. While the details of how we might predict whether a branch will be taken or an exception will be raised are outside the scope of our present discussion, the mechanism for dealing with mispredictions is not.

A mechanism for arbitrated precise-exception–handling, used in the MiniMIPS processor, has been presented by Manohar, Martin, and the author [47]; a similar one by Furber *et al.* [88]. The SPAM processor uses such an arbitrated mechanism for normal branches; since it does not have exceptions, there is no need for a precise-exception mechanism; but having handled branches in this way, adding exceptions should be easy.

The details of the mechanism are available in the paper, but the basic idea is very simple: the *PCUNIT* generates the sequence of program-counter values that we *a priori* believe to be the most likely. The corresponding instructions are fetched from instruction memory and executed. Results are written back to the register file and data memory in program order; if the control flow takes an unanticipated turn, the instructions that were fetched but should not be executed are yet executed, but the results of these executions are discarded. Finally, the *PCUNIT* is informed that the control flow has changed; it then begins fetching the instructions corresponding to the updated control flow.

As is easily understood from the preceding description, the arbitrated mechanism is flexible and could accommodate a wide variety of predicted control-flows. In practice, we have as yet only used it predicting a straightline control-flow.[1] In other words, the processor fetches instructions sequentially, assuming (in the MiniMIPS) no exceptions or (in the SPAM) no branches; if the assumption turns out to have been wrong, the unwanted instructions are discarded and fetching begins from the exception-handler address (in the MiniMIPS) or from the branch-target address (in the SPAM).

The arbitrated mechanism allows informing the *PCUNIT* of control-flow changes only when they occur; it becomes unnecessary to inform it, for each instruction that does *not* change the control flow, that they do not occur. This means that the fetching of instructions is effectively decoupled from the executing of them.

In the SPAM processor, the canceling of unwanted instructions and the informing of the *PCUNIT* of control-flow changes are handled by the writeback unit, *WB*. When a branch is executed and an impending control-flow change becomes apparent (in the *EXEC*), this information passes via the *WB* on a channel to the *PCUNIT*. As we noted, the communications on this channel are conditional. In Figure 8.2, this is illustrated by the channel's being drawn dashed.

8.2.3 Byte skewing

Classic QDI design-styles, such as the one used in the design of the Caltech asynchronous microprocessor [53] and the Philips Tangram system [9], treat QDI-system design in a control-centric way: first, the control structures that are necessary for implementing bare, dataless handshakes are

[1] We should point out that the MiniMIPS processor has a second mechanism, different from the one described here, for performing branch prediction; this branch predictor uses the slightly more sophisticated backward-taken–forward-not-taken ("BTFN") predictor.

designed, and then the bare channels are widened to carry data; logic for computing can be inserted in appropriate places.[2]

While this method of designing the circuits elegantly takes us from a collection of small processes that implement only the handshakes to processes that communicate (and compute) with data, the large drawback is that the slack of the system is fixed at the time that the control is designed, unless special measures are taken. For instance, handshakes between units (which for control circuitry consist only of bare wires but are much more complicated in the finished system) can limit the performance of a system.

One of the main innovations of the MiniMIPS processor project was the slack-elastic design-style [55]. The slack-elastic style allows the introducing of slack gradually during the decomposition instead of all at once at the beginning; among other things, this allows our breaking the handshake cycles into smaller pieces, thus achieving higher system throughput.

In the MiniMIPS, we distributed the control to the datapath via a logarithmic tree. In other words, if control information is required at the level of bit processes (or more commonly, at the level of 1-of-4 processes), this information is copied out in a pipelined tree. Normally, a four-way copy would copy the control information to each of the bytes, and the bytes would be designed as single, large processes, with the bit or 1-of-4 "processes" actually being fragments and not complete processes (i.e., the smallest part of the design that communicates with its environment entirely on channels is the byte-sized process).

The MiniMIPS logarithmic tree is not the only way of distributing the control. If throughput is the only concern and the latency of computation is a secondary issue (e.g., in DSP applications), each bit of the datapath can be made to copy the received control at the same time as it performs its data computation. This approach, called *bit skewing,* was used in the asynchronous filter designed by Lines and Cummings [18].

Any number of compromises can be imagined. Figure 8.3 shows three ways of distributing the control. Importantly, in a slack-elastic system, which way we finally choose is not visible in the high-level description. Datapath processes are marked "D" in the figure; the remaining processes are simple copies, although the initial copy can sometimes be combined with the preceding process. For the SPAM implementation, we choose method (c), which combines aspects of both the logarithmic-tree method and the bit-skewing method. We call this *byte skewing.*[3]

[2] A good intuitive understanding of this procedure can be had by comparing with how the Incas built suspension bridges across gullies in the Andes. First, they would send a runner through the jungle with a light rope; having done this, they would pull the rope up from the jungle floor until it ran across the gully. Then they would use the thin rope to haul a much thicker rope across, a rope thick and strong enough to carry the persons, llamas, etc., that needed to cross the bridge. The bare handshakes are the "thin ropes" and the full-fledged data channels with computation are the "thick ropes."

[3] We should note that the datapath operation's being implemented in eight-bit chunks in (a) and in two-bit chunks in (c) is an issue separate from that of byte skewing. The chief reason we choose to implement the operations in two-bit chunks in the SPAM processor is because many of the algorithms used for compiling PL1 programs (mainly the BDD code) require resources that are roughly exponential in the size of the process being compiled; hence, it is much easier to compile these smaller pieces automatically than it would be to compile the eight-bit MiniMIPS chunks.

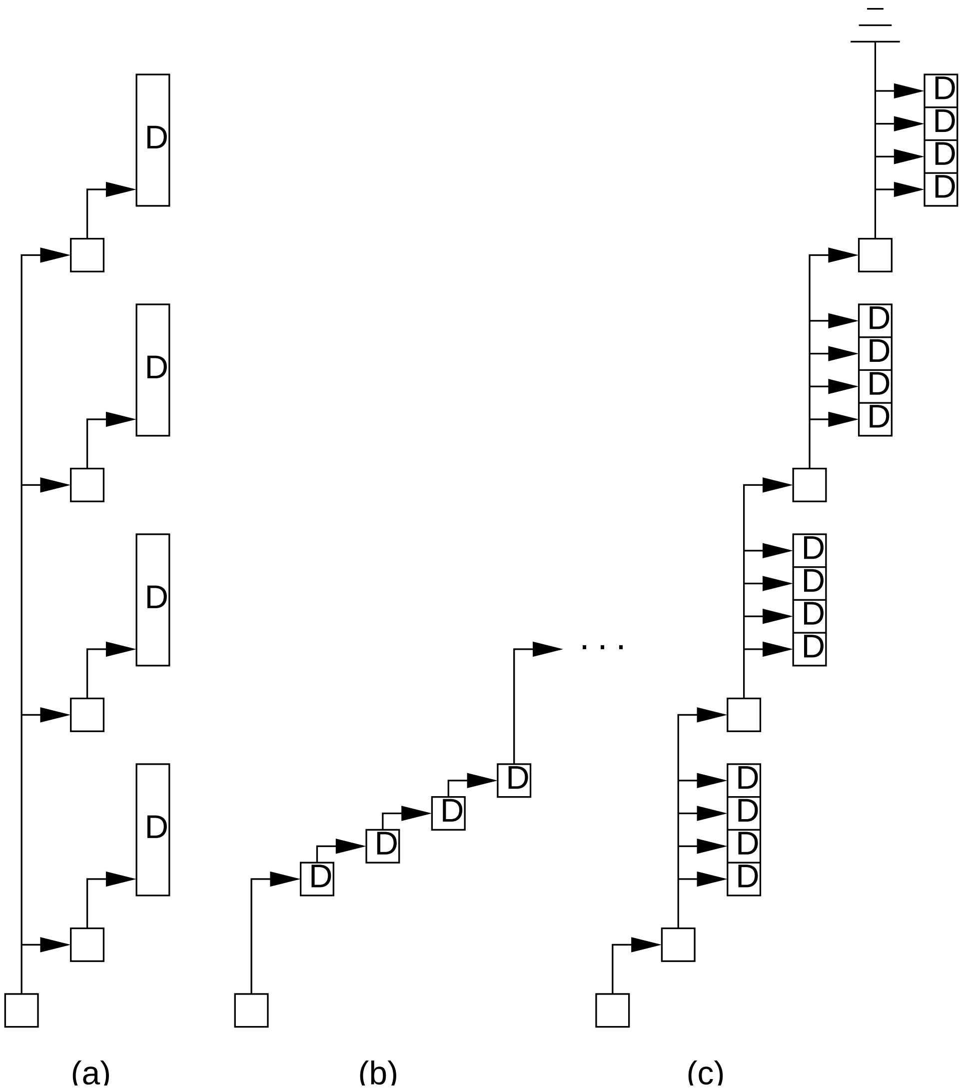

Figure 8.3: Three ways of distributing control, shown on a hypothetical datapath operating on 32 bits encoded as 16 1-of-4 values. (a) MiniMIPS method: two-stage copy to four byte-wide processes. (b) Asynchronous-filter method: linear tree (list) of control copies to 16 processes operating on 1-of-4 data (bit skewing). (c) SPAM method: linear tree of control copies to four four-way copies and thence to 16 processes operating on 1-of-4 data.

The reason we should avoid method (b) in a general-purpose processor design should be obvious: the latency penalty is simply not acceptable. But what is wrong with (a), the MiniMIPS method? Compared with it, byte skewing as in (c) has the following advantages:

- The method is easily scalable; going from 32 bits to 64 bits is simply a matter of arraying more cells. This is why we have "grounded" the top of the figure: by using a bit bucket here, we pay a small penalty of unnecessary data-replication but gain the benefit of being able to array identical datapath-cells. Scaling the datapath for method (a) involves adding an additional level to the tree as well as new wires that cross the datapath (new wiring slots must be found for these).

- The wires are shorter—no wires cross the entire width of the datapath.

- There are fewer wires; instead of $O(\log n)$ sets of wires, each enough for crossing the entire width of the datapath, there is only one such set. In the limit of wide datapaths, (c) will hence use less energy than (a).

- Byte skewing allows for simpler implementations of many arithmetic operations, e.g., addition.

- The layout is far simpler.

The importance of the shorter wires and the simpler layout should not be underestimated.

Naysayers would retort that byte skewing adds to the latency of computing, which in itself is enough for them to say no; this is true, but only to an extent. Comparing (a) and (c) in Figure 8.3, the latency difference for control to arrive at the top bit of the datapath is really only two stages (we should not count the extra pipelining that was added for other reasons); at the same time, we should realize that control, generally speaking, arrives *sooner* at the less-significant bits. In any case, the naysayers' argument is weak: the added latency matters only on branches, and the amount of added latency is insignificant compared with the average time between branches; it seems likely that the throughput advantage and simple design of the byte-skewed control distribution will outweigh it.[4]

In the SPAM implementation, byte-skewing is used in many places where it might not at first seem obvious that it is a good idea. For instance, the bits of the instruction word are rearranged so that the register identifiers `rx` and `ry` come out of the memory before the other bits of the instruction word. This way, producing the instruction operands early is possible; indeed, *earlier* than would be possible using the logarithmic-tree control distribution of the MiniMIPS.

We should also note that the second stage of the control distribution tree in (a) in many ways behaves *electrically* like a four-way copy, even though it may not do so logically. Finally, implementing the operations in this finer-grained way adds extra pipelining to the processor, the desire for which should be clear from our going from $18\,{}^2\!/_3$ transitions per cycle in the MiniMIPS to 10–12 in the SPAM.

[4]In the SPAM processor, the only arithmetic operation that gets slower under byte skewing is shifting right. But of course shifting left becomes simpler and gets faster.

8.3 Design examples

We shall now study two parts of the SPAM design to see two different ways that we can design large STAPL-based systems in. First, we shall study the *PCUNIT* of the SPAM processor; this we shall be able to understand completely as a composition of PL1 programs. Secondly, we shall study the *REGFILE*; this design example will show that the compilation methods that we gave in the previous chapter are not the only way that STAPL circuits can be compiled in.

8.3.1 The *PCUNIT*

The sequential CHP of a non-arbitrated *PCUNIT* would be

$$
\begin{array}{l}
pc \;:=\; init\_pc; \\
*[\; IMEM\_ADDR!pc;\; pc \mathrel{+}= 4; \\
\quad DOBRANCH?d; \\
\quad [\; d \longrightarrow pc \;:=\; branch\_target \;[]\; \neg d \longrightarrow \mathbf{skip} \\
\quad] \\
\;];
\end{array}
$$

the *PCUNIT* learns by reading *DOBRANCH* whether it has to branch. With the arbitrated mechanism, the program becomes instead

$$
\begin{array}{l}
pc \;:=\; init\_pc,\; va \;:=\; \mathbf{false}; \\
*[\; IMEM\_ADDR!pc,\; VA!va;\; pc \mathrel{+}= 4,\; va \;:=\; \mathbf{false}; \\
\quad [\; \overline{DOBRANCH} \longrightarrow pc \;:=\; branch\_target - 4,\; va \;:=\; \mathbf{true},\; DOBRANCH \\
\quad []\; \neg\overline{DOBRANCH} \longrightarrow \mathbf{skip} \\
\quad] \\
\;];
\end{array}
$$

the reader is referred to Manohar, Nyström, and Martin [47] for the purpose of the *VA* channel and the implementation of the negated probe $\neg\overline{DOBRANCH}$. We further add a channel, *EXPC*, for informing the *EXEC* of what it needs for computing the target of relative branches and another for reading the as yet unspecified *branch_target*, which gets us to

$$
\begin{array}{l}
pc \;:=\; init\_pc,\; va \;:=\; \mathbf{false}; \\
*[\; IMEM\_ADDR!pc,\; EXPC!pc,\; VA!va;\; pc \mathrel{+}= 4,\; va \;:=\; \mathbf{false}; \\
\quad [\; \overline{DOBRANCH} \longrightarrow BRANCH\_TARGET?pc,\; va \;:=\; \mathbf{true},\; DOBRANCH \\
\quad []\; \neg\overline{DOBRANCH} \longrightarrow \mathbf{skip} \\
\quad] \\
\;],
\end{array}
$$

where we have assumed that the branch target is computed elsewhere.

```
define pcunit_noarb() (1of(2) d; 1of(4)[16] branchto; 1of(4)[16] expc;
                       1of(4)[16] imem_addr; 1of(2) va)
{
  1of(2) bc, dup_ctrl;
  1of(4)[16] incpc, incpc2, genpc, newpc, pc2;
  1of(2)[32] addend, aug, genpc2;

  pc_sel32() psel(bc, incpc2, genpc, newpc);
  pc_copy() pcopy(newpc, imem_addr, expc, pc2);
  pc_incr() pinc(pc2, incpc); /* INCPC <- PC2 + 8 */

  /* initialize tokens:  output of incrementer gets 8, input gets 4 */
  <i:16: [ i != 1 -> reset1of(4,0) r_pc2[i](pc2[i]), r_incpc[i](incpc[i]);] >
  reset1of(4,1) r_pc2_1(pc2[1]); reset1of(4,2) r_incpc_1(incpc[1]);

  slack(4,16,3) sm_incpc(incpc,incpc2); /* slack match common case */

  /* branch path */
  addend_dup() pdup(dup_ctrl, branchto, addend);

  <i:32: [ i!=2 -> zero_gen(2) a[i](aug[i]);][ i==2 -> alternator() a2(aug[2]);]>
  pc_adder() padd(addend, aug, genpc2);

  <i:16: buf_2to4 b24_pc[i]({genpc2[2*i],genpc2[2*i+1]},genpc[i]); >

  1of(2) p_ns, p_s;
  singlewidth_slack(2,4) p_s_slack( p_ns , p_s ); reset1of(2,0) r_s_slack(p_s);

  pcunitctrl() p(d, bc, dup_ctrl, va, p_s, p_ns);
}
```

Figure 8.4: Top-level CAST decomposition of SPAM *PCUNIT* (without arbiter).

8.3.1.1 Adding slack

Originally, the implementation of the *PCUNIT* program used in the SPAM processor was designed with an amount of pipelining that could be chosen when the system is reset. This was accomplished by using a fixed datapath and a number of initial tokens that could be chosen at reset time. Considering only the *pc*-increment function of the *PCUNIT*, we could write this as the program:

$$*[L?oldpc \;\; ; \;\; R!(oldpc \;\; + \;\; n \; * \; 4)] \quad \| \quad SLACK(R, L) \;\; ,$$

where the process *SLACK* implements a high-slack channel. At reset time, this channel is initialized with n tokens, $init\_pc$, $init\_pc + 4$, $init\_pc + 8, \ldots$, $init\_pc + 4(n-1)$, corresponding to the first n *pc*-values.[5]

It turns out, however, that a much simpler design is obtained if the number of tokens is fixed. In the program that we shall see, $n = 2$.

[5] As mentioned in Appendix B, $init\_pc = 8$ in the SPAM architecture.

