INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
THE POWER OF THE HALF-ADDER FORM

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

David R. Lutz, B.S., M.A., M.S.

* * * * *

The Ohio State University

1996

Dissertation Committee:
D. N. Jayasimha
Timothy J. Carlson
Ten-Hwang Lai
Füsun Ö zgüner
P. Sadayappan

Approved by

[Signature]
Adviser

Department of Computer
and Information Science
To Karen, Michael, Rachel, and Rebecca
ACKNOWLEDGMENTS

I give my sincerest thanks to all of the following:

- D. N. (Jay) Jayasimha, my advisor, for his encouragement, patience, hard work, insight, and friendship.

- the extraordinary teachers I have had, especially David Ray and Doug Sturm of Bucknell University, and D. N. Jayasimha, Wolfgang Küchlin, Ten-Hwang (Steve) Lai, Tim Long, and P. (Saday) Sadayappan of the Ohio State University, for giving so much of themselves to me and to other students.

- Wolfgang Küchlin, for introducing me to computer algebra and arithmetic.

- Tim Carlson, for his careful reading which led to a greatly improved thesis.

- the remaining members of my committee, Ten-Hwang Lai, Füsun Özgüner, and P. Sadayappan, for their suggestions and comments.

- Thong Luu, my supervisor at Bell Labs, for allowing me to spend Fridays at school for a year so that I could finish my degree.

- Al Aburto of the Naval Command, Control and Ocean Surveillance Center RDT&E Division in San Diego, CA, for maintaining the benchmarks we used in our simulations.
• the Ohio Supercomputing Center, for giving us access to an Alpha workstation for running our simulations.

• Kiran Kumar Bondalapati, for helping to set-up, run, and analyze our simulations.

• my children, Michael, Rachel, and Rebecca, for enduring my frequent absences.

• and especially to Karen Roller, my wife, for her continual support through this long and sometimes difficult process.
VITA

1980 ............................................. B.S. Mathematics

1980 ............................................. M.A. Mathematics

1990 ............................................. M.S. Computer Science

1980-present ................................. Member of Technical Staff,
               Bell Labs

Publications


Fields of Study

Major Field: Computer and Information Science

Studies in:

- Computer Architecture  Prof. D. N. Jayasimha
- Parallel Computing    Prof. D. N. Jayasimha, Prof. P. Sadayappan
- Algorithms           Prof. Wolfgang Küchlin, Prof. Ten-Hwang Lai
# Table of Contents

DEDICATION ................................................................. ii  
ACKNOWLEDGMENTS .................................................. iii  
VITA ............................................................................. v  
LIST OF TABLES ......................................................... ix  
LIST OF FIGURES ......................................................... x  

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Introduction ................................................ 1</td>
</tr>
<tr>
<td>1.1</td>
<td>Notation .......................................................... 5</td>
</tr>
<tr>
<td>1.2</td>
<td>Problem Definitions ........................................... 6</td>
</tr>
<tr>
<td>1.3</td>
<td>Overview of Related Work ................................... 8</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Half-Adder Based Counters ................................... 8</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Carry-Generation Detection and Addition ................. 10</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Early Zero Detection ......................................... 11</td>
</tr>
<tr>
<td>II</td>
<td>Theory .................................................................. 14</td>
</tr>
<tr>
<td>2.1</td>
<td>Model ............................................................... 14</td>
</tr>
<tr>
<td>2.2</td>
<td>Carry-Save Form ................................................ 18</td>
</tr>
<tr>
<td>2.3</td>
<td>Half-Adder Form ................................................ 21</td>
</tr>
<tr>
<td>III</td>
<td>Modulo-K Counters .............................................. 27</td>
</tr>
<tr>
<td>3.1</td>
<td>Counting in Half-Adder Form ................................ 28</td>
</tr>
<tr>
<td>3.2</td>
<td>Detecting the Final Increment in Constant Time ........ 29</td>
</tr>
</tbody>
</table>
3.3 Enhancements ................................................................. 35
3.4 Comparison to Other Constant-Time Counters .................. 36

IV Carry Generation Detection, Comparison, and Addition ........ 39
4.1 Carry Generation Detection .................................................. 40
  4.1.1 The Prefix-And Method .................................................. 41
  4.1.2 The Carry-Lookahead Method and Brent's Method .......... 44
  4.1.3 The Partial Prefix-And Method ...................................... 48
  4.1.4 Evaluation of the Four Methods .................................... 53
4.2 Comparison ........................................................................... 56
4.3 Addition .............................................................................. 59
  4.3.1 16-Bit PPA Adder .......................................................... 61
  4.3.2 32 and 64-Bit PPA Adders ............................................. 64
  4.3.3 Subtraction ................................................................. 66
4.4 Implementation Issues ....................................................... 67

V Early Branch Condition Resolution ....................................... 69
5.1 Early Zero Detection .......................................................... 70
  5.1.1 Addition, Subtraction, and Zero Detection .................... 70
  5.1.2 Cost and Delay ............................................................ 75
  5.1.3 Other Operations ........................................................ 76
5.2 Sign, Carry, and Overflow Detection .................................... 77
  5.2.1 Sign, Carry, and Overflow in Half-Adder Form .............. 77
  5.2.2 Sign, Carry, and Overflow for Complemented Problems ... 79
5.3 Architectural Impact .......................................................... 82
  5.3.1 Branch Penalties .......................................................... 82
  5.3.2 New Instructions .......................................................... 84
  5.3.3 Speculative Execution ................................................... 85
5.4 Simulation ............................................................................ 87
  5.4.1 Methodology ............................................................... 87
  5.4.2 Results and Discussion ................................................ 92

VI Conclusion ........................................................................... 98

BIBLIOGRAPHY ................................................................. 104
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Notation</td>
</tr>
<tr>
<td>2</td>
<td>Representation of tristate buffers and 2-input multiplexers</td>
</tr>
<tr>
<td>3</td>
<td>Detecting the 67th clock pulse in h-a form</td>
</tr>
<tr>
<td>4</td>
<td>Time at which each bit of S is set when counting to 67</td>
</tr>
<tr>
<td>5</td>
<td>Stages and logic levels for a PG64 module constructed from PG4 modules</td>
</tr>
<tr>
<td>6</td>
<td>Asymptotic delay and cost for group carry-generation detection</td>
</tr>
<tr>
<td>7</td>
<td>Carry generation delay and cost for some small values of $n$</td>
</tr>
<tr>
<td>8</td>
<td>Description of raw simulation data</td>
</tr>
<tr>
<td>9</td>
<td>Formulas for simulation analysis</td>
</tr>
<tr>
<td>10</td>
<td>Benchmarks</td>
</tr>
<tr>
<td>11</td>
<td>Raw simulation data</td>
</tr>
<tr>
<td>12</td>
<td>Simulation analysis</td>
</tr>
<tr>
<td>13</td>
<td>New upper bounds</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>43</td>
</tr>
<tr>
<td>8</td>
<td>46</td>
</tr>
<tr>
<td>9</td>
<td>47</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>51</td>
</tr>
<tr>
<td>12</td>
<td>61</td>
</tr>
<tr>
<td>13</td>
<td>62</td>
</tr>
<tr>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td>Page</td>
<td>Subject</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>15</td>
<td>8-bit prefix-and using recursive doubling</td>
</tr>
<tr>
<td>16</td>
<td>Combined adder/subtractor/zero-detector</td>
</tr>
<tr>
<td>17</td>
<td>Example for discussion of conditions</td>
</tr>
<tr>
<td>18</td>
<td>Early branch condition detector</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

The fundamental problem in computer architecture is how to build a better computer. Modern computers are both pipelined and superscalar, and are limited in speed by the latency of their operations and the amount of instruction-level parallelism (ILP) available in programs. Decreasing latency and increasing ILP are primary goals for computer architects.

In this thesis, we introduce and study the half-adder form, a new intermediate representation of sums that can be used to decrease the latency of some of the most basic operations in computers. Informally, a sum is in half-adder form if it is the result of adding two n-bit two's complement numbers with a linear array of n half adders (precise definitions will be given in section 2.3). This addition requires only a small, constant delay, yet the half-adder form has enough structure so that we can use it to detect many of the properties of the final sum. Furthermore, use of the half-adder form speeds up the computation of the final sum by nearly a factor of two.

We begin by developing the theory of the half-adder form, and then applying this theory to some of the most basic operations in computers. These operations include modulo-k counting; comparing two numbers to see which is greater; detecting
whether a sum is zero, positive, or negative; detecting carry out and overflow; and computing the sum or difference of two numbers. The result is new algorithms and circuit designs that dramatically decrease the latency of these operations. For some operations, the reduction in latency is great enough to allow concurrent execution of other instructions that depend on their results. For example, our technique of early zero detection allows a branch that depends on whether a sum is zero to be executed concurrently with the computation of the sum. For these operations, our algorithms result in both decreased latency and increased ILP.

We measure our results with respect to a variation of the boolean circuit model in which we charge one unit of delay for: (1) any two-input logical function, or (2) any single level of tristate buffers. Item (2) allows us to select one of \( k \) inputs when exactly one input is enabled. A complete description of and justification for the model is given in section 2.1.

Our main contributions are as follows:

- The theory of half-adder form. Numbers in half-adder form have some surprising and useful properties. While these properties are used here to solve some important problems, we believe that many other applications exist. Some possibilities are discussed in chapter VI.

- A design for a programmable frequency divider that has a small fixed delay irrespective of the word-length. Our counter is simultaneously simpler and faster than any other programmable frequency divider.
- Two new designs for detecting whether an $n$-bit sum has generated a carry. Both designs require only $\log n + 3$ logic levels, which is nearly twice as fast as the best existing implementations. Besides being faster, our designs are simpler, require fewer gates, and have a more regular implementation than earlier designs.

- A new adder, the partial prefix-and adder, that computes an $n$-bit sum in only $\log n + 4$ logic levels, again nearly twice as fast as the best existing implementations.

- A new design for an early zero detector, i.e., a sum equals zero detector that detects whether the sum is zero before the sum is computed. The early zero detector requires only $\log n + 3$ gate delays, and does not add any extra delay to the computation of the sum.

- Designs for sign and overflow detection requiring only $\log n + 4$ logic levels.

- Architectural changes for making use of the early availability of branch conditions. The suggested changes have a very low cost, consisting mainly of forwarding hardware for a few condition bits, and a clearly measurable benefit.

More minor contributions include the following:

- The design of an integrated unit that performs addition and subtraction, early zero detection for the sum or difference, and detects carry-out and overflow from the sum or difference. The integrated unit provides zero detection in $\log n + 3$ logic levels; and provides the sum or difference as well as the sign, carry, and overflow conditions in $\log n + 5$ logic levels.
• The model described in section 2.1. Many circuits, for example adders and multiplexers, are not well described in terms of gates. Our model provides a useful abstraction that allows us to analyze and improve the behavior of real circuits.

The remainder of this chapter is devoted to establishing our notation, defining the problems to be solved, and reviewing the relevant literature.

Chapter II presents our model, and the definitions and basic theory for carry-save form and half-adder form.

Chapter III applies and extends the theory of chapter II to construct a programmable, constant-time modulo-k counter.

Chapter IV contains our most important result: a fast and efficient way to compute whether or not a sum generates a carry. Fast carry-generation detection leads to fast comparison and addition, and we present algorithms and circuit designs for these problems.

Chapter V shows how to use the theory of half-adder form for the early detection of branch conditions, presents some architectural implications of this early detection, and shows some of the benefits of these changes via simulation.

Finally, chapter VI presents some ideas for future applications of the h-a form.

Some of the material in this thesis has been published or accepted for publication elsewhere. In particular, much of chapter III will appear in [LJ], some of chapter IV have been published in [LJ96a], and parts of chapter V will appear in [LJ96b].
1.1 Notation

We use lower-case letters to represent numbers, upper-case letters to represent \( n \)-bit two’s complement numbers, and subscripted lower-case letters to represent bits. Unless specified otherwise, \( X + Y \) and \( X - Y \) represents signed two’s complement addition and subtraction. The meaning of other symbols is given in Table 1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \wedge )</td>
<td>bitwise and</td>
</tr>
<tr>
<td>( \lor )</td>
<td>bitwise or</td>
</tr>
<tr>
<td>( \oplus )</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td>( \bar{x}_i )</td>
<td>not ( x_i )</td>
</tr>
<tr>
<td>( \bar{X} )</td>
<td>one’s complement of ( X )</td>
</tr>
</tbody>
</table>

The low order bit in a word is bit zero. Unless specified otherwise, logarithms are base two, and \( n \) is assumed to be a power of two. This last assumption allows us to express the minimum depth of a \( 2n \)-input function as \( \log n + 1 \) instead of the more correct but less lucid \( \log[2n] \).

In circuits we use the standard gate notations for AND, OR, and XOR gates. NOT gates are not part of our representations. Instead, inversion is indicated by a small circle at the input or output to a gate. Tristate buffers and 2-input multiplexers are denoted as shown in table 2.
Table 2: Representation of tristate buffers and 2-input multiplexers

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tristate buffer</td>
<td>$e_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>two-input multiplexer</td>
<td>$e_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

1.2 Problem Definitions

We provide specific algorithms and circuit designs for the following problems: programmable modulo-$k$ counting, carry-generation detection, comparison, addition, sign detection, overflow detection, and early zero detection.

Our design goals for all circuits include the following:

1. low latency. The circuit should be as fast as possible.

2. simplicity. The circuit should be simple to implement and use.

3. small area and a regular implementation.

For all problems, assume that $X$ and $Y$ are $n$-bit two’s complement numbers, and $c_{in}$ is an optional carry-in bit.

The *programmable modulo-$k$ counting problem* is to create a circuit that provides an output pulse for every $k$ input pulses, where $k$ can be specified as an input to the circuit. This kind of counter can also be referred to as a programmable frequency
divider. A variation of the problem is a circuit that allows \( k \) to change on every output pulse.

The **carry-generation detection problem** is to determine whether or not there is a carry out from the \( n \)-bit sum \( X + Y + c_{in} \). Detecting carry generation is the most important part of the comparison, addition, sign detection, and overflow detection problems.

The **comparison problem** is to determine whether \( X = Y \) or \( X > Y \). Other conditions, namely \( X < Y \), \( X \leq Y \), \( X \geq Y \), or \( X \neq Y \), are easily determined given these two conditions.

