EVERYTHING I NEEDED TO KNOW ABOUT ASYNCHRONOUS REGISTER FILES, I LEARNED IN KINDERGARTEN

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Jedi Master

by

David Fang

January 2004

ABSTRACT

At the heart of practically every modern microprocessor core sits some form of register file, whose purpose is to hold and supply intermediate results of computations to other computation units. As register files grow in size and in the number of ports to support increasing instruction-level parallelism (ILP), it becomes extremely difficult to meet timing requirements in clocked designs, and the energy consumed by accesses increases significantly. Asynchronous microprocessors share many of the same design issues, however, we have at our disposal a different family of techniques due to the robust and modular nature of self-timed design.

Starting with a sequential specification of a typical asynchronous register file, we decompose the specification into fine-grain parallel processes for the core, bypass and control that implement the specified register file. To improve the throughput of the core, we vertically pipeline the read and write ports into smaller blocks of data, and we describe the locking mechanism that maintains pipelined mutual exclusion among reads and writes. Using standard handshaking expansion templates, we synthesize quasi-delay insensitive production rules that describe the circuits for the pipelined core ports. This initial design serves as the basis for comparison for the transformations presented in the remainder of the thesis.

The key contributions are described in detail throughout the remainder of the thesis. We extend the base design to support a width-adaptive datapath representation, which leads to significant energy reduction by conditionally communicating higher significant bits of integers, with little performance degradation. We show how the bypass can be extended to reduce core accesses with alternative implementations of the hard-wired zero register and bypass-forwarding of duplicate operands using Port Priority Selection. We show the improvement in speed and energy gained by splitting the register core into two banks. As an alternative to banking, which is interconnect-limited, we present the technique of nesting the register core into non-uniform banks without increasing the interconnect requirement to facilitate faster accesses to more frequently used registers and slower accesses to less frequently used registers, and thus, achieve average-case improvement. We have laid out the explored design space of register files in TSMC .18μm technology, and present performance and energy results for all register cores simulated using a variant of `spice`.

Biographical Sketch

`[scrolling text, with theme music]`

A seemingly long time ago in a part of the country far away...

The author graduated from Franklin Regional Senior High School in Murrysville, PA with Honors with Highest Distinction in the class of 1997. He enrolled at the California Institute of Technology in 1997, ambitiously intending to tackle electrical engineering, physics, and a twist of applied mathematics, but only managed to graduate with a Bachelor of Science in Electrical Engineering with Honors in 2001.

The roots of his interest in asynchronous VLSI trace back to the EE/CS181abc class he took (and endured) as an undergrad, taught by Prof. Alain Martin and his research group members. The author received a National Defense Science and Engineering Graduate Fellowship, sponsored by the Office of Naval Research. Since the summer of 2001, the author has been a student of the Computer Systems Laboratory in the Cornell Electrical and Computer Engineering Department, as an apprentice of Jedi Prof. Rajit Manohar, a former student of Prof. Martin. The change of climate has been likened to leaping out of the frying pan and into the freezer.

To supplement his background in asynchronous VLSI, the author is minoring in computer science, and maintains interest in computer architecture and compilers, which spans the hardware and software aspects of computer engineering. Aside from being passionate about his work, he also maintains strong interest in music and dance — but only when time permits, of course.

Presently, he lives his life asynchronously, not conforming to any normal or regulated sleeping schedule. He aspires to settle down in a place that's sunnier and warmer than Ithaca, NY someday.

dedicated to Master Yoda

Acknowledgments

First, I'd like to thank my advisor, Prof. Rajit Manohar, for his continuous guidance, support, and his patience throughout the lifetime of this undertaking. Thanks to my committee members, Profs. Martin Burtscher and Radu Rugina, for taking the time to parse the entire thesis through their pipelines and provide constructive feedback.

The experience of the Master's project would not have been as enjoyable without constant encouragement and a sense of humor from my fellow computer engineers in the Computer Systems Laboratory. It is my pleasure and honor to be in the company of "Asynchronians" John Teifel, Virantha Ekanayake, Clint Kelly, and David Biermann.

A nod goes to an innumerable group of friends who, with some success from time to time, have lured me away from my arduous studies to enjoy life as it should be enjoyed. Without them, I would've lost my wits long ago. (No comical retorts, please!)

Finally, the work for thesis was entirely funded by the National Defense Science and Engineering Graduate Fellowship with sponsorship from the Office of Naval Research. You have fed this poor hungry grad student in support of his work. Thank you!

Table of Contents

List of Tables

List of Figures

List of Programs

List of Abbreviations

CHP	Concurrent Hardware Processes
CMOS	complementary metal-oxide semiconductor
CRT	constant response time
DRAM	dynamic random access memory
HSE	handshaking expansion
ISA	instruction set architecture
ILP	instruction-level parallelism
NFET	n-diffusion field effect transistor
PFET	p-diffusion field effect transistor
PCEVFB	precharge enable-valid full-buffer (Section 4.1)
PCEVHB	precharge enable-valid half-buffer (Section 4.1)
PCFB	precharge full-buffer (Section 1.1)
PCHB	precharge half-buffer (Section 1.1)
PRS	production rule set
QDI	quasi-delay insensitive
SCMOS	scalable CMOS
SDI	scalable-delay insensitive
SRAM	static random access memory
TSMC	Taiwan Semiconductor Manufacturing Company
VLSI	very large scale integration
WAD	width-adaptive datapath (Chapter 5)

Preface

This preface has been written in the style of Frequently Asked Questions (FAQ).

"Should I read this thesis?" Absolutely. Asynchronous register files were intended for children and adults of all ages. This thesis has something to offer for everyone. To the layperson, the thesis works through the systematic design of a complex subsystem with a divide-and-conquer approach. To a theoretician, this thesis illustrates a direct application of mathematical transformations for synthesizing a complex system whose correctness can be formally proven. To an engineer or architect, this thesis surveys a large design space with the goal of designing (perhaps jointly) for high performance and low energy. To a circuit designer, this thesis demonstrates the modularity with which one can design a moderately complex system of robust, self-timed circuits. Good asynchronous microprocessor design requires approaches that are different than those of traditional synchronous designs. For anyone who is familiar with asynchronous circuit design, this thesis gives a tremendous amount of detail of how new and old optimizations can be applied to existing designs. A reasonable goal for anyone's first reading is to understand the fundamental ideas without getting lost in the forest of details.

"What do I need to know?" Among the plethora of program code sequences and circuit diagrams and long-winded passages of text, one will find figures that contain references to other figures or programs. Their hierarchical placement is no accident. They have been placed to guide the reader through the bog of detail. For a general understanding, one only needs to follow the figures that outline the systematic division of each task into subproblems. As writing the thesis was a non-linear task, reading through the thesis may be aided by back-pedaling along the paved road. To venture slightly off the path will require one to become familiar with the CHP program syntax described in Appendix A. The same syntax is also used to describe the individual communication protocol actions, known as handshaking expansions (HSE). Since we present only digital circuits, being able to count to 1 should (in theory) be enough to understand all of circuits in the thesis. In only a few passages will we mention analog circuit concepts.

"How is this thesis organized?" Chapter 1 provides background for register files and asynchronous circuit design. Chapter 2 divides the task of designing an asynchronous register file into smaller concurrent processes. Once the these fine-grain processes are 'simple' enough, i.e., there exist straightforward template-based implementations, we divide large work up into pipelined blocks to improve the throughput in Chapter 3. In Chapter 4, we design circuits for the pipeline block, which we use as the baseline for comparison of optimizations. Nothing in the first four chapters is new because a similar design already exists in the Caltech MiniMIPS. Chapter 5 presents a variable-width implementation that aims to reduce energy by storing and communicating compressed integers instead of full-width integers. Chapters 6 and 7 present alternative implementations that reduce accesses to the core to further reduce energy. Chapters 4.4 through 9 describe transformations that make the core operate faster and with even less energy. Chapters 5 and 9 are the most significant contributions of this thesis. Many of the details in deriving circuits in the style of Chapter 4 have been placed in the Appendices for the latter chapters, because the details are analogous, and need not clutter the text further. Even more detail has been pushed into the technical report [11], the 'companion' to the thesis.

"When I press my finger on the references, why don't the pages automatically turn?" Blame technology, no printer I am aware of supports hyper-linked printouts.

"Do I get free food out of this?" Only if you come to the defense.

David Fang
`fang@csl.cornell.edu`
September, 2003

Chapter 1

Introduction

The core of typical modern microprocessors is equipped with a register file, whose purpose is to provide extremely fast-access storage for a relatively small amount of data. Register files commonly sit at the smallest and fastest end of the memory hierarchy, followed by possibly one or multiple levels of cache, then main memory (often DRAM), and finally disk. The register file is partially exposed as part of the instruction set architecture (ISA) to the compiler, whose job is to allocate available registers and schedule instructions as efficiently as possible for code performance. Out-of-order superscalarprocessors are capable of dynamically renaming registers, and can exploit greater instruction-level parallelism (ILP) by mapping logical registers to a greater number of physical registers to support more in-flight instructions.

While today's mainstream microprocessors are designed synchronously around a fixed clock frequency, self-timed or asynchronous microprocessor have demonstrated competitiveness in performance and energy efficiency. In both synchronous and asynchronous designs, the register file has been cited as a throughput bottleneck and a significant consumer of energy.

It is in the interest of the microprocessor community to investigate techniques for accelerating register file accesses and reducing their energy consumption. In this thesis, we present the systematic design of register files for asynchronous microprocessors and traverse the design space of optimizations in search of faster and lower energy designs. The most important contributions of this thesis are 1) a width-adaptive implementation of a vertically pipelined register core that saves considerable energy by conditionally communicating and storing higher significant bits of data, and 2) the introduction of non-uniform access register organizations that do not increase the interconnect requirement nor do they complicate the controlling environment.

1.1 Background

1.1.1 Asynchronous Circuit Synthesis

As integrated circuit technology continues to improve at an exponential rate, it becomes more and more difficult to design and verify large-scale synchronous circuit designs. As wire delays become more significant with the shrinking of feature sizes, timing model parameters must be corrected for each new silicon process. Asynchronous design methodologies have been proposed as a solution to the increasing difficulty of designing and verifying large-scale synchronous circuits. The self-timed nature of asynchronous circuits makes designing large systems very modular, robust and portable between process technologies. With local communication handshakes replacing the global clock, functional units are no longer constrained by external timing, and may potentially speed up.

Power consumption is gaining attention as more modern applications demand chips that require a minimal amount of energy to operate. Not only does global clock distribution become more difficult due to clock skew, but it also contributes as much as 40% of the total core power [6, 15]. Techniques for reducing power consumption in synchronous designs include clock gating, which shuts off the clock to idle components, and low-swing operation, which reduces the relative change in voltage on selected nodes [22]. An often cited advantage of asynchronous designs is that energy is only consumed when work is done, such as computation or data movement, and hence requires no global clock distribution. Idle asynchronous circuits require no phase-locked loop oscillator to keep a global clock continually running.

Timing models. There exist many timing models for asynchronous circuits, including delay-insensitive (DI), speed-insensitive (SI), quasi-delay insensitive (QDI), scalable-delay-insensitive (SDI), and bounded-delay [46]. Of particular interest is the QDI model, which only assumes unbounded gate delays and isochronic forks on wires, and is the most conservative delay model for which one can design useful asynchronous computing circuits [29, 30]. One primary advantage of QDI is that no timing verification or analysis is required to confirm the correctness of a circuit. QDI is sometimes criticized as being too conservative and requiring more circuit overhead to guarantee delay insensitivity, in contrast to non-QDI asynchronous designs, which have the additional difficulty of having to verify timing assumptions about the speed of circuits.

Voltage scaling. One of the benefits of asynchronous design is the natural property of continuous voltage scaling, which allows one to tradeoff performance for power reduction by lowering the voltage. Synchronous designs must take a two-step approach to voltage scaling: reducing clock frequency before lowering voltage, or raising the voltage before increasing the frequency. However, delay-insensitivity allows voltage scaling to occur *while the circuits are operating*, simply by turning the supply voltage knob, without threatening the reliability of the system — practically zero performance overhead in changing the level of operation. Analytical methods for designing asynchronous pipelines for energy efficiency (as opposed to only performance) based on pipeline dynamics have been proposed [49].

Synthesis. Without the need for timing verification, synthesizing QDI circuits is a relatively straightforward procedure [29]. Figure 1.1 illustrates the design flow of a QDI system. One begins with a sequential functional specification of an entire system, such as a microprocessor. We use the traditional *Communicating Hardware Processes* or CHP, a variant of Hoare's CSP language, to specify the behavior of concurrent communicating processes which compose a system [17]. A summary of CHP notation can be found in Appendix A. The first several phases are a series of semantic-preserving transformations and decompositions of CHP programs into fine-grain processes. Compositions of these processes behave like parallel programs with only point-to-point communications. The system specification is decomposed into individual functional units, which exposes the underlying architecture and functional support of the system.

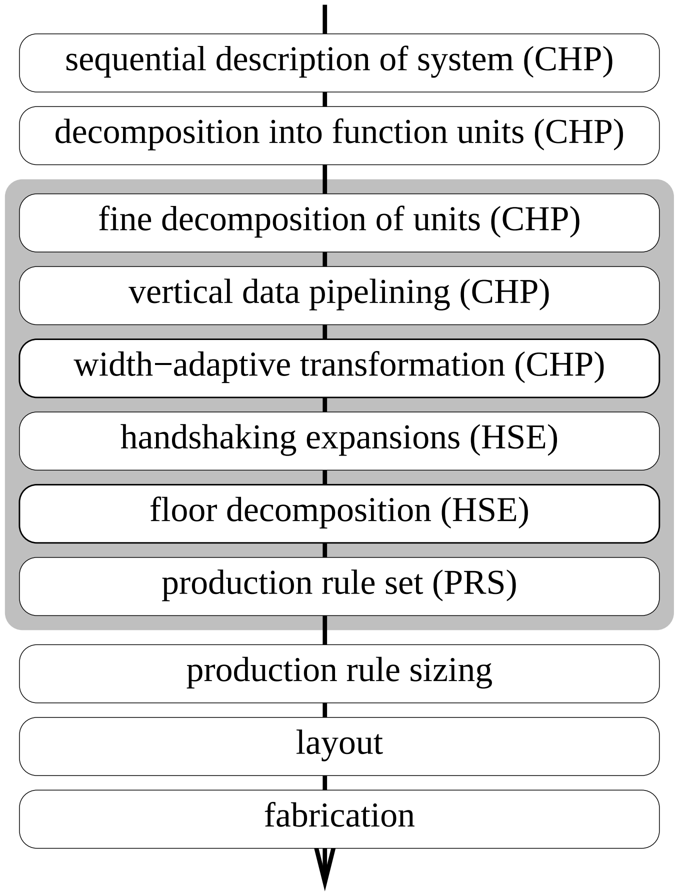

sequential description of system (CHP)
decomposition into function units (CHP)
fine decomposition of units (CHP)
vertical data pipelining (CHP)
width−adaptive transformation (CHP)
handshaking expansions (HSE)
floor decomposition (HSE)
production rule set (PRS)
production rule sizing
layout
fabrication

Figure 1.1: Flow diagram of QDI synthesis procedure for an asynchronous system. The design of the register file follows the steps shaded in gray.

The intermediate phases can be considered refinements of the functional units.

Functional units can be further decomposed into control processes and data processes. Typically at this point, a numerical representation is chosen for data storage and communication, such as binary dual-rail (1of2) or quad-rail (1of4), although the representation may remain abstract. 1ofN or one-hot codes (and compositions thereof) are commonly used to encode values in delay-insensitive channel protocols. Asynchronous 1ofN communication actions strictly alternate between producing a value by raising a single rail and returning all rails to neutral or null (return-to-zero).

Vertical pipelining, which we will discuss in detail in Chapter 3, decouples control from data processes with the goal of improving throughput on a wide datapath [25]. We design our register file to use the *width-adaptive datapath* representation, which enables energy-efficient communication of compressed integers. We introduce width adaptivity as a transformation on non-width-adaptive pipelined processes in Chapter 5.

Communicating protocol actions of the fine-grain processes can be expressed as *handshaking expansions* (HSE), which can be translated into delay-insensitive production rules. Rather than prove the delay-insensitivity of production rules for every instance of communication, we can apply template compilation for just a few common HSEs to cover all communication actions. The advantage is that since the templates are proven correct in the general case, their specific instances are automatically correct. The Caltech MiniMIPS datapath primarily used the four-phase handshaking expansions of the *precharge half-buffer* (PCHB) and *full-buffer* (PCFB), whose protocols are listed in HSE Programs 1.1 and 1.2, and delay-insensitive circuit templates shown respectively in Figures 1.2 and 1.3 [23, 31]. In the HSEs, $[L]$ represents waiting for the presence of data on channel L, or data *validity*, and $[\neg L]$ represents input data *neutrality*. $R\uparrow$ represents sending data on the output channel R, and $R\downarrow$ represents resetting R. Acknowledgment signals are represented by superscript a or e. A positive input acknowledgment ($L^a\uparrow$ or $L^e\downarrow$) is returned when the input data token is no longer needed, and a request acknowledgment ($L^a\downarrow$ or $L^e\uparrow$) is returned when the process is ready to accept the next token. We refer to the portion of the HSE up to the positive input acknowledgment as the *set phase*, and everything thereafter including the input request as the *reset phase* of the expansion. The *en* signal in the PCFB represents an internal signal used to make states uniquely distinguishable for production rule synthesis. In the circuit figures, the dashed line represents an abstract completion tree for wider channels, which may invert the sense of the channel validity (R^v, L^v) signals. The generalization of the PCHB reshuffling to functions of multiple inputs and outputs is listed in HSE Program 1.3 and illustrated in Figure 1.4. Each output $R[m]$ is computed by a function $f_m()$, which depends on inputs $L[0..n]$.

Program 1.1 Equivalent HSEs: precharge half-buffer (PCHB)

```
    $*[[\neg R^a \wedge L]; R\uparrow; L^a\uparrow; [R^a]; R\downarrow; [\neg L]; L^a\downarrow]$
    $*[[R^e \wedge L]; R\uparrow; L^e\downarrow; [\neg R^e]; R\downarrow; [\neg L]; L^e\uparrow]$
```

Program 1.2 Equivalent HSEs: precharge full-buffer (PCFB)

$*[[\neg R^a \wedge L]; R\uparrow; L^a\uparrow; en\downarrow; (([R^a]; R\downarrow), ([\neg L]; L^a\downarrow)); en\uparrow]$

$*[[R^e \wedge L]; R\uparrow; L^e\downarrow; en\downarrow; (([\neg R^e]; R\downarrow), ([\neg L]; L^e\uparrow)); en\uparrow]$

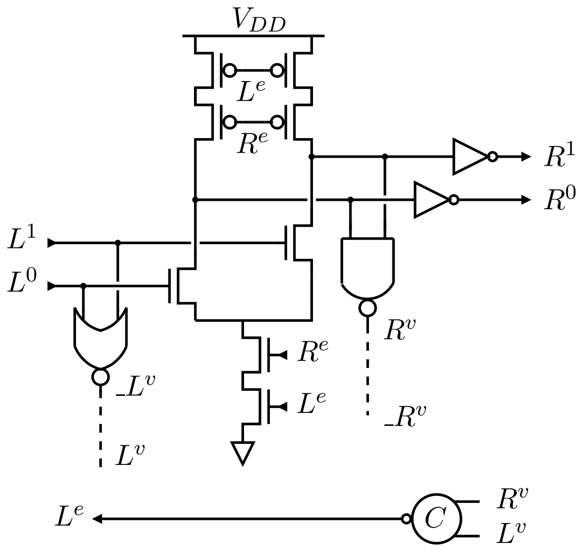


Figure 1.2: Precharge half-buffer (PCHB) with active-low acknowledgments

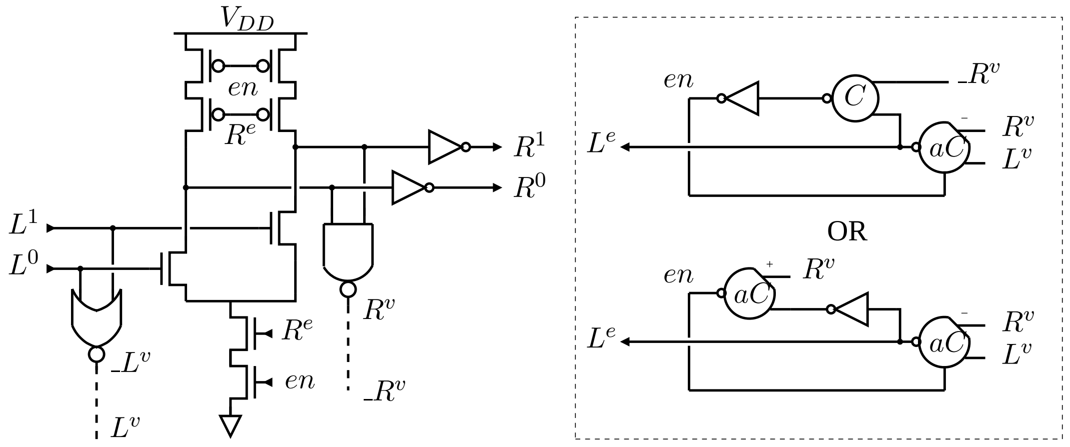


Figure 1.3: Two equivalent implementations of a precharge full-buffer (PCFB) with active-low acknowledgments

One additional phase we use in the synthesis of the register file is *floor decomposition*, which aids us in physically mapping the production rule set onto a partitioned plane for purposes of circuit layout, but more importantly, identifies and isolates circuit modifications introduced by various register file transformations and optimizations from a higher level. In the end, we have a complete set of production rules, which, by proof of semantic-preserving process transformations and delay-insensitive handshaking circuit templates, correctly implements

Program 1.3 HSE of a PCHB template for a function of multiple inputs and multiple outputs

```
$*[\langle\| \forall m : [R[m]^e \wedge L[0..n]]; R[m] := f_m(L[0..n])\rangle;$
  $L[0..n]^e\downarrow;$
  $\langle\| \forall m : [\neg R[m]^e]; R[m]\downarrow\rangle;$
  $[\langle\wedge\forall n : \neg L\rangle]; L[0..n]^e\uparrow$
$]$
```

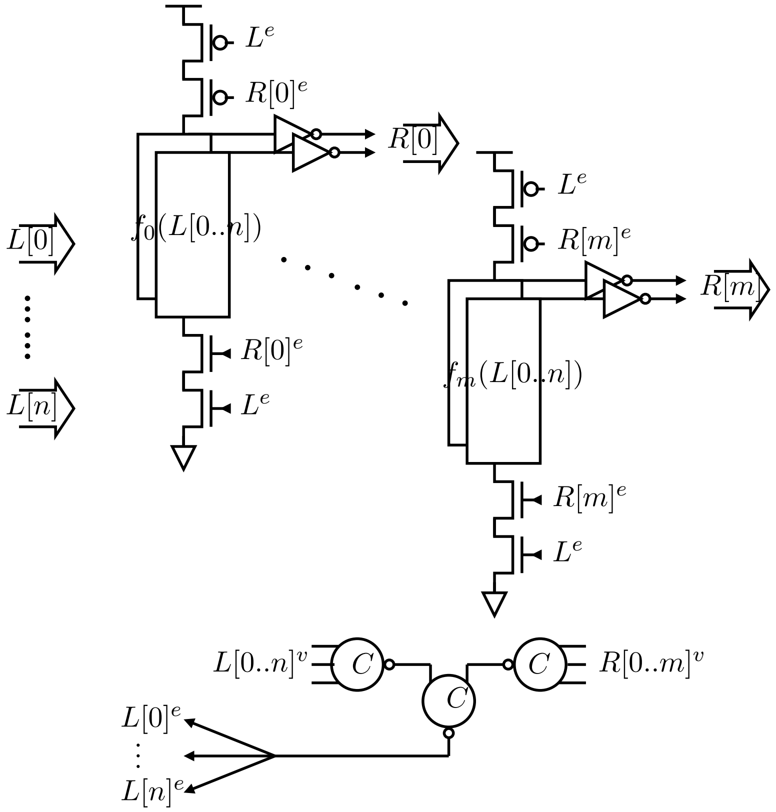


Figure 1.4: Abstract PCHB circuit template for a function with n inputs and m output channels.

the entire system as originally specified.

1.1.2 Register File Models

Area. Register files are most commonly modeled as small multiported memory arrays, with each cell storing one bit of information. Each cell is accessed by at least one (vertical) word line per port, typically two (horizontal) lines per bit per write port and at least one line per bit per read port [12,57,58]. While the number of gates of a register cell is only linearly proportional to the number of ports, the area scales linearly in both dimensions, therefore the cell area scales quadratically with the number of ports.

Capacitance. Capacitance governs the performance and energy characteristics of a register file. Sources of capacitance include gate fan-in and fan-out, wires, and parasitic diffusion capacitances. The capacitances on word and bit lines determine their switching rates and energies. The wire and parasitic components become increasingly significant as feature sizes shrink with advancing technology.

Speed. Assuming that gate fan-in/out loads can be switched with a properly amplified buffer chain, their delay is proportional to the log of the (lumped) capacitance [42]. Two other components of a register file's access time are word line and bit line delay. Both of these delays scale linearly with the number of ports because the length of the wires is determined by the size of each cell. For a monolithic array, the worst-case bit line delay is proportional to the number of register words in a bank, while the worst-case bit-line delay is proportional to the word size (architecture width in bits). The linearly scaling components of delay dominate the total delay for sufficiently large register files. As the number of ports, the word size, and the number of registers per banks increase, it becomes increasingly difficult for synchronous designs to support single-cycle accesses to large register files. Multi-cycle register files present their own problems because of the multiple levels of bypassing required, and their negative impact on the branch misprediction penalty [1].

Energy. On each access to the register file, one word line is switched for every set of bit lines switched, so the energy dissipated by bit lines is far more significant than the energy from the word line. The gate and diffusion capacitance components are proportional to the number of registers (transistors) sharing the same bit lines, whereas the wire capacitance is linearly proportional to both the number of registers and the number of ports [42]. The bit line loads of heavily-ported register files are dominated by wire capacitance. Although there exist circuit techniques for reducing energy dissipation, such as reducing voltage swing, differential voltage sensing, and current sensing, they only reduce energy by constant factors [57, 58]. Architectural changes in the bypass and register file organization have been proposed to reduce the number of ports and the number of accesses to the register file [1, 15, 36, 42, 58].

1.2 Overview

In Chapter 2, we formally initially decompose the sequential specification of the register file into three major concurrent processes: the core, bypass, and control. These coarse-grained processes are then decomposed into fine-grain processes. In Chapter 3, we introduce the vertical pipeline transformation, which decomposes a single logical data channel into smaller physical channels to improve throughput of data communications. We describe how register locking is implemented in the core of the register file, which preserves pipelined mutual exclusion among shared variables and channels while allowing control handshake to complete with constant response time. In Chapter 4, we transform the pipelined core processes into handshaking expansions using slightly different full-buffer and half-buffer templates, work through various floor decompositions in detail, and synthesize circuit production rules. We also show results for two sizes of register cores to quantify the benefits of register banking. Subsequent chapters in this thesis skip the floor decomposition steps, however, details for all floor decompositions are provided in a separate technical report for the interested and patient reader [11]. After Chapter 4, the reader should have a good understanding of how the Caltech MiniMIPS register file was designed [31], which closely resembles our initial base design of the register core. The core base design is used as a basis of comparison for the optimizations and transformations presented in the rest of the thesis. Table 1.1 summarizes which of the decomposed processes are affected by each transformation. An 'x' denotes where a process requires modification for a particular transformation.

Table 1.1: Register file components affected by various transformations

base design processes	Control	Bypass	Core
vertical pipelining (Ch. 3)		x	x
register banking (Sec. 4.4)	x	x	
width-adaptivity (Ch. 5)		x	x
register 0 read (Ch. 6)	x	x	
register 0 write (Ch. 6)	x		
port priority select (Ch. 7)	x	x	
unbalanced trees (Ch. 8)			x
register nesting (Ch. 9)			x

The following chapters present techniques and transformations for reducing energy consumed by the register file. In Chapter 5, we apply the width-adaptive representation to the (already vertically pipelined) register file, which reduces switching activity and energy by suppressing communication of leading 0s and 1s in integers on the datapath [25]. In Chapter 6, we examine some transformations that reduce energy consumption on read and write accesses to the hard-wired register zero. In Chapter 7, we apply a transformation to the bypass and control to suppress redundant copies of operands in the core.

The final chapters present transformations for increasing the throughput of the vertical control pipeline. In Chapter 9, we describe register array nesting, which introduces variable access time registers, but requires no change to the bypass or control. In these last two chapters, we also combine the new techniques with width adaptivity and the optimizations presented in the earlier chapters, which introduces some subtle cross-cutting issues.

Appendix A is a summary of the notation used in CHP programs. Appendices B through E contain program listings for various processes of the register file. Appendix G describes the global and local conventions used for resetting circuits. We have included listings for all derived production rules for the register core in Appendix H. Finally, all results that appear throughout the thesis are collected together in organized tables in Appendix J.

Chapter 2

Process Specification and Decomposition

We start the design process given a sequential behavioral specification of the register file. Using Martin's synthesis procedure, we decompose the original specification into smaller processes that can then be easily translated into production rules [29]. Our register file decomposition follows very closely to that of the Caltech MiniMIPS. since both architectures are based on the MIPS R3000 [31]. By the end of this chapter, we will have a set of fine-grain pipelined processes, whose concurrent operation correctly implements the sequential specification. In Chapter 3, we describe how to improve the throughput of data-communicating processes with vertical pipelining. In Chapter 4, we translate the final processes into handshaking expansions and production rules for the base design circuits.

2.1 Sequential Specification

Our RISC-based architecture specifies two read ports X, Y and two write ports $Z[0]$, $Z[1]$ for the register file, although in our in-order architecture, at most one value is written back at a time on any instruction iteration. Figure 2.1 shows a schematic of the channel interface between the register file and its neighboring processes.

One way one might write a sequential specification for the register file is shown in CHP Program 2.1. The program has two distinct phases: an operand read phase and a writeback phase. $reg[0..31]$ are the integer values of the 32 general purpose registers. RS, RT, and RD are the channels that respectively encode the indices (ranging from 0 to 31) of two source operands and a destination operand issued by the decode unit. An iteration of the register file operation begins with receiving the index variables rs, rt, and rd on their respective channels. In the read phase, a non-**null** value on rs or rt tells the register file to output the appropriate index register values on the respective X and Y operand buses. A **null** value on rs or rt means that there is no need to read an operand. In the writeback phase, the register file receives an exception status result from the writeback unit on each

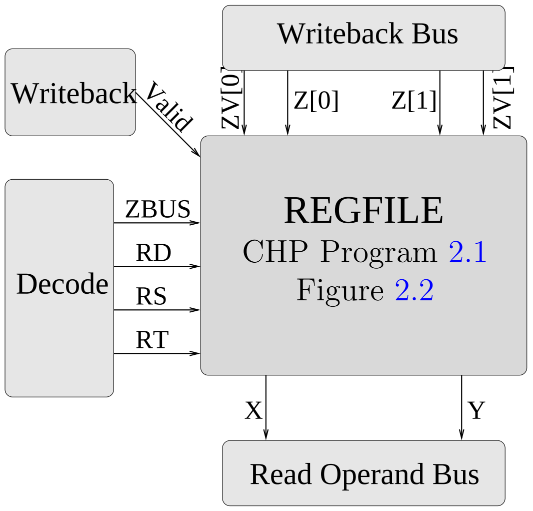


Figure 2.1: Register file's channel interface with its environment

Program 2.1 CHP: register file

```
$REGFILE \equiv$
    $*[RS?rs,\ \ RT?rt,\ \ RD?rd;$
       $[rs \neq \textbf{null} \longrightarrow X!reg[rs]\ \ [\!]\ \ \textbf{else} \longrightarrow \textbf{skip}],$
       $[rt \neq \textbf{null} \longrightarrow Y!reg[rt]\ \ [\!]\ \ \textbf{else} \longrightarrow \textbf{skip}];$
       $Valid?val;$
       $[rd \neq \textbf{null} \longrightarrow ZBUS?zbus; ZV[zbus]?zv;$
          $[zv \longrightarrow Z[zbus]?t\ \ [\!]\ \ \textbf{else} \longrightarrow \textbf{skip}];$
          $[val \wedge zv \wedge (rd \neq 0) \longrightarrow reg[rd] := t\ \ [\!]\ \ \textbf{else} \longrightarrow \textbf{skip}]$
       $[\!]\textbf{else} \longrightarrow \textbf{skip}$
       $]$
    $]$
```

iteration on channel *Valid*. **null** on *rd* means that no writeback result is expected. If a result is expected, the decode also communicates *ZBUS*, which indicates from which writeback bus data will be received. Execution units send a validity over the writeback bus on ZV, accompanied by data on Z if the result is valid. If the final result is valid, then the value of t is written into the *reg* array. Since the MIPS instruction set architecture (ISA) sometimes exchanges the source and destination register instruction fields, we require that the decode rearrange operands into their corresponding logical channels if necessary.

While this initial specification suffices for correctness, it is restricted to operating in alternating read-write phases, whereas the register file has the potential to perform both phases simultaneously in the absence of data dependences. Since the writeback result for an instruction must arrive some time after the operands issue, the register file can concurrently issue operands from one instruction while writing back values from a previous instruction. This phase-overlapping can lead to a situation where a register is read and written at the same time. The correct thing to do is to suppress reading the stale values and forward the recent writeback value to the operand bus. This is precisely what a *bypass* mechanism does. Program 2.2 lists the modified sequential specification of the register file with exposed bypass functionality.

Program 2.2 CHP: register file with explicit bypass

```
$REGFILE \equiv$
  $*[RS?rs,\ \ RT?rt,\ \ RD?rd;$
    $zx := (rs \neq \textbf{null}) \wedge (rs = z) \wedge (z \neq 0),$
    $zy := (rt \neq \textbf{null}) \wedge (rt = z) \wedge (z \neq 0);$
    $[rs \neq \textbf{null} \longrightarrow [zx \longrightarrow X!t\ \ [\!]\ \neg zx \longrightarrow X!reg[rs]]$
    $[\!]\textbf{else} \longrightarrow \textbf{skip}$
    $],$
    $[rt \neq \textbf{null} \longrightarrow [zy \longrightarrow Y!t\ \ [\!]\ \neg zy \longrightarrow Y!reg[rt]]$
    $[\!]\textbf{else} \longrightarrow \textbf{skip}$
    $];$
    $Valid?val;$
    $[rd \neq \textbf{null} \longrightarrow ZBUS?zbus;$
      $ZV[zbus]?zv;$
      $[zv \longrightarrow Z[zbus]?t\ \ [\!]\ \ \textbf{else} \longrightarrow \textbf{skip}];$
      $[val \wedge zv \wedge (rd \neq 0) \longrightarrow reg[rd] := t\ \ [\!]\ \ \textbf{else} \longrightarrow \textbf{skip}]$
    $[\!]\textbf{else} \longrightarrow \textbf{skip}$
    $];$
    $z := rd$
  $]$
```

In the read phase of Program 2.2, z holds the index of the register written from the previous iteration, and t saves the result of the last value written back. Local boolean variables zx and zy indicate whether or not t should be bypassed to the X or Y output buses in place of the value read from the core. For non-bypassed reads, X and Y receive their values directly from the reg array. Finally, the index rd is saved in z for comparison with the read indices for the following iteration. Note that it would also be correct to postpone $RD?rd$ until immediately after $Valid?$ since rd is not used until the writeback phase.

In the rare event that the register file receives **false** on $Valid$ from the writeback unit, indicating that an exception has occurred, it does not matter what output is produced because subsequent values are ignored and discarded until the instruction stream becomes ‘valid-again.’ The specification of the register file’s environment guarantees that the instruction that precedes the first valid-again instruction sends $RD!$**null**. The precise exception mechanism is orthogonal to the design of the register file, but the interested reader is invited to read how exceptions work in the MiniMIPS [28].

The register access control for the delayed writeback of the bypass now guarantees that the same register is never read and written in the same loop iteration, therefore it is safe to overlap the read and writeback phases. As we decompose the control component of the register file in Section 2.5, we will express precisely which actions can be parallelized.

2.2 Primary Decomposition

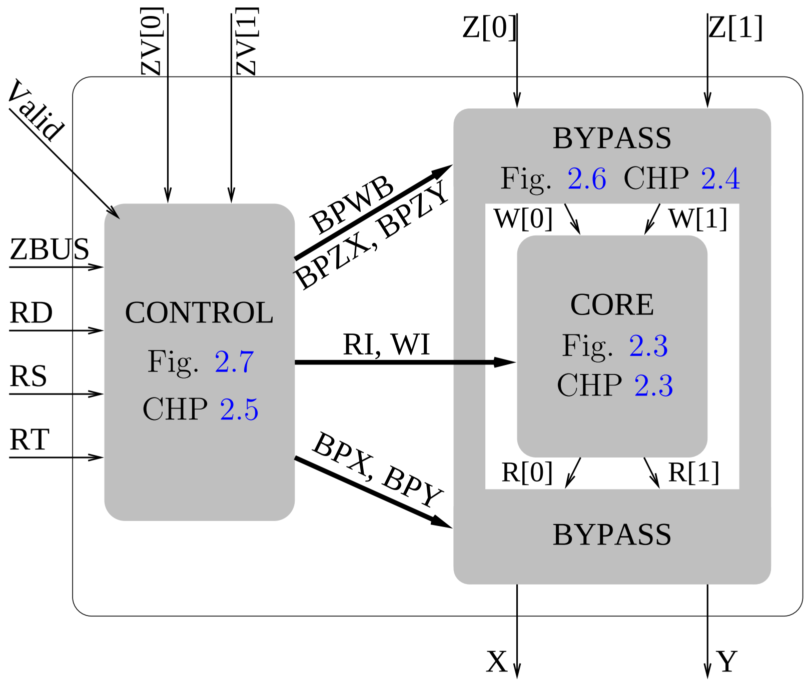


Figure 2.2: Schematic of the Register File process decomposition.

Our first step in decomposition is to isolate variables into separate processes, also known as *projection* [26]. We move all instances of the *reg* array into the *CORE* process, and move all instances of the *t* variable into the *BYPASS*. All remaining control variables will remain in the *CONTROL* process. The resulting parallel composition is:

$$REGFILE \equiv CONTROL \parallel BYPASS \parallel CORE$$

Figure 2.2 illustrates a schematic of the decomposition of the register file. The *CONTROL* must guarantee that it issues only safe and exclusive indices to the *CORE* such that on each iteration:

1. no two write ports write to the same $reg[i]$ in *CORE*

2. no $reg[i]$ is ever being concurrently read and written in *CORE*

The first requirement is already satisfied by the original specification, since the two write ports are mutually exclusive, however, the multiported *CORE* we present is capable of supporting multiple concurrent writebacks in other designs. The second requirement is guaranteed by comparing the source and destination indices to invoke bypass-forwarding when read and write indices match.

Core. The sole purpose of the *CORE* process is to provide accessible storage for the *reg* shared variables. To simplify the *CORE* as much as possible, we minimize the interface to the *CONTROL* to a single channel per port that communicates the index (which may be **null**) at the start of each iteration. Each port that receives a valid index performs a corresponding register read or write. We decompose *CORE* further in Section 2.3.

Bypass. The *BYPASS*, shown in CHP Program 2.4, provides an interface between the *CORE* and the datapath buses and receives steering controls from the *CONTROL*. We have introduced auxiliary variables x', x'', y', and y'' to differentiate the uses and definitions of temporary variable t. When $rd \neq \textbf{null}$ (and hence $z \neq \textbf{null}$ on the following iteration), *CONTROL* sends the *BYPASS* the conditional writeback signal *BPWB*, and the conditional copy signals *BPZX*, and *BPZY*, determined by *zbus*. We decompose *BYPASS* further in Section 2.4.

Control. To compose *CONTROL*, we take Program 2.2 and replace all uses of t with communications to the *BYPASS*, and replace all uses of *reg* with communications to the *CORE*. The result is listed in Program 2.5. We have rewritten the guards for the case statements of the read phase with equivalent guards in terms of zx and zy. For the X port, $zx \Rightarrow (rs \neq \textbf{null})$, and $\neg zx \Rightarrow (rs \neq z) \vee (z = \textbf{null}) \vee (z = 0)$. If $rs \neq \textbf{null}$, we guarantee that the bypass sends some output to the X bus, either from the writeback bypass or the core. If $z \neq \textbf{null}$, we guarantee that the token on the writeback bus is received and thus consumed. The same arguments hold symmetrically for the Y port and zy. We transform and decompose *CONTROL* further in Section 2.5. One can easily verify in the *CONTROL* that, between the writeback phase of one iteration and the read phase of the following iteration, the *BYPASS* steering signals are always issued coherently, i.e., token production and consumption are balanced in all processes, although the communication on the control channels need not be synchronized. The *CONTROL* is the only process that is specific to our architecture; the *BYPASS* and *CORE* processes that follow can be used in a more general class of architectures.

Program 2.3 CHP: register core

```
$CORE \equiv$
    $*[\, WI[0]?wi[0],\ WI[1]?wi[1],\ RI[0]?ri[0],\ RI[1]?ri[1];$
      $[ri[0] \neq \textbf{null} \longrightarrow R[0]!reg[ri[0]] \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}],$
      $[ri[1] \neq \textbf{null} \longrightarrow R[1]!reg[ri[1]] \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}],$
      $[wi[0] \neq \textbf{null} \longrightarrow$
        $[wi[0] = 0 \longrightarrow W[0]? \ \ [\!] \ \ \textbf{else} \longrightarrow W[0]?reg[wi[0]] \ ]$
      $[\!] \ \textbf{else} \longrightarrow \textbf{skip}$
      $],$
      $[wi[1] \neq \textbf{null} \longrightarrow$
        $[wi[1] = 0 \longrightarrow W[1]? \ \ [\!] \ \ \textbf{else} \longrightarrow W[1]?reg[wi[1]] \ ]$
      $[\!] \ \textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$
```

Program 2.4 CHP: register file bypass (sequential)

```
$BYPASS \equiv$
    $*[[\overline{BPWB[0]} \wedge \overline{BPZX[0]} \wedge \overline{BPZY[0]} \longrightarrow$
        $BPWB[0]?w0, BPZX[0]?zx0, BPZY[0]?zy0, Z[0]?t;$
        $[w0 \longrightarrow W[0]!t \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}],$
        $[zx0 \longrightarrow x' := t \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}],$
        $[zy0 \longrightarrow y' := t \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}]$
      $[\!]\overline{BPWB[1]} \wedge \overline{BPZX[1]} \wedge \overline{BPZY[1]} \longrightarrow$
        $BPWB[1]?w1, BPZX[1]?zx1, BPZY[1]?zy1, Z[1]?t;$
        $[w1 \longrightarrow W[1]!t \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}],$
        $[zx1 \longrightarrow x'' := t \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}],$
        $[zy1 \longrightarrow y'' := t \ \ [\!] \ \ \textbf{else} \longrightarrow \textbf{skip}]$
      $];$
      $BPX?mx, BPY?my;$
      $[mx = \text{"}z0\text{"} \longrightarrow x := x'$
      $[\!]mx = \text{"}z1\text{"} \longrightarrow x := x''$
      $[\!]mx = \text{"}core\text{"} \longrightarrow R[0]?x$
      $],$
      $[my = \text{"}z0\text{"} \longrightarrow y := y'$
      $[\!]my = \text{"}z1\text{"} \longrightarrow y := y''$
      $[\!]my = \text{"}core\text{"} \longrightarrow R[1]?y$
      $];$
      $X!x,\ Y!y$
    $]$
```

Program 2.5 CHP: register file control

```
$CONTROL \equiv$
    $z := \textbf{null}$;
    $*[RS?rs,\ \ RT?rt,\ \ RD?rd$;
      $zx := (rs \neq \textbf{null}) \wedge (rs = z) \wedge (z \neq 0)$,
      $zy := (rt \neq \textbf{null}) \wedge (rt = z) \wedge (z \neq 0)$;
      $[zx \longrightarrow RI[0]!\textbf{null}, BPZX[zbus]!\textbf{true}$,
        $[zbus = 0 \longrightarrow BPX!\text{"}z0\text{"}\ \ [\!]\ \textbf{else} \longrightarrow BPX!\text{"}z1\text{"}]$
      $[\!]\neg zx \longrightarrow\ \ RI[0]!rs$,
        $[z \neq \textbf{null} \longrightarrow BPZX[zbus]!\textbf{false}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}]$,
        $[rs \neq \textbf{null} \longrightarrow BPX!\text{"}core\text{"}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}]$
      $]$,
      $[zy \longrightarrow RI[1]!\textbf{null}, BPZY[zbus]!\textbf{true}$,
        $[zbus = 0 \longrightarrow BPY!\text{"}z0\text{"}\ \ [\!]\ \textbf{else} \longrightarrow BPY!\text{"}z1\text{"}]$
      $[\!]\neg zy \longrightarrow\ \ RI[1]!rt$,
        $[z \neq \textbf{null} \longrightarrow BPZY[zbus]!\textbf{false}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}]$,
        $[rt \neq \textbf{null} \longrightarrow BPY!\text{"}core\text{"}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}]$
      $]$;
      $Valid?val$;
      $[rd \neq \textbf{null} \longrightarrow ZBUS?zbus; ZV[zbus]?zv$;
        $[val \wedge zv \longrightarrow BPWB[zbus]!\textbf{true}, WI[zbus]!rd, WI[\neg zbus]!\textbf{null}$
        $[\!]\ \textbf{else} \longrightarrow BPWB[zbus]!\textbf{false}, WI[zbus]!\textbf{null}, WI[\neg zbus]!\textbf{null}$
        $]$
      $[\!]\ \textbf{else} \longrightarrow \textbf{skip}$
      $]$;
      $z := rd$
    $]$
```

2.3 Register Core

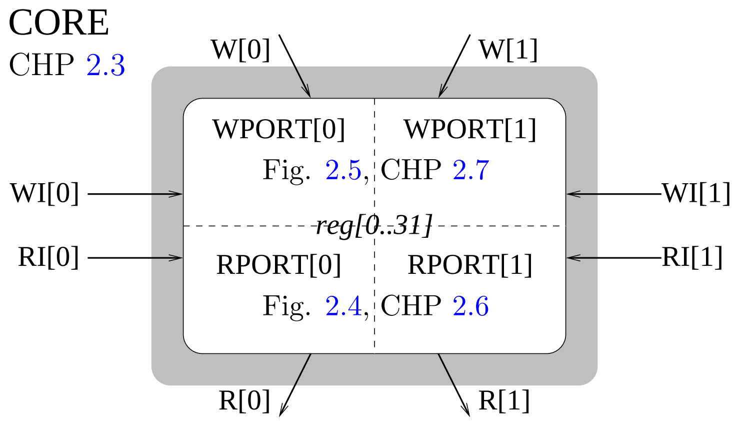


Figure 2.3: Schematic of the *CORE* decomposition

The decomposition of the *CORE* is relatively straightforward. The *CORE* has exclusive use of the *reg* array of variables. We are assured that the *CONTROL* will issue only compatible indices on any iteration, and that uses of the local index variables $ri\,[0..1]$ and $wi\,[0..1]$ are independent. Thus, we can decompose the *CORE* into concurrent processes corresponding to the ports, as shown in Figure 2.3:

$CORE \equiv RPORT\,[0] \parallel RPORT\,[1] \parallel WPORT\,[0] \parallel WPORT\,[1]$

The *reg* array variables are now shared among all the port processes. Program 2.6 defines a core read port and Program 2.7 defines a core write port process. For now, we must guarantee that the index controls on *RI* and *WI* do not become decoupled because decoupling could lead to violations of read-write exclusion for the registers. Therefore, we can only complete the receive actions $RI?$ and $WI?$ after the reads and writes have completed.[1]

Program 2.6 CHP: core read port

```
$CORE.RPORT\,[i] \equiv$
  $*[ri\,[i] := \overline{RI\,[i]};$
    $[(ri\,[i] \neq \textbf{null}) \wedge (ri\,[i] \neq 0) \longrightarrow R\,[i]!reg\,[ri\,[i]]$
    $[\!]\,(ri\,[i] \neq \textbf{null}) \wedge (ri\,[i] = 0) \longrightarrow R\,[i]!0$
    $[\!]$ $\textbf{else} \longrightarrow \textbf{skip}$
    $];$
    $RI\,[i]?$
  $]$
```

Furthermore, each port can be decomposed into a demux and data component for each register as follows:

[1]We read and use the value of a channel without acknowledging the channel with the notation $var := \overline{CHAN}$.

Program 2.7 CHP: core write port

```
$CORE.WPORT[j] \equiv$
  $*[wi[j] := \overline{WI[j]};$
    $[wi[j] \neq$ **null** $\longrightarrow W[j]?x;$
      $[wi[j] \neq 0 \longrightarrow reg[wi[j]] := x$ ▯ **else** $\longrightarrow$ **skip**$]$
    ▯ **else** $\longrightarrow$ **skip**
    $];$
    $WI[j]?$
  $]$
```

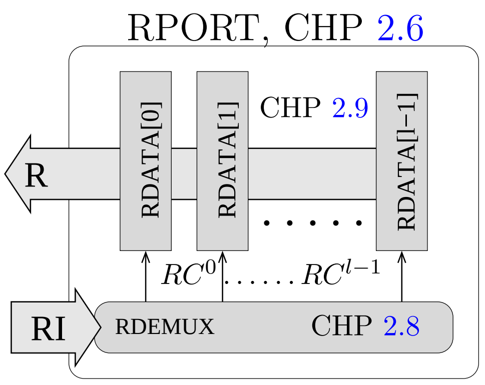


Figure 2.4: Schematic of read port

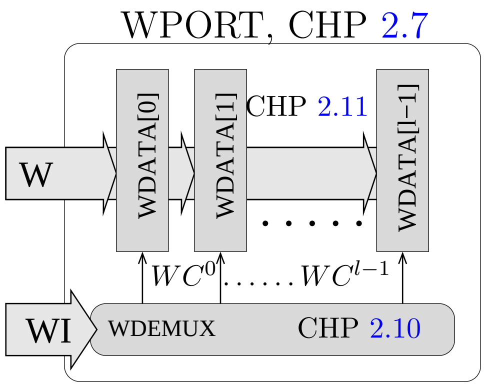


Figure 2.5: Schematic of write port

$$CORE.RPORT[i] \equiv RDEMUX[i] \parallel \langle \parallel \forall l : RDATA[l] \rangle$$

$$CORE.WPORT[j] \equiv WDEMUX[j] \parallel \langle \parallel \forall l : WDATA[l] \rangle$$

where *RDEMUX*, *RDATA*, *WDEMUX*, and *WDATA* are listed as Programs 2.8, 2.9, 2.10, 2.11, respectively. Now R is a shared output data channel, whose exclusive use is guaranteed by the read port's demux, as only one register is selected at a time per port. W is a shared input data channel, whose exclusive use is guaranteed by the write port's demux. RC and WC are exclusive, decoded select channels indexed by register line l and port number i. RC and WC can be interpreted as 1ofN-encoded channels that each use a single acknowledge.

By inserting the demuxes between the *CONTROL* and access to the *reg* array, we have introduced another potential pipeline stage between the control and shared data. Simply completing the receives on $RC?$ and $WC?$ in the *DATA* processes or $RI?$ and $WI?$ in the *DEMUX* processes in the beginning of the iteration without additional synchronization measures between the ports can decouple the ports and may lead to a situation where read-write exclusion of the *reg* array variables is violated.

In the *RDATA* process, for a read operation to remain atomic, we cannot complete the communication on $RC?$ before reading from *reg* has completed. We postpone completing the communication on $RC?$ until we are guaranteed that the read is complete, while allowing the read to start as soon as the probe of RC is true.

Analogously, to keep a write to *reg* atomic in the *WDATA* process, we postpone completing *WC*? until after the write is complete, while letting the write start when the probe of *WC* is positive. Specifying the *DEMUX*es in the same manner preserves the guarantee that reading and writing are completed before the input control tokens to the *CORE* are consumed and removed. In Chapter 3, we will discuss how to pipeline the read and write ports with locking to preserve exclusion in the presence of decoupling.

The MIPS ISA specifies that register zero (*reg*[0]) be hard-wired to the value 0. *reg*[0] is the only register that does not require read-write exclusion because its value is constant. Thus, it is safe to complete the *RC*? communication before sending 0 in *RDATA*, and safe to complete the *WC*? communication before completing the non-modifying write in *WDATA*.

In Chapter 3, we discuss the details of vertically pipelining *RDATA* and *WDATA* while preserving exclusion. In Chapter 4, we translate the pipelined design into handshaking expansions and production rules of circuits.

Program 2.8 CHP: read port demux

```
$RPORT[i].RDEMUX \equiv$
    $*[ri[i] := \overline{RI[i]};$
      $[ri[i] \neq \textbf{null} \longrightarrow RC[ri[i], i]! \ [] \ \textbf{else} \longrightarrow \textbf{skip}];$
      $RI[i]?$
    $]$
```

Program 2.9 CHP: single-register read port

```
$RPORT[i].RDATA[l(\neq 0)] \equiv$
    $*[\overline{RC[l, i]}; R[i]!reg[l]; RC[l, i]?]$

$RPORT[i].RDATA[0] \equiv$
    $*[RC[0, i]?; R[i]!0]$
```

Program 2.10 CHP: write port demux

```
$WPORT[j].WDEMUX \equiv$
    $*[wi[j] := \overline{WI[j]};$
      $[wi[j] \neq \textbf{null} \longrightarrow WC[wi[j], j]! \ [] \ \textbf{else} \longrightarrow \textbf{skip}];$
      $WI[j]?$
    $]$
```

Program 2.11 CHP: single-register write port

$WPORT\,[j]\,.\,WDATA\,[l(\neq 0)] \equiv$
    $*[\,\overline{WC\,[l,j]}\,;\ W\,[j]\,?reg\,[l]\,;\ WC\,[l,j]\,?\,]$

$WPORT\,[j]\,.\,WDATA\,[0] \equiv$
    $*[\,WC\,[0,j]\,?;\ W\,[j]\,?\,]$

2.4 Register Bypass

The remaining decomposition of the *BYPASS* revolves around an observation from the dataflow analysis of CHP Program 2.4: t is always written before it is read, so t is never live on exit from any iteration, neither are x', x'', y', or y'' because sending a variable on a channel counts as a use. Thus, all the bypass components are independent and may be decomposed as follows:

$$BYPASS \equiv BPZ\,[0] \parallel BPZ\,[1] \parallel BPZX \parallel BPZY$$

where the writeback (conditional copy) processes $BPZ\,[0]$ and $BPZ\,[1]$ (which are equivalent) are defined in Program B.1 and read-output merge processes *BPZX* and *BPZY* (also equivalent) are defined in Program B.2. Figure 2.6 illustrates the process decomposition of the bypass. The $BPZ\,[0..1]$ and $BPZX\,/\,Y$ processes are simple enough that we can translate them into canonical handshaking expansions. Since the *BYPASS* processes fit the templates for conditional output and conditional input [23], their synthesis into production rules is straightforward and uninteresting. Thus, we omit the remainder of the syntheses for the *BYPASS* from this thesis.

2.5 Register Control

We now finish the transformation and decomposition of *CONTROL*. We have observed in Program 2.5 that *rd* is not used until the writeback phase, and that a copy of it is saved in z. Also note that Program 2.5 requires z to be initialized to **null** before the main loop begins. If we peel the program loop back by one writeback phase, we eliminate the need for *rd* by receiving a delayed copy of *rd* with $RD'?z$. *CONTROL* now issues the write index of the previous iteration. The resulting program is listed as Program 2.12. The process that sends a delayed copy of *rd* on channel RD' is

$$CONTROL.RDCOPY \equiv RD'!\mathbf{null}; *[\,RD?rd;\ RD'!rd\,]$$

which is a simple buffer with an initial output token.

The read and write phases of *CONTROL* can now execute concurrently because we have eliminated dependencies across loop iterations. We decompose *CONTROL* into:

$$CONTROL \equiv RDCOPY \parallel RSCOMP \parallel RTCOMP \parallel WBCTRL \parallel ZBCOPY$$

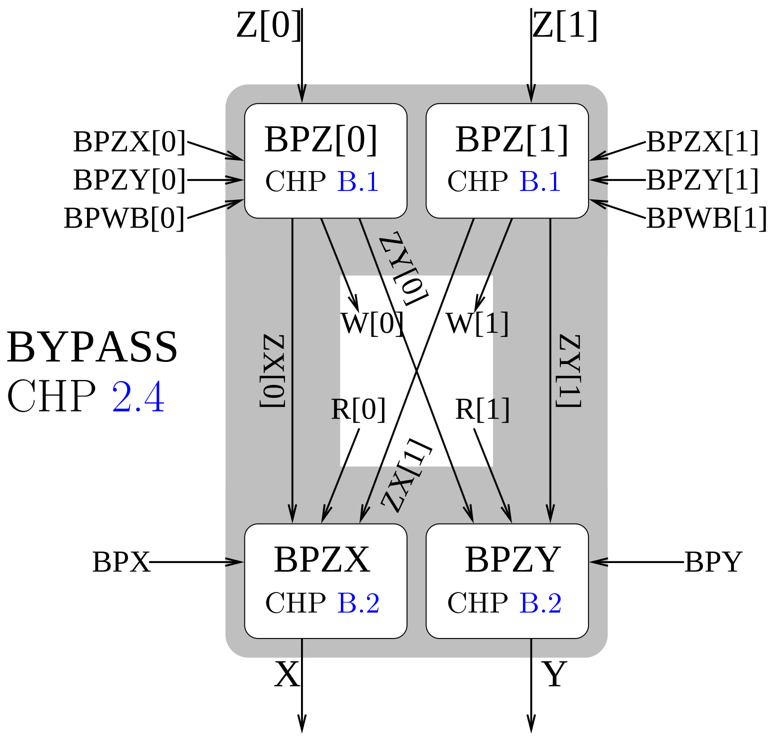


Figure 2.6: Schematic of the bypass decomposition

as illustrated in Figure 2.7. *RSCOMP* and *RTCOMP* (Program C.1) compare the source and destination indices to coordinate register reading between the *BYPASS* and *CORE*. *WBCTRL* (Program C.2) determines whether or not a writeback value will be committed to the *CORE*. These processes require private copies of *zbus* and z, so we introduce *ZBCOPY* and transform *RDCOPY* into a copy-buffer, as listed in Program C.3. These processes are simple enough to be synthesized as buffered logical functions with multiple inputs and outputs as described in [23], thus we omit the remainder of their syntheses from this thesis.

2.6 Summary

In this chapter, we have demonstrated how to decompose the sequential specification of our register file into fine-grain concurrent processes. Through semantic-preserving transformations, we have also proven that the parallel composition of the processes correctly implements the original behavioral specification. In Chapter 3, we further pipeline the data-driven components of the *BYPASS* and *CORE* for improved concurrency and throughput. In Chapter 4, we synthesize the pipelined processes into the production rules set that constitutes the base design register file.

Program 2.12 CHP: register file control, after rolling back one writeback phase

```
$CONTROL \equiv$
  $*[RS?rs,\ \ RT?rt,\ \ RD'?z,\ \ Valid?val;$
    $zx := (rs \neq \textbf{null}) \wedge (rs = z) \wedge (z \neq 0),$
    $zy := (rt \neq \textbf{null}) \wedge (rt = z) \wedge (z \neq 0);$
    $[z \neq \textbf{null} \longrightarrow ZBUS?zbus; ZV[zbus]?zv;$
      $[val \wedge zv \longrightarrow BPWB[zbus]!\textbf{true}, WI[zbus]!z, WI[\neg zbus]!\textbf{null}$
      $[\!]\textbf{else} \longrightarrow BPWB[zbus]!\textbf{false}, WI[zbus]!\textbf{null}, WI[\neg zbus]!\textbf{null}$
      $]$
    $[\!]\ \textbf{else} \longrightarrow \textbf{skip}$
    $],$
    $[zx \longrightarrow RI[0]!\textbf{null}, BPZX[zbus]!\textbf{true},$
      $[zbus = 0 \longrightarrow BPX!\text{"}z0\text{"}\ \ [\!]\ \textbf{else} \longrightarrow BPX!\text{"}z1\text{"}]$
    $[\!]\neg zx \longrightarrow\ \ RI[0]!rs,$
      $[z \neq \textbf{null} \longrightarrow BPZX[zbus]!\textbf{false}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}],$
      $[rs \neq \textbf{null} \longrightarrow BPX!\text{"}core\text{"}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}]$
    $],$
    $[zy \longrightarrow RI[1]!\textbf{null}, BPZY[zbus]!\textbf{true},$
      $[zbus = 0 \longrightarrow BPY!\text{"}z0\text{"}\ \ [\!]\ \textbf{else} \longrightarrow BPY!\text{"}z1\text{"}]$
    $[\!]\neg zy \longrightarrow\ \ RI[1]!rt,$
      $[z \neq \textbf{null} \longrightarrow BPZY[zbus]!\textbf{false}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}],$
      $[rt \neq \textbf{null} \longrightarrow BPY!\text{"}core\text{"}\ \ [\!]\ \textbf{else} \longrightarrow \textbf{skip}]$
    $]$
  $]$
```

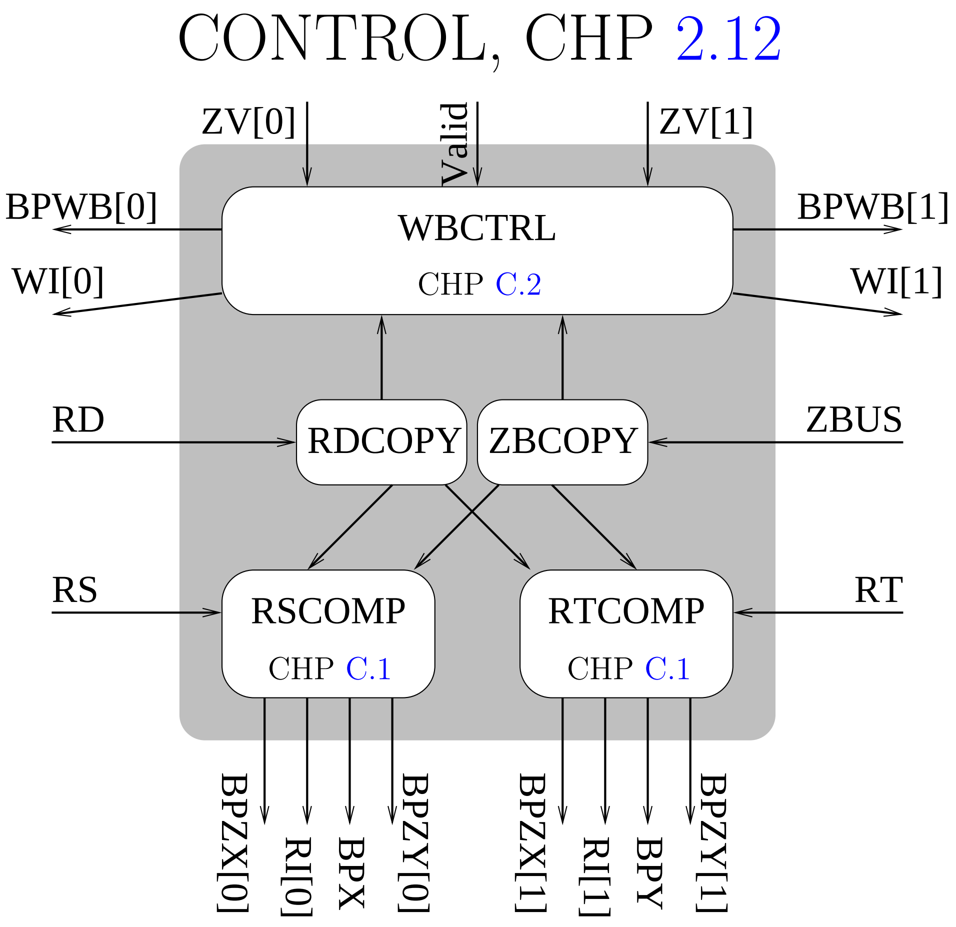


Figure 2.7: Schematic of the control decomposition

Chapter 3

Vertical Pipelining

Thus far, we have defined fine-grain processes of the register file that are independent of the data width or representation. Now we address the impact of data width on the cycle time and performance of the *CORE* and *BYPASS* processes. One of the limitations of QDI design is that local handshake cycle times include set and reset delays through channels' completion trees, therefore completing across wider data channels is slower than completing across narrower data channels. A secondary contributor to the cycle time is the delay of driving long word select wires that are shared vertically across all bits of a register line.

Vertically pipelining the *CORE* and *BYPASS* results in completion trees over narrower bundles of data, by decomposing a single logical data channel into a collection of constituent physical (or logical) channels. Vertical pipelining is a prime example of a process transformation that is motivated by consequences of physical implementation. In this chapter, we formalize the vertical pipeline transformation of the *CORE* and *BYPASS* at the CHP program level. We also discuss the necessary precautions for preserving atomicity of pipelined reads and writes to the *CORE*. Pipelined mutual exclusion guarantees coherent ordering among reads and writes to shared variables [27]. In Chapter 4, we break down the communication actions between vertically pipelined stages into handshaking expansions and finally synthesize them into production rules for the base designs. These initial base designs will serve as the basis of comparison for the optimizations presented in the remainder of this thesis.

3.1 Preliminary Concepts

The goal of pipelining is to improve performance by shortening critical paths, which applies to both synchronous and asynchronous designs. In synchronous designs, critical paths are the slowest paths between clocked register latches and hence dictate the maximum rate at which the system may be safely clocked. In self-timed asynchronous designs, however, critical paths may be determined by the pipeline dynamics of token-hole occupancy, but ultimately, system cycle times are bounded from below by the slowest cycle times of communication handshakes on frequently-used paths [53].

