High-Level Synthesis and Rapid Prototyping of Asynchronous VLSI Systems

Thesis by

Catherine Grace Wong

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

2004

(Defended May 21, 2004)

All of us are better when we are loved.

— Alistair McLeod

To my parents, Andrew and Nancy Wong.

Acknowledgements

I would very much like to begin by thanking my advisor, Alain Jean Martin, who introduced me to a world without clocks. He has been a wonderful mentor, guiding me with wisdom, patience, humour, and care. This thesis would not have been possible without his inspiration and support, and I shall always appreciate his willingness to make time (even foregoing sleep on Saturday mornings!) to discuss my research. The best of what I learned at Caltech, and what I continue to learn, can be summed up by his lessons: to be intellectually daring, and to strive for elegance in all things.

Other professors have been inspiring as well. In particular, I thank André DeHon for his teaching an excellent class at Caltech on electronic design automation, and for his feedback on my thesis and other research papers. I would also like to thank the other professors on my Ph.D. committee, Mani Chandy and Jason Hickey, for their helpful critiques and advice. I am grateful to Jonathan Rose, my undergraduate advisor at the University of Toronto, for introducing me to reconfigurable computing, and for offering suggestions on the asynchronous FPGA research presented here. I also thank Tarek Abdelrahman and Corinna Lee, two professors who both inspired and encouraged me to pursue research in computer hardware when I was an undergraduate student.

My years in graduate school have been enriched and enlivened by my fellow students in the asynchronous VLSI group: Mika Nyström, Andrew Lines, Paul Pénzes, Robert Southworth, Uri Cummings, Eitan Grinspun, Matt Hanna, Karl Papadantonakis, Piyush Prakash, Wonjin Jang, and Jonathan Dama. By listening, encouraging, commenting and debating, they have all contributed to and improved the quality of the research presented here. I thank them for their camaraderie, and will truly miss group meetings together every Friday at 9:20 am. Mika especially has been a great part of my life at Caltech, and I am thankful for not only his detailed comments on this work, but for his friendship.

Other graduate students in the Computer Science department have also kept me sane (or happily insane, depending on your point of view). Nathan Litke, Mark Meyer, Eve Schooler, Ilja Friedel, and Jessi Stumpfel all shone their lights in my life in their own special ways, making the department feel like a home. And, needless to say, all would have been chaos without the help of our friendly department administrative staff. In particular, I'm not sure where I would be (stranded in Japan?) were it not for the aid of Diane Goodfellow, Betta Dawson, and Jeri Chettum. Thank you!

I have been blessed with many amazing friendships throughout my time in Los Angeles. I thank every kindred spirit who played with me, prayed with me, fed me, chauffered me, and cheered me on with motivational messages written in Rice Krispie square letters!

Most importantly, I thank my family for their unfailing love and support.

My big sister has always been there to listen, to laugh, to send me books, and to pick me up when I'm down. I treasure and am eternally grateful for our peerless "culture of two." Her home and family have been a refuge: her husband is the big brother I always wanted, and their children are an endless source of delight.

My parents have always challenged me to do great things, but have never left me in doubt of their unconditional love. From arithmetic lessons with Greenie the squeaky bath-toy dinosaur, to the everyday examples of the grace and heart with which they live their own lives, they have taught and shaped me in innumerable ways. Life has always been joyful in the Wong household! I dedicate this thesis wholeheartedly to my mother and my father, and I thank God for them every day.

Lastly, I thank my fiancé Peter: my rock, and my love. Our time together at Caltech has been a wonderful prelude; I stand alongside him full of hope, faith, and joy as the rich music of our life together is about to begin.

Abstract

This thesis introduces data-driven decomposition (DDD), a new method for the high-level synthesis of asynchronous VLSI systems and the first method to target high-performance asynchronous circuits. Given a sequential description of circuit behavior, DDD produces an equivalent network of communicating processes that can each be directly implemented as fine-grained asynchronous pipeline stages. Control and datapath are integrated within each pipeline stage of the final system.

We present many aspects of the synthesis of asynchronous VLSI systems, including general circuit templates that DDD uses to estimate low-level performance and energy metrics while optimizing the concurrent system. We also introduce a new circuit model and new techniques for slack matching, a performance optimization that inserts pipelining into a system to modify asynchronous handshake dynamics and increase throughput. The entire method is then applied to a complex control unit from an asynchronous 8051 microcontroller, as an example.

This thesis also introduces a new architecture for an asynchronous field-programmable gate array (FPGA). The architecture is cluster-based and, unlike most FPGA designs, contains an entirely delay-insensitive interconnect. The basic reconfigurable cells of this FPGA fit the asynchronous pipeline-stage circuit-template used by DDD, and the reconfigurable clusters include circuitry that implements features assumed by an optimization phase of DDD, which reduces the energy consumption of the system.

Contents

List of Figures

Chapter 1

Introduction

Almost 40 years after the initial observation of Moore's law, the data density of chips continues to increase. Today, VLSI chips are among the most complex systems in technology. Fragility accompanies complexity, though, and today's chips are so finely tuned that small irregularities or errors can render sophisticated systems useless. Set apart from performance or power targets, managing complexity is the most important challenge facing the VLSI community. Our focus is on creating methods for VLSI system design that can handle the high concurrency of complex systems and separate the issue of system correctness from performance assumptions.

1.1 The Advantages of Asynchrony

Aside from complexity, increased power consumption and decreased robustness are currently the most pressing issues that VLSI designers grapple with. In synchronous design, long the mainstay of VLSI systems, global clock signals are implicated in both problems. Clock activity consumes a significant amount of energy, especially when long clock distribution wires must switch. Meanwhile, both the uncertainties in propagation delays of clock signals across a chip and the reduced device reliability resulting from shrinking transistor sizes contribute to timing uncertainties in the system. These uncertainties force designers to pad global clock periods with large safety margins, slowing down the entire system.

These issues are often addressed by moving away from global synchronization and replacing the single system clock with a collection of local clocks. Local clock signals do not span as great distances across a chip, and can be stopped and started independently of each other to reduce dynamic energy consumption while continuing to allow computations in other parts of the system. However, the creation of multiple clock domains introduces complexities at domain interfaces that reduce modularity, and does not remove the potential of a single fabrication or design error from casting the entire system into disarray.

The alternative to synchronous design is asynchronous design, which eschews clocks entirely. Asynchronous systems can be modeled as message-passing networks. Global synchronization is replaced with local handshakes between the communicating asynchronous circuits. Practical evidence of the energy and speed advantages of asynchrony (described in detail below) is provided by the results of the most recent large asynchronous design: an 8051 microcontroller. Compared to synchronous 8051s all running at the same operating voltage in the same fabrication technology, the asynchronous chip both runs twice as fast as the advertised high-speed synchronous 8051, and consumes only 75% of the energy of the advertised low-power synchronous 8051 [41].

Some design issues—most notably the problems related to clock-tree distribution—disappear completely when a system is implemented asynchronously. Other issues, such as high dynamic energy consumption, can be alleviated by the elimination of regularly switching clock networks (as well as by the absence of glitches, that are not allowed in asynchronous logic). There can be performance advantages as well, as fast computations in the system are no longer held back by slower computations that restrict global-clock periods. An asynchronous system is free to run in "average-case," rather than "worst-case," time. If a component runs slowly but infrequently, it is not necessary to use every means possible to bring the system at large up to the desired speed; so synthesis tools do not need to work so hard.

The elimination of the global clock cuts the number of timing assumptions in the system by varying degrees, depending on the style of asynchrony chosen. As will be discussed, in the quasi delay-insensitive (QDI) asynchronous style of this thesis, only one easily manageable timing assumption remains. Other less conservative asynchronous design styles such as *bundled data* have enough timing assumptions that although theoretically they would run faster than QDI systems, practically the safety margins necessitated by uncertainty force them to run at a slower pace. In any case, one immediate advantage of the elimination of some timing assumptions is that fewer or even no timing assumptions need be reconciled across component interfaces. As the data density of chips increases and systems-on-chip become more prevalent, this boost to the modularity and re-usability of asynchronous circuits makes them more attractive.

Perhaps the greatest current advantage of asynchronous design is increased robustness: the lack of timing assumptions separates issues of performance from issues of correctness. This allows asynchronous systems to continue functioning despite variations in process technology or other physical parameters such as voltage or temperature (some asynchronous systems have been shown to operate properly at sub-threshold voltages [42]), and enables further energy savings by allowing energy to be traded off against performance through voltage scaling without any need for special circuitry or ramping protocols. It also simplifies system synthesis, matching particularly well with high-level synthesis.

One drawback of asynchronous VLSI is the area overhead required by extra circuitry and wires to eliminate timing assumptions by encoding a signal's validity within the signal itself. This overhead cost is offset by the fact that shrinking feature sizes both reduces the cost of area and increases the time overhead due to uncertainty that is being eliminated by the extra circuitry. Even more than area, the major obstacle preventing asynchronous VLSI from becoming a generally viable and desired alternative to clocked chips is the current lack of design and test tools for high-performance asynchronous systems. The research presented in this thesis seeks to remove a significant part of this obstacle.

1.2 VLSI Synthesis

While their details differ, the synthesis flows for synchronous and asynchronous VLSI can be organized into analogous stages. Both design methods naturally begin with architectural design and system specification. They then move on through behavioral, or high-level, synthesis, where algorithmic descriptions are analyzed and structured into separate components, usually each at the level of fine-grained pipeline stages. Finally, actual circuits are generated by logic synthesis and physical design, and the overall system must be verified.

Specifically, the asynchronous design algorithms used in this thesis belong to the Caltech synthesis method for asynchronous VLSI [37]. This method consists of a series of semantics-preserving program transformations that ultimately convert an initial behavioral system specification into the equivalent of a transistor netlist. The method is correct by construction, and every chip designed using this approach (including the fastest working asynchronous microprocessors to date [39]) has been functional on first silicon.

In synchronous VLSI, where correctness is inextricably linked to performance, much of the focus is on design and verification at lower levels of synthesis (logic and physical), where it is essential to achieve precise timing closure. In asynchronous VLSI however, small variations in circuit delays and other low-level details can be easily tolerated by the system without significant degradation in performance.

Much of the design emphasis in the synthesis of asynchronous VLSI is instead placed at higher levels, when algorithms are decomposed into networks of small components, or processes. This transformation is called *process decomposition*. With the absence of clocks and glitches, and the necessity of all communication being acknowledged in local synchronizing handshakes, the inter-process communications mapped during high-level synthesis have a significant effect on the energy consumption of the final system. Also, while high performance is not necessary for correctness in asynchronous systems, it is still obviously desirable. The throughput of asynchronous systems is most dependent upon what is known as *pipeline dynamics* or *handshaking dynamics*, as determined during high-level synthesis.

The relatively dim spotlight on high-level synthesis in synchronous design, combined with the lack of mature design tools for high-performance asynchronous design has relegated behavioral synthesis tools to the status of constantly emerging technology.

1.3 Contributions

The first contribution of this thesis is *data-driven decomposition (DDD)*, a new method for process decomposition that transforms a sequential program into a network of communicating processes by analyzing the data dependencies in the original program. Basic DDD is independent of the intended application of the target system. It is in fact more closely related to work on compiling programs for data-flow architectures [2, 18, 31, 32, 16] and on data flow graphs for optimizing software compilers [3, 12, 46, 49] than to other hardware synthesis methods. Unlike software or data flow single-assignment language compilers, DDD is intended for use in the generation of actual circuits. We present specific optimizations that make it the first behavioral synthesis method for high-performance asynchronous VLSI systems.

While other asynchronous CAD tools exist, they are either syntax-directed and cannot generate processes that are small enough for high throughput [4, 5, 9], or they begin at a level lower than behavioral synthesis [11, 19, 28]. Until now, the fastest asynchronous microprocessors have been decomposed manually, a task that is painstaking and whose results are highly dependent upon the experience and intuition of the designer.

Within the framework of DDD for asynchronous VLSI, we

- present generalized circuit templates that can be used to estimate the performance, energy consumption and size of high-level processes when they are implemented as asynchronous pipeline stages;
- demonstrate new techniques for both high-level sequential programs and concurrent systems to reduce the energy consumption of the final asynchronous VLSI system;
- and demonstrate new techniques to optimize the throughput of an asynchronous system while still minimizing energy consumption.

The second contribution of this thesis is the presentation of a new clustered architecture for asynchronous field-programmable gate arrays (FPGAs). As VLSI systems grow more complex, system design time also grows and re-programmability is an increasingly attractive option in a world where hardware development cycles can no longer keep pace with mask costs, and the technological advancements and demands of applications. The modularity and robustness of asynchronous systems makes them ideal vehicles for reconfigurable computation since they fit easily on systems-on-chip and adapt easily to changing requirements for performance and power.

Unlike most previous asynchronous reconfigurable designs [22, 25, 33, 45], our architecture integrates datapath with control, and has a fully delay-insensitive interconnect. The lack of global timing assumptions across the interconnect eases the strain on place and route tools considerably by eliminating the necessity for complete timing closure. The main logic cells of this FPGA are based on the same asynchronous pipeline stages targeted by DDD. Together, DDD and the reconfigurable asynchronous architecture present new opportunities for designers seeking to synthesize asynchronous VLSI systems.

1.4 Organization

The organization of this thesis is as follows:

1. We develop the method of data-driven decomposition (DDD), which transforms a sequential algorithm into an equivalent system of communicating processes. The first half of DDD eliminates false syntactic data dependencies from the sequential algorithm, and the second half analyzes the real data dependencies to decompose the algorithm into a concurrent system (Chapter 2).

2. We introduce basic concepts of asynchronous VLSI including quasi delay-insensitivity and synchronization through handshakes. We demonstrate the formal synthesis of asynchronous VLSI and the generation of common circuits. We create general circuit templates for asynchronous pipeline stages and demonstrate how they can be used to estimate low-level circuit performance metrics from high-level algorithms (Chapter 3).

3. Making use of the new asynchronous circuit templates, we present optimizations to basic DDD that generate practical asynchronous VLSI systems. These optimizations include "distillation,"

which identifies scenarios where conditional communications can be exploited to redesign asynchronous pipeline stages so that they are idle and consume no dynamic energy for extended periods of time (Chapter 4).

4. Given a decomposed network of asynchronous pipeline stages, we introduce a new clustering phase that analyzes circuit performance estimates and pipeline dynamics to perform "recomposition" (energy optimization) and "slack matching" (asynchronous throughput optimization) simultaneously. These transformations can be considered analogous to partitioning and retiming in synchronous VLSI systems. Their end result is an energy-efficient asynchronous system capable of running at specified target speeds (Chapter 5).

5. We demonstrate the complete DDD method on a large control unit from an asynchronous 8051 microcontroller and compare its results to those of manual decomposition (Chapter 6).

6. We present a new architecture for asynchronous FPGAs that includes basic logic cells and clusters of logic cells with added functionality, and demonstrate how a typical asynchronous microprocessor unit can be implemented on the architecture. We analyze tradeoffs in the design of delay-insensitive programmable interconnect, and introduce a parameterized model of the FPGA architecture for use in future work (Chapter 7).

1.5 Notation

This programming language *CHP (Communicating Hardware Processes)* is used throughout this thesis [37]. A more complete description of its syntax is provided in Appendix A, but we introduce it briefly here. CHP is a high-level hardware description language that includes communications primitives and concurrent processes. It is a simple imperative language, and none of its constructs are explicitly tailored for hardware implementation.

CHP *variables* can be integers, enumerations, or arrays. A *process* is a single imperative program that manipulates variables. Communications primitives can be used to transfer data and synchronize computations with other processes. *Communications channels* are dedicated between two processes, and have only one send and one receive port. A *system* consists of a group of processes, each running concurrently and sharing information through communications across *channels*. Shared variables are not allowed between processes in a system.

The major constructs of the CHP language used in this thesis are

Basic Statements:

- $x := expr$: Assignment.

  Assign the value of expression *expr* to variable x.

- $A?a$: Input communication.

  Assign the value on input channel A to variable a.

- $B!b$: Output communication.

  Send the value of variable b on channel B.

- **skip**: Do nothing.

Composition of Statements:

- $A; B$: Execute A and then B in sequence.

- A, B: Execute A and B in parallel. Binds more strongly than ";."

- $A \parallel B$: Execute A and B in parallel. Binds less strongly than ";."

Control Structures:

- $[g \rightarrow A [] h \rightarrow B]$: Deterministic selection statement.

  If g is true, execute A. If h is true, execute B. The program waits for one of g and h to be true; g and h must be mutually exclusive.

- $*[g \rightarrow A [] h \rightarrow B]$: Deterministic repetition statement.

  If g is true, execute A. If h is true, execute B. Repeat this behavior until both g and h are false; g and h must be mutually exclusive.

- `*[`A`]`: Unconditional repetition statement.

  Repeat A indefinitely. Equivalent to "`*[`**true** $\rightarrow$ A`]`."

Chapter 2

Data-Driven Decomposition

Process decomposition transforms a sequential program into an equivalent network of communicating processes. This chapter introduces *data-driven decomposition (DDD)*, a new method for the process decomposition of deterministic processes. DDD transforms programs into single-assignment form and uses data-dependency analysis when applying the projection technique to decompose a program into a distributed system. We present the backbone of the DDD method, which is generally applicable not only to asynchronous VLSI design but also to synchronous design and parallel programming in software.

2.1 Introduction

The goals of process decomposition are to expose concurrency, to pipeline computations, and to divide the original sequential program into a network of processes, making any further transformations (such as low-level synthesis) easier. DDD performs process decomposition by analyzing data dependencies in the original program to remove unnecessary synchronization and then partitioning the program into an equivalent system of target processes. If decomposition is applied as part of a circuit-synthesis flow, DDD can factor in performance metrics such as cycle time and energy consumption. Source and target processes are expressed in CHP.

As an example, the program

$$P \;\equiv\; *[A?a, B?b;\;\; x := f(a,b);\;\; X!x;\;\; C?c, D?d;\;\; y := g(c,d);\;\; Y!y;\;\; Z!h(x,y)]$$

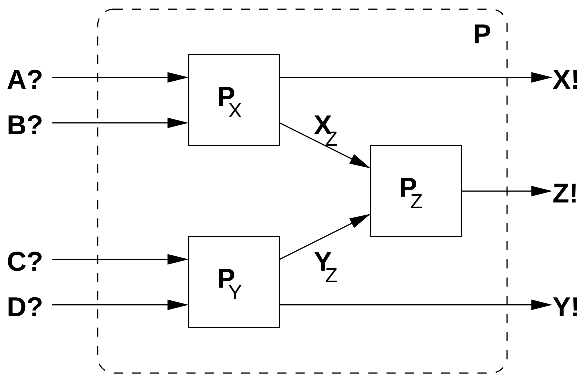


Figure 2.1: Example of process decomposition.

can be decomposed into the system given below (see Figure 2.1):

$$
\begin{aligned}
P &\ \triangleright\ P_X \ \| \ P_Y \ \| \ P_Z \\
P_X &\ \equiv\ *[\ A?a,\ B?b;\ x := f(a,b);\ X!x,\ X_Z!x\ ] \\
P_Y &\ \equiv\ *[\ C?c,\ D?d;\ y := g(c,d);\ Y!y,\ Y_Z!y\ ] \\
P_Z &\ \equiv\ *[\ X_Z?x,\ Y_Z?y;\ Z!h(x,y)\ ]
\end{aligned}
$$

Decomposition has introduced concurrency to the program, as the computations of x and of y now occur in parallel. Also, if further synthesis is to be performed on the system, the target processes P_X, P_Y and P_Z are more easily compiled.

There are three main phases in DDD:

1. **DSA conversion:** The first phase converts sequential programs into dynamic single-assignment (DSA) form by splitting variables so that each is *defined* (assigned a value) only once during execution. This removes unnecessary synchronization resulting from two unrelated variable instances possessing the same name, leaving only inherent data dependencies in the sequential program.

2. **Projection:** The second phase of DDD partitions the DSA sequential program into a new system with one process for every variable in the code. Thus data dependencies are made explicit, as each decomposed process includes all computations of its variable, input communications with processes whose variables are used in the computations, and output communications with

processes that use its computation results in their computations.

3. **Clustering:** The final phase of DDD applies only when DDD is used to generate circuit systems. DDD clusters decomposed processes back together, simultaneously reducing communications overhead (particularly reducing energy consumption) and optimizing system throughput.

We begin presenting DDD by introducing basic concepts that are used in our data-driven method (Section 2.2). DDD itself is introduced in Section 2.3, where we outline its first phase: transforming sequential programs into DSA form. The DSA program can then be partitioned into parallel processes through application of the projection technique, described in Section 2.4. Finally, Section 2.5 presents related work, and Section 2.6 summarizes the basic methods for the first two phases of DDD. Additional asynchronous VLSI optimizations for these first two phases are described in Chapter 4, and the final phase of clustering for asynchronous VLSI is presented in Chapter 5.

2.2 Basic Concepts for Data-Driven Decomposition

This section introduces the key concepts behind data-driven decomposition, such as program execution, program equivalence, and reaching definitions. These concepts set up the formal presentation of the DDD method in following sections.

2.2.1 Program Execution

DDD manipulates deterministic programs that can be either *terminating* or *non-terminating*. A terminating program contains no unconditional loops. We consider only non-terminating programs P that fit the following template:

$$P \;\equiv\; P_{init};\; \texttt{*[}\; P_{loop}\; \texttt{]}$$

where P_{init} and P_{loop} are themselves terminating programs. (Nested unconditional loops are not sensible. As will be described later in this chapter, if P_{loop} contains conditional loops, it can be treated as a terminating program.)

We define a *trace* of a CHP program to be a sequence of basic statements (assignments and communications) that occurs when the program is executed. A deterministic program with parallel compositional operators can have multiple traces, since statements on either side of the operators can be executed in any order. For example, the code "$S1, S2$" specifies that statements $S1$ and $S2$ can be executed in any order, and so both traces "$S1; S2$" and "$S2; S1$" can occur. The *general trace* of a program P represents all possible execution traces of P, and is denoted as $\mathbf{tr}(P)$.

When reasoning formally about order and program transformations, we make use of the *order relation* $\prec$. Given two instances of statements $S1$ and $S2$, $S1 \prec S2$ indicates that statement $S1$ always precedes statement $S2$ in every possible execution trace of the program. (The notation $S2 \succ S1$ is equivalent to $S1 \prec S2$.) If $S1 \prec S2$, then either there is at least one semicolon between them in the code, or $S1$ and $S2$ appear in different iterations of the same loop with $S1$ in the earlier iteration. In the example code "$S1, S2; S3$," we have $S1 \prec S3$ and $S2 \prec S3$ but no such order relation between $S1$ and $S2$. Given a set of statements S_i from a program P, we define $\min\limits_{\prec}\{S_i\}$ to be the set of statements that can be executed first among the statements of S_i in general trace $\mathbf{tr}(P)$. Similarly, we define $\max\limits_{\prec}\{S_i\}$ to be the set of statements that can be executed last among the statements of S_i in general trace $\mathbf{tr}(P)$.

2.2.2 Program Equivalence

DDD consists of a sequence of program transformations. The program transformations of the DSA phase all transform sequential programs into new sequential programs. The program transformations of the projection phase transform sequential programs into concurrent programs. The correctness of any transformation $P \rhd Q$ depends on the *program equivalence* of the original program P and the newly-generated program Q. (We use the notation "$P \rhd Q$" to indicate that program P is transformed into program Q.) We call two programs equivalent if their observable behavior is the same. This section focuses on sequential program equivalence; the notion of equivalence between sequential and concurrent programs is discussed in Section 2.4.

The observable behavior of a non-terminating CHP program P is its *communications trace* (the projection of its trace onto communication actions). Since there are no shared variables between CHP programs, communication channels are a program's only interface to the outside world, and all other program variables are strictly local. The observable behavior of a terminating CHP program includes its communications trace, but can also include "results" that are stored in the program's variables (called "result variables") at the end of execution. If temporary variables have been used in the program, the program designer may wish to designate only a subset of variables as result variables.

We define $P.V$ to be the set of all variables used in a program P, and $P.V_{res} \subseteq P.V$ be the set of result variables for a terminating program P. For non-terminating programs P, we define $P.V_{init}$ to be the set of variables that are assigned values (whether through regular assignment statements or input communications) in the initial code P_{init}. $P.V_{res}$ consists of any variables whose values can be used before being defined in an iteration of P_{loop}.

Finally, for every channel C in P, we define the *individual communications trace* $\mathbf{tr}(P)\lceil C$ to be the projection of the communications trace **tr** on only communications using C. Two individual communications traces are equivalent only if their lengths are identical, and the sequence of values communicated are identical.

Let us now apply a transformation to some deterministic program P such that $P \rhd Q$. For purposes of equivalence, we place only two restrictions on the transformation. First, the program interface (its input and output communication channels) remains unchanged, so every channel C in program P has a unique corresponding channel C_Q in program Q. Secondly, if the program is terminating, every result variable x in the original program P has a set of designated corresponding result variables $V_Q(x)$ in the transformed program.

We can now define program equivalence for terminating programs.

Definition 1 **[terminating-program equivalence]** *Given a deterministic terminating program P, if $P \rhd Q$ then $P \overset{pgm}{\equiv} Q$ if, for every possible trace of P and of Q:*

- *For any variable $x \in P.V_{res}$, the value stored in x at the completion of $\mathbf{tr}(P)$ equals the values stored in variables $y \in V_Q(x)$ at the completion of $\mathbf{tr}(Q)$.*

- *For every channel C in P,* $\mathbf{tr}(P)\lceil C \equiv \mathbf{tr}(Q)\lceil C_Q$

□

Moving on to programs with infinite traces, consider the general program P which is a non-terminating program that can be specified using terminating programs P_{init} and P_{loop}.

Definition 2 [non-terminating-program equivalence] *Given a deterministic non-terminating program* $P \equiv P_{init}; *[P_{loop}]$*, if* $P \rhd Q$ *then* $P \overset{pgm}{\equiv} Q$ *if:*

- $P_{init} \overset{pgm}{\equiv} Q_{init}$*, and*

- $(P_{loop})^n \overset{pgm}{\equiv} (Q_{loop})^n$*, for all iteration indices n.*

□

Since $P_{loop} \overset{pgm}{\equiv} Q_{loop}$, given the same inputs, they always perform the same communications and compute the same results. We must still guarantee that they have the same inputs for each iteration. The program P is non-terminating, so the "inputs" to each iteration of P_{loop} are the values stored in the result variables, which can be used before they are defined in an iteration of P_{loop}. Thus, for every variable $x \in P_{loop}.V_{res}$, the variable instances in Q_{loop} that correspond to any instances of x prior to its first assignment in P_{loop} must all have the same name as the variables in Q_{loop} that correspond to the instance of x on the LHS of its last assignment in P_{loop}.

2.2.3 Assignments and Data Dependencies

DDD uses assignments as its basic unit of reasoning. In this context, it considers both regular assignments and communication statements to be "assignments," and both regular variables and communication channels to be "variables." We say that variable x *directly depends* on variable y if y is used in an assignment to x. More specifically, x uses y if:

- y appears on the RHS of a regular assignment statement to x ("$x := y + 1$"),

- y is an input channel and x is its input variable ("$y?x$")

  (x is on the LHS and y on the RHS of this assignment),

$$\begin{array}{lll@{\qquad}ll@{\qquad}ll}
P \equiv & y := 0 & //\,\mathsf{S1} & Q \equiv & y_1 := 0 & Q' \equiv & y_1 := 0 \\
 & Y?y & //\,\mathsf{S2} & & Y?y_2 & & Y?y_2 \\
 & x := y + 1 & //\,\mathsf{S3} & & x := y_2 + 1 & & x := y_1 + 1
\end{array}$$

Figure 2.2: Three programs demonstrating the importance of preserving reaching definitions. If x is a result variable, P and Q are equivalent but P and Q' are not. If x is not a result variable, all three are equivalent.

- x is an output channel and y appears in the expression that it sends ("$x!f(y)$")

  (x is on the LHS and y on the RHS of this assignment), or

- x is assigned a value within a guarded command and y appears in its guard condition

  ("$\texttt{[}y \rightarrow x{\uparrow}\,\square\,\neg y \rightarrow \textbf{skip}\texttt{]};$" y is on the RHS and x is on the LHS of such an assignment).

In general, x *depends* on y if and only if

- x directly depends on y, or

- x directly depends on variable z, and z depends on y.

2.2.4 Reaching Definitions

We use *reaching definitions* [1, 10] to reason about whether a transformation changes the flow of data in a program. Since changing data flow can lead to different values' being output or stored in result variables, reaching definitions play an important role in correctness proofs for transformations.

For example, consider the program P from Figure 2.2. There are two definitions of y in this program (S1 and S2) but the results of only assignment S2 are used in statement S3. We therefore say that S2 *reaches* S3, or that the reaching definition of y in S3 is S2. Now consider program transformations that rename variable y in the code. In one case, P is transformed into Q and in another, P is transformed into Q'. Only P and Q are equivalent (assuming that x is the result variable) because when the variable on the LHS of assignment S2 was renamed, the same renaming was applied to all statements reached by S2. Thus, the reaching definition was preserved.

Definition 3 [reaching definition] *The reaching definition of x in statement S is*

$$\mathbf{RD}(x,S) \equiv \max_{\prec} \{T \in P : T \prec S \ \cap\ T.LHS \equiv x\}$$

If there are no assignments to x that precede statement S, then $\mathbf{RD}(x,S) \equiv \perp$. $\square$

Consider a program P with variable x and statements S and T. If $T \equiv \mathbf{RD}(x,S)$ then, using the notation of Hoare triples, for every possible execution trace of P:

$$\{x \neq X\}; \ \ T; \ \ \{x = X\} \ \ \ldots \ \ \{x = X\}; \ \ S$$

where the condition $x = X$ holds true for some value X throughout "...." In the example of Figure 2.2 then, $\mathbf{RD}(y, \mathsf{S3}) \equiv \mathsf{S2}$. Note that for a self-referencing assignment such as "$x := x + 1$," the reaching definition for x in this statement is a prior assignment, and not the assignment itself.

The remainder of this section considers transformations that insert and rename variables, insert and reorder statements, but do not delete or alter the operations performed in statements. We call this class of transformations *operation-preserving transformations*. Since statements are never deleted, then if $P \overset{op}{\triangleright} Q$, every statement S in P has a corresponding statement S_Q in Q. Variables in P can be renamed in multiple ways but statement operations cannot be changed, so we index variables by their instances within statements. Thus, if statement S uses variables $v_S\texttt{[}i\texttt{]}$, each with an index i that is unique for S, then $v_S\texttt{[}i\texttt{]}$ has a corresponding variable $v_{S_Q}\texttt{[}i\texttt{]}$ in S_Q.

We can now say that a transformation *preserves reaching definitions* if:

$$\mathbf{RD}(v_S[i], S) \equiv T \ \ \Rightarrow \ \ \mathbf{RD}(v_{S_Q}[i], S_Q) \equiv T_Q$$

When program transformations insert copy statements into the code, reaching definitions can be changed without affecting the correctness of the transformation. For example, when a copy statement "$x := y$" is inserted, even if y is replaced by x in all statements reached by the copy statement, the reaching definitions of the program have changed. Since we use the preservation of reaching definitions in our correctness proofs for DSA transformations that include the insertion of copy statements, we extend the concept to introduce *effective reaching definitions.*

Definition 4 [effective reaching definition] *Let S_{cp} be a chain of copy statements inserted into P when $P \rhd Q$, and let $S_{cp}.V$ be the set of variables used and defined in this chain. For any statement S_Q in Q and variable $v \in S_{cp}.V$ where $\mathbf{RD}(v, S_Q) \in S_{cp}$, the effective reaching definition of v in S_Q is*

$$\mathbf{RD}_{\mathit{eff}}(v, S_Q) \equiv \mathbf{RD}((\min_{\prec}\{S_{cp}\}).RHS, \min_{\prec}\{S_{cp}\})$$

□

We say that a transformation "preserves effective reaching definitions" if

$$\mathbf{RD}(v_S[i], S) \equiv T \;\;\Rightarrow\;\; \mathbf{RD}(v_{S_Q}[i], S_Q) \equiv T_Q \;\;\cup\;\; \mathbf{RD}_{\mathit{eff}}(v_{S_Q}[i], S_Q) \equiv T_Q$$

For example, given the program

$$P \;\;\equiv\;\; a := 0,\;\; B?b;\;\; x := a + b$$

let SA, SB, and SX be the assignments to a, b, and x, respectively. Also, let $v_{SX}\texttt{[0]} \equiv a$ and $v_{SX}\texttt{[1]} \equiv b$. Thus, $\mathbf{RD}(v_{SX}\texttt{[0]}, SX) \equiv SA$ and $\mathbf{RD}(v_{SX}\texttt{[1]}, SX) \equiv SB$. Let $P \rhd Q$, and

$$Q \;\;\equiv\;\; a := 0,\;\; B?b_1;\;\; b_2 := b_1;\;\; x := a + b_2$$

Now, SA_Q, SB_Q, and SX_Q correspond to the assignments to a, b_1, and x. We also have $v_{SX_Q}[0] \equiv a$ and $v_{SX_Q}[1] \equiv b_2$, where b_2 is a new variable defined by a new copy statement. Despite the new variable names and copy statement, this transformation has preserved effective reaching definitions since $\mathbf{RD}(v_{SX_Q}[0], SX_Q) \equiv SA_Q$ and $\mathbf{RD}_{\mathit{eff}}(v_{SX_Q}[1], SX_Q) \equiv SB_Q$.

2.2.5 DSA Transformations

The compiler transformations used by DSA conversion are variable-renaming and copy-propagation. Consider the following program:

$$P \;\;\equiv\;\; A?x;\;\; B?y;\;\; X!f(x, y);\;\; C?x;\;\; Y!g(x)$$

As an example of renaming variables, the variable x may be split into two new variables x_1 and x_2:

$$P_{var} \;\equiv\; A?x_1;\;\; B?y;\;\; X!f(x_1, y);\;\; C?x_2;\;\; Y!g(x_2)$$

Meanwhile, copy propagation adds a copy assignment to the program. It is often combined with variable renaming:

$$P_{cpp} \;\equiv\; A?x;\;\; B?y;\;\; X!f(x, y);\;\; C?x;\;\; y := x;\;\; Y!g(y)$$

DSA transformations do not remove statements or alter the structure of operations performed by statements in any way. They are therefore operation-preserving transformations (as described at the end of the previous section), and we can use the following notation: if $P \rhd Q$ then $S \rhd S_Q$ and $v_S\texttt{[}i\texttt{]} \rhd v_{S_Q}\texttt{[}i\texttt{]}$ for statements S in P and variables $v_S\texttt{[}i\texttt{]}$ in S.

Theorem 1 *Let us apply an operation-preserving transformation to a terminating deterministic program P creating a new program Q. If the transformation preserves the effective reaching definitions of P, then $P \overset{pgm}{\equiv} Q$.*

Proof: Assume $P \overset{pgm}{\not\equiv} Q$. Then, by Definition 1, there must be either a difference in the values of result variables, a different value sent on an output channel, or guard conditions evaluating differently and changing the number of communications executed on an input or output channel. Within the constraints of operation-preserving transformations, these scenarios can only be caused by the existence of at least one assignment S in the program (whether to a regular variable or to an output channel) whose result has changed because of the transformation from P to Q. Let the first such assignment be to some variable x in P.

Now, the transformation cannot change any operations in P, and all of the statements are deterministic. Therefore, S can only assign a different value than S_Q if the values of the variables used to compute x have changed. Let $v_S\texttt{[}i\texttt{]}$ represent the variables used to compute x, and let $T \equiv \mathbf{RD}(v_S[i], S)$ in P. Since the transformation preserves effective reaching definitions, $T_Q \equiv \mathbf{RD}(v_{S_Q}[i], S_Q)$ or $T_Q \equiv \mathbf{RD}_{\mathit{eff}}(v_{S_Q}[i], S_Q)$. But, using the same logic as above, T can only assign different values to $v_S\texttt{[}i\texttt{]}$ than T_Q assigns to $v_{S_Q}\texttt{[}i\texttt{]}$ if the input values to *these* assignments differ.

By induction, $P \overset{pgm}{\not\equiv} Q$ only if the initialization assignments or the environment sources attached to the programs' input channels also differ. But the transformation is prohibited from making such changes. Therefore, all assignments S in P write the same values as assignments S_Q in Q and we must have $P \overset{pgm}{\equiv} Q$, contradicting the initial assumption. $\Box$

We will use Theorem 1 to prove the correctness of DSA transformations on terminating fragments of CHP code. The results will also be used later to build correctness proofs for transformations on non-terminating code.

2.3 Dynamic Single Assignment

This section presents the first major phase of DDD: transforming the original code to a sequential program in dynamic single-assignment (DSA) form. Given a program whose main body (disregarding initialization statements) is enclosed in a non-terminating loop, the program is in DSA form if at most one assignment is executed per variable per iteration. We first provide the motivation for such a transformation, and then illustrate how it can be applied to CHP programs by studying the language's three basic control structures: straightline code, selection statements, and repetition statements.

2.3.1 Motivation

Data-driven decomposition is concerned with exposing concurrency in an algorithm. It therefore relies on data dependencies rather than syntactic constraints often included by designers of sequential programs. One of these constraints is the use of the same variable name for variables that are actually unrelated. For example, the program

$$x := 0;\ \ z := f(x);\ \ A?x;\ \ Y!x$$

can be transformed into

$$x := 0;\ \ z := f(x)\ \ \|\ \ A?y;\ \ Y!y$$

without changing its semantics. In general, when there are multiple assignments to the same variable in a program, different instances of the variable are considered unrelated if they have different reaching definitions, or the reaching definition of the first instance is *killed* before the second instance.

Definition 5 [kill] *A statement S kills the definition of statement T when*

$$(S.LHS \equiv T.LHS) \text{ and } (\mathbf{RD}(T.LHS, S) \equiv T)$$

□

Let us rewrite a program so that every time the definition of a variable is killed, the LHS of the killing assignment and all future appearances of the variable in the code are renamed. There is no reordering involved, and this does not change any reaching definitions in the program, but unrelated variable instances have now been decoupled.

The advantage of such a renaming is that in removing unnecessary syntactic dependencies, it helps identify situations where statements can be reordered correctly, adding concurrency to the programming. Another advantage of the renaming is that if it is used in DDD, when each variable is projected into its own process, the processes now truly do execute only one assignment per iteration of the outer loop. This relieves some concerns about generating a decomposed network where the processes are still too large for post-DDD synthesis. (Chapter 3 explains how processes that execute more than one assignment per iteration require additional circuitry when using the standard compilation template.) This chapter describes the DSA transformation, which is based on renaming.

Definition 6 [DSA form] *Given a terminating program P, P is in DSA form when, for every assignment statement S in P,*

$$\mathbf{RD}(S.LHS, S) \equiv \perp$$

If P is a non-terminating program, it is in DSA form when both P_{init} and P_{loop} are individually in DSA form. □

2.3.2 DSA Indices

First, we introduce some terminology. The first phase of DDD transforms a program P into an equivalent program in DSA form, called P_{DSA}. Part of this transformation involves "splitting" variables x from the original program P by replacing them with multiple new variables x_n for P_{DSA}, each with a unique integer subscript n. The new variables are called *DSA variables.* The subscript of a DSA variable x_n is called its *DSA index.*

Even when DDD splits original variables into separate DSA variables, we maintain a connection between them, purely to reason about DSA transforms.

Definition 7 [DSA reaching definition] *Consider a program P containing the variable x. If $P \rhd P_{DSA}$ and $x \rhd X_{DSA}$ (where $X_{DSA} \equiv \{x_1, x_2, \ldots\}$), then the DSA reaching definition of x for any statement S in P_{DSA} is as follows:*

$$\mathbf{RD}_{DSA}(x, S) \equiv \max_{\prec} \{T : T \prec S \ \cap \ T.LHS \in X_{DSA}\}$$

If there are no assignments to $x_i \in X_{DSA}$ that precede statement S, then $\mathbf{RD}_{DSA}(x, S) \equiv \perp$. $\square$

We call the DSA index of $(\mathbf{RD}_{DSA}(x, S))$.LHS the *reaching DSA index* of x in S. If $\mathbf{RD}_{DSA}(x, S) \equiv \perp$ then the reaching DSA index is 0. Now, we also present the following definitions:

Definition 8 [initial DSA index] *Given a variable x and a fragment of CHP code C from a program P, let $N_0^C[x]$ be the reaching DSA index of x immediately before C_{DSA} in program P_{DSA}. This value is also called the initial DSA index of x in C.* $\square$

Definition 9 [final DSA index] *Given a variable x and a fragment of CHP code C from a program P, let $N_\infty^C[x]$ be the reaching DSA index of x immediately after C_{DSA} in program P_{DSA}. This value is also called the final DSA index of x in C.* $\square$

For example, given the CHP

$$\begin{aligned} P &\equiv A?a;\ \ S;\ \ X!a \\ S &\equiv X!a;\ \ B?b;\ \ a := a + b \end{aligned}$$

DSA conversion produces

$$\begin{aligned} P_{DSA} &\equiv A?a_1;\ \ S_{DSA};\ \ X!a_2 \\ S_{DSA} &\equiv X!a_1;\ \ B?b;\ \ a_2 := a_1 + b \end{aligned}$$

with initial and final DSA reaching indices for S_{DSA}:

$$\begin{aligned} N_0^{S_{DSA}}\texttt{[}a\texttt{]} &= 1 \\ N_\infty^{S_{DSA}}\texttt{[}a\texttt{]} &= 2 \end{aligned}$$

These concepts are helpful in developing methods to transform CHP programs into DSA form. The rest of this section illustrates how the DSA transformation can be applied in general to the three main control structures of CHP, and collectively to all deterministic CHP programs.

2.3.3 Straightline Code

Straightline code contains no selection or repetition statements but rather only basic CHP statements separated by either sequential or parallel operators. For such code to be in DSA form, every assignment in the text must have a different variable on its LHS. If a variable x has multiple assignments in the code, the DSA transformation splits it into multiple new DSA variables x_n, each with unique integer subscripts n.

Consider general straightline code S. For every variable x in S, let us define an indexing function

$$n(i) = N_0^S[x] + i$$

The code can be converted to DSA form ($S \triangleright S_{DSA}$) using the following method:

1. For all variables x in S, rename the variable on the LHS of the i^{th} assignment to x "$x_{n(i)}$."
2. For all statements *reached* by the i^{th} assignment to x in S, rename all instances of x on the RHS "$x_{n(i)}$." (For any statements in S that use x before it has been defined, replace x with $x_{n(0)}$.)

An example of the application of this algorithm (with $N_0^S\texttt{[}a\texttt{]} = 0$) is

$$S \equiv A?a;\;\; b := a,\;\; c := \neg a;\;\; a := a+1,\;\; D!a;\;\; a := 1;\;\; d := f(a)$$

$$S_{DSA} \equiv A?a_1;\;\; b := a_1,\;\; c := \neg a_1;\;\; a_2 := a_1+1,\;\; D!a_2;\;\; a_3 := 1;\;\; d := f(a_3)$$

(Note that if only one assignment to a variable x appears in the original code, that variable does not need to actually be renamed x_1.)

Theorem 2 *Let S be deterministic straightline code. If $S \rhd S_{DSA}$ as described above, then $S \overset{pgm}{\equiv} S_{DSA}$ and S_{DSA} is in DSA form.*

Proof: S is a terminating program. The method in question alters programs only by renaming variables within the code. It is therefore a operation-preserving transformation. By construction (step 2), whenever a variable x on the LHS of an assignment is renamed, all instances of x used in statements reached by that definition are also renamed to match. Also by construction, whenever x is used in S before it has been defined by S, its DSA variable uses the initial DSA index. Thus the method preserves all reaching definitions from S and so, by Theorem 1, $S_{DSA} \overset{pgm}{\equiv} S$.

It remains to show that S_{DSA} fulfills the property required of a DSA program in Definition 6. For every assignment to a variable x in S, the method replaces x on the LHS with a new variable x_n with a unique index n. (The index is unique because a deterministic program always maintains a strict order between assignments to the same variable.) Since each assignment has a unique variable on its LHS, $\mathbf{RD}(A.LHS, A) \equiv \perp$ for all assignments A in S_{DSA}. Therefore S_{DSA} is indeed a DSA program. $\Box$

2.3.4 Selection Statements

Selection statements contain different branches of control, of which only one is executed per program iteration. The command of each branch (guarded command) can be straightline code containing numerous assignments, or even more general code that can also include nested selections or loops. Recalling Definition (6) for DSA programs, a selection in a DSA program can therefore contain multiple assignments to the same variable, as long as no more than one assignment appears in each guarded command.

Let us consider only selection statements with straightline code as commands for now, and build up the more general solution later in Section 2.3.7. For the i^{th} branch in a selection G, we label the guard condition G_i and the command (straightline code) S_i. Our method for converting a stand-alone selection statement into DSA form is to apply the straightline code DSA transformation independently to every command S_i in G. When this method is applied to

$$G \;\equiv\; [\; G_1 \longrightarrow \quad S_1 \;\;[\!]\;\; \ldots \;\;[\!]\;\; G_i \longrightarrow \quad S_i \;\;[\!]\;\; \ldots \;]$$

the resulting structure is

$$G_{DSA} \;\equiv\; [\; G_1 \longrightarrow \;(S_1)_{DSA} \;\;[\!]\;\; \ldots \;\;[\!]\;\; G_i \longrightarrow \;(S_i)_{DSA} \;\;[\!]\;\; \ldots \;]$$

If the selection does stand alone then this transformation is in fact enough. However, selections generally appear in the middle of a larger series of statements; variables used in a selection can have initial values assigned prior to the selection and can also be used in other computations after the selection. DDD must therefore take additional steps to ensure that the entire DSA program is correct.

Throughout the rest of this section, we will illustrate different steps referring to the CHP programs P_1, P_2 and P_3 from Figure 2.3. We label the selection statement in each program "G." The code may not be that efficient from a programming perspective, but it serves well as a simple example for DSA transformation!

2.3.4.1 DSA Indices from Pre-Selection Code

When transforming a selection statement G to DSA form, we must replace all variables v that appear in the guard conditions of G with the DSA variables $v_{N_0^G[v]}$. This makes the indexing of new DSA variables in the selection statement consistent with DSA indices used in any pre-selection code.

In Figure 2.3, this transformation is among those applied to P_1 resulting in new selection statement P_2. The initial DSA indices used for variables in G are

$$N_0^G[a] = N_0^G[b] = N_0^G[c] = 1$$

```
P₁ ≡ *[ $A?a$,  $B?b$,  $C?c$;
        [ $a \wedge b \wedge c \longrightarrow$  $C?c$;  $D!c$;  $c := 5$;  $f := 2*c$;  $F!f$
        [] $a \wedge \neg b \longrightarrow$  $G?c$;  $D!c$
        [] **else** $\longrightarrow$  **skip**
        ];
        $X!a$,  $Y!c$
      ]

P₂ ≡ *[ $A?a_1$,  $B?b_1$,  $C?c_1$;
        [ $a_1 \wedge b_1 \wedge c_1 \longrightarrow$  $C?c_2$;  $D!c_2$;  $c_3 := 5$;  $f_1 := 2*c_3$;  $F!f_1$
        [] $a_1 \wedge \neg b_1 \longrightarrow$  $G?c_2$;  $D!c_2$
        [] **else** $\longrightarrow$  **skip**
        ];
        ...
      ]

P₃ ≡ *[ $A?a_1$,  $B?b_1$,  $C?c_1$;
        [ $a_1 \wedge b_1 \wedge c_1 \longrightarrow$  $C?c_2$;  $D!c_2$;  $c_3 := 5$;  $f_1 := 2*c_3$;  $F!f_1$
        [] $a_1 \wedge \neg b_1 \longrightarrow$  $G?c_2$;  $D!c_2$;  $c_3 := c_2$
        [] **else** $\longrightarrow$  $c_3 := c_1$
        ];
        $X!a_1$,  $Y!c_3$
      ]
```

Figure 2.3: CHP selection statement and its transformation to DSA form.

$$N_0^G[f] = 0$$

Thus, a, b and c have all been assigned values prior to the selection statement G, but variable f has not. Any variable that appears in a guard condition of G must have a DSA index greater than 0 when the program enters the selection. Note that since the guard conditions cannot contain any assignments to variables, $N_0^G[v] = N_0^{S_i}[v]$ for every variable v and every command S_i in the selection G.

2.3.4.2 DSA Indices for Post-Selection Code

It now remains to ensure that the DSA indices chosen for the selection statement are consistent with those used for variables in code that follows the selection statement. Let us suppose that a variable v appears in multiple branches of the selection statement and is assigned to a different number of times in each one. Since each guarded command is converted independently to DSA form, the reaching definition of original variable v may be different at the end of different commands. For example, in P_2, the reaching definition of original variable c is now "$c_3 := 5$" at the end of the first control branch, "$G?c_2$" at the end of the second branch, and the pre-selection assignment "$C?c_1$" at the end of the third. What new variable name—c_1, c_2 or c_3—should be used to replace c in post-selection statements?