The **addition problem** is to compute an \( n \)-bit two's complement number \( T = X + Y + c_{in} \), plus a carry-out bit \( c_{out} \). The carry-in bit is needed for subtraction, and the carry-out bit is needed for overflow detection. Both bits are also useful if one wishes to perform multi-word-length additions. The condition detecting problems are to discover certain properties of \( T \), without necessarily computing \( T \). The **sign detection problem** is to determine the sign of \( T \). The **early zero detection problem** is to determine whether \( T = 0 \) before the computation of \( T \) has completed. The **overflow detection problem** is to determine whether there is an overflow in the computation of \( T \). This is typically detected by determining the carry into the sign bit and the carry out of the sign bit: overflow occurs when the two carries are not equal.
1.3 Overview of Related Work

Many number representations have been proposed for doing computer arithmetic. Some of the most widely known are signed-digit, residue, logarithmic, floating point, binary coded decimal, carry-save, signed magnitude, and one’s and two’s complement. Two’s complement and IEEE floating point are by far the most commonly used. Other representations are usually limited to specific applications if they are used at all. Good overviews of these representations can be found in any of several computer arithmetic texts [Hwa79, Sco85, Kor93].

Half-adder form is a new representation, and it has not been studied in its own right. Of the forms that have been studied, it is most closely related to carry-save form (see section 2.2), and the propagate and generate functions used in addition (see section 4.1.2).

The problems that we apply half-adder form to have been studied, and continue to be studied despite many years of attention. The reason for this continued interest is that the problems are so fundamental that the rewards for even small improvements are great. We examine previous solutions to these problems in three sections, corresponding to our chapters III, IV, and V: modulo-$k$ counters, carry-generation detection and addition, and early zero detection.

1.3.1 Half-Adder Based Counters

Recall that a programmable modulo-$k$ counter provides an output pulse for every $k$ input pulses, where $k$ can be specified as an input to the circuit. The problem with
constructing such a counter is long carry chains. A modulo-$k$ counter must compare
the number of input pulses it receives with some known value, and long carry chains
may change most or all of the bits of the count on a single input pulse. Since so
many bits can change on any input, we might expect that any $n$-bit frequency divider
would require $O(\log n)$ delay.

Thus it was surprising when Ercegovac and Lang demonstrated a constant-time
counter and frequency divider [EL89]. The counter is based on $n$ half adders, and
is constructed out of subcounters of increasing lengths. Each subcounter maintains
two counts: the current count, and a new count, the count that will be displayed
the next time the subcounter receives a carry in. The length of the subcounters are
chosen so that the half adders can produce the new count by the time it is needed,
and a ring counter causes the new count to be instantiated at the proper time. When
combined with a clever $\text{count} = k$ detection scheme, the counter becomes a constant-
time frequency divider, although it is not fully programmable because some values of
$k$ require hardware changes.

Vuillemin improved on Ercegovac and Lang's counter by reducing the number of
latches and the area [Vui91]. His improvements come with some cost, requiring more
work at the design stage because a careful timing analysis is needed to ensure that all
of the counts are updated at the proper time. Vuillemin's counter also only works for
$k = 2^n$, i.e., it is not programmable. We provide a much more detailed look at both
counters in chapter III, where we present the first fully programmable constant-time
frequency divider.
1.3.2 Carry-Generation Detection and Addition

There are a large number of papers written about addition, and good overviews of early work are available [Hwa79, WF82, Swa90]. Many important papers related to addition have also been reprinted [Swa76, Swa80, Swa90]. For the early history of adders, the reader is referred to these sources.

The main types of adders are ripple carry, conditional sum, carry select, carry skip, and carry lookahead, all of which are described in any computer arithmetic text, for example [Kor93] or [Sco85]. There is also a good discussion of adders in appendix A of [HP96].

The "best" adder seems to depend on a given technology, although some variation of carry-lookahead is usually the fastest in practice. The current dominant technology is CMOS VLSI. Mead and Conway claimed that carry-lookahead addition was inappropriate for VLSI [MC80], but Brent and Kung disproved this assertion [BK82]. A fairly detailed layout for a contemporary carry-lookahead adder is given by Weste and Eshraghian [WE93]. The fastest modern adders are carry-lookahead/carry-select hybrids such as found in the AMD Am29050 [LS92], or the DEC Alpha [D+92].

The fastest theoretical adders with non-exponential size are those of Brent and Krapchenko. Brent’s adder [Bre70] uses $O(n \log n)$ gates and has depth $\log n + O(\sqrt{\log n})$. Krapchenko’s adder [Weg87] uses $9n$ gates and has depth bounded by $\log n + 7\sqrt{2\log n} + 16$. Brent’s adder has better performance for practical $n$. As far as we know, neither method has ever been implemented.
Both Brent’s and Krapchenko’s adders are close to the theoretical limits proved by Winograd [Win65] and Spira [Spi73]. In terms of two-input gates using boolean logic, these bounds say that \( n \)-bit addition as defined in section 1.2 requires at least \( \log n + 2 \) logic levels.

Several authors have looked at carry-lookahead addition as an example of the prefix problem [LF80, LD94, Lei92]. Given some associative operation \( \circ \), the prefix problem is to compute all of the products \( X_1 \circ X_2 \circ \ldots \circ X_i \) for \( i = 1, 2, \ldots, n \).

In chapter IV we also look at addition as a prefix problem, but we use a simpler function than has been used in the past, namely a 2-input AND. This will give us an adder requiring only \( \log n + 4 \) logic levels, which is very close to the lower bound in the previous paragraph, although our result uses a slightly different model.

### 1.3.3 Early Zero Detection

The most frequently executed conditional branch instructions for most architectures are “branch on zero” and “branch on nonzero” [HP96]. The value to be tested is typically the result of an addition or subtraction, and in the usual implementation, zero detection cannot complete until after the sum or difference is known. Since detecting zero requires the examination of \( n \) bits, and since most of these bits are not available until after the sum or difference has been computed, the zero condition usually has the highest latency among branch conditions.

The first general solution to early zero detection in sums is presented by Weinberger [Wei75]. The method is based on half adders, and results from simplifying the equations for all possible half-adder outputs that could result in a zero sum. The
resulting expression is quite complicated, but still faster than addition followed by normal zero detection.

A partial solution to the problem of early zero detection is given by Losq and Rao [LR82], who note that determining the zero condition for subtraction is easy. A subtraction yields a zero if and only if the original inputs are equal, a condition that can be checked with \( n \times \text{XNOR} \) gates and an \( n \)-input AND.

MIPS processors make heavy use of this partial solution [Cho89]. The MIPS instruction set requires zero detection less often than most other instruction sets because it has two-argument branch-on-equal or branch-on-not-equal instructions. The MIPS approach provides some of the benefits of early zero detection, but has some significant disadvantages. One disadvantage is that it reduces the number of offset bits in the the branch statement. Another is that it is less general than early zero detection – it solves the problem of detecting whether \( X - Y = 0 \) but not the problem of detecting whether \( X + Y = 0 \). Finally, the MIPS solution is not usable for other existing instruction sets.

Vassiliadis and Putrino designed an improved zero sum detector that operates in parallel with the adder [VP89]. The method is based on finding a more-easily computed expression that is zero if and only if the sum is zero. The expression is computed using full adders and is approximately 35% faster than the adder. Phillips and Vassiliadis extended this result to 3-input ALUs [PV93].

More recently, several authors have independently discovered some closely related methods to compute the more general condition \( X + Y = T \) for an arbitrary \( T \) [CL92,
VPB93, LJ94, Par94]. The simplest and best of these methods will be described in section 2.2.

All of these approaches to early zero detection are different from the one taken in chapter V. Our approach is faster, more general, more closely integrated with the ALU, and can be easily applied to existing instruction sets.
CHAPTER II

Theory

In this chapter we present our model, some of the theory of carry-save form that led to this work, and most importantly, the basic theory of h-a form.

A significant portion of this thesis is devoted to gate-level descriptions of new algorithms. We were faced very early with the problem of finding a technology-independent method for comparing our work with other designs, especially with respect to delay. We have settled on two approaches. For circuits with very easy implementations, we simply list the delay in terms of its component functions: e.g., comparing two n-bit numbers for equality requires the delay of a two-input XNOR and an n-input AND. For more complex circuits, we use the following model.

2.1 Model

Winograd proved lower bounds on addition time using a model in which any d-valued logical function of τ d-valued inputs can be computed in unit time [Win65]. Other important papers that consider addition time, such as [Bre70] and [Spi73], followed Winograd’s lead. A good justification of this approach can be found in [WF82]. We begin with a special case of this model, namely one in which any boolean function
of two inputs can be computed in unit time. We then extend the model to allow unit-time selection of one of \( k \) inputs when exactly one of the inputs is enabled.

More formally, our model is a variation of the boolean circuit model [LD90]: circuits are represented as directed acyclic graphs (DAGs) consisting of input nodes (nodes with in-degree 0), operation nodes, and tristate nodes. Input nodes perform no work and require no delay. Operation nodes compute any 2-input logical function with unit delay. Tristate nodes also have unit delay and will be described below. No restriction is placed on the fanout of any type of node.

Any of the nodes can be designated as output nodes. The depth or level of a node \( v \) is the maximum length along all the paths between the input nodes and \( v \). The depth of the circuit is the maximum depth of all of the output nodes. Since all of the non-input nodes have unit delay, the delay for a circuit is given by its depth.

Tristate nodes are an extension to the original model. They are meant to model the concepts of tristate buffers and pass transistors, which are widely used in VLSI, but which are not well explained by the boolean circuit model. A tristate node represents a set of \( k \) tristate buffers attached to a single output wire. Each tristate buffer \( j \) is a circuit with two inputs, \((e_j, v_j)\), and a single output that can be in one of three states: 0, 1, or \( Z \). The first two states are self explanatory. The third state, \( Z \), is the high impedance or logically disconnected state. If \( e_j = 0 \), then the output of the tristate buffer is \( Z \). If \( e_j = 1 \), then the output is \( v_j \). The circuit designer must guarantee that exactly one of the \( k \) tristate buffers is not in the high impedance state.
A tristate node uses \( k \) control bits to select one of \( k \) input bits with unit delay. The input to a tristate node consists of \( k \) ordered pairs of bits, \((e_0, v_0), (e_1, v_1), \ldots, (e_{k-1}, v_{k-1})\). Just as the circuit designer must guarantee that exactly one tristate buffer is not in the high impedance state, the algorithm designer must guarantee that for some \( i, e_i = 1 \), and \( e_j = 0 \) for \( j \neq i \). The output of the tristate node is \( v_i \).

We represent DAGs using traditional circuit symbols, with extensions for tristate buffers and two-input multiplexers as shown in table 2.

![Figure 1: 4:1 multiplexer](image)

**Example:** Consider the 4:1 multiplexer in figure 1. The AND gates are operation nodes which are used to decode the inputs \( s_0 \) and \( s_1 \). The dashed box represents a tristate node, which uses the output of the decoder to select one of the input \( v_i \) in unit time. The delay for this circuit is 2. A similar multiplexer that does not use tristate nodes is shown in figure 2. This second multiplexer has a less realistic delay of 4.
Like most models, this one is a simplification of a more complex reality. In reality, different logic functions require different amounts of delay, fanout is limited, and tristate nodes have some limitations with respect to number of inputs and number of consecutive levels in which they can be cascaded. Quantifying these and other important factors would require making many technology-dependent assumptions. Instead of doing this, we have followed the lead of [Bre70] and [Win65], and simply assumed that delay is proportional to the depth of the circuit.

Despite the simplifying assumptions of the model, we believe that the circuits we present are practical. We have been much more conservative than the model allows when constructing our circuits, especially with respect to the number of tristate buffers in a tristate node, the fanout of tristate nodes, and the fanout of operation nodes. The largest tristate node we use in this thesis has 9 inputs, and it would be
perfectly feasible to build all of our circuits using tristate nodes with a maximum of 5 inputs. Only the final tristate node is on the critical path for any of our circuits, so if tristate nodes require more delay in a particular technology, this has only a small constant effect on our overall delay. The slack time between the production of a tristate output and its use also allows the insertion of drivers, effectively making the fanout one for all except the final tristate node. We have opted for simplicity of exposition over minimization of fanout for operation nodes in our diagrams, but there are some easy changes that can be made to the circuits to restrict fanout to a maximum of two on most logic levels. These changes, along a more detailed discussion of fanout will be presented when we discuss addition in section 4.4.

In summary, we believe that the model is a useful starting point when comparing the speed of various algorithms, especially when it is used conservatively as in this thesis. Of course determining exact speedups will require implementations. The model’s extension to include tristate nodes is also useful because it allows us to compare algorithms that cannot be adequately described in terms of gates.

2.2 Carry-Save Form

An n-bit carry-save adder (CSA) consists of n independent full adders. It takes three n-bit two’s complement numbers as inputs, and produces two outputs: an n-bit sum and an n-bit carry. Let \( X = x_{n-1} \ldots x_1 x_0 \), \( Y = y_{n-1} \ldots y_1 y_0 \), and \( Z = z_{n-1} \ldots z_1 z_0 \) be n-bit words with low order bits \( x_0, y_0, \) and \( z_0 \). An n-bit CSA produces a carry
word $U = u_{n-1} \ldots u_1 u_0$ and a sum word $V = v_{n-1} \ldots v_1 v_0$ such that

$$u_i = (x_{i-1} \land y_{i-1}) \lor (x_{i-1} \land z_{i-1}) \lor (y_{i-1} \land z_{i-1}) \quad (2.1)$$

$$v_i = x_i \oplus y_i \oplus z_i \quad (2.2)$$

The high order carry bit, $u_n$, is not usually considered to be part of $U$. Note that $u_0$ is always 0, and that $U + V = X + Y + Z$ (recall that all sums are modulo $2^n$).

**Definition**: $(U, V)$ is in $n$-bit carry-save form (or $n$-bit c-s form) if there exist $n$-bit $X$, $Y$ and $Z$ satisfying equations 2.1 and 2.2. We write $(U, V) = \text{csa}(X, Y, Z)$, and we usually omit the modifier "$n$-bit" unless it is necessary for clarity.

**Example**: Figure 3 shows a 4-bit carry-save adder, together with its inputs and outputs. Note that $u_0$ is not an output.

![4-bit carry-save adder](image)

Figure 3: 4-bit carry-save adder

When $U$ and $V$ are added with a carry-propagate adder (CPA), the resulting number $U + V$ is the carry-propagate form (or c-p form) of the sum.

Historically, carry-save addition has been used for a limited set of intermediate calculations, with the most common example being the accumulation of the partial
products of a multiplication [Wal64]. More recently, carry-save adders have been used to compare a sum to a given value, i.e., to compute whether or not \( X + Y = Z \). The basic method has been discovered independently by many authors [CL92, LJ94, Par94, PV93, VPB93, VP89], although only formally proved by us [LJ94]. The method relies on the fact that for two's complement numbers, it is easy to detect when a sum is \(-1\).