As we have mentioned in Section 1.1.2, one of the significant components in access time is the wire delay of the word lines, which is attributed to the gate fanout and capacitance of wiring across the entire array of bit rows. Figure 3.1a shows a monolithic, unpipelined register core with full-width fanout word lines. A common technique for driving long word lines is using an amplification chain for speed. In QDI designs, completion trees pose a greater threat to handshake cycle times. The MiniMIPS used a *pipelined completion* datapath, as shown in Figure 3.1b, where control signals are wire-copied to each of 8-bit blocks of the datapath, which in turn, generates local copies of control within each block. Pipelined completion results in narrower completion trees and reduced control fanout per block, and hence reduces the cycle time of all units on the datapath, including the register file. The pipelined completion blocks are synchronized by the copy-control which collects the acknowledgments of across all blocks.

One disadvantage of pipelined completion is that control signals still need to be wired across the entire datapath width to generate local control copies in each block. Doubling the word line interconnect requirement can mean requiring more metal layers or nearly doubling the word pitch of (horizontally) arrayed structures such as register cells, which already suffer from large bit line loads. We design our register files with *vertical pipelining*, as shown in Figure 3.1c, which also benefits from reduced data completion trees and reduced control fanout, but explicitly propagates control from block to block, and thus requires no full-width word line interconnect.[1] A design with both traditional (horizontal) and vertical pipelining is said to be *two-dimensionally pipelined* or *orthogonally pipelined*.

Pipelined completion and vertical pipelining incur the same circuit overhead over an unpipelined design. The area and energy overhead of control copying that is incurred depends on the granularity of pipelining. We can trade area and energy for increased throughput with finer pipelines, however, the improvement is limited by diminishing returns. The circuits for the vertically pipelined register file are actually *identical* to those of the pipelined completion design. Thus the expected energy difference between them is accounted for by the additional wire interconnect of copying control to each block in pipelined completion designs.

Since the *CORE* read and write port processes communicate on both control and data handshakes, their cycle times are limited by the slower of the two handshakes. Completion trees for the 1of32-channel control handshake can also limit the maximum throughput of the ports. Section 4.4, and Chapters 8 and 9 present different techniques for speeding up the control handshake.

A balanced distribution of word line control (such as wire-copying with pipelined completion) keeps register file accesses *block-aligned* (Figure 3.2) so that the bit lines are driven (within some timing margin) simultaneously. In synchronous aligned datapaths, all bits of each datum are communicated in the same clock

[1]We call this vertical because traditional pipeline diagrams show pipeline stages flowing horizontally from left to right.

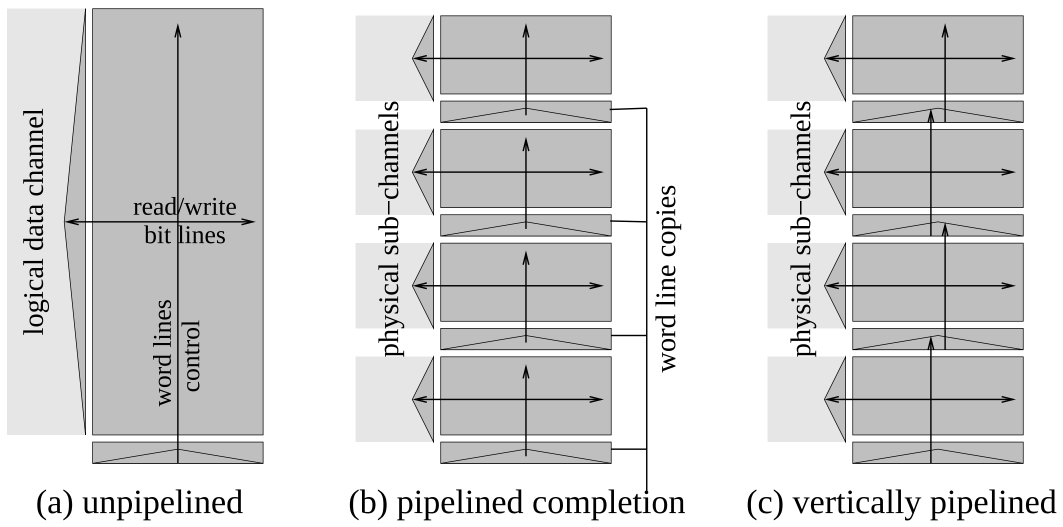


Figure 3.1: a) an unpipelined core completes across the full data-width for each data handshake whereas b) with pipelined completion, control signals are copied via copy-trees or wires to several blocks, and data completion detection is confined within each block, and c) a vertically pipelined core propagates control in a linear pipeline of blocks, and thus, does not require additional interconnect. In each subfigure, data is communicated horizontally and a decoded control arrives from the bottom. The thin rectangles represent control repeaters. The triangles in the figures represent completion detection trees in QDI asynchronous designs.

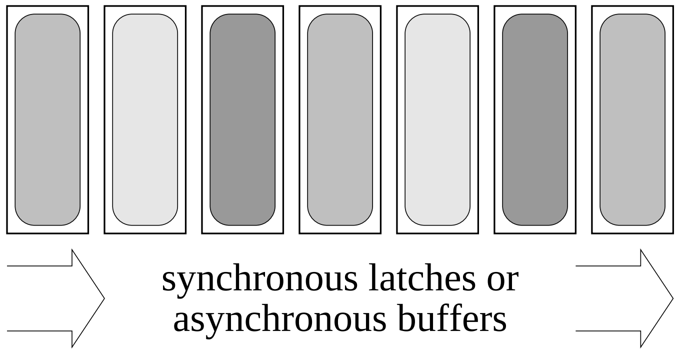


Figure 3.2: Synchronous or asynchronous block-aligned datapath communication

cycle, and in asynchronous aligned datapaths, all bits of each datum are synchronized by the same acknowledge.[2] However, such synchronization is not required in a delay-insensitive design; correctness is preserved even with unaligned control distribution, where bit lines may fire in an arbitrary sequence with unbounded delay. The timing characteristics depend entirely on the topology of control distribution. Vertical pipelining of asynchronous designs introduces a latency in propagating

[2] Multiple acknowledge wires may be used, as long as they are synchronized somewhere in the datapath [31].

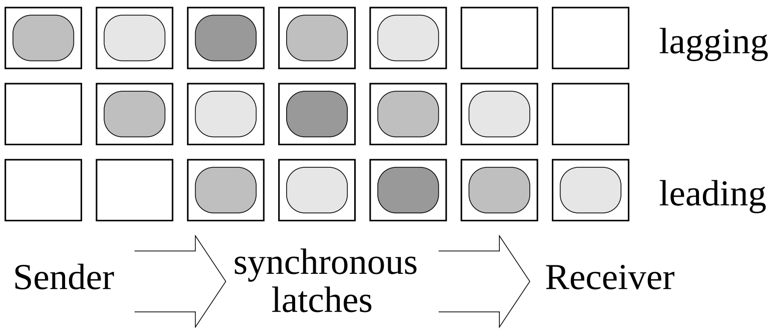


Figure 3.3: Synchronous parallel skewed vertical pipeline operation

control through pipeline stages. Data tokens on different physical sub-channels of the same logical data channel are no longer synchronized or aligned; rather, they are *block-skewed*, where there is some phase delay between communication of the block sub-channels.[3] We have chosen to propagate control from the least significant to the most significant blocks, the same direction as a ripple-carry. In Section 4.3.2, we show that the vertical latency per stage is only two gate delays through a single domino stage.[4] Block-skewed vertical pipelining in an asynchronous datapath allows the bottom (leading) blocks to start processing successive tokens while the top (lagging) blocks finish preceding tokens, as illustrated in Figure 3.4.

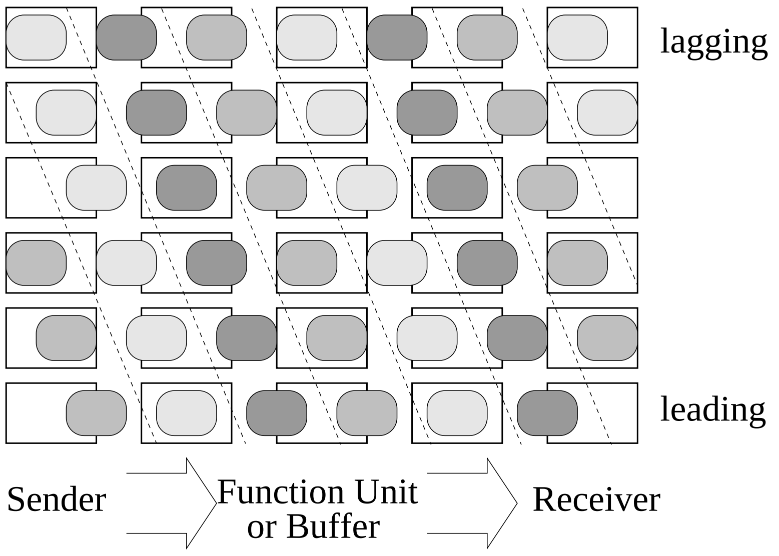


Figure 3.4: Snapshot of vertically pipelined, block-skewed datapath communication. Like-shaded rectangles correspond to the same logical token. Control to functional units are issued from the bottom, and propagated to the upper blocks, thus the lower blocks will *lead* the upper blocks.

[3] Called “byte-skewed” in Nyström’s dissertation [34], which specifies the granularity.
[4] True only when control propagation is independent of arriving data

Another quantity worth considering in choosing granularity is the total *vertical skew*, the difference between arrival times of control at the first and last pipeline stage. Operations that depend on results from the most significant blocks, such as conditional branches dependent on a compare, may slow down as the total vertical skew increases with finer pipeline granularity. Since conditional branches occur relatively frequently, a long vertical skew may adversely affect the overall system performance.

Asynchronously pipelining the core ports that use the array of shared variables *reg* without further modification may lead to a violation of mutual exclusion and ordering. Consider a 1-read, 1-write ported register core to which a write control token is issued followed by a read control token. Suppose the write port is stalled waiting for a result from a long latency operation. If the dependent read port races ahead of the stalled write, it will read out a stale value from the register, even though the control tokens were issued and completed sequentially. This violation of a flow dependence is commonly known as a read-after-write (RAW) hazard in traditional synchronous pipelines [38]. The reverse situation where a later write overtakes an earlier read (anti-dependence) is a WAR hazard, which can occur if the read operation is stalled. Violations of output dependences are write-after-write (WAW) hazards. One solution for synchronous pipelines is to detect such dependences and stall the pipeline as long as necessary to guarantee correctness. Synchronous bypasses can also forward dependent results to respective functional units to reduce the number of cycles stalled. The asynchronous solution of *pipeline locking* is analogous to the synchronous counterpart. The advantage of asynchronous locking is that the stall time is not restricted to any clock granularity; a stalled operation may resume as soon as it is unlocked, without waiting for the next clock edge. By restricting ourselves to only semantic-preserving transformations, the absence of data hazards in the sequential specifications automatically guarantees hazard-free concurrent implementations. In Section 3.5, we formalize the notion of locking in our concurrent process specifications.

3.2 Related Work

It is possible for synchronous designs to leverage vertical pipelining. Canal et al. proposed a byte-parallel, skewed micro-architecture for variable width operands on the datapath, in which higher order bytes are conditionally computed and communicated an entire clock cycle behind lower order bytes [4]. Although their proposed pipeline is optimized for full-width throughput, their vertical latency per byte is an entire clock cycle (Figure 3.3), which is many times longer than the delay through a single domino stage. The synchronous byte-parallel skewed architecture requires more latching activity and bypass forwarding overhead. We will revisit Canal's proposed architectures in Chapter 5 when we discuss our asynchronous implementation of significance compression.

A synchronous, vertically pipelined design could potentially use multi-phase overlapping clocks to reduce the vertical delay per block [16]. The advantage of self-timed vertical pipelining is that timing is governed by local handshakes, not by any global constraints or margins, and is robust with delay-insensitive design. Vertical pipelining incurs vertical control latching overhead in both synchronous and asynchronous designs. However, no additional (horizontal) data latches are required in the asynchronous datapath because the self-timed nature preserves ordering and dependences, although asynchronous FIFOs or buffers may be added in critical places to improve throughput [23].

There have been several approaches to preserving read-write ordering presented in past asynchronous designs. Paver (1992) et al. implemented a locking mechanism based on Sutherland's Micropipelines (bundled-data, bounded-delay timing model) to prevent RAW hazards in the AMULET1 processor [39, 46]. However, their design still had to stall on dependent operands. The AMULET2 included a lock FIFO along with a bypass mechanism where a writeback result could be use directly as an operand [14]. However, the design was further complicated by the necessary logic for determining the conditional execution of an ARM instruction. The AMULET3 adds a reorder buffer for out-of-order execution, but does not preclude the situation of concurrent reading and writing to a register [13]. Instead, they guarantee that the read register value will be overridden by the unconditionally forwarded value from the bypass and only require that the indeterminate values caused by conflict do not dissipate excess power.

The Caltech QDI MiniMIPS (1997) implemented pipeline locking in each stage of control copy to guarantee pipelined read-write exclusion and ordering [27, 31]. Even though the MiniMIPS' style of pipelining (pipelined completion) is different than what we present in our designs, the same underlying principles lead to the same production rules; the difference lies in the way we connect our pipelined blocks. In this chapter, we describe pipeline locking detail and apply it to our register file's control propagation.

The ASPRO-216 (1997) standard-cell QDI microprocessor architecture supported out-of-order writeback in the register file [41]. The only synchronization they required was a locking scheme to preserve read-after-write dependencies. The SDI TITAC-2 (1997), based on the MIPS R2000, included a register file with a read-after-write sequencer to stall the read of a concurrent read and write to the same register [48].

The Asynchronous Lattice Filter (1994) is the earliest example of a fine-grained, two-dimensionally pipelined asynchronous *bit-skewed* datapath [9]. Each *bit* of the datapath constituted a vertical pipeline stage, which practically eliminated completion trees on the entire datapath. The price paid for extremely low cycle time is pipeline area and energy overhead for every bit. We compromised with a granularity of four bits per vertical pipeline stage in all of the designs we explored in this thesis.

The argument for block-skewing was also presented in Nyström's dissertation in his proposed Single Pulse Asynchronous Microprocessor (SPAM) architecture [34]. Supporting arguments for block-skewing included: simplicity of arithmetic computation in most cases, which leads to simple layout, and trivial scalability to arbitrary-width datapaths because of the constant-overhead interconnect requirements without linearly scaling long-wires.

3.3 Pipeline Templates

We describe the template for the vertical pipeline transformation. Starting with a general unpipelined process as shown in Program 3.1, one simply divides full-width actions into partial-width actions, and propagates control from one stage to the next, in the manner shown in Program 3.2. Channel subscripts i and o differentiate between input and output control channels. While this transformation preserves the semantics of the original specification, its performance suffers from having each stage wait until control propagations (C_o!) are complete before finishing the receive actions (C_i?), i.e., the pipeline consists of *non-constant response time* (non-CRT) stages. The more vertical pipeline stages we use, the slower the cycle time! If the individual program actions from the unpipelined process are independent of one another (as is the case when no variables are shared), then one may complete the receive actions *concurrently* with the control send actions, as shown in Program 3.3. In the presence of shared variables, we employ explicit locks to guarantee that each pipeline stage receives input controls in the same order as the unpipelined version, even with constant-response-time (CRT) pipeline stages, which suffices to preserve the original semantics. The general form of locked pipelines is listed in Program 3.4. In Section 3.5, we will work out the example of pipeline locking in the *CORE* in detail and translate the *lock* and *unlock* conditions.

Program 3.1 CHP: template for an unpipelined process

```
*[($C_i$[1]?c[1], ..., $C_i$[$j$]?c[$j$]);
  $\langle$complete width actions $1 \ldots j\rangle$
 ]
```

Program 3.2 CHP: template for a non-CRT vertically pipelined process

```
*[(c[1] := $\overline{C_i[1]}$, ..., c[$j$] := $\overline{C_i[j]}$);
  $\langle$partial width actions $1 \ldots j\rangle$;
  ($C_o$[1]!c[1], ..., $C_o$[$j$]!c[$j$]);
  ($C_i$[1]?, ..., $C_i$[$j$]?)
 ]
```

3.4 Pipelined Bypass

Since the *BYPASS* processes share no variables between sub-processes, we can safely complete control reception before control propagation, because FIFO operation of control tokens suffices to guarantee correct operation. Programs B.4

Program 3.3 CHP: template for a CRT vertically pipelined process, with independent actions

```
*[($C_i$[1]?$c$[1], ..., $C_i$[$j$]?$c$[$j$]);
  $\langle$independent partial width actions $1 \ldots j\rangle$;
  ($C_o$[1]!$c$[1], ..., $C_o$[$j$]!$c$[$j$])
 ]
```

Program 3.4 CHP: template for a non-CRT vertically pipelined process, with locking

```
*[(($unlocked(1) \wedge c$[1] := $\overline{C_i\texttt{[1]}}$), ..., ($unlocked(j) \wedge c$[$j$] := $\overline{C_i\texttt{[}j\texttt{]}}$));
  $\langle$partial width actions $1 \ldots j\rangle$;
  (($lock(1)$; $C_o$[1]!$c$[1]), ..., ($lock(j)$; $C_o$[$j$]!$c$[$j$]));
  ($C_i$[1]?, ..., $C_i$[$j$]?);
  $unlock(1 \ldots j)$
 ]
```

and B.3 are the respective vertically pipelined versions of Programs B.1 and B.2. The terminal block (at the most significant position) of a vertical pipeline does not propagate any control; the CHP for the terminal read bypass omits the merge control BPX_o and the CHP for the terminal write bypass omits the copy controls $BPWB_o$, $BPZX_o$, and $BPWB_o$.

Production rules of the pipelined bypass processes can be synthesized by applying any standard QDI template such as PCHB or PCFB, thus we omit their derivation.

3.5 Pipelined Mutual Exclusion: Core

We have isolated the use of shared variables to only the *CORE* processes. The final step in transforming the register file core into pipelined processes is to apply pipeline locking to protect the use of the *reg* shared variables against data hazards. Program D.1 describes the read and write multi-ported process composition for a single register line. In this program, R and W represent full-width output and input channels shared across all registers. The demuxes guarantee that only one register per port is communicating on these channels, and accessing a particular *reg*[l] at any time.

Following the template transformation from Section 3.3, we divide the full-width operation into partial width operations, which results in Program D.2. The *reg*[l] variables are now divided into blocks of arbitrary bit granularity. The least significant blocks are controlled directly by the demuxes, and the most significant blocks omit control propagation.

Since Program D.2 is non-CRT, at most one control token may occupy a port at any given time, therefore the *CORE* (including the demuxes) preserves the mutual exclusion guaranteed by the *CONTROL*. Our goal, however, is to pipeline

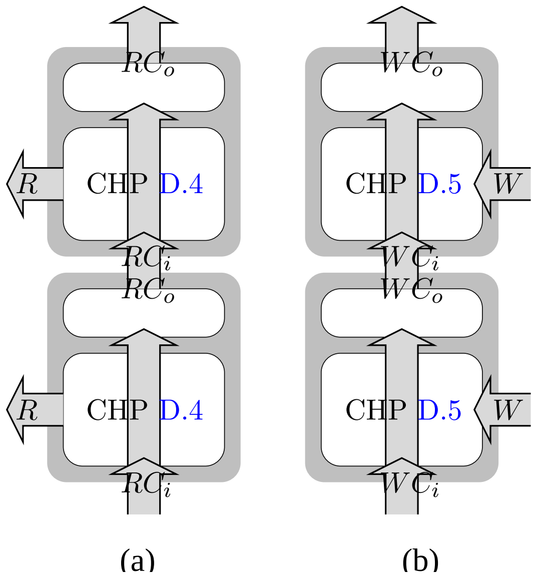


Figure 3.5: Pipelined core process blocks for a) reading and b) writing

the *CORE* in a manner that allows CRT while maintaining atomicity and mutual exclusion among accesses to shared variables.

We apply the transformations given by the theorems in [27] (whose template is shown in Program 3.5), which results in CHP Program D.3. We have introduce shared lock variables rx and wx and auxiliary channels RC' and WC' that guard the use of reg and control propagation actions. In the read port, we unlock $rx\downarrow$ after $R!reg$ and RC_o to guarantee exclusive use of reg and of the control rails. Analogously, we unlock $wx\downarrow$ after $W?reg$ and WC_o in the write port. When we introduce data-dependent control propagation in Chapter 5, this specification will be slightly modified.

Program 3.5 CHP: template for pipelined process with locking at the receivers

$$*[[\overline{C'}]; lock; C'; (\langle data\ \ action\rangle, C_o); unlock]$$
$$\| *[[\overline{C_i} \wedge unlocked]; C'; C_i]$$

To illustrate locking from the read port block, $rx\uparrow$ guards RC', so that after RC' is communicated, we are assured that WC' cannot occur until after $rx\downarrow$, thus we can acknowledge the input control RC_i. When $R!reg$ is finished, unlocking $rx\downarrow$ completes the iteration. The actions in the write port block occur analogously. The new block specifications are able to complete communication on the input control channels without having to wait for acknowledgment on the output control channels, thus we have CRT so we can expect pipelined, block-skewed behavior and performance on the datapath as described with Figure 3.4.

Currently, CHP Program D.3 specifies that the control *receivers* are responsible for maintaining the lock. But since we have multiple receivers (control and data bit cells) for each sender (control only), it would be more efficient to maintain locking at the *sender* and leverage the read-write exclusion of the input control channels. We show the template for this new transformation in Program 3.6. Since the word line controls RC and WC for a single register are guaranteed to be read-write exclusive, we no longer need to guard $\langle$*data actions*$\rangle$ (here, uses of *reg*) with lock variables; the locking variables *rx* and *wx* appear only in the control propagation component, so the data components are much simplified. C' is just a local copy of the C_i for the data actions. We have introduced a synchronization on C'' to signal to the control when the actions are finished, before completing the receive communications. In the next chapter we show that C'' translates to only a validity signal as opposed to a complete handshake. Applying this template results in Programs D.4 and D.5, which show the respective pipeline-locked read and write port processes with the locking maintained by the control-sending component. The terminal blocks for the read and write ports omit control propagation channels RC_o and WC_o and lock variables.

Program 3.6 CHP: template for pipelined process with locking at the sender

```
   *[$C'$; $\langle data\ \ action \rangle$; $C''$]
|| *[[$\overline{C_i}$]; $C'$; [$unlocked$]; $lock$; ($C_o$, ($C''$; $C_i$)); $unlock$
    ]
```

Shifting the responsibility of locking to the sender means that the *DEMUX*es from Programs 2.8 and 2.10 require the same locking mechanism as the control propagation components of the pipelined ports, if we want to decouple the input control from the decoded output. With the addition of locking variables and guards, the resulting CHP for the *DEMUX*es are shown in Programs D.6 and D.7.

It is admittedly somewhat clumsy to express this transformation precisely in high-level CHP without exposing the phases of handshaking. In Section 4.1, we fully specify the synchronizations required to correctly implement communication handshakes on the control and data channels.

3.6 Register Zero

For completeness, we include the CHP decomposition of the core read and write blocks for the hard-wired zero register. Since the value of *reg*[0] is constantly 0, all writes to this register are non-modifying. Reads and writes to the zero register may be freely re-ordered, thus, we can pipeline accesses to register zero without lock variables. The pipelined zero-register block is specified in CHP Program D.8.

In Chapter 6, we present alternatives for implementing register zero in the *CORE* by modifying the *BYPASS* and *CONTROL*.

3.7 Summary

We began this chapter by motivating vertical pipelining as a way to decrease the cycle time of data communication. We introduced vertical pipelining as a transformation of a single logical data channel into smaller physical channels, which results in narrower completion trees in each pipeline stage. The transformation of the *BYPASS* is straightforward because no shared variables are accessed. However, in order for the *CORE* to maintain pipelined, mutually-exclusive access to the shared channels and variables, we employed pipeline locking to preserve read-write ordering as issued by the *CONTROL* while allowing constant response time (CRT) in the vertical control pipeline. Although CRT vertical pipelining allows acknowledgment upon partial completion of a full-width data action, mutual exclusion preserves the atomicity of the full-width actions with shared variables.

With a new transformation, we shifted the responsibility of locking to the component that sends the control, which simplified the data read and write components by guaranteeing that the word select control channels for each register maintained read-write exclusion. Finally, we briefly described the difference for the hard-wired zero register in the core. The transformations presented in this chapter closely follow those used in the design of the Caltech MiniMIPS [31] — the intention here is to provide sufficient detail for understanding the high-level program transformations that impact the low-level synthesis. In Chapter 4, we break the pipelined read and write ports down into handshaking expansions and synthesize circuit production rules for the base design of the register file *CORE*.

Chapter 4

Core Base Design

In the last chapter, we concluded with CHP specifications of the vertically pipelined register core read and write ports. Now we synthesize these pipelined block processes into production rules for the base design of the *CORE*. The majority of the remainder of this thesis focuses on optimizing the *CORE* for throughput and energy efficiency. Since the *CONTROL* process depends largely on the architectural specification, we omit the derivation and optimization of its production rules from this thesis. Since the *BYPASS* processes' syntheses is straightforward and uninteresting, we also omit their production rules. The *CORE* alone spans a significant design space to explore, and therefore requires careful attention to design well. By focusing exclusively on the *CORE*, we are setting up for the optimizations in the remainder of the thesis with a detailed foundation for the base design.

4.1 Template Handshaking Expansions

The next step in synthesis is to expand the communication actions of the CHP processes into *handshaking expansions* or HSE. There are many QDI handshaking expansion templates that one could apply to synthesize the pipelined *CORE* processes [24]. The optimal choice of buffer reshuffling depends on the communication environment and the additional functionality required for a given CHP process. Ultimately, our choices are driven by circuit-level implications.

A vertical pipeline stage of a read or write port can be regarded as a control buffer with added register functionality. Though we have written the pipelined core processes as independent CRT (3.5) processes (with shared variables and channels), recall that each port process is controlled by exclusive 1of32-encoded channels, as specified by the demuxes. For each port, all registers share the same communication control with the environment, since the read and write data channels are shared. Signals that are wired across the array will have considerable fanout and load. For speed and energy and area, we would like to minimize the number of signals that are shared across the all registers.

We are, however, averse to implementing the core control propagation with the traditional PCHB and PCFB reshufflings because both require two enable signals in the precharge-domino stage for the output rails [23]; we use a variation that combines them to one signal, which also reduces the size of the transistor stacks.

We avoid recomputing the 1of32 control validity twice (once as input, once as output) by using a single shared validity signal that acts as output completion and as an input validity to the successor stage [53]. The shared validity signal also has the benefit of roughly halving the energy spent in detecting validity completion across each control channel. To maintain QDI, we must start the validity completion after the output inverter, as opposed to starting with a NAND gate like in the PCHB and PCFB. Non-QDI variations are feasible and have been evaluated [35], however, we restrict ourselves to only QDI circuits. Now the channel consists of data rails, and acknowledgment rail, and a validity rail. One could apply this transformation to the original PCFB and PCHB to yield equivalent reshufflings that use an extra validity rail in the handshake protocol.

We are interested in reshufflings that require only a single signal in the precharge stack. Two such template reshufflings that fit our criteria are the *precharge enable-valid full-buffer* or PCEVFB (Program 4.1, Figure 4.1) and *precharge enable-valid half-buffer* or PCEVHB (Program 4.2, Figure 4.2). *'Enable'* means that the channel response is active-low, whereas *'acknowledge'* is active-high. *'Valid'* means that the shared completion signal is active-high, whereas *'neutral'* is active-low.[1] Any combination of these is equivalent since they have the same underlying sequence of actions. The major difference between these reshufflings and the PCHB and PCFB is that the en actions have been postponed to succeed the R^e actions, which allows us to use only en in the precharge stage of the control rails. R^v serves both as the output validity of the current stage, and as the input validity of the successor. There are other possible reshufflings that meet our requirements, but rather than exhaust all possibilities, we restrict our attention to the PCEVFB and PCEVHB for the remainder of this thesis. Production rules for the enable-neutral PCENFB and PCENHB variations of the circuits presented are provided in the Appendix of the technical report [11], but are omitted from this thesis.

Program 4.1 Equivalent HSEs: precharge enable-valid full-buffer (PCEVFB)

$*[[\neg R^a]; en\uparrow; [L]; R\uparrow; L^a\uparrow; [R^a]; en\downarrow; R\downarrow, ([\neg L]; L^a\downarrow)]$
$*[[R^e]; en\uparrow; [L]; R\uparrow; L^e\downarrow; [\neg R^e]; en\downarrow; R\downarrow, ([\neg L]; L^e\uparrow)]$

Program 4.2 Equivalent HSEs: precharge enable-valid half-buffer (PCEVHB)

$*[[\neg R^a]; en\uparrow; [L]; R\uparrow; L^a\uparrow; [R^a]; en\downarrow; R\downarrow; [\neg L]; L^a\downarrow]$
$*[[R^e]; en\uparrow; [L]; R\uparrow; L^e\downarrow; [\neg R^e]; en\downarrow; R\downarrow; [\neg L]; L^e\uparrow]$

[1]There is currently no standard naming convention. This is what we have arbitrarily chosen for this thesis.

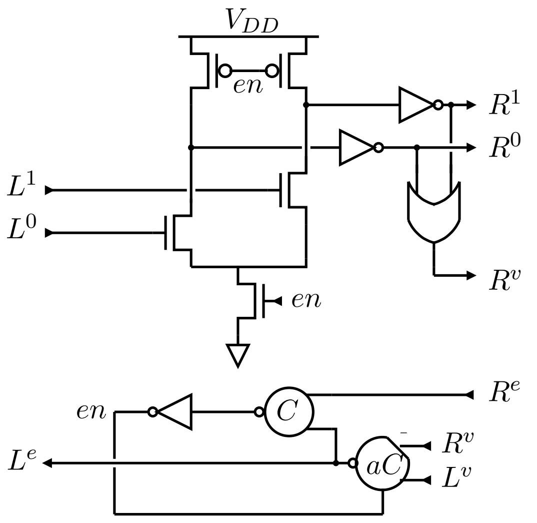


Figure 4.1: Precharge enable-valid full-buffer (PCEVFB) template

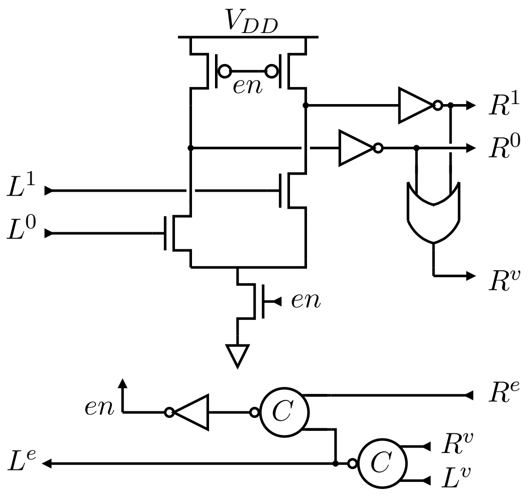


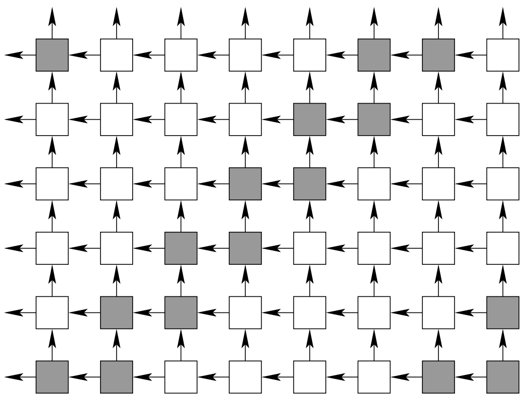
Figure 4.2: Precharge enable-valid half-buffer (PCEVHB) template

4.1.1 Half-Buffer vs. Full-Buffer

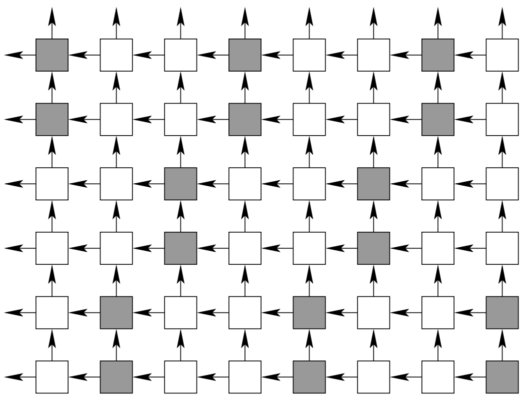
We are left to choose between the half-buffer and full-buffer reshufflings for the read and write port processes. Often cited reasons for preferring full-buffering are that its cycle time is shorter by roughly two transitions and that it provides greater slack per stage [23]. However, where throughput and slack are not critical, half-buffering has the advantage of being simpler to design due to its symmetry.

A vertically data-pipelined asynchronous datapath has the advantage of decoupling the data-pipeline from the control-pipeline. As discussed in Chapter 3, the major benefit from two-dimensional pipelining is that throughput is improved by the reduction in size of completion trees and reduction in fanout of control signals. Another property of two-dimensional pipelining is that the decoupled horizontal and vertical pipelines may have different buffering. Figure 4.3 illustrates the four combinations of buffering for an two-dimensional control-data pipeline.

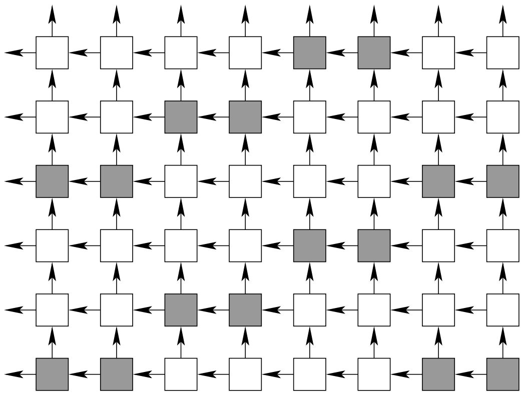
For example, high slack is preferred in the horizontal data direction to better accommodate in-flight data tokens in a cyclic datapath. It is reasonable to fix the data buffering of our design space to full-buffering. This only makes a minor difference for the read port, and makes no difference to the write port because is produces no data output tokens. Since the vertical control pipeline does not form a cycle, buffering the control pipeline with high slack is not critical. In the base design, it may seem obvious that full-buffering is a better choice for throughput, because the reshuffling allows more concurrency in the handshake. For optimization comparisons in the remainder of this thesis, we show results for both control bufferings.



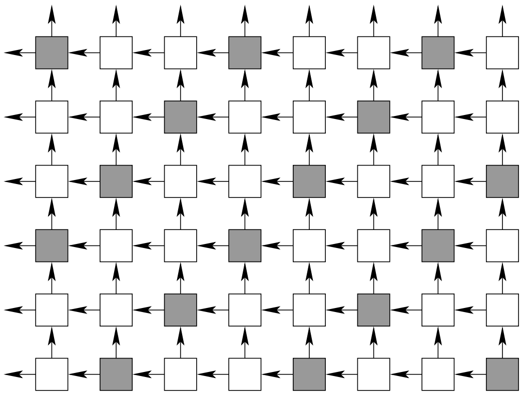
(a) half-buffer control, half-buffer data



(b) half-buffer control, full-buffer data



(c) full-buffer control, half-buffer data



(d) full-buffer control, full-buffer data

Figure 4.3: Examples of two-dimensional pipelining. The control pipeline is vertical, and the data pipeline is horizontal. Tokens are represented by diagonal bands of colored rectangles.

4.1.2 Core Read Port HSE

Recall from Programs D.3 and D.4 that the core read port pipelines take one control channel as an input, and produce a control output and a data output; a single input forks to two outputs. Applying the PCEVFB reshuffling to this process, we obtain Program 4.3. The $RC\uparrow$ action represents setting one of the 32 word select lines, and $R_o\uparrow$ represents setting all of the dual-rails of a pipeline block of the read port.

While this looks like a sufficiently simple expansion, there is one serious problem. Recall that in our base design, we have an array of 32 registers, and hence our production rules will have an array of precharge stages for control propaga-

Program 4.3 HSE: PCEVFB data-independent read port (version 1)

```
*[[$R_o^e \wedge RC_o^e$]; $ren\uparrow$; [$RC_i$]; ($R_o\uparrow$, [$unlocked() \longrightarrow lock$; $RC_o\uparrow$]); $RC_i^e\downarrow$;
  [$\neg R_o^e \wedge \neg RC_o^e$]; $ren\downarrow$; ($R_o\downarrow$, ($unlock$; $RC_o\downarrow$), ([$\neg RC_i$]; $RC_i^e\uparrow$))
 ]
```

tion, each of which require *ren* as an input. As we will show in Section 4.3.1, each register bit cell will also require *ren* as an input. Multiply the number of registers by the number of bits controlled by a single block (say four), and we are looking at a branching factor of over 160, a serious threat to our cycle time! To reduce the fanout, we decouple the control and data and give them separate enables ren_C and ren_D. This transformation gives us Program 4.4. Note that we can apply the same transformation to decouple each bit line, i.e., give each bit line its own ren_D with a fanout of roughly 32.

Program 4.4 HSE: PCEVFB data-independent read port (version 2)

```
*[(([$R_o^e$]; $ren_D\uparrow$), ([$RC_o^e$]; $ren_C\uparrow$));
  [$RC_i$]; ($R_o\uparrow$, [$unlocked() \longrightarrow lock$; $RC_o\uparrow$]); $RC_i^e\downarrow$;
  (([$\neg R_o^e$]; $ren_D\downarrow$), ([$\neg RC_o^e$]; $ren_C\downarrow$));
  ($R_o\downarrow$, ($unlock$; $RC_o\downarrow$), ([$\neg RC_i$]; $RC_i^e\uparrow$))
 ]
```

However, we have not finished exploiting all the available concurrency. Program 4.4 still enforces the orderings $ren_C\uparrow \prec R_o\uparrow$ and $ren_D\uparrow \prec RC_o\uparrow$ and their inverses, which are unnecessary. By removing the sequential synchronizations around [RC_i], we can decouple the output setting actions. In the reset phase, removing the synchronization after $ren_C\downarrow$ and $ren_D\downarrow$ decouples the $R_o\downarrow$ and $RC_o\downarrow$ actions. Finally we must check that ren_C and ren_D have reset before requesting the next control token with $RC_i^e\uparrow$ and restarting the cycle. The final transformed result is Program 4.5. One can generalize the same transformation to decouple each bit line and generate a separate ren_D for each bit line.

Program 4.5 HSE: PCEVFB data-independent read port (version 3)

```
*[(([$R_o^e$]; $ren_D\uparrow$; [$RC_i$]; $R_o\uparrow$),
   ([$RC_o^e$]; $ren_C\uparrow$; [$RC_i \wedge unlocked()$]; $lock$; $RC_o\uparrow$));
  $RC_i^e\downarrow$;
  (([$\neg R_o^e$]; $ren_D\downarrow$; $R_o\downarrow$),
   ([$\neg RC_o^e$]; $ren_C\downarrow$; $unlock$; $RC_o\downarrow$),
   ([$\neg RC_i \wedge \neg ren_D \wedge \neg ren_C$]; $RC_i^e\uparrow$)
  )
 ]
```

If we apply the same transformations we used for the PCEVFB on the PCEVHB reshuffling of the core read port, the result is HSE Program 4.6, which keeps the data output full-buffered while control propagation remains half-buffered. Since the data output handshake with the read bypass is straightforward, we chose to fix the data handshake as a full-buffer for better throughput throughout the remainder of the thesis.

Program 4.6 HSE: PCEVHB data-independent read port with full-buffered data output, and half-buffered control output (version 3)

```
$*[(([R_o^e]; ren_D\uparrow; [RC_i]; R_o),$
    $([RC_o^e]; ren_C\uparrow; [RC_i \wedge unlocked()]; lock; RC_o\uparrow));$
  $RC_i^e\downarrow;$
  $(([\neg R_o^e]; ren_D\downarrow; R_o\downarrow),$
    $([\neg RC_o^e]; ren_C\downarrow; unlock; RC_o\downarrow; [\neg ren_D \wedge \neg RC_i]; RC_i^e\uparrow));$
 ]
```

The terminal block of the pipelined read port has a single control input and a single data output. The HSE shown Program 4.7 is just Program 4.5 stripped of the RC_o control output channel, and is equivalent to a full-buffer because $RC_i^e\uparrow$ does not wait for $R_o\downarrow$.

Program 4.7 HSE: terminal block of read port

```
$*[[R_o^e]; ren_D\uparrow; [RC_i]; R_o\uparrow; RC_i^e\downarrow;$
  $(([\neg R_o^e]; ren_D\downarrow; R_o\downarrow), ([\neg RC_i \wedge \neg ren_D]; RC_i^e\uparrow))$
 ]
```

4.1.3 Core Write Port HSE

Program 4.8 HSE: PCEVFB data-independent write port (version 1)

```
$*[[WC_o^e]; wen\uparrow; [WC_i];$
  $([unlocked() \longrightarrow lock; WC_o\uparrow], [W_i \longrightarrow \langle write\rangle]);$
  $(WC_i^e\downarrow, W_i^e\downarrow);$
  $[\neg WC_o^e]; wen\downarrow; ((unlock; WC_o\downarrow), ([\neg WC_i \wedge \neg W_i]; (WC_i^e\uparrow, W_i^e\uparrow)))$
 ]
```

Applying the PCEVFB reshuffling to the write port (Programs D.3 and D.5), we obtain Program 4.8. The join process for the write port is asymmetric, unlike the fork process of the read port. Control propagation is independent from the arriving data, i.e., $WC_o\uparrow$ does not have to wait for $[W_i]$, whereas writing to the actual register must wait for $[WC_i \wedge W_i]$. (We will introduce data-dependent control in Chapter 5.)

The key observation is that since writing does not produce an output, there is no need for $\langle write\rangle$ to be guarded by wen; only the output control propagation requires *wen* in its guards. After decoupling register writing from control propagation, the result is Program 4.9. We have introduced a new variable *wvc* that indicates when writing is complete, and combined $WC_i^e \equiv W_i^e$ because they share the same guards.

Note that the expansion for the data-writing component is completely independent of the reshuffling chosen for the control propagation. We can short-cut through the same derivation for the PCEVHB reshuffling, which results in Program 4.10.

Program 4.9 HSE: PCEVFB data-independent write port (version 4)

```
$*[[WC_o^e]; wen\uparrow; [WC_i \wedge unlocked()]; lock; WC_o\uparrow; [wvc]; WC_i^e\downarrow;$
  $[\neg WC_o^e]; wen\downarrow; ((unlock; WC_o\downarrow), ([\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow))$
 $]$
$*[[WC_i \wedge W_i]; \langle write \rangle; wvc\uparrow; [\neg W_i]; wvc\downarrow]$
```

Program 4.10 HSE: PCEVHB data-independent write port (version 4)

```
$*[[WC_o^e]; wen\uparrow; [WC_i \wedge unlocked()]; lock; WC_o\uparrow; [wvc]; WC_i^e\downarrow;$
  $[\neg WC_o^e]; wen\downarrow; unlock; WC_o\downarrow; [\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow$
 $]$
$*[[WC_i \wedge W_i]; \langle write \rangle; wvc\uparrow; [\neg W_i]; wvc\downarrow]$
```

The terminal block for the write port pipeline is listed in Program 4.11 is the same as Program 4.9 without the WC_o control propagation channel. Since the terminal write block only takes in a control input and a data input, and produces no output, buffering does not apply to the terminal block.

Program 4.11 HSE: terminal block of write port

```
$*[[WC_i \wedge wvc]; WC_i^e\downarrow; [\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow]$
$*[[WC_i \wedge W_i]; \langle write \rangle; wvc\uparrow; [\neg W_i]; wvc\downarrow]$
```

4.2 Floor Decomposition

In the *floor decomposition* phase of synthesis, we partition the handshaking expansions into components that correspond to the physical placement of production rules. As we explore the design space of different optimizations, we can isolate the modifications to specific regions of the floor decomposition. This makes production rule and layout generation conveniently modular for our study on register core optimizations.

Figure 4.4 shows how the *CORE.BLOCK* process is physically decomposed into four quadrants: the register data cell array, the control propagation and completion array, data interface and communication array, and the handshake control. The chosen reshuffling (full or half buffer) only affects the handshake control block. As we floor-decompose the read and write ports, we will show how the various control and data signals fit into the figure.

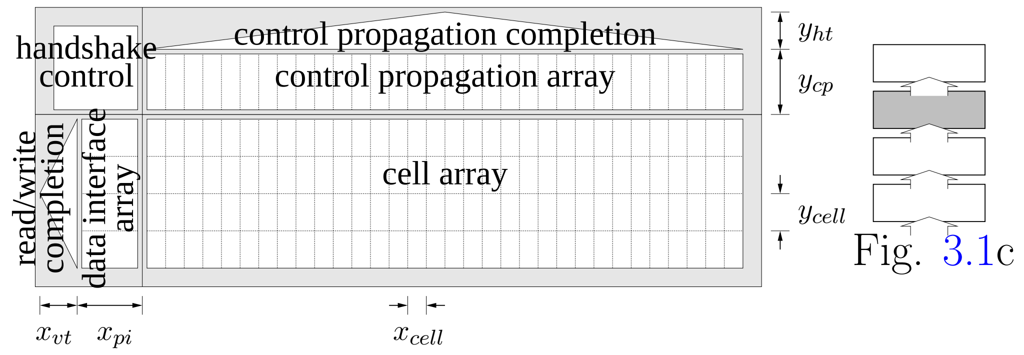


Figure 4.4: Floorplan of a vertically pipelined register core block for reading and writing, from Figure 3.1c

4.2.1 Decomposed Reading

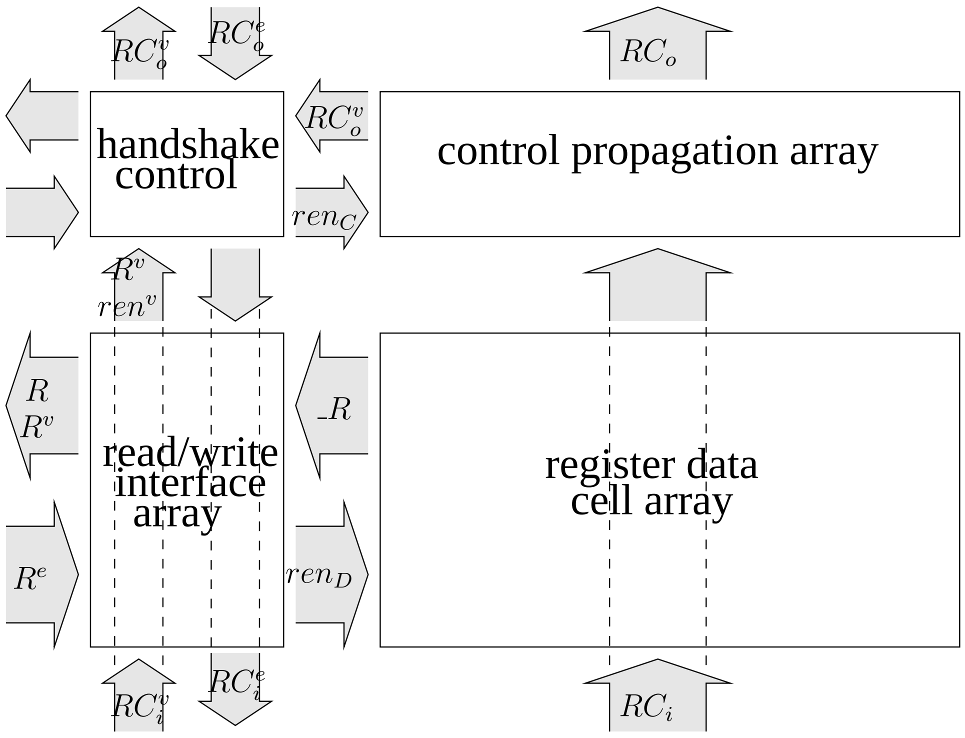


Figure 4.5: Floor decomposition of a read port block, shown with channel signals and some internal signals at component boundaries

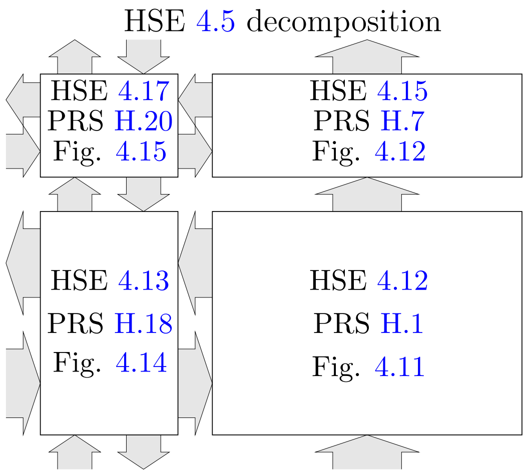


Figure 4.6: Floor decomposition of a PCEVFB read port

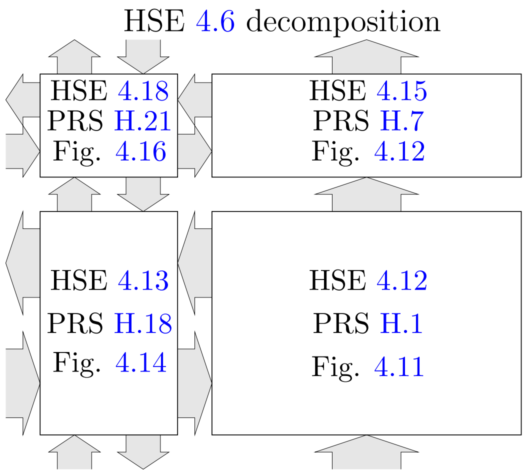


Figure 4.7: Floor decomposition of a PCEVHB read port

Figure 4.5 shows the relative placement of channel signals and internal nodes at the boundaries of the floor decomposition components. Figures 4.6 and 4.7 outline the steps that follow in synthesizing production rules from the full-buffer and half-buffer handshaking expansions of the core read port. As we apply floor decomposition to the HSEs of the various core ports, we often find it necessary or convenient to lower the abstraction of certain signals and actions to expose the CMOS-implementability requirement. For CMOS, a production rule set (PRS) may contain only guarded actions whose output is opposite to the sense of their input, i.e., each production rule must be inverting. In the read port floor decomposition, we introduce a pseudo-channel, $\_R$, which represents the inverted sense of the R shared data channel. $\_R$ will be wire-shared across the register cell array, which means we can use only NFETs (stronger than PFETs) in the cells, which use RC_i as active high inputs. The R data channel rails which connect to the environment, will be driven by inverters, which has the advantage of high gain amplification.

Program 4.12 HSE: the register read cell array component, set-only

```
$REG\_DATA_{read}\texttt{[}b, l\texttt{]} \equiv$
    $*\texttt{[[}RC_i\texttt{[}l\texttt{]} \wedge ren_D\texttt{[}b\texttt{]]}; \_R\texttt{[}b\texttt{]}\downarrow; \texttt{[}\_R\texttt{[}b\texttt{]]]}$
```

Program 4.13 HSE: the register read data interface with $\_R$ reset

```
$REG\_INTRFC_{read}\texttt{[}b\texttt{]} \equiv$
    $*\texttt{[[}R^e \wedge RC_i^e\texttt{]}; ren_D\texttt{[}b\texttt{]}\uparrow; \texttt{[}\neg\_R\texttt{[}b\texttt{]]}; R\texttt{[}b\texttt{]}\uparrow;$
      $\texttt{[}\neg R^e \wedge \neg RC_i^e\texttt{]}; ren_D\texttt{[}b\texttt{]}\downarrow; \_R\texttt{[}b\texttt{]}\uparrow; R\texttt{[}b\texttt{]}\downarrow$
     $\texttt{]}$
```

The read cell, shown in Program 4.12, is replicated in a 2-dimensional array, once per register word line (32), and once per bit line (4). The read data interface, Program 4.13, is replicated (vertically) once per bit line. Each bit line will have its own ren_D, $\_R$, and R while R^e and RC_i^e are wire-shared across all the vertical array of data interface cells. Since the pull-up production rule for $\_R\uparrow$ is independent of the register word selected by RC_i, we can use a single $\neg ren_D$ pull-up on $\_R$ instead of replicating the same rule in every cell.

Program 4.14 HSE: completion tree for R^v in read port

```
$REG\_CT_{R^v} \equiv$
    $*[[\langle \wedge \forall b :: R[b] \rangle]; R^v\uparrow; [\langle \wedge \forall b :: \neg R[b] \rangle]; R^v\downarrow]$
```

The R^v read validity signal is completed across all interface cells in a block in Program 4.14, and is connected up to the handshake control for this block and as an output request to the data environment. With the full-buffer reshuffling on channel R, since $R^v\downarrow$ (which follows $ren_D\downarrow$) is not checked before $RC_i^e\uparrow$, the handshake control and read data interface must check $ren_D\downarrow$ before requesting the next input control token with $RC_i^e\uparrow$ to prevent $\_R\downarrow$ from prematurely firing before it is reset.

The control propagation arrays are similar for read and for writing, as listed in Program 4.15. We use $lock_r\uparrow$ and $lock_w\uparrow$ to denote the locking actions, and $lock_r\downarrow$ and $lock_w\downarrow$ to denote the unlocking actions. The *unlocked* conditions correspond to the conjunctions of the individual *lock* variables specified in Program D.3. In Section 4.3.2, we will implement the locking and unlocking actions as production rules. The control validity completion trees are listed in Program 4.16 — they are just OR-trees. Without loss of generality, we have written the all control validity signals in the active-high *valid* sense (denoted X^v) as opposed to the *neutral* sense (denoted X^n).

Program 4.15 HSE: the register control propagation array (read and write)

```
$REG\_CTRL\_PROP_{read}[l] \equiv$
    $*[[RC_i[l] \wedge ren_C \wedge \langle unlocked[l] \rangle]; lock_r[l]\uparrow; RC_o[l]\uparrow; RC_o^v\uparrow;$
      $[\neg ren_C]; lock_r[l]\downarrow; RC_o[l]\downarrow; RC_o^v\downarrow$
     ]
$REG\_CTRL\_PROP_{write}[l] \equiv$
    $*[[WC_i[l] \wedge wen \wedge \langle unlocked[l] \rangle]; lock_w[l]\uparrow; WC_o[l]\uparrow; WC_o^v\uparrow;$
      $[\neg wen]; lock_w[l]\downarrow; WC_o[l]\downarrow; WC_o^v\downarrow$
     ]
```

What remains of the original HSE is the handshake control that coordinates the communication with the environment via requests and acknowledges. A full-buffer version of the control is listed in Program 4.17, and a half-buffer version is listed in Program 4.18. By careful floor decomposition, we have isolated production rule

Program 4.16 HSE: completion tree for control propagation array in the read and write port

```
$REG\_CT_{RC} \equiv$
    $*[[\langle \vee\forall l :: RC_o[l] \rangle]; RC_o^v\uparrow; [\langle \wedge\forall l :: \neg RC_o[l] \rangle]; RC_o^v\downarrow]$
$REG\_CT_{WC} \equiv$
    $*[[\langle \vee\forall l :: WC_o[l] \rangle]; WC_o^v\uparrow; [\langle \wedge\forall l :: \neg WC_o[l] \rangle]; WC_o^v\downarrow]$
```

differences between our chosen buffer reshufflings to only the handshake control quadrant; the other three quadrants are independent of the reshuffling! The read handshake control takes control validity completion signals RC_i^v and RC_o^v, a data validity completion R^v, and internal enable completion ren^v from Program 4.19 as inputs from the other quadrants of the read block.

An elegant result of the handshake expansion of Programs 4.5 and 4.6 is that the data-output acknowledge R^e communicates with only the read data interface array, and that the control-output acknowledge RC_o^e communicates with only the handshake control. Our control-data decomposition allows the control propagation reset phase and the data interface reset phase to proceed concurrently and independently of each other. The handshake control needs to check only $R^v\uparrow$ before $RC_i^e\downarrow$ because R^v is sent to the (data receiving) environment as a request and checked in both directions by R^e. R^e is, in turn, symmetrically checked by ren_D, which is completed with ren_C at ren^v. We will show that an advantage to synchronizing control and data at ren^v is that ren^v is off of the critical path.

Program 4.17 HSE: the register read handshake control (full buffer)

```
$REG\_HSEN_{read,fullbuf} \equiv$
    $*[[RC_o^e]; ren_C\uparrow; [RC_o^v \wedge RC_i^v \wedge ren^v \wedge R^v]; RC_i^e\downarrow;$
      $[\neg RC_o^e]; ren_C\downarrow; [\neg RC_i^v \wedge \neg ren^v]; RC_i^e\uparrow; [\neg RC_o^v]$
     $]$
```

Program 4.18 HSE: the register read handshake control (half buffer)

```
$REG\_HSEN_{read,halfbuf} \equiv$
    $*[[RC_o^e]; ren_C\uparrow; [RC_o^v \wedge RC_i^v \wedge ren^v \wedge R^v]; RC_i^e\downarrow;$
      $[\neg RC_o^e]; ren_C\downarrow; [\neg RC_o^v \wedge \neg RC_i^v \wedge \neg ren^v]; RC_i^e\uparrow$
     $]$
```

Program 4.19 HSE: completion tree for *ren* signals in read port

```
$REG\_CT_{ren} \equiv$
    $*[[ren_C \wedge \langle \forall b :: ren_D[b] \rangle]; ren^v\uparrow; [\neg ren_C \wedge \langle \forall b :: \neg ren_D[b] \rangle]; ren^v\downarrow]$
```

We summarize the responsibilities of each block in the remainder of this subsection by describing sequences of events that need to be enforced by the composition of the individual quadrants. We describe actions that are ordered by the 'mini-handshakes' between neighboring quadrants.

Cell array to read data interface array. The cell array communicates the register value to the interface on the $\_R$ inverted data rails when ren_D is high. $\_R$ is reset by $\neg ren_D$. Together, the handshake control and interface array must guarantee the following total order of events (for both reshufflings): $*[ren_D\uparrow; \_R\downarrow; ren_D\downarrow; \_R\uparrow]$. The cell production rule for $\_R\downarrow$ is guarded by ren_D, which enforces the first order relation. The second ordering is enforced by: $\_R\downarrow \prec R^v\uparrow \prec (RC_i^e\downarrow, R^e\downarrow) \prec ren_D\downarrow$. The third ordering is satisfied by the production rule $\neg ren_D \mapsto \_R\uparrow$. The final ordering will be enforced by: $\_R\uparrow \prec R^v\downarrow \prec R^e\uparrow \prec ren_D\uparrow$.

One crucial requirement, which places a constraint on the production rules, is that the ren_D signal which is completed into ren^v *must* be the same ren_D signal that is connected across the register cell array, *not* an amplified copy thereof, because the handshake control uses $\neg ren^v$ to guarantee that all ren_Ds have reset and therefore cut-off the $\_R\downarrow$ actions. In other words, the $ren_D\uparrow$ and $ren_D\downarrow$ actions must remain *atomic*. Otherwise, ren^v cannot guarantee that $\_R\downarrow$ actions have been cut-off before requesting the next input token with $RC_i^e\uparrow$ without an additional timing assumption. Failure to guarantee this can lead to violations of word select exclusion and idempotence.

Cell array to control propagation. There are no communication actions between the cell array and the control propagation array. Both arrays share the read word line signals $RC_i[l]$ as inputs, therefore the handshake control needs to guarantee that the input completion signal, RC_i^v, is checked before RC_i^e, i.e., that $RC_i^v\uparrow \prec RC_i^e\downarrow$ and $RC_i^v\downarrow \prec RC_i^e\uparrow$.

Read data interface array to handshake control. The data interface array communicates the completion signals R^v and ren^v to the handshake control, while the handshake control communicates RC_i^e to the interface array and the successor block. The actions of ren^v and RC_i^e are ordered by the conjunction of the following cycles:

1. $RC_i^v\uparrow \prec RC_i^e\downarrow \prec RC_i^v\downarrow \prec RC_i^e\uparrow \prec RC_i^v\uparrow$
2. $ren^v\uparrow \prec RC_i^e\downarrow \prec ren_D\downarrow \prec ren^v\downarrow \prec RC_i^e\uparrow \prec ren_D\uparrow \prec ren^v\uparrow$
3. $ren^v\uparrow \prec RC_i^e\downarrow \prec ren_C\downarrow \prec ren^v\downarrow \prec RC_i^e\uparrow \prec ren_C\uparrow \prec ren^v\uparrow$

Since ren_D directly guards the production rules for $\_R\downarrow$ and ren_C guards the firings of RC_o, the $\neg ren^v$ guard for $RC_i^e\uparrow$ guarantees that the same control token will not cause R and RC_o to fire more than once, i.e., each input token is *idempotent*.

Handshake control to control propagation. The handshake control for the read port sends ren_C to the control propagation array, which eventually responds through RC_o^v. Both buffer reshufflings impose the following ordering: $ren_C\uparrow \prec RC_o^v\uparrow \prec RC_o^e\downarrow \prec ren_C\downarrow \prec RC_o^v\downarrow \prec RC_o^e\uparrow \prec ren_C\downarrow$.

The floor decomposition can be applied to the terminal read port block in a similar fashion. The register cell array and the data interface array are identical to those of the non-terminal blocks, because the data functionality is the same. The first difference is that there is no control propagation array quadrant. The handshake control quadrant is much simplified after the output control handshake is eliminated from the HSE in Program 4.17. The resulting HSE is shown in Program 4.20.

Program 4.20 HSE: the terminal block's read handshake control

```
$REG\_HSEN_{read,last} \equiv$
    *[[$RC_i^v \wedge ren^v \wedge R^v$]; $RC_i^e\downarrow$; [$\neg RC_i^v \wedge \neg ren^v$]; $RC_i^e\uparrow$
     ]
```

We finish deriving complete production rules for the core read port in Section 4.3.

4.2.2 Decomposed Writing

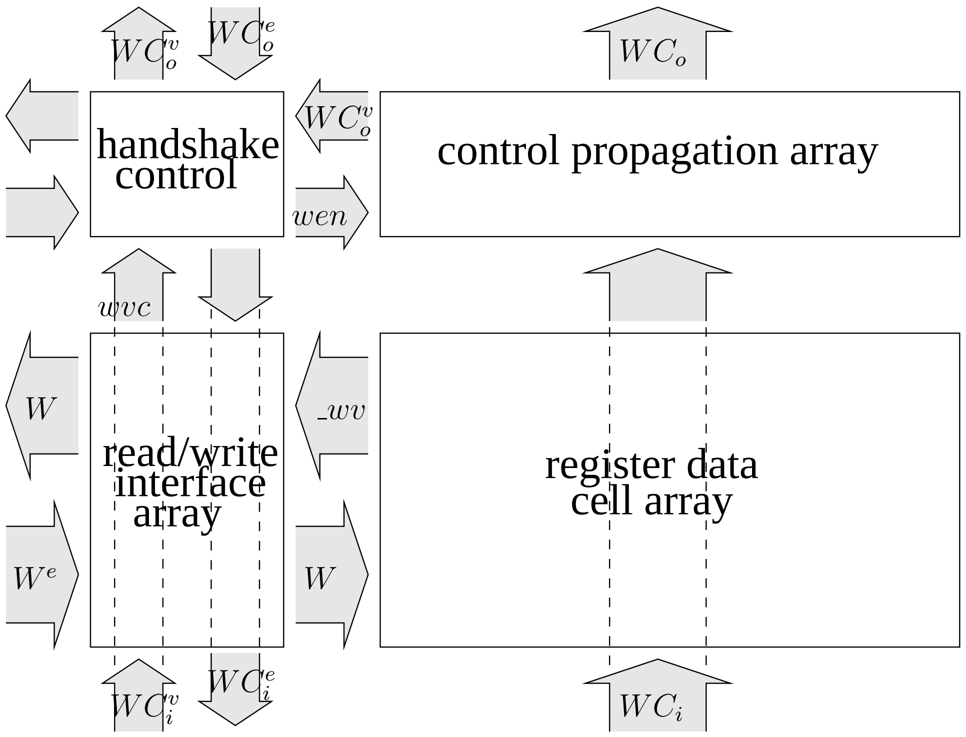


Figure 4.8: Floor decomposition of a write port block, shown with channel signals and some internal signals at component boundaries

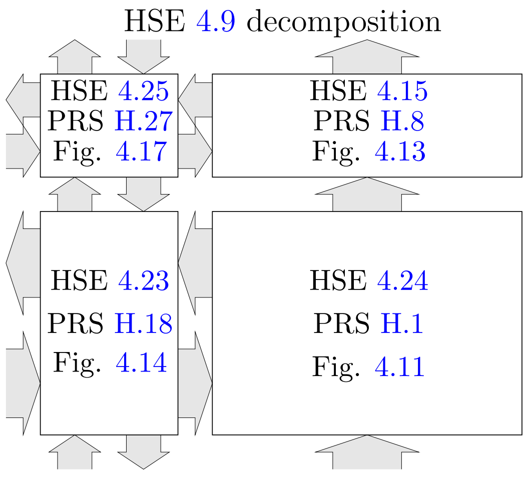


Figure 4.9: Floor decomposition of a PCEVFB write port

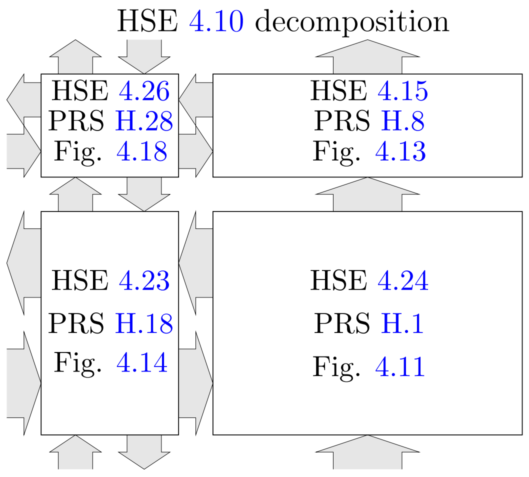


Figure 4.10: Floor decomposition of a PCEVHB write port

The core write port is floor-decomposed in the same manner as the read port. Figure 4.8 shows the relative placement of channel signals and internal nodes at the boundaries of the floor decomposition components. Figures 4.9 and 4.10 outline the steps that follow in synthesizing production rules from the full-buffer and half-buffer handshaking expansions of the core write port. First we extract the writing of the data to a register into a cell component, listed in Program 4.21. Each cell in the array takes a write word line $WC_i\,\texttt{[}l\texttt{]}$ and core data $W_i\,\texttt{[}b\texttt{]}$ as input. The $\_wv\,\texttt{[}b\texttt{]}$ signals indicate when the write to a bit cell is finished. Each $\_wv\,\texttt{[}b\texttt{]}$ is reset (high) after the input bit rails return to their neutral state. All $\_wv\,\texttt{[}b\texttt{]}$'s for each block are completed together in the completion tree in Program 4.22. Since resetting $\_wv\,\texttt{[}b\texttt{]}\!\uparrow$ is independent of the select line, we can move the reset action to the write data interface in Program 4.23. What remains in the cell is Program 4.24, which can be easily implemented with mostly NFETs. We derive the cell-writing production rules in Section 4.3.1.

wvc signals the completion across all $_wv\,\texttt{[}b\texttt{]}$s for a single block in Program 4.22. Note that wvc is not shared with the environment, so no input validity W^v is needed from environment data sender.