To determine the answer, we must either carry information about the guarded conditions outside of the selection statement, or join all of the different definitions of v from the different commands back into a single variable definition for a single thread of control. (The terminology used here is deliberately similar to that of join nodes, or ϕ-nodes, used in static single-assignment compiler analysis [12]. The techniques used to merge multiple threads of control back into one are different, however.)

We choose the second option, and appoint the new DSA variable with the largest index to hold the definition that reaches post-selection statements. In other words, for a selection statement G with commands S_i,

$$N_\infty^G[v] = \max_i N_\infty^{S_i}[v] \tag{2.1}$$

The DSA index of v immediately following the selection statement G is $N_\infty^G[v]$, and all statements outside of G whose reaching definition of v is in G should use $v_{N_\infty^G\texttt{[}v\texttt{]}}$ on the RHS of their assignments.

Thus, the final DSA indices for variables in G in P_2 are

$$N_\infty^G[c] = 3$$

$$N_\infty^G[a] = N_\infty^G[f] = 1$$

Note that since a is not assigned any values in G, $N_0^G\texttt{[}a\texttt{]} = N_\infty^G\texttt{[}a\texttt{]}$. As it turns out, we can ignore $N_\infty^G[f]$ since in our example's original program P_1, f is never used after the selection statement.

For the variables that *are* used after the selection, we must ensure that $v_{N_\infty^G\texttt{[}v\texttt{]}}$ is always defined at the end of G! DDD accomplishes this using the following method:

1. For every command S_i and every variable v assigned a value in G, if $N_\infty^{S_i}[v] < N_\infty^G[v]$ then

insert the copy statement $v_{N_\infty^G[v]} := v_{N_\infty^{S_i}[v]}$ to the end of the command

2. For every statement S and every variable v where $G \prec S$ and $\mathbf{RD}(v, S)$ is in G, change all instances of v on the RHS of S into $v_{N_\infty^G\texttt{[}v\texttt{]}}$

The DSA indices inside selection statement G are now consistent with the selection's surrounding code. When the above method is applied to P_2, the resulting process is P_3 in Figure 2.3.

2.3.4.3 Proof of Correctness

Theorem 3 *Let G be a selection statement in a program P. If $P \triangleright P_{DSA}$ and $G \triangleright G_{DSA}$ as described above, then $G \overset{pgm}{\equiv} G_{DSA}$ and G_{DSA} is in DSA form.*

Proof: The transformation in this section is limited to variable renaming and copy propagation at the end of guarded commands. It is therefore an operation-preserving transformation, and we use Theorem 1 to prove program equivalence by proving the preservation of effective reaching definitions from P to P_{DSA}.

Since only one command S_i in G is ever executed at a time, the application of the DSA straightline transformation to S_i guarantees both that $(S_i)_{DSA}$ is in DSA form and that $S_i \overset{pgm}{\equiv} (S_i)_{DSA}$ by Theorem 2.

DDD replaces variables v that are used in guards G_i with $v_{N_0^G\texttt{[}v\texttt{]}}$. By Definition 8, $N_0^G\texttt{[}v\texttt{]}$ is the reaching DSA index of v at the beginning of G, so $\mathbf{RD}(v, G_i) \equiv \mathbf{RD}(v_{N_0^G\texttt{[}v\texttt{]}}, (G_i)_{DSA})$. Reaching definitions in the guards are therefore preserved. Since there are no assignments in guards, the replacement of v with $v_{N_0^G\texttt{[}v\texttt{]}}$ does not affect the DSA form of the selection.

It remains to consider any copy statements inserted at the end of S_i. Any variable $v_{N_\infty^G\texttt{[}v\texttt{]}}$ on the LHS of these statements cannot have been defined before in S_i (otherwise $N_\infty^{S_i} \geq N_\infty^G$ and the assignment would not exist), so this insertion leaves G_{DSA} in DSA form. The new assignment kills the last definition of v in S_i, but is merely a copy statement that propagates the value of that definition, and thus leaves the effective reaching definition intact. The inserted assignments therefore do not change the semantics of the selection, and $G_{DSA} \overset{pgm}{\equiv} G$. □

2.3.5 Repetition Statements

While it is not necessary for the correctness of the method, we assume that all sequential programs P to which DDD is applied are non-terminating. Any statements in P_{init} can be considered the reset protocol of the program; since the program's loop is non-terminating, we do not consider any statements that follow it. Nested non-terminating loops are nonsensical so we also ignore such programs. Hence, when we speak of applying a DSA transformation to repetition statements in CHP, we are referring to either a loop with only terminating programs inside, or nested loops where the inner loop is conditional.

We begin by demonstrating how DDD converts loops with terminating bodies into DSA form, and then how DDD handles nested loops.

2.3.5.1 Loops with Terminating Bodies

Consider a program

$$P \;\equiv\; A;\; *[\; G \longrightarrow \; S \;];\; Z$$

where A, S and Z are all terminating programs that do not contain any repetition statements. Traditional "non-terminating programs" fall into this class (their guard condition $G \equiv \textbf{true}$, making Z superfluous). Note that conditional loops with multiple branches

$$*[\; G_1 \longrightarrow \; S_1 \;[]\; G_2 \longrightarrow \; S_2 \;[]\; \ldots \;[]\; G_N \longrightarrow \; S_N \;]$$

can also be rewritten to fit into this class of programs:

$$*[\; G_1 \vee G_2 \vee \ldots \vee G_N \longrightarrow \; [G_1 \longrightarrow \; S_1 \;[] \ldots []\, G_N \longrightarrow \; S_N] \;]$$

Let x be a variable that is defined in A and has multiple definitions in S. The presence of a definition in A implies that x is used in S before it is defined in S. If x is split when $S \rhd S_{DSA}$, the DSA variable that is used at the beginning will have a different DSA index from the DSA variable defined at the end. The program will therefore be incorrect. For example, if

$$x := x_{init};\; *[\; X!x;\; C?x;\; Y!x;\; D?x \;]$$

is transformed into

$$x_0 := x_{init};\ \ *[\ \ X!x_0;\ \ C?x_1;\ \ Y!x_1;\ \ D?x_2\ \ ]$$

then the variable x_0 does not hold the value of x_2 after an iteration but instead always keeps the initial value x_{init}. DDD's solution is to change the name of the variable defined in the initial code and insert a copy statement for DSA variables of x at the beginning of the main loop. The transformed program is therefore

$$x_2 := x_{init};\ \ *[\ \ x_0 := x_2;\ \ X!x_0;\ \ C?x_1;\ \ Y!x_1;\ \ D?x_2\ \ ]$$

Consider a program P containing a loop as above, where x represents all variables used in S before they are defined in S. DDD begins by individually converting $A \triangleright A_{DSA}$, $S \triangleright S_{DSA}$ and $Z \triangleright Z_{DSA}$ such that for all variables x in P, $N_\infty^{A_{DSA}}[x] = N_0^{S_{DSA}}[x]$ and $N_\infty^{S_{DSA}}[x] = N_0^{Z_{DSA}}[x]$. It transforms $G \triangleright G_{DSA}$ by replacing all instances of x in G with DSA variable $x_{N_\infty^{A_{DSA}}[x]}$. The intermediate program that results is

$$A_{DSA};\ \ *[\ \ G_{DSA} \longrightarrow\ \ S_{DSA}\ \ ];\ \ Z_{DSA}$$

(Note that when x is used before being defined in S_{DSA}, its DSA index is $N_\infty^{A_{DSA}}[x]$.) DDD then inserts two copy statements into the code:

$$A_{DSA};\ \ x_{N_\infty^{S_{DSA}}[x]} := x_{N_\infty^{A_{DSA}}[x]};\ \ *[\ \ G_{DSA} \longrightarrow\ \ x_{N_\infty^{A_{DSA}}[x]} := x_{N_\infty^{S_{DSA}}[x]};\ \ S_{DSA}\ \ ];\ \ Z_{DSA}$$

Theorem 4 *Let $P \equiv A; *[G \rightarrow S]; Z$ be a non-terminating program with terminating components A, S, and Z. If $P \triangleright P_{DSA}$ following the DDD method given above, then $P \overset{pgm}{\equiv} P_{DSA}$ and P_{DSA} is in DSA form.*

Proof: We begin by noting that, for all variables x used before being defined in S,

$$N_\infty^{A_{DSA}}[x] = N_0^{S_{DSA}}[x] \leq N_\infty^{S_{DSA}}[x]$$

Therefore $x_{N_\infty^{A_{DSA}}[x]}$ is used but not defined anywhere in the body of S_{DSA}, and $x_{N_\infty^{S_{DSA}}[x]}$ is also not defined anywhere in the body of A_{DSA}, and so the insertion of the two copy statements does not affect the individual DSA form of either S_{DSA} or of A_{DSA}. Therefore, by Definition 6, since the code preceding the loop, the code in the loop body, and the code following the loop are all individually in DSA form, P_{DSA} is in DSA form as well.

Now, since they are all terminating programs, $A \overset{pgm}{\equiv} A_{DSA}$, $S \overset{pgm}{\equiv} S_{DSA}$ and $Z \overset{pgm}{\equiv} Z_{DSA}$ by Theorems 2 and 3. To fulfill program equivalence for non-terminating programs as specified in Definition 2, it remains to show that the names of all variables $x_i \in X_{DSA}$ used in the body of P_{DSA}'s loop before being defined in the loop match the names of variables on the LHS of the l ast assignments to any DSA variable $x_i \in X_{DSA}$ in A_{DSA} and S_{DSA}.

But the first definition to a DSA variable in the loop body is actually the inserted copy assignment at the beginning of the loop. The only DSA variable used before the definition is therefore on the RHS of the definition itself, $x_{N_{\infty}^{S_{DSA}}\texttt{[}x\texttt{]}}$. By definition, this is the DSA variable last assigned a value in S_{DSA}. Because of the copy statement inserted immediately preceding the loop, it is also the last DSA variable assigned a value in A_{DSA}. Therefore, $P \overset{pgm}{\equiv} P_{DSA}$. □

2.3.5.2 Nested Conditional Loops

The case of conditional repetition statements nested within a non-terminating program is illustrated by the following code:

$$P \;\equiv\; A_{init};\; \texttt{*[}\; A;\; \texttt{*[}G \;\longrightarrow\; S\texttt{]};\; Z\; \texttt{]}$$

We present two methods of transforming such loops into DSA form: one straightforward but inefficient, the other more complicated but also more generally applicable.

The first method, transforming the entire program into a state machine, is inefficient in terms of the communications that it will require in the eventual decomposed system. We include it because it is general and straightforward and can be useful in simple cases. The method converts nested loops into selection statements through the addition of a state bit to the process. Then, the DSA transformation for selection statements can be performed to rewrite the entire program in DSA form.

Consider the general program P for repetition statements given above. We introduce a state bit x that is true when the program is executing statements within the loop, and false when the program is on the outside of the loop. With the addition of a large encompassing selection statement, we can now push the repetitive behavior of the inner loop out onto the unconditional main loop.

```
$A_{init}$, $x\downarrow$;
*[ [$\neg x$          $\longrightarrow$ $A$, $x\uparrow$
   [] $x \wedge G$    $\longrightarrow$ $S$
   [] $x \wedge \neg G$ $\longrightarrow$ $Z$, $x\downarrow$
 ] ]
```

While this is a valid transformation, it is not ideal because all program variables now directly depend upon x. When the variables are all split into their own processes by DDD, the value of x will need to be explicitly communicated to all of them. Even when hardware optimizations introduced in Chapter 4 are applied to the decomposed system, the system may be inefficient with its communications. Still, the nested repetitions have been removed, and the program can be converted to DSA form as illustrated in the previous section.

A more efficient way of handling nested loops is to perform an early decomposition of the sequential program. The code within the loop is moved to one process while the code outside of the loop is kept in another. An example of such a solution is

```
$P$       $\equiv$ $A_{init}$; *[ $A$; *[$G \longrightarrow S$]; $Z$ ] $\triangleright$ $P_{out}$ $\|$ $P_{in}$
$P_{out}$ $\equiv$ $A_{init}$; *[ $A$; $AS!$; $SZ?$; $Z$ ]
$P_{in}$  $\equiv$ $x\downarrow$; *[ [$\neg x \longrightarrow AS?$ [] $x \longrightarrow$ **skip**];
                    [ $G \longrightarrow S$, $x\uparrow$
                    [] $\neg G \longrightarrow SZ!$, $x\downarrow$
                  ] ]
```

where AS and SZ are newly- introduced channels.

There is no added concurrency in this decomposition. Synchronizing semicolons in the sequential program have been replaced with synchronizing communications in the decomposed system, and the

```
$P \equiv$ $\mathbf{A_{init}}$; $a_{use} := f_{aa}(a_{init})$;
    *[ $a_{def} :=$ $\mathbf{A}(a_{use})$;
       $g_{use} := f_{ag}(a_{def})$, $s_{use} := f_{as}(a_{def})$, $z_{use} := f_{az}(a_{def})$;
       *[ $\mathbf{G}(g_{use}) \longrightarrow$ $s_{def} :=$ $\mathbf{S}(s_{use})$;
                         $g_{use} := f_{sg}(s_{def})$, $z_{use} := f_{sz}(s_{def})$, $s_{use} := f_{ss}(s_{def})$ ];
       $z_{def} :=$ $\mathbf{Z}(z_{use})$;
       $a_{use} := f_{za}(z_{def})$
    ]

$P \equiv$ $P_{in}$ $\|$ $P_{out}$

$P_{out} \equiv$ $\mathbf{A_{init}}$; $a_{use} := f_{aa}(a_{init})$;
    *[ $a_{def} :=$ $\mathbf{A}(a_{use})$;
       $g_{use} := f_{ag}(a_{def})$, $s_{use} := f_{as}(a_{def})$, $z_{use} := f_{az}(a_{def})$;
       $AS!\{s_{use}, g_{use}\}$; $SZ?z_{use}$;
       $z_{def} :=$ $\mathbf{Z}(z_{use})$;
       $a_{use} := f_{za}(z_{def})$
    ]

$P_{in} \equiv x\downarrow$;
    *[ [ $\neg x \longrightarrow$ $AS?\{s_{use}, g_{use}\}$ ▯ $x \longrightarrow$ **skip** ];
       [ $\mathbf{G}(g_{use}) \longrightarrow$ $s_{def} :=$ $\mathbf{S}(s_{use})$;
                         $g_{use} := f_{sg}(s_{def})$, $z_{use} := f_{sz}(s_{def})$, $s_{use} := f_{ss}(s_{def})$, $x\uparrow$
       ▯ $\neg$ $\mathbf{G}(g_{use}) \longrightarrow$ $SZ!z_{use}$, $x\downarrow$
    ] ]
```

Figure 2.4: DSA transformation of repetition statements.
Example of early decomposition, including data.

two implementations are semantically equivalent. The advantage of this solution over the general state machine is that only short communication statements, and not arbitrarily long series, are contained within selection statements. Thus, less communication will be required between guard variables and decomposed processes in the eventual distributed system. Both processes can now be converted into DSA form and decomposed separately.

We demonstrated the abstract split of computations above, but have not yet included the data communications required between processes. A more realistic representation of the scenario that explicitly includes variables that are used and defined by each CHP series is given in Figure 2.4. In this example, the notation "$v_{def} := \mathbf{V}(v_{use})$" indicates that the CHP series $\mathbf{V}$ uses variables from the set v_{use} and contains assignments to variables from the set v_{def}. The notation "$w_{use} := f_{vw}(v_{def})$" is included to explicit indicate which variables assigned values in the series $\mathbf{V}$ are used in the ensuing series **W**.

The decomposition shown in Figure 2.4 is correct: the two new processes together form the equivalent of the original sequential program. Both new processes (P_{in} and P_{out}) contain only straightline code and selection statements, and so can be converted into DSA form as demonstrated in the previous sections. Now DDD can apply the projection technique of Section 2.4 to both processes separately and then recombine the resulting systems for the clustering heuristic of Chapter 5.

2.3.6 Special Cases

We have not yet mentioned arrays, a special type of variable in CHP. When the original CHP specification describes circuit behavior, arrays often represent special structures such as memories or register files, and their detailed manipulation is left for Chapter 4, which describes DDD hardware optimizations.

When the original program is not describing circuit behavior, we can generally consider an array $X\,\texttt{[}0\ldots N\texttt{]}$ to be a collection of variables $x_0 \ldots x_N$, and replace any array operations with selection statements before applying DDD. Thus a statement "$y := X\,\texttt{[}k\texttt{]}$" can be rewritten as

$$\texttt{[}\;\; k = 0 \longrightarrow \;\; y := x_0 \;\; \square \;\; k = 1 \longrightarrow \;\; y := x_1 \;\; \square \;\; \ldots \;\; \square \;\; k = N \longrightarrow \;\; y := x_N \;\; \texttt{]}$$

and a statement "$X\,\texttt{[}k\texttt{]} := y$" can be rewritten as

$$\texttt{[}\;\; k = 0 \longrightarrow \;\; x_0 := y \;\; \square \;\; k = 1 \longrightarrow \;\; x_1 := y \;\; \square \;\; \ldots \;\; \square \;\; k = N \longrightarrow \;\; x_N := y \;\; \texttt{]}$$

2.3.7 Putting It Together

When converting general deterministic CHP programs into DSA form, the individual transformations demonstrated for various control structures can be combined as follows. First, remove any nested loops from the program using the techniques given in Section 2.3.5. Then the only structures left in the processes are CHP series that combine selection statements with straightline code. Sections 2.3.3 and 2.3.4 showed how to transform these structures into DSA form. Since the methods for both structures depend on the same concepts of initial and final DSA indices for variables in the structure

($N_0^S[v]$ and$N_\infty^S[v]$), they can be used together when dealing with general code that combines and even nests straightline code and selection statements.

Suppose straightline code S appears in parallel with a selection statement G. The situation where variables are assigned values in both S and G never arises because it would be an instance of nondeterminism in the original code. Thus, all deterministic CHP programs can be converted to DSA form.

2.4 Using the Projection Technique

Now that we have converted an original sequential process to DSA form, we can begin breaking it into target processes. The basic units of our decomposition are assignment statements, and the basic tool is the method of *projection* [34]. Projection is a decomposition technique in which a CHP process is syntactically projected onto disjoint sets of its variables. The resulting *images* are the new processes that together form a concurrent system which is functionally equivalent to the source sequential process. DDD directs the projection technique by manipulating a program's syntax to achieve the semantically desired results.

A simple example of partitioning by projection begins with the process

$$P \equiv *[\ A?a;\ \ B?b;\ \ X!f(a);\ \ Y!g(b)\ ]$$

When P is projected onto the two sets $\{A?, a, X!\}$ and $\{B?, b, Y!\}$ the resulting system is $P \rhd P1 \parallel P2$, where

$$P1 \equiv *[\ A?a;\ \ X!f(a)\ ]$$

$$P2 \equiv *[\ B?b;\ \ Y!g(b)\ ]$$

All synchronization between the two computations has been removed by this transformation. The correctness of projection has been proved under the assumption of *slack elasticity* [35]. In a CHP system, the *slack* of a communication channel specifies the maximum of outstanding messages (the amount of buffering) allowed on that channel. A system is slack elastic if its correctness is preserved when the slack on its channels is increased. An open system S is *locally slack elastic* if, when composed in parallel with an environment such that the new system is closed, adding slack to any channel in the closed system introduces no non-determinism to S. All deterministic programs are locally slack elastic and so, according to Manohar [34], projection performed on deterministic programs is correct.

Historically, projection has been used to verify the equivalence between sequential processes and systems that have been decomposed by hand. Projection is a tool; until now, no guiding framework has been provided demonstrating effective and general ways to apply the tool to process decomposition. Where in the past designers have relied on experience and intuition in choosing the different variable sets for projection, DDD uses data-dependency analysis to specify sets of variables for every process in the projected concurrent system. DDD's methods are described in Sections 2.4.1-2.4.4.

2.4.1 Dependency Sets

In projection, a process's variables (including communication channels) are partitioned into disjoint *projection sets.* The original process is then decomposed into a system containing one new process per set. For example, consider the process

$$P \;\equiv\; *[A?a, B?b, C?c;\;\; X!(a+b);\;\; [c \longrightarrow D?d;\;\; y := c+d \;\;[\!]\neg c \longrightarrow \textbf{skip}]]$$

The variables of P can be partitioned into two projection sets: $\{A?, a, B?, b, X!\}$ and $\{C?, c, D?, d, y\}$. When the original process is projected onto these sets, the resulting system is

$$P \;\triangleright\; P_x \;\;\|\;\; P_y$$

$$P_x \;\equiv\; *[\;\; A?a,\;\; B?b;\;\; X!(a+b)\;\;]$$

$$P_y \;\equiv\; *[\;\; C?c;\;\; [c \longrightarrow ?d;\;\; y := c+d \;\;[\!]\neg c \longrightarrow \textbf{skip}]\;\;]$$

There are rules about how variables can be grouped into projection sets. If instead of P, the program from the previous example had been Q_1 from Figure 2.5, we would not have been able to split $X!$ and y into different projection sets since both variables use b in their assignments. In this case, a copy of b would have needed to be inserted into the program before projection, as in the

$$
\begin{array}{ll}
Q_1 \equiv & *[\ A?a,\ \ B?b,\ \ C?c;\ \ X!(a+b); \\
& \quad [\ c \longrightarrow \ \ D?d;\ \ y := b + c + d \ \ [] \ \neg c \longrightarrow \ \ \textbf{skip}\ ]\ ]
\end{array}
$$

$$
\begin{array}{ll}
Q_2 \equiv & *[\ A?a,\ \ B?b,\ \ C?c;\ \ b_y := b;\ \ X!(a+b); \\
& \quad [\ c \longrightarrow \ \ D?d;\ \ y := b_y + c + d \ \ [] \ \neg c \longrightarrow \ \ \textbf{skip}\ ]\ ]
\end{array}
$$

Figure 2.5: Example CHP processes for projection.

rewritten process Q_2. Now, $X!$ and y could have been split into different processes using completely disjoint projection sets.

DDD creates one process for every variable in the DSA sequential program, and so projection sets are based on the *dependency sets* of each variable. For a variable v, its dependency set $DS(v)$ consists of all of the variables used in its assignment. The only variables that do not have dependency sets (and thus do not have their own processes in the decomposed system) are input-communication channels. The dependency sets for all of the variables in the program Q_2 are listed below:

$$
\begin{array}{lcl}
DS(a) & \equiv & \{A?\} \\
DS(b) & \equiv & \{B?\} \\
DS(c) & \equiv & \{C?\} \\
DS(b_y) & \equiv & \{b\} \\
DS(X!) & \equiv & \{a, b\} \\
DS(d) & \equiv & \{c, D?\} \\
DS(y) & \equiv & \{b_y, c, d\}
\end{array}
$$

2.4.2 Copy Variable and System Channel Insertion

A variable's dependency set forms the core of its projection set in DDD, but other syntactic adjustments are required to make the transformation formally correct. For example, if a variable appears in the dependency sets of several other variables, copy variables need to be inserted in the sequential code. And if projection is used to pipeline the program, regular assignments such as "$x := a$" need to be replaced by distributed assignments: parallel communication statements on newly created system channels, as in "$X_A!a, X_A?x$." This section describes both of these steps.

2.4.2.1 Inserting Copy Variables

If a variable appears in the dependency sets of several other variables, DDD inserts new *copy variables* into the program that can each be projected out into a different process. In general, if a variable v appears in the dependency sets of variables a_i, then DDD creates a new copy variable v_{cp} and inserts the assignment $v_{cp} := v$ immediately after the original assignment to v. (During projection, the creation of this variable creates a separate copy process for v.) DDD then continues by inserting other assignments of the form $v_{a_i} := v_{cp}$ for each a_i after the assignment to v_{cp}. Finally, to maintain program correctness, DDD replaces each instance of v on the RHS of the assignment to a_i with the new copy variable v_{a_i}.

For example, in the CHP example given below, DDD inserts copy variables a_{cp}, a_x and a_B to transform the program R_1 into R_2 before assigning projection sets.

$$R_1 \equiv *[\ A?a;\ \ x := a + 1,\ \ B!a;$$
$$[a > 0 \longrightarrow\ \ Z!f(x)\ \ [\!]\ \ \textbf{else} \longrightarrow\ \ \textbf{skip}\ \ ]\ \ ]$$

$$R_2 \equiv *[\ A?a;\ \ a_{cp} := a; a_x := a_{cp}, a_B := a_{cp}, a_Z := a_{cp};\ \ x := a_x + 1,\ \ B!a_B;$$
$$[\ \ a_Z > 0 \longrightarrow\ \ Z!f(x)\ \ [\!]\ \ \textbf{else} \longrightarrow\ \ \textbf{skip}\ \ ]\ \ ]$$

Consider a slightly more complicated scenario where a variable v appears in a guard condition for a selection statement command containing multiple assignments to different variables. A copy of v can be inserted for each of the assignments, but now they must all be incorporated in the guard condition of the rewritten program. DDD accomplishes this by replacing all instances of "v" in the guard condition with a conjunction of the new copy variables, and all instances of "$\neg v$" with a conjunction of the inverses of the new copy variables. Thus, the process R_3 in the example below can be rewritten as R_4.

$$R_3 \equiv *[\ \ ...\ \ v := f(w, x);\ \ ...$$
$$[\ \ v \longrightarrow\ \ A?a,\ \ b := f(...)\ \ [\!]\ \ \neg v \longrightarrow\ \ \textbf{skip}\ \ ]\ \ ...\ \ ]$$

$$R_4 \equiv *[\; \ldots\; v := f(w,x);\;\; v_{cp} := v; v_a := v_{cp}, v_b := v_{cp}; \ldots$$
$$[\; v_a \wedge v_b \longrightarrow \;\; A?a,\;\; b := f(\ldots)\; [\!] \;\neg v_a \wedge \neg v_b \longrightarrow \;\; \textbf{skip}\;]\; \ldots\;]$$

Thus far, we have been considering communication channels to be the equivalent of regular variables in DDD. There are practical differences that must be acknowledged by our methods. While DDD can copy regular variables to create coherent projection sets, input-communication channels cannot be “copied” or “split.” Instead, DDD rewrites the code so that whenever an input channel is used, the value read in is always stored in the same variable. If this is not the case in the original code, DDD introduces a new *input variable* to the code for this purpose, and then inserts assignments from this new variable to the original variables immediately afterwards. (This may cause the program to no longer be in DSA form.) The input channel therefore does not directly appear in the dependency or projection sets of any variable other than the new input variable, and no split needs to be considered. In the example below, the process R_5 is rewritten as R_6:

$$R_5 \equiv *[\;\; V?a;\;\; x := f(v)\; \ldots\; ;$$
$$V?b,\;\; W?c;\;\; Z!(b+c)\;]$$

$$R_6 \equiv *[\;\; V?v_{in};\;\; a := v_{in};\;\; x := f(a)\; \ldots\; ;$$
$$V?v_{in},\;\; W?c;\;\; b := v_{in};\;\; Z!(b+c)\;]$$

Once the appropriate copy and input variables have been inserted into the sequential program, DDD rewrites all of the dependency sets.

Note that almost all transformations described in this section amount to copy propagation accompanied by variable renaming. (Variable renaming preserves effective reaching definitions.) The sole exception is the transformation that replaces variables in guard expressions with conjunctions of copies of themselves. In this last transformation, if $x = x_1 = x_2$, then $x_1 \wedge x_2 \equiv x$ and $\neg x_1 \wedge \neg x_2 \equiv \neg x$ are always true, so the guard replacement never alters the computations performed by the program. Therefore, the transformations described here fall into the class of operation-preserving transformations and since they preserve effective reaching definitions, by Theorem 1, the new program is equivalent to the original.

```
$Q_3 \equiv$ *[ $A?a,\ B?b,\ C?c;$
          $B_{cp}!b,\ B_{cp}?b_{cp},\ C_{cp}!c,\ C_{cp}?c_{cp};$
          $A_X!a,\ A_X?a_x,$
          $B_X!b_{cp},\ B_X?b_X,\ B_y!b_{cp},\ B_y?b_y,$
          $C_d!c_{cp},\ C_d?c_d,\ C_y!c_{cp},\ C_y?c_y;$
          $X!(a_X + b_X);$
          [$c_d \wedge c_y \longrightarrow\ D?d;\ D_y!d,\ D_y?d_y;\ y := b_y + c_y + d_y$
          ▯$\neg c_d \wedge \neg c_y \longrightarrow$ skip
        ] ]
```

Figure 2.6: Rewriting Q_1 to create disjoint projection sets.

2.4.2.2 Internal Communication Channel Insertion

For projection to be formally correct, if a decomposed process includes a communications statement then the statement must exist in the original sequential code. Given an original program where variable b depends directly on variable a, DDD must perform projection so that the value computed in the decomposed process for a is explicitly communicated to the new process for b. This is accomplished by inserting the required communication statements into the sequential program before projection.

If an assignment of the form "$a_b := a_{cp}$" has already been inserted into the program, then DDD can simply rewrite this assignment as "$A_b!a_{cp}, A_b?a_b$," where A_b is a new communication channel created for use internally within the decomposed system. Both the regular assignment and the distributed assignment assign the value of a to a_b.

If, on the other hand, $DS(b)$ is the only dependency set in which the variable a appears, then DDD creates a dummy variable a_b, inserts the concurrent communications statement (as above) into the program immediately following the assignment to a, and replaces a on the RHS of the assignment to b with a_b. Figure 2.6 demonstrates the insertion of internal communications by rewriting the process Q_1 from Figure 2.5 and creating the projection channels A_X, B_{cp}, B_X, B_y, C_{cp}, C_d, C_y, and D_y.

Again, all transformations in this section are copy propagation transformations accompanied by variable renaming to preserve effective reaching definitions. (Rewriting regular assignments as communication statements can be considered simply another form of copy propagation, using the new channel as a temporary variable.) By Theorem 1 then, the resulting programs are equivalent to the old.

2.4.3 Performing Projection

Now the dependency sets for all non-copy variables are disjoint. DDD forms the projection sets $PS(v)$ for each variable v from the original program in the following manner:

1. Include the variable v itself in $PS(v)$.
2. For every variable a in $DS(v)$
   - include a in $PS(v)$, and
   - include internal input channel port $A_v?$ in $PS(v)$.
3. For every variable z in whose dependency set v_z belongs, include the internal output channel port $V_z!$ in $PS(v)$.

Note that for all communication channels C, the input port ($C?$) and output port ($C!$) are treated as two separate variables. With the projection sets thus created, DDD can project the sequential program onto each set.

We now prove the correctness of the projection phase of DDD. The transformation is considered correct when the communication traces on each channel from the original sequential program are identical to the communication traces on the same channels in the decomposed system.

Theorem 5 *After applying DDD's projection phase to a deterministic program, the final concurrent system is equivalent to the original sequential program.*

Proof: (Sketch) We have already demonstrated that every sequential transformation (copy variable insertion, system channel insertion) performed on the original program results in new sequential programs that are program equivalent to the old. It remains to prove that the final step, transforming the sequential program into a concurrent system, is correct.

The sequential program is deterministic, and therefore locally slack elastic. It has been proved elsewhere that when projection is correctly applied to a locally slack elastic program, the projected system is a valid implementation of the original program [34]. The correct application of projection entails using projection sets that are both *complete* (i.e, the union of all projection sets includes every variable and channel in the sequential program) and *disjoint* (i.e., no item appears in more than one projection set). The items included in projection sets are regular variables, original input channels, original output channels, and internal channels (both input and output).

First, consider the completeness of the projection sets. All regular variables receive their own projection set except for those that are assigned copy variables and are generated specifically for other variables to depend on and therefore include in their projection sets. All original output-channel ports also receive their own projection set. Original input-channel ports are always attached to a single input variable and included in their projection set. Finally, all internal-channel ports are also linked to variables in the program and included in their projection sets. The projection sets are therefore complete.

Now, any variables that appear in multiple dependency sets are split into new copy variables specifically to keep projection sets disjoint. Original output-channel ports appear in their own individual projection sets only and if original input-channel ports are used for different variables in the original program, special input variables are inserted so that they need only appear in a single projection set. Finally, internal channels are created to communicate data between two variables only, so their ports appear only in one projection set each. Therefore, every variable and channel port appears in only one projection set and the projection sets created by DDD are disjoint.

The partition designed by DDD is therefore both complete and disjoint, and the projection is correct. □

Continuing the example from Figures 2.5 and 2.6, the projection sets of the variables in Q_3 are as follows:

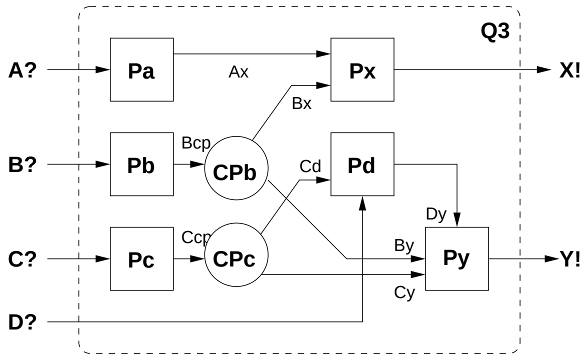


Figure 2.7: Final projection of example program Q_3.

$$PS(a) \equiv \{\ A?,\ a,\ A_X!\ \}$$

$$PS(b) \equiv \{\ B?,\ b,\ B_{cp}!\ \}$$

$$PS(b_{cp}) \equiv \{\ B_{cp}?,\ b,\ B_X!,\ B_y!\ \}$$

$$PS(c) \equiv \{\ C?,\ c,\ C_{cp}!\ \}$$

$$PS(c_{cp}) \equiv \{\ C_{cp}?,\ c,\ C_d!,\ C_y!\ \}$$

$$PS(X) \equiv \{\ a_X,\ A_X?,\ b_X,\ B_X?,\ X!\ \}$$

$$PS(d) \equiv \{\ c_d,\ C_d?,\ D?,\ d,\ D_y!\ \}$$

$$PS(y) \equiv \{\ b_y,\ B_y?,\ c_y,\ C_y?,\ d_y,\ D_y?,\ y\ \}$$

When projection is applied to process Q_3, the resulting system is as follows:

$$P_a \equiv *[\ A?a;\ A_X!a\ ]$$

$$P_b \equiv *[\ B?b;\ B_{cp}!b\ ]$$

$$CP_b \equiv *[\ B_{cp}?b;\ B_X!b,\ B_y!b\ ]$$

$$P_c \equiv *[\ C?c;\ C_{cp}!c\ ]$$

$$CP_c \equiv *[\ C_{cp}?c;\ C_d!c,\ C_y!c\ ]$$

$$P_X \equiv *[\ A_X?a_X,\ B_X?b_X;\ X!(a_X + b_X)\ ]$$

$$P_d \equiv *[\ C_d?c_d;\ [\ c_d \longrightarrow\ D?d;\ D_y!d\ \,[]\ \neg c_d \longrightarrow\ \textbf{skip}\ ]\ ]$$

$$P_y \equiv *[\ B_y?b_y,\ C_y?c_y;\ [c_y \longrightarrow\ D_y?d_y;\ y := b_y + c_y + d_y\ \,[]\neg c_y \longrightarrow\ \textbf{skip}\ ]\ ]$$

$$Q_3 \rhd P_a \parallel P_b \parallel P_c \parallel P_X \parallel P_d \parallel P_y$$

This system is illustrated in Figure 2.7.

2.4.4 Looking Ahead

Note that there are several inefficiencies in the system implementing Q_3. First, the processes P_a, P_b and P_c are not necessary since they neither perform any computation nor copy their input variables to multiple processes. Such processes are called *L-R buffers* and their elimination is placed in context in Chapter 4. (After elimination, the channel $A_X?$ in process P_X is replaced with $A?$.) Secondly, consider the process P_d. When we strip away the selection statement, this process also reads in a value under a certain condition and then outputs the same value under the same condition. This description is very similar to that of a simple buffer, which can be eliminated. The distillation transformation identifies and handles such scenarios, and is also presented in Chapter 4.

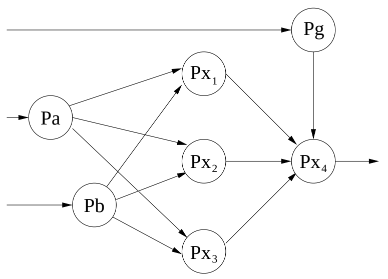
2.5 Related Work: Static Single Assignment Form

The DSA form used by DDD is similar to the static single-assignment (SSA) form commonly used in software compilers in conjunction with control dependence graphs [8, 12, 13]. While DSA programs are limited to a variable being assigned a value at most once during *execution*, in SSA programs only one assignment is allowed to a variable in the *program text*. Hence, even if assignments to a variable appear in different branches of a selection statement, that variable must be split into new variables with one assignment each. At the end of any selection statement, a *ϕ-function* is used to gather the split variables from different branches back into a single variable.

For example, consider a CHP program containing the selection statement:

```
[ $g = 0 \longrightarrow$   $x := f_1(a,\ b)$
▯ $g = 1 \longrightarrow$   $x := f_2(a,\ b)$
▯ $g = 2 \longrightarrow$   $x := f_3(a,\ b)$
]
```

the DSA version of this CHP fragment is actually unchanged since none of the shown variables are



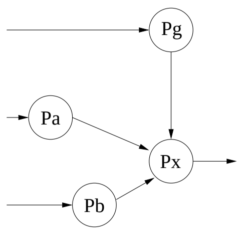


Figure 2.8: SSA and DSA forms.
The system on the left implements the SSA form of a selection statement while the system on the right implements the DSA form of a selection statement.

ever assigned a value more than once during execution. However, when the fragment is converted into SSA form, we have

```
[  $g = 0 \longrightarrow$   $x_1 := f_1(a,\ b)$
▯  $g = 1 \longrightarrow$   $x_2 := f_2(a,\ b)$
▯  $g = 2 \longrightarrow$   $x_3 := f_3(a,\ b)$
];  $x_4 := \phi(x_1, x_2, x_3)$
```

The systems resulting from the DDD projections of these two program fragments are shown in Figure 2.8.

As we can see, the DSA system uses fewer processes and channels. If DDD is used for hardware synthesis, the DSA system is certainly more efficient than the SSA system. The SSA form is more suitable for software and compiler applications, where communications are not as relatively expensive, and optimizations of the larger code typical of these applications can benefit more from the simpler data structures and reasoning arising from the true single-assignment form of SSA.

2.6 Summary

This chapter has presented the fundamental steps for the first two phases of DDD: transforming the program into DSA form, and applying projection to decompose the DSA program.

Recall that DDD requires slack-elastic programs to guarantee correctness. The method performs the following steps:

1. **Dynamic Single Assignment Transformation**
   (a) Perform early decomposition to eliminate nested loops from the program.
   (b) Transform the resulting programs into DSA form by rewriting selection statements and straightline code.
2. **Projection**
   (a) Build variable dependency sets for each program. If any variables appear in multiple sets, insert the appropriate copy and input variables.
   (b) Insert distributed assignments into the sequential code in order to prepare it for projection.
   (c) Build the new dependency and projection sets for the program and apply the technique of projection.

The results of these fundamental steps are a decomposed system that is semantically equivalent to the original sequential program, with added concurrency. This system may include unnecessary processes, unnecessary communications, and may not be optimized for circuit performance. Further optimizations and additional techniques for DDD are described in Chapters 4 and 5.

Chapter 3

Asynchronous Circuits and Synthesis

Now that we have presented DDD for general process decomposition, we focus our attention on process decomposition for the high-level synthesis of asynchronous VLSI systems. Process decomposition is the first step in the asynchronous design flow and the skill with which it is performed greatly impacts the performance and energy consumption of the final hardware. For DDD to generate systems that can be implemented as fast and energy-efficient asynchronous circuits, low-level circuit information must be incorporated in its high-level transformations.

This is the first of three chapters that present a version of DDD tailored specifically for use in the design of asynchronous hardware. We begin in this chapter by providing a general introduction to asynchronous VLSI circuits and synthesis. We then present templates for a family of fast asynchronous circuits. These templates allow DDD to estimate low-level circuit performance metrics without requiring formal logic synthesis. The ensuing chapters describe the actual modifications to DDD for use in asynchronous design.

3.1 Quasi Delay-Insensitivity

By definition, asynchronous systems eschew a global clock signal, but they may still make many different timing assumptions to synchronize their actions. The most conservative style of asynchronous design is *delay-insensitive (DI)*, which makes no timing assumptions and guarantees the correctness of computations for any set of wire- and gate-delays. It has been shown that the class of completely DI systems is quite limited, excluding most circuits of interest [36].

Quasi delay-insensitive (QDI) design makes only one kind of timing assumption, and is the most conservative approach commonly found in asynchronous VLSI systems. QDI systems include *isochronic forks*, where the assumption is made that when a certain wire splits, signals propagate along the different wire paths with similar delays. The addition of this one timing assumption allows entire microprocessors to be built. In fact, the fastest working asynchronous microprocessors to date are QDI [42]. (Other asynchronous design styles exist with more timing assumptions [21] but, as with clocked circuits, the safety margins required to ensure correctness hinder their performance.) The Caltech synthesis techniques (both manual and automated) described in this thesis localize isochronic forks to the extent that their assumptions are easily met.

The QDI design style enhances some of the inherent advantages of asynchronous design, and adds others too:

- **Low Power**: Unlike asynchronous design styles with more timing assumptions, no delay lines or similar elements are required to match delays along different paths for correctness. Hence, QDI circuits stop switching completely when idle, reducing idle dynamic-energy consumption to zero. From the perspective of synchronous VLSI, this is equivalent to "perfect" clock gating.

- **Robustness**: Independence from delays allows systems to remain correct no matter how physical parameters affect performance. With only the minimal timing assumption of isochronic forks, QDI systems are robust to variations in physical parameters such as voltage, temperature, and fabrication. (Variations in fabrication are becoming increasingly prevalent as feature size shrinks.) In practice, the voltage of QDI systems can be scaled during runtime to trade off energy and speed without requiring any dedicated circuitry or ramp-down protocols. QDI microprocessors have been demonstrated running correctly at sub-threshold voltages [42].

- **Modularity**: Using the Caltech synthesis flow for QDI design (both the existing manual approaches and the new DDD techniques), isochronic forks are almost always localized within individual circuits, leaving the system interconnect delay-insensitive. Modular design with QDI systems is therefore easier than with synchronous components that may have different clock domains, or with less conservative asynchronous components where different timing constraints may need to be met at the interfaces. Increased modularity also promotes the re-usability of circuits designed in the asynchronous QDI style.

The main disadvantages of QDI design are an area penalty caused by the extra circuitry and wiring required to implement delay-insensitivity, and a current lack of synthesis tools for automated design. The area penalty can increase the energy consumption of a system, but this effect is usually dwarfed by the other low-power advantages of asynchronous and QDI design. The lack of automated synthesis tools is, of course, addressed in part by DDD.

3.2 Communications and Handshakes

We regard an asynchronous VLSI system as a distributed system where modules (or *processes*) communicate data and synchronize computations via message-passing communications over dedicated channels. A communications channel is unidirectional and connects a sender process to a receiver process. QDI systems have no global clock signal or other timing assumption for processes to distinguish between old messages and new messages on channels. Channels therefore alternate between "valid" and "neutral" states, and the validity of a signal on channel wires (i.e., the presence of a message on the channel), is encoded within the signal itself.

3.2.1 Channel Encodings

Channel encodings used in QDI systems include not only different states for every possible message value, but also a special *neutral state* to signify the absence of valid data, and possibly one or more forbidden states. The most common encoding is *e1ofN*, which consists of $N+1$ *rails* (wires): N *data rails* that encode the message, and one *enable rail* used in communication handshakes. On a channel between two communicating processes, data rails are set by the sender while the enable rail

C.0	C.1	State
false	**false**	Neutral
true	**false**	Data = 0
false	**true**	Data = 1
true	**true**	Illegal

(a)

_C.0	_C.1	State
true	**true**	Neutral
false	**true**	_Data = 0
true	**false**	_Data = 1
false	**false**	Illegal

(b)

Figure 3.1: Channel Encodings.
Different encodings for one bit of information: (a) Data rails for an e1of2 channel C, or all rails for a 1of2 channel C. (b) All rails for a _1of2 channel $\_C$.

is set by the receiver.

An e1ofN channel C's enable rail is labeled $C.e$, and is used in handshakes to signal both the reception of valid data and then the readiness for more. In a possibly more familiar context, an enable rail is equivalent to an inversion of the acknowledge signal commonly used in data/acknowledgment handshakes for off-chip protocols. The acknowledge signal is inverted to simplify QDI circuits.

The N data rails of an e1ofN channel C (labeled $C.0, C.1, \ldots, C.N-1$) form a one-hot code that can express N different messages. When all rails are false, the channel is in its neutral state with no message present. Only one rail can be true at a time. If multiple data rails are true, the channel is in an illegal state. This data encoding is illustrated in Figure 3.1a. Figure 3.1b presents a *_1ofN* data encoding consisting only of data rails that together implement an inverted one-hot code. This encoding is not commonly used to communicate between processes but rather to store data internally within QDI circuits.

Thus, we can express one bit of information using an e1of2 (or *dual-rail*) channel C and its three wires: data rails $C.0$ and $C.1$, and enable rail $C.e$. Similarly, two bits of information can be expressed using an e1of4 channel (with five wires), and three bits of information can be expressed using either an e1of8 channel (with nine wires) or, alternatively, the combination of an e1of2 and an e1of4 channel (with eight wires total). Usually, bytes or words are encoded using a combination of e1of4 channels. For example, a one-byte channel can be implemented by four e1of4 channels, each carrying two bits of information. This is for reasons of efficiency. (For a channel encoding, $4 \times (4+1) = 20$ wires is more efficient than $2^8 + 1 = 257$ wires.)

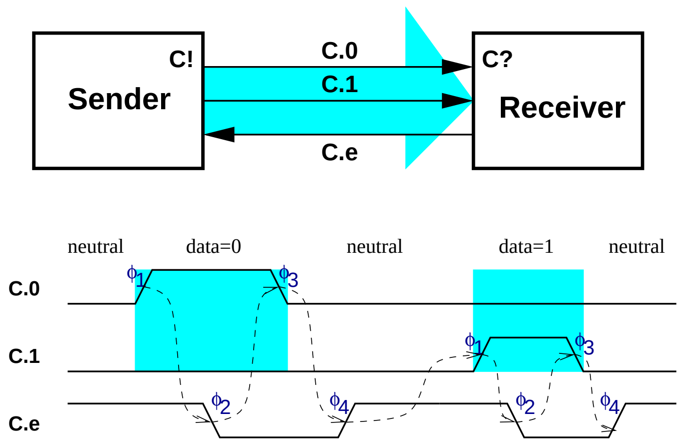


Figure 3.2: Four-phase handshake on a one-bit communication channel encoded as e1of2.

3.2.2 Handshakes

Communications on e1ofN channels are implemented with a four-phase handshake protocol, illustrated in Figure 3.2 and given below:

- Receiver:
  1. Set phase: Wait for one of the data rails to be true (valid data)
  2. Set phase: Set the enable rail to false
  3. Reset phase: Wait for all data rails to be false (neutral data)
  4. Reset phase: Reset the enable rail to true
- Sender:
  1. Set phase: Wait for the enable rail to be true
  2. Set phase: Set the message's data rail to true (set valid data)
  3. Reset phase: Wait for the enable to be false
  4. Reset phase: Reset data rails to false (reset neutral data)

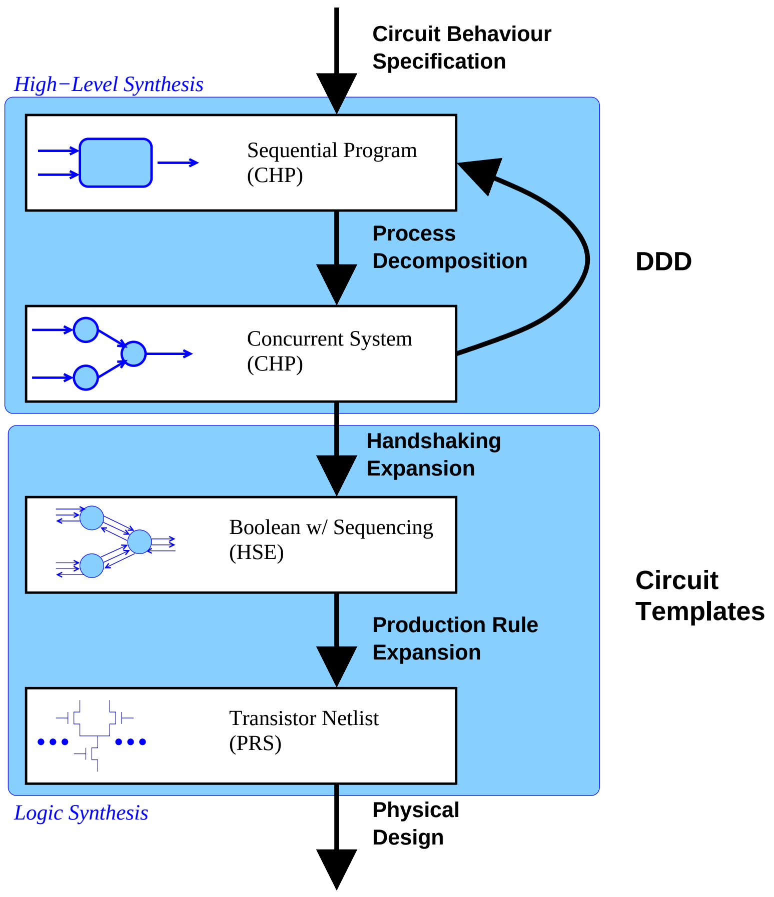


Figure 3.3: Formal synthesis flow for asynchronous VLSI design.