**Lemma 2.1** Let \( X \) and \( Y \) be \( n \)-bit two's complement numbers. Then \( X + Y = -1 \Leftrightarrow X \) and \( Y \) differ in every bit position.

**Proof:**

\[
X + Y = -1 \Leftrightarrow Y = X \Leftrightarrow X \) and \( Y \) differ in every bit position. \( \Box \)

The test in lemma 2.1 is much faster than an addition, requiring only the delay of an XOR gate and an \( n \)-input AND. Extending this test to allow comparison with an arbitrary value is straightforward, as shown in lemma 2.2.

**Lemma 2.2** Let \( X, Y, \) and \( Z \) be \( n \)-bit two's complement numbers. Then \( X + Y = Z \Leftrightarrow X + Y + Z = -1 \)

**Proof:**

\[
X + Y = Z \Leftrightarrow X + Y - Z = 0 \tag{2.3}
\]
\[
\Leftrightarrow X + Y - Z - 1 = -1 \tag{2.4}
\]
\[
\Leftrightarrow X + Y + \overline{Z} = -1 \tag{2.5}
\]

The last equivalence holds because \( Z \) is a two's complement number and so \(-Z = \overline{Z} + 1\). \( \Box \)
Theorem 2.1 follows immediately from lemmas 2.1 and 2.2.

**Theorem 2.1** Let $X, Y, \text{ and } Z$ be $n$-bit two's complement numbers, and let $(U, V) = csa(X, Y, Z)$. Then $X + Y = Z \iff U \text{ and } V \text{ differ in every bit position.}$

Computing $(U, V) = csa(X, Y, Z)$ can be done with the delay of a full-adder plus an inverter, so the total delay in establishing whether $X + Y = Z$ is only a small constant value plus the delay of an $n$-input AND.

Some architectural implications of theorem 2.1 are given in [CL92, LJ94, PV93, VPB93, VP89].

Note that theorem 2.1 does not rely on any special properties of c-s form. CSAs are used merely to change a 3-input sum to a 2-input sum. In fact, numbers in c-s form have no special properties — the only property that can disqualify an arbitrary pair $(U, V)$ from being in c-s form is if $u_q = 1$. All of the interesting information in theorem 2.1 comes from an analysis of the 2-input sum, leading us to suspect that something more fundamental is going on with 2-input sums. We address this suspicion in the next section.

### 2.3 Half-Adder Form

An $n$-bit half adder consists of $n$ independent half adders. It takes two $n$-bit two's complement numbers as inputs, and produces two outputs: an $n$-bit sum and an $n$-bit carry. Let $X = x_{n-1} \ldots x_1 x_0$, and $Y = y_{n-1} \ldots y_1 y_0$ be $n$-bit words with low order bits $x_0$ and $y_0$. An $n$-bit half adder produces a carry word $C = c_{n-1} \ldots c_1 0$ and a sum
word $S = s_{n-1} \ldots s_1 s_0$ such that

$$
c_i = x_{i-1} \land y_{i-1}
$$

(2.6)

$$
s_i = x_i \oplus y_i
$$

(2.7)

The high order carry bit, $c_n$, is not usually considered to be part of $C$, but is useful when discussing carries or overflow. Clearly $C + S = X + Y$ (recall that all sums are module $2^n$). If we include $c_n$, the equality also holds for integers. The low order carry bit, $c_0$, is always 0.

**Definition:** $(C, S)$ is in $n$-bit half-adder form (or $n$-bit h-a form) if there exist $n$-bit $X$ and $Y$ satisfying equations 2.6 and 2.7. We write $(C, S) = ha(X, Y)$, and we usually omit the modifier "$n$-bit" unless it is necessary for clarity.

**Example:** Figure 4 shows a 4-bit half adder, together with its inputs and outputs. Note that $c_0$ is not an output of the half adder.

![Figure 4: 4-bit half adder](image)

The following lemma give us a more convenient way to prove that a pair of numbers is in h-a form.
Lemma 2.3 Suppose $C$ and $S$ are two $n$-bit two’s complement numbers. $(C, S)$ is in h-a form $\Leftrightarrow c_0 = 0$ and $c_i \land s_{i-1} = 0$ for $i = 1, 2, \ldots, n - 1$.

Proof:

[$\Rightarrow$] This is immediate from the definition.

[$\Leftarrow$] Set $c_n = 0$, and construct $X$ and $Y$ according to the following algorithm:

for $i = 0, 1, \ldots, n - 1$ do

if $s_i = 1$ then $x_i \leftarrow 0$ and $y_i \leftarrow 1$

else $x_i \leftarrow y_i \leftarrow c_{i+1}$.

Then $X, Y, C$, and $S$ satisfy equations 2.6 and 2.7, so $(C, S)$ is in h-a form. □

Example: The number $(C, S) = (2, 1) = (0010, 0001)$ is not in h-a form because $c_1 \land s_0 = 1$.

Corollary 2.1 Let $(C, S)$ be a number in h-a form. The number of bits in $(C, S)$ that are set to one is less than or equal to $n$.

Proof: The bit $c_0 = 0$. All of the other bits of $(C, S)$ are in one of the pairs $(c_i, s_{i-1})$, where $i \in \{1, \ldots, n\}$. By lemma 2.3, at most one bit is set in each of these pairs. □

The following lemma is useful in many proofs, allowing us to assume that part of an existing number in h-a form is also in h-a form.

Lemma 2.4 If $(C, S)$ is in $n$-bit h-a form, then for any $m$ such that $1 \leq m \leq n$, the low order $m$ bits of $(C, S)$ are also in h-a form.
Proof: By definition, there exist $n$-bit $X$ and $Y$ satisfying equations 2.6 and 2.7. The low order $m$ bits of $X$ and $Y$ satisfy these equations for the low order $m$ bits of $(C,S)$. □

Carry-save and half-adder forms are examples of redundant representations of numbers. Half-adder form is much less redundant than c-s form.

Lemma 2.5 The number of different representations of numbers in $n$-bit c-s form is $2^{2n-1}$.

Proof: Except for bit $c_0$, which must be zero, the value of any bit can be zero or one, independent of the value in any other bit. □

Lemma 2.6 The number of different representations of numbers in $n$-bit h-a form is $2 \times 3^{n-1}$.

Proof: Bit $c_0$ must be zero and bit $s_{n-1}$ can be zero or one, independent of the value of any other bits. For each remaining pair $(c_i, s_{i-1})$, the values can be $(0, 0), (0, 1),$ or $(1, 0)$. There are $n-1$ such pairs. □

Remark: By lemma 2.6, there are an average of $(3/2)^{n-1}$ h-a form representations for each $n$-bit number. For 32-bit numbers, this is approximately 300,000 representations per number. For 64-bit numbers, this is over 100 billion representations per number.

Despite the large average number of representations, there are some numbers with far fewer representations, including the following number whose representation is unique.
Theorem 2.2 Let \((C, S)\) be a number in h-a form. Then \(C + S = -1 \Leftrightarrow S = -1\).

Proof:

\([\Rightarrow]\) \((C, S)\) is in h-a form, so there exist \(X\) and \(Y\) such that \(X + Y = -1\) and \((C, S)\) is the result of adding \(X\) and \(Y\) with a half adder. By the definition of a two's complement number, \(X + Y = -1 \Leftrightarrow Y = \overline{X}\). Then by equation 2.7, \(S = X \oplus \overline{X} = -1\).

\([\Leftarrow]\) By the definition of h-a form, only one of \(c_i\) and \(s_{i-1}\) can be set for \(i = 1, \ldots, n\), so \(C = 0\), and \(C + S = -1\). \(\square\)

Even more surprising than the uniqueness of \(-1\) in h-a form is the fact that we can detect it without examining \(C\). Theorem 2.2 is a very powerful result, and it is one of the most important parts of this thesis. It provides the basis for fast modulo-\(k\) counting in chapter III, fast equality checking in chapter IV, and early zero detection in chapter V.

Besides \(-1\), the largest negative number, the only other number with a unique representation in h-a form is \(2^{n-1} - 1\), the largest positive number.

Corollary 2.2 Let \((C, S)\) be a number in \(n\)-bit h-a form. Then \(C + S = 2^{n-1} - 1 \Leftrightarrow S = 2^{n-1} - 1\).

Proof:

\([\Leftarrow]\) Is identical to the proof in theorem 2.2.

\([\Rightarrow]\) Let \(C'\) be the low order \(n - 1\) bits of \(C\), and let \(S'\) be the low order \(n - 1\) bits of \(S\). By lemma 2.4, \((C', S')\) is in h-a form, and by theorem 2.2, \(S' = 2^{n-1} - 1\), and \(C' = 0\). Since \(s_{n-2} = 1\), then \(c_{n-1} = 0\), and since \(0 < C + S < 2^{n-1}\), \(s_{n-1} = 0\). \(\square\)
The following theorem proves that $2^n - 1$ and $2^{n-1} - 1$ are the only numbers with unique representations in h-a form.

**Theorem 2.3** Let $(C, S)$ be a number in n-bit h-a form, and suppose $C + S \neq -1$ and $C + S \neq 2^{n-1} - 1$. Then the representation $(C, S)$ is not unique, i.e., there exist $(C', S')$ in h-a form such that $(C, S) \neq (C', S')$ and $C + S = C' + S'$.

**Proof:** By theorem 2.2 and corollary 2.2, there exists $i < n - 1$ such that $s_i = 0$.

We consider 4 cases. For each case, we define a new number $(C', S')$ that is in h-a form, and whose bits have the same values as the corresponding bits of $(C, S)$, except for certain specified bit positions.

- **case 1:** there exists $j > i$ such that $c_j = s_j = 1$. Set $c'_j = s'_j = 0$. If $j < n - 1$, then set $c'_{j+1} = 1$.
- **case 2:** Not case 1, and there is some $j > i$ such that $s_j = 1$. Pick the smallest such $j$, and set $s'_j = 0$ and $c'_j = 1$.
- **case 3:** Not case 1 or 2, and there is some $j > i$ such that $c_j = 1$. Pick the largest such $j$, and set $s'_j = 1$ and $c'_j = 0$.
- **case 4:** Not case 1, 2 or 3, i.e., for all $j > i$, $c_j = s_j = 0$. Set $c'_{n-1} = s'_{n-1} = 1$.

In all 4 cases, $(C', S')$ is in h-a form, $C' \neq C$, $S' \neq S$, but $C' + S' = C + S$. □

While there is considerably more to be said about the theory of h-a form, we will present it in the following chapters rather than here. The reason for this is that the theory seems more natural when accompanied by its motivating problems.
CHAPTER III

Modulo-K Counters

Our first application of half-adder form will be to the theory of counting. A modulo-$k$ counter, or frequency divider, is a binary counter that provides an output pulse for every $k$ clock pulses. Frequency dividers are used in many different circuits, and are distinguished from other counters in that the output pulse is much more important than the count itself. In many cases, the count is not even available. A simple example of such a frequency divider is a ring counter [EL85].

Desirable properties for modulo-$k$ counters (in this chapter we use the words counter and frequency divider interchangeably) include the following: (1) A fast period, or time between clock pulses; (2) programmability, i.e., having a modulus that is not fixed in hardware; (3) simplicity, both in implementation and in use; and (4) a small area and a regular implementation. In this chapter we introduce a counter that has all four of these properties. The counter's most notable feature is its simplicity: it not only uses less hardware than competing designs, but it is easier to understand and implement, and far easier to program.
3.1 Counting in Half-Adder Form

Figure 5 is a programmable modulo-\( k \) counter based on Theorem 2.2. The basic idea is to count up from \( k \) to \( 2^n - 1 \).\(^1\) Initially, \((C, S)\) is set to \((0, k)\). At each clock pulse (the clock input is not shown), the sum in \((C, S)\) increases by one because of the input to the low order half adder. Since the increment value is one, the incrementer can be constructed out of half adders, and \((C, S)\) is always in h-a form. After \( k \) clock pulses \( C + S = 2^n - 1 \), and by theorem 2.2, \( S = 2^n - 1 \). This reduces the detector to an \( n \)-input AND of the bits of \( S \).

![Diagram of a half-adder frequency divider](image)

**Figure 5: 4-bit h-a form frequency divider**

Two observations will help in the understanding of this counter. First, the "1" that is input to the low-order half adder is *not* part of \( C \). By definition, \( c_0 = 0 \), and

---

\(^1\)For this chapter, all numbers are considered to be unsigned. Equality is still measured modulo \( 2^n \).
since a zero has no effect on the sum, the “c₀ input” to the low order half adder is available for other purposes. We use it to cause the sum to increase on every cycle.

Second, figure 5 is a synchronous counter, not a ripple counter. The clock pulse (which is not shown in the diagram) causes all of the half adders to generate their sums simultaneously.

Example: Table 3 shows the contents of the C and S registers for a 7-bit h-a form counter that detects when 67 clock pulses have been received. Initially (C, S) is set to (0,67) = (0,60). After 67 clock pulses, S = 127 = 2⁷ – 1.

The frequency divider can handle any output multiple in the range from 1 to 2ⁿ.

The period with which the counter can be operated is lower bounded by the sum of the delays due to a latch, a half adder, and an n-input AND gate. The next section investigates whether the n-input AND can be eliminated.