Program 4.21 HSE: the register write cell array component

$REG\_DATA_{write}\,\texttt{[}b,l\texttt{]} \equiv$

$\quad *\texttt{[[}WC_i\,\texttt{[}l\texttt{]} \wedge W_i\,\texttt{[}b\texttt{]]};\langle write\,\texttt{[}b,l\texttt{]}\rangle;\_wv\,\texttt{[}b\texttt{]}\!\downarrow;\texttt{[}\neg W_i\,\texttt{[}b\texttt{]]};\_wv\,\texttt{[}b\texttt{]}\!\uparrow\texttt{]}$

Program 4.22 HSE: completion tree for wvc in read port

$REG\_CT_{wvc} \equiv$

$\quad *\texttt{[[}\langle \wedge \forall b :: \neg\_wv\,\texttt{[}b\texttt{]}\rangle\texttt{]};wvc\!\uparrow;\texttt{[}\langle \wedge \forall b :: \_wv\,\texttt{[}b\texttt{]}\rangle\texttt{]};wvc\!\downarrow\texttt{]}$

Program 4.23 HSE: resetting the write validity bitline

```
$REG\_INTRFC_{write}\,[b] \equiv$
    $*[[\neg W_i\,[b]];\ \_wv\,[b]\uparrow]$
```

Program 4.24 HSE: the register write cell array component (set-only)

```
$REG\_DATA_{write}\,[b,l] \equiv$
    $*[[WC_i\,[l] \wedge W_i\,[b]];\ \langle write\,[b,l]\rangle;\ \_wv\,[b]\downarrow;\ [\_wv\,[b]]]$
```

The write control propagation array is nearly identical to that of the read control and is listed in Program 4.15. Again, the implementation of the locking scheme is isolated in the control propagation array and has no effect on any other quadrant. The write control signals are completed in an OR-tree as shown in Program 4.16. WC_o^v serves as the output validity for this control block and as a input validity to the successor block.

Program 4.25 HSE: the register write handshake control (full buffer)

```
$REG\_HSEN_{write,fullbuf} \equiv$
    $*[[WC_o^e];\ wen\uparrow;\ [WC_o^v \wedge WC_i^v \wedge wvc];\ WC_i^e\downarrow;$
      $[\neg WC_o^e];\ wen\downarrow;\ [\neg WC_i^v \wedge \neg wvc];\ WC_i^e\uparrow;\ [\neg WC_o^v]$
     $]$
```

Program 4.26 HSE: the register write handshake control (half buffer)

```
$REG\_HSEN_{write,halfbuf} \equiv$
    $*[[WC_o^e];\ wen\uparrow;\ [WC_o^v \wedge WC_i^v \wedge wvc];\ WC_i^e\downarrow;$
      $[\neg WC_o^e];\ wen\downarrow;\ [\neg WC_o^v \wedge \neg WC_i^v \wedge \neg wvc];\ WC_i^e\uparrow$
     $]$
```

Finally, the handshake controls for the write port, which communicates with the write data environment and successor and predecessor control blocks via request and acknowledge, are listed in Programs 4.25 (full-buffer version) and 4.26 (half-buffer version). Recall that WC_i^e acknowledges both the predecessor block and the data sender (W_i^e). WC_i^v is the incoming validity from the predecessor block. Again, the choice of reshuffling only affects the handshake control quadrant, and does not affect the specification of the other three quadrants.

We summarize the communication interfaces and responsibilities between the various floor-quadrants of the write port.

Cell array to write data interface. The data interface itself does little work other than reset the bit-validity $\_wv\,[b]$ and complete wvc. Since each cell in the array takes $WC_i\,[l]$ and $W_i\,[b]$ as input, the handshake control must synchronize the input control and data tokens so that each token is consumed exactly once.

From Programs 4.9 and 4.10, the second parts are implemented by the cell and data interface array. The handshake control part guarantees that WC_i cannot remain active for the duration of more than one data token W_i because the shared acknowledge WC_i^e ($\equiv W_i^e$) always waits for $[\neg WC_i \wedge \neg wvc]$. The conjunction of the following orderings guarantees the idempotence and synchronization between control and data tokens:

1. $wvc\uparrow \prec WC_i^e\downarrow \prec wvc\downarrow \prec WC_i^e\uparrow \prec wvc\uparrow$
2. $WC_i^v\uparrow \prec WC_i^e\downarrow \prec WC_i^v\downarrow \prec WC_i^e\uparrow \prec WC_i^v\uparrow$

Cell array to control propagation. The cell array and control propagation array do not communicate any signals with each other, but they share the input control WC_i. Again, the handshake control guarantees that the use of an input control token is synchronized.

Write data interface to handshake control. The data interface only communicates wvc to the handshake control, but the handshake control communicates no signal to the data interface.

Handshake control to control propagation. The handshake control and control propagation array communicate a 4-phase handshake with wen and WC_o^v. Together with the output acknowledge, WC_o^e, they enforce the ordering: $wen\uparrow \prec WC_o^v\uparrow \prec WC_o^e\downarrow \prec wen\downarrow \prec WC_o^v\downarrow \prec WC_o^e\uparrow \prec wen\uparrow$.

The terminal write port block is floor-decomposed similarly. The register cell and data interface array are identical to those of the non-terminal blocks, and there is no control propagation array. The handshake control quadrant is much simplified after the output control handshake is eliminated from the HSE in Program 4.25. The resulting HSE is shown in Program 4.20.

Program 4.27 HSE: the terminal block's write handshake control

```
$REG\_HSEN_{write,last} \equiv$
    $*[[WC_i^v \wedge wvc];\ WC_i^e\downarrow;\ [\neg WC_i^v \wedge \neg wvc];\ WC_i^e\uparrow]$
```

In this section, we have described in detail the process of partitioning actions of a handshaking expansion into a floor decomposition, which is an intermediate step in translating a process (with a chosen template handshaking expansion) into production rules. While this step is not altogether necessary for synthesis, it is used here as an aid to help the reader visualize the mapping of a process to its physical implementation. In the interest of exploring many variations of the register core, we have identified the components that are subject to change as we apply different optimizations. It is to our advantage that we can modularly modify individual quadrants without affecting the correctness of a design, as long as we preserve orderings of actions across the interfaces. We show the production rules for the read and write port in Section 4.3.

4.3 Production Rule Synthesis

Now we take each of the quadrants of the floor decompositions described in the previous section and translate them into production rules for circuits.

4.3.1 Core Register Cells

Figure 4.11 shows our QDI read-write register cell. The production rules for this cell are identical to that used in the MiniMIPS [31]. None of the variations in the remainder of this thesis modify the register cell. The production rules are also listed in Program H.1.

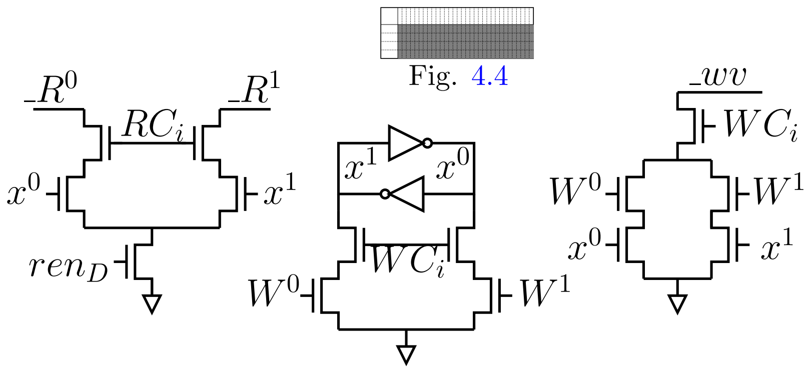


Figure 4.11: QDI Register core cell. Only one read and one write port are shown.

Storage. Each register cell stores one bit of data. The bit is stored internally as a pair of cross-coupled inverters. The only PFETs in the register cell are the pull-ups of the coupled nodes, which we denote as x^0 and x^1. Since we do not care how the values are initialized on power-up, we omit reset circuitry.

Reading. The read output for each port of the cell is the active-low dual-rail channel $\_R$. ren acts as the bit line enable, and RC_i is the register line select. The production rules for $\_R^0\downarrow$ and $\_R^1\downarrow$ are exclusively NFETs. Each read port contributes one register (word) select wire-track and three bit wire-tracks.

Writing. The write input for each port to the cell is the active-high dual-rail channel W, and the register line select WC_i. The QDI register cell is larger than a typical synchronous register cell because of both the dual-rail encoding and the additional circuitry to detect write completion, $\_wv$. With the exception of the cross-coupled inverters, the production rules for writing to x^0, x^1, and $\_wv$ all use NFETs. Each write port contributes one register (word) select wire-track and three bit wire-tracks.

Register Zero. The hard-wired register zero requires no storage, and only requires a circuit for pulling down the $\_R^0$ rail for reading. A write to the zero register is non-modifying, and just immediately returns with $\_wv\downarrow$. The production rules for the zero register cell are listed in Program H.2.

While the area per port of our QDI register cell is larger than that of most traditional synchronous register cells, all register cells scale linearly in both dimensions by the number of ports, hence quadratically overall. The area of heavily ported register cells is dominated by wires, not gates, therefore the traditional area models for synchronous register files also applies here, but with a different number of wire tracks per port. For comparison, the register cell presented by Tseng [50] is most similar to ours because both read and write data lines are dual-rail. Their cell has fewer transistors because they use a pass-gate transformation to convert the write bit lines to active-low. Zyuban and Kogge's model for register file energy complexity models cell ports as having a dual-rail write and monorail read line [57,58]. Rixner, Dally, et al. model register cells with the minimal single transistor and single bit line per (unified read-and-write) port [42]. The area, energy, and delay models for register arrays mentioned in Section 1.1.2 apply the same way to our QDI register core, but with more wire tracks per port.

Beyond the scope of this thesis, but worth exploring, are many possibilities for non-QDI register files that can take advantage of various (smaller) cells and analog circuit techniques for reducing energy by carefully adding timing assumptions.

4.3.2 Control Propagation

The read and write control propagation for the base design core are unconditional and independent of data. We translate the generalized lock condition from Programs D.4 and D.5 into additional guards on the production rules for $\_RC_o\downarrow$ and $\_WC_o\downarrow$. We observe that these inverted outputs themselves can be used as the locking variables rx and wx (in the inverted sense)! This makes the production rules for locking very convenient, because we do not need to introduce any additional nodes. The locking guards guarantee mutual exclusion between output controls RC_o and WC_o and between multiple WC_os. Read-write and write-write exclusion on the controls guarantees exclusive access to the x^0 and x^1 shared internal state variables in the register cells. A nice result of this floor decomposition is that the implementation of the locking scheme only affects the control propagation array; the handshake control quadrant is entirely independent of the locking scheme.

Figures 4.12 and 4.13 show the precharge-domino circuitry for pipeline-locked unconditional control propagation for 2-read, 2-write ported registers. The PRSs are also listed in Programs H.7 and H.8. It is clear from the PRS that the vertical latency per stage of control propagation is only two gate delays through the precharge stack.

The validity signals RC_o^v and WC_o^v are computed using OR-trees. For a bank of 32 registers, we use a 4-level tree of NOR2-NAND4-NOR2-NAND2, starting with RC_o[0..31] and WC_o[0..31].[2]

[2] A non-QDI completion tree could start completing across $\_RC_o$ and $\_WC_o$, as long as the output driving inverter is faster than the completion tree, i.e., the output of the completion trees guarantee, by timing assumption, the validity of the control rails.

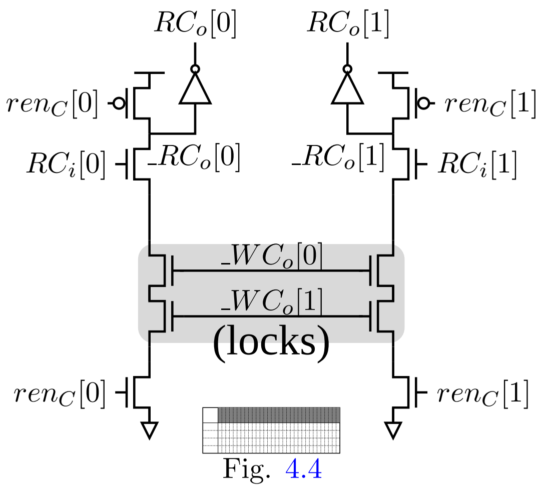


Figure 4.12: Pipeline-locked read control propagation, shown for two ports.

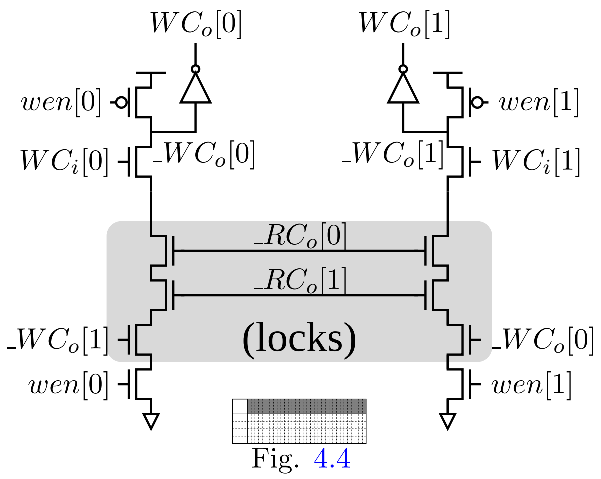


Figure 4.13: Pipeline-locked write control propagation, shown for two ports.

4.3.3 Data Interface Cell

The read and write data interface cell is illustrated in Figure 4.14, and the production rules are also listed in Program H.18.

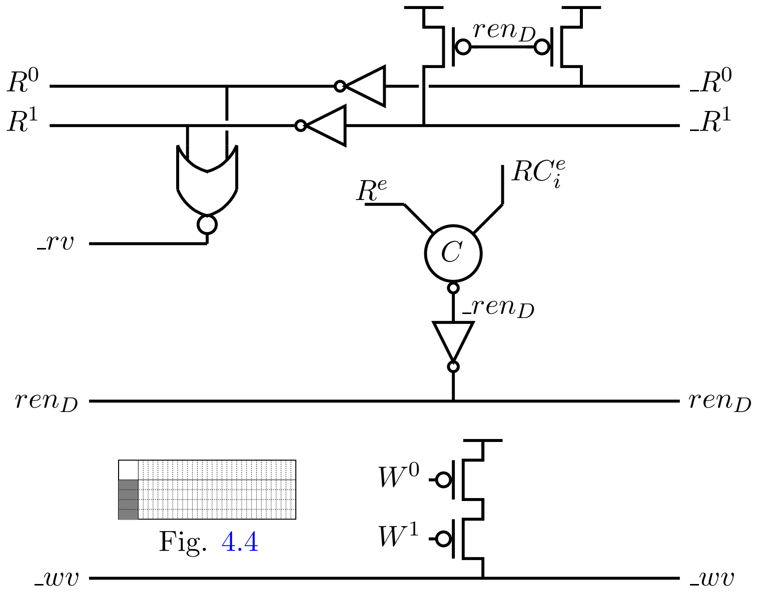


Figure 4.14: Read and write data interface for a single port of a bit line (resets are not shown)

Reading. The $\neg ren_D$ reset of $\_R\uparrow$ complements the pull-down inside the register cell. Completion detection on channel R begins after the output inverters with a 2-input NOR gate.[3] The C-element combines the acknowledges R^e and RC_i^e which guarantees correct ordering of the full-buffer and half-buffer handshaking expansions. ren_D, which has a large load across the register cell array, is driven by a high-gain inverter.

Writing. The write interface production rule is a simple pull-up on $\_wv$ when the input data W is neutral. $\_wv$ is then checked by the completion tree for wvc.

One can visualize the relative placement of a row of the register cell array to the right of the interface cell, and the block-wide completion trees for $\_wv$, $\_rv$, and ren_D to the left of the interface cell array, as shown in Figure 4.4. The completion trees are just two- or three-level trees of C-elements.

4.3.4 Handshake Control

For our core base designs, we consider the full-buffer and half-buffer reshufflings of the read and write ports. We show the production rules for the handshake controls and point out the differences between the two reshufflings.

Full-buffered reading. (Program H.20, Figure 4.15)
We derive stable production rules from the partial handshaking expansion given in Program 4.17. The actions for ren_C are symmetrically guarded by the input and output acknowledges RC_i^e and RC_o^e. ren_C is then checked by the control-data enable completion, ren^v. ren^v and RC_i^v symmetrically guard RC_i^e. The full-buffered reshuffling allows $RC_i^e\uparrow$ (request for next input) before the output is reset, $\neg RC_o^v$. $\neg ren^v$ guarantees that ren_C and all ren_D have reset so that no RC_o and R can fire again until the output receivers have reset their acknowledges R^e and RC_o^e. $\neg RC_o^v$ is checked symmetrically by the successor's acknowledge, RC_o^e, and R^v is checked symmetrically by the data receiving environment, so there is no need to check them again locally.

Half-buffered reading. (Program H.21, Figure 4.16)
Stable production rules for the half-buffer reshuffling are similarly derived from the partial handshaking expansion in Program 4.18. Recall that we have chosen to keep the data communication full-buffered so we need not wait for data neutrality $\neg R^v$ before requesting the next control input. The only significant difference from the full-buffer is that the control output neutrality $\neg RC_o^v$ is checked before requesting the next input control token with $RC_i^e\uparrow$, which is a difference of a single PFET.

Terminal Reading Block. (Program H.26)
The production rules for the terminal block's read port are trivial from the HSE in Program 4.20.

Full-buffered writing. (Program H.27, Figure 4.17)

[3] A non-QDI detection could start completing on $\_R$ using a NAND gate with the conservative timing assumption that the output inverters reset low faster than the completion tree resetting, i.e., that output neutrality "guarantees" the data rails have reset.

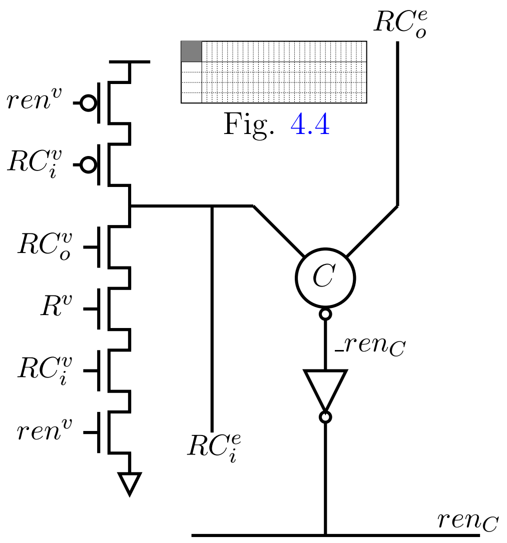


Figure 4.15: Read handshake control for full-buffered unconditional control propagation. (resets are not shown)

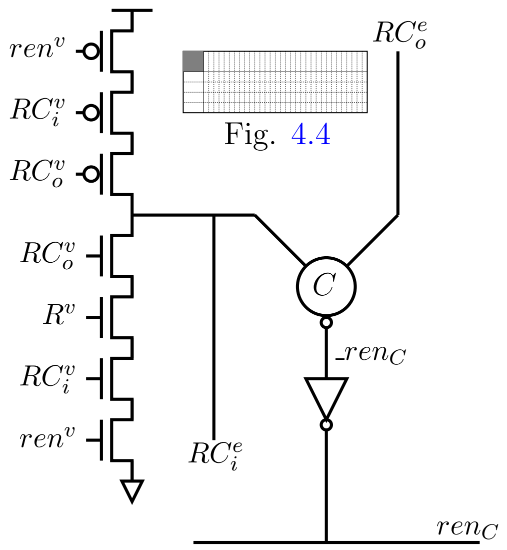


Figure 4.16: Read handshake control for half-buffered unconditional control propagation. (resets are not shown)

We derive stable production rules for the full-buffer write port handshake control from the expansion in Program 4.25. The firings of *wen* are symmetrically guarded by the input and output acknowledges WC_i^e and WC_o^e. Input data and control validity *wvc* and WC_i^v are both checked symmetrically before WC_i^e fires. Since $\neg WC_o^v$ is not checked before requesting the next input token with $WC_i^e\uparrow$, we need to check *wen* symmetrically before WC_i^e to guarantee that each input token is used and acknowledged exactly once. Otherwise, if *wen* remains high (reset low too slow) during the reset phase, another input token may come along and cause another $WC_o\uparrow$ to fire (and possibly WC_o^v) which is a violation of exclusion on the use of the shared data channel W.

Half-buffered writing. (Program H.28, Figure 4.18)
We derive stable production rules for the half-buffer write port handshake control from the expansion in Program 4.26. The first difference from the full-buffer reshuffling is that $\neg WC_o^v$ is checked before requesting the next input with $WC_i^e\uparrow$. The symmetric guard of WC_o^v enforces the ordering $wen\uparrow \prec WC_o^v\uparrow \prec WC_o^e\downarrow \prec wen\downarrow \prec WC_o^v\downarrow \prec WC_o^e\uparrow \prec wen\uparrow$. Therefore, *wen* need not guard WC_i^e to guarantee idempotence and exclusion.

Terminal Reading Block. (Program H.26)
The production rules for the terminal block's write port are trivial from the HSE in Program 4.27.

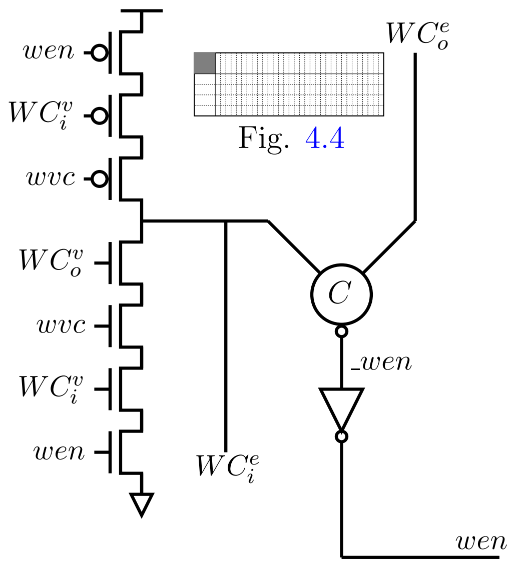


Figure 4.17: Write handshake control for full-buffered unconditional control propagation. (resets are not shown)

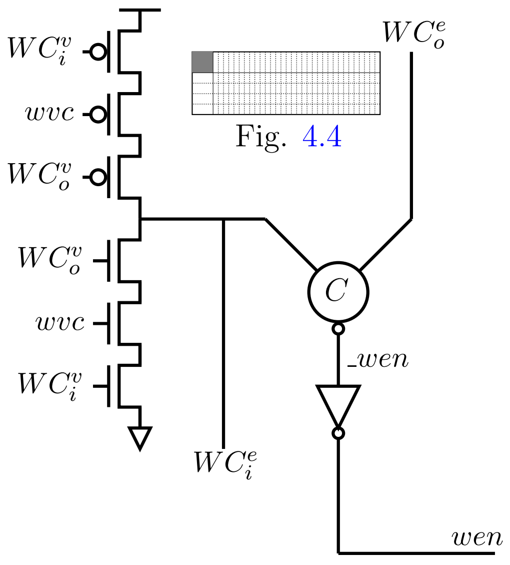


Figure 4.18: Write handshake control for half-buffered unconditional control propagation. (resets are not shown)

4.3.5 Circuit Variations and Optimizations

The production rules we have just described are not exactly the ones we implemented and for which simulation results are presented. There is a class of circuit optimizations we used to modify the completion of validity signals, which reduces the transistor stacks on RC_i^e and WC_i^e without increasing the number of transitions per cycle. For fairness of comparison of the actual implementations, we applied these transformations uniformly to all versions of the read and write ports circuits presented in this thesis. However, we list the original derived production rules in Appendix H because they correspond exactly to the partial handshaking expansions from the floor decompositions of the read and write ports, and hence, are easier to understand. Knowledge of these circuit optimizations is not crucial to understanding this thesis. We describe the circuit optimizations in full detail in the technical report [11].

4.4 Banking

Before we present the results of the base design register cores, we describe the impact of banking the register core on the *BYPASS* and *CONTROL* components of the register file. As memory structures such as register files, SRAMs, and DRAMs increase in the number of bits and words, access times slow down due to increased capacitive load on shared bit lines and word select lines. In Chapter 3, we alleviated the load on shared word lines with vertical pipelining. The traditional solution for reducing load on bit lines is banking, splitting an array into sub-arrays.

In Chapter 9, we describe a different type of partitioning that has non-uniform access times.

4.4.1 Related Work

Many modern SRAMs and DRAMs are heavily banked to support fast access times. Banking also enables rapid concurrent access to different banks, which can be leveraged by non-conflicting sequential memory access patterns, as is often used with signal processing applications [21]. Banking can offer excellent average-case performance and has been demonstrated in an asynchronous DRAM design [10].

As superscalarmicroprocessors exploit greater and greater instruction-level parallelism (ILP), the number of registers required to support in-flight instructions increases, as does the number of ports required to support wider issue [12, 18, 52]. From Section 1.1.2, we have seen models of how performance and energy of register files scale with size, and how larger register files can severely limit cycle times. Modern processors bank their register files to make sure their access times meet critical path timing requirements [20, 37, 45, 55]. Banking register files provides an alternative to adding more read and write ports to the register cells, which helps especially when accesses to different banks are (statically or dynamically) scheduled together.

While our register control allows concurrent accesses to different ports, it cannot issue simultaneous accesses to different banks of the same port. This is only a limitation of our control specification, which is guided by the number of buses on the datapath. Other architectures may be able to take advantage of multiple banks and multiple ports by scheduling (statically or dynamically) concurrent read and write accesses to different banks of each port, and multiply the number of effective ports when banks do not conflict. Nonetheless, even for a single-issue in-order processor, banking still offers an improvement in performance and reduction in energy in the core.

4.4.2 Core Banking

Aside from speeding up access times, an additional motivation for banking our register core is that the read and write cycle time of a block of 4 bits by 32 registers is limited by the control handshake cycle, which includes the time of setting and resetting through the control propagation arrays' completion trees. Recall that for a bank of 32 registers, we completed the validity in a four-stage OR-tree.

When we bank the *CORE* process, all we do is duplicate each core process, and halve the number of registers in each bank. Figure 4.20 illustrates the schematic for dual-banked register core read and write operation. The only change that this may introduce is that the sense of the control propagation completion signals RC^v and WC^v may become inverted to active-low RC^n or WC^n signals.[4]

[4] Production rules for the active-low validity reshufflings (PCENFB and PCENHB)

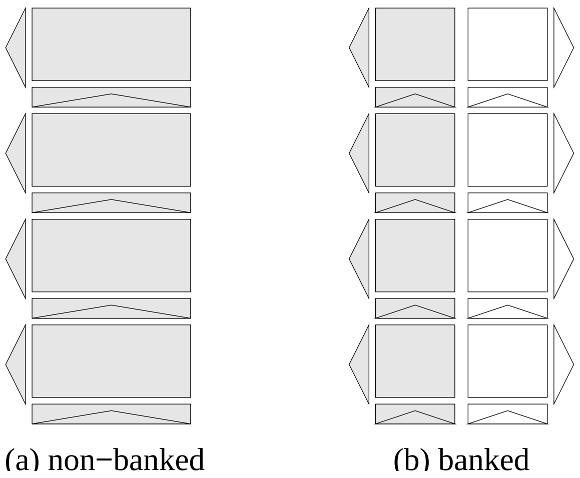


Figure 4.19: Banking the register file is a common method for reducing access energy and delay by reducing the load on bit lines

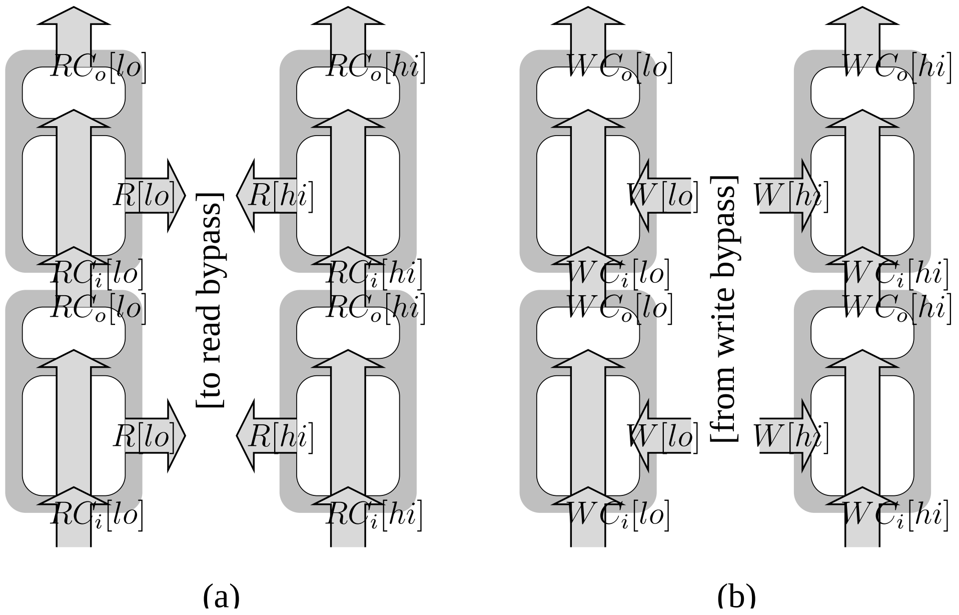


Figure 4.20: Block diagram of vertically pipelined, banked read and write processes. For the 32 register architecture, the *lo* bank contains registers 0 through 15, and the *hi* bank contains registers 16 through 31.

For our study, we divide the register core into two symmetric banks. In general, one could divide the register file into any number of banks, at the cost of adding the hardware for control (the handshake control and data interface array) for each bank. The speedup gained by banking diminishes as the number of banks increase

for all handshake control circuits appear in the Appendix of the technical report [11], but are omitted from this thesis. For this thesis, we just add inverters to force the shared validity signals to be active-high.

and size of each bank decreases. Let us not forget that each bank that we introduce adds a set of channels, which needs to be multiplexed or de-multiplexed by the bypass interface to the operand buses. Now we need to modify the read and write bypasses to accommodate the channels for each bank.

4.4.3 Bypass Banking

Figure 4.21 illustrates the new decomposition of the *BYPASS* for a dual-banked register file. For comparison, the original *BYPASS* for the unbanked register core is shown in Figure 2.6. The bypass forwarding channels *BPZX*[0..1] and *BPZY*[0..1] remain the same as before, but the number of channels between the core and bypasses have doubled. The CHP modifications that are introduced as a result are very simple.

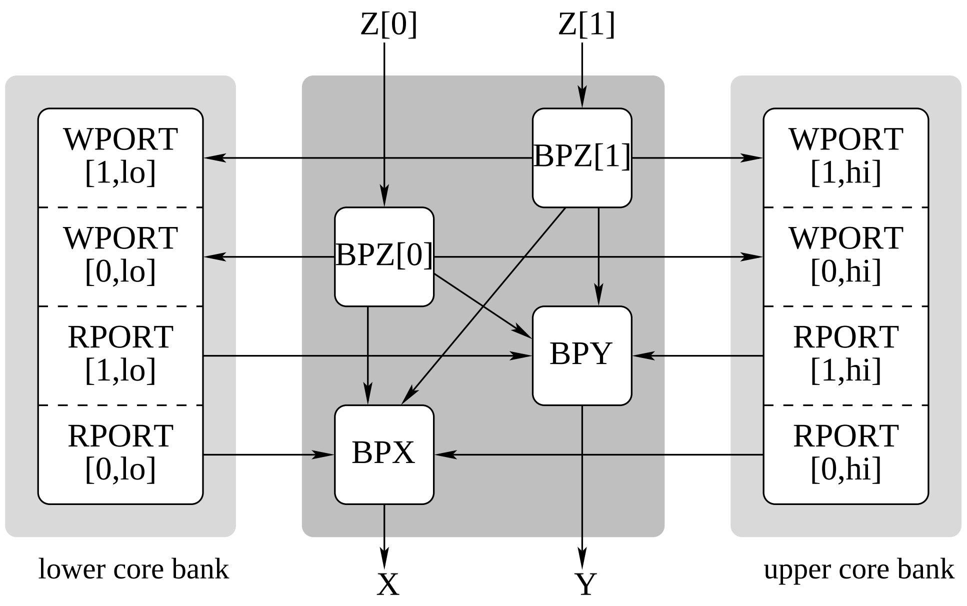


Figure 4.21: Bypass decomposition for dual-banked register core. Control channels are not shown.

Recall that the read bypass is just a controlled merge process from Program B.2. Each bank adds another channel from which a source operand may be read, so naturally, we just add one more case to the merge, which results in Program B.9. *BPX* and *BPY* now communicate *core*[*lo*] or *core*[*hi*] to distinguish between the upper and lower banks of the respective read ports. A heavily banked design might use a multi-stage merge if a single-stage N-way merge becomes too slow.

The original writeback-bypass, Program B.1, is a controlled conditional copy. We modify the case that conditionally writes back to the core to *split* the data to one of the banks of each write port. The resulting CHP is Program B.10. Channels *BPWB*[0..1] now communicate one of three values: *lo* or *hi* to copy a value to a bank of a write port, or **false** to discard a value.

Throughout the remainder of the thesis, we will show that banking the *CORE* and *BYPASS* can be easily adopted in conjunction with other transformations. The resulting transformed bypasses still fit into well-known function templates, thus, QDI production rule synthesis is straightforward.

4.4.4 Control Modifications

The last step is to direct bank accesses in the *CONTROL*. We have essentially moved part of the register index demuxing into the control process. We change the communication actions on *BPX* and *BPY* from Program C.1 to be conditional on the value of the respective indices rs and rt, which results in CHP Program C.4.

For the writeback bypass control, after we change the *BPWB* communication to be conditional on the bank index, the result is Program C.5.

Obviously, if we encode the index channels (range 32) in binary, we can use a dual rail to distinguish between banks; comparator logic is unnecessary in the bypass controls. A result of decoding the bank outside of the core is that the each core bank's demux will be simplified and faster.

Again this slight modification in the *CONTROL* is compatible with the transformations introduced in the later Chapters. Synthesis of QDI production rules follows from straightforward application of known function templates.

4.5 Results

Here we present the performance and energy results for our base design register core, a 32-bit x 32-word bank pipelined vertically into 4-bit blocks, and for the 16-word, banked version, both laid out in TSMC .18μm technology. We used the same layout in both designs without resizing transistors to equalize path delays. The layout dimensions of the various components, labeled in Figure 4.4, are listed in Table 4.1. The height of the base design's control propagation cell is $y_{cp_{std}}$.

Table 4.1: Layout component dimensions, corresponding to Figures 4.4 and 9.2.

dim.	λ	λ/x_{cell}	dim.	λ	λ/y_{cell}
x_{cell}	65	1.00	y_{cell}	210	1.00
x_{pi}	268	4.12	$y_{cp_{std}}$	380	1.81
x_{vt}	109	1.68	$y_{cp_{WAD}}$	401	1.91
x_{ni}	240	3.69	y_{ht}	140	0.67

For comparison, the Caltech MiniMIPS was not banked, used a block granularity of 8-bits, and was designed in HP's .6μm CMOS process from MOSIS [31]. From `spice` simulations, the MiniMIPS was anticipated to operate at 280 MHz and 4 W at 3.3 V, and was projected to operate at 560 MHz and 2.4 W with HP's .18μm process at 1.8 V [32].

We simulated the core circuits for 25 ns using a variant of `spice`.[5] Since we measure energy by linearly interpolating the average rate at which charge flows from the power supply, there will be some miniscule numerical error. The timing measurements have been validated for the targeted technology. The number of transitions per cycle, measured with `prsim` is the number of signal inversions in a control handshake assuming unit gate delays, and is only meant to give a rough estimate of performance. The frequency (or throughput) is simply the reciprocal of the cycle time. The energy we report in all tables is the amount of energy dissipated *per iteration per block*. It is important to note that energy reported for the banked designs only includes the energy consumed by a single bank, and does not include the static energy consumed by the other bank.

Another important metric for performance is the *latency* of port operations. For read ports, the read latency is the measured delay from bit line enable (ren_D) and word line select (RC_i) to data output ($R_o\uparrow$). Shorter read latency allows functional units to receive inputs earlier and produce outputs earlier (especially in asynchronous systems) and also reduces the branch mispredict penalty. Write latency is a delay that matters only to asynchronous write ports that use a write validity signal to detect write completion (as opposed to using delay assumptions). Write latency is measured as the delay from write bit line (W) and write word line (WC_i) to write validity ($\_wv\downarrow$), which depends on whether or not the internal cross-coupled inverters are toggled. Since we simulate maximum write switching, the write latencies we report include the toggle-time and the time for the validity signal.

In addition to performance and energy, we also compute the voltage-invariant metric $E\tau^2$, which quantifies energy efficiency [49]. A system with a lower $E\tau^2$ is superior in performance compared to one with higher $E\tau^2$ when operating at equal energies by voltage-scaling, and is also lower in energy when throughputs are equalized by voltage-scaling.

We expect the most significant speedup to come from the reduction in load on the shared bit lines $\_R$, ren_D, W, and $\_wv$, which were among the slowest observed critical transitions in the non-banked designs. The control completion trees for RC_o^v and WC_o^v are implemented as three-stage, 16-input OR-tree with a fourth stage inverter to correct the sense of the validity signal, which has the same depth as the four-stage, 32-input OR-tree, so the number of inverter transitions per cycle remains the same. However, the reduced path effort will result in slightly reduced delay.

Since the majority of energy per block is consumed by the data components of the read and write ports, halving the number of sharers on all bit lines (by banking) results in a significant reduction in dynamic energy based on reduced capacitance, and also reduces the substrate leakage current in the NMOS-dominant register cell

[5] The absolute energies reported by our simulator have not been validated and are in fact much higher than the expected energies for this technology, however the relative energies, which are more important to this thesis, are valid.

array, and hence, reduce static power dissipation.

4.5.1 Reading

Table 4.2: Read-access performance and energy comparisons for the base design register file, for a block size of 4 bits x 32 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	22	1.953	512.2	0.323	26.90	102.5
full	20	1.862	537.0		26.59	92.2

Table 4.3: Read-access performance and energy comparisons for a register bank with a block size of 4 bits x 16 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	22	1.821	549.1	0.222	15.92	52.8
full	20	1.698	588.8		15.78	45.5

Since the read port is dual-railed and hence symmetric, the value being read has no impact on the cycle time and energy. In analog simulation, we allow the internal cross-coupled bits to reset randomly by metastability. Tables 4.2 and 4.3 list the simulation results for the full-buffer and half-buffer reshufflings of the core read port. These results also appear in Table J.2 for comparison with the other read port variations presented throughout the thesis. Table J.4 compares the performance and energy of half-buffered and full-buffered read ports across the entire design space. The register cell and interface arrays are same for full and half-buffers, hence, the read latencies are the same.

Comparing reshufflings: unbanked, 32 registers. The full-buffer reshuffling is only 4.9% faster than the half-buffer version and consumes only 1.1% less energy per iteration than the half-buffer version. Overall, the full-buffer read port is 11.2% more energy-efficient than the half-buffer read port.

Comparing reshufflings: banked, 16 registers. The full-buffer reshuffling is only 7.2% faster than the half-buffer version and consumes only 0.9% less energy per iteration than the half-buffer version. Overall, the full-buffer read port is 16.0% more energy-efficient than the half-buffer read port.

Comparing bank sizes: half-buffer reshuffling. For the half-buffer reshuffling, reducing the bank size from 32 to 16 results in a 7.2% speedup in cycle time, 40.8% reduction in energy per cycle per block, which amounts to a 94.2% improvement in energy efficiency.

Comparing bank sizes: full-buffer reshuffling. For the full-buffer reshuffling, reducing the bank size from 32 to 16 results in a 9.6% speedup in cycle time, 40.7% reduction in energy per cycle per block, which amounts to a 102.7% improvement in energy efficiency.

The most significant improvement in performance is the reduced read latency, which is 0.686 of the unbanked design's read latency, a reduction of about 100 ps. For larger, and more heavily-ported register banks, the benefit of banking is expected to increase dramatically.

4.5.2 Writing

Table 4.4: Write-access performance and energy comparisons for the base design register file, for a block size of 4 bits x 32 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	22	2.488	402.0	0.528	27.81	172.1
full	20	2.444	409.2		27.45	163.9

Table 4.5: Write-access performance and energy comparisons for a register bank with a block size of 4 bits x 16 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	22	2.179	458.9	0.417	11.23	53.3
full	20	2.118	472.1		11.30	50.7

In simulating the write port (both digitally and analog), we wrote alternating ones-complement values to the core. Recall that a bit-toggling write to a register cell takes two more transitions than a non-toggling write. Since the write-validity signals are all checked through a completion tree, at least one bit-flip in a block is required to achieve the reported cycle times, which is reasonably probable. However, the energies reported for writing are worst-case figures because energy depends on the writing activity factor.

Tables 4.4 and 4.5 list the simulation results for the full-buffer and half-buffer reshufflings of the core write port. These results also appear in Table J.12 for comparison with the other write port variations. Table J.15 compares the performance and energy of half-buffered and full-buffered write ports across the entire design space. The register cell and interface arrays are same for full and half-buffers, hence, the write latencies are the same.

Comparing reshufflings: unbanked, 32 registers. The full-buffer reshuffling is only 1.8% faster than the half-buffer version and consumes only 1.3% less energy per iteration than the half-buffer version. Overall, the full-buffer read port is 5.0% more energy-efficient than the half-buffer write port.

Comparing reshufflings: banked, 16 registers. The full-buffer reshuffling is only 2.9% faster than the half-buffer version and consumes only −0.6% less energy per iteration than the half-buffer version. Overall, the full-buffer read port is 5.2% more energy-efficient than the half-buffer write port.

Comparing bank sizes: half-buffer reshuffling. For the half-buffer reshuffling, reducing the bank size to 16 results in a 14.2% speedup in cycle time, 59.6% reduction in energy per cycle per block, which amounts to a 222.6% improvement in energy efficiency.

Comparing bank sizes: full-buffer reshuffling. For the full-buffer reshuffling, reducing the bank size from 32 to 16 results in a 15.4% speedup in cycle time, 58.8% reduction in energy per cycle per block, which amounts to a 223.4% improvement in energy efficiency.

The write latency of the banked write port is 0.686 of the unbanked write port's write latency, a reduction of about 110 ps. For larger, and more heavily-ported register banks, the benefit of banking is expected to increase dramatically.

4.6 Summary

In this chapter, we have worked through a step-by-step synthesis of the read and write port circuits for a pipelined register file. The transformations presented in the remainder of the thesis make use of the floor decompositions in this chapter by introducing minor modifications in very few components. We have presented simulation results for the non-banked and banked designs of the register core. Banking is clearly beneficial to improving performance and reducing energy, as long as the resulting modifications in the *CONTROL* and *BYPASS* are not limited by the interconnect complexity that arises from the increased number of channels.

The circuits derived in the remainder of the thesis will be presented in less detail because they follow the same principles we have used in this chapter. The results from this chapter will serve as the baseline for comparisons with other transformations and optimizations presented throughout the thesis.

Chapter 5

Width Adaptivity

In this chapter, we encode the numbers communicated on the datapath and stored in the register file using a width-adaptive representation. This change is motivated by the observation that numbers in a CPU core require on average far fewer bits to represent than the full-width of the datapath. We can leverage this fact to reduce the amount of switching activity (and hence energy) on a CPU datapath by compressing the representation of leading zeros and ones on the datapath with a *width-adaptive datapath* (WAD) representation [25].

The high-level CHP program transformations we have used were independent of the numerical encoding in the datapath. When we vertically pipelined the register core and bypass in Chapter 3, we exposed the full-width of the datapath in defining the size and number of pipelined blocks. The underlying binary representation was exposed only when we derived the production rules in Chapter 4. The MiniMIPS register core and bypass were designed with the exact same transformations and the traditional full-width binary representation [31]. We now transform the register core and bypass processes from the full-width binary into the width-adaptive binary representation.

5.1 Related Work

Numerical compression on the datapath is an old concept, however, the use of width adaptivity in asynchronous architectures was first presented by Manohar [25]. Analogous studies in the synchronous domain include clock-gating as a means of suppressing switching activity on the datapath [2, 3], and byte-serial, byte-semi-parallel, byte-parallel implementations, which leverage synchronous vertical pipelining [4]. In width-adaptive MIPS studies, datapath switching activities were reduced by 2/3 [25], and among other similar studies in the synchronous 32-bit architectures, switching activity and energy savings range from 30 to 80%. With wider datapath architectures, such as the 64-bit Alpha 21264, one can expect even greater reduction in datapath activity. The primary disadvantage of the synchronous implementations is that control is significantly complicated with the addition of bypassing and forwarding paths. We show that width adaptivity in

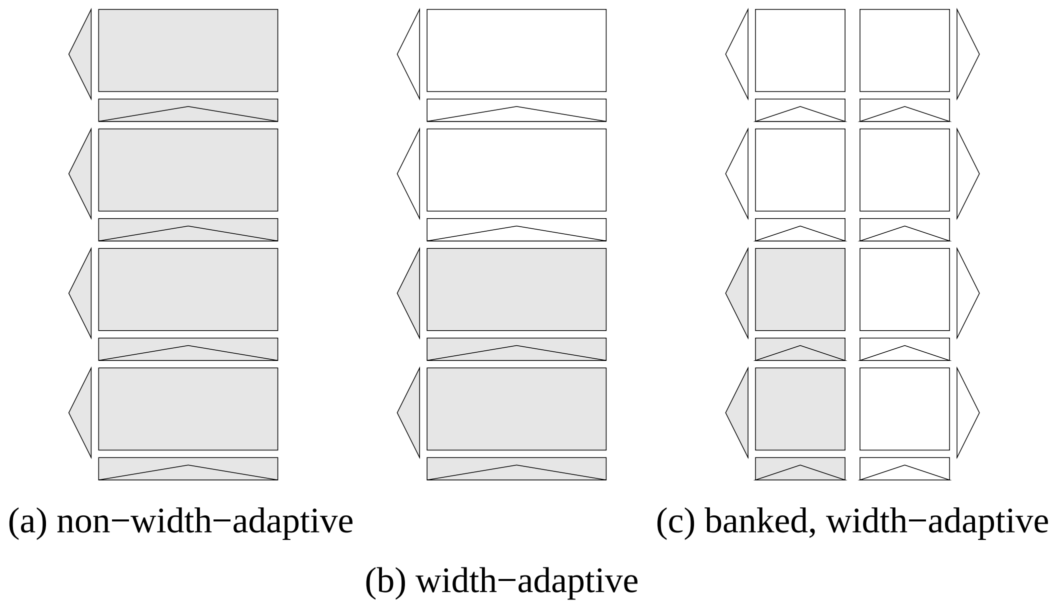


Figure 5.1: Switching activity in a) a non-width adaptive register file, b) a width-adaptive register file, and c) a banked width-adaptive register file.

our asynchronous pipeline is entirely transparent and thus requires no change to the non-width-adaptive register control.

Vertical pipelining, as described in Chapter 3, is conducive to width adaptivity implementations in asynchronous designs, because the pipeline stages delineate natural boundaries at which numbers may be terminated by compression.[1] Each block of data is extended by an additional delimiter bit to encode where the number terminates. A smaller block size gives finer granularity for terminating compressible numbers along with a shorter cycle time, but incurs a greater energy overhead cost in storing delimiter bits, propagating control, and an increased total block latency across a full-width number. A tradeoff study between WAD granularity and energy is presented by Manohar [25]. Although each vertical pipeline stage is an opportunity to encode a block width-adaptively, one may select *any* subset of pipeline stages to transform into WAD. For this thesis, we restrict our design space to the same four-bit block granularity inherited from vertical pipelining, and uniformly transform all pipeline stages using WAD.

Compressible numbers may also be arbitrarily expanded by storing and communicating higher significant blocks with the understood bits, which gives them redundant representations. In the course of manipulating integers through functional units, compressible integers may become expanded, which accounts for sub-optimal energy savings. Manohar proposed and compared several re-compression schemes to narrow the gap from optimal energy savings [25].

[1] Unpipelined width-adaptive functional units (called WAD-aligned) are described in the WAD paper [25], however, we omit them from our register file study.

5.2 WAD Encoding

A WAD number's width is encoded in its physical representation. Higher bits beyond the delimiter may be interpreted either normally or as leading 0's or 1's, depending on the value of the delimiter. WAD datapaths and functional units save considerable energy by suppressing switching activity of higher significant bits when they are *understood* without communication. Table 5.1 summarizes the encoding of the delimiter bit with the MSB for a WAD block. Figure 5.2 illustrates a few examples of width-adaptive representations of integers.

Table 5.1: The encoding of width-adaptive datapath (WAD) blocks

delim. bit	MSB	next block	control
0	0	normal	propagate
0	1	normal	propagate
1	0	$\overline{0}$	terminate
1	1	$\overline{1}$	terminate

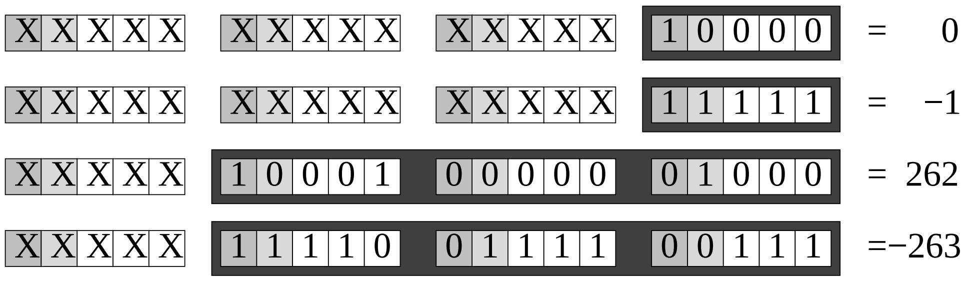

X X X X X X X X X X X X X X X 1 0 0 0 0 = 0
X X X X X X X X X X X X X X X 1 1 1 1 1 = −1
X X X X X 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 = 262
X X X X X 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 =−263

Figure 5.2: Examples of width-adaptive representation of integers. The delimiter bits are darkly shaded, and the MSBs are lightly shaded. X's represent 'don't cares.' Only darkly bordered bits are communicated.

5.3 CHP Transformations

Changing from the standard binary representation to a WAD representation does not affect the *CONTROL* processes; only the *BYPASS* and *CORE* need to be adapted. We begin with the vertically pipelined processes for the bypass and core from Chapter 3, and modify the control propagation actions to become conditional. We introduce two evaluation conditions in the guards of the new CHP programs: $p(\ldots)$ represents the propagation condition, where the delimiter bit of a data block is 0, and $t(\ldots)$ is the termination condition, where the delimiter bit is 1. Recall that the CHP template for a non-WAD pipeline stage with no shared variables was listed as Program 3.3. With only local variables, FIFO operation suffices to preserve the semantics of the unpipelined program, thus the receive actions on the control channels may precede the corresponding send actions. We can write the template for the WAD transformation as Program 5.1, in which control propagation is conditional. For pipelines with shared variables, we use locks to preserve the original semantic orderings. Applying width adaptivity, the template Program 3.4 transforms into Program 5.2. The *lock* action and *unlocked* condition maintain the same meanings as in the non-WAD pipelines.

Program 5.1 CHP: template for a width-adaptive vertical pipeline, with independent actions

```
$*[(C_i[1]?c[1],\ldots,C_i[j]?c[j]);$
  $\langle \textit{independent partial width actions } 1\ldots j\rangle;$
  $[p(\ldots)\longrightarrow(C_o[1]!c[1],\ldots,C_o[j]!c[j])$
  $[\!]\, t(\ldots)\longrightarrow \textbf{skip}$
  $]$
 $]$
```

Program 5.2 CHP: template for a width-adaptive vertically pipeline, with locking

```
$*[((\textit{unlocked}(1)\wedge c[1] := \overline{C_i[1]}),\ldots,(\textit{unlocked}(j)\wedge c[j] := \overline{C_i[j]}));$
  $\langle \textit{partial width actions } 1\ldots j\rangle;$
  $[p(\ldots)\longrightarrow((\textit{lock}(1);\,C_o[1]!c[1]),\ldots,(\textit{lock}(j);\,C_o[j]!c[j]))$
  $[\!]\, t(\ldots)\longrightarrow \textbf{skip}$
  $];$
  $(C_i[1]?,\ldots,C_i[j]?);$
  $\textit{unlock}(1\ldots j)$
 $]$
```

5.3.1 Bypass

The transformation from a non-WAD to WAD pipelined bypass is relatively simple at the CHP level. Since the *BYPASS* uses no shared variables, we simply apply template Program 5.1 to the non-WAD bypass read and writeback processes, Programs B.3 and B.4, which results in Programs B.5 and B.6.

The handshaking expansions for the WAD bypass processes are straightforward from applying any QDI handshake template, thus we omit them (and their production rules) from this thesis.

5.3.2 Core

Since the core uses locking to protect the shared variables, we use template Program 5.2 to transform the core read and write processes into their width-adaptive versions, which are listed respectively in Programs D.9 and D.10. Note that for the

WAD read and write ports, locking is only required in the propagation condition case, because the control cannot possibly violate exclusion in the termination case. Figure 5.3 illustrates the delimiter bit modification needed to implement width-adaptive read and write. For the read port, propagation depends on the value of the delimiter bit in the selected register, and for the write port, propagation depends on the delimiter bit of the incoming number.

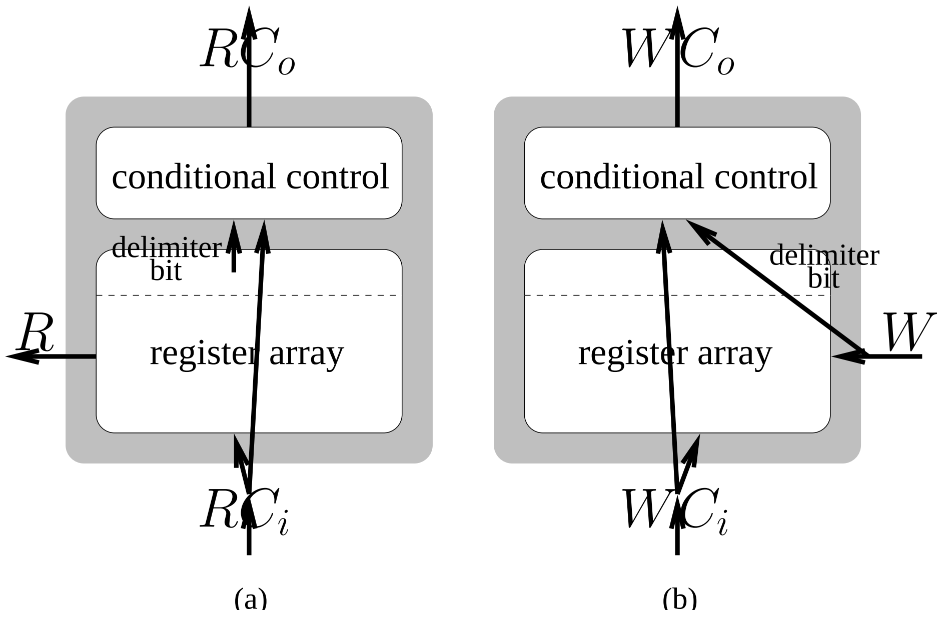


Figure 5.3: Block diagram of a width-adaptive register core a) read port and b) write port

Now we are ready to re-apply template handshaking expansions and floor decomposition to the modified read and write port processes, in the same manner as in Sections 4.1 and 4.2.

5.4 Template Handshaking Expansions

—(

The primary difference between HSEs for the non-WAD and WAD core ports is that the control output actions are conditional, therefore acknowledgment is only conditionally dependent on the control output. Conditional outputs are a simple extension to the general buffer reshuffling templates, which is described in Lines' thesis [23]. The *BYPASS* handshaking expansions follow directly from simple application of handshaking templates with conditional outputs. In this section, we discuss some subtleties of the HSEs for the *CORE* port processes. The HSEs we show in this chapter are the final results of transformations similar to those detailed in Section 4.1. The initial and intermediate versions of these HSEs are derived in greater detail in the technical report [11].

Program 5.3 HSE: PCEVFB WAD pipeline stage template with locking. C_o is an conditional output channel with locking, U_o is an unconditional output channel.

```
$*[(([U_o^e];\ en_U\uparrow;\ [C_i];\ U_o\uparrow),$
    $([C_o^e];\ en_C\uparrow;$
        $[C_i \wedge p(\ldots) \wedge unlocked() \longrightarrow lock;\ C_o\uparrow \,[\!]\, t(\ldots) \longrightarrow \textbf{skip}]));$
  $C_i^e\downarrow;$
  $(([\neg U_o^e];\ en_U\downarrow;\ U_o\downarrow),$
    $([(p(\ldots) \wedge \neg C_o^e) \vee t(\ldots)];\ en_C\downarrow;\ unlock;\ C_o\downarrow),$
    $([\neg C_i \wedge \neg ren_U \wedge \neg en_C];\ C_i^e\uparrow))$
 $]$
```

Program 5.4 HSE: PCEVHB WAD pipeline stage template with locking. C_o is an conditional output channel with locking, U_o is an unconditional output channel.

```
$*[(([U_o^e];\ en_U\uparrow;\ [C_i];\ U_o\uparrow),$
    $([C_o^e];\ en_C\uparrow;$
        $[C_i \wedge p(\ldots) \wedge unlocked() \longrightarrow lock;\ C_o\uparrow \,[\!]\, t(\ldots) \longrightarrow \textbf{skip}]));$
  $C_i^e\downarrow;$
  $(([\neg U_o^e];\ en_U\downarrow;\ U_o\downarrow),$
    $([(p(\ldots) \wedge \neg C_o^e) \vee t(\ldots)];\ en_C\downarrow;\ unlock;\ C_o\downarrow;$
      $[\neg C_i \wedge \neg ren_U \wedge \neg en_C];\ C_i^e\uparrow))$
 $]$
```

Program 5.3 shows the full-buffer template for a process with one unconditional input channel, one unconditional output channel, and one conditional output channel with locking. Program 5.4 shows the half-buffer counterpart. We have applied decoupling transformations so that each output channel has its own *en* internal enable [11]. These templates may be trivially generalized for an arbitrary number of the used channels. Since these HSE templates are correct in the general case, there is no need to prove the correctness of every specific instance thereof.

5.4.1 Core Read Port HSE

After introducing the WAD transformation to the read port, the control output is conditional on the value of the delimiter bit of the selected register, $p(reg)$. Program E.1 is the HSE result after applying the template HSE Program 5.3. We only need to lock in the case when control is propagated, and the reset phase, starting with $ren_C\downarrow$, only waits for the output acknowledge $\neg R_o^e$ when control is propagated. For output full-buffered data output with half-buffered control propagation, the result is HSE Program E.2.

5.4.2 Core Write Port HSE

The non-WAD write port receives an input control and input data and unconditionally produces a control output. After the WAD transformation, the control output is conditional on the delimiter bit of the input data, $p(W)$. We present two variation of the WAD write port, using different transformation templates.

Unconditional internal enable. One way to make the output conditional is to apply template HSE Programs 5.3 and 5.3, which results in adding the propagation condition guard $p(W_i)$ before $WC_o\uparrow$, as shown in Program E.3 (PCEVFB), and Program E.4 (PCEVHB). We refer to this version as the *unconditional write-enable* or *uwen* variation. Since the write-action does not generate a data output token, only control propagation requires its own internal enable *wen*. In the control terminating case, the **skip** action does not actually have to wait for WC_i because it does not matter which register word was selected in the current pipeline stage. Thus, WC_i appears in the guard expressions for $WC_o\uparrow$ and $\langle write \rangle$, but not for **skip**. $p(W_i)$ is actually redundant in the guard before $wen\downarrow$ because it would already be implied by waiting for the output acknowledge $\neg WC_o^e$, i.e., no acknowledge would arrive if no control token was ever sent.

Conditional internal enable. An alternate HSE template for pipeline stages with conditional output is shown in Program 5.5 (for full-buffer). The difference from template Program 5.3 is that the internal enable for the conditional output, en_C, is raised conditionally, whereas in the former variation, en_C is raised unconditionally. Another difference is that the propagation and termination conditions are only checked in the set phase and never checked during the reset phase, which will lead to simpler circuits. In the control termination case, the sequence $C_o^e\downarrow \prec en_C\downarrow \prec unlock \prec C_o\downarrow$ is entirely vacuous because $en_C\uparrow$ never fires, so $C_o\uparrow$ never fires, therefore $C_o^e\downarrow$ never acknowledges. Both templates transform unconditional output channels to conditional, however they translate into different circuits, which we will compare at the end of this chapter. Applying this alternate template to the WAD write port results in Program E.5 (PCEVFB) and Program E.6 (PCEVHB). We refer to this version as the *conditional write-enable* or *cwen* variation.

5.4.3 HSE Summary

We have shown that the HSEs of the WAD versions of the core process closely resemble their non-WAD counterparts, thus we should expect that their floor decompositions are also similar, and therefore their production rules for circuits have much in common. Rather than present exhaustive, repetitive floor decompositions for the width-adaptive core read and write port HSEs, we cut straight to production rule synthesis in this chapter. The components of the resulting floor decompositions appear in Appendix F. Detailed and comprehensive floor decompositions (of the same fashion as those presented from Section 4.2) are provided in the technical report [11].

Program 5.5 HSE: PCEVFB WAD pipeline stage template with locking and conditional internal enable. C_o is an conditional output channel with locking, U_o is an unconditional output channel.

```
*[(([$U_o^e$]; $en_U\uparrow$; [$C_i$]; $U_o\uparrow$),
    [$C_o^e \wedge p(\ldots) \longrightarrow en_C\uparrow$; [$C_i \wedge unlocked()$]; $lock$; $C_o\uparrow$
      [] $t(\ldots) \longrightarrow$ **skip**]);
   $C_i^e\downarrow$;
   (([$\neg U_o^e$]; $en_U\downarrow$; $U_o\downarrow$),
    ([$\neg C_o^e$]; $en_C\downarrow$; $unlock$; $C_o\downarrow$),
    ([$\neg C_i \wedge \neg ren_U \wedge \neg en_C$]; $C_i^e\uparrow$))
  ]
```

We provide the following figures as roadmaps from complete HSEs to floor decomposed components and production rules. Figures 5.4 and 5.5 show the decompositions of the WAD read port for the PCEVFB and PCEVHB reshufflings. Figures 5.6 and 5.7 show the decompositions of the WAD write port with unconditional write-enable for the PCEVFB and PCEVHB reshufflings. Figures 5.8 and 5.9 show the decompositions of the WAD write port with conditional write-enable for the PCEVFB and PCEVHB reshufflings.