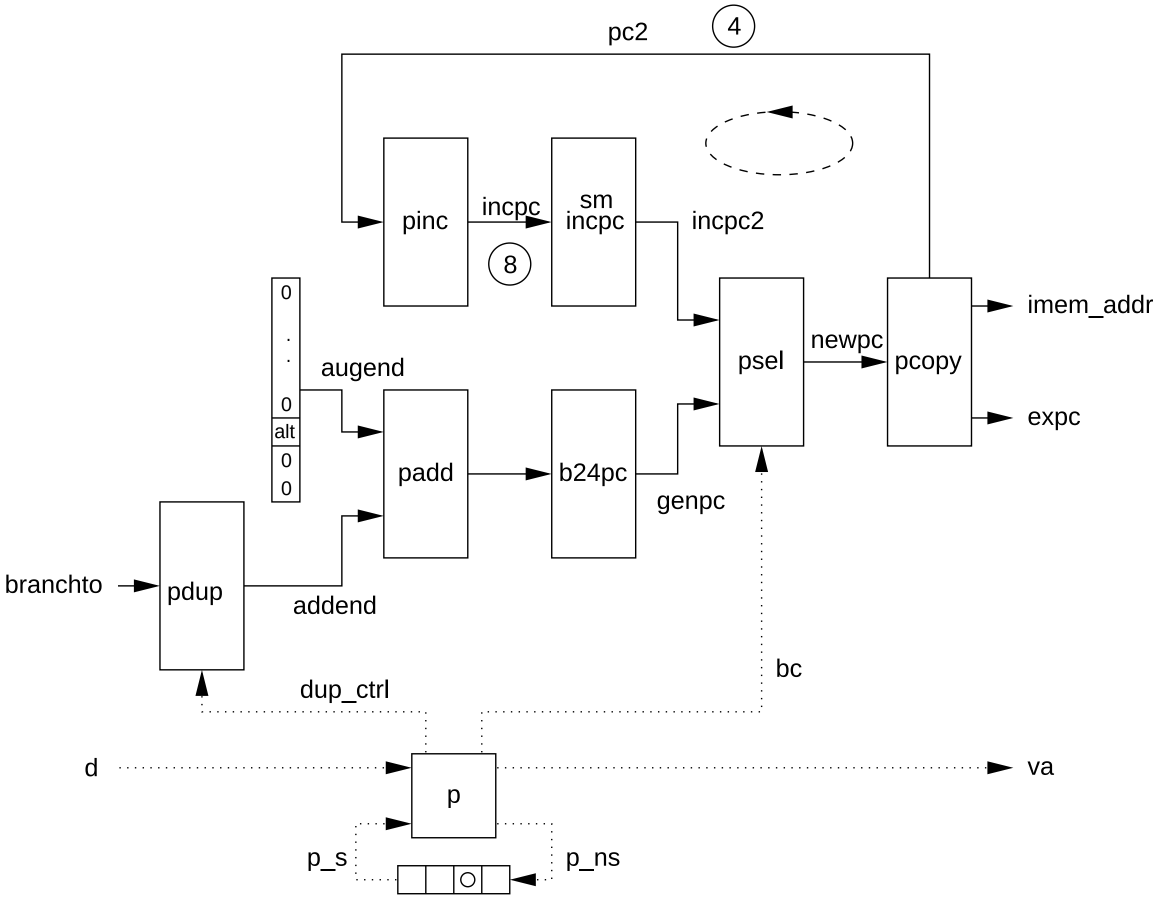

Figure 8.5: Process graph of *PCUNIT*. Data channels are drawn solid; control channels dotted. Initial tokens are shown as circles.

```
define pc_sel(e1of2 c; e1of4 incpc, genpc, newpc)
{
  communicate {
    true -> c?, incpc?;
    c == 0 -> newpc!incpc;
    c == 1 -> newpc!genpc, genpc?;
  }
}
```

Figure 8.6: PL1 program for a single 1-of-4 process of *psel*.

8.3.1.2 CAST decomposition

The top-level CAST decomposition of the *PCUNIT* (without the arbiter—the arbitrated branch-mechanism is handled outside this program) is shown in Figure 8.4. This program corresponds exactly to the CHP above, except that two *pc*-operations are in progress at the same time; the transformations used for getting hither are described by Pénzes [69].

8.3.1.3 Subprocesses

The process graph is illustrated in Figure 8.5. The top cycle in the figure is the one usually exercised: an old *pc* appears on *pc*2; *pinc* increments it by eight (since two tokens are in the *pc*-increment loop, this is the right amount to increment by); *sm_incpc* slack-matches it so that all the processes are given enough time to reset their handshakes; *psel* selects it (as long as there has been no branch); *pcopy* copies it to the various places it is needed. The bottom path, from *branchto* to *newpc*, is only used during branches. This allows a simple ripple-carry adder's being used for *padd*. The unit that follows *padd*, *b*24_*pc*, converts the result of the addition from 32 1-of-2 codes (bits) to the 16 1-of-4 codes usually used for representing the *pc* value. All the processes are byte skewed; for instance, the lower bits of an operation in *pcopy* in time overlap the higher bits in *psel*.

Branches are handled by discarding the two tokens in the *pc* loop and regenerating them. When a branch has been detected, *pdup* sends the branch target address received on *branchto* twice on the *addend* channel. The *augend* channel meanwhile carries the two tokens 0 and 4. (The alternator process driving bit 2 of *augend* accomplishes `*[`*augend*!0; *augend*!4`]`.)

As is clear from the above, *psel* is what we can call an "asymmetric select" process. It either simply reads and copies *incpc*2 to *newpc* or else it reads and discards the value on *incpc*2 and reads and copies the value on *genpc* to *newpc* (on branches). The PL1 code for a single bit of *psel* is shown in Figure 8.6.

The most complex of the *PCUNIT* processes is the control process `pcunitctrl`; this was implemented with a single PL1 program, seen in Figure 8.7.[6]

[6]The only reason that the state variable `s` in this program was implemented using a feedback loop is that the PL1 compiler as yet does not handle state variables properly; making the replacement manually would save a few

```
define pcunitctrl(e1of2 d, selctl, dctl, wbva, s, ns)
{
  /* EVENTUALLY c(d) = c(wbva) = c(selctl) */
  invariant { s == 1 #> d != 1 }
  communicate {
    true -> s?, d?;

    /* normal op */
    s == 0 && d == 0 -> ns!0, wbva!0, selctl!0;

    /* start branching */
    s == 0 && d == 1 -> ns!1, wbva!1, selctl!1, dctl!1 /* copy */;

    /* stop branching */
    s == 1 -> ns!0, wbva!0, selctl!1, dctl!0 /* pass */;
  }
}
```

Figure 8.7: PL1 program for `pcunitctrl`.

The reason that slack-matching is required (*sm_incpc*) is that the *PCUNIT* needs to produce a new *pc* every ten transitions, so the loop *pinc*–*sm_incpc*–*psel*–*pcopy*–$\cdots$ should take twenty transitions, but *pinc* takes only ten transitions; hence there are six transitions left (*psel* and *pcopy* only take two each) that need to be absorbed if we want the system to be able to run at full speed.

8.3.1.4 32-bit incrementer

The most interesting of the datapath units is the incrementer. This unit computes, on 1-of-4 data, $pc2 := pc + 8$. As mentioned above, it does this in ten transitions (i.e., five stages). However, it is still a very simple unit—the byte skewing allows this. The incrementer consists of three types of cells: a bottom adder cell for adding the actual increment, a carry cell that is specialized for adding zero plus a carry in, and a buffer for slack-matching.

Because of the byte skewing, carrying across a byte boundary costs no extra latency; however, carrying within a byte does cost. Hence, carrying across byte boundaries is done with a rippling carry, and carrying within bytes is done with a carry forwarded to the next pipeline stage. By increasing the number of bits that can be incremented at the same time, we can minimize the number of carries that need to be done within bytes, which will thus minimize the number of stages required for the increment. It appears to be practical to increment pairs of 1-of-4 codes. Hence, the incrementer overall gets the structure seen in Figure 8.8. In the figure, only the carry paths have been drawn; "S" signifies a slack-matching stage (i.e., a buffer), and "I" signifies an incrementer stage. The diagram has been redrawn in Figure 8.9; the beneficial effects of the byte skewing are here clear: the structure behaves *in time* as a pure carry-forward incrementer.

transistors, but it would also make modifying the program more difficult.

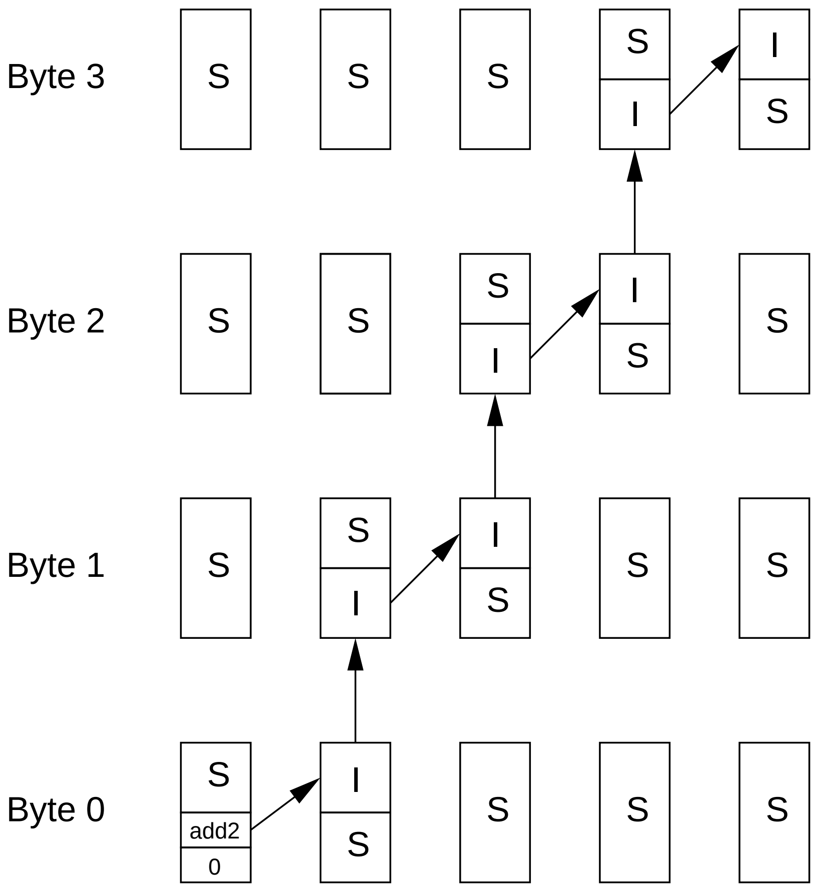

Figure 8.8: Block diagram of pc incrementer; layout alignment. Flow of data is from left to right.

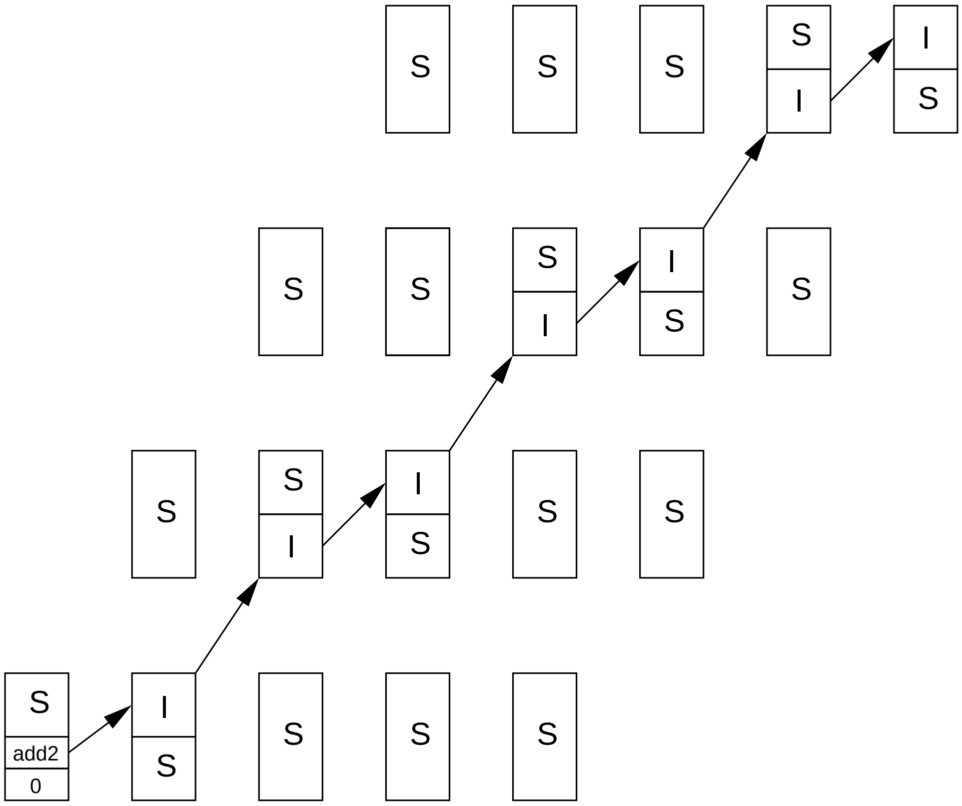

Figure 8.9: Block diagram of pc incrementer; time alignment.

8.3.1.5 Implementation and simulation

The author produced layout for the *PCUNIT* described here using the `magic` layout tool, using design rules for the HP/MOSIS 0.6-μm process (see Section 3.2.2). Most of the layout was "quick and dirty"; the transistors were sized by hand to avoid bad cases of static charge-sharing (the circuits included charge-sharing avoidance measures, as well) and for reasonable performance. Shared-transistor networks were used where performance was an issue.

The complete *PCUNIT* contains 54,786 transistors (this includes weak transistors in staticizers/bleeders and the transistors used for charge-sharing avoidance). The simulation results we shall see were obtained using the `aspice` circuit simulator without considering wiring resistance or capacitance. Because of the byte-skewed design-style and its relatively short wires, it seems likely that wiring resistance would not be an issue, even in more recent deep-submicron technologies; the extra wiring capacitance would cause a performance loss of from 20–40 percent, depending on how much the designer cares about speed relative to energy.[7]

Spice simulations show that the STAPL *PCUNIT* runs at about 1 GHz in its unwired state; this is about three times as fast as the QDI unit used in the MiniMIPS. Given that the MiniMIPS would be capable of operating at about 220 MHz if a layout bug were fixed, we should expect a fabricated STAPL *PCUNIT* to run at 650–700 MHz in the same technology.

Some simulation results are shown in Figure 8.10 and Figure 8.11. Figure 8.10 shows `expc[1]`, i.e., bits 2 and 3 of the *pc*, just after reset. Figure 8.11 illustrates the latency of detecting a branch from the arbiter input's rising at $t = 12$ ns to the control for `psel`'s being produced at $t \approx 13.3$ ns; the datapath's producing the first branched-to *pc* value takes 2–5 more stages, so the total latency is about 2 ns. Each 40-ns simulation takes about four hours to run on a 1 GHz single-processor Intel Pentium III Xeon computer with 512 megabytes of memory, running FreeBSD 4.2.

The current consumption of the *PCUNIT* is shown in Figure 8.12 and in Figure 8.13; Figure 8.12 shows the current consumption when there is no branching, whereas Figure 8.13 shows it for constant branching. For the no-branching case, the power consumption is about $1.2 \text{ A} \times 3.3 \text{ V} \approx 4$ W, or about 4 nJ per operation. While this may seem a high number (the MiniMIPS fetch unit uses about 2.6 nJ per instruction [70]), we must remember that the circuits were not carefully designed at the analog level, that they run at 1 GHz, and that whereas the power consumption is high, at least the noise-inducing dI/dt is very low. Finally, the latency due to byte skewing is illustrated in Figure 8.14; in this figure, we can see that `expc[15]` is produced about 0.5 ns later than `expc[1]`.

[7] It is fairly easy to show that if a circuit is well-balanced in the sense that its different parts run all at about the same speed and respond to sizing in about the same way and we are sizing the circuit for minimum Et^n, where n is some constant, then we should expect the optimal transistor-sizing to yield a speed that is roughly $n/(n+1)$ of the simulated speed without parasitics. For many applications, $n = 2$ is a reasonable choice [55]; this choice can also be justified on theoretical grounds, as long as we are allowed to vary the supply voltage of the system being designed. For $n = 2$ we should expect the optimally sized circuit to run about 70 percent as fast as the ones we are presenting here. (Note that the circuits we present here are not *entirely* unloaded—some wires are present, and some transistor parasitics, e.g. edge capacitances, are also present.)

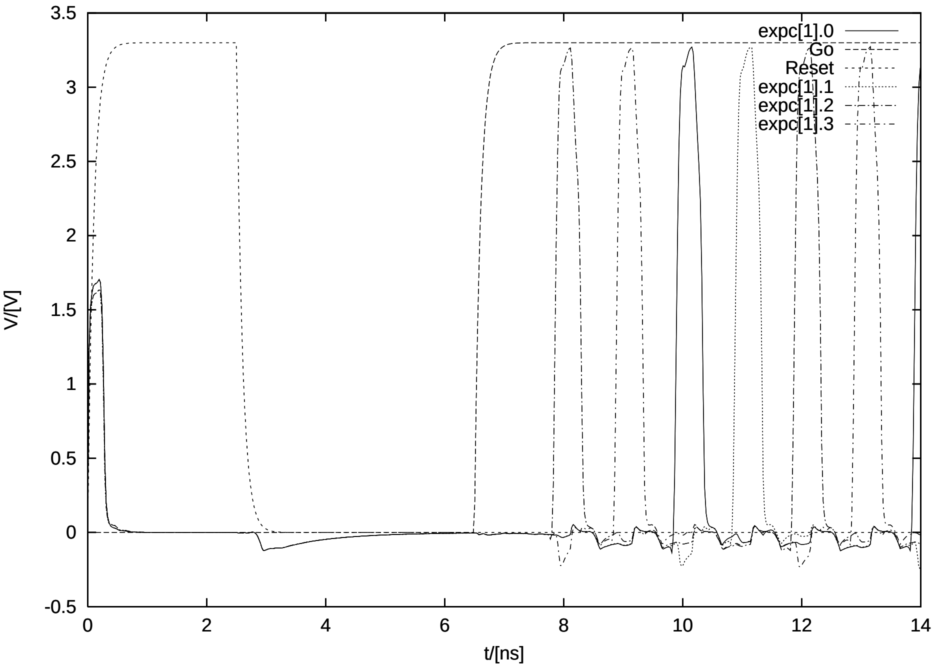

Figure 8.10: Behavior of `expc[1]` after reset; no branches.

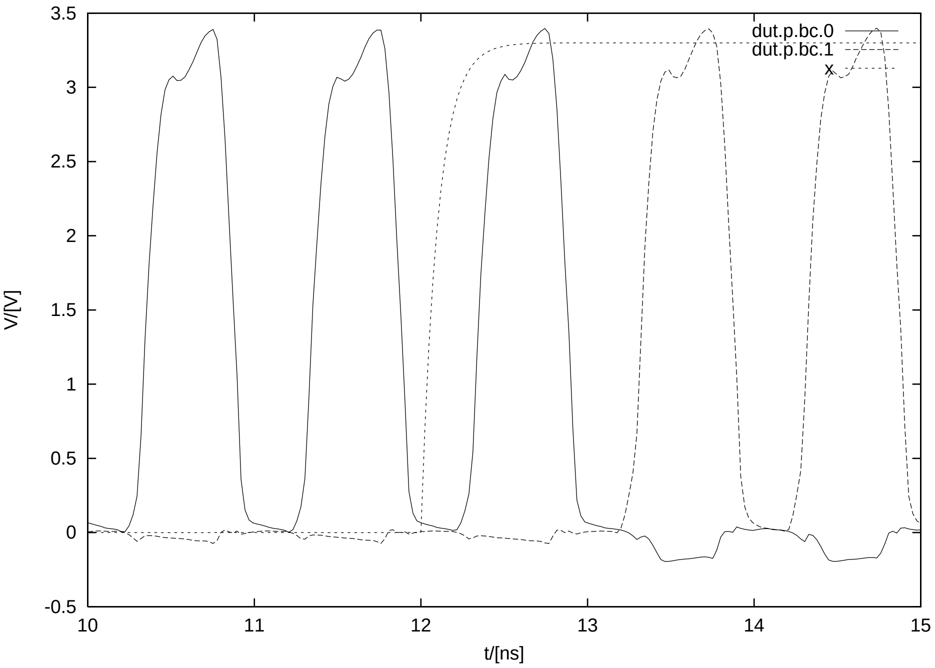

Figure 8.11: Behavior of control for pc-selector `psel`; a branch is reported at t =12 ns.