3.2 Detecting the Final Increment in Constant Time

A closer examination of Table 3 shows that, at least for this example, the high order bits of S are stable as the number of clock pulses approaches 67. The highest order bit, s₆, does not change after it is set in step 10. The next highest order bit, s₅, does not change after it is set in step 41. Similarly, s₄ is fixed at step 56, s₃ is fixed at step 63, and s₂ is fixed at step 66. Only the two low-order bits of S change when the 67th pulse is received. The pattern appears to be that the bits of S are set in order from high to low; and that once a leftmost (i.e., high order) sequence of bits is set, it remains set until all bits of S are set. We prove that this pattern holds in
Table 3: Detecting the 67th clock pulse in h-a form

<table>
<thead>
<tr>
<th>time</th>
<th>c</th>
<th>s</th>
<th>time</th>
<th>c</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
<td>011110</td>
<td>34</td>
<td>000001</td>
<td>101110</td>
</tr>
<tr>
<td>1</td>
<td>000000</td>
<td>011111</td>
<td>35</td>
<td>000000</td>
<td>101111</td>
</tr>
<tr>
<td>2</td>
<td>000010</td>
<td>011110</td>
<td>36</td>
<td>000001</td>
<td>101110</td>
</tr>
<tr>
<td>3</td>
<td>000000</td>
<td>011111</td>
<td>37</td>
<td>000010</td>
<td>101110</td>
</tr>
<tr>
<td>4</td>
<td>000010</td>
<td>011111</td>
<td>38</td>
<td>000101</td>
<td>101100</td>
</tr>
<tr>
<td>5</td>
<td>000010</td>
<td>011111</td>
<td>39</td>
<td>001000</td>
<td>101001</td>
</tr>
<tr>
<td>6</td>
<td>000101</td>
<td>011100</td>
<td>40</td>
<td>010001</td>
<td>100010</td>
</tr>
<tr>
<td>7</td>
<td>001000</td>
<td>011001</td>
<td>41</td>
<td>010001</td>
<td>110001</td>
</tr>
<tr>
<td>8</td>
<td>010010</td>
<td>010010</td>
<td>42</td>
<td>000001</td>
<td>110010</td>
</tr>
<tr>
<td>9</td>
<td>100010</td>
<td>000000</td>
<td>43</td>
<td>000000</td>
<td>110011</td>
</tr>
<tr>
<td>10</td>
<td>000010</td>
<td>100010</td>
<td>44</td>
<td>000001</td>
<td>110010</td>
</tr>
<tr>
<td>11</td>
<td>000000</td>
<td>100011</td>
<td>45</td>
<td>000010</td>
<td>110010</td>
</tr>
<tr>
<td>12</td>
<td>000010</td>
<td>100010</td>
<td>46</td>
<td>000101</td>
<td>110001</td>
</tr>
<tr>
<td>13</td>
<td>000101</td>
<td>100010</td>
<td>47</td>
<td>000000</td>
<td>110101</td>
</tr>
<tr>
<td>14</td>
<td>000101</td>
<td>100000</td>
<td>48</td>
<td>000010</td>
<td>110101</td>
</tr>
<tr>
<td>15</td>
<td>000000</td>
<td>100101</td>
<td>49</td>
<td>001001</td>
<td>110101</td>
</tr>
<tr>
<td>16</td>
<td>000010</td>
<td>100100</td>
<td>50</td>
<td>000001</td>
<td>110101</td>
</tr>
<tr>
<td>17</td>
<td>000101</td>
<td>100101</td>
<td>51</td>
<td>000000</td>
<td>110111</td>
</tr>
<tr>
<td>18</td>
<td>000101</td>
<td>100100</td>
<td>52</td>
<td>000001</td>
<td>110110</td>
</tr>
<tr>
<td>19</td>
<td>000000</td>
<td>100111</td>
<td>53</td>
<td>000100</td>
<td>110101</td>
</tr>
<tr>
<td>20</td>
<td>000010</td>
<td>100111</td>
<td>54</td>
<td>001010</td>
<td>110100</td>
</tr>
<tr>
<td>21</td>
<td>000101</td>
<td>100111</td>
<td>55</td>
<td>010000</td>
<td>110011</td>
</tr>
<tr>
<td>22</td>
<td>001010</td>
<td>100100</td>
<td>56</td>
<td>000010</td>
<td>110001</td>
</tr>
<tr>
<td>23</td>
<td>001000</td>
<td>100011</td>
<td>57</td>
<td>000100</td>
<td>110001</td>
</tr>
<tr>
<td>24</td>
<td>000010</td>
<td>101001</td>
<td>58</td>
<td>000001</td>
<td>110100</td>
</tr>
<tr>
<td>25</td>
<td>000100</td>
<td>101000</td>
<td>59</td>
<td>000000</td>
<td>110111</td>
</tr>
<tr>
<td>26</td>
<td>000010</td>
<td>101010</td>
<td>60</td>
<td>000001</td>
<td>110110</td>
</tr>
<tr>
<td>27</td>
<td>000000</td>
<td>101011</td>
<td>61</td>
<td>000100</td>
<td>110101</td>
</tr>
<tr>
<td>28</td>
<td>000010</td>
<td>101101</td>
<td>62</td>
<td>000101</td>
<td>111000</td>
</tr>
<tr>
<td>29</td>
<td>000010</td>
<td>101101</td>
<td>63</td>
<td>000000</td>
<td>111101</td>
</tr>
<tr>
<td>30</td>
<td>000101</td>
<td>101000</td>
<td>64</td>
<td>000010</td>
<td>111100</td>
</tr>
<tr>
<td>31</td>
<td>000000</td>
<td>101111</td>
<td>65</td>
<td>000100</td>
<td>111101</td>
</tr>
<tr>
<td>32</td>
<td>000010</td>
<td>101110</td>
<td>66</td>
<td>000001</td>
<td>111100</td>
</tr>
<tr>
<td>33</td>
<td>000010</td>
<td>101101</td>
<td>67</td>
<td>000000</td>
<td>111101</td>
</tr>
</tbody>
</table>
general with a series of lemmas. We also prove that there is sufficient time between the setting of adjacent bits \( s_i \) and \( s_{i-1} \) to allow the detection of \( S = 2^n - 1 \) to be completed in constant time.

For the following lemmas, assume that \((C, S)\) is in h-a form; that the initial \( n+1 \)-bit unsigned sum of \( C \) and \( S \) is less than \( 2^n \); and that \((C, S)\) is incremented at each clock pulse using the incremener in figure 5.

**Lemma 3.1** Bit \( c_n \) changes from 0 to 1 when \( C + S = 2^n + n - 1 \), i.e., exactly \( n \) pulses after \( C + S = 2^n - 1 \).

**Proof:** Clearly \( c_n = 0 \) while \( C + S < 2^n \). By theorem 2.2, when \( C + S = 2^n - 1 \) then \( S = 2^n - 1 \) and \( C = 0 \). The first clock pulse after \( S = 2^n - 1 \) causes a carry into bit \( c_1 \), the second pulse causes a carry into bit \( c_2 \), and the \( n \)th pulse causes \( c_n \) to change from 0 to 1. At this point \( C + S = 2^n - 1 + n \). □

**Example:** The first 10 entries in the left-hand side of table 3 illustrate lemma 3.1 for \( n = 6 \). Initially, \( C + S = 60 < 63 \). At time 3, \( C + S = 63 = 2^6 - 1 \). At time 9, which is \( n = 6 \) steps later, there is a carry into bit \( c_6 \).

Lemma 3.1 tells us what the sum of two words is when a carry is generated. By lemma 2.4, we can also tell when there will be a carry into any of the lower-order bits.

**Definition:** An \( n \)-bit word \( A \) is the *prefix-and* of \( S \) means that for each bit \( a_i \) of \( A \), \( a_i = 1 \) if and only if \( s_{n-1} = s_{n-2} = \ldots = s_i = 1 \).

**Example:** If \( S = 111011 \), then \( A = 111000 \).
Lemma 3.2 Let $A$ be the prefix-and of $S$. If $a_j = 1$, then the value of $a_j$ does not change until after $C + S = 2^n - 1$.

Proof: Since $a_j = 1$, then $s_j = s_{j+1} = \ldots = s_{n-1} = 1$, and by the definition of h-a form, $c_j = c_{j+1} = \ldots = c_{n-1} = 0$. Since initially $C + S < 2^n - 1$, the initial sum of the low order $j$ bits of $C$ and $S$ is less than $2^j - 1$. By lemma 2.4, the low order $j$ bits of $C$ and $S$ are also in h-a form. By lemma 3.1, $c_j \neq 1$ until after all of the low order $j$ bits of $S$ are 1, i.e., after $C + S = 2^n - 1$. □

Lemma 3.2 says that $A$ is nondecreasing until the terminating condition for the count is achieved. Lemma 3.3 establishes the number of pulses that occur between the setting of two adjacent bits of $A$.

Lemma 3.3 Let $A$ be the prefix-and of $S$. If $a_{j+2} = 1$, $a_{j+1} = 0$, and $C+S < A+2^{j+1}$, then bit $s_j$ is set $2^j - 1$ clock pulses after bit $s_{j+1}$ is set.

Proof: $C + S < A + 2^{j+1}$ and $a_{j+2} = 1$ implies that the low order $j + 2$ bits of of $(C, S)$ sum to less than $2^{j+1}$. Applying lemma 3.1 to these bits we see that bit $c_{j+1}$ is set when the count in those bits is $2^{j+1} + (j + 1) - 1 = 2^{j+1} + j$. By equation 2.7, bit $s_{j+1}$ is set one pulse later, when the count in those bits is $2^j + j + 1$.

At this point, the count in the low order $j + 1$ bits is $j + 1$, which is less than $2^j$ for $j > 1$. Applying lemma 3.1 to the low order $j + 1$ bits of $(C, S)$, we see that bit $c_j$ is set when the count in those bits is $2^j + j - 1$, and bit $s_j$ is set one pulse later when the count in those bits is $2^j + j$. This occurs exactly $2^j + j - (j + 1) = 2^j - 1$ clock pulses after bit $s_{j+1}$ is set. □
Example: Table 4 shows the number of pulses received when each of the bits of \( S \) change from 0 to 1 in table 3. Column three gives the value of \( 2^j - 1 \) for each \( j \), which we can see is exactly the number of pulses received between the setting of bit \( j + 1 \) and bit \( j \).

Table 4: Time at which each bit of \( S \) is set when counting to 67

<table>
<thead>
<tr>
<th>bit(j)</th>
<th>time</th>
<th>( 2^j - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>41</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>63</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>67</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>67</td>
<td>0</td>
</tr>
</tbody>
</table>

Lemma 3.3 says that except for the two low-order bits, there is sufficient time between the setting of adjacent bits to allow an iterative comparison to establish whether or not \( S = 2^n - 1 \). Together, the lemmas in this section allow for constant-time counting and detection.

Theorem 3.1 For an \( n \)-bit incrementer constructed as in figure 5, and for any count \( k, 1 \leq k \leq 2^n \), detection of the terminating condition \( C + S = 2^n - 1 \) can be done in constant time.

Proof: We will show that if \( A \) is initialized to be the prefix-and of \( S \), then \( A \) can be maintained as the prefix-and of \( S \) in constant time. This reduces the test for the terminating condition to a constant-time test of whether or not \( a_0 = 1 \).
Initially $(C, S) = (0, \bar{k})$ and $A$ is set to the prefix-and of $\bar{k}$ ($A$ must be precomputed).

On each clock pulse, let $a_{n-1} = s_{n-1}$, $a_{n-2} = s_{n-1} \land s_{n-2}$, and $a_i = s_i \land s_{i+1} \land a_{i+2}$ for $i = 0, 1, \ldots, n - 3$. Since the count is constantly increasing, we must have $s_{n-1} = 1$ after zero or more clock pulses. By lemma 3.2, the number of bits that are set in the prefix-and of $S$ is nondecreasing until after the terminating conditions are met. By lemma 3.3, the number of bits that are set in the prefix-and of $S$ cannot increase by more than two on any clock pulse. Suppose $a_{i+1} = 1$, $a_i = 0$, and on the next clock pulse $s_i$ and $s_{i-1}$ are both set. Then $a_i = a_{i+2} \land s_{i+1} \land s_i$ and $a_{i-1} = a_{i+1} \land s_i \land s_{i-1}$ are set after the delay of a three-input AND gate. This completes the proof since the delay of a 3-input AND gate is constant. □

Figure 6: Constant time detector

Figure 6 is a 9-bit detector constructed as in the proof of theorem 3.1. Since we are using 3-input AND gates, only the even bits of $A$ have to be computed.
The precomputation of $A$ to the prefix-and of $k$ is only necessary when counting to very small numbers on large counters. For example, if $k \geq n/2$, which is not a very severe restriction on a counter that can handle a modulus of $2^n$, then we can safely initialize $A$ to 0. By the time $n/2$ clock cycles have passed, $A$ will contain the correct prefix-and of $S$.

3.3 Enhancements

The circuits discussed in this chapter create an output pulse $k$ clock pulses after initialization. If we need an output pulse after every $k$ clock pulses, then the counter should be initialized to count to $k - 1$, and the output pulse should be used to reinitialize the counter.

A variation of the frequency divider is a counter that produces a fixed sequence of output pulses. Instead of producing an output every $k$ cycles, the counter is required to cycle through $m$ different output values $(K_0, K_1, \ldots, K_{m-1})$. A small modification to the frequency divider in figure 5 allows it to perform this function. The values $(K_0, K_1, \ldots, K_{m-1})$ are stored in a memory. Initially, $(C, S) = (0, K_0)$. Whenever the $i$th output pulse is generated, $(C, S)$ is set to $(0, K_i)$.

The proofs in section 3.2 assume that every clock pulse causes the counter to increment, but this is not necessary. Sometimes the increment is not tied to the clock, and an input value could be zero. This can easily be accommodated by changing the fixed “1” that is input to the low-order half adder to an input that is zero or one. The output pulse will now be generated when $k$ ones have been received. Constant time
detection will still work. Indeed the worst case for constant-time detection occurs when the input is always one – if the input is sometimes zero then there is more time to do the detection.

3.4 Comparison to Other Constant-Time Counters

As noted in section 1.3, Ercegovac and Lang have produced the only comparably fast arbitrary length frequency divider [EL89]. They solve the carry propagation problem by dividing the counter into subcounters of increasing lengths. Each subcounter has two registers, a ring counter, and an incremeneter. The first register holds the current count. The second register holds the current count plus 1. A ring counter of length $r$ is incremented on each clock pulse. When the ring counter reaches $r$, the value in register 2 is moved to register 1, the incremeneter (which is constructed out of half adders) is used to increment the value in register 2, and the ring counter continues to increment. The lengths of the subcounters and ring counters are picked so that the incremeneters have time to complete the worst case carry propagation before the ring counter reaches $r$. Since the next count value is always precomputed, the correct count is always available.

To count to an arbitrary value $k$, subvalues corresponding to one less than what each subcounter will hold when the count reaches $k$ are computed. A latch is set when each subvalue is reached and the ring counter for the corresponding subcounter reaches $r$. When the latches are set for all of the subcounters, an output pulse is
generated. Since the high order subcounters finish before the low order subcounters, the conditions for the output pulse can be detected in constant time.

The situation is somewhat more complicated when the value to be detected in a subcounter ends in more than one zero. In this case hardware changes have to be made to allow detection of two less than what the subcounter will hold when the count reaches $k$.

This frequency divider has some disadvantages with respect to ours. It uses more than twice as many latches and occupies more than twice the area. It is much harder to program, and as noted above, requires hardware changes in order to count to certain values. Recall that our counter is programmed to output a pulse after $k$ clock pulses simply by setting $(C, S) = (0, k)$.