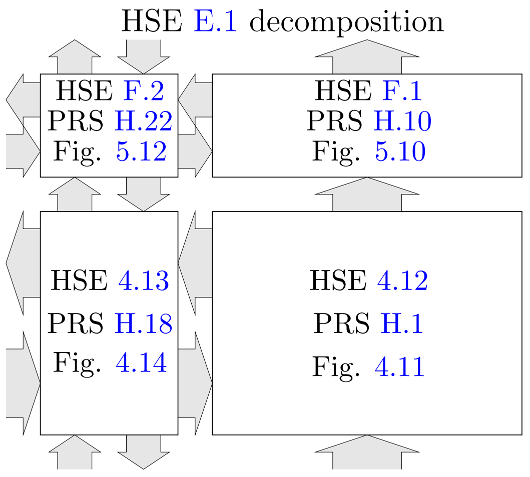


Figure 5.4: Floor decomposition of a PCEVFB WAD read port

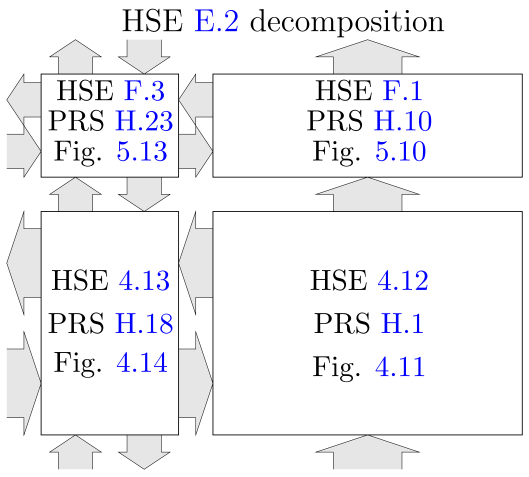


Figure 5.5: Floor decomposition of a PCEVHB WAD read port

—)

5.5 Width-Adaptive Production Rules

The partial handshaking expansions of the WAD read and write ports are very similar because the only change we introduced was conditional control propagation. The data components of the floor decomposition are the same except that we have

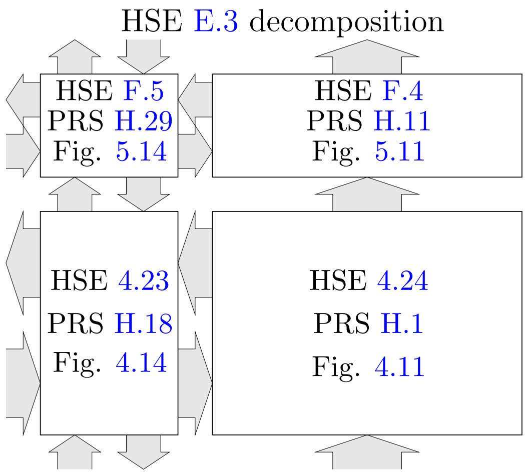


Figure 5.6: Floor decomposition of a PCEVFB WAD write port (unconditional write-enable)

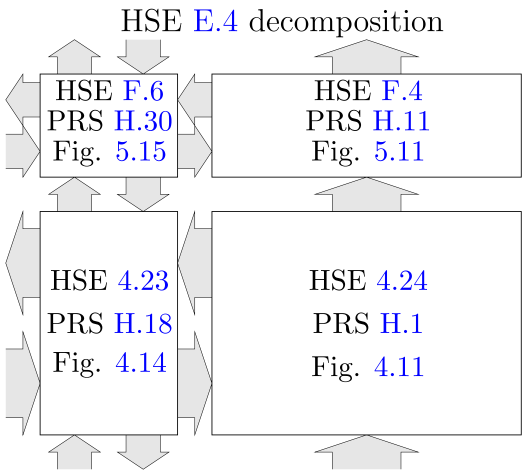


Figure 5.7: Floor decomposition of a PCEVHB WAD write port (unconditional write-enable)

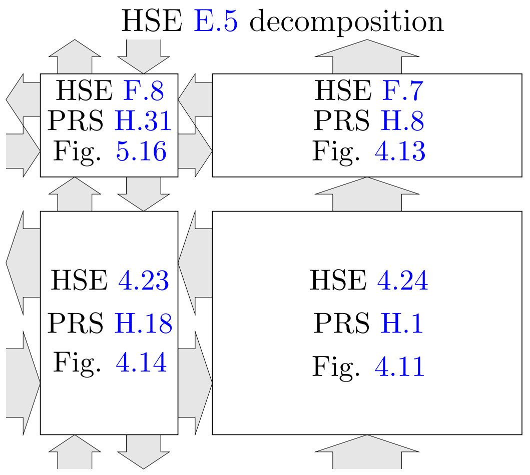


Figure 5.8: Floor decomposition of a PCEVFB WAD write port (conditional write-enable)

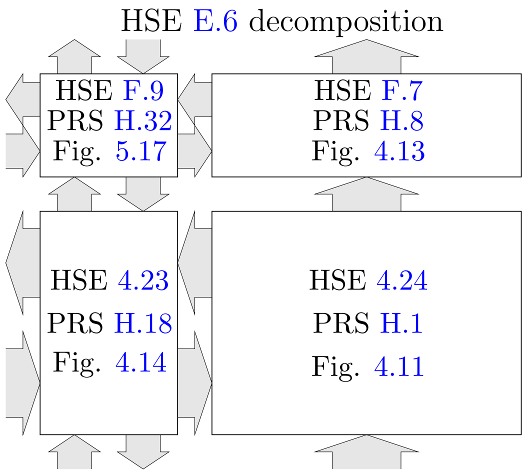


Figure 5.9: Floor decomposition of a PCEVHB WAD write port (conditional write-enable)

added one more row of storage per pipeline stage for the delimiter bits. Since the data components of the HSEs have not been changed by width adaptivity (except for the number of bit lines), their production rules remain unchanged from the non-WAD base design. Only the control propagation elements and the handshake controls have been adapted with slight modifications to support width adaptivity.

5.5.1 WAD Control Propagation

The introduction of conditional control propagation leads to the addition of at most only a single n-transistor (per port per word line) to the original precharge stage for unconditional control propagation. The other additional circuitry detects and signals the **skip** condition for control termination in the read control propagation array.

WAD Read Control. (Program H.10, Figure 5.10) The WAD read control propagation production rules have an additional series NFET, which implements the dx^0 guard of $RC_o\uparrow$, which is translated from `[`$p(reg) \mapsto \ldots RC_o\uparrow$`]`. The translation of the locking condition has not changed from the base design. The new production rules for $\_RC_o^f$ implement the **skip** action in the termination case, and requires no locking. $\_RC_o^f$ is shared across the entire control propagation array.

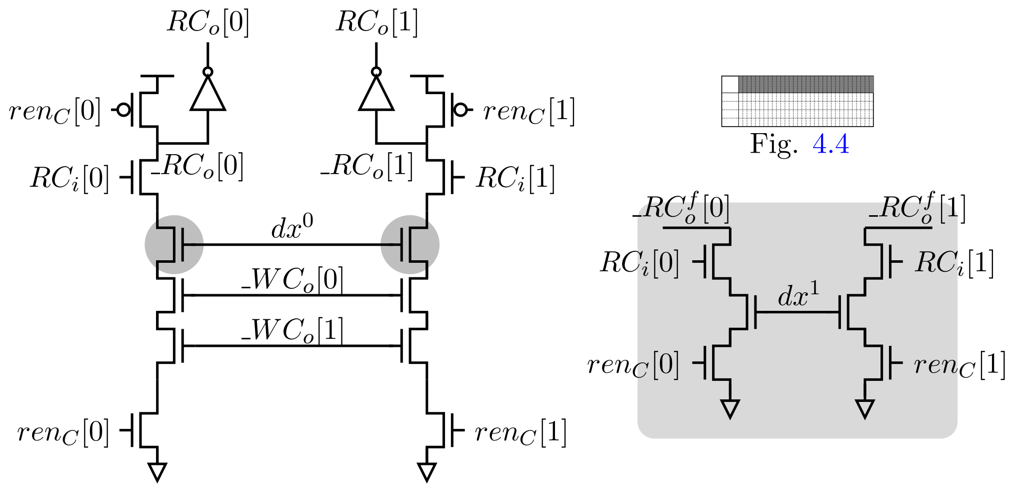


Figure 5.10: Width-adaptive pipeline-locked read control propagation, two ports shown. Shaded circuits are modifications introduced by WAD.

WAD Write Control, Unconditional Write-Enable. (Program H.11, Figure 5.11) The WAD write control propagation for the unconditional write-enable adds one series NFET to implement the dW^0 guard of $WC_o\uparrow$, which is translated from `[`$p(W) \mapsto \ldots WC_o\uparrow$`]`. The additional NFET adds little to no area in comparison to the non-WAD write control propagator.

WAD Write Control, Conditional Write-Enable. (Program H.8, Figure 4.13) With the conditional write-enable reshuffling, for every iteration where $wen\uparrow$, we are guaranteed that the input WC_i will arrive (eventually) and cause $WC_o\uparrow$ to fire, thus the control propagation *behaves* like an unconditional control propagation with respect to wen. Since the partial HSE of the non-WAD and WAD-cwen write ports are equivalent, we can use the exact same circuit as shown in Figure 4.13.

Register Zero. Recall that the most compact width-adaptive binary representation of the value 0, using blocks of 4 bits, is just 10000, where the 1 represents the terminating delimiter bit.[2] Only one block's worth of bits needs to be commu-

[2] The delimiter bit of a WAD zero register will be hard-wired to 1 instead of 0.

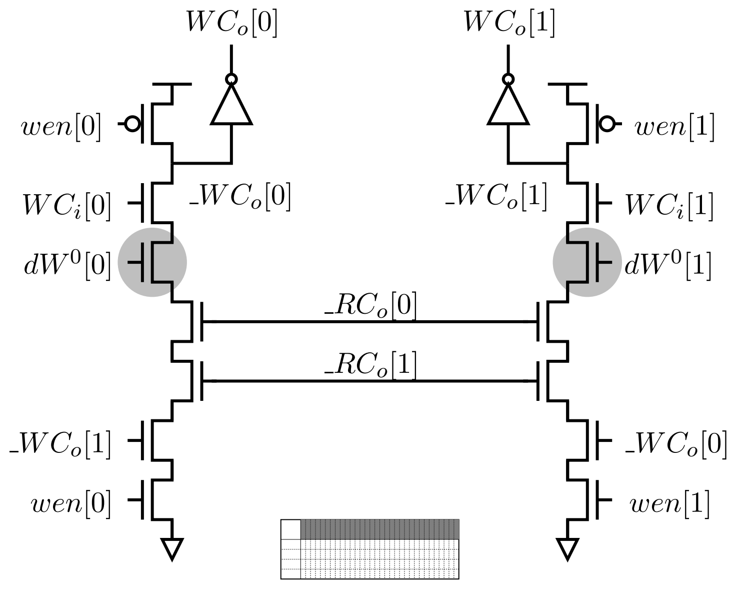


Figure 5.11: Width-adaptive pipeline-locked write control propagation, for unconditional write-enable, two ports shown. Shaded circuits are modifications introduced by WAD.

nicated from the core for a read from register zero.[3] The higher significant blocks require no circuits for driving the output for register zero, and can therefore may omit the production rules for $\_R{\downarrow}$ from Program H.2. Since read control propagation is omitted beyond the least significant block, read control completion trees will require one less input. This makes read-accesses to a WAD register zero, which are somewhat frequent, extremely energy-efficient on the datapath. However, we still need production rules for a non-modifying write to register zero, because an input data token may take an arbitrary number of WAD blocks to represent. The write control propagation may be non-locking since there can be no data hazards through register zero. In Chapter 6, we will discuss alternative implementations of the zero register outside of the core.

5.5.2 WAD Read Handshake Control

The PRSs for the WAD read handshake control are listed in the following PRS Programs: PCEVFB H.22 (Figure 5.12), PCEVHB H.23 (Figure 5.13). The notable difference between the unconditional and WAD versions are the production rules for the terminating condition, $\_RC_o^f$ and RC_o^f. The input is acknowledged with $RC_i^e{\downarrow}$ after the output control is valid RC_o^v or control is terminated RC_o^f. The

[3] A possible alternative implementation of the zero value may place a delimiter bit *below* the least significant bit to indicate whether the value is zero or non-zero. The tradeoff would be that zero values have been made more efficient at the expense of adding one more bit of switching to all non-zero values.

read-enable ren_C is reset after the output control is acknowledged $\neg RC_o^e$, but only when control is propagated, otherwise it is bypassed by $\neg\_RC_o^f$ in the termination case.

Note that for full-buffering, we use a variation where $\neg RC_o^f$ is checked *before* requesting the next token with $RC_i^e\uparrow$. This may seem like half-buffering in the terminating case, however, RC_o^f is not a true output, and one may argue by transition count that $RC_o^f\downarrow$ is unlikely to be on the critical path of the reset phase. The other option is to complete RC_o^v and RC_o^f together with a NOR gate and check the result before $\_ren\downarrow$, but this incurs more overhead circuits to keep the system QDI.

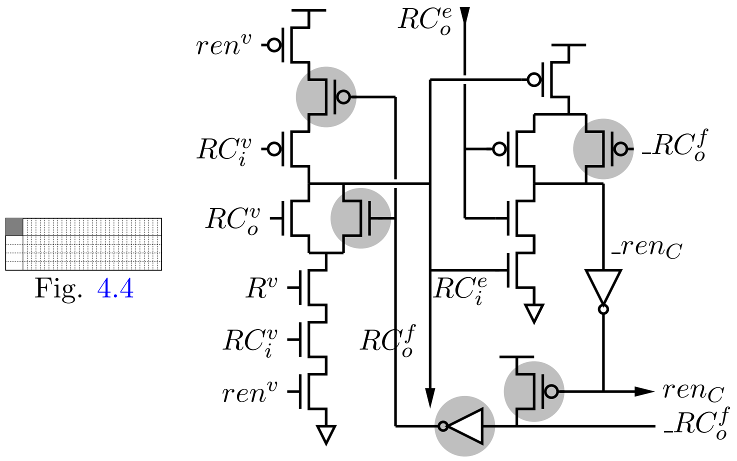


Figure 5.12: Single port of a width-adaptive read handshake control, PCEVFB reshuffling (resets not shown). Shaded circuits are modifications introduced by WAD.

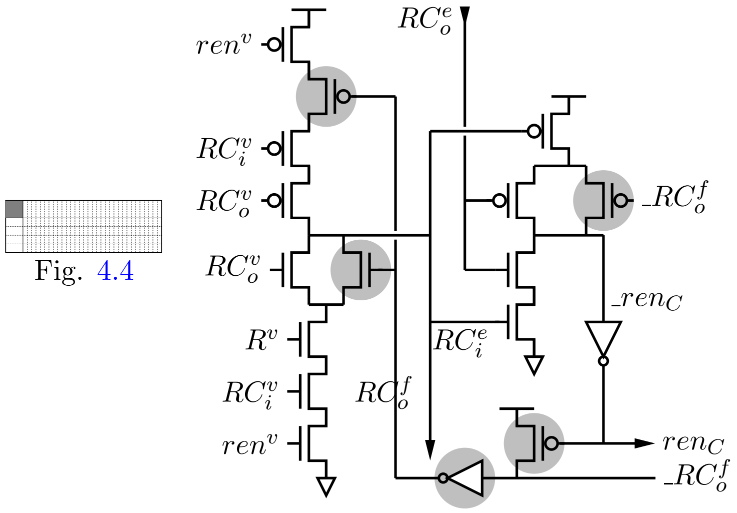


Figure 5.13: Single port of a width-adaptive read handshake control, PCEVHB reshuffling (resets not shown). Shaded circuits are modifications introduced by WAD.

5.5.3 WAD Write Handshake Control

Unconditional Write-Enable

The PRSs for the WAD write handshake control with unconditional write-enable are listed in the following Programs: PCEVFB H.29 (Figure 5.14), PCEVHB H.30 (Figure 5.15). The control termination condition is detected by WC_o^f and its complement. The input acknowledge $WC_i^e\downarrow$ is sent after the output is valid WC_o^v or control is terminated WC_o^f. The write-enable *wen* is reset after the control output is acknowledged $WC_o^e\downarrow$, but only if control is propagated, otherwise the acknowledge check is bypassed by $\neg\_WC_o^f$. For the full-buffer, we use the variation where $\neg WC_o^f$ is checked *before* requesting the next input token with $WC_i^e\uparrow$. A rough transition count of the cycle reveals that WC_o^f is very unlikely to be on the critical path of the reset phase.

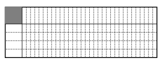


Fig. 4.4

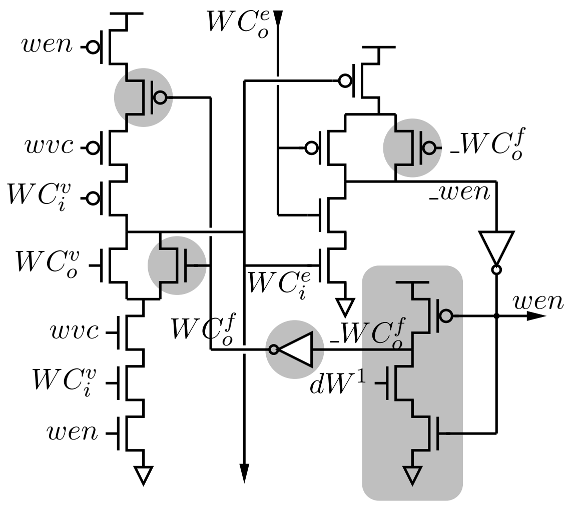


Figure 5.14: Single port of a width-adaptive write handshake control, with unconditional write-enable, PCEVFB reshuffling (resets not shown). Shaded circuits are modifications introduced by WAD.

Conditional Write-Enable

The PRSs for the WAD write handshake control with conditional write-enable are listed in the following Programs: PCEVFB H.31 (Figure 5.16), PCEVHB H.32 (Figure 5.17). For the conditional write-enable reshuffling, $wen\uparrow$ is guarded by the propagation condition, dW^0, so there is no need to locally compute WC_o^f for the termination condition. The termination condition, dW^1, bypasses the wait for the control output validity WC_o^v before the input is acknowledged $WC_i^e\downarrow$. The circuits for both the full-buffer and half-buffer are noticeably simpler than the unconditional write-enable counterparts. In the control terminating block, *wen* remains low, therefore WC_o^v remains low, and RC_o^e remains high, so the entire right-half of the circuits in Figure 5.16 and 5.17 remains idle, which saves

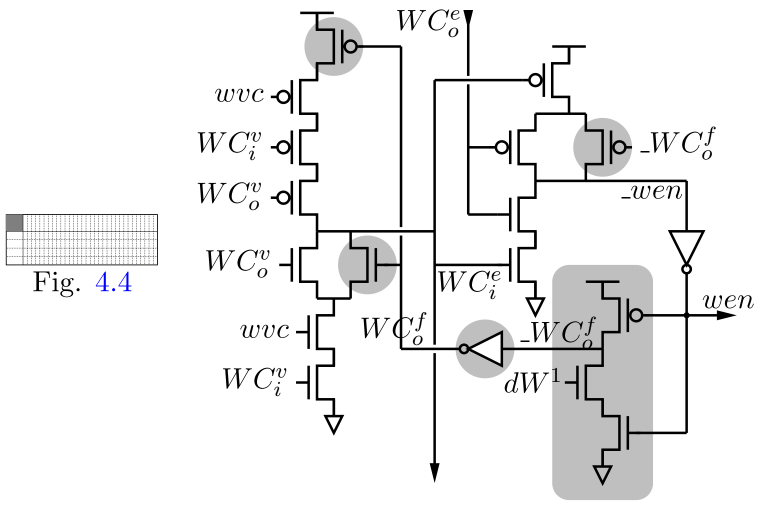


Figure 5.15: Single port of a width-adaptive write handshake control, with unconditional write-enable, PCEVHB reshuffling (resets not shown). Shaded circuits are modifications introduced by WAD.

some energy (in the terminating case) compared to the unconditional write-enable variations.

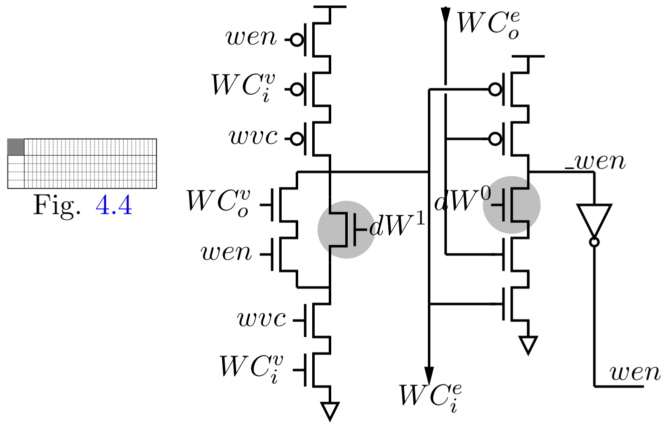


Figure 5.16: Single port of a width-adaptive write handshake control, with conditional write-enable, PCEVFB reshuffling (resets not shown). Shaded circuits are modifications introduced by WAD.

5.5.4 PRS Comparison of WAD Write Ports

With production rules for both variations of the WAD write control we can speculatively compare their performance and energies. In the case of control propagation, we note that $wen\uparrow$ occurs later for the conditional write-enable than with the unconditional write-enable because it must wait for the data dW to arrive. Assuming that subsequent actions in the WAD write port cycles are similar (same

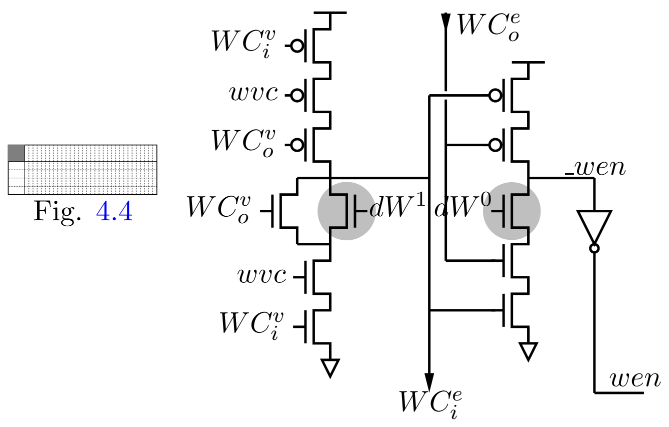


Figure 5.17: Single port of a width-adaptive write handshake control, with conditional write-enable, PCEVHB reshuffling (resets not shown). Shaded circuits are modifications introduced by WAD.

transition count), one can expect the conditional write-enable version to have a slightly longer handshake cycle time, and slower vertical latency per block (roughly, four transitions instead of two).

However, a slower write port operation may not noticeably slow down the entire datapath. A slower register write would slow down the datapath every time a dependent register read stalled on the same register, because the performance of reading would be limited by the write's cycle time and vertical latency of unlocking. We expect this to be a rare case because the bypass already forwards dependent operands past the core and through to the operand bus, so the core writeback remains off the critical path. The width-adaptive bypass can be implemented with a two-transition vertical latency per block by using the unconditional bypass-enable variation of reshuffling, analogous to the unconditional write-enable. Thus, we can tolerate a slightly slower writeback operation in the core.

We expect the conditional write-enable version to consume less energy than the unconditional write-enable version, because dW^0 is not wired to input gates across an entire array for every block, and *wen* (which fans out across the control array) is not switched in the terminating block of a WAD write. We will show below that the handshake control circuit for the conditional write-enable is slightly simpler and therefore smaller.

5.6 Results

In this section we present results for the WAD implementations of the register core. We compare the WAD results with the non-WAD base design for both banked (16 registers) and unbanked (32 registers) register cores, and we show the impact of width adaptivity on performance and energy.

5.6.1 Area

Recall that the block floorplan for the WAD designs is very similar to that of the non-WAD base design. The most significant difference is that each block has an additional row of register cells for the delimiter bit. The layout dimensions correspond to the labels in Figure 4.4 and entries in Table 4.1. The height of the WAD control propagation cell (for both unconditional and conditional write-enable) is $y_{cp_{WAD}}$, which is only 5% larger than the non-WAD counterpart.

5.6.2 Reading

For the read port simulations, we simulate only control propagation cases for all blocks because the termination cases skip the output handshake and operate with fewer cycle transitions, and hence will never limit the overall cycle time. Thus, we allow all non-delimiter register bits to reset randomly with metastability, but force the delimiter bits to reset to 0 to guarantee propagation. Since the register cell and interface array circuits have not changed, the read latencies remain the same as before, as listed in Tables 4.2 and 4.3.

Table 5.2: Read-access performance and energy comparisons for the WAD register file, for a block size of 4 bits x 32 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	22	2.149	465.4	34.10	157.5
full	20	2.014	496.4	33.18	134.6

Table 5.3: Read-access performance and energy comparisons for the WAD register file, for a block size of 4 bits x 16 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	22	2.025	493.8	19.88	81.6
full	20	1.872	534.3	19.61	68.7

Tables 5.2 and 5.3 show simulation results for both reshufflings of the WAD read port with, respectively, 32 and 16 registers per bank. The same results also appear in Table J.3. The relative improvements from banking read ports for other points in the design space are shown in Table J.7.

Comparing WAD: unbanked, 32 registers. For the half-buffer reshuffling, the WAD version is 9.1% slower than the non-WAD version and consumes 26.8%

more energy per block. For the full-buffer reshuffling, the WAD version is 7.6% slower than the non-WAD version and consumes 24.8% more energy per block.

Comparing WAD: banked, 16 registers. For the half-buffer reshuffling, the WAD version is 10.1% slower than the non-WAD version and consumes 24.9% more energy per block. For the full-buffer reshuffling, the WAD version is 9.3% slower than the non-WAD version and consumes 24.3% more energy per block.

The increase in energy per block fits our expectations because width adaptivity adds one more bit line per block, which was originally four bits. Since the average case width of 32-bit integers is far less than 80% of the full-width [25], width adaptivity would result in overall energy savings, because fewer blocks are activated. Even after we account for the combined effect of performance and energy with energy efficiency ($E\tau^2$), which is worse by 46.0% to 51.0% per block, a WAD read port is still expected to be more energy-efficient than a non-WAD read port.

Complete comparisons between all WAD designs of the read port and their non-WAD counterparts are given in Tables J.5 (half-buffered) and J.6 (full-buffered). Typical throughput degradation from adding width adaptivity ranges from 7 to 10%, however, we expect that performance gap may be reduced with more aggressive transistor sizing (at the cost of more energy), since we reused as much layout as possible from the non-WAD read port. Table J.4 contains comparisons between half and full buffering for the WAD read port. Full buffers are typically 5 to 9% faster than the half buffer versions.

Comparing banking: WAD half-buffer reshuffling. For the half-buffer reshuffling, reducing the bank size to 16 results in a 6.1% speedup in throughput, 41.7% reduction in energy per cycle per block, which amounts to a 93.1% improvement in energy efficiency.

Comparing banking: WAD full-buffer reshuffling. For the full-buffer reshuffling, reducing the bank size from 32 to 16 results in a 7.6% speedup in throughput, 40.9% reduction in energy per cycle per block, which amounts to a 95.9% improvement in energy efficiency.

5.6.3 Writing, Unconditional Write-Enable

For all write port simulations, we simulate only control propagation cases for the same reason as with the read port, thus we write only 0s in the delimiter bit position for all blocks, while all other bits toggle between alternating data tokens to simulate worst-case writing energy. Since the register cell and interface array circuits have not changed, the write latencies remain the same as before, as listed in Tables 4.4 and 4.5. The same results also appear in Table J.13. The relative improvements from banking write ports for other points in the design space are shown in Table J.18.

Comparing WAD: unbanked, 32 registers. Table 5.4 shows simulation results for both reshufflings of the WAD write port with unconditional write-enable.

Table 5.4: Write-access performance and energy comparisons for the WAD register file, with the unconditional write-enable variation, for a block size of 4 bits x 32 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	22	2.601	384.5	35.07	237.3
full	20	2.604	384.0	34.90	236.7

For the half-buffer reshuffling with unconditional write-enable, the WAD version is 4.3% slower than the non-WAD version, consumes 26.1% more energy per block, and is less energy-efficient by 37.9% per block. For the full-buffer reshuffling with unconditional write-enable, the WAD version is 6.2% slower than the non-WAD version, consumes 27.1% more energy per block, and is less energy-efficient by 44.4% per block.

Table 5.5: Write-access performance and energy comparisons for the WAD register file, with the unconditional write-enable variation, for a block size of 4 bits x 16 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	22	2.288	437.0	13.17	69.0
full	20	2.281	438.5	13.46	70.0

Comparing WAD: banked, 16 registers. Table 5.5 shows simulation results for both reshufflings of the WAD write port with unconditional write-enable. For the half-buffer reshuffling with unconditional write-enable, the WAD version is 4.8% slower than the non-WAD version, consumes 17.2% more energy per block, and is less energy-efficient by 29.3% per block. For the full-buffer reshuffling with unconditional write-enable, the WAD version is 7.1% slower than the non-WAD version, consumes 19.1% more energy per block, and is less energy-efficient by 38.1% per block.

Comparing banking: WAD half-buffer reshuffling. For the half-buffer unconditional write-enable reshuffling, reducing the bank size to 16 results in a 13.7% speedup, 62.5% reduction in energy per cycle per block, which amounts to a 244.1% improvement in energy efficiency.

Comparing banking: WAD full-buffer reshuffling. For the full-buffer unconditional write-enable reshuffling, reducing the bank size from 32 to 16 results in a 14.2% speedup, 61.4% reduction in energy per cycle per block, which amounts to a 238.3% improvement in energy efficiency.

5.6.4 Writing, Conditional Write-Enable

Table 5.6: Write-access performance and energy comparisons for the WAD register file, with the conditional write-enable variation, for a block size of 4 bits x 32 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	24	2.556	391.3	34.40	224.7
full	22	2.636	379.4	36.04	250.4

Table 5.7: Write-access performance and energy comparisons for the WAD register file, with the conditional write-enable variation, for a block size of 4 bits x 16 registers

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	24	2.243	445.9	13.03	65.5
full	22	2.320	431.1	13.51	72.7

Tables 5.6 and 5.7 show simulation results for both reshufflings of the WAD write port with conditional write-enable.

Comparing WAD: unbanked, 32 registers. For the half-buffer reshuffling with conditional write-enable, the WAD version is 2.7% slower than the non-WAD version, consumes 23.7% more energy per block, and is less energy-efficient by 30.5% per block. For the full-buffer reshuffling with conditional write-enable, the WAD version is 7.3% slower than the non-WAD version, consumes 31.3% more energy per block, and is less energy-efficient by 52.8% per block.

Comparing WAD: banked, 16 registers. For the half-buffer reshuffling with conditional write-enable, the WAD version is 2.8% slower than the non-WAD version, consumes 16.0% more energy per block, and is less energy-efficient by 22.9% per block. For the full-buffer reshuffling with conditional write-enable, the WAD version is 8.7% slower than the non-WAD version, consumes 19.6% more energy per block, and is less energy-efficient by 43.4% per block.

Table 5.7 show simulation results for both reshufflings of the WAD write port with conditional write-enable. The same results also appear in Table J.14. The relative improvements from banking write ports for other points in the design space are shown in Table J.18.

Comparing banking: WAD half-buffer reshuffling. For the half-buffer conditional write-enable reshuffling, reducing the bank size to 16 results in a 14.0% speedup, 62.1% reduction in energy per cycle per block, which amounts to a 242.8% improvement in energy efficiency.

Comparing banking: WAD full-buffer reshuffling. For the full-buffer conditional write-enable reshuffling, reducing the bank size from 32 to 16 results in a 13.6% speedup, 62.5% reduction in energy per cycle per block, which amounts to a 244.4% improvement in energy efficiency.

With the expected number of active width-adaptive blocks, a width-adaptive write port (both conditional and unconditional write-enable) will consume significantly less energy than a non-width-adaptive write port, and still be slightly more energy-energy efficient.

Complete comparisons between all WAD designs of the write port (including unconditional and conditional write-enable) and their non-WAD counterparts are given in Tables J.16 (half-buffered) and J.17 (full-buffered). As expected, implementing width adaptivity incurs up to around 25% overhead in energy per block because of the additional delimiter bit. However, taking into account the typical compression of a 32-bit integer, width adaptivity (even with four-bit block granularity) achieves an overall reduction in energy consumption. The conditional write-enable variations typically consume 1 to 3% less energy than the unconditional write-enable variations (for both reshufflings) because dW^0 does not fan out across the entire control propagation array and the handshake control circuit is simpler. Another less significant reduction in energy (which was not simulated) results from the fact that *wen* is not raised in the width-adaptive terminal block of the write port, which would probably save another 1 to 2% in only the terminal block.

Table J.15 compares half and full buffering for the WAD write ports. Interestingly, there is little difference in performance between the full and half buffered WAD write ports with unconditional write-enable, but for the conditional write-enable variations, the half buffer actually outperforms the full buffer by around 3%, and consumes 1 to 5% less energy. The differences are too small to conclude whether one reshuffling is superior to the other because of freedom in transistor sizing.

5.7 Summary

In this chapter, we have shown the transformation from a standard vertically pipelined register core into a width-adaptive core, which adds one more bit-slice to the base design block and makes control propagation for the read and write port blocks conditional. New circuits for the WAD core were derived for the control propagation array and handshake control, while all others remained the same, even for the banked design. More importantly, width adaptivity is entirely transparent to the *CONTROL* for the register core, and thus requires no modification (and hence, no complication) in the *CONTROL*. Our simulation results show that simply implementing the width-adaptive read and write ports results in a little performance loss from the increase in complexity, and an increase in block energy overhead proportional to the relative increase in the number of bits. The savings from typical integer compression overcomes the overhead, which makes width adaptivity a good solution for reducing energy on the datapath.

Chapter 6

Register Zero

The MIPS architecture specifies that register zero is hard-wired to the value 0. This chapter focuses on possible alternatives to implementing a hard-wired zero register. In Chapter 4, we described how to implement the zero register in the core, and gave a set of production rules for the register cell. In Chapter 5, we described how the control for reading from register zero was simplified by width adaptivity. This chapter is organized into two parts: the first part describes the high-level CHP transformation that moves the functionality for reading register zero into the *CONTROL* and *BYPASS*, and the second part describes the *CONTROL* modification that moves the functionality for non-modifying writes into the bypass.

6.1 Related Work

The zero register is frequently sourced as an operand and used as a destination when a result from an execution unit is discarded. A survey of some SPECInt95 benchmarks run on RISC machines showed that as many as 40% of register writes and 25% of register reads reference register zero [50]. It is also useful for synthesizing new instructions from existing instructions by using one operand as 0. These are a few of the reasons register zero was introduced in many early RISC architectures [19].

In the MiniMIPS, reads from register zero came from the core and passed through the read bypass before reaching the bus [31], and non-modifying writes to register zero passed through the writeback bypass and were consumed in the core. Our non-WAD implementation of the core zero register, including the non-locking control propagation, uses the same production rules as those in the MiniMIPS, except that we have connected the blocks in a vertical pipeline instead of using pipelined completion .

Tseng showed that moving the zero register to the bypass instead of the core saved 18% to 26% energy depending on the access frequency [50]. The majority of the energy reduction came from reducing bit line activity, but some of the reduction may be attributed to the reduced bit line capacitance from having one fewer register share each bit line. In their single-railed (synchronous) register file, the switching activity of the register core depended on the value of the operand being sourced, whereas in our dual-rail register core, each bit will always switch one bit line. Thus, we expect our relative energy savings from reading our of the core to be less than that found by Tseng. They also save energy on the non-modifying writes by conditionally suppressing write bit line switching. However, given that energy dissipated by single-railed writing to the register core depends whether or not the write bit line is discharged, we expect greater relative energy savings for suppressed writes to dual-rail register core.

6.2 Reading Register Zero

We express the same energy reducing techniques mentioned before as transformations of our register file CHP decomposition. While the original specification of the register file Program 2.2 remains unchanged, the decomposition changes slightly in the *BYPASS* and the *CONTROL*. In this section, we work with the finely decomposed processes in Chapter 2 without having to re-decompose the register file from the top. The *CORE* process is somewhat simplified after removing the conditional check for the zero index, but as we have seen in the production rule synthesis of the core, this only translates to not using a special zero register in the core cell array.

6.2.1 Bypass Modifications

The new decomposition of the *BYPASS* and *CONTROL* keeps the same channel interfaces as the original decomposition detailed in Chapter 2. Our original decomposition of the *BYPASS* finished with the read bypasses given in Program B.2. We encode an additional value "*zero*" on the control channels BPY and BPX. Program B.7 shows the read bypass with an additional case for sourcing the 0 value to the operand buses X and Y.

Recall that the original read bypass fit the template of a standard conditional input or merge process, for which production rule synthesis (for many reshufflings) is straightforward [23]. All we have done is add one more input case to the merge process, except that the "*zero*" case doesn't actually require a data input token. One benefit of sourcing a value from the bypass is that a bypass-sourced value can arrive at the operand bus roughly two transitions sooner than a value sourced from the register core. A simple synchronous datapath would reap little or no gain from having a value available on the bus a fraction of a clock cycle earlier, but an asynchronous datapath may begin useful work as soon as a value is available without constraint to a discrete time-granularity.

6.2.2 Control Modifications

Now that the read bypass can support sourcing the hard-wired zero value, we need to update the *CONTROL* to detect a read access to register zero, send the "*zero*" control to the read bypass, and suppress control to the *CORE*. The original read bypass control processes are listed in Program C.1. The new read bypass control is shown in Program C.6. In the $\neg zx$ and $\neg zy$ sub-cases, we now compare the source index with 0. If register zero is indexed, the control sends "*zero*" to the bypass, and suppresses sending the index token to the core. When the read bypass receives "*zero*", it will not expect any input from the core, thus we have preserved the semantic flow of tokens in and out of the system composed of the new bypass and control.

6.2.3 Impact of Width-Adaptivity

The modified bypass can be vertically pipelined as easily as the original bypass, by applying the template transformation. Since width adaptivity is completely transparent to the control, The same control can be used for both non-WAD and WAD register files. A WAD bypass would require the register zero modification only at the least significant block, because sourcing 0 always terminates control at the first block; the remaining pipeline stages remain the same as the non-WAD versions.

We observed at the end of Section 5.5.1 that a read from a WAD register zero consumes only a fraction of the core energy from reading a non-WAD register zero. For a 32 bit register file with four-bit width-adaptive granularity, only the least significant block is communicated, thus consuming only about 1/8th the energy of a full-width read. Thus, the relative energy savings of suppressing register zero reads is greater for a non-WAD register cores than to a WAD cores. However, in both cases, the zero value appears on the operand buses sooner, so both designs would equally benefit in performance.

6.3 Writing Register Zero

The original specification for the register file already includes a case for suppressing writes to the register core in the writeback bypass, shown in Program B.1. The *BPWB* channel communicates whether or not the value received on Z is committed to the register core. The only modification required is the control process that communicates on *BPWB*.

6.3.1 Control Modifications

The original writeback control is Program C.2 and the new writeback control is specified in Program C.7. We have added a new case for when register zero is accessed as a destination. The control must still read a token on the $ZBUS_{WB}$ input channel and read the validity bit that accompanies it on ZV, but we send **false** on *BPWB* to suppress copying a non-modifying write unnecessarily to the register core. Note that the control can tell the writeback-bypass to discard the result independent of the validities *val* and *zv*. We still communicate **null** on the *WI* index channels to the core to synchronize the demuxes, and guarantee read-write exclusion of the core port indices.

One could arguably reduce energy further for read accesses from register zero by hard-wiring zero values directly at the inputs of the execution units, at the cost of adding complexity to the decode and execution units. Likewise, terminating write accesses to register zero as early as the outputs of execution units can further reduce energy consumed by the buses. However, in this thesis, we restrict ourselves to evaluating techniques that do not affect the original sequential specification.

6.3.2 Impact of Width-Adaptivity

Since the writeback bypass remains unchanged, width adaptivity introduces the same transformation as shown in Section 5.3.1. Each non-modifying write to register zero that terminates in the bypass saves core energy. The amount of core energy saved depends on the frequency of writes to register zero, and how many blocks are communicated for the width-adaptive versions. A non-WAD core unconditionally receives a full-width input spanning all blocks, whereas a WAD core receives variable-width inputs, spanning fewer blocks. Thus, the relative energy savings is greater for a non-WAD register file.

6.4 Summary

This chapter presented some alternative implementations (at the CHP level) of a hard-wired zero register that reduce core energy consumption. The *CORE* and *BYPASS* changes proposed are compatible with other transformations, such as width adaptivity, and the optimizations presented in the remainder of the thesis.

Chapter 7

Port Priority Selection

As register files continue to grow well beyond 32 physical registers, and into hundreds of physical registers with increasing number of ports to accommodate increasing instruction-level parallelism (ILP), their energy consumption becomes increasingly significant in the energy budget for a processor core. In this chapter, we present another transformation that potentially reduces core energy consumption in the register file, *Port Priority Selection* or PPS, introduced and patented by Sun Microsystems [40]. The general idea behind PPS is that multiple copies of the same register value need not be simultaneously read from the core, rather, a single copy may be fetched from one port (the one with highest 'priority' among those requesting the same index) and duplicated external to the core, as shown in Figure 7.1. We present PPS as a high-level transformation of the *BYPASS* and *CONTROL* in the context of our dual-ported asynchronous register file.

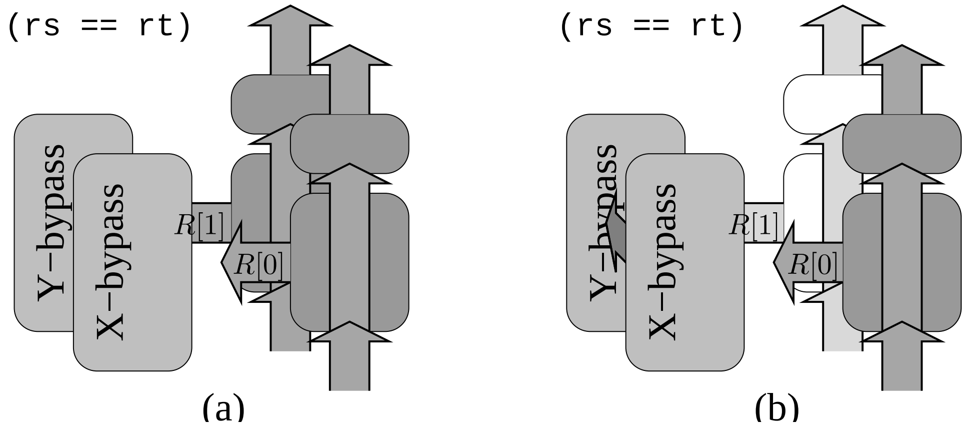


Figure 7.1: a) A traditional multi-ported register file may retrieve the same register through different ports, whereas b) a PPS implementation may reduce energy by suppressing redundant read accesses to the core.

7.1 Related Work

Zyuban and Kogge modeled the benefit of using PPS in multi-ported register files in superscalararchitectures and concluded that PPS (along with other energy-reducing analog circuit techniques) would potentially reduce energy consumed by heavily ported register cores by large factors [57, 58]. Another motivation for PPS is that by making read ports exclusive, they open the opportunity for using the same bit lines for (time-multiplexed) reading and writing, which greatly reduces the number of ports. The register cell we present does not support time multiplexed sharing of ports, but such designs may be of interest as the demand for ports increases. A general and efficient implementation of the port priority and operand copy logic is explained in the Sun patent [40]. However, we only need to implement the same logic for dual-read and dual-write ported registers.

7.2 Bypass Modifications

We start by modifying the (read) *BYPASS* processes to support operand copying. Suppose, without loss of generality, that the X port has higher priority than the Y port, meaning when both ports normally request the same register, only port X will read the operand from the core, port Y receives its copy from X. A schematic of the decomposed PPS read bypass is illustrated in Figure 7.2. The writeback bypasses remain unchanged from the original design and are not shown. We have introduced a new channel XY over which a value is copied from one port to the other operand bus. Another new channel from the *CONTROL*, *PPS*, tells the higher priority port X whether or not to copy its read value to Y.

We rewrite the decomposed CHP for the read bypass as listed in CHP Program B.8. We have included the changes from Chapter 6 to support the read bypass sourcing of the hard-wired zero value. In the *BPZX* read bypass, we have added a receive communication on channel PPS_{BPX}? which controls the conditional copy on XY!. In the *BPZY* read bypass, we have extended the *BPY* channel to communicate one more exclusive signal "*fromX*", which selects XY? as the source of input.

The *BPZX* read bypass is now a three-way merge with two outputs, one of which is conditional. The behavior of *BPZX* fits into a class of generalized function templates for which handshaking expansions and production rules are straightforward and requires no further analysis. The *BPZY* read bypass is simply extended to a four-way merge, for which template synthesis is well-known. We omit production rules for the new read bypasses from this thesis.

7.3 Control Modifications

In this section, we describe the changes necessary in the *CONTROL* to correctly operate the bypass that supports PPS. There are several variations that may work,

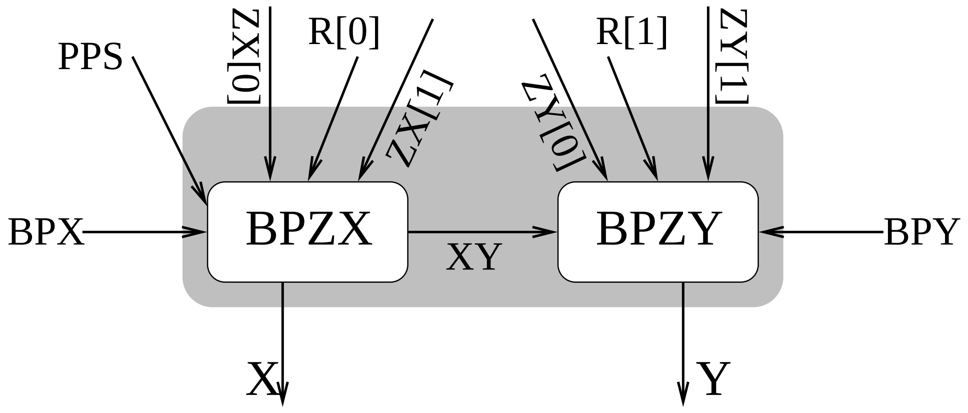


Figure 7.2: Modified read bypass decomposition for Port Priority Selection

but we present only one. We work directly with the decomposed *CONTROL* processes from Section 2.5.

The new decomposition is: $CONTROL \equiv RDCOPY \parallel RSRTEQ \parallel RSCOMP \parallel RTCOMP \parallel WBCTRL \parallel ZBCOPY \parallel RSCOPY \parallel RTCOPY$, and is shown in Figure 7.3. (For comparison, the original decomposition is shown in Figure 2.7.) The *RDCOPY*, *WBCTRL*, and *ZBCOPY* processes remain unchanged. We have added *RSRTEQ* (Program C.8) to compare when $rs = rt$, which needs copies of *RS* and *RT* from *RSCOPY* and *RTCOPY*, shown below.

$$CONTROL.RSCOPY \equiv *[RS?rs; RS_{RS}!rs, RS_{EQ}!rs]$$
$$CONTROL.RTCOPY \equiv *[RT?rt; RT_{RT}!rt, RT_{EQ}!rt]$$

In *RSRTEQ*, *eq* compares *rs* against *rt*, which determines when there is an opportunity to use the PPS.

We give the CHP for *RSCOMP* and *RTCOMP* in Program C.9. One can easily verify that when $rs \neq rt$ (*eqs* and *eqt* are false), the *CONTROL* and *BYPASS* processes behave exactly as they did in the original decomposition, without port priority selection. Now we verify the behavior when $rs = rt$. First we look at *RSCOMP*, the control for the *X* port. When the bypass forwarding condition is true for both read bypasses ($zx \wedge zy \Rightarrow rs = rt \Rightarrow eqs \wedge eqt$), we always suppress port copying at the read bypasses because the writeback already copies the dependent operand to both read bypasses; bypass-forwarding always overrides PPS. When an operand is bypassed, the *RTCOMP* only sends "$z0$" or "$z1$" to *BPYZ*. Changing the control to suppress copying at the writeback would involve more modifications than are necessary.

Note that in this version, if $rs = rt = 0$, we use the hard-wired zero at the bypasses without copying, because both read bypasses already support sourcing 0. Copying zero from *X* to *Y* would unnecessarily complicate control further. Finally, when $rs \neq 0 \wedge rt \neq 0 \wedge rs = rt$, we activate port copying on *XY* in the read bypasses by communicating $PPS_{BPX}!eqs$ (which is true) from *RSCOMP*, and $BPY!$"*fromX*" from *RTCOMP*. Thus, we have proven that the new register file

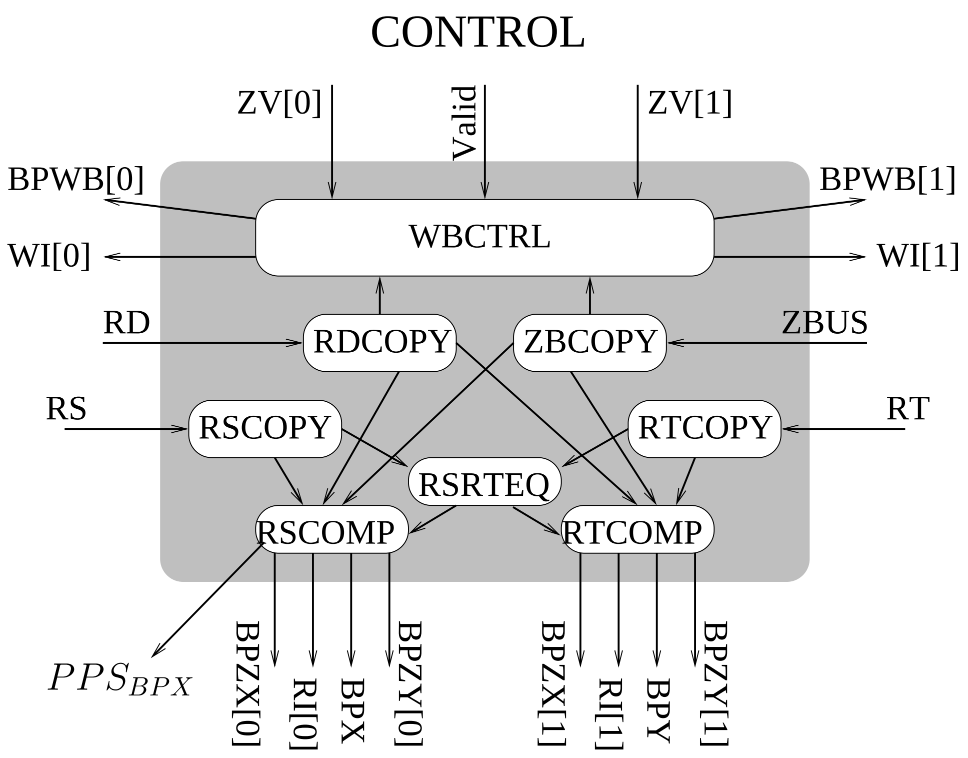


Figure 7.3: Schematic of Control decomposition for port priority select

decomposition of the *CONTROL* and *BYPASS* correctly implements priority port selection while adhering to overall behavior required by the original sequential specification. From here, synthesis into QDI production rules from the current decomposition is straightforward.

7.4 Summary

Having two read ports and two write ports in a register file is not considered heavily ported in comparison to register files found in modern superscalarmicroprocessors. The frequency of instructions that source two identical operands (out of a possible 32 registers) alone may not be sufficient to warrant the use of PPS. However, in modern and future generations of synchronous and asynchronous processors that increase the number of registers and buses to leverage increasing ILP, PPS may play an important role in reducing the number of accesses to the register file core(s) and the energy per access. We have demonstrated the ease with which PPS is specified and implemented asynchronously for a small number of ports, but a more general and scalable method may be required for more heavily ported register file architectures.

Chapter 8

Non-Uniform Control Completion

One of the limitations to symmetric banking that we have pointed out is that it introduces more channels, hence more wiring and interconnect requirements. It becomes difficult to place and route a single bypass in relation to a large number of banks, because the physical implementation is mapped onto a plane and a finite number of metal layers for routing. In the remaining two chapters, we turn to techniques that potentially speed up the register file access times and throughput in the *average* case *without* changing the *CORE*'s external interface, and hence, requiring no more channels. Our approach in this chapter is to leverage register usage distributions to give more frequently used registers higher throughput, while allowing infrequent registers to operate with a lower throughput.

In a synchronous design, unless the register file contains a critical path, the datapath is unlikely to speed up by making certain register accesses faster. However, if the slowest cycle introduced by nesting still meets the cycle time requirement, then one can potentially conserve energy by dynamically changing the load or drive strength of signals [50].

An asynchronous design, on the other hand, is not constrained to any global timing requirements, so introducing non-uniform register accesses has greater potential to reduce energy *and* gain performance in the average case. Moreover, robust delay-insensitive asynchronous systems can tolerate any variation in access times, therefore maintaining correctness comes at no additional complexity or retiming.

8.1 Register Statistics

Most architectures have designated conventions for register allocation, which are exposed to the register allocator of a compiler. For example, the MIPS register conventions are described in Table 8.1. One of the consequences of register conventions is that certain registers are used far more frequently than others. The most frequently used MIPS registers are bolded in Table 8.1. Typically, the 16 most frequently used registers on 32-register in-order machines running integer benchmarks account for over 90% of all register accesses [50]. We show the 20 most frequently read and written MIPS registers (sorted by frequency) in Table 8.2.[1] The top 16 registers constitute 99% of all accesses to the register file. The statistics always depend on the architecture and the compiler that generated the code.

Table 8.1: MIPS register conventions

name	reg#	convention
\$zero	0	**constant** 0
\$at	1	reserved for compiler
\$v0-\$v1	2–3	**results**
\$a0-\$a3	4–7	**arguments**
\$t0-\$t7	8–15	(callee-saved) temps
\$s0-\$s7	16–23	**caller-saved**
\$t8-\$t9	24–25	(callee-saved) temps
\$k0-\$k1	26–27	reserved for OS
\$gp	28	global pointer
\$sp	29	**stack pointer**
\$fp	30	**frame pointer**
\$ra	31	**return address**

More sophisticated out-of-order execution machines have register renaming hardware which dynamically re-maps logical registers to physical registers [38]. Dynamic register renaming may further increase the fraction of accesses represented by the most frequent half of physical registers if the renamer keeps track of a separate free-list per partition, and always allocates the first available fast register before allocating a slow register.

To evaluate potential speedup, we ask the following questions:

- Supposing we sorted registers by their usage frequency, what fraction of all accesses would be represented by the N most frequently used registers across a range of choices of N?

- What combination of speedup and slowdown for the respective partitions would result in a net speedup?

We let r_h represent the fraction of all register accesses represented by the most frequent half of registers (say, 16 out of 32), and normalize the baseline uniform access register file's cycle time to 1. τ_f (< 1) represents the normalized cycle time of the fast partition of the nested design, assumed to contain the most frequently

[1] Averaged across SPECInt95 benchmarks with training inputs: 099.go, 129.compress, 134.perl, 124.m88ksim, 130.li, 147.vortex, 126.gcc, 132.ijpeg, compiled with `gcc-2.95.3 -O3`, run on a MIPS simulator

Table 8.2: Cumulative dynamic usage frequencies of the 20 most read and written MIPS registers

N	reg	read%	cumul.	reg	write%	cumul.
1	0	32.95	32.95	0	27.63	27.63
2	3	14.90	47.85	2	18.96	46.59
3	2	12.59	60.44	3	18.35	64.94
4	30	9.78	70.22	4	9.57	74.51
5	5	8.03	78.25	5	6.83	81.34
6	4	6.36	84.61	6	4.87	86.21
7	29	4.73	89.34	31	2.95	89.16
8	16	2.36	91.70	29	2.52	91.68
9	31	1.73	93.43	16	2.04	93.72
10	17	1.43	94.86	30	1.67	95.39
11	6	1.19	96.05	14	1.49	96.88
12	28	0.91	96.96	1	1.03	97.91
13	14	0.85	97.81	17	0.52	98.43
14	1	0.60	98.41	18	0.41	98.84
15	18	0.48	98.89	7	0.30	99.14
16	7	0.29	**99.18**	19	0.22	**99.36**
17	19	0.23	99.41	8	0.13	99.49
18	20	0.15	99.56	20	0.13	99.62
19	21	0.13	99.69	9	0.08	99.70
20	8	0.09	99.78	21	0.08	99.78

used registers, and τ_g (> 1) represents the slow cycle time. The first-order average cycle time τ is given as[2]:

$$\tau = r_h \tau_f + (1 - r_h)\tau_g$$

If $\tau < 1$, then the average cycle time for the nested design is faster than that of the non-nested base design. An analogous calculation can also be done for normalized energy using E_f (< 1) and E_g (> 1). Given performance and energy measurements of a non-uniform access register file, one can compute breakeven probabilities for r_h to determine when nesting is likely to be beneficial:

$$\hat{r}_h = \frac{\tau_g - 1}{\tau_g - \tau_f}$$

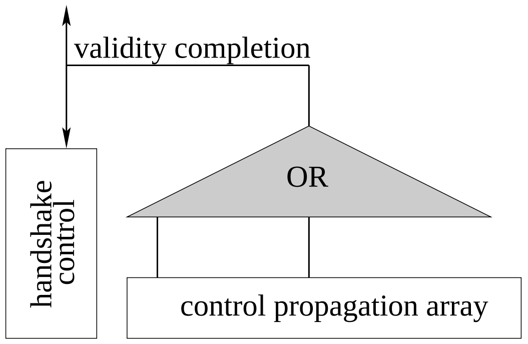
[2] This is ignoring hysteresis effects of cycle times from transitioning between fast and slow accesses, which may result in, effectively, slightly longer cycle times.

Since benchmarks only represent averages over a limited subset of benchmarks, one should also consider the performance *sensitivity* in the neighborhood of the breakeven probability $\hat{r}_h$, which is heavily dependent on the slower cycle time τ_g.

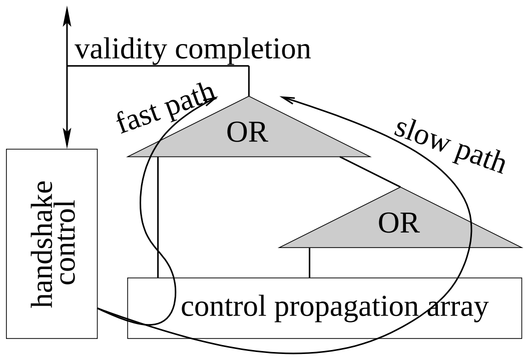
8.2 Unbalancing Completion Trees

We can already create non-uniform cycle time accesses to the register core without any transformations at the CHP or HSE levels. Thus far in this thesis, all of the control propagation completion trees we have used have been balanced trees of equal depth. Since the function of completion trees is just to guarantee validity of signals, one has a lot of freedom in their implementation. Changing the implementation does not affect the abstraction of the overall QDI asynchronous handshaking, thus, the correctness remains automatically preserved. Figure 8.1 illustrates the conceptual difference between balanced and unbalanced trees. The unbalanced tree contains a fast path (with lower tree depth than the balanced tree) and a slow path through both subtrees. Not only does the faster path reduce cycle time, but it also reduces energy in the average case.

The simplest unbalanced tree we introduce is a two-level tree, with leaves at one of two distances from the root. The top subtree is almost the same as the bottom subtree, except that it can take in one more input from the root of the bottom subtree. For the unbanked register core with 32 registers, this translates a 16-input OR-tree connected to a 17-input OR-tree.



(a) Balanced completion trees



(b) Unbalanced completion trees

Figure 8.1: The balanced completion tree has all paths of equal length, whereas the unbalanced tree shown has fast and slow paths which account for non-uniform cycle times. Data components are not shown.

One need not stop at introducing two levels in the unbalanced completion tree; trees may be designed with arbitrary balancing. Just as one constructs optimal Huffman codes based on symbol probabilities, one can analogously design completion trees to take advantage of any register (or datapath bus) usage distribution to optimize for performance or energy. One fact to bear in mind is that completion trees constitute only a fraction of the cycle time and cycle energy, thus, there may exist opportunities elsewhere in the circuits to apply unbalanced design in favor of more common paths.

8.3 Results

We have simulated all previous designs of the read and write ports redesigned with unbalanced completion trees. 32-input OR trees have been split into two levels of 16-input OR trees, and 16-input OR trees (from Chapter 4.4) have been split into two levels of 8-input OR trees. Since the 16-input OR trees were implemented in four stages of gates, we do not expect the unbanked (size 32) read and write ports with unbalanced trees to operate much (if at all) faster than those with balanced trees. Unbalancing completion trees in this case does not create a shorter fast path, so adding a longer slow path is unlikely to offer speedup or energy reduction. These simulations are uninteresting, but nonetheless they appear in the write port results tables in Appendix J in row entries with width 32a. However, the 8-input OR trees were implemented in only two stages of gates, thus we expect the banked (size 16) read and write ports with unbalanced trees to operate faster through the fast paths than those with balanced trees. Unbalanced trees create an opportunity to achieve average case speedup if sufficiently many accesses hit in the fast path. We also compute the breakeven probabilities (described in Section 8.1) for which average non-uniform accesses will be faster than uniform accesses. The read and write latencies remain the same because the bank sizes of the register cell arrays have not been changed.

Something to bear in mind about the breakeven probabilities for the results we show is that the potential gains and losses from non-uniform read accesses as a result of unbalanced completion trees is rather insignificant for our small register banks. However, completion trees for larger banks have greater potential to benefit (or worsen) by unbalancing.

8.3.1 Non-WAD Reading

Table 8.3 shows the non-WAD read port performance and energy results for the half-buffer and full-buffer reshufflings. The same results also appear in Table J.2. Performance and energy differences between non-uniform and uniform read port accesses and their breakeven probabilities appear together in Table J.11, listed in rows with width 16a. (The ‘a’ in ‘16a’ stands for *asymmetric* control completion with unbalanced completion trees.)

Half-buffer. The fast path’s cycle time is 0.994 of the uniform-access cycle time, and the slow path’s cycle is 1.070 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 91.6% of accesses must hit in the fast path. The fast path’s energy per cycle is 0.980 of the uniform-access energy, and the slow path's energy per cycle 1.024 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 55.0% of accesses must hit in the fast path.

Full-buffer. The fast path's cycle time is 0.995 of the uniform-access cycle time, and the slow path's cycle is 1.043 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 88.9% of accesses must hit in the fast path. The fast path's energy per cycle is 0.978 of the uniform-access energy, and the slow path's energy per cycle 1.008 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 26.6% of accesses must hit in the fast path.

Table 8.3: Read-access performance and energy comparisons for the non-uniform non-WAD register file with 16 registers. Upper numbers are figures for the faster half.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	18	1.809	552.7	15.60	51.1
	22	1.949	513.2	16.31	61.9
full	16	1.689	592.0	15.43	44.0
	20	1.771	564.6	15.90	49.9

8.3.2 Non-WAD Writing

Table 8.4 shows the non-WAD write port performance and energy results for the half-buffer and full-buffer reshufflings. The same results also appear in Table J.12. Performance and energy differences between non-uniform and uniform write port accesses and their breakeven probabilities appear together in Table J.22, listed in rows with width 16a. (The 'a' in '16a' stands for *asymmetric* control completion with unbalanced completion trees.)

Half-buffer. The fast path's cycle time is 0.998 of the uniform-access cycle time, and the slow path's cycle is 0.997 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 0.0% of accesses must hit in the fast path. The fast path's energy per cycle is 0.986 of the uniform-access energy, and the slow path's energy per cycle 0.998 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 0.0% of accesses must hit in the fast path.

Full-buffer. The fast path's cycle time is 0.999 of the uniform-access cycle time, and the slow path's cycle is 1.018 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 95.7% of accesses must hit in the fast path. The fast path's energy per cycle is 0.954 of the uniform-access energy, and the slow path's energy per cycle 0.987

of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 0.0% of accesses must hit in the fast path.

Table 8.4: Write-access performance and energy comparisons for the non-uniform non-WAD register file with 16 registers. Upper numbers are figures for the faster half.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	20	2.175	459.8	11.08	52.4
	22	2.172	460.3	11.22	52.9
full	20	2.116	472.5	10.78	48.3
	20	2.156	463.8	11.15	51.8

8.3.3 WAD Reading

Table 8.5 shows the WAD read port performance and energy results for the half-buffer and full-buffer reshufflings. The same results also appear in Table J.3. Performance and energy differences between non-uniform and uniform read port accesses and their breakeven probabilities appear together in Table J.11, listed in rows with width 16a.

Half-buffer. The fast path's cycle time is 0.978 of the uniform-access cycle time, and the slow path's cycle is 1.076 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 77.7% of accesses must hit in the fast path. The fast path's energy per cycle is 0.970 of the uniform-access energy, and the slow path's energy per cycle 1.018 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 38.0% of accesses must hit in the fast path.

Full-buffer. The fast path's cycle time is 0.994 of the uniform-access cycle time, and the slow path's cycle is 1.038 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 87.1% of accesses must hit in the fast path. The fast path's energy per cycle is 0.982 of the uniform-access energy, and the slow path's energy per cycle 1.011 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 38.2% of accesses must hit in the fast path.

8.3.4 WAD Writing

Table 8.6 shows the WAD, unconditional write-enable write port performance and energy results for the half-buffer and full-buffer reshufflings and Table 8.7 shows the same results for the conditional write-enable variation. The same results also

Table 8.5: Read-access performance and energy comparisons for the non-uniform WAD register file with 16 registers. Upper numbers are figures for the faster half.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	18	1.981	504.8	19.29	75.7
	22	2.179	458.8	20.25	96.2
full	16	1.861	537.3	19.26	66.7
	20	1.942	514.9	19.83	74.8

appear in Tables J.13 and J.14. Performance and energy differences between non-uniform and uniform write port accesses and their breakeven probabilities appear together in Table J.22, listed in rows with width 16a.

Half-buffer, unconditional write-enable. The fast path's cycle time is 0.998 of the uniform-access cycle time, and the slow path's cycle is 1.009 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 81.1% of accesses must hit in the fast path. The fast path's energy per cycle is 0.996 of the uniform-access energy, and the slow path's energy per cycle 1.000 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 9.0% of accesses must hit in the fast path.

Full-buffer, unconditional write-enable. The fast path's cycle time is 0.999 of the uniform-access cycle time, and the slow path's cycle is 1.017 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 94.4% of accesses must hit in the fast path. The fast path's energy per cycle is 0.993 of the uniform-access energy, and the slow path's energy per cycle 0.988 of the uniform-access energy. (This figure may be the result of numerical noise.) For the average energy of the non-uniform accesses to beat the uniform access energy, 0.0% of accesses must hit in the fast path, because both cases consume less energy. It is entirely possible to have slightly reduced energy in the slow path of the unbalanced 16-input OR-tree because the number of logic gates (and in our case, their sizes) through the slow path is the same as a balanced four-stage, 16-input OR-tree, but the *wiring* is reduced because connections are more localized for unbalanced trees.

Half-buffer, conditional write-enable. The fast path's cycle time is 1.000 of the uniform-access cycle time, and the slow path's cycle is 1.025 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 98.1% of accesses must hit in the fast path. The fast path's energy per cycle is 0.970 of the uniform-access energy, and the slow path's energy per cycle 1.003 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 9.2% of accesses must hit

Table 8.6: Write-access performance and energy comparisons for the non-uniform WAD (unconditional write-enable) register file with 16 registers. Upper numbers are figures for the faster half.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	20	2.283	438.0	13.12	68.4
	22	2.310	433.0	13.17	70.3
full	20	2.278	438.9	13.36	69.3
	20	2.319	431.2	13.30	71.5

Table 8.7: Write-access performance and energy comparisons for the non-uniform WAD (conditional write-enable) register file with 16 registers. Upper numbers are figures for the faster half.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	22	2.242	446.1	12.63	63.5
	24	2.299	434.9	13.07	69.1
full	20	2.313	432.4	12.81	68.5
	22	2.383	419.6	13.21	75.0

in the fast path.

Full-buffer, conditional write-enable. The fast path's cycle time is 0.997 of the uniform-access cycle time, and the slow path's cycle is 1.027 of the uniform-access cycle time. For the average cycle time of the non-uniform accesses to beat the uniform access cycle time, 90.3% of accesses must hit in the fast path. The fast path's energy per cycle is 0.948 of the uniform-access energy, and the slow path's energy per cycle 0.978 of the uniform-access energy. For the average energy of the non-uniform accesses to beat the uniform access energy, 0.0% of accesses must hit in the fast path, because both cases consume less energy.

8.4 Summary

We have shown that by unbalancing completion trees to create non-uniform access registers in the same bank, there is little potential to achieve significant speedup or energy reduction, however, even the worst cases through the slow paths are not much worse than those with balanced completion trees. The breakeven probabilities may be slightly misleading because they are highly sensitive to small gains and losses (normalized cycle times and energies close to 1.0). Nevertheless, tree unbalancing will be important for the next chapter, when we introduce a more extreme form of non-uniform access registers, nesting.