3.3 Asynchronous VLSI Synthesis Overview

The Caltech synthesis method for asynchronous VLSI consists of a series of semantics-preserving program transformations [37]. System behavior is initially specified as a sequential program in the high-level CHP language. The transformations are then applied successively, each generating a lower-level circuit description. The final output of the formal method can be used as a transistor netlist. The method is illustrated in Figure 3.3.

The first program transformation is *process decomposition*, in which the original process with the sequential program is transformed into a concurrent system of communicating processes, still expressed in the high-level language. This transformation serves three main purposes: dividing the process to enable its conquering by lower-level program transformations, pipelining computations, and introducing concurrency into the system. It is performed repeatedly, until the processes are deemed to be small enough for tractable further synthesis. If the new processes still use wide channels (for example, communicating bytes of data), the transformation of *vertical decomposition* is performed to split a process and its channels into multiple "slices" that operate on smaller channel encodings.

After process decomposition, we have a concurrent system that may still be implemented as software, synchronous hardware, or asynchronous hardware. The next transformation of *handshaking expansion* brings the system into the realm of asynchronous design, as high-level communications are mapped onto QDI handshake protocols. The handshaking expansion language HSE is a subset of CHP where all data types are boolean (or a collection of booleans), and the only possible actions are to wait for a boolean condition to become true, or to assign a value to a boolean variable or channel rail. Selection and repetition control structures remain in the HSE language.

The standard four-phase receive protocol described in the previous section for e1of2 input communication "$A?a$" is expressed in HSE as

$$[A.0 \;\vee\; A.1];\; A.e{\downarrow};\; [\neg A.0 \;\wedge\; \neg A.1];\; A.e{\uparrow}$$

where "$[B]$" indicates "wait until boolean condition B is true." Meanwhile, the send protocol for a four-phase handshake of an output communication "$X!0$" expressed in HSE is as follows:

$$[X.e];\; X.0{\uparrow};\; [\neg X.e];\; X.0{\downarrow}$$

Some of the intricacies of handshaking expansion lie in the interleaving of handshakes for different channels so that a minimum amount of data requires explicit storage. For example, given the original CHP "$A?x; X!x$", where all encodings are e1of2, the handshakes for the receive and send actions could result in the following HSE:

$$[A.0 \longrightarrow x.0{\uparrow} \;\vee A.1 \longrightarrow x.1{\uparrow}];\; A.e{\downarrow};\; [\neg A.0 \wedge \neg A.1];\; A.e{\uparrow};$$
$$[X.e];\; [x.0 \longrightarrow X.0{\uparrow} \;[\!]\; x.1 \longrightarrow X.1{\uparrow}];\; [\neg X.e];\; X.0{\downarrow},\; X.1{\downarrow}$$

However, the variable x must now be explicitly stored within the process. If instead the handshakes for the two communications are interleaved:

$$[X.e];\ [A.0 \longrightarrow X.0\uparrow \ [\!]\ A.1 \longrightarrow X.1\uparrow];\ A.e\downarrow;$$
$$[\neg A.0 \wedge \neg A.1 \wedge \neg X.e];\ X.0\downarrow, X.1\downarrow$$

the same value is always communicated, but no intermediate variable is required.

Handshaking expansion makes QDI communications explicit and expresses behavior at the level of circuit nodes and wires. However, HSE still assumes an implicit sequencing of actions (using the semicolon operator) that does not exist in actual circuits. The final transformation of *production-rule expansion* converts each sequential HSE program into a concurrent set of actions, or production rules. Production rules have the form "$B \rightarrow x\uparrow$" or "$B \rightarrow x\downarrow$", where B is a boolean guard condition, and $x\uparrow$ is the equivalent of assignment $x :=$ **true** to boolean variable x, and $x\downarrow$ is the equivalent of the assignment $x :=$ **false**. PRS assignments can only be executed, or *fired*, when their guard conditions evaluate to true.

Generated PRS can be treated as a transistor netlist. As a simple example, the PRS (production-rule set)

$$\begin{array}{lcl} a \wedge b & \rightarrow & x\downarrow \\ \neg a \vee \neg b & \rightarrow & x\uparrow \end{array}$$

implements a nand-gate. PRS is executed in the following manner: note the production rules whose guard conditions currently evaluate to true, and fire one of them. There is no sequencing between the rules and so all ordering must be made explicit within the guard conditions of the production rules.

As described in greater detail elsewhere [38], a production-rule set is correct when it is both *stable* and *non-interfering*. Stability requires that once any guard evaluates to true, it remains true until after the rule has fired. Non-interference requires that given production rules "$B1 \rightarrow x\uparrow$" and "$B2 \rightarrow x\downarrow$" for a variable x, "$\neg B1 \vee \neg B2$" is always true. Creating correct PRS for an HSE process often involves inserting new variables into the HSE to enforce stability and non-interference.

The formal synthesis flow outlined here can be used to generate many different types of QDI circuits. DDD automates process decomposition. Some tools exist to perform low-level handshaking and production-rule expansion, but there is enough freedom in the general transformations that applying the tools to anything but small and simple processes is infeasible. Practically, QDI designers limit themselves to a family of circuits with a set interleaving of handshakes. This limits the freedom of handshaking and production rule expansion, but also makes the transformations tractable. Examples of formal low-level synthesis for this circuit family are provided, along with less formal circuit templates, in the following section.

3.4 Asynchronous Pipeline Stages

High-performance QDI systems use fine-grained *asynchronous pipeline stages* as their standard building blocks. These pipeline stages are based on the simple CHP process "$*[A?a; X!f(a)]$," but can be considerably more complex. Consider a process that receives data on multiple input channels, computes new values using that data, and sends the results on multiple output channels. The computations and communications can appear in any pattern, be executed conditionally in selection statements, and repeated indefinitely in loops. All such behavior can be expressed in CHP and implemented in hardware by an asynchronous pipeline stage that integrates control with datapath.

CHP processes are not always intended to be synthesized into hardware systems, but processes that do describe circuit behaviors usually have the following basic form (familiar from Chapter 2):

$$P \;\equiv\; P_{init};\; *[\; P_{loop} \;]$$

P_{init} consists of assignments and output communications that are executed once to initialize the system. The main loop body P_{loop} represents the repeated behavior of the circuit after initialization. When we mention an "iteration" of a circuit, we refer to the CHP in P_{loop} itself, not including the outer unconditional loop. Henceforth, we assume that all CHP programs fit this form.

We specify a strict set of requirements for CHP processes that are directly implementable as asynchronous pipeline stages. These requirements are sufficient but not necessary; many CHP programs that do not conform can still be transformed at lower levels of synthesis into such stages. Nevertheless, we present these specifications so that later we can easily and formally demonstrate that DDD always creates networks of processes that can each be implemented as single asynchronous pipeline stages.

Definition 10 [Strict CHP requirements for asynchronous pipeline stages] *A CHP process* $P \equiv P_{init};$`*[`P_{loop}`]` *can be directly implemented as an asynchronous pipeline stage if the following conditions are true:*

- *Both* P_{init} *and* P_{loop} *are terminating programs*
- P_{init} *contains only assignments or output communications*
- *No channel is used more than once in any execution trace of either* P_{init} *or* P_{loop}
- *In any trace of* P_{loop}*, all input communications precede all output communications*

□

While nested loops are not allowed by this specification, nested selection statements and conditional communications are, so long as no channel is used more than once in any selection branch.

3.5 Precharged Half-Buffers

Among the various types of asynchronous pipeline stages, *precharged half-buffers (PCHBs)* offer the most attractive combination of speed and compactness. The fastest asynchronous microprocessors to date use PCHBs for more than 90% of their circuits. PCHBs both compute and store data. They derive their name from the facts that they are precharged circuits and that, because of their handshaking expansions, two are required to store one message. (These details are described further in Chapter 5.) PCHB circuits are the target of our synthesis method for asynchronous systems, and also the basis of our reconfigurable asynchronous architecture.

We have already specified the requirements for a CHP process to be directly implementable as an asynchronous pipeline stage. (Under physical constraints of circuit size such as the maximum number of transistors allowed in series, asynchronous pipeline stages can be implemented as PCHBs.)

Next, we present low-level language specifications of PCHB circuits and demonstrate how they can be generated using traditional handshaking and production-rule expansion. This formal compilation is no longer required, as circuit templates can be used to generate PCHBs. We therefore introduce the general PCHB template that DDD uses to incorporate low-level circuit information efficiently when performing high-level synthesis.

3.5.1 Traditional Compilation

Recall the basic send ($R!x$) and receive ($L?x$) handshakes for four-phase communication

$$[R.e];\ R.d\uparrow;\ [\neg R.e];\ R.d\downarrow$$

and

$$[L.d];\ L.e\downarrow;\ [\neg L.d];\ L.e\uparrow$$

(where, given a channel C, $C.e$ represents its enable rail and $C.d$ represents its collective data rails). For a simple buffer "$*[L?x; R!x]$", the PCHB interleaving of these handshakes is

$$[R.e];\ [L.d];\ R.d\uparrow;\ L.e\downarrow;\ [\neg R.e];\ R.d\downarrow;\ [\neg L.d];\ L.e\uparrow$$

Thus, for example, given a CHP process $*[A?a, B?b; X!(a \wedge b)]$ (where all channels encode booleans as e1of2), we can compile it into a PCHB circuit by first writing its handshaking expansion as follows:

```
*[ [$X.e$]; [$A.0 \vee A.1$], [$B.0 \vee B.1$];
   [ $A.0 \vee B.0 \longrightarrow X.0\uparrow$ ▯ $A.1 \wedge B.1 \longrightarrow X.1\uparrow$ ]; $A.e\downarrow, B.e\downarrow$;
   [$\neg X.e$]; $X.0\downarrow, X.1\downarrow$; ([$\neg A.0 \wedge \neg A.1$]; $A.e\uparrow$), ([$\neg B.0 \wedge \neg B.1$]; $B.e\uparrow$)
 ]
```

The next step in traditional synthesis is to compile the HSE into a stable and non-interfering set of production rules. For the result to be implementable in CMOS technology, the production rules must be inverting: rules "$B \to x\uparrow$" must contain only negated nodes such as "$\neg a$" in guard B, while rules "$B \to x\downarrow$" must contain only unnegated nodes. The presence of HSE such as "$[A.0 \vee B.0 \to X.0\uparrow \ldots$" already indicates that the compiled PRS will not be CMOS-implementable. We therefore insert inverse variables into the HSE:

$$\begin{array}{lcl}
A.e \wedge B.e \wedge X.e \wedge (A.0 \vee B.0) & \rightarrow & \_X.0\downarrow \\
A.e \wedge B.e \wedge X.e \wedge (A.1 \wedge B.1) & \rightarrow & \_X.1\downarrow \\
\neg\_X.0 & \rightarrow & X.0\uparrow \\
\neg\_X.1 & \rightarrow & X.1\uparrow \\
(A.0 \vee A.1) \wedge (X.0 \vee X.1) & \rightarrow & A.e\downarrow \\
(B.0 \vee B.1) \wedge (X.0 \vee X.1) & \rightarrow & B.e\downarrow \\
\neg A.e \wedge \neg B.e \wedge \neg X.e & \rightarrow & \_X.0\uparrow \\
\neg A.e \wedge \neg B.e \wedge \neg X.e & \rightarrow & \_X.1\uparrow \\
\_X.0 & \rightarrow & X.0\downarrow \\
\_X.1 & \rightarrow & X.1\downarrow \\
(\neg A.0 \wedge \neg A.1) \wedge (\neg X.0 \wedge \neg X.1) & \rightarrow & A.e\uparrow \\
(\neg B.0 \wedge \neg B.1) \wedge (\neg X.0 \wedge \neg X.1) & \rightarrow & B.e\uparrow
\end{array}$$

Figure 3.4: CMOS-implementable PRS.
For the CHP process $*[A?a, B?b; X!(a \wedge b)]$. Modifications are required to reduce the number of p-transistors in series.

```
*[ [$X.e$]; [$A.0 \vee A.1$], [$B.0 \vee B.1$];
    [ $A.0 \vee B.0 \longrightarrow$ $\_X.0\downarrow$; $X.0\uparrow$ [] $A.1 \wedge B.1 \longrightarrow$ $\_X.1\downarrow$; $X.1\uparrow$ ]; $A.e\downarrow, B.e\downarrow$;
    [$\neg X.e$]; $\_X.0\uparrow, \_X.1\uparrow$; $X.0\downarrow, X.1\downarrow$; ([$\neg A.0 \wedge \neg A.1$]; $A.e\uparrow$), ([$\neg B.0 \wedge \neg B.1$]; $B.e\uparrow$)
 ]
```

Using techniques described by Martin [38], we compile this HSE to generate the PRS given in Figure 3.4. This PRS is CMOS-implementable but contains gates with as many as four p-transistors in series, which is unadvisable for performance reasons. We therefore insert more intermediate variables to the circuit to reduce the number of p-transistors in series. These variables include channel data validities and a local circuit enable signal *en*. The final PRS, which remains CMOS-implementable as well as stable and non-interfering, is given in Figure 3.5. The final result is not the most efficient PRS possible, but serves as an gentle introductory example to traditional low-level compilation methods for asynchronous VLSI. The circuit corresponding to the PRS is illustrated in Figure 3.6.

Staticizers (keeper circuits featuring cross-coupled inverters, one of which is weak) do not appear in the PRS but are added to non-combinational nodes. Muller C-element gates are common in general asynchronous circuit design and are present in the PCHB. C-elements are not combinational. Their outputs only switch when all inputs agree, and otherwise retain their previous value. These gates must therefore be followed by staticizers. If a C-element's output is fed into an inverter, then that

$$\begin{array}{lcl}
A.0 \vee A.1 & \rightarrow & \_Av\downarrow \\
B.0 \vee B.1 & \rightarrow & \_Bv\downarrow \\
\neg\_Av & \rightarrow & Av\uparrow \\
\neg\_Bv & \rightarrow & Bv\uparrow \\
en \wedge X.e \wedge (A.0 \vee B.0) & \rightarrow & \_X.0\downarrow \\
en \wedge X.e \wedge (A.1 \wedge B.1) & \rightarrow & \_X.1\downarrow \\
\neg\_X.0 & \rightarrow & X.0\uparrow \\
\neg\_X.1 & \rightarrow & X.1\uparrow \\
\neg\_X.0 \vee \neg\_X.1 & \rightarrow & Xv\uparrow \\
Av \wedge Xv & \rightarrow & A.e\downarrow \\
Bv \wedge Xv & \rightarrow & B.e\downarrow \\
\neg A.e \wedge \neg B.e & \rightarrow & \_en\uparrow \\
\_en & \rightarrow & en\downarrow \\
\neg A.0 \wedge \neg A.1 & \rightarrow & \_Av\uparrow \\
\neg B.0 \wedge \neg B.1 & \rightarrow & \_Bv\uparrow \\
\_Av & \rightarrow & Av\downarrow \\
\_Bv & \rightarrow & Bv\downarrow \\
\neg en \wedge \neg X.e & \rightarrow & \_X.0\uparrow \\
\neg en \wedge \neg X.e & \rightarrow & \_X.1\uparrow \\
\_X.0 & \rightarrow & X.0\downarrow \\
\_X.1 & \rightarrow & X.1\downarrow \\
\_X.0 \wedge \_X.1 & \rightarrow & Xv\downarrow \\
\neg Av \wedge \neg Xv & \rightarrow & A.e\uparrow \\
\neg Bv \wedge \neg Xv & \rightarrow & B.e\uparrow \\
A.e \wedge B.e & \rightarrow & \_en\downarrow \\
\neg\_en & \rightarrow & en\uparrow
\end{array}$$

Figure 3.5: Final CMOS-implementable PRS for the CHP process $*[A?a, B?b; X!(a \wedge b)]$.

inverter is incorporated into the staticizer circuit. This is the case with the staticizer for the circuit node "$\_en$" in Figure 3.6.

3.5.2 Circuit Templates

DDD does not perform traditional compilation from CHP to HSE to PRS when designing systems of PCHBs. Instead, it exploits the regularity of PCHB circuits by creating templates (based upon formal compilations) from which it can easily estimate circuit performance metrics such as cycle time and energy consumption. These templates do not always represent the most efficient design for a specific CHP process, but can be generally and reasonably applied to any CHP directly implementable as an asynchronous pipeline stage. A PCHB circuit comprises three main components: *computation networks*, *validity trees*, and *completion networks*.

While the lack of clock distribution trees saves energy, the bulk of energy savings in QDI systems

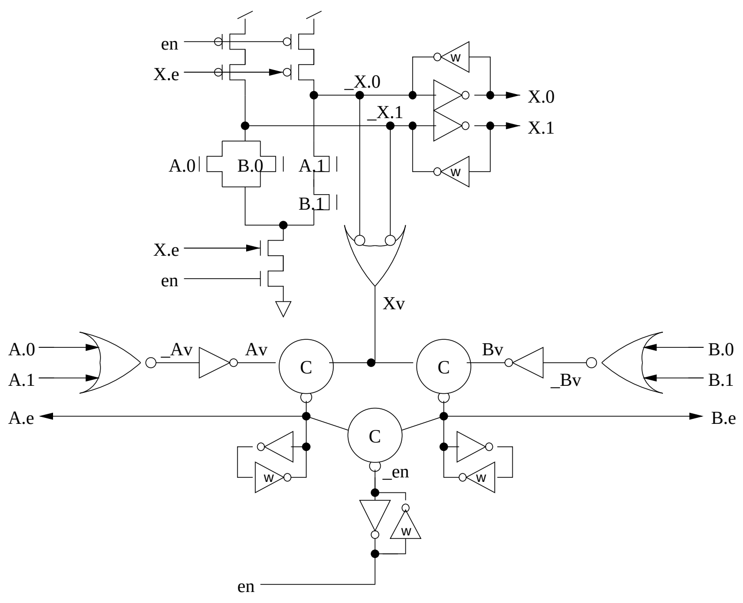


Figure 3.6: PCHB corresponding to the PRS compiled from $*[A?a, B?b; X!(a \wedge b)]$. Synthesized using traditional techniques, this is not the most efficient implementation (fewer gates are required if the validity trees for channels A and B are combined), but serves well as an introduction to formal asynchronous VLSI synthesis.

compared to synchronous systems arises from the absence of unnecessary "don't care" communications. The communications of a QDI system consume much more energy than the computations. Since channels are always reset to neutral encodings and all communications must be acknowledged, communications where the data is simply thrown away are wasteful and expensive indeed.

QDI systems avoid "don't cares" by using *conditional communications*, where a PCHB performs certain inputs or outputs only if data received on other channels fulfills certain logical conditions. For conditional outputs, the PCHB may compute but ultimately discard output data. For conditional inputs, the PCHB may ignore and fail to acknowledge valid data on the input channel. In the latter case, any data (which can have arrived early since there is no global clock synchronizing the different processes) is left on the channel's data rails, awaiting the iteration when other control data indicates that it should be received.

While additional circuitry is required in each PCHB to implement conditional communications, the overall energy savings are still significant. Not only can switching be reduced on interconnect wires, but conditional communications can render entire processes idle with no dynamic energy consumption for long periods of time. For example, consider the case where process A receives data from processes B and C. Both B and C have conditional outputs, where data is only set when some system variable g is true. When g is true, A receives its inputs, performs its computations, and sends out its specified outputs. When g is false, however, A receives no inputs at all and therefore performs no computations or output communications, but simply remains idle. Such situations are considered in further detail in Chapter 4.

We begin this section by considering PCHBs with only unconditional communications.

3.5.2.1 Unconditional Communications

A basic circuit template for PCHBs with only unconditional communications is presented in Figure 3.7.

Each PCHB output channel has a *computation network* in which the values sent on the channel are computed. The computation networks resemble precharge domino logic seen in synchronous circuit design. Instead of a global clock, it is the local enable signal *en* along with the output channel enable that guards the power supplies. The computation for an output channel $X!$ is performed in a pulldown network of n-transistors. The pullup network of p-transistors is small and does not perform any computations, simply precharging the output nodes instead.

The output nodes $\_X.d$ of the computation network use a _1ofN encoding, and are inverted to produce data rails $X.d$. (These inverters form part of the staticizers that store the outputs of the non-combinational precharged stage.) The resulting even parity of the *forward latency* (e.g., from input channel data rails $A.d$ to output channel data rails $X.d$) enables computations in neighboring pipeline stages always to occur in n-transistor networks, instead of alternating stages using slower p-transistor networks. The PCHB computation networks for output channels $X!$ and $Y!$ are shaded in Figure 3.7.

Beside the computation network, each input and output channel in a PCHB has its own *validity tree* that checks for the presence or absence of data on the channel's data rails. The trees are composed of combinational gates that together form the equivalent of a large or-gate (with inverted

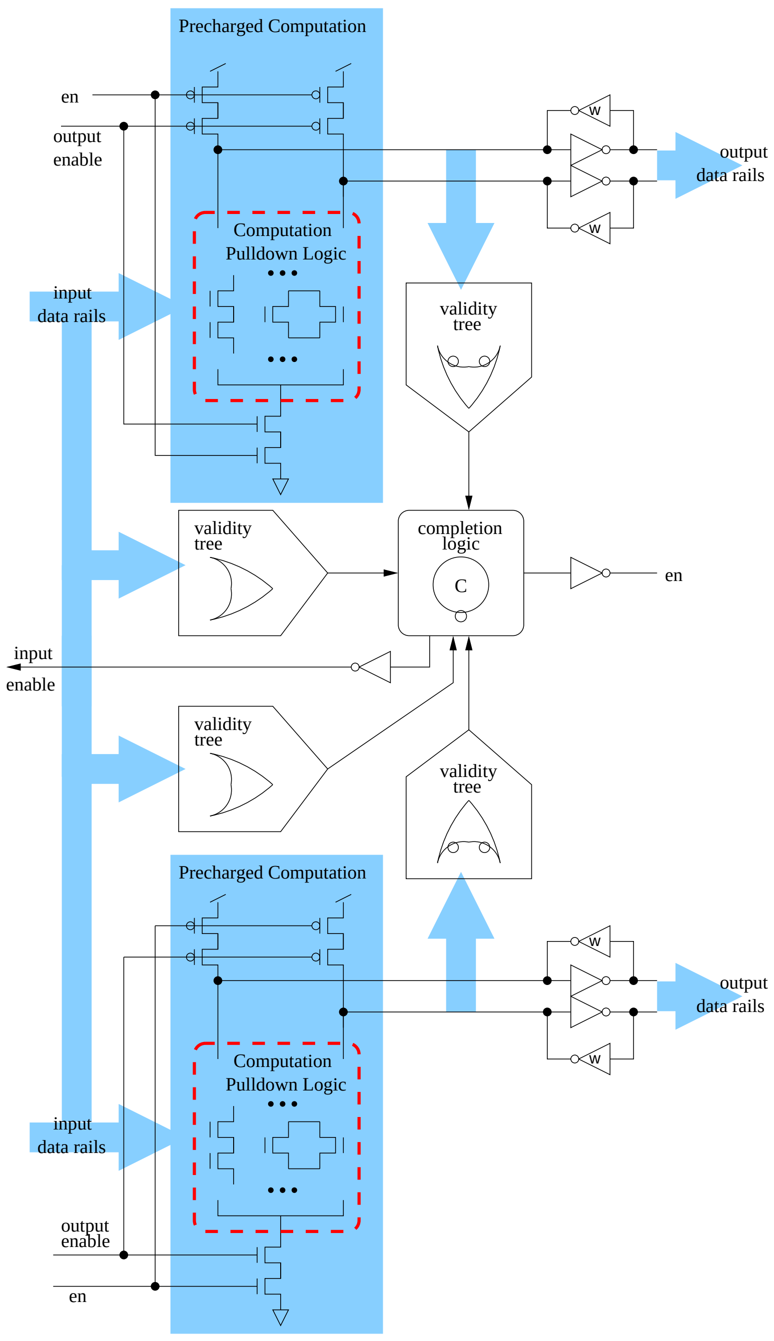


Figure 3.7: General template for an unconditional precharged half-buffer circuit.

inputs for output-channel validity). Regardless of the number of computation networks in which it is used, an input channel has only one validity tree per PCHB.

Although other configurations are possible, DDD assumes in its cycle-time estimations that each channel has its own validity tree. Input and output channel validities are gathered together

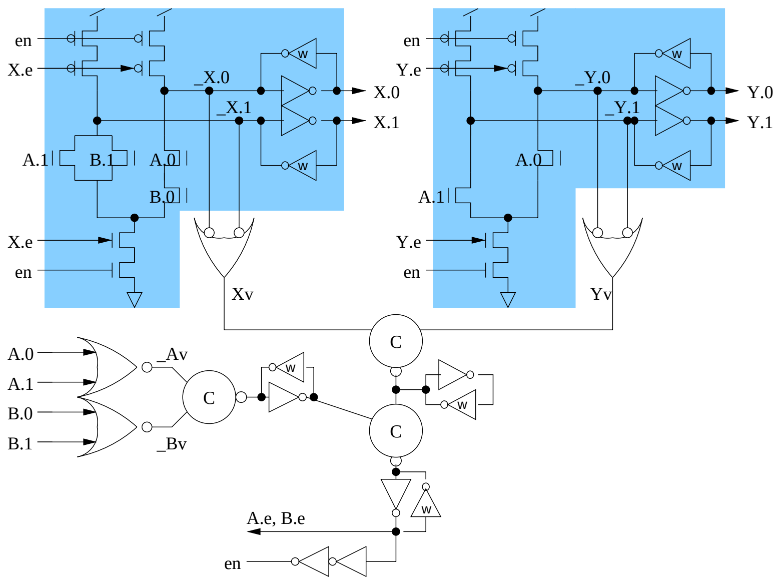


Figure 3.8: Template-designed PCHB circuit.
For the following CHP process with dual-rail channels: $*[A?a, B?b; X!(a \vee b), Y!a]$. The computation networks for the two output channels are shaded.

in a *completion network*, to generate signals for input channel enable rails, and the local enable signal *en* that guards the power supplies of every computation network. An input enable depends on the validities of the input channel and all output channels that use the input variable in their computations. The local enable signal depends upon all of the input enables, and unites the various computation networks in the PCHB. Because of their potentially large fanout, the input enables and the signal *en* are usually immediately preceded by at least one inverter, and possibly by two for CMOS parity. An example of a template-designed PCHB for a CHP process is given in Figure 3.8.

3.5.2.2 Conditional Outputs

There are several ways of implementing conditional communications. We present techniques that can be applied in general to any CHP process that fits the requirements for asynchronous pipeline stages.

The design problem presented by conditional outputs reduces to the fact that communications on input channels must still be acknowledged if no output is performed. We accomplish this by adding an extra state for the PCHB's internal representation of conditional output channels that signifies that the channel is ready for communication but the condition is such that no communication should take place. This extra state is included in the output validity checks so that even if no output handshakes occur, the input handshakes can note this extra state and complete their handshakes without deadlock.

Our approach to generally implementing this extra conditional output state is to add an extra "dummy" rail to the internal _1ofN encoded data computed by the precharge stage. This extra rail is also an input to the output channel validity tree. It is active (low) when no communications occur on the output channel so that the input channels can still be acknowledged and the internal enable signal reset for the next iteration. As a simple example, consider the process $*[A?a, G?g; [g = 0 \rightarrow$ **skip**$\,[\!]\, g = 1 \rightarrow X!a]]$. The basic handshaking expansion is

```
*[ $[X.e]$; $[A.d], [G.d]$;
    $[G.1 \wedge A.d \longrightarrow$ $\_X.d\downarrow$; $X.d\uparrow$; $A.e\downarrow, G.e\downarrow$; $[\neg X.e]$
    $[\!]\, G.0 \longrightarrow$ $\_X.N\downarrow$; $A.e\downarrow, G.e\downarrow$
    $]$; $\_X.d\uparrow, \_X.N\uparrow$; $X.d\downarrow$; $[\neg A.d \wedge \neg G.d]$; $A.e\uparrow, G.e\uparrow$
 ]
```

where $X.d$ represents the collective data rails of channel X, $\_X.d$ represents the inverted data rails for output channel X, and $\_X.N$ represents the extra output rail for conditional output channel X. The traditional compilation of this HSE into CMOS-implementable production rules inserts a local enable and validity variables.

The final PCHB circuit is illustrated in Figure 3.9. (Staticizers are not shown.) Note in particular that the precharge network for the dummy rail $\_X.2$ contains only the local circuit enable signal *en* and not the output channel enable $X.e$. (This is necessary to avoid deadlock, since the output channel X is inactive when $\_X.2$ is pulled down, and its enable $X.e$ will not be lowered.) Thus, the template for PCHBs with conditional outputs is identical to those for unconditional PCHBs, except for the dummy inverted output rail, the logic associated with its pulldown and pullup, and its connection to the output validity tree.

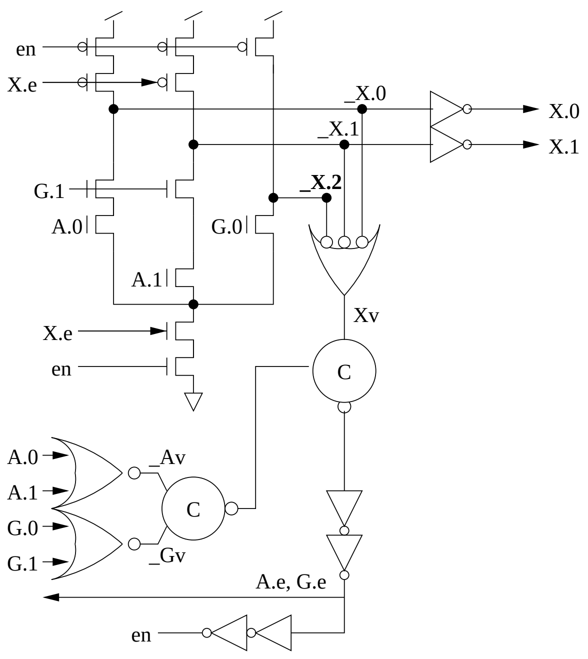


Figure 3.9: Example of a PCHB with conditional output $X!$.

3.5.2.3 Conditional Inputs

It is more expensive to implement conditional inputs than conditional outputs. Complexities and additional circuitry arise because in addition to computing the input condition and modifying the validity circuitry, conditional inputs require that the enable rail for the input channel be suppressed and replaced in the PCHB's completion tree. (This must be accomplished without introducing deadlock or unacknowledged transitions within the PCHB circuit.)

A simple but general way to fulfill all these requirements is to compute the input condition as if it were another dual-rail output (i.e., perform the computation in a separate precharged stage guarded by the local enable signal). The results of this computation are used in three distinct parts of the PCHB:

- **Condition Validity:** The validity of the condition computation is included in the completion trees of any other input channels used in the condition expression. In this sense, the condition computation is treated as another output computation.
- **Suppressing Input Channel Validity:** If an input channel contains a valid message but is

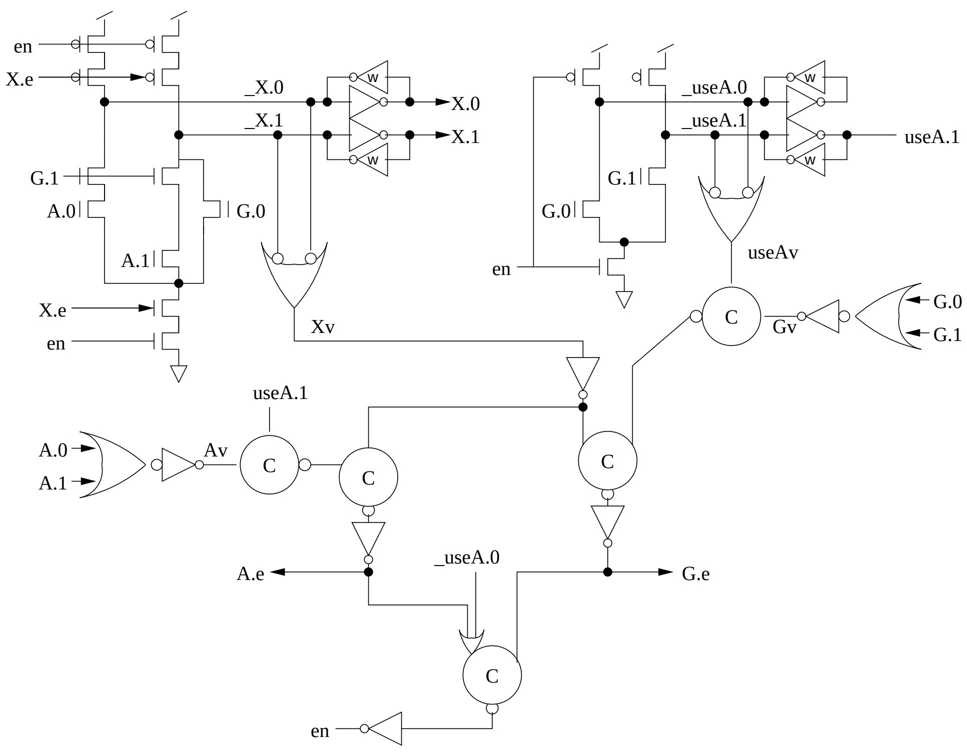


Figure 3.10: Example of a PCHB with conditional input $A?$.

not to be used during this iteration, the positive condition rail is used to prevent the conditional input-channel validity from propagating in the PCHB. This prevents acknowledgment of the data on the input channel, and postpones communications for another iteration.

- **Replacing Input Channel Enable:** When the conditional input channel is not required, the negative condition rail is used to replace the input enable in the local completion tree, preventing deadlock.

We use the process

$$*[G?g;\ [g=0 \longrightarrow\ X!1\ [\!]\ g=1 \longrightarrow\ A?a;\ X!a]]$$

to illustrate these points.

For the conditional input channel $A?$, the new precharged network has outputs that are encoded in an internal _1of2 channel called $\_useA$, with $\_useA.1$ being false if the condition is met (i.e., the input should occur) and $\_useA.0$ being false if the condition is not met (i.e., the input should not occur). The basic handshaking expansion is

```
*[ [$R.e$]; [$G.d$];
    [ $G.0 \longrightarrow$  $\_useA.0\downarrow$,  $\_R.1\downarrow$;  $R.1\uparrow$;  $G.e\downarrow$;  $\_useA.0\uparrow$,  $\_R.1\uparrow$;  $R.1\downarrow$;  [$\neg G.d$]
    [] $G.1 \longrightarrow$  $\_useA.1\downarrow$,  [$A.d$];  $\_R.d\downarrow$;  $R.d\uparrow$;  $A.e\downarrow$,  $G.e\downarrow$;
                $\_useA.1\uparrow$,  $\_R.d\uparrow$;  $R.d\downarrow$;  ([$\neg A.d$]; $A.e\uparrow$),  ([$\neg G.d$]; $G.e\uparrow$)
 ] ]
```

The CMOS-implementable PCHB compiled from this handshaking expansion is illustrated in Figure 3.10. It includes a validity signal *_useAv* for the input-channel condition nodes. This condition validity is treated as an output-channel validity and is included in the generation of the input enable for the control channel used to compute the condition G. Note that while unconditional input channels can copy the same input enable signal, input enables for conditional channels must be kept separate so that they may switch under different conditions.

Elsewhere, in the PCHB's completion circuitry, condition signal *useA*.1 prevents the propagation of input-channel validity signal Av in the PCHB when no input on A should occur. (If an input arrives early on A, Av may evaluate to true even though A is not to be used during this cycle.) This prevents any inputs on A from being incorrectly acknowledged by $A.e$.

In cases where the completion network for $A.e$ actually comprises multiple levels of C-elements, both Av and *useA*.1 must be included as inputs together for the same C-element at a leaf of the tree. Otherwise, the signal on Av may propagate through C-elements and even if it is eventually suppressed later in the tree by by *useA*.1, transitions on internal C-element tree nodes may be unacknowledged (i.e., have no successor transitions). Such behavior is not allowed in QDI systems [36]. Avoiding this problem in conditional completion trees can be tricky in general, and avoiding it in an optimal fashion is a complex enough problem to involve random heuristics and be dealt with using separate tools. By setting up the conditional inputs as we have described here, ensuring that the two pertinent signals share a C-element is straightforward.

Finally, in every iteration, either A is used and $A.e$ is lowered, or A is not used and *_useA*.0 is lowered. Thus, including signal *_useA*.0 along with $A.e$ in the final completion circuitry generating local signal *en* prevents deadlock in the PCHB whenever the input channel A is not used.

Thus, templates for conditional inputs include an extra entire precharge stage for each conditional input, and extra inputs to the completion networks of the PCHB.

3.5.2.4 State-holding Bits

Finally, state-holding bits may be explicitly added to DDD processes when they are rewritten to avoid multiple communications per iteration. For example, $*[A?a; X!a; A?b; Y!b]$ may be rewritten as

$$s\downarrow;\ *[\ A?a;\ [\neg s \longrightarrow\ X!a,\ s\uparrow\ [\!]\ s \longrightarrow\ Y!b,\ s\downarrow\ ]]$$

The main loop body of this program uses new variable s before assigning it a value. Such variables can be implemented by including internal registers in the PCHB. Section 4.1.2 describes how DDD inserts state-holding bits when transforming processes to make them directly implementable as PCHBs. Specific circuit techniques for implementing general state-holding bits are given by Lines [29].

3.6 Performance Metrics

PCHB templates allow DDD, a high-level synthesis algorithm, to perform circuit-level optimizations on a system without design iterations involving actual low-level synthesis. Given the CHP specification and channel encodings of a process, we can estimate the cycle time and dynamic energy consumption when it is eventually implemented as a PCHB.

The three circuit characteristics estimated by DDD are cycle time, energy consumption, and size limitations in terms of transistors in series. We measure the cycle time of a PCHB in units of *transitions*, where the firing of a production rule equals one transition. While a path with more transitions can be faster than a path with fewer transitions but larger nodes, transition counts provide a quick way to estimate the performance of a circuit without delving into lower-level analog details.

The *internal cycle-time* of a PCHB is defined as the number of transitions that occur between consecutive resets of the local enable signal en. Since this number can depend on the size of other PCHBs with which our circuit performs communications handshakes, we assume that simple left-right PCHB buffers are placed at each of the input and output channels when measuring the internal cycle-time of a PCHB. Since an odd number of transitions are required in CMOS to set and then reset values, transition counts for PCHB cycle times are usually twice an odd number. (Recent asynchronous microprocessor designs have operated at cycle times of 18 and 22 transitions [39, 41].) The major exception to this rule are processes that include state-holding bits, which can have cycle times of twice an even number—the average number of transitions for the set and reset phases which have different odd numbers of transitions.

DDD estimates the cycle time of a circuit by considering the number of input and output channels, the width and conditionality of each channel, and the maximum fanin for any type of logic gate. The maximum fanin (set for different target technologies) determines the height of validity trees and completion networks. Given the PCHB templates from the previous section, this information is sufficient to estimate the number of transitions required per cycle.

For example, consider an unconditional PCHB with N_{in} input channels labeled $A_0 \ldots A_{N_{in}-1}$ and N_{out} output channels labeled $X_0 \ldots X_{N_{out}-1}$. Each channel C has $C.N$ data rails. We assume for now that the circuit is "fully connected" (all of the outputs depend upon all of the inputs). The computation in this section is not guaranteed to return the minimum achievable cycle time because it assumes that *all* channel validities must be generated before input enable signals can be generated ("monolithic completion"). This is our base circuit; more complicated scenarios involving partially connected circuits, early channel validities, and conditional communications are presented elsewhere [53].

Let τ_{vin} be the maximum number of transitions between the circuit's receiving data and its generating a validity signal for any input channel. Let τ_{vout} be the maximum number of transitions between the circuit's receiving data and its generating a validity signal for any output channel. Let τ_{valid} be the maximum number of transitions between the circuit's receiving data and its generating

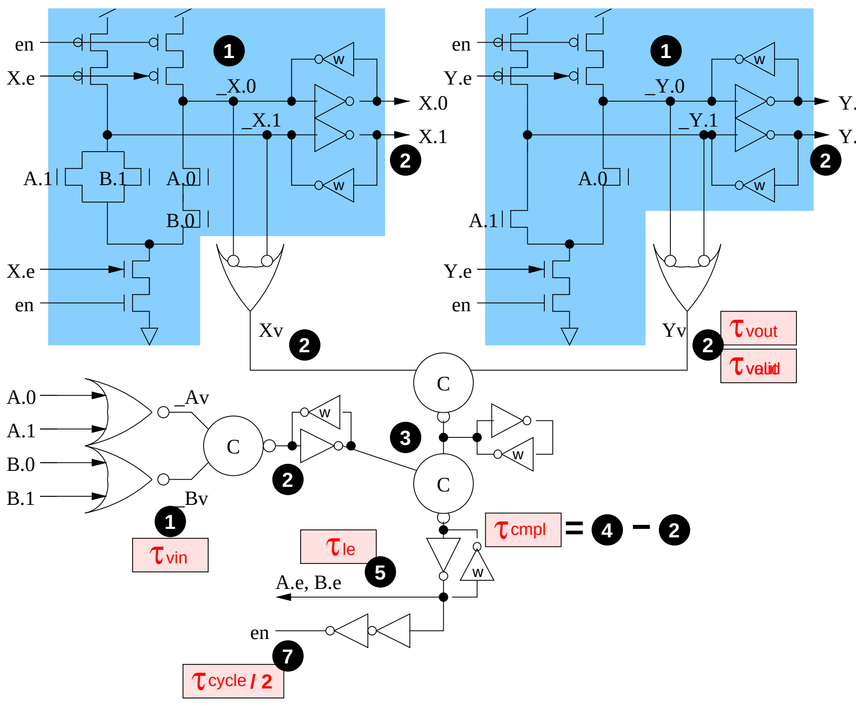


Figure 3.11: PCHB circuit annotated with transition counts and delays used to estimate cycle time.

validity signals for any channel. Let τ_{cmpl} be the number of transitions required to gather all of the channel validities into a single signal (this is the completion tree). Let τ_{le} be the number of transitions required to generate the left enable signal. Let τ_{cycle} be the cycle time. These delays are all annotated in Figure 3.11 for the PCHB given in Figure 3.8.

We introduce the function

$$\mathit{makeOdd}(x) = x + (x+1) \bmod 2$$

to achieve the desired transistor count parities for our inverting CMOS circuits. We also can easily compute the height of trees with fanin N that compute input channel validities, output channel validities, and enable signals using functions $\mathit{Height}(\mathbf{iv}, N)$, $\mathit{Height}(\mathbf{ov}, N)$, and $\mathit{Height}(\mathbf{ce}, N)$, respectively. (Input validity trees consist of alternating nor- and nand-gates; output validity trees consist of alternating nand- and nor-gates; enable trees consist of C-elements.) We have the following:

$$\begin{aligned}
\tau_{vin} &= \max_{0 \le i < N_{in}} \{ \, \mathit{Height}(\mathbf{iv}, A_i.N) \, \} \\
\tau_{vout} &= 1 + \max_{0 \le i < N_{out}} \{ \, \mathit{Height}(\mathbf{ov}, X_i.N) \, \} \\
\tau_{valid} &= \max(\tau_{vin}, \; \tau_{vout}) \\
\tau_{cmpl} &= \mathit{Height}(\mathbf{ce}, \; N_{in} + N_{out}) \\
\tau_{le} &= \mathit{makeOdd}(\, \tau_{valid} + \tau_{cmpl} \,) \\
\tau_{cycle} &= 2 \times (\tau_{le} + 2)
\end{aligned}$$

DDD estimates the dynamic energy consumption of a circuit by counting the number of gates that switch during every cycle, and noting the number of inputs for each gate. The energy consumed by standard gates (nor, nand, and C-elements) can be cataloged, by the number of inputs, for the fabrication technology ahead of time. DDD can then access this information to perform energy estimations for use in its final clustering phase, where multiple PCHBs can be grouped into a single PCHB to reduce energy consumption. As will be discussed further in Chapter 5, the changes in energy consumption of pulldown networks during clustering are insignificant compared to the changes in the validity tree and completion networks. Therefore DDD's estimates will be sufficiently accurate since these components comprise mostly standard gates.

In addition to cycle time, the size of a PCHB can be limited by the number of transistors in series in its pulldown computation network. The upper limit on the number of transistors required in series is set by the fanin of a PCHB process. A production rule guard containing the logic "$A.0 \wedge A.1$" is nonsensical—it will never evaluate to true for e1ofN channels. Hence, if a PCHB has N_{in} input channels, the number of transistors in series in a computation network can be no more than $N_{in} + 2$, where the extra two transistors are the local enable *en* and the output channel enable "feet" transistors.

3.7 Summary

We have presented an overview of the synthesis of asynchronous VLSI systems and introduced the precharged half-buffer as the basic building block of high-performance quasi delay-insensitive design. After demonstrating the traditional compilation of PCHBs, we introduced general templates for PCHBs both with and without conditional communications. We defined the requirements for CHP processes to be implemented as asynchronous pipeline stages, and have set the stage for demonstrating how DDD decomposes programs into networks of fine-grained PCHB circuits. After demonstrating traditional compilation of PCHBs, we introduced circuit templates for general PCHBs both with and without conditional communications. Finally, we illustrated how these templates can be used to estimate the cycle time, energy, and size limitations of any PCHB process generated by DDD.

Chapter 4

DDD Optimizations for Asynchronous VLSI

This chapter presents a set of modifications to basic DDD (as described in Chapter 2) that optimize target systems implemented as hardware, and specifically as networks of asynchronous PCHB circuits. In practice, asynchronous designers usually set maximum cycle times for systems, and then work to achieve the minimum energy consumption for that cycle time. DDD can use PCHB templates to accomplish the same goal automatically.

We begin by showing that all processes generated by the DSA and projection phases of DDD can either be implemented as a PCHB, or easily rewritten to be implemented as a PCHB. In terms of performance, these first two phases of DDD need only produce decomposed processes that, when implemented as PCHBs, individually meet the desired cycle time. The DSA transformation already helps control the size of DDD processes. Processes with wide channels that encode, for instance, bytes of information typically need to be vertically decomposed to meet desired cycle times. In rare cases where vertically-decomposed processes are still not fast enough, designers must consider function decomposition to allow DDD to "horizontally decompose" the process and solve the problem.

Most of the modifications in this chapter focus on reducing the switching activity and hence the energy consumption of the decomposed system. (The remainder either isolates expensive computations to avoid redundancy, or isolates memories, which are implemented using specialized circuits.) Communications consume the bulk of energy in asynchronous systems, and so we present methods to reduce both the number and the activity factors of communication channels required by the decomposition. QDI circuits consume no dynamic energy when they are idle, and so we also introduce techniques to reduce the activity of computational processes. The DDD modifications are presented in chronological order, and are integrated within basic DDD before, after, and in between the DSA and projection phases.

4.1 DDD Generation of PCHBs

Recall the list of requirements for CHP processes in the hardware form "P_{init};*[P_{loop}]" to be directly implementable as PCHBs (Definition 10). The main requirements are that there be no nested loops, that P_{init} not contain any input communications, that no more than one communication is executed on any channel during a single iteration, and that all input communications precede all output communications in every iteration. DDD creates CHP processes that either meet these requirements or can be easily rewritten to meet them. This section addresses each PCHB requirement.

Before projection, DDD rewrites the original program by converting it into DSA form, removing nested loops, and inserting both copy and projection variables. Projection is a syntactic translation. Thus, regarding the first requirement, since nested loops have been removed from the sequential program, none of the decomposed processes can contain nested loops. Therefore, all of their subprograms P_{init} and P_{loop} are terminating programs. Similarly, for the second requirement, since we have limited the initial code P_{init} of the sequential program to containing regular assignments and output communications only, no decomposed processes can contain input communications in their initial code. This fulfills the second requirement for a process to be directly implementable as a PCHB.

Whether or not DDD processes satisfy the last two PCHB requirements depends upon the type of process under consideration. Every process in the eventual DDD concurrent system corresponds to either a single regular variable or to a single channel port in that pre-projection sequential program. We consider the two possibilities separately.