Vuillemin improved on Ercegovac and Lang's counter by reducing the number of latches and the area [Vui91], but only for the modulo-$2^n$ case. Vuillemin's counter is similar to Ercegovac and Lang's in that both rely on a staged construction, with the stages for the low order bits being smaller than the stages for the high order bits. Both counters also precompute the next count for each stage, and then instantiate the new count when the time comes for a carry into the stage. Vuillemin achieves his reduction in latches by doing a clever timing analysis that shows that the ring counters are not necessary, and that they can be replaced with a few AND gates. This reduction in hardware comes at the expense of some added complexity at the design stage.
Changing Vuillemin's counter to handle arbitrary moduli would require adding comparison circuitry, and it would probably end up looking very much like the frequency divider in [EL89], although with fewer latches. It would still require more circuits and area than our design, and would suffer from all of the other disadvantages listed above.

A major difference between these counters and ours is that, for our counter, the state of the counter is determined by the count (and possible the state of the prefix-and when the value in the counter is very near to \(2^n - 1\)). Both Ercegovac and Lang's counter and Vuillemin's counter have additional state that is not directly accessible. This extra state makes these counters difficult to program, and in fact precludes programming for arbitrary counts without changing hardware.

Both of these counters have the advantage of maintaining the count in non-redundant form, although for frequency dividers, we believe this to be unnecessary. If we need to know when the count has reached certain values, then we can use the method discussed in the second paragraph of section 3.3 to generate output pulses on those values. Indeed, that method is much faster than any method that has to read the count and compare it to a value.
CHAPTER IV

Carry Generation Detection, Comparison, and Addition

Our most important application of half-adder form is to the theory of carry generation detection, and especially to the construction of adders. Addition is the most fundamental operation in computers. All of the basic integer and floating point arithmetic functions require binary addition, and most comparisons are performed using addition. Every operation requires an adder to update the program counter, and many operations require an adder to generate memory addresses. Furthermore, the latency of addition appears to be closely tied to the cycle time of the fastest processors, a relationship we will explore in chapter VI.

The organization of this chapter is as follows: Section 4.1 presents two new algorithms for determining whether or not a sum in half-adder form generates a carry. It also compares these algorithms with the two best existing algorithms for performing the same task. Section 4.2 uses the algorithms in section 4.1 to improve the speed of comparison. Section 4.3 discusses the addition problem, and gives possible constructions for 16, 32, and 64 bit adders. Finally, section 4.4 discusses some issues that are not measured by our model, but which are important in implementations.
4.1 Carry Generation Detection

The surprising result of this section is that if \((C, S)\) is in h-a form, determining whether or not \(C + S\) generates a carry depends on only one bit of \(C\), and that the location of that bit is completely determined by \(S\). Consider the structure of \(S\): either \(S = -1\), or else it consists of a sequence of zero or more high order one bits followed by a zero bit. Suppose this high order zero bit is at position \(i - 1\). We will show in theorem 4.1 that bit \(c_i\) determines whether or not \(C + S\) generates a carry: the rest of \(C\) can be ignored.

Recall from chapter III that when we say an \(n\)-bit word \(A\) is the prefix-and of \(S\), we mean that for each bit \(a_i\) of \(A\), \(a_i = 1\) if and only if \(s_{n-1} = s_{n-2} = \ldots = s_i = 1\).

**Theorem 4.1** Let \((C, S)\) be in h-a form, and let \(A\) be the prefix-and of \(S\). If \(A = 0\) then there is no carry-out of \(C + S\). Otherwise, let \(i\) be the lowest order bit such that \(a_i = 1\). Then \(c_i = 1 \Leftrightarrow \) there is a carry-out of \(C + S\).

**Proof:** If \(A = 0\), then \(s_{n-1} = 0\) and an easy induction shows that there can be no carry-out of \(C + S\).

So suppose that \(A \neq 0\).

\([\Rightarrow]\) \(a_i = 1\) means that \(s_{n-1} = s_{n-2} = \ldots = s_i = 1\). Since \(c_i = 1\), and since the carry propagates through \(s_i, s_{i+1}, \ldots, s_{n-1}\), then there is a carry-out of \(C + S\).

\([\Leftarrow]\) By the definition of \(A\), \(s_i = s_{i+1} = \ldots s_{n-1} = 1\). By lemma 2.3 (section 2.3), only one of \(s_k\) and \(c_{k+1}\) can be set, so \(c_{i+1} = c_{i+2} = \ldots = c_{n-1} = 0\), and no carry is
generated at positions $i + 1, i + 2, \ldots, n - 1$. Since $s_{i-1} = 0$, there is no carry into position $i$. Since there is a carry-out of position $n - 1$, we must have $c_i = 1$. □

**Example:** Given $X$ and $Y$ as in equations 4.1 and 4.2, then $C$, $S$, and $A$ are given in equations 4.3, 4.4, and 4.5. Note that there is exactly one transition from 1 to 0 in $A$, and that the carry out from $C + S$ (a carry is generated in this example) is determined by the corresponding bit of $C$.

$$X = \begin{align*}
  &1101101100111001 \quad (4.1) \\
  &Y = \begin{array}{c}
  0010010011001111
  \end{array} \quad (4.2)
  \end{align*}$$

$$C = \begin{array}{c}
  000000000010010
  \end{array} \quad (4.3)$$

$$S = \begin{array}{c}
  11111111110110
  \end{array} \quad (4.4)$$

$$A = \begin{array}{c}
  1111111111000
  \end{array} \quad (4.5)$$

4.1.1 The Prefix-And Method

Our first solution to the carry-generation detection problem is a direct application of theorem 4.1.

**Algorithm 4.1** Let $X$ and $Y$ be $n$-bit numbers, and let $c_{in}$ be an optional carry-in bit. To determine whether there is a carry-out from $X + Y + c_{in}$, do the following:

1. Compute $(C, S) = ha(X, Y)$.

2. Compute the prefix-and $A$ of $S$
3. Do the following in parallel:

- if $a_{n-1} = 0$ then return $c_n$.
- if $a_0 = 1$ then return $c_{in}$.
- For $i$ in $1, 2, ..., n-1$ do in parallel
  
  if $a_i \land \overline{a_{i-1}} = 1$ then return $c_i$

Step 1 of the above algorithm can be implemented by a single-level circuit consisting of $n$ AND gates and $n$ XOR gates. Step 2 can be implemented in $\log n$ levels by using a parallel prefix circuit [LF80]. Step 3 can be implemented in two levels, the first of which computes the logical function $a_i \land \overline{a_{i-1}}$, and the second of which uses tristate buffers to select one of the $c_i$ based on the values obtained in the first level. This works because by theorem 4.1, exactly one of the $n + 1$ if statements in step 3 is true. The total delay is $\log n + 3$, which is close to the theoretical lower bound of $\log n + 2$ that is imposed by fanin considerations (including $c_{in}$, there are $2n + 1$ inputs).

Figure 7 shows an 8-bit circuit based on this method. To simplify the diagram, the computation of $C$ is not shown (we will soon show that $C$ does not have to be computed). $S$ is computed by the 8 XOR gates at level 1. The dotted box performs an 8-bit parallel prefix-and at levels 2 through 4, the AND gates at level 5 determine the position $i$ of the lowest order 1 in the prefix-and and the tristate node at level 6 transmits $c_i$ to the output. If there is no 1 in the prefix-and, then $c_8$ is output.
Figure 7: 8-bit prefix-and carry generation detection

Note that the problem solved in six logic levels in figure 7 is an 8-bit carry generation with a carry-in bit. The fastest competing method for this problem is carry-lookahead, which we discuss in the next section. Carry-lookahead requires eight logic levels for this problem. It is interesting to observe that carry lookahead algorithms can also be viewed as using prefix computations [BK82, LF80]. In both cases log \(n\) steps are required, but our algorithm is asymptotically nearly twice as fast because each step is a 2-input AND as opposed to a 2-input AND followed by a dependent 2-input OR.
We end this section with a minor simplification to our algorithm. Since we are only using one bit of $C$, and since we can prove that the bits of $X$ and $Y$ that produce that bit have identical values, we do not have to compute $C$.

**Corollary 4.1** (to theorem 4.1) Let $S = X \oplus Y$, let $x_{-1}$ be the carry-in bit (0 if there is no carry-in), and let $A$ be the prefix-and of $S$. If $a_{n-1} = 0$ then set $i = n$. Otherwise let $i$ be the lowest order bit such that $a_i = 1$. Then $x_{i-1} = 1 \Leftrightarrow$ there is a carry-out of $X + Y$.

**Proof:** Suppose $i$ is the bit such that $a_i \land \overline{a_{i-1}} = 1$. Then $s_{i-1} = 0$, and so $c_i = x_{i-1} = y_{i-1}$. □

To apply corollary 4.1 to figure 7, simply replace any bit labeled $c_i$ with $x_{i-1}$.

In the next section, we examine both carry lookahead addition, which up until now has been the best practical method for carry generation detection, and Brent’s method, which has been the best theoretical method. These methods will help to put the prefix-and method in context, and more importantly, will enable us to improve the prefix-and method.

### 4.1.2 The Carry-Lookahead Method and Brent’s Method

Carry-lookahead adders calculate the value at each bit based on carry propagate and generate signals defined for $i \leq j < k$ as follows:

\[
p_{i,j} = x_i \oplus y_i \quad (4.6)
\]

\[
g_{i,j} = x_i \land y_i \quad (4.7)
\]
\[ p_{i,k} = p_{i,j} \land p_{j+1,k} \]  \hspace{1cm} (4.8)

\[ g_{i,k} = g_{j+1,k} \lor (p_{j+1,k} \land g_{i,j}) \]  \hspace{1cm} (4.9)

The propagate bit \( p_{i,k} \) indicates that a carry into bit \( i \) will propagate through bit \( k \), and the generate bit \( g_{i,k} \) indicates that a carry is generated by the sum of the bits in positions \( i \) through \( k \). Most discussions of carry-lookahead addition define \( p_{i,i} \) as \( x_i \lor y_i \). This is technically incorrect, because if \( x_i = y_i = 1 \) then a carry is not propagated through position \( i \). However, a carry is generated at position \( i \), and it is easy to see that this simplification makes no difference in any result \( g_{i,k} \). Since the \( g_{i,k} \) are the only results that matter in carry-lookahead addition, and since \( \text{OR} \) is usually easier to implement than \( \text{XOR} \), the incorrect definition is more popular than the correct definition. We require the correct definition, \( p_{i,i} = x_i \oplus y_i \), because we need the true values of \( p_{i,k} \). Since our model charges the same for an \( \text{XOR} \) and an \( \text{OR} \), this change does not affect any complexity results.

Note that if \((C, S) = ha(X, Y)\), and \( A \) is the prefix-and of \( S \), then

\[ p_{i,i} = s_i \]  \hspace{1cm} (4.10)

\[ p_{i,n-1} = a_i \]  \hspace{1cm} (4.11)

\[ g_{i,i} = c_{i+1} \]  \hspace{1cm} (4.12)

If \( p_{i,k} \) is set, then if there is a carry into bit \( i \) of \((C, S)\), it will propagate from bit \( i \) to bit \( k \). If \( g_{i,k} \) is set, then when \( C \) and \( S \) are added, there is a carry out of bit \( k \).

The delay for detecting whether there is a carry out of \( X + Y + c_{in} \) is \( 2\log n + 2 \), almost twice as slow as Winograd’s lower bound[Win65].
An earlier paper by Brent [Bre70] suggests a faster way. The method is to calculate part of the prefix-and $A$ of $P$, while simultaneously computing the largest generate groups that can be computed with the same delay as $A$. As soon as the necessary subset of $A$ has been computed, we can dispense with the (relatively slow) calculation of the $g_{i,j}$, and simply use a tree of OR gates to see if any of the $g_{i,j} \land a_j = 1$.

Figure 8 shows a 64-bit circuit based on Brent’s method. Only the bits of $A$ that are used as inputs to the final tree of OR gates need to be computed, which reduces the

Brent’s notation is very different from that given here, and he uses the alternate definition for $p_{i,i}$, so it is not obvious the circuit presented here is essentially the same as his. We only realized the similarity when we discovered that the time bounds are identical.
size of the prefix-and problem. Each \( p_{i,i+7} \) is the 8-input AND of \( p_{i,i}, p_{i+1,i+1}, \ldots, p_{i+7,i+7} \), and each \( g_{i,i+7} \) can be computed by any method that finishes by the time \( A \) has been computed. It takes three units of delay to compute the inputs to the prefix-and, and another three units of delay to compute the prefix-and, so the \( g_{i,i+7} \) are not needed until level 7. This means that the \( g_{i,i+7} \) can be computed by definition as in figure 9.

![Figure 9: 8-bit calculation of generate step](image)

The time required for this circuit as shown is \((3/2)\log n + 2\): with one step to compute the propagate and generate inputs, \( \log n \) steps to compute \( A \), one step to compute \( g_{i,j} \land a_j \), and then \( \log \sqrt{n} \) steps to compute the tree of OR gates. The size of the tree of OR gates is based on the size of the groups \( g_{i,j} \) that can be computed in the...
time it takes to compute $A$. Using equation 4.9 to produce the $g_{i,j}$, as we have done in figure 9, allows us to compute groups of size $\sqrt{n}$. The asymptotic time bound is better than $(3/2) \log n + 2$ because for large $n$, we can use Brent's method recursively to calculate larger generate groups in the allotted time. Using the results in [Bre70], it is straightforward to show that the asymptotic delay is $\log n + O(\sqrt{\log n})$. While this is clearly better than the delay of $2 \log n + 2$ required by carry-lookahead, it is slower and more complicated than our prefix-and method.

4.1.3 The Partial Prefix-And Method

The prefix-and circuit given in section 4.1.1 is cumbersome for large $n$, and it does not provide partial results that can be reused when carries have to be computed for many bit positions. Our second solution, a combination of the prefix-and method with some of the ideas from carry-lookahead addition, remedies both of these problems.

Suppose we have produced propagate and generate bits according to equations 4.6 and 4.7, and suppose that these bits are grouped into contiguous disjoint groups. If we use equations 4.8 and 4.9 to produce propagate and generate bits for each such group, then we end up with smaller propagate and generate words. We will show that these new words are closely related to h-a form, and that we can use theorem 4.1 to find out if the new words (and by extension, the old words) generate a carry.

The proof begins with a lemma that shows that the group propagate and generate signals for any group of consecutive bits cannot both be 1.

