The Impact of Asynchrony on Computer Architecture

Rajit Manohar

Computer Science Department
California Institute of Technology

Caltech-CS-TR-98-12

The Impact of Asynchrony on Computer Architecture

Thesis by

Rajit Manohar

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

California Institute of Technology
Pasadena, California

1998
(Submitted June 25, 1998)

Acknowledgments

I thank Alain J. Martin for being my advisor, for teaching me about concurrency and asynchronous circuit design, and for being a person with whom I could always talk. He took me on as a graduate student on extremely short notice, and encouraged me throughout my stay at Caltech.

I thank the members of my thesis committee: Yaser S. Abu-Mostafa, Al Barr, K. Mani Chandy, and Alain J. Martin.

Friday mornings were devoted to group meetings. I would like to thank Uri Cummings, Marcel van der Goot, Peter Hofstee, Tony Lee, Andrew Lines, Mika Nyström, Paul Penzes, Robert Southworth, José Tierno, and other members of the asynchronous VLSI group for all the insightful (and animated!) discussions, and what they taught me through them.

I often spent time thinking about topics that were not directly (and sometimes not even indirectly) related to asynchronous circuit design. I thank Alain J. Martin for giving me this freedom, and Yaser S. Abu-Mostafa, Donald Dabdub, Boris Dimitrov, Robert Harley, Rohit Khare, K. Rustan M. Leino, and Paolo A.G. Sivilotti for providing some of the distractions.

I thank Marcel van der Goot, Alain J. Martin, Eve Schooler, Paolo A.G. Sivilotti, and Robert Southworth for their careful reading of this thesis.

I thank Tzu-Yi Chen, Donald Dabdub, Marie Johnson, Robert Harley, Rohit Khare, Berna Massingill, Adam Rifkin, and Eve Schooler for conversation and friendship.

This research was supported by the Defence and Advanced Research Projects Agency under the office of Army Research and in part by a National Semiconductor Corporation graduate fellowship.

Abstract

The performance characteristics of asynchronous circuits are quite different from those of their synchronous counterparts. As a result, the best asynchronous design of a particular system does not necessarily correspond to the best synchronous design, even at the algorithmic level. The goal of this thesis is to examine certain aspects of computer architecture and design in the context of an asynchronous VLSI implementation.

We present necessary and sufficient conditions under which the degree of pipelining of a component can be modified without affecting the correctness of an asynchronous computation.

As an instance of the improvements possible using an asynchronous architecture, we present circuits to solve the prefix problem with average-case behavior better than that possible by any synchronous solution in the case when the prefix operator has a right zero. We show that our circuit implementations are area-optimal given their performance characteristics, and have the best possible average-case latency.

At the level of processor design, we present a mechanism for the implementation of precise exceptions in asynchronous processors. The novel feature of this mechanism is that it permits the presence of a data-dependent number of instructions in the execution pipeline of the processor.

Finally, at the level of processor architecture, we present the architecture of a processor with an independent instruction stream for branches. The instruction set permits loops and function calls to be executed with minimal control-flow overhead.

Contents

Chapter 1.

INTRODUCTION

"It is a little off the beaten track, isn't it?"
—Sir Arthur Conan Doyle, *The Red Headed League*

Asynchronous switching circuits have been used since the 1940's. The Illiac, designed by the University of Illinois Digital Computer Laboratory in the late 1950's,[18] is an example of a computer that contained both synchronous and asynchronous switching circuits. The computer had "end signals" (now called "acknowledge" signals) that indicated the completion of an action.

Early concepts in the design of fundamental mode circuits were contributed by D.A. Huffman in the 1950's.[18] The circuits to be designed were specified using flow tables from which the excitation functions for all variables in the circuit were generated. An extension of fundamental mode circuits, known as burst-mode circuits, are still used by the asynchronous design community. Both these design styles use feedback paths with carefully matched delays to store state information.

A theory of speed-independent asynchronous switching circuits was developed by D.E. Muller in the early 1960's as an attempt to abstract from the difficulties of designing circuits that depended heavily on their precise physical implementation. His model assumed that transistor networks may have arbitrary delay, and that the propagation delay through wires is negligible compared to the delay through the network.

As complex asynchronous circuits became difficult to design because of the problem of hazards in switching signals, they were replaced by synchronous circuits. By the time computers became widespread in the 1970's, synchronous switching circuits had emerged as the prevalent design style. Indeed in the proceedings of the

Caltech conference on VLSI in 1979, the chair of the session on self-timed logic, Charles E. Molnar, noted that:

"The appearance of this session on Self-Timed Logic in a Conference on Very Large Scale Integrated System Design may warrant some explanation."[21]

Modern asynchronous circuit design probably began when concerns arose regarding problems with the physical realization of large-scale synchronous systems. In 1979, Seitz proposed a design methodology for self-timed circuits wherein the circuit was to be decomposed into equipotential regions—regions where delays in wires could be considered negligible, with explicit modeling of signal propagation delay between such regions.[22]

The first method for the synthesis of asynchronous circuits whose correct functioning did not depend on the delays of gates and which permitted multiple concurrent switching signals was introduced by Martin.[15] The approach is inspired by the observation that a VLSI chip is a fine-grained concurrent computation. Computations are modeled using CHP (Communicating Hardware Processes) programs that describe their behavior algorithmically. (Appendix 1 contains a brief description of the notation; a detailed description of the semantics is provided by van der Goot.[6]) Asynchronous quasi delay-insensitive (QDI) circuits are synthesized from these programs using semantics-preserving transformations. We apply this approach to the design of asynchronous circuits in this thesis.

Asynchronous QDI circuits are robust to variations in temperature, voltage, and fabrication process parameters. For example, the Caltech microprocessor fabricated in $1.6\mu m$ CMOS technology (design voltage $5\,V$) is functional at all voltages from $0.6\,V$ (sub-threshold) to $12\,V$ (punch-through).[16] More recently, an asynchronous pipelined lattice-structure filter[2] was fabricated and is functional in $0.8\mu m$ CMOS technology (design voltage $3.3\,V$) from $1.1\,V$ to $4.9\,V$. Both chips are also functional at temperatures ranging from $77K$ to $350K$.

Asynchronous implementation strategies for complex VLSI systems are interesting for other reasons as well. Asynchronous circuits exhibit average-case behavior. As a result, we can choose implementations that improve the overall performance of the circuit even if they make the worst-case performance worse. For example, an N-bit ripple-carry asynchronous adder has an average case latency of $O(\log N)$, the same order as a more complex synchronous carry-lookahead adder.[1]

Asynchronous implementations of a system consume less power than synchronous implementations. If we assume that the physical implementation of a circuit dissipates power when a particular signal in the circuit changes, we can show that any computation must dissipate an amount of power that depends on the entropy of the specification of the circuit.[25] What may not be so obvious is that we can achieve this bound within a constant factor by using an asynchronous implementation.[25]

Today, the most important issue in the design of a large system is the management of complexity. Complex asynchronous circuits can be described by relatively concise CHP programs that completely specify their behavior.[15] Using a formal transformational approach, the final circuit is designed using correctness-preserving transformations from an initial compact *sequential* specification—one that is easily verified. As a result, the behavior of a circuit is understood in a modular and hierarchical fashion—by understanding sequential programs, program transformations, and program composition.[15]

1.1. Computer Architecture

Computer architecture is the specification and algorithmic design of the hardware in a computer system. Given the specification of a system, a designer is confronted with a number of possible implementation strategies. The choices made by a designer are guided by the relative merits of the different strategies. The figure of merit is typically the performance of the resulting implementation.

Existing studies into the design of computer architecture have been made with the assumption that the target hardware is synchronous, since almost all circuits designed today are synchronous. However, the rationale for the design choices made by synchronous designers need not apply when designing an asynchronous circuit to perform the same computation.

In synchronous design, the performance of the system is determined by the clock frequency. If any component of the system is slow, the entire system must be slowed down to ensure the system operates correctly. This affects system throughput if some part of it does not operate at the desired clock frequency. A well-designed asynchronous circuit with the same properties will operate at the speed of the slow component only when the slow component is used.

Since the clock is used to discretize the time domain, differences in performance among components are measured in clock intervals. Asynchronous implementation methods take advantage of subtle performance differences—differences that arise when the number of transistors in series vary depending on the data, for instance.

Normally, this difference is too small to be utilized by a synchronous design, whereas an asynchronous circuit will adapt its performance based on the value of its input even in cases where such small variations occur. Therefore, asynchronous design is the ideal vehicle to implement one of the most pervasive principles of computer architecture: make the common case fast.

1.2. Contributions

This thesis presents original contributions to asynchronous design and architecture in the following areas:

I. High-level Design [Chapter 2]
We present necessary and sufficient conditions under which the slack or degree of pipelining of a computation can be changed. The results are then used to demonstrate the correctness of the new program transformations introduced in the design of a high-performance asynchronous MIPS processor.[17]

II. Architectural Optimizations for the Average-Case [Chapters 3, 4]
We present asynchronous solutions to two problems in processor architecture:

1. We present circuits to solve the prefix problem with average-case behavior better than that possible by any traditional synchronous solution. The problem is used to construct an asynchronous adder with average-case latency better than any previously known solution. We show that the resulting circuits have optimal asymptotic average-case latency.
2. We present a distributed mechanism for the implementation of precise exceptions in an asynchronous processor that permits a data-dependent number of instructions in the main execution pipeline. This mechanism was used in the design of a high-performance asynchronous MIPS processor.[17]

III. Processor Architecture [Chapter 5]
We present a novel processor architecture for handling the problem of control dependencies introduced in an instruction stream due to the presence of branch instructions. We show how this architecture can co-exist with existing techniques for handling this problem.

Chapter 2.

Slack Elasticity

"Yes! Another couple of weeks to slack!"

—Robert Harley

We present necessary and sufficient conditions under which we can modify the slack on a channel in an asynchronous computation without changing its behavior. These results can be used to modify the degree of pipelining in an asynchronous system.

We specify a distributed computation using CHP (Appendix 1 contains a summary of the notation), and restrict our attention to systems that do not share variables among concurrent processes. The processes in the computation interact by exchanging messages over first-in first-out channels. Each channel in the computation has a fixed amount of *slack*, or buffering, which specifies the maximum number of outstanding messages on a channel.

The CHP specification of a process completely characterizes both the computation it performs as well as its synchronization behavior. For instance, we can specify a process that performs addition with the following CHP:

$$*[\ (A?x \| B?y);\ \ C!(x+y)\ ]$$

Unfortunately, for performance reasons, this specification can be very restrictive in practice. If $\mathbf{c}X$ is the number of completed actions on channel X, the specification includes the property that

$$0 \le \mathbf{c}A - \mathbf{c}C \le 1$$

In other words, the specification includes the fact that an implementation cannot accept its next set of inputs on channel A without producing an output on channel C. This restriction causes the throughput of an asynchronous delay-insensitive circuit that implements the computation to degrade as $1/\log N$, where N is the number of bits used to represent x. However, it is possible that this property of the specification is not critical—namely, modifying it to the weaker

$$0 \leq \mathbf{c}A - \mathbf{c}C \leq \log N$$

does not affect the correctness of the computation. In that case, we can prevent the throughput degradation by pipelining the computation—a significant improvement.

It is often necessary to adjust the amount of pipelining in an asynchronous computation to optimize its performance based on the timing behavior of the components of the system.[26] Quite often, the transformation amounts to changing the slack of various channels in the computation.[11] Ideally this transformation should be applied after the high-level design is completed, since we may not have the necessary timing information until the physical design of the system has been simulated. Such transformations, in general, involve examining the entire asynchronous system instead of just a single process.

We address the issues raised above by examining the following question: when can we change the slack of communication channels in the system without modifying behaviors of the system? This single transformation can be used to show the correctness (or lack thereof) of a number of different program transformations. Changing the slack of a synchronization channel is a non-trivial operation. Consider the following example in which channels A, X, and Y are slack-zero channels.

$$X;A \;\|\; A;Y \;\|\; [\; \overline{X} \longrightarrow X;Y;\; \text{“}\textit{good}\text{”} \;[]\; \overline{Y} \longrightarrow Y;X;\; \text{“}\textit{bad}\text{”} \;]$$

The only possible computation is the sequence $X; A; Y;$ “*good*.” However, if we introduce slack on channel A, we now have the possibility $A; Y; X;$ “*bad*.”

When we are permitted to add slack to a channel in the system, we say that the particular channel is *slack elastic*. If every channel in the system is slack elastic, the system is said to be slack elastic.

2.1. Semantic Framework

We assume that the computation of interest is described by a collection of CHP processes communicating via first-in first-out channels. The processes do not share any variables; all interaction is via message-passing using single-sender single-receiver channels. Let X be a command causing an "X-action" when executed. We define $\mathbf{c}X$ to be the number of *completed* X-actions since the beginning of a computation.

2.1.1. Synchronization

(X, Y) form a pair of *synchronization primitives* if the difference $(\mathbf{c}X - \mathbf{c}Y)$ is bounded.[13] Formally, there exist two integer constants $\mathbf{k}X$ and $\mathbf{k}Y$ such that at least one of the two constants is finite, and:

$$-\mathbf{k}Y \leq \mathbf{c}X - \mathbf{c}Y \leq \mathbf{k}X \qquad \text{(SAFETY REQUIREMENT)}$$

The quantity $K = \mathbf{k}X + \mathbf{k}Y$ is called the *synchronization slack*.[13]

The *probe* of a synchronization primitive can be used to determine if the action can complete.[14] Formally,

$$\overline{X} \Rightarrow (\mathbf{c}X - \mathbf{c}Y < \mathbf{k}X) \quad \wedge \quad (\mathbf{c}X - \mathbf{c}Y < \mathbf{k}X) \Rightarrow \Diamond\overline{X}$$

$$\overline{Y} \Rightarrow (-\mathbf{k}Y < \mathbf{c}X - \mathbf{c}Y) \quad \wedge \quad (-\mathbf{k}Y < \mathbf{c}X - \mathbf{c}Y) \Rightarrow \Diamond\overline{Y}$$

where $\overline{X}$ denotes the probe of synchronization primitive X, and $\Diamond E$ means that expression E becomes true eventually. Once a probe of a synchronization becomes true, it remains true until the primitive is executed. Probes can only occur in the guards of selection statements.

The value $\mathbf{q}X$ is defined as the number of X-actions currently *suspended*. The progress requirement on synchronization primitives states that the set of suspended actions is minimal, i.e., the completion of any non-empty subset of suspended actions would violate the safety requirement.[13] Formally, if (X, Y) form a pair of synchronization primitives,

$$\mathbf{q}X = 0 \vee \mathbf{q}Y = 0 \qquad \text{(PROGRESS REQUIREMENT)}$$

CHP communication channels that carry data can be described using this framework. A CHP channel C has two *ports* associated with it: $C!$, a sender port, and $C?$, a receiver port. $(C!, C?)$ form a pair of synchronization primitives. We define $\mathbf{s}C!$ to be the sequence of data values that have been sent on the sender port, and $\mathbf{s}C?$ the sequence of received values. Let $|s|$ be the length of sequence s. Then, $|\mathbf{s}C!| = \mathbf{c}C!$ and $|\mathbf{s}C?| = \mathbf{c}C?$.

2.1.2. Computations and Behaviors

We restrict our attention to systems that satisfy the properties listed below; their need will become evident in the sections that follow.

- the system is *closed*, i.e., we have specified the CHP processes of interest and their environment;

- the system is deadlock-free;
- negated probes of the sender port of channels are not used in the computation;
- if a sender port is probed, the probe will be true infinitely often.

An execution *trace* is a particular interleaving of atomic actions that can occur during execution of the system. The system is completely characterized by the set of possible traces that can occur.[24] We only consider the *complete* traces of the system.[8] The execution of processes is assumed to be weakly fair, and the selection statement is assumed to be unfair. (Appendix 1 contains a more detailed description of the model.)

Given a concurrent system, we are not interested in the possible interleavings of actions that occur in a trace. Rather, we are interested in the sequence of data values that are produced on certain channels of the system, given the sequence of values being sent on other channels. For instance, in the earlier example of the process that performs addition, we might only be interested in the fact that the data values sent on channel C correspond to the sum of the values received on channels A and B. To this end, we define a *behavior* of a system in terms of the possible traces that can occur.

A behavior in our model is primarily characterized by the sequence of values that are sent and received on the channels of the system. Since processes in the system can only interact using communication channels, behaviors capture the data values that are exchanged by interacting processes. Therefore, behaviors can be used to describe the input/output characteristics of processes in the system. In addition, we would like to specify a computation without specifying the synchronization behavior as far as possible. In our model, the only ordering between values that have been sent on various channels that can be inferred from the behavior itself is the ordering preserved by the FIFO nature of the individual channels.

Since the sequences of values sent and received on channels can be infinite, behaviors capture the notion of weakly fair execution. The notion of weak fairness in behaviors corresponds to the next value (if any) that can be sent/received on a channel being sent/received eventually.

The other component of a behavior is the sequence of non-deterministic choices made by processes in the system, since these choices can affect the data values being sent on channels. The only construct in CHP that introduces such choices is the selection statement.

We assume that all the channels in the system are initialized empty, i.e., for all channels c, $\mathbf{k}c? = 0$. The initialization of variables and channels is assumed to be part of the CHP program for each process. Therefore, the actual initial values of variables do not affect the behavior, because every variable is assigned a value before it is used.

Given the sequence of choices made by a process and the sequence of values that have been received by the process, we can completely determine the local state of a process. Therefore, our model does not include the local state of the process as part of a behavior.

Definition 2.1. (*decision point*)
Given a trace, a decision point for a process p is a point between two actions in the trace where p has selected a guard of a selection statement for execution and several guards of the selection are true.

A decision point is characterized by a tuple $(n, sel, gset, alt)$, where n is the occurrence index of the selection statement in the execution of p, sel denotes the selection statement, $gset$ is the set of guards of the selection statement that are true, and alt is the alternative chosen by p.

Decision points of the system correspond to places where a non-deterministic choice is made. We assume we have no control over the mechanism used to implement this choice; therefore, the choice made by the computation is assumed to be unfair.

Definition 2.2. (*behavior*)
Given a trace, the corresponding behavior $\mathcal{B}$ of a system is a function that maps each channel c in the system to the pair of sequences of values $(\mathbf{s}c?, \mathbf{s}c!)$ that occurred in the trace, and each process to its set of decision points in the trace.