We should point out that the circuit is a simplistic one: the slack-matching of the incrementer is done with standard left-right buffers. Since the number of tokens is known at compile time, we could easily use higher-slack buffers that use less energy and fewer transistors for the same amount of slack. It seems likely that nearly half the energy could thus be saved. A little less easily, the *PCUNIT* could be redesigned to have the same input-output specification but to use an algorithm optimized for the average case; studies of the MiniMIPS have shown that even greater savings would be possible in this way.

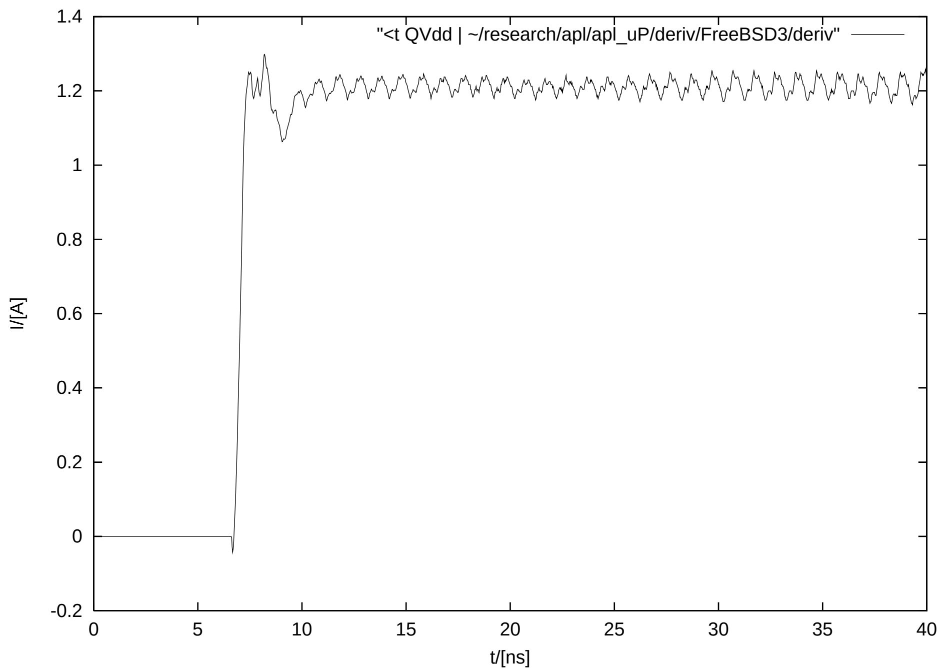

Figure 8.12: Current draw of *PCUNIT* in amperes; no branching. `Go` active at $t = 6.5$ ns.

The most difficult part of the *PCUNIT* for the circuit designer is the *pc* incrementer. In our decomposition, this unit is used on every instruction fetch; hence the number of 1-of-4 codes that can be incremented in a single stage of logic to a large extent determines how fast the whole processor can run, for a given degree of speculation. For this reason, carefully designing the *pc*-incrementer stage so it achieves the highest possible throughput and the smallest possible latency becomes necessary.

In the domino-logic design-style that we use, the circuits perform fastest if transistors are shared in the pulldown paths; in the *pc* incrementer this sharing is necessary if we want to get acceptable performance. The sharing leads to large internal parasitic capacitances and hence to difficulties with charge sharing. An example of the bad effects of charge sharing is seen in Figure 8.15. The figure shows one of the output-logic nodes of the more-significant incrementer-domino as the carry-in

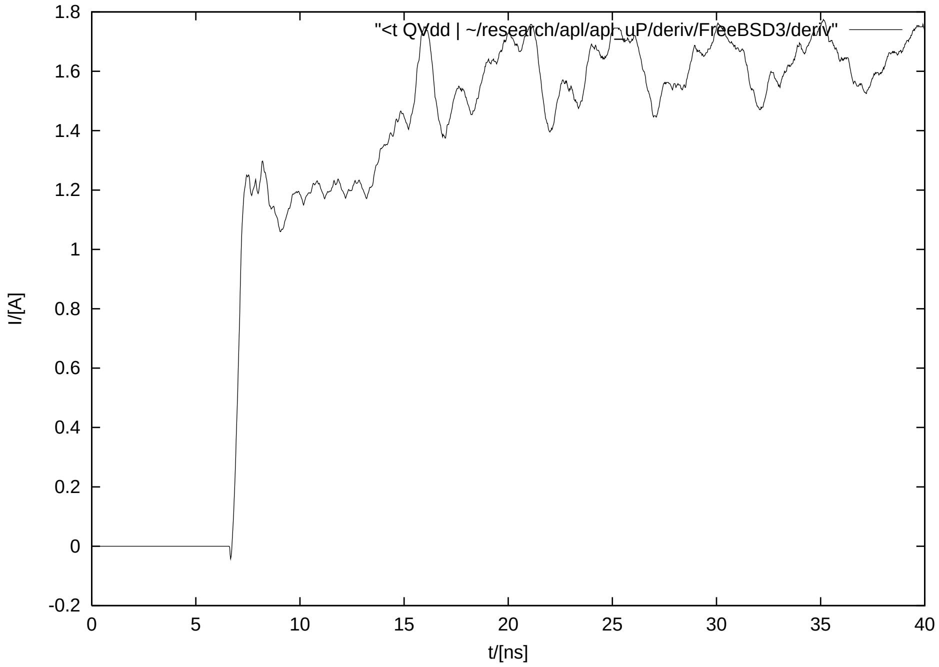

Figure 8.13: Current draw of *PCUNIT* in amperes; constant branching after $t = 12$ ns. `Go` active at $t = 6.5$ ns.

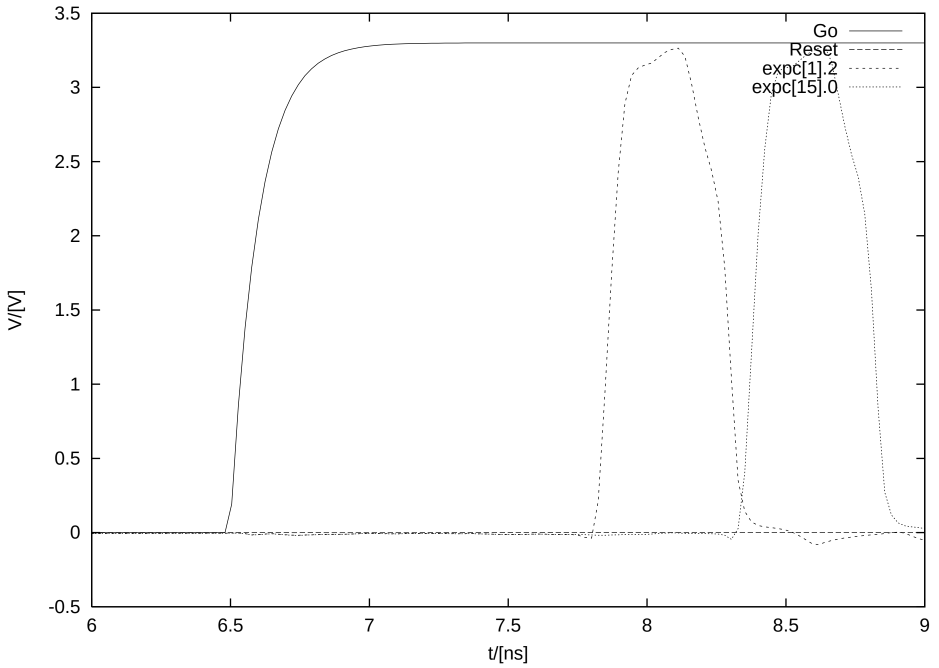

Figure 8.14: Arrival of least and most significant 1-of-4 codes of pc.

changes (the output goes from zero to one); because the incrementer computed a result of zero on the previous cycle, the internal nodes are charged up, and hence the figure shows almost the worst-case charge-sharing possible in this circuit.

The circuit diagram of the pull-down logic is shown in Figure 8.16; the node that is the source of our charge-sharing troubles is marked "X" in the figure. The p-transistors to *Vdd* and the parallel resistors (implemented by weak p-transistors to *Vdd* with their gates grounded) are used for reducing the effects of charge sharing by charging the internal nodes away from *GND* when the circuit resets. By sizing them larger, we can reduce or eliminate the charge-sharing problem, at the cost of a slower, more power-hungry circuit.

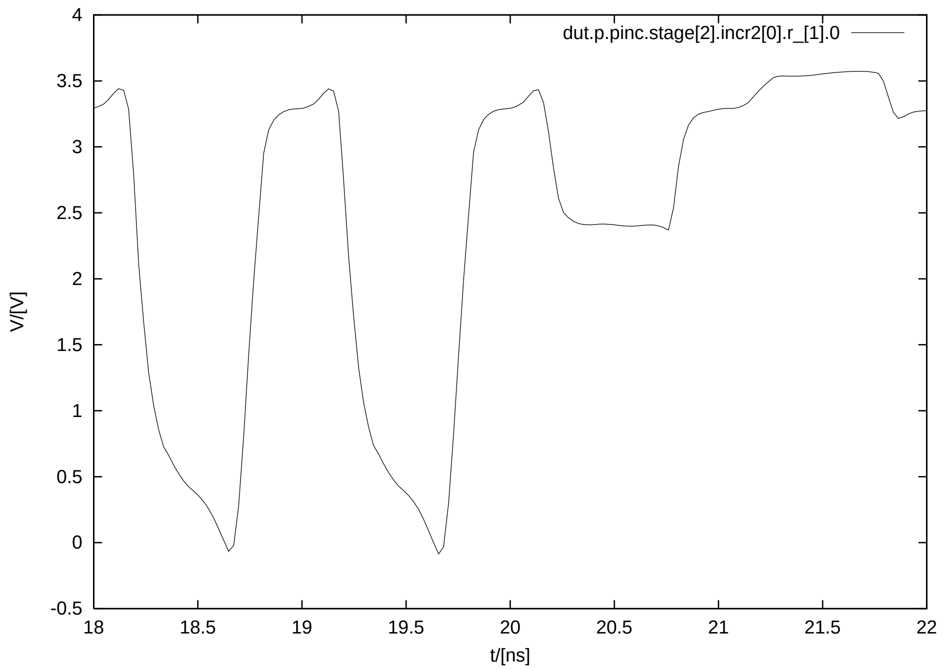

Figure 8.15: Charge sharing in the *pc* incrementer.

The *PCUNIT* was implemented mainly with PL1 processes. Those things that were not designed as PL1 processes either already existed (they were simple, hand-compiled processes like the copy processes and merge processes required in any STAPL design of moderate complexity) or they were hand-designed for flexibility (e.g., the 2×1-of-4 code incrementer cell was parameterized to allow easy experimenting with different arrangements; the result of compiling a PL1 program implementing the finally chosen design would have been similar if not identical to the hand-designed circuit with the finally chosen parameters).

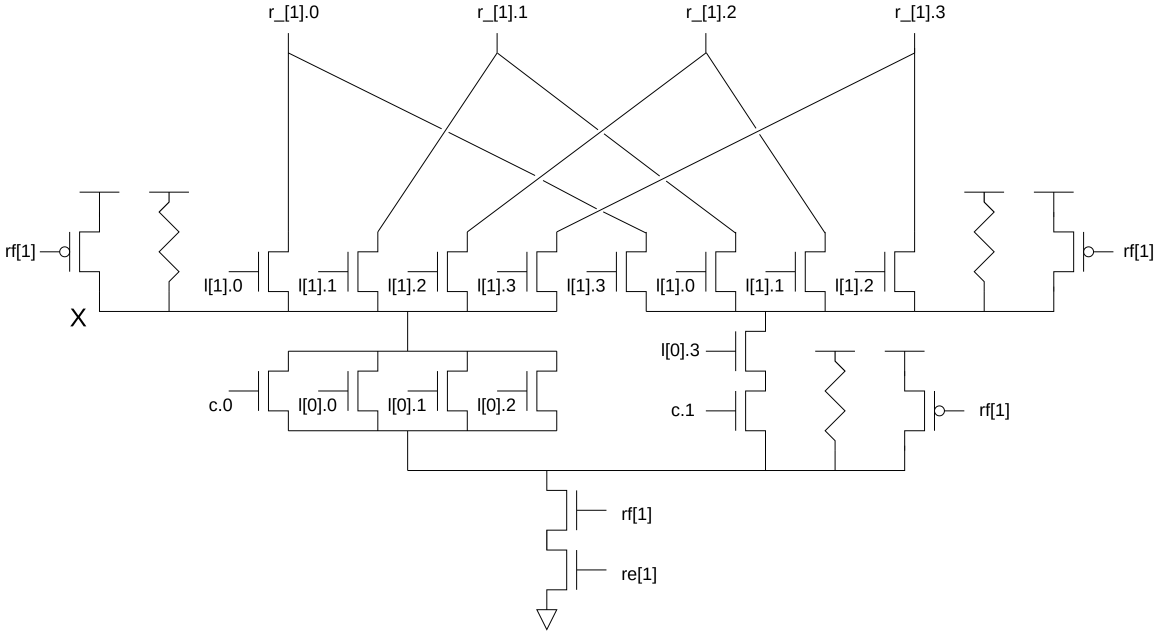

Figure 8.16: Circuit diagram of compute logic for the upper 1-of-4 code in *pc*-incrementer.

8.3.2 The *REGFILE*

In the *PCUNIT* design that we have just seen, we were able to decompose the large-scale unit into a collection of PL1 processes, i.e., into a collection of processes whose implementation exactly followed the rules of Chapters 5–7. This already suggests that the STAPL technique and the PL1 language let us build digital logic systems. One cause for concern is that the transistor count for STAPL circuits is fairly high. While technology changes are making this more and more acceptable in logic circuits, high transistor-counts are still cause for concern in array circuits, e.g., SRAMs and DRAMs.

The SPAM processor implementation has three array structures: an instruction memory, a data memory, and a register file. As a first step in applying APL techniques to the design of such circuits, the register file is a ten-transition-per-cycle APL circuit, using single-track handshaking.[8] The design that we shall see here uses an additional timing assumption for the purpose of reducing the transistor count; it implements an 8×8-bit register array in a single process, which would not be possible if we were to strictly follow the rules of Chapter 6. It also uses a higher-level design-trick inherited from the MiniMIPS design for the purpose of increasing the slack: a special type of buffer is used for distributing the register control to the register file in such a way that conflicting register accesses (i.e., reads and writes, or writes and writes, referring to the same register) are properly

[8]The instruction memory and data memory are simplified versions of the register file: the instruction memory has one read port and no write port; the data memory has one read port and one write port. The register file itself of course has two read ports and one write port.

sequenced, but other accesses can be performed out of order.

8.3.2.1 *REGFILE* specification

The SPAM *REGFILE* has 8 registers numbered 0–7, of which register 0 is always zero (it may be written, but such writes will be ignored); it has two read ports, x and y, and one write port, z.

Because of the SPAM architecture's orthogonal instruction set, there is nothing very mysterious about the *REGFILE*: it is simply consulted for the execution of every instruction. Hence, its CHP specification is

$$
\begin{array}{l}
\mathit{REGFILE} \equiv \\
\mathit{gpr}\,[0] := 0; \\
*[\ \ I?i; \\
\quad X!\mathit{gpr}\,[i.rx]\,,\,Y!\mathit{gpr}\,[i.ry]\,; \\
\quad [\ \ i.rz \ = \ 0 \ \longrightarrow \ Z?\_ \\
\quad [\!]\ i.rz \ \neq \ 0 \ \longrightarrow \ Z?\mathit{gpr}\,[i.rz] \\
\quad] \\
\ ]\ .
\end{array}
$$

We should like to implement the *REGFILE* in a way that allows the reading and writing of registers in the core to be performed concurrently; the register core will then be specified as:

$$
\begin{array}{l}
\mathit{REGCORE} \equiv \\
\mathit{gpr}\,[0] := 0; \\
*[\ \ I?i; \ \ X!\mathit{gpr}\,[i.rx]\,,\,Y!\mathit{gpr}\,[i.ry]\,,\,Z?\mathit{gpr}\,[i.rz]\ \ ]
\end{array}
$$

If we can implement the *REGCORE* thus, we shall be able to use simpler circuit realizations of the register bits than the general state-bit described in Section 6.4.3.1 (the general state-bit can be read and written at the same time, whence it is necessary to copy the value between iterations so that the reading does not result in the new value or confusion).

The main thing that raises concern here is that a register may be read and written on the same iteration of *REGFILE*, but this is not true of the *REGCORE* program. A register-bypass mechanism solves this problem: we copy the input value three ways, delay the write to the register file by one iteration, and if the same register is read on the iteration following the one it was written on, the value is read from the bypass unit rather than from the register core. The bypass mechanism also reduces the read latency for reads of registers that have lately been written. The mechanism is essentially identical to the one used in the MiniMIPS.

```
define regfile()(1of(2)[3] rx, ry, rz; 1of(4)[16] x, y; 1of(4)[16] z0, z1, z2;
                 1of(2) reg_wb)
{
  1of(4)[16] corex, corey, corez0;
  1of(8) cx, cy, cz; 1of(2) bx, by, bxs, bys;

  regctrl() rct(rx,ry,rz, reg_wb,   cx, cy, cz,   bx, by);
  slack(2,1,3) s_bx({bx},{bxs}), s_by({by},{bys});
  bypass() b(bxs, bys, x, y, z0, z1, z2, corex, corey, corez0);
  reg_core(true) rco({ cx,cy }, cz, , corez0);

  rco.r[0..15,0] = corex[0..15]; rco.r[0..15,1] = corey[0..15];
}
```

Figure 8.17: Top-level CAST decomposition of SPAM *REGFILE*.