**Lemma 4.1** Given $i \leq k$, $p_{i,k} \land g_{i,k} \neq 1$. 
Proof: The proof is by induction on the length of the interval \((i, k)\). By equations 4.6 and 4.7 this is true for \(i = k\).\(^3\) So assume it is true for all intervals of length less than \(k - i\). For \(k > i\), there exists \(j\) such that \(i \leq j < k\). By equation 4.8 and 4.9, 
\[
pi,k = pi,j \land p_{j+1,k} \quad \text{and} \quad gi,k = g_{j+1,k} \lor (p_{j+1,k} \land gi,j).
\]

If \(pi,k \neq 1\), then we are done, so assume \(pi,k = 1\). Then \(pi,j = 1\) and \(p_{j+1,k} = 1\). By the induction hypothesis, \(g_{i,j} = g_{j+1,k} = 0\), and so \(gi,k = g_{j+1,k} \lor (p_{j+1,k} \land gi,j) = 0\). □

Theorem 4.2 Let \(k_0 = 0, k_1, k_2, \ldots, k_{m-1} = n - 1\) be a strictly increasing sequence of numbers. Set \(c_0 = 0\) and for \(i = 0, \ldots, m - 1\) let \(s_i = p_{k_i,k_{i+1}-1}\) and \(c_{i+1} = g_{k_i,k_{i+1}-1}\). Then \((C, S)\) is in \(m\)-bit h-a form.

Proof: By lemma 2.3, it suffices to show that \(s_i \land c_{i+1} \neq 1\). This is immediately true by lemma 4.1. □

Theorem 4.2 allows us to apply theorem 4.1 recursively to determine whether or not a sum generates a carry. As an example, we construct a 16-bit group propagate and generate detector (PG16) out of 4-bit group propagate and generate detectors (PG4).

Our convention for PGk blocks will be that the propagate bits enter at the top, and the generate bits enter at the right. The \(k\)-bit AND of the propagate bits is output at the left, and the \(k\)-bit group generate bit is output at the bottom. If a carry-in bit \((c_{in})\) is not shown for a block, it is hard-wired to 0.

\(^3\)This would not be true if we had defined \(pi,i = x_i \lor y_i\).
Figure 10: Module PG4: 4-bit prefix-and propagate and generate

Figure 10 shows module PG4. The prefix-and of the propagate bits is computed in the first two levels. At level three, the AND gates locate any transitions from 1 to 0 in the prefix-and and at level 4 the appropriate generate bit is selected.

Figure 11 shows a 16-bit propagate and generate module PG16 constructed out of two stages of PG4 modules. We will see that the two stages overlap in time, so that the generate outputs are available after passing through 6 logic levels, even though each stage contains 4 logic levels. The input propagate and generate bits are divided into 4-bit groups. Note that the thicker gray lines in our figures represent multiple wires.
Figure 11: Module PG16: 16-bit propagate and generate constructed out of PG4 modules

(four in this case), as opposed to the thin black lines which represent single wires. At levels 1 and 2, the PG4 modules in the first stage compute the 4-bit prefix-and of their input propagate bits, and output the AND of all four propagate bits. At levels 3 and 4, the second stage PG4 module computes the 4-bit prefix-and of these output bits, and the first stage PG4 modules compute the 4-bit group generate output. At the end of level 4, the 4-bit group generate bits are available. At level 5, the second stage PG4 module locates the transition point from 1 to 0 in its prefix-and and at level 6 the appropriate group generate bit from the PG4 modules in the first stage is selected.
Replacing the first stage PG4 modules in figure 11 with PG16 modules produces PG64, a 64-bit propagate and generate module. In general, a $4^k$-bit propagate and generate unit can be constructed out of $k$ stages of PG4 modules, and it will produce its generate output in $\log_2(4^k) + 2 = 2k + 2$ logic levels (an additional level is required to produce the input $p_{i,i}$).

We refer to this method as the partial prefix-and method because only part of the prefix-and is computed. It has the same delay as the prefix-and method, but is easier to lay out and requires fewer gates because only part of the prefix-and is computed (Detailed comparisons of the methods are given in section 4.1.4). It is also much easier to use in the construction of adders, as we will see in section 4.3.

The reader may wonder why the same construction does not work for the usual carry-lookahead method. Ignoring the carry-in bit, carry-lookahead can also be used in a PG4 module that has only 4 logic levels. The crucial difference is that in our construction, the generate inputs are only used in the final logic level of each stage, while in the usual carry-lookahead method, the generate inputs are used in every logic level in every stage.

Table 5: Stages and logic levels for a PG64 module constructed from PG4 modules

<table>
<thead>
<tr>
<th>Bit</th>
<th>Stage</th>
<th>logic level produced</th>
<th>logic level required</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit propagate</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>16-bit propagate</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>64-bit propagate</td>
<td>3</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>4-bit generate</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>16-bit generate</td>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>64-bit generate</td>
<td>3</td>
<td>8</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 5 lists the stages and logic levels at which the various propagate and generate bits are produced and used for a PG64 module constructed out of PG4 modules. For example, the 16-bit generate bit is produced in stage 6, but is not required until stage 8. The table clearly shows that 4 and 16-bit generate computations are not on the critical path. For modules constructed out of larger PGk modules there is even more slack: a generate bit produced at level $j$ is not used until level $j + \log k$.

For technologies in which larger tristate nodes require more than unit time, this slack means that the tristate nodes will not slow down the computation except for a constant factor in the last level.

### 4.1.4 Evaluation of the Four Methods

We compare the costs and delay of the four algorithms in two ways: first, the asymptotic analysis, and second, analysis for $n \leq 64$.

In order to account for a carry-in bit in our analysis of Brent’s method or the carry-lookahead method, we have added one additional logic level and 2 additional gates. This carry-in is free for either of the prefix-and methods, but requires extra delay for the other methods when the words have size $2^k$. Since this is the most common word size, we have included the extra delay in our analysis. Our cost and delay estimates also include the time required to produce any required propagate and generate inputs.

We use gates as our cost measure. Other measures, such as area and design complexity, are preferable, but for the designs we propose these measures are completely dependent on the layout of the prefix-and, which is outside of the scope of this thesis.
Apart from the prefix-and, our two designs are very compact and regular, consisting almost entirely of linear arrays of identical gates connecting adjacent gates. Because of this regularity, even the naive prefix-and shown in figure 7 would result in circuits using a smaller area than that used by carry-lookahead or Brent’s method.

Most of the work in the prefix-and methods (and to a lesser extent, in Brent’s method) is in computing the prefix-and. The circuit in figure 7 uses the simplest implementation of parallel prefix. It computes its results in \( \log n \) steps, but uses \((n/2)\log n\) gates, some with large fanout. More sophisticated implementations are available. For large \( n \), [LF80] show how to limit the size of the circuit to \( 4n \) without increasing the time bound. And while it is not relevant in our model, the technique of recursive doubling [KS73] gives a \( \log n \) step prefix-and that has a fanout limited to 2 at any level. This last technique, an example of which is given in section 4.4, may be faster in some technologies.

The costs are evaluated as follows:

1. Calculation of \( p_i \) (or \( s_i \)): \( n \) XOR gates.

2. Calculation of \( g_i \) (or \( c_i \)): \( n \) AND gates.

3. Calculation of a \( k \)-bit prefix-and: \( \min(4k, (k/2)\log k) \) AND gates. The first number is an asymptotic bound from [LD90], while the second number is from a naive algorithm that works well for small \( k \).

4. Calculation of a \( k \)-bit OR : \( (k - 1) \) OR gates.

5. Calculation of a \( k \)-bit group propagate: \( (k - 1) \) AND gates.
6. Calculation of a $k$-bit group generate given all necessary propagate values: $(2k - 2)$ gates ($k - 1 \text{ AND and } k - 1 \text{ OR}$).

7. Calculation of the carry bit using \text{AND} gates and a $k$-input tristate node: $2k$ gates ($(k - 1) \text{ AND gates and } (k + 1) \text{ tristate buffers}$).

The carry lookahead method involves computing $p_i$ and $g_i$, an $n$-bit group propagate, and an $n$-bit group generate, for a total cost of $5n - 3$.

The exact asymptotic cost for Brent’s method is difficult to establish because it uses an irregular recursion. Brent claims that the cost is less than $7n$. For $n \leq 64$, Brent’s method involves computing $p_i$ and $g_i$, \sqrt{n} calculations of a \sqrt{n}-bit group propagate, calculation of a \sqrt{n}-bit prefix-and, \sqrt{n} calculations of a \sqrt{n}-bit group generate, \sqrt{n} \text{ AND gates to combine the propagate and generate bits, and a } \sqrt{n} \text{-bit OR, for a total cost of } 5n - \sqrt{n} + (\sqrt{n}/2) \log \sqrt{n} - 1.$

The prefix-and method involves computing $s_i$ (by corollary 4.1, $c_i$ is not required), an $n$-bit prefix-and, and calculation of the final carry bit using an $n$-input tristate node, for a total cost of $7n$. For small $n$, the cost is $3n + (n/2) \log n$.

The partial prefix-and method involves computing 4 problems of size $n/4$, a 4-bit prefix-and, and calculation of the final carry bit using a 4-input tristate node, for a total \textit{cost}(n) = $4 \times \textit{cost}(n/4) + 12$. The basis of this recursive cost measure can be any of the other methods. Using the prefix-and method for the recursion basis ($n = 4$, \textit{cost}(4) = 16), gives an asymptotic cost of $5n - 4$. 


Table 6: Asymptotic delay and cost for group carry-generation detection

<table>
<thead>
<tr>
<th>Method</th>
<th>Delay</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carry-Lookahead</td>
<td>$2 \log n + 2$</td>
<td>$5n - 1$</td>
</tr>
<tr>
<td>Brent</td>
<td>$\log n + O(\sqrt{\log n})$</td>
<td>$&lt; 7n$</td>
</tr>
<tr>
<td>Prefix-And</td>
<td>$\log n + 3$</td>
<td>$&lt; 7n$</td>
</tr>
<tr>
<td>Partial Prefix-And</td>
<td>$\log n + 3$</td>
<td>$5n - 4$</td>
</tr>
</tbody>
</table>

Table 6 summarizes the asymptotic cost and delay for the four methods. The partial prefix-and method is clearly superior to the other methods in the asymptotic case.

Table 7 shows the cost and delay for $n = 4, 16, 64$. Again the partial prefix-and method is superior, although the regularity and simplicity of the prefix-and method make it attractive for small $n$. The exact tradeoff point between the two methods is dependent on the implementation and technology used.

Table 7: Carry generation delay and cost for some small values of $n$

<table>
<thead>
<tr>
<th>Method</th>
<th>$n=4$</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Steps</td>
<td>Gates</td>
<td>Steps</td>
<td>Gates</td>
<td>Steps</td>
<td>Gates</td>
</tr>
<tr>
<td>Carry-lookahead</td>
<td>6</td>
<td>19</td>
<td>10</td>
<td>79</td>
<td>14</td>
<td>319</td>
</tr>
<tr>
<td>Brent</td>
<td>6</td>
<td>20</td>
<td>9</td>
<td>81</td>
<td>12</td>
<td>325</td>
</tr>
<tr>
<td>Prefix-And</td>
<td>5</td>
<td>16</td>
<td>7</td>
<td>80</td>
<td>9</td>
<td>384</td>
</tr>
<tr>
<td>Partial Prefix-And</td>
<td>NA</td>
<td>NA</td>
<td>7</td>
<td>76</td>
<td>9</td>
<td>316</td>
</tr>
</tbody>
</table>

4.2 Comparison

One of the simplest applications of carry generation detection is to the problem of comparison. We typically think of comparisons as being centered around the number
0. Because of theorem 2.2, it is faster and simpler to examine the relationship of the numbers with respect to \(-1\).

**Assumption:** For the proofs in this section and in chapter V, we assume that in the absence of overflow, two's complement arithmetic gives the same results as integer arithmetic. This simplifies our proofs by allowing us to consider just two cases: one with overflow, and one without overflow.

**Theorem 4.3** Let \( X \) and \( Y \) be two's complement numbers. Then \( X = Y \Leftrightarrow X + \overline{Y} = -1 \). Similarly, \( X > Y \Leftrightarrow \) the \((n+1)\)-bit sum \( X + \overline{Y} > -1 \).

**Proof:**

\[
X = Y \Leftrightarrow X - Y = 0 \tag{4.13}
\]
\[
\Leftrightarrow X + \overline{Y} + 1 = 0 \tag{4.14}
\]
\[
\Leftrightarrow X + \overline{Y} = -1 \tag{4.15}
\]

The result for inequalities is slightly more complicated because for inequalities we have to consider the possibility of overflow.

If there is no overflow,

\[
X > Y \Leftrightarrow X - Y > 0 \tag{4.16}
\]
\[
\Leftrightarrow X + \overline{Y} + 1 > 0 \tag{4.17}
\]
\[
\Leftrightarrow X + \overline{Y} > -1 \tag{4.18}
\]

If there is overflow, then relationship 4.16 fails because overflow causes a positive difference to appear negative, and a negative difference to appear positive. A solution
to this problem is to include an extra bit in the computation so that the difference cannot overflow. (Another solution would be to reverse the sense of the comparison in the presence of overflow). Only one extra bit is required because the maximum difference between any two $n$-bit two's complement numbers is $2^n - 1$. This difference cannot overflow as long as we compute it using $(n+1)$-bit two's complement numbers.

The conversion from $n$-bit two's complement numbers to $(n+1)$-bit two's complement numbers is accomplished by extending the sign bit, i.e., if $X = x_{n-1} \ldots x_1 x_0$, then the $n+1$ bit representation of $X$ is $X' = x_n x_{n-1} \ldots x_1 x_0$, where $x_n = x_{n-1}$. Let $X'$ and $Y'$ be the sign-extended $(n+1)$-bit representation of $X$ and $Y$, let $(C, S) = ha(X, Y)$, and let $(C', S') = ha(X', Y')$. The sign of $X' + Y'$ is given by bits $c'_n, s'_n$, and the carry-out of the sum of the low order $n$ bits of $C'$ and $S'$. All of these values can be computed directly from $C$ and $S$ since $c'_n = c_n$, $s'_n = s_{n-1}$ and the carry out of the sum of the low order $n$ bits of $C'$ and $S'$ is equal to the carry out of the sum of $C$ and $S$.

Thus all of the information needed to compare $X$ and $Y$ is contained in $(C, S) = ha(X, Y)$. This is significant because $ha(X, Y)$ can be computed using only one unit of delay. The following algorithm implements the comparison method of theorem 4.3, using the fast equality check from theorem 2.2 to test whether $X = Y$, and the fast carry generation of theorem 4.1 to computer whether $X > Y$.

**Algorithm 4.2** *Compute whether $X > Y$ or $X = Y$.*

---

4Recall that NOT gates are not counted in our model, since any 2-input logical function involving a negated input can be translated into a function that requires only the non-negated input.
1. Compute \((C, S) = ha(X, Y)\).

2. Use algorithm 4.1 to find \(c_{out}\), the carry out of \(C + S\). Also compute \(e_{xy}\), the AND of \(S\).

3. Set \(sign = c_{out} \oplus s_{n-1}\).

If \(e_{xy} = 1\), then \(X = Y\), and if \(sign = 0\), then \(X > Y\).