Given a channel c and process p, we denote $(\mathbf{s}c?, \mathbf{s}c!)$ by $\mathcal{B}.c$, and the set of decision points corresponding to p by $\mathcal{B}.p$. The behavior corresponding to a trace is unique. However, multiple traces can map onto the same behavior, since different interleavings of actions that do not interact with one another will be reduced to the same behavior.

Definition 2.3. (*system*)
A system is a closed, deadlock-free collection of CHP processes and is defined by the set of behaviors that can occur during execution.

Any collection of deadlock-free processes will have at least one possible behavior. Therefore, a system will be the empty set just when it does not contain any processes.

Example. Consider the system shown below, where all channels have zero slack.

```
        *[ $X!0$ ] || *[ $Y!1$ ] || *[ $Z?w$ ]  || $p$
$p \equiv$   *[[$\overline{X} \longrightarrow X?x;\ Z!x;$ [$\overline{Y} \longrightarrow Y?x; Z!x$ | $\neg\overline{Y} \longrightarrow$ skip]
        |$\overline{Y} \longrightarrow Y?y;\ Z!y;$ [$\overline{X} \longrightarrow X?y; Z!y$ | $\neg\overline{X} \longrightarrow$ skip]
        ]]
```

It has, among others, a trace that corresponds to the sequence

$$(X!0\|X?x); (Z!0\|Z?w); (Y!1\|Y?x); (Z!1\|Z?w)\ldots$$

where the first guard $\overline{X} \to$ ... is chosen for execution with $\overline{Y}$ being true in the outer selection statement, and $\overline{Y} \to$ ... is chosen in the inner selection statement. The behavior corresponding to this trace maps Y to the pair of sequences $([1,1,\ldots],[1,1,\ldots])$, X to $([0,0,\ldots],[0,0,\ldots])$, Z to $([0,1,0,1,\ldots],[0,1,0,1,\ldots])$, and the process p to $\{(0, selout, \{\overline{X},\overline{Y}\},\overline{X}),\ (1, selout, \{\overline{X},\overline{Y}\},\overline{X}),\ldots\}$, where *selout* is the outer selection statement that selects between $\overline{X}$ and $\overline{Y}$, and the labels $\overline{X}$ and $\overline{Y}$ refer to the alternatives in the selection statement. ✳

2.1.3. Specifications and Observability

The specification of a closed CHP program is a set of behaviors. Usually, a specification does not completely specify the sequence of values sent and received on all channels of the system. Accordingly, we classify the channels of the system into *internal* and *external* channels, depending on whether or not the data values sent on those channels are part of the specification. All properties of interest must be specified only using the quantities s$E!$ and s$E?$, where E is an external channel.

Example. It is possible that we may not be able to observe certain properties of a computation, since behaviors do not contain as much information as the sequence of actions in the computation. For example, consider the two processes

```
   *[ $NCS_1$; $CS_1$ ]
|| *[ $NCS_2$; $CS_2$ ]
```

where NCS_1 and NCS_2 are non-critical sections, and CS_1 and CS_2 are critical sections. We cannot directly observe the property that two processes access their critical sections CS_i in an exclusive manner, since we can only observe the sequence of values on channels. However, we can make the mutual exclusion property visible by the introduction of a third process and an external channel C as follows:

$$\begin{array}{ll} & *[\ NCS_1;\ \ A!1; A!1; CS_1\ ] \\ \| & *[\ NCS_2;\ \ B!2; B!2; CS_2\ ] \\ \| & *[[\overline{A} \longrightarrow A?x\ [\!]\ \overline{B} \longrightarrow B?x];\ \ C!x\ ] \end{array}$$

By observing the sequence of values on channel C, we can determine if mutual exclusion is maintained. For instance, if sequence $1, 2, 1, 2, \ldots$ is possible, we have violated the mutual exclusion requirement. ⁂

Definition 2.4. (*smaller set of decision points*)
Given two sets of decision points D_1 and D_2 for a process p, we say that $D_1 \sqsubseteq D_2$ iff for every decision point $(n, sel, gset_1, alt) \in D_1$, there exists $(n, sel, gset_2, alt) \in D_2$ such that $gset_1 \subseteq gset_2$.

The relation "$\sqsubseteq$" on sets of decision points orders them in terms of the number of non-deterministic choices that were possible.

Definition 2.5. (*implementation*)
We say that a system implements a specification if for each behavior $\mathcal{B}_{sys}$ of the system, there exists a behavior $\mathcal{B}_{spec}$ in the specification such that for all external channels e , $\mathcal{B}_{sys}.e = \mathcal{B}_{spec}.e$ and for all processes p, $\mathcal{B}_{sys}.p \sqsubseteq \mathcal{B}_{spec}.p$.

This implementation relation is different from the traditional implementation relations used in trace theory and other models of concurrent programming because it does not include the synchronization behavior of the computation.

Example. Consider the following two systems:

$S_0 \equiv\ *[\ X!0\ ]\ \|\ *[\ Y!0\ ]\ \|\ *[\ X?x\ ]\ \|\ *[\ Y?y\ ]$

$S_1 \equiv\ *[\ X!0;\ Y!0\ ]\ \|\ \textbf{skip}\ \|\ *[\ X?x\ ]\ \|\ *[\ Y?y\ ]$

The computations specified by S_0 and S_1 are indistinguishable under our model because the sequence of values sent and received on channels X and Y remain unchanged, and both systems have no decision points. Standard concurrency models will differentiate them because the communications on X and Y cannot be executed in parallel in S_1, and because of the additional bound $0 \leq \mathbf{c}X - \mathbf{c}Y \leq 1$ in system S_1. Under most models, S_1 would be a valid implementation of S_0, but S_0 would not be a valid implementation of S_1. ⁂

We now present the theorems that enable a large number of transformations, including the introduction and elimination of pipelining, data-flow style process decomposition, and pipelined control distribution.

2.2. Main Results

Throughout this section we will use $\mathcal{S}$ to denote the set of possible behaviors of the system of interest, p to denote a process in the system, and c to denote a channel in the system.

Lemma 2.6. (*monotonicity*)
Let $\mathcal{S}^+$ be the system obtained from $\mathcal{S}$ by increasing the slack on a particular channel. Then $\mathcal{S} \subseteq \mathcal{S}^+$.

Proof: Consider any behavior of $\mathcal{S}$. This behavior corresponds to some execution trace of system $\mathcal{S}$. It suffices to show that this execution trace is possible in $\mathcal{S}^+$. Let c be the channel whose slack was increased from $\mathbf{k}c!$ to $\mathbf{k}c! + n$. By definition, traces from $\mathcal{S}$ satisfy $\mathbf{c}c! - \mathbf{c}c? \leq \mathbf{k}c!$. If all guards that are true in $\mathcal{S}$ are still true in $\mathcal{S}^+$, then these traces still exist in $\mathcal{S}^+$.

Increasing slack does not change the probe of the receiver end of the channel (by definition). The probe of a sender is monotonic with slack (by definition). Since we disallow negated probes of sender ports, this implies that all guards of selection statements are monotonic with slack. Hence, every trace of $\mathcal{S}$ occurs in $\mathcal{S}^+$.

A true probe on a sender port can be postponed in $\mathcal{S}^+$ until the point when it becomes true in $\mathcal{S}$; we know that the probe will eventually become true in $\mathcal{S}$ because we have assumed such probes will be true infinitely often. Thus the decision points for this trace in $\mathcal{S}^+$ can be made identical to those in $\mathcal{S}$. □

Lemma 2.6 shows that the set of behaviors is monotonic with the slack on the channels. Note that all the restrictions on computations that were mentioned in the previous section are needed for this proof.

Theorem 2.7. (*decreasing slack*)
Decreasing the slack of a channel preserves the correctness of computations if and only if it does not introduce deadlock.

Proof: Let $\mathcal{S}^-$ be the system obtained from $\mathcal{S}$ by decreasing the slack of a channel. If $\mathcal{S}^-$ is deadlock-free, $\mathcal{S}^- \subseteq \mathcal{S}$ by lemma 2.6. By definition 2.5, $\mathcal{S}^-$ implements $\mathcal{S}$. □

Definition 2.8. (*extension*)
A behavior $\mathcal{B}'$ is said to be an extension of behavior $\mathcal{B}$ iff:

$$(\forall c :: \mathcal{B}.c = \mathcal{B}'.c) \wedge$$
$$(\exists p_0 :: (\forall p : p \neq p_0 : \mathcal{B}.p = \mathcal{B}'.p) \wedge \mathcal{B}.p_0 \neq \mathcal{B}'.p_0 \wedge \mathcal{B}.p_0 \sqsubseteq \mathcal{B}'.p_0)$$

Intuitively, the extension of a behavior corresponds to the same data behavior but with at least one additional choice which did not exist in the original behavior.

We now show that the only way in which increasing the slack on a channel can affect the computation is by increasing non-determinism.

Theorem 2.9. (*increasing slack*)
Let $\mathcal{S}^+$ be the system obtained from $\mathcal{S}$ by increasing the slack of a channel. Then either $\mathcal{S} = \mathcal{S}^+$, or there exists a behavior $\mathcal{B}^+ \in (\mathcal{S}^+ - \mathcal{S})$ that is an extension of a behavior in $\mathcal{S}$.

Proof: By lemma 2.6, $\mathcal{S} \subseteq \mathcal{S}^+$. Therefore, either $\mathcal{S} = \mathcal{S}^+$, or there exists $\mathcal{B}_0 \in \mathcal{S}^+ - \mathcal{S}$. Assume such a $\mathcal{B}_0$ exists. Now $\mathcal{B}_0$ differs from every behavior in $\mathcal{S}$ in either the sequence of values sent on some channel or in the set of decision points for some process in $\mathcal{S}$. This implies that the local state of some process from $\mathcal{S}^+$ differs from the local state that could occur in $\mathcal{S}$. Consider the first point in execution when this occurs. The only non-deterministic construct in CHP is the selection statement, and therefore the only way a new local state could occur is because of a new true guard in a selection statement. By the same argument as in lemma 2.6, the guards true in $\mathcal{S}$ will eventually become true in $\mathcal{S}^+$. Therefore, we can pick an alternative of the selection statement that is possible in $\mathcal{S}$, and continue execution as in the original system $\mathcal{S}$. This new behavior is the required extension. □

The strength of Theorem 2.9 lies in the fact that if we can show that we cannot possibly introduce new decision points, this implies that adding slack does not change the behavior of a computation.

We now present some corollaries of the results of the previous section that can be used to reason about a large class of CHP programs.

2.3. Subsidiary Results

The monotonicity lemma coupled with Theorem 2.9 permits us to make the following statement that is very useful in practice.

Corollary 2.10. (*sandwich theorem*)
If a system satisfies its specification when the slack on channel c is k and is unchanged when the slack on channel c is l ($> k$), it satisfies its specification when the slack on c is s, for all s satisfying $k \leq s \leq l$.

Proof: The set of behaviors (and therefore the implementation relation) is monotonic with slack. Therefore, if the system is correct with c having slack k and the system is unchanged by increasing the slack to l, the system is unchanged all slack s satisfying $k \leq s \leq l$, concluding the proof. □

When computations are entirely deterministic, we expect we can introduce slack on any channel without affecting correctness.

Corollary 2.11. (*deterministic computations*)
If the guards in selection statements are syntactically mutually exclusive and there are no probed channels, the system has only one behavior.

Proof: Since the computation is deterministic, the sequence of values sent on channels is always the same and there are no decision points. □

A selection statement with probed channels in its guards is said to exhibit *maximal non-determinism* if all the guards can be true whenever the selection statement is executed.

Corollary 2.12. (*maximal non-determinism*)
If all selection statements with probes have maximal non-determinism, the system is slack elastic.

Proof: The set of decision points of the system cannot be increased, so by Theorem 2.9 we can increase the slack on any channel without changing the behavior of the system. □

Corollary 2.12 is extremely useful in practice. The design of the MIPS R3000 processor undertaken by our group satisfies its requirements.

Consider the problem of measuring the slack of a channel c. To be able to measure the slack of c, we must be provided with a collection of processes to which c is connected, and a single channel which produces one output on channel *result*: *true*, if the slack of c is equal to a specified value, say k, or *false* otherwise. We claim that this task is impossible under the assumptions of the model.

Corollary 2.13. (*impossibility of measuring slack*)
It is not possible to measure the slack of a communication channel.

Proof: Assume that a collection of deadlock-free processes can be used to answer the question "is the slack of channel c equal to k?" Consider the closed system $\mathcal{S}$ where we observe channel *result*, and make c have slack k. The only possible output on *result* is *true*, by our assumption. Let $\mathcal{S}^+$ be the system, where we add slack 1 to channel c. By Theorem 2.7, $\mathcal{S}$ implements $\mathcal{S}^+$. Therefore, *result* can produce the value *true* in $\mathcal{S}^+$—a contradiction. □

More generally, if a system can be used to compute any relationship between the slack of a set of channels, then the relation must be trivial—i.e., the system always outputs *true* or always outputs *false*.

2.4. Shared Variables

When a CHP program uses shared variables, we can eliminate them by the introduction of a process which controls access to the shared variable via communication channels. Each reader and writer of the shared variable is given a separate channel to access the particular variable. If the writes to the shared variable are not mutually exclusive, or reads and writes are not mutually exclusive, then the violation of mutual exclusion will be visible in a decision point in the selection statement of the process that implements the shared variable.

A special type of shared variable that might be used is in a *synchronizer*. A synchronizer is the following CHP process:

```
*[[ $x \wedge \overline{V}$  $\longrightarrow$  $V$!true
 |$\neg x \wedge \overline{V}$  $\longrightarrow$  $V$!false
 ]]
```

where x is a variable which can change from true to false, or from false to true at any instant. Since the variable is shared, it can change while being evaluated, and both x and $\neg x$ can evaluate to true.

The synchronizer is used when we have no control over when x can change. Therefore, the results we have presented earlier still apply, as the set of decision points for this process is maximal. Therefore, using a synchronizer does not affect slack elasticity.

2.5. Applications

When designing asynchronous systems, we can increase the slack on a particular channel under the conditions outlined above. We now present some important transformations that can be shown to be semantics-preserving using the results derived above.

2.5.1. Pipelining

Pipelining is a technique whereby the computation of a function is distributed over a number of stages so as to reduce the cycle time of the system—increasing the throughput—at the cost of increasing the latency of the computation.

Consider the following program:

$*[\ L?x;\ \ R!g(f(x))\ ]$

We introduce pipelining when we transform it into the program shown below:

$*[\ L?x;\ \ I!f(x)\ ]\ \|\ *[\ I?y;\ \ R!g(y)\ ]$

It should be clear that we can apply this transformation if and only if we are permitted to increase the slack on channels L or R. Under those conditions, we can formally pipeline a computation as follows:

$*[L?x; R!g(f(x))]$

$=$ { add slack 1 to channel R, introducing internal channel I }

$*[L?x; I!g(f(x))]\ \|\ *[I?y; R!y]$

$=$ { distribute computation }

$*[L?x; I!f(x)]\ \|\ *[I?t; R!g(t)]$

2.5.2. Control Distribution

In designing a delay-insensitive system, we face a problem when attempting to design datapaths where the quantities being manipulated are composed of a large number of bits. The problem is illustrated by examining the circuit implementation of the following program:

$*[\ L?x;\ \ R!x\ ]$

Before we send value x on channel R, we must be sure that all the bits used to represent x have been received on channel L. The circuit that waits for all the bits to have been received has a cycle time that is proportional to $\log N$, where N is the number of bits. As a result, as we increase the number of bits in x, the system throughput will decrease.

Instead, we examine an alternative implementation strategy. We implement channel L using an array of $\Theta(N)$ channels, where the individual channels have a fixed number of bits. As a result, we transform the program shown above into:

$*[\ (\|i :: L[i]?x[i]);\ \ (\|i :: R[i]!x[i])\ ]$

We have moved the performance problem from the implementation of the communication action on a channel to the implementation of the semicolon that separates the L and R actions. However, we observe that there is no data-dependency between channels $L[i]$ and $R[j]$ when $i \neq j$. We will attempt to remove the synchronization between the parts of the program that are not data-dependent.

We introduce a process that enforces the sequencing specified by the program above. The original program is equivalent to:

$(\|i :: *[\ S[i] \bullet L[i]?x[i];\ \ S[i] \bullet R[i]!x[i]\ ])$
$\|\ *[\ (\|i :: S[i])\ ]$

since the $S[i]$-actions ensure that the actions on channels L and R are properly sequenced.

Now, we increase the slack on channels $S[i]$. If we let the slack on channels $S[i]$ go to infinity, the program shown above is equivalent to:

$(\|i :: *[\ L[i]?x[i];\ \ R[i]!x[i]\ ])$

Therefore, we can transform the original program into this one if and only if we can add slack on channels $S[i]$. Observe that we now have $\Theta(N)$ *independent* processes, and increasing N will not affect the throughput of the system.

This transformation can be generalized into a technique for control-data decomposition. Traditional techniques for the decomposition of a process into control and data consist of replacing actions $D!x$ and $D?x$ by Ds and Dr, and introducing processes

$*[\ Ds \bullet D!x\ ]\ \|\ *[\ Dr \bullet D?x\ ]$

If x is an N-bit binary integer, then once again the cycle time of communication actions on the D channels would be $\Theta(\log N)$.

Instead, we apply the following transformation. Let I_i be the possible channels that write to variable x, and let O_i be the possible channels that read from variable

x. We replace actions $I_i?x$ with $C!(\mathbf{true}, i); D1$, and $O_i!x$ with $C!(\mathbf{false}, i); D1$. We introduce the following processes:

$$*[\ C?(b,k);\ \ [b \longrightarrow I_k?x\ \ [\!]\neg b \longrightarrow O_k!x];\ \ D2\ ]\ \|\ *[\ D2; D1\ ]$$

This transformation does not affect the correctness of the computation. By splitting up the input and output channels into an array of $\Theta(N)$ channels that carry a constant number of bits of data, we can transform the computation as follows:

$$\begin{aligned}&(\|i :: *[\ \ C[i]?(b,k);\ \ [b \longrightarrow I_k[i]?x[i]\ \ [\!]\neg b \longrightarrow O_k[i]!x[i]]; D2[i]])\\ \|\ &*[\ (\|\ i :: D2[i]);\ \ (\|\ i :: D1[i])\ ]\end{aligned}$$

and modify $C!(b,k); D1$ into $(\|i :: C[i]!(b,k)); (\|\ i :: D1[i])$.