Chapter 9

Core Partitioning via Nesting

Last chapter, we described how to achieve non-uniform register access cycle times by unbalancing the completion trees of the control propagation output. With sufficiently skewed register usage distributions, a non-uniform cycle time register file may be faster on average than the balanced cycle time version. However, merely unbalancing the completion trees has no impact on the speed of the read and write bit lines because they are still shared across a large array of registers. Often times, a designer cares about keeping low latency, the delay from word line to data availability, of read and write operations. (This is especially crucial in large memories such as DRAMs.) Data dependencies create natural cycles on the datapath, whose performance may depend on the sum of latencies across all (horizontal) data pipeline stages on the datapath.

In this chapter we continue with the idea of non-uniform access time register cores, with the addition of non-uniform bit-line latencies. To achieve this, we must partition the bit lines to reduce the shared load in the common case, while providing longer latency, yet QDI, accesses to another partition, while preserving the original channel interface requirements. We call this method *nesting* because it effectively creates a hierarchy of register banks through the interface of a single register file. At the top of the hierarchy will be a partition that is smaller and hence faster than the unpartitioned register core, and deeper in the hierarchy can be other partitions that are slower to access. If one can arrange accesses to a nested register core to utilize the faster partition most of the time, then there is potential to achieve an overall reduction of latency in the average case.

9.1 Related Work and Applications

The Cray-1 implemented two-levels of registers that required explicit instructions to transfer data between levels [43, 44]. Swenson and Patt (1988) also proposed using hierarchical register files to cater to the demand for relatively few fast register backed by a large number of slower registers [47]. They observed that sections of code with high ILP did not suffer much loss in performance from having multi-cycle register accesses, whereas serially dependent sections of code (low ILP) were not accelerated by increasing the number of registers. Proper scheduling of serially dependent instructions through a small set of fast registers is likely to speed up execution of these critical sections of code, while a larger and longer latency register file can catch accesses that might otherwise use the cache or main memory.

Rixner, Dally, et al. evaluated the use of hierarchical register files (instead of the cache) for media applications that exhibit little data reuse (temporal locality) by prefetching directly into lesser-ported register files with more registers [42]. They also provided models for how the delay, energy, and area of hierarchical register files scale with number of registers and ports.

For synchronous designs *without* support for variable, multi-cycle register accesses, reducing register read latency on some register accesses is unlikely to speed up the datapath, because operands are synchronized at the same latches regardless of their true arrival times. However, if the slowest path introduced by nesting (e.g., with a pass gate on the bit line) meets the cycle time requirement, then significant bit line energy may be saved on average [50].

Synchronous designs that support multi-cycle register files introduce multi-level register bypasses, which further increases bypassing delay, branch mispredict penalties, and register lifetimes in superscalar processors [1]. As feature sizes shrink, multi-level bypasses are more likely to limit clock frequencies because of their wire-dominated interconnect requirements [36]. An alternative to full-bypassing a multi-cycle register file is to use only partial bypassing to reduce the bypass complexity, however this introduces cycles during which data becomes momentarily unavailable, which pushes significant complexity into the issue logic [8].

The benefit of a QDI asynchronous implementation of variable latency (and cycle time) register files is that the speed up of a partition of the register file may lead to an average reduction of forward latency through the datapath, and hence speedup, *without* additional retiming or bypass-forwarding support.

Cooper and Harvey proposed a compiler-controlled memory (CCM), which serves as a separate memory space apart from the cache, which is available to use by a compiler for spilling registers without polluting the cache [7]. One possible application for a slower second-level register file may be to serve as a CCM to support the primary register file, when it cannot afford to be enlarged due to timing constraints.

As parallel architectures further increase the demand for number of registers and ports, even banked (or other uniformly partitioned) designs suffer from the increase in interconnect requirements. Capitanio, et al. explored the tradeoff between full connectivity and limited connectivity in early VLIW (very-long instruction word) machines [5]. Zalamea, et al. proposed a two-level hierarchy of heterogeneous register files for VLIW machines [56]. Their second-level register file (called R2) had greater capacity but fewer ports than the primary register file (R1), and was not directly accessible by the functional units. R2 served as an intermediate memory between the R1 and the L1 cache, which required explicit load/store operations to move data between levels. They chose the largest sizes for R1 and R2 that allowed the access times to fit in the target cycle time.

Perhaps one of the most register-demanding parallel architectures proposed is that of simultaneous multi-threading (SMT), which supports issuing of instructions from multiple logical threads to fill instruction slots when ILP alone is insufficient to fill the issue bandwidth [51]. An SMT core requires at least as many physical registers as the number of logical registers per thread times the number of threads, and often requires more (sometimes exceeding 200 or 300) to keep the issue queue from stalling due to a shortage of free registers for dynamic allocation. Instead of resorting to uniformly multi-cycle latency register file accesses, implementations with variable access-time registers (assuming some intelligent register allocation policy) may offer average-case speedup.

Other possible applications for adding a slower second level of registers (that interface through the same hardware as the primary register file) include extending an ISA with special purpose registers, providing privileged registers for kernel instructions, or system profiling. For processors that keep multiple versions of internal state for swapping, backing-up, or checkpointing [33], a nested register file would provide a means to save and restore state with no impact on the interconnect requirement for implementation.

9.2 Nesting CHP Decomposition

The elegance of nesting register core partitions is that the transformation is exclusively local to the *CORE*. The channel interfaces to the *CORE* and *BYPASS* remain unchanged, therefore *CORE* nesting is completely transparent to *CORE*'s environment.

A diagram of vertically pipelined, nested read and write port operations is illustrated in Figure 9.1. Within the *CORE* we split the shared data channels, the read and write bit lines, into two halves: $\_R$ and W remain the same as the non-nested data channels, but are now shared among half as many registers, and $\_IR$ and IW are the new inner copies of the channels, shared among the other half of the registers.[1] We divide the word select control channels similarly: RC and WC represent the select channels for the outer, non-nested half of the registers, and IRC and IWC controls the inner, nested half.

Another way to interpret the nesting transformation is using width adaptivity, from Chapter 5. With WAD, the *data* values have variable length. Analogously, with nesting, the control *indices* to an arrayed structure (the core), have variable length, i.e., a nested array is *depth-adaptively* addressed with a width-adaptive index, which encodes the depth into the nested array.

[1] Recall that the $\_R$ is the inverted pseudo-channel whose bit rails are shared across the register array for each read port.

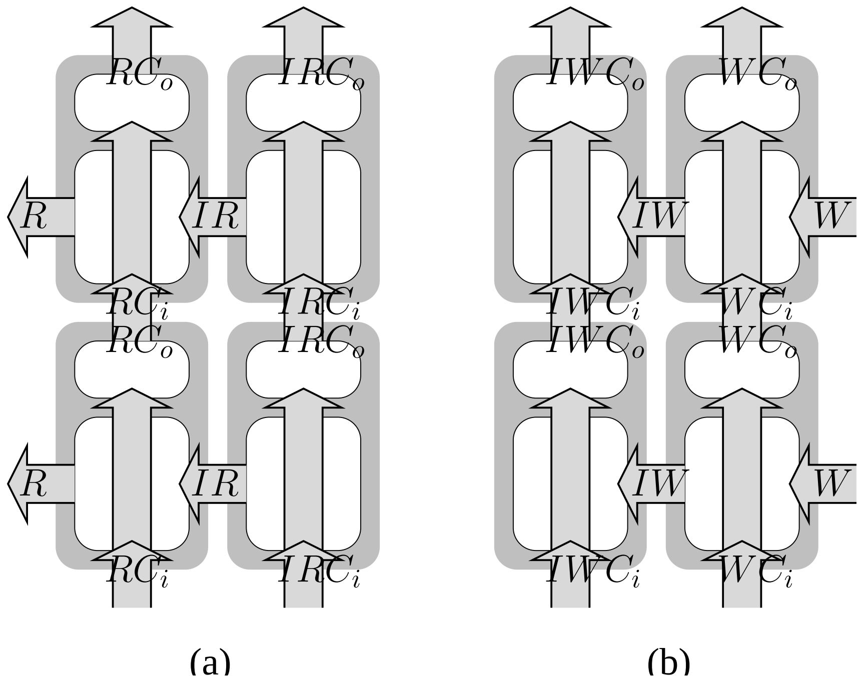


Figure 9.1: Block diagram of vertically pipelined, and nested read and write processes.

9.2.1 Unconditional Control Propagation

Following the CHP template for vertically pipelining with locking, Program 3.6, we extend the template for data-nested process in CHP Program 9.1. Read and write accesses to the outer partition (which we arbitrarily designate as the lower 16 registers) behave like the normal non-nested reads and writes, and do not communicate with the inner partition. Read and write accesses to the inner partition, however, will also activate the outer partition through a *CONNECT* interface process. Note that the CHP for the inner and outer *BLOCK*s are actually equivalent; the inner partition is connected to its own set of private channels. The resulting non-WAD nested read port is shown in Program D.11 and the nested write port is shown in Program D.12.

The *CONNECT* processes are activated by the inner partition word lines (1of16 channels) IRC_i and IWC_i. On a read access to the inner partition, data originates in the inner partition and is forwarded to the outer partition $R!(IR?)$ by *CONNECT*.[2] Analogously, a write to the inner partition is forwarded from the outer partition $IW!(W?)$ by the interconnect. Instead of using a true handshake in the interconnect, we devise a lightweight interconnect between the partitions that requires no completion trees in the inner partition and minimizes the modifications in the outer partition.

The process specification of the demux for the nested core is listed in Pro-

[2] We have written this communication in this manner to specify that the data token is *not* buffered, i.e. there is no full handshake between the inner and outer partitions.

Program 9.1 CHP: template for pipelined, non-WAD, nested process with locking at the sender

```
$BLOCK_{outer} \equiv$
        $*[C'; \langle outer\ \ data\ \ action \rangle; C'']$
    $\| *[[\overline{C_i}]; C'; [unlocked()]; lock; (C_o, (C''; C_i)); unlock]$
$CONNECT \equiv$
        $*[[\overline{IC_i}]; (\langle relay\ \ inner/outer\ \ data \rangle, IC_o, IC_i)]$
$BLOCK_{inner} \equiv$
        $*[IC'; \langle inner\ \ data\ \ action \rangle; IC'']$
    $\| *[[\overline{IC_i}]; IC'; [unlocked()]; lock; (IC_o, (IC''; IC_i)); unlock]$
```

gram D.13. When we index the inner partition, not only do we need to communicate the inner partition select lines IRC or IWC, we still need to communicate to the outer partition with RC^{inner} or WC^{inner} that we will be using the outer partition's shared data channels $\_R$ and W. RC^{inner} and WC^{inner} act like 17th word select lines for the outer partition and will be mutually exclusive with the other select lines RC and WC (now also 1of16 channels) to maintain exclusion for driving $\_R$ and reading W.

9.2.2 WAD Control Propagation

Nesting and width adaptivity are orthogonal transformations, and we give the template for their combined transformation in Program 9.2. The $CONNECT$ data interface is the same as the non-WAD version. Again, the data components of the $BLOCK$ processes for the inner and outer partition accesses are equivalent; each partition connects to its own set of channels. Applying this transformation results in Program D.14 for the WAD nested read port, and Program D.15 for the WAD nested write port. Control propagation for the inner partition read depends on $p(reg[l])$, which is the same as the non-nested version. Control propagation for the inner partition write depends on $p(IW)$, which is just the inner partition's copy of the input write delimiter bits.

9.3 Handshaking Expansion Modifications

In this section, we present the handshaking expansions for the various nested read and write ports without going into the details of their derivation. Their derivations are repetitive and mostly follow the same style as those of the non-nested designs, such as control-data decoupling. New transformations revolve around the fact that internal actions may be freely re-ordered and decoupled as long as the communication interface to the environment is preserved. Another guideline is to keep the existing components unchanged if possible, and otherwise introduce minimal changes while maintaining correctness. Step-by-step derivations of the nested

Program 9.2 CHP: template for pipelined, WAD, nested process with locking at the sender

```
$BLOCK_{outer} \equiv$
      $*[C'; \langle outer\ \ data\ \ action \rangle; C'']$
    $\| *[[\overline{C_i}]; C';$
        $[p(\ldots) \wedge unlocked() \longrightarrow lock; (C_o, (C''; C_i)); unlock$
        $[]\, t(\ldots) \longrightarrow C''; C_i$
      $]]$
$CONNECT \equiv$
      $*[[\overline{IC_i}]; (\langle relay\ \ inner/outer\ \ data \rangle, IC_o, IC_i)]$
$BLOCK_{inner} \equiv$
      $*[IC'; \langle inner\ \ data\ \ action \rangle; IC'']$
    $\| *[[\overline{IC_i}]; IC';$
        $[p(\ldots) \wedge unlocked() \longrightarrow lock; (IC_o, (IC''; IC_i)); unlock$
        $[]\, t(\ldots) \longrightarrow IC''; IC_i$
      $]]$
```

read and write port HSEs appear in the technical report [11].

9.3.1 Unconditional Read Control Propagation

Recall that the read port's HSE fit the template for a control-data fork process, which we transformed into Program 4.5 with full-buffering and Program 4.6 with full-buffered data output and half-buffered control propagation. Read accesses to the outer partition should still behave like their non-nested counterparts for all reshufflings.

Now we extend the HSE to include accesses to the inner partition, and combine all of the sub-processes in CHP Program D.11. The final HSE is listed in Program E.7. We have decoupled $iren_D$ and $iren_C$ as we have done before with ren_D and ren_C in Section 4.1.2. The action $IRC\uparrow$ represents the raising of one of the inner partition's register read select control rails. The old RC_i has been broken down into the cases $RC_{i,outer}$ and $RC_{i,inner}$, which are mutually exclusive. RC_{inner} is implicitly raised when IRC is used. During a read access to the inner partition, the input acknowledges, RC_i^e and IRC_i^e, which both acknowledge $\downarrow$, are decoupled from each other, and $IRC_i^e\downarrow$ need not check data output validity $R_o\uparrow$ nor $IR_o\uparrow$. In the reset phase of the data component, we allow $IR_o\downarrow$ and $R_o\downarrow$ to reset independently of one another.

To prove that the new HSE is compatible with the data environment, we show that by factoring out all the actions on $iren$ and the internal channels IRC and IR in the $RC_{i,inner}$ cases, we are left with the HSE of Program 4.5. It is no coincidence that the control and data actions in the $RC_{i,inner}$ cases resemble the same action sequences in the same phases of the non-nested handshaking expansion. This selfsimilarity leads to equivalent components in the floor decomposition.

We can analogously derive the HSE for the nested half-buffer read port (with full-buffered data output), listed in Program E.8. One feature of interest is that while the RC channel is half-buffered, we intentionally keep the IRC communication full-buffered, i.e., $IRC_i^e\uparrow$ does not wait for $IRC_o\downarrow$ in the reset phase. Since accesses to the inner partition are expected to be slower than the outer partition accesses, and the inner handshake is control-limited in cycle time, we choose full-buffering as the faster option for the inner partition.

The HSEs we have shown here, however, are not the final versions we use for floor decomposition. The final major transformation involves re-ordering the HSE so that $iren_D\downarrow \prec ren_D\downarrow$ and $iren_C\downarrow \prec ren_C$, to facilitate maintaining atomicity of $ren_C\downarrow$ and $ren_D\downarrow$. The same transformation is used for both non-WAD and WAD, nested read ports. A full-length discussion of this transformation can be found in the floor decomposition section of the corresponding chapter in the technical report [11].

9.3.2 Unconditional Write Control Propagation

We left off with the CHP Program for the nested write port in Program D.12. The pipelined write port's HSE fit the template for a control-data join process, which we transformed into Program 4.9 with full-buffering and Program 4.10 with half-buffered control propagation. Write accesses to the outer partition should behave like the non-nested write port for both reshufflings. Following the same transformations we used in Section 4.1.3, we have decoupled the data component from the control propagation and wen, introduced a write-validity signal wvc, combined the input acknowledges into WC_i^e ($\equiv W_i^e$) to obtain Program E.9 for the full-buffered reshuffling. The action $IWC\uparrow$ represents the raising of one of the inner partition's register write select control rails. Again, WC_i has been broken down into the cases $WC_{i,outer}$ and $WC_{i,inner}$, which are mutually exclusive. WC_{inner} is implicitly raised when IWC is used.

We can verify that the nested write port is compatible with the original data environment by factoring out the actions of $iwen$, and the internal channels IWC, and IW in the $WC_{i,inner}$ cases. The guarded action sequences for writes to the inner and outer partitions are similar, which comes as no surprise. We decouple the acknowledging actions of $IWC_i^e\downarrow$ and $WC_i^e\downarrow$ for greater concurrency. The handshake on IWC is full-buffer-like in that IWC_i^e need not wait for $IWC_o\downarrow$. Note that on a write to the inner partition, $IWC_i^e\downarrow$ need not wait for $IW\uparrow$ and $IWC_i^e\uparrow$ need not wait for $IW\downarrow$, which means that we have *completely* decoupled the control and data in the inner partition. However, control and data are still always synchronized with WC_i and wvc before $WC_i^e\downarrow$ because they share the same acknowledge.

The half-buffer version of the nested write port is similarly derived in Program E.10. Again, for greater concurrency, we keep IWC full-buffered by not waiting for $\neg IWC_o$ before requesting the next input with $IWC_i^e\uparrow$ in the reset phase.

9.3.3 WAD Read Control Propagation

Now we apply the width-adaptive transformation to the nested read port. After decoupling the inner partition enables into $iren_C$ and $iren_D$ and applying the same transformations used for the non-WAD nested read port in Section 9.3.1, the resulting HSE for the full-buffer reshuffling is Program E.11. Control propagation for the inner partition is conditional on the value of the delimiter bit of the accessed register in the inner partition. $p(reg)$ denotes the propagation condition and $t(reg)$ denotes control termination. Resetting the inner enable $iren\downarrow$ only waits for the inner partition acknowledge $\neg IRC_o^e$ in the propagation case, analogous to the outer partition's reset of $ren\downarrow$.

The half-buffered version of the WAD nested read port can be derived with the same routine transformations, and is listed in Program E.12. The data output handshake remains full-buffered.

9.3.4 WAD Write Control Propagation

Recall that in Section 5.4.2, we presented two reshufflings for the WAD write port: the unconditional write-enable and conditional write-enable variations, depending on when $wen\uparrow$ was allowed to set. With nesting, we introduce the inner partition's write-enable $iwen$, which introduces a choice for when $iwen\uparrow$ is allowed to be set, thus yielding three possible reshufflings of the WAD nested write port. Recall that the conditional write-enable variation for the non-nested write port had a simpler handshake control circuit and slightly slower cycle time. Assuming writes to the inner partition to be less frequent than writes to the outer partition, and critical writes to be already bypassed to the operand buses, we can afford to use the slower conditional write-enable variation for the inner partition with little expected loss in performance. Thus, we restrict our attention to the subset of two *conditional inner write-enable* reshufflings of the WAD, nested write port.

The final HSE for the WAD, nested, unconditional outer write-enable, full-buffered write port is shown in Program E.13, and the preliminary HSE for the conditional outer write-enable, full-buffered version is shown in Program E.15. (This time, we do not show the half-buffered counterparts because they are trivially similar.) For both HSEs, the $iwen\uparrow$ is conditional on the inner partition's delimiter bit.

The data components and full-buffer reset phases of the control component for both versions appear identical at the HSE level. The main difference lies in when $wen\uparrow$ occurs in the setting phase of the outer partition control, but we also point out more subtle differences.

In the unconditional outer write-enable version, the propagation condition $p(W_i)$ directly guards $WC_o\uparrow$, whereas IWC_o need not be guarded directly by

$p(IW)$ because $iwen\uparrow$ already implies $p(IW)$. We allow the inputs to be acknowledged ($IWC_i^e\downarrow$ and $WC_i^e\downarrow$) independently of one another. The resetting of $wen\downarrow$ only waits for $\neg WC_o^e$ in the propagation case and in the termination case for the inner partition, $[\neg IWC_o^e]; iwen\downarrow$ is vacuous because $iwen$ is never raised.

For the conditional outer write-enable, the $p(IW)$ guard is actually redundant because wen and $WC_{i,inner}$ already imply $p(IW)$, however the $t(IW)$ guard of **skip** is still needed. The sequence in the control termination case of the reset phase, both $[\neg WC_o^e]; wen\downarrow; unlock; WC_o\downarrow$ and $[\neg IWC_o^e]; iwen\downarrow; unlock; IWC_o\downarrow$ are vacuous sequences because $wen\uparrow$ and $iwen\uparrow$ never occur, so the behavior reduces to that of the terminal block of the write port. Even though we have separated the control and data in both partitions, we still share the inner partition's delimiter rails of IW because the inner partition's control enable $iwen$ is conditional on $p(IW)$. We take this into account in the floor decomposition and production rule generation.

9.4 Floor Decomposition

Since one of our goals is to introduce as little modification as possible to the non-nested designs to achieve nested designs, floor decomposition helps to identify which components the non-nested and nested designs have in common, which components require modification or replacement, and what new components are necessary to implement nesting. For nesting, we introduce a new floorplan, shown in Figure 9.3 for the read port and Figure 9.4 for the write port. The left halves of these figures are exactly the same as the floorplans shown in Figures 4.5 and 4.8. All we have done is split the old control propagation array and data cell array into inner and outer banks, and introduced nested interconnect components. The nested interconnect is a new component which will behave like another register and control propagation cell from the outer partition's perspective, and will behave like an external data interface array from the inner partition's perspective.

9.4.1 Read Data Nesting

We start with the data decomposition of the non-WAD core read port. Figures 9.5 and 9.6 give a visual outline of the final floor decomposition for the PCEVFB and PCEVHB reshufflings. This section discusses the decomposition of the bottom halves of these floor decompositions, the data array, interface and interconnect.

We showed in Chapter 5 that the width-adaptive transformation introduced no modifications to the partial HSE of the data cell array component, aside from sharing the internal delimiter bits, dx^0 and dx^1, to the control propagation array.

We also showed that no modifications are necessary in the cell arrays when we introduce nesting. If we compare Programs E.7 and E.8 for the non-width-adaptive versions, and Programs E.11 and E.12 for the width-adaptive versions, the guard for setting the inner partition's shared read rails $IR_o\uparrow$ is $[iren_D \wedge IRC_i \wedge reg]$, and the guard for resetting the read rails $IR_o\downarrow$ is $[\neg iren_D]$, which is independent

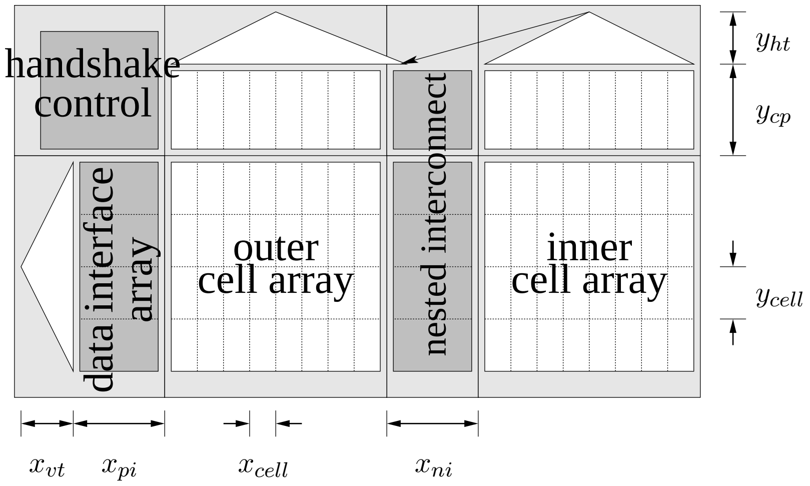


Figure 9.2: Floorplan of a nested 4-bit x 16-word pipeline block of the register core, with the outer partition on the left side and the inner partition on the right. New or modified components that arise from nesting are darkly shaded, while all other components corresponding to Figure 4.4 remain unchanged. The WAD, nested floorplan includes one more row of delimiter bit cells in the cell array. The dimensions for the various components are listed in Table 4.1.

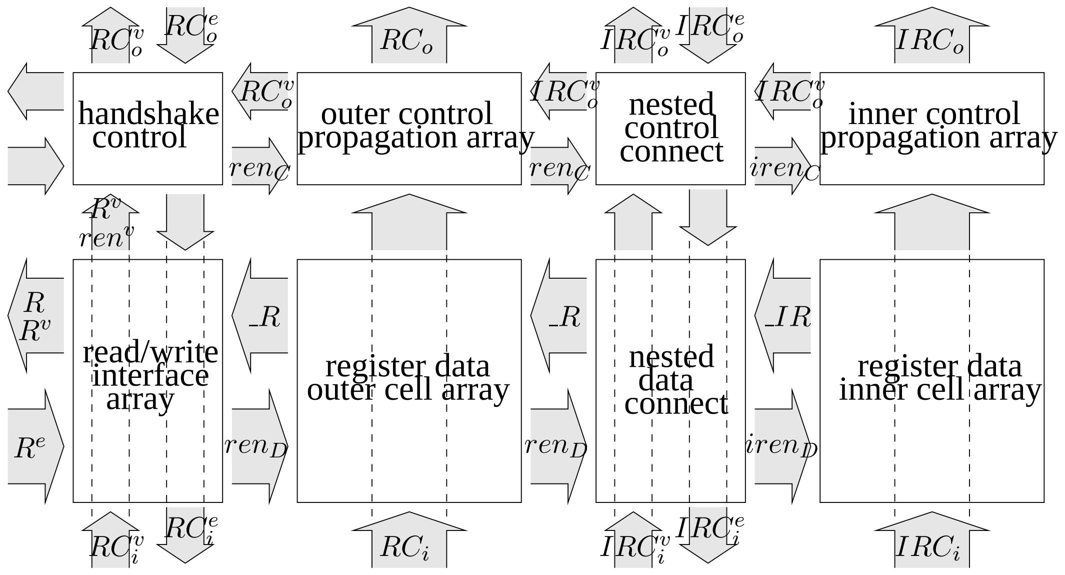


Figure 9.3: Floor decomposition of a data-nested core read port

of IRC_i. This means that the partial HSE for the inner partition's cell array is equivalent to that of the outer partition, which was shown in Program 4.12. Therefore, we can use the same template HSE as the old cell array for both the inner and outer partitions, and just connect the inner cell array to the inner partition's control signals, $iren_D$ and IRC_i, and register state variables.

Since all read accesses to the inner partition also use the outer partition's

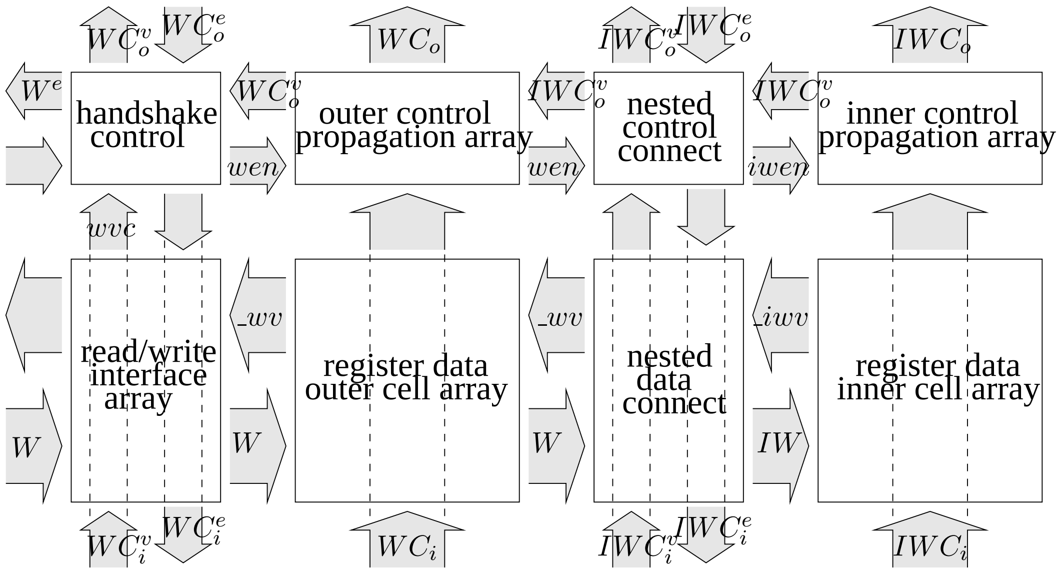


Figure 9.4: Floor decomposition of a data-nested core write port

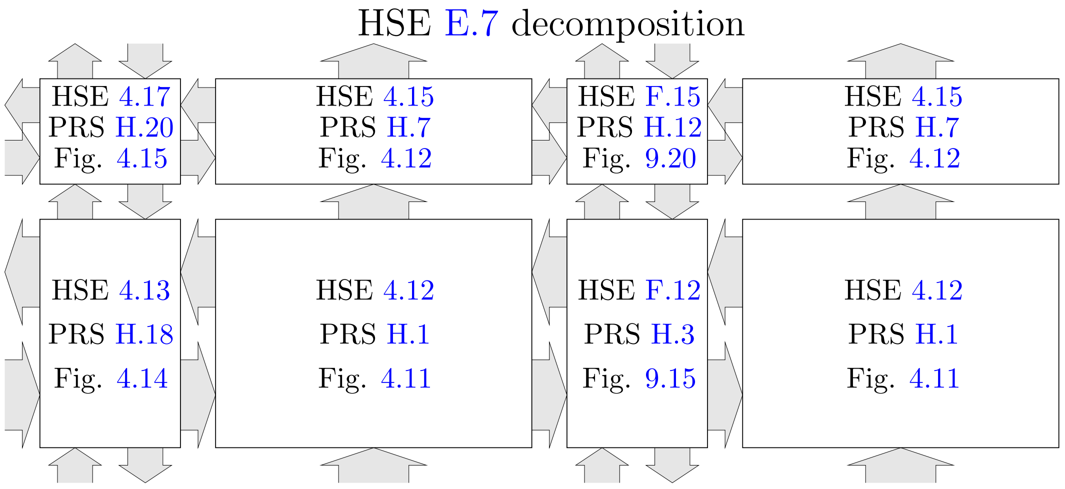


Figure 9.5: Floor decomposition of a PCEVFB nested read port

shared read channel $\_R$, we need to complete the nested connect interface to the outer partition. The guards for the outer partition's read rails R_o now appear in the outer partition and inner partition cases. The guards for the outer partition case remain unchanged from the non-nested designs, so we examine the guards from the inner partition. Comparing across the same HSEs mentioned above, the inner partition's guard for setting $R_o\uparrow$ is always $[ren_D \wedge IR_o]$, and the guard for resetting $R_o\downarrow$ remains as $[\neg ren_D]$. For setting $R_o\uparrow$ we do not need an explicit guard of $RC_{i,inner}$ because $IR_o\uparrow$ already implies $RC_{i,inner}$.

Because the partition cases are mutually exclusive, we have also guaranteed that no other control for the outer partition can drive $R_o\uparrow$ while the inner partition is selected. It will require a little additional work to guarantee that the inner partition

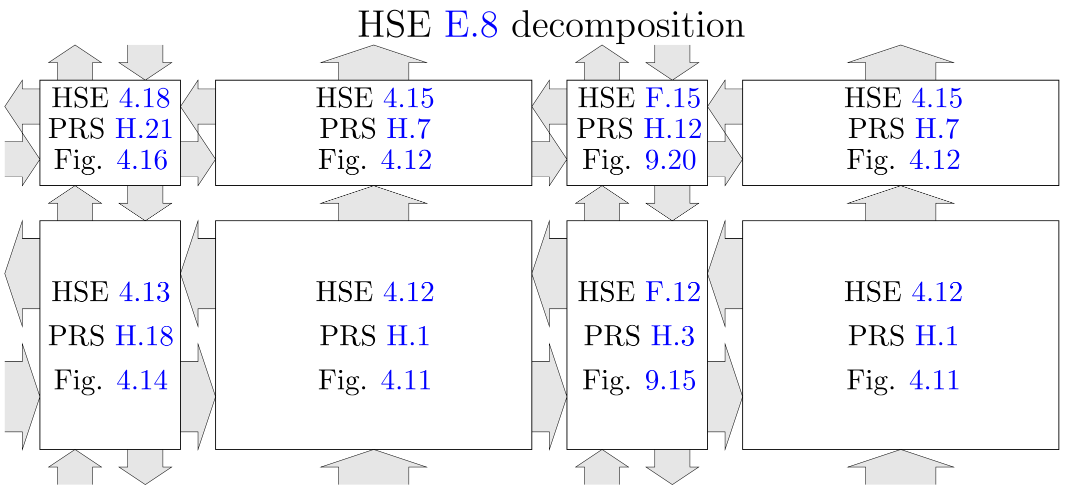


Figure 9.6: Floor decomposition of a PCEVHB nested read port

does not attempt to drive $R_o\uparrow$ during an outer partition access because IR_o is not directly guarded by $RC_{i,inner}$. We need to guarantee that $IR_o\downarrow$ has reset before enabling the outer partition to drive $R_o\uparrow$. To accomplish this, we introduce a new signal IR_o^v, which represents the validity of the inner partition's read rails, and obeys the sequencing $[IR_o]; IR_o^v\uparrow; [\neg IR_o]; IR_o^v\downarrow$. We explicitly add a $[\neg IR_o^v]$ guard before $ren_D\uparrow$ to guarantee that the inner partition has stopped driving $R_o\uparrow$ before allowing an outer partition access. This is very reminiscent of the locking technique to ensure pipelined mutual exclusion between the partitions' access to R_o. One necessary consequence of unlocking $ren\uparrow$ with $[\neg IR_o^v]$ is that we have to connect an inner partition signal all the way out to the outer partition's data interface. We later prove that this is the only necessary modification to the data interface cell. The partial HSE for the read data interface for nested data arrays is shown in Program F.10. For comparison, the original read data interface is shown in Program 4.13.

Since the inner partition input acknowledge $IRC_i^e\downarrow$ does not wait for $IR_o\uparrow$ we do not need a completion tree across all the read bit lines in the block of the inner partition. We guarantee that the inner partition input control IRC is not prematurely reset before IR_o is used with the ordering (where the s subscript denotes signals corresponding to the successor block): $IRC_i\uparrow \prec IR_o\uparrow \prec R_o\uparrow \prec R_o^v\uparrow \prec RC_i^e\downarrow \prec ren_{s,C}\downarrow \prec iren_{s,C}\downarrow \prec IRC_i\downarrow$. The advantage of enforcing this sequence is that we can reuse the data output validity completion tree of the outer partition instead of adding a completion tree just for the inner partition.

Finally, we describe the partial HSE for the nested read data interconnect, listed in Program F.12. From the outer partition's perspective, the interconnect component behaves like a 17th register because it drives $\_R\downarrow$ just like any other register cell. From the inner partition's perspective, the interconnect component serves the purpose of interfacing data to the 'environment' (the outer partition) with a handshake-like communication at bit granularity, not block granularity. The

HSE for this component is a result of a re-ordering transformation that enforces the following ordering: $ren_D\uparrow \prec iren_D\uparrow \prec iren_D\downarrow \prec ren_D\downarrow$. The discussion of this transformation can be found in the technical report [11]. After this final transformation, the HSE for the data component of the read port is listed in Program 9.3.

Program 9.3 HSE: data component of read port with nested data, after final transformations

```
$*[([R_o^e \wedge RC_i^e \wedge \neg IR_o^v]; ren_D\uparrow);$
  $[RC_{i,inner} \longrightarrow iren_D\uparrow; [IRC_i]; IR_o\uparrow; IR_o^v\uparrow; iren_D\downarrow; R_o\uparrow$
  $[] RC_{i,outer} \longrightarrow R_o\uparrow$
  $];$
  $[\neg R_o^e \wedge \neg RC_i^e]; ren_D\downarrow;$
  $([RC_{i,inner} \longrightarrow IR_o\downarrow; IR_o^v\downarrow$
   $[] RC_{i,outer} \longrightarrow$ **skip**
   $],$
   $R_o\downarrow)$
 $]$
```

We have derived the HSE specification of a nested data array with modest modifications to the outer partition data interface and the specification of the new nested interconnect cell for interfacing to inner partition reads. The same HSEs work for both the WAD and non-WAD nested read ports.

9.4.2 Non-WAD Write Data Nesting

Now we turn our attention to the data decomposition of the non-WAD core write port. Figures 9.7 and 9.8 give a visual outline of the final floor decomposition for the PCEVFB and PCEVHB reshufflings. This section discusses the decomposition of the bottom halves of these floor decompositions, the data array, interface and interconnect.

When we introduced width adaptivity to the write port in Chapter 5, we showed that the data component of the HSE requires no modifications (aside from writing one additional delimiter bit) because writing is unconditional and the control and data input acknowledges are always synchronized. Nesting adds a new dimension to our design space and requires careful attention, particularly in the case of width adaptivity. In this section, we focus on the data component of the floor decomposition for only the nested, non-width-adaptive write port. We will return to data component of the nested, width-adaptive write port after we have discussed the corresponding control component of the floor decomposition in Sections 9.4.6 and 9.4.6.

We start by analyzing the expansions for the nested non-width-adaptive write ports shown in Programs E.9 (full-buffered) and E.10 (half-buffered), whose data components are identical in HSE. As expected, write accesses to the outer partition

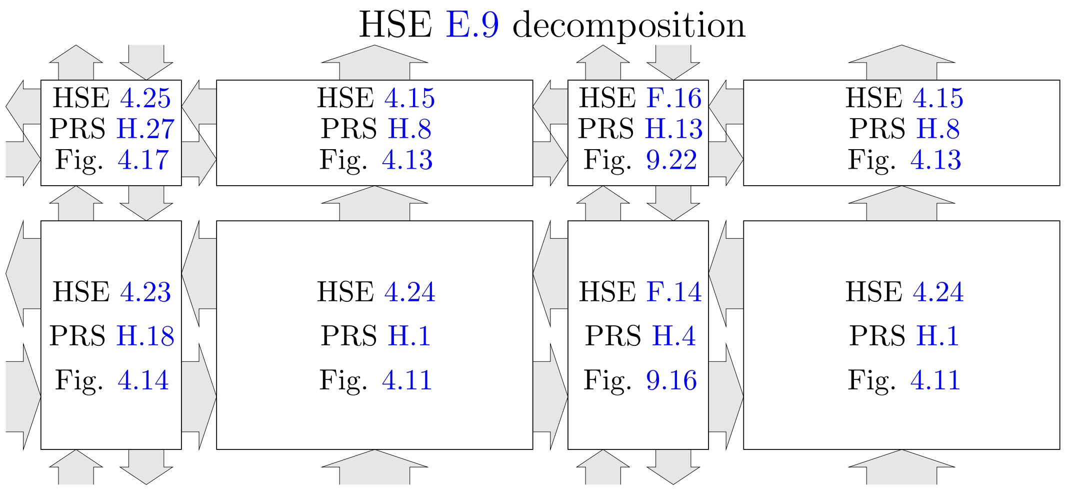


Figure 9.7: Floor decomposition of a PCEVFB nested write port

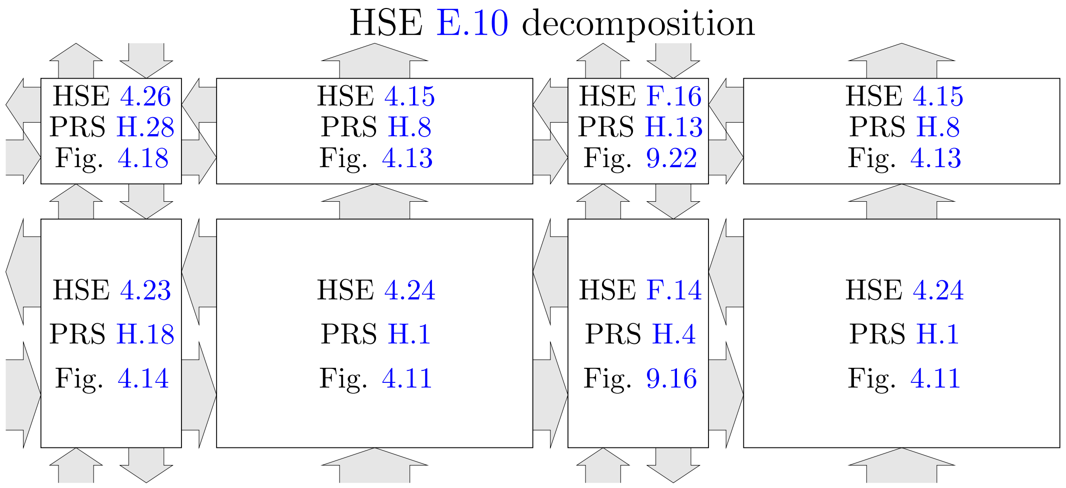


Figure 9.8: Floor decomposition of a PCEVHB nested write port

behave exactly the same as writes to a non-nested data array. Since $WC_{i,inner}$ implies IWC_i, the guard for $\langle write_{inner} \rangle$ is $[IWC_i \land IW]$, which is analogous to the guard for writing to the outer partition, $[WC_i \land W_i]$. We expose the write validity variables $\_wv$ and $\_iwv$ to signal when a write is complete to the outer and inner partitions, just as we did in Section 4.2.2. After we factor out the control propagation component, the data component is shown in HSE Program F.13, which covers the data write interface, inner and outer write data arrays, and nested interconnect.

On a write to the outer partition the sequence $IW\downarrow; \_iwv\uparrow$ is vacuous because the inner partition is never activated. To make writing the inner partition behave like writing to any other register in the outer partition, we impose several sequences on writes to the inner partition. Since $\_iwv$ signals that the write to the inner partition has completed, we can use $\neg\_iwv$ as a guard for $\_wv\downarrow$. The resetting of

$IW\downarrow$ must occur after write has become visible to the outer partition, hence it must wait until $\_wv\downarrow$. Since $IW\downarrow$ is a local channel, not controlled by an environment, we are free to reset it without having to wait for $W_i\downarrow$. Resetting $IW\downarrow$ is independent of the selected register in the inner partition, so $\neg IW$ may directly guard the resetting of $\_iwv\uparrow$, analogous to $\neg W_i$ guarding $\_wv\uparrow$ in the outer partition's data interface. However, the outer partition needs to wait until the inner partition has finished resetting before resetting its validity. Since the write data interface's only guarded event is resetting $\_wv\uparrow$, the only way we can check that the inner partition has reset (without adding more events) is by strengthening the guard of $\_wv\uparrow$ with $\_iwv$. The consequence of this requirement is that we need to connect $\_iwv$ (or some derivative thereof) across the outer partition's array and to the data interface for each port, but this is the only additional wire connection that is needed.[3] The HSE for the data interface for writing to a nested array is shown in Program F.11, and the HSE for the nested interconnect between the inner and outer arrays of the write port is shown in Program F.14.

Since each bit line completes its own handshake with the inner partition, we have guaranteed that $\_iwv$ is already checked in both directions, therefore we have eliminated the need for completion trees across $\_iwv$.

9.4.3 Non-WAD Read Control Nesting

We now present the floor decomposition of the control component of the non-WAD nested read port, depicted in the upper halves of Figures 9.5 and 9.6. In Section 4.2.1, the control component consisted of the handshake control and the control propagation array. With the nested transformation, the control array is broken up into an inner partition and an outer partition, and we introduce a nested control interconnect in between the partitions, as shown in Figure 9.3.

To get a clearer picture of what the control handshake is doing, we take Program E.7 and factor out the data component's actions, which leaves us with Program 9.4 for the full-buffer reshuffling. Recall that in both cases, we chose to full-buffer the inner partition's control handshake for more concurrency and better performance. We already understand that an access to the outer partition behaves exactly like a non-nested handshake, which is described in detail in Chapter 4. From the outer partition's perspective, we want an access to the inner partition to look like a outer partition access, to minimize or eliminate change to the handshake control. We are left to dissect the handshake for an access to the inner partition, which looks like simultaneous handshakes on channels RC and IRC.

One important difference between the control and data components is that the

[3] One could argue that wiring $\_iwv$ across the array is unnecessary since the reset guard, W_i, is already connected across the array, so responsibility for resetting $\_wv\uparrow$ may be shifted to the nested connect component, where $\_iwv$ is locally available. Doing so would add wire-delay on W_i, which slightly slows down the reset of $\_wv\uparrow$ on every access cycle through the outer partition.

Program 9.4 HSE: PCEVFB control component only of the data-independent read port with nested data

```
*[[$RC_o^e$]; $ren_C\uparrow$;
  [$RC_{i,inner} \longrightarrow$ [$IRC_o^e$]; $iren_C\uparrow$; [$IRC_i \wedge unlocked() \longrightarrow lock$; $IRC_o\uparrow$];
     ($IRC_i^e\downarrow$, ([$R_o$]; $RC_i^e\downarrow$))
  ▯$RC_{i,outer} \longrightarrow$ [$unlocked() \longrightarrow lock$; $RC_o\uparrow$]; [$R_o$]; $RC_i^e\downarrow$
  ];
  (([$\neg RC_o^e$]; $ren_C\downarrow$;
   [$RC_{i,inner} \longrightarrow$ [$\neg IRC_o^e$]; $iren_C\downarrow$; (($unlock$; $IRC_o\downarrow$), ([$\neg IRC_i$]; $IRC_i^e\uparrow$))
   ▯$RC_{i,outer} \longrightarrow unlock$; $RC_o\downarrow$
   ]),
   ([$\neg ren_D \wedge \neg ren_C \wedge \neg RC_i$]; $RC_i^e\uparrow$)
  )
 ]
```

nested control handshake is able to use two channel acknowledges, RC_i^e and IRC_i^e, on accesses to the inner partition. Using two acknowledges allows us to keep the following ordering: $ren_C\uparrow \prec iren_C\uparrow \prec ren_C\downarrow \prec iren_C\downarrow$, which is proven in the technical report [11], whereas the data component had to interchange $iren_D \prec ren_D\downarrow$. Since the guards for control propagation are equivalent in the outer and inner arrays, we can re-use the original unconditional control propagation elements in both partitions of the nested design. As we described in Chapter 8, we connect the root of the inner tree, IRC_o^v, to an input of the outer tree to form an unbalanced completion tree whose result is RC_o^v. This simplifies the floor decomposition by guaranteeing that $RC^v \Rightarrow IRC^v$ and $\neg RC^v \Rightarrow \neg IRC^v$, which ultimately allows us to re-use the *original* non-nested read handshake control in the outer partition of the nested design. Details of this argument can be found in the technical report [11].

9.4.4 WAD Read Control Nesting

Introducing width adaptivity to the nested read port will require a slight modification to the read handshake control and the control nested interconnect. However, we are able to preserve the interface that accesses to the inner partition should behave like accesses to the outer partition, but slower. In the case of control propagation for a WAD, nested read port access, the action sequences should mirror those of the non-WAD, nested read port, whose control propagation is unconditional. After we factor out the data component from the HSE Programs E.11, we are left with Program 9.5 for the full-buffered reshuffling. Figures 9.9 and 9.10 outline the floor decompositions for the PCEVFB and PCEVHB reshufflings respectively.

We observe that the guarded actions for **skip** in the terminations cases of both the inner and outer control arrays are equivalent to that of the non-nested

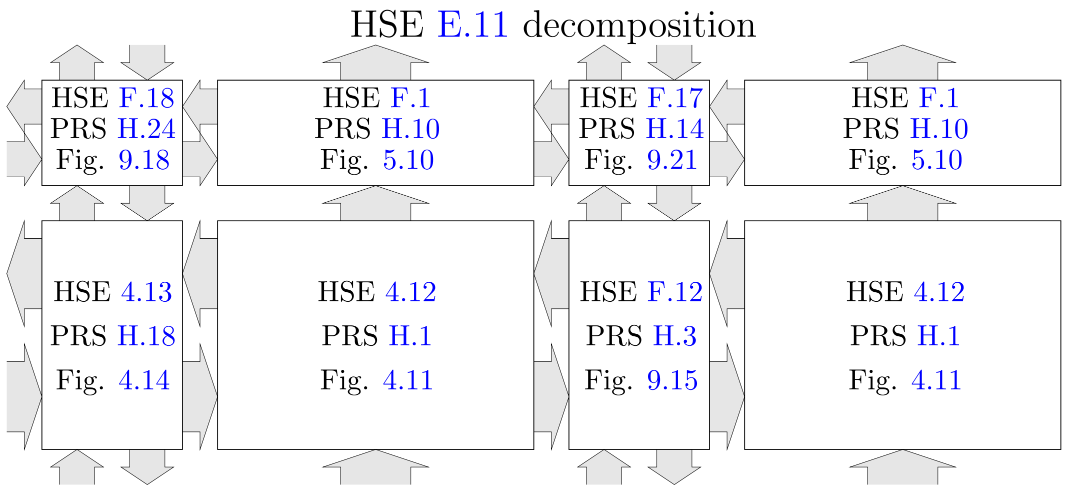


Figure 9.9: Floor decomposition of a PCEVFB WAD nested read port

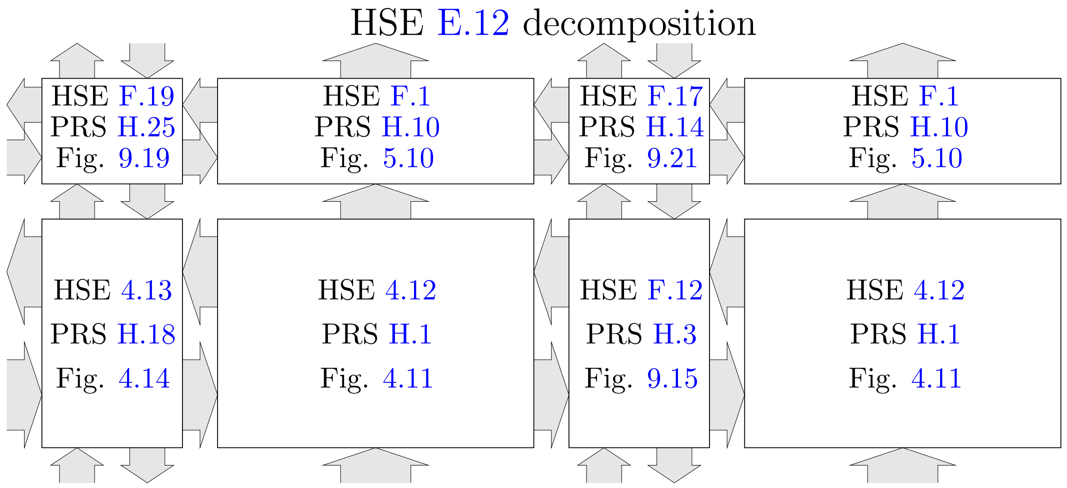


Figure 9.10: Floor decomposition of a PCEVHB WAD nested read port

counterpart, therefore we can re-use the non-nested WAD control propagation array in both partitions. The width-adaptive version of the read control nested interconnect is given in Program F.17. We introduce signals RC_o^f and IRC_o^f to represent the control termination cases. We guard RC_o^f with IRC_o^f, which makes terminating accesses to the inner partition appear like a terminating access to the outer partition. Since the interconnect controls a handshake on IRC, adding width adaptivity to the nested interconnect is analogous to making the original non-nested read handshake control width-adaptive in Chapter 5.

Finally, only the outer partition's handshake control component remains. Since the use of RC_o^f is now shared, we must guarantee exclusive use between the two partitions, just as we did for the $\_R$ in the data component. The only modification that is required is a check for $\neg IRC_o^f$ before $ren_C\uparrow$, analogous to checking $\neg IR_o^v$ before ren_D in the data component. The final HSE for the WAD, nested variation of

Program 9.5 HSE: PCEVFB control component of WAD read port with nested data

```
*[[$RC_o^e$]; $ren_C\uparrow$;
  [$RC_{i,inner} \longrightarrow$ [$IRC_o^e$]; $iren_C\uparrow$; [$IRC_i$];
    [$p(reg) \wedge unlocked() \longrightarrow lock; IRC_o\uparrow$ ▯ $t(reg) \longrightarrow$ **skip**];
    ($IRC_i^e\downarrow$, ([$R_o$]; $RC_i^e\downarrow$))
  ▯$RC_{i,outer} \longrightarrow$ [$p(reg) \wedge unlocked() \longrightarrow lock; RC_o\uparrow$ ▯ $t(reg) \longrightarrow$ **skip**];
    [$R_o$]; $RC_i^e\downarrow$
  ];
  (([$(p(reg) \wedge \neg RC_o^e) \vee t(reg)$]; $ren_C\downarrow$;
    [$RC_{i,inner} \longrightarrow$ [$(p(reg) \wedge \neg IRC_o^e) \vee t(reg)$]; $iren_C\downarrow$;
      (($unlock; IRC_o\downarrow$), ([$\neg IRC_i$]; $IRC_i^e\uparrow$))
    ▯$RC_{i,outer} \longrightarrow unlock; RC_o\downarrow$
    ]),
   ([$\neg ren_D \wedge \neg ren_C \wedge \neg RC_i$]; $RC_i^e\uparrow$))
 ]
```

the read handshake control is given in Program F.18 for the full-buffer reshuffling. The final wait on $\neg IRC_o^f \wedge \neg IRC_o^v$ in Program F.17 becomes unnecessary because $\neg IRC_o^v$ is checked by $RC_o^v\downarrow \prec RC_o^e\uparrow \prec ren_C\uparrow$, and $\neg IRC_o^f$ is now explicitly checked by ren_C in the outer partition's read handshake control. A detailed discussion of the various synchronization actions is given in technical report [11].

9.4.5 Non-WAD Write Control Nesting

The control component of the non-WAD nested write port is given in the upper half of HSE Program E.9 for the full-buffered version. It is worth pointing out that the control HSE closely resembles that of the full-buffered non-WAD nested read control component, shown in Program 9.4. If we factor out the respective guards of *wvc* and R_o, which only apply to outer partition accesses, we find that their remainders are in fact *equivalent*. Since the only differences arise in events that are in the handshake control of the outer partition, we can use the exact same floor decomposition for the control propagation array and the control nested connect. The same decomposition preserves the interface that writes to the inner partition appears like outer partition writes to the handshake control. Program F.16 shows the partial HSE for the control nested interconnect between the write control propagation arrays of the inner and outer partitions.

The write control propagation arrays are the same as those for non-nested write control propagation. We use the same technique of connecting the inner partition completion signal IWC_o^v as an input to the completion tree of the outer partition, so that $WC_o^v \Rightarrow IWC_o^v$ and $\neg WC_o^v \Rightarrow \neg IWC_o^v$. Since the outer partition's handshake control cannot distinguish between inner and outer write control propagation, and the control decomposition already guarantees correct ordering, we can use the original non-nested write handshake controls, HSE Programs 4.25

(full-buffer) and 4.26 (half-buffer), for the nested write port's handshake control.

9.4.6 WAD Write Control Nesting

Unconditional Outer Write-Enable

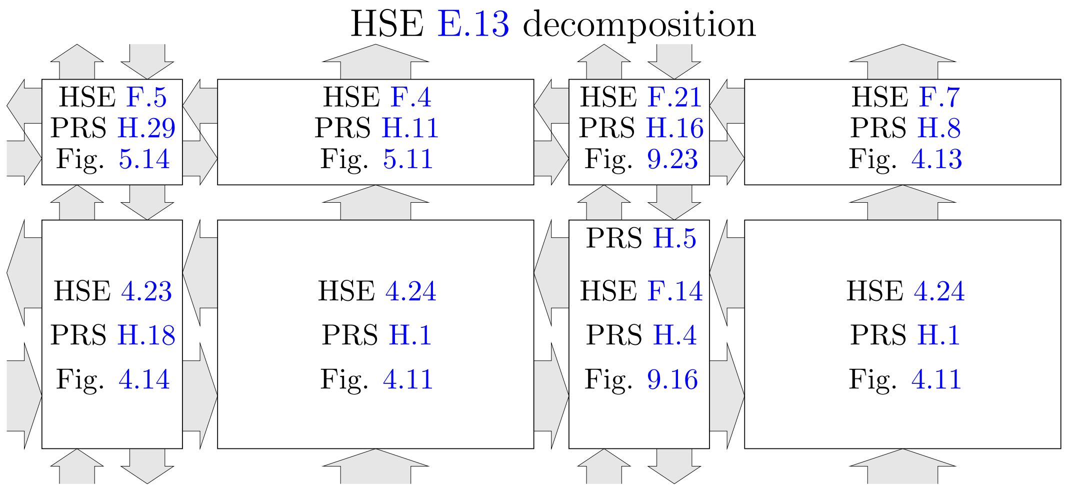


Figure 9.11: Floor decomposition of a PCEVFB WAD nested write port, (unconditional outer write-enable)

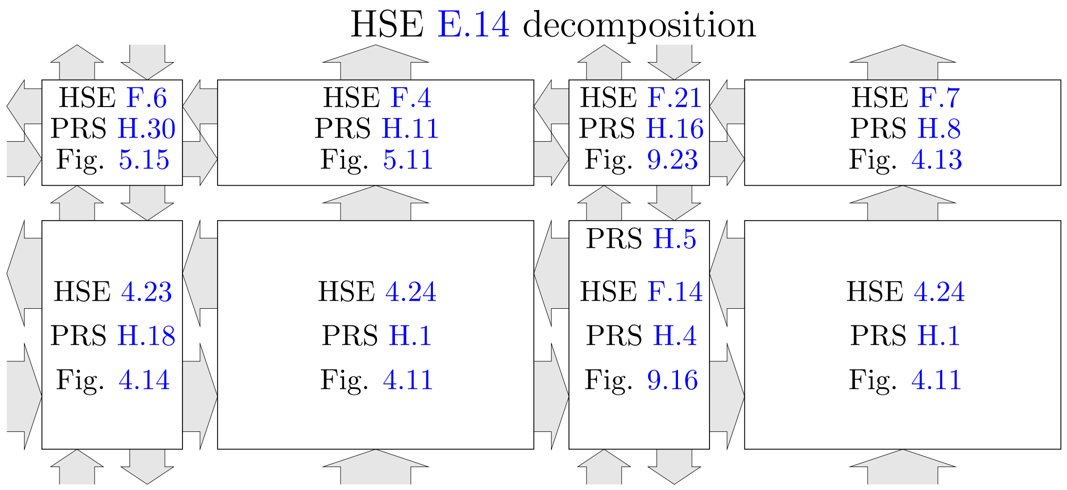


Figure 9.12: Floor decomposition of a PCEVHB WAD nested write port, (conditional outer write-enable)

We left off in Section 9.3.4 with the HSEs for WAD nested write ports shown in Program E.13 full-buffered with an unconditional outer write-enable, and Program E.15 full-buffered with a conditional outer write-enable. In both cases, we have chosen to only raise the inner write-enable in the control propagation case, and the handshake on IWC is full-buffered. Figures 9.11 and 9.12 show the outline of the floor decompositions for the PCEVFB and PCEVHB reshufflings of the WAD write port with unconditional outer write-enable.

In the case with unconditional outer write-enable and conditional inner write enable, the guards for control propagation differ between the inner and outer partition: $IWC_o\uparrow$ is not guarded by dIW^0 because *iwen* already implies propagation, whereas $WC_o\uparrow$ is guarded by dW^0 because *wen* does not imply control propagation. Thus, the HSE of the inner partition control propagation array is equivalent to that of the base design's with unconditional propagation, and the HSE of the outer partition control propagation array is equivalent to the WAD non-nested array with unconditional write-enable.

Since the nested interconnect performs the functions of a handshake control with respect to the inner partition, we can modify the existing handshake control HSE for conditional write-enableto obtain Program F.21. Using dIW^1 as a guard in the write control nested interconnect requires that $\neg IWC_i^e$ is checked before $dIW^1\downarrow$ is reset in the write data nested interconnect. The HSE for the modified data interconnect is shown in Program F.20.

As usual, This guarantees that $IWC^v \prec WC^v$ on writes to the inner partition. Since we have introduced no shared control variables between partitions, we have preserved the interface of making write access to the inner partition indistinguishable from writes to the outer partition from the perspective of the outer handshake control. Thus, we can re-use the WAD non-nested write handshake control for the outer partition of the nested design.

Conditional Outer Write-Enable

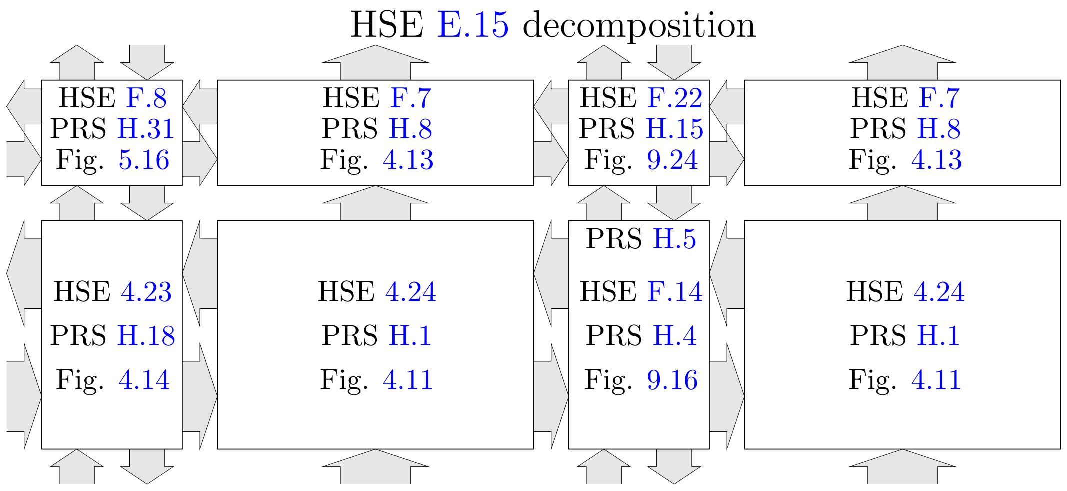


Figure 9.13: Floor decomposition of a PCEVFB WAD nested write port, (conditional outer write-enable)

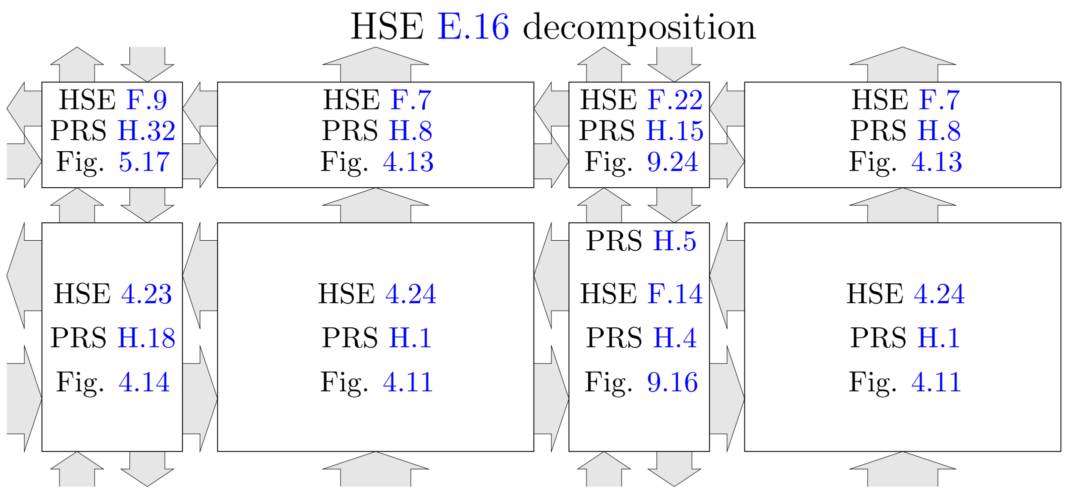


Figure 9.14: Floor decomposition of a PCEVHB WAD nested write port, (conditional outer write-enable)

Figures 9.13 and 9.14 outline the floor decompositions for the PCEVFB and PCEVHB reshufflings of the WAD write port with conditional outer write-enable. We have already argued that the inner control propagation array is equivalent to the write control array for the conditional write-enablevariation. Since the control nested interconnect performs the functions of the inner handshake control, we can adapt HSE of the WAD write handshake control for conditional write-enable to interface with the outer partition's handshake control. The resulting HSE for the control nested interconnect is Program F.22. Having preserved the interface of keeping write accesses to either partition indistinguishable, we can re-use the the WAD non-nested handshake control (without modification) as the outer partition's handshake control of the nested design.

9.5 Production Rules

The floor decomposition of the nested read and write ports revealed that the majority of partial HSE components required little or no change from the non-nested versions. To recapitulate the similarities, the following components are *exactly* the same as those of the non-nested designs from Chapters 4 and 5:

- read- and write-ported register cells that store internal state
- unconditional read and write control propagation array elements
- WAD read and write control propagation array elements
- handshake controls for unconditional read control propagation
- handshake controls for unconditional write control propagation

- handshake controls for WAD write control propagation
  (both conditional and unconditional outer write-enable)

In this section, we synthesize the new and modified HSEs into circuit production rules.

9.5.1 Read Data Nested Interconnect

In Section 9.4.1, we introduced the nested interconnect cell between the read ports of the inner and outer register arrays, whose partial HSE is listed in Program F.12. This HSE already exposes partial implementation in CMOS production rules by using $\_IR$ and IR to represent the internal inner data channel. We translated $RC_{i,inner}$ as the inner input control validity IRC_i^v. To guarantee that $iren_D\downarrow$ occurs before $\_R\downarrow$, we introduce its complement $iren_{\_D}$, which guards $\_R\downarrow$. $iren_{\_D}$ guarantees that $\_IR^v\downarrow$ has cut-off $ren_C\uparrow$ in the outer partition. Because we make no timing assumptions about $iren_{\_D}\downarrow$, we have to check every transition, so the most convenient place to check $\neg iren_{\_D}$ is before $IR\uparrow$. We have guaranteed stability by making $iren_{\_D}\uparrow$ the last possible transition in the interconnect cell before responding the the outer partition with $\_R\downarrow$. When the outer partition resets $ren_D\downarrow$, $\_IR\uparrow$ is allowed to reset, which leads to $\_IR^v\uparrow$, the final transition in the reset phase, which unlocks $ren_C\uparrow$ in the outer partition. The circuit for the nested interconnect is shown in Figure 9.15, and the PRS are also listed in Program H.3.