4.1.1 Variable Processes

By the DSA transformation, DDD processes that represent regular variables assign only one value to that variable per iteration: variables with more than one assignment executed per iteration of the original main loop are split into multiple DSA variables.

Input and output channels for DDD-variable processes either exist in the original sequential specification, or were inserted by projection and are internal to the decomposed system. In the case of original channels, output channels are always isolated in their own channel processes (Section 2.4.1). If original input channels are used more than once per main-loop iteration, DDD separates them into their own process (Section 2.4.2.1). Therefore any original channels that appear in a DDD-variable process are input channels and are used at most once per iteration.

Now consider the case of channels inserted by projection. If variable x depends upon variable a then the projection phase inserts a communication on internal channel “A_X” immediately following any assignment to a (Section 2.4.2). Since the program is in DSA form, there can be only one assignment to a executed during any iteration of the main loop. Hence, there can be only one communication on internal channel A_X during any iteration of the main loop. Therefore, any internal channels inserted by projection that appear in a DDD-variable process (such as a process P_x for x) are used at most once per iteration.

It remains to show that all input communications precede all output communications in every iteration of a DDD-variable process. Given a DDD process for variable x, all input channels receive variables used in the computation of x and all output channels send the results of the assignment to x to other variables for use in their assignments (Section 2.4.3). By construction, every variable is assigned a value before it is used in every iteration (Section 2.3.5). Thus, all input communications in the variable process must precede the assignment to x, which must precede all output communications.

Therefore DDD-variable processes fulfill all necessary CHP requirements to be directly implemented as asynchronous pipeline stages (Definition 10).

4.1.2 Channel Processes

DDD-channel processes exist for both input and output channels from the initial CHP specification. If the initial specification includes multiple communications in sequence on a channel per iteration, then the DDD process will also execute multiple communications in sequence on that channel per iteration. As such, DDD-channel processes may not at first fulfill the asynchronous pipeline stage requirements given above, but can be rewritten to do so.

Consider a DDD process P_X for channel X, where X is used in multiple communications per iteration. First note that X is the only channel in this process that may have multiple communications per iteration. As reasoned in the previous section, channels from the original sequential program are isolated in their own DDD processes when they are used multiple times and, by construction, internal channels inserted by projection can be used at most once per iteration. To rewrite this process so that it may be directly implemented as a PCHB, we introduce state in such a way that only one communication on X is performed per state, and only one state is executed per iteration. This transformation goes as follows.

Let $P_X \equiv P_{init}; \texttt{*[}P_{loop}\texttt{]}$. We begin by assuming that P_{init} and P_{loop} are straightline series; selection statements can be easily incorporated into the basic transformation later. Let S_X be any communication on channel X. Let T be any statement in parallel with S_X.

For every S_X and T, insert a semicolon so that if X is an input channel $T \prec S_X$ and if T is an output channel then $S_X \prec T$. This reordering preserves reaching definitions because there can be no data dependencies between parallel statements in a deterministic CHP program. By Theorem 1, the new process is program-equivalent to the original process P_X since we have only reordered statements and not changed the operations performed by any statements, For example,

$$A?a;\;\; X!a,\;\; B?b;\;\; X!b \quad \triangleright \quad A?a;\;\; X!a;\;\; B?b;\;\; X!b$$

Now there is a strict order relation between any communication on X and all other statements in the code.

We group all statements in the process into three classes: input communications S_I, regular assignment statements S_A, and output communications S_O. If statements belonging to different classes are in parallel, insert semicolons between them establishing class order relations $S_A \prec S_O \prec S_I$. (We assume that the program environment is designed so that this does not introduce deadlock.) Now the only statements that may be in parallel with each other in the process belong to the same class. Again, since the program is deterministic, reaching definitions are preserved and the transformed process is program-equivalent to the original. For example,

$$\begin{aligned} & R?r;\ \ s := f(r);\ \ P?p,\ \ q := g(r),\ \ S!s;\ \ Q!g(p,q),\ \ T!p \\ \rhd\ & R?r;\ \ s := f(r);\ \ q := g(r);\ \ S!s;\ \ P?p;\ \ Q!g(p,q),\ \ T!p \end{aligned}$$

Consider P_{loop}. It can now be written as

$$(P_I)^1;\ \ (P_A)^1;\ \ (P_O)^1;\ \ (P_I)^2;\ \ (P_A)^2;\ \ (P_O)^2;\ \ \ldots;\ \ (P_I)^N;\ \ (P_A)^N;\ \ (P_O)^N$$

where P_I is a sequence containing only input communications S_I, P_A is a sequence containing only regular assignments S_A, and P_O is a sequence containing only output communications S_O. Each individual sequence may be empty and contain no statements at all. If X is an input channel then communications S_X belong in sequences P_I; if X is an output channel then communications S_X belong in sequences P_O. No sequence can contain more than one communication on X. For example, if X is an output channel then the code "$(S_X)^n; (S_X)^{n+1}$" may be written as "$(P_I)^k; (P_A)^k; (P_O)^k; (P_I)^{k+1}; (P_A)^{k+1}; (P_O)^{k+1}$" where $(P_O)^k \equiv (S_X)^n$, $(P_O)^{k+1} \equiv (S_X)^{n+1}$, and $(P_I)^k$, $(P_A)^k$, $(P_I)^{k+1}$ and $(P_A)^{k+1}$ are empty sequences. This provides a clear separation for different communications on the same channel.

We can now introduce state to the process through a new variable s, which is initialized to zero and incremented after every sequence of output communications $(P_O)^n$, except the last, where it is reset to zero. Including assertions, the process can be rewritten as follows:

```
$P_X \;\equiv\; P_{init};\;$ *[ $P_{loop}$ ]

$P_X \;\triangleright\; P_{init},\; s := 0;$

        *[ $\{s = 0\}\; (P_I)^1;\; (P_A)^1;\; (P_O)^1;\; s := 1;$
           $\{s = 1\}\; (P_I)^2;\; (P_A)^2;\; (P_O)^2;\; s := 2;$
           $\{s = 2\}\; \ldots\; s := N - 1;$
           $\{s = N - 1\}\; (P_I)^N;\; (P_A)^N;\; (P_O)^N;\; s := 0$
        ]
```

The added assignments to s do not affect any reaching definitions from the original code, and so the rewritten process is program-equivalent to the old process. It is easy to see that we may go one step further and transform the body of the program from a straightline series to a selection statement:

```
$P_X \;\triangleright\; P_{init},\; s := 0;$

        *[ [ $s = 0 \longrightarrow\; (P_I)^1;\; (P_A)^1;\; (P_O)^1;\; s := 1$
           [] $s = 1 \longrightarrow\; (P_I)^2;\; (P_A)^2;\; (P_O)^2;\; s := 2$
           [] $s = 2 \longrightarrow\; \ldots\; s := N - 1$
           [] $s = N - 1 \longrightarrow\; (P_I)^N;\; (P_A)^N;\; (P_O)^N;\; s := 0$
        ] ]
```

Now every iteration of the main loop executes only one guarded command. By construction, all input communications precede all output communications in each guarded command. Recall that in a DDD-channel process for X, no channel other than X can be used in multiple communications. By construction, no guarded command contains more than one communication on X. If X is an output channel and there are multiple communications on X within the initial code P_{init}, all sequences "$(P_I)^k; (P_A)^k; (P_O)^k$" with $k > 1$ are moved into P_{loop} and given their own guarded command. (Input communications do not exist in P_{init}.) Thus, this transformation generates a process that satisfies the third and fourth requirements of Definition 10 and can, under electrical constraints, therefore be directly implemented as a PCHB.

Further transformations can create more efficient PCHBs. Consider the following example:

```
    *[ $A?a,\ B?b;\ X!f(a);\ C?c;\ d := g(b,c);\ X!h(d)$ ]
$\triangleright$ $s := 0;$
    *[ [ $s = 0 \longrightarrow\ A?a,\ B?b;\ X!f(a);\ s := 1$
       [] $s = 1 \longrightarrow\ C?c;\ d := g(b,c);\ X!h(d);\ s := 0$
     ] ]
```

The new process is program-equivalent to the original and satisfies the conditions for asynchronous pipeline stages. However, the value of variable b is assigned in one iteration and used in the next, requiring explicit storage using state bits either within the PCHB or on a feedback loop outside the PCHB. If the process is rewritten again as

```
    $s := 0;$
    *[ [ $s = 0 \longrightarrow\ A?a;\ X!f(a);\ s := 1$
       [] $s = 1 \longrightarrow\ B?b,\ C?c;\ d := g(b,c);\ X!h(d);\ s := 0$
     ] ]
```

then no extra state bits are required.

In general then, the transformation of DDD-channel processes can also reorder statements so that they are assigned to sequences (and thus guarded commands) in such a way as to minimize the number of state bits required. Usually, input communications are moved as late as possible without changing reaching definitions, while output communications are moved as early as possible without changing reaching definitions. Assignment statements, which typically have multiple variables on their RHS but only one on their LHS, are usually moved as early as possible (without changing reaching definitions). This is an attempt to place assignments in the same guarded command as the assignments to the variables on their RHS.

We have shown that all CHP processes generated by the DSA and projection phases of DDD can be rewritten to be directly implementable as asynchronous pipeline stages.

4.2 Isolating Hardware Units

This section presents the first modification of the DDD method for asynchronous VLSI. It introduces techniques for DDD to handle "expensive" system resources such as memories and arithmetic execution units. The circuits that implement memories or register files are more complicated than regular PCHBs, and should be implemented separately. We therefore introduce a methodical transformation that isolates arrays (the usual CHP representation of memories). The PCHBs required to compute the functions can be quite large, and should be treated as a system resource to be re-used in time as opposed to repeated in area. Our modification to DDD therefore allows designers to flag functions that should be isolated for re-use during decomposition. Both modifications are applied before DSA conversion and projection.

4.2.1 Arrays

We have previously shown how DDD can rewrite statements using arrays as selection statements treating each array element as a separate variable (Section 2.3.6). When the sequential program containing these selection statements are decomposed, each array element receives its own process, with dedicated channels between each of them and the array index process. This solution is usually prohibitively expensive for systems that are to be implemented in hardware. In CHP specifying circuit behavior, arrays usually represent system memories or register files, structures implemented using specialized circuits that do not fit the PCHB template.

For hardware design, we therefore isolate arrays at the very beginning of synthesis. DDD accomplishes this by rewriting the original sequential program and inserting variables in such a way that after the first two phases of DDD are applied, the entire array is in its own process that can be synthesized separately from the rest of the system. The target array process for some array A includes a control channel OP_A, an index channel IDX_A, a read channel R_A, and a write channel W_A. Its CHP is as follows:

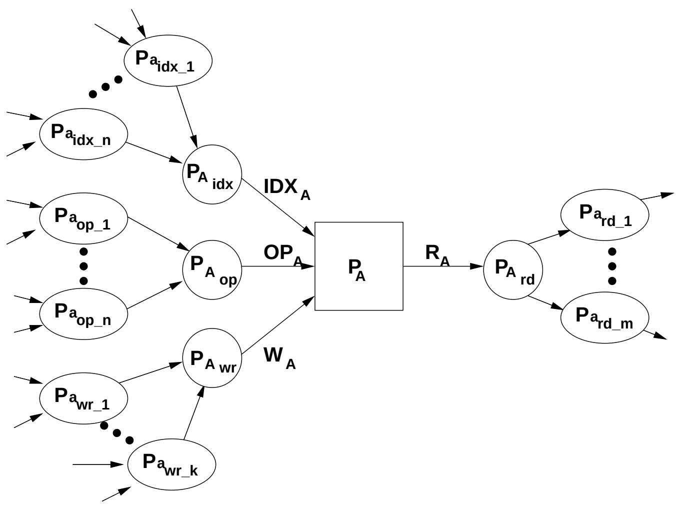


Figure 4.1: Isolating arrays.
System decomposed from sequential program containing k write operations to and $m = n - k$ read operations from array A.

```
$P_A \equiv$ *[ $OP_A?op,\ \ IDX_A?i;$
        [ $op = \textbf{read} \longrightarrow\ \ R_A!A$[$i$]
        ▯ $op = \textbf{write} \longrightarrow\ \ W_A?A$[$i$]
      ] ]
```

DDD rewrites the sequential program P by splitting it into two concurrent processes P_A and $P_{\bar{A}}$, where the first process is as given above and the second is the sequential program with array A excised. The program $P_{\bar{A}}$ is generated by transforming array manipulations from P into communications on the array channels. This involves the insertion of new variables a_{op}, a_{idx}, and a_{rd} or a_{wr} into the program. Assignments where arrays appear on the RHS are turned into array read operations:

$$v := f(\ A[g(i)],\ j\ )$$

$$\triangleright\ a_{op} := \textbf{read},\ \ a_{idx} := g(i);\ \ OP_A!a_{op},\ \ IDX_A!a_{idx};\ \ R_A?a_{rd};\ \ v := f(a_{rd}, j)$$

Assignments to array elements are turned into array write operations:

$$A[g(i)] := f(v)$$

$$\triangleright\ a_{op} := \textbf{write},\ \ a_{idx} := g(i),\ \ a_{wr} := f(v);\ \ OP_A!a_{op},\ \ IDX_A!a_{idx},\ \ W_A!a_{wr}$$

While the inclusion of special variables (a_{op}) as well as special channels ($OP_A!$) may seem redundant, the reasoning is as follows. Consider an array A that is used multiple times in the original program. If the DDD processes for the array channels such as $OP_A!$ are to be directly implementable as PCHBs, they require the state transformation presented in Section 4.1.2. In order to avoid any single DDD process's requiring excess circuitry that could increase its cycle time and thus the system cycle time, DDD attempts to separate computations from state-holding bits. Thus, a separate variable a_{idx} allows computations of functions such as $g(i)$ to be isolated in a non-state-holding process. Similarly, if array accesses occur within selection statements, a separate variable a_{op} allows the computations of the guard conditions to be separated from the state-holding process for channel OP_A. If such caution proves unnecessary, the excess processes will be efficiently recomposed, using methods presented in Chapter 5.

Consider a sequential program P containing n statements using an array A: k write operations and $m = n - k$ read operations. Figure 4.1 illustrates the system that results from applying the array transformation described here followed by DSA conversion and projection. Note that processes $P_{A_{op}}$, $P_{A_{idx}}$, $P_{A_{wr}}$ and $P_{A_{rd}}$ all contain state-holding bits to distinguish between the multiple array accesses required in one iteration of the original program.

4.2.2 Functional Units

DDD decomposes programs using variables as the basic units, not functions. If the same function is used in computing values for different variables, different processes in the decomposed system may end up looking very similar. On the other hand, if all variables queued up to use the one process that executed every instance of a function, expensive state-holding mechanisms would be required, and lack of resources could slow the entire system down. Our compromise is to allow designers to specify which functions—typically expensive arithmetic units—should be isolated and reused by different variables, while letting all others be repeated throughout the system as needed.

Functions can be handled using an approach similar to that for array read operations. For example, we assume that a function f with two arguments will be isolated by an unconditional process

$$P_f \;\equiv\; \texttt{*[}\; FA?f_a,\;\; FB?f_b;\;\; FOUT!f(f_a, f_b)\; \texttt{]}$$

We therefore set up processes that merge the various arguments and forward them on channels FA and FB, and another process that reads the function evaluation result on $FOUT$ and splits it for the result variables. Thus, when function f is flagged in the code below but function g is not:

$$\ldots\;\; v := f(h_1(a), h_2(b)) \;\;\wedge\;\; g(c,d);\;\; w := f(h_2(r), h_1(s)) \;\;\vee\;\; g(t,u)\;\; \ldots$$

$$\triangleright\;\; \ldots\;\; f_a := h_1(a), f_b := h_2(b);\;\; FA!f_a,\;\; FB!f_b;\;\; FOUT?f_{out};\;\; v := f_{out} \wedge g(c,d);$$

$$f_a := h_2(r), f_b := h_1(s);\;\; FA!f_a,\;\; FB!f_b;\;\; FOUT?f_{out};\;\; w := f_{out} \vee g(t,u)\;\; \ldots$$

After DSA conversion and projection, the system includes the following processes:

$$P \;\;\triangleright\;\; P_f \;\;\|\;\; P_{FA} \;\;\|\;\; P_{FB} \;\;\|\;\; P_{FOUT} \;\;\|\;\; P_v \;\;\|\;\; P_w \;\;\|\;\; P_{fa_1} \;\;\|\;\; P_{fa_2} \;\;\|\;\; P_{fb_1} \;\;\|\;\; P_{fb_2}$$

$$P_{fa_1} \;\equiv\; A_{fa_1}?a;\;\; FA1_{FA}!h_1(a)$$

$$P_{fa_2} \;\equiv\; B_{fa_2}?b;\;\; FA2_{FA}!h_2(b)$$

$$P_{fb_1} \;\equiv\; R_{fb_1}?r;\;\; FB1_{FB}!h_2(r)$$

$$P_{fb_2} \;\equiv\; S_{fb_1}?s;\;\; FB2_{FA}!h_1(s)$$

$$P_{FA} \;\equiv\; FA1_{FA}?f_a;\;\; FA!f_a;\;\; FA2_{FA}?f_a;\;\; FA!f_a$$

$$P_{FB} \;\equiv\; FB1_{FB}?f_b;\;\; FB!f_b;\;\; FB2_{FB}?f_b;\;\; FB!f_b$$

$$P_{FOUT} \;\equiv\; FOUT?f_{out};\;\; FOUT_v!f_{out};\;\; FOUT?f_{out};\;\; FOUT_w!f_{out}$$

$$P_v \;\equiv\; FOUT_v?f_{out},\;\; C_v?c,\;\; D_v?d;\;\; v := f_{out} \wedge g(c,d)$$

$$P_w \;\equiv\; FOUT_w?f_{out},\;\; T_w?t,\;\; U_w?u;\;\; w := f_{out} \wedge g(t,u)$$

Note that the evaluation of function f appears in one process only, while the evaluation of g appears in both P_v and P_w.

4.3 Reducing System Communications

Reducing the number of communications in a system can greatly reduce the energy consumption. Of course, we cannot change the specification of the original sequential program, and so the communications on external channels must remain the same. However, we can encode expressions in new variables to reduce the number of new channels required in the decomposed system, and we can make communications on other channels conditional.

These measures decrease energy consumption in three ways:

- by decreasing the actual number of channels in the system;
- by reducing the wire load that is switched per cycle;
- by making entire modules conditional.

The first two ways are described in this section; the last is addressed in Section 4.4. The transformations occur after the DSA phase of DDD, and before or during projection.

4.3.1 Encoding Guard Expressions

Our first technique encodes guards of selection statements in fewer variables. The purpose of the transformation is to reduce the number and size of physical channels required in the decomposed system, given that every variable assigned a value within a selection statement depends upon the variables in guard conditions. For example, consider the following process.

```
*[ $G_0?g_0, G_1?g_1, G_2?g_2, G_3?g_3;\ \ A?a, B?b, C?c;$
   [ $f(g_0, g_1, g_2, g_3) \longrightarrow\ \ X!(a \wedge b), Y!(b \wedge c), z := a \wedge c$
   []$\neg f(g_0, g_1, g_2, g_3) \longrightarrow\ \ z := b \vee c$
   ];  $Z!(a \vee z)$
 ]
```

Would there be more or fewer channels in the decomposed system if the guard conditions were encoded as follows?

```
*[ $G_0?g_0, G_1?g_1, G_2?g_2, G_3?g_3;\ \ A?a, B?b, C?c;$
    $h := f(g_0, g_1, g_2, g_3);$
    [ $h \longrightarrow\ \ X!(a \wedge b),\ \ Y!(b \wedge c),\ \ z := a \wedge c$
    ▯$\neg h \longrightarrow\ \ z := b \vee c$
    ]; $Z!(a \vee z)$
 ]
```

The answer depends on the size of the variables g_i and changes depending on both the number of variables assigned a value in the selection (three: $X!$, $Y!$ and z), and on the number of guarded commands in the selection statement (two).

We begin encoding guards by assigning a communications cost to every variable in the sequential code. A variable that can hold K different values can be communicated on a 1ofK channel. Practically speaking, we always break channels up into a group of channels of more manageable size (e.g., 1 byte variables are not communicated on a 1of256 channel but rather on four 1of4 channels). Let us choose some base channel size 1of$B$. Normally, $B = 4$, but any reasonable value (say, $B \leq 6$) can be chosen for this purpose. Consider a variable x that can assume K_x different values. The channel required to communicate x can be implemented as $\lceil \log_B K_x \rceil$ different 1of$B$ channels. This variable is therefore assigned a communications cost of $C_x = \lceil \log_B(K_x) \rceil$.

Scanning through the sequential program, for every selection statement G, we have

- V_G = set of all guard variables in the selection;
- N_G = # conditions in the selection;
- A_G = # variables assigned a value within the selection.

Let h be the variable that encodes the guard conditions. Now, let us compute C_{enc} (the communications cost when guard conditions are encoded in h), and C_{unenc} (the cost when they are left unencoded):

$$C_{V_G} = \sum_{\forall v_i \in V_G} C_{v_i}$$

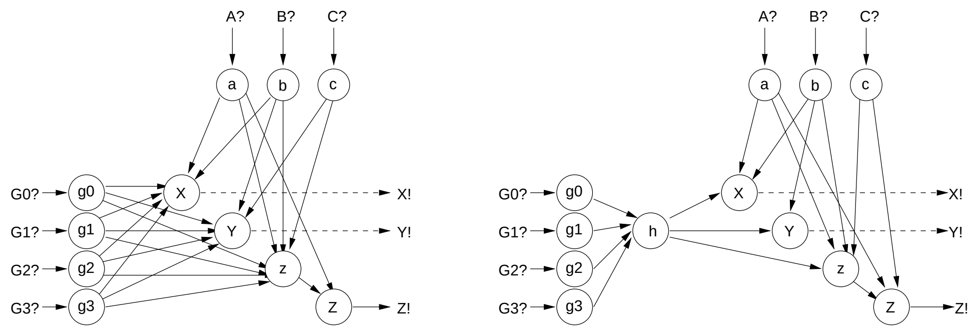


Figure 4.2: Systems without and with guard encoding.

$$C_h = \lceil \log_B(N_G) \rceil$$

$$C_{unenc} = C_{V_G} \cdot A_G$$

$$C_{enc} = C_{V_G} + C_h \cdot A_G$$

If $C_{enc} < C_{unenc}$ then we know to encode the guard conditions of the selection in question. If not, we leave the selection unencoded. The systems in Figure 4.2 demonstrate the possible communications savings when guards using g_i are encoded in h using the technique described here.

Returning to our example, we see that $N_G = 2$ and $A_G = 3$. Let $V_{v_i} = 4$ for $\forall v_i \in V_G$. Then $C_{V_G} = 4$, $C_h = 1$, $C_{unenc} = 12$, and $C_{enc} = 7$. In this case, encoding the guard conditions reduces the communications cost of the selection by almost half! In contrast, when $V_a = V_b = 4$, the process

*[$A?a,\ B?b;$ [$a \wedge b \longrightarrow\ x\uparrow$ [] $\neg a \vee \neg b \longrightarrow\ x\downarrow$]]

is an example of a selection for which it is better not to encode the guard conditions ($C_{unenc} = 2$, $C_{enc} = 3$).

The encoding of guard conditions takes place after an initial DSA transformation but before projection. Nested selection statements should be flattened before guard encoding.

4.3.1.1 Removing Nested Selections

Nested selection statements are allowed in CHP but are often not necessary. For example, the selection statement

$[\ g \longrightarrow\ x := f(a);\ \ [\ h \longrightarrow\ \ Y!x\ \ [\!]\ \neg h \longrightarrow\ \ \textbf{skip}\ ]$

$[\!]\ \neg g \longrightarrow\ \ x := 0;\ \ [\ h \longrightarrow\ \ Y!x\ \ [\!]\ \neg h \longrightarrow\ \ \textbf{skip}\ ]$

$]$

can be rewritten as

$[\ g\ \wedge\ h \longrightarrow\ \ x := f(a);\ \ Y!x$

$[\!]\ g\ \wedge\ \neg h \longrightarrow\ \ x := f(a)$

$[\!]\ \neg g\ \wedge\ h \longrightarrow\ \ x := 0;\ \ Y!x$

$[\!]\ \neg g\ \wedge\ \neg h \longrightarrow\ \ x := 0$

$]$

To keep manipulations—including guard encoding—of selection statements and their variables semantically clear, DDD removes any selection nesting where the guards of the inner selection are not conditionally input in different communication patterns in outer levels of nesting. For example, the nesting

$[\ g \longrightarrow\ \ x := f(a),\ \ H1?h;\ \ [\ h \longrightarrow\ \ Y!x\ \ [\!]\ \neg h \longrightarrow\ \ \textbf{skip}\ ]$

$[\!]\ \neg g \longrightarrow\ \ x := 0,\ \ H2?h;\ \ [\ h \longrightarrow\ \ Y!x\ \ [\!]\ \neg h \longrightarrow\ \ \textbf{skip}\ ]$

$]$

must remain intact. The removal of nested selections is performed prior to guard encoding.

4.3.1.2 Removing Basic Selections

Similarly to unnecessarily nested selections, basic selection statements are also often unnecessary. For example, the code

$[\ g \longrightarrow\ \ x := f(a,b)\ \ [\!]\ \neg g \longrightarrow\ \ x := f(c,d)\ \ ]$

contains no conditional communications and so can be easily rewritten as

$x := f'(g,a,b,c,d)$

If such a communicationless selection statement contains assignments to multiple variables, it is wise to check if guard encoding results in a better system before rewriting the selection statement. This removal can therefore take place after guard encoding, or in each individual process following projection. Then, no selection statements exist in the code unless they contain communication statements.

4.3.2 Conditional Communications

We can reduce the energy consumption of a system by reducing the activity level on some of its internal channels. This can be accomplished within the DDD framework by altering the way in which copy variables are inserted in the sequential program during projection (Section 2.4.2). Consider a variable v on the LHS of an assignment that appears within a selection statement G. Instead of inserting copy variables for every variable that depends upon v, DDD can insert copy variables only for variables that use v in the same branch as the assignment. Variables that use v in other branches of G do not require copies of v within the current branch.

Consider the following program, which is already in DSA form:

```
$P \equiv$ *[ $G?g,\ A?x_0;\ \ Y!x_0;$
      [ $g = 0 \longrightarrow\ \ B?b;\ \ x_1 := f_1(x_0, b);\ \ C?c;\ \ x_2 := f_2(x_1, c)$
      [] $g = 1 \longrightarrow\ \ B?b;\ \ W!b,\ \ x_2 := x_0$
      [] $g = 2 \longrightarrow\ \ x_2 := x_0$
      ]; $Z!x_2$
    ]
```

We want to ensure that we send defined values of variables only when they are actually used in the computation implemented by the receiving module. To illustrate, we see that in P the variable x_1 and the channels $W!$ and $Z!$ all depend upon b. However, x_1 is only assigned a value when $g = 0$ and $W!b$ is only executed when $g = 1$. We therefore place the projection assignments for intermediate channels B_{x_1} and B_W as follows:

```
*[ $G?g,\ \ A?x_0;\ \ Y!x_0;$
    [ $g = 0 \longrightarrow\ \ B?b;\ \ B_{x_1}!b,\ \ B_{x_1}?b_{x_1};$
                  $x_1 := f_1(x_0, b_{x1});\ \ C?c;\ \ x_2 := f_2(x_1, c)$
    ▯ $g = 1 \longrightarrow\ \ B?b;\ \ B_W!b,\ \ B_W?b_W;\ \ W!b_W,\ \ x_2 := x_0$
    ▯ $g = 2 \longrightarrow\ \ x_2 := x_0$
    ]; $Z!x_2$
 ]
```

After projection, the process implementing assignments to variable b is

```
$P_b \equiv$ *[ $G_b?g;$
        [ $g = 0 \longrightarrow\ \ B?b;\ \ B_{x_1}!b$
        ▯ $g = 1 \longrightarrow\ \ B?b;\ \ B_W!b$
        ▯ $g = 2 \longrightarrow$ skip
     ] ]
```

4.4 Reducing System Computations

This section describes a post-decomposition synthesis phase that "tidies up" the system and reduces energy consumption. There are two transformations involved: *distillation*, where unnecessary control structures are removed from the system; and *elimination*, where processes that do not perform any computation or copy function are removed.

4.4.1 Motivation

Given a sequential program, DDD produces a concurrent system of processes. Ideally, every process serves a purpose whether it be performing a computation, splitting or merging channels, or even just copying values. However, the first two phases of DDD can produce systems with processes that are nothing more than simple L-R buffers. DDD may well end up inserting such buffers in a throughput optimization during its clustering phase, but we do not wish to add unnecessary constraints on where the buffers should appear. DDD therefore removes these extra buffers from the decomposed system. (Any deadlock that arises because of this removal of slack is also handled during DDD's clustering phase.) This *elimination* is performed in a separate phase after projection because new buffers can be created by a more complicated post-projection transformation, called distillation.

Distillation removes unnecessary control structures from the system. This can result in the removal of guard variables and the channels on which they are communicated, reducing the system's dynamic energy consumption. The energy reduction extends beyond fewer communications, to fewer computations as well. If only one channel is used in the execution of a process iteration and all others are conditionally suppressed, that process still consumes dynamic switching activity in its computation stages (recall that conditional outputs are implemented through an additional computation stage output) and its completion tree (which suppresses the generation of conditional input enables but must still generate the local enable signal *en*).

When a process contains a selection statement where one of the guarded commands is "$g \rightarrow \textbf{skip}$" with no actions, if the selection statement can be properly removed, then the process no longer consumes any dynamic switching activity in its computation stages or completion tree when the condition g is true. Thus the channels carrying the guard variables have been eliminated, and computations have been eliminated from the system too. This is the motivation for distillation, and the following sections describe the situations in which selections can be "properly removed," and the act of removing them.

As a simple illustration, consider the following system:

$$
\begin{aligned}
\mathit{SYSTEM} &\equiv P_{ctrl} \;\|\; P_{send} \;\|\; P_{recv} \\
P_{ctrl} &\equiv *[\; G?g;\; G_{send}!g,\; G_{recv}!g \;] \\
P_{send} &\equiv *[\; G_{send}?g,\; A?a;\; [\; g \longrightarrow B!f(a) \;[\!]\; \neg g \longrightarrow \textbf{skip} \;]\;] \\
P_{recv} &\equiv *[\; G_{recv}?g;\; [\; g \longrightarrow B?b;\; C!h(b) \;[\!]\; \neg g \longrightarrow \textbf{skip} \;]\;]
\end{aligned}
$$

Communications only occur on channel B when g is true. In fact, processes on both sides of the channel check this condition before performing their input or output statement. One of these checks is redundant: if we envision a channel as a pipe with valves on both ends, only one of the valves need be closed for the flow through the pipe to cease. If we choose to distill P_{recv}, then the resulting system is equivalent:

$$SYSTEM \;\triangleright\; SYSTEM2 \;\equiv\; P2_{ctrl} \;\|\; P2_{send} \;\|\; P2_{recv}$$

$$P2_{ctrl} \;\equiv\; *[\; G?g;\;\; G_{send}!g \;]$$

$$P2_{send} \;\equiv\; *[\; G_{send}?g,\;\; A?a;\;\; [\; g \longrightarrow \;\; B!f(a) \;\; [\!] \;\; \neg g \longrightarrow \;\; \mathbf{skip} \;]\;]$$

$$P2_{recv} \;\equiv\; *[\; B?b;\;\; C!h(b) \;]$$

Now elimination can be performed on the new L-R buffer $P2_{ctrl}$, reducing energy consumption in the system even further:

$$SYSTEM \;\triangleright\; SYSTEM3 \;\equiv\; P3_{send} \;\|\; P2_{recv}$$

$$P3_{send} \;\equiv\; *[\; G?g,\;\; A?a;\;\; [\; g \longrightarrow \;\; B!f(a) \;\; [\!] \;\; \neg g \longrightarrow \;\; \mathbf{skip} \;]\;]$$

$$P2_{recv} \;\equiv\; *[\; B?b;\;\; C!h(b) \;]$$

Distillation and elimination have transformed a system with three processes (two with expensive conditional communication circuitry) that are all active whenever an input arrives on $G?$, into an equivalent system of two processes (only one with conditional communication circuitry) where one is active whenever an input arrives on $G?$ but the other is active only when that input evaluates to true.

In general, when there are conditions on both sides of a channel, we prefer to remove the conditions from the receiving end, leaving a conditional output on the sending end. The reason for this is that conditional inputs are more expensive to implement than conditional outputs. Conditional inputs require an entirely new computation stage that switches whether the condition is true or not, as well as extra gates in the completion tree to generate individual conditional input enables. In contrast, conditional outputs simply require an extra computation stage output whose path to ground only consumes dynamic energy when the condition is false, and an increase in size of a validity gate. Experiments on a simple system using e1of2 channels indicate that during distillation, a system that keeps its conditional input consumes 45% more energy than a system that keeps its conditional output.

4.4.2 Distillation

Distillation eliminates unnecessary control from a system by either removing selection statements from processes or moving conditional computations to new processes with fewer levels of nested selections. Aside from guard variable channels eliminated along with selections, the communication traces for individual channels in the system are unchanged by distillation because of redundancy in the control of systems generated by DDD.

In a DDD-generated system, when an internal channel is used conditionally in one process, the same condition is also always checked in the process on the other end of the channel. An internal communication arises from single assignment in the sequential code; the input and output projection communications inserted by DDD replace this assignment and thus are always performed under the same condition. Upon decomposition, any guard variables that compute the communication's condition are sent to the processes representing both the variable being sent and the variable being received. For example, consider the DDD transformations below:

$$
\begin{array}{l}
\quad *[\ldots\ [\ g \longrightarrow\ b := a\ [\!]\ \neg g \longrightarrow\ \textbf{skip}\ ]\ \ldots] \\
\triangleright\ *[\ldots\ [\ g \longrightarrow\ A_b!a,\ A_b?b\ [\!]\ \neg g \longrightarrow\ \textbf{skip}\ ]\ \ldots] \\
\triangleright\ *[\ldots\ [\ g \longrightarrow\ A_b!a\ [\!]\ \neg g \longrightarrow\ \textbf{skip}\ ]\ \ldots] \\
\quad \|\ *[\ldots\ [\ g \longrightarrow\ A_b?b\ [\!]\ \neg g \longrightarrow\ \textbf{skip}\ ]\ \ldots]
\end{array}
$$

In the final system, communications on internal channel A_b are checked for the condition $g = \mathit{true}$ in both processes attached to its ports. (DDD also assumes that conditions are checked on both sides of external channels; the user must override this assumption if it is not the case.) When distilling a process, DDD can therefore focus solely on the behavior of the process under consideration, and can ignore the behavior any other processes.

A selection statement can be eliminated without affecting a system's computations if its guards are used only to distinguish between action and non-action, and not between different communication patterns or computations when action is specified. Since processes on both sides of a channel evaluate the conditions under which communications occur, the action/non-action checks in the receiving process are redundant, and the selection statement (including guard variables and channels) can be removed from the receiving process, if not used for other purposes.

Formally, consider a process P that contains a channel C. Let us define the guard condition $P.C.G$ to be the boolean condition under which C is used in P. If C does not appear in any selection statements, then $P.C.G \equiv \mathbf{true}$. If C does appear in selection statements, then $P.C.G$ is a combination of the guard conditions for the commands in which it is used. (Nested selections lead to a conjunction of guard conditions while C's being used in multiple branches of the same selection leads to a disjunction of guard conditions.)

DDD performs distillation by creating a dependency graph for process P. This graph contains one node for each channel in P. Edges can exist only between an input channel node and an output channel node. If a variable received on an input channel is used in a computation for a value sent on an output channel, *not including selection statement guard conditions*, then an edge exists between the nodes representing those two channels. The resulting bipartite graph indicates the data dependencies between process channels, but ignores control dependencies that could be eliminated along with selection statements during distillation.

The dependency graph may be composed of multiple unconnected subgraphs. Each maximally-connected subgraph is called a process *component*, and each component is considered separately for distillation. Assume for now that P contains no nested selections. A component can be distilled if:

- all channels $P.C$ in the component have equivalent guard conditions $P.C.G$, and
- output channels in the component always send values computed by the same function(s) when their guard condition is true.

The first requirement is that control variables do not distinguish between different communication patterns. The second requirement is that control variables do not distinguish between different computations. If both of these requirements are fulfilled, then the component can be distilled. If the component is one of multiple components, then we say that the process is *partially distilled*, and the component is decomposed into its own process, with no selection statement. If there is only one component for the entire process, then the process is distilled and the selection statement eliminated.

For example, let

```
$P \equiv$ *[ $G?g;$
        [ $g = 0 \longrightarrow$   $A?a;$  $X!f(a)$
        ▯ $g = 1 \longrightarrow$   $A?a,$  $B?b;$  $X!f(a),$  $Y!b$
        ▯ $g = 2 \longrightarrow$   $B?b;$  $Y!(\neg b)$
      ] ]
```

There are two components for this process: one consisting of $A?$ and $X!$; the other consisting of $B?$ and $Y!$. The component containing A and X fulfills the two requirements for distillation given above, but the component containing B and Y does not because, although $P.B.G \equiv P.Y.G \equiv g = 1 \vee g = 2$, the function output on Y differs for that channel's two active cases. Since the conditional communications on A are controlled by the process on the other side of the channel, and the guard variable g is not needed to distinguish between any other behavior (whether communications patterns or computations) of A or the other channels in its component, then the component containing A can be distilled. This results in the following system:

```
*[ $A?a;$  $X!f(a)$ ]  $\|$
*[ $G?g;$  [ $g = 0 \longrightarrow$ **skip** ▯ $g = 1 \longrightarrow$  $B?b;$  $Y!b$ ▯ $g = 2 \longrightarrow$  $B?b;$  $Y!(\neg b)$ ] ]
```

P has been partially distilled.

Let us now move on to consider processes with nested selections. Distillation considers the removal of selections one nesting level at a time, starting with the outermost selection and moving inwards. To determine whether distillation can remove a selection level, we create a dependency graph for each nested level, omitting edges for guard variables only at that or at outer levels. Each connected component can only be distilled if, at the current level of nesting:

- the projection of each channel's guard conditions on variables used in the guards of the current nested selection are all equivalent, and
- output channels in the component always send values computed by the same function(s) in

communications *at this level of nesting.* If no functions are used at this level and all communications take place in inner selections, then the second condition is considered to be fulfilled.

Once distillation is not possible at a level of nesting, no further distillation is possible at nesting levels that are further in.

For example, consider the process

$$
\begin{array}{l}
Q \equiv *\texttt{[}\ \ G?g; \\
\qquad \texttt{[}\ \ g \longrightarrow\ \ A?a,\ \ H?h;\ \ \texttt{[}h = 0 \longrightarrow\ \ X!f_1(a)\ \ \texttt{[]}h = 1 \longrightarrow\ \ X!f_2(a)\ \ \texttt{[]}h = 2 \longrightarrow \textbf{skip}\ \ \texttt{]} \\
\qquad \texttt{[]}\ \ \neg g \longrightarrow\ \ \textbf{skip} \\
\quad \texttt{]}\ \ \texttt{]}
\end{array}
$$

Distillation begins by considering the outermost selection statement. The component at this level includes channels $A?$, $X!$, and $H?$. ($H?$ is not used for guard variables at this level of nesting, and is considered part of the component.) Now,

$$
\begin{array}{lcl}
Q.A.G & \equiv & Q.H.G \ \equiv \ g \\
Q.X.G & \equiv & g \wedge (h = 0 \vee h = 1)
\end{array}
$$

The projection of the guard conditions of all channels in this component on guard variable g are equivalent:

$$Q.A.G \lceil g \ \equiv \ Q.H.G \lceil g \ \equiv \ Q.X.G \lceil g \ \equiv \ g$$

There are no communications on the only output channel X at this level, so there are no functions to compare. Therefore, distillation can remove the outer selection, resulting in the following new process:

$$*\texttt{[}\ \ A?a,\ \ H?h;\ \ \texttt{[}h = 0 \longrightarrow\ \ X!f_1(a)\ \ \texttt{[]}h = 1 \longrightarrow\ \ X!f_2(a)\ \ \texttt{[]}h = 2 \longrightarrow\ \ \textbf{skip}\texttt{]}\ \ \texttt{]}$$

Moving onto the next level of nesting, $H?$ is now a guard channel, and the new dependency graph has a single component consisting of channels A and X. When the guard conditions of these channels are projected on variable h, they are not equivalent: $Q.A.G\lceil h \equiv \textbf{true}$ and $Q.X.G\lceil h \equiv (h = 0 \vee h = 1)$. No further distillation is possible.

Transformation	No. Processes	No. Channels
Initial DDD	15	39
Elimination I	13	37
Distillation	13	35
Elimination II	11	33

Figure 4.3: Results of elimination and distillation on MiniMIPS WriteBack unit.

4.4.3 Elimination

After the distillation transformation has been applied to the system, we can search for and remove processes that are no more than L-R buffers. In fact, this transformation can be performed in conjunction with distillation. A process is a L-R buffer if it is unconditional, has only one input and one output channel, and always computes the identity function. Elimination reduces the energy consumption of a system, and possibly the overall latency as well.

4.4.4 Example

We applied the elimination and distillation transformations to a system decomposed by DDD that implements the WriteBack unit from the asynchronous MiniMIPS. This unit has multiple control channels, and a two-byte datapath. After DDD, there were 15 processes and 39 channels in the decomposed system. Elimination removed two processes and two channels from the system immediately. Two more processes were then distilled (deleting two control channels), resulting in simple buffers that were then eliminated (removing two more data channels from the system). After the transformations, the new system comprised 11 processes and 33 channels. Of the six channels that were deleted, one was a two-byte datapath channel, one was an $e1of6$, two were $e1of2$ channels, and two more were $e1of3$ channels. These results are summarized in Figure 4.3.

4.5 Future Work

There are other program transformations not completely implemented by DDD that are helpful when synthesizing asynchronous circuits.

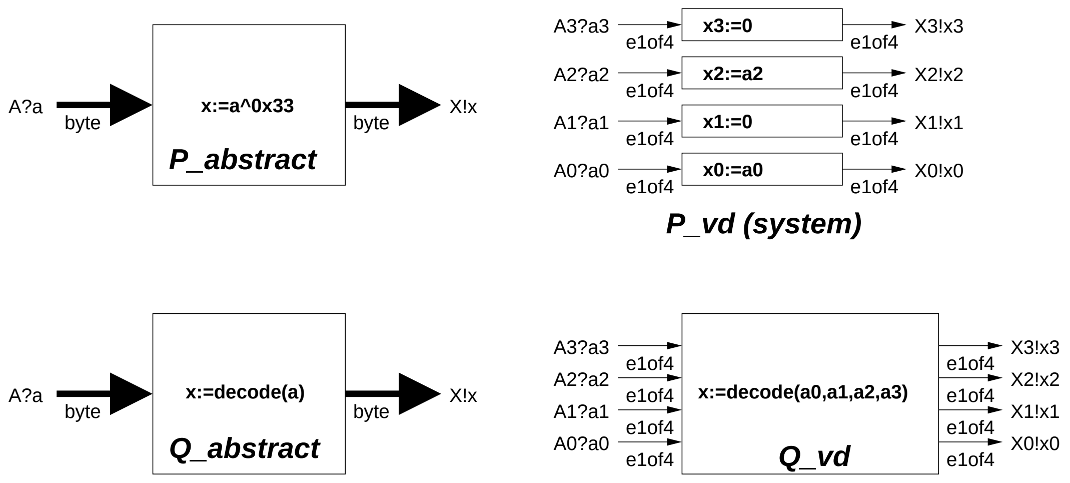


Figure 4.4: Vertical decomposition of decomposable (P) and non-decomposable (Q) processes.

The first such transformation is *vertical decomposition*, which is performed immediately before guard encoding. If a CHP process contains variables that are byte- or word-sized, their variables and their channels are usually split into hardware-implementable e1of4 encodings instead. For example, a byte is sliced into four e1of4 variables encoding two bits each. If the computations performed on the variable are decomposable into separate computations for each e1of4 slice, then the process is also broken into pieces, with the control variables copied to all of them. Otherwise, the process remains intact but the channels are still split into separate entities. Both of these situations are illustrated in Figure 4.4. Currently, DDD does not determine whether functions are separable or not, and the user must rewrite the function for the smaller variables manually.

Other transformations that may be added to DDD in the future include the pipelining of single functions, and the use of arbiters to handle non-determinism. If a single computation is deemed too large to fit into a PCHB circuit with a given cycle time, function decomposition can be performed to break the computation into pieces that can be assigned to new intermediate variables. If non-deterministic selection statements exist in the original code, their results can be stored in a variable and that variable used as a guard in the subsequent deterministic selection statement. The non-deterministic selection can then be isolated and removed, leaving behind a deterministic locally slack elastic system to which DDD can be applied. The synthesis of the non-deterministic process involves arbiter circuits and can be handled separately from DDD, much as array memories are now.

4.6 Summary

We have presented modifications to DDD that are tailored to produce systems that can be implemented as energy- and area-efficient asynchronous QDI circuits. The entire method, including the modifications, is as follows:

1. **Isolation of Arrays and Large Compute Functions**
2. **DSA Transformation**
   (a) Early decomposition of nested loops
   (b) Transformation of straightline series and selection statements
3. **Vertical Decomposition**
4. **Guard Encoding**
5. **Projection**
   (a) Build dependency sets
   (b) Insert copy variables and projection communications
   (c) Build new dependency and projection sets, apply projection
6. **Distillation**
7. **Elimination**

Note that arrays and expensive functions must be isolated before the DSA transformation because their isolation introduces new channels and variables to the sequential program. If their isolated process is accessed more than once per sequential iteration, the channels and variables will be affected by the DSA transformation. Meanwhile, the DSA transformation and vertical decomposition can be performed in any order relative to each other, but must both precede the guard encoding step introduced in this chapter. Guard encoding depends upon the number of variables assigned values in a selection statement, and this number can be affected by both DSA and vertical decomposition. Lastly, the distillation technique can only be applied after a concurrent system has been projected from the sequential program. Since distillation can convert complicated processes into simple ones, to eliminate L-R buffers in a single pass, elimination follows distillation.

Chapter 5

Clustering Asynchronous Processes

We have presented the first two phases of DDD, where sequential programs are decomposed into systems of concurrent processes, each implementable as a PCHB. We now focus on DDD's last phase, clustering. Clustering recomposes the DDD-generated processes into larger processes to improve system efficiency. It is performed in conjunction with slack matching, a performance optimization for asynchronous systems. The global optimization problem that combines recomposition and slack matching is complex; DDD addresses it with the randomized heuristic of simulated annealing. This chapter describes the circumstances under which individual processes can and should be recomposed, the system model used to implement slack matching, a new method for slack matching homogeneous systems, and the cost functions used in the simulated annealing.

5.1 Motivation

One way of optimizing asynchronous systems is to specify a maximum cycle time before process decomposition, and then to strive to create a decomposed system that meets the target with minimum global energy consumption. The first two phases of DDD limit processes to performing computations for only one variable, and hence usually produce systems where each individual process can easily meet reasonable cycle time constraints. We define the *individual cycle time* of a process to be the cycle time when its environment consists solely of simple L-R buffers.The system's cycle time is different, and can depend on the amount of available slack.

Communication actions typically consume much more energy than computations, especially in

QDI systems with four-phase handshakes. It is therefore important to select a process granularity for the decomposed system that strikes a balance between processes' being small enough to achieve the desired cycle time and their being large enough that communications overhead is not needlessly large. For example, an assignment comprising multiple branches of a selection statement and containing significant logic in both the assigned and guard expressions certainly forms the basis for a process being large enough to stand on its own in the final decomposed system. However, in CHP used to design asynchronous chips, assignments are often unconditional and no more complex than a single static assignment. Basic processes that do little more than input two values and output their conjunction do not offer enough computation to be worth the three communications performed on the computation's behalf.

DDD systems are in fact often over-decomposed: multiple processes can be combined into one and still meet realistic target cycle times. The clustering phase of DDD presented here seeks out such situations and recomposes processes to reduce energy consumption and latency in the new system. Clustering is performed assuming that the processes are implemented as PCHB circuits; DDD uses the PCHB template to estimate the cycle time and energy consumption of a process given only the high-level CHP specification.

Breaking the original circuit specification into small processes only to regroup some of them back into larger processes may seem wasteful, but there are two major advantages to this approach. The first is that while we generally cluster processes to minimize communications and energy consumption, other metrics can be chosen by designers instead, requiring only a change in the cost functions used during the clustering heuristic. For example, in a hypothetical system where Vdd can be set independently for each module, DDD clustering can be used to minimize the system's energy complexity, Et^2. (This metric is independent of voltage to first order, and thus significant when comparing circuits that function in environments where voltage scaling is performed to trade energy off against performance [40].)