The parallel control distribution $(\|i :: C[i]!(b,k))$ can be converted into a control distribution *tree*, where the value (b,k) is copied from one channel to $\Theta(N)$ leaves. We will call the control distribution tree *copytree*(R,C), where R is the root of the tree, and C is the array of channels above. The computation is equivalent to:

$$\begin{aligned}&(\|i :: *[\ \ C[i]?(b,k);\ \ [b \longrightarrow I_k[i]?x[i]\ \ [\!]\neg b \longrightarrow O_k[i]!x[i]]; D2[i]])\\ \|\ &*[\ (\|\ i :: D2[i]);\ \ (\|\ i :: D1[i])\ ]\\ \|\ &\mathit{copytree}(R, C)\end{aligned}$$

where we replace $(\|i :: C[i]!(b,k))$ with $R!(b,k)$. Now, we are ready to introduce slack on channels $D1[i]$ and $D2[i]$. When we let the slack on these channels go to infinity, we are left with:

$$\begin{aligned}&(\|i : *[\ \ C[i]?(b,k);\ \ [b \longrightarrow I_k[i]?x[i]\ \ [\!]\neg b \longrightarrow O_k[i]!x[i]\ \ ])\\ \|\ &\mathit{copytree}(R, C)\end{aligned}$$

and the main control distribution turns into $R!(b,k)$. If we examine the net effect of this transformation, we observe that all semicolons that wait for $\Theta(N)$ actions to complete have been eliminated. Therefore, the throughput of the system no longer depends on N—a significant improvement. We have increased the latency of control distribution (although asymptotically the latency of the control distribution is still $\Theta(\log N)$, we have increased constant factors). However, we should not fail to observe that we have introduced communication on channel R; the number of bits sent on R is $\Theta(\log(|I| + |O|))$, where $|I|$ is the number of channels that write to variable x, and $|O|$ is the number of channels that read from variable x.

2.5.3. General Function Decomposition

In general, if we have a computation graph which is supposed to implement a function that has a simple sequential specification, we can show its correctness by introducing "ghost channels" which sequence all the actions in the computation graph. A single process that sequences all the actions in the computation is introduced, so that the resulting system mimics the behavior of the sequential program. Adding slack to the ghost channels introduced for sequencing permits the processes in the computation graph to proceed in parallel; when we add infinite slack to the sequencing channels, we have a computation that behaves exactly like the original computation without the sequencer process, and the ghost channels can be deleted without modifying the behavior of the computation. Therefore, showing the correctness of the original computation can be reduced to showing whether adding slack on the ghost channels modifies the behavior of the system.

Example. Suppose we would like to demonstrate that the following CHP program implements a first-in first-out buffer:

$$*[\ L?x;\ \ U!x;\ \ L?x;\ \ D!x\ ]\ \|\ *[\ U?y;\ \ R!y;\ \ D?y;\ \ R!y\ ]$$

We begin by closing the system with the introduction of two processes which send data on channel L and receive data from channel R. Next, we introduce a sequencer process which sequences the actions in the computation. The resulting system is shown below.

$$\begin{array}{l} \quad i := 0; *[\ L!i;\ \ i := i+1\ ]\ \|\ *[\ R?w\ ] \\ \|\ *[\ L?x \bullet S_1;\ \ U!x \bullet S_2;\ \ L?x \bullet S_4;\ \ D!x \bullet S_5\ ] \\ \|\ *[\ U?y;\ \ R!y \bullet S_3;\ \ D?y;\ \ R!y \bullet S_6\ ] \\ \|\ *[\ S_1;\ \ S_2;\ \ S_3;\ \ S_4;\ \ S_5;\ \ S_6\ ] \end{array}$$

The sequencer process restricts the computation so that only one interleaving is possible, namely the sequence

$$(L!0\|L?x); (U!x\|U?y); (R!y\|R?w); (L!1\|L?x); (D!x\|D?y); (R!y\|R?w);$$
$$(L!2\|L?x);\ \ldots$$

which clearly implements a first-in first-out buffer, since the sequence of values sent on R is the same as the sequence of values received on L. We can increase the slack on channels S_i without modifying its behavior because the computation is deterministic. In the limit of infinite slack on the channels S_i for all i, the sequencer process does not enforce any synchronization between the actions, and we can eliminate the sequencer process entirely leaving us with the original computation. Therefore, the original computation implements a first-in first-out buffer. ⁜

2.6. A Recipe for Slack Elastic Programs

Corollary 2.12 can be used as a guideline for the design of programs that are guaranteed to be slack elastic. Ensuring slack elasticity of the design is important in order to be able to postpone decisions related to the amount of pipelining to be used in an implementation. In the design of an asynchronous MIPS processor, we found it necessary to adjust the slack on communication channels after most of the physical layout was complete because we did not have accurate estimates of the timing behavior of the processes we used until analog simulations were performed.

There are two selection statements in CHP. Selection statements that are described using the thick bar "▯" indicate that the guards are mutually exclusive. If such selection statements do not use any probes in their guards, they cannot be the cause of the introduction of new decision points. Selection statements that use the thin bar "|" indicate that their guards might not be mutually exclusive. If such selection statements are maximally non-deterministic—i.e., if the computation meets its specification irrespective of the alternative chosen when the selection is encountered, then they will not be the cause of erroneous computations. If we follow these two guidelines, we will be guaranteed that the computation is slack elastic. Every process in the high-level description of the asynchronous MIPS processor we designed satisfied these criteria.

2.7. Discussion

If a computation is entirely deterministic, if a computation has non-determinism only because of local variables, or if a computation is such that its specification permits all possible decision points that might occur in the computation to occur, we know that the computation is slack elastic. In general, however, only some of the channels in a computation are slack elastic. We characterize the cases when a channel is not slack elastic in this section.

By Theorem 2.9, we are guaranteed that erroneous computations are only introduced by the extension of a behavior that used to occur before slack was introduced. This implies that we now have more true guards in a selection statement than were possible earlier. In addition, this must be a result of the probe of a channel being true that was not true in the original computation, since that is the only way an external transformation can affect another process.

Consider the case when two probes are supposed to be mutually exclusive in a computation. For simplicity, consider the case when the two probes are in the guard of a selection statement, as shown below.

$p_1 \equiv \; ... \; [\overline{A} \longrightarrow \; ... \; A \; ... \; [] \overline{B} \longrightarrow \; ... \; B \; ... \;] \; ...$

We are given that these probes are mutually exclusive. This implies that the actions A and B on the two channels are ordered in some manner. Without loss of generality, consider the case when A occurs before B. For this ordering to be preserved, when B is attempted, we must be sure that action A has completed. There are only two processes in the system which can determine that A has completed—p_1 shown above, and the process that attempts action A.

If actions A and B are in a single process p_0, then the ordering between A and B is guaranteed by a semi-colon in process p_0. If that is the only mechanism used to enforce the ordering, channel A is not slack elastic, because adding slack on channel A could violate mutual exclusion between the guards in p_1 as A could complete before p_1 selected the guarded command $\overline{A} \rightarrow ...A...$ for execution, permitting $\overline{B}$ and $\overline{A}$ to be true simultaneously.

The actions on A and B might be ordered by a chain of synchronization actions. The completion of A would initiate this chain, and the action B would be blocked by a synchronization channel at the end of the chain. This chain can be initiated by the completion of A in process p_1, or in the process which attempts action A. In the latter case, channel A is not slack elastic for the same reason as above. However, if process p_1 initiates the chain of synchronization events, channel A is slack elastic since it is the completion of A in process p_1 that causes action B to be initiated. We are guaranteed that $\overline{A}$ is false when the chain of events triggering B begins. In either case, we can introduce slack in the chain of synchronization actions so long as the action that B is waiting for cannot complete before the action initiated after the completion of A is executed.

2.8. Related Work

To prove the results presented in this chapter, we used a new model to describe a computation by introducing the notion of a behavior. Observe that we cannot use a traditional trace theoretic model to describe the computation—indeed, the results of this chapter would be false under a standard trace theoretical semantics. The reason for this is that using trace theoretic specifications is an over-specification for the cases in which we are interested. Trace theory captures synchronization behavior—the very behavior we are attempting to change. Increasing the slack on a channel will increase the set of possible traces, thus violating the traditional subset refinement relation of trace theory. Therefore, adding slack will not be a refinement in traditional trace theoretic models.

In van de Snepscheut's work on trace theory, slack independence is defined to be correctness under arbitrary slack.[24] This work is closely related to delay-insensitivity in asynchronous circuits. However, no attempt is made to determine when computations have such properties, which is one of the contributions of our work. In addition the model only includes demonic choice, which is insufficient to model selection statements with probed channels in their guards.

Misra and Chandy[19] describe computations using only the sequence of communication actions on channels; their model is based on Hoare's theory of traces. Their model specifies the interleaving of actions among different channels and as a result resembles other approaches that consider interleavings of actions among processes. They use projection to extract the sequence of values sent/received on an individual channel from the trace and use relations among such sequences to express many program properties.

Chapter 3.

Parallel Prefix

"What's one and one and one and one and one and one and one and one and one and one?" "I don't know," said Alice, "I lost count." "She can't do Addition," the Red Queen interrupted.

—Lewis Carroll, *Through the Looking Glass*

We present asynchronous circuits to solve the prefix problem with $O(N \log N)$ circuit size, $O(\log N)$ worst-case latency, and $O(1)$ cycle time. If the prefix operation has a right zero, the asynchronous solution has an average-case latency of $O(\log \log N)$. The construction can be used to obtain an $O(1)$ cycle time asynchronous adder with $O(N \log N)$ circuit size and $O(\log \log N)$ average-case latency. We prove that our circuits have optimal asymptotic average-case latency.

Let $\otimes$ be an associative operation. The *prefix problem* is to compute, given $x_1, x_2, \ldots, x_N$, the results $y_1, y_2, \ldots, y_N$, where $y_k = x_1 \otimes x_2 \otimes \cdots \otimes x_k$, for $1 \leq k \leq N$.[9]

We construct asynchronous solutions to the prefix problem that are similar to their synchronous counterparts. We improve the average-case performance of the asynchronous solution by using two competing methods for solving the prefix problem and picking the one that arrives earliest to produce the output. This technique reduces the average-case latency from $O(\log N)$ to $O(\log \log N)$ when the prefix operator has a right zero, a significant improvement. We show that our solutions have optimal asymptotic average-case latency.

A number of problems can be formulated as a prefix problem. Ladner and Fisher show how the prefix problem can be used to parallelize the computation of an arbitrary Mealy machine.[9] Leighton discusses a number of different problems that can be solved using prefix computations.[10] As a concrete application, we use the construction to obtain an asynchronous adder which has $O(1)$ cycle time, $O(N \log N)$ circuit size, $O(\log N)$ worst-case latency, and $O(\log\log N)$ average-case latency.

3.1. Traditional Solution

To formulate the prefix problem in terms of an asynchronous CHP program, we assume that the inputs $x_1, x_2, \ldots, x_N$ arrive on input channels $X_1, X_2, \ldots, X_N$ respectively, and that the outputs $y_1, y_2, \ldots, y_N$ are to be produced on output channels $Y_1, Y_2, \ldots, Y_N$ respectively. The problem can be restated in terms of reading the values x_i from the input channels, computing the y_i values, and sending these values on the appropriate output channels. In terms of CHP, the immediate solution that leaps to mind is the following program:

```
*[ $X_1?x_1$,  $X_2?x_2$,  …,  $X_N?x_N$;
   $Y_1!x_1$,  $Y_2!(x_1 \otimes x_2)$,  …,  $Y_N!(x_1 \otimes x_2 \otimes \cdots \otimes x_N)$
]
```

This program is very inefficient for a number of reasons, the most obvious being that there are $O(N^2)$ $\otimes$-operations, which correspond to $O(N^2)$ circuit elements; but it will serve as a specification for the problem.

For the purposes of this chapter, we will assume that the operation $\otimes$ has an identity e. This is merely an aid to clarity—it does not detract from the construction in any way.

Since we know input value x_i at position i, we can solve the prefix problem if we can determine $x_1 \otimes \cdots \otimes x_{i-1}$ at position i. Assume we had a method that computed the prefixes we needed for a problem of size n. We will extend it to compute the prefixes we need of size $2n$ as follows. We begin by using $x_{2i-1} \otimes x_{2i}$ as the input to the n-input prefix computation graph. The result of this operation would be to compute values $x_1 \otimes \cdots \otimes x_{2i}$ at output position $i+1$. We can now solve the prefix problem of size $2n$ by producing $x_1 \otimes \cdots \otimes x_{2i}$, and $x_1 \otimes \cdots \otimes x_{2i+1}$. The program to do this is described by

```
$UP(L, R, U, V, Ld, Rd) \equiv$
    *[ $L?x, R?y$;  $U!(x \otimes y)$;  $V?p$;  $Ld!p, Rd!(p \otimes x)$ ]
```

where the channels U and V correspond to the input and output stages of the prefix computation graph of half the size. From the structure of the solution, it is clear that the computation graph is a tree. Repeating this observation, all that remains is to provide a solution to the prefix problem of size 2—the root of the tree, and to read the inputs and produce the final outputs.

The V channel at the root of the tree requires the empty prefix—the identity e. The output U of the root is not used by any other process. Thus, we simplify the root process to:

$$ROOT(L, R, Ld, Rd) \equiv$$
$$*[\ L?x, R?y;\ \ Ld!e, Rd!x\ ]$$

where e is the identity of $\otimes$. The leaves of the prefix computation tree read the inputs, their prefix (from the tree), and produce the appropriate output. A leaf process is written as:

$$LEAF(X, U, V, Y) \equiv$$
$$*[\ X?x;\ U!x;\ \ V?y;\ Y!(y\otimes x)\ ]$$

Part of the computation graph for the prefix problem when $N = 4$ is shown in Figure 3.1.

Observe that the sequencing between $U!(x\otimes y)$ and $V?p$ is enforced by the environment of the UP process. We can therefore split the process into two parts that execute in parallel. However, the obvious split would cause variable x to be shared between the two processes. We introduce a local channel C which is used to copy the value of x. The new UP process is:

$$UP(L, R, U, V, Ld, Rd) \equiv$$
$$*[\ L?x, R?y;\ \ U!(x\otimes y),\ \ C!x\ ]\ \|\ *[\ C?c, V?p;\ \ Ld!p, Rd!(p\otimes c)\ ]$$

These two processes are identical! Therefore, we write:

$$UP2(A, B, C, D) \equiv$$
$$*[\ A?x, B?y;\ \ C!(x\otimes y), D!x\ ]$$

$$UP(L, R, U, V, Ld, Rd) \equiv UP2(L, R, U, C)\ \|\ UP2(V, C, Rd, Ld)$$

Similarly, we can rewrite the $LEAF$ process as:

$$LEAF(X, U, V, Y) \equiv$$
$$*[\ X?x;\ U!x, C!x]\ \|\ *[\ C?c,\ V?y;\ Y!(y\otimes c)\ ]$$

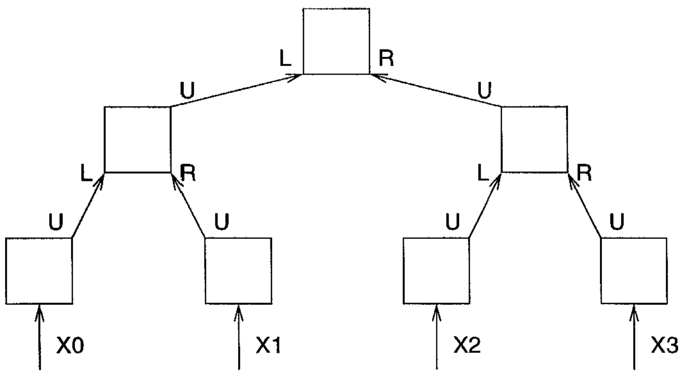

Figure 3.1. Solution to the prefix problem.

Since each node in the tree contains a constant number of $\otimes$ computations and there are $O(N)$ bounded fan-in nodes in the tree, there are $O(N)$ $\otimes$-computation circuits in the solution. Since the tree is of depth $O(\log N)$, the latency and cycle time of this solution is $O(\log N)$.

3.2. Pipelining

The solution presented above has a cycle time of $\Theta(\log N)$ since the prefix computation tree can only perform one prefix computation at a time. We can pipeline the computation to permit the tree to operate simultaneously on multiple inputs and reduce the cycle time to $O(1)$.

Consider a single *UP* node in the prefix computation tree. There are no pipeline stages between the two halves of process *UP*, since they communicate through a slack-zero channel C. However, the second process that is part of *UP* cannot complete its computation until it receives a value on channel V. This value is computed by a circuit which has a number of pipeline stages proportional to the depth of *UP* in the tree. Therefore, even though there are $O(\log N)$ pipeline stages on the computation for V, we cannot have $O(\log N)$ computations being performed by the tree since channel C has zero slack. Therefore, we introduce buffering on C proportional to the depth of the node in the tree. Logically, it is simpler to visualize the computation by "unfolding" the tree into two parts—the up-going phase, and down-going phase—as shown in Figure 3.2. The vertical arrows are the internal channels C, and two boxes connected by vertical arrows correspond to a single node in the tree.

It is clear that one must add $2d - 1$ stages of buffering on the internal channel C for a node that is d steps away from the root for the circuit to be pipelined in

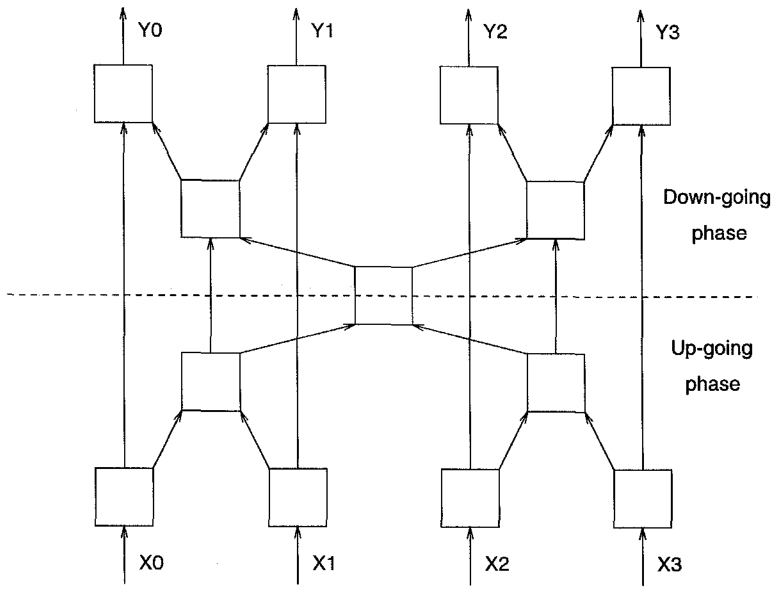

Figure 3.2. Unpipelined prefix computation.

a manner that permits $2\lg N + 1$ prefix operations to be performed simultaneously. Figure 3.3 shows the tree after the appropriate buffers have been introduced.