Step 1 of the above algorithm requires one logic level. Step 2 of the algorithm requires \(\log n + 2\) logic levels to compute \(c_{out}\). The value \(e_{xy}\) is computed in \(\log n\) logic levels as part of the computation of \(c_{out}\). Steps 3 requires one logic level. The total delay for setting the sign and equality bits is \(\log n + 4\) logic levels. This is sufficient for determining any of the standard relationships \((<, \leq, =, \neq, \geq, >)\), although some of these relationships require one additional logic level.

### 4.3 Addition

Recall our formulation of the addition problem: given two \(n\)-bit two's complement numbers \(X\) and \(Y\), and a carry-in bit \(c_{in}\), the addition problem is to compute an \(n\)-bit two's complement number \(T = X + Y + c_{in}\), plus a carry-out bit \(c_{out}\). The carry-in bit is needed for subtraction, and the carry-out bit is needed for overflow detection. Both bits are also useful if one wishes to perform multiword-length additions.

Given a fast carry generate detector, as in figure 7 or figure 11, it is theoretically straightforward to construct a fast solution to the addition problem. For each bit position \(i\), find out whether the group consisting of all lower-order bits generates a
carry. The final sum bit is the XOR of \( s_i \) and this carry bit. The final sum can be computed in \( \log n + 4 \) levels: \( \log n + 3 \) levels to generate the longest group carry, and one level for the final XOR.

As described, the adder would be impractical, requiring an \( i \)-bit carry generation detector for each bit \( i \). Fortunately, the same problem exists for carry-lookahead adders, and solutions are well known. Carry-lookahead adders typically only determine the carry-in for a small subset of the bit positions, and then use some other method to propagate the carries to the remaining bits. The fastest of these other methods is to use carry-select adders. A carry-select adder consists of two adders: the first computes its sum assuming that \( c_{in} = 0 \), and the second computes the same sum assuming that \( c_{in} = 1 \). Two-input multiplexers are used to chose between the two sums when the value of \( c_{in} \) is known. If the size of the groups is chosen so that the carry-select adder can complete by the time the carry generation has completed, then the selection of the correct values for \( T \) takes only one additional logic level.

In this section, we construct practical 16, 32, and 64 bit adders. Our method is to use partial prefix-and carry generation to compute the carries on 4-bit boundaries, and 4-bit carry-select adders to complete the addition. Carries could be computed for other than 4-bit boundaries, but the carry generation algorithm is so fast that if the carry-select adders were any larger, they would become the bottleneck. Each \( n \)-bit addition will complete in \( \log n + 4 \) logic levels.
4.3.1 16-Bit PPA Adder

Our first task is to modify PG16 to output generate values on the 4-bit boundaries; i.e., to produce the 4, 8, 12, and 16-bit generate values. PG16 already produces the 4 and 16-bit values, so our problem is to produce the 8 and 12-bit values.

![Diagram of Module G3: 3-bit internal generate](image)

Figure 12: Module G3: 3-bit internal generate

Our solution to the 12-bit problem is to use PG4 modules in the first stage and a PG3 module in the second stage. In order to simplify our diagram, we replace PG3 with the module G3 shown in figure 12. Module G3 is internally identical to PG3,
but the inputs and outputs have been changed as follows: (1) generate bits that are input from the right are output on the left, (2) the carry-in bit is hardwired to 0, and (3) the group propagate bit is no longer one of the outputs. Using module G3 allows us to reduce the number of input and output labels required in our diagrams.

Module G2 is similarly constructed to find the 8-bit generate value. Since any Gk module is internally identical to a PGk module, we will always do our analysis in terms of PGk modules.

![Diagram](image)

Figure 13: Module PG16/4: 16-bit propagate and 4, 8, 12, and 16-bit generate

Plugging modules G2 and G3 into PG16 (figure 11) gives us the required functionality. No additional logic levels are required to produce the extra generate bits. The
new module will be denoted PG16/4, and is shown in figure 13. In general, we use PGn/k to represent a PPA module that outputs carries on the k, 2k, ..., and n-bit boundaries.

At this point we can use module PG16/4 along with 4-bit carry-select adders to construct a 16-bit PPA adder, as shown in figure 14. Module 2ADD4 is a 4-bit carry-select adder. It computes two 4-bit sums: the first assuming that the carry-in bit is 0, and the second assuming that the carry-in bit is 1. The generate values from module PG16/4 select between the two sums by driving two-input multiplexers.

The adders in 2ADD4 should be designed so that they can complete their two sums in the time required to produce the generate bits. In practice, these delay
requirements can only be determined as part of an implementation. For the purposes of this chapter, we will simply find a design that completes within the number of logic levels required to produce the generate bits.

In the 16-bit adder, the generate bits are produced in 7 logic levels. A four-bit ripple-carry adder (with carry in, but no carry out) requires 8 logic levels, and so is unsuitable. A solution is to use a carry-lookahead circuit for the fourth bit, and use ripple-carry for the first three bits. In this case, the four bit adder completes in 6 levels.

The entire 16-bit adder occupies only 8 logic levels: 1 to generate the propagate and generate bits, 6 to complete module PG16/4, and 1 for the 2-input multiplexers.

4.3.2 32 and 64-Bit PPA Adders

It is easy to expand the 16-bit PPA adder to handle 32 bits. One way is to construct a carry generation module PG32/4 consisting of eight PG4 module in the first stage, and modules PG8, G7, G6, G5, G4, G3, and G2 in the second stage. While we have not shown all of these modules, they should be easy to extrapolate from figures 7, 10, and 12. The construction of PG32/4 is similar to that given in figure 13. The generate outputs, which come on four bit boundaries, again drive 4-bit carry-select adders. The adder requires 9 logic levels: one to generate the propagate and generate bits, seven to complete module PG32/4, and one for the 2-input multiplexers.

The most straightforward scheme for a 64-bit adder is to construct it out of two levels of PG8 modules, and use 8-bit carry-select adders. The carry generation unit PG64/8 would consist of eight PG8 modules in the first stage, and modules PG8, PG7,
PG6, PG5, PG4, PG3, and PG2 in the second stage. The problem with this scheme is in the construction of the 2ADD8 modules. The carry generation is complete after 9 levels, and we are unaware of any simple 8-bit adder that computes its sum (with carry in) in only 9 levels. An 8-bit ripple carry adder requires 16 levels. Of course it is possible to use a more complex 8-bit adder, and this may be the best approach in practice. It is interesting, however, to explore a design using three stages of PG4 modules. The three stages are as follows:

Stage 1: 16 PG4 modules. The output of this stage is sixteen instances of 4-bit group propagate and generate signals.

Stage 2: 4 PG4, 4 PG3, and 4 PG2 modules. The output of this stage is four instances of 16/4 group propagate and generate signals.

Stage 3: 4 PG4, 4 G3, and 4 G2 modules. The output of this stage is the 64/4 group generate signals.

The first two stages are basically four copies of PG16/4, except we also output the 4, 8, and 12-bit propagate signals from each PG16/4.

The stage 3 PG4 modules output the carries on the 52, 56, 60, and 64-bit boundaries. Each of the four PG4 modules gets propagate and generate signals from the 16, 32, and 48-bit boundaries. The only difference is in the high order inputs, which will come from 4, 8, 12, and 16-bit boundaries of the high order PG16/4.

The stage 3 G3 modules output the carries on the 36, 40, 44, and 48-bit boundaries. Each of the 4 G3 modules gets propagate and generate signals from the 16 and 32-bit
boundaries. The only difference is in the high order inputs, which will come from 4, 8, 12, and 16-bit boundaries of the second highest order PG16/4.

Similarly, the stage 3 G2 modules output the carries on the 20, 24, 28, and 32-bit boundaries. The 4, 8, 12, and 16-bit boundaries are simply passed through from the low-order PG16/4 module in stages 1 and 2.

The combined output of the stage 3 modules consists of all of the carries on the 4-bit boundaries. These carries are used to select 4-bit sums from each of sixteen 4-bit carry-select adders.

4.3.3 Subtraction

Since the PPA adder is constructed to accept a carry-in bit, subtraction is easily performed using the standard formula $X - Y = X + \overline{Y} + 1$. Like addition, subtraction of $n$-bit words requires only $\log n + 4$ logic levels.

What is unusual about the PPA adder is that subtraction incurs no additional delay for words of length $2^k$. Note that for words of this size, the carry chain from carry in to carry out has length $2^k + 1$. For carry-lookahead, computing a carry chain of length $2^k + 1$ requires two additional logic levels as compared to computing a carry chain of length $2^k$. The extra carry-in bit is free with PPA carry generation. While this benefit only applies to words of length $2^k$, it is worth mentioning because most words have this length.
4.4 Implementation Issues

In this section we discuss several details of the PPA adder that are beyond the scope of our model, but which may be helpful in an implementation.

It is sometimes useful to invert bits during a computation. Generate bits are not used until the final level of the PPA adder, and so they can be inverted whenever it is convenient. Of course the number of inversions must be a multiple of two. Propagate bits can be inverted between stages. For stages with inverted inputs, prefix-or should be used instead of prefix-and, and the tristate node should select the low-order zero instead of the low-order one.

In some technologies the number of inputs to a tristate node is limited, and the number of consecutive levels in which tristate nodes can be cascaded is also limited. A PGk module has \( k + 1 \) inputs to its tristate node. None of our designs has \( k > 8 \), and it is certainly feasible to construct adders with maximum \( k = 4 \). For stages with maximum \( k > 2 \), there is always at least one level of slack between tristate nodes. An inverter can be placed in this logic level to prevent any cascading of tristate buffers.

Fanout is an important consideration in some technologies, although it is usually considered less important than fanin. The prefix-and function can be implemented so that the fanout is no more than two at any logic level by using the technique of recursive doubling[KS73]. This technique does not increase the depth of the circuit, although it does increase the number of gates and the wiring complexity. An example of computing an 8-bit prefix-and using this technique is given in figure 15. For PG4 modules, the difference is trivial, consisting of one additional AND gate.
For any PGk module, the only other value with fanout greater than two at any logic level is the low order prefix-and bit, e.g., the bit labeled $p_{0,3}$ in figure 10. This bit has a fanout of three or greater, depending on how many modules require it as an input. The fanout of the generate bits is not a problem because the generate bits are only used once in each module, and because the production of the generate bits is not on the critical path, allowing for extra driver levels.

Ways of dealing with the intermodule fanout of propagate bits are technology dependent, and can affect the choice of modules used to implement the adder. Larger modules have lower fanout, and fewer levels at which fanout greater than two occurs. For example, a PG64/8 module will have lower fanout than a PG64/4 module. It will also only have fanout problems at one level instead of at two levels.
CHAPTER V

Early Branch Condition Resolution

Our final application of half-adder form is to the architectural problem of resolving branch conditions. Conditional branches have a negative impact on computer performance, and reducing the latency of branch conditions allows us to reduce this impact. Using half-adder form, we can compute most branch conditions with only slightly more delay than is required to determine if a word is zero. This means that the operation on which a branch depends can potentially execute in the same cycle as the branch. Speculative execution, a technique that has been introduced to avoid some branch penalties, also benefits from this early resolution.

The organization of this chapter is as follows: Section 5.1 shows how to convert a problem in zero detection to a problem in -1 detection, presents a combined adder/subtractor/zero-detector, and compares the costs and delays of the new unit to a typical ALU. Section 5.2 extends the early zero detector to compute other branch conditions by applying the fast carry generation results from chapter IV. Section 5.3 deals with some of the architectural implications of early branch condition resolution, specifically with respect to branch penalties, instruction set design, and speculative
execution. Finally, section 5.4 describes our simulations, which quantify some of the performance benefits that can be realized by using early branch condition resolution.

5.1 Early Zero Detection

As noted in section 1.3.3, the most important condition, and the one that normally has the highest latency, is the “sum = zero” condition. Unfortunately zero is not particularly easy to detect in half-adder form. Since -1 is easy to detect, our strategy will be to convert problems in zero detection to closely related problems in -1 detection.

5.1.1 Addition, Subtraction, and Zero Detection

Observe that for two’s complement numbers, \( X + Y = 0 \Leftrightarrow \overline{X} + \overline{Y} = -1 \), and of course, \( X - Y = 0 \Leftrightarrow \overline{X} - \overline{Y} = -1 \). Since complementation is easy, obtaining the correct sum or difference is trivial once we have the complemented sum or difference. If the left-hand side of the complemented problems (i.e., \( \overline{X} + \overline{Y} \) or \( \overline{X} - \overline{Y} \)) can be expressed in h-a form, we can use theorem 2.2 to detect equality with -1, which immediately gives us zero detection for the original problems.

The only remaining problem is finding a fast way to compute the complemented sum or difference in h-a form. Ideally, the computation should accomplished using only a few, simple constant-time operations. The operations that meet these criteria include complementation, adding two numbers with a half adder, and adding one to a number in h-a form. This last operation is fast because if \((C, S)\) is in h-a form, \(c_0 = 0\), which allows us to add one to \((C, S)\) by setting \(c_0 = 1\).
The following theorem shows how to compute $X + Y$ using just these "fast" operations.

**Theorem 5.1** For two's complement numbers, $X + Y = X + Y + 1$.

**Proof:** The proof of this theorem, as well as the following theorem, relies on the property of two's complement numbers that $-X = X + 1$.

\[
X + Y = -(X + Y) - 1 = -X - 1 - Y - 1 + 1 = X + Y + 1 \quad \Box
\]

The complete procedure for addition and early zero detection using the method of theorem 5.1 is as follows:

**Algorithm 5.1** Given $X$ and $Y$, compute $X + Y$ and determine whether $X + Y = 0$.

1. Compute $(C, S) = ha(X, Y)$, and set $c_0 = 1$.

2. Do the following in parallel:
   - Compute $S' = C \oplus S$, and $m_s = s'_0 \land s'_1 \land \ldots \land s'_{n-1}$.
   - Compute the sum $C + S$.

3. If $m_s = 1$, then $X + Y = 0$. The sum is $X + Y = \overline{C + S}$.

The XOR in step 2 of algorithm 5.1 is required because when we set $c_0 = 1$ in step 1, $(C, S)$ is no longer in h-a form, and hence we cannot use theorem 2.2 for -1
detection. We use XOR gates instead of half adders in step 2 because the result is not used for anything except -1 detection, and for -1 detection the carry word is not needed.

We might suspect that fewer than $2n$ bits would have to be examined in step 2, since $(C, S)$ is so close to h-a form. The following example shows that there is no shortcut.