8.3.2.2 *REGFILE* decomposition

The *REGFILE* is decomposed into three main pieces: the bypass unit, the register core, and the register control; the decomposition is shown in Figures 8.17 and 8.18. The register control and bypass are further decomposed into sets of PL1 processes, which are then compiled into STAPL circuits. The register core is a hand-compiled circuit that obeys the STAPL timing constraints.

Note that we have split the input channel Z into three: $Z0$, $Z1$, and $Z2$. It turned out that the unit merging the results from the different execution units (arithmetic, function block, shifter, and so on) was a simple one and could easily take on additional functions. By combining the copying of Z that would normally have to occur in *REGFILE* with the merging function, we are able to remove one stage of pipelining from the execution loop, at the cost of this minor cluttering of the *REGFILE* interface.

8.3.2.3 Register-core cell

The register-core cell holds eight bits in eight registers (64 state bits) in a single process. The read and write ports may be treated as separate units; this is possible because the register control issues only non-conflicting reads and writes to the core (recall that this was the purpose of introducing the bypass).

The (two–read-port, one–write-port) core cell consists of five distinct parts: two read-port subcells, one write-port subcell, one dummy-write subcell, and the state bits themselves. A block diagram is shown in Figure 8.19.

The state bits are organized in pairs; this allows generating the read outputs directly as 1-of-4 codes in the core. The circuitry used for each pair of state bits is shown in Figure 8.20. The arrangement of the state bits, the word (i.e., byte) lines, bit lines, pulse generators, etc. is shown

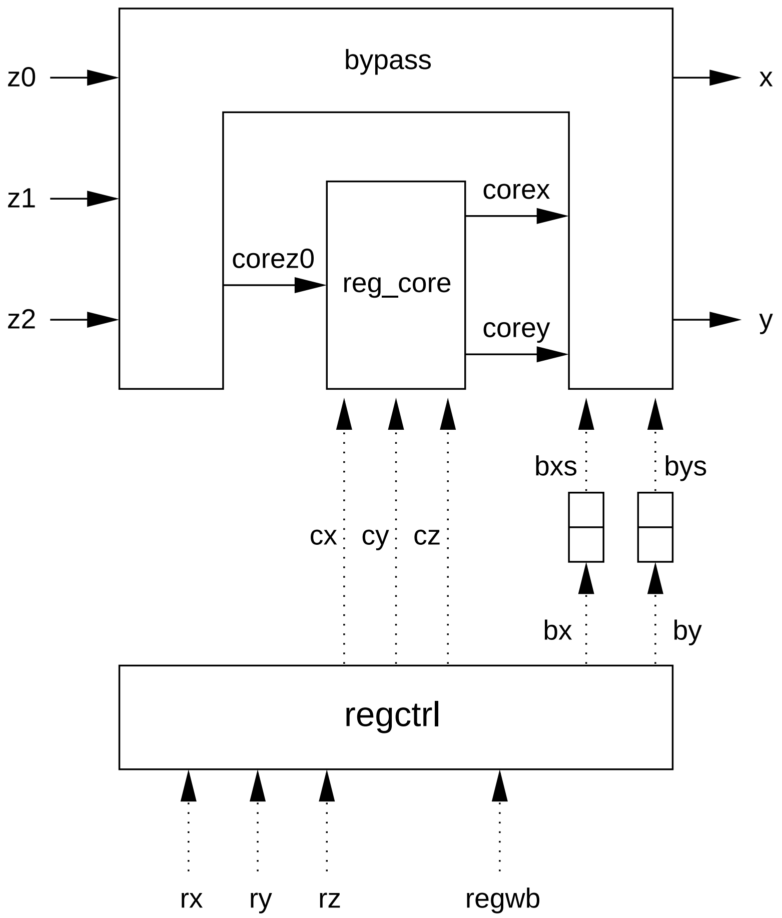

Figure 8.18: Process graph of *REGFILE*. Data channels are drawn solid; control channels dotted.

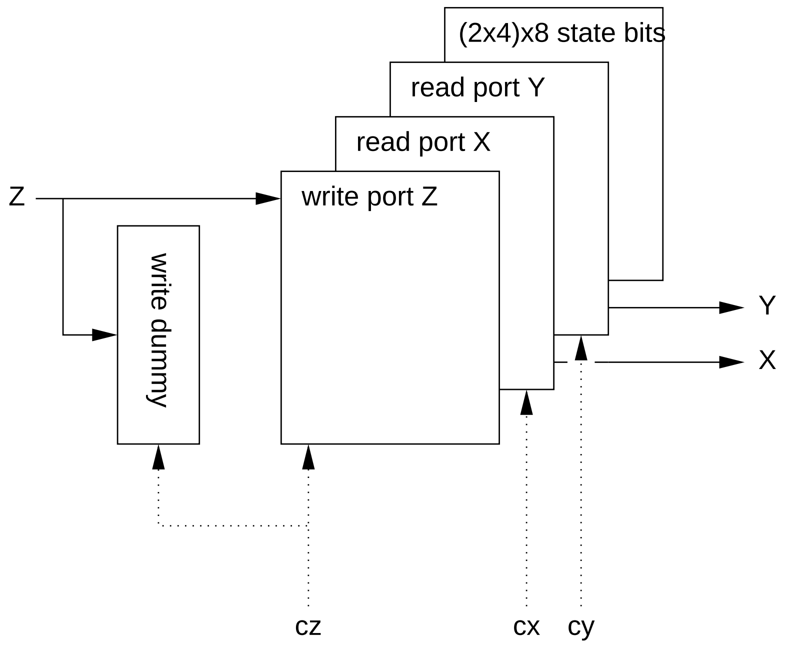

Figure 8.19: Block diagram of 8×8 register-core cell; input and output channels are each four 1-of-4 codes.

in Figure 8.21.[9] As usual, the control wires have been drawn dotted. There are in reality four data wires for each of X, Y, and Z.

Simulations show that this register file operates at about the same speed as the logic circuitry we have seen before, i.e., about 1 GHz in 0.6-μm CMOS without wiring, according to our simulations; this indicates that the speed for fabricated parts would be 650–700 MHz.

8.4 Performance measurements on the SPAM implementation

The design of the SPAM processor is complete to the PRS level. Using the assembler mentioned in Appendix B, we can assemble programs for it and run them on the simulator. We shall study the results of running two small programs on the SPAM: first, a simple straightline program that tests the maximum speed of the SPAM processor; and secondly, the program shown in Figure 8.22, which computes the nth Fibonacci number. The results were obtained by the author's simulating

[9] José Tierno has kindly pointed out to the author that this register file could easily be extended to 32 registers while maintaining almost the same performance by making four of the 8×8 bit cores we have here but then ganging them together simply by putting the output p-transistors in parallel (some modifications to the control circuitry would also be needed). This would allow a MIPS- or Alpha-style 32-entry register file in a single pipeline stage, operating at ten transitions per cycle.

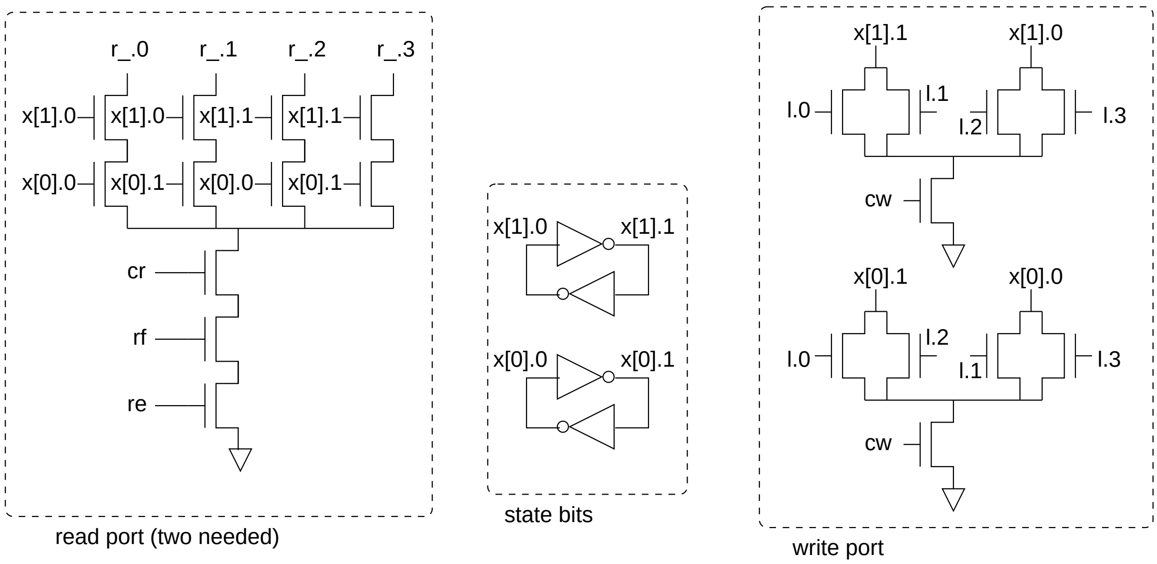

Figure 8.20: Circuitry associated with each pair of state bits in register core. Dummy-write circuitry not shown.

the SPAM processor design at the PRS level on a 1 GHz Pentium III Xeon computer with 512 MB of RAM, using the `csim` production-rule simulator; the simulation runs at several instructions per second.[10]

8.4.1 Straightline program

The first program consists of the assembly instruction `and r0=r0,r0` repeated enough times to fill the memory (our implementation has 512 bytes of instruction memory; i.e., the instruction is repeated 128 times). Because logical operations can proceed at ten transitions per cycle, and because the *PCUNIT* can fetch at that speed, we should expect the processor to be able to execute this trivial program at that speed. Running the simulation shows that this is not so. If we average over 260 instructions, the SPAM processor runs at $12\,^2\!/_3$ transitions per cycle.

The reason that the SPAM processor cannot completely manage its intended ten transitions per cycle is to be found in the register file–execution unit loop. In an effort to keep the SPAM simple, the implementation was designed with only a single writeback bus (Z); this causes slack-matching problems because the execution unit–bypass–execution unit loop is too long: to manage full throughput, we should have to have at most five pipeline stages in this loop; we have seven. This problem was avoided in the MiniMIPS partly by splitting the writeback bus Z into two separate busses that are used alternately; this technique could be used in the SPAM.[11] The fact remains,

[10] The `csim` simulator was written by Matthew Hanna and Eitan Grinspun as part of the MiniMIPS project.

[11] The other reason that this problem was less troublesome in the MiniMIPS was that many circuits used in the

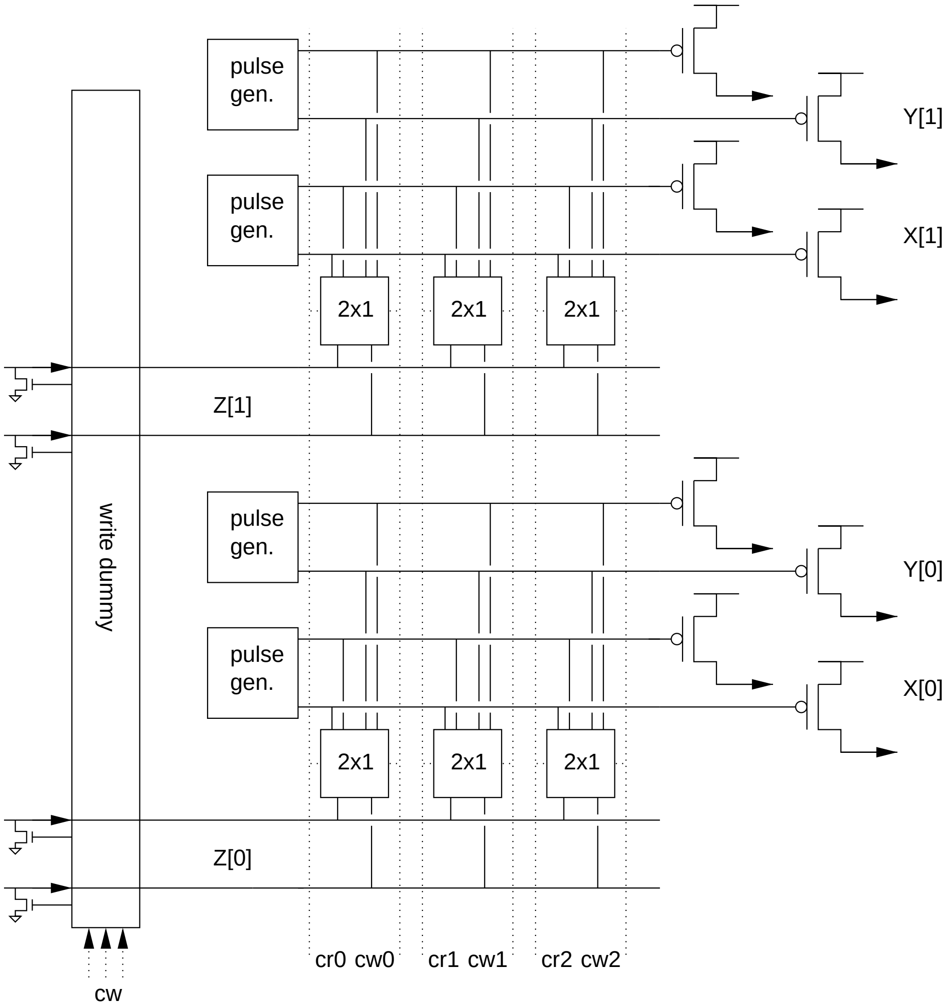

Figure 8.21: Overall arrangement of register-core cell. A two 1-of-4–code tall, three-register wide chunk is shown.

```
;;; Compute the n+1th Fibonacci number
.=0x8
        li      r1=15           ; n
        li      r2=0            ; fib(0)
        li      r3=1            ; fib(1)
Start:  beq     r1, End
        add     r4=r3,r2        ; fib(f+1) := fib(f) + fib(f-1)
        or      r2=r3,0
        or      r3=r4,0
        sub     r1=r1,1
        jmp     r0=Start
End:    hlt                     ; halt the processor--result is in r2
```

Figure 8.22: SPAM program for computing Fibonacci numbers.

however, that back-to-back data-dependent instructions in the SPAM simply cannot execute at full speed, because the latency through our units is slightly too long, notwithstanding our efforts to keep the forward latency short.

8.4.2 Computing Fibonacci numbers

The second program we try is shown in Figure 8.22. It computes the sixteenth Fibonacci number using the simple iterative algorithm.

The Fibonacci-number program needs to execute $5 + 6n$ instructions; 95 for $n = 15$. Simulating it, we see that it takes 4048 transitions to execute the program (to the point where zero is written to `r1`—92 instructions after reset). In this time, we should have been able to execute roughly 400 instructions (or at least 300, taking into account or $12\,^2\!/_3$ transitions' minimum cycle time). We should expect that the arbitrated-branch mechanism steals a few cycles for every branch, but at worst that should take us up to 150 instructions or so. Obviously something else is stealing the time. What is the problem here?

If we examine an execution trace using `csim`'s `critical` feature, which deduces what transitions are on the critical path, we see that the culprit is the instruction `sub r1=r1,1`. The pertinent part of the trace is shown in Figure 8.23.

Examining these production rules, we see that the critical path is the carry chain of the datapath adder. But was not the point of using a simple ripple-carry adder that we could demonstrate the superior average-case performance of asynchronous design?

It turns out that asynchronous intuition fails us here. It is true that on average, for *random data,* the longest sequence of carries in a ripple-carry adder goes only as $\log n$, where n is the width of

MiniMIPS were in fact faster than the $18\,^2\!/_3$ cycles per transition that the processor as a whole managed. Since we are with the SPAM design aiming at the entire processor's executing at the maximum speed of the individual blocks, this option is not open to us. However, "binary tree FIFOs" [18] and similar structures have the effect of absorbing slack mismatches even in STAPL systems operating at ten transitions per cycle overall.

```
~dut.e.a.a.f[6].d_.1 -> dut.e.a.a.d[7].1+ at 405100
dut.e.a.a.f[6].fd & dut.e.a.a.f[6].de & dut.e.a.a.b[6].1 & dut.e.a.a.d[6].1 -> dut.e.a.a.f[6].d_.1- at 405000
~dut.e.a.a.f[5].d_.1 -> dut.e.a.a.d[6].1+ at 404900
dut.e.a.a.f[5].fd & dut.e.a.a.f[5].de & dut.e.a.a.b[5].1 & dut.e.a.a.d[5].1 -> dut.e.a.a.f[5].d_.1- at 404800
~dut.e.a.a.f[4].d_.1 -> dut.e.a.a.d[5].1+ at 404700
dut.e.a.a.f[4].fd & dut.e.a.a.f[4].de & dut.e.a.a.b[4].1 & dut.e.a.a.d[4].1 -> dut.e.a.a.f[4].d_.1- at 404600
~dut.e.a.a.f[3].d_.1 -> dut.e.a.a.d[4].1+ at 404500
dut.e.a.a.f[3].fd & dut.e.a.a.f[3].de & dut.e.a.a.b[3].1 & dut.e.a.a.d[3].1 -> dut.e.a.a.f[3].d_.1- at 404400
~dut.e.a.a.f[2].d_.1 -> dut.e.a.a.d[3].1+ at 404300
dut.e.a.a.f[2].fd & dut.e.a.a.f[2].de & dut.e.a.a.b[2].1 & dut.e.a.a.d[2].1 -> dut.e.a.a.f[2].d_.1- at 404200
~dut.e.a.a.f[1].d_.1 -> dut.e.a.a.d[2].1+ at 404100
dut.e.a.a.f[1].fd & dut.e.a.a.f[1].de & dut.e.a.a.b[1].1 & dut.e.a.a.d[1].1 -> dut.e.a.a.f[1].d_.1- at 404000
~dut.e.a.a.f[0].d_.1 -> dut.e.a.a.d[1].1+ at 403900
dut.e.a.a.f[0].fd & dut.e.a.a.f[0].de & dut.e.a.a.a[0].1 & dut.e.a.carry_in.1 -> dut.e.a.a.f[0].d_.1- at 403800
```

Figure 8.23: Part of the critical-path transition-trace of running the program of Figure 8.22. Time goes upwards; each transition delay is counted as 100 time units.

the adder in bits; it is furthermore true that most numbers that are added by the average program may have better behavior even than that because they are more commonly small than large. Hence, for adding or subtracting random numbers to and from each other or for adding small numbers to each other, an asynchronous ripple-carry adder is a good and far simpler alternative to a carry-lookahead or carry-select adder. But what we are doing in the Fibonacci program is *subtracting* small integers from each other. In this case, the ripple-carry adder achieves its worst-case performance; and consistently so.