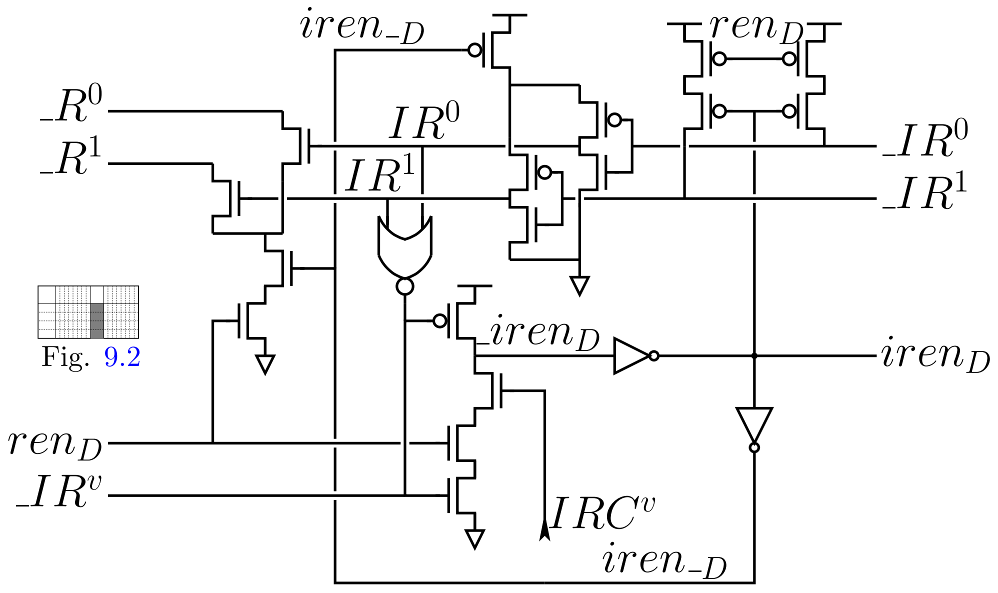


Figure 9.15: The interconnect circuit between inner and outer register partitions for a single nested read port

It is unfortunate that the data latency for a read access to the inner partition is up to nine transitions slower than a read accesses to the outer partition, a seemingly high penalty, but this is the price we must pay for QDI robustness. Remember that the idea behind nesting is that the most frequent accesses hit in the faster outer partition while less frequent accesses go through the slower inner partition. In the HSE floor decomposition, we have given some hints about where timing assumptions would be relatively safe and beneficial, should the need for a faster inner partition arise. The more ambitious (and hence, less conservative) designer is invited to explore the use of timing assumptions to make the inner partition read accesses faster while maintaining a high degree of robustness.

9.5.2 Write Data Nested Interconnect

We presented the nested interconnect cell between the write ports of the inner and outer register arrays in Section 9.4.2. The partial HSE is listed in Program F.14. We introduce the inverted dual-rail $\_IW$ and the active-high validity iwv to synthesize CMOS production rules. We allow $\_IW\uparrow$ to reset as soon as the outer partition sees validity $\_wv\downarrow$, which allows the inner partition to reset concurrently with the outer partition's handshake. Eventually the outer partition's data interface checks that the inner partition has reset $iwv\downarrow$ before requesting the next input. The rest of the production rules are straightforward from the HSE. The circuit is shown in Figure 9.16, and the PRS are listed in Program H.4.

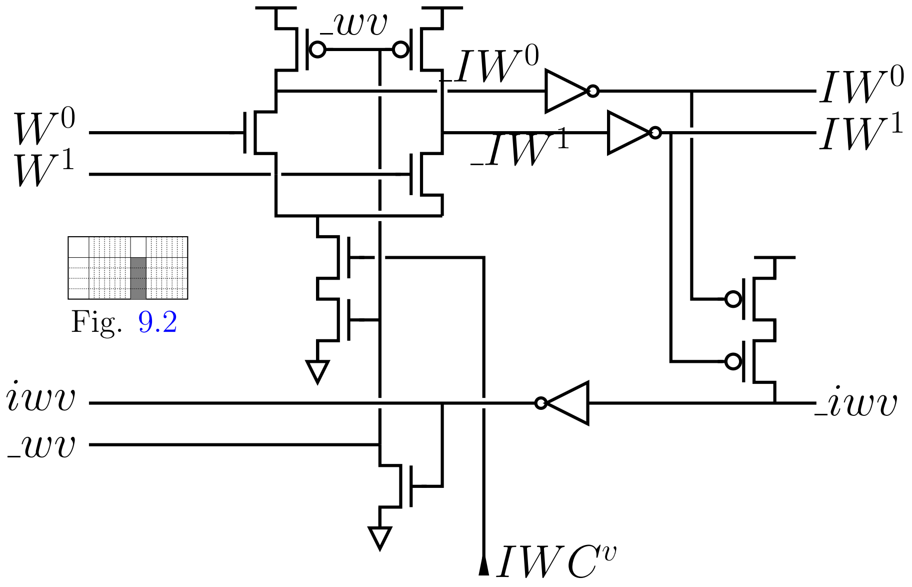


Figure 9.16: The interconnect circuit between inner and outer register partitions for a single nested write port

The time between $W\uparrow$ and $\_wv\downarrow$ on a bit-flipping write to the outer partition is roughly 3 transitions. The same delay for a bit-flipping write to the inner partition is roughly seven transitions, which is a less drastic than the difference in read latency between partitions. We will show in Section 9.6 how this impacts the cycle times.

In Section 9.4.6, we showed that the inner partition's delimiter bit of the write port dIW is shared to the width-adaptive control's nested interconnect, and therefore needed to wait for the inner acknowledge $IWC_i^e\downarrow$ before resetting $dIW\downarrow$, as specified in HSE Program F.20. This translates to a simple modification in the PRS, shown in Program H.6. Then in Section 9.4.6, we only needed to share one rail of the delimiter bit dIW^1, which translates to another slight modification in the PRS, shown in Program H.5.

9.5.3 Read/Write Nested Data Interface

We showed in Sections 9.4.1 and 9.4.2 that the data interface for the outer partition needed slight modification to accommodate nested read and write accesses in HSE Programs F.10 and F.11. For synthesis into CMOS production rules, we replace the $\neg IR_o^v$ guard with the signal $\_IR_o^v$, and the $\_iwv$ guard with $\neg iwv$. The resulting circuit is shown in Figure 9.17 and the PRS are listed in Program H.19.

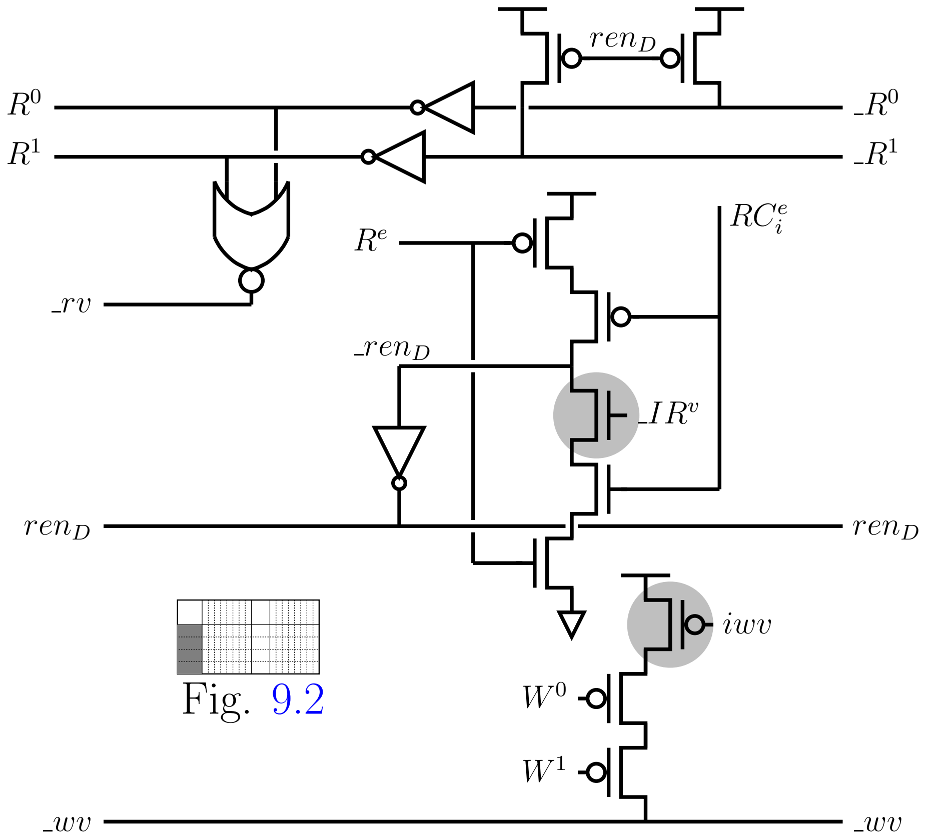


Figure 9.17: The data interface cell adapted to accommodate nested read and write register arrays, shown for a single port. Shaded transistors are modifications introduced by nesting.

9.5.4 WAD Nested Read Handshake Control

In Section 9.4.4, we concluded that the only modification required to convert a non-nested WAD read handshake control to the nested version is to strength the guard of $ren_C\uparrow$ with $\neg IRC_o^f$. This translates into adding a single series NFET in the production rule for $\_ren_C\downarrow$ whose guard is $\_ircof$ from the inner partition. The resulting WAD nested read handshake controls for the PCEVFB and PCEVHB reshufflings are shown respectively in Figures 9.18 and 9.19. Their PRSs are listed as Programs H.24 and H.24.

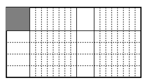


Fig. 9.2

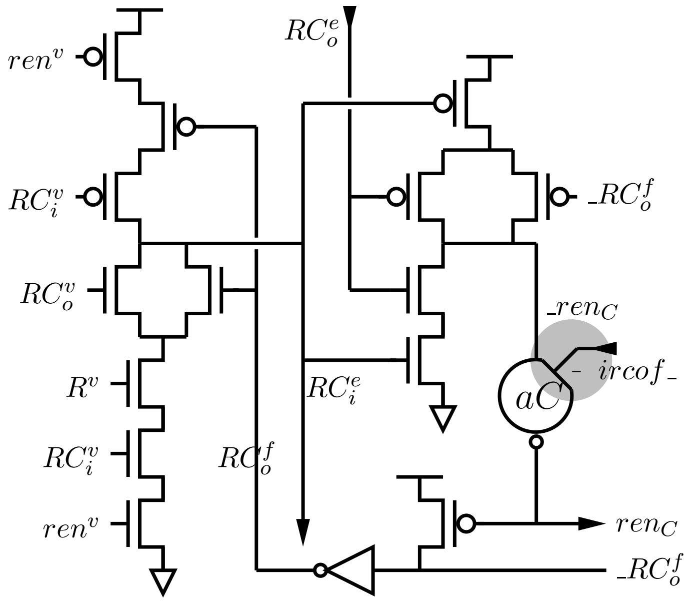


Figure 9.18: PCEVFB WAD nested read handshake control circuit. The shaded circuit is a modification introduced by WAD nesting.

9.5.5 Unconditional Read Control Nested Interconnect

In Section 9.4.3, we derived the partial HSE for the control interconnect component between the control propagation arrays of the inner and outer partitions for a non-WAD read port in Program F.15. Synthesis into CMOS production rules is straightforward after we introduce the intermediate signal $\_iren_C$. The inverter $\_iren_C$ is good for strongly driving $iren_C$, which is shared across the inner read control propagation array. The circuit is shown in Figure 9.20 and the PRS is listed in Program H.12.

9.5.6 WAD Read Control Nested Interconnect

In Section 9.4.4, we derived the partial HSE for the control interconnect component between the control propagation arrays of the inner and outer partitions for a WAD

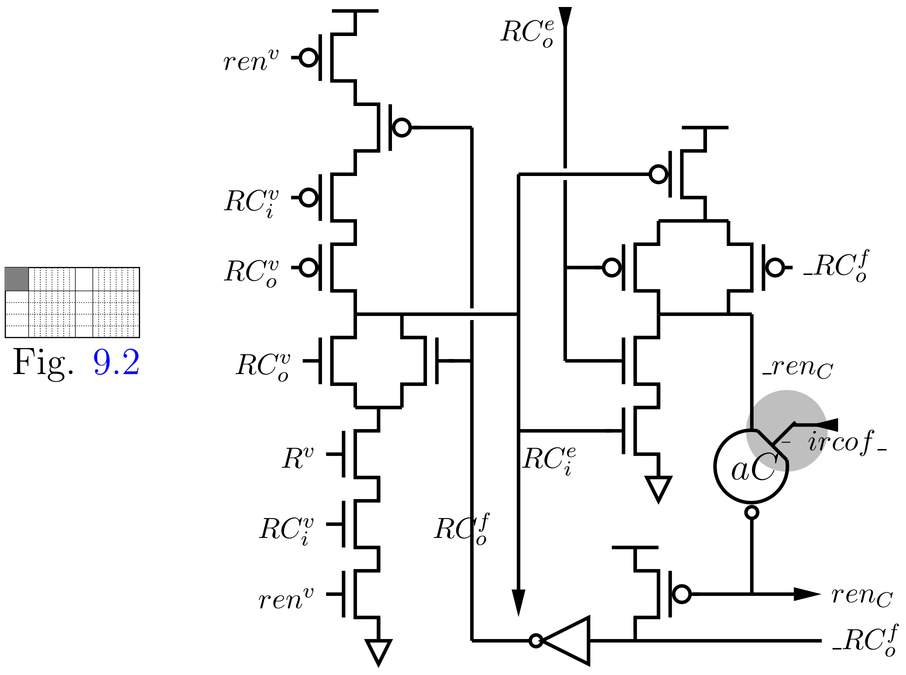


Figure 9.19: PCEVHB WAD nested read handshake control circuit. The shaded circuit is a modification introduced by WAD nesting.

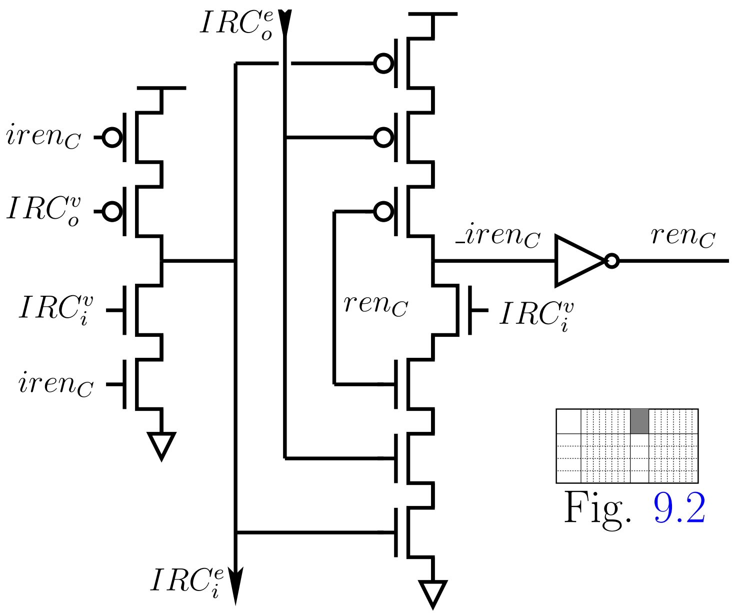


Figure 9.20: The control interconnect circuit between the inner and outer partitions' control propagation arrays for a non-WAD nested read port

read port in Program F.17. We need to introduce a few complementary signals to implement CMOS production rules. The circuit is shown in Figure 9.21 and the PRS is listed in Program H.14.

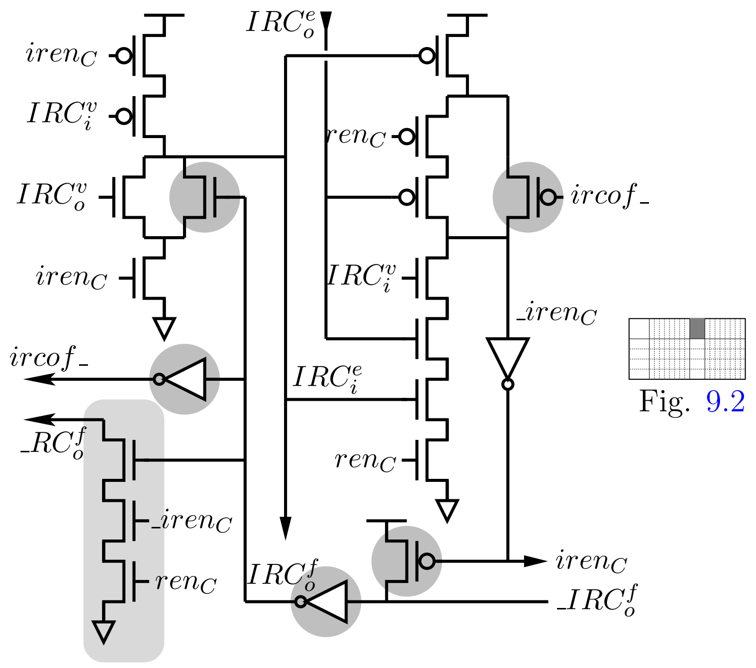


Figure 9.21: The control interconnect circuit between the inner and outer partitions' control propagation arrays for a WAD nested read port. Shaded circuits are modifications introduced by WAD.

$ircof\_$ is in inverted copy of IRC_o^f and is connected to the outer partition's handshake control to unlock $ren_C\uparrow$. We use $ircof\_$ to bypass waiting for the inner partition's output acknowledge IRC_o^e before $\_iren_C\uparrow$. On a control terminating access to the inner partition, $\_iren_C$ guarantees to the outer partition that $\neg ircof\_$ is stable before responding with $\_RC_o^f\downarrow$. A control propagating access to the inner partition behaves exactly like the non-WAD version of the control's nested interconnect in the previous subsection.

9.5.7 Unconditional Write Control Nested Interconnect

In Section 9.4.5, we observed that the HSE for the control interconnect between the inner and outer write control propagation array of the non-WAD write port was identical to that of the non-WAD read port. Therefore, their circuits should also be identical. We show the write control interconnect in Figure 9.22, and give the PRS in Program H.13.

9.5.8 WAD Write Control Nested Interconnect

Unconditional Outer Write-Enable

We left off in Section 9.4.6 with the partial HSE for the control interconnect component for the WAD nested write port with an unconditional outer write-enable,

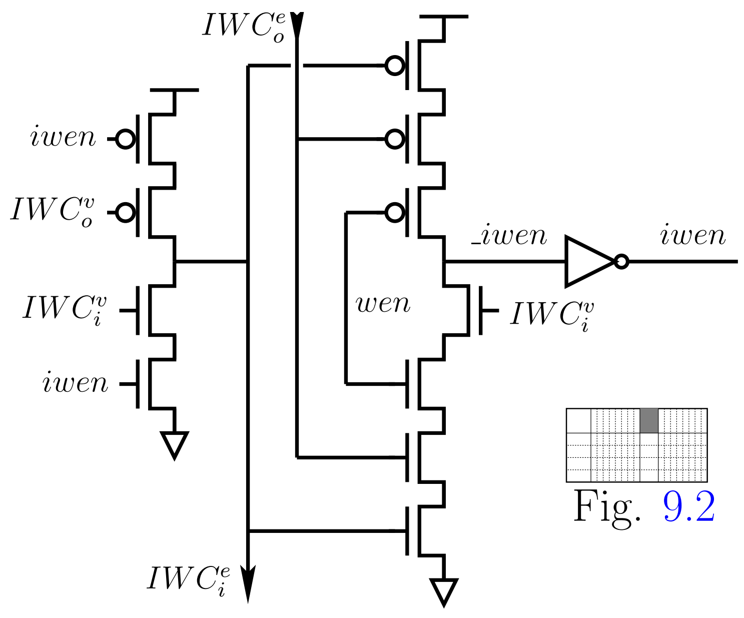


Figure 9.22: The control interconnect circuit between inner and outer partitions' control propagation arrays for a non-WAD nested write port

shown in Program F.21. Since the HSE was only a slight modification from the non-nested WAD write handshake control with conditional write-enable, we expect the production rules to look similar. As a result, the circuit synthesis is only a slight modification. The circuit is shown in Figure 9.23, and the PRS is listed in Program H.16.

Conditional Outer Write-Enable

In Section 9.4.6, we showed the partial HSE for the control interconnect component for the WAD nested write port with a conditional outer write-enable in Program F.22. Again, the HSE was only a slight modification from the non-nested WAD write handshake control with conditional write-enable. Therefore, the circuit synthesis is only a slight modification. The circuit is shown in Figure 9.24, and the PRS is listed in Program H.15.

9.6 Results

We have simulated all previous designs of the register core read and write ports, but with nested partitioning. We include results for the unbanked, nested core with 16 registers in the inner and outer partitions, and results for the banked, nested core with 8 registers in both partitions. In the tables in Appendix J, we refer to the former core as 32n (32-nested), and the latter as 16n (16-nested). Since register core banking and nesting are independent transformations, they can easily be combined to create really fast access and low energy registers. Figure 9.25 illustrates how

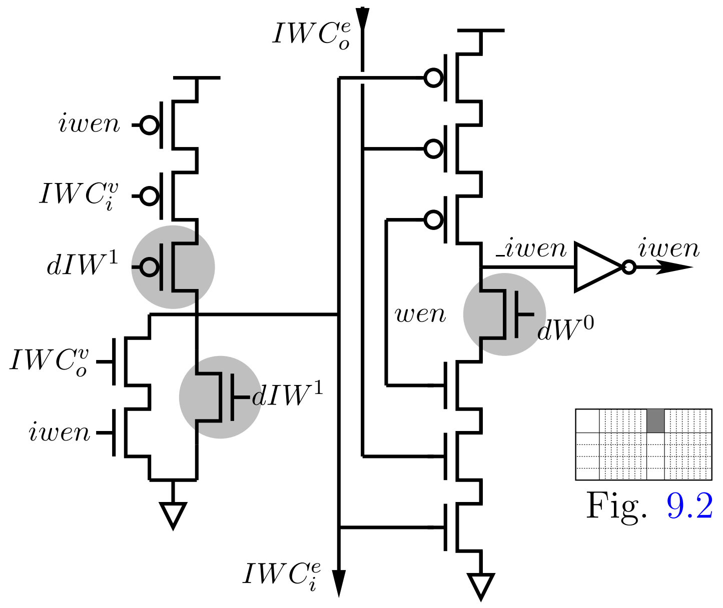


Figure 9.23: The control interconnect circuit between the inner and outer partitions' control propagation arrays for a WAD nested write port with an unconditional outer write-enable. The shaded circuits are modifications introduced by WAD.

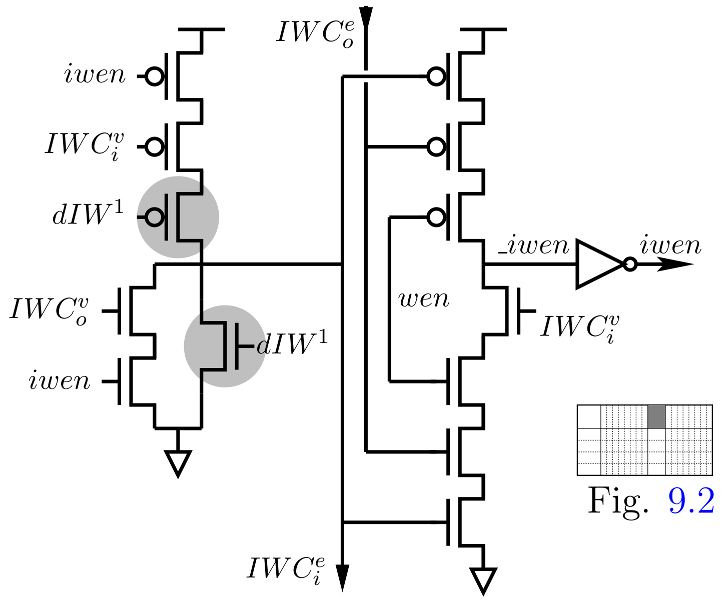


Figure 9.24: The control interconnect circuit between the inner and outer partitions' control propagation arrays for a WAD nested write port with a conditional outer write-enable. The shaded circuits are modifications introduced by WAD.

vertically pipelined, banked and nested read and write ports operate.

With nesting, we observe a greater difference in performance and energy between the partitions than we saw with just unbalancing completion trees. In each

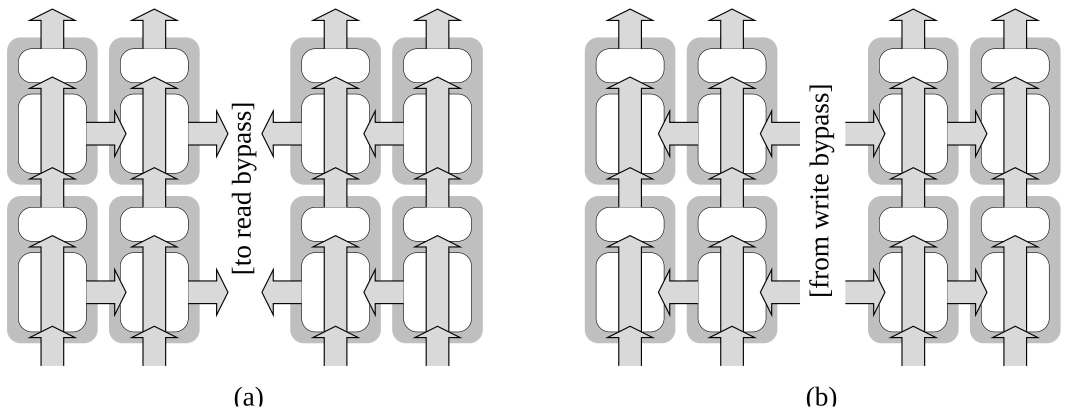


Figure 9.25: Vertically pipelined, banked and nested read and write ports.

subsection, we compute breakeven probabilities for when the average-case nested accesses is superior to uniform accesses. These probabilities are more significant than the corresponding probabilities from Chapter 8 because there is a greater gain in the fast case, and a higher penalty for the slow case. Bear in mind (from Section 8.1) that the most frequently used 16 out of 32 MIPS registers constituted around 99% of all dynamic read and write register accesses. We show that in all cases of reading and writing, the breakeven probabilities of our nested designs fall below this critical probability, which makes a case for nesting the asynchronous register files that we target.

We also evaluate the impact of adding an inner partition (of 16 registers) to an existing bank of 16 registers. (In Appendix J, this corresponds to comparing w=16 against w=32n.) Nesting has a nice property that the number of registers in a deeper partition has no impact on the performance and dynamic energy because the nested interconnect isolates load from the outer partition. Thus, the only negative impact on performance and energy from adding a partition is the constant cost of the nested interconnect. The number of registers in each partition will, however, affect the amount of static power dissipated, which is included in all of the reported energy figures. The numbers presented in these sections are collected in Table J.10 for reading and in Table J.21 for writing.

9.6.1 Area

The layout dimensions for the various components corresponding to Figure 9.2 are listed in Table 4.1. The width of the nested interconnect cell is 3.69 times the width of a register cell. This is the the only transistor area overhead associated with nesting. (The $\_IR^v$ and IW^v wires run over the outer cell array.) If one were to recursively nest multiple levels of register banks, each nesting boundary would incur this constant overhead in area.

9.6.2 Non-WAD Reading

Table 9.1 shows the performance and energy results for the half-buffer and full-buffer reshufflings of the core read port with a total of 32 registers, and Table 9.2 shows the same results for read ports with a total of 16 registers. The same results appear in Table J.2 in row entries with widths 32n and 16n, respectively. The relative performance and energy comparisons with the uniform-access read ports are computed in Table J.11, along with their breakeven probabilities. The baselines for comparison are non-nested core ports with the same number of total registers.

32-nested, half-buffer. The fast (outer) partition's read cycle time is 1.090 of the baseline uniform-access cycle time, and the slow (inner) partition's cycle time is 2.175 of the uniform-access cycle time. The fast cycle time comes as a surprise, because the fast partition is essentially half of the size of the non-nested design with single transistor modifications and a completion tree with less path effort. The reason for the extremely high penalty for slow accesses is because nearly an entire data handshake completes in the inner partition before the outer partition proceeds, which is a consequence of keeping the system strictly QDI. To reduce the penalty, one could make careful timing assumptions in the nested interconnect to avoid time-critical event-orderings, at the sacrifice of the robustness of delay insensitivity. The fast partition's read latency is 0.668 of the uniform-access read latency, while the slow read latency is 4.043 of the uniform-access read latency. For the average nested read latency to beat the uniform-access read latency, at least 90.2% of accesses must hit in the fast partition. Recall from Section 9.4.1 that the high penalty in latency is due to the fact that the inner bank must complete most of its cycle *before* it can reply with the data to the outer partition. The fast partition's cycle energy is 0.775 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.394 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 63.7% of accesses must hit in the fast partition.

32-nested, full-buffer. The fast partition's read cycle time is 1.009 of the baseline uniform-access cycle time, and the slow partition's cycle time is 2.106 of the uniform-access cycle time. Again, the fast partition cycle time is surprisingly slower than that of the non-nested cycle time. The fast partition's cycle energy is 0.746 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.350 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 58.0% of accesses must hit in the fast partition.

16-nested, half-buffer. The fast partition's read cycle time is 0.966 of the baseline uniform-access cycle time, and the slow partition's cycle time is 2.040 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 96.8% of accesses must hit in the fast partition. The fast partition's read latency is 0.733 of the uniform-access read latency, while the slow read latency is 5.180 of the uniform-access read latency. For the average

Table 9.1: Read-access performance and energy comparisons for the nested register file with 16 registers per partition. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	22	2.128	470.0	0.216	20.86	94.4
	46	4.247	235.4	1.308	37.51	676.6
full	20	1.880	531.9	0.216	19.84	70.1
	38	3.922	255.0	1.308	35.90	552.3

Table 9.2: Read-access performance and energy comparisons for the nested register file with 8 registers per partition. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	18	1.759	568.5	0.163	14.47	44.8
	38	3.714	269.2	1.149	24.98	344.6
full	16	1.630	613.5	0.163	14.09	37.4
	32	3.103	322.3	1.149	23.25	223.9

nested read latency to beat the uniform-access read latency, at least 94.0% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.909 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.569 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 86.2% of accesses must hit in the fast partition.

16-nested, full-buffer. The fast partition's read cycle time is 0.960 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.827 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 95.3% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.893 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.474 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 81.6% of accesses must hit in the fast partition.

Impact of adding a nested partition. Recall that the impact of nesting on bit line latency comes from the additional parasitic load of the nested interconnect on each read bit line $\_R$. For reading, we measured an increase in read latency of a bit line by -2.6%, or a *decrease* of 6 ps. This decrease in read latency is purely an artifact of measuring signal delay as the time difference between the last of multiple arriving inputs to the output transition, which does not model the Charlie Effect of transistors [54]. This difference is small enough to be considered noise in the data. The insignificant change in read latencies is very promising to asynchronous designs whose performance can be limited by the total forward latency through the datapath as opposed to the cycle time of local handshakes.

For the half-buffer reshuffling, adding an inner partition results in a 16.8% increase in cycle time and a 31.0% increase in energy per block per iteration. For the full-buffer reshuffling, adding an inner partition results in a 10.7% increase in cycle time and a 25.8% increase in energy per block per iteration.

9.6.3 Non-WAD Writing

Table 9.3 shows the performance and energy results for the half-buffer and full-buffer reshufflings of the core write port with a total of 32 registers, and Table 9.4 shows the same results for write ports with a total of 16 registers. The same results appear in Table J.12 in row entries with widths 32n and 16n, respectively. The relative performance and energy comparisons with the uniform-access write ports are computed in Table J.22, along with their breakeven probabilities. The baselines for comparison are non-nested core ports with the same number of total registers.

32-nested, half-buffer. The fast partition's write cycle time is 0.942 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.592 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 91.1% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.587 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.074 of the uniform-access cycle energy. The fast partition's write latency is 0.818 of the uniform-access write latency, while the slow write latency is 2.074 of the uniform-access write latency. For the average nested write latency to beat the uniform-access write latency, at least 85.5% of accesses must hit in the fast partition. For the average nested cycle energy to beat the uniform-access cycle energy, at least 15.2% of accesses must hit in the fast partition. The significant energy reduction in the outer partition shows that the energy of the data access dominated that of control propagation. If one can tolerate slow cycles, then much energy is reduced by simply isolating the capacitance of the inner partition with the nested interconnect.

32-nested, full-buffer. The fast partition's write cycle time is 0.938 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.492 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 88.9% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.583 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.042 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 9.2% of accesses must hit in the fast partition.

16-nested, half-buffer. The fast partition's write cycle time is 0.980 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.644

Table 9.3: Write-access performance and energy comparisons for the nested register file with 16 registers per partition. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	22	2.344	426.7	0.432	16.32	89.7
	46	3.960	252.5	1.095	29.86	468.3
full	20	2.293	436.1	0.432	16.01	84.2
	38	3.647	274.2	1.095	28.60	380.3

Table 9.4: Write-access performance and energy comparisons for the nested register file with 8 registers per partition. Upper numbers are the figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	20	2.136	468.1	0.375	10.71	48.9
	36	3.583	279.1	0.963	19.48	250.0
full	20	2.079	481.0	0.375	10.49	45.3
	30	2.964	337.4	0.963	17.68	155.3

of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 97.0% of accesses must hit in the fast partition. The fast partition's write latency is 0.899 of the uniform-access write latency, while the slow write latency is 2.310 of the uniform-access write latency. For the average nested write latency to beat the uniform-access write latency, at least 92.8% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.953 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.734 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 94.0% of accesses must hit in the fast partition.

16-nested, full-buffer. The fast partition's write cycle time is 0.981 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.399 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 95.6% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.928 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.565 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 88.7% of accesses must hit in the fast partition.

Impact of adding a nested partition. Recall that the impact of nesting on bit line latency comes from the additional gate load of the nested interconnect on each write bit line W. For writing, we measured an increase in write latency of a bit line by 3.6%, or 15 ps, which is a very low overhead for nesting.

For the half-buffer reshuffling, adding an inner partition results in a 7.5% increase in cycle time and a 45.3% increase in energy per block per iteration. For the full-buffer reshuffling, adding an inner partition results in a 8.3% increase in cycle time and a 41.7% increase in energy per block per iteration.

9.6.4 WAD Reading

Table 9.5 shows the performance and energy results for the half-buffer and full-buffer reshufflings of the WAD core read port with a total of 32 registers, and Table 9.6 shows the same results for WAD read ports with a total of 16 registers. The same results appear in Table J.3. The relative performance and energy comparisons with the uniform-access read ports are computed in Table J.11, along with their breakeven probabilities.

32-nested, half-buffer. The fast partition's read cycle time is 1.087 of the baseline uniform-access cycle time, and the slow partition's cycle time is 2.168 of the uniform-access cycle time. The read latencies for the WAD read port are the same as those for the non-WAD read port. The fast partition's cycle energy is 0.767 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.375 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 61.7% of accesses must hit in the fast partition.

32-nested, full-buffer. The fast partition's read cycle time is 1.011 of the baseline uniform-access cycle time, and the slow partition's cycle time is 2.042 of the uniform-access cycle time. The fast partition's cycle energy is 0.749 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.338 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 57.5% of accesses must hit in the fast partition.

Table 9.5: Read-access performance and energy comparisons for the WAD nested register file with 16 registers per partition. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	22	2.335	428.3	26.17	142.7
	46	4.659	214.6	46.89	1017.9
full	20	2.037	490.9	24.87	103.2
	38	4.114	243.1	44.40	751.6

Table 9.6: Read-access performance and energy comparisons for the WAD nested register file with 8 registers per partition. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	18	1.964	509.3	18.04	69.5
	38	4.081	245.1	31.22	519.9
full	16	1.802	554.8	17.66	57.4
	32	3.498	285.9	29.52	361.2

16-nested, half-buffer. The fast partition's read cycle time is 0.970 of the baseline uniform-access cycle time, and the slow partition's cycle time is 2.015 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 97.1% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.907 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.570 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 86.0% of accesses must hit in the fast partition.

16-nested, full-buffer. The fast partition's read cycle time is 0.963 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.869 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 95.9% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.900 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.505 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 83.5% of accesses must hit in the fast partition.

Impact of adding a nested partition. The impact of adding an inner partition of 16 register on the read latency is the same as that for the non-WAD read ports, in Section 9.6.2. For the half-buffer reshuffling, adding an inner partition results in a 15.3% increase in cycle time and a 31.6% increase in energy per block per iteration. For the full-buffer reshuffling, adding an inner partition results in a 8.8% increase in cycle time and a 26.8% increase in energy per block per iteration.

These results show that nesting partitions has roughly the same impact on performance and energy for width-adaptive read ports as it does on non-WAD read ports. The relative impacts of nesting on performance and energy between the half-buffer and full-buffer variations are similar. The absolute energy figures show that nesting combined with width adaptivity can potentially reduce read port energy by 2/3 if the majority of accesses hit in the fast partition.

9.6.5 WAD Writing, Unconditional Outer Write-Enable

Table 9.7 shows the performance and energy results for the half-buffer and full-buffer reshufflings of the WAD write port (unconditional outer write-enable) with a total of 32 registers, and Table 9.8 shows the same results for WAD write ports with a total of 16 registers. The same results appear in Table J.13. The relative performance and energy comparisons with the uniform-access read ports are computed in Table J.22, along with their breakeven probabilities.

32-nested, half-buffer. The fast partition's write cycle time is 0.943 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.583 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 91.1% of accesses must hit in the fast partition. The write latencies for the WAD write port are the same as those for the non-WAD write port. The fast partition's cycle energy is 0.552 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.027 of the uniform-access cycle energy. It is interesting to note that a slow access actually consumes *less* energy than a uniform access (of equal size). One possible reason is because the cell array is partitioned, the substrate leakage current in each partition is halved, thereby making each partition easier to staticize and keep signals away from the threshold voltage of the gates connected to the bit lines, which in turn, reduces the sub-threshold leakage of the affected nodes — leakage in divided cell arrays is easier to conquer.

32-nested, full-buffer. The fast partition's write cycle time is 0.943 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.459 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 88.9% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.557 of the baseline uniform-access cycle energy, and the slow partition's energy is 0.977 of the uniform-access cycle energy. Again, we see that the energy of an inner partition access can be lower that of the unpartitioned access.

Table 9.7: Write-access performance and energy comparisons for the WAD nested register file with 16 registers per partition, unconditional outer write-enable variation. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
half	22	2.453	407.6	19.35	116.5
	46	4.117	242.9	36.04	610.8
full	20	2.456	407.2	19.45	117.3
	38	3.801	263.1	34.11	492.7

Table 9.8: Write-access performance and energy comparisons for the WAD nested register file with 8 registers per partition, unconditional outer write-enable variation. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	20	2.245	445.4	12.60	63.5
	36	3.740	267.4	22.82	319.2
full	20	2.238	446.9	12.54	62.8
	30	3.211	311.5	21.18	218.3

16-nested, half-buffer. The fast partition's write cycle time is 0.981 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.634 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 97.1% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.957 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.733 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 94.5% of accesses must hit in the fast partition.

16-nested, full-buffer. The fast partition's write cycle time is 0.981 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.408 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 95.6% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.932 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.574 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 89.4% of accesses must hit in the fast partition.

Impact of adding a nested partition. The impact of adding an inner partition of 16 register on the write latency is the same as that for the non-WAD write ports, in Section 9.6.3. For the half-buffer reshuffling, adding an inner partition results in a 7.2% increase in cycle time and a 47.0% increase in energy per block per iteration. For the full-buffer reshuffling, adding an inner partition results in a 7.7% increase in cycle time and a 44.6% increase in energy per block per iteration.

9.6.6 WAD Writing, Conditional Outer Write-Enable

Table 9.9 shows the performance and energy results for the half-buffer and full-buffer reshufflings of the WAD write port (unconditional outer write-enable) with a total of 32 registers, and Table 9.10 shows the same results for WAD write ports with a total of 16 registers. The same results appear in Table J.14. The relative performance and energy comparisons with the uniform-access read ports are computed in Table J.22, along with their breakeven probabilities.

32-nested, half-buffer. The fast partition's write cycle time is 0.940 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.585 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 90.7% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.554 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.032 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 6.7% of accesses must hit in the fast partition, which is a sure win in energy, even if the partition accesses are evenly distributed.

32-nested, full-buffer. The fast partition's write cycle time is 0.943 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.453 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 88.9% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.539 of the baseline uniform-access cycle energy, and the slow partition's energy is 0.950 of the uniform-access cycle energy, thus all accesses will benefit in decreased energy compared to the non-nested write port.

Table 9.9: Write-access performance and energy comparisons for the WAD nested register file with 16 registers per partition, conditional outer write-enable variation. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	24	2.403	416.2	19.06	110.0
	46	4.052	246.8	35.50	582.8
full	22	2.486	402.2	19.43	120.1
	38	3.831	261.0	34.25	502.6

Table 9.10: Write-access performance and energy comparisons for the nested register file with 8 registers per partition, conditional outer write-enable variation. Upper numbers are figures for the faster outer partition.

buf	trans./ cycle	cycle (ns)	freq. (MHz)	energy/cycle (pJ)	$E\tau^2$ ($10^{-30}Js^2$)
half	22	2.197	455.2	12.41	59.9
	36	3.656	273.5	22.67	303.0
full	20	2.268	440.9	12.60	64.8
	30	3.203	312.2	21.20	217.5

16-nested, half-buffer. The fast partition's write cycle time is 0.980 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.630

of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 96.9% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.953 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.739 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 94.0% of accesses must hit in the fast partition.

16-nested, full-buffer. The fast partition's write cycle time is 0.978 of the baseline uniform-access cycle time, and the slow partition's cycle time is 1.381 of the uniform-access cycle time. For the average nested cycle time to beat the uniform-access cycle time, 94.5% of accesses must hit in the fast partition. The fast partition's cycle energy is 0.933 of the baseline uniform-access cycle energy, and the slow partition's energy is 1.569 of the uniform-access cycle energy. For the average nested cycle energy to beat the uniform-access cycle energy, at least 89.4% of accesses must hit in the fast partition.

Impact of adding a nested partition. The impact of adding an inner partition of 16 register on the write latency is the same as that for the non-WAD write ports, in Section 9.6.3. For the half-buffer reshuffling, adding an inner partition results in a 7.1% increase in cycle time and a 46.3% increase in energy per block per iteration. For the full-buffer reshuffling, adding an inner partition results in a 7.2% increase in cycle time and a 43.8% increase in energy per block per iteration.

These results show that nesting partitions has the same impact on performance and energy for width-adaptive write ports as it does on non-WAD write ports. The relative impacts on performance and energy between the half-buffer and full-buffer variations are similar. The absolute energy figures show that nesting combined with width adaptivity can potentially reduce write port energy by 2/3 if the majority of accesses hit in the fast partition.

9.7 Summary

In this final chapter, we have shown the feasibility of quasi-delay insensitively (QDI) partitioning a register bank for non-uniform accesses *without* changing the interconnect requirements, having built upon the basic idea from Chapter 8. Not only did nesting create non-uniform cycle time accesses, but nesting also introduced non-uniform read and write latencies, which may result in an average-case speedup of the forward path of data through the datapath. The intention of nesting is to exploit typical register usage frequencies to speed up the most common access while allowing less frequently used registers to slow down.

We have derived the necessary circuit transformations from extensions of the read and write port CHP, which were lowered into handshaking expansions, and finally synthesized into production rules. By directing all accesses to the inner partition through the outer partition, reusing unbalanced control validity completion trees, and using 'lightweight' nested interconnects, the inner partition behaves like a slower register to the outer partition, thus we were able to *minimize* (sometimes eliminate) the changes to the original circuits for the non-nested designs. Had we implemented full handshakes in the interconnect, more modifications to the outer partition's control would have been required, which may have potentially reduced the speedup of fast accesses.

For all read and write ports in the design space, we have evaluated the differences in performance and energy introduced by nested partitioning, and computed the breakeven probabilities that indicate when nesting is favorable. We found nesting to be always beneficial because the breakeven probabilities never exceeded the critical probability of 99%.

We measured the impact of adding a partition of registers to an existing bank of registers. We conclude that one can add an *arbitrary* size inner partition and while incurring only a small, constant performance penalty on the critical outer partition. In other words, the inner partition of a nested asynchronous register file may grow arbitrarily large while maintaining a constant performance level for a fixed subset of registers (in the outer partition) *with no external complexity*!

The performance penalty for accessing the slow partition is relatively high because of the conservativeness of the nested interconnect circuits that arises from the QDI timing model. More aggressive designs can leverage timing assumptions to reduce the slow access penalties, which would make make nesting even more appealing for achieving average-case speedup for both throughput and latency. However, in the interest of reducing energy, large register banks will benefit more from nesting because the energy is dominated by the bit line capacitances.

Finally, we have demonstrated that nesting is completely compatible with all previous register file techniques: vertical pipelining, width adaptivity, and banking. The property that nesting does not increase the channel interface (and external interconnect requirements) makes it very appealing as a local optimization that can be applied on top of all other optimizations.

Chapter 10

Conclusion

10.1 Recapitulation

We have completed a lengthy tour of asynchronous register file design. Our journey began with the basic background for QDI asynchronous design and models for the expected performance and energy consumption of register files. We then worked through a specific example of how a typical asynchronous register file is specified and decomposed into fine-grain concurrent processes, which can then be compiled into robust circuits using known templates. For improved performance, we pipelined the *BYPASS* and *CORE* processes to operate on four bits of data per pipeline stage. To preserve the ordering and mutual-exclusion semantics guaranteed by the *CONTROL* for reading and writing, we used pipeline-locking in the control pipeline stages of the *CORE*. After writing out the handshaking expansions (HSE) that describe the communication actions on the control and data channels, we synthesized circuit production rules for the first version of the pipelined asynchronous register file. The results for our initial design (including the banked version) from Chapter 4 served as the baseline for comparison with other versions of register files presented in the rest of the thesis. The first part of the thesis contributes a reasonably detailed design of a QDI asynchronous register file.

In Chapter 5, we introduced width-adaptive versions of the register file that saved considerable energy in the average case by reducing the number of blocks that switch and communicate. Width adaptivity is entirely transparent to the register file control and results in little performance loss from the additional delimiter bit per block. The design of the width-adaptive register file is the second major contribution of this thesis.

We discussed some alternative implementations of the *BYPASS* that reduce the number of access to the core: consuming and producing register zero operations in the *BYPASS* (Chapter 6) and suppressing redundant copies from the core by copying operands at the bypass (Port Priority Selection, in Chapter 7). These alternatives require only simple modifications to the *CONTROL* processes.

We presented various register core organizations for improving performance and energy. We have already seen that banking register files results in faster cycle times and latencies and reduced energy because of reduced bit-line loads and reduced path effort in the handshake cycle. However, the cost of banking lies in the use of more channels, which may require multiplexing outside of the core. For a small number of banks, the cost of multiplexing may be absorbed in the operand read bypass.

When one cannot afford to add more banks, one can leverage typical register use distributions to make non-uniform access registers with the purpose of accelerating commonly used registers while allowing infrequently used registers to be slower. We first introduced unbalanced control completion trees (with a fast path and a slow path) to demonstrate the potential for average case reduction of cycle time with *minimal* modifications and computing breakeven probabilities to assess when unbalancing would be beneficial. However, unbalancing completion trees alone did not offer much speedup because the read and write latencies were not affected.

To improve cycle times and latencies further, we designed *nested* register files, which effectively isolated the load for half of the register array into its own inner partition, which was connected to the fast outer partition through a delay-insensitive interconnect. The interconnect was designed to make accesses to the inner partition behave just like any other register (except for timing) from the outer partition's perspective. The delay-insensitive nature allows one to arbitrarily connect deeper nests of register banks with no complexity in retiming. Nesting introduced a much larger potential for reducing cycle time and latency through accessing the fast registers, but also induced a greater penalty for accessing slow registers. Significant energy can be saved by nesting because much of energy consumed is due to the register array bit-line capacitance, which has been cut in half for the outer partition. Not only is nesting useful for achieving average-case improvement, but it also means that one can arbitrarily extend a register file without slowing down accesses to a fixed subset of registers; the addition of a large slower partition has a only a *small constant* impact on the fast outer partition! The most important point to take away from this thesis is that unbalancing completion trees and delay-insensitive nesting have the design advantage of requiring *no additional external interconnects* to the core, i.e., they are entirely local transformations that incur no complexity in retiming or interconnect. Non-uniformly nested register-file designs are the final and most significant contribution of this thesis. Finally, we have demonstrated that all of the key transformations we have shown in this thesis can be synthesized in *any* combination.

Designing around a QDI asynchronous timing model requires one to re-assess techniques that may not otherwise be considered in the synchronous domain. One lesson we have learned from the thesis is that the modular nature with which we design asynchronous register files allows us to isolate and combine circuit modifications introduced by various transformations.

10.2 Choice

```
Consult your local computer engineer to see which
asynchronous register file is right for you.
```

After all this work, it is only natural to ask, "So which among all surveyed asynchronous register files is the best?" The answer is a resounding "Really, it depends." It depends on the metric of interest, the overall architecture, and the characteristics of the applications. For minimal area (maximal density) and simplicity, an non-vertically-pipelined register file as described in Chapter 2 is the most appealing. For energy only, an unpipelined register file has no control propagation overhead, however a vertically pipelined, *width-adaptive* register file has greater potential to reduce the number of bits communicated on the datapath. Banking uniformly reduces the energy per port operation, while nesting reduces the energy of accesses to the fast partition. For performance, one needs to determine whether the cycle time or the read latency of register file accesses is more critical. Vertical pipelining reduces the cycle time of the data handshakes, banking uniformly reduces the cycle time of control handshakes, and unbalancing control completion trees and nesting reduces the best-case cycle time of control handshakes. Banking reduces all read latencies uniformly, whereas (non-uniform) nesting reduces the best-case read latencies. Deciding whether or not to nest depends on the register usage pattern of a given application, which depends partially on the register allocator of the compiler if registers are statically assigned. Designing for a joint metric such as energy efficiency ($E\tau^2$) requires a more careful assessment of the tradeoff between energy and performance.

10.3 Future Work

Our survey of asynchronous register files was confined to a very tight design space. After showing the initial base design, we restricted ourselves to using the same register cell by re-using the same general floor decomposition of the same underlying handshake. For the basic register cell alone, there are many performance-, density-, and energy-improving techniques common to synchronous register files that we have not addressed. A follow-up study of how analog circuit techniques may be used with asynchronous circuits may reveal interesting combinations that compound the benefits we have shown in this thesis.

Since we have designed our register files around the one of the most conservative (arguably overkill) timing models, QDI, an important question to ask may be: what timing assumptions may offer improved performance while sustaining maximal robustness? Such timing assumptions would impact our decomposition of handshaking expansions into production rules by replacing many production rules with implicit timing-based guarantees that specific events in the handshake sequence have occurred. For example, production rules for write-validity $\_wv$ in each cell may be removed with the assumption that by the time the write control and data validity signals have arrived, the write to the corresponding cell has completed, similar to that used in the design of an asynchronous DRAM [10]. We have hinted in Chapter 9 where similar timing assumptions would benefit the design without compromising robustness.

Much of our argument for non-uniform access register files pivots around the probability distribution of typical logical register accesses. The next step would be to ask whether or not register nesting would be beneficial to architectures that dynamically rename logical registers to physical registers. Many questions arise regarding the potential use of non-uniform access registers: What allocation and de-allocation strategies would skew register usage distributions to favor a small subset of physical registers? If we expose such non-uniformity as part of the ISA, how can a compiler optimize the use of non-uniform registers? Can the hardware or software schedule accesses to slower registers earlier to hide their latency? The fact one may add an arbitrary number of registers to an inner partition of a nester register file without slowing the outer partition invites architectural studies on potential uses for larger and slower register banks, particularly for asynchronous designs. Finally, at the architectural level, we have demonstrated nesting as a method for implementing robust asynchronous non-uniform access register files with *no* additional control complexity or retiming outside of the core — inconceivable for synchronous designs with multi-cycle register accesses. Non-uniform register accesses in asynchronous microprocessor designs should promote architecture and application studies for heterogenous register-file organizations.

Appendix A

Summary of CHP Notation

The CHP notation we use is based on Hoare's CSP [17]. A full description of CHP and its semantics can be found in [29]. What follows is a short and informal description.

- Assignment: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := true$, and $a\downarrow$ for $a := false$.
- Selection: $[G1 \rightarrow S1 \, [] \, ... \, [] \, Gn \rightarrow Sn]$, where Gi's are boolean expressions (guards) and Si's are program parts. The execution of this command corresponds to waiting until one of the guards is *true*, and then executing one of the statements with a *true* guard. The notation $[G]$ is short-hand for $[G \rightarrow skip]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \rightarrow S1 \, [] \, ... \, [] \, Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the *true* guards and executing the corresponding statement, repeating this until all guards evaluate to *false*. The notation $*[S]$ is short-hand for $*[true \rightarrow S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is *true* iff a communication over channel X can complete without suspending.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.
- Simultaneous Composition: $S \bullet T$ both S and T are communication actions and they complete simultaneously.

Appendix B

Bypass CHP

This appendix includes detailed CHP for many variations of the register file *BYPASS* described throughout the thesis.

Unless otherwise specified, the CHP programs for *BYPASS*.*BPZ*[1] (and the respective *BLOCK* pipeline versions) are equivalent to those of *BYPASS*.*BPZ*[0] with their own set of local variables, and the CHP programs for *BYPASS*.*BPZY* are equivalent to those of *BYPASS*.*BPZX*.

B.1 Base Design

The decomposition for the *BYPASS* of the base design is discussed in Section 2.4.

Program B.1 CHP: register file writeback bypass

```
$BYPASS.BPZ[0] \equiv$
    $*[BPWB[0]?w0, BPZX[0]?zx0,$
        $BPZY[0]?zy0, Z[0]?z0;$
      $[w0 \longrightarrow W[0]!z0 \ [\!] \ \textbf{else} \longrightarrow \textbf{skip}],$
      $[zx0 \longrightarrow ZX[0]!z0 \ [\!] \ \textbf{else} \longrightarrow \textbf{skip}],$
      $[zy0 \longrightarrow ZY[0]!z0 \ [\!] \ \textbf{else} \longrightarrow \textbf{skip}]$
    ]
```

Program B.2 CHP: register file read bypass

```
$BYPASS.BPZX \equiv$
   $*[BPX?mx;$
     $[mx = ”z0” \longrightarrow ZX[0]?x$
     $[]mx = ”z1” \longrightarrow ZX[1]?x$
     $[]mx = ”core” \longrightarrow R[0]?x$
     $];$
     $X!x$
    $]$
```

B.2 Vertically Pipelined

Vertically pipelining of the *BYPASS* is discussed in Section 3.4.

Program B.3 CHP: pipelined register file read bypass

```
$BYPASS.BPZX.BLOCK \equiv$
  *[$BPX_i?mx$;
    $BPX_o!mx$,
    ([$mx =$ "$z0$" $\longrightarrow ZX[0]?x$
    [] $mx =$ "$z1$" $\longrightarrow ZX[1]?x$
    [] $mx =$ "$core$" $\longrightarrow R[0]?x$
    ];
    $X!x$)
   ]
```

Program B.4 CHP: pipelined register file writeback bypass

```
$BYPASS.BPZ[0].BLOCK \equiv$
  *[$BPWB_i[0]?w0, BPZX_i[0]?zx0,$
      $BPZY_i[0]?zy0, Z[0]?z0$;
    [$w0 \longrightarrow W[0]!z0$ [] else $\longrightarrow$ skip],
    [$zx0 \longrightarrow ZX[0]!z0$ [] else $\longrightarrow$ skip],
    [$zy0 \longrightarrow ZY[0]!z0$ [] else $\longrightarrow$ skip],
    $BPWB_o[0]!w0$,  $BPZX_o[0]!zx0$,  $BPZY_o[0]!zy0$
   ]
```

B.3 Width-Adaptive

The width-adaptive CHP-level transformation is discussed in Section 5.3.

Program B.5 CHP: WAD read bypass

```
$BYPASS.BPZX \equiv$
    $*[BPX_i?mx;$
      $[mx = \text{"}z0\text{"} \longrightarrow ZX[0]?x$
      $[] mx = \text{"}z1\text{"} \longrightarrow ZX[1]?x$
      $[] mx = \text{"}core\text{"} \longrightarrow R[0]?x$
      $];$
      $(X!x,\ \ [p(x) \longrightarrow BPX_o!mx\ \ [] t(x) \longrightarrow \textbf{skip}])$
    $]$
```

Program B.6 CHP: WAD writeback process

```
$BYPASS.BPZ[0] \equiv$
    $*[BPWB_i[0]?w0, BPZX_i[0]?zx0,$
        $BPZY_i[0]?zy0, Z[0]?z0;$
      $[w0 \longrightarrow W[0]!z0\ \ []\ \ \textbf{else} \longrightarrow \textbf{skip}],$
      $[zx0 \longrightarrow ZX[0]!z0\ \ []\ \ \textbf{else} \longrightarrow \textbf{skip}],$
      $[zy0 \longrightarrow ZY[0]!z0\ \ []\ \ \textbf{else} \longrightarrow \textbf{skip}],$
      $[p(z0) \longrightarrow BPWB_o[0]!w0, BPZX_o[0]!zx0, BPZY_o[0]!zy0$
      $[] t(z0) \longrightarrow \textbf{skip}$
      $]$
    $]$
```

B.4 Register Zero

The modification for sourcing zero from the read bypass is discussed in Section 6.2.1.

Program B.7 CHP: register file read bypass with source for hard-wired zero

```
$BYPASS.BPZX \equiv$
  $*[BPX?mx;$
    $[mx = \text{"}z0\text{"} \longrightarrow ZX[0]?x$
    $[]mx = \text{"}z1\text{"} \longrightarrow ZX[1]?x$
    $[]mx = \text{"}zero\text{"} \longrightarrow x := 0$
    $[]mx = \text{"}core\text{"} \longrightarrow R[0]?x$
    $];$
    $X!x$
   $]$
```

B.5 Port Priority Select

The Port Priority Select modification for the read bypass is discussed in Section 7.2.

Program B.8 CHP: read bypasses with port priority select

```
$BYPASS.BPZX \equiv$
    $*[BPX?mx, PPS_{BPX}?pp;$
      $[mx = "z0" \longrightarrow ZX[0]?x$
      $[] mx = "z1" \longrightarrow ZX[1]?x$
      $[] mx = "zero" \longrightarrow x := 0$
      $[] mx = "core" \longrightarrow R[0]?x$
      $];$
      $X!x,$
      $[pp \longrightarrow XY!x \; [] \; \textbf{else} \longrightarrow \textbf{skip}]$
    $]$

$BYPASS.BPZY \equiv$
    $*[BPY?my;$
      $[my = "z0" \longrightarrow ZY[0]?y$
      $[] my = "z1" \longrightarrow ZY[1]?y$
      $[] my = "zero" \longrightarrow y := 0$
      $[] my = "fromX" \longrightarrow XY?y$
      $[] my = "core" \longrightarrow R[1]?y$
      $];$
      $Y!y$
    $]$
```

B.6 Banking

The bypass modifications for accommodating banked cores is described in Section 4.4.3.

Program B.9 CHP: register file read bypass, for a dual-banked core

```
$BYPASS.BPZX \equiv$
  $*[BPX?mx;$
    $[mx = \text{"}z0\text{"} \longrightarrow ZX[0]?x$
    $[]\, mx = \text{"}z1\text{"} \longrightarrow ZX[1]?x$
    $[]\, mx = \text{"}core[lo]\text{"} \longrightarrow R[0, lo]?x$
    $[]\, mx = \text{"}core[hi]\text{"} \longrightarrow R[0, hi]?x$
    $];$
    $X!x$
  $]$
```

Program B.10 CHP: register file writeback bypass, for dual-banked register core

```
$BYPASS.BPZ[0] \equiv$
  $*[BPWB[0]?w0, BPZX[0]?zx0,$
      $BPZY[0]?zy0, Z[0]?z0;$
    $[w0 = \text{"}lo\text{"} \longrightarrow W[0, lo]!z0$
    $[]\, w0 = \text{"}hi\text{"} \longrightarrow W[0, hi]!z0$
    $[]\, \textbf{else} \longrightarrow \textbf{skip}],$
    $[zx0 \longrightarrow ZX[0]!z0 \;[]\; \textbf{else} \longrightarrow \textbf{skip}],$
    $[zy0 \longrightarrow ZY[0]!z0 \;[]\; \textbf{else} \longrightarrow \textbf{skip}]$
  $]$
```

Appendix C

Control CHP

This appendix includes the CHP for various versions of the register file *CONTROL* described throughout the thesis.

C.1 Base Design

The decomposition for the *CONTROL* of the base design is discussed in Section 2.5.

Program C.1 CHP: register bypass control for base design

```
$CONTROL.RSCOMP \equiv$
  $*[RS?rs, RD_{RS}?zs;$
    $[zs \neq \textbf{null} \longrightarrow ZBUS_{RS}?zbs \;[]\; \textbf{else} \longrightarrow \textbf{skip}];$
    $zx := rs \neq \textbf{null} \wedge rs = zs \wedge zs \neq 0;$
    $[rs \neq \textbf{null} \longrightarrow$
      $[zx \longrightarrow RI[0]!\textbf{null}, BPZX[zbs]!\textbf{true}$
        $[zbs = 0 \longrightarrow BPX!\text{"}z0\text{"} \;[]\; \textbf{else} \longrightarrow BPX!\text{"}z1\text{"}]$
      $[] \neg zx \longrightarrow RI[0]!rs,$
        $[zs \neq \textbf{null} \longrightarrow BPZX[zbs]!\textbf{false} \;[]\; \textbf{else} \longrightarrow \textbf{skip}],$
        $BPX!\text{"}core\text{"}$
      $]$
    $[] \;\textbf{else} \longrightarrow \textbf{skip}$
    $]$
  $]$

$CONTROL.RTCOMP \equiv$
  $*[RT?rt, RD_{RT}?zt;$
    $[zt \neq \textbf{null} \longrightarrow ZBUS_{RT}?zbt \;[]\; \textbf{else} \longrightarrow \textbf{skip}];$
    $zy := rt \neq \textbf{null} \wedge rt = zt \wedge zt \neq 0;$
    $[rt \neq \textbf{null} \longrightarrow$
      $[zy \longrightarrow RI[1]!\textbf{null}, BPZY[zbt]!\textbf{true}$
        $[zbt = 0 \longrightarrow BPY!\text{"}z0\text{"} \;[]\; \textbf{else} \longrightarrow BPY!\text{"}z1\text{"}]$
      $[] \neg zy \longrightarrow RI[1]!rt,$
        $[zt \neq \textbf{null} \longrightarrow BPZY[zbt]!\textbf{false} \;[]\; \textbf{else} \longrightarrow \textbf{skip}],$
        $BPY!\text{"}core\text{"}$
      $]$
    $[] \;\textbf{else} \longrightarrow \textbf{skip}$
    $]$
  $]$
```

Program C.2 CHP: register writeback control of base design

```
$CONTROL.WBCTRL \equiv$
    $*[RD_{WB}?zw,\ Valid?val;$
      $[zw \neq \textbf{null} \longrightarrow ZBUS_{WB}?zbw;\ \ ZV[zbw]?zv;$
        $[val \wedge zv \longrightarrow BPWB[zbw]!\textbf{true},\ WI[zbw]!zw,\ WI[\neg zbw]!\textbf{null}$
        $[\!] \ \textbf{else} \longrightarrow BPWB[zbw]!\textbf{false},\ WI[zbw]!\textbf{null},\ WI[\neg zbw]!\textbf{null}$
        $]$
      $[\!] \ \textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$
```

Program C.3 CHP: destination copy process

```
$CONTROL.RDCOPY \equiv$
    $RD_{RS}!\textbf{null},\ RD_{RT}!\textbf{null},\ RD_{WB}!\textbf{null};$
    $*[RD?rd;$
      $RD_{RS}!rd,\ RD_{RT}!rd,\ RD_{WB}!rd$
    $]$
$CONTROL.ZBCOPY \equiv$
    $*[ZBUS?zb;$
      $ZBUS_{RS}!zb,\ ZBUS_{RT}!zb,\ ZBUS_{WB}!zb$
    $]$
```

C.2 Banking

The control modification for supporting banked register core and bypass is discussed in Section 4.4.4.

Program C.4 CHP: register bypass control for dual-banked register core

```
$CONTROL.RSCOMP \equiv$
    $*[RS?rs, RD_{RS}?zs;$
      $[zs \neq \textbf{null} \longrightarrow ZBUS_{RS}?zbs \;[]\; \textbf{else} \longrightarrow \textbf{skip}];$
      $zx := rs \neq \textbf{null} \wedge rs = zs \wedge zs \neq 0;$
      $[rs \neq \textbf{null} \longrightarrow$
        $[zx \longrightarrow RI[0]!\textbf{null}, BPZX[zbs]!\textbf{true}$
          $[zbs = 0 \longrightarrow BPX!\text{"}z0\text{"} \;[]\; \textbf{else} \longrightarrow BPX!\text{"}z1\text{"}]$
        $[]\neg zx \longrightarrow RI[0]!rs,$
          $[zs \neq \textbf{null} \longrightarrow BPZX[zbs]!\textbf{false} \;[]\; \textbf{else} \longrightarrow \textbf{skip}],$
          $[rs < 16 \longrightarrow BPX!\text{"}core[lo]\text{"} \;[]\; \textbf{else} \longrightarrow BPX!\text{"}core[hi]\text{"}]$
        $]$
      $[]\; \textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$

$CONTROL.RTCOMP \equiv$
    $*[RT?rt, RD_{RT}?zt;$
      $[zt \neq \textbf{null} \longrightarrow ZBUS_{RT}?zbt \;[]\; \textbf{else} \longrightarrow \textbf{skip}];$
      $zy := rt \neq \textbf{null} \wedge rt = zt \wedge zt \neq 0;$
      $[rt \neq \textbf{null} \longrightarrow$
        $[zy \longrightarrow RI[1]!\textbf{null}, BPZY[zbt]!\textbf{true}$
          $[zbt = 0 \longrightarrow BPY!\text{"}z0\text{"} \;[]\; \textbf{else} \longrightarrow BPY!\text{"}z1\text{"}]$
        $[]\neg zy \longrightarrow RI[1]!rt,$
          $[zt \neq \textbf{null} \longrightarrow BPZY[zbt]!\textbf{false} \;[]\; \textbf{else} \longrightarrow \textbf{skip}],$
          $[rt < 16 \longrightarrow BPY!\text{"}core[lo]\text{"} \;[]\; \textbf{else} \longrightarrow BPY!\text{"}core[hi]\text{"}]$
        $]$
      $[]\; \textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$
```

Program C.5 CHP: register writeback control for a banked register core

```
$CONTROL.WBCTRL \equiv$
    $*[RD_{WB}?zw,\ Valid?val;$
      $[zw \neq \textbf{null} \longrightarrow ZBUS_{WB}?zbw;\ \ ZV[zbw]?zv;$
        $[val \wedge zv \longrightarrow$
          $[zw < 16 \longrightarrow BPWB[zbw]!\text{"}lo\text{"}\ \ [\!]\ \textbf{else} \longrightarrow BPWB[zbw]!\text{"}hi\text{"}],$
          $COREWB[zbw]!\textbf{true},\ WI[zbw]!zw,\ WI[\neg zbw]!\textbf{null}$
        $[\!]\ \textbf{else} \longrightarrow BPWB[zbw]!\textbf{false},\ COREWB[zbw]!\textbf{false},$
          $WI[zbw]!\textbf{null},\ WI[\neg zbw]!\textbf{null}$
        $]$
      $[\!]\ \textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$
```

C.3 Register Zero

The control modification to support reading 0 from the bypass is discussed in Section 6.2.2. The control modification to support consuming writes to register zero at the writeback bypass is discussed in Section 6.3.1.