The cycle time of the pipelined prefix computation with buffers does not depend on the number of inputs, but on the time it takes to perform the $\otimes$ operation. The latency of the computation block is proportional to the number of stages, and is therefore $2\lg N + 1$ stages both with and without the buffers. However, we have increased the circuit size from $O(N)$ to $O(N \log N)$ since we have introduced $O(N \log N)$ buffers.

3.3. Reducing the Average-Case Latency

If the prefix computation is not used very often, the observed performance depends on the latency of the prefix computation—a quantity that is not reduced by adding buffers to the computation tree. In this section, we present a technique that reduces the average-case latency of the prefix computation in certain cases. We begin by considering a simple solution to the prefix problem.

The simplest way to perform the prefix computation is in a sequential fashion. Since we have n different input channels, we use n processes, one for each input channel, connected in a linear fashion as shown in Figure 3.4.

The stage for x_k receives y_{k-1} on channel L from the previous stage and x_k on

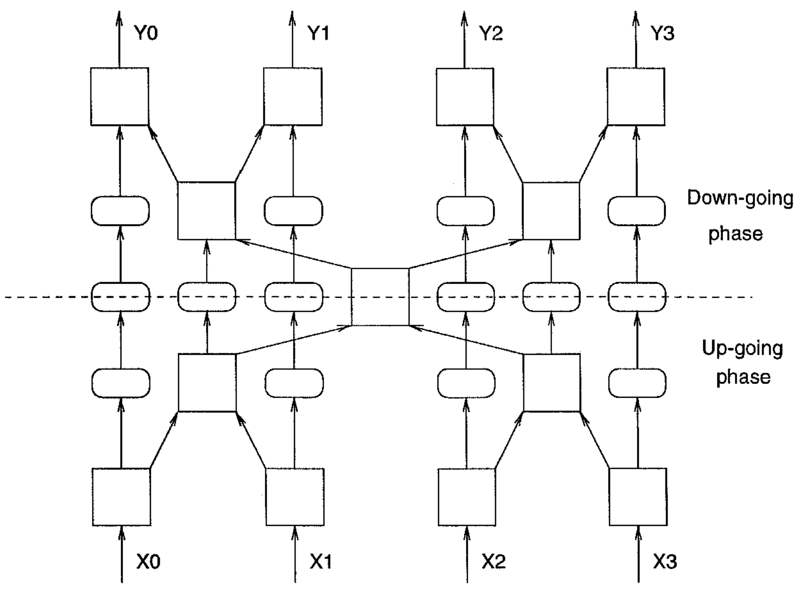

Figure 3.3. Pipelined prefix computation with buffers.

channel X_k and produces y_k on channel Y_k as well as channel R which connects it to the next stage. The CHP for an intermediate stage of such a solution is given by:

```
$SERIAL(X,Y,L,R) \equiv$
        $*[\ X?x, L?p;\ \ Y!(p \otimes x), R!(p \otimes x)\ ]$
```

However, we know that the input on channel X arrives much sooner than the input on channel L. Given this information, is it possible to produce the outputs on Y and R before receiving the input on L?

Suppose we know that a is a right zero of the prefix operation, i.e., $x \otimes a = a$ for all values of x. Now, if the input on channel X is equal to a, we can produce the output on Y and R *before* reading the value on L. We rewrite *SERIAL* as:

```
$SERIAL(X,Y,L,R) \equiv$
        $*[\ X?x;$  $[\ x = a \longrightarrow Y!a, R!a, L?p$
                    $[]\ x \neq a \longrightarrow L?p;\ \ Y!(p \otimes x), R!(p \otimes x)$
                    $]$
        $]$
```

The time taken for this solution to produce the output is *data-dependent*. In the best case (when all inputs are a), the time from receiving the inputs to producing the output is constant—much better than the prefix computation tree, and in the worst

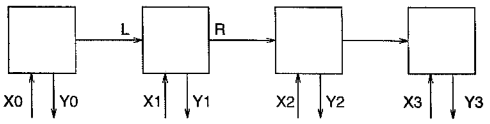

Figure 3.4. Serial prefix computation.

case, the time taken is $O(N)$—much worse than the prefix computation tree which only takes $O(\log N)$ time.

The solution we adopt is to *combine* both the prefix computation tree and the serial computation into a single computation. The two computations compete (in time) against one another, and we can pick the solution that arrives first. This technique has a worst-case latency of $O(\log N)$, but a best-case latency of $O(1)$.

We begin with the unpipelined prefix computation corresponding to Figure 3.2. The CHP for the *LEAF* process used by the prefix computation tree is:

```
$LEAF(X, U, V, Y) \equiv$
        $*[\ X?x;\ U!x,\ C!x]\ \ \|\ \ *[\ C?c,\ V?y;\ Y!(y \otimes c)\ ]$
```

Observe that the value received along channel V for a leaf which receives x_k as input is the same as the value received along channel L by the corresponding process in the serial computation shown in Figure 3.4.

We introduce channels L and R from the serial computation into the prefix computation tree. The output Y from the leaf process is simply copied on outgoing channel R. Since the values received on L and on the corresponding V are the same, we combine these two channels externally using a merge process that picks the first input that arrives, as follows:

```
$MERGE(L, V, M) \equiv$
        $*[[\overline{L} \longrightarrow L?y; M!y,\ V?yy$
          $|\overline{V} \longrightarrow V?y; M!y, L?yy$
         $]]$
```

The new *LEAF* process is:

```
$LEAF(X, U, V, L, R, Y) \equiv$
        $*[\ X?x;\ U!x,\ C!x]\ \ \|\ \ MERGE(L, V, M)\ \ \|\ \ SERIAL(C, Y, M, R)$
```

The compilation of *SERIAL* depends on the structure of $\otimes$. The compilation of the *MERGE* procedure that picks the first input is given below:

```
$*[[\neg Ma]; [v(L) \vee v(V)]; M \Uparrow; ([v(L)]; La\uparrow), ([v(V)]; Va\uparrow);$
  $[Ma]; M \Downarrow; ([n(L)]; La\downarrow), ([n(V)], Va\downarrow)$
 $]$
```

This circuit has an efficient implementation because we know that the value being received on both L and V will be the same.

Finally, using a similar transformation, we can replace process *UP* in the prefix computation tree by one that also has a serial computation phase. The original *UP* process was:

```
$UP(L, R, U, V, Ld, Rd) \equiv$
        $*[\ L?x, R?y;\ U!(x\otimes y),\ C!x\ ]\ \|\ *[\ C?c, V?p;\ Ld!p, Rd!(p\otimes c)\ ]$
```

The value to be sent along the "right" channel for the serial computation, namely *SR*, is given by $p \otimes x \otimes y$. We therefore introduce an additional internal channel C', along which the value $x \otimes y$ is sent. Finally, the "left" channel for the serial computation, namely *SL*, is merged with V using the same *MERGE* process shown above. We obtain:

```
$UP(SL, SR, L, R, U, V, Ld, Rd) \equiv$
          $*[\ L?x, R?y;\ U!(x\otimes y),\ C'!(x\otimes y),\ C!x\ ]$
        $\|\ MERGE(SL, V, M)$
        $\|\ *[\ C?c, M?p;\ M1!p, Ld!p, Rd!(p\otimes c)\ ]$
        $\|\ *[\ C'?d;\ [d = a \longrightarrow SR!a, M1?p\ [\!]\ d \neq a \longrightarrow M1?p; SR!(p\otimes d)]\ ]$
```

Since this solution follows from the unpipelined version of the prefix computation, its cycle time is $O(\log N)$. To improve its cycle time this time, we need to add buffering to both channels C and C'. This transformation will once again increase the circuit size from $O(N)$ to $O(N \log N)$. For reasons to be discussed in the following section, we use binary tree buffers to implement the buffering on channels C and C' instead of linear buffers.

3.4. Analysis of the Average Case

The latency of the prefix computation is data-dependent. We therefore need some information about the input distribution to determine the average-case latency. Consider process *SERIAL* shown below that is part of the prefix computation.

```
$SERIAL(X,Y,L,R) \equiv$
    *[ $X?x$;  [ $x = a \longrightarrow Y!a, R!a, L?c$
               [] $x \neq a \longrightarrow L?c;\ \ Y!(c \otimes x), R!(c \otimes x)$
               ]
    ]
```

When $x \neq a$, the output on Y and R depends on the input c. We call this the *propagate* case, since the output of the process depends on the input c. Let the probability of a particular input being a be p, and let this distribution be independent across all the n inputs. If the inputs remain independently distributed, the analysis below is applicable even if the probability of the input being a at input position i might vary (as long as it remains non-zero), since we can pick p to be the smallest value as a conservative approximation.

Theorem 3.1.
If the inputs of the prefix computation are independently distributed with non-zero probability of an input being a right zero, the average-case latency of the modified asynchronous prefix computation is $O(\log\log N)$, where N is the input size.

Proof: Let $L(N)$ be the latency through a prefix computation with N inputs. We assume that the prefix computation uses a k-ary tree for the purpose of this analysis. We can write:

$$L(N) = \min\left(ms, L\left(\frac{N}{k}\right) + h\right)$$

where m is the length of the longest sequence of "propagate" inputs, s is the delay through a single stage of the serial "propagate" chain at the leaves of the tree, and h is the delay through one stage of the tree. The first part of the formula comes from the serial computation, and the latter from the tree computation. To expand $L(\frac{N}{k})$, observe that at the next stage in the tree, m will be replaced by m/k since we are considering the *same* input. Applying this expansion recursively, we obtain:

$$L(N) = \min_{m \geq k^i}\left(\frac{ms}{k^i} + ih\right)$$

In particular, choosing $m = k^i$ we obtain:

$$L(N) \leq s + \frac{h}{\log k}\log m$$

The average latency is bounded above by:

$$\mathrm{E}[L(N)] \leq s + \frac{h}{\log k}\mathrm{E}[\log m]$$

To compute the expected value of $\log m$, observe that

$$\mathrm{E}[\log m] \leq \log \mathrm{E}[m]$$

since the expected value of the logarithm of a random variable is the logarithm of the geometric mean of the variable. Since the arithmetic mean is always at least the geometric mean and log is increasing (m is always non-negative), the above inequality follows. We can bound $\mathrm{E}[L(N)]$ from above if we determine $\mathrm{E}[m]$.

When $p = 1/2$, we know that $\mathrm{E}[m] \leq \log_2 N$.[1] A simple extension of the proof shows that

$$\mathrm{E}[m] \leq \lceil \log_{1/(1-p)} N \rceil + \frac{1}{pN} = O(\log N)$$

when $0 < p < 1$ (a complete proof is given in Appendix 2). Therefore, the average latency through the prefix computation is bounded above by:

$$\begin{aligned} \mathrm{E}[L(N)] &\leq s + \frac{h}{\log k}\log\left(\lceil \log_{1/(1-p)} N \rceil + \frac{1}{pN}\right) \\ &= O(\log\log N) \end{aligned}$$

concluding the proof. □

When the prefix computation operates with $O(1)$ cycle time, the value of s given above is a function of N. Since we add $2d - 1$ stages of buffering at depth d in the tree for the serial computation part as well, the value of s is bounded above by a function that depends on the latency of a buffer of size $O(\log N)$. Since we have used a binary tree buffer to implement the slack on the internal channels, the latency of a buffer of size $O(\log N)$ is $O(\log\log N)$. Therefore, the additional buffering required to reduce the cycle time of the circuit does not increase the order of the average-case latency.

3.5. Reducing the Area Overhead

The $O(\log\log N)$ average-case latency adder has $O(N\log N)$ additional circuit size because of the additional buffering required. In this section we show how the area overhead of the prefix computation circuit can be reduced by using the fact that the input distribution is independent.

On examination of the analysis for average-case latency, we make the following observation. The way we achieve an average-case latency of $O(\log\log N)$ is as follows. We traverse up the tree computation $O(\log\log N)$ steps. At this point, the average propagate-chain length is $O(1)$, and we use the serial part of the computation. In another $O(\log\log N)$ steps, we propagate the results down the tree. This permits us to complete the prefix computation with a latency of $O(\log\log N)$ steps. Therefore, we should be able to achieve the same average-case latency with lower area overhead by using the serial part of the computation only at one stage of the prefix computation tree.

Assume that we add the serial phase of the computation at only one level of the prefix computation that is d steps away from the leaves of the tree. The latency is given by:

$$L(N) = \min\left(d\cdot h + ms/k^d, h\log_k N\right)$$

On average, the latency would be:

$$\begin{aligned}\mathrm{E}[L(N)] &\leq \min\left(d\cdot h + s/k^d\cdot \mathrm{E}[m], h\log_k N\right)\\ &= \min\left(d\cdot h + s/k^d\cdot \log_{1/(1-p)} N, h\log_k N\right)\end{aligned}$$

We attempt to determine the minimum value of this function by differentiating the first part of the minimum expression with respect to d. We obtain:

$$\begin{aligned}d_{min} &= \log_k\left(s/h\cdot \ln k \log_{1/(1-p)} N\right)\\ &= O(\log\log N)\end{aligned}$$

When we add a serial phase to this stage of the prefix computation tree, the average-case latency is given by:

$$\begin{aligned}\mathrm{E}[L_{min}(N)] &\leq \frac{h}{\ln k} + \log_k\left(s/h\cdot \ln k \log_{1/(1-p)} N\right)\\ &= O(\log\log N)\end{aligned}$$

Since we added a serial phase $O(\log\log N)$ steps away from the leaves, the additional area required to permit the computation to run at full throughput is $O(N)$ since we have $O(N/\log N)$ nodes, with $O(\log(N/\log N))$ buffering required for each of them.

If we are willing to sacrifice throughput, we can reduce the area overhead even further and still have $O(\log\log N)$ average-case latency. Observe that we no longer need the tree computation beyond d_{min}. If we simply eliminate the tree after that depth, we still have the same average-case latency! However, we have increased the worst-case latency to $O(N/\log N)$, which may or may not be acceptable in practice. However, we have a significant savings in area—we save an additional $O(N)$ in circuit size, compensating for the $O(N)$ area overhead for adding the serial phase of the computation at depth d_{min}. The actual area necessary will depend on the exact circuit implementation used in either case.

3.6. Application to Binary Addition

The prefix computation can be used to construct a binary *kpg*-adder.[9] To perform binary addition at bit position i, the carry-in for that bit-position must be known. The carry-in computation can be formulated as a prefix computation as follows.

Suppose bit i of the two inputs are both zero. Then no matter what the carry-in is, the carry-out of the stage is zero—a *kill* (k). Similarly, if the two inputs are both one, the carry-out is always one—a *generate* (g). Otherwise, the stage *propagates* (p) the carry-in. To determine the carry-out of two adjacent stages, one can use the following $\otimes$ operation. The vertical column represents the *kpg* code for the least significant bit.

$\otimes$	k	p	g
k	k	k	g
p	k	p	g
g	k	g	g

Table 4.1. Prefix operator for kpg addition

Observe that the *kpg* code has the property that both k and g are right zeros of the prefix operator. Therefore, we can use the techniques discussed above to reduce the latency of binary addition. From the previous section, we observe that the average latency through such an adder is $O(\log\log N)$.

3.7. Area Optimality

We have designed a parallel prefix computation block which has $O(1)$ cycle time and $O(\log N)$ worst-case latency. The circuit has $O(N \log N)$ size. In this section, we show that we cannot do any better in the general case.

We assume that the prefix computation circuit has the following properties:

1. It can be used repeatedly;
2. It does not store information about its history, i.e., it cannot use information from any previous input to compute its next output;
3. Output y_k cannot be generated without knowledge of x_k.

Under these assumptions, we conclude:

Theorem 3.2.

Let $C(N)$ be a family of circuits that solve an N-input prefix problem, with $r(N)$ being the worst-case ratio of their latency and cycle time over all possible input values. Then the size of the circuits, $S(N)$, is $\theta(N \max(1, r(N)))$.

Proof: The circuit cannot have less than $\theta(N)$ size since it has N inputs and N outputs, and must store at least one bit per output.

Consider a consecutive sequence of inputs all of which have the worst-case ratio of latency to cycle time. Let the latency for the input be l, and the cycle time be τ. If the cycle time is τ, then after τ seconds, the circuit must be able to accept its next input. Since the latency is l, the circuit must have $\frac{l}{\tau}$ pending prefix computations internally. Since each prefix computation requires $\theta(N)$ size to store information for N different outputs, we conclude that the circuit must have $\theta(N\frac{l}{\tau})$ size. □

Corollary 3.3.

A full-throughput N-input parallel prefix computation circuit has $\theta(N \log N)$ size.

Proof: The worst-case latency of any parallel prefix computation circuit is $\theta(\log N)$. Since the cycle time is constant, $r(N) = \theta(\log N)$, concluding the proof. □

From these two observations, we conclude that all the circuits we presented to solve the prefix problem have asymptotically optimal circuit size. The unpipelined circuits have $O(N)$ circuit size, and the full-throughput circuits have $O(N \log N)$ circuit size.

3.8. Latency Optimality

Let V be the set of values that x_i might take. To analyze the delay through a prefix computation circuit, we partition V into two parts: a subset consisting of *propagate*-type values, and one consisting of non-propagate values. The set P of propagate-type values is the maximal set characterized by the property that $|V \otimes p| > 1$ for all $p \in P$, i.e., $x \otimes p$ depends on the value x, where $V \otimes x$ is the set $\{s \otimes x \mid s \in V\}$. In the case of a binary adder, the input p is the only input of propagate type (see Table 4.1).