The way we can improve the performance of the Fibonacci program is familiar to every hacker: we unroll the loop. Thus we remove a large fraction of the subtract instructions (and incidentally also of the branches, which themselves are a bit costly). The improved program is shown in Figure 8.24. Executing it takes 2324 transitions, a speedup of 1.7. But the critical path still mainly goes through the adder.

8.4.3 Energy measurements

The author has run the unrolled Fibonacci program through the `esim` production-rule and energy simulator.[12] Using a fanout-weighted transition-counting model that has been calibrated on the MiniMIPS, running the unrolled program took 21.5×10^6 `esim` energy units. Pénzes's observations [70] of the MiniMIPS suggest that each energy unit corresponds to about 160 fJ; the author calibrated the model against the *PCUNIT*, which took 4.82×10^6 of the energy units to fetch 126 instructions, suggesting that the number is a bit smaller for the SPAM (about 105 fJ), if we assume that the energy cost of adding all the wiring will approximately cancel the benefit of more careful sizing. (Recall that the *PCUNIT* uses about 4 nJ per operation; see Section 8.3.1.5.) This is not surprising since the MiniMIPS does not have the STAPL feedback-path transitions; also `esim` overestimates the energy dissipation for the SPAM because the gate-sharing information was not included in the measurements. The unrolled Fibonacci program executes 60 instructions; in the simulation it fetches

[12]The author is indebted to Paul Pénzes for his providing the `esim` simulator and helping to run it.

```
;;; Compute the n+1th Fibonacci number, unrolled
.=0x8
        li      r1=15           ; n
        li      r2=0            ; fib(0)
        li      r3=1            ; fib(1)
        and     r4=r1,1
        beq     r4, StartEven
StartOdd:
        add     r4=r3,r2        ; fib(f+1) := fib(f) + fib(f-1)
        or      r2=r3,0
        or      r3=r4,0
        sub     r1=r1,1
StartEven:
        beq     r1, End
        add     r4=r3,r2        ; fib(f+1) := fib(f) + fib(f-1)
        or      r2=r4,0
        add     r5=r4,r3        ; fib(f+2) := fib(f+1) + fib(f)
        or      r3=r5,0
        sub     r1=r1,2
        jmp     r0=StartEven
End:    hlt                     ; halt the processor--result is in r3
```

Figure 8.24: SPAM program for computing Fibonacci numbers, unrolled once.

126. We can hence estimate that the energy per operation is 37 nJ per effective instruction or 18 nJ per fetched instruction. The MiniMIPS consumes about 34 nJ per arithmetic instruction and about 21 nJ for a no-operation instruction [70].

8.4.4 Summary of SPAM implementation's performance

The SPAM implementation performs reasonably well. In fact, for programs that do not use the ripple-carry adder in the worst-case way of the Fibonacci program (unfortunately, all programs with short for-loops use the adder thus), the performance is very good: $12^2/_3$ transitions per cycle would correspond to an average fetching rate of over 500 MHz in the now-obsolete 0.6-μm CMOS technology. (This is an honest figure for a hypothetical fabricated chip.) At the same time, we have identified a few serious bottlenecks: first, the loop restricting the speed to $12^2/_3$ transitions per cycle could be removed by redesigning the writeback mechanism along the lines of the MiniMIPS; secondly, changing the datapath adder from a ripple-carry adder to a traditional carry-lookahead adder would bring us much closer to the goal of executing real programs at ten transitions per cycle.[13] We should finally remember that the bottlenecks we have identified are not due to the STAPL realization but rather due to a microarchitecture that is a bit too simple to be able to achieve the performance

[13] Note that we should not redesign the adder without first fixing the writeback bottleneck; the author has tried replacing the ripple-carry adder with a simple carry-lookahead adder (with full carry-lookahead across eight bits), and this resulted in the Fibonacci program's running slightly slower because of the added latency of the carry-lookahead adder.

target.

8.4.5 Comparison with QDI

We have now seen the design of a large, concurrent system using the STAPL circuit family. The speed and energy advantages of QDI design have been established before [55]; can we compare the two design styles?

The only way of truly fairly comparing STAPL and QDI would be taking a single specification, e.g., the SPAM architecture, and implementing it as well as possible using each of the design styles. We have not done this, and obviously it would be a lot of work to do so; most likely no way of doing it would even convince the skeptics.

Nevertheless, we should not shirk our duty of comparing STAPL with previously known techniques. There are four chief dimensions of interest: ease of design, speed, energy consumption, and reliability; reliability may include tolerance to design errors and noise, and the ability of operating over a wide range of environmental conditions.

8.4.5.1 Ease of design

Are STAPL circuits easier to design than their QDI counterparts? The PL1 language shows that it is easy to design STAPL circuits, so easy a computer can do it well. But given the similarities between STAPL and QDI circuits, it would be easy to write the same software tools for QDI circuits (indeed the work is already in progress). And QDI circuits are easier to generalize: there is a compilation technique that will take us from CHP all the way to PRS. We must also remember that STAPL circuits are more sensitive to sizing; it is not clear how important this is for the designer, since QDI sizing must also be verified before fabrication.

8.4.5.2 Speed

Do STAPL circuits run faster than QDI circuits? Undoubtedly. The SPAM example shows that something as large as a microprocessor can be designed with circuits that all run at ten transitions per cycle, whereas it would be very difficult to do so in less than 18 with only QDI circuits. The reason for the difference is that STAPL circuits remove many waits that are necessary for maintaining QDI protocols and replace them with timing assumptions. Furthermore, STAPL circuits load their inputs less than do QDI circuits, because they generally do not need the completion circuitry that is needed in QDI circuits. The SPAM processor parts that we have simulated run three times as fast as similar parts from the MiniMIPS.

It should be noted that STAPL circuits do not offer a magic bullet for dealing with latency issues. The latency through a STAPL stage is somewhat smaller than through the same stage of

QDI computation, because of the lesser loading of the inputs; but the difference is minor. Some might say that STAPL circuits make the job harder for the microarchitect, much as the faster improvement in processor speed compared with the improvement in DRAM access-time has made his job harder.

8.4.5.3 Energy

How do STAPL and QDI circuits compare on energy? To first order, there is no reason to believe that they should use very different amounts of energy. The reason is that STAPL circuits have most of the paths that are present in QDI circuits: the logic is the same, much of the output completion is the same. There is no input completion, nor are there acknowledge wires, but on the other hand, the QDI circuits do not have pulse generators. Estimating the energy lost to interference is difficult, but so is estimating the energy lost to "shoot-through" current in the combinational completion-networks in QDI circuits.

There is a little more to this story, however. A circuit carrying out the STAPL handshake uses the same wires for receiving data as it does for acknowledging it; hence in the limit of single one-hot codes, there are only half as many transitions on the interface nodes of STAPL circuits as there are for QDI circuits. But in QDI circuits, one can combine the acknowledges by synchronizing at a slightly larger grain-size: e.g., in the MiniMIPS, most "elementary" processes (meaning those that communicate with their environment entirely on channels) are eight-bit processes, thus amortizing the cost of the acknowledge across all eight bits. But in STAPL circuits, little would be gained by this. Hence the STAPL circuits invite deeper pipelining.

In short, this means that while STAPL and QDI circuits can be built that look nearly identical, that may be an artificial thing to do. And hence we cannot say that, given a high-level specification, its well-designed STAPL and QDI implementations will dissipate the same amount of energy per operation. We cannot, for instance, say that STAPL circuits will run three times faster *and* use the same amount of energy as QDI circuits: the higher degree of pipelining in STAPL circuits will use a little more energy than that.

Let us evaluate STAPL and QDI circuits using the Et^2 metric; this metric captures the fact that by our varying the supply voltage of a CMOS circuit, any speed improvement can be traded for roughly twice that improvement in energy. The 1:2 tradeoff is reasonably accurate for a wide range of operating voltages. The Et^2 metric was introduced in the context of the MiniMIPS by Martin *et al.* [55]; it is more fully explored by Martin [57]. STAPL circuits are about three times faster for the circuits we have studied; the transistor count is about twice as high, and there is an extra handshake for each 1-of-4 code that is not present for the QDI implementations that we compared with. Hence, conservatively estimating ($E \mapsto 2E$, $t \mapsto t/3$) the improvement in Et^2 gives that STAPL circuits improve by a factor of about five; to first order, the change in At^2 (a metric introduced by Thompson [82]) would be about the same. How this comparison would turn out if we compared PL1-described STAPL with similarly generated QDI circuits is less clear, because some part of the higher transistor-count of the STAPL circuits is due to our using higher-level design tools rather than to the circuit family itself. (The MiniMIPS processor, whence come the QDI circuits we are comparing with, was designed entirely by hand.)

8.4.5.4 Reliability

How reliable are STAPL circuits? Since we have not yet seen a fabricated chip, this is not the easiest question to answer. The timing assumptions in STAPL circuits are definitely cause for concern. In simulations, the author has noticed that STAPL circuits, when their supply voltage is lowered away from the nominal, appear to stop working sooner (at a higher voltage: $\approx 1 \times V_T$) than do QDI circuits (the author has in the lab successfully operated the 1989 QDI, 1.6-μm microprocessor [53] with $Vdd \approx V_T/3$). These failures are due to the various circuit delays' not changing at the same rate as the supply voltage is changed; the STAPL circuits could be engineered to be more tolerant to supply-voltage changes by making sure that σ decreases and ξ increases as the supply voltage changes instead of the other way around. Recall that the single-track handshake involves four timing constraints; these are captured by $\sigma_{\mathbf{true}}$, $\sigma_{\mathbf{false}}$, $\xi_{\mathbf{true}}$, and $\xi_{\mathbf{false}}$ (see Figure 5.7).

As for injected noise, it does not seem that either STAPL circuits or QDI circuits are particularly trustworthy. Both design styles make much use of dynamic logic; both design styles achieve high performance when gate sharing is used, which leads to charge sharing and thence to problems with the dynamic nodes. The STAPL circuits do use more dynamic stages than do the QDI circuits, but on the other hand, charge-sharing–avoiding circuits are easier to design in STAPL circuits.

STAPL circuits are less tolerant to design errors than are QDI circuits. In a STAPL circuit, a single mis-sized transistor can easily lead to complete system failure, whereas in QDI circuits, the same thing can usually only happen under extreme circumstances (a badly mis-sized transistor on a three-transition feedback-loop, for instance). This, however, is something that we can understand theoretically, and we saw in Section 6.8.4 how we might deal with the issue.

Chapter 9

Related Work

Quidquid latine dictum sit, altum viditur.

— *Anonymous*

This thesis is about asynchronous pulse logic: in essence, it argues that APL circuits are possible; that they will compute reliably; and that their performance is better than that of QDI circuits, for about the same degree of design difficulty. We have had to cover a wide range of subjects to make the argument stick.

9.1 Theory

In Chapter 3, we developed a theory for the functioning of APL circuits. No comprehensive theory exists that manages establishing the connection from device physics to digital asynchronous logic. There have been small steps towards one for QDI circuits. Martin [52] suggested that as long as signals ramp monotonically (and quickly), the circuits can be proved to be correct realizations of the specifications. Later, van Berkel discovered that not arranging the signals to behave well can cause problems [7]. Greenstreet has come further than others in making the connection complete; his work relies on dynamical-systems theory (which to some extent ours does too) [31]. Our work differs from these authors' in that we have chosen the pulse as the basic abstraction, whereas Martin and van Berkel considered the transition (which is natural since they were dealing with QDI systems) and Greenstreet used a more complicated (and more powerful) dynamical-systems model.

9.2 STAPL circuit family

We next, in Chapters 5 and 6, developed a family of practical circuits for implementing any digital specification. These circuits are similar to the "asP" circuits studied by Greenstreet, Molnar, Sutherland, Ebergen, and others [32, 61, 80]. The work at Sun (Sutherland, Ebergen, and others) seems aimed mainly at very fast (six transitions per cycle) FIFO controls intended for driving a bundled-data datapath. Our work differs considerably from this in that we design the entire system, control as well as datapath, using the STAPL model (see section below on PL1 language). These other authors also have not studied circuits as complicated or powerful as ours; it seems unlikely that the circuits that we have studied should be possible with only six transitions per cycle. The "IPCMOS" circuits studied by Schuster *et al.* at IBM Yorktown Heights [74] are in essence the same as the Sun circuits, although the low-level realizations differ slightly.

Single-track handshakes have been studied before [8, 83]. The chief difference between the earlier work and the STAPL family is that the STAPL family is aimed at implementing in a single template all the functions that we should care to see in a single process; earlier single-track circuits have generally started from a QDI specification and gradually worked in the timing assumptions. We have instead started with the idea of a pulse with well-known characteristics, and then we built up the entire family around that pulse.

Earlier work with "post-charge logic," whose circuit realizations are similar to the pulsed asynchronous, was done in synchronous contexts by Proebsting [71], Simon [77, 17], and is today an active field of research and development. The work in this thesis was in part inspired by the work on the AMD K6 processor by Draper *et al.* [22].

The STAPL circuits are also in some respects similar to the "neuromorphic" circuits pioneered by Mead and his group [58]; the "silicon neuron" of Mahowald and Douglas [44] is the closest to compare with. The silicon neuron integrates analog inputs and then generates a carefully timed spike, which resets its inputs and which can be integrated by other neurons; in basic principle it is similar to the STAPL template. The details, however, differ markedly: the silicon neuron is built using transistors operated in (slow) analog configurations (especially if they are operated in the subthreshold regime; this would be done to make the modeling easier), it uses inputs that are encoded differently (i.e., as analog levels), and it is much slower (the speed difference is as much as six decades). To some extent, the slowness is intentional on the part of Mahowald and Douglas; their claim is that since the silicon neuron is intended for building "machines that interact with real-world events in the same way as biological nervous systems," this is the right thing to do. Nevertheless, the author cannot deny that the present work has been strongly influenced by the idea that forms the basis of the silicon neuron, viz. the idea of waiting for inputs to arrive and then, when appropriate, generating a single output pulse of well-known characteristics; as we saw earlier, the careful internal-pulse–timing is one of the main differences between the STAPL family and earlier single-track–handshake asynchronous circuits.

9.3 PL1 language

The PL1 language is used for specifying the behavior of small asynchronous processes that are then composed (using the CAST language) into a larger system. The PL1 language describes processes that are similar to the computational elements described in the 1970s and 1980s by Dennis and other authors in the dataflow computer field [19]. Necessary conditions for deterministic behavior of these systems were implicit in much of their work; the ideas of slack-elasticity were later elucidated by Manohar [45], who proved the properties of slack-elasticity for deterministic and certain nondeterministic systems that our work and the MiniMIPS depend on. Slack-elasticity in deterministic systems was to the author's knowledge first used in the Caltech Asynchronous Filter [18]; how to compile the processes needed for building such systems was described by Lines [43], but some of the ideas are already present in Burns's [14] and Williams's [85] work.

The asynchronous slack-elastic method of designing hardware itself was explored in the framework of QDI design, first proposed by Martin [51] as the most conservative, yet realizable compromise between speed-independent and delay-insensitive design methods. (We mentioned some of the background to the earliest work in the Introduction.) QDI design has really been the main inspiration for this work: this was the way the first [53] and largest and fastest [55] (i.e., the MiniMIPS) asynchronous microprocessors were designed. The byte skewing we described for the control distribution of the SPAM is a hybrid of the bit skewing used in the Asynchronous Filter [18] and the pipelined completion used in the MiniMIPS [55].

Our pulsed circuits were initially inspired by the problems of compiling from HSE to PRS via ER systems; CHP, HSE, and PRS were described by Martin [48, 54]; ER systems are due to Burns [14] and were extended by Lee [42]. Taking a different approach to the problem, Myers and Meng [64] and Takamura and others [81] have described methods for introducing timing assumptions into a QDI system. However, these authors essentially start from QDI systems and attempt to improve their performance by removing unnecessary transistors, whereas the method described in this thesis leads to quite different circuits, since it does not make the QDI detour.

The ideas of modular, delay-insensitive asynchronous design owe much to basic work in concurrency. Hoare's CSP language [36], itself related to Dijkstra's guarded-command language [21], is the basis of our CHP. Chandy and Misra's UNITY language [16] may be thought of as a more powerful (and hence not directly implementable) version of production-rule sets; UNITY is also an application of the guarded-command language. There is a definite scale of semantics: Dijkstra's guarded-command language allows arbitrary sequential statements to be executed; UNITY allows arbitrary atomic assignments (but no sequencing); and the production-rule language allows only single, boolean assignments. In a sense, the STAPL processes themselves are higher-level "production rules" with more powerful semantics than in the usual PRS model; in this sense the STAPL model is most similar to UNITY.

9.4 SPAM microprocessor

The SPAM microprocessor is itself not revolutionary. It is essentially a simplified MIPS processor; this allows us to take maximum advantage of the experiences of the MiniMIPS project. The register-locking mechanism is the same as that used in the MiniMIPS, and the arbitrated-branch mechanism was inspired by the MiniMIPS exception mechanism; it is to a lesser degree similar to the arbitrated branch and exception system of the AMULET1 processor [88].

Chapter 10

Lessons Learned

If a man will begin with certainties, he shall end in doubts; but if he will be content to begin with doubts, he shall end in certainties.

— *Francis Bacon,* The Advancement of Learning *(1605)*

10.1 Future Work

Things remain to be done. The SPAM processor demonstration is yet unfinished, and it would be more convincing with working silicon. This is the most immediate task.