The second advantage of recomposition is that the performance optimization of slack matching can be applied simultaneously to the same decomposed system of processes. While the first two phases of DDD may produce processes that can individually fulfill cycle-time constraints, the system as a whole may not contain enough buffering, or slack, to meet the global performance target. We therefore insert L-R buffers at select points of the system, and attempt to do so while adding as little energy consumption as possible. If clustering were performed before slack matching, processes could be recomposed only for a chain of buffers to be later inserted in their place, at a greater energy cost. Linking the two transformations eliminates such unhelpful cancellations from the synthesis flow.

We begin this chapter by defining system models and describing methods for performing the slack-matching optimization. We then describe situations in which process recomposition is advantageous, and demonstrate how low-level circuit information is extracted from high-level CHP specifications to compute the performance costs and energy savings of clusters. Finally, we introduce the clustering heuristic for DDD that combines recomposition with slack matching, describe approaches to special cases. Results from our prototype clustering tool are presented in Chapter 6.

5.2 Recomposition of PCHB Processes

Let us consider the individual action of process recomposition. Using the circuit template given in Chapter 3, we can study the effects of combining two PCHB processes. Our current goal is to lower the energy consumption of the concurrent system. Recomposition can achieve this in two ways:

- reducing the number of communications (and hence the amount of switching) in the circuit;
- removing redundancy by merging processes whose circuits have identical parts.

We specify the situations where combining processes leads to significant energy savings within the circuit. We also present size limits for recomposed PCHBs.

5.2.1 Clustering in Series

Recomposition can significantly reduce energy when a communication channel exists between the two processes being clustered. This scenario is illustrated in Figure 5.1. If the composition of the two original computations is not too large for a single PCHB (as will be discussed later in this section),

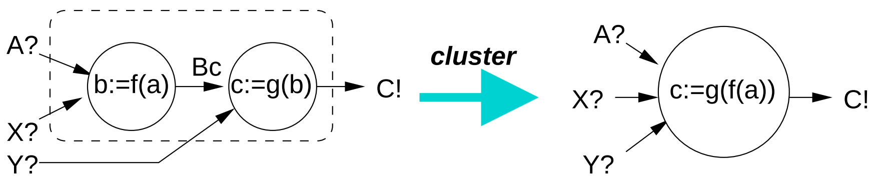


Figure 5.1: Clustering two processes in series.

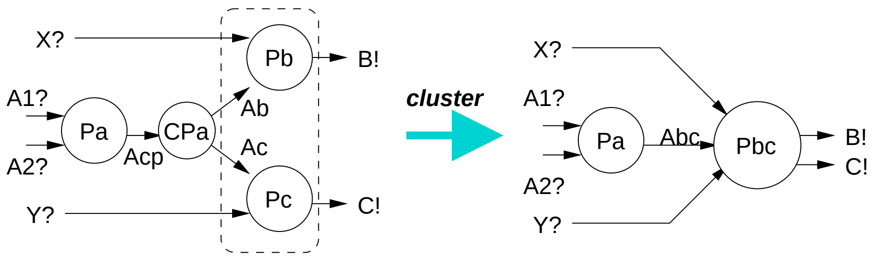


Figure 5.2: Clustering in parallel two processes that share inputs.

then combining the two processes eliminates a communication channel and the validity trees on both sides of the channel. The wider the channel, the more beneficial the recomposition.

5.2.2 Clustering in Parallel

Consider the case where two processes share the same input, as illustrated in Figure 5.2. In the basic decomposed system, the inputs appear on different channels originating from the same copy process. If the two processes are merged, one of the channels, and possibly the copy process itself, will be eliminated. This alone contributes towards the first goal of clustering for low energy.

There are further savings from clustering processes that share inputs. The two original processes used redundant validity trees for this input. Merging them eliminates one copy of the tree, fulfilling the second goal of recomposition. Thus, combining basic processes that share the same input leads to energy savings. If the processes are the only ones to which that input value is sent (and therefore a copy process is eliminated), or if the input value is particularly wide (and therefore has large validity trees), this clustering is especially desirable.

Since there are no shared variables or shared communication channels in our concurrent systems, processes can never share the same output. If their output channels have a common destination process, however, their output enable signals $R.e$ can be shared. No internal circuit savings arise from this sharing, but externally we need to route only one acknowledge signal instead of two. This eliminates switching on one wire and can simplify the routing problem, but does not have significant impact on the energy consumption of the system. Our clustering heuristic groups together processes with characteristics that offer greater reductions in energy; DDD focuses on clustering processes that share inputs or are arranged in series.

5.2.3 Limits to Cluster Size

A process is deemed too large for hardware implementation if either of the following conditions apply:

- it has too many transistors in series, or
- its validity/completion trees have too many levels to be implemented in a fast asynchronous pipeline stage.

The first problem is straightforward: if the computational logic is complex then the circuit may have more pulldown transistors in series than is advisable in the target technology for synthesis. The pulldown network should not require more transistors in series than the sum of foot transistors (at most one for the internal enable and one for the enable of the output channel being computed) and the total number of input channels to the process. To determine the number of transistors in series more precisely, we can find the maximum number of conjunctions in the assigned and guard expressions for communications on each channel, and add the largest of these values to the number of foot transistors.

When two processes are combined because they share input channels, their pulldown networks remain separate since their outputs are not shared. The maximum number of transistors in series therefore remains the same. However, when two processes are clustered in series, the pulldown network for the computational logic can grow in size. Such a recomposition may be disallowed because it exceeds the maximum number of transistors in series.

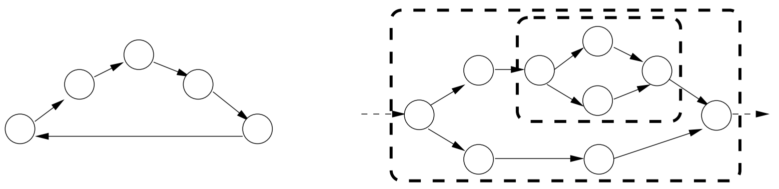


Figure 5.3: Basic slack matching structures.
Ring of PCHBs (left) and Reconvergent Fanout (two nested cases, right).

The second limit to the size of a recombined process involves the maximum cycle time specified by the designer prior to decomposition. Deep completion or validity trees in a process could increase its cycle time, possibly slowing down the entire system. The depth of these trees depends on both the size of their fanin (which in turn depends upon the width and number of channels), and on the maximum number of transistors allowed in series (which limits the fanin for each individual gate). All of the information required to estimate a recomposed process's cycle time can be determined from its CHP specification (see Chapter 3).

5.3 Slack Matching

Slack matching is a system optimization where simple L-R buffers are inserted into a QDI system to increase system throughput. The slack matching solutions presented in this section are only for individual structures with slack constraints, as illustrated in Figure 5.3. The two types of slack-matching structures are *rings* and *reconvergent fanout* (composed of a fork process, a join process, and parallel branches in between the two.)

The more general problem of slack matching systems composed of multiple structures is addressed in the following section, as part of the global clustering heuristic. Not all asynchronous systems can be slack matched without duplication of processes. While the class of such unsolvable systems has not been rigorously defined, we have shown elsewhere that the presence of supercycles of slack constraints (groups of more than two rings or reconvergent fanout structures that share edges) is a necessary condition for insolvability [54].

Slack matching is often described as analogous to the retiming of synchronous VLSI systems for optimal throughput [27, 17, 43, 30]. However, the pipeline dynamics arising from asynchronous handshakes make it more complex. While retiming can be performed in polynomial time, the general slack matching problem has been proved to be NP-complete [24].

The *static slack* of a pipeline is the maximum number of messages that can be introduced to the pipeline (with none being removed) before deadlock occurs. A pipeline of N PCHB circuits has a static slack of $s = N/2$, because PCHB handshake protocols allow valid data on the rails of only alternating (and not adjacent) PCHBs composed in series. This is the origin of the name "half-buffer." A pipeline holding m messages is deadlock-free when $0 < m < s$. While static slack determines the number of messages that a system may hold without deadlock, the additional concept of *dynamic slack* determines the range of messages that can be held for optimal throughput.

This section introduces the FBI model for PCHBs that forms the basis of our slack analysis of asynchronous systems. We classify and present methods for determining the critical cycles of FBI systems, and then use this knowledge in dynamic slack-matching for optimal throughput in structures containing basic slack constraints: rings and reconvergent fanout. Separate techniques are presented for homogeneous systems (composed of PCHBs with identical timing characteristics) and heterogeneous systems (composed of PCHBs whose timing characteristics may differ).

5.3.1 FBI Delay Model for PCHBs

For the purposes of slack matching, we can characterize circuits using graphs whose vertices represent circuit elements (production rules or production rule sets) and whose edges represent transitions on circuit nodes. One example of such a graph is the "FBI delay model" for PCHB circuits. The name of this model is derived from the fact that it expresses the forward-latency cycles (F), backward-latency cycles (B), and internal cycles (I) of a system of PCHBs. We begin by focussing on rings of PCHBs each with only one input and one output channel, but the model can be extended to general PCHBs.

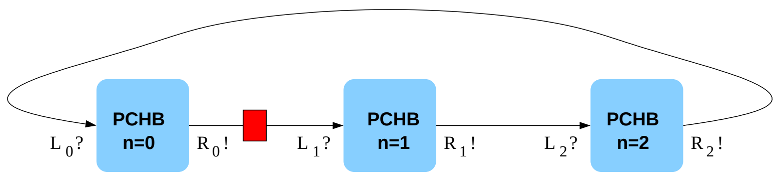


Figure 5.4: Ring of PCHB stages with N=3.

5.3.1.1 FBI Graphs

Consider a ring of three PCHB pipeline stages containing one message, as illustrated in Figure 5.4. To determine the critical cycle time of this system, we must represent the PCHBs using a more detailed model that includes handshake behavior. The FBI model corresponding to the ring in Figure 5.4 is illustrated in Figure 5.5.

Each vertex in the graph has a label ϕ_n, where

$$\phi \in \Phi \equiv \{fu,\ fd,\ bu,\ bd,\ iu,\ id,\ vu,\ vd\} \tag{5.1}$$

and $0 \leq n < N$ is a PCHB index in a ring of N PCHBs. The meanings of these labels in reference to a PCHB circuit are illustrated in Figure 5.6. The *delay* of the circuit element represented by a vertex ϕ_n is $\delta(\phi_n)$, and each vertex delay in the FBI model is defined as follows. Given a PCHB with index n, let the notation "$\texttt{[}B\texttt{]}$" indicate that boolean expression B evaluates to *true*:

- $\delta(fu_n)$ is the delay from $\texttt{[}en_n \wedge R.e_n \wedge L.d_n\texttt{]}$ to both $R.d_n\!\uparrow$ and $rv_n\!\uparrow$
- $\delta(vu_n)$ is the delay from $\texttt{[}L.d_n\texttt{]}$ to $lv_n\!\uparrow$
- $\delta(bd_n)$ is the delay from $\texttt{[}lv_n \wedge rv_n\texttt{]}$ to $L.e_n\!\downarrow$
- $\delta(id_n)$ is the delay from $\texttt{[}\neg L.e_n\texttt{]}$ to $en_n\!\downarrow$
- $\delta(fd_n)$ is the delay from $\texttt{[}\neg en_n \wedge \neg R.e_n\texttt{]}$ to both $R.d_n\!\downarrow$ and $rv_n\!\downarrow$
- $\delta(vd_n)$ is the delay from $\texttt{[}\neg L.d_n\texttt{]}$ to $lv_n\!\downarrow$
- $\delta(bu_n)$ is the delay from $\texttt{[}\neg lv_n \wedge \neg rv_n\texttt{]}$ to $L.e_n\!\uparrow$

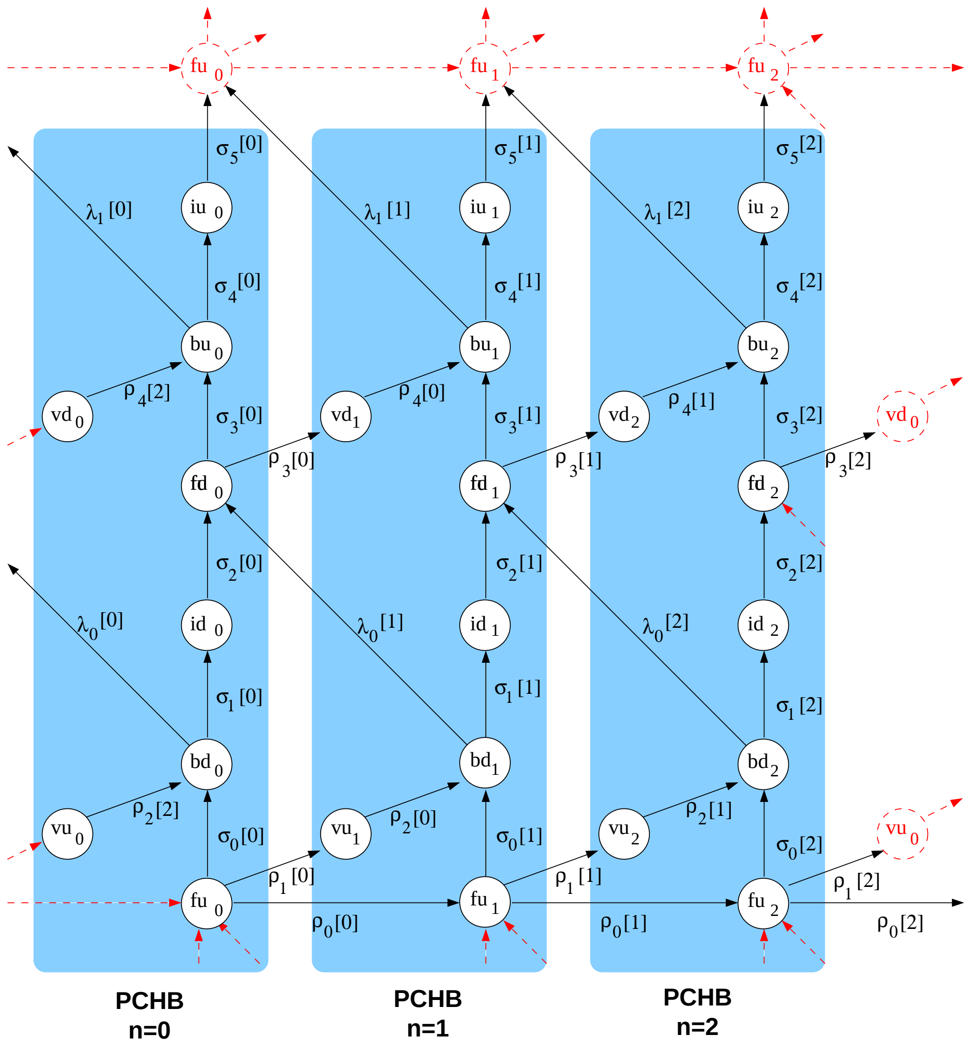


Figure 5.5: FBI model for ring of PCHB stages with N=3.
Dashed vertices and edges are redundant and included to demonstrate wraparound connections.

- $\delta(iu_n)$ is the delay from $[L.e_n]$ to $en_n\uparrow$

Delays are expressed in units of transition counts, and "$L.d$" represents the data rails of communication channel L. The use of the same delays fu_n and fd_n for the generation of both $R.d$ and rv is based on the practical assumption that the maximum channel size for circuits in a given technology is no greater than the maximum number of n-transistors allowed in series. Hence, the validity of an output channel can be computed in a single gate. The input-validity delays vu_n and vd_n are not similarly included with fu_n and fd_n because of the asymmetry of PCHB pulldown and pullup networks: the set phase for output data $R.d$ depends upon input data $L.d$ but the reset phase does

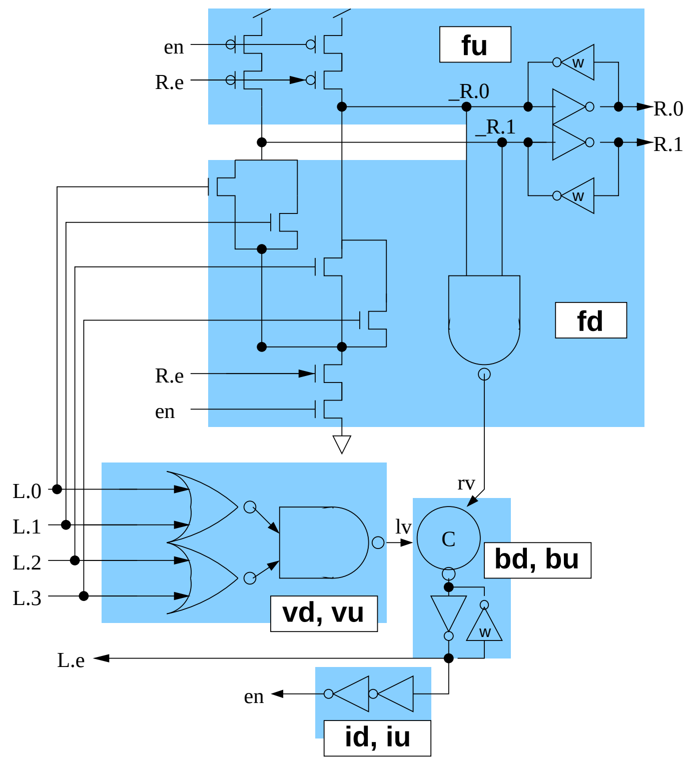


Figure 5.6: Illustrating the FBI vertices on a PCHB circuit.

not.

The edges in the graph are labelled with names $e(n)$, where $0 \leq n < N$ is the PCHB index and

$$e \in \varepsilon \equiv \{\lambda_0, \lambda_1, \sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \rho_0, \rho_1, \rho_2, \rho_3, \rho_4\} \tag{5.2}$$

The symbols in ε indicate whether an edge points to the PCHB on the left (λ_j), stays within the current PCHB (σ_j), or points to the PCHB on the right (ρ_j). Each edge has a *source* and a *sink* vertex. For example, in Figure 5.5, we have $src(\lambda_0\texttt{[}n\texttt{]}) = bd_n$ and $snk(\lambda_0\texttt{[}n\texttt{]}) = fd_{n-1}$. A *path* P is a sequence of edges $P = e_0 e_1 e_2 ...$, where $snk(e_j) = src(e_{j+1})$. For paths $P = e_0 e_1 ... e_L$, the *path source* is $src(P) = src(e_0)$ and the *path sink* is $snk(P) = snk(e_L)$. A *cycle* is a path P for which $src(P) = snk(P)$. A *simple cycle* P is a cycle for which all $src(e_j)$ are unique over all $e_j \in P$ (i.e., in which no vertex is appears more than once). The path delay of a path P is

$$\delta(P) = \sum_{j=0}^{L} \delta(src(e_j)) \tag{5.3}$$

5.3.1.2 Messages and Tokens

A ring with N PCHBs and static slack $s = N/2$ may contain $0 \leq m \leq s$ *messages.* Each message is represented in the ring's FBI system by multiple *tokens* travelling around forward-latency cycles, backward-latency cycles, internal cycles, and handshake cycles concurrently. A token on an edge in the FBI model represents a boolean condition that evaluates to *true* and is a conjunction term in the guard of a production rule that has not yet fired. (We require that disjunctive logic be encapsulated within an FBI vertex: a production rule whose guard contains disjunctions must be grouped in the same vertex with the production rules that set its guard variables.) When all fanin edges to a vertex contain tokens, the *true* conditions together contribute to enabling all production rules for that vertex. Figure 5.7 annotates each edge with the boolean condition that it represents.

Tokens traverse the graph as follows: if the fanin edges of a vertex all contain a token, a token is removed from each fanin edge and a token is placed on each fanout edge of the vertex. (This behavior is similar to that of Petri nets [44].) Thus, the number of tokens on a cycle is always conserved, and the number of tokens on any cycle at reset is the number of tokens flowing on that cycle when the system is in steady state.

Tokens are introduced to the system by reset circuitry. Specifically, *pre-tokens* are placed on edges corresponding to production rules that are initialized by the Reset signal. For example, given the production rule "$\mathit{Reset} \rightarrow en{\downarrow}$," a pre-token is placed on corresponding edge σ_2. Pre-tokens traverse the graph as regular tokens do. When the system has stabilized and is frozen at the end of the Reset phase (the progress of pre-tokens may be impeded by production rules such as "$\neg \mathit{Reset} \wedge \neg\_en \rightarrow en{\uparrow}$"), any pre-tokens at the inputs of production rules whose firings would be non-vacuous are converted into actual tokens.

Thus, we can specify the FBI model of a system of N PCHBs as $G = \langle V, E, R\rangle$, where, if we

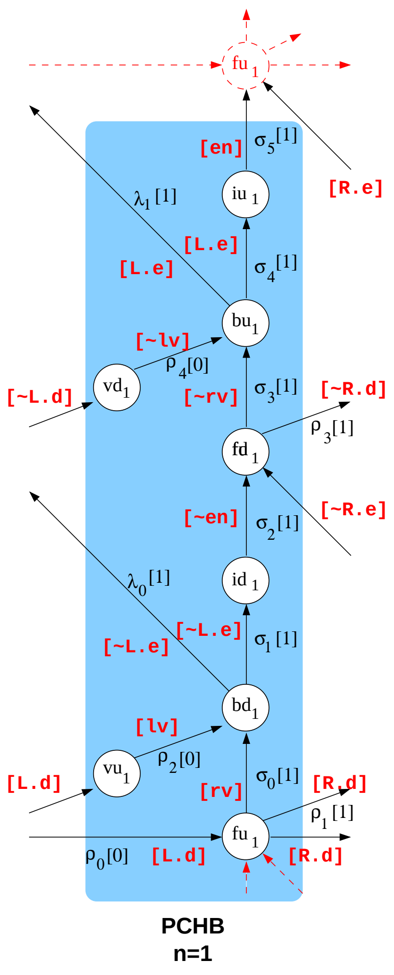


Figure 5.7: FBI graph for a PCHB annotated with edge boolean conditions for tokens. "L.d" represents valid data on the data rails of channel "L;" "L.e" is the enable (inverted acknowledgment) rail of channel "L."

define $S_N \equiv \{n \mid n < N \cap n \in \mathbf{N}\}$, then

- $V = \Phi \times S_N$ is the set of vertices
- $E = \varepsilon \times S_N$ is the set of edges
- $R \subseteq E$ is the set of edges on which tokens exist at the end of the Reset phase.

For PCHBs, each buffer is initialized either to send a message, to receive a message, or to a state with no message initialization. The location of tokens for these three cases of PCHBs at the end of the Reset phase is illustrated in Figure 5.8. For a PCHB with index n in an FBI graph, the following conditions hold:

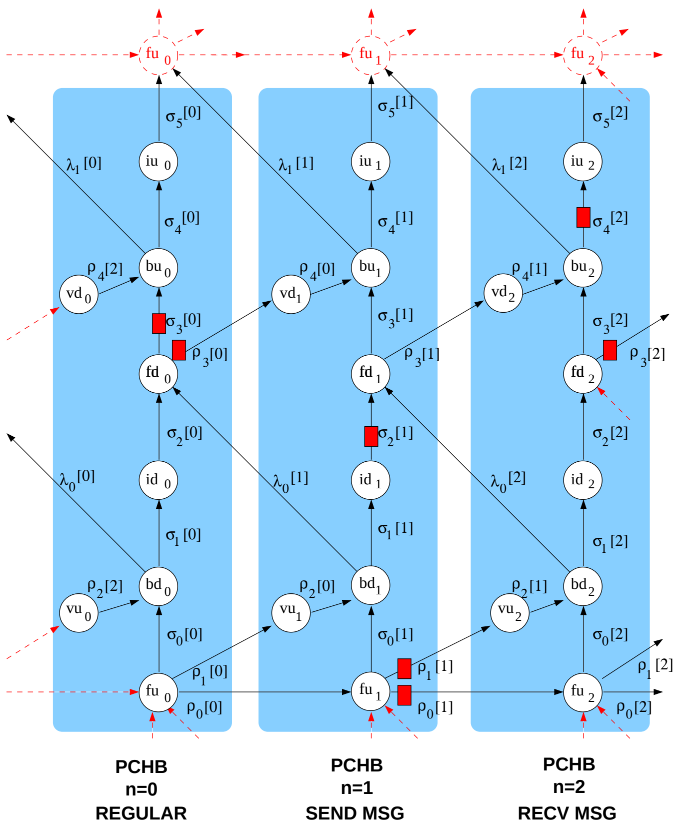


Figure 5.8: FBI graph for a PCHBs showing placement of initial tokens.

- $R_{reg}[n] \equiv \{\sigma_3[n], \rho_3[n]\}$
- $R_{snd}[n] \equiv \{\sigma_2[n], \rho_0[n], \rho_1[n]\}$
- $R_{rcv}[n] \equiv \{\sigma_4[n], \rho_3[n]\}$

5.3.2 Critical Cycles

A system's throughput depends on the length of its FBI cycles, and the number of tokens on these cycles.

5.3.2.1 Common Cycles

Lines [29] categorized the following cycles: forward-latency cycles, backward-latency cycles, internal cycles, and handshake cycles. Before proceeding with further analysis, we use the FBI model to define these commonly-mentioned cycles in asynchronous QDI systems. Other cycles exist in the FBI system and we explore them methodically in the following section, but many real-life PCHB systems fall into a category that renders these *common cycles* significant in determining the critical cycle time of a system.

Given a ring of N PCHBs that contains m messages, the *common forward-latency cycle* is $\tau_f^\dagger$:

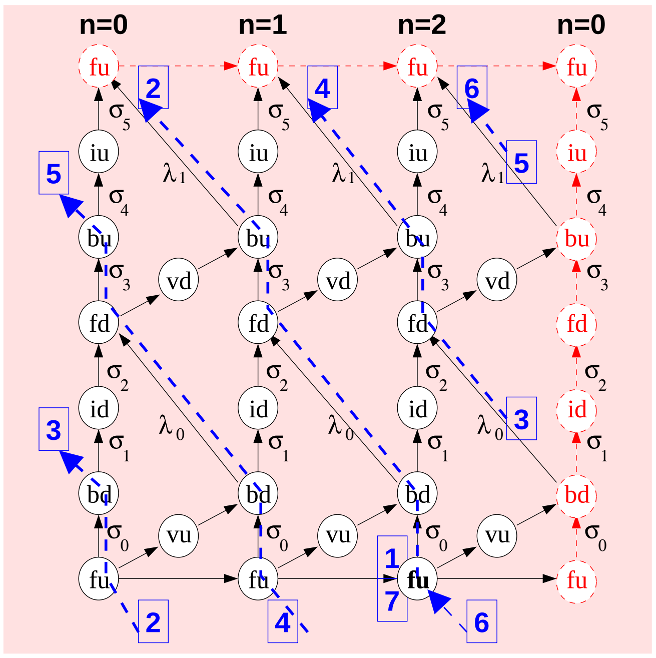
$$\begin{aligned} C_f &= \rho_0[0]\ \rho_0[1]\ \ldots \rho_0[N-1] \\ \delta_f^\dagger &= \delta(C_f) \\ \tau_f^\dagger &= \frac{\delta_f^\dagger}{m} \end{aligned} \tag{5.4}$$

(The notation $\delta_f^\dagger$ and $\tau_f^\dagger$ is used to distinguish these common cycle definitions from the more general definitions δ_f and τ_f presented in the following section.)

The *common backward-latency cycle* is traversed by message acknowledgment tokens moving backwards around the ring (i.e., in the direction of decreasing PCHB indices). Another perspective is that there are absences of messages, or $s-m$ "holes," moving backwards around the ring as the messages move forward. If N is odd, there is one cycle that traverses each PCHB twice. If N is even, there are two cycles that each traverse each PCHB once. These scenarios are illustrated in Figure 5.9.

$$\begin{aligned} C_{b_{odd}} &= \sigma_0[N-1]\ \lambda_0[N-1]\ \sigma_3[N-2]\ \lambda_1[N-2]\ \ldots\ \lambda_0[0] \\ &\quad\ \sigma_3[N-1]\ \lambda_1[N-1]\ \ldots\ \sigma_3[0]\ \lambda_1[0] \\ C_{b_{evn0}} &= \sigma_0[N-1]\lambda_0[N-1]\sigma_3[N-2]\lambda_1[N-2]\sigma_0[N-3]\ldots\sigma_3[0]\lambda_1[0] \\ C_{b_{evn1}} &= \sigma_3[N-1]\lambda_1[N-1]\sigma_0[N-2]\lambda_0[N-2]\sigma_3[N-3]\ldots\sigma_0[0]\lambda_0[0] \\ \delta_b^\dagger &= \begin{cases} \frac{1}{2}\delta(C_{b_{odd}}) & \text{if } N \bmod 2 = 1 \\ \max\{\delta(C_{b_{evn0}}), \delta(C_{b_{evn1}})\} & \text{if } N \bmod 2 = 0 \end{cases} \end{aligned} \tag{5.5}$$

$$\tau_b^\dagger = \frac{\delta_b^\dagger}{s-m} \tag{5.6}$$



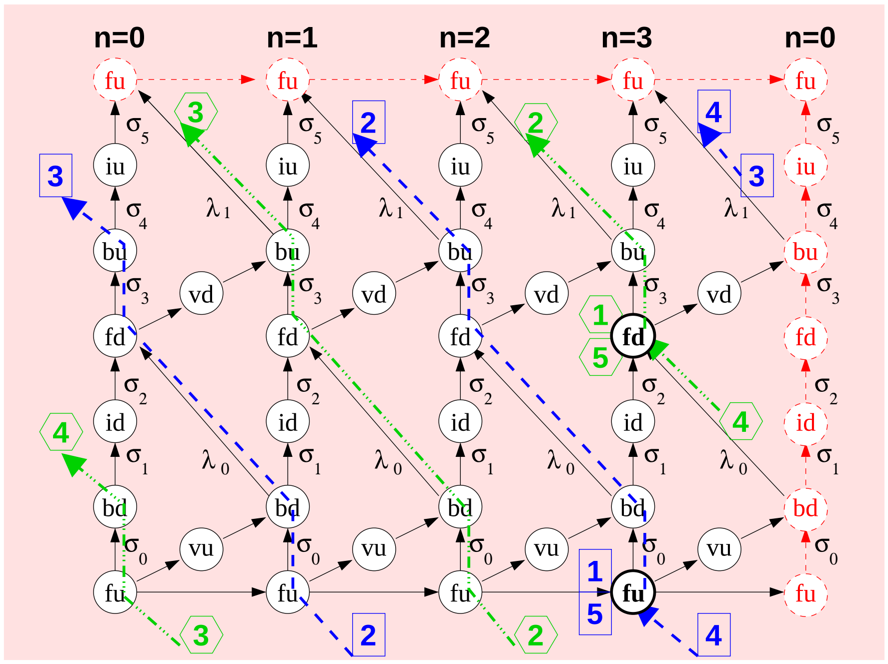


Figure 5.9: Backward-latency cycles.
The FBI system on the top has an odd number of PCHBs. Starting from vertex fu (n=2), a common backward-latency cycle that traverses each PCHB twice is marked on the graph. (The graph is a torus; follow the numbers on the path as they wrap around the edges.) The FBI system on the bottom has an even number of PCHBs. There are two common backward-latency cycles marked on the graph: one starting at vertex fu (n=3) and the other starting at vertex fd (n=3). Each traverses each PCHB once.

The *common longest internal cycle* is the longest cycle that remains within any single PCHB in the ring.

$$\begin{aligned} C_i[n] &= \sigma_0[n]\ \sigma_1[n]\ \sigma_2[n]\ \sigma_3[n]\ \sigma_4[n]\ \sigma_5[n] \\ \tau_i^\dagger &= \max_n\{\delta(C_i[n])\} \end{aligned} \tag{5.7}$$

Handshake cycles occur between two adjacent (communicating) PCHBs. The *common longest handshake cycle* is

$$\begin{aligned} C_h[n] &= \rho_1[n]\ \rho_2[n]\ \lambda_0[n+1]\ \rho_3[n]\ \rho_4[n]\ \lambda_1[n+1] \\ \tau_h^\dagger &= \max_n\{\delta(C_h[n])\} \end{aligned} \tag{5.8}$$

5.3.2.2 Critical Cycles

Consider an FBI system for a linear series of N PCHBs. If we begin by ignoring PCHB indices then, by inspection, all paths with source fu_n and sink fu_m are expressed by the following regular expression P:

$$P = (F^*C^*)^*$$

where

$$\begin{aligned} F &= \rho_0 \\ C &= [\sigma_0|\rho_1\rho_2]\ [\sigma_1\sigma_2|\lambda_0]\ [\sigma_3|\rho_3\rho_4]\ [\sigma_4\sigma_5|\lambda_1] \end{aligned}$$

Let us consider the regular expression C more closely. C always begins at some vertex fu_n in the FBI graph. If n is the index of the current PCHB in the ring, when traversing edges in the FBI system we have

$$\begin{array}{ccc}
\{n = K\} & \lambda_j & \{n = (K-1) \bmod N\} \\
\{n = K\} & \sigma_j & \{n = K\} \\
\{n = K\} & [\rho_0 \mid \rho_1\rho_2 \mid \rho_3\rho_4] & \{n = (K+1) \bmod N\}
\end{array}$$

Since there are a finite number of edges in any instance of C, we can assign all possible paths fitting the pattern C into one of the following subpatterns:

$$\begin{array}{ccc}
\{n = K\} & C_0 & \{n = K\} \\
\{n = K\} & C_{+1} & \{n = (K+1) \bmod N\} \\
\{n = K\} & C_{+2} & \{n = (K+2) \bmod N\} \\
\{n = K\} & C_{-1} & \{n = (K-1) \bmod N\} \\
\{n = K\} & C_{-2} & \{n = (K-2) \bmod N\}
\end{array}$$

We can now write our general pattern specification for paths with source fu_n as

$$P = (F^*[C_0|C_{+1}|C_{+2}|C_{-1}|C_{-2}]^*)^*$$

Paths that fit subpattern C_n traverse more than one edge, begin at fu_N, and end at fu_{N+n} without traversing any other nodes fu_m, where $m \neq N$ and $m \neq N + n$. Paths in subpattern F traverse only one edge $\rho_0\texttt{[}n\texttt{]}$.

Given this pattern for traversing an FBI graph, we find the simple cycle in the graph with the largest *cycle time*, which is defined as the cycle delay divided by number of tokens on the cycle. This section presents an analysis for a general heterogeneous ring of L-R buffers; the following sections make assumptions on the PCHBs to reduce the possible cycles put forth here into the four common cycles described in Section 5.3.2.1.

Case 1: Stationary Cycles $\{n = K\}\ C_0\ \{n = K\}$

Any pattern P that includes C_0 must include a cycle since $src(C_0) = snk(C_0) = fu_K$. Any path in C_0 must contain twice the number of edges ρ_j as it does edges λ_j. There are six such paths (all simple cycles) for any PCHB[n]:

$$\begin{aligned}
C_0^0[n] &= \sigma_0[n]\ \sigma_1[n]\ \sigma_2[n]\ \sigma_3[n]\ \sigma_4[n]\ \sigma_5[n]\\
C_0^1[n] &= \sigma_0[n]\ \sigma_1[n]\ \sigma_2[n]\ \rho_3[n]\ \rho_4[n]\ \lambda_1[n+1]\\
C_0^2[n] &= \rho_1[n]\ \rho_2[n]\ \lambda_0[n]\ \sigma_3[n]\ \sigma_4[n]\ \sigma_5[n]\\
C_0^3[n] &= \rho_1[n]\ \rho_2[n]\ \lambda_0[n]\ \rho_3[n]\ \rho_4[n]\ \lambda_1[n+1]\\
C_0^4[n] &= \rho_1[n]\ \rho_2[n]\ \sigma_1[n+1]\ \sigma_2[n+1]\ \sigma_3[n+1]\ \lambda_1[n+1]\\
C_0^5[n] &= \sigma_0[n]\ \lambda_0[n]\ \rho_3[n-1]\ \rho_4[n-1]\ \sigma_4[n]\ \sigma_5[n]
\end{aligned}$$

Analysis of the reset sets in Section 5.3.1.2 shows that all possible internal cycles C_0 contain one token. Thus, the maximum cycle time is as follows:

$$\tau_{C_0} = \max\left\{\delta(C_0^j[n])\ :\ 0{\le}n{<}N,\ 0{\le}j{<}6\right\} \tag{5.9}$$

Note that τ_{C_0} includes all common internal cycles C_i[n] and handshake cycles C_h[n] from Section 5.3.2.1.

Case 2: Consecutive C-iterations With Direction Change:

$[C_{+1}|C_{+2}][C_{-1}|C_{-2}]$ **or** $[C_{-1}|C_{-2}][C_{+1}|C_{+2}]$

All of the scenarios in this case reduce to stationary cycles. In other words, all eight paths in $[C_{+1}|C_{+2}][C_{-1}|C_{-2}]$ and $[C_{-1}|C_{-2}][C_{+1}|C_{+2}]$ contain a simple cycle c, where either $c \in C_0$ or $rot(c) \in C_0$ (where $rot(c)$ is any rotation of the cycle c). To demonstrate this fact, let us first note that given a path $c \in C$, where $src(c) = fu_n$, the following edges must be a part of c:

$$c \in C_{+1}[n] \quad\Rightarrow\quad \rho_1[n]\rho_2[n] \in c \cap (\sigma_2[n+1] \in c \cup \rho_4[n] \in c)\ \cup\ \rho_3[n]\rho_4[n] \in c$$

$$c \in C_{+2}[n] \quad \Rightarrow \quad \rho_1[n]\rho_2[n] \in c \,\cap\, \rho_3[n+1]\rho_4[n+1] \in c$$

$$c \in C_{-1}[n] \quad \Rightarrow \quad \sigma_0[n]\lambda_0[n] \in c \,\cup\, \sigma_3[n]\lambda_1[n] \in c$$

$$c \in C_{-2}[n] \quad \Rightarrow \quad \sigma_0[n]\lambda_0[n]\sigma_3[n-1]\lambda_1[n-1] \in c$$

Now we can consider each possible iteration. Using the statements above to determine whether an edge is in the iteration or not, and the FBI graph to determine the sources and sinks of these edges, we can check for simple cycles by looking for vertices that appear twice in the path:

1. $P_C = C_{+1}[n]C_{-1}[n+1]$ or $P_C = C_{-1}[n+1]C_{+1}[n]$:

   We see that $snk(\rho_2[n]) = src(\lambda_0[n+1])$, $snk(\sigma_2[n+1]) = src(\sigma_3[n+1])$, $src(\rho_3[n]) = snk(\lambda_0[n+1])$, and $snk(\rho_4[n]) = src(\lambda_1[n+1])$. Therefore a simple cycle $c \in P_C$ exists s.t. $rot(c) \in C_0$.

2. $P_C = C_{+2}[n]C_{-2}[n+2]$ or $P_C = C_{-2}[n+2]C_{+2}[n]$:

   Since $src((\rho_3[n+1]) = snk(\lambda_0[n+2])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_0$.

3. $P_C = C_{+1}[n]C_{-2}[n+1]$:

   Since $snk((\rho_2[n]) = src(\lambda_0[n+1])$ and $src(\rho_3[n]) = src(\sigma_3[n])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_0$.

4. $P_C = C_{-2}[n+2]C_{+1}[n]$:

   Since $snk((\sigma_2[n+1]) = snk(\lambda_0[n+2])$, and $snk(\rho_4[n]) = src(\lambda_1[n+1])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_0$.

5. $P_C = C_{-1}[n+1]C_{+2}[n]$:

   Since $src((\lambda_0[n+1]) = snk(\rho_2[n])$, and $src(\sigma_3[n+1]) = src(\rho_3[n+1])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_0$.

6. $P_C = C_{+2}[n]C_{-1}[n+2]$:

   Since $src((\rho_3[n+1]) = snk(\lambda_0[n+2])$ and $snk(\rho_4[n+1]) = src(\lambda_1[n+2])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_0$.

Therefore all possible iterations in this case include a simple cycle, and the maximum cycle time for all subpatterns of C in this case is τ_{C_0}.

Case 3: Forward-Latency Paths: $\{n = K\}$ $[F|C_{+1}|C_{+2}]$ $\{n \geq (K+1) \bmod N\}$

All iterations of P in these subpatterns move forward. Cycles can exist only when the forward-latency path spans the entire ring of N PCHBs at least once. Given a source vertex fu_n, there is only one possible path $c \in F\texttt{[}n\texttt{]}$ $(\rho_0\texttt{[}n\texttt{]})$, four possible paths $c \in C_{+1}\texttt{[}n\texttt{]}$ (labelled $C^0_{+1}[n]$ – $C^3_{+1}[n]$), and one possible path $c \in C_{+2}\texttt{[}n\texttt{]}$ $(\rho_1[n]\rho_2[n]\sigma_1[n+1]\sigma_2[n+1]\rho_3[n+1]\rho_4[n+1])$. The number of tokens on each possible path ranges from zero to two, and depends upon whether the PCHBs traversed are initialized to send a value, to receive a value, or neither.

Note that if our pattern consists solely of edges $\rho_0\texttt{[}n\texttt{]}$ and $P = F^*$, then F can be repeated only N times before a simple cycle is created. In fact, this is the common forward-latency cycle C_f from Section 5.3.2.1, and since m messages in the ring result in m initial tokens on edges labelled ρ_0, (there are m PCHBs that initially send tokens and use R_{snd}), the forward-latency cycle-time is therefore commensurate with the previous common cycle definition of $\tau_f^\dagger$:

$$\frac{1}{m}\sum_{n=0}^{N-1} \delta(fu_n)$$

Returning to paths including subpatterns other than F, since each possible subpattern in this case moves forward, we can from our FBI graph $G = \langle V, E, R\rangle$ construct a new acyclic graph $A_F(G)$ and use it to determine the maximum cycle time for this case. Since forward-latency simple-cycles can traverse the PCHB ring once or twice (depending on whether N is even or odd), we unroll the ring in $A_F(G)$ so that each PCHB has three vertices in the new acyclic graph. This is demonstrated in Figure 5.10.

The PCHB vertices in $A_F(G)$ are connected by edges representing the paths in $F\texttt{[}n\texttt{]}$, $C_{+1}[n]$ and $C_{+2}[n]$. Each edge is annotated with the delay of and number of tokens on the paths in G. Thus, given $G = \langle V, E, R\rangle$, we create $A_F(G) = \langle V_F, E_F, D_F, T_F\rangle$, where

- $V_F = N \times \{1, 2, 3\}$

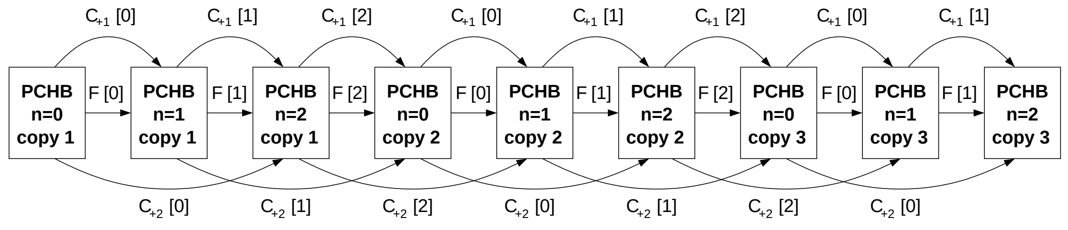


Figure 5.10: Acyclic forward-latency graph $A_F(G)$.

- $E_F = \{F, C^0_{+1}, C^1_{+1}, C^2_{+1}, C^3_{+1}, C_{+2}\} \times N$ represents a path c, where $src(c) = fu_n$
- $D_F : E_F \rightarrow \mathbf{N}$ is the delay of the edge's path
- $T_F : E_F \rightarrow \mathbf{N}$ is the number of tokens on the edge's path

It remains therefore to search $A_F(G)$ to determine the path with source and sink vertices both labelled with the same PCHB index $0 \leq n < N$ that has the maximum quotient of total path delay divided by the total number of reset tokens on the path. (Some combinations of successive edges belonging to the class C_{+1} contain internal cycles and can be ignored by the heuristic.) This quotient is the newly-defined forward-latency cycle-time τ_f for the system.

Case 4: Composite Backward-Latency Paths:

$\{n = K\}\ [C_{-1} \mid C_{-2} \mid C_{-2}FC_{-2}]\ \{n \leq (K-1) \bmod N\}$

First we note that, to avoid simple cycles from the subpattern C_0, the step F is allowed only if it is between two instances of a path from C_{-2}. Any other subpatterns involving F are left for Case 5. We therefore create a new class

$$\{n = K\}\ C_{-3} = C_{-2}FC_{-2}\ \{n = (K-3) \bmod N\}$$

We can now rewrite the pattern for this case as $[C_{-1}|C_{-2}|C_{-3}]$. There are four possible paths $c \in C_{-1}$, one possible path $c \in C_{-2}$, and one possible path C_{-3}.

Similarly to what we did for the forward handshake cycles of Case 3, we create an acyclic graph $A_B(G)$ from the FBI model in which edges represent paths from either $C_{-1}[n]$, $C_{-2}[n]$ or $C_{-3}[n]$. Again, it remains to search the acyclic graph to determine the path with matching source and sink and the largest quotient of path delay divided by number of tokens (i.e., the backward-latency cycle-time τ_b).

Case 5: Direction Change Including F-iterations:

$C_{-1}F$ or FC_{-1} or $[F|C_{+1}|C_{+2}]FC_{-2}$ or $C_{-2}F[F|C_{+1}|C_{+2}]$

Let us begin by considering the two subpatterns that use F and C_{-1}, as well as the two subpatterns that include C_{-2} and two instances of F. All of these subpatterns begin and end at the same node, forming a simple cycle. There are five cycles whose rotations create all paths for these subpatterns:

$$\begin{aligned}
C^0_{-F}[n] &= \rho_0[n]\ \sigma_0[n+1]\ \sigma_1[n+1]\ \sigma_2[n+1]\ \sigma_3[n+1]\ \lambda_1[n+1] \\
C^1_{-F}[n] &= \rho_0[n]\ \sigma_0[n+1]\ \lambda_0[n+1]\ \sigma_3[n]\ \sigma_4[n]\ \sigma_5[n] \\
C^2_{-F}[n] &= \rho_0[n]\ \sigma_0[n+1]\ \lambda_0[n+1]\ \rho_3[n]\ \rho_4[n+1]\ \lambda_1[n+1] \\
C^3_{-F}[n] &= \rho_0[n]\ \rho_1[n+1]\ \rho_2[n+2]\ \lambda_0[n+2]\ \sigma_3[n+1]\ \lambda_1[n+1] \\
C^4_{-F}[n] &= \rho_0[n]\ \rho_0[n+1]\ \sigma_0[n+2]\ \lambda_0[n+2]\ \sigma_3[n+1]\ \lambda_1[n+1]
\end{aligned}$$

Let C_{-F} be the set containing all of these cycles, all of which contain exactly one token.

All other subpatterns in this case reduce to a rotation of one of the five cycles listed above.

1. $P_C = C_{+1}[n]F[n+1]C_{-2}[n+2]$: Since $snk((\sigma_2[n+1]) = snk(\lambda_0[n+2])$, and $snk((\rho_4[n]) = snk(\sigma_3[n+1])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_{-F}$.

2. $P_C = C_{-2}[n+2]F[n]C_{+1}[n+1]$: Since $snk((\rho_2[n+1]) = snk(\sigma_0[n+2])$, and $src((\rho_3[n+1]) = snk(\lambda_0[n+2])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_{-F}$.

3. $P_C = C_{+2}[n]F[n+2]C_{-2}[n+3]$: Since $snk((\rho_4[n]) = snk(\sigma_3[n+2])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_{-F}$.

4. $P_C = C_{-2}[n]F[n-2]C_{+2}[n-1]$: Since $snk((\sigma_0[n]) = snk(\rho_2[n-1])$, there exists a simple cycle $c \in P_C$ s.t. $rot(c) \in C_{-F}$.

Thus,

$$\tau_{C_{-F}} = \max\left\{\delta(C^j_{-F}[n])\ :\ 0\leq n<N,\ 0\leq j<5\right\} \tag{5.10}$$

is the maximum cycle time for all subpatterns in this case.

5.3.2.3 Basic Homogeneous Cases

Let us consider the results of our critical-cycle analysis when applied to a subset of homogeneous systems. In a *homogeneous system* of PCHBs, every PCHB has identical delays in the FBI model. In the subset considered in this section, we have $\delta(fu_n) = \delta(fd_n) = \delta(vu_n) = \delta(vd_n) = f$, $\delta(bu_n) = \delta(bd_n) = b$, and $\delta(iu_n) = \delta(id_n) = i$ for all $0 \leq n < N$. We call such systems *basic homogeneous.*

Basic homogeneous systems are a practical reduction of homogeneous systems. Usually, only processes containing state bits have different delay values for the set and reset phases of the cycle. In PCHBs with single inputs and outputs, channel sizes are limited by the maximum number of n-transistors allowed in series in the technology. Thus, input validities can be computed in the same number of transitions as forward latencies.