Given an input vector $\mathbf{x} = (x_1, \ldots, x_n)$, a propagate sequence is a subvector $(x_i, x_{i+1}, \ldots, x_j)$ such that $x_i \otimes x_{i+1} \otimes \cdots \otimes x_j \in P$. We define $m(\mathbf{x})$ to be the length of the longest propagate sequence in $\mathbf{x}$. For example, $m(k, k, p, g, p, p) = 2$ since there are two consecutive p values in the vector.

Theorem 3.4.
The average-case latency through any prefix computation circuit is $\theta(\mathrm{E}[\log m(\mathbf{x})])$, *where* m *is defined as above.*

Proof: Given an input vector, let the longest propagate sequence in it be at positions i through j. This implies that the outputs at positions i through j must depend on the input at position i. Therefore, the information content in the input at position i must be communicated to $j - i + 1 = m(\mathbf{x})$ different output positions. This information cannot propagate faster than $\log m(\mathbf{x})$, concluding the proof. □

By Theorem 3.4, the prefix computation circuit we have designed has asymptotically optimal average-case latency. Note that the result does not depend on the input distribution.

Consider the case of binary addition. The argument used in the proof of Theorem 3.4 was based on an analysis of the input to output dependencies; this analysis holds no matter how the binary adder is constructed, and therefore the result also applies to binary addition. In particular, this implies that an adder constructed in this manner has the best possible asymptotic average-case latency characteristics for any input distribution.

If the set P is closed under $\otimes$ then inputs from P will result in long propagate sequences, slowing down the prefix computation. Note that $p, q \in P$ implies $p \otimes q \in P$ is quite a natural property for a prefix operator to have since it is associative. Suppose $x \otimes a$ depends on x and $x \otimes b$ depends on x. Then $x \otimes (a \otimes b) = (x \otimes a) \otimes b$. Since

$x \otimes a$ depends on x, it is natural to expect that $(x \otimes a) \otimes b$ would depend on x. The example in Table 4.2 shows that this is not true in general.

$\otimes$	0	1	2
0	0	0	0
1	0	0	1
2	0	1	2

Table 4.2. P may not be closed under $\otimes$

In this case, the elements 1 and 2 are contained in the set P for this operator. However, $1 \otimes 1 = 0 \notin P$. Since the set $\{2\}$ is closed under $\otimes$, the input can have long sequences of 2's in it, which will slow down the prefix computation. We formalize this observation below.

Let Q_1, ..., Q_q be maximal subsets of P that are closed under $\otimes$. Intuitively, the members of these Q-sets make the prefix computation slow since long sequences of values from a fixed set Q_i will result in large values of $m(\cdot)$. The operator in Table 4.2 has only one such Q-set, namely $\{2\}$. The fact that we may have more than one maximal Q-set is illustrated by Table 4.3, whose prefix operator has two such sets $\{1\}$ and $\{2\}$.

$\otimes$	0	1	2
0	0	0	0
1	0	1	0
2	0	0	2

Table 4.3. $\otimes$ may have more than one Q-set

By the definition of Q-sets, there exists at least one such set since the trivial set $\emptyset \subseteq P$ is closed under $\otimes$.

Lemma 3.5.

$|Q_1| = 0$ if and only if $x_1 \otimes \cdots \otimes x_{|P|+1} \notin P$ for all $x_i \in V$.

Proof: If the RHS holds, then clearly only the empty set is closed under $\otimes$. Assume that the RHS does not hold, i.e., $x_1 \otimes \cdots \otimes x_{|P|+1} \in P$ for some $x_i \in V$. Then, $x_1 \otimes \cdots \otimes x_j \in P$ for all j, $1 \leq j \leq |P|+1$. Since we have $|P|+1$ possible values for j, we are guaranteed that $x_1 \otimes \cdots \otimes x_a \otimes \cdots \otimes x_b = x_1 \otimes \cdots \otimes x_a$ for some $a < b$. Let $x = x_1 \otimes \cdots \otimes x_a$, and $y = x_{a+1} \otimes \cdots \otimes x_b$. This shows that $x \otimes y^k = x$ for all values $k \geq 0$. Therefore, $y^k \in P$ for all values $k \geq 1$, showing the existence of a set $\{y\} \subseteq P$ that is closed under $\otimes$, concluding the proof. □

If $|Q_1|$ is empty, then Lemma 3.5 shows that no matter what vector we pick from $V^{|P|+1}$, multiplying the elements from the vector results in an element that is not in P. If this is the case, then we can solve the prefix problem in constant time by splitting the input into blocks of size $|P|+1$ and solving the problem for each block independently. Indeed, Lemma 3.5 can be used to determine if this is the case since the RHS of the equivalence stated in Lemma 3.5 can be easily checked.

3.9. Related Work

Asynchronous adders were originally studied by Burks et al.[1] who showed that the average-case latency through a ripple-carry binary adder (assuming that the inputs were independently distributed and the zero-one probabilities were equal) was bounded by $\log_2 N$, where N is the number of bits being added. Winograd[27] showed that a lower bound on the worst-case time complexity for binary addition is $O(\log_2 N)$, where N is the number of bits in the input. The prefix problem and the formulation of binary addition as a prefix problem was proposed by Ladner and Fischer.[9] Gemmell and Harchol[5] present a method for adding two binary numbers "mostly correctly," with an error probability ϵ. They show lower bounds on the latency of such adders to be $O(\log\log(N/\epsilon))$. Our circuits always produce the correct answer with $O(\log\log N)$ latency. Gemmell and Harchol also claim an $O(\log N)$ lower bound on the average-case latency of binary addition in their abstract, which we have shown to be incorrect in this chapter by providing a construction for a $O(\log\log N)$ binary adder; closer inspection reveals that their lower bound only applies to "VRTC" (variable running time correct) circuits, showing that asynchronous circuits for addition have better latency characteristics than those constructed by their method.

Chapter 4.

PRECISE EXCEPTIONS

"I must go back and see after some executions I have ordered." —Lewis Carroll, *Alice in Wonderland*

The presence of precise exceptions in a processor leads to complications in its design. Recent processor architectures have sacrificed this requirement for performance reasons at the cost of software complexity. We present an implementation strategy for precise exceptions that does not block the instruction fetch when exceptions do not occur; the cost of the exception handling mechanism is only encountered when an exception occurs during execution—an infrequent event.

Ordinarily, a processor executes a sequence of instructions without interruption. Conceptually the instructions are executed one after another, with some instructions that modify the control flow. However, this stream of execution can be interrupted in two different ways: by interrupts—external asynchronous events that are generated by various I/O devices, and by exceptions.

Exceptions are used for a number of reasons. They are used to enforce protection between different address spaces so that a process that is running cannot access memory that belongs to another process. They are used to prevent a program from executing certain special instructions. They are used to begin the execution of special operating system subroutines (traps). They are used when some functionality is implemented partly in hardware and partly in software: hardware page tables (TLB), partial implementations of IEEE 754/854 floating-point arithmetic, etc.

When an exception or interrupt is encountered, a processor aborts the normal instruction sequence by jumping to a fixed address (or one of a fixed set of addresses) in memory. This point in memory contains a software routine, the exception handler, that services the exception or interrupt. The hardware is said to implement *precise exceptions* just when the state of the processor seen by the exception handler is the same as the state of the processor *before* execution of the instruction that caused the exception or interrupt was attempted. As a result, after the service routine has executed, we can restart execution of a program from the point where the exception occurred (if appropriate) without affecting program behavior.

The implementation of such an exception mechanism is complicated by the fact that a processor is typically heavily pipelined, and therefore even if a particular instruction has raised an exception, a number of instructions following it may have been partially executed. As a result, modern high-performance architectures such as the MIPS R8000, DEC Alpha, and Power-2 do not implement precise exceptions in hardware.

In this chapter we present a mechanism for the implementation of precise exceptions for asynchronous processors. An interesting feature of this mechanism is that it permits the presence of a data-dependent number of instructions in the main execution pipeline.

4.1. An Overview of a Processor

In this section we will provide a generic description of an asynchronous processor that does not have interrupts or exceptions. For simplicity, we assume that we have a "Harvard architecture"—i.e., the instruction and data memories are not synchronized.

A processor is comprised of a number of "units" (which is the traditional terminology for "process") that communicate with each other. A processor conceptually has a unit that generates the sequence of program counter values—the "IF" unit, a unit that decodes the instruction stream—the "DE" unit, a part that executes the decoded instructions—the "EX" units, and a place on the processor that stores state—the "RF" unit.

The instruction fetch *IF* generates a program counter value which is sent to the memory. The memory returns an instruction that is sent to the decode *DE*. This unit decodes the instruction and sends the appropriate control information to all the other units: the instruction to be executed is sent to the appropriate execution unit

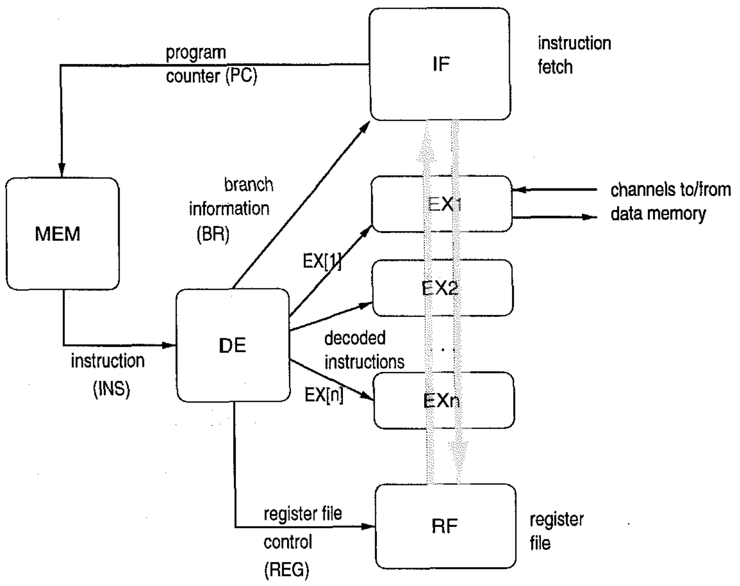

Figure 4.1. Information flow in a processor without exceptions.

EX_i; information about what state is needed and modified by the instruction is sent to the register file *RF*; information about control flow is sent to *IF*. The flow of information is shown in Figure 4.1. The sequential CHP description of the processor is given below:

```
PROC ≡
    *[    IF :   pc := "next pc";
        MEM :   i := imem[pc];
         DE :   id := decode(i);
       EXEC :   "read operands";
                "execute instruction";
                "write results"
    ]
```

When this CHP program is decomposed using standard techniques,[15] the different parts of the processor shown in Figure 4.1 can execute concurrently. [16] In particular, the *EXEC* is decomposed into a number of different execution units and a register file. Once the control information is dispatched to the execution units and the register file by *DE*, the instruction can execute and asynchronously complete execution. Since we have multiple execution units running concurrently, there can be a data-dependent number of instructions executing at any given time. The number of instructions executing in parallel is limited by data-dependencies between instructions, the number of communication channels between the register file and the execution units, and the number of execution units.

4.2. Implementing Precise Exceptions

The introduction of exceptions or external interrupts complicates the execution of instructions in a number of ways. When an instruction raises an exception, the exception must be detected and reported to the *IF*, since the processor must begin execution of the exception handler. In addition, the *RF* and data memory interface must be notified of the exception so that subsequent instructions do not modify the state of the processor until the exception handler begins execution.

The result of each instruction is modified so that it includes whether the instruction raised an exception. This exception bit is computed by the execution units. The simplest modification to *PROC* that includes a precise exception-handling mechanism is shown below:

```
$EPROC_0 \equiv$
        $e :=$ false;
      *[   IF :  $[\neg e \longrightarrow pc := "next\ pc" \ [] \ e \longrightarrow pc := "exception\ pc"]$;
          MEM :  $i := imem[pc]$;
           DE :  $id := decode(i)$;
         EXEC :  $"read\ operands"$;
                 $"execute\ instruction"$;
                 $e := "exception\ condition"$;
           WB :  $[\neg e \longrightarrow "write\ results"$
                 $[] \ e \longrightarrow "set\ exception\ flags,\ save\ pc"$
                 ]
       ]
```

A problem with this scheme is that the value of e computed by *EXEC* affects the next *pc* value, since it is used by *IF*. As a result, parallelizing this program would not introduce any concurrency between *IF* and *EXEC* because *IF* would have to wait for *EXEC* to complete before computing the next *pc*, and *EXEC* would have to wait for the next *pc* to be computed before it could receive the decoded instruction.

Since the case $e =$ **true** is rare, we would like to optimize the program so that we break the dependency between *IF* and *EXEC* when $e =$ **false**. To do so, we introduce a slack 1 channel *EX* that is used to notify *IF* of the presence of an exception (Note: the probe is the only construct in our language which can be used to guarantee the “execute eventually” semantics that we need for this mechanism, since the selection statement is unfair). *IF* will detect the presence of an exception by probing channel *EX*. A naive (and incorrect) modification of $EPROC_0$ that contains this transformation is shown below:

```
*[    IF :   [$\neg\overline{EX} \longrightarrow pc := "next\ pc"$
             | $\overline{EX} \longrightarrow pc := "exception\ pc", EX$
             ];
     MEM :   (...)
    EXEC :   (...)
      WB :   [$\neg e \longrightarrow "write\ results"$
             ▯$e \longrightarrow "set\ exception\ flags,\ save\ pc", EX$
             ]
 ]
```

Although this breaks the dependence between *IF* and *EXEC*, *EXEC* might execute instructions that were invalid—instructions not executed by $EPROC_0$—since the sequence of *pc* values might have changed. *EXEC* only executes invalid instructions after *EX* has been executed by *WB*, and before *EX* is executed by *IF*. We introduce variable *va* that is set to **true** when *EX* is executed by *IF* (*va* stands for valid-again; the variable signals the transition from invalid to valid instructions), and variable *valid* that is set to **false** when *EX* is executed by *WB*. This transformation is shown below:

```
     $valid\uparrow$;
*[    IF :   [$\neg\overline{EX} \longrightarrow va\downarrow, pc := "next\ pc"$
             | $\overline{EX} \longrightarrow va\uparrow, pc := "exception\ pc", EX$
             ];
     MEM :   (...)
    EXEC :   (...)
      WB :   [$\neg e \longrightarrow "write\ results", valid\uparrow$
             ▯$e \longrightarrow "set\ exception\ flags,\ save\ pc", valid\downarrow, EX$
             ]
 ]
```

The processor is executing invalid instructions whenever *valid* is **false** and *va* is **false**.

To eliminate any state change that might occur when the processor executes invalid instructions, we modify *WB* to a **skip** when $\neg valid \wedge \neg va$ is true. This is the only modification necessary, since all state changes are performed by *WB*. The resulting program is a correct implementation of the processor, and is shown below:

```
$EPROC_1 \equiv$
          $valid\uparrow$;
        *[    $IF$ :    [$\neg\overline{EX} \longrightarrow va\downarrow, pc := "next\ pc"$
                        | $\overline{EX} \longrightarrow va\uparrow, pc := "exception\ pc", EX$
                        ];
             $MEM$ :    $i := imem[pc]$;
              $DE$ :    $id := decode(i)$;
            $EXEC$ :    $"read\ operands"$;
                        $"execute\ instruction"$;
                        $e := "exception\ condition"$;
              $WB$ :    [$valid \vee va \longrightarrow$
                          [$\neg e \longrightarrow "write\ results", valid\uparrow$
                          ▯$e \longrightarrow "set\ exception\ flags,\ save\ pc",\ valid\downarrow,\ EX$
                          ]
                        ▯$\neg valid \wedge \neg va \longrightarrow$ skip
                        ]
         ]
```

The fact that exceptions are observed eventually is guaranteed by the progress condition on probes. Channel *EX* (initially empty) must have slack ≥ 1 for this program to be deadlock-free.

Another problem that needs to be resolved is that when exceptions do occur, the exception flags that need to be set are typically not stored in the execution unit that raised the exception. To avoid synchronization across execution units, we can transform $EPROC_1$ into a program in which the update of exception flags is performed by an ordinary instruction in the execution pipeline. This "fake exception instruction" (with a *pc* value that is determined by inverting the "next pc" computation) corresponds to the instruction which has *va* set to **true**. The information about exception flags is once again sent to the execution unit by a special channel which has slack ≥ 1. Therefore, when *va* is true, we are no longer executing an actual instruction; we only have to execute normally when *valid* is true. The result of these transformations is shown below:

```
$EPROC_2 \equiv$
          $valid\uparrow$;
    *[    $IF$ :     [$\neg\overline{EX} \longrightarrow va\downarrow, pc :=$ "next pc"
                     | $\overline{EX} \longrightarrow va\uparrow, pc :=$ "before exception pc", $EX$
                     ];
          $MEM$ :    $i := imem[pc]$;
          $DE$ :     $id := decode(i)$;
          $EXEC$ :   [$\neg va \longrightarrow$ "read operands";
                        "execute instruction";
                        $e :=$ "exception condition";
                     ▯$va \longrightarrow EINFO?(flags, epc)$; "set exception flags, save epc"
                     ];
          $WB$ :     [$valid \longrightarrow$
                       [$\neg e \longrightarrow$ "write results", $valid\uparrow$
                       ▯$e \longrightarrow EINFO!($"exception flags"$, pc)$, $EX$, $valid\downarrow$
                       ]
                     ▯$\neg valid \longrightarrow valid := va$
                     ]
    ]
```

4.2.1. Process Decomposition

When $EPROC_2$ is decomposed into a number of concurrent processes, we have to ensure that the sequence of va values and e values received by the part labeled WB (for writeback) is preserved. When the $EXEC$ is decomposed into a number of concurrent processes, the program order of the instructions is lost and we have a number of independent processes that will be communicating e-values to WB.

We keep track of the instruction order by introducing a queue which identifies which execution unit is executing the next instruction in program order. This queue is read by WB to determine which execution unit e-value is to be read next. This queue is also a convenient place to store the va bit. The queue is written by DE, since it is responsible for decoding instructions in program order. The pc value used by the writeback is also stored in a separate queue that is connected to IF (which is the process that computes pc values).