Furthermore, there remain several unanswered questions. For instance, in Section 4.1, we mentioned a possible and tantalizing way of dealing with interference: with the ordering of transistor strengths opposite to the (worst-case) ordering we chose, it might be possible to build pulsed circuits that, instead of failing, operate as NMOS circuits when the inputs are stable for too long. Another issue that needs to be explored further is the kinds of waveforms we should use for characterizing the pulse generators; speaking from intuition, the rectangular pulse shapes we used do not fit the actual pulses observed as well as certain trapezoidal shapes, whence we should expect trapezoidal test-pulses to yield a better quantitative understanding of the behavior of the pulsed circuits. Finally, we should of course be happy to remove the infinite-gain inertial-delay; are there more reasonable conditions that the circuits can be understood under?

When it comes to the circuit family itself, several questions are unanswered. Is there a more parsimonious way of implementing the STAPL handshakes? What can we do about tolerating noise (Section 6.8)—should we have some feedback from the output in the pulse generator, less internal feedback, or some other feedback arrangement (see the description of "load lines" in Section 6.8.4)?

There is a vast space of circuit designs that has not been explored: one big question is if we should always design the circuits so that the transition counts match up; in Section 5.2.1, we note that it may not always be optimal to maintain this assumption; either way, there are serious timing verification issues that need to be tackled if we want to ensure that a given circuit design satisfies the single-track–handshake constraint. Our simulations suggest that the circuit family we have given works, but that is not enough: we should like to know more exactly how the transistor delays relate to the handshake constraints (e.g., Eqs. 5.2 and 5.3). What we need to do here is show how each s and x relates to the transistor delays themselves.

As far as programming tools go, the connection between the PL1 language and CAST has not yet been made complete; the PL1 compiler also has several annoying "features" that make it somewhat difficult to use.

The SPAM implementation could, as we noted at the end of Chapter 8, easily be improved. The current design is very promising: the circuits are very fast, and we almost achieve the design objective of ten transitions per cycle. But it only works well on straightline code and on code that does not subtract small numbers from each other. What needs to be done to improve it is fairly obvious: a change to the writeback mechanism, a new adder, and—in the slightly longer term—more sophisticated branch-prediction.

10.2 Conclusion

In this thesis, we have seen the development of STAPL, a new way of building digital logic systems. The discussion has gone from a simple experiment, through elementary theory, more specific circuit theory, a family of circuits that realizes the theory, automatic design tools, and finally to a microprocessor-design example. As we said in the Introduction, we should follow Carver Mead and make the handling of system complexity the touchstone for this new implementation and realization technology. The contribution of this thesis is making it possible to build modular asynchronous systems without sacrificing performance.

So the question is: how did we do with the SPAM processor? Overall, the results were encouraging: it was easy to design the processor with the new PL1 language and the old CAST language together; the circuit performance was spectacularly good for such a comprehensive circuit technology; the performance problems that we ran into were not related to the design style, and they could easily be remedied. The chief drawbacks of our new design-style are the high transistor-count (our SPAM example has about twice as many transistors for implementing the same function as the corresponding hand-compiled MiniMIPS parts) and the high power-consumption; the transistor count is something we should not worry about (at least for logic; the *REGFILE* design shows that it is possible to save transistors by compiling things by hand), and the power consumption is mainly due to the speed of the circuits. In terms of the Et^2 and At^2 metrics, the parts of the SPAM processor design that we studied suggest that the STAPL circuits are a definite improvement over all previously known implementation technologies: the improvement is a factor of five over the MiniMIPS, which itself is as good as any other single-issue 32-bit microprocessor [55].

Appendix A

PL1 Report

We have no very useful techniques for protecting the system from software bugs. We are reduced to the old-fashioned method of trying to keep the bugs from getting into the software in the first place. This is a primary reason for programming the system in PL/I...
— *Vyssotsky, Corbató, Graham:* Structure of the Multics Supervisor *(1965)*

A.1 Introduction

This report describes Pipeline Language 1 (PL1), a specification language for processes used in highly concurrent VLSI systems. The design of PL1 was inspired by developments in asynchronous digital VLSI, especially slack elasticity and pipelined completion, but the main concepts are also familiar from earlier work on the representation of computations as dataflow graphs.

A.1.1 Scope

We shall make frequent reference to *processes* in this report. Traditionally, processes are thought of as the sequential building blocks of a parallel system. Restricting internal concurrency is too narrow a view, and we take the position that processes are simply parts of a parallel system that communicate with each other on channels. Arbitrary shared variables are hence not allowed between processes. The reader that is satisfied with using shared variables is urged to ignore the metaphysical implications of something's being a process; he can simply take the process as a syntactic construct that we introduce for structuring a concurrent system.

Programs written in PL1 describe processes, not entire systems. The hierarchy required for describing an entire system is expounded in some other language, such as the CAST language [46, 78] or a general-purpose language like C or Modula-3.

A.1.2 Structure of PL1

The PL1 language is defined by proceeding through several levels. At the lowest level are the syntactic tokens, such as keywords and identifiers. These tokens are combined to make expressions and actions. Finally, the expressions and actions are arranged to make a process description.

We discuss the syntax of the language first and the semantics later.

A.2 Syntax elements

We describe the PL1 syntax bottom-up: We start with the lexical definition of tokens and proceed later to the grammatical definition of language components.

The lexical components of a PL1 program are comments, keywords, integers, identifiers, expression operators, and special operators. Out of these components are built expressions and actions.

We shall use regular expressions [1] for describing the lexical elements of PL1.

A.2.1 Keywords

The following are keywords with special meaning in PL1: **true**, **false**, **void**, **define**, **communicate**, **goto**, **go to**, **invariant**. Keywords may not be used as identifiers.

A.2.2 Comments

A comment is, as in the C language, started by `/*` . The comment includes all the text to the next occurrence of `*/` . Comments do not nest. The text in a comment is without meaning in the language.

A.2.3 Numericals

Numerical data is limited to integers and can be expressed either in hexadecimal (base 16) or in decimal. Hexadecimals begin with the special sequence `0x`.

$< \mathit{numerical} >:=$ `[0-9][0-9]*` $|$ `0x[0-9a-f][0-9a-f]*`

A.2.3.1 Boolean numbers

For convenience, the keyword **true** is understood, in all contexts, as the integer `1`, and the keyword **false** is likewise understood as the integer `0`.

$< \mathit{integer} >:=\ < \mathit{numerical} >|\mathbf{true}|\mathbf{false}$

A.2.4 Identifiers

As identifiers, anything is legal that is *not* a keyword, a numerical, a type identifier (see Section A.3.1), or the sequence `0x`, and further is described by

< *identifier* >:= `[$a-zA-Z_0-9][$a-zA-Z_0-9]*` .

A.2.5 Reserved special operators

PL1 reserves a few tokens as special operators—they can appear outside expressions, with different meaning than when they appear within expressions. The special operators are `->`, `=`, `!`, and `?` . Of these, `?` and `->` can never appear in an expression.

A.2.6 Expression operators

The PL1 expression operators are inspired by C. Operators are either unary or binary. Note that `-` can be either the unary negation operator or the binary subtraction operator; the distinction is made in the grammar. A similar duality applies to `+`.

< *unary operator* >:= `-` | `+` | `!` | `~`

< *binary operator* >:= `*` | `/` | `%` | `<<` | `>>` | `+` | `-` | `<` | `<=` | `=<` | `>` | `>=` | `=>` | `==` | `!=` | `&` | `^` | `|` | `&&` | `||` | `#>`

A.2.7 Expression syntax

Expressions are written as infix or prefix strings of operations on integers and identifiers, as in normal mathematical notation or as in C.[1]

< *expression* >:= < *identifier* > | < *integer* > | < *unary operator* >< *expression* > |
< *expression* >< *binary operator* >< *expression* > | `(`< *expression* >`)`

A.2.8 Actions

PL1 programs execute *actions.* Three types of actions are defined: send actions, receive actions, and state actions.

< *action* >:= < *send action* > | < *receive action* > | < *state action* >

< *action subject* >:=< *identifier* >

< *action object* >:=< *expression* >

< *send action* >:= < *action subject* >`!`< *action object* >

< *receive action* >:= < *action subject* >`?`

< *state action* >:= < *action subject* >`=`< *action object* >

[1] Note that parentheses are allowed in expressions. The parentheses that denote optional elements in the grammar are set in a larger and lighter font.

A.2.8.1 Implicit declaration by actions

Making an identifier the subject of a send action implicitly declares the identifier as an output channel. Conversely, making an identifier the subject of a receive action implicitly declares the identifier as an input channel.

A.3 PL1 process description

The actions and expressions are arranged to make a PL1 process description. For completeness, we also define declarations and invariants.

A.3.1 Declarations

All PL1 variables must be mentioned in declarations before being used. Declarations can be of two kinds: argument declarations and local declarations. Argument declarations declare the input-output channels of a process; thus, argument declarations define variables that have a type denoted by channel type-identifiers. Conversely, local declarations define variables whose types are denoted by local type-identifiers. Currently we define a restricted set of data types, viz.,

$< \mathit{channel\ type\ identifier} >:=$ `e1of[1-9][0-9]*`

and

$< \mathit{local\ type\ identifier} >:=$ `1of[1-9][0-9]*` .

We currently also enforce the further restriction that all variables are of type `1of`x or `e1of`x where $x = 2^n$ for some non-negative integer $n < N$, where N is implementation-dependent. The restriction that x must be a power of two may be removed in a future implementation, but the restriction that x must be bounded is likely to remain.
Thus:

$< \mathit{argument\ decl} >:=$

$\qquad < \mathit{channel\ type\ identifier} > < \mathit{identifier} > (\texttt{,} < \mathit{identifier} >) \cdots$

$< \mathit{local\ decl} >:=$

$\qquad < \mathit{local\ type\ identifier} > < \mathit{identifier} > (\texttt{,} < \mathit{identifier} >) \cdots$

No syntax is provided for making the explicit distinction between input and output channels. However, the implementation will enforce the distinction by checking that either only receive or only send actions are performed on a given channel.

The distinction between argument types and local types is intrinsic to the language, but *the specific data types provided are subject to change.*

A.3.2 Communication statement

The communication statement joins a communication condition, in the form of an expression, with the relevant actions.

$< \mathit{guard} >:=< \mathit{expression} >$

$< \mathit{communication\ statement} >:= < \mathit{guard} > \texttt{->} < \mathit{action} > (\texttt{,} < \mathit{action} >) \cdots$

A.3.3 Process communication-block

A PL1 process consists of the following parts, in order: a process declaration, a list of local declarations, a list of invariant statements, and a communication block. Each component is optional except the process declaration itself.

$< \mathit{communication\ block} >:= \textbf{communicate}\ \{$

$\quad\quad < \mathit{communication\ statement} >\big(; < \mathit{communication\ statement} >\big)\cdots$

$\quad\quad \}$

$< \mathit{invariant} >:= \textbf{invariant}\ \{\ < \mathit{expression} >\ \}$

$< \mathit{process} >:=$

$\quad\quad \textbf{define} < \mathit{identifier} >\ \texttt{(}\big(< \mathit{argument\ decl} >\big(; < \mathit{argument\ decl} >\big)\cdots\big)\texttt{)}\{$

$\quad\quad\quad \big(< \mathit{local\ decl} >\big(; < \mathit{local\ decl} >\big)\cdots\big)$

$\quad\quad\quad \big(< \mathit{invariant} >\big(< \mathit{invariant} >\big)\cdots\big)$

$\quad\quad\quad \big(< \mathit{communication\ block} >\big)$

$\quad\quad \}$

The process is the highest-level syntactic element in PL1. The interactions between processes are handled externally to the language.

A.4 Semantics

The semantics of PL1 may be broadly divided into three categories: expression semantics, action semantics, and concurrency semantics.

A.4.1 Expression semantics

All PL1 expressions are evaluated as two's-complement binary quantities.

We have already covered the syntactic appropriateness of the various PL1 language operators. The operations defined in the grammar have the following meanings defined in the following tables.

A.4.1.1 Binary operators

The binary operators in PL1 have operator precedence as in C. In the following table, the precedence is indicated by grouping, with the precedence falling as we descend down the table. All binary operators are left-associative. (In contrast to C, the right-associative assignment is not an operator as such in PL1; it is instead part of an action statement.)

<table>
<tr><th>Operator</th><th>Interpretation</th><th>Operand(s)</th><th>Result</th></tr>
<tr><td>*</td><td>Multiplication</td><td>integer</td><td>integer</td></tr>
<tr><td>/</td><td>Division</td><td>integer</td><td>integer</td></tr>
<tr><td>%</td><td>Remainder</td><td>integer</td><td>integer</td></tr>
<tr><td>+</td><td>Addition</td><td>integer</td><td>integer</td></tr>
<tr><td>-</td><td>Subtraction</td><td>integer</td><td>integer</td></tr>
<tr><td><<</td><td>Left shift</td><td>integer</td><td>integer</td></tr>
<tr><td>>></td><td>Right shift</td><td>integer</td><td>integer</td></tr>
<tr><td><</td><td>Less than</td><td>integer</td><td>boolean</td></tr>
<tr><td><=,=<</td><td>Less than or equal</td><td>integer</td><td>boolean</td></tr>
<tr><td>></td><td>Greater than</td><td>integer</td><td>boolean</td></tr>
<tr><td>>=,=></td><td>Greater than or equal</td><td>integer</td><td>boolean</td></tr>
<tr><td>==</td><td>Equal</td><td>integer</td><td>boolean</td></tr>
<tr><td>!=</td><td>Not equal</td><td>integer</td><td>boolean</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>integer</td><td>integer</td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>integer</td><td>integer</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>integer</td><td>integer</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>boolean</td><td>boolean</td></tr>
<tr><td>||</td><td>Logical OR</td><td>boolean</td><td>boolean</td></tr>
<tr><td>#></td><td>Logical IMPLIES</td><td>boolean</td><td>boolean</td></tr>
</table>

A.4.1.2 Unary operators

The unary operators have higher precedence than any binary operators and are listed in the following table.

Operator	Interpretation	Operand(s)	Result
!	Logical NOT	boolean	boolean
~	Bitwise NOT	integer	integer
-	Negation	integer	integer
+		integer	integer

Because of the syntax of actions, expression operators have higher precedence than delimiters used in actions.

A.4.1.3 Boolean type-coercion

As in C, coercion between boolean values and integer values is done as follows:

1. A boolean result used as an operand to an integer operator is interpreted as `1` if it evaluates to **true** and as `0` if it evaluates to **false**.
2. An integer result used as an operand to a boolean operator is interpreted as **false** if it evaluates to `0` and as **true** in all other cases.

These are the same rules as are used for converting the constants **true** and **false** to integers.

A.4.1.4 Integer type-coercion

If the size (in bits) of the result of an evaluation does not match the size of the variable that it is assigned to or the size of the channel that it is sent on, the result is either sign-extended (if it is too narrow) or bitwise truncated (if it is too wide). The use of negative quantities is, in general, discouraged since all built-in datatypes are unsigned.

A.4.1.5 Use of channel identifiers

An identifier used in an expression that refers to a channel or to a state variable evaluates to the current value of the channel or state variable in question. If there is no current value (because none has been sent on that channel), then the expression does not evaluate. There is no way of accessing a value corresponding to the "undefined" or "no-data" state of a channel. The channel value cannot change during the current round of execution because it can only be updated after it has been removed by the receiving process.

A.4.2 Action semantics

Three types of variables with associated actions are defined in PL1. Send actions are defined for output channels, receive actions are defined for input channels, and assignment actions are defined for state variables. Channels between processes are first-in–first-out.

A.4.2.1 Receive actions

When a receive action is enabled for an input channel, the value present on the input channel will be disposed of, after it has been used in any expressions that it appears in. On the next round of execution of the process, the next value will be provided, if necessary.

A.4.2.2 Send actions

When a send action is enabled for an output channel, a value equal to the current value of the expression that is the object of the send action will be sent on the channel.

A.4.2.3 Assignment actions

When an assignment action is enabled for a state variable with an object expression that evaluates to w, the value present in the state variable on the current round will be disposed of. On the next round of execution of the process, the next value $v_{i+1} = w$ will be provided, if necessary.

A.4.3 Execution semantics

The semantics of a PL1 process may be defined in terms of an *execution.* The execution of the process may either fail, in which case no actions are performed, or it may succeed, in which case all enabled actions are performed concurrently. If the execution fails, it will be retried at a later time.

The execution of a PL1 process can be thought of as the infinite loop:

Wait until it can be determined, for each guard, whether it evaluates to **true** *or* **false***;*
Wait until all values required for computing action objects are available;
Concurrently execute all enabled actions.

The execution of a PL1 process may succeed only if enough operands are available such that it is possible to evaluate *all* communication-statement guards either to **true** or to **false** (using the type-coercion rules, if necessary) and if all values required for computing the objects of the send and assignment actions are available. If these conditions do not obtain, the execution will fail. The evaluation of the guards and the variables required for computing the objects of the actions may be performed concurrently; likewise, the actions may be performed concurrently. However, the evaluation of the guards and the variables required for computing the objects of the actions strictly precedes the actions themselves—this ensures that the guards and action objects are stable.

A.4.4 Invariants

Invariants are provided as a convenience. The programmer indicates that some predicate will hold as a precondition of the execution of a program, given that the involved values may be computed. The invariant may be used to simplify the implementation, and the implementation may optionally check that the invariant is always satisfied and else abort the computation in an implementation-dependent way.

A.4.5 Semantics in terms of CHP

The execution semantics of a PL1 program may be described in terms of the extended CHP language, which includes the value probe and peek [66].

A.4.5.1 The channel peek

The *peek* ¿ works like a receive, except that it leaves the channel in the state it was in before the peek was executed.

$$\{\overline{X}\}X¿x\{\overline{X}\}$$

A.4.5.2 Channel values

We use the idea of the value on a channel for defining the value probe. The same idea is also used for defining the semantics of expressions in PL1. The value on a channel X, $\mathbf{val}(X)$ may be defined in terms of Hoare triples as follows:

$$\{\mathbf{val}(X) = v \wedge \overline{X}\}X?x\{x = v\}$$

$$\{\mathbf{val}(X) = v \wedge \overline{X}\}X¿x\{x = v\}$$

(But of course $X?x$ and $X¿x$ have different effects on the next value that shall be seen on the channel.)