Referring to the common cycles defined previously in Section 5.3.2.1, their equations reduce to the following:

$$\tau_i^\dagger = 2f + 2b + 2i \tag{5.11}$$

$$\tau_h^\dagger = 4f + 2b \tag{5.12}$$

$$\tau_f^\dagger = \frac{Nf}{m} \tag{5.13}$$

$$\tau_b^\dagger = \frac{N(f+b)}{s-m} \tag{5.14}$$

where m is the number of messages in the ring. Note that the backward time τ_b is equal for both even- and odd-length rings.

Theorem 6 *Given an FBI representation of a basic homogeneous ring of PCHBs, the critical cycle time is* $\max\{\tau_f^\dagger, \tau_b^\dagger, \tau_i^\dagger, \tau_h^\dagger\}$.

Proof: For the internal cycles in our FBI system,

$$\tau_{C_0} = 2f + 2b + 2\max\{f, i\} = \max\{\tau_i, \tau_h\}$$

For the forward-latency cycles in our FBI system,

$$\begin{aligned}
\delta_F &= f \\
\delta_{C_{+1}} &= 3f + 2b + \max\{f, i\} \\
\delta_{C_{+2}} &= 4f + 2b + 2i
\end{aligned}$$

Without loss of generality, let the critical forward-latency cycle include n_2 instances of C_{+2}, n_1 instances of C_{+1}, and n_f instances of F. Consider simple cycles within one traversal of the ring of N PCHBs. By inspection, all subpatterns C_{+1} contain one reset token except subpatterns originating at a PCHB that sends an initial message, which contain two reset tokens. Similarily, all subpatterns C_{+2} contain one reset token except subpatterns that pass through a PCHB with an initial send, which also contain two reset tokens. Thus, given $N = 2n_2 + n_1 + n_f$,

$$\begin{aligned}
\frac{n_2\delta_{C_{+2}} + n_1\delta_{C_{+1}} + n_f\delta_F}{n_2 + n_1 + m} &= \frac{(4n_2 + 3n_1 + n_f)f + (2n_2 + 2n_1)b + 2n_2 i + n_1 \max\{f + i\}}{n_2 + n_1 + m} \\
&= \frac{n_2\tau_i^\dagger + n_1 \max\{\tau_h^\dagger, \frac{1}{2}(\tau_h^\dagger + \tau_i^\dagger)\} + (2n_2 + n_f)f}{n_2 + n_1 + m} \\
&\leq \frac{n_2\tau_i^\dagger + n_1 \max\{\tau_h^\dagger, \tau_i^\dagger\} + m\tau_f^\dagger}{n_2 + n_1 + m} \\
&\leq \max\{\tau_i^\dagger, \tau_h^\dagger, \tau_f^\dagger\}
\end{aligned}$$

If two traversals of the ring are required, then we set $2N = 2n_2 + n_1 + n_f$ and expect $n_2 + n_1 + 2m$ tokens in the ring. The analysis that follows is similar to the above and yields the same final result.

When considering backward-latency cycles, by inspection, subpattern C_{-1} contains one token unless the source PCHB is an initial receiver, in which case it may contain zero tokens. Meanwhile, subpattern C_{-2} contains one token, unless it passes through an initial receiver, in which case it contains zero tokens. Finally, subpattern C_{-3} contains two tokens, unless it passes through an initial receiver, in which case it contains one token. Since we are searching for the maximum possible cycle time, given $N = 3n_3 + 2n_2 + n_1$,

$$\begin{aligned}
\delta_{C_{-1}} &= 2f + 2b + \max\{f, i\} \\
\delta_{C_{-2}} &= 2f + 2b \\
\delta_{C_{-3}} &= 5f + 4b \\
\frac{n_3\delta_{C_{-3}} + n_2\delta_{C_{-2}} + n_1\delta_{C_{-1}}}{2n_3 + n_2 + n_1 - m} &= \frac{(5n_3 + 2n_2 + 2n_1)f + (4n_3 + 2n_2 + 2n_1)b + n_1 \max\{f, i\}}{2n_3 + n_2 + n_1 - m} \\
&= \frac{N(f + b) + (2n_3 + n_1)f + (n_3 + n_1)b + n_1 \max\{f, i\}}{2n_3 + n_2 + n_1 - m} \\
&= \frac{(s - m)\tau_b^\dagger + \frac{n_1}{2}\max\{\tau_i^\dagger, \tau_h^\dagger\} + \frac{n_3}{2}\tau_h^\dagger}{2n_3 + n_2 + n_1 - m} \\
&= \frac{(\frac{3n_3+2n_2+n_1}{2} - m)\tau_b^\dagger + \frac{n_1}{2}\max\{\tau_i^\dagger, \tau_h^\dagger\} + \frac{n_3}{2}\tau_h^\dagger}{2n_3 + n_2 + n_1 - m} \\
&\leq \max\{\tau_b^\dagger, \tau_i^\dagger, \tau_h^\dagger\}
\end{aligned}$$

If multiple traversals of the ring are required, similar analyses can be performed changing the definition of N and the number of tokens expected in the denominator, yielding identical final results.

And finally, for the cycles including F and a direction change,

$$\tau_{C_{-F}} = \max\{3f + 2b + \max\{f, i\},\ 4f + 2b\} \leq \max\{\tau_i^\dagger, \tau_h^\dagger\}$$

Therefore every possible critical cycle in a basic homogeneous ring of PCHBs collapses into the maximum of the four common cycles: internal $\tau_i^\dagger$, handshake $\tau_h^\dagger$, forward latency $\tau_f^\dagger$, and backward latency $\tau_b^\dagger$. $\square$

5.3.3 Dynamic Slack

While static slack expresses the maximum number of messages that can be present in a pipeline without deadlock, the *dynamic slack* of a pipeline is the range of number of messages that a pipeline can hold while operating at maximum throughput [29]. This range depends not only on the number of PCHBs in the pipeline but also on the forward and backward latencies, and the maximum internal and handshake cycle-times of the pipeline. The insertion of L-R buffers into a pipeline of complex PCHBs performing computations can affect all of these values, possibly increasing the dynamic slack of the system. (Previous analyses of asynchronous pipelines have been performed by Williams [52] and Lines [29]).

We make use of the following definitions for general heterogeneous systems:

$$\delta_f = m\,\tau_f \tag{5.15}$$

$$\delta_b = (s-m)\,\tau_b \tag{5.16}$$

where τ_f and τ_b are the cycle times of critical cycles discovered by the heuristics presented in Section 5.3.2.2 (cases 3 and 4). The maximum throughput of a system is the reciprocal of the critical cycle time, and is achieved at an intersection of internal, handshake, forward-latency or backward-latency time constraints. The equations in this section assume that the common cycles defined in Section 5.3.2.1 are the critical ones but can be reformulated to incorporate the critical cycles found for general heterogeneous systems in the previous section.

Let us therefore begin by defining

$$\begin{aligned} d &= m \quad s.t.\ \tau_f = \tau_b \\ &= \frac{N\delta_f}{2(\delta_b + \delta_f)} \end{aligned} \tag{5.17}$$

where m is the number of messages in the system. If a system is indeed limited by its latency cycles (i.e., if $\max\{\tau_i, \tau_h\} \leq \tau_f|_{m=d}$), then d is the dynamic slack and the system runs at maximum throughput when $m = d$. However, if the system is instead limited by either its internal or its handshake cycles, the system runs at maximum throughput whenever m is within dynamic slack range $[d_{min}, d_{max}]$ for that cycle time, where

$$d_{min} = m \quad s.t.\ \tau_f = \tau_{C_0}$$

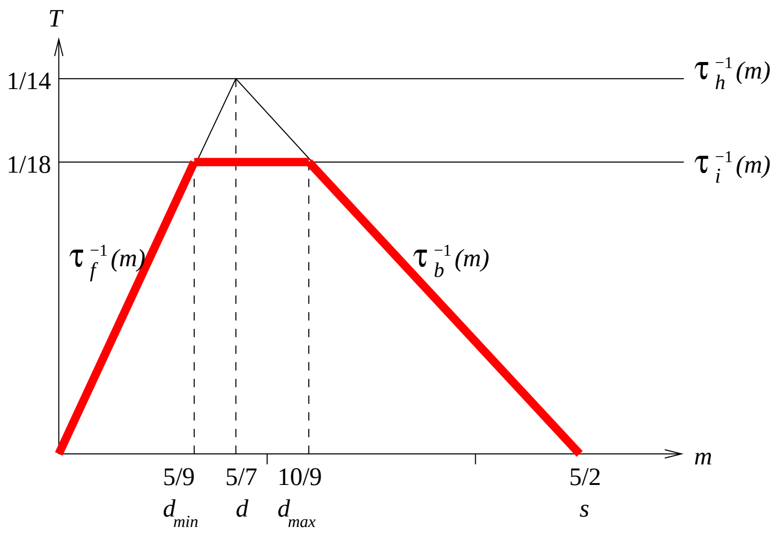


Figure 5.11: Throughput vs. number of messages.
For a basic homogeneous ring of five PCHBs with $f = 2$, $b = 3$, $i = 2$. While $m \leq d_{min}$, $\tau_{crit} = \tau_f$, while $d_{min} \leq m \leq d_{max}$, $\tau_{crit} = \tau_i$, and while $d_{max} \leq m$, $\tau_{crit} = \tau_b$.

$$= \frac{\delta_f}{\tau_{C_0}} \tag{5.18}$$

and

$$\begin{aligned} d_{max} &= m \quad s.t. \;\; \tau_b = \tau_{C_0} \\ &= \frac{N}{2} - \frac{\delta_b}{\tau_{C_0}} \end{aligned} \tag{5.19}$$

Note that for the system to run at some cycle time $\tau \geq \tau_{C_0}$, we can substitute τ for τ_{C_0} in the definitions above and in any equations that depend upon these definitions. As described by Lines [29], the plot of a pipeline's throughput T against the number of messages m resembles either a triangle with a peak at $m = d$ and $T_{max} = \frac{1}{\tau_{crit}}$, or a trapezoid, where the throughput is at T_{max} for all $d_{min} \leq m \leq d_{max}$. A trapezoidal case is illustrated in Figure 5.11. Note that, as is usually the case in heterogeneous systems and always the case in basic homogeneous systems,

$$d_{min} \leq d \leq d_{max} \iff \delta_b \leq \frac{N\tau_{C_0} - 2\delta_f}{2} \tag{5.20}$$

Given a system composed of multiple pipelines and rings, the system's maximum throughput is the lowest individual maximum throughput of any of its pipelines or rings. Therefore, when slack matching this system, our task is to insert L-R buffers to decrease the critical cycle time of slower rings and pipelines, and to align the dynamic slacks or ranges of dynamic slack so that each component is operating at this system maximum throughput. This often requires knowledge of the number of messages that will be in a ring or a pipeline.

As an example, in basic homogeneous systems,

$$\begin{aligned} d &= \frac{Nf}{2(2f+b)} = \frac{Nf}{\tau_h} && (5.21)\\ d_{min} &= \frac{Nf}{2(f+b+\max\{f,i\})} = \frac{Nf}{\max\{\tau_i,\tau_h\}} && (5.22)\\ d_{max} &= \frac{N\max\{f,i\}}{2(f+b+\max\{f,i\})} = \frac{N\max\{f,i\}}{max\{\tau_i,\tau_h\}} && (5.23) \end{aligned}$$

Since $d_{min} \leq d_{max}$ and $f \leq \max\{f,i\}$, we know that basic homogenous systems are always limited by their internal cycle or by their handshake cycle. Structures in these systems are therefore always slack matched when they contain m tokens and $d_{min} \leq m \leq d_{max}$.

5.3.3.1 Base Case: Ring of PCHBs

Consider first a basic homogeneous ring containing N PCHBs. We wish to determine the optimal number n_s of slack buffers to be inserted in the ring so that it runs at maximum throughput. If we know that the ring will always contain $m = M$ messages, then the constraint $d_{min} \leq m \leq d_{max}$ for maximum throughput for homogeneous pipelines presented at the end of the previous section is fulfilled when

$$\frac{M\max\{\tau_i,\tau_h\}}{\max\{f,i\}} \leq (N+n_s) \leq \frac{M\max\{\tau_i,\tau_h\}}{f}$$

(In situations when $f \geq i$, the above inequality becomes an equality equivalent to previous claims of optimality that link cycle time, stage latency, and number of stages, where each stage has identical cycle time [42, 39, 41].) Thus, the minimum number of slack buffers that should be inserted into a basic homogeneous ring for maximum throughput is as follows:

$$n_s = \left\lceil \frac{M \max\{\tau_i, \tau_h\}}{\max\{f, i\}} - N \right\rceil \tag{5.24}$$

If we wish the ring to run at some cycle time $\tau \geq \tau_{crit}$, then we can substitute τ for $\max\{\tau_i, \tau_h\}$ in the above equation.

Now consider a heterogeneous ring currently containing N PCHBs with critical stationary cycle τ_{C_0}, forward-latency delay δ_f, and backward-latency delay δ_b. We wish to determine the optimal number n_s of slack buffers to be inserted in the ring so that it runs at maximum throughput. Assume that slack buffers have short internal cycles that do not increase the maximum τ_{C_0} of the system and have equal set and reset delays ($f_s = \delta(fu_s) = \delta(fd_s)$ and $b_s = \delta(bd_s) = \delta(bu_s)$).

If we know that the ring will always contain $m = M$ messages, then the constraint for maximum throughput for homogeneous pipelines presented at the end of the previous section is fulfilled when $\min\{d_{min}, d\} \leq m \leq \max\{d, d_{max}\}$. For the purposes of illustration, let us consider the following scenario:

$$\begin{array}{rcl}
d_{min} \leq & m & \leq d_{max} \\
\dfrac{\delta_f + n_s f_s}{\tau_{C_0}} \leq & M & \leq \dfrac{N + n_s}{2} - \dfrac{\delta_b + n_s(f_s + b_s)}{\tau_{C_0}} \\
\dfrac{\tau_{C_0}(2M - N) + 2(\delta_b)}{\tau_{C_0} - 2(f_s + b_s)} \leq & n_s & \leq \dfrac{M\tau_{C_0} - \delta_f}{f_s}
\end{array}$$

The minimum number of slack buffers required for the heterogeneous ring to run at maximum throughput is therefore

$$n_s = \left\lceil \frac{\tau_{C_0}(2M - N) + 2(\delta_b)}{\tau_{C_0} - 2(f_s + b_s)} \right\rceil \tag{5.25}$$

5.3.3.2 Base Case: Reconvergent Fanout

Consider a system where one PCHB sends tokens on two different output channels connected to two separate pipelines that reconverge in a PCHB with two inputs at the end. An example is shown in Figure 5.12. The two pipelines in this case are called P ($S \rightarrow P1 \rightarrow P2 \rightarrow P3 \rightarrow P4 \rightarrow P5 \rightarrow M$)

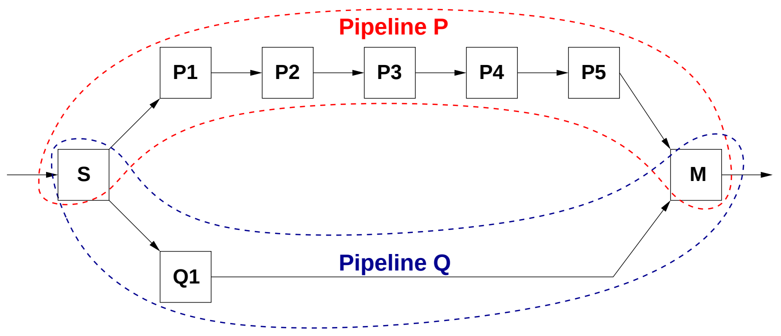


Figure 5.12: Reconvergent fanout example.

and Q ($S \rightarrow Q1 \rightarrow M$). Note that the processes M and S are included in both pipelines.

If P and Q are not slack matched, a possible plot of throughput as a function of the number of messages is shown in Figure 5.13. Note that the system's highest attainable throughput is less than the maximum attainable throughput of the individual pipelines. In this case, since the maximum throughputs for each individual pipeline are identical, we slack match the system by inserting L-R buffers so that the dynamic slack ranges of the two pipelines overlap and the pipelines can run at maximum combined throughput. If the maximum throughputs for the individual pipelines differ, we slack match the system by inserting L-R buffers so that the throughput functions for the pipelines intersect at the lowest of the maximum individual throughputs. This is the highest throughput at which the system as a whole can run. (A system can only be as fast as its slowest pipeline.) This alternate scenario is illustrated in Figure 5.14. For the remainder of this section, we assume that the maximum throughputs for each individual pipeline are identical.

In a basic homogeneous system, d, d_{min}, d_{max}, τ_f, and τ_b are all proportional to N, while τ_h and τ_i have constant identical values for both pipelines. In our example, we have $N_Q < N_P$ and so if $d_{max}(Q) < d_{min}(P)$, we must add n_s slack buffers to Q until their dynamic slack ranges overlap.

$$\begin{aligned} d_{max}(Q) &\geq d_{min}(P) \\ (N_Q + n_s)\max\{f, i\} &\geq N_P f \\ n_s &\geq \frac{N_P f}{\max\{f, i\}} - N_Q \end{aligned} \tag{5.26}$$

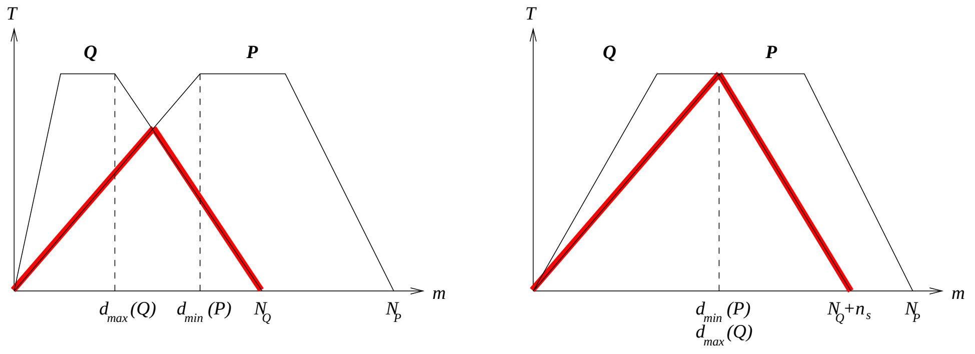


Figure 5.13: Slack matching reconvergent fanout for pipelines with identical maximum throughputs. The plot on the left shows system throughputs before slack matching. The maximum throughputs attainable by pipelines P and Q individually are both greater than the maximum throughput attainable by the system as a whole. (System throughput is indicated by the thick line.) The plot on the right is for the slack matched system. Slack matching adds buffers to Q so that the dynamic slack ranges of the two pipelines overlap, and the system can run as quickly as its individual pipelines.

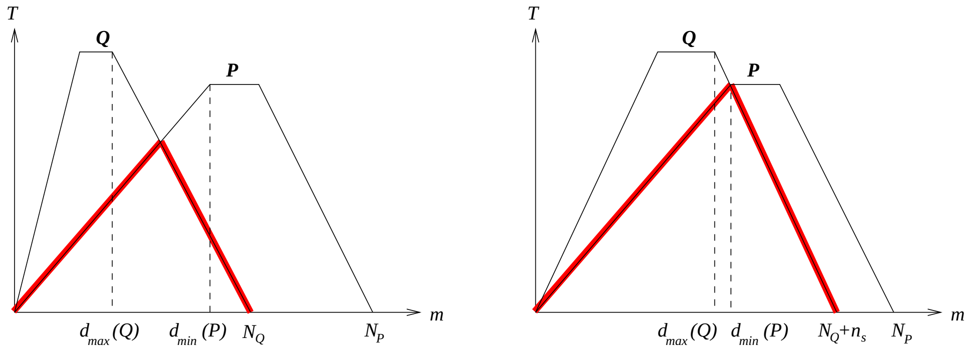


Figure 5.14: Slack matching reconvergent fanout for pipelines with different maximum throughputs. The plot on the left shows system throughputs before slack matching. The maximum throughputs attainable by pipelines P and Q individually differ, and the maximum throughput attainable by the system (thick line) is lower than both of them. The plot on the right is for the slack matched system. Slack matching adds buffers to Q so that the two throughput functions intersect at the lower of the two maximum individual throughputs. The system can now run as quickly as its slowest individual pipeline.

Note that if $f \geq i$, this is tantamount to saying that both pipelines should contain the same number of stages.

In the heterogeneous case, without loss of generality, assume that if the pipelines' ranges of dynamic slack do not overlap, then $d_{max}(Q) < d_{min}(P)$. We wish to add the mininum number of n_s slack buffers to Q so that $d_{max}(Q) \geq d_{min}(P)$. (For completeness, we actually check that $\max\{d(Q), d_{max}(Q)\} \geq \min\{d_{min}(P), d(P)\}$ but for the purpose of illustration, we let the sim-

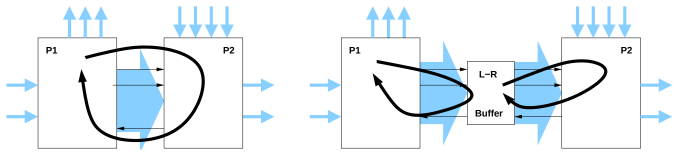


Figure 5.15: Breaking up a long handshake with a slack buffer.

pler condition suffice.) Let $\delta_f(Q)$ be the current forward-latency delay and $\delta_b(Q)$ be the current backward-latency delay of Q, and let $\delta_f(P)$ be the forward latency of P. For the slack buffers, assume again that $f_s = \delta(fu_s) = \delta(fd_s)$ and $b_s = \delta(bu_s) = \delta(bd_s)$). Then,

$$\begin{aligned} d_{max}(Q) &\geq d_{min}(P) \\ \frac{N_Q + n_s}{2} - \frac{\delta_b(Q) + n_s(f_s + b_s)}{\max\{\tau_i, \tau_h\}} &\geq \frac{\delta_f(P)}{\max\{\tau_h, \tau_i\}} \\ n_s &\geq \frac{2(\delta_f(P) + \delta_b(Q)) - N_Q \max\{\tau_h, \tau_i\}}{\max\{\tau_h, \tau_i\} - 2(f_s + b_s)} \end{aligned} \tag{5.27}$$

Thus we can determine the minimum number of slack buffers that need be added to a branch of reconvergent fanout for maximum throughput.

5.4 Clustering Heuristic

Given a decomposed system created by the DSA and projection phases of DDD, the clustering phase of DDD recomposes processes and slack matches the resulting system to run at any cycle time less than or equal to τ_{max} (specified by the designer). In doing so, DDD also attempts to minimize the system communications energy consumption. It is assumed that τ_{max} is greater than or equal to the longest internal cycle time τ_i of any PCHB in the decomposed system, and that if any handshake cycles τ_h in the system exceed τ_{max}, then a slack buffer can be inserted in between the two PCHBs to break the long handshake into two shorter handshakes whose cycle times do not exceed τ_{max}. (See Figure 5.15). If either of these two assumptions is not true, the target cycle time cannot be achieved by the DDD-generated system.

5.4.1 Preliminaries

For clustering, DDD expresses the decomposed system as a graph $G = \langle V, E \rangle$. Each edge $e \in E$ represents a communication channel from a source $src(e) \in V$ to a sink $snk(e) \in V$. Each vertex $v \in V$ either represents a PCHB or serves as a dummy source for edges representing system input-channels or a dummy sink for edges representing system output-channels. The number of data rails in a channel is $width(e)$. Paths, path sources, and path sinks are defined as for FBI graphs in Section 5.3.1.1. The length N_P of a path P is given by the number of vertices traversed by the path, $src(P)$ and $snk(P)$ inclusive. The designer may place latency constraints upon the system during clustering, requiring that the longest forward-latency path between a specific system input channel and system output channel cannot exceed δ transitions.

Rings of vertices in G can be one of three types generated by DDD. The first type of ring includes DSA variables that were used before being defined in an iteration of the DSA sequential program. The second type of ring includes processes with internal state bits that represent external system channels used multiple times within the original sequential program. The third type of ring includes processes for complex functions that are used multiple times in the sequential program and have been isolated from the rest of the system.

Prior to slack matching, DDD eliminates these rings and transforms G into an acyclic graph $G' = \langle V', E' \rangle$. Each ring of PCHBs is removed by selecting an edge in the ring, and changing the edge's source from the original vertex to a new dummy source with the same FBI values as the orginal vertex. The result is a straight path, called a *ring path*, with a length one greater than that of the original cycle. DDD then places a latency constraint on this path, the maximum value of which is determined by using techniques from Section 5.3 to slack match the original ring to meet a specific cycle time. After the clustering phase is complete, DDD restores the ring from the original graph, each edge containing the newly-determined number of slack buffers.

As illustrated in Figure 5.16, DSA-variable rings are broken between vertices for x_0 and x_n, where n is the maximum DSA index for x. The source of the edge between these two vertices is changed from process P_{x_n} to new dummy source process P'_{x_n}. For rings that include external system

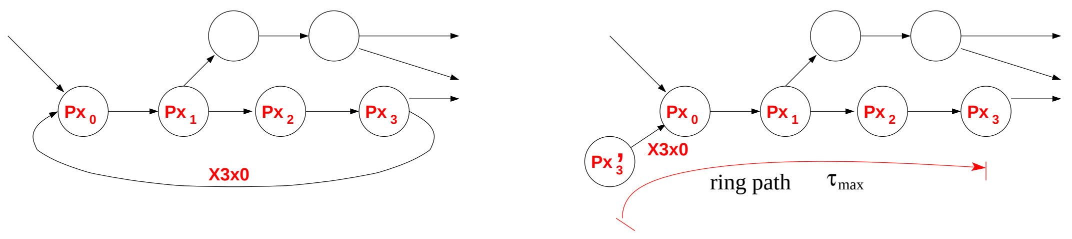


Figure 5.16: Removing cycles for clustering.
The ring in the PCHB graph on the left implements a chain of DSA variables $x_0 \ldots x_3$, where the value of x_3 is assigned to x_0 at the beginning of the DSA program. The ring is transformed into an acyclic path with a dummy source that has the same FBI values as the variable with the maximum DSA index, as shown in the graph on the right. A latency constraint derived from slack matching the ring to run at τ_{max} can now be placed on the path between vertices Px'_3 and Px_3.

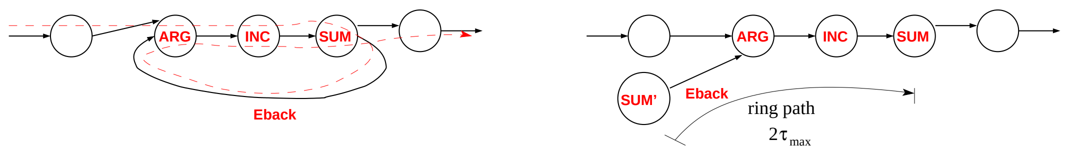


Figure 5.17: Incrementer example of removing cycles for clustering.
The ring in the PCHB graph on the left implements an incrementer process INC that has been isolated but is used twice per iteration of the original sequential program. The processes ARG and SUM each contain one state bit implementing two states, and collect incrementer inputs and distribute incrementer outputs, respectively. The ring is transformed into an acyclic path with a dummy source that has the same FBI values as SUM, as shown in the graph on the right. A latency constraint derived from slack matching the ring to run at $2\tau_{max}$ (because the entire ring is traversed only once for every two times the vertices ARG, INC, and SUM are traversed) can now be placed on the path between vertices SUM' and SUM.

input channels used multiple times, the ring is broken immediately before the input channel's vertex. (If an external output channel is used multiple times, rings containing its vertex cannot exist within the system alone.) And finally, in the case of rings that include isolated processes used multiple times, the ring is broken immediately before the process that gathers inputs for the isolated process.

In both of the last two cases, the ring is slack matched for the cycle time $N_{st} \cdot \tau_{max}$, where the entire ring is traversed only once for every N_{st} times that its stateholding processes are traversed. This allows the overall system to run at τ_{max}. A scenario with an isolated incrementer that is used twice per sequential program iteration is presented in Figure 5.17.

Once acyclic graph G' has been created, if any of the latency constraints (either user-specified or ring-derived) are exceeded, then processes must be clustered in series until they are met. DDD

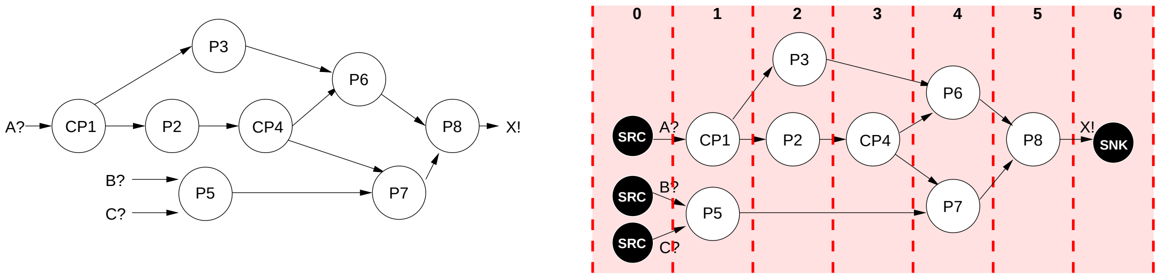


Figure 5.18: Clustering schedule.
The original graph (left) can be initilaly configured using an ASAP (as soon as possible) schedule (right).

uses a greedy heuristic to choose which edges should be removed by clustering in series. Processes should not be clustered if the new internal cycles τ_i or handshake cycles τ_h generated are greater than τ_{max}. If it is not possible to reduce path lengths to meet latency constraints without increasing the maximum cycle time of the system, then either the latency constraints or the target cycle time must be relaxed.

After generating a homogeneous acyclic graph G', DDD creates a table of L_{max} *columns*, where L_{max} is the length of the longest path in G'. Every vertex is assigned to a column such that for all $v \in V$, $0 \leq col(v) < L_{max}$. For the column assignments to be legal, the following *order constraint* must be fulfilled:

$$\forall e \in E' : \quad col(snk(e)) > col(src(e)) \tag{5.28}$$

Now, the *span* of an edge $e \in E'$ is defined as follows:

$$span(e) = col(snk(e)) - col(src(e)) - 1 \tag{5.29}$$

We also define a *schedule* of G' to be a set of column assignments to the vertices of V'. An example of an initial schedule is given in Figure 5.18.

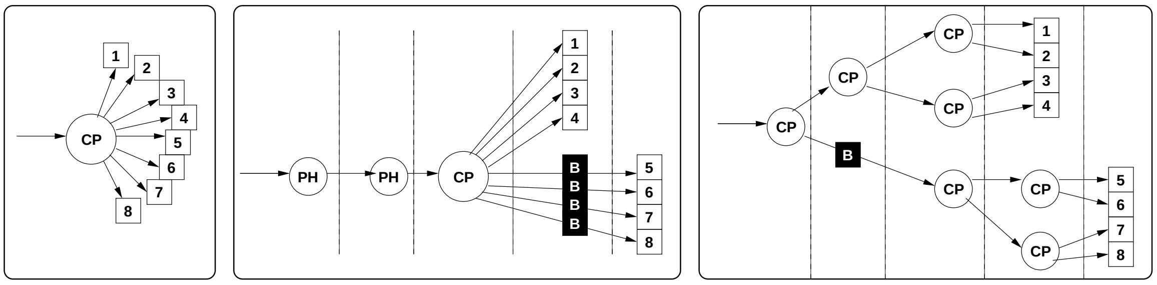


Figure 5.19: Splitting copy processes that exceed maximum fanout for clustering.

5.4.2 Copy Processes

Although they do not include any computation, there is a size limit for copy processes because large fanout leads to both electrical slowdowns and tall completion trees. The maximum fanout of a copy process can be set depending on the fabrication technology and desired cycle time. DDD splits any process in the decomposed system that exceeds this fanout limit into a tree of copy processes.

It is not advisable to assign the copy channels to different leaf processes before the rest of the clustering stages assign processes to columns, since the processes on the receiving end of the channels could be placed in columns that are far apart. The copy process would then be constrained to appear in a column before the earliest of the receiving columns, and the channels to the later receivers would inefficiently require the insertion of many slack matching buffers. It is therefore better to wait until after the receiving processes have been assigned before grouping them together for leaf copy processes.

However, we must still create spacer processes to hold columns for every level of the tree when the copy process is eventually split. Thus, if an eight-way copy process needs to be split in a system with a maximum copy fanout of two, the process is first converted into three processes in series, where the first two are place-holder buffers (PH) and the last is still an eight-way copy. The clustering proceeds to assign all of the processes to columns, and then the three copy processes are turned into a three-level tree (if they span more than three columns, the branching is saved for the last two columns). The eight receiving processes are assigned to leaf copy processes in groups of two, according to how close their new columns are. Figure 5.19 illustrates the steps in this procedure.

5.4.3 Simulated Annealing

DDD uses *simulated annealing* to recompose processes and reduce energy consumption while ensuring that the system's critical cycle time is no greater than τ_{max}. In simulated annealing [26], a system is randomly perturbed and if the perturbation decreases the overall system cost, then the new configuration is accepted. If the move increases the overall system cost, then the new configuration is accepted with a probability that decreases with time. The number of perturbations and probabilities of acceptance are set by an annealing schedule, and whenever the "temperature" is changed, the configuration with minimum system cost using the previous temperature is chosen as the starting point for the next temperature. We have chosen this randomized heuristic because the cost function and size constraints for clustering do not fit the formats required by linear or quadratic programming, and greedy non-randomized heuristics are easily trapped in local minima, producing subpar results.

For DDD clustering, the processes are initially assigned columns using ASAP (as soon as possible) scheduling, where each vertex is assigned to the column with the lowest possible index such that the assignment still maintains the order constraint. The possible individual annealing moves are as follows:

- move a process to a different column;
- move a process to a different column and cluster it in series with another process in that column;
- cluster a process in parallel with another process in the same column that shares at least one input;
- remove a process from a cluster in which it was included by recomposition in parallel;
- remove a process from a cluster in which it was included by recomposition in series and, to maintain the order constraint, move it to a different column.

Whenever a move is selected by the heuristic, DDD slack matches the system to meet the target cycle time and then computes the cost difference to determine whether or not to accept the move. The cost function chosen for clustering is the energy consumed by communications, including slack buffers and input validity trees for channels. The energy cost of computations is either not affected by recomposition (when performed in parallel), or small compared to the change in communication costs (when performed in series). Thus,

$$f_{obj} = \sum_{e \in E'} N_{slack}(e) f_{buf}(width(e)) + \sum_{e \in E'} f_{val}(width(e)) \tag{5.30}$$

is the objective function to be minimized by simulated annealing. $N_{slack}(e)$ will be determined by slack matching and is the number of slack buffers on the edge e, $f_{buf}(x)$ is the energy consumed by a slack buffer for e1of$x$ channels, and $f_{val}(x)$ is the energy consumed by a validity tree with x inputs. Both f_{buf} and f_{val} can be experimentally determined for the technology targeted by DDD. As we will demonstrate in the following sections, N_{slack} is easy to determine for homogeneous systems but harder for heterogeneous systems.

5.4.4 Clustering Homogeneous Systems

Consider a system where for all $n: \ 0 \leq n < N$, $\delta(fu_n) = fu$, $\delta(bd_n) = bd$, $\delta(id_n) = id$, $\delta(fd_n) = fd$, $\delta(bu_n) = bu$, $\delta(iu_n) = iu$, $\delta(vu_n) = vu$, and $\delta(vd_n) = vd$. Note that this case is more general than the basic homogeneous case presented for slack matching structures in Section 5.3.3. Within the framework of simulated annealing, our task is to determine the number of slack buffers $N_{slack}(e)$ required for each edge $e \in E'$ so that the critical cycle time of the system is no greater than τ_{max}. Since it has already been established that all internal and handshake cycle times are less than or equal to τ_{max}, it remains only to ensure that the dynamic slack ranges of different branches of reconvergent fanout overlap, and that the ring paths contain the proper number of slack buffers to run at τ_{max} (given the initial number of messages M in the rings).

First, we consider slack matching in cases of reconvergent fanout. Consider the base case of reconvergent fanout with two branches P and Q. Without loss of generality, let $N_Q \leq N_P$. Let us add $span(e)$ slack buffers to every edge e in branch P, so that there are N'_P vertices in that branch—one for every column between the branch source and sink. This structure is optimally slack-matched when $d_{min}(P) \leq d_{max}(Q) \leq d_{max}(P)$. In a homogeneous system, this is equivalent to

$$r \cdot N'_P \leq N'_Q \leq N'_P$$

where

$$r = \frac{2 \cdot fu}{\tau_{max} - fu - fd - bu - bd}$$

and N'_Q is the sum of N_Q and the total number of slack buffers that will be added to the branch Q. Every column spanned by an edge in branch Q now contains a vertex in branch P. So, to slack match the branches while adding the least number of buffers to Q, we add

$$N_{slack}(e) = \lceil r \ \cdot \ span(e) \rceil \tag{5.31}$$

slack buffers to each edge e in branch Q, giving us $N'_Q = N_Q + \sum N_{slack}(e)$ vertices in the branch. The inequalities $r \cdot N'_P \leq N'_Q \leq N'_P$ hold, and the structure has been slack matched. Note that we never need add more than *span*(e) slack buffers to an edge e during slack matching. Therefore, slack matching homogeneous reconvergent fanout does not change the column assignments for any vertices from the pre-slack-matched structure.

When considering ring paths, DDD must choose a length constraint for slack matching. The ring can operate at τ_{max} when $d_{min} \leq M \leq d_{max}$. For a ring path P in a homogeneous system, this translates to

$$\frac{fu \cdot N_P}{\tau_{max}} \leq M \leq \frac{N_P}{2} - \frac{N_P\,(fu + bd + fd + bu)}{2\tau_{max}}$$

The minimum length of the path for throughput τ_{max}^{-1} is therefore

$$N_P \ \geq \ \frac{2\tau_{max} M}{\tau_{max} - (fu + bd + fd + bd)} = r \cdot \frac{\tau_{max} M}{fu}$$

We can therefore make use of the same slack-matching procedure for rings as for branches of reconvergent fanout, by setting the length constraint for the ring path as follows:

$$N_P = 1 + \frac{\tau_{max} M}{fu} \tag{5.32}$$

Here, P includes the dummy source of the ring path. Now if $\lceil r \cdot span(e) \rceil$ L-R buffers are placed on every edge e in the ring, the total length of the ring will fulfill the inequality

$$N_P \geq r \cdot \frac{\tau_{max} M}{fu}$$

Thus, when slack matching a homogeneous system, DDD first adds slack buffers to the longest branches in any reconvergent fanout structure so that every column traversed by the branches contains a vertex. Then, DDD simply sets $N_{slack}(e) = \lceil r \cdot span(e) \rceil$ for every edge $e \in E'$. This computation is simple enough to be incorporated into determining cost function for simulated annealing. Note that clustering can only be performed legitimately within this framework if the new clustered processes retain the homogeneous FBI values $\phi \in \Phi$.

5.4.5 Clustering Heterogeneous Systems

Clustering heterogeneous systems is more complex than clustering homogeneous systems. While we still make use of the table of columns to enforce legal moves during clustering-annealing steps, the concept of edge spans cannot be used to determine the number of slack buffers because dynamic slack values for ring paths and branches of reconvergent fanout are no longer linear in the path length. Instead, after a configuration change is proposed by a simulated annealing move, DDD slack matches the system in the manner presented below.

For every edge $e \in E'$, let the current number of slack buffers on the edge be $N_{slack}(e)$. DDD creates for each edge e a set $P(e)$ of paths that include e and potentially need to be individually slack matched. These paths can be either ring paths or branches of reconvergent fanout. Each path $p \in \bigcup_{e \in E'} P(e)$ is analyzed independently, and then DDD assigns to $N_{slack}(p)$ the number of additional slack buffers required on that path for the system to run at τ_{max}. If the system is slack matched then these values are all zero.

When slack matching a system for energy, we are concerned both with the number of buffers required, and the widths of the channels on which the buffers are placed. In current technologies, experiments show that the energy consumed by the smallest slack buffer is greater than the difference in energy consumption between the smallest slack buffer and a slack buffer for the widest allowed channel. Therefore, given the choice of placing a single buffer on a wide channel or placing two buffers on narrow channels, DDD always opts to place a single buffer. In general, DDD aims at first slack matching the system with as few buffers as possible, and secondly, at placing those buffers on the narrowest channels available.

Thus, to slack match a heterogeneous system, DDD first creates set $E_{slack} \subseteq E'$ where

$$E_{slack} \equiv \{e \text{ s.t. } \forall p \in P(e) \ : \ N_{slack}(p) > 0\} \tag{5.33}$$

Let $e_{max} \in E_{slack}$ be the edge with maximum cardinality $|P(e)|$. DDD then adds

$$\min_{p \in P(e_{max})} \{N_{slack}(p)\}$$

buffers to edge e_{max} and updates all path constraints $N_{slack}(p)$ for all edges e in E' accordingly. This step is repeated until the system is slack matched or E_{slack} is empty. In the latter case, it is necessary to add buffers to an edge e even though some paths $p \in P(e)$ already have $N_{slack}(p) = 0$. DDD adds the extraneous buffers; as long as $N \geq d_{min} \ \Rightarrow \ \delta_f \leq M\tau_{max}$, the system will still run at the desired throughput.

5.5 Summary

This chapter presented a clustering phase for DDD which includes both the recomposition of processes to improve latency and energy consumption, and the slack matching of the entire system to improve system throughput. Recomposition is most beneficial in systems of PCHBs when the processes being combined are connected in series, or share the same input. Processes are restricted in size by constraints involving the number of transitors allowed in series, which affects both computation pulldown networks and completion gate fanin (and thus the process's individual cycle time).

Both the slack matching optimization in general and dynamic slack matching in particular were presented in this chapter. We introduced the FBI model for PCHB systems, for use in determining critical cycles in the system, so as to perform dynamic slack matching. We demonstrated dynamic slack matching for both homogeneous and heterogeneous basic slack matching structures (rings and reconvergent fanout). Finally, we presented the simulated annealing heuristic used to combine recomposition and slack matching. Results of this heuristic are given in the next chapter.

Chapter 6

Case Study: Instruction Fetch Unit

We have applied the techniques of DDD to the designing of the instruction fetch unit of the Lutonium, an asynchronous 8051 microcontroller [41]. The unit combines control with a 16-bit datapath for the program counter. It is the limiting factor on instruction throughput of the entire microcontroller, and in a custom design consumes roughly 12% of the energy of the microcontroller core. Prior to DDD, its manual decomposition required weeks for a designer to perfect.

6.1 Initial Specification

The Fetch unit communicates with the instruction memory, the microcontroller branch unit, and the instruction decode. It is responsible for instruction decoding (of variable length instructions), generation of the next program counter, read and write accesses to instruction memory, and interrupt handling. The fetch control is complicated by the fact that although instructions can be one to three bytes in length, they are always fetched from memory two bytes at a time. Unaligned instructions can therefore introduce speculation to the fetching.

The unit is shown in context within the Lutonium in Figure 6.1. Its original sequential CHP specification, including declaration types as used in the `chpsim` simulator, is given below.

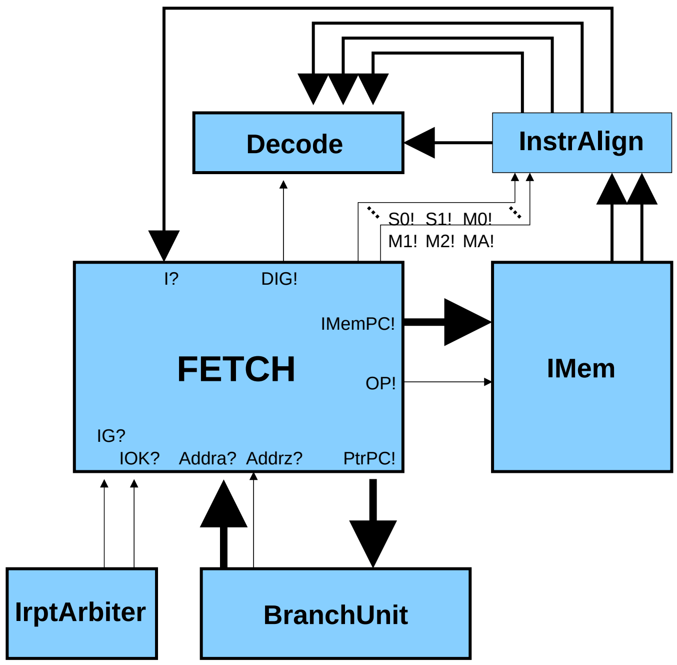


Figure 6.1: Fetch unit of the Lutonium 8051 microprocessor.

```
process fetch()(
    I?                   :  byte;    // Instruction from Memory
    IMemPC!              :  word;    // Next program counter, to Memory
    OP!                  :  e1of3;   // Control Operation, to Memory
    Addrz?               :  e1of2;   // lsb of Address from Branch Unit
    Addra?, PtrPC!       :  word;    // Addresses to/from Branch Unit
    S0!, S1!             :  e1of5;   // Instr Alignment Control (Mem to Decode)
    M0!, M1!, M2!, MA!   :  e1of2;   // Instr Alignment Control
    IG?                  :  e1of2;   // Possible Interrupt Warning
    IOK?                 :  e1of6;   // Interrupt Confirmation
    DIG!                 :  e1of2    // Interrupt Confirmation, to Decode
)
chp {
   var i          :  byte;    // Instruction
   var pca, aa    :  word;    // Program counter and branch address
   var pcz, az    :  e1of2;   // lsb of program counter and branch address
   var iLen       :  e1of3;   // Instruction length
   var ig         :  e1of2;   // Interrupt warning
   var irpt       :  e1of6;   // Actual interrupt
   var newpc      :  e1of4;   // Branch information decoded from instruction

        $pca := 0,\ pcz := 0,\ IMemPC!0,\ OP!0,\ S0!0,\ M0!0;$
        *[ $I?i;\ iLen := ilength(i);$
           $newpc\ :=\ idecode(i);$

           [ $pcz = 0 \longrightarrow$
              [ $iLen = 0\ \wedge\ newpc = 0 \longrightarrow\ pcz := 1$
              ▯ $iLen = 1\ \wedge\ newpc = 0 \longrightarrow\ pcz := 0,\ pca := pca + 1,\ S1!1,\ M1!1$
              ▯ $iLen = 2\ \wedge\ newpc = 0 \longrightarrow\ pcz := 1,\ pca := pca + 1,\ S1!1,\ M1!1;$
                                              $IMemPC!pca,\ OP!0,\ S0!2,\ M2!0$
        [] $iLen = 0 \;\wedge\; newpc = 1 \longrightarrow\; pcz := 1,\; S1!4;\; PtrPC!\{pca[31..1], pcz\}$
        [] $iLen = 1 \;\wedge\; newpc = 1 \longrightarrow\; pcz := 0,\; pca := pca + 1,\; S1!1,\; M1!1;\; PtrPC!\{pca[31..1], pcz\}$
        [] $iLen = 2 \;\wedge\; newpc = 1 \longrightarrow\; pcz := 1,\; pca := pca + 1,\; S1!1,\; M1!1;\; PtrPC!\{pca[31..1], pcz\},$
                                        $IMemPC!pca,\; OP!0,\; S0!2,\; M2!0; S1!4$

        [] $newpc = 2 \longrightarrow\; pcz := 1,\; S1!4;\; PtrPC!\{pca[31..1], pcz\}$
        [] $newpc = 3 \longrightarrow\; pcz := 1,\; S1!4$
        ]

    [] $pcz = 1 \longrightarrow\; pca := pca + 1;$
        [ $iLen = 0 \;\wedge\; newpc = 0 \longrightarrow\; pcz := 0$
        [] $iLen = 1 \;\wedge\; newpc = 0 \longrightarrow\; pcz := 1;\; IMemPC!pca,\; OP!0,\; S0!1,\; M1!0$
        [] $iLen = 2 \;\wedge\; newpc = 0 \longrightarrow\; pcz := 0;\; IMemPC!pca,\; OP!0,\; S0!1,\; M1!0,\; S1!2,\; M2!1,$
                                        $pca := pca + 1$

        [] $iLen = 0 \;\wedge\; newpc = 1 \longrightarrow\; pcz := 0;\; PtrPC!\{pca[31..1], pcz\}$
        [] $iLen = 1 \;\wedge\; newpc = 1 \longrightarrow\; pcz := 1;\; IMemPC!pca,\; OP!0,\; S0!1,\; M1!0,\; S1!4,$
                                        $PtrPC!\{pca[31..1], pcz\}$

        [] $iLen = 2 \;\wedge\; newpc = 1 \longrightarrow\; pcz := 0;\; IMemPC!pca,\; OP!0,\; S0!1,\; M1!0,\; S1!2,\; M2!1,$
                                        $pca := pca + 1;\; PtrPC!\{pca[31..1], pcz\}$
        [] $newpc = 2 \longrightarrow\; pcz := 0;\; PtrPC!\{pca[31..1], pcz\}$
        [] $newpc = 3 \longrightarrow\; pcz := 0$
        ]
    ];

    $IG?ig;\; DIG!ig,$ [$ig = 1 \longrightarrow\; IOK?irpt$ [] $ig = 0 \longrightarrow\; irpt := 0$];
    [ $irpt! = 0 \longrightarrow$
        [ $newpc = 0 \longrightarrow$ [$pcz = 1 \longrightarrow\; S1!4$ [] $pcz = 0 \longrightarrow$ **skip**]
        [] $newpc = 1 \longrightarrow\; Addra?pca,\; Addrz?pcz$
        [] $newpc = 2 \longrightarrow\; Addra?aa,\; Addrz?az;\; IMemPC!\{aa[31..1], az\},\; OP!0,$
                          [$az = 0 \longrightarrow\; S0!3,\; MA!0, S1!4$ [] $az = 1 \longrightarrow\; S0!4, S1!3,\; MA!1$]
        [] $newpc = 3 \longrightarrow\; Addra?aa,\; Addrz?az;\; IMemPC!\{aa[31..1], az\};$
                          [$az = 0 \longrightarrow\; OP!1$ [] $az = 1 \longrightarrow\; OP!2$]
        ]; $newpc := 1,\; PtrPC!\{pca[31..1], pcz\};\; Addra?pca,\; Addrz?pcz$

    [] $irpt = 0 \longrightarrow$
        [ $newpc = 0 \longrightarrow$ **skip**
        [] $newpc = 1 \longrightarrow\; Addra?pca,\; Addrz?pcz$
        [] $newpc = 2 \longrightarrow\; Addra?aa,\; Addrz?az;\; IMemPC!\{aa[31..1], az\},\; OP!0,$
                          [$az = 0 \longrightarrow\; S0!3,\; MA!0, S1!4$ [] $az = 1 \longrightarrow\; S0!4, S1!3,\; MA!1$]
        [] $newpc = 3 \longrightarrow\; Addra?aa,\; Addrz?az;\; IMemPC!\{aa[31..1], az\},$
                          [$az = 0 \longrightarrow\; OP!1$ [] $az = 1 \longrightarrow\; OP!2$]
        ]
    ];

    [ $pcz = 0 \longrightarrow\; IMemPC!pca,\; OP!0,\; S0!0,\; M0!0$
    [] $pcz = 1 \;\wedge\; newpc! = 0 \longrightarrow\; IMemPC!pca,\; OP!0,\; S0!4,\; S1!0,\; M0!1$
    [] $pcz = 1 \;\wedge\; newpc \;\; = 0 \longrightarrow\; S1!0,\; M0!1$
    ]
  ]

}
```

During every cycle, the instruction (*i*) is input and decoded (*iLen*, *newpc*). The next 16-bit program counter is computed (a concatenation of the variable *pca* and the lsb variable *pcz*) and either sent to the instruction memory, or to an external unit for storage in case of branches. The instruction bytes read from the program memory are aligned through output control channels ($S0!$, $S1!$, $M0!$, etc.) before being used within the fetch or sent to the external decode unit.