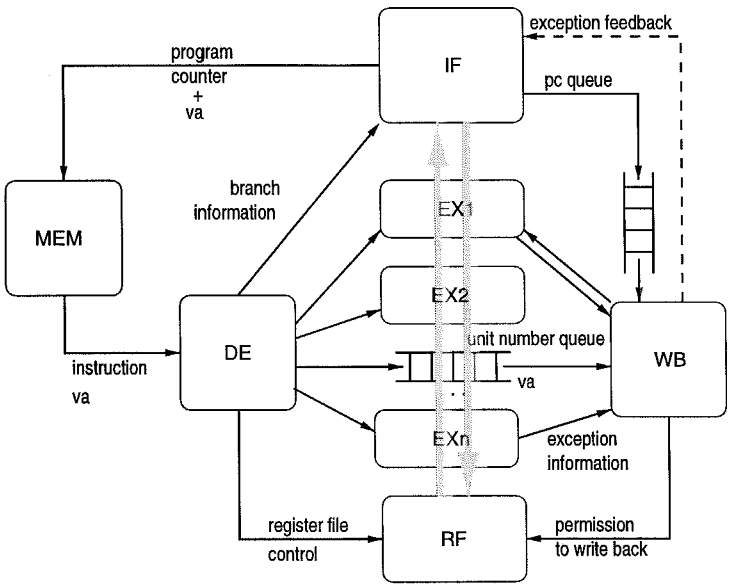

Figure 4.2. Information flow in a processor with exceptions.

When decomposed, the part of the processor where writes occur is distributed among a number of concurrent processes which need to know whether the writes are to be performed or not. Therefore, *WB* also needs information regarding where the next write is scheduled to take place; this information is sent to it by the queue connected to *DE*. The parts of the processor where writes occur are modified to read a channel from the writeback that informs them as to whether writes are permitted to occur or not. Figure 4.2 shows the modifications to the processor architecture.

4.2.2. Evaluating a Probe

The non-standard part of the program described above is the non-deterministic selection statement in *IF*. In this section we provide a circuit implementation for a process which can be used to implement this part of the exception mechanism. We can replace the program fragment *IF* by *IF'* by the introduction of a process that probes channel *EX*.

```
$IF'$ :  $E?x$;
      $[\neg x \longrightarrow va\downarrow, pc := "next\ \ pc"$
      $[] x \longrightarrow va\uparrow, pc := "before\ \ exception\ \ pc"$
      $]$
```

The process that contains the arbitrated selection statement is shown below.

$$*[[\overline{EX} \longrightarrow E!\textbf{true}, EX \quad | \neg\overline{EX} \longrightarrow E!\textbf{false}\ ]]$$

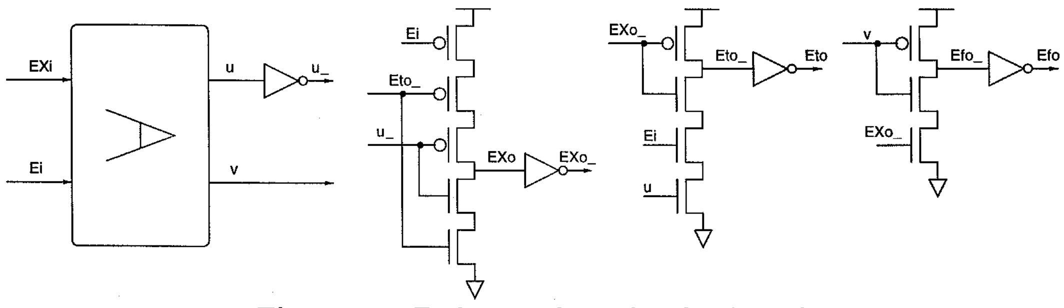

Figure 4.3. Evaluating the probe of a channel.

We now provide a circuit implementation for this particular process. Assuming that channels *EX* and *E* are both passive, we can write the following handshaking expansion:

```
$*[[\ EXi \longrightarrow [Ei]; Eto\uparrow; [\neg Ei]; EXo\uparrow; Eto\downarrow; [\neg EXi]; EXo\downarrow$
 $|\ Ei \longrightarrow Efo\uparrow; [\neg Ei]; Efo\downarrow$
$]]$
```

We have eliminated the check for $\neg EXi$ in the handshaking expansion for the second guarded command. The reason we can eliminate this check is that the hardware implementation of a two-way arbitrated selection statement is weakly fair. Therefore, if *EXi* is true, the first alternative in the selection will execute eventually. We introduce variables u and v to model the arbitration that is required by the above handshaking expansion.

```
$*[[\ EXi \longrightarrow u\uparrow; [u]; [Ei]; Eto\uparrow; [\neg Ei]; EXo\uparrow;$
                  $Eto\downarrow; [\neg EXi]; u\downarrow; [\neg u]; EXo\downarrow$
 $|\ Ei \longrightarrow v\uparrow; [v]; Efo\uparrow; [\neg Ei]; v\downarrow; [\neg v]; Efo\downarrow$
$]]$
```

We apply process factorization to obtain:

```
$*[[\ EXi \longrightarrow u\uparrow;\ [\neg EXi];\ u\downarrow$
 $|\ Ei \longrightarrow v\uparrow;\ [\neg Ei];\ v\downarrow$
$]]$
$\|$
$*[[\ u \longrightarrow [Ei]; Eto\uparrow; [\neg Ei]; EXo\uparrow; Eto\downarrow; [\neg u]; EXo\downarrow$
 $[]\ v \longrightarrow Efo\uparrow; [\neg v]; Efo\downarrow$
$]]$
```

The first process shown in the decomposition above is an arbiter between *EXi* and *Ei*; the compilation of the second process results in the bubble-reshuffled production rules shown below.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
Reset\_ \wedge u \wedge EXo\_ \wedge Ei & \to & Eto\_\!\downarrow & \neg u\_ \wedge \neg Eto\_ \wedge \neg Ei & \to & EXo\!\uparrow \\
\neg Reset\_ \vee \neg EXo\_ & \to & Eto\_\!\uparrow & u\_ \wedge Eto\_ & \to & EXo\!\downarrow \\
Eto\_ & \to & Eto\!\downarrow & EXo & \to & EXo\_\!\downarrow \\
\neg Eto\_ & \to & Eto\!\uparrow & \neg EXo & \to & EXo\_\!\uparrow
\end{array}
$$

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
Reset\_ \wedge EXo\_ \wedge v & \to & Efo\_\!\downarrow & u & \to & u\_\!\downarrow \\
\neg Reset\_ \vee \neg v & \to & Efo\_\!\uparrow & \neg u & \to & u\_\!\uparrow \\
Efo\_ & \to & Efo\!\downarrow & & & \\
\neg Efo\_ & \to & Efo\!\uparrow & & &
\end{array}
$$

The CMOS implementation is shown in Figure 4.3. For clarity, the reset transistors and staticizers are not shown.

4.2.3. Slack Elasticity

We observe that the mechanism for precise exceptions just discussed is slack elastic because it exhibits the property of maximal non-determinism; the only selection statement we have introduced that contains probed communication actions is the one containing $\overline{EX}$ in *IF*, and the correctness of the computation does not depend on the alternative that is chosen for execution.

4.3. Unpipelining the Processor

The scheme outlined in the previous section is used in the MiniMIPS, a stripped-down version of a MIPS R3000 microprocessor.[17] Observe that when the instruction that has the valid-again bit set to **true** is being executed, the instructions in the exception handler are already being processed by the *DE* unit and *MEM* unit. However, it is possible that the state changes introduced by the valid-again instruction modify the effect of the *MEM* and *DE* unit. For instance, the exception handler typically begins execution in kernel mode. The valid-again instruction would modify the mode to kernel mode; however, this affects the accessible address space and the instructions that are allowed to be executed.

To illustrate the problem, we decompose $EPROC_2$ into two processes. For the rest of the discussion, all variables shown will be local variables unless otherwise specified. We obtain the program shown below by process decomposition:[15]

```
$IF_2 \equiv$
        *[    $IF$ :    [$\neg\overline{EX} \longrightarrow va\downarrow, pc :=$ "next pc"
                        | $\overline{EX} \longrightarrow va\uparrow, pc :=$ "before exception pc", $EX$
                        ];
             $MEM$ :    $i := imem[pc]$;
                        $I!(i, pc, va)$;
                        $J?control$
         ]

$EXEC_2 \equiv$
          $valid\uparrow$;
        *[    $DE$ :    $I?(i, pc, va)$;
                       $id := decode(i)$;
            $EXEC$ :    [$\neg va \longrightarrow$ "read operands";
                            "execute instruction";
                            $e :=$ "exception condition";
                        [] $va \longrightarrow EINFO?(flags, epc)$; "set exception flags, save epc"
                        ];
              $WB$ :    [$valid \longrightarrow$
                          [$\neg e \longrightarrow$ "write results", $valid\uparrow$
                          [] $e \longrightarrow EINFO!($"exception flags"$, pc)$, $EX$, $valid\downarrow$
                          ]
                        [] $\neg valid \longrightarrow valid := va$
                        ]
                        $J!$"control information"
         ]
```

The channels I and J are used to communicate non-shared variables between IF_2 and $EXEC_2$. Since we are assuming that the operation $i := imem[pc]$ is complicated by a mechanism that is affected by the state change on exceptions, we assume the existence of a set of shared variables that are modified by the operation "set exception flags." To preserve correctness, we are forced to ensure that IF_2 cannot overlap its execution with the rest of the processor by making $J!$ the last action in $EXEC_2$—a significant performance penalty.

We would like to complete the $J!$ communication action just after the instruction has been decoded, because that is when the information that it sent to IF_2 is computed. This would permit most of the computation in $EXEC_2$ to overlap with IF_2. However, it would violate mutual exclusion between reading and writing the shared variables mentioned above when an exception occurs.

Observe that we *know* when mutual exclusion might be violated—the bit *va* is true if this might occur! We therefore introduce a new synchronization channel EX' that is used to block *IF* on this condition. We can now complete $J!$ as soon as the instruction has been decoded; the case of shared variables that arises on the rare occasion when exceptions occur is handled by the new synchronization introduced. The final program is shown below. Since the modifications we have introduced are minor, we use (...) to indicate the parts that are unchanged.

```
$IF_2 \equiv$
        *[   $IF$ :   (...)
           $MEM$ :   (...)
                     $J?control;$
                     $[va \longrightarrow EX' \;[]\; \neg va \longrightarrow \textbf{skip}]$
         ]

$EXEC_2 \equiv$
          $valid\uparrow;$
        *[   $DE$ :   $I?(i, pc, va);$
                     $id := decode(i);$
                     $J!''control\ \ information''$
          $EXEC$ :   (...)
             $WB$ :   (...)
                     $[va \longrightarrow EX' [] \neg va \longrightarrow \textbf{skip}]$
         ]
```

Observe that when *va* is true, we effectively unpipeline the execution of instructions since all preceding instructions are forced to complete before the first instruction of the exception handler is dispatched.

4.4. An Optimization

In most processors, a large number of instructions are guaranteed to terminate normally. When all instructions being executed by one execution unit are guaranteed to terminate normally, we can eliminate the communication between the execution unit and the writeback. This optimization permits the writeback to process an instruction without waiting for any information from some execution units. This optimization was used in the asynchronous MiniMIPS processor where the function block and shifter never raise exceptions.[17]

When we can quickly (relative to the time taken to execute the instruction completely) determine that an instruction will not raise an exception, reporting this value to the writeback can improve performance of the processor. This is especially important for instructions which have high execution latency, such as those involved in floating-point arithmetic. The MiniMIPS executes a number of different arithmetic instructions in the adder unit. This unit can raise exceptions in rare cases (instruction traces show that the ratio of instructions that raise exceptions to those that do not is less than 10^{-4}). The unit was optimized so that the latency of exception reporting in the common cases was 40% of the worst-case latency.

4.5. Related Work

In synchronous processors, the clock globally synchronizes all actions, and therefore exception detection is implicitly synchronized with fetching instructions from memory. As a result, synchronous processors implement precise exceptions by allowing a deterministic number of instructions to execute before the exception status of an instruction is checked. The absence of a global clock allows us to break this synchronization.

The AMULET is a self-timed clone of the ARM processor.[4] However, it does not have multiple execution units which simplifies the design of an exception handling mechanism. "Fred" is an asynchronous processor with multiple execution units.[20] This processor does not implement precise exceptions.

Chapter 5.

The Branch Processor

"Sometimes I think the only universal in the computing field is the fetch-execute cycle." —Alan J. Perlis

We present a novel processor architecture which uses two independent instruction streams: one for the main processor, which consists of the instructions that perform the actual computation, and one for the branch processor, which determines the sequence of program counter values used to fetch instructions for the main processor. The two instruction streams only synchronize when necessary—in the case when the direction of a branch is known only at runtime.

A major bottleneck in the execution of instructions in modern processors is the process of computing program counters to determine which instruction to execute next, followed by fetching the specified instructions from memory. In traditional architectures, each instruction contains information about the sequence of program counters that constitute the program. As a result, one cannot compute which program counter is to be generated next without examining the preceding instruction—since it might be a branch. Traditionally, branch delay slots are introduced into the design to alleviate this problem. At the hardware level, aggressive branch prediction techniques are used to guess which instruction will be fetched next. When prediction fails, the hardware has to cancel the result of speculative execution.

Branches in programs correspond to subroutine calls, loops, and **if** statements. In the cases of fixed length loops and subroutine calls, we know how the branches behave when the program is compiled. In this chapter we present a processor architecture that eliminates branches in these cases by providing more information to the processor. The architecture is inspired by an asynchronous design style; however, it can be implemented using a synchronous design style as well.

5.1. Existing Instruction Sets

Consider a traditional instruction set such as the one used by the MIPS R3000. For simplicity of exposition, we will assume that the processor does not have a branch delay slot. The processor has a number of different instruction types, and different instructions are used to compute different functions. However, each instruction implicitly encodes control flow information. An instruction such as

```
pc:   addu r1,r2,r3
```

implicitly encodes that the next program counter is pc+4. An instruction such as

```
pc:   bne r1,r3,L
```

encodes that the next program counter is either pc+4, or L, depending on whether or not registers r1 and r3 are equal.

A processor computes the sequence of program counter values. However, from the point of view of the density of the encoding, existing instruction sets encode this information very inefficiently, since most of the time one has to examine an instruction simply to determine that the next instruction to be executed is at pc+4. Consider a simple FORTRAN 77 loop:

```
         do 10 i = 1, 100
           c(i) = a(i)+b(i)
10       continue
```

This loop would be compiled into a number of instructions, with a single branch at the end of the loop. Most of the instructions would simply correspond to "pc:=pc+4" as far as the sequence of program counters is concerned. We can statically determine that the sequence of instructions that implement the body of the loop will be executed a hundred times; however, this information is not encoded in the instruction set. It is this deficiency in instruction sets that we will remedy in this chapter.

5.2. A New Approach

We introduce a completely separate and independent instruction sequence that encodes the sequence of program counter values. A program is therefore compiled into

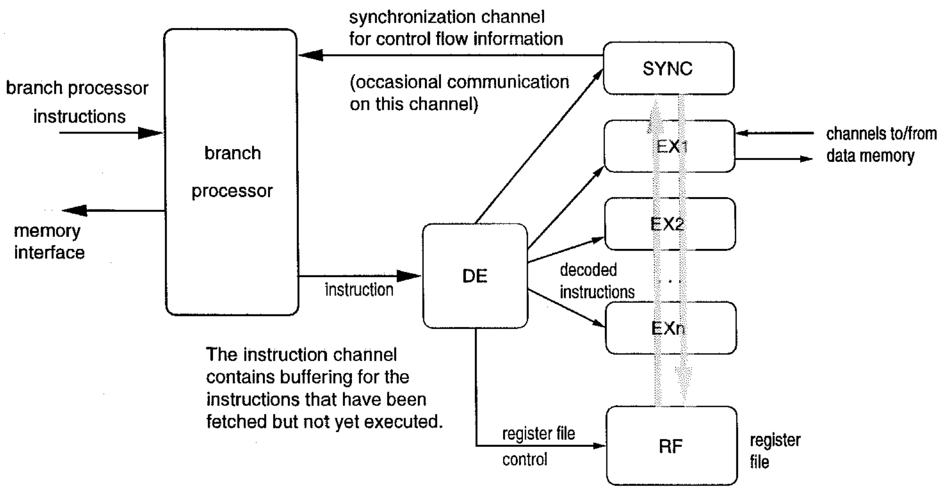

Figure 5.1. Processor shown in Chapter 4 modified to the new architecture.

two instruction streams: one which determines the computations to be performed, and one which determines the flow of control.

The processor that executes the first instruction stream corresponds to a traditional processor, and we will refer to it as the data processor. The second instruction stream will be executed on a separate processor which we call the *branch processor*. Conceptually, we have a sequence of instructions that computes the sequence of program counter values. The program counter sequence is used to fetch instructions from memory. The data processor receives this sequence of instructions. The branch processor also needs feedback from the data processor when executing code that has conditional branches. This feedback channel occasionally synchronizes the two processors. Figure 5.1 shows how the processor discussed in Chapter 4 would be modified to have such an architecture. Contrast it with the processor shown in Figure 4.1, which corresponds to a traditional instruction set.

In Chapter 4, we broke the dependence of the instruction fetch on exception information by communicating on the exception channel only when exceptions occur. In this chapter, we break the dependence of the instruction fetch on the instructions being executed in the data processor except when the data values computed in the data processor affect control flow. We expect performance improvements from the following factors: latency tolerance for cache misses, parallel execution of branch instructions and ordinary instructions, and early knowledge of the sequence of program counter values.

5.3. A Sample Instruction Set

In this section we present a sample instruction set for the branch processor. Since control flow in programs normally follows a call/return pattern, we include a hardware stack in the branch processor that is used for storing program counter values.

There are times when control flow information is only available at run time; to be able to execute programs in which this is the case, we introduce a single instruction in the main data processor called `send!`. This instruction sends a data value from the data processor to the branch processor via the synchronization channel. It must be matched by a branch processor instruction that reads the data from this channel; instructions which read values from this channel have a "?" appended to them.

In what follows, *addr* refers to the address of instructions to be executed on the data processor, and *braddr* refers to addresses for branch processor instructions.

Block Fetch. Block fetch instructions are introduced to compress control flow information within basic blocks. Instruction

`fetch` *addr*,*N*

means "fetch and execute *N* instructions that begin at address *addr*." If we know statically that we have *N* instructions that will be executed sequentially, we can compress this control-flow information using this single instruction.

This instruction can be used to implement straight-line microcode. A sequential stream of instructions that implements a complex task can be invoked without increasing code size significantly by using a single `fetch` instruction. Using this instruction can result in a smaller instruction cache footprint for a program in the case when common code can be shared among different parts of the program.