A.4.5.3 The value probe

Slack elasticity allows the value probe

$$\overline{<, i :: X_{\mathrm{i}} >: P(<, i :: X_{\mathrm{i}} >)}$$

to be defined for one channel as

$$\overline{X : P(X)} \equiv \overline{X} \wedge P(X)\Big|_{\mathrm{X} \to \mathbf{val}(\mathrm{X})}$$

and extended to predicates involving multiple channels as

$$\overline{X, Y : S(X) \wedge S(Y)} \equiv \overline{X : S(X)} \wedge \overline{Y : S(Y)} \quad (*)$$

$$\overline{X, Y : S(X) \vee S(Y)} \equiv \overline{X : S(X)} \vee \overline{Y : S(Y)} \quad (\dagger).$$

An alternative definition is possible by defining the value probe directly for multiple channels and replacing the equivalence with $\equiv_\infty$ in $(*)$ and $(\dagger)$, where $\equiv_\infty$ denotes equivalence under infinite slack.

Alternatively, a direct definition of the value probe is possible:

$$\{\overline{X : P(X)}\}X?v\{P(v)\}$$

$$\{\overline{X} \wedge \neg\overline{X : P(X)}\}X?v\{\neg P(v)\}$$

$$\{\sim \overline{X : P(X)}\}X?v\{\neg P(v)\}$$

However, in PL1 the concept of **val**(X) is ubiquitous, since it is used directly in expression evaluation.

A.4.5.4 Semantics in terms of value probe

To define the semantics of the PL1 process, we must specify what is meant by "waiting until it can be determined, for each guard, whether it evaluates to **true** or **false**." We therefore introduce the tilde operator as follows:

$$\sim \overline{X : P(X)} \equiv \overline{X : \neg P(X)}$$

For instance,

$$\sim \overline{X, Y : S(X) \vee S(Y)} = \overline{X : \neg S(X)} \wedge \overline{Y : \neg S(Y)}.$$

At this point, we can define the semantics of the PL1 program. The program

$$\textbf{communicate}\{G_0 \to C_0; \ldots; G_n \to C_n\}$$

where the C_i's do not use variables and no action is enabled more than once on a single iteration of the program is defined as

$$
\begin{array}{l}
*[\;\; [< \wedge i : n : G_i \;\vee\; \sim G_i >]; \\
\quad <\| \; i : n : [G_i \;\longrightarrow\; C_i \;▯\; \sim G_i \;\longrightarrow\; \textbf{skip}] > \\
\;]\;.
\end{array}
$$

If the C_i's use variables, these must be renamed so that there is no conflict in executing the C_i's concurrently. We introduce the notation **vars**(X) for the set of variables that the action X depends on. The program definition is then

$$
\begin{array}{l}
*[\ \ [<\wedge i : n : G_{\mathrm{i}}\ \vee\ \sim G_{\mathrm{i}} >];\\
\quad <\| \ i : n : [G_{\mathrm{i}} \ \longrightarrow\ < v : v \in \mathbf{vars}(C_i) : L_v ¿ \lambda_v > \ \ [] \ \sim G_{\mathrm{i}} \ \longrightarrow\ \mathbf{skip}] >;\\
\quad <\| \ i : n : [G_{\mathrm{i}} \ \longrightarrow\ C_{\mathrm{i}}\big|_{v:v\in\mathbf{vars}(C_i):v\rightarrow\lambda_v} \ \ [] \ \sim G_{\mathrm{i}} \ \longrightarrow\ \mathbf{skip}] >\\
\ \ ]\ ,
\end{array}
$$

where L_v denotes the input channel associated with the name v and λ_v is a temporary local variable; the notation $v \rightarrow \lambda_v$ means that we replace each variable v with the temporary λ_v.

If any actions are enabled more than once on a single iteration, the actions must have the same action objects (i.e., the same values for sends); multiply enabled actions behave like the single execution of one of the actions.

A.4.6 Slack elasticity

Slack elasticity allows leeway in terms of the exact ordering of actions by PL1 programs. If a system is slack elastic, then it does not matter when values are sent on channels, as long as they are sent in the right order. The informal definition of the execution semantics of PL1, as well as the definition in terms of CHP, provides the least amount of slack possible. Given that the system being designed is slack-elastic, the only way in which the specification could be violated by the implementation is through the introduction of deadlock. Since the PL1 semantics as defined here has the least slack of all possible implementations, any slack-elastic system that behaves correctly and avoids deadlock with the PL1 semantics will behave correctly and avoid deadlock using any legal implementation.

In practice, an implementation of a PL1 process in a slack-elastic system is allowed to produce output values as soon as they can be determined, which can be before all the guards have been checked. This property can be used to great effect, e.g., in production-rule implementations.

A.5 Examples

A process that repeatedly sends the value `1` on its output channel would be written as follows:

```
define bitgen(e1of2 r)
{
    communicate {
        true -> r!1;
    }
}
```

A full-adder would be written as follows:

```
define fulladder(e1of2 a,b,c; e1of2 s,d)
{
    communicate {
        true -> s!(a+b+c)&0x1,d!(!!((a+b+c)&0x2)),a?,b?,c?;
    }
}
```

In the mysterious expression `d!(!!((a+b+c)&0x2))`, the first exclamation mark denotes the send communication, whereas the next two are C-style inversions. (The value of the expression `!!x` is zero if `x` is zero and one otherwise.)

A two-input merge would be written as follows:

```
define merge(e1of2 l0,l1,s; e1of2 r)
{
    communicate {
        true -> s?;
        s == 0 -> r!l0, l0?;
        s == 1 -> r!l1, l1?;
    }
}
```

A contrived example PL1 program that does nothing very interesting (except illustrate some of the syntax of the language) is shown here:

```
define contrivedExample(e1of2 l0, l1, c; e1of2 r, z)
{
    invariant { l0 + l1 + 2*c > 1 }
    communicate {
        !(c == 1) -> r!l0, l0?, z!(c + l0);
        c == 1 && l1 == 0 -> r!l1, z!1;
        c == 1 && l1 == 1 -> r!0;
        c == 1 -> l1?;
        true -> c?;
    }
}
```

Illustrating the use of state variables, we may write an alternator as follows:

```
define alternator(e1of2 r)
{
    1of2 s;
    communicate {
        true -> s=!s,r!s;
    }
}
```

This page intentionally left blank.

Appendix B

SPAM Processor Architecture Definition

Nevertheless, The year's penultimate month *is not in truth a good way of saying* November.
— *H.W. Fowler,* A Dictionary of Modern English Usage *(1926)*

B.1 Introduction

This appendix describes the Simple Pulsed Asynchronous Microprocessor (SPAM) architecture. SPAM is a simple 32-bit RISC architecture intended for hardware demonstration projects. Its design reflects a desire of making a high-performance implementation as easy as possible. This is not without merit on the software level; for instance, as a result of the desire of keeping the hardware as simple as possible, the instruction set of the SPAM processor is completely orthogonal; i.e., all instructions use the same addressing mode and instruction format.

B.2 SPAM overview

The SPAM architecture defines eight general-purpose registers, `gpr[0]` through `gpr[7]`, of which `gpr[0]` is always read as zero, although it may be written by any instruction. Apart from these, the processor state consists only of the program counter, `pc`. The instructions provided are arithmetic instructions, load-store instructions, and `pc`-changing instructions. Changes to `pc` take effect immediately—there is no "branch delay slot." The architecture does not define floating-point operations, interrupts, or exceptions.

B.3 SPAM instruction format

All SPAM instructions have the same format. The instruction format is a four-operand RISC format with three register operands and a single immediate operand. The opcode format has two fields, which are also the same across all instructions. These fields are the operation unit and the operation function. The operation "Y-mode," which determines the addressing mode used for conjuring operand `opy`, is further defined in a fixed position in the instruction.

SPAM instructions are 32 bits wide. Considering a SPAM instruction i as a 32-bit array of bits, we identify the fields of the instruction:

1. The `opcode` $= i[31 \ldots 27]$, further grouped into:
   (a) The unit number `unit` $= i[31 \ldots 30]$.
   (b) The function `fxn` $= i[29 \ldots 27]$.
2. The Y-mode `ymode` $= i[26 \ldots 25]$.
3. The result register number `rz` $= i[24 \ldots 22]$.
4. The X-operand register number `rx` $= i[21 \ldots 19]$.
5. The Y-operand register number `ry` $= i[18 \ldots 16]$.
6. The immediate field `imm` $= i[15 \ldots 0]$.

B.4 SPAM instruction semantics

Because the SPAM instruction set is orthogonal, we may define the semantics of instructions in a modular way. An instruction execution consists of the following steps:

1. Generating the operands:

   $$\texttt{opx} := \texttt{gpr[}i\texttt{.rx]} \text{ and } \texttt{opy} := \texttt{YMODE(}i\texttt{.ymode)(gpr[}i\texttt{.ry],}i\texttt{.imm)}$$

2. Computing the result:

   $$\texttt{opz} := \texttt{OP(}i\texttt{.opcode)(opx,opy)}$$

   (a) Computing the next `pc`:

   $$\texttt{pc} := \texttt{PCOP(}i\texttt{.opcode)(pc,opx,opy)}$$

3. Writing back `opz`:

   $$\texttt{gpr[}i\texttt{.rz]} := \texttt{opz}$$

All instructions are executed in these three steps. Hence, all instructions produce a result that is written back in the register file; if the value is not needed for further computation, it should be discarded by setting i`.tz` to zero (in the assembly language, this can be accomplished by leaving out the target register). In what follows, we shall mainly deal with how `opz` is computed (i.e., the part above denoted by `OP`), since all else is the same for all instructions, except that branches also need to compute `pc` (denoted by `PCOP`).

B.4.1 Operand generation

The first operand, `opx`, is always the contents of `gpr[`i`.rx]`. The second operand, `opy`, is computed from the contents of `gpr[`i`.ry]` and the immediate field, depending on i`.ymode`.

Allowable values for i`.ymode` are as follows, where *sext* signifies sign extension:

i`.ymode` Mnemonic	Decimal value	Operand generated
YMODE_REG	0	opy := gpr[i.ry]
YMODE_IMM	1	opy := *sext*(i.imm)
YMODE_IMMSHIFT	2	opy := i.imm << 16
YMODE_REGIMM	3	opy := gpr[i.ry] + *sext*(i.imm)

B.4.2 Operation definitions

Operations are defined on two's-complement numbers. There are no flags or condition codes. We group the operations by unit:

B.4.2.1 ALU operations i.unit = UNIT_ALU = **0**

All ALU operations take two operands and produce one result. The *bitwise_NOR* is included in the instruction set for the express purpose of computing the bitwise inverse of `opx` using a zero operand for `opy`.

Mnemonic	Name	i.fxn	Operation
`add`	Add	0	`opz := (opx + opy)`$_{31\ldots0}$
`sub`	Subtract	1	`opz := (opx - opy)`$_{31\ldots0}$
`nor`	NOR	4	`opz :=` *bitwise_NOR*`(opx,opy)`
`and`	AND	5	`opz :=` *bitwise_AND*`(opx,opy)`
`or`	OR	6	`opz :=` *bitwise_OR*`(opx,opy)`
`xor`	Exclusive OR	7	`opz :=` *bitwise_XOR*`(opx,opy)`

B.4.2.2 Branch operations i.unit = UNIT_BRCH = **1**

Branch operations include unconditional jumps (`jmp`) and the halt instruction (`hlt`). All branch operations unconditionally produce the same result, namely the value of `pc`, right-shifted by two; this value is used for `opz`. Likewise, a branch taken will branch to the address denoted by `opy` incremented by one and left-shifted by two. The shifting avoids having to define the behavior of alignment errors and allows larger immediate branch-offsets.

Note that the mechanism described for branch addresses allows a simple compilation of function call-return linkage. The function-call jump saves the current PC, and then the function-return jump calls back through the saved address. Coroutine linkage is compiled similarly. (The SPAM architecture leaves unspecified function-parameter–linkage conventions and register-save masks, etc.)

The `hlt` instruction halts the machine. An external action, not defined within the architecture, is required for restarting it.

Conditional branches branch on the value of `opx`.

Mnemonic	Name	i.fxn	Branch if	Target
`hlt`	Halt	0	**true**	$\perp$
`beq`	Branch on Equal	1	opx $= 0$	$(\texttt{opy}_{29\ldots0}+1)\|00$
`bne`	Branch on Not Equal	2	opx $\neq 0$	$(\texttt{opy}_{29\ldots0}+1)\|00$
`bgt`	Branch on Greater Than	3	opx > 0	$(\texttt{opy}_{29\ldots0}+1)\|00$
`blt`	Branch on Less Than	4	opx < 0	$(\texttt{opy}_{29\ldots0}+1)\|00$
`ble`	Branch on Less or Equal	5	opx ≤ 0	$(\texttt{opy}_{29\ldots0}+1)\|00$
`bge`	Branch on Greater or Equal	6	opx ≥ 0	$(\texttt{opy}_{29\ldots0}+1)\|00$
`jmp`	Jump	7	**true**	$(\texttt{opy}_{29\ldots0}+1)\|00$

B.4.2.3 Memory operations i.unit = UNIT_DMEM = 2

Only two memory operations are defined: load word, `lw`; and store word, `sw`. The address of the memory access is determined by `opy`. On a memory load, `opx` is ignored; whereas on a store, it becomes the value stored. A store returns `opy` (the computed address) as `opz`; this allows coding postincrement and postdecrement addressing-modes in a single instruction.

Mnemonic	Name	i.fxn	Operation
`lw`	Load Word	0	`opz := dmem[opy]`
`sw`	Store Word	1	`dmem[opy] := opx, opz := opy`

B.4.2.4 Shifter operations i.unit = UNIT_SHFT = 3

The SPAM architecture defines a restricted shifter that is capable only of logical shifts. Arithmetic shifts must be simulated using `blt`. The SPAM shifter can shift by one or eight. Shifts-by-eight are provided so that byte memory-operations can proceed at a reasonable speed.

Mnemonic	Name	i.fxn	Operation
`sr1`	Shift Right by One	0	opz := $0\|\texttt{opy}_{31\ldots1}$
`sr8`	Shift Right by Eight	1	opz := $00000000\|\texttt{opy}_{31\ldots8}$
`sl1`	Shift Left by One	2	opz := $\texttt{opy}_{30\ldots0}\|0$
`sl8`	Shift Left by Eight	3	opz := $\texttt{opy}_{23\ldots0}\|00000000$

B.4.2.5 Undefined operations

Operations not yet defined are reserved for future expansion and must not be used. The behavior of the undefined operations is UNDEFINED (the machine may take any action, which includes the possibility of its hanging [3]).

B.4.2.6 System reset

The mechanism for causing a system reset is implementation-dependent. On system reset, the processor starts execution with `pc` = 8 and arbitrary data in all general-purpose registers except `gpr[0]`.

B.5 Assembly-language conventions

The SPAM architecture uses a simple, understandable assembly-language syntax that is free from the traditional confusion about which register identifier names the operand and which names the result.

B.5.1 The SPAM assembly format

The SPAM assembly format is best illustrated with an example (this example mainly illustrates the syntax of the format; a more reasonable program from the point of view of efficiency is the Fibonacci program of Section 8.4.2):

```
;;; Compute sum of 100 first integers
;;; Do some other things to test the processor
.=0x8
             jmp Start           ; comment
.=0x100
Start:
             li r1=100
             li r2=0U            ; upper immediate
             jmp r3=Detour       ; comment
Label:                           ; comment
             add r2=r1,r2
             sw r2,(100)
             lw r2=(r1+0x3ff)
             lw r2=(100)
             sub r1=r1,1
             bne r1,Label
             hlt
             jmp zero            ; shouldnt get executed
             nop
.=0x200                          ; test comment
Detour:      jmp r3
```

B.5.1.1 Assembly instruction syntax

In the example, we see the use of some standard assembler conventions, such as the use of "`.`" for setting the desired memory location of the current instruction. We also see that the syntax of the instructions is $<$ *mnemonic* $><$ *result register* $>=<$ *operands* $>$. Register indirect and indexed register-indirect memory-instructions are written with parentheses, similarly to the MIPS assembly format.

Labels can be used directly by the branches. Any field not specified will be assembled as zero; this has several benefits—e.g., not specifying the target register of an operation makes the target `gpr[0]`, which means that the result shall be discarded.

B.5.1.2 Specification of immediates

Immediates are specified either in decimal or in hexadecimal. Hexadecimal numbers must be preceded with the string `0x` to flag their base. Following an immediate with the roman capital `U` flags it as being an "upper" immediate; i.e., it will be shifted 16 bits left before it is used.

B.5.1.3 Pseudo-instructions

There are also several *pseudo-instructions* in the example program that are understood by the assembler and mapped to the machine-language instructions presented earlier. The pseudo-instructions understood by the assembler are as follows:

Pseudo-instruction	Name	Operation
`li rz=opy`	Load immediate	`or rz=r0,opy`
`nop`	No operation	`add r0=r0,r0`
`not rz=opy`	NOT	`nor rz=0,opy`

Notice that the `nop` pseudo-instruction conveniently assembles to an all-zeros instruction word.

Appendix C

Proof that Definition 3.2 Defines a Partial Order

—How now, you secret, black, and midnight hags! What is't you do?

—A deed without a name.

— *William Shakespeare,* Macbeth *(c. 1605)*

In an earlier version of the manuscript, Definition 3.2 was claimed to apply to the equivalence classes under translation of all functions $\{f : t \to V\}$. This is not quite true. Consider[1] $f(x) = x$ and $g(x) = x + I(x)$, which do not obey the restriction

$$\big(\exists k, F :: (\forall x : |x| > k : f(x) < F) \wedge (\exists x :: f(x) > F)\big). \tag{C.1}$$

It is here clear that by Definition 3.2, $\tau(g) \leq \tau(f)$ and $\tau(f) \leq \tau(g)$, yet under the normal concept of equality, it is not the case that $\tau(f) = \tau(g)$; in other words, it is not true that $\exists\delta :: \big(\forall x :: g(x) = f(x-\delta)\big)$. The three requirements for a relation's being a partial order, viz. reflexivity ($a \leq a$), transitivity ($a \leq b \wedge b \leq c \Rightarrow a \leq c$), and anti-symmetry ($a \leq b \wedge b \leq a \Rightarrow a = b$), are thus not satisfied for equivalence classes under translation unless we change to a different notion of equality; specifically, our $\leq$ is not anti-symmetric.

Changing the notion of equality would however upset the definition of $\mathcal{F}$; we should also notice that for anything that intuitively looks like a "pulse" (as well as for many other functions), Definition 3.2 seems quite reasonable in conjunction with the normal definition of equality. It is hence simpler to restrict the functions under consideration.