**Example:** There are many representations of -2 in h-a form, and the "1" bits can be in either $C$ or $S$. For an example of a sum whose bits are mostly in $C$, consider $(C, S) = ha(63, -65)$, which produces the following:

\[
\begin{align*}
63 &= 00111111 \\
-65 &= 10111111 \\
C &= 01111110 \\
S &= 10000000
\end{align*}
\]

For an example of a sum in which the one bits are all in $S$, consider $(C, S) = ha(-2, 0)$.

The "difference = 0" problem is easier than the "sum = 0" problem.

**Theorem 5.2** For two's complement numbers, $\overline{X - Y} = \overline{X} + Y$.

**Proof:**

\[
\begin{align*}
\overline{X - Y} &= -(X - Y) - 1 \\
&= \overline{X} + Y \quad \Box
\end{align*}
\]

The subtraction procedure is as follows:
Algorithm 5.2 Given $X, Y$, compute $X - Y$ and determine whether $X - Y = 0$.

1. Compute $(C, S) = ha(\overline{X}, Y)$

2. Do the following in parallel:
   - Compute $m_s = s_0 \land s_1 \land \ldots \land s_{n-1}$.
   - Compute the sum $C + S$

3. If the $m_s = 1$, then $X - Y = 0$. The difference is $X - Y = C + S$.

The "difference = 0" condition can also be detected by comparing the original inputs for equality. The problem with this method is that it is not integrated with the adder, and it is not applicable to addition. As we will soon see, the method given above can be integrated into a carry-lookahead adder with almost no additional hardware. Furthermore, addition and subtraction and zero detection can be performed with the same circuit.

A combined $n$-bit adder/subtractor/zero-detector is given in figure 16. The "add" bit is set to one if the operation is an addition, and zero if the operation is a subtraction: it is then used to select the appropriate input operand, $Y$ or $\overline{Y}$ (this selection is accomplished using $n$ XOR gates). The selected operand is then added to $X$ with a half-adder, producing the sum $(C, S)$. Since $(C, S)$ is in h-a form, $c_0 = 0$, and so we can easily add one to the sum by setting $c_0 = 1$. Since we only want to add one if the operation is an addition, we set $c_0 = \text{add}$.

In order to test whether $C + S = -1$, we need to have $(C, S)$ in h-a form. The result of this sum is not used elsewhere, and since -1 detection does not require any
information from the carry word, it suffices to compute $S' = C \oplus S$. At this point, we can apply theorem 2.2 to detect whether the sum or difference is zero. We do this by computing the $n$-bit AND of $S'$. If the output is one, the original sum or difference is zero.
While we compute $S'$ and the AND of $S'$, we also compute the sum of $C$ and $S$. We have specified a carry-lookahead adder (CLA), although other adders could be used. The sum $C + S$ has to be inverted to get the answer to the original sum or difference.

### 5.1.2 Cost and Delay

We assume some familiarity with adders in this section, at least at the level of detail presented in [HP96] or [Kor93]. While figure 16 looks like it contains a considerable amount of hardware above and beyond the adder, we will show that most of this hardware is already present in current ALUs.

Consider the blocks beginning at the top of the figure. The XOR gates used to choose the appropriate argument for addition and subtraction ($\overline{Y}$ and $Y$) are required in current ALUs. The next block is a half adder. The first step in a CLA (and in many other adders) is to compute propagate and generate bits $p_{i,i} = x_i \oplus y_i$ and $g_{i,i} = x_i \land y_i$. This step can be eliminated for numbers in h-a form, because $p_{i,i} = s_i$ and $g_{i,i} = c_{i+1}$. The adder itself is unchanged by our scheme, although we do require a final inversion to get the correct sum or difference. The remaining hardware consists of $n$ XOR gates, followed by an $n$-input AND. The AND would be required for any scheme doing zero detection. Thus the only new hardware is $n$ XOR gates, $n$ inverters to compute $\overline{X}$, and $n$ inverters to get the correct sum.

The delay for addition or subtraction is virtually unchanged. All of the hardware that is common adds no extra delay. The computation of $\overline{X}$ also adds no delay because it occurs at the same time as the computation of $\overline{Y}$. In some technologies there may be extra delay due to one extra fanout for $C$ and $S$, and the final inversion
of $C + S$, but this effect is likely to be small, and compensated for by the fact that
the sum $C + S$ does not have to be fed into a zero detector.

The delay for zero detection will of course be substantially improved. In current
implementations, the sum must usually complete before we can check whether all of
the sum bits are 0. Since the low-order bits are typically available before the high
order bits, part of the zero detection can be performed earlier. With the fastest
adders, however, a substantial part of the computation occurs after the the sum is
complete.

The delay for the zero detector in figure 16 is equal to the delay of an inverter,
two XOR gates, a half-adder, and an $n$-input AND, which is $\log n + 3$ logic levels in our
model. A typical branch on zero statement in an existing processor already requires
the $n$-input AND, so if a tiny extra delay can be tolerated, the branch can also use the
output of our zero detector. In section 5.3, we will explore some of the architectural
implications of this early availability.

5.1.3 Other Operations

Other functions for which zero detection may be needed include logical functions,
multiplication, and division. Logical functions are so fast that the logic function and
an $n$-input NOR can easily complete with a delay no greater than the delay for the
scheme given in section 5.1.1. Early zero detection for multiplication can be handled
by checking if either of the arguments is zero. For division, only the numerator needs
to be checked, although the denominator should probably be checked to avoid dividing
by zero. Early zero detection is rarely needed for multiplication and division, and the
simplest scheme may be to add zero to the product or quotient and allow the adder to complete the detection.

5.2 Sign, Carry, and Overflow Detection

The four most common integer conditions are zero, sign, carry out, and overflow. Other conditions are either trivial (e.g., the odd or even condition on the Alpha) or can be easily computed using these four. We would like to extend our early zero detection scheme so that all of the conditions can be detected early. In order to simplify the discussion, we first consider sign, carry, and overflow for any sum in half-adder form, and only then examine the relationship of these conditions with respect to complemented problems.

5.2.1 Sign, Carry, and Overflow in Half-Adder Form

Let \((C, S)\) be in \(n\)-bit h-a form, as shown in figure 17. Since we are discussing the carry and overflow conditions, we retain bit \(c_n\). In order to represent both addition and subtraction, we also consider bit \(c_{in}\). For addition, \((C, S) = ha(X, Y), c_{in} = 0,\) and \(C + S = X + Y\). For subtraction, \((C, S) = ha(X, \overline{Y}), c_{in} = 1,\) and \(C + S = X - Y\).

\[
\begin{align*}
C &= c_n c_{n-1} c_{n-2} \cdots c_2 c_1 c_0 \\
S &= s_{n-1} s_{n-2} \cdots s_2 s_1 s_0
\end{align*}
\]

Figure 17: Example for discussion of conditions

Now suppose that \(c_{out}\) is defined to be the carry out of the low order \(n - 1\) bits of \((C, S)\), including \(c_{in}\) and \(c_{n-1}\). Using one of the prefix-and carry-generation detection
methods from chapter IV, we can compute $c_{out}$ in $\log n + 2$ steps. We now show that
that the three conditions are completely determined by $c_n, s_{n-1}$, and $c_{out}$.

The sign of $C + S$ is given by $s_{n-1} \oplus c_{out}$.

The carry condition is made easy by the fact that $(C, S)$ is in h-a form.

**Theorem 5.3** If $s_{n-1} = 0$ then carry $= c_n$, else carry $= c_{out}$.

**Proof:** If $s_{n-1} = 0$, then no carry will propagate past position $n - 1$, so the only possible carry bit is $c_n$.

If $s_{n-1} = 1$, then $c_n = 0$. If $c_{out} = 0$, then there is no carry out of $C + S$. If $c_{out} = 1$, then there is a carry out of $C + S$. Thus the carry condition is given by $c_{out}$. □

Overflow for the addition of unsigned numbers is given by the carry out. For signed numbers, overflow occurs when numbers have the same sign, and the resulting sum has a different sign. This is typically calculated by taking the XOR of the carry into the sign bit and the carry out of the sign bit. This calculation is also made easier by the fact that $(C, S)$ is in h-a form.

**Theorem 5.4** If $c_{out} = 0$ then overflow $= c_n$, else overflow $= \overline{c_n} \land \overline{s_{n-1}}$.

**Proof:** The carry into the sign bit is $c_{out}$.

If $c_{out} = 0$, then overflow occurs when there is a carry out of the sign bit. In this case, the only possible carry out of the sign bit is given by bit $c_n$, so overflow $= c_n$.

If $c_{out} = 1$, then overflow occurs when there is no carry out of the sign bit. An analysis of all of the three possible values for the pair $(c_n, s_{n-1})$ shows that there is no carry out of the sign bit only when $c_n = s_{n-1} = 0$. □
5.2.2 Sign, Carry, and Overflow for Complemented Problems

In order to use early zero detection, we are computing \( \overline{X + Y} \) instead of \( X + Y \) (or \( \overline{X - Y} \) instead of \( X - Y \)), and this affects some of the conditions. Of course the sign bit in the complemented problem must be inverted in order to be correct for the original problem. The situation for \( \text{the} \) carry and overflow bits is not quite as obvious.

**Theorem 5.5** The carry out of the low order \( k \) bits of \( X + Y \) is not equal to the carry out of the low order \( k \) bits of \( \overline{X + Y} \) for any \( k \in \{1, 2, \ldots, n\} \). Similarly, the carry out of the low order \( k \) bits of \( X - Y \) is not equal to the carry out of the low order \( k \) bits of \( \overline{X - Y} \) for any \( k \in \{1, 2, \ldots, n\} \).

**Proof:** For the computation of \( X + Y \), let \( (C, S) = ha(X, Y) \), with \( c_{in} = 0 \). By theorem 5.1, the computation of the complemented problem is given by \( (C', S') = ha(\overline{X}, Y) \), with \( c'_{in} = 1 \). Let \( T = C + S + c_{in} \), and \( T' = C' + S' + c'_{in} \). For any \( k < n \), \( t_k \) is expressed as \( s_k \oplus c_{outk} \), where \( c_{outk} \) is the carry out of the low order \( k \) bits of \( X + Y + c_{in} \), and \( t'_k \) is expressed as \( s'_k \oplus c'_{outk} \), where \( c'_{outk} \) is the carry out of the low order \( k \) bits of \( \overline{X + Y} + c'_{in} \). By theorem 5.1, \( t_k = t'_k \). Note that \( s_k = x_k \oplus y_k = \overline{x_k} \oplus \overline{y_k} = s'_k \), and so we must have \( c_{outk} \neq c'_{outk} \).

For \( k = n \), note that \( c_{outn} = c_n \vee (s_{n-1} \wedge c_{out(n-1)}) \), and \( c'_{outn} = c'_n \vee (s'_{n-1} \wedge c'_{out(n-1)}) \). Note that \( s_{n-1} = s'_{n-1} \), so there are two cases to consider:

**Case 1:** \( s_{n-1} = s'_{n-1} = 0 \). Then either \( x_{n-1} = y_{n-1} = 0 \), in which case \( c_n = 0 = c'_n \), or \( x_{n-1} = y_{n-1} = 1 \), in which case \( c_n = 1 = c'_n \). In either case, \( c_{outn} \neq c'_{outn} \).
Case 2: $s_{n-1} = s'_{n-1} = 1$, and so $c_n = c'_n = 0$. As we proved earlier, $c_{out(n-1)} \neq c'_{out(n-1)}$, and so $c_{outn} \neq c'_{outn}$.

For the computation of $X - Y$, let $(C', S') = ha(X, Y')$, with $c_{in} = 1$. By theorem 5.2, the computation of the complemented problem is given by $(C', S') = ha(X, Y)$, with $c'_{in} = 0$. Let $T = C + S + c_{in}$, and $T' = C' + S' + c'_{in}$. For any $k < n - 1$, $t_k$ is expressed as $s_k \oplus c_{outk}$, where $c_{outk}$ is the carry out of the low order $k$ bits of $X + Y + c_{in}$, and $t'_k$ is expressed as $s'_k \oplus c'_{outk}$, where $c'_{outk}$ is the carry out of the low order $k$ bits of $X + Y + c'_{in}$. By theorem 5.2, $t_k = \overline{t'_k}$. Note that $s_k = x_k \oplus \overline{y_k} = \overline{x_k} \oplus y_k = s'_k$, and so we must have $c_{outk} \neq c'_{outk}$.

For $k = n$, the proof is the same as that given for addition. □

In particular, the carry computed by theorem 5.3 must be inverted to apply to the original problem.

**Theorem 5.6** The computation of $X + Y$ overflows $\Leftrightarrow$ the computation of $X - Y$ overflows. Similarly, the computation of $X - Y$ overflows $\Leftrightarrow$ the computation of $X + Y$ overflows.

**Proof:** Let $c_{in}$ and $c_{out}$ be the carries into and out of the sign bit in the computation of $X + Y$ or $X - Y$, and let $c'_{in}$ and $c'_{out}$ be the carries into and out of the sign bit in the computation of the corresponding complemented problem $(X + Y$ or $X - Y$).

Recall that overflow is true exactly when $c_{in} \neq c_{out}$. By theorem 5.5, $c_{in} \neq c'_{in}$ and $c_{out} \neq c'_{out}$, so $c_{in} \neq c_{out} \Leftrightarrow c'_{in} \neq c'_{out}$. □

Applying theorems 5.3, 5.4, 5.5, and 5.6 to figure 16, we get figure 18. Note that the sign and carry bits are inverted, and that the zero and overflow bits are not.
The delay for zero detection is $\log n + 3$ logic levels, which is unchanged from figure 16. The delay for our new conditions is $\log n + 5$ logic levels, only two levels more than are required for zero detection. Of course nothing prevents us from replacing the PG unit with a PPA adder, which would also give us the sum or difference in $\log n + 5$ logic levels.
5.3 Architectural Impact

Since branches and the way they are handled figure so prominently in CPU performance, faster detection of branch conditions has a major architectural impact. In this section we consider the impact of early branch condition resolution on branch penalties, instruction set design, instruction-level parallelism (ILP), and some forms of speculative execution. For simplicity, our examples use a typical five-stage pipeline (IF, ID, EX, MEM, and WB) like that given by Hennessy and Patterson [HP96]. Similar arguments can be presented for longer pipelines. We also consider both single- and multiple-issue implementations.

5.3.1 Branch Penalties

There are three basic methods for transferring branch conditions from an operation to a branch instruction:

1. The conditions can be stored in global status bits (condition codes). This method typically requires that the cycle time be long enough to accommodate an operation plus zero detection. Early branch condition resolution may allow us to reduce the cycle time on machines for which the ALU execution stage is the critical path. It also increases the opportunity for instruction-level parallelism. In particular, the operation generating the condition may be able to execute at the same time as the branch using the condition.

2. The conditions can be evaluated by examining