Loops. To permit simple looping constructs to be implemented without significant overhead, we introduce the following two instructions:

`push` *baddr*, *N*
`dec`

The `push` instruction stores the pair $\langle baddr, N\rangle$ on the hardware stack. Branch processor execution continues with the next instruction.

`dec` examines the pair $\langle baddr, N\rangle$ stored on the top of the stack, and decrements N. If the result is zero (or negative), stack is popped; otherwise, the branch processor begins execution at address *baddr*. For example, the code corresponding to a loop that executes a sequence of 15 instructions 10 times would be:

```
   push A, 10;
A: fetch addr, 15;
   dec
```

The number of iterations in a loop is not always known at compile time. To permit the execution of loops with iteration counts determined at run time, we introduce the following instruction:

`pushN?` *baddr*

This instruction receives the next data value from the synchronization channel and uses it as the loop count N (as in the normal `push` instruction); other than that it behaves like a `push` instruction.

When breaking out of a loop, the hardware stack still has state information in it which needs to be destroyed. The pop instruction explicitly pops the top of the hardware stack.

Function Calls. Function calls are implemented with the `call` instruction. `call` *baddr* pushes $\langle nextpc, 1\rangle$ onto the branch processor stack (*nextpc* is the program counter address immediately following the `call`) and transfers control to *baddr*. Returning from a function is implemented by a `ret` instruction, that jumps to the address on the top of the stack and pops the stack.

To be able to efficiently execute a function call to an address determined at run time (this occurs when executing a function determined by looking at a function pointer stored in a table, or in the case of dynamic dispatch of methods in object-oriented languages), we introduce the `call?` instruction. This instruction reads the address to branch to from the synchronization channel, and otherwise behaves like a `call`.

Data-dependent Control Flow. The push and pop instructions can be used to implement control flow in loops. To handle arbitrary branches, we introduce goto instructions of two flavors:

`goto` *baddr*
`goto?`

The first instruction unconditionally changes the branch processor execution address to *baddr*. The second instruction reads the address to branch to from the synchronization channel.

Instruction	Stall	Purpose
`fetch` *addr*, *N*	n	fetch and execute block of instructions
`push` *baddr*, *N*	n	push loop counter
`dec`	n	decrement/pop loop counter
`pushN?` *baddr*	y	push loop counter, value unknown at compile time
`pop`	n	break out of loop
`call` *baddr*	n	function call
`call?`	y	function call, target unknown at compile time
`ret`	n	return from function call
`goto` *baddr*	n	arbitrary control flow
`goto?`	y	goto with target unknown at compile time
`if?` *baddr*	y	conditional branches
`fetch?` *addr*, *N*	y	block predication
`send!` *data*	y	data processor communication

Table 5.1. Instruction-set summary.

When control flow depends on computation in the data processor, the synchronization channel is used to determine the direction of the branch. The `if?` instruction is used for this purpose.

`if?` *baddr*

The instruction reads a value from the synchronization channel and continues execution at address *baddr* if the value received is non-negative. Otherwise, execution continues with the next branch processor instruction.

Predicated Execution. The instruction set for the branch processor will improve the performance of execution only if matching `send!` are executed early in the data processor. Programs containing short sequences of instructions interspersed with conditional branches that depend on the computation just performed would not be executed very efficiently. However, in such cases, predicated execution—executing instructions conditionally—could be used to improve performance.[12] This approach has been successfully used to efficiently execute code with frequently occurring branches.

We provide a simple mechanism for predicating a block of instructions. The instruction `fetch?` *addr*,*N* is used for this purpose. If the value received from the data processor is non-negative, then the block of *N* instructions stored at address *addr* are executed; otherwise, the instruction behaves like a nop.

5.4. Sample Branch Processor Code

Table 5.1 has a summary of the new instructions we have introduced. We now provide examples showing how code would be generated for the branch processor.

Example. Consider the following FORTRAN program fragment:

```
      do 10 i = 1, 100
        c(i) = a(i)+b(i)
10    continue
```

Compiling this piece of code using `f2c` and the GNU C compiler for an R3000 processor results in the following assembly code.

```
E: r2:=1; i:=r2;
   r8:=c-4; r7:=a-4; r6:=b-4;
L: r2:=i;
   r5:=r2+1; r2:=r2*4;
   r3:=r2+r7; r4:=r2+r6;
   r3:=mem[r3]; r4:=mem[r4];
   r2:=r2+r8;
   i:=r5;
   r5:=(r5<101);
   r3:=r3+r4;
   mem[r2]:=r3;
   if r5 goto L
```

In a branch processor architecture, the underlined instructions shown above would be deleted. In addition, the following branch processor code would be generated:

```
     fetch E, 5;
     push L1,100;
L1:  fetch L, 11;
     dec
```

In this example, the branch processor does not synchronize with the data processor because the control flow can be determined when the program is compiled. ✳

Example. Consider the same program with a modification that permits the program to exit the loop early.

```
      do 10 i = 1, 100
        c(i) = a(i)+b(i)
        if (c(i) .ge.  0) goto 11
10    continue
11    ...
```

The compiled version of this program is shown below.

```
E: r2:=1; i:=r2;
   r8:=c-4; r7:=a-4; r6:=b-4;
L: r5:=i;
   r3:=r5*4;
   r2:=r3+r7; r4:=r3+r6;
   r2:=mem[r2]; r4:=mem[r4];
   r3:=r3+r8;
   r2:=r2+r4;
   mem[r3]:=r2;
   if r2>=0 goto M;      ⇒ send! r2
P: r2:=r5+1;
   i:=r2;
   r2:=(r2<101);
   if r2 goto L
M: ...
```

In a branch processor architecture, the underlined instructions would be deleted, and in one case replaced by the **send!** instruction shown. The additional branch processor code would be:

```
     fetch E, 5;
     push L1, 100;
L1:  fetch L, 10;
     if? B;
     fetch P, 2;
     dec;
     push L1, 1;
 B:  pop
     ...
```

An examination of the code generated reveals that the **send!** r2 can be reordered with two instructions to obtain:

```
r2:=mem[r2]; r4:=mem[r4];
r2:=r2+r4; send! r2;
r3:=r3+r8;
mem[r3]:=r2
...
```

This transformation would improve the performance because the **send!** action occurred earlier. ⁎

5.5. Deadlock, Exceptions, and Context Switching

The state in the branch processor architecture is distributed, since we have two streams of instructions that are being executed concurrently. In addition, we have instructions that synchronize the branch processor and data processor. In this section we examine some of the consequences of such an architecture, presenting solutions to the new issues raised in such an architecture.

5.5.1. Deadlock

Since our architecture includes explicit instructions that synchronize the data processor and branch processor, incorrect code could deadlock the hardware. To avoid this problem, we must be able to detect the occurrence of deadlock and correct the problem.

Deadlock can occur in two different cases in the branch processor architecture.

- A receive on the synchronization channel is blocked because there is no matching send and the channel is empty;
- A send on the synchronization channel is blocked because the channel is full and there is no matching receive.

Every `send!` instruction must be `fetch`ed before the corresponding receive is executed in the branch processor. Therefore, the first case can only be caused by an incorrect program. This possibility can be prevented by using a correct compiler.

The second case could occur if multiple sends have been dispatched in advance, causing the synchronization channel to become full before any receives could be executed. This case could also be prevented by a compiler. The compiler must keep track of the number of outstanding `send!` operations at any point in the program, and ensure that the number of pending send operations does not exceed the hardware limit.

Although both cases of deadlock can be prevented using appropriate compilation techniques, we might want to be able to execute arbitrary programs on the hardware without causing the processor to deadlock. We discuss some deadlock-detection techniques below.

Deadlock can be detected by using a timing assumption or by running a deadlock detection algorithm. Simple timing assumptions include assuming that the processor has deadlocked if instructions have not been decoded for a long interval—say a microsecond. We could also execute a simple termination detection algorithm to detect deadlock.[3] In the latter case, we only have to involve the two ends of the synchronization channel in the termination detection algorithm along with counters to detect that there are no data values in transit from the branch processor to the data processor.

If a receive action is blocked forever, this implies that the code being executed on the branch processor is erroneous. In this case, we must be able to begin execution of the exception handler. If a `send!` action is blocked forever, this could imply that the code generated by the compiler is erroneous; however, if the compiler can predetermine the sequence of `send!` actions, the processor might deadlock because the synchronization channel is full. In this case, we should gracefully recover by permitting the program to continue execution while draining the values stored in the synchronization channel.

To permit program execution in the presence of blocked `send!` instructions, we must be able to save (and restore) the values stored in the synchronization channel to memory. Therefore, we must include a mechanism that will memory-map the synchronization channel, and treat the hardware queue as an optimized implementation of this queue. With such an implementation, a blocked `send!` action will cause execution to fail only when a process exhausts the virtual memory on the system (or, alternatively, exceeds its resource limits).

5.5.2. Exceptions and Context Switching

The processor architecture just proposed has state stored in both the data processor and the branch processor. The processor must include the capability of storing the entire state to memory. The state of the data processor can be saved to and restored from memory in the same way as in traditional processors. The state of the branch processor is stored in the contents of the branch processor stack and the contents of the synchronization channel between the data processor and branch processor. The hardware stack as well as the synchronization channel is be memory-mapped; therefore, we can save and restore the state of these parts of the branch processor using load and store instructions from the data processor. As we have a mechanism for saving and restoring the state of the processor, we can implement context switching.

Exceptions can be handled using a mechanism based on the one presented in Chapter 4. The `send!` instructions must be treated as instructions that modify the state of the processor. In addition, if an exception is encountered in the middle of a block fetch instruction, we must be able to restore execution from the middle of the block. This implies that the branch processor should keep track of pending block fetch instructions so that they can be restarted after an exception is handled.

Exceptions that occur in the branch processor itself (such as address translation errors or stack underflows) can be handled by sending them to the data processor with a special bit set indicating a branch processor exception. The instruction will be executed as a **nop** in the data processor, and raise an exception in the usual way. Making the writeback unit in the data processor handle branch processor exceptions ensures that exceptions are handled in program order.

5.6. Performance Comparison

To be able to provide some intuition as to when the branch processor performs well, we begin by providing a high-level description for the branch processor instruction set shown in the previous section. The interaction between the branch processor and the data processor occurs via two channels: PC, the channel on which program counter values are sent to the data processor, and $SYNC$, the channel used to read data values from the data processor.

The program for the branch processor is shown below. Variable *bpc* is the program counter for branch processor instructions, and S is a stack. A stack element has an *addr* field and an N field. We use three stack operations: $Top(S)$ is the top element of stack S; $Push(S, addr, N)$ pushes the pair $\langle addr, N\rangle$ onto the stack and returns a new stack; $Pop(S)$ returns deletes the top element of stack S and returns the new stack. We have omitted overflow and underflow detection from the program for the sake of clarity.

```
$bpc, S := init\_bpc, \epsilon$;
*[ $(i, addr, N), bpc := bmem[bpc],\ \ bpc + 1$;
   [$recv(i) \longrightarrow SYNC?x$ ▯ $\neg recv(i) \longrightarrow$ skip];
   [fetch($i$) $\longrightarrow$  *[ $N > 0 \longrightarrow PC!addr;\ \ addr, N := addr + 1, N - 1$ ]
   ▯push($i$) $\longrightarrow$  $S := Push(S, addr, N)$
   ▯dec($i$) $\longrightarrow$ [ $Top(S).N > 1 \longrightarrow bpc := Top(S).addr$;
                                   $Top(S).N := Top(S).N - 1$
                 ▯ $Top(S).N \leq 1 \longrightarrow S := Pop(S)$
                 ]
   ▯pushN?($i$) $\longrightarrow S := Push(S, addr, x)$
   ▯pop($i$) $\longrightarrow$  $S := Pop(S)$
   ▯call($i$) $\longrightarrow$  $S := Push(S, bpc, 1);\ \ bpc := addr$
```

```
        []call?(i) $\longrightarrow$  $S := Push(S, bpc, 1)$;  $bpc := x$
        []ret(i) $\longrightarrow$ $bpc := Top(S).addr$; $S := Pop(S)$
        []goto(i) $\longrightarrow$  $bpc := addr$
        []goto?(i) $\longrightarrow$  $bpc := x$
        []if?(i) $\longrightarrow$  [ $x \geq 0 \longrightarrow bpc := addr$ [] $x < 0 \longrightarrow$ skip ]
        []fetch?(i) $\longrightarrow$  [ $x \geq 0 \longrightarrow$
                          *[ $N > 0 \longrightarrow PC!addr$;  $addr, N := addr + 1, N - 1$ ]
                        [] $x < 0 \longrightarrow$ skip
                        ]
        []else $\longrightarrow$ skip
        ]
    ]
```

Since program counter values are computed by the branch processor, the data processor simply reads the *PC* channel to determine which instruction should be executed next. The high-level CHP for the data processor is shown below.

```
*[      IF :  $PC?pc$;
       MEM :  $i := imem[pc]$;
        DE :  $id := decode(i)$;
      EXEC :  "read operands";
              [send!(i) $\longrightarrow$ $SYNC!$"data"
              []else $\longrightarrow$" execute instruction"; "write results"
              ]
]
```

5.6.1. Program-Counter Computation

The branch processor can be compared to the instruction fetch in a standard processor. A simplified version of the instruction fetch for the asynchronous MiniMIPS is shown below. The channel *SYNC* corresponds to the channel from the core of the processor that is used to communicate register values and immediate values to the instruction fetch; we have introduced an additional *COND* channel on which condition codes for branches are sent to the instruction fetch.

```
    $PC!init\_pc$;  $pc := init\_pc$
    *[ $I?i; pc := pc + 1$
       [$i =$" nextpc" $\longrightarrow$ skip
```

```
        $[]i =" jump" \longrightarrow SYNC?x; pc := x$
        $[]i =" branch" \longrightarrow SYNC?x; COND?c;$
                $[c \longrightarrow pc := pc + x \ []\neg c \longrightarrow$ skip ]
        ];
        $PC!pc$
    ]
```

The branch processor can compute program counter values earlier than this simple instruction fetch because we have eliminated the communication $I?i$ which synchronizes the instruction fetch with the rest of the data processor on every instruction. Instead, the branch processor only synchronizes with the data processor when necessary. In the example of a simple loop, we eliminate all synchronization, permitting the branch processor to fetch instructions without any feedback from the data processor.

The branch processor program is more complex than the simple instruction fetch because it has more instructions to decode. This decoding overhead is quite small when compared to the overhead in accessing branch processor memory by the "$(i, addr, N) := bmem[bpc]$" statement. Accessing a large on-chip cache has a latency that is approximately equal to the cycle time τ of the processor. This is the additional overhead we encounter when using a branch processor.

The slowest possible execution of the branch processor architecture corresponds to the case when the last $PC!$ communication in the branch processor fetched a `send!` instruction, and the next branch processor instruction is either `pushN?`, `call?`, `goto?`, `if?`, or `fetch?`. In this case, the branch processor waits for the `send!` instruction to be fetched, decoded, and executed. Let the time taken to fetch, decode, and execute the `send!` instruction be τ_0. We analyze the branch processor overhead for each potentially slow instruction.

- `pushN?`. Since the value of bpc is not data-dependent on the value received on $SYNC$, the branch processor can continue execution without actually having to wait for the data on $SYNC$ to arrive.
- `call?` and `goto?`. The branch processor waits for the data on $SYNC$ to arrive before it can fetch the next branch processor instruction. The next data processor instruction has an additional data processor latency of $\tau_0 + \tau$ seconds.
- `if?`. If the value received on $SYNC$ is negative, the stall is τ_0 seconds because the next branch processor instruction can be speculatively read from branch processor memory. If the value received is non-negative, the branch processor

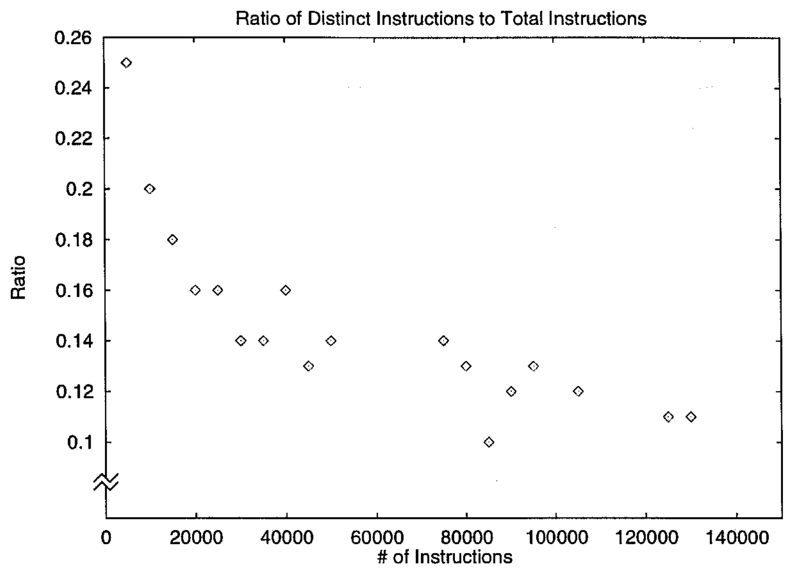

Figure 5.2. Ratio of distinct instructions to total instructions.

will have to read a new value from branch processor memory incurring an additional data processor latency of $\tau_0 + \tau$ seconds.

- `fetch?`. If the value received on *SYNC* is negative, the stall is τ_0 seconds. If the value received on *SYNC* is positive, the stall is τ_0 seconds because the next program counter values are available immediately.

To summarize, we expect that in the worst case the branch processor stalls for $\tau_0 + \tau$ seconds for branches that are taken, `goto?` and `call?` instructions, and τ_0 seconds for branches that are not taken and `fetch?` instructions.

In a non-speculative traditional microprocessor, the latency of fetching the branch instruction and executing it (which is about the same as τ_0) is typically avoided by the introduction of $\lceil \tau_0/\tau \rceil$ branch delay slots. If we directly translate a standard instruction set to branch processor code by replacing branches by `send!` instructions, we observe that each `send!` instruction will be followed by $\lceil \tau_0/\tau \rceil$ instructions that correspond to the branch delay slot. Therefore, the only additional stall the branch processor encounters is τ, which would be completely hidden if the original architecture had an additional branch delay slot.