Our restriction (C.1) solves the problem. We shall prove the following theorem.

Theorem C.1 *For equivalence classes under translation of continuous functions that satisfy (C.1), Definition 3.2 defines a partial order.*

[1] $I(x)$ represents the unit-step function; see Eq. 3.2, p. 26.

Proof. Reflexivity and transitivity hold; this is true by inspection. We prove anti-symmetry for equivalence classes of continuous functions $\tau(f)$ and $\tau(g)$, where f and g are arbitrary representatives of the chosen classes. Assume that there exist $g(x)$ and $f(x)$ satisfying the "pulse restriction" (C.1) such that $\tau(g) \leq \tau(f) \wedge \tau(f) \leq \tau(g)$. We have by $\tau(g) \leq \tau(f) \wedge \tau(f) \leq \tau(g)$ that

$$\exists \delta_1, \delta_2 :: f(x - \delta_1) \leq g(x) \leq f(x - \delta_2). \tag{C.2}$$

Hence[2]

$$\exists \delta_1, \delta_2 :: f(x - \delta_1) \leq f(x - \delta_2), \tag{C.3}$$

or more simply stated,

$$\exists \delta :: f(x) \leq f(x - \delta). \tag{C.4}$$

We say that either (1) $\delta = 0$ (i.e., $\delta_1 = \delta_2$ and hence $\forall x :: g(x) \leq f(x - \delta_1) \wedge f(x - \delta_1) \leq g(x)$, in other words $\forall x :: g(x) = f(x - \delta_1)$); or (2) f is *weird.*

But no continuous f that satisfies (C.1) is weird. Consider the part of the domain of f where f equals or exceeds F. There is a smallest value l where f equals F; this is defined by $f(l) = F \wedge \big(\forall x : x < l : f(x) < F\big)$; likewise there is a greatest m defined by $f(m) = F \wedge \big(\forall x : x > m : f(x) < F\big)$; we know that l and m exist because f is continuous and $\forall x : x < -k \vee x > k : f(x) < F$ by (C.1). Now consider $f(x) = f(x - \delta)$. If $\delta > 0$, then the equation cannot hold because $f(l - \delta) < F = f(l)$, and likewise for m and $\delta < 0$. Hence any f satisfying (C.1) is non-weird and $\forall x :: g(x) = f(x - \delta_1)$.

Thus $\tau(g) \leq \tau(f) \wedge \tau(f) \leq \tau(g) \Rightarrow \exists \delta :: \big(\forall x :: g(x) = f(x - \delta)\big)$, or more succinctly,

$$\tau(g) \leq \tau(f) \wedge \tau(f) \leq \tau(g) \Rightarrow f \sim g \tag{C.5}$$

under Definition 3.2 and translation equivalence; in other words,

$$\tau(f) \leq \tau(g) \wedge \tau(g) \leq \tau(f) \Rightarrow \tau(f) = \tau(g), \tag{C.6}$$

i.e., $\leq$ is anti-symmetric over the equivalence classes under translation containing continuous functions satisfying (C.1). *Q.E.D.*

C.1 Remark on Continuity

We have proved that our definition of $\leq$ establishes a partial order over equivalence classes under translation of *continuous* functions satisfying the restriction C.1. It is not difficult to generalize the proof so that it covers functions that are not continuous but still satisfy (C.1), but the argument

[2]This simple & crucial observation was made by Karl Papadantonakis.

becomes considerably more opaque. The chief difficulty is that we have asserted that "there is a smallest value l defined by $f(l) = F \wedge \big(\forall x : x < l : f(x) \le F\big)$"; this need no longer be true if f has discontinuities, since we might have that $f(x) < F$ for $x \le x_0$ and $f(x) > F$ for $x > x_0$; i.e., f exceeds F over an open interval. Changing the definition of l to make it clear that l exists would consequentially mean using a more complicated argument for showing that f is non-weird, because it may be that directly evaluating $f(l)$ does not yield the desired value; in other words, we should have to consider the values of f in a neighborhood of l instead of just at l itself.

We omit the more general proof because only continuous functions are of physical interest, and the extra complications would only obscure the basic idea of the restriction C.1. Yet we should remember that the testing pulses that we used in Chapter 3 were not continuous, so putting the theory on entirely solid ground would require either finishing this proof or changing those pulses to be continuous—neither of which would change the essence of the mathematics.

This page intentionally left blank.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, Mass.: Addison-Wesley, 1986.

[2] Phillip E. Allen and Douglas R. Holberg. *CMOS Analog Circuit Design.* Oxford: Oxford University Press, 1987.

[3] Alpha Architecture Committee. *Alpha Architecture Reference Manual*, third edition. Boston, Mass.: Digital Press, 1998.

[4] Abraham Ankumah. Designing an Energy-Efficient Asynchronous 80C51 Microcontroller. B.S. thesis, California Institute of Technology, Division of Engineering and Applied Science, Department of Electrical Engineering, 2001.

[5] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. *Revised Report on the Algorithmic Language* ALGOL 60. Berlin: Springer-Verlag, 1969.

[6] Ganesh Balamurugan and Naresh R. Shanbhag. The Twin-Transistor Noise-Tolerant Dynamic Circuit Technique. *IEEE Journal of Solid State Circuits*, **36**(2), February 2001.

[7] Kees (C. H.) van Berkel. Beware the Isochronic Fork. *Integration, the VLSI Journal.* **13**(2), June 1992, pp. 103–128.

[8] Kees van Berkel and Arjan Bink. Single-Track Handshake Signaling with Application to Micropipelines and Handshake Circuits. In *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 1996.

[9] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation*, pp. 384–389, 1991.

[10] Dileep Bhandarkar. *Alpha Implementations and Architecture: complete reference and guide.* Newton, Mass.: Digital Press, 1996.

[11] Gerrit A. Blaauw and Frederick P. Brooks, Jr. *Computer Architecture: Concepts and Evolution.* Reading, Mass.: Addison-Wesley, 1997.

[12] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, **C-35**(8), August 1986, pp. 677–691.

[13] Steven M. Burns. *Automated Compilation of Concurrent Programs into Self-timed Circuits.* M.S. thesis, Caltech CS-TR-88-2, California Institute of Technology, 1988.

[14] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits.* Ph.D. thesis, California Institute of Technology, 1991.

[15] Steven M. Burns and Alain J. Martin. Performance Analysis and Optimization of Asynchronous Circuits. In Carlo H. Séquin, ed., *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference.* Los Alamitos, Calif.: IEEE Computer Society Press, 1991.

[16] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Reading, Mass.: Addison-Wesley, 1988.

[17] T. I. Chappell, B. A. Chappell, S. E. Schuster, J. W. Allan, S. P. Klepner, R. V. Joshi, and R. L. Franch. A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE Journal of Solid State Circuits*, **26**(11), November 1991.

[18] U. V. Cummings, A. M. Lines, and A. J. Martin. An Asynchronous Pipelined Lattice Structured Filter. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 1994.

[19] Jack B. Dennis. Data Flow Supercomputers. *Computer*, November 1980, pp. 48–56. IEEE Computer Society, 1980.

[20] Digital Equipment Corporation. *PDP-6 Arithmetic Processor 166 Instruction Manual.* Maynard, Mass.: Digital Equipment Corporation, c. 1960. This can currently be obtained from Tom Knight's web page at MIT: `http://www.ai.mit.edu/people/tk/pdp6/pdp6.html`

[21] Edsger W. Dijkstra. *A Discipline of Programming.* Englewood Cliffs, N.J.: Prentice-Hall, 1976.

[22] Don Draper, Matt Crowley, John Holst, Greg Favor, Albrecht Schoy, Jeff Trull, Amos Ben-Meir, Rajesh Khanna, Dennis Wendell, Ravi Krishna, Joe Nolan, Dhiraj Mallick, Hamid Partovi, Mark Roberts, Mark Johnson, and Thomas Lee. Circuit Techniques in a 266-MHz MMX-Enabled Processor. *IEEE Journal of Solid-State Circuits*, **32**(11), November 1997, pp. 1650–1664.

[23] J. Ebergen. Squaring the FIFO in GASP. In *ASYNC 2001: Proceedings of the Seventh International Symposium on Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 2001.

[24] Raphael A. Finkel. *Advanced Programming Language Design.* Menlo Park, Calif.: Addison-Wesley, 1996.

[25] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. *Proceedings of the VII Banff Workshop: Asynchronous Hardware Design*, August 1993.

[26] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. *Proceedings of the IEEE*, **87**(1), 1999.

[27] Theodore W. Gamelin and Robert Everist Greene. *Introduction to Topology,* second edition. Mineola, N.Y.: Dover Publications, 1999.

[28] J. D. Garside, S. Temple, and R. Mehra. The AMULET2e cache system. In *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 1996.

[29] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Reading, Mass.: Addison Wesley, 1985.

[30] Marcel R. van der Goot. *Semantics of VLSI Synthesis.* Ph.D. thesis, California Institute of Technology, 1996.

[31] Mark R. Greenstreet and Ian Mitchell. Reachability Analysis Using Polygonal Projections. In *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*, March 1999, pp. 103–116. In series: *Lecture Notes in Computer Science*, 1569. Berg en Dal, the Netherlands: Springer-Verlag, 1999.

[32] Mark R. Greenstreet and Tarik Ono-Tesfaye. A Fast ASP* RGD Arbiter. In *Proceedings of the Fifth International Symposium on Research in Asynchronous Circuits and Systems*, Barcelona, Spain. Los Alamitos, Calif.: IEEE Computer Society Press, 1999.

[33] Matthew Hanna. *CHP.* Unpublished, California Institute of Technology Computer Science Department, 2000. May be obtained from the author.

[34] Scott Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, **83**(1), 1995.

[35] John Hennessey and David Patterson. *Computer Architecture: A Quantitative Approach,* first ed. San Mateo, Calif.: Morgan-Kaufmann, 1990.

[36] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, 1978.

[37] H. Peter Hofstee. Synchronizing Processes. Ph.D. thesis, California Institute of Technology; Division of Engineering and Applied Science, 1995.

[38] J. H. Hubbard and B. H. West. *Differential Equations: A Dynamical Systems Approach (Ordinary Differential Equations).* New York, N.Y.: Springer-Verlag, 1991.

[39] John H. Hubbard and Barbara Burke Hubbard. *Vector Calculus, Linear Algebra, and Differential Forms: A Unified Approach.* Upper Saddle River, N.J.: Prentice-Hall, 1999.

[40] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Englewood Cliffs, N.J.: Prentice-Hall, 1992.

[41] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, second ed. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

[42] Tak Kwan (Tony) Lee. *A General Approach to Performance Analysis and Optimization of Asynchronous Circuits.* Ph.D. thesis, Caltech-CS-TR-95-07. California Institute of Technology, Division of Engineering and Applied Science, 1995.

[43] Andrew Lines. *Pipelined Asynchronous Circuits.* M.S. thesis, California Institute of Technology CS-TR-95-21, 1995.

[44] Misha Mahowald and Rodney Douglas. A silicon neuron. *Nature*, **354**, 19/26 December 1991, pp. 515–518.

[45] Rajit Manohar. *The Impact of Asynchrony on Computer Architecture.* Ph.D. thesis, CS-TR-98-12, California Institute of Technology, July 1998.

[46] Rajit Manohar. *Cast: Caltech Asynchronous Tools.* TEXinfo documentation package. Unpublished, California Institute of Technology Department of Computer Science/Cornell University Department of Electrical Engineering, 1998–2001.

[47] Rajit Manohar, Mika Nyström, and Alain J. Martin. Precise Exceptions in Asynchronous Processors. In Erik Brunvand and Chris Myers, eds., *Proceedings of the 2001 Conference on Advanced Research in VLSI (ARVLSI 2001).* Los Alamitos, Calif.: IEEE Computer Society Press, 2001.

[48] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

[49] A. J. Martin. Synthesis of Asynchronous VLSI Circuits. In J. Staunstrup, ed., *Formal Methods for VLSI Design*. North-Holland, 1990.

[50] Alain J. Martin. *Asynchronous Circuits for Token-Ring Mutual Exclusion.* Caltech Computer Science Technical Report CS-TR-90-09. Pasadena, Calif.: California Institute of Technology Computer Science Department, 1990.

[51] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W. J. Dally, ed., *Sixth MIT Conference on Advanced Research in VLSI.* Cambridge, Mass.: MIT Press, 1990.

[52] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive VLSI circuits. In C. A. R. Hoare, ed., *Developments in Concurrency and Communication*, in UT Year of Programming Series, pp. 1–64. Englewood Cliffs, N.J.: Addison-Wesley, 1990.

[53] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, ed., *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373. Cambridge, Mass.: MIT Press, 1991.

[54] Alain J. Martin. *Synthesis of Asynchronous VLSI Circuits.* Caltech Computer Science Technical Report CS-TR-93-28. Pasadena, Calif.: California Institute of Technology Computer Science Department, 1993.

[55] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Pénzes, R. Southworth, U. Cummings, and T. K. Lee. The Design of an Asynchronous MIPS R3000 Processor. In R. B. Brown and A. T. Ishii, eds., *Proceedings of the 17th Conference on Advanced Research in VLSI.* Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

[56] Alain J. Martin, Andrew M. Lines, and Uri V. Cummings. Pipelined Completion for Asynchronous Communication. U.S. Patent 6,038,656. Mar. 14, 2000.

[57] Alain J. Martin. Towards an energy complexity of computation. *Information Processing Letters*, **77**(2001), pp. 181–187.

[58] Carver Mead. *Analog VLSI and Neural Systems.* Reading, Mass.: Addison-Wesley, 1989.

[59] Carver A. Mead. VLSI and Technological Innovation. In Charles L. Seitz, ed., *Proceedings of the Caltech Conference on Very Large Scale Integration,* Pasadena, Calif.: California Institute of Technology Computer Science Department, 1979.

[60] Carver Mead and Lynn Conway, *Introduction to VLSI Systems.* Reading, Mass.: Addison-Wesley, 1980.

[61] C. E. Molnar, I. W. Jones, W. S. Coates, J. K. Lexau, S. M. Fairbanks, and I. E. Sutherland. Two FIFO Ring Performance Experiments. *Proceedings of the IEEE*, **87**(1), 1999.

[62] Gordon E. Moore. Are We Really Ready for VLSI? In Charles L. Seitz, ed., *Proceedings of the Caltech Conference on Very Large Scale Integration,* Pasadena, Calif.: California Institute of Technology Computer Science Department, 1979.

[63] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *The Annals of the Computation Laboratory of Harvard University. Volume XXIX: Proceedings of an International Symposium on the Theory of Switching, Part I.*, 1959, pp. 204–243.

[64] C. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, **1**(2), June 1993, pp. 106–119.

[65] Greg Nelson, ed. *Systems Programming with Modula-3.* Englewood Cliffs, N.J.: Prentice Hall, 1991.

[66] Mika Nyström. *Pipelined Asynchronous Cache Design.* M.S. thesis, California Institute of Technology CS-TR-97-21, 1997.

[67] Ad Peeters and Kees van Berkel. Synchronous Handshake Circuits. In *ASYNC 2001: Proceedings of the Seventh International Symposium on Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 2001.

[68] Ad M. G. Peeters. *Single-Rail Handshake Circuits.* Ph.D. thesis, University of Eindhoven (Technische Universiteit Eindhoven), 1996.

[69] Paul I. Pénzes. *The design of high performance asynchronous circuits for the Caltech MiniMIPS processor.* M.S. thesis, California Institute of Technology, 1999.

[70] Paul I. Pénzes. Private communication, 2001.

[71] Robert J. Proebsting. Speed Enhancement Techniques for CMOS Circuits. U.S. Patent 4,985,643. Jan. 15, 1991.

[72] William F. Richardson. *Architectural Considerations in a Self-Timed Processor Design.* Ph.D. thesis, Department of Computer Science, University of Utah, 1996.

[73] C. T. Sah. Characteristics of the Metal-Oxide-Semiconductor Transistors. *IEEE Transactions on Electron Devices*, **ED-11**, 1964, p. 324.

[74] S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous Interlocked Pipelined CMOS Circuits Operating at 3.3–4.5MHz[sic! should be GHz; may be

corrected in final version]. Paper 17.3 in *Technical Digest of 2000 IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, Calif., 2000.

[75] Charles L. Seitz. “System timing,” Chapter 7 in [60].

[76] William M. Siebert. *Circuits, Signals, and Systems.* Cambridge, Mass.: MIT Press, 1986.

[77] Thomas D. Simon. *Fast CMOS Buffering With Post-Charge Logic.* S.M. thesis, Massachusetts Institute of Technology, 1994.

[78] Robert Southworth, Matthew Hanna, and Rajit Manohar. *CAST 2.000.* Unpublished, California Institute of Technology Computer Science Department, 2000. May be obtained from the author.

[79] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits.* San Francisco, Calif.: Morgan Kaufmann, 1999.

[80] I. Sutherland and S. Fairbanks. GasP: A Minimal FIFO Control. In *ASYNC 2001: Proceedings of the Seventh International Symposium on Asynchronous Circuits and Systems.* Los Alamitos, Calif.: IEEE Computer Society Press, 2001.

[81] A. Takamura, M. Kuwako, M. Imai, T. Fuji, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on Scalable-Delay-Insensitive model. In *Proceedings of the International Conference on Computer Design (ICCD)*, 1997.

[82] C. D. Thompson. Area-Time Complexity for VLSI. In Charles L. Seitz, ed., *Proceedings of the Caltech Conference on Very Large Scale Integration,* Pasadena, Calif.: California Institute of Technology Computer Science Department, 1979.

[83] José Tierno. Private communication, 2001.

[84] Stephen A. Ward and Robert H. Halstead, Jr. *Computation Structures.* Cambridge, Mass.: MIT Press, 1990.

[85] Ted E. Williams. *Self-Timed Rings and their Application to Division.* Ph.D. thesis, Computer Systems Laboratory, Stanford University, 1991.

[86] Catherine G. Wong. *A Graphical Method for Process Decomposition.* M.S. thesis, California Institute of Technology, 2000.

[87] Catherine G. Wong and Alain J. Martin. Data-Driven Process Decomposition for the Synthesis of Asynchronous Circuits. Submitted for publication to ICECS 2001 (Sept. 2001).

[88] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple. AMULET1: An Asynchronous ARM Microprocessor. *IEEE Transactions on Computers*, **46**(4), April 1997, pp. 385–397.