6.2 Sequential Transformations

Multiple instances of addition by one ("$pca := pca + 1$") are possible in certain iterations of the sequential code. These operations are therefore replaced in the original text by communications on channels linked to a newly isolated 16-bit incrementer ("$INC!pca; SUM?pca$").

There are no nested loops within the Fetch unit (indeed, nested loops are rare in hardware specifications), and so we proceed to convert the sequential program into dynamic single-assignment form. The original variables *pca*, *pcz*, and *newpc* are split into five, four, and two new DSA variables, respectively. Input channels *Addra*?, *Addrz*?, and the new incrementer channel *SUM*? are all used multiple times within the program, and so new non-DSA input variables *addra_in*, *addrz_in* and *sum_in* are inserted for their eventual projection. (For example, an original incrementer instruction may now be written as "$INC!pca\_0; SUM?sum\_in; pca\_1 := sum\_in.$")

Next, since the incrementer will be implemented as two separate 8-bit incrementers, we vertically decompose our 16-bit channels and variables into two 8-bit versions. While byte encodings are eventually implemented as four e1of4 encodings, we decide to leave them intact prior to clustering to ensure that they will always finish in the same cluster. After clustering, they will be vertically decomposed further, and during clustering any control channels to byte processes are weighted more heavily to acknowledge their extra switching load. This choice makes the layout more regular and easier to design, and is an example of the tradeoffs that often arise when performing a combination of manual and automated synthesis.

It now remains to perform guard encoding before DDD's projection phase decomposes the sequential program. From the original code, we see that there are four main selection statements in the sequential code. (Nested selections are considered part of their outermost selection statements.) When we apply the guard-encoding tests to these selections, three of the four should be encoded. The computations for each selection are given in Figure 6.2. The channels and variables of the final

Guard Variables	N_G	A_G	C_{V_G}	C_{unenc}	C_{enc}	Encoded Variables
pcz_0, $iLen$, $newpc\_0$	16	12	3	36	27	gx (e1of4), gy (e1of4)
ig	2	1	1	1	2	Unencoded
$irpt$, $newpc\_0$	8	14	3	42	31	hx (e1of2), hy (e1of4)
pcz_3, $newpc\_1$	3	5	2	10	7	jx (e1of3)

Figure 6.2: Guard encoding for the Fetch example.

Variable Type	Name	Encoding
Input Channels	$I?$, $Addra\_B0?$, $Addra\_B1?$, $SUM\_B0?$, $SUM\_B1?$	byte
	$Addrz?$, $IG?$	e1of2
	$IOK?$	e1of6
Output Channels	$IMemPC\_B0!$, $IMemPC\_B1!$, $PtrPC\_B0!$, $PtrPC\_B1!$	byte
	$INC\_B0!$, $INC\_B1!$	byte
	$M0!$, $M1!$, $M2!$, $MA!$, $DIG!$	e1of2
	$OP!$	e1of3
	$S0!$, $S1!$	e1of5
Variables	i, aa_B0, aa_B1	byte
	pca_B0_0, pca_B0_1, pca_B0_2, pca_B0_3, pca_B0_4	byte
	pca_B1_0, pca_B1_1, pca_B1_2, pca_B1_3, pca_B1_4	byte
	sum_in_B0, sum_in_B1, $addra\_in\_B0$, $addra\_in\_B1$	byte
	pcz_0, pcz_1, pcz_2, pcz_3, az	e1of2
	ig, hy, $addrz\_in$	e1of2
	$iLen$, jx	e1of3
	$newpc\_0$, $newpc\_1$, gx, gy, hy	e1of4
	$irpt$	e1of6

Figure 6.3: Channels and variables in final DSA version of the Fetch.

DSA program are given in Figure 6.3.

6.3 Decomposition and System Transformations

When the projection phase of DDD transforms the final DSA program, the initially generated system has 88 processes (including copy processes and copy place-holding processes) and 219 channels. The first transformations that DDD applies to the decomposed system are distillation and elimination. For example, the process P_{az} should be distilled:

```
$P_{az} \equiv$ *[ $HX\_az?hx,\ \ HY\_az?hy;$
              [ $hx = 0 \wedge hy = 0\ \ \longrightarrow\ \ \textbf{skip}$
              [] $hx = 0 \wedge hy = 1\ \ \longrightarrow\ \ \textbf{skip}$
              [] $hx = 0 \wedge hy = 2\ \ \longrightarrow\ \ ADDRZ\_az?addrz\_in;\ \ az := addrz\_in;$
                                        $AZ\_imempcb0!az,\ \ AZ\_s0!az,\ \ AZ\_ma!az,\ \ AZ\_s1!az,\ \ AZ\_op!az$
              [] $hx = 0 \wedge hy = 3\ \ \longrightarrow\ \ ADDRZ\_az?addrz\_in;\ \ az := addrz\_in;$
                                        $AZ\_imempcb0!az,\ \ AZ\_s0!az,\ \ AZ\_ma!az,\ \ AZ\_s1!az,\ \ AZ\_op!az$
              [] $hx = 1 \wedge hy = 0\ \ \longrightarrow\ \ \textbf{skip}$
              [] $hx = 1 \wedge hy = 1\ \ \longrightarrow\ \ \textbf{skip}$
              [] $hx = 1 \wedge hy = 2\ \ \longrightarrow\ \ ADDRZ\_az?addrz\_in;\ \ az := addrz\_in;$
                                        $AZ\_imempcb0!az,\ \ AZ\_s0!az,\ \ AZ\_ma!az,\ \ AZ\_s1!az,\ \ AZ\_op!az$
              [] $hx = 1 \wedge hy = 3\ \ \longrightarrow\ \ ADDRZ\_az?addrz\_in;\ \ az := addrz\_in;$
                                        $AZ\_imempcb0!az,\ \ AZ\_s0!az,\ \ AZ\_ma!az,\ \ AZ\_s1!az,\ \ AZ\_op!az$
              ] ]
```

In this process, the same operations are being performed for variables *addrz_in* and *az* regardless of the values of *hx* and *hy*. The channels HX_az0 and HY_az are therefore eliminated, and the simpler distilled version of P_{az} generated:

```
$P'_{az} \equiv$ *[ $ADDRZ\_az?addrz\_in;\ \ az := addrz\_in;$
        $AZ\_imempcb0!az,\ \ AZ\_s0!az,\ \ AZ\_ma!az,\ \ AZ\_s1!az,\ \ AZ\_op!az$ ]
```

The new version is now a simple copy process.

An example of partial distillation can be found in the process P_{MA}:

```
*[ $HX\_ma?hx,\;\; HY\_ma?hy;$
   [ $hx = 0 \wedge hy = 0 \;\longrightarrow\; \textbf{skip}$
   ▯ $hx = 0 \wedge hy = 1 \;\longrightarrow\; \textbf{skip}$
   ▯ $hx = 0 \wedge hy = 2 \;\longrightarrow\; AZ\_ma?az;$ [$az = 0 \;\longrightarrow\; MA!(0)$ ▯ $az = 1 \;\longrightarrow\; MA!(1)$]
   ▯ $hx = 0 \wedge hy = 3 \;\longrightarrow\; AZ\_ma?az$
   ▯ $hx = 1 \wedge hy = 0 \;\longrightarrow\; \textbf{skip}$
   ▯ $hx = 1 \wedge hy = 1 \;\longrightarrow\; \textbf{skip}$
   ▯ $hx = 1 \wedge hy = 2 \;\longrightarrow\; AZ\_ma?az;$ [$az = 0 \;\longrightarrow\; MA!(0)$ ▯ $az = 1 \;\longrightarrow\; MA!(1)$]
   ▯ $hx = 1 \wedge hy = 3 \;\longrightarrow\; AZ\_ma?az$
   ] ]
```

In this case, the control variable *hy* is necessary to distinguish between the operations that need to be performed but variable *hx* is not. We therefore eliminate the channel *HX_ma* from the system, and the newly simplified process is

```
*[ $HY\_ma?hy;$
   [ $hy = 0 \;\longrightarrow\; \textbf{skip}$
   ▯ $hy = 1 \;\longrightarrow\; \textbf{skip}$
   ▯ $hy = 2 \;\longrightarrow\; AZ\_ma?az;$ [$az = 0 \;\longrightarrow\; MA!(0)$ ▯ $az = 1 \;\longrightarrow\; MA!(1)$]
   ▯ $hy = 3 \;\longrightarrow\; AZ\_ma?az$
] ]
```

Together, the distillation and elimination techniques result in a system with 55 processes and 168 channels, reducing the clustering tool's computed energy cost of the system by roughly 15%. These results are summarized in Figure 6.4.

Next the decomposed system is clustered, using a simulated annealing tool that simultaneously performs recomposition and homogeneous slack matching. Since the system is heterogeneous, with no processes having lesser delays than the intended slack buffers, the results are over slack-matched, and consume more energy than necessary. The results of this experiment are therefore worse than

Transformation	No. Processes	No. Channels	Normalized Energy Cost
Manual Decomp	12	45	1
DSA/Projection	88	219	7.3
Distill/Elim I	55	168	6.1
Cluster I	36	124	2.8
Distill/Elim II	27	94	2.1

Figure 6.4: Results of clustering and distillation.

can be expected with a heterogeneous clustering tool, and as such are very encouraging. When given a maximum cycle time of 22 transitions (the same target cycle time used in the manually decomposed Lutonium Fetch unit), the clustered system contains 36 regular processes (excluding slack buffers) and 124 channels, with a further reduction of 55% in energy cost compared to the unclustered distilled system.

Since the system has changed, we can perform a second round of distillation, elimination, and clustering. The final result is a system with 27 processes (excluding slack buffers) and 94 channels, with another 23% reduction in energy cost compared to the first clustered system. Overall, even taking into account the pessimistic slack matching, clustering has therefore reduced the energy cost of the system by 70%. The final system is illustrated in Figure 6.5.

Figure 6.4 summarizes the results of the distillation and clustering transformations, and compares them to the manual decomposition that was performed (over a period of weeks) for the original Lutonium Fetch unit. Both decompositions began with the same initial CHP. We can see that the DDD system has the same throughput but roughly twice the energy as the manually decomposed system. Heterogeneous slack matching would reduce this energy factor somewhat. These results are more than satisfactory, and shall only improve as the capabilities of the tools expand.

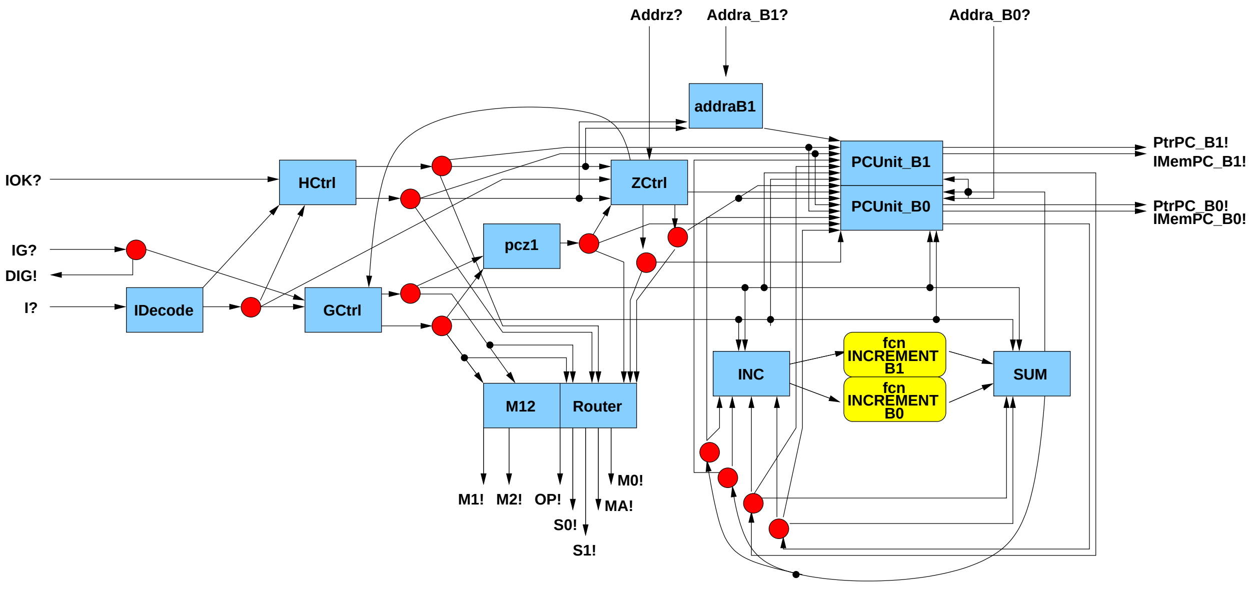


Figure 6.5: DDD example.
The system that results when DDD is applied to the Fetch unit of the asynchronous 8051 microcontroller. The circles represent simple copy processes.

Chapter 7

Reconfigurable Asynchronous Circuits

This chapter presents an architecture for an asynchronous QDI field-programmable gate array (FPGA). The basic reconfigurable cell is an asynchronous precharged half-buffer stage with conditional inputs and outputs. These *logic cells* are then grouped together to form a larger computation block, or *cluster*, with internal copies and feedback. DDD can be modified to map systems onto this FPGA architecture. Beyond its standard algorithms, DDD can use encoding and vertical decomposition techniques to guarantee that every process generated by its DSA and clustering phases can be mapped to a basic FPGA cell. Then, clustering and slack matching can be performed to map these cells to FPGA clusters without actually recomposing their processes. Together, DDD and the reconfigurable architecture presented here make the rapid prototyping of high-performance asynchronous VLSI systems possible.

The logic cells and clusters initially presented in this chapter are dual rail. After demonstrating how designs can be implemented on this architecture, we discuss the merits of different channel encodings, and create models to analyze general asynchronous QDI reconfigurable architectures.

7.1 Motivation and Background

Programmable logic is becoming increasingly attractive as the decrease in CMOS feature-size continues to improve the size and performance of programmable devices. High power consumption and difficult timing closure represent the current downside of reprogrammability. Since two of the main advantages of asynchronous logic are low power consumption and the absence of the global time constraints imposed by clocks, it seems natural to apply asynchronous technology to the design of FPGAs.

Timing issues present difficulties in synchronous FPGA design because the mapping procedure (partitioning, placement, and interconnect routing) may cause violation of necessary timing constraints. Eliminating the clock greatly relaxes the constraints, but there are still some timing requirements to be met in asynchronous design. As exposited in Chapter 1, the different asynchronous techniques distinguish themselves through their timing assumptions. The QDI style used in this thesis makes the weakest timing assumptions. The circuits are completely delay-independent except for some forks—called *isochronic forks*—in which the propagation delays on the different branches of the fork are similar.

Automatic placement by a mapping procedure could violate the delay assumption on isochronic forks. An important advantage of our joint FPGA architecture and DDD synthesis method is that DDD guarantees that all isochronic forks are local to the logic cells and therefore are unaffected by the placement procedure. Communication between cells is entirely delay-insensitive: It is implemented as a four-phase handshake, where the data to be transmitted is encoded in an e1ofN code. Hence, the mapping procedure does not have to meet any timing requirement, which makes this architecture particularly suitable for dynamically configurable systems.

Of course, the mapping algorithms will consider performance metrics such as latency and path-length. In fact, homogeneous slack matching can be performed. The advantage of delay-insensitive design can be seen when one component of the system does not meet timing constraints but is off the critical path or used only infrequently. Instead of dedicating computation time to eliminating timing errors from this component, it can be left as is, and the whole system will still function correctly and close to the desired speed.

Several proposals for asynchronous FPGAs have already appeared in the literature [45, 22, 33, 25, 48, 51]. Some of the approaches provide the same level of functionality as our PCHB pulldown logic

(3-4 input look-up tables, called *LUTs*), but most do not offer an implementation of the input and output handshakes and sequencing—say, the equivalent of the control part of the PCHB. In these alternative proposals, the control has to be implemented explicitly, exposing the issues of timing assumptions and isochronic forks, and adding large efficiency penalties in terms of area, cycle time, and energy. In all but [48] and [51], either timing assumptions are needed to separate the control from the datapath, or the implementation of the sequencing results in unpredictable placement of isochronic forks.

Rettberg and Kleinjohann offer a delay-insensitive FPGA architecture [48] but it has been proved that the class of entirely delay-insensitive circuits is very limited [36]. Recently, Teifel and Manohar have published a new design that also uses PCHBs but uses multiple types of PCHBs with different communication patterns as basic cells [51]. When systems are mapped, the majority of these cells may be unusable. In contrast, the architecture presented in this chapter uses one general cell that can be programmed with different communication patterns. This single cell is more complex, but is always usable when a system is being mapped on the FPGA. It may be that architectural choices are each better suited for different classes of applications. A clearer picture will emerge when we combine our synthesis tools with the area and performance estimation tools under development, based on the general architectural model described at the end of this chapter.

The rest of this chapter describes a new architecture for asynchronous QDI FPGAs that is cluster-based. An overview of the architecture is shown in Figure 7.1.

7.2 Logic Cells

Our basic logic cell consists of a single PCHB circuit with three dual-rail input channels and one dual-rail output channel. Each cell contains twelve programmable SRAM cells: eight to configure the cell's computation and four to configure the cell's communication patterns. The CHP specification of the cell is given in Figure 7.2. Sb, Sc, Si and Sz are the programmable bits that configure communications, while S is an array of the eight programmable bits that control the computation. The CHP specification is given in Figure 7.2.

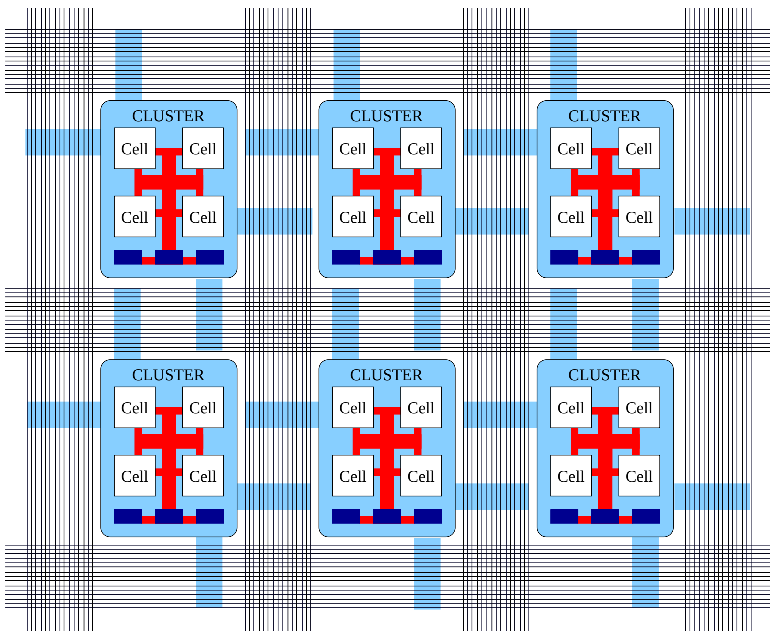


Figure 7.1: Quasi delay-insensitive asynchronous FPGA.

```
$Cell \equiv$  *[$A?a$;
                 [$\neg Sb \longrightarrow$  $b := 0, c := 0$
                 [] $Sb \wedge \neg Si \longrightarrow$ [$\neg Sc \longrightarrow B?b, c := 0$ [] $Sc \longrightarrow B?b, C?c$]
                 [] $Sb \wedge$  $Si \longrightarrow$
                      [$\neg Sc \longrightarrow$ [$\neg a \longrightarrow B?b, c := 0$ [] $a \longrightarrow b := 0, c := 0$]
                      [] $Sc \longrightarrow$ [$\neg a \longrightarrow B?b, c := 0$ [] $a \longrightarrow b := 0, C?c$]
                      ]
                 ];
                 [$\neg Sz \longrightarrow Z!f(S, a, b, c)$
                 [] $Sz \longrightarrow$ [$a \longrightarrow Z!f(S, a, b, c)$ [] $\neg a \longrightarrow$ **skip**]
                 ]
               ]
```

Figure 7.2: CHP specification for our asynchronous FPGA cell.
Basic logic cells are grouped together in clusters which contain additional conditionality and serve as the interface to the interconnect. *Sb*, *Si* and *Sc* are SRAM cells that program input conditions; *Sz* is an SRAM cell that programs output conditions; *S* is an array of eight SRAM cells that together program the computation.

7.2.1 Reconfigurable PCHB Circuit

The basic logic cell largely follows the PCHB template given in Chapter 3 and shown in Figure 3.7. There is a computation stage that consists of a pulldown network of n-transistors and a simple pullup network of p-transistors. Input and output channel validity trees are also present, along with

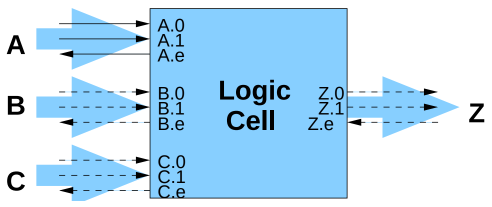


Figure 7.3: Logic cell for the asynchronous FPGA.
The cell includes three dual-rail input channels, and one dual-rail output channel. Dashed lines indicate channels on which communications may be conditional.

completion logic to generate input enable signals. Again, these components must account for data dependencies when conditional communications are programmed.

We limit the cell to one output channel as multiple output channels would require multiple separate precharged computation stages that could go unused. (Area is an important factor when designing FPGA cells since they are repeated across the chip.) Instead, by grouping single-output logic cells inside clusters where they can share input channels, there is some redundancy in validity circuitry when multiple outputs are computed from the same inputs, but no unusable circuitry when there is no sharing (as is often the case). We limit the cell to three input channels because with four input channels, the pulldown networks of the computation logic grow too long for high performance.

For our prototype design, we choose dual-rail encoding since its binary nature keeps low the number of SRAM cells required to program the compute function. This encoding also allows us to implement inverting functions in the routing network by simply swapping the connections to the two data rails, instead of using up an entire logic cell to perform this task. Other channel encoding choices are discussed later in Section 7.5.2. The basic circuit for the cell is illustrated in Figures 7.3—7.6. (These are not an exact representation of the circuit: details such as the reset circuitry and staticizers for each of the C-elements have been omitted.)

There are several deviations from the original PCHB template. One difference is that instead of using both the output channel enable $Z.e$ and the local signal enable en in the pullup network and foot transistors of the precharged computation stage, we combine the output enable with internally-generated signals via a C-element into a single new signal, “go.” This reduces the number of transistors in series of the pulldown network which, because of the extra logic required to program

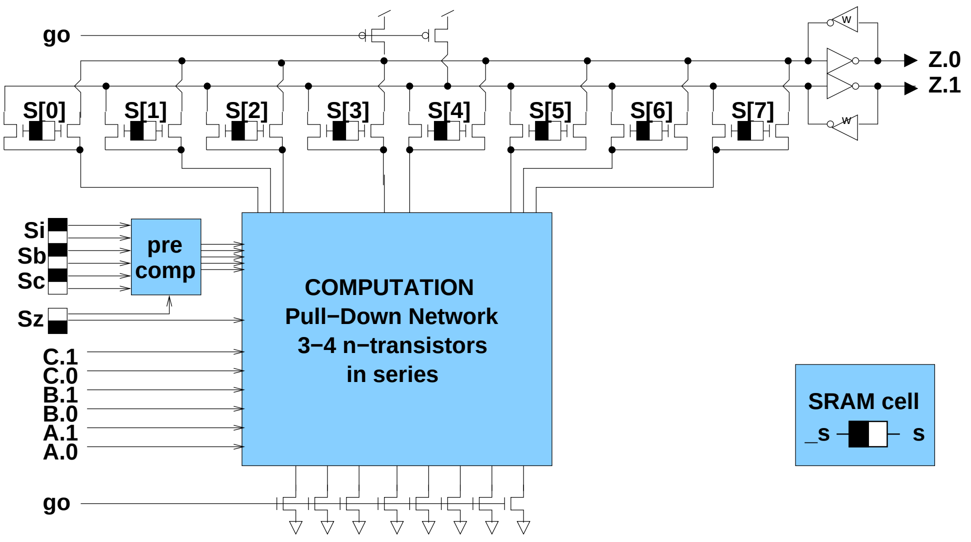


Figure 7.4: Reconfigurable cell: computation circuitry.

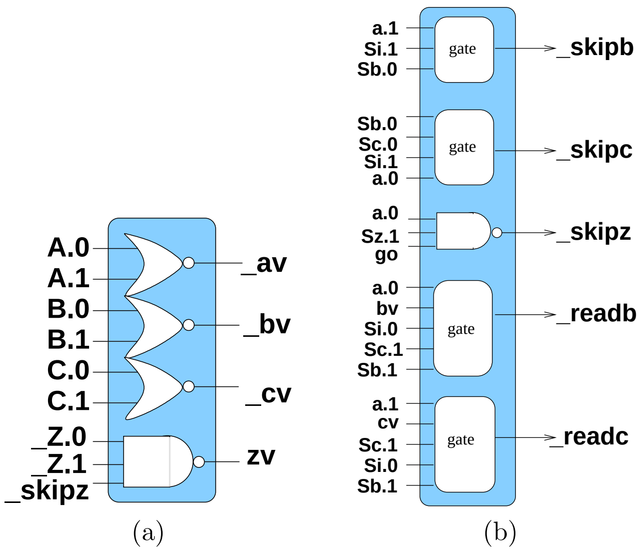


Figure 7.5: Reconfigurable cell.
(a) Input and output validity generation. (b) Circuitry for conditional communications.

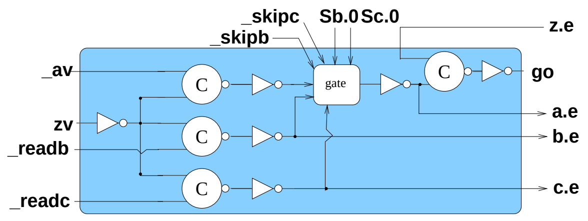


Figure 7.6: Reconfigurable cell: completion circuitry.

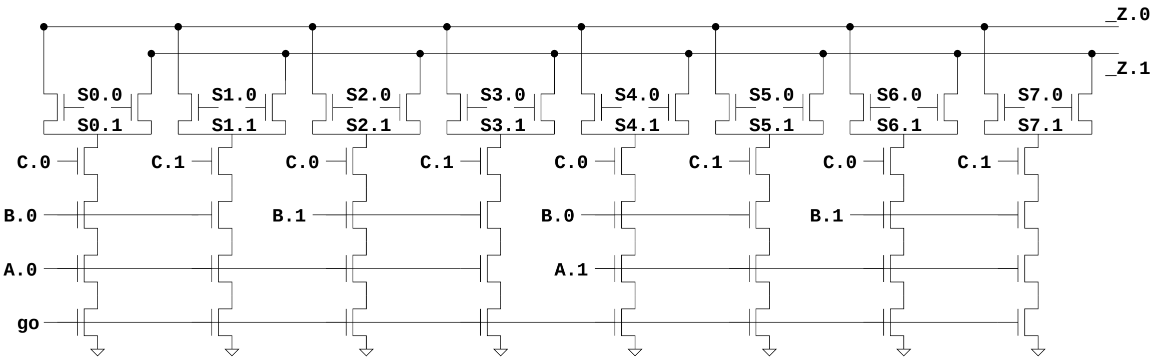


Figure 7.7: Pulldown networks: chosen configuration.
SRAM cells placed next to the outputs in a sample pulldown network for programmable PCHB with unconditional communications. (48 transistors, no charge-sharing problems.)

the computations, is already large. The size of the computation network also leads to the second deviation: instead of a single foot transistor, multiple feet provide parallel paths to ground. This alleviates otherwise inevitable charge-sharing problems. Finally, the "pre-comp" unit in Figure 7.4 consists of combinational gates that have been extracted from the pulldown network to reduce the number of n-transistors in series. Since the unit depends only on SRAM-cell outputs, it does not add to the forward latency or the cycle time of the PCHB after the cell has been programmed.

Meanwhile, for conditional communications, instead of the general solutions presented in Chapter 3, we create a specific implementation for our cell. This involves the use of non-precharged gates to generate the programmable conditions, as shown in Figure 7.5. The output condition signal *_skipz* is the equivalent of the extra dummy rail presented in earlier sections, and is similarly used in the output validity. The internal signals *_readb* and *_readc* are analogous to *_useb*.1 and *_usec*.1 in the general solution for conditional inputs, while *_skipb* and *_skipc* are analogous to *_useb*.0 and *_usec*.0, respectively.

Note that in our pulldown computation networks, we chose to place the SRAM transistors at the top (next to the output nodes) rather than at the bottom (next to the foot transistor for *go*). The two alternate implementations for a three-input logic cell with no conditional communications are shown in Figures 7.7 and 7.8. The second design has large internal nodes and likely requires additional internal precharge p-transistors (not shown in the diagram) to avoid charge-sharing problems. Since

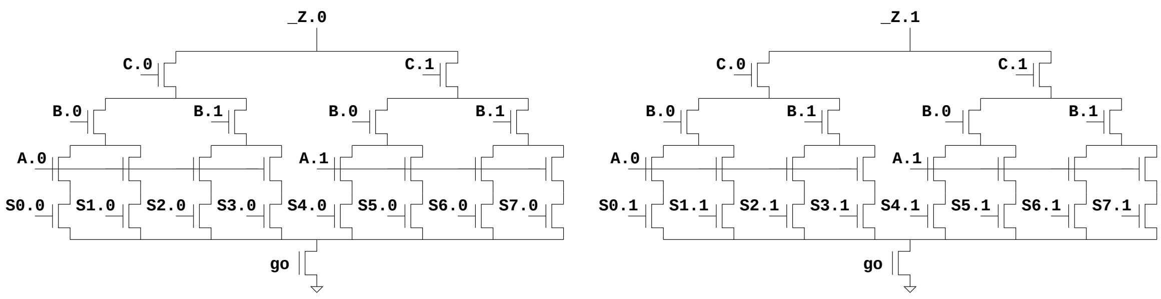


Figure 7.8: Pulldown networks: alternate configuration.
SRAM cells placed next to the foot transistors in a sample pulldown network for programmable PCHB with unconditional communications. (46 transistors, not including extra precharge transistors to alleviate charge-sharing problems.)

aside from the additional precharge transistors the two configurations have roughly the same size, we choose the configuration without charge-sharing problems for our basic cell.

This decision is rejustified later when we consider e1of4 channels, since even without considering extra precharge transistors, the second configuration uses over 10% more area than the first. In general, if we consider basic cells with K e1ofM input channels, then the computation configuration with SRAM transistors on the top requires

$$N_{top}(M,K) = M^K(M\lceil log_2(M)\rceil + K + 1) = M^{K+1}\lceil log_2(M)\rceil + (K+1)M^K$$

transistors, while the second implementation, with SRAM next to the foot, requires

$$N_{bot}(M,K) = M(M^K\lceil log_2(M)\rceil + \sum_{i=0}^{K} M^i) = M^{K+1}\lceil log_2(M)\rceil + \frac{M(M^{K+1}-1)}{M-1}$$

transistors.

7.2.2 Communication Patterns

Using the programmable bits Sb, Si, Sc and Sz, the cell can be configured to use either one, two, or three inputs. The cell's input channel $A?$ is always unconditional (a communication must occur on every cycle) but communications on all of the other channels—input channels $B?$ and

Sb	Sc	Si	Sz	a	b	c	$Z!$
0	X	X	0	X	0	0	$f(S,a,0,0)$
0	X	X	1	0	0	0	—
0	X	X	1	1	0	0	$f(S,1,0,0)$
1	0	0	0	X	$B?$	0	$f(S,a,b,0)$
1	0	0	1	0	$B?$	0	—
1	0	0	1	1	$B?$	0	$f(S,1,b,0)$
1	1	0	0	X	$B?$	$C?$	$f(S,a,b,c)$
1	1	0	1	0	$B?$	$C?$	—
1	1	0	1	1	$B?$	$C?$	$f(S,1,b,c)$
1	0	1	0	0	$B?$	0	$f(S,0,b,0)$
1	0	1	0	1	0	0	$f(S,1,0,0)$
1	0	1	1	0	$B?$	0	$f(S,0,b,0)$
1	0	1	1	1	0	0	—
1	1	1	0	0	$B?$	0	$f(S,0,b,0)$
1	1	1	0	1	0	$C?$	$f(S,1,0,c)$
1	1	1	1	0	$B?$	0	$f(S,0,b,0)$
1	1	1	1	1	0	$C?$	—

Figure 7.9: Communication patterns for our asynchronous FPGA cell.

$C?$, and output channel $Z!$—can be programmed to depend upon the value input upon $A?$. These conditional communications allow each cell to be configured to fit ten different communication patterns, some of which include data-dependency. From these patterns, a cell can not only implement basic computation blocks but also, either singly or in combination with another cell, two basic control circuits in asynchronous QDI design: the controlled merge and the controlled split. The importance of these cells is demonstrated in the *FBlock* example of Section 7.4.

7.2.3 Performance and Area

We have created layout and performed analog SPICE simulations for our logic cell in TSMC 0.18-μm technology. The simulations show that a cell operates with a cycle time of 190-235 MHz (depending upon its configuration), and consumes anywhere from 2.1-3.1 pJ/cycle. The results are summarized in Figure 7.10.

The FPGA cell layout, including twelve bits of SRAM, is shown in Figure 7.11. Its dimensions are 334x323-lambda (1079 μm^2). While the design of the asynchronous logic cell may seem bulky compared to basic logic cells in synchronous FPGAs, we note that QDI designs face a similar area penalty in custom VLSI design yet outperform their synchronous counterparts in both speed and

Sb	Sc	Si	Sz	communication	cycle time MHz	energy pJ/cycle
0	X	X	0	$A?,Z!$	206	2.4
0	X	X	1	$A?,(Z!)$	235	2.1
1	0	0	0	$A?,B?,Z!$	195	2.8
1	0	0	1	$A?,B?,(Z!)$	222	2.4
1	0	1	0	$A?,(B?),Z!$	200	2.6
1	0	1	1	$A?,(B?),(Z!)$	217	2.4
1	1	0	0	$A?,B?,C?,Z!$	190	3.1
1	1	0	1	$A?,B?,C?,(Z!)$	220	2.7
1	1	1	0	$A?,(B?,C?),Z!$	199	2.8
1	1	1	1	$A?,(B?,C?,Z!)$	214	2.6

Figure 7.10: FPGA simulation results.
Results of SPICE simulations for different configurations of the basic cell in 0.18-μm technology. Parentheses indicate that communications depend upon the data input on channel $A?$.

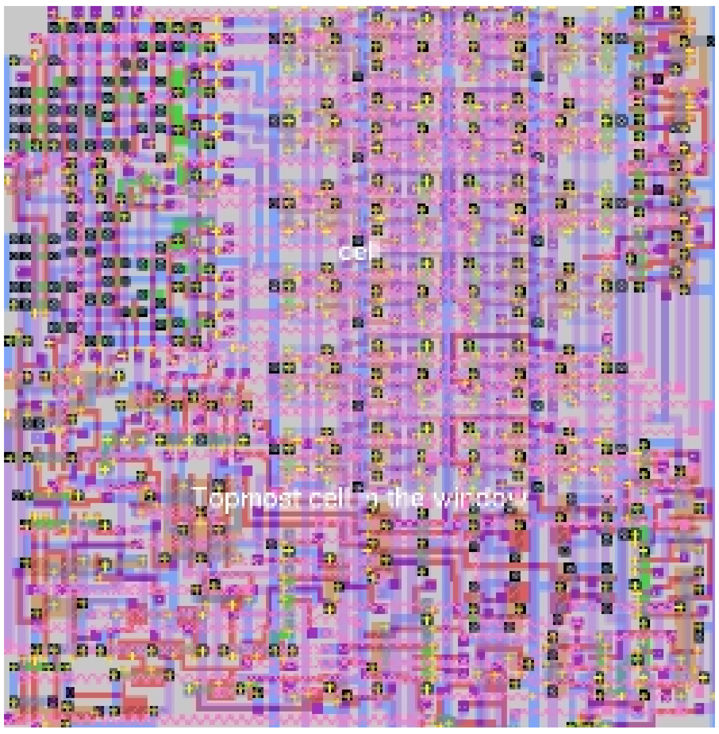


Figure 7.11: Layout for the basic FPGA cell, including twelve programmable SRAM bits.

energy consumption [42]. The interconnect area penalty that arises from the multiple wires of a dual-rail channel is also offset in part by the omission of a clock tree in an asynchronous FPGA.

7.3 Cluster Design

The logic cells serve as versatile building blocks for an asynchronous FPGA. However, additional functionality and interfacing can help implement programs and embed the cells into an interconnect mesh more efficiently. To incorporate features such as channel replication, slack-matching buffers for performance optimization, and initial tokens generated at reset, we group cells together into structures called *clusters*. While a more general model is presented in Section 7.5, a cluster in this section consists of four basic logic cells, $L[0]$–$L[3]$ with additional programmable SRAM and circuitry to implement the new features. This design is illustrated in Figure 7.12, and its features are described below.

7.3.1 Copying Channels

QDI channels cannot be split or shared between PCHBs without additional completion circuitry to acknowledge every transition on the channel. If the data rails of a channel are split and copied to two different input ports, then the two enable rails from those ports must be collected in a C-element with that gate's output serving as the new enable rail for the copied channel. Clusters include extra enable rails and C-elements so that any input channel can be copied either to $L[0]$ and $L[1]$, to $L[2]$ and $L[3]$, or to all four logic cells. To configure the cluster and copy input channels, instead of connecting separate enable rails (e.g., both $A[0].e$ and $A[1].e$) to the interconnect mesh, connect instead their copy enable generated by a C-element (e.g., cpA_01_e).

In cases where a channel needs to be copied to two cells in different clusters, an extra enable rail is also provided for each output channel. This rail is sent through a C-element with the original output enable rail and a multiplexer programmed by the SRAM bit $Scpz_i$ is used to choose the enable signal from either the original single output enable signal or the combined copy enables.

7.3.2 Feedback Channels

There are many cases when the output channel of a logic cell in a cluster is required as an input to another cell (or, perhaps to several other logic cells) in that same cluster. One example is when

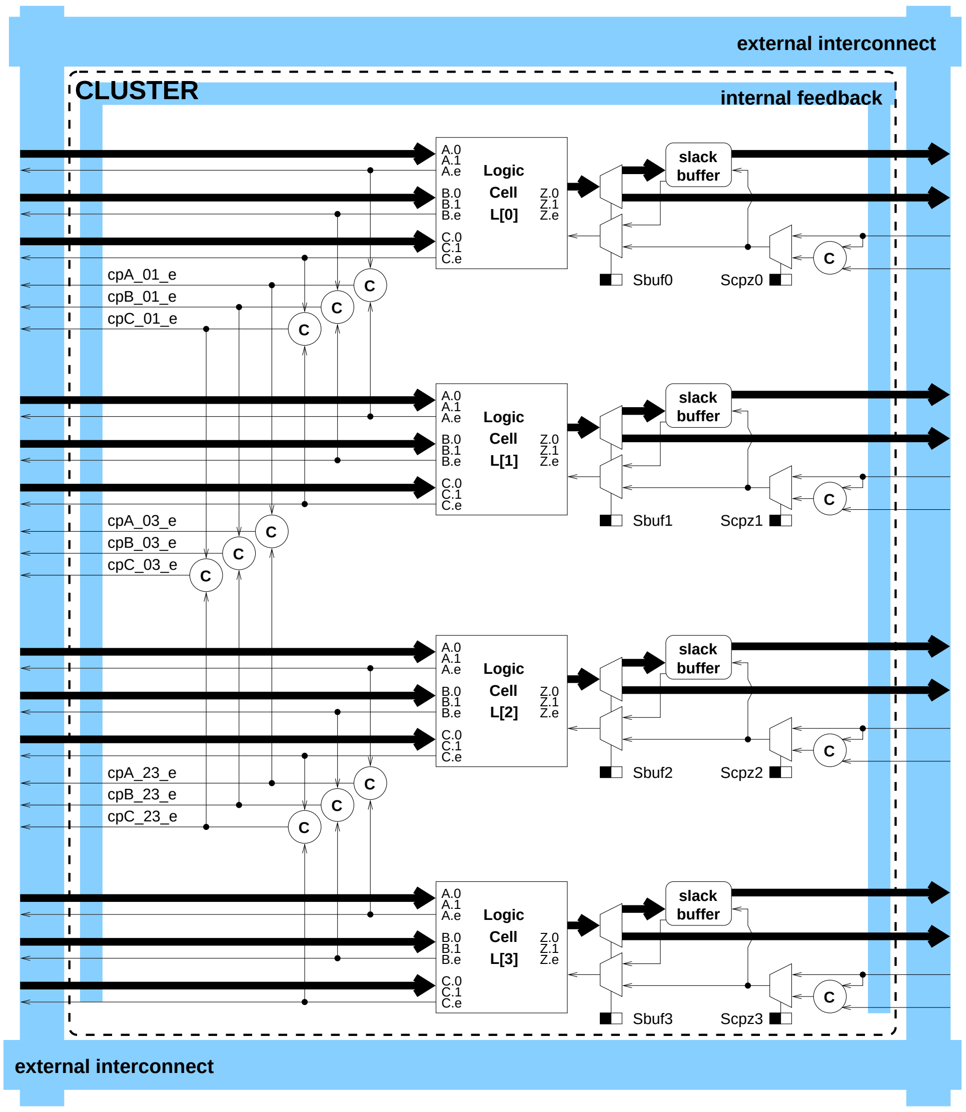


Figure 7.12: Cluster block diagram.
Includes circuitry implementing channel copying, feedback, slack buffers, and initial reset tokens.

multiple bits of a full adder are grouped together in one cluster and the carry-out of one bit must be fed into multiple cells implementing the next bit. To implement this scenario as simply as possible, feedback channels are provided within the cluster. The logic cells can be programmed to connect to these channels in the same way that they can connect to external interconnect—the new channels are completely local.

7.3.3 Buffering and Initial Tokens

Since all cells in the FPGA are identical, optimizing the performance of a system implemented on the FPGA requires homogeneous slack matching. While the logic cell can be programmed to serve as a slack buffer, this is its simplest configuration and wastes much of its circuitry. We therefore insert small dedicated slack buffers to the system. Each output channel contains a QDI "slack buffer" that the output signals may be programmed (using the SRAM bits $Sbuf_i$) to either skip or pass through.

These slack buffers actually serve a dual-purpose and are also used at reset, when a system may be initialized with tokens of data on certain channels. Because such channels require slightly different completion circuitry, we choose to include the circuitry in the PCHBs, where it can be implemented more efficiently than in the basic cells. Thus, the slack buffers can be programmed with different reset token configurations: no token, initial token zero, and initial token one.

7.3.4 Summary

Aside from SRAM cells used to connect channels to interconnect, the cluster described in this section contains 60 programmable SRAM cells: twelve within each logic cell, four to select whether or not each logic cell output will pass through a slack buffer, four to choose whether or not each output channel will be copied, and four to program initial tokens on the output channels.

The programmable features allow clusters to be used in a variety of ways, ranging from four independent computation cells to an eight-way copy for a single channel. In attempting to provide a flexible cluster architecture for our FPGA, we may have added too much functionality to the clusters. For example, although the bypass feature has proved very useful in decompositions thus far, it is not yet clear whether slack-matching buffers (and their programmable initial tokens) are really required for every logic cell, whether there should be fewer such buffers per cluster, or whether they should be moved out of the clusters entirely and inserted into the switches in the routing network. These are all issues for further study and experimentation.

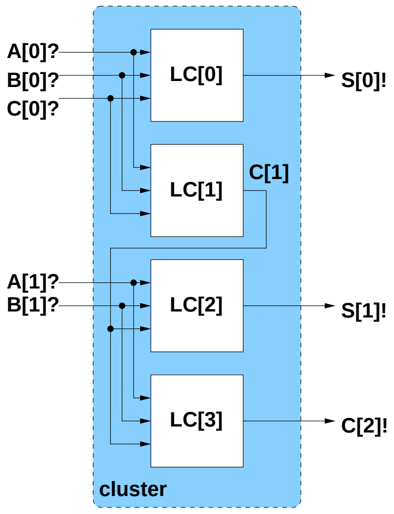


Figure 7.13: FPGA full adder example.
One cluster implementing two bits of a full adder with input channels A and B, sum output channel S, and carry channels C.

7.4 Mapping Examples

In this section, we demonstrate different possible configurations of the clustered architecture described in Sections 7.2 and 7.3 by decomposing programs of different sizes and mapping them to cells and clusters. We start by considering a simple full adder and then a four-bit ALU. The final example is a 32-bit datapath unit taken from an asynchronous microprocessor.

7.4.1 Full Adder and ALU

One bit of a ripple-carry full adder can be implemented using two dual-rail logic cells. When multiple bits are required, copy and feedback channels allow each cluster to implement two bits of the adder. A possible cluster setup is shown in Figure 7.13.

Building upon this full adder, consider the implementation of a four-bit ALU that can perform addition, subtraction, logical AND, and logical OR. Such a system requires two clusters to implement the full adder cells, one cluster to negate the subtrahend in case of subtraction, and one cluster to implement the boolean functions. We also need three more clusters to conditionally send the inputs and opcodes to either the full adder or the boolean computation blocks, and one cluster to merge the outputs of these two sets of blocks. The sum total of cells required to implement this four-bit ALU is 32. The cells can be mapped to eight clusters.

7.4.2 Microprocessor Execution Unit

We have decomposed the *FBlock* execution unit from the asynchronous MiniMIPS [39] for implementation on our clustered architecture. The *FBlock* is a classic datapath unit (32-bits wide) with simple control. Its initial high-level CHP specification is as follows:

```
$FBlock($ $C?$            $:$ $e1of2$[5];
          $X?, Y?, Z!$      $:$ $e1of2$[32];
          $ImL?, ImH?$      $:$ $e1of2$[16] $)$ $\equiv$
*[ $C?c$;
    [$c \in \{and, or, xor, nor\} \longrightarrow$  $X?x,\ Y?y,$  $Z!op(c)(x, y)$
    ▯$c \in \{andi, ori, xori\} \longrightarrow$  $X?x, ImL?il;$  $Z!op(c)(x, il)$
    ▯$c = lui \longrightarrow$  $ImH?ih;$  $Z!(ih*2^{16})$
  ] ]
```

The final decomposition of this unit consists of 834 logic cells, grouped in 209 clusters. All but two of the clusters are fully populated. 661 of the cells take advantage of conditional communications. On a related note, only 93 of the cells are active on every cycle. Hence, in the absence of a clock, only 93 of the 834 cells are consuming dynamic energy every cycle. These results help justify the additional circuitry inserted into PCHBs to implement conditional communications. A high-level view of the decomposed system is shown in Figure 7.14.