5.6.2. Memory Access

We have introduced an additional memory read for branch processor instructions. This memory read is unsynchronized with the memory for data processor instructions

Cache Size (words)	Original Instructions	Unique Instructions
1024	0.0%	2.3%
2048	0.7%	2.3%
4096	2.3%	57.1%
8192	2.3%	78.5%
16384	50.9%	99.4%

Table 5.2. Percentage of programs with 100% cache hits.

or the data memory. However, most modern processors have a single off-chip memory. Therefore, we may have increased the instruction memory bandwidth requirements to off-chip memory.

However, data processor instructions no longer contain information about which instruction has to be executed next. Therefore, common code can be shared without any code replication. All that needs to be replicated is the branch processor `fetch` instruction for the block of shared code. Therefore, the branch processor could improve instruction cache performance by reducing cache misses in the instruction cache.

To maximize instruction sharing (and, incidentally, maximize branch processor code size), each unique data processor opcode would be stored once. This implies that an upper bound on the number of instructions required to be stored in the instruction cache is given by the number of distinct instructions in the program.

We collected instruction count statistics for 267 executables that were compiled using the GNU C compiler for an R3000-based DECstation. Figure 5.2 shows the ratio of the number of distinct instruction opcodes to the total number of instruction opcodes in executables of varying sizes. The figure illustrates that the number of distinct opcodes grows at a rate that is less than linear in the size of the executable.

Table 5.2 shows the percentage of programs that would completely fit in an instruction cache depending on whether we count total instructions or the number of unique instructions in the program. In a branch processor architecture, most programs would fit in a typical instruction cache (8K words). Therefore, we would significantly reduce the number of instruction cache misses in the data processor. At the same time, we would increase cache misses for the branch processor. We can bound the number of cache misses for the branch processor by the number of cache misses for the original instruction set, since each ordinary instruction would be translated into at most one branch processor instruction.

Therefore, the additional memory bandwidth requirements for a branch processor can be reduced significantly by sharing instructions from the data processor—at a performance cost. This conservative analysis shows that introducing a branch processor will not have a large impact on the instruction memory bandwidth required by the processor.

5.7. Speculation

The new architecture specified here can be used in conjunction with existing techniques for addressing performance problems due to control-flow dependencies. The techniques we mention here are branch prediction and prefetching. Both these techniques attempt to improve performance by predicting what the program will execute.

Incorporating branch prediction into this architecture corresponds to guessing the value being sent on the feedback channel for `if?` instructions. Since simple loops no longer contribute branch instructions, the effectiveness of branch prediction will be decreased because the cases which can be easily predicted (loops) are no longer present.

Prefetch instructions attempt to hide the latency of cache misses by dispatching reads to the caches before the data value is actually needed. These prefetch instructions can be inserted into the instruction stream of both the branch processor (for instruction cache prefetches) and the data processor (for data cache prefetches).

Instructions that support software-controlled speculation can be introduced to improve the performance of the branch processor architecture. The instruction

`sfetch` *addr*, *N*

means "fetch and speculatively execute *N* instructions that begin at address *addr*." These instructions are fetched from memory and dispatched to the data processor. The `commit` instruction informs the data processor if the last speculatively executed block should be permitted to modify the state of the processor. Therefore, the sequence "`sfetch` *addr*, *N*; `commit true`" is equivalent to "`fetch` *addr*, *N*." The sequence "`sfetch` *addr*, *N*; `commit false`" is equivalent to a **skip**.

Speculative execution is used to begin execution of a block of code before knowing whether it should be executed. The condition under which the code should be permitted to execute is computed in the data processor, and sent back to the branch processor via a `send!` instruction. Often, this information determines which of "`commit true`" or "`commit false`" should be executed. To optimize this case,

Instruction	Stall	Purpose
`sfetch` *addr*, *N*	n	fetch and speculatively execute instructions
`sfetch?` *addr*, *N*	y	optimized speculative execution
`commit true`	n	commit results of last speculative execution
`commit false`	n	discard results of last speculative execution

Table 5.3. Instructions supporting speculative execution.

we introduce the `sfetch?` instruction. "`sfetch?` *addr*,*N*" behaves like `sfetch`. In addition, it receives a value from the data processor and uses this value to determine which commit instruction should be executed. It would be equivalent to the following branch processor code:

```
    sfetch addr,N;
    if? A;
    commit false; goto B;
A:  commit true
B:  ...
```

The instructions for supporting software-controlled speculative execution are summarized in Table 5.3.

5.8. Compilation Issues

Existing compilation techniques can be used to generate code for the branch processor. In the worst-case, a standard instruction set can be translated directly into branch processor instructions by replacing conditional branches with `send!` and `if?` pairs, and using `fetch` instructions to dispatch instructions within a basic block.

Loop Detection. Both fixed length and variable length loops can be detected by modern compilation systems. Most programming languages have constructs for simple iterated loops, simplifying the problem of loop detection. Therefore, a compiler can generate `push` instructions for loops. In addition, subroutine call and returns are explicit in the language. Therefore, these instructions can be easily generated by standard compilation systems. Indeed, the branch processor instruction set is easier to map to because the call and return semantics are provided by the hardware directly.

Peephole Optimization. Peephole optimization can be used to move a `send!` instruction before any other instructions in the data processor that it depends on.

Recall that early `send!` instructions will improve the performance of the branch processor architecture.

Code Sharing. Loop unrolling and loop peeling are transformations used to improve the performance of programs. Both transformations replicate the body of the loop in order to statically determine the direction of some of the branches in the loop body. Observe that such program transformations replicate code just in the branch processor; streams of instructions in the data processor can be re-used because they no longer encode any control flow information. This implies that we will not worsen instruction cache performance by applying such transformations.

We can also think of `fetch` instructions as providing a simple interface for implementing microcode. A sequence of instructions stored at fixed addresses in memory can be used to create complex "instructions" of the form of `fetch` *addr*, N. The effect of executing these instructions would be to execute the sequence of instructions stored at the specified memory address, providing the same effect as an architecture that included programmable microcode.

5.9. Related Work

There is a wealth of research in techniques for alleviating the problem of control dependencies. Smith[23] describes a number of standard approaches to predicting control flow behavior using branch prediction. More recently, complex two-level predictors have been proposed to improve the accuracy of branch prediction.[28] The branch processor approach provides more control flow information to the processor, permitting the processor to speculate less often. Vector architectures are used to provide limited support for fixed length loops. However, not all fixed length loops are vectorizable. For instance, loops that contain irregular array accesses (such as loops containing sparse-matrix operations) cannot be vectorized even though their control flow can be determined statically. In addition, vector architectures do not support nested loops, nor do they support efficient call and return functionality. The branch processor approach can be used in conjunction with existing techniques to improve the performance of processors.

Chapter 6.

CONCLUSION

"Begin at the beginning, and go on till you come to the end; then stop." —Lewis Carroll, *Alice in Wonderland*

This thesis has explored several important aspects of asynchronous architecture and design—an exploration that has yielded promising results.

In the area of high-level design of asynchronous systems, we presented conditions under which an asynchronous computation could be pipelined. The conditions were used to show the correctness of a number of program transformations that introduce concurrency in an asynchronous system. The conditions presented were general enough to be satisfied by the high-level design of a complete asynchronous microprocessor and were used to justify various program transformations used in its design.

We presented latency optimal circuits for solving the prefix problem. The circuits have the best possible asymptotic latency given arbitrary input distributions. Pipelined versions of the circuits were presented, showing that the improvement in latency was not attained at the cost of decreasing the throughput of the prefix computation.

We presented a mechanism for the implementation of precise exceptions in an asynchronous microprocessor. The mechanism was used in the design of a high-performance asynchronous microprocessor. We also presented circuits for the implementation of the non-standard component of the exception mechanism that involved using an arbitration device.

We presented a novel processor architecture for addressing the problem of control flow dependencies. The architecture introduced multiple unsynchronized instruction streams for controlling the execution of a sequential program, with one instruction stream specialized for control flow.

Throughout this thesis, we attempted to eliminate synchronization and dependencies whenever possible. We presented a method to reason about transformations that reduce synchronization between different parts of a computation. The prefix computation circuits took advantage of the presence of input values that eliminated data-dependencies to reduce average-case latency. The exception mechanism eliminated synchronization between parts of the processor except when an exception was encountered—which is an uncommon event. The branch processor approach eliminated synchronization between the control and data part of a program when the control flow was not data-dependent.

6.1. Future Work

Can we construct efficient algorithms for determining when channels in a system are slack elastic? Even if this problem cannot be solved in general, it would suffice to construct sound algorithms that could analyze frequently occurring cases.

Can we use data-dependent optimizations to improve the performance of other arithmetic units? Can we use correlations between successive inputs to improve the performance of arithmetic units?

How does the branch processor perform on standard benchmarks? A full evaluation of this processor would involve writing a compiler that would generate appropriate branch processor code. Unfortunately, an evaluation of the branch processor architecture cannot be done by simply translating standard assembly code to branch processor code in a naive manner, since information about loops that is available to intermediate stages of a compiler is not present in the assembly code.

What instruction sets should asynchronous computers use? It should be possible to improve the average-case performance of asynchronous systems by choosing an instruction set that is optimized appropriately.

Appendix 1.

Communicating Hardware Processes

The notation we use to describe hardware, called "Communicating Hardware Processes" (CHP), is based on Hoare's CSP.[7] A complete formal semantics of the language can be found in van der Goot's thesis.[6] What follows is a short and informal language summary.

Simple statements and expressions.

- Skip: **skip**. This statement does nothing.
- Assignment: $x := E$. This statement means "assign the value of E to x." When E is **true**, we abbreviate $x := E$ to $x\uparrow$, and when E is **false** we abbreviate $x := E$ to $x\downarrow$. Setting elements of a vector x of boolean-valued variables to **true** in a concurrent manner is denoted $x \Uparrow$, while setting elements to **false** is denoted $x \Downarrow$.
- Communication: $X!e$ means send the value of e over channel X; $Y?x$ means receive a value over channel Y and store it in variable x. When we are not communicating data values over a channel, the directionality of the channel may be unimportant. In this case, the statement X denotes a synchronization action on port X.
- Probe: The boolean $\overline{X}$ is true if and only if a communication over channel X can complete without suspending.

Compound statements.

- Selection: $[G_1 \rightarrow S_1 \,\square\, \ldots \,\square\, G_n \rightarrow S_n]$, where G_i's are boolean expressions (guards) and S_i'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 shorthand for $[G \rightarrow \textbf{skip}]$, 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: $*[G_1 \rightarrow S_1 ▯ \ldots ▯ G_n \rightarrow S_n]$. 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 shorthand for $*[\textbf{true} \rightarrow S]$. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "▯."
- Sequential Composition: $S; T$. The execution of this command corresponds to executing S followed by T. The semicolon binds tighter than the parallel composition operator $\|$, but weaker than the comma or bullet.
- Parallel Composition: $S \| T$ or S, T. The execution of this command corresponds to executing commands S and T in parallel. The $\|$ operator binds weaker than the bullet or semicolon. The comma binds tighter than the semicolon but weaker than the bullet.
- Simultaneous Composition: $S \bullet T$ (read "S bullet T"). The execution of this command corresponds to executing the actions S and T such that they complete simultaneously. Typically, the two actions are communication actions only, and the implementation of the bullet corresponds to replacing S by $S; S$ and T by $T; T$ and then picking an interleaving of the "doubled" actions, like $S; T; S; T$.

The concurrent execution of a collection of CHP processes is assumed to be *weakly fair*—every continuously enabled action will be given a chance to execute eventually. The selection statement is assumed to be demonic, and it therefore *not* fair. Consider the following four processes:

$$\begin{array}{ll} & *[\ X!0\ ] \ \|\ *[\ Y!1\ ] \\ \| & *[[\overline{X} \longrightarrow X?x\ ▯\ \overline{Y} \longrightarrow Y?x\ ];\ Z!x\ ] \\ \| & *[\ W!2\ ] \end{array}$$

Since the selection statement is not fair, Z is permitted to output an infinite sequence of zeros. However, both $Z!x$ and $W!2$ will execute eventually, since parallel composition is assumed to be weakly fair.

Appendix 2.

Analysis of Parallel Prefix

Given an N-input prefix computation, let c_N be the length of the longest sequence of propagate inputs. We would like to determine the expected value of c_N, assuming that the n inputs are independent, identically distributed random variables and that the probability of an input being of propagate type is $(1-p)=q$. We use a simple generalization of the reasoning presented by Burks et al.[1] Clearly, the expected value of c_N is given by:

$$\sum_{k=1}^{N} \Pr[c_N \geq k] \qquad (*)$$

where $\Pr[c_N \geq k]$ is the probability that the length of the longest sequence of propagate inputs is at least k.

The probability $\Pr[c_N \geq k]$ consists of two parts: (a) the probability that the first $(N-1)$ inputs have a sequence of propagate inputs at least k; (b) the probability that the first $N-1$ don't have such a sequence but adding the nth input produces a sequence of length k. We can therefore write:

$$\Pr[c_N \geq k] = \Pr[c_{N-1} \geq k] + q^k(1-q)\,(1-\Pr[c_{N-k-1} \geq k])$$

The second term (which corresponds to part b) is obtained by observing that of the n inputs, the last k inputs are of type propagate, and the input at position $N-k$ is not of type propagate. We also need to take into account the fact that the first $N-k-1$ positions do not have a propagate sequence of length at least k. Repeatedly expanding the first term, we obtain:

$$\begin{aligned}\Pr[c_N \geq k] &= \Pr[c_k \geq k] + q^k(1-q) \sum_{i=k+1}^{N} (1 - \Pr[c_{i-k-1} \geq k]) \\ &= q^k + q^k(1-q)(N-k) - q^k(1-q) \sum_{i=k+1}^{N} \Pr[c_{i-k-1} \geq k] \\ &\leq q^k + q^k(1-q)(N-k) \\ &\leq q^k + q^k(1-q)N\end{aligned}$$

To complete the proof, we note that $\Pr[c_N \geq k] \leq 1$. We split the range of the summation $(*)$ into two parts.

$$\begin{aligned}\sum_{k=1}^{N} \Pr[c_N \geq k] &= \sum_{k=1}^{K-1} \Pr[c_N \geq k] + \sum_{k=K}^{N} \Pr[c_N \geq k] \\ &\leq K - 1 + \sum_{k=K}^{N} (q^k + q^k(1-q)N) \\ &= K - 1 + (1 + (1-q)N) \sum_{k=K}^{N} q^k \\ &= K - 1 + \frac{q^K(1+(1-q)N)(1-q^{N-K+1})}{1-q} \\ &\leq K - 1 + \frac{q^K}{1-q} + Nq^K\end{aligned}$$

Pick K such that $Nq^K \leq 1$, i.e., pick $K = \lceil \log_{1/q} N \rceil$. We obtain:

$$\begin{aligned}\sum_{k=1}^{N} \Pr[c_N \geq k] &\leq \lceil \log_{1/q} N \rceil + \frac{1}{N(1-q)} \\ &= O(\log N)\end{aligned}$$

It is clear that we can extend this proof to different values q_i for the different input positions by overestimating q_i to be the largest possible value. In that case, we have:

$$\begin{aligned}\sum_{k=1}^{N} \Pr[c_N \geq k] &\leq \lceil \log_{1/q_{\max}} N \rceil + \frac{1}{N(1-q_{\max})} \\ &= O(\log N)\end{aligned}$$

References

1. A.W. Burks, H.H. Goldstein, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Institute for Advanced Study, Princeton, N.J., June 1946.

2. Uri V. Cummings, Andrew M. Lines, and Alain J. Martin. An asynchronous pipelined lattice-structure filter. In *Proceedings of the First International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 126–133, November 1994.

3. E.W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations, *Information Processing Letters*, **11**(1), August 1980.

4. S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. A micropipelined ARM. *Proceedings of the VII Banff Workshop: Asynchronous Hardware Design*, August 1993.

5. Peter Gemmell and Mor Harchol. Tight Bounds on Expected Time to Add Correctly and Add Mostly Correctly. *Information Processing Letters*, **49**:77–83, 1994. A misleading version of this paper appeared as a University of California at Berkeley technical report CSD-93-737, 1993.

6. Marcel van der Goot. The Semantics of VLSI Synthesis. Ph.D. thesis CS-TR-95-08, California Institute of Technology, 1996.

7. C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, 1978.

8. Kevin S. van Horn. *An Approach to Concurrent Semantics Using Complete Traces.* M.S. thesis 5236:TR:86, California Institute of Technology, 1986.

9. Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the Association for Computing Machinery*, **27**(4):831–838, October 1980.

10. F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan-Kaufmann, 1992.

11. Andrew Matthew Lines. Pipelined Asynchronous Circuits. M.S. thesis, California Institute of Technology, 1996.

12. Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-mei W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

13. Alain J. Martin. An Axiomatic definition of synchronization primitives. *Acta Informatica*, **16**:219–235, 1981.

14. Alain J. Martin. The Probe: An addition to communication primitives. *Information Processing Letters*, **20**:125–130, 1985.

15. Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

16. Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373, MIT Press, 1991.

17. 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. *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

18. Raymond E. Miller. *Switching Theory*, Volumes 1 and 2. John Wiley and Sons, 1965.

19. Jayadev Misra and K. Mani Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, **SE-7**(4):417–426, July 1981.

20. William F. Richardson. Architectural Considerations in a Self-Timed Processor Design. Ph.D. thesis, Department of Computer Science, University of Utah, 1996.

21. Charles L. Seitz (editor). *Proceedings of the Caltech Conference on Very Large Scale Integration*, 1979.

22. Charles L. Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems*, by Carver Mead and Lynn Conway, Addison-Wesley, 1979.

23. J.E. Smith. A study of branch prediction strategies. *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

24. Jan L.A. van de Snepscheut. *Trace theory and VLSI design*. Lecture Notes in Computer Science 200, Springer-Verlag, 1985.

25. José A. Tierno, Rajit Manohar, and Alain J. Martin. The Energy and Entropy of VLSI Computations. *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996.

26. Ted Eugene Williams. Self-timed Rings and their Application to Division. Ph.D. thesis, Computer Systems Laboratory, Stanford University, May 1991.

27. S. Winograd. On the Time Required to Perform Addition. *Journal of the Association of Computing Machinery*, **12**(2):277–285, April 1965.

28. T.-Y. Yeh and Y.N. Patt. Two-level adaptive branch prediction. *Proceedings of the 24th Annual ACM/IEEE Symposium on Microarchitecture*, 1991.