Program C.6 CHP: register bypass control for reading 0 from the bypass

```
$CONTROL.RSCOMP \equiv$
    $*[RS?rs, RD_{RS}?zs;$
      $[zs \neq \textbf{null} \longrightarrow ZBUS_{RS}?zbs \;[]\; \textbf{else} \longrightarrow \textbf{skip}];$
      $zx := rs \neq \textbf{null} \wedge rs = zs \wedge zs \neq 0;$
      $[rs \neq \textbf{null} \longrightarrow$
        $[zx \longrightarrow RI[0]!\textbf{null}, BPZX[zbs]!\textbf{true}$
          $[zbs = 0 \longrightarrow BPX!\text{"}z0\text{"} \;[]\; \textbf{else} \longrightarrow BPX!\text{"}z1\text{"}]$
        $[] \neg zx \longrightarrow$
          $[rs \neq 0 \longrightarrow RI[0]!rs \;[]\; \textbf{else} \longrightarrow RI[0]!\textbf{null}],$
          $[zs \neq \textbf{null} \longrightarrow BPZX[zbs]!\textbf{false} \;[]\; \textbf{else} \longrightarrow \textbf{skip}],$
          $[rs = 0 \longrightarrow BPX!\text{"}zero\text{"} \;[]\; \textbf{else} \longrightarrow BPX!\text{"}core\text{"}]$
        $]$
      $[] \;\textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$
```

Program C.7 CHP: register writeback control

```
$CONTROL.WBCTRL \equiv$
    $*[RD_{WB}?zw, Valid?val;$
      $[zw = \textbf{null} \longrightarrow \textbf{skip}$
      $[] zw = 0 \longrightarrow ZBUS_{WB}?zbw; ZV[zbw]?zv;$
        $BPWB[zbw]!\textbf{false}, WI[zbw]!\textbf{null}, WI[\neg zbw]!\textbf{null}$
      $[] \;\textbf{else} \longrightarrow ZBUS_{WB}?zbw; ZV[zbw]?zv;$
        $[val \wedge zv \longrightarrow BPWB[zbw]!\textbf{true}, COREWB[zbw]!\textbf{true},$
          $WI[zbw]!zw, WI[\neg zbw]!\textbf{null}$
        $[] \;\textbf{else} \longrightarrow BPWB[zbw]!\textbf{false}, COREWB[zbw]!\textbf{false},$
          $WI[zbw]!\textbf{null}, WI[\neg zbw]!\textbf{null}$
        $]$
      $]$
     $]$
```

C.4 Port Priority Select

Register file control modification for Port Priority Select is described in Section 7.3.

Program C.8 CHP: priority port comparator

```
$CONTROL.RSRTEQ \equiv$
    $*[RS_{EQ}?rs, RT_{EQ}?rt;$
      $eq := rs = rt \wedge rt \neq \mathbf{null};$
      $EQ_{RS}!eq, EQ_{RT}!eq$
    $]$
```

Program C.9 CHP: register bypass control, with priority port select

```
$CONTROL.RSCOMP \equiv$
    $*[RS_{RS}?rs, RD_{RS}?zs, EQ_{RS}?eqs;$
      $[zs \neq \textbf{null} \longrightarrow ZBUS_{RS}?zbs \;[\!]\; \textbf{else} \longrightarrow \textbf{skip}];$
      $zx := rs \neq \textbf{null} \wedge rs = zs \wedge zs \neq 0;$
      $[rs \neq \textbf{null} \longrightarrow$
        $[zx \longrightarrow RI[0]!\textbf{null}, BPZX[zbs]!\textbf{true},$
          $[zbs = 0 \longrightarrow BPX!\text{"}z0\text{"} \;[\!]\; \textbf{else} \longrightarrow BPX!\text{"}z1\text{"}],$
          $PPS_{BPX}!\textbf{false}$
        $[\!]\neg zx \longrightarrow$
          $[rs \neq 0 \longrightarrow RI[0]!rs \;[\!]\; \textbf{else} \longrightarrow RI[0]!\textbf{null}],$
          $[zs \neq \textbf{null} \longrightarrow BPZX[zbs]!\textbf{false} \;[\!]\; \textbf{else} \longrightarrow \textbf{skip}],$
          $[\;\; rs = 0 \longrightarrow BPX!\text{"}zero\text{"}, PPS_{BPX}!\textbf{false}$
          $[\!]\; \textbf{else} \longrightarrow BPX!\text{"}core\text{"}, PPS_{BPX}!eqs$
          $]$
        $]$
      $[\!]\; \textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$

$CONTROL.RTCOMP \equiv$
    $*[RT_{RT}?rt, RD_{RT}?zt, EQ_{RT}?eqt;$
      $[zt \neq \textbf{null} \longrightarrow ZBUS_{RT}?zbt \;[\!]\; \textbf{else} \longrightarrow \textbf{skip}];$
      $zx := rs \neq \textbf{null} \wedge rs = zt \wedge zt \neq 0;$
      $[rt \neq \textbf{null} \longrightarrow$
        $[zy \longrightarrow RI[1]!\textbf{null}, BPZY[zbt]!\textbf{true}$
          $[zbt = 0 \longrightarrow BPY!\text{"}z0\text{"} \;[\!]\; \textbf{else} \longrightarrow BPY!\text{"}z1\text{"}]$
        $[\!]\neg zy \longrightarrow$
          $[rt \neq 0 \wedge \neg eqt \longrightarrow RI[1]!rt \;[\!]\; \textbf{else} \longrightarrow RI[1]!\textbf{null}],$
          $[zt \neq \textbf{null} \longrightarrow BPZY[zbt]!\textbf{false} \;[\!]\; \textbf{else} \longrightarrow \textbf{skip}],$
          $[\;\; rt = 0 \longrightarrow BPY!\text{"}zero\text{"}$
          $[\!]\;\; rt \neq 0 \wedge eqt \longrightarrow BPY!\text{"}fromX\text{"}$
          $[\!]\; \textbf{else} \longrightarrow BPY!\text{"}core\text{"}$
          $]$
        $]$
      $[\!]\; \textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$
```

Appendix D

Core CHP

This appendix includes CHP programs for various transformations of the register file *CORE* presented throughout the thesis.

D.1 Pipelined Core

The pipeline transformations and locking mechanisms for the *CORE* are discussed in Section 3.5.

Program D.1 CHP: unpipelined multiported register

```
$RPORT[i].RDATA[l] \equiv$
    $*[[\overline{RC[l,i]}]; R[i]!reg[l]; RC[l,i]]$
$WPORT[j].WDATA[l] \equiv$
    $*[[\overline{WC[l,j]}]; W[j]?reg[l]; WC[l,j]]$
$REGDATA[l] \equiv$
     $\langle \| \forall i : RPORT[i].RDATA[l] \rangle$
    $\| \langle \| \forall j : WPORT[j].WDATA[l] \rangle$
```

Program D.2 CHP: pipelined, multiported register block

```
$RPORT[i].RDATA[l].BLOCK \equiv$
    $*[[\overline{RC_i[l,i]}]; (R[i]!reg[l], RC_o[l,i]); RC_i[l,i]]$
$WPORT[j].WDATA[l].BLOCK \equiv$
    $*[[\overline{WC_i[l,j]}]; (W[j]?reg[l], WC_o[l,j]); WC_i[l,j]]$
$REGDATA[l].BLOCK \equiv$
     $\langle \| \forall i : RPORT[i].RDATA[l].BLOCK \rangle$
    $\| \langle \| \forall j : WPORT[j].WDATA[l].BLOCK \rangle$
```

Program D.3 CHP: pipelined register block with locking

```
$RPORT[i].RDATA[l].BLOCK \equiv$
        $*[[\overline{RC'[l,i]}]; rx[l,i]\uparrow; RC'[l,i];$
          $(R[i]!reg[l], RC_o[l,i]); rx[l,i]\downarrow$
        $]$
      $\| *[[\overline{RC_i[l,i]} \wedge \langle \wedge \forall j : \neg wx[l,j] \rangle];$
          $RC'[l,i]; RC_i[l,i]$
        $]$
$WPORT[j].WDATA[l].BLOCK \equiv$
        $*[[\overline{WC'[l,j]}]; wx[l,j]\uparrow; WC'[l,j];$
          $(W[j]?reg[l], WC_o[l,j]); wx[l,j]\downarrow$
        $]$
      $\| *[[\overline{WC_i[l,j]} \wedge \langle \wedge \forall i : \neg rx[l,i] \rangle \wedge \langle \wedge \forall k \neq j : \neg wx[l,k] \rangle];$
          $WC'[l,j]; WC_i[l,j]$
        $]$
$REGDATA[l].BLOCK \equiv$
      $\langle \| \forall i : RPORT[i].RDATA[l].BLOCK \rangle$
    $\| \langle \| \forall j : WPORT[j].WDATA[l].BLOCK \rangle$
```

Program D.4 CHP: pipelined register read port with locking at the sender

```
$RPORT[i].RDATA[l].BLOCK \equiv$
        $*[RC'[l,i]; R[i]!reg[l]; RC''[l,i]]$
      $\| *[[\overline{RC_i[l,i]}]; RC'[l,i];$
          $[\langle \wedge \forall j : \neg wx[l,j] \rangle]; rx[l,i]\uparrow;$
          $(RC_o[l,i], (RC''[l,i]; RC_i[l,i])); rx[l,i]\downarrow$
        $]$
```

Program D.5 CHP: pipelined register write port with locking at the sender

```
$WPORT[j].WDATA[l].BLOCK \equiv$
        $*[WC'[l,j]; W[j]?reg[l]; WC''[l,j]]$
      $\| *[[\overline{WC_i[l,j]}]; WC'[l,j];$
          $[\langle \wedge \forall i : \neg rx[l,i] \rangle \wedge \langle \wedge \forall k \neq j : \neg wx[l,k] \rangle]; wx[l,j]\uparrow;$
          $(WC_o[l,j], (WC''[l,j]; WC_i[l,j])); wx[l,j]\downarrow$
        $]$
```

Program D.6 CHP: read port demux, with locking

```
$RPORT[i].RDEMUX \equiv$
    $*[RI[i]?ri[i];$
     $[ri[i] \neq \textbf{null} \longrightarrow [\langle\wedge\forall j : \neg wx[l,j]\rangle];$
        $rx[l,i]\uparrow; RC[ri[i],i]!; rx[l,i]\downarrow$
     $[\!]\textbf{else} \longrightarrow \textbf{skip}$
     $]$
    $]$
```

Program D.7 CHP: write port demux, with locking

```
$WPORT[j].WDEMUX \equiv$
    $*[WI[j]?wi[j];$
      $[wi[j] \neq \textbf{null} \longrightarrow [\langle\wedge\forall i : \neg rx[l,i]\rangle \wedge \langle\wedge\forall k \neq j : \neg wx[l,k]\rangle];$
         $wx[l,j]\uparrow; WC[wi[j],j]!; wx[l,j]\downarrow$
      $[\!]\textbf{else} \longrightarrow \textbf{skip}$
      $]$
    $]$
```

Program D.8 CHP: pipelined, multiported zero-register block

```
$RPORT[i].RDATA[0].BLOCK \equiv$
    $*[RC_i[0,i];(R[i]!0, RC_o[0,i])]$
$WPORT[j].WDATA[0].BLOCK \equiv$
    $*[WC_i[0,j];(W[j]?, WC_o[0,j])]$
$REGDATA[0].BLOCK \equiv$
     $\langle\| \forall i : RPORT[i].RDATA[0].BLOCK\rangle$
    $\| \langle\| \forall j : WPORT[j].WDATA[0].BLOCK\rangle$
```

D.2 WAD Core

The width-adaptive transformation for the *CORE* is discussed in Section 5.3.2.

Program D.9 CHP: WAD read port, without locking in the termination case

```
$RPORT[i].RDATA[l].BLOCK \equiv$
      $*[[\overline{RC_i[l,i]}]; R[i]!reg[l]]$
    $\| *[[p(reg[l]) \wedge \overline{RC_i[l,i]} \wedge \langle \forall j : \neg wx[l,j] \rangle \longrightarrow$
          $rx[l,i]\uparrow; (RC_o[l,i], RC_i[l,i]); rx[l,i]\downarrow$
       $[] t(reg[l]) \wedge \overline{RC_i[l,i]} \longrightarrow RC_i[l,i]$
      $]]$
```

Program D.10 CHP: WAD register write port, without locking in the terminating case

```
$WPORT[i].WDATA[l].BLOCK \equiv$
      $*[[\overline{WC_i[l,j]}]; W[j]?reg[l]]$
    $\| *[[p(\overline{W[j]}) \wedge \overline{WC_i[l,j]} \wedge \langle \forall i : \neg rx[l,i] \rangle \wedge \langle \forall k \neq j : \neg wx[l,k] \rangle \longrightarrow$
          $wx[l,j]\uparrow; (WC_o[l,j], WC_i[l,j]); wx[l,j]\downarrow$
       $[] t(\overline{W[j]}) \wedge \overline{WC_i[l,j]} \longrightarrow WC_i[l,j]$
      $]]$
```

D.3 Nested Core

The nesting transformation for the base register *CORE* is discussed in Section 9.2.1.

Program D.11 CHP: nested partitions read, with unconditional pipeline-locked control propagation

```
$RPORT[i].RDATA[l].BLOCK_{outer} \equiv$
    $*[RC'[l,i]; R[i]!reg[l]; RC''[l,i]]$
  $\| *[[\overline{RC_i[l,i]}]; RC'[l,i]; [\langle\wedge\forall j : \neg wx[l,j]\rangle];$
      $rx[l,i]\uparrow; (RC_o[l,i], (RC''[l,i]; RC_i[l,i])); rx[l,i]\downarrow$
     $]$
$RPORT[i].RDATA.CONNECT \equiv$
    $*[[\overline{IRC_i[i]} \wedge RC^{inner}[i]];$
     $(R[i]!(IR[i]?), IRC_o[i], IRC_i[i])$
    $]$
$RPORT[i].RDATA[l].BLOCK_{inner} \equiv$
    $*[IRC'[l,i]; IR[i]!reg[l]; IRC''[l,i]]$
  $\| *[[\overline{IRC_i[l,i]}]; IRC'[l,i]; [\langle\wedge\forall j : \neg wx[l,j]\rangle];$
      $rx[l,i]\uparrow; (IRC_o[l,i], (IRC''[l,i]; IRC_i[l,i])); rx[l,i]\downarrow$
     $]$
```

Program D.12 CHP: nested partition write, with unconditional pipeline-locked control propagation

```
$WPORT[j].WDATA[l].BLOCK_{outer} \equiv$
    $*[WC'[l,j]; W[j]?reg[l]; WC''[l,j]]$
  $\| *[[\overline{WC_i[l,j]}]; WC'[l,j]; [\langle\wedge\forall i : \neg rx[l,i]\rangle \wedge \langle\wedge\forall k \neq j : \neg wx[l,k]\rangle];$
      $wx[l,j]\uparrow; (WC_o[l,j], (WC''[l,j]; WC_i[l,j])); wx[l,j]\downarrow$
     $]$
$WPORT[j].WDATA.CONNECT \equiv$
    $*[[\overline{IWC_i[j]} \wedge WC^{inner}[j]];$
     $(IW[j]!(W[j]?), IWC_o[j], IWC_i[j])$
    $]$
$WPORT[j].WDATA[l].BLOCK_{inner} \equiv$
    $*[IWC'[l,j]; IW[j]?reg[l]; IWC''[l,j]]$
  $\| *[[\overline{IWC_i[l,j]}]; IWC'[l,j]; [\langle\wedge\forall i : \neg rx[l,i]\rangle \wedge \langle\wedge\forall k \neq j : \neg wx[l,k]\rangle];$
      $wx[l,j]\uparrow; (IWC_o[l,j], (IWC''[l,j]; IWC_i[l,j])); wx[l,j]\downarrow$
     $]$
```

Program D.13 CHP: read and write demuxes for nested partitioning, port i

```
$RPORT[i].RDEMUX_{nested} \equiv$
  $*[ri[i] := \overline{RI[i]};$
    $[ri[i] \neq \textbf{null} \longrightarrow$
      $[ri[i] < 16 \longrightarrow RC^{ri[i]}[i]!$
      $[\!] ri[i] \geq 16 \longrightarrow RC^{inner}[i]!, IRC^{ri[i]}[i]!$
      $]$
    $[\!]$ $\textbf{else} \longrightarrow \textbf{skip}$
    $];$
    $RI[i]?$
  $]$
$WPORT[i].WDEMUX_{nested} \equiv$
  $*[wi[i] := \overline{WI[i]};$
    $[wi[i] \neq \textbf{null} \longrightarrow$
      $[wi[i] < 16 \longrightarrow WC^{wi[i]}[i]!$
      $[\!] wi[i] \geq 16 \longrightarrow WC^{inner}[i]!, IWC^{wi[i]}[i]!$
      $]$
    $[\!]$ $\textbf{else} \longrightarrow \textbf{skip}$
    $];$
    $WI[i]?$
  $]$
```

D.4 WAD Nested Core

The nesting transformation for the width-adaptive register *CORE* is discussed in Section 9.2.2.

Program D.14 CHP: nested partitions read, with WAD pipeline-locked control propagation

```
$RPORT[i].RDATA[l].BLOCK_{wad,outer} \equiv$
    $*[RC'[l,i]; R[i]!reg[l]; RC''[l,i]]$
  $\|\ *[[\overline{RC_i[l,i]}]; RC'[l,i];$
      $[p(reg[l]) \wedge \langle \forall j : \neg wx[l,j] \rangle \longrightarrow$
        $rx[l,i]\uparrow; (RC_o[l,i], (RC''[l,i]; RC_i[l,i])); rx[l,i]\downarrow$
      $[] t(reg[l]) \longrightarrow RC''[l,i]; RC_i[l,i]$
    $]]$
$RPORT[i].RDATA.CONNECT \equiv$
    $*[[\overline{IRC_i[i]} \wedge RC^{inner}[i]];$
     $(R[i]!(IR[i]?), IRC_o[i], IRC_i[i])$
    $]$
$RPORT[i].RDATA[l].BLOCK_{wad,inner} \equiv$
    $*[IRC'[l,i]; IR[i]!reg[l]; IRC''[l,i]]$
  $\|\ *[[\overline{IRC_i[l,i]}]; IRC'[l,i];$
      $[p(reg[l]) \wedge \langle \forall j : \neg wx[l,j] \rangle \longrightarrow$
        $rx[l,i]\uparrow; (IRC_o[l,i], (IRC''[l,i]; IRC_i[l,i])); rx[l,i]\downarrow$
      $[] t(reg[l]) \longrightarrow IRC''[l,i]; IRC_i[l,i]$
    $]]$
```

Program D.15 CHP: nested partition write, with WAD pipeline-locked control propagation

```
$WPORT\,[j]\,.WDATA\,[l]\,.BLOCK_{outer} \equiv$
        $*[\,WC'\,[l,j]\,;\;W\,[j]\,?reg\,[l]\,;\;WC''\,[l,j]\,]$
      $\|\;*[\,\overline{WC_i\,[l,j]}\,;\;WC'\,[l,j]\,;$
          $[\,p(\overline{W\,[j]}) \wedge \langle \forall i : \neg rx\,[l,i] \rangle \wedge \langle \forall k \neq j : \neg wx\,[l,k] \rangle \longrightarrow$
             $wx\,[l,j]\uparrow;(\,WC_o\,[l,j]\,,(\,WC''\,[l,j]\,;\;WC_i\,[l,j]\,));wx\,[l,j]\downarrow$
          $[\!]\,t(\overline{W\,[j]}) \longrightarrow WC''\,[l,j]\,;\;WC_i\,[l,j]$
        $]]$
$WPORT\,[j]\,.WDATA.CONNECT \equiv$
        $*[\,[\,\overline{IWC_i\,[j]} \wedge WC^{inner}\,[j]\,]\,;$
          $(IW\,[j]!(W\,[j]?),IWC_o\,[j]\,,IWC_i\,[j]\,)$
        $]$
$WPORT\,[j]\,.WDATA\,[l]\,.BLOCK_{inner} \equiv$
        $*[IWC'\,[l,j]\,;IW\,[j]\,?reg\,[l]\,;IWC''\,[l,j]\,]$
      $\|\;*[\,\overline{IWC_i\,[l,j]}\,;IWC'\,[l,j]\,;$
          $[\,p(\overline{IW\,[j]}) \wedge \langle \forall i : \neg rx\,[l,i] \rangle \wedge \langle \forall k \neq j : \neg wx\,[l,k] \rangle \longrightarrow$
             $wx\,[l,j]\uparrow;(IWC_o\,[l,j]\,,(IWC''\,[l,j]\,;IWC_i\,[l,j]\,));wx\,[l,j]\downarrow$
          $[\!]\,t(\overline{IW\,[j]}) \longrightarrow IWC''\,[l,j]\,;IWC_i\,[l,j]$
        $]]$
```

Appendix E

Core HSE

This appendix includes the handshaking expansions (HSE) for the various versions of the register file *CORE* presented throughout the thesis.

E.1 Pipelined Core

The pipeline transformations and locking mechanisms for the *CORE* are presented in Sections 4.1.2 and 4.1.3.

E.2 WAD Core

The width-adaptive transformation for the *CORE* is discussed in Section 5.4.

Program E.1 HSE: PCEVFB WAD read port

```
$*[(([R_o^e]; ren_D\uparrow; [RC_i]; R_o\uparrow),$
    $([RC_o^e]; ren_C\uparrow;$
        $[RC_i \wedge p(reg) \wedge unlocked() \longrightarrow lock; RC_o\uparrow [] t(reg) \longrightarrow \textbf{skip}]));$
  $RC_i^e\downarrow;$
  $(([\neg R_o^e]; ren_D\downarrow; R_o\downarrow),$
    $([(p(reg) \wedge \neg RC_o^e) \vee t(reg)]; ren_C\downarrow; unlock; RC_o\downarrow),$
    $([\neg RC_i \wedge \neg ren_D \wedge \neg ren_C]; RC_i^e\uparrow))$
 $]$
```

Program E.2 HSE: PCEVHB WAD read port with full-buffered data output, and half-buffered control propagation

```
$*[(([R_o^e]; ren_D\uparrow; [RC_i]; R_o),$
    $([RC_o^e]; ren_C\uparrow;$
        $[RC_i \wedge p(reg) \wedge unlocked() \longrightarrow lock; RC_o\uparrow [] t(reg) \longrightarrow \textbf{skip}]));$
  $RC_i^e\downarrow;$
  $(([\neg R_o^e]; ren_D\downarrow; R_o\downarrow),$
    $([(p(reg) \wedge \neg RC_o^e) \vee t(reg)]; ren_C\downarrow;$
      $unlock; RC_o\downarrow; [\neg ren_D \wedge \neg RC_i]; RC_i^e\uparrow))$
 $]$
```

Program E.3 HSE: PCEVFB WAD write port, with unconditional write-enable

```
$*[[WC_o^e]; wen\uparrow;$
  $[WC_i \wedge unlocked() \wedge p(W_i) \longrightarrow lock; WC_o\uparrow [] t(W_i) \longrightarrow \textbf{skip}];$
  $[wvc]; WC_i^e\downarrow;$
  $[(p(W_i) \wedge \neg WC_o^e) \vee t(W_i)]; wen\downarrow;$
  $((unlock; WC_o\downarrow), ([\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow))$
 $]$
$*[[WC_i \wedge W_i]; \langle write \rangle; wvc\uparrow; [\neg W_i]; wvc\downarrow]$
```

Program E.4 HSE: PCEVHB WAD write port, with unconditional write-enable

```
*[[$WC_o^e$]; $wen\uparrow$;
   [$WC_i \wedge unlocked() \wedge p(W_i) \longrightarrow lock$; $WC_o\uparrow$ ▯ $t(W_i) \longrightarrow \textbf{skip}$];
   [$wvc$]; $WC_i^e\downarrow$;
   [$(p(W_i) \wedge \neg WC_o^e) \vee t(W_i)$]; $wen\downarrow$; $unlock$; $WC_o\downarrow$;
   [$\neg WC_i \wedge \neg wvc$]; $WC_i^e\uparrow$
  ]
*[[$WC_i \wedge W_i$]; $\langle write \rangle$; $wvc\uparrow$; [$\neg W_i$]; $wvc\downarrow$]
```

Program E.5 HSE: PCEVFB WAD write port, with conditional write-enable

```
*[[$WC_o^e \wedge p(W_i) \longrightarrow wen\uparrow$; [$WC_i \wedge unlocked()$]; $lock$; $WC_o\uparrow$
      ▯ $t(W_i) \longrightarrow \textbf{skip}$];
   [$wvc$]; $WC_i^e\downarrow$;
   [$\neg WC_o^e$]; $wen\downarrow$;
   (($unlock$; $WC_o\downarrow$), ([$\neg WC_i \wedge \neg wvc$]; $WC_i^e\uparrow$))
  ]
*[[$WC_i \wedge W_i$]; $\langle write \rangle$; $wvc\uparrow$; [$\neg W_i$]; $wvc\downarrow$]
```

Program E.6 HSE: PCEVHB WAD write port, with conditional write-enable

```
*[[$WC_o^e \wedge p(W_i) \longrightarrow wen\uparrow$; [$WC_i \wedge unlocked()$]; $lock$; $WC_o\uparrow$
      ▯ $t(W_i) \longrightarrow \textbf{skip}$];
   [$wvc$]; $WC_i^e\downarrow$;
   [$\neg WC_o^e$]; $wen\downarrow$; $unlock$; $WC_o\downarrow$;
   [$\neg WC_i \wedge \neg wvc$]; $WC_i^e\uparrow$
  ]
*[[$WC_i \wedge W_i$]; $\langle write \rangle$; $wvc\uparrow$; [$\neg W_i$]; $wvc\downarrow$]
```

E.3 Non-WAD Nested Core

The template nesting transformations for the non-WAD read and write ports are discussed in Sections 9.3.1 and 9.3.2.

Program E.7 HSE: PCEVFB data-independent read port with nested data

```
$*[(([R_o^e]; ren_D\uparrow);$
   $[RC_{i,inner} \longrightarrow iren_D\uparrow; [IRC_i]; IR_o\uparrow; R_o\uparrow [] RC_{i,outer} \longrightarrow R_o\uparrow]),$
  $(([RC_o^e]; ren_C\uparrow);$
   $[RC_{i,inner} \longrightarrow [IRC_o^e]; iren_C\uparrow; [IRC_i \wedge unlocked() \longrightarrow lock; IRC_o\uparrow];$
     $(IRC_i^e\downarrow, ([R_o]; RC_i^e\downarrow))$
   $[] RC_{i,outer} \longrightarrow [unlocked() \longrightarrow lock; RC_o\uparrow]; [R_o]; RC_i^e\downarrow$
   $]);$
  $(([\neg R_o^e]; ren_D\downarrow;$
    $([RC_{i,inner} \longrightarrow iren_D\downarrow; IR_o\downarrow [] RC_{i,outer} \longrightarrow$ skip$], R_o\downarrow)),$
   $([\neg RC_o^e]; ren_C\downarrow;$
     $[RC_{i,inner} \longrightarrow [\neg IRC_o^e]; iren_C\downarrow; ((unlock; IRC_o\downarrow), ([\neg IRC_i]; IRC_i^e\uparrow))$
     $[] RC_{i,outer} \longrightarrow unlock; RC_o\downarrow$
     $]),$
   $([\neg ren_D \wedge \neg ren_C \wedge \neg RC_i]; RC_i^e\uparrow)$
  $)$
 $]$
```

Program E.8 HSE: PCEVHB data-independent read port with nested data, full-buffered data output, and half-buffered control propagation

```
$*[(([R_o^e]; ren_D\uparrow);$
    $[RC_{i,inner} \longrightarrow iren_D\uparrow; [IRC_i]; IR_o\uparrow; R_o\uparrow [] RC_{i,outer} \longrightarrow R_o\uparrow]),$
  $(([RC_o^e]; ren_C\uparrow);$
    $[RC_{i,inner} \longrightarrow [IRC_o^e]; iren_C\uparrow; [IRC_i \land unlocked() \longrightarrow lock; IRC_o\uparrow];$
      $(IRC_i^e\downarrow, ([R_o]; RC_i^e\downarrow))$
    $[] RC_{i,outer} \longrightarrow [unlocked() \longrightarrow lock; RC_o\uparrow]; [R_o]; RC_i^e\downarrow$
    $]);$
  $(([\neg R_o^e]; ren_D\downarrow;$
    $([RC_{i,inner} \longrightarrow iren_D\downarrow; IR_o\downarrow [] RC_{i,outer} \longrightarrow$ **skip**$], R_o\downarrow)),$
   $([\neg RC_o^e]; ren_C\downarrow;$
     $[RC_{i,inner} \longrightarrow [\neg IRC_o^e]; iren_C\downarrow;$
       $((unlock; IRC_o\downarrow; [\neg ren_D \land \neg RC_i]; RC_i^e\uparrow), ([\neg IRC_i]; IRC_i^e\uparrow))$
     $[] RC_{i,outer} \longrightarrow unlock; RC_o\downarrow; [\neg ren_D \land \neg RC_i]; RC_i^e\uparrow$
     $])$
  $)$
 $]$
```

Program E.9 HSE: PCEVFB data-independent write port, with nested data

```
$*[[WC_o^e]; wen\uparrow;$
   $[WC_{i,inner} \longrightarrow [IWC_o^e]; iwen\uparrow; [unlocked()]; lock; IWC_o\uparrow;$
     $(IWC_i^e\downarrow, ([wvc]; WC_i^e\downarrow))$
   $[] WC_{i,outer} \longrightarrow [unlocked()]; lock; WC_o\uparrow; [wvc]; WC_i^e\downarrow;$
   $];$
   $[\neg WC_o^e]; wen\downarrow;$
   $[WC_{i,inner} \longrightarrow [\neg IWC_o^e]; iwen\downarrow;$
     $((unlock; IWC_o\downarrow), [\neg IWC_i \longrightarrow IWC_i^e\uparrow, ([\neg WC_i \land \neg wvc]; WC_i^e\uparrow)])$
   $[] WC_{i,outer} \longrightarrow (unlock; WC_o\downarrow), ([\neg WC_i \land \neg wvc]; WC_i^e\uparrow)$
   $]$
 $]$
$\|$
$*[[WC_{i,inner} \longrightarrow [W_i]; IW\uparrow; \langle write_{inner}\rangle$
   $[] WC_{i,outer} \longrightarrow [W_i]; \langle write_{outer}\rangle$
   $];$
   $wvc\uparrow;$
   $[\neg W_i]; (IW\downarrow; wvc\downarrow)$
 $]$
```

Program E.10 HSE: PCEVHB data-independent write port, with nested data

```
*[[$WC_o^e$]; $wen\uparrow$;
  [$WC_{i,inner} \longrightarrow$ [$IWC_o^e$]; $iwen\uparrow$; [$unlocked()$]; $lock$; $IWC_o\uparrow$;
     ($IWC_i^e\downarrow$, ([$wvc$]; $WC_i^e\downarrow$))
  [] $WC_{i,outer} \longrightarrow$ [$unlocked()$]; $lock$; $WC_o\uparrow$; [$wvc$]; $WC_i^e\downarrow$
  ];
  [$\neg WC_o^e$]; $wen\downarrow$;
  [$WC_{i,inner} \longrightarrow$ [$\neg IWC_o^e$]; $iwen\downarrow$;
     (($unlock$; $IWC_o\downarrow$; [$\neg WC_i \wedge \neg wvc$]; $WC_i^e\uparrow$),
      ([$\neg IWC_i$]; $IWC_i^e\uparrow$))
  [] $WC_{i,outer} \longrightarrow$ $unlock$; $WC_o\downarrow$; [$\neg WC_i \wedge \neg wvc$]; $WC_i^e\uparrow$
  ]
 ]
||
*[[$WC_{i,inner} \longrightarrow$ [$W_i$]; $IW\uparrow$; $\langle write_{inner}\rangle$
  [] $WC_{i,outer} \longrightarrow$ [$W_i$]; $\langle write_{outer}\rangle$
  ];
  $wvc\uparrow$;
  [$\neg W_i$]; ($IW\downarrow$; $wvc\downarrow$)
 ]
```

E.4 WAD Nested Core

The template nesting transformations for the WAD read and write ports are discussed in Sections 9.3.3 and 9.3.4.

Program E.11 HSE: PCEVFB WAD read port with nested data

```
$*[(([R_o^e]; ren_D\uparrow);$
    $[RC_{i,inner} \longrightarrow iren_D\uparrow; [IRC_i]; IR_o\uparrow; R_o\uparrow [] RC_{i,outer} \longrightarrow R_o\uparrow]),$
  $(([RC_o^e]; ren_C\uparrow);$
    $[RC_{i,inner} \longrightarrow [IRC_o^e]; iren_C\uparrow; [IRC_i];$
      $[p(reg) \wedge unlocked() \longrightarrow lock; IRC_o\uparrow [] t(reg) \longrightarrow \textbf{skip}];$
      $(IRC_i^e\downarrow, ([R_o]; RC_i^e\downarrow))$
    $[] RC_{i,outer} \longrightarrow [p(reg) \wedge unlocked() \longrightarrow lock; RC_o\uparrow [] t(reg) \longrightarrow \textbf{skip}];$
      $[R_o]; RC_i^e\downarrow$
    $]);$
  $(([\neg R_o^e]; ren_D\downarrow;$
    $([RC_{i,inner} \longrightarrow iren_D\downarrow; IR_o\downarrow [] RC_{i,outer} \longrightarrow \textbf{skip}], R_o\downarrow)),$
   $([(p(reg) \wedge \neg RC_o^e) \vee t(reg)]; ren_C\downarrow;$
      $[RC_{i,inner} \longrightarrow [(p(reg) \wedge \neg IRC_o^e) \vee t(reg)]; iren_C\downarrow;$
        $((unlock; IRC_o\downarrow), ([\neg IRC_i]; IRC_i^e\uparrow))$
      $[] RC_{i,outer} \longrightarrow unlock; RC_o\downarrow$
      $]),$
   $([\neg ren_D \wedge \neg ren_C \wedge \neg RC_i]; RC_i^e\uparrow)$
  $)$
 $]$
```

Program E.12 HSE: PCEVHB WAD read port with nested data, full-buffered data output, and half-buffered control propagation

```
$*[(([R_o^e]; ren_D\uparrow);$
    $[RC_{i,inner} \longrightarrow iren_D\uparrow; [IRC_i]; IR_o\uparrow; R_o\uparrow [] RC_{i,outer} \longrightarrow R_o\uparrow]),$
  $(([RC_o^e]; ren_C\uparrow);$
    $[RC_{i,inner} \longrightarrow [IRC_o^e]; iren_C\uparrow; [IRC_i];$
      $[p(reg) \wedge unlocked() \longrightarrow lock; IRC_o\uparrow [] t(reg) \longrightarrow \textbf{skip}];$
      $(IRC_i^e\downarrow, ([R_o]; RC_i^e\downarrow))$
    $[] RC_{i,outer} \longrightarrow [p(reg) \wedge unlocked() \longrightarrow lock; RC_o\uparrow [] t(reg) \longrightarrow \textbf{skip}];$
      $[R_o]; RC_i^e\downarrow$
    $]);$
  $(([\neg R_o^e]; ren_D\downarrow;$
     $([RC_{i,inner} \longrightarrow iren_D\downarrow; IR_o\downarrow [] RC_{i,outer} \longrightarrow \textbf{skip}], R_o\downarrow)),$
    $([(p(reg) \wedge \neg RC_o^e) \vee t(reg)]; ren_C\downarrow;$
      $[RC_{i,inner} \longrightarrow [(p(reg) \wedge \neg IRC_o^e) \vee t(reg)]; iren_C\downarrow;$
        $(unlock; IRC_o\downarrow; ([\neg ren_D \wedge \neg RC_i]; RC_i^e\uparrow), ([\neg IRC_i]; IRC_i^e\uparrow))$
      $[] RC_{i,outer} \longrightarrow unlock; RC_o\downarrow; [\neg ren_D \wedge \neg RC_i]; RC_i^e\uparrow$
      $])$
  $)$
 $]$
```

Program E.13 HSE: PCEVFB WAD write port, with nested data, unconditional outer write-enable, conditional inner write-enable variation

```
$*[[WC_o^e]; wen\uparrow;$
   $[WC_{i,inner} \longrightarrow$
     $[p(IW) \wedge IWC_o^e \longrightarrow iwen\uparrow;$
     $[unlocked()]; lock; IWC_o\uparrow [] t(IW) \longrightarrow \textbf{skip}];$
     $(IWC_i^e\downarrow, ([wvc]; WC_i^e\downarrow))$
   $[] WC_{i,outer} \longrightarrow$
     $[p(W_i) \wedge unlocked() \longrightarrow lock; WC_o\uparrow [] t(W_i) \longrightarrow \textbf{skip}];$
     $[wvc]; WC_i^e\downarrow;$
   $];$
   $[(p(W_i) \wedge \neg WC_o^e) \vee t(W_i)]; wen\downarrow;$
   $[WC_{i,inner} \longrightarrow [\neg IWC_o^e]; iwen\downarrow;$
     $((unlock; IWC_o\downarrow), [\neg IWC_i \longrightarrow IWC_i^e\uparrow, ([\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow)])$
   $[] WC_{i,outer} \longrightarrow (unlock; WC_o\downarrow), ([\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow)$
   $]$
 $]$
$\|$
$*[[WC_{i,inner} \longrightarrow [W_i]; IW\uparrow; \langle write_{inner}\rangle [] WC_{i,outer} \longrightarrow [W_i]; \langle write_{outer}\rangle];$
  $wvc\uparrow; [\neg W_i]; (IW\downarrow; wvc\downarrow)$
 $]$
```

Program E.14 HSE: PCEVHB WAD write port, with nested data, unconditional outer write-enable, conditional inner write-enable variation

```
$*[[WC_o^e]; wen\uparrow;$
  $[WC_{i,inner} \longrightarrow$
    $[p(IW) \wedge IWC_o^e \longrightarrow iwen\uparrow;$
    $[unlocked()]; lock; IWC_o\uparrow [] t(IW) \longrightarrow \textbf{skip}];$
    $(IWC_i^e\downarrow, ([wvc]; WC_i^e\downarrow))$
  $[] WC_{i,outer} \longrightarrow$
    $[p(W_i) \wedge unlocked() \longrightarrow lock; WC_o\uparrow [] t(W_i) \longrightarrow \textbf{skip}];$
    $[wvc]; WC_i^e\downarrow;$
  $];$
  $[(p(W_i) \wedge \neg WC_o^e) \vee t(W_i)]; wen\downarrow;$
  $[WC_{i,inner} \longrightarrow [\neg IWC_o^e]; iwen\downarrow;$
    $((unlock; IWC_o\downarrow), [\neg IWC_i \longrightarrow IWC_i^e\uparrow, ([\neg IWC_o \wedge \neg WC_i \wedge \neg wvc]; WC_i^e\uparrow)]])$
  $[] WC_{i,outer} \longrightarrow unlock; WC_o\downarrow; [\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow$
  $]$
 $]$
$\|$
$*[[WC_{i,inner} \longrightarrow [W_i]; IW\uparrow; \langle write_{inner}\rangle [] WC_{i,outer} \longrightarrow [W_i]; \langle write_{outer}\rangle];$
  $wvc\uparrow; [\neg W_i]; (IW\downarrow; wvc\downarrow)$
 $]$
```

Program E.15 HSE: PCEVFB WAD write port, with nested data, conditional outer write-enable, conditional inner write-enable variation

```
$*[[p(W_i) \wedge WC_o^e \longrightarrow wen\uparrow;$
    $[WC_{i,inner} \longrightarrow$
      $[p(IW) \wedge IWC_o^e \longrightarrow iwen\uparrow;$
      $[unlocked()]; lock; IWC_o\uparrow [] t(IW) \longrightarrow \textbf{skip}];$
      $(IWC_i^e\downarrow, ([wvc]; WC_i^e\downarrow))$
    $[] WC_{i,outer} \longrightarrow [unlocked()]; lock; WC_o\uparrow; [wvc]; WC_i^e\downarrow;$
    $]$
  $[] t(W_i) \longrightarrow \textbf{skip}$
  $];$
  $[\neg WC_o^e]; wen\downarrow;$
  $[WC_{i,inner} \longrightarrow [\neg IWC_o^e]; iwen\downarrow;$
    $((unlock; IWC_o\downarrow), [\neg IWC_i \longrightarrow IWC_i^e\uparrow, ([\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow)]])$
  $[] WC_{i,outer} \longrightarrow (unlock; WC_o\downarrow), ([\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow)$
  $]$
 $]$
$\|$
$*[[WC_{i,inner} \longrightarrow [W_i]; IW\uparrow; \langle write_{inner}\rangle [] WC_{i,outer} \longrightarrow [W_i]; \langle write_{outer}\rangle];$
  $wvc\uparrow; [\neg W_i]; (IW\downarrow; wvc\downarrow)$
 $]$
```

Program E.16 HSE: PCEVFB WAD write port, with nested data, conditional outer write-enable, conditional inner write-enable variation

```
$*[[p(W_i) \wedge WC_o^e \longrightarrow wen\uparrow;$
    $[WC_{i,inner} \longrightarrow$
      $[p(IW) \wedge IWC_o^e \longrightarrow iwen\uparrow;$
      $[unlocked()]; lock; IWC_o\uparrow [] t(IW) \longrightarrow \textbf{skip}];$
      $(IWC_i^e\downarrow, ([wvc]; WC_i^e\downarrow))$
    $[] WC_{i,outer} \longrightarrow [unlocked()]; lock; WC_o\uparrow; [wvc]; WC_i^e\downarrow;$
    $]$
  $[] t(W_i) \longrightarrow \textbf{skip}$
  $];$
  $[\neg WC_o^e]; wen\downarrow;$
  $[WC_{i,inner} \longrightarrow [\neg IWC_o^e]; iwen\downarrow;$
    $((unlock; IWC_o\downarrow), [\neg IWC_i \longrightarrow IWC_i^e\uparrow, ([\neg IWC_o \wedge \neg WC_i \wedge \neg wvc]; WC_i^e\uparrow)])$
  $[] WC_{i,outer} \longrightarrow unlock; WC_o\downarrow; [\neg WC_i \wedge \neg wvc]; WC_i^e\uparrow$
  $]$
 $]$
$\|$
$*[[WC_{i,inner} \longrightarrow [W_i]; IW\uparrow; \langle write_{inner}\rangle [] WC_{i,outer} \longrightarrow [W_i]; \langle write_{outer}\rangle];$
  $wvc\uparrow; [\neg W_i]; (IW\downarrow; wvc\downarrow)$
 $]$
```

Appendix F

Partial HSEs of the Core

This appendix contains partial HSEs for the various components of the *CORE* floor decompositions throughout the thesis.

F.1 Non-WAD Core

The partial HSEs for the base design (non-width-adaptive) core appear in the text of Section 4.2.

F.2 WAD Core

The WAD floor decompositions follow very closely to the non-WAD floor decompositions and are not repeated in the thesis. We provide the resulting HSEs for the decomposed control components in this section.

F.2.1 Reading Control

Program F.1 HSE: WAD read control propagation array, where the termination condition only sets RC_o^f

```
$REG\_CTRL\_PROP_{read,wad}[l] \equiv$
    $*[[RC_i[l] \wedge ren_C];$
      $[dx^0[l] \wedge \langle unlocked[l] \rangle \longrightarrow lock_r[l]\uparrow; RC_o[l]\uparrow; RC_o^v\uparrow$
      $[] dx^1[l] \longrightarrow RC_o^f\uparrow];$
      $[\neg ren_C]; lock_r[l]\downarrow; RC_o[l]\downarrow; RC_o^v\downarrow$
    $]$
```

Program F.2 HSE: WAD read handshake control (full buffer)

```
$REG\_HSEN_{read,wad,fullbuf} \equiv$
    $*[[RC_o^e]; ren_C\uparrow; [(RC_o^f \vee RC_o^v) \wedge RC_i^v \wedge ren^v \wedge R^v]; RC_i^e\downarrow;$
      $[RC_o^f \vee \neg RC_o^e]; ren_C\downarrow; (RC_o^f\downarrow, ([\neg RC_i^v \wedge \neg ren^v]; RC_i^e\uparrow)); [\neg RC_o^v]$
    $]$
```

Program F.3 HSE: WAD read handshake control (full buffered propagation, half-buffered termination)

```
$REG\_HSEN_{read,wad,fullbuf} \equiv$
    $*[[RC_o^e]; ren_C\uparrow; [(RC_o^f \vee RC_o^v) \wedge RC_i^v \wedge ren^v \wedge R^v]; RC_i^e\downarrow;$
      $[RC_o^f \vee \neg RC_o^e]; ren_C\downarrow; RC_o^f\downarrow; [\neg RC_i^v \wedge \neg ren^v]; RC_i^e\uparrow; [\neg RC_o^v]$
    $]$
```

F.2.2 Writing Control, Unconditional Write-Enable

Program F.4 HSE: the WAD write control propagation array, for unconditional write-enable

```
$REG\_CTRL\_PROP_{write,wad,uwen}\,[l] \equiv$
    $*[[\,WC_i\,[l] \wedge wen \wedge dW^0\,[l] \wedge \langle unlocked\,[l]\rangle];$
        $lock_w\,[l]\uparrow;\ WC_o\,[l]\uparrow;\ WC_o^v\uparrow;$
      $[\neg wen];\ lock_w\,[l]\downarrow;\ WC_o\,[l]\downarrow;\ WC_o^v\downarrow$
     $]$
```

Program F.5 HSE: WAD write handshake control, with unconditional write-enable (full buffer)

```
$REG\_HSEN_{write,wad,uwen,fullbuf} \equiv$
    $*[[\,WC_o^e];\ wen\uparrow;\ [dW^1 \longrightarrow WC_o^f\uparrow [\!]\,\textbf{else} \longrightarrow \textbf{skip}];$
      $[(\,WC_o^f \vee WC_o^v) \wedge WC_i^v \wedge wvc];\ WC_i^e\downarrow;$
      $[\,WC_o^f \vee \neg\, WC_o^e];\ wen\downarrow;\ (\,WC_o^f\downarrow,\ [\neg\, WC_i^v \wedge \neg wvc];\ WC_i^e\uparrow);\ [\neg\, WC_o^v]$
     $]$
```

Program F.6 HSE: WAD write handshake control, with unconditional write-enable (full buffer propagation, half buffer termination)

```
$REG\_HSEN_{write,wad,uwen,fullbuf} \equiv$
    $*[[\,WC_o^e];\ wen\uparrow;\ [dW^1 \longrightarrow WC_o^f\uparrow [\!]\,\textbf{else} \longrightarrow \textbf{skip}];$
      $[(\,WC_o^f \vee WC_o^v) \wedge WC_i^v \wedge wvc];\ WC_i^e\downarrow;$
      $[\,WC_o^f \vee \neg\, WC_o^e];\ wen\downarrow;\ WC_o^f\downarrow;\ [\neg\, WC_i^v \wedge \neg wvc];\ WC_i^e\uparrow;\ [\neg\, WC_o^v]$
     $]$
```

F.2.3 Writing Control, Conditional Write-Enable

Program F.7 HSE: WAD write control propagation array, with conditional write-enable

```
$REG\_CTRL\_PROP_{write}$ [$l$] $\equiv$
    *[[$WC_i$[$l$] $\wedge$ $wen$ $\wedge$ $\langle unlocked$[$l$]$\rangle$]; $lock_w$[$l$]$\uparrow$; $WC_o$[$l$]$\uparrow$; $WC_o^v\uparrow$;
      [$\neg wen$]; $lock_w$[$l$]$\downarrow$; $WC_o$[$l$]$\downarrow$; $WC_o^v\downarrow$
     ]
```

Program F.8 HSE: WAD write handshake control, conditional write-enable (full buffer)

```
$REG\_HSEN_{write,wad,cwen,fullbuf}$ $\equiv$
    *[[$dW^0 \longrightarrow$ [$WC_o^e$]; $wen\uparrow$ [] $dW^1 \longrightarrow$ skip];
      [($dW^1 \vee WC_o^v$) $\wedge$ $WC_i^v$ $\wedge$ $wvc$]; $WC_i^e\downarrow$;
      [$\neg WC_o^e$]; $wen\downarrow$; [$\neg WC_i^v \wedge \neg wvc$]; $WC_i^e\uparrow$; [$\neg WC_o^v$]
     ]
```

Program F.9 HSE: WAD write handshake control, conditional write-enable (half buffer)

```
$REG\_HSEN_{write,wad,cwen,halfbuf}$ $\equiv$
    *[[$dW^0 \longrightarrow$ [$WC_o^e$]; $wen\uparrow$ [] $dW^1 \longrightarrow$ skip];
      [($dW^1 \vee WC_o^v$) $\wedge$ $WC_i^v$ $\wedge$ $wvc$]; $WC_i^e\downarrow$;
      [$\neg WC_o^e$]; $wen\downarrow$; [$\neg WC_o^v \wedge \neg WC_i^v \wedge \neg wvc$]; $WC_i^e\uparrow$;
     ]
```

F.3 Non-WAD Nested Core

The decompositions of the non-WAD nested core read and write ports are discussed in Section 9.4.

F.3.1 Modified Data Interface

The modifications to the peripheral data interface required by the nesting transformation are described in Sections 9.4.1 and 9.4.2.

Program F.10 HSE: the register read data interface with $\_R$ reset, modified for use with nested data arrays

```
$REG\_INTRFC_{read,nested}[b] \equiv$
    $*[[R^e \wedge RC_i^e \wedge \neg IR_o^v]; ren_D[b]\uparrow; [\neg\_R[b]]; R[b]\uparrow;$
      $[\neg R^e \wedge \neg RC_i^e]; ren_D[b]\downarrow; \_R[b]\uparrow; R[b]\downarrow$
     $]$
```

Program F.11 HSE: resetting the write validity bitline

```
$REG\_INTRFC_{write,nested}[b] \equiv$
    $*[[\neg W_i[b] \wedge \_iwv]; \_wv[b]\uparrow]$
```

F.3.2 Nested Data Interconnect

The partial HSEs of the nested data interconnect component are described in Sections 9.4.1 and 9.4.2.

Program F.12 HSE: the nested interconnect component between the inner and outer partition of the nested read port data array

```
*[[$ren_D \wedge C_{i,inner}$]; $iren_D\uparrow$; [$\neg$ _$IR$]; $IR\uparrow$; $IR^v\uparrow$; $iren_D\downarrow$; _$R\downarrow$;
  [$\neg ren_D$]; _$IR\uparrow$; $IR\downarrow$; $IR^v\downarrow$
 ]
```

Program F.13 HSE: a single bit of the data component of a data-independent control-data join, with nested data

```
*[[$WC_{i,inner} \longrightarrow$ [$W_i$]; $IW\uparrow$; $\langle write_{inner}\rangle$; _$iwv\downarrow$
  [] $WC_{i,outer} \longrightarrow$ [$W_i$]; $\langle write_{outer}\rangle$
  ];
  _$wv\downarrow$;
  $IW\downarrow$; _$iwv\uparrow$;
  [$\neg W_i$]; _$wv\uparrow$
 ]
```

Program F.14 HSE: the nested interconnect component between the inner and outer partition of the nested write port array

```
*[[$D_i \wedge C_{i,inner}$]; $ID\uparrow$; [$\neg$ _$iwv$]; _$wv\downarrow$;
  $ID\downarrow$; _$iwv\uparrow$
 ]
```

F.3.3 Nested Control Interconnect

The partial HSEs of the non-WAD nested control interconnect component are described in Sections 9.4.3 and 9.4.5.

Program F.15 HSE: nested interconnect component between the inner and outer partitions' non-WAD read control propagation arrays

```
*[[$ren_C \wedge IC_i^v \wedge IC_o^e$]; $iren_C\uparrow$; [$IC_o^v$]; $IC_i^e\downarrow$;
  [$\neg ren_C \wedge \neg IC_o^e$]; $iren_C\downarrow$; [$\neg IC_i^v$]; $IC_i^e\uparrow$; [$\neg IC_o^v$]
 ]
```

Program F.16 HSE: nested interconnect component between the inner and outer partitions' non-WAD write control propagation arrays

```
*[[$wen \wedge IWC_i^v \wedge IWC_o^e$]; $iwen\uparrow$; [$IWC_o^v$]; $IWC_i^e\downarrow$;
  [$\neg wen \wedge \neg IWC_o^e$]; $iwen\downarrow$; [$\neg IWC_i^v$]; $IWC_i^e\uparrow$; [$\neg IWC_o^v$]
 ]
```

F.4 WAD Nested Core

The decompositions of the WAD nested core read and write ports are discussed in Section 9.4.

F.4.1 Reading

The partial HSEs for the control components of the WAD nested read port are discussed in Section 9.4.4.

Program F.17 HSE: nested interconnect component between the inner and outer partitions' WAD read control propagation arrays

```
*[[$ren_C \wedge IRC_i^v \wedge IRC_o^e$]; $iren_C\uparrow$;
  [$IRC_o^f \longrightarrow RC_o^f\uparrow, IRC_i^e\downarrow$ ▯ $IRC_o^v \longrightarrow IRC_i^e\downarrow$];
  [$(\neg ren_C \wedge \neg IRC_o^e) \vee IRC_o^f$]; $iren_C\downarrow$;
  [$\neg IRC_i^v$]; $IRC_i^e\uparrow$; [$\neg IRC_o^v \wedge \neg IRC_o^f$]
 ]
```

Program F.18 HSE: WAD nested read handshake control (full buffer)

$REG\_HSEN_{read,wad,nested,fullbuf} \equiv$

```
*[[$RC_o^e \wedge \neg IRC_o^f$]; $ren_C\uparrow$; [$(RC_o^f \vee RC_o^v) \wedge RC_i^v \wedge ren^v \wedge R^v$]; $RC_i^e\downarrow$;
  [$RC_o^f \vee \neg RC_o^e$]; $ren_C\downarrow$; $(RC_o^f\downarrow$, ([$\neg RC_i^v \wedge \neg ren^v$]; $RC_i^e\uparrow$)); [$\neg RC_o^v$]
 ]
```

Program F.19 HSE: WAD nested read handshake control (full-buffered propagation, half-buffered termination)

$REG\_HSEN_{read,wad,nested,fullbuf} \equiv$

```
*[[$RC_o^e \wedge \neg IRC_o^f$]; $ren_C\uparrow$; [$(RC_o^f \vee RC_o^v) \wedge RC_i^v \wedge ren^v \wedge R^v$]; $RC_i^e\downarrow$;
  [$RC_o^f \vee \neg RC_o^e$]; $ren_C\downarrow$; $RC_o^f\downarrow$; [$\neg RC_i^v \wedge \neg ren^v$]; $RC_i^e\uparrow$; [$\neg RC_o^v$]
 ]
```

F.4.2 Writing

The partial HSEs for the control components of the WAD nested write ports are discussed in Sections 9.4.6 and 9.4.6. Note that the nested data connect for the *delimiter bit* (HSE Program F.20) is slightly modified from the interconnect for the non-delimiter bits (HSE Program F.14).

Program F.20 HSE: the nested interconnect component between the delimiter bit of the inner and outer partition of the nested write port array

```
$*[[dW_i \wedge IWC_i^v]; dIW\uparrow; [\neg\_iwv]; \_wv\downarrow;$
  $[\neg IWC_i^e]; dIW\downarrow; \_iwv\uparrow$
 ]
```

Program F.21 HSE: control nested interconnect between inner and outer partitions of WAD nested write handshake control, unconditional outer write-enable

```
$*[[dIW^0 \longrightarrow [IWC_o^e \wedge wen]; iwen\uparrow [] dIW^1 \longrightarrow \textbf{skip}];$
  $[(dIW^1 \vee IWC_o^v) \wedge IWC_i^v]; IWC_i^e\downarrow;$
  $[\neg IWC_o^e \wedge \neg wen]; iwen\downarrow; [\neg IWC_i^v]; IWC_i^e\uparrow; [\neg IWC_o^v]$
 ]
```

Program F.22 HSE: control nested interconnect between inner and outer partitions of WAD nested write handshake control, conditional outer write-enable

```
$*[[dIW^0 \longrightarrow [IWC_o^e \wedge wen]; iwen\uparrow [] dIW^1 \longrightarrow \textbf{skip}];$
  $[(dIW^1 \vee IWC_o^v) \wedge IWC_i^v]; IWC_i^e\downarrow;$
  $[\neg IWC_o^e \wedge \neg wen]; iwen\downarrow; [\neg IWC_i^v]; IWC_i^e\uparrow; [\neg IWC_o^v]$
 ]
```

Appendix G

Reset Convention

This appendix explains the reset signals that are found in the production rules for the register file.

G.1 Global Reset Signals

The global reset convention we use for our circuits follows closely to those presented in Nyström's dissertation, which was an answer to the problems found in the Caltech MiniMIPS' reset convention [31, 34]. The MiniMIPS used only two reset signals, *Reset* and *Reset*_, to clear and initialize the state of the pipelines. They allowed moments of interference with the assumption that interference would be resolved in a limited amount of time. The major problem with that scheme was the inevitable timing assumption about the delay from *Reset*↓ to *Reset*_↑ and their respective rise and fall (slew) rates. For the same reasons given by Nyström, we introduce new reset signals to avoid timing problems [34].

The reset convention we use occurs in multiple phases. The first step we take is to cut-off critical production rules with series transistors gated by global reset signals. In NFET pull-down rules, one uses _*sReset* in series to cut-off, and in PFET pull-up rules, one uses *sReset* in series to cut-off. (The *s* is for *series*.) In our production rules, we forbid the use of *sReset* because of the negative impact on performance of series PFETs. Thus, eliminating this type of reset signal restricts the places where we may cut-off to only pull-down production rules. No matter what the state of pipelines is in the entire asynchronous system, applying the series cut-off reset will cause the system to halt in a limited amount of time. The longest path between series cut-off resets characterizes the upper bound on the time it takes to stabilizes to the halted state.

Assuming that all nodes with series cut-off resets are properly staticized, one may safely apply *parallel* resets to force nodes to switch into certain states. We use _*pReset*↓ to set a node high, and *pReset*↑ to set a node low.[1] (The *p* is for

[1] With careful planning, we were able to entirely eliminate *pReset* from the core production rule set, which left only two global reset signals to route in the core.

parallel.) The requirement is that before the parallel resets can be applied, that the opposing transistor network is guaranteed to be off, either directly by the series cut-off resets or by propagation. (A similar optimization may be used in the staticizers to cut-off the opposing weak transistors.) The global parallel resets force a set of nodes into a state that propagates throughout the system, which eventually arrives at a known initial state. (This time may characterized by paths between reset nodes.) In the initial state, the parallel resets may be de-asserted leaving staticizers to hold the state of dynamic nodes.

Once the parallel resets are no longer driving, the final step in the reset protocol is to de assert the series cut-off resets, which allows the system to proceed. With independently-driven global reset signals, the *only* timing assumptions that enter into the reset protocol are the durations of each reset phase, which may be (arbitrarily) generously long to guarantee safety and non-interference.

To summarize:

1. Assert series cut-off reset signals
2. Assert parallel reset signals to switch critical nodes
3. De-assert parallel reset signals
4. De-assert series cut-off reset signals

G.2 Handshake Protocol Reset State

This section addresses the question: *where* in the PCEVFB and PCEVHB reshufflings' HSEs is it *best* to reset? More specifically, what should the state of the input acknowledge and internal state enable be? Since our sub-system does not initialize with any tokens on start-up, the data outputs must be neutral on reset.

Our goal in choosing the handshake reset convention is to minimize the amount of reset circuitry, especially in nodes along the critical path. Each series transistor added weakens the driving strength of a transistor stack, and each parallel transistor added contributes the the parasitic capacitance on a node. We evaluate our choices, based on the state of the (active-low) acknowledge signal and internal state enable on reset.

Acknowledge low, enable low. Advantages: With the low internal enable automatically forces the precharge stacks of the data rails to reset to neutral (low output after the inverter), which makes data resetting very efficient and fast because the resetting of data does not ripple from pipeline stage to stage. The fact that the acknowledgment and enable are the same sense eliminates the need for opposing reset cut-offs on the path from acknowledgment to enable.[2]

[2] This is true when the internal enable is two gate transitions after the acknowledgments.

Disadvantages: The natural state of the (active-low) acknowledge is high, indicating to the input-senders that this stage is ready to accept new input tokens. One would need to force the acknowledge to be low, which implies additional series gating in the opposing pull-up network. Since the acknowledgment path is likely to be on the critical path of the handshake cycle time, we did not choose this convention.

Acknowledge low, enable high. Advantages: None.

Disadvantages: (same as with enable low) In addition, one would lose the advantage of not having to reset the output because the internal enable used for the precharge is high. The fact that the acknowledgment is opposite in sense to the enable means that cut-off and parallel resets are required in the internal enable production rules. As we've seen in the production rule derivation for the read port, the internal enables *ren* already have huge fanout to overcome. This is just a poor choice.

Acknowledge high, enable low. Advantages: With enable low, no additional reset circuits are needed to reset data low. Acknowledge-high is the natural state of the acknowledge when a stage is ready to accept new inputs.

Disadvantages: The fact that the acknowledgment is opposite in sense to the enable means that cut-off and parallel resets are required in the internal enable production rules.

Acknowledge high, enable high. Advantages: (same as with previous case) The fact that the acknowledgment and enable are the same sense eliminates the need for opposing reset cut-offs on the path from acknowledgment to enable.

Disadvantages: With enable high, one needs reset circuits in the output of the precharge data stages. However, that overhead can be minimized by cutting off the pull-down of the feedback staticizer with *_sReset* and resetting high with a minimum-size *_pReset*, resulting in low data on reset. However, each precharge stage depends on the previous stage being reset, which creates a ripple-dependency on the data reset, possibly prolonging the middle phase of resetting. Since the reset time is not critical to the performance of our system, this is not an issue.

Conclusion: Choosing to reset with acknowledgment high and enable high resulted in the least overhead for reset circuits. We have argued that the disadvantages of this convention are far outweighed by its advantages. This convention almost eliminates reset overhead from nodes that are likely to be on the critical path. We have strongly leveraged the transformation where reset signals may be implemented in the staticizers to minimize the negative impact on important signals.

It should be noted that our optimal choice is not necessarily the general optimal choice. When choosing a reset convention, one must consider the chosen reshuffling, and its implications on implementation at the circuit level.

Appendix H

Core PRS

This appendix contains the production rules for all variations of the register core. The PRS presented here, however, do not correspond *exactly* to the circuits we have laid out and simulated. These circuits given here and throughout the text are presented for ease of understanding because they correspond precisely to the partial handshaking expansions of the floor decompositions. The actual circuits resulted from transformations that moved around completion logic to reduce fanout along the critical path of the handshake, particularly in the acknowledgment generation. These optimizations are described only in the technical report [11].

H.1 Register Cell Array

The register cell is illustrated in Figure 4.11.

Program H.1 PRS: core register cell, single ported

$$\begin{array}{lcl}
\neg x^1 & \rightarrow & x^0\uparrow \\
\neg x^0 & \rightarrow & x^1\uparrow \\
x^1 & \rightarrow & x^0\downarrow \\
x^0 & \rightarrow & x^1\downarrow \\
WC_i \wedge W^0 & \rightarrow & x^1\downarrow \\
WC_i \wedge W^1 & \rightarrow & x^0\downarrow \\
W^0 \wedge x^0 \wedge WC_i & \rightarrow & \_wv\downarrow \\
W^1 \wedge x^1 \wedge WC_i & \rightarrow & \_wv\downarrow \\
ren \wedge RC_i \wedge x^0 & \rightarrow & \_R^0\downarrow \\
ren \wedge RC_i \wedge x^1 & \rightarrow & \_R^1\downarrow
\end{array}$$

Program H.2 PRS: core register cell hard-wired to zero, single ported

$$\begin{array}{lcl}
(W^0 \vee W^1) \wedge WC_i^0 & \rightarrow & \_wv\downarrow \\
ren_D \wedge RC_i^0 & \rightarrow & \_R^0\downarrow
\end{array}$$

H.2 Data Nested Interconnect

PRS H.3 is illustrated in Figure 9.15, and PRS H.4 is illustrated in Figure 9.16.