7.5 Architectural Models and Interconnect

We have presented the design of a basic e1of2 logic cell, and of a cluster that contains four such cells as well as four slack buffers. However, there are many other design points to consider when constructing a fast, energy- and area-efficient asynchronous FPGA. We have begun studying different

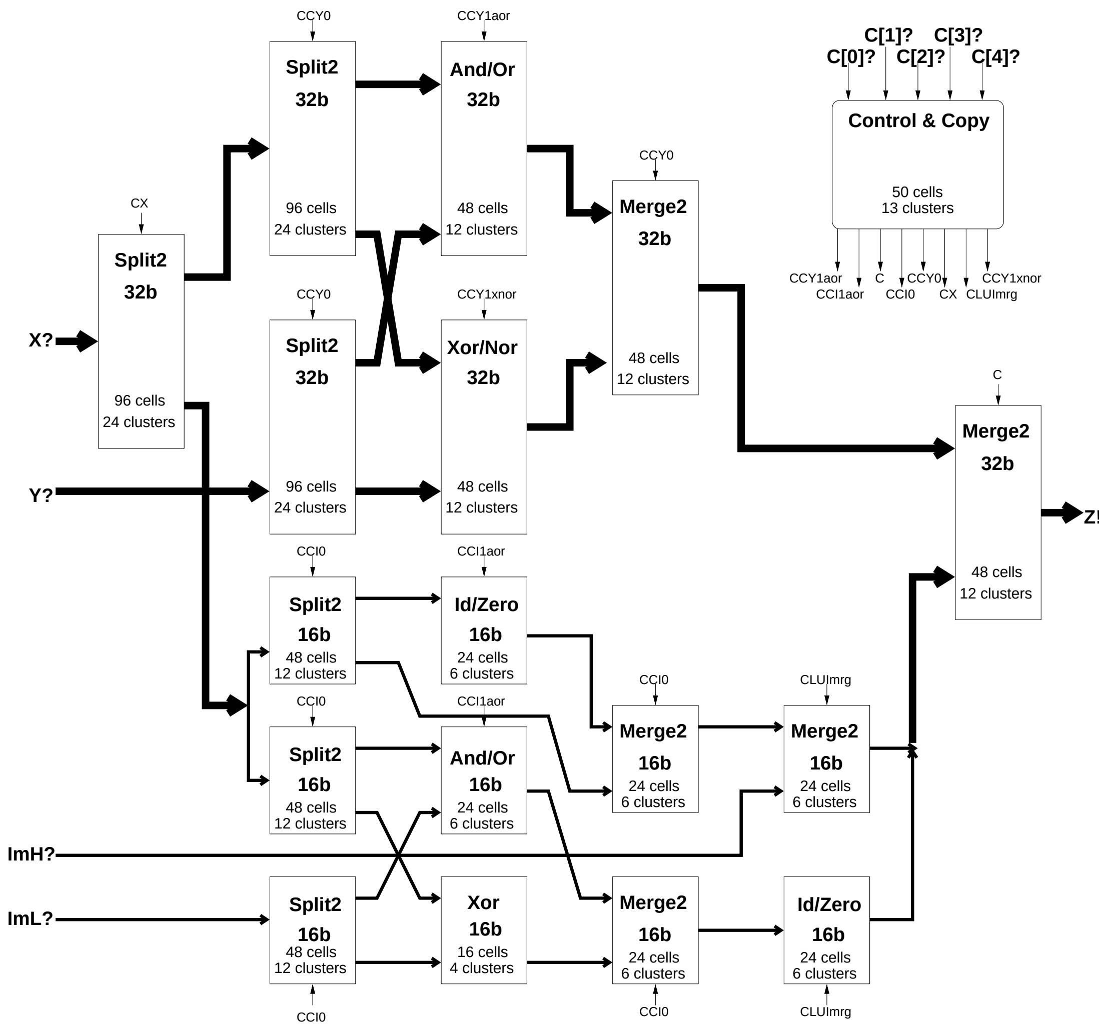


Figure 7.14: Decomposition of the *FBlock* execution unit from the asynchronous MiniMIPS.

designs with different channel types (performance considerations limit us to considering e1of2 and e1of4 encodings), varying cluster configurations, interconnect switches and routing architectures. This section describes our approach to creating a parameterized model (including both computation blocks and interconnect) for an asynchronous FPGA, and notes some initial results and trends concerning area that have surfaced when comparing synchronous FPGAs with asynchronous FPGAs that use e1of2 and e1of4 channel encodings.

In our analysis, we use the minimum-width transistor area estimation technique presented by Betz, Rose and Marquardt [6]. This method estimates total circuit area by using an experimentally-derived function to compute the area required by a transistor of a given size in units of the area

M (e1ofM)	K (fanin)	#SRAM cells	Total Area	Area: SRAM/total	Datapath Width	#Functions
2	3	12	400	0.18	1b	2^8
4	3	132	1777	0.45	2b	4^{64}
4	2	34	516	0.40	2b	4^{16}

Figure 7.15: Comparison of three reconfigurable asynchronous logic cells.
The logic cells are parameterized by channel width (M) and fanin (K). Area estimates are given in units of minimum-width transistors.

required by a minimally-sized transistor in the same technology.

7.5.1 Logic Cell Comparisons

At the logic cell level, the three-input e1of2 cell described in section 7.2 requires twelve SRAM bits and roughly 2.5 times the area of an equivalent three-input synchronous cell (which includes a 3-LUT, a multiplexer, and nine SRAM bits). Recall that the asynchronous e1of2 cell includes circuitry that allows conditional communications by only generating enable signals under certain data-dependent conditions. This circuitry may not be necessary in a synchronous design, but contributes to the energy-efficiency of our circuits since the majority of energy in asynchronous chips is consumed in communications and not computation. Also, recall that conditional communications on channels allow entire cells to be used (i.e., be "active") only conditionally. Since there is no global clock, asynchronous QDI cells do not consume any dynamic energy when they are inactive.

Compared to the e1of2 cell, an equivalent e1of4 cell (with the same control logic but twice the datapath) requires 132 SRAM bits and more than five times the area. The e1of4 cell can be programmed to compute 4^{4^3} different functions while the e1of2 version can compute only 2^{2^3}. However, details at the analog circuit level may cause the e1of4 cell to run slower than the smaller e1of2 cell. A comparison of the three cells is given in Figure 7.15.

7.5.2 Interconnect

Asynchronous circuits require more wires than synchronous circuits to communicate information. (In QDI asynchronous design, $N + 1$ wires are required in a channel that encodes $\lceil \log_2 N \rceil$ bits.)

Channel Type	#Gates per channel switch	#SRAM cells per channel switch	#Channels per $2n$-bits of datapath
sync/1-bit	1	1	$2n$
sync/2-bit	2	1	n
async/e1of2	3	1	$2n$
async/e1of4	5	1	n

Figure 7.16: Interconnect switches for different channel encodings.

However, the general interconnect of asynchronous FPGAs can be conceptually modeled after that of synchronous FPGAs, with an asynchronous channel being the equivalent of a synchronous wire. The area required by asynchronous interconnect does not simply scale with the number of wires. While each switch for an e1ofN channel involves $N + 1$ wires and programmable gates (either pass-gates or tri-state buffers), only one SRAM cell is required for configuration. Figure 7.16 characterizes the interconnect switches for some basic synchronous and asynchronous channel encodings. We include synchronous interconnect where each wire can be routed individually, synchronous interconnect where the wires are grouped in pairs to carry 2-bit quantities and are always routed together, and the asynchronous e1of2 and e1of4 channel encodings.

Consider the situation in which interconnect area is dominated by switches and not wires. (Conventional experience indicates this is the case, and while the growing number of metal layers in new technologies help us increase the density of wiring networks, switches currently require space on the substrate and cannot be easily stacked [15].) The specific computations performed in this section assume a network that uses 50% pass-gates and 50% tri-state buffers. Since tri-state buffers are usually more than twice the size of pass-gates, the numerical results will vary depending on the fraction of pass-gates used in interconnect. However, for each switching network architecture considered here, the relative ordering of the switching areas required by the various channel encodings remains the same no matter what combination of pass-gates and tri-state buffers are used.

We begin by noting that both synchronous 1-bit and e1of2 interconnect require the same number of channels to route a given datapath. Given a switching network architecture then, both types of interconnect will use the same number of switches; only the area per channel switch will differ.

Following the circuit area estimation technique described earlier, e1of2 switches require, on average, about 2.2 times the area of their synchronous 1-bit counterparts. Hence, no matter what type of switching network is chosen (i.e., no matter whether the number of switches grows linearly or super-linearly with the number of inputs and outputs to the network [50]), e1of2 interconnect will use up roughly 2.2 times as much area as synchronous 1-bit interconnect. Similarly, e1of4 interconnect will use up about 2.1 times the area as synchronous 2-bit interconnect, regardless of the type of switching network chosen. (The comparisons in this section ignore the wires required to distribute clock signals throughout a synchronous FPGA.)

In contrast, when comparing the two asynchronous encodings, e1of4 interconnect requires fewer channels but larger individual channel switches than e1of2 interconnect. The ratio of total interconnect areas for the two channel-encoding schemes therefore depends on the switching network architecture chosen. For example, consider a network that uses $O(N^2)$ switches, where N is the number of network sources and sinks. Given this switching architecture and equally wide datapaths, e1of4 interconnect uses up only 39% of the area of e1of2 interconnect. When the switching network uses $O(N)$ switches, e1of4 interconnect still requires less area than e1of2 interconnect, but at the higher fraction of 77%.

Typically, routing resources take up 90% and logic cells only 10% of the area of current synchronous FPGAs. If this ratio holds true for asynchronous FPGAs then, ignoring clusters and focusing on this section's area comparisons for logic cells and switching networks, an asynchronous e1of2 FPGA requires roughly 2.2 times the area of its synchronous 1-bit equivalent. (The fact that e1of2 cells and switching networks are similar factors larger than their 1-bit synchronous counterparts points to the ratio remaining roughly the same.)

The computation is more complicated when comparing e1of2 and e1of4 asynchronous FPGAs. One reason is the effect of different switching network architectures on the area required by e1of4 interconnect. Another is that while e1of4 cells can handle twice the datapath of e1of2 cells, the percentage of cells that make use of this doubled datapath depends on the nature of the system being implemented. In a datapath element such as the *FBlock* described in section 7.4, close to 95%

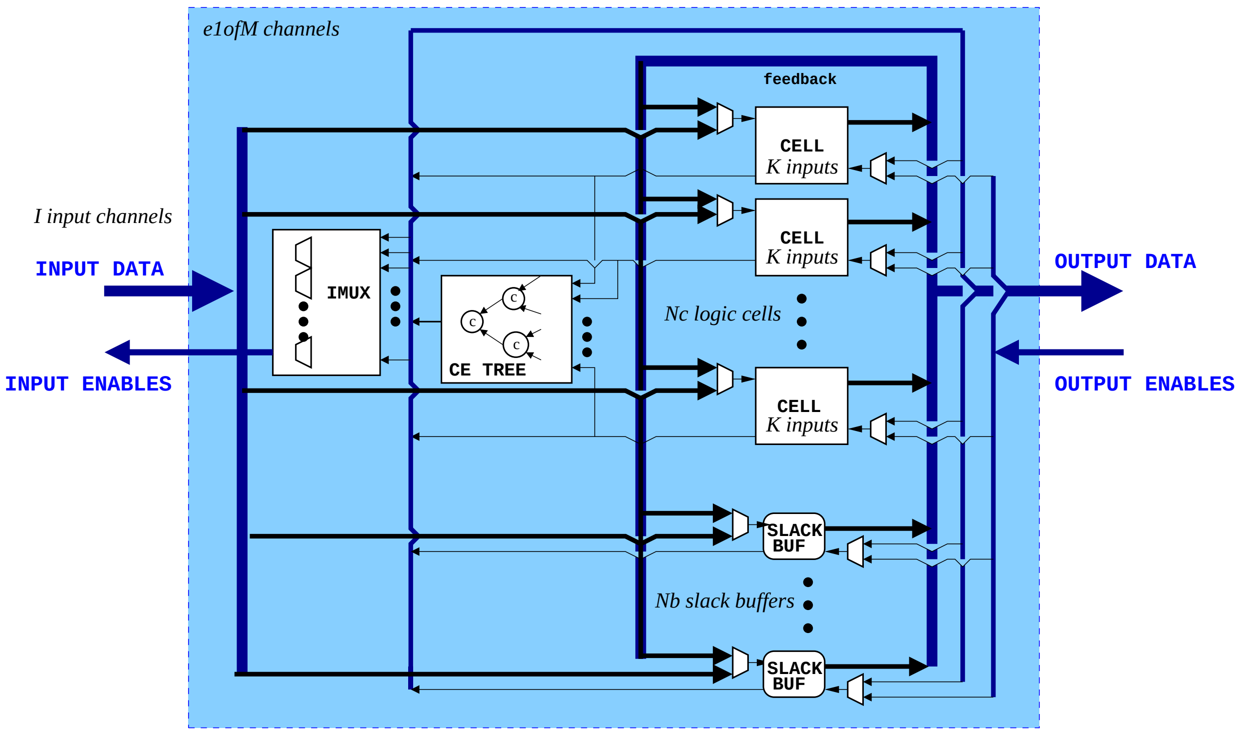


Figure 7.17: Structured cluster design.
The cells and slack buffers are PCHBs, while the C-element tree and input enable multiplexer (IMUX) generate and route enable signals for channels entering the cluster from the interconnect.

of the cells would take advantage of the larger cells. In a less regular control element, that percentage might drop. We are working on combining our synthesis and area estimation tools to study this issue further.

7.5.3 Parameterized Cluster Design

We cluster cells in FPGAs to save area by taking advantage of locality in mapped applications, and to relieve the burden on global physical design tools by reducing the size of the problems they face. We have already presented one possible cluster architecture, but wish to analyze others. We therefore apply structure to our clusters and use this structure to parameterize their design. Parameters include the number of cells, the number of slack buffers, the fanin into the cluster, the fanin into each cell, and the size of each channel.

The structure that we have chosen for our clusters is displayed in Figure 7.17. This cluster is very general: instead of tying slack buffers to logic cells, the numbers of both components present in the cluster are independent, and any cell can be connected to any buffer or even a chain of buffers through the internal cluster network of feedback channels. Meanwhile, instead of only specific two-way or four-way copies being programmable, the cluster allows incoming channels to be copied to any number and combination of logic cells by gathering all of their input enables together in a C-element tree ("CE TREE" in Figure 7.17).

Finally, the programmable input enable multiplexer ("IMUX" in Figure 7.17)) allows the cluster to have a lower fanin than the total fanin of its internal logic cells and slack buffers. For example, even if a cluster has four three-input logic cells and one slack buffer (with a total of 13 internal input ports), the architecture may allow only a fanin of eight channels from interconnect for the cluster. Interconnect comprises 90% of the area of typical FPGAs, so often the utilization levels of logic cells are sacrificed to reduce the amount of interconnect required [14].

Consider a cluster based on e1ofM channels and K-input logic cells. In our parameterized model, let there be N_c such logic cells (each with area $A_c(M,K)$) and N_b e1ofM slack buffers (with area $A_b(M)$). The total fanin of the cluster logic cells and slack buffers is therefore $I_{max} = K \cdot N_c + N_b$. Let $I \leq I_{max}$ be the fanin of the cluster.

Feedback (FB) channels in the cluster have $N_c + N_b$ possible sources (one for each output of a logic cell or slack buffer) and $K \cdot N_c + N_b$ possible sinks (one for each input of a cell or buffer). Each logic cell or slack buffer input can therefore choose from $I + N_c + N_b$ possible input channels. Meanwhile, all logic cell and slack buffer outputs can be copied to two different channels in the interconnect. C-elements that feeds into the output enable multiplexer implement these copies.

We present the parameterized logic cell component of a cluster in Figure 7.18. The component includes both the logic cell itself, and multiplexers that provide a programmable interface with the rest of the cluster. Every logic cell component includes $K \cdot M$ multiplexers for its input channel data rails, each with $I + N_c + N_b$ inputs. Only K SRAM cells are required to program these multiplexers, however, as all data rails must be connected to the same channel. Input enable rails are passed through electrical buffers (as opposed to asynchronous pipeline stage buffers) to increase their drive

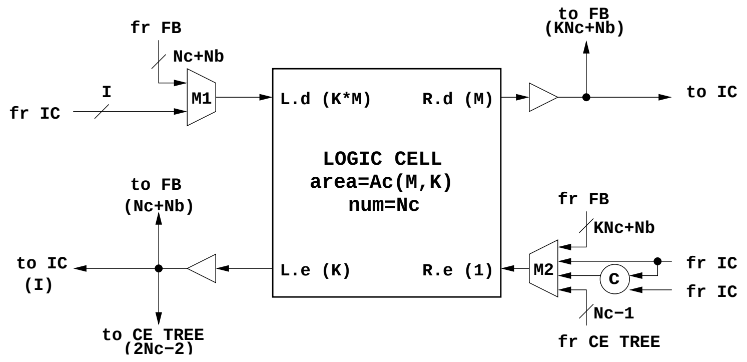


Figure 7.18: Logic cell slice for a cluster based on K-input $e1ofM$ cells.

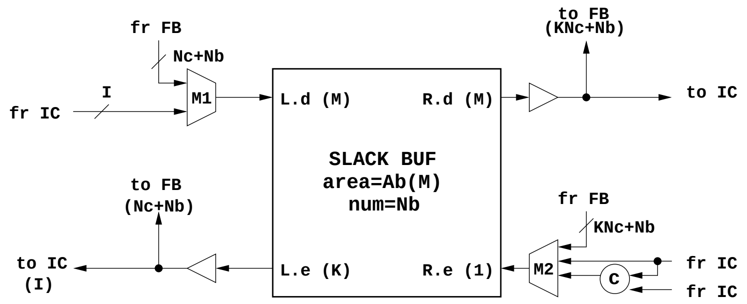


Figure 7.19: Slack buffer slice for a cluster based on $e1ofM$ channels.

strength before they are copied to the IMUX, to the feedback channels, and to the C-element tree. On the output port of the logic cell, the data rails are copied to both the interconnect and the feedback channels. The output enable has multiplexer with inputs from feedback channels, the C-element tree, and two inputs (one generated by a copy C-element) from the interconnect. This multiplexer can be programmed by two SRAM cells.

We can already see that the cost of generality for the cluster is quite high; if it proves to be exorbitantly high, the model can easily be modified to limit the number of feedback channels, for example, and trade off area against flexibility. The parameterized slack buffer, shown in Figure 7.19, is similar to the logic cell. However, it has only one input instead of K, and since channels cannot be copied to slack buffer input ports, has less possible connections for its input data rails and enables.

For the C-element tree, we make the following design decisions at the outset. Any input channel from the interconnect or feedback channel from within the cluster can be copied. Channels can be

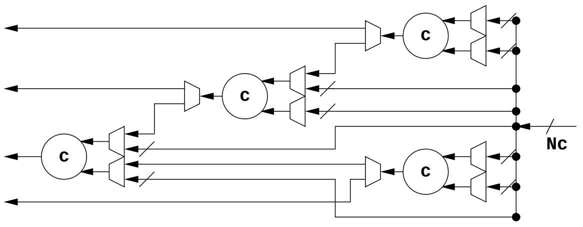


Figure 7.20: C-element tree slice.
C-element tree for a cluster with $N_c = 5$. The tree can be programmed to handle a 5-way copy, a 4-way copy, a 3-way copy, a 3-way and a 2-way copy, and two 2-way copies.

copied only to logic cell input ports, and not to slack buffer input ports. (The slack buffers will be used more often in series than in parallel.) Finally, if each K-input logic cell has input channel ports labeled C_k, where $0 \leq k < K$, then a channel can only be copied to different logic cell input ports that have the same index k. The only reason to distinguish between the different input ports of a logic cell is if the condition of communication for that channel is important. While there are cases where an unconditional input for one cell will be a copy of a conditional input in another cell, they are not common. Therefore, to save area and keep our cluster mapping free of such conditional muddy waters, we insist that all input ports of the copy channels match. (This suits the common case of vertical decomposition.)

Input copies require a programmable tree of C-elements that gather all the copy acknowledgments into a single acknowledgement signal. Since copy channels are restricted to one of the K possible logic cell input ports, we may use K identical trees that each handle N_c inputs. For now, we wish to keep the C-element trees as general as possible—they should be able to handle an N_c-way copy, two $\frac{N_c}{2}$-way copies, etc. We therefore keep the tree as balanced as possible, allow outputs from every C-element in the tree, and make all N_c possible input acknowledgements available to every C-element in the tree. A C-element tree for a cluster where $N_c = 5$ is illustrated in figure 7.20.

Let $demux(x)$, $mux(x)$, and $celem(x)$ return the areas required by demultiplexers, multiplexers, and C-elements with x inputs, respectively. In general, the area of such a C-element tree is

$$A_{CE}(N_c) \quad = \quad (N_c - 1) \cdot celem(2) + (N_c - 2) \cdot demux(2)$$

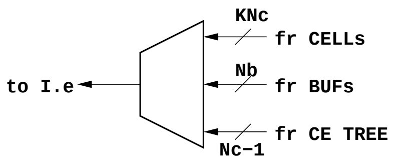


Figure 7.21: Input multiplexer slice for a parameterized cluster.

$$+N_c \cdot mux(N_c) + (N_c - 2) \cdot mux(N_c + 1) \tag{7.1}$$

where the area functions for demultiplexers and multiplexers include the SRAM cells required to program them.

Finally, if I channels enter the cluster from the interconnect and are distributed among the N_c logic cells and N_b slack buffers, then I multiplexers are required to gather all of the possible acknowledgement wires (including possible copies) and choose the appropriate one to send back out to the interconnect. Such a multiplexer has $K \cdot N_c + N_b + N_c - 1$ inputs, and is shown in figure 7.21.

Combining the information for the various cluster components, we can use the minimum-width transistor technique to estimate the area of a cluster given the parameters M, K, N_c, N_b and I. Following standard practise, the buffers that strengthen signals to be sent out to the interconnect (shown in figures 7.18 and 7.19) are not included in the cluster area but are instead credited towards the routing area of an FPGA. Multiplexers are assumed to consist of 6-transistor SRAM cells and minimum-sized n-transistors. Then, the area functions for the logic cell and slack buffer slices are

$$\begin{aligned}
A_{LC}(M,K,N_c,N_b,I) &= A_c(M,K) + M \cdot K \cdot mux(N_c + N_b + I) \\
&\quad + mux(K \cdot N_c + N_b + N_c + 2) + celem(2) && (7.2) \\
A_{SB}(M,K,N_c,N_b,I) &= A_b(M) + M \cdot mux(N_c + N_b + I) \\
&\quad + mux(K \cdot N_c + N_b + N_c - 1 + 2) + celem(2) && (7.3)
\end{aligned}$$

The area function for C-element trees was given in a previous section, and so the total area of the cluster is

$$\begin{aligned} A_{cluster}(M,K,N_c,N_b,I) &= N_c \cdot A_{LC}(M,K,N_c,N_b,I) + N_b \cdot A_{SB}(M,N_c,N_b,I) \\ &\quad + A_{CE}(N_c) + I \cdot mux(K \cdot N_c + N_b + N_c - 1) \end{aligned} \tag{7.4}$$

7.6 Summary

We have presented a clustered architecture for an asynchronous FPGA. The system is QDI, and places no timing constraints on any placement and routing tools that target this FPGA. Examples of how to implement several different circuits on this architecture have been given, including one for a large datapath unit from an asynchronous microprocessor. We have considered different channel encoding options for interconnect, and set up a parameterized model for clusters to explore different architectures. While there are many issues open to further research (which we will discuss in Chapter 8), the cell and cluster designs presented here offer a good general basis for future asynchronous FPGAs.

Chapter 8

Conclusion

We have presented *data-driven decomposition* (DDD), a new method that can transform a sequential algorithm into a distributed network of circuits optimized for both speed and energy. While its early phases can be applied to convert both software and hardware algorithms into concurrent systems, we have also introduced techniques to DDD that make it the first high-level synthesis technique to target the pipeline stages used in high-performance asynchronous VLSI systems. In addition, we have presented a new architecture for asynchronous FPGAs that is quasi delay-insensitive, and whose interconnect is entirely delay insensitive. Together, DDD and reconfigurable asynchronous architectures represent a significant step forward in the automated design and rapid prototyping of asynchronous VLSI systems, allowing more designers to exploit the many advantages of asynchrony.

8.1 Summary of New Synthesis Methods

DDD uses data-dependency analysis to decompose a sequential program into a concurrent system of communicating processes. The method can be applied to any deterministic CHP program and is summarized in Figure 8.1.

The first main phase of DDD transforms the original program into *dynamic single-assignment* (DSA) form, to limit the size of processes in the distributed system and to eliminate unnecessary syntactic constraints, thus exposing concurrency in the algorithm. We have shown how deterministic programs written in CHP can be systematically rewritten in DSA form by constructing a method based on the three control structures in CHP: straightline series, selection statements, and repetition

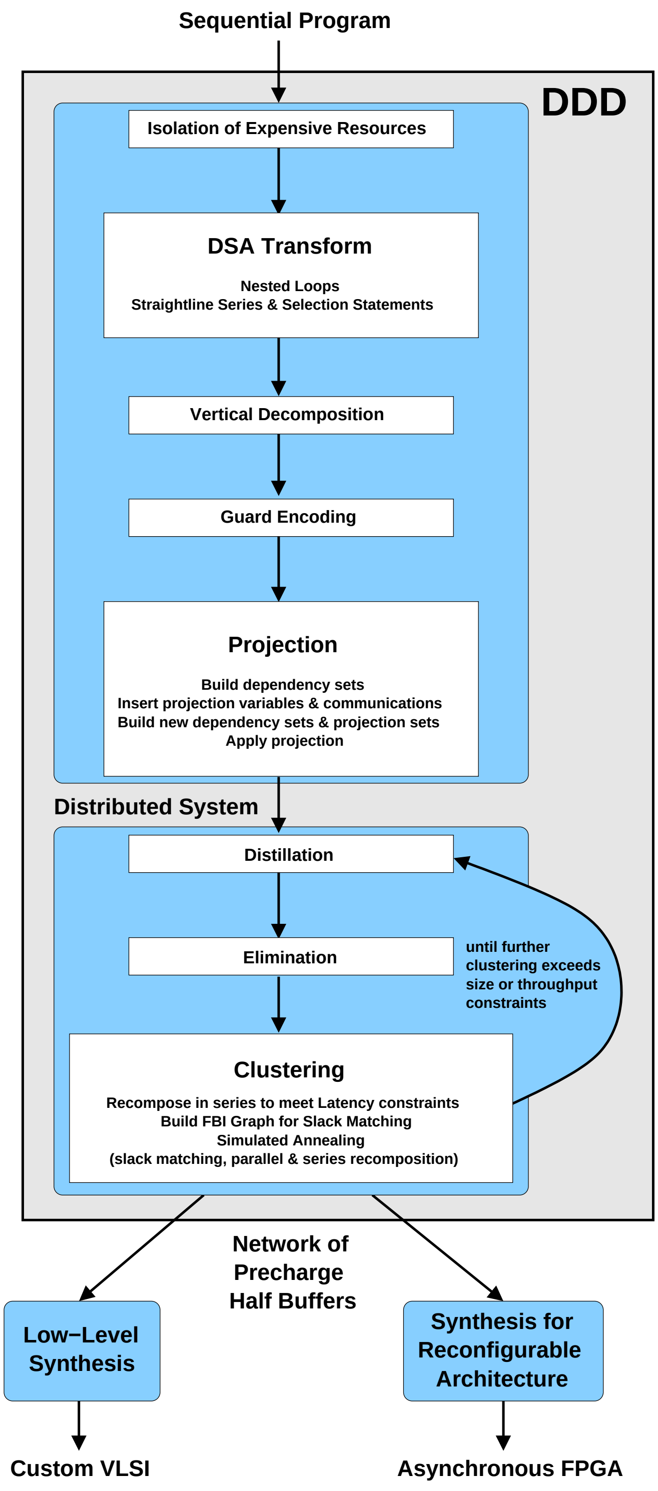

Sequential Program
DDD
Isolation of Expensive Resources
DSA Transform
Nested Loops
Straightline Series & Selection Statements
Vertical Decomposition
Guard Encoding
Projection
Build dependency sets
Insert projection variables & communications
Build new dependency sets & projection sets
Apply projection
Distributed System
Distillation
Elimination
until further clustering exceeds size or throughput constraints
Clustering
Recompose in series to meet Latency constraints
Build FBI Graph for Slack Matching
Simulated Annealing
(slack matching, parallel & series recomposition)
Network of Precharge Half Buffers
Low−Level Synthesis
Synthesis for Reconfigurable Architecture
Custom VLSI
Asynchronous FPGA

Figure 8.1: DDD for the high-level synthesis of asynchronous VLSI systems.

statements.

After the DSA transformation has been applied along with other optimizations such as the *isolation* of expensive resources (area optimization), *vertical decomposition* (performance optimization), and *guard encoding* (energy optimization), DDD splits the single sequential description into an equivalent concurrent system of communicating processes. This is accomplished using the syntactic transformation of *projection* (until now primarily used for verification purposes). DDD provides a semantic framework in which data-dependency analysis can be performed, allowing projection to be systematically applied to decompose the sequential program.

We have presented parameterizable circuit templates for *precharged half-buffers* (PCHBs), the family of asynchronous pipeline stages used in most high-performance asynchronous VLSI systems, and we have shown that all processes in the decomposed system generated by DDD can be implemented directly as PCHBs. DDD now uses these templates to perform energy and throughput optimizations on the distributed system generated by projection. First, DDD removes redundant control circuitry through *distillation*, which individually transforms certain processes with conditional communications so that the entire processes are idle more often, consuming no dynamic energy. Then, DDD performs *elimination*, removing L-R buffers from the system.

At this point, each individual process implements the computations for a single variable and is simple enough for lower-level synthesis to be feasible. However, this simplicity is often accompanied by a communications overhead in energy consumption that is sizable relative to the energy consumed by the actual computations being performed. We have therefore presented DDD's *clustering* phase where, to reduce system energy consumption, DDD recomposes select processes to eliminate communications and their related wires and circuitry from the system.

Clustering performs the *slack matching* performance-optimization in conjunction with *recomposition*. This transformation inserts additional pipelining into the system to alter handshake dynamics and increase system throughput. We have introduced the FBI model to analyze the handshake dynamics in a system and shown that in certain systems, the set of possible critical cycles can be limited to four basic cycles. We have demonstrated how recomposition and slack matching can be combined using a simulated-annealing heuristic, reducing system energy costs by as much as 70% without decreasing system performance.

8.2 Future Work

While the first two phases of DDD are largely independent of whether the final system is implemented in software or hardware, the clustering phase of DDD is tailored specifically for high-performance asynchronous VLSI systems. In the future, DDD can be reformulated for synchronous VLSI systems as well, by changing methods of low-level performance estimation and replacing the slack-matching transformation with retiming. Also, currently, DDD can be applied only to deterministic sequential programs. Although the majority of programs used in asynchronous VLSI design are deterministic, future research incorporating non-determinism and *arbitration* will make the DDD method complete.

The focus of DDD as presented in this thesis is, through its performance estimations and optimizations, often on the communications within an asynchronous system. Many future improvements to the method therefore involve the computations within an asynchronous system. Currently, DDD can isolate computations that the designer deems costly. Also, DDD can automatically perform vertical decomposition on processes that either do not involve computation, or include computations that the designer indicates can be "evenly sliced." The bitwise AND is an example of a computation that can be evenly sliced, since each vertically decomposed computation is identical and can operate independently.

Future extensions to DDD may involve function decomposition, allowing the method to automatically distribute a single computation across multiple variables (and, eventually, processes) to improve the energy or performance of the final system. Function decomposition is also necessary for vertical decomposition to be automatically applied in cases where computations cannot be evenly sliced. Although most of the energy consumed in asynchronous systems is in communications, automatic function synthesis would allow DDD to make more accurate energy estimations for PCHBs by including computation pulldown networks.

In the clustering phase of DDD, the creation of a more efficient method for slack matching systems, particularly heterogeneous ones, will be a fertile area for further research. Future work may modify its approach and cost functions to optimize other metrics. One possible such metric is the voltage-independent metric Et^2, which grows more attractive as power concerns lead to more solutions involving dynamic voltage scaling. Another interesting addition to DDD is the inclusion of layout considerations into the recomposition heuristic. While vertical decomposition often allows slices of layout to be generated and then duplicated, clustering two such slices may result in a reduction in energy but increase the number of individual processes for which physical layout must be designed. In other cases, clustering can reduce the regularity of the system, also making the physical design of a system more difficult. Also, DDD clustering currently seeks to minimize energy while maintaining a specified system throughput.

Of course, physical design is not necessary when using asynchronous FPGAs. As has already been discussed, there is much work yet to be done in using the parameterized reconfigurable constructs presented here to explore different FPGA architectures with different criteria in mind. Specifically, the relationships between cluster area, interconnect area, and logic utilization should be determined. It is likely that irregular microprocessor designs will be best implemented on one type of FPGA architecture, while DSP designs will be best implemented on another type.

The DDD tool can be combined with area- and performance-estimation tools for FPGAs to examine all possibilities, including more adventurous interconnect schemes that exploit delay-insensitivity and the flexibility of asynchronous design. For the DDD tool to be used in such experiments, we must add constraints to the current DDD method so that decomposition phases target the CHP used by the basic reconfigurable cell, and the clustering phase targets the specific reconfigurable clusters of the FPGA. Finally, since the delay-insensitivity of our reconfigurable interconnect reduces pressure on mapping tools, the architecture presented here is attractive for FPGAs that can be dynamically reprogrammed. The use of asynchronous VLSI technology in evolutionary hardware is an exciting path that has not yet been explored.

Appendix A

CHP Notation

CHP is a high-level hardware description language that includes communications primitives and concurrent processes. It is based on Hoare's CSP (communicating sequential processes) language for parallel programming [23]. In our overall synthesis method, we begin by describing the behavior of circuits and systems using CHP and indeed, the initial process for a simple microprocessor can fit into a single page of code. CHP is a simple language though, and none of its constructs are explicitly tailored for hardware implementation.

This appendix is intended as a brief and informal introduction to the basic CHP features and structures used in this dissertation. A formal description of the CHP language can be found in [38].

A.1 Basic Constructs

CHP *variables* can be integers, enumerations, or arrays. Numerous other variable types can be defined as well, but in the interests of clarity, this dissertation limits variables to those three basic types.

A *process* is a single imperative program that manipulates variables. While communications primitives can be used to communicate with other processes, they are not necessary to transfer information within a single process.

A *system* consists of a group of processes, each running concurrently and sharing information through communications across *channels*. Shared variables are not allowed between processes in a system. If two processes share information, it must be communicated explicitly between them.

Without these synchronizing communications, different processes in a system can run independently at their own pace.

Communications channels are dedicated between two processes. Channel types exist that correspond to every type and size of variable. They have only two ports and are, in most cases, unidirectional (i.e., the process attached to one port is always the sender of information and the process attached to the other port is always the receiver of information.) The exception to unidirectionality is when the channel is used purely for the synchronization of processes, and no data is communicated. In this case, at the CHP level, the channel is considered to have no "direction" at all.

A.2 Basic Statements

The simplest statement in CHP is "**skip**," which does nothing.

Aside from that "no-op," the most basic statements in CHP are *regular assignments*, where the value of some expression is assigned to a variable. Examples of regular assignments are "$x := 0$" and "$x := a + b$," for variables x, a, and b. The variables being written are considered to be on the "left-hand side" (LHS) of an assignment; the variables that appear in the expression being evaluated are considered to be on the "right-hand side" (RHS) of the assignment. (If the assignment statement is labeled "A," then the abbreviations "$A.LHS$" and "$A.RHS$" may also be used to refer to the set of variables on the LHS or RHS of A.) Expressions can be boolean, arithmetic, or otherwise defined by users. Short forms often used for assignments to boolean variables are "$a\uparrow$" (equivalent to "$a :=$ **true**") and "$a\downarrow$" (equivalent to "$a :=$ **false**").

Communications primitives can also be used in assignment statements. For example, the send statement "$C!(a + b)$" evaluates an expression containing variables a and b and assigns the result to (sends it out on) the output channel C. Thus, C is on the LHS of the assignment while a and b are on the RHS. Similarly, the receive statement "$D?x$" reads a value off of input channel D and assigns it to the variable x. In this case, x is on the LHS of the assignment and channel D is on the RHS.

A.3 Composition of Statements

Statements can be composed in sequence and in parallel. The sequential operator is the semicolon. The CHP "$A; B$" means "execute first A and then B in sequence."

There are two parallel operators: the comma and the parallel bars. For example, "A, B" means "execute A and B in any order." The comma is a stronger compositional operator than the semicolon. Meanwhile, the code "$A \parallel B$" also means execute the two statements in parallel (i.e., any order), but the parallel bars are a weaker compositional operator than the semicolon.

Any fragment of CHP code that consists of only basic statements and sequential and parallel compositional operators is called *straightline code*. Its name is derived from the fact that there are no branches of control—each of the basic statements will be executed right after each other, in some partial order.

A.4 Control Statements

More complex than the basic statements introduced earlier, *selection statements* and *repetition statements* (also called loops) are control structures in CHP. Both make use of *guarded commands*, where each guarded command consists of a boolean condition (the guard) and a sequence of statements (the command). In such a structure, whenever one of the boolean conditions evaluates to true, the command associated with that guard is executed. If multiple guards can evaluate to true at the same time, one is chosen randomly and the structure is nondeterministic. If the guards are mutually exclusive, then the structure is deterministic. This thesis deals only with deterministic guarded commands.

Selection statements contain multiple guarded commands. When a CHP program enters this structure, it stalls until at one of the guards becomes true. At that point the true guard's corresponding command is executed and then the program exits the selection statement. A syntactic example for a deterministic selection statement is

$$[\ g_1 \longrightarrow\ S_1\ \talloblong\ g_2 \longrightarrow\ S_2\ \talloblong\ g_3 \longrightarrow\ S_3\ ]$$

In this structure, at most one of the boolean expressions g_1, g_2 and g_3 can be true at any given time. When guard gi evaluates to true, then its command S_i is executed. If $g_3 \equiv$"**else**," then one of the three guards always evaluates to true and the program never stalls at the selection statement.

The repetition statement, or loop, can also contain guarded commands. A program only enters the loop if one of the guards is true. Otherwise, it skips the loop entirely and moves on to the next statement. Once in the loop, the program executes the command corresponding to the true guard and when finished, evaluates all of the guards again. The program continues within the loop until none of the guards are true, at which point it exits the repetition statement. The syntax of a loop is illustrated by

$$\texttt{*[}\ g_1 \longrightarrow \ S_1 \ \talloblong\ g_2 \longrightarrow \ S_2 \ \talloblong\ g_3 \longrightarrow \ S_3 \ \texttt{]}$$

There is an implicit semicolon at the end of every guarded command in the loop. This sequences the end of one iteration with the beginning of the next.

A very simple loop is the non-terminating loop "$\texttt{*[}\textbf{true} \rightarrow S\texttt{]}$," usually shortened to "$\texttt{*[}S\texttt{]}$." Such a loop means "repeat S forever," and is often used an outer loop for programs describing circuit behavior that repeats indefinitely. In such situations, when we refer to "one iteration" of the circuit's program, we mean one iteration of this main unconditional loop.

Selection and repetition statements can be nested within each other, with no limit. When we refer to a *sequence of statements* (for example, the command within guarded commands), we mean any combination of basic, selection, and repetition statements arranged using both sequential and parallel operators.

A.5 Applicability of DDD

This section has introduced all of the constructs of the CHP language that are used in this thesis, and indeed the majority of the CHP language itself. The DDD method described in this work can be applied to any deterministic CHP program. Any combination of the structures described here fits that description.

Bibliography

[1] A.V. Aho, R. Sethi and J.D. Ullman. “Compilers: Principles, Techniques, and Tools.” Addison-Wesley, 1986.

[2] S.J. Allan and A.E. Oldehoeft. A Flow Analysis Procedure for the Translation of High-Level Languages to a Data Flow Language. *IEEE Transactions on Computers*, vol. C-29, No. 9, September 1980.

[3] R.A. Ballance, A.B. Maccabe and K.J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. *Proc. ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,* June 1990.

[4] A. Bardsley and D.A. Edwards. The Balsa Asynchronous Circuit Synthesis System. *Forum on Design Languages*, 2000.

[5] C.H. van Berkel and R.W.J.J. Saeijs. Compilation of Communicating Processes Into Delay-Insensitive Circuits. *Proc. International Conference on Computer Design,* 1988.

[6] V. Betz, J. Rose and A. Marquardt. Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, 1999.

[7] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. Synthesis of Embedded Software from Synchronous Dataflow Specifications. *Journal of VLSI Signal Processing* 21, 1999.

[8] M.M. Brandis and H. Mossenboock. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 6 November 1994.

[9] S.M. Burns and A.J. Martin. Synthesis of Self-Timed Circuits by Program Transformation. In G.J. Milne, ed., *The Fusion of Hardware Design and Verification,* North-Holland, 1988.

[10] J-F. Collard. Reasoning About Program Transformations: Imperative Programming and Flow of Data. Springer-Verlag, 2003.

[11] J. Cortadella, M. Kishinevsky et al. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, Vol. E80-D, No. 3, March 1997.

[12] R. Cytron, J. Ferrante, et al. Efficiently Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, October 1991.

[13] R.K. Cytron and J. Ferrante. Efficiently Computing ϕ-Nodes On-The-Fly. *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 3, May 1995.

[14] A. DeHon. Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100*Proc. International Symposium on Field Programmable Gate Arrays (FPGA'99)*, February 1999.

[15] A. DeHon. Rent's Rule Based Switching Requirements, *Proc. System-Level Interconnect Prediction (SLIP'01)*, 2001.

[16] J.B. Dennis. Data Flow Supercomputers. *IEEE Computer Magazine*, November 1980.

[17] P. Duncan, S. Swamy and R. Jain. Low-Power DSP Circuit Design Using Retimed Maximally Parallel Architectures. *Proc. 1st Symposium on Integrated Systems*, March 1993.

[18] R.A. Finkel. "Advanced Programming Language Design." Addison-Wesley, 1996.

[19] R.M. Fuhrer, S.M.!Nowick et al. MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines. Columbia University CS Tech Report CUCS-020-99, 1999.

[20] H. van Gageldonk, K. van Berkel, and A. Peeters. An Asynchronous Low-Power 80C51 Microcontroller. *Proc. 4th Intl Symposium on Advanced Research in Asynchronous Circuits and Systems,* April 1998.

[21] S. Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, **83**(1), 1995.

[22] S. Hauck, S. Burns, G. Borriello and C. Ebeling. A FPGA for Implementing Asynchronous Circuits. *IEEE Design and Test of Computers*, 11(3), 1994.

[23] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, Vol. 21 No. 8, August 1978.

[24] S. Kim and P. Beerel. Pipeline Optimization for Asynchronous Circuits: Complexity Analysis and an Efficient Optimal Algorithm. *Proc. International Conference on Computer-Aided Design*, 2000.

[25] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Schiozawa, M. Inamori and K. Nagami. PCA-1: A Fully Asynchronous, Self-Reconfigurable LSI, *Proc. Seventh Intl Symposium on Asynchronous Circuits and Systems*, March 2001.

[26] S. Kirkpatrik, C.D. Gellatt, Jr. and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.

[27] C.E. Leiserson and J.B. Saxe. Retiming Synchronous Circuitry. *Algorithmica, 6:5-35*, 1991.

[28] M. Ligthart, K. Fant, et al. Asynchronous Design Using Commercial HDL Synthesis Tools. *Proc. Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000.

[29] A.M. Lines. Pipelined Asynchronous Circuits, M.S. Thesis, Dept of Computer Science, California Institute of Technology, 1995.

[30] K.N. Lalgudi and M.C. Papaefthymiou. Fixed-Phase Retiming for Low Power Design. *Proc. International Symposium on Low Power Electronics and Design*, 1996.

[31] J.R. McGraw. The VAL Language: Description and Analysis. ACM TOPLAS, January 1982.

[32] J.R. McGraw, S. Skedzielewski et al. SISAL: Streams and Iteration in a Single-Assignment Language. Lawrence Livermore National Laboratories, Report M-146, 1983.

[33] K. Maheswaran. Implementing Self-Timed Circuits in Field Programmable Gate Arrays. Master's thesis, U.C.Davis, 1995. September 1996.

[34] R. Manohar, T.K. Lee and A.J. Martin. Projection: A Synthesis Technique for Concurrent Systems. *Proc. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1999.

[35] R. Manohar and A.J. Martin. Slack Elasticity in Concurrent Computing. *Proc. 4th International Conference on the Mathematics of Program Construction*, in J. Jeuring ed., Lecture Notes in Computer Science 1422 Springer Verlag, 1998.

[36] A.J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI, ed. W.J. Dally,* MIT Press, 1990.

[37] A.J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C.A.R. Hoare, ed: *Developments in Concurrency and Communication, UT Year of Programming Series.* Addison-Wesley, 1990.

[38] A.J. Martin. Synthesis of Asynchronous VLSI Circuits. Caltech Computer Science Technical Report CS-TR-93-28, 1993.

[39] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings and T.K. Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. *Proc. 17th Conference on Advanced Research in VLSI*, 1997.

[40] A.J. Martin, M. Nyström and P. Penzes. Et^2: A Metric For Time and Energy Efficiency of Computation. In R. Melhem and R. Graybill ed., “Power-Aware Computing,” Kluwer Academic Publishers, 2001.

[41] A.J. Martin, M. Nyström, K. Papadantonakis, P.I. Penzes, P. Prakash, C.G. Wong, J. Chang, K.S. Ko, B. Lee, E. Ou, J. Pugh, E. Talvala, J.T. Tong, and A Tura. The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller, *Proc. 9th IEEE Intl Symposium on Advanced Research in Asynchronous Circuits and Systems*, May 2003.

[42] A.J. Martin, M. Nyström and C.G. Wong. Three Generations of Asynchronous Microprocessors. *IEEE Design & Test of Computers, Special Issue on Clockless VLSI*, November-December 2003.

[43] J. Monteiro, S. Devadas and A. Ghosh. Retiming Sequential Circuits for Low Power. *Proc. International Conference on Computer-Aided Design (ICCAD)*, 1993.

[44] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, **77**(4), 1989.

[45] R.E. Payne. Asynchronous FPGA Architectures. *IEE Proceedings on Computers and Digital Techniques*, Special Issue on Asynchronous Processors, September 1996.

[46] K. Pingali, M. Beck, et al. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. *Proc. 18th annual ACM Symposium on Principles of Programming Languages*, 1991.

[47] K. Rath and S.D. Johnson. Toward a Basis for Protocol Specification and Process Decomposition. *Proc. 1993 IFIP Conference on Hardware Description Languages and their Applications,* April 1993.

[48] A. Rettberg and B. Kleinjohann. A Fast Asynchronous Reconfigurable Architecture for Multimedia Applications, *Proc. 14th Symposium on Integrated Circuits and Systems Design*, September 2001.

[49] A. Rogers and K. Pingali. Process Decomposition Through Locality of Reference. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[50] R. Rubin and A. DeHon. Design of FPGA Interconnect for Multilevel Metalization, *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA2003)*, February 2003.

[51] J. Teifel and R. Manohar. Programmable Asynchronous Pipeline Arrays. *Proc. 13th Intl Conference on Field Programmable Logic and Applications*, September 2003.

[52] T. Williams. Latency and Throughput Tradeoffs in Self-Timed Asynchronous Pipelines and Rings, *Stanford Technical Report* CSL-TR-90-431, May 1990.

[53] C.G. Wong. Performance Estimations for Asynchronous Pipeline Stages. In preparation.

[54] C.G. Wong. Slack-Matching Systems. In preparation.

[55] C.G. Wong and A.J. Martin. High-Level Synthesis of Asynchronous Systems by Data-Driven Decomposition. *Proc. 40th Design Automation Conference (DAC)*, June 2003.

[56] C.G. Wong, A.J. Martin, and P. Thomas. An Architecture for Asynchronous FPGAs. *Proc. IEEE International Conference on Field-Programmable Technology (FPT'03)*, 2003.