Program H.3 PRS: delay-insensitive interface cell bewteen the data bits of inner and outer banks of a nested register array, shown for a single read port

$$
\begin{array}{lcl}
\neg \_pReset & \rightarrow & \_IR^0\uparrow \\
\neg \_pReset & \rightarrow & \_IR^1\uparrow \\
\neg \_pReset & \rightarrow & \_iren_D\uparrow \\
IRC_i^v \wedge ren_D \wedge \_IR^v & \rightarrow & \_iren_D\downarrow \\
\neg \_iren_D & \rightarrow & iren_D\uparrow \\
iren_D & \rightarrow & iren\_{}_D\downarrow \\
\neg \_IR^0 \wedge \neg iren\_{}_D & \rightarrow & IR^0\uparrow \\
\neg \_IR^1 \wedge \neg iren\_{}_D & \rightarrow & IR^1\uparrow \\
IR^0 \vee IR^1 & \rightarrow & \_IR^v\downarrow \\
\neg \_IR^v & \rightarrow & \_iren_D\uparrow \\
\_iren_D & \rightarrow & iren_D\downarrow \\
\neg iren_D & \rightarrow & iren\_{}_D\uparrow \\
IR^0 \wedge ren_D \wedge iren\_{}_D & \rightarrow & \_R^0\downarrow \\
IR^1 \wedge ren_D \wedge iren\_{}_D & \rightarrow & \_R^1\downarrow \\
\neg ren_D \wedge \neg iren_D & \rightarrow & \_IR^0\uparrow \\
\neg ren_D \wedge \neg iren_D & \rightarrow & \_IR^1\uparrow \\
\_IR^0 & \rightarrow & IR^0\downarrow \\
\_IR^1 & \rightarrow & IR^1\downarrow \\
\neg IR^0 \wedge \neg IR^1 & \rightarrow & \_IR^v\uparrow
\end{array}
$$

Program H.4 PRS: delay-insensitive interface cell bewteen the data bits of inner and outer banks of a nested register array, shown for a single write port

$\neg\_pReset$	$\rightarrow$	$\_IW^0\uparrow$
$\neg\_pReset$	$\rightarrow$	$\_IW^1\uparrow$
$W^0 \wedge \_wv \wedge IWC_i^v$	$\rightarrow$	$\_IW^0\downarrow$
$W^1 \wedge \_wv \wedge IWC_i^v$	$\rightarrow$	$\_IW^1\downarrow$
$\neg\_IW^0$	$\rightarrow$	$IW^0\uparrow$
$\neg\_IW^1$	$\rightarrow$	$IW^1\uparrow$
$\neg\_IW^v$	$\rightarrow$	$IW^v\uparrow$
IW^v	$\rightarrow$	$\_wv\downarrow$
$\neg\_wv$	$\rightarrow$	$\_IW^0\uparrow$
$\neg\_wv$	$\rightarrow$	$\_IW^1\uparrow$
$\_IW^0$	$\rightarrow$	$IW^0\downarrow$
$\_IW^1$	$\rightarrow$	$IW^1\downarrow$
$\neg IW^0 \wedge \neg IW^1$	$\rightarrow$	$\_IW^v\uparrow$
$\_IW^v$	$\rightarrow$	$IW^v\downarrow$

Program H.5 PRS: delay-insensitive interface cell bewteen the delimiter bits of inner and outer banks of a nested register array, used with conditional outer write-enable, shown for a single write port

$\neg\_pReset$	$\rightarrow$	$\_dIW^0\uparrow$
$\neg\_pReset$	$\rightarrow$	$\_dIW^1\uparrow$
$dW^0 \wedge \_wv \wedge dIWC_i^v$	$\rightarrow$	$\_dIW^0\downarrow$
$dW^1 \wedge \_wv \wedge dIWC_i^v$	$\rightarrow$	$\_dIW^1\downarrow$
$\neg\_dIW^0$	$\rightarrow$	$dIW^0\uparrow$
$\neg\_dIW^1$	$\rightarrow$	$dIW^1\uparrow$
$\neg\_dIW^v$	$\rightarrow$	$dIW^v\uparrow$
dIW^v	$\rightarrow$	$\_wv\downarrow$
$\neg\_wv$	$\rightarrow$	$\_dIW^0\uparrow$
$\neg\_wv \wedge \neg IWC_i^e$	$\rightarrow$	$\_dIW^1\uparrow$
$\_dIW^0$	$\rightarrow$	$dIW^0\downarrow$
$\_dIW^1$	$\rightarrow$	$dIW^1\downarrow$
$\neg dIW^0 \wedge \neg dIW^1$	$\rightarrow$	$\_dIW^v\uparrow$
$\_dIW^v$	$\rightarrow$	$dIW^v\downarrow$

Program H.6 PRS: delay-insensitive interface cell bewteen the delimiter bits of inner and outer banks of a nested register array, used with unconditional outer write-enable, shown for a single write port

$$\begin{array}{lcl}
\neg\_pReset & \rightarrow & \_dIW^0\uparrow \\
\neg\_pReset & \rightarrow & \_dIW^1\uparrow \\
dW^0 \wedge \_wv \wedge dIWC_i^v & \rightarrow & \_dIW^0\downarrow \\
dW^1 \wedge \_wv \wedge dIWC_i^v & \rightarrow & \_dIW^1\downarrow \\
\neg\_dIW^0 & \rightarrow & dIW^0\uparrow \\
\neg\_dIW^1 & \rightarrow & dIW^1\uparrow \\
\neg\_dIW^v & \rightarrow & dIW^v\uparrow \\
dIW^v & \rightarrow & \_wv\downarrow \\
\neg\_wv \wedge \neg IWC_i^e & \rightarrow & \_dIW^0\uparrow \\
\neg\_wv \wedge \neg IWC_i^e & \rightarrow & \_dIW^1\uparrow \\
\_dIW^0 & \rightarrow & dIW^0\downarrow \\
\_dIW^1 & \rightarrow & dIW^1\downarrow \\
\neg dIW^0 \wedge \neg dIW^1 & \rightarrow & \_dIW^v\uparrow \\
\_dIW^v & \rightarrow & dIW^v\downarrow
\end{array}$$

H.3 Control Propagation Array

The unconditional read and write control propagators (for two ports) are illustrated respectively in Figures 4.12 and 4.13. The WAD read and write control propagators are illustrated respectively in Figures 5.10 and 5.11 (unconditional write-enable).

Program H.7 PRS: unconditional read control propagation with locking, for two ports with $p = 0, 1$ $(q = 1 - p)$

$$\begin{array}{lll}
\neg\_pReset & \rightarrow & \_RC_o[p]\uparrow \\
ren[p] \wedge \_WC_o[p] \wedge \_WC_o[q] \wedge RC_i[p] & \rightarrow & \_RC_o[p]\downarrow \\
\neg\_RC_o[p] & \rightarrow & RC_o[p]\uparrow \\
\neg ren[p] & \rightarrow & \_RC_o[p]\uparrow \\
\_RC_o[p] & \rightarrow & RC_o[p]\downarrow
\end{array}$$

Program H.8 PRS: unconditional write control propagation with locking, for two ports with $p = 0, 1$ $(q = 1 - p)$

$$\begin{array}{lll}
\neg\_pReset & \rightarrow & \_WC_o[p]\uparrow \\
wen[p] \wedge \_WC_o[q] \wedge \_RC_o[p] \wedge \_RC_o[q] \wedge WC_i[p] & \rightarrow & \_WC_o[p]\downarrow \\
\neg\_WC_o[p] & \rightarrow & WC_o[p]\uparrow \\
\neg wen[p] & \rightarrow & \_WC_o[p]\uparrow \\
\_WC_o[p] & \rightarrow & WC_o[p]\downarrow
\end{array}$$

Program H.9 PRS: unconditional read/write control propagation without locking (for register 0), for a single port

$$\begin{array}{lll}
\neg\_pReset & \rightarrow & \_WC_o^0\uparrow \\
\neg\_pReset & \rightarrow & \_RC_o^0\uparrow \\
ren \wedge RC_i^0 & \rightarrow & \_RC_o^0\downarrow \\
\neg\_RC_o^0 & \rightarrow & RC_o^0\uparrow \\
\neg ren & \rightarrow & \_RC_o^0\uparrow \\
\_RC_o^0 & \rightarrow & RC_o^0\downarrow \\
wen \wedge WC_i^0 & \rightarrow & \_WC_o^0\downarrow \\
\neg\_WC_o^0 & \rightarrow & WC_o^0\uparrow \\
\neg wen & \rightarrow & \_WC_o^0\uparrow \\
\_WC_o^0 & \rightarrow & WC_o^0\downarrow
\end{array}$$

Program H.10 PRS: WAD conditional read control propagation with locking, for two ports with $p = 0, 1$ $(q = 1 - p)$

$$\begin{array}{ll}
\neg\_pReset & \rightarrow\ \_RC_o[p]\uparrow \\
ren[p] \wedge \_WC_o[p] \wedge \_WC_o[q] \wedge RC_i[p] \wedge dx^0 & \rightarrow\ \_RC_o[p]\downarrow \\
ren[p] \wedge RC_i[p] \wedge dx^1 & \rightarrow\ \_RC_o^f[p]\downarrow \\
\neg\_RC_o[p] & \rightarrow\ RC_o[p]\uparrow \\
\neg ren[p] & \rightarrow\ \_RC_o[p]\uparrow \\
\_RC_o[p] & \rightarrow\ RC_o[p]\downarrow
\end{array}$$

Program H.11 PRS: WAD conditional write control propagation with locking, and unconditional write-enable *wen*, for two ports with $p = 0, 1$ $(q = 1 - p)$

$$\begin{array}{ll}
\neg\_pReset & \rightarrow\ \_WC_o[p]\uparrow \\
wen[p] \wedge \_WC_o[q] \wedge \_RC_o[p] \wedge \_RC_o[q] \wedge WC_i[p] \wedge dW^0 & \rightarrow\ \_WC_o[p]\downarrow \\
wen[p] \wedge WC_i[p] \wedge dW^1 & \rightarrow\ \_WC_o^f[p]\downarrow \\
\neg\_WC_o[p] & \rightarrow\ WC_o[p]\uparrow \\
\neg wen[p] & \rightarrow\ \_WC_o[p]\uparrow \\
\_WC_o[p] & \rightarrow\ WC_o[p]\downarrow
\end{array}$$

H.4 Control Nested Interconnect

The nested control interconnects for the non-WAD read and write ports are illustrated in Figures 9.20 and 9.22 respectively. The nested control interconnects for the WAD read and write ports are illustrated in Figures 9.21 and 9.23 respectively.

Program H.12 PRS: delay-insensitive interface cell between inner and outer banks of nested, unconditional read control propagation array, single port

$$\begin{array}{lcl}
\neg\mathit{\_pReset} & \rightarrow & \mathit{\_iren}_C\uparrow \\
\neg\mathit{\_pReset} & \rightarrow & \mathit{IRC}_i^e\uparrow \\
\mathit{iren}_C \wedge \mathit{IRC}_o^v & \rightarrow & \mathit{IRC}_i^e\downarrow \\
\neg\mathit{IRC}_i^e \wedge \neg\mathit{ren}_C \wedge \neg\mathit{IRC}_o^e & \rightarrow & \mathit{\_iren}_C\uparrow \\
\mathit{\_iren}_C & \rightarrow & \mathit{iren}_C\downarrow \\
\neg\mathit{iren}_C \wedge \neg\mathit{IRC}_i^v & \rightarrow & \mathit{IRC}_i^e\uparrow \\
\mathit{IRC}_i^e \wedge \mathit{IRC}_o^e \wedge \mathit{ren}_C \wedge \mathit{IRC}_i^v & \rightarrow & \mathit{\_iren}_C\downarrow \\
\neg\mathit{\_iren}_C & \rightarrow & \mathit{iren}_C\uparrow
\end{array}$$

Program H.13 PRS: delay-insensitive interface cell between inner and outer banks of nested, unconditional write control propagation array, single port

$$\begin{array}{lcl}
\neg\mathit{\_pReset} & \rightarrow & \mathit{\_iwen}\uparrow \\
\neg\mathit{\_pReset} & \rightarrow & \mathit{IWC}_i^e\uparrow \\
\mathit{wen} \wedge \mathit{IWC}_i^v \wedge \mathit{IWC}_i^e \wedge \mathit{IWC}_o^e & \rightarrow & \mathit{\_iwen}\downarrow \\
\neg\mathit{\_iwen} & \rightarrow & \mathit{iwen}\uparrow \\
\mathit{iwen} \wedge \mathit{IWC}_o^v & \rightarrow & \mathit{IWC}_i^e\downarrow \\
\neg\mathit{wen} \wedge \neg\mathit{IWC}_i^e \wedge \neg\mathit{IWC}_o^e & \rightarrow & \mathit{\_iwen}\uparrow \\
\mathit{\_iwen} & \rightarrow & \mathit{iwen}\downarrow \\
\neg\mathit{iwen} \wedge \neg\mathit{IWC}_i^v & \rightarrow & \mathit{IWC}_i^e\uparrow
\end{array}$$

Program H.14 PRS: delay-insensitive interface cell between inner and outer banks of nested, WAD read control propagation array, single port

$$\begin{array}{lcl}
\neg\_pReset & \rightarrow & \_iren_C\uparrow \\
\neg\_pReset & \rightarrow & \_IRC^f_o\uparrow \\
\neg\_pReset & \rightarrow & IRC^e_i\uparrow \\
\neg\_IRC^f_o & \rightarrow & IRC^f_o\uparrow \\
IRC^f_o & \rightarrow & ircof\_\downarrow \\
ren_C \wedge IRC^f_o \wedge \_iren_C & \rightarrow & \_RC^f_o\downarrow \\
iren_C \wedge (IRC^v_o \vee IRC^f_o) & \rightarrow & IRC^e_i\downarrow \\
\neg IRC^e_i \wedge ((\neg ren_C \wedge \neg IRC^e_o) \vee \neg ircof\_) & \rightarrow & \_iren_C\uparrow \\
\_iren_C & \rightarrow & iren_C\downarrow \\
\neg ren_C \wedge \neg iren_C & \rightarrow & \_IRC^f_o\uparrow \\
\_IRC^f_o & \rightarrow & IRC^f_o\downarrow \\
\neg IRC^f_o & \rightarrow & ircof\_\uparrow \\
\neg iren_C \wedge \neg IRC^v_i & \rightarrow & IRC^e_i\uparrow \\
IRC^e_i \wedge IRC^e_o \wedge ren_C \wedge IRC^v_i & \rightarrow & \_iren_C\downarrow \\
\neg\_iren_C & \rightarrow & iren_C\uparrow
\end{array}$$

Program H.15 PRS: delay-insensitive interface cell between inner and outer banks of nested, WAD write control propagation array, with conditional outer write-enable, single port

$$\begin{array}{lcl}
\neg\_pReset & \rightarrow & \_iwen\uparrow \\
\neg\_pReset & \rightarrow & IWC^e_i\uparrow \\
wen \wedge IWC^v_i \wedge IWC^e_i \wedge IWC^e_o & \rightarrow & \_iwen\downarrow \\
\neg\_iwen & \rightarrow & iwen\uparrow \\
idW^1 \vee (iwen \wedge IWC^v_o) & \rightarrow & IWC^e_i\downarrow \\
\neg wen \wedge \neg IWC^e_i \wedge \neg IWC^e_o & \rightarrow & \_iwen\uparrow \\
\_iwen & \rightarrow & iwen\downarrow \\
\neg iwen \wedge \neg IWC^v_i \wedge \neg idW^1 & \rightarrow & IWC^e_i\uparrow
\end{array}$$

Program H.16 PRS: delay-insensitive interface cell between inner and outer banks of nested, WAD write control propagation array, with unconditional outer write-enable, single port

$$\begin{array}{lcl}
\neg\_pReset & \rightarrow & \_iwen\uparrow \\
\neg\_pReset & \rightarrow & IWC^e_i\uparrow \\
wen \wedge IWC^e_i \wedge IWC^e_o \wedge dW.0 \wedge IWC^v_i & \rightarrow & \_iwen\downarrow \\
\neg\_iwen & \rightarrow & iwen\uparrow \\
idW^1 \vee (iwen \wedge IWC^v_o) & \rightarrow & IWC^e_i\downarrow \\
\neg wen \wedge \neg IWC^e_i \wedge \neg IWC^e_o & \rightarrow & \_iwen\uparrow \\
\_iwen & \rightarrow & iwen\downarrow \\
\neg iwen \wedge \neg IWC^v_i \wedge \neg idW.1 & \rightarrow & IWC^e_i\uparrow
\end{array}$$

Program H.17 PRS: delay-insensitive interface cell between inner and outer banks of nested, WAD write control propagation array, with unconditional outer write-enable, single port

$$\begin{array}{lcl}
\neg \_pReset & \rightarrow & \_iwen\uparrow \\
\neg \_pReset & \rightarrow & IWC_i^e\uparrow \\
wen \wedge IWC_i^e \wedge IWC_o^e \wedge idW.0 & \rightarrow & \_iwen\downarrow \\
\neg \_iwen & \rightarrow & iwen\uparrow \\
idW^1 \vee (iwen \wedge IWC_o^v) & \rightarrow & IWC_i^e\downarrow \\
\neg wen \wedge \neg IWC_i^e \wedge \neg IWC_o^e & \rightarrow & \_iwen\uparrow \\
\_iwen & \rightarrow & iwen\downarrow \\
\neg iwen \wedge \neg IWC_i^v \wedge \neg idW.1 \wedge \neg idW.0 & \rightarrow & IWC_i^e\uparrow
\end{array}$$

H.5 Data Interface Array

The non-nested data interface circuits are illustrated in Figure 4.14, and the nested version is illustrated in Figure 9.17.

Program H.18 PRS: read/write data interface cell for a single port of a bit line

$$
\begin{array}{lcl}
\neg\_pReset & \rightarrow & \_R^0\uparrow \\
\neg\_pReset & \rightarrow & \_R^1\uparrow \\
\_R^0 & \rightarrow & R^0\downarrow \\
\_R^1 & \rightarrow & R^1\downarrow \\
R^0 \vee R^1 & \rightarrow & \_rv\downarrow \\
\neg RC_i^e \wedge \neg R^e & \rightarrow & \_ren_D\uparrow \\
\_ren_D & \rightarrow & ren_D\downarrow \\
\neg ren_D & \rightarrow & \_R^0\uparrow \\
\neg ren_D & \rightarrow & \_R^1\uparrow \\
\neg\_R^0 & \rightarrow & R^0\uparrow \\
\neg\_R^1 & \rightarrow & R^1\uparrow \\
\neg R^0 \wedge \neg R^1 & \rightarrow & \_rv\uparrow \\
RC_i^e \wedge R^e & \rightarrow & \_ren_D\downarrow \\
\neg\_ren_D & \rightarrow & ren_D\uparrow \\
\neg W^0 \wedge \neg W^1 & \rightarrow & \_wv\uparrow
\end{array}
$$

Program H.19 PRS: read/write data interface cell for a single port of a nested bit line

```
$\neg \_pReset$                              $\rightarrow \_R^0\uparrow$
$\neg \_pReset$                              $\rightarrow \_R^1\uparrow$
$\_R^0$                                      $\rightarrow R^0\downarrow$
$\_R^1$                                      $\rightarrow R^1\downarrow$
$R^0 \vee R^1$                               $\rightarrow \_rv\downarrow$
$\neg RC_i^e \wedge \neg R^e$                $\rightarrow \_ren_D\uparrow$
$\_ren_D$                                    $\rightarrow ren_D\downarrow$
$\neg ren_D$                                 $\rightarrow \_R^0\uparrow$
$\neg ren_D$                                 $\rightarrow \_R^1\uparrow$
$\neg \_R^0$                                 $\rightarrow R^0\uparrow$
$\neg \_R^1$                                 $\rightarrow R^1\uparrow$
$\neg R^0 \wedge \neg R^1$                   $\rightarrow \_rv\uparrow$
$RC_i^e \wedge R^e \wedge \_IR^v$            $\rightarrow \_ren_D\downarrow$
$\neg \_ren_D$                               $\rightarrow ren_D\uparrow$
$\neg W^0 \wedge \neg W^1 \wedge \neg IW^v$  $\rightarrow \_wv\uparrow$
```

H.6 Read Handshake Control

H.6.1 Unconditional Read Handshake Control

The half-buffer unconditional read handshake control is illustrated in Figure 4.16, and the full-buffer version is illustrated in Figure 4.15.

Note: all these handshake control production rules may are reused for the nested, non-WAD variations without modification!

Program H.20 PRS: read handshake control for unconditional control propagation, PCEVFB reshuffling

$$\begin{array}{lcl}
\neg\_pReset & \rightarrow & RC_i^e\uparrow \\
ren^v \wedge RC_i^v \wedge RC_o^v \wedge R^v & \rightarrow & RC_i^e\downarrow \\
\neg RC_i^e \wedge \neg RC_o^e & \rightarrow & \_ren_C\uparrow \\
\_ren_C & \rightarrow & ren_C\downarrow \\
\neg ren^v \wedge \neg RC_i^v & \rightarrow & RC_i^e\uparrow \\
RC_i^e \wedge RC_o^e & \rightarrow & \_ren_C\downarrow \\
\neg\_ren_C & \rightarrow & ren_C\uparrow
\end{array}$$

Program H.21 PRS: read handshake control for unconditional control propagation, PCEVHB reshuffling

$$\begin{array}{lcl}
\neg\_pReset & \rightarrow & RC_i^e\uparrow \\
ren^v \wedge RC_i^v \wedge RC_o^v \wedge R^v & \rightarrow & RC_i^e\downarrow \\
\neg RC_i^e \wedge \neg RC_o^e & \rightarrow & \_ren_C\uparrow \\
\_ren_C & \rightarrow & ren_C\downarrow \\
\neg ren^v \wedge \neg RC_i^v \wedge \neg RC_o^v & \rightarrow & RC_i^e\uparrow \\
RC_i^e \wedge RC_o^e & \rightarrow & \_ren_C\downarrow \\
\neg\_ren_C & \rightarrow & ren_C\uparrow
\end{array}$$

H.6.2 WAD Read Handshake Control

The half-buffer WAD read handshake control is illustrated in Figure 5.13, and the full-buffer version is illustrated in Figure 5.12.

Program H.22 PRS: read handshake control for WAD conditional control propagation, PCEVFB reshuffling

$$
\begin{array}{lcl}
\neg \mathit{\_pReset} & \rightarrow & RC_i^e\uparrow \\
\neg \mathit{\_pReset} & \rightarrow & \mathit{\_RC}_o^f\uparrow \\
\neg \mathit{\_RC}_o^f & \rightarrow & RC_o^f\uparrow \\
ren^v \wedge RC_i^v \wedge R^v \wedge (RC_o^v \vee RC_o^f) & \rightarrow & RC_i^e\downarrow \\
\neg RC_i^e \wedge (\neg RC_o^e \vee \neg \mathit{\_RC}_o^f) & \rightarrow & \mathit{\_ren}_C\uparrow \\
\mathit{\_ren}_C & \rightarrow & ren_C\downarrow \\
\neg ren_C & \rightarrow & \mathit{\_RC}_o^f\uparrow \\
\mathit{\_RC}_o^f & \rightarrow & RC_o^f\downarrow \\
\neg ren^v \wedge \neg RC_i^v \wedge \neg RC_o^f & \rightarrow & RC_i^e\uparrow \\
RC_i^e \wedge RC_o^e & \rightarrow & \mathit{\_ren}_C\downarrow \\
\neg \mathit{\_ren}_C & \rightarrow & ren_C\uparrow
\end{array}
$$

Program H.23 PRS: read handshake control for WAD conditional control propagation, PCEVHB reshuffling

$$
\begin{array}{lcl}
\neg \mathit{\_pReset} & \rightarrow & RC_i^e\uparrow \\
\neg \mathit{\_pReset} & \rightarrow & \mathit{\_RC}_o^f\uparrow \\
\neg \mathit{\_RC}_o^f & \rightarrow & RC_o^f\uparrow \\
ren^v \wedge RC_i^v \wedge R^v \wedge (RC_o^v \vee RC_o^f) & \rightarrow & RC_i^e\downarrow \\
\neg RC_i^e \wedge (\neg RC_o^e \vee \neg \mathit{\_RC}_o^f) & \rightarrow & \mathit{\_ren}_C\uparrow \\
\mathit{\_ren}_C & \rightarrow & ren_C\downarrow \\
\neg ren_C & \rightarrow & \mathit{\_RC}_o^f\uparrow \\
\mathit{\_RC}_o^f & \rightarrow & RC_o^f\downarrow \\
\neg ren^v \wedge \neg RC_i^v \wedge \neg RC_o^f \wedge \neg RC_o^v & \rightarrow & RC_i^e\uparrow \\
RC_i^e \wedge RC_o^e & \rightarrow & \mathit{\_ren}_C\downarrow \\
\neg \mathit{\_ren}_C & \rightarrow & ren_C\uparrow
\end{array}
$$

H.6.3 Nested WAD Read Handshake Control

The half-buffer WAD nested read handshake control is illustrated in Figure 9.19, and the full-buffer version is illustrated in Figure 9.18.

Program H.24 PRS: read handshake control for nested, WAD conditional control propagation, PCEVFB reshuffling

$$
\begin{array}{lcl}
\neg\_pReset & \rightarrow & RC_i^e\uparrow \\
\neg\_pReset & \rightarrow & \_RC_o^f\uparrow \\
\neg\_RC_o^f & \rightarrow & RC_o^f\uparrow \\
ren^v \wedge RC_i^v \wedge R^v \wedge (RC_o^v \vee RC_o^f) & \rightarrow & RC_i^e\downarrow \\
\neg RC_i^e \wedge (\neg RC_o^e \vee \neg\_RC_o^f) & \rightarrow & \_ren_C\uparrow \\
\_ren_C & \rightarrow & ren_C\downarrow \\
\neg ren_C & \rightarrow & \_RC_o^f\uparrow \\
\_RC_o^f & \rightarrow & RC_o^f\downarrow \\
\neg ren^v \wedge \neg RC_i^v \wedge \neg RC_o^f & \rightarrow & RC_i^e\uparrow \\
RC_i^e \wedge RC_o^e \wedge ircof\_ & \rightarrow & \_ren_C\downarrow \\
\neg\_ren_C & \rightarrow & ren_C\uparrow
\end{array}
$$

Program H.25 PRS: read handshake control for nested, WAD conditional control propagation, PCEVHB reshuffling

$$
\begin{array}{lcl}
\neg\_pReset & \rightarrow & RC_i^e\uparrow \\
\neg\_pReset & \rightarrow & \_RC_o^f\uparrow \\
\neg\_RC_o^f & \rightarrow & RC_o^f\uparrow \\
ren^v \wedge RC_i^v \wedge R^v \wedge (RC_o^v \vee RC_o^f) & \rightarrow & RC_i^e\downarrow \\
\neg RC_i^e \wedge (\neg RC_o^e \vee \neg\_RC_o^f) & \rightarrow & \_ren_C\uparrow \\
\_ren_C & \rightarrow & ren_C\downarrow \\
\neg ren_C & \rightarrow & \_RC_o^f\uparrow \\
\_RC_o^f & \rightarrow & RC_o^f\downarrow \\
\neg ren^v \wedge \neg RC_i^v \wedge \neg RC_o^f \wedge \neg RC_o^v & \rightarrow & RC_i^e\uparrow \\
RC_i^e \wedge RC_o^e \wedge ircof\_ & \rightarrow & \_ren_C\downarrow \\
\neg\_ren_C & \rightarrow & ren_C\uparrow
\end{array}
$$

H.6.4 Read Handshake Control Termination

The read handshake control for the terminal block is independent of control-buffering since there is no control output. The same production rules work for the non-WAD and WAD, non-nested and nested variations.

Program H.26 PRS: read handshake control for the terminal block.

$$\begin{array}{lcl} \neg\_pReset & \rightarrow & RC_i^e\uparrow \\ ren^v \wedge RC_i^v \wedge Rv & \rightarrow & RC_i^e\downarrow \\ \neg ren^v \wedge \neg RC_i^v & \rightarrow & RC_i^e\uparrow \end{array}$$

H.7 Write Handshake Control

H.7.1 Unconditional Write Handshake Control

The half-buffer unconditional write handshake control is illustrated in Figure 4.18, and the full-buffer version is illustrated in Figure 4.17.

Note: all the following PRS can be used for the nested versions without modification!

Program H.27 PRS: write handshake control for unconditional control propagation, PCEVFB reshuffling

$$\begin{array}{lcl}
\neg\_pReset & \rightarrow & WC_i^e\uparrow \\
wen \wedge wvc \wedge WC_i^v \wedge WC_o^v & \rightarrow & WC_i^e\downarrow \\
\neg WC_i^e \wedge \neg WC_o^e & \rightarrow & \_wen\uparrow \\
\_wen & \rightarrow & wen\downarrow \\
\neg wen \wedge \neg wvc \wedge \neg WC_i^v & \rightarrow & WC_i^e\uparrow \\
WC_o^e \wedge WC_i^e & \rightarrow & \_wen\downarrow \\
\neg\_wen & \rightarrow & wen\uparrow
\end{array}$$

Program H.28 PRS: write handshake control for unconditional control propagation, PCEVHB reshuffling

$$\begin{array}{lcl}
wvc \wedge WC_i^v \wedge WC_o^v & \rightarrow & WC_i^e\downarrow \\
\neg WC_i^e \wedge \neg WC_o^e & \rightarrow & \_wen\uparrow \\
\_wen & \rightarrow & wen\downarrow \\
\neg wvc \wedge \neg WC_i^v \wedge \neg WC_o^v & \rightarrow & WC_i^e\uparrow \\
WC_o^e \wedge WC_i^e & \rightarrow & \_wen\downarrow \\
\neg\_wen & \rightarrow & wen\uparrow
\end{array}$$

H.7.2 WAD Write Handshake Control, Unconditional Enable

The half-buffer WAD write handshake control with unconditional write-enable is illustrated in Figure 5.15, and the full-buffer version is illustrated in Figure 5.14.

Note: all the following PRS can be used for the nested versions without modification!

Program H.29 PRS: write handshake control for WAD control propagation, with unconditional write-enable *wen*, PCEVFB reshuffling

$$
\begin{array}{lcl}
\neg\_pReset & \rightarrow & WC_i^e\uparrow \\
\neg\_pReset & \rightarrow & \_WC_o^f\uparrow \\
wen \wedge dW^1 & \rightarrow & \_WC_o^f\downarrow \\
\neg\_WC_o^f & \rightarrow & WC_o^f\uparrow \\
wen \wedge WC_i^v \wedge wvc \wedge (WC_o^v \vee WC_o^f) & \rightarrow & WC_i^e\downarrow \\
\neg WC_i^e \wedge (\neg WC_o^e \vee \neg\_WC_o^f) & \rightarrow & \_wen\uparrow \\
\_wen & \rightarrow & wen\downarrow \\
\neg wen & \rightarrow & \_WC_o^f\uparrow \\
\_WC_o^f & \rightarrow & WC_o^f\downarrow \\
\neg wen \wedge \neg WC_i^v \wedge \neg wvc \wedge \neg WC_o^f & \rightarrow & WC_i^e\uparrow \\
WC_o^e \wedge WC_i^e & \rightarrow & \_wen\downarrow \\
\neg\_wen & \rightarrow & wen\uparrow
\end{array}
$$

Program H.30 PRS: write handshake control for WAD control propagation, with unconditional write-enable *wen*, PCEVHB reshuffling

$$
\begin{array}{lcl}
\neg\_pReset & \rightarrow & \_WC_o^f\uparrow \\
wen \wedge dW^1 & \rightarrow & \_WC_o^f\downarrow \\
\neg\_WC_o^f & \rightarrow & WC_o^f\uparrow \\
WC_i^v \wedge wvc \wedge (WC_o^v \vee WC_o^f) & \rightarrow & WC_i^e\downarrow \\
\neg WC_i^e \wedge (\neg WC_o^e \vee \neg\_WC_o^f) & \rightarrow & \_wen\uparrow \\
\_wen & \rightarrow & wen\downarrow \\
\neg wen & \rightarrow & \_WC_o^f\uparrow \\
\_WC_o^f & \rightarrow & WC_o^f\downarrow \\
\neg WC_i^v \wedge \neg wvc \wedge \neg WC_o^v \wedge \neg WC_o^f & \rightarrow & WC_i^e\uparrow \\
WC_o^e \wedge WC_i^e & \rightarrow & \_wen\downarrow \\
\neg\_wen & \rightarrow & wen\uparrow
\end{array}
$$

H.7.3 WAD Write Handshake Control, Conditional Enable

The half-buffer WAD write handshake control with conditional write-enable is illustrated in Figure 5.17, and the full-buffer version is illustrated in Figure 5.16.

Note: all the following PRS can be used for the nested versions without modification!

Program H.31 PRS: write handshake control for WAD control propagation, with conditional write-enable *wen*, PCEVFB reshuffling

$$\begin{array}{lcl}
\neg \_pReset & \rightarrow & \_wen\uparrow \\
WC_o^e \wedge WC_i^e \wedge dW^0 & \rightarrow & \_wen\downarrow \\
\neg \_wen & \rightarrow & wen\uparrow \\
WC_i^v \wedge wvc \wedge (wen \wedge WC_o^v \vee dW^1) & \rightarrow & WC_i^e\downarrow \\
\neg WC_i^e \wedge \neg WC_o^e & \rightarrow & \_wen\uparrow \\
\_wen & \rightarrow & wen\downarrow \\
\neg wen \wedge \neg WC_i^v \wedge \neg wvc & \rightarrow & WC_i^e\uparrow
\end{array}$$

Program H.32 PRS: write handshake control for WAD control propagation, with conditional write-enable *wen*, PCEVHB reshuffling

$$\begin{array}{lcl}
\neg \_pReset & \rightarrow & \_wen\uparrow \\
WC_o^e \wedge WC_i^e \wedge dW^0 & \rightarrow & \_wen\downarrow \\
\neg \_wen & \rightarrow & wen\uparrow \\
WC_i^v & \rightarrow & wciv\_\downarrow \\
\neg \_wvc \wedge \neg wciv\_ & \rightarrow & wvciv\uparrow \\
wvciv \wedge (WC_o^v \vee dW^1) & \rightarrow & WC_i^e\downarrow \\
\neg WC_i^e \wedge \neg WC_o^e & \rightarrow & \_wen\uparrow \\
\_wen & \rightarrow & wen\downarrow \\
\neg WC_i^v & \rightarrow & wciv\_\uparrow \\
\_wvc \wedge wciv\_ & \rightarrow & wvciv\downarrow \\
\neg wvciv \wedge \neg WC_o^v & \rightarrow & WC_i^e\uparrow
\end{array}$$

H.7.4 Write Handshake Control Termination

The write handshake control for the terminal block is independent of buffering since there is no output. The same production rules work for both unconditional and WAD, non-nested and nested variations.

Program H.33 PRS: write handshake control for control termination

$$\begin{array}{lcl} wvc \wedge WC_i^v & \rightarrow & WC_i^e\downarrow \\ \neg wvc \wedge \neg WC_i^v & \rightarrow & WC_i^e\uparrow \end{array}$$

Appendix I

Mine Eyes Have Seen The Glory

This appendix is intentionally left blank. Nobody likes references to Appendix “I” anyways.

Appendix J

Big Honkin' Tables

Table J.1 summarizes the symbols used in the tables throughout this appendix. For double-row table entries without a separation line, the upper row contains numbers for the faster of the non-uniform accesses and the lower row contains numbers for the slower.

For non-uniform access comparisons and breakeven analysis in Tables J.11 and J.22, widths 32a and 32n use width 32 as the baseline for comparison, and widths 16a and 16n use width 16 as the baseline for comparison. Breakeven probabilities $\hat{r}$ are computed as described in Section 8.1.

For a read operation, 'latency' is defined and measured as the delay from the time when $ren_D \wedge RC_i$ becomes true to the time $R_o\uparrow$ rises half-way between the supply rails, which includes the falling transition time for the read bit line, $\_R_o$. For a write operation, 'latency' is the delay from the write input condition to the write-validity condition (per bit line), which is measured as the delay from the time $W_i \wedge WC_i$ is true to the time $\_wv\downarrow$ falls half-way between the supply rails.

Table J.1: Data table symbols

type	symbol	definition
o	R	read port operation
(oper.)	W	write port operation
	S	standard or non-width-adaptive (read and write)
f	W	width-adaptive (read)
(format)	Wu	width-adaptive, unconditional write-enable (write)
	Wc	width-adaptive, conditional write-enable (write)
	32	single bank of 32 registers, balanced completion tree
	32a	single bank of 32 registers, unbalanced completion tree
w	32n	16-reg. bank nested inside 16-reg. bank, unbalanced tree
(width)	16	single bank of 16 registers, balanced completion tree
	16a	single bank of 16 registers, unbalanced completion tree
	16n	8-reg. bank nested inside 8-reg. bank, unbalanced tree
b	H	precharge enable-valid half-buffer (PCEVHB) reshuffling
(buf)	F	precharge enable-valid full-buffer (PCEVFB) reshuffling
τ_H, E_H		cycle time and energy of half-buffer variation
τ_F, E_F		cycle time and energy of full-buffer variation
τ_S, E_S		cycle time and energy of standard (non-WAD) variation
τ_W, E_W		cycle time and energy of WAD variation
l_f, τ_f, E_f		latency, cycle time, and energy of faster partition
l_s, τ_s, E_s		latency, cycle time, and energy of slower partition
l_0, τ_0, E_0		latency, cycle time, and energy of baseline (unpartitioned)
$\hat{r}_l$, $\hat{r}_\tau$, $\hat{r}_E$		break-even probability distribution for non-uniform access registers

Table J.2: All non-WAD read port performance and energy results

o	f	w	b	tr./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	en./cy. (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
R	S	32	H	22	1.953	512.2	0.323	26.90	102.5
			F	20	1.862	537.0	0.323	26.59	92.2
		32a	H	22	1.955	511.5	0.323	27.06	103.4
				30	2.315	431.9	0.323	29.77	159.6
			F	20	1.862	537.0	0.323	26.74	92.7
				28	2.079	480.9	0.323	28.50	123.2
		32n	H	22	2.128	470.0	0.216	20.86	94.4
				46	4.247	235.4	1.308	37.51	676.6
			F	20	1.880	531.9	0.216	19.84	70.1
				38	3.922	255.0	1.308	35.90	552.3
		16	H	22	1.821	549.1	0.222	15.92	52.8
			F	20	1.698	588.8	0.222	15.78	45.5
		16a	H	18	1.809	552.7	0.222	15.60	51.1
				22	1.949	513.2	0.222	16.31	61.9
			F	16	1.689	592.0	0.222	15.43	44.0
				20	1.771	564.6	0.222	15.90	49.9
		16n	H	18	1.759	568.5	0.163	14.47	44.8
				38	3.714	269.2	1.149	24.98	344.6
			F	16	1.630	613.5	0.163	14.09	37.4
				32	3.103	322.3	1.149	23.25	223.9

Table J.3: All WAD read port performance and energy results

o	f	w	b	tr./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	en./cy. (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
R	W	32	H	22	2.149	465.4	0.323	34.10	157.5
			F	20	2.014	496.4	0.323	33.18	134.6
		32a	H	22	2.151	464.9	0.323	34.16	158.1
				30	2.557	391.1	0.323	37.67	246.3
			F	20	2.014	496.5	0.323	33.26	134.9
				28	2.321	430.9	0.323	36.06	194.2
		32n	H	22	2.335	428.3	0.216	26.17	142.7
				46	4.659	214.6	1.308	46.89	1017.9
			F	20	2.037	490.9	0.216	24.87	103.2
				38	4.114	243.1	1.308	44.40	751.6
		16	H	22	2.025	493.8	0.222	19.88	81.6
			F	20	1.872	534.3	0.222	19.61	68.7
		16a	H	18	1.981	504.8	0.222	19.29	75.7
				22	2.179	458.8	0.222	20.25	96.2
			F	16	1.861	537.3	0.222	19.26	66.7
				20	1.942	514.9	0.222	19.83	74.8
		16n	H	18	1.964	509.3	0.163	18.04	69.5
				38	4.081	245.1	1.149	31.22	519.9
			F	16	1.802	554.8	0.163	17.66	57.4
				32	3.498	285.9	1.149	29.52	361.2

Table J.4: Impact of chosen buffering on read port performance and energy

o	f	w	$\tau_H/\tau_F - 1$	$1 - E_F/E_H$	$\frac{E_H\tau_H^2}{E_F\tau_F^2} - 1$
R	S	32	4.9%	1.1%	11.2%
		32a	5.0%	1.2%	11.5%
			11.3%	4.3%	29.5%
		32n	13.2%	4.9%	34.6%
			8.3%	4.3%	22.5%
		16	7.2%	0.9%	16.0%
		16a	7.1%	1.1%	16.0%
			10.0%	2.5%	24.1%
		16n	7.9%	2.6%	19.5%
			19.7%	6.9%	53.9%
R	W	32	6.7%	2.7%	17.0%
		32a	6.8%	2.7%	17.2%
			10.2%	4.3%	26.8%
		32n	14.6%	5.0%	38.3%
			13.2%	5.3%	35.4%
		16	8.2%	1.4%	18.7%
		16a	6.4%	0.2%	13.5%
			12.2%	2.0%	28.5%
		16n	8.9%	2.1%	21.3%
			16.7%	5.4%	43.9%

Table J.5: Impact of width-adaptivity on half-buffer read port performance and energy

o	f	w	b	$\tau_W/\tau_S - 1$	$E_W/E_S - 1$	$\frac{E_W \tau_W^2}{E_S \tau_S^2} - 1$
R	W	32	H	9.1%	26.8%	53.6%
		32a		9.1%	26.2%	52.8%
				9.5%	26.5%	54.3%
		32n		8.9%	25.5%	51.1%
				8.8%	25.0%	50.4%
		16		10.1%	24.9%	54.5%
		16a		8.7%	23.7%	48.2%
				10.6%	24.1%	55.3%
		16n		10.4%	24.7%	55.4%
				9.0%	25.0%	50.9%

Table J.6: Impact of width-adaptivity on full-buffer read port performance and energy

o	f	w	b	$\tau_W/\tau_S - 1$	$E_W/E_S - 1$	$\frac{E_W \tau_W^2}{E_S \tau_S^2} - 1$
R	W	32	F	7.6%	24.8%	46.0%
		32a		7.5%	24.4%	45.5%
				10.4%	26.6%	57.7%
		32n		7.7%	25.3%	47.1%
				4.7%	23.7%	36.1%
		16		9.3%	24.3%	51.0%
		16a		9.2%	24.8%	51.5%
				8.8%	24.7%	50.0%
		16n		9.6%	25.3%	53.2%
				11.3%	27.0%	61.4%

Table J.7: Impact of bank size on read port performance and energy

o	f	w	b	$\tau_{32}/\tau_{16} - 1$	$1 - E_{16}/E_{32}$	$\frac{E_{32}\tau_{32}^2}{E_{16}\tau_{16}^2} - 1$
R	S	16	H	7.2%	40.8%	94.2%
			F	9.6%	40.7%	102.7%
		16a	H	8.1%	42.3%	102.5%
				18.8%	45.2%	157.7%
			F	10.2%	42.3%	110.6%
				17.4%	44.2%	147.0%
		16n	H	21.0%	30.6%	110.9%
				14.4%	33.4%	96.4%
			F	15.3%	29.0%	87.3%
				26.4%	35.2%	146.7%
R	W	16	H	6.1%	41.7%	93.1%
			F	7.6%	40.9%	95.9%
		16a	H	8.6%	43.5%	108.8%
				17.3%	46.3%	156.1%
			F	8.2%	42.1%	102.2%
				19.5%	45.0%	159.6%
		16n	H	18.9%	31.1%	105.1%
				14.2%	33.4%	95.8%
			F	13.0%	29.0%	79.9%
				17.6%	33.5%	108.1%

Table J.8: Impact of bank size on read latency

o	w	l_{16}/l_{32}
R	16,16a	0.686
	16n	0.753
	16n	0.878

Table J.9: Impact of nesting on read latency

o	w	l_f/l_0	l_s/l_0	$\hat{r}_l$
R	32n	0.668	4.043	90.2%
	16n	0.733	5.180	94.0%

Table J.10: Impact of extending a bank with nesting on read port performance and energy

o	f	b	$\frac{\tau_{32n}}{\tau_{16}} - 1$	$\frac{l_{32n}}{l_{16}} - 1$	$\frac{E_{32n}}{E_{16}} - 1$
R	S	H	16.8%	-2.6%	31.0%
		F	10.7%		25.8%
	W	H	15.3%		31.6%
		F	8.8%		26.8%

Table J.11: Impact of non-uniform accesses on read port performance and energy

o	f	w	b	τ_f/τ_0	τ_s/τ_0	$\hat{r}_\tau$	E_f/E_0	E_s/E_0	$\hat{r}_E$
R	S	32a	H	1.001	1.186	100.0%	1.006	1.107	100.0%
			F	1.000	1.117	100.0%	1.006	1.072	100.0%
		32n	H	1.090	2.175	100.0%	0.775	1.394	63.7%
			F	1.009	2.106	100.0%	0.746	1.350	58.0%
		16a	H	0.994	1.070	91.6%	0.980	1.024	55.0%
			F	0.995	1.043	88.9%	0.978	1.008	26.6%
		16n	H	0.966	2.040	96.8%	0.909	1.569	86.2%
			F	0.960	1.827	95.3%	0.893	1.474	81.6%
R	W	32a	H	1.001	1.190	100.0%	1.002	1.105	100.0%
			F	1.000	1.152	100.0%	1.002	1.087	100.0%
		32n	H	1.087	2.168	100.0%	0.767	1.375	61.7%
			F	1.011	2.042	100.0%	0.749	1.338	57.5%
		16a	H	0.978	1.076	77.7%	0.970	1.018	38.0%
			F	0.994	1.038	87.1%	0.982	1.011	38.2%
		16n	H	0.970	2.015	97.1%	0.907	1.570	86.0%
			F	0.963	1.869	95.9%	0.900	1.505	83.5%

Table J.12: All non-WAD write port performance and energy results

o	f	w	b	tr./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	en./cy. (pJ)	$E\tau^2$ $(10^{-30}Js^2)$
W	S	32	H	22	2.488	402.0	0.528	27.81	172.1
			F	20	2.444	409.2	0.528	27.45	163.9
		32a	H	22	2.488	402.0	0.528	27.95	173.0
				30	2.484	402.6	0.528	29.20	180.2
			F	20	2.444	409.2	0.528	27.95	166.9
				28	2.471	404.7	0.528	29.80	182.0
		32n	H	22	2.344	426.7	0.432	16.32	89.7
				46	3.960	252.5	1.095	29.86	468.3
			F	20	2.293	436.1	0.432	16.01	84.2
				38	3.647	274.2	1.095	28.60	380.3
		16	H	22	2.179	458.9	0.417	11.23	53.3
			F	20	2.118	472.1	0.417	11.30	50.7
		16a	H	20	2.175	459.8	0.417	11.08	52.4
				22	2.172	460.3	0.417	11.22	52.9
			F	20	2.116	472.5	0.417	10.78	48.3
				20	2.156	463.8	0.417	11.15	51.8
		16n	H	20	2.136	468.1	0.375	10.71	48.9
				36	3.583	279.1	0.963	19.48	250.0
			F	20	2.079	481.0	0.375	10.49	45.3
				30	2.964	337.4	0.963	17.68	155.3

Table J.13: All WAD-uwen write port performance and energy results

o	f	w	b	tr./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	en./cy. (pJ)	$E\tau^2$ ($10^{-30} Js^2$)
W	Wu	32	H	22	2.601	384.5	0.528	35.07	237.3
			F	20	2.604	384.0	0.528	34.90	236.7
		32a	H	22	2.602	384.3	0.528	34.64	234.5
				30	2.637	379.2	0.528	36.41	253.2
			F	20	2.604	384.0	0.528	34.80	236.0
				28	2.648	377.7	0.528	36.39	255.1
		32n	H	22	2.453	407.6	0.432	19.35	116.5
				46	4.117	242.9	1.095	36.04	610.8
			F	20	2.456	407.2	0.432	19.45	117.3
				38	3.801	263.1	1.095	34.11	492.7
		16	H	22	2.288	437.0	0.417	13.17	69.0
			F	20	2.281	438.5	0.417	13.46	70.0
		16a	H	20	2.283	438.0	0.417	13.12	68.4
				22	2.310	433.0	0.417	13.17	70.3
			F	20	2.278	438.9	0.417	13.36	69.3
				20	2.319	431.2	0.417	13.30	71.5
		16n	H	20	2.245	445.4	0.375	12.60	63.5
				36	3.740	267.4	0.963	22.82	319.2
			F	20	2.238	446.9	0.375	12.54	62.8
				30	3.211	311.5	0.963	21.18	218.3

Table J.14: All WAD-cwen write port performance and energy results

o	f	w	b	tr./ cycle	cycle (ns)	freq. (MHz)	latency (ns)	en./cy. (pJ)	$E\tau^2$ $(10^{-30} Js^2)$
W	Wc	32	H	24	2.556	391.3	0.528	34.40	224.7
			F	22	2.636	379.4	0.528	36.04	250.4
		32a	H	24	2.558	391.0	0.528	34.16	223.5
				32	2.655	376.6	0.528	36.20	255.2
			F	22	2.632	380.0	0.528	34.87	241.5
				30	2.712	368.7	0.528	36.52	268.6
		32n	H	24	2.403	416.2	0.432	19.06	110.0
				46	4.052	246.8	1.095	35.50	582.8
			F	22	2.486	402.2	0.432	19.43	120.1
				38	3.831	261.0	1.095	34.25	502.6
		16	H	24	2.243	445.9	0.417	13.03	65.5
			F	22	2.320	431.1	0.417	13.51	72.7
		16a	H	22	2.242	446.1	0.417	12.63	63.5
				24	2.299	434.9	0.417	13.07	69.1
			F	20	2.313	432.4	0.417	12.81	68.5
				22	2.383	419.6	0.417	13.21	75.0
		16n	H	22	2.197	455.2	0.375	12.41	59.9
				36	3.656	273.5	0.963	22.67	303.0
			F	20	2.268	440.9	0.375	12.60	64.8
				30	3.203	312.2	0.963	21.20	217.5

Table J.15: Impact of chosen buffering on write port performance and energy

o	f	w	$\tau_H/\tau_F - 1$	$1 - E_F/E_H$	$\frac{E_H\tau_H^2}{E_F\tau_F^2} - 1$
W	S	32	1.8%	1.3%	5.0%
		32a	1.8%	-0.0%	3.6%
			0.5%	-2.0%	-1.0%
		32n	2.2%	2.0%	6.5%
			8.6%	4.2%	23.1%
		16	2.9%	-0.6%	5.2%
		16a	2.8%	2.7%	8.6%
			0.8%	0.6%	2.1%
		16n	2.8%	2.0%	7.8%
			20.9%	9.2%	61.0%
W	Wu	32	-0.1%	0.5%	0.2%
		32a	-0.1%	-0.5%	-0.6%
			-0.4%	0.1%	-0.7%
		32n	-0.1%	-0.5%	-0.7%
			8.3%	5.4%	24.0%
		16	0.3%	-2.2%	-1.5%
		16a	0.2%	-1.8%	-1.4%
			-0.4%	-0.9%	-1.7%
		16n	0.3%	0.5%	1.2%
			16.5%	7.2%	46.2%
W	Wc	32	-3.0%	-4.8%	-10.3%
		32a	-2.8%	-2.1%	-7.4%
			-2.1%	-0.9%	-5.0%
		32n	-3.4%	-1.9%	-8.4%
			5.8%	3.5%	15.9%
		16	-3.3%	-3.7%	-9.9%
		16a	-3.1%	-1.4%	-7.4%
			-3.5%	-1.0%	-7.9%
		16n	-3.1%	-1.5%	-7.6%
			14.1%	6.4%	39.3%

Table J.16: Impact of width-adaptivity on half-buffer write port performance and energy

o	f	w	b	$\tau_W/\tau_S - 1$	$E_W/E_S - 1$	$\frac{E_W \tau_W^2}{E_S \tau_S^2} - 1$
W	Wu	32	H	4.3%	26.1%	37.9%
	Wc			2.7%	23.7%	30.5%
	Wu	32a		4.4%	24.0%	35.6%
				5.8%	24.7%	40.5%
	Wc			2.7%	22.2%	29.2%
				6.4%	24.0%	41.6%
	Wu	32n		4.5%	18.6%	29.9%
				3.8%	20.7%	30.4%
	Wc			2.5%	16.8%	22.7%
				2.3%	18.9%	24.5%
	Wu	16		4.8%	17.2%	29.3%
	Wc			2.8%	16.0%	22.9%
	Wu	16a		4.7%	18.4%	30.5%
				5.9%	17.5%	32.8%
	Wc			3.0%	14.0%	21.1%
				5.5%	16.5%	30.5%
	Wu	16n		4.9%	17.7%	30.0%
				4.2%	17.2%	27.7%
	Wc			2.8%	16.0%	22.6%
				2.0%	16.4%	21.2%

Table J.17: Impact of width-adaptivity on full-buffer write port performance and energy

o	f	w	b	$\tau_W/\tau_S - 1$	$E_W/E_S - 1$	$\frac{E_W\tau_W^2}{E_S\tau_S^2} - 1$
W	Wu	32	F	6.2%	27.1%	44.4%
	Wc			7.3%	31.3%	52.8%
	Wu	32a		6.2%	24.5%	41.4%
				6.7%	22.1%	40.2%
	Wc			7.1%	24.7%	44.7%
				8.9%	22.5%	47.6%
	Wu	32n		6.6%	21.5%	39.4%
				4.1%	19.2%	29.5%
	Wc			7.8%	21.4%	42.7%
				4.8%	19.7%	32.2%
	Wu	16		7.1%	19.1%	38.1%
	Wc			8.7%	19.6%	43.4%
	Wu	16a		7.1%	23.9%	43.6%
				7.0%	19.3%	37.9%
	Wc			8.5%	18.9%	42.0%
				9.5%	18.5%	44.8%
	Wu	16n		7.1%	19.5%	38.5%
				7.7%	19.8%	40.6%
	Wc			8.3%	20.2%	43.1%
				7.5%	20.0%	40.1%

Table J.18: Impact of bank size on write port performance and energy

o	f	w	b	$\tau_{32}/\tau_{16} - 1$	$1 - E_{16}/E_{32}$	$\frac{E_{32}\tau_{32}^2}{E_{16}\tau_{16}^2} - 1$
W	S	16	H	14.2%	59.6%	222.6%
			F	15.4%	58.8%	223.4%
		16a	H	14.4%	60.4%	230.0%
				14.3%	61.6%	240.4%
			F	15.5%	61.4%	245.8%
				14.6%	62.6%	251.1%
		16n	H	9.7%	34.4%	83.5%
				10.5%	34.8%	87.3%
			F	10.3%	34.5%	85.7%
				23.0%	38.2%	144.9%
W	Wu	16	H	13.7%	62.5%	244.1%
			F	14.2%	61.4%	238.3%
		16a	H	13.9%	62.1%	242.9%
				14.2%	63.8%	260.3%
			F	14.3%	61.6%	240.4%
				14.2%	63.5%	256.8%
		16n	H	9.3%	34.9%	83.3%
				10.1%	36.7%	91.4%
			F	9.7%	35.5%	86.9%
				18.4%	37.9%	125.7%
W	Wc	16	H	14.0%	62.1%	242.8%
			F	13.6%	62.5%	244.4%
		16a	H	14.1%	63.0%	252.1%
				15.5%	63.9%	269.3%
			F	13.8%	63.3%	252.3%
				13.8%	63.8%	258.0%
		16n	H	9.4%	34.9%	83.6%
				10.8%	36.2%	92.4%
			F	9.6%	35.1%	85.3%
				19.6%	38.1%	131.1%

Table J.19: Impact of bank size on write latency

o	w	l_{16}/l_{32}
W	16,16a	0.790
	16n	0.868
	16n	0.880

Table J.20: Impact of nesting on write latency

o	w	l_f/l_0	l_s/l_0	$\hat{r}_l$
W	32n	0.818	2.074	85.5%
	16n	0.899	2.310	92.8%

Table J.21: Impact of extending a bank with nesting on write port performance and energy

o	f	b	$\frac{\tau_{32n}}{\tau_{16}} - 1$	$\frac{l_{32n}}{l_{16}} - 1$	$\frac{E_{32n}}{E_{16}} - 1$
W	S	H	7.5%	3.6%	45.3%
		F	8.3%		41.7%
	Wu	H	7.2%		47.0%
		F	7.7%		44.6%
	Wc	H	7.1%		46.3%
		F	7.2%		43.8%

Table J.22: Impact of non-uniform accesses on write port performance and energy

o	f	w	b	τ_f/τ_0	τ_s/τ_0	$\hat{r}_\tau$	E_f/E_0	E_s/E_0	$\hat{r}_E$
W	S	32a	H	1.000	0.998	0.0%	1.005	1.050	100.0%
			F	1.000	1.011	99.2%	1.018	1.086	100.0%
		32n	H	0.942	1.592	91.1%	0.587	1.074	15.2%
			F	0.938	1.492	88.9%	0.583	1.042	9.2%
		16a	H	0.998	0.997	0.0%	0.986	0.998	0.0%
			F	0.999	1.018	95.7%	0.954	0.987	0.0%
		16n	H	0.980	1.644	97.0%	0.953	1.734	94.0%
			F	0.981	1.399	95.6%	0.928	1.565	88.7%
W	Wu	32a	H	1.000	1.014	100.0%	0.988	1.038	75.6%
			F	1.000	1.017	99.0%	0.997	1.043	93.7%
		32n	H	0.943	1.583	91.1%	0.552	1.027	5.8%
			F	0.943	1.459	88.9%	0.557	0.977	0.0%
		16a	H	0.998	1.009	81.1%	0.996	1.000	9.0%
			F	0.999	1.017	94.4%	0.993	0.988	0.0%
		16n	H	0.981	1.634	97.1%	0.957	1.733	94.5%
			F	0.981	1.408	95.6%	0.932	1.574	89.4%
W	Wc	32a	H	1.001	1.039	100.0%	0.993	1.052	88.2%
			F	0.998	1.029	94.6%	0.967	1.013	28.6%
		32n	H	0.940	1.585	90.7%	0.554	1.032	6.7%
			F	0.943	1.453	88.9%	0.539	0.950	0.0%
		16a	H	1.000	1.025	98.1%	0.970	1.003	9.2%
			F	0.997	1.027	90.3%	0.948	0.978	0.0%
		16n	H	0.980	1.630	96.9%	0.953	1.739	94.0%
			F	0.978	1.381	94.5%	0.933	1.569	89.4%

Appendix K

Top Ten Lists

The Top 10 Signs That You Should Stop Writing the Master's Thesis

10. You get so bored of writing, that you start adding Top 10 lists to the appendix.
9. You actually find the lists funny.
8. None of your peers has ever written a PhD dissertation as long.
7. You need to rent a cart to deliver copies of the thesis to your committee members.
6. You start hoarding and concealing reams of paper for "the big printing day."
5. Your desk creaks and sags from its sheer weight.
4. Your committee members give you feedback on only the abstract.
3. The thesis advisor mutters something about binding an encyclopedia in twenty volumes.
2. Your wife is pregnant and asks when you're going to graduate and get a real job.
1.
```
ERROR: TeX capacity exceeded.  Out of main_memory.
PDF output incomplete.
```

Bibliography

[1] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[2] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, January 1999.

[3] David Brooks and Margaret Martonosi. Value-based clock gating and operation packing: Dynamic strategies for improving processor power and performance. *ACM Transactions on Computer Systems*, 18(2):89–126, May 2000.

[4] Ramon Canal, Antonio González, and James E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 181–190, Monterrey, CA, December 2000.

[5] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned register files for VLIWs: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 292–300, December 1992.

[6] J. W. Chung, D. Kao, C. Cheng, and T. Lin. Optimization of power dissipation and skew sensitivity in clock buffer synthesis. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED '95)*, pages 179–184, Dana Point, CA, 1995.

[7] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, October 1998.

[8] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual*

International Symposium on Computer Architecture, pages 316–325, Vancouver, Canada, June 2000.

[9] Uri Cummings, Andrew Lines, and Alain Martin. An asynchronous pipelined lattice structure filter. In *Proceedings of the 1st Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, November 1994.

[10] Virantha N. Ekanayake. Asynchronous memories. Master's thesis, Cornell University, 2002.

[11] David Fang. Detailed decompositions of asynchronous register files. Technical Report CSL-TR-2003-1037, Cornell University, December 2003.

[12] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[13] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *Proceedings of the 1998 International Conference on Computer Design*, pages 247–252, Austin, TX, October 1998.

[14] S. B. Furber, J. D. Garside, S. Temple, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proceedings of the 3rd Annual International Symposium on Asynchronous Circuits and Systems*, pages 290–299, April 1997.

[15] Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th Design Automation Conference (DAC '98)*, San Francisco, CA, 1998.

[16] D. Harris. *Skew-Tolerant Circuit Design*. Morgan Kaufmann, 2001.

[17] C. Anthony R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[18] M. Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, 1991.

[19] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[20] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[21] M. Lewis and L. Brackenbury. Exploiting typical DSP access patterns for a low power multiported register bank. In *Proceedings of the 7th Annual International Symposium on Asynchronous Circuits and Systems*, Salt Lake City, UT, March 2001.

[22] Hai Li, Yiran Chen, T. N. Vijaykumar, and Kaushik Roy. Deterministic clock gating for microprocessor power reduction. In *Proceedings of the 9th IEEE Symposium on High-Performance Computer Architecture*, Anaheim, CA, February 2003.

[23] Andrew M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1995.

[24] Rajit Manohar. An analysis of reshuffled handshaking expansions. In *Proceedings of the 7th Annual International Symposium on Asynchronous Circuits and Systems*, Salt Lake City, Utah, March 2001.

[25] Rajit Manohar. Width-adaptive data word architectures. In *Proceedings of the 19th Conference on Advanced Research in VLSI*, Salt Lake City, Utah, March 2001.

[26] Rajit Manohar, Tak-Kwan Lee, , and Alain J. Martin. Projection: A synthesis technique for concurrent systems. In *Proceedings of the 5th Annual International Symposium on Asynchronous Circuits and Systems*, pages 125–134, Barcelona, Spain, April 1999.

[27] Rajit Manohar and Alain J. Martin. Pipelined mutual exclusion and the design of an asynchronous microprocessor. Technical Report CSL-TR-2001-1017, Cornell Computer Systems Lab, November 2001.

[28] Rajit Manohar, Mika Nyström, and Alain J. Martin. Precise exceptions in asynchronous processors. In *Proceedings of the 19th Conference on Advanced Research in VLSI*, Salt Lake City, Utah, March 2001.

[29] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4), 1986.

[30] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Proceedings of the 6th Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[31] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri V. Cummings, and Tak Kwan Lee. The design of an asynchronous MIPS R3000. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[32] Alain J. Martin, Mika Nyström, Paul Penzes, and Catherine Wong. Speed and energy performance of an asynchronous MIPS R3000 microprocessor. Technical Report CaltechCSTR:2001.012, Caltech Computer Science Department, September 2001.

[33] José Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture*, Istanbul, Turkey, November 2002.

[34] Mika Nyström. *Asynchronous Pulse Logic*. PhD thesis, California Institute of Technology, May 2001.

[35] Recep O. Ozdag and Peter A. Beerel. High-speed QDI asynchronous pipelines. In *Proceedings of the 7th Annual International Symposium on Asynchronous Circuits and Systems*, pages 13–22, Manchester, UK, April 2002.

[36] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity effective superscalar processors. In M. Hill, N. Jouppi, and G. Sohi, editors, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[37] D. B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, pages 8–15, April 1996.

[38] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[39] N. Paver, P. Day, S. B. Furber, J. D. Garside, and J.V. Woods. Register locking in an asynchronous microprocessor. In *Proceedings of the 1992 International Conference on Computer Design*, pages 351–355, Boston, MA, October 1992.

[40] A. Podlensky, G. Kristovsky, and A. Malshin. Multiport register file memory cell configuration for read operation. U.S. Patent 5,657,291, Sun Microsystems, Inc., Mountain View, CA, August 1997.

[41] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: a standard-cell QDI 16-bit RISC asynchronous processor. In *Proceedings of the 4th Annual International Symposium on Asynchronous Circuits and Systems*, San Diego, CA, March/April 1998.

[42] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture*, pages 375–386, January 2000.

[43] Richard M. Russell. The Cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[44] Richard M. Russell. The Cray-1 computer system. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, *Readings in Computer Architecture*, pages 40–49. Morgan Kaufmann, 2000.

[45] S. P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC microprocessor. *IEEE Micro*, pages 8–17, October 1994.

[46] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[47] John A. Swenson and Yale N. Patt. Hierarchical register for scientific computing. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 346–353, Saint Malo, France, 1988.

[48] Akihiro Takamura, Masashi Kuwako, Masashi Imai, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya. TITAC-2: A 32-bit asynchronous microprocessor based on scalable-delay-insensitive model. In *Proceedings of the 1997 International Conference on Computer Design*, pages 288–294, October 1997.

[49] John Teifel, David Fang, David Biermann, Clinton Kelly IV, and Rajit Manohar. Energy-efficient pipelines. In *Proceedings of the 8th Annual International Symposium on Asynchronous Circuits and Systems*, Manchester, UK, April 2002.

[50] Jessica Hui-Chun Tseng. Energy-efficient register file design. Master's thesis, MIT, 1999.

[51] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[52] Steven Wallace and Nader Begherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques '96*, pages 179–184, Boston, MA, October 1996.

[53] T. E. Williams. *Self-Timed Rings and their Application to Division.* PhD thesis, Stanford University, May 1991.

[54] Anthony J. Winstanley, Aurelien Garivier, and Mark R. Greenstreet. An event spacing experiment. In *Proceedings of the 8th Annual International Symposium on Asynchronous Circuits and Systems*, pages 47–56, Manchester, UK, April 2002.

[55] K. C. Yeager. MIPS R10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, April 1995.

[56] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Two-level hierarchical register file organization for VLIW processors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 137–146, Monterrey, CA, December 2000.

[57] V. Zyuban and P. Kogge. The energy complexity of register files. Technical Report 97-20, Notre Dame CSE, December 1997.

[58] V. Zyuban and P. Kogge. The energy complexity of register files. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED '98)*, pages 305–310, August 1998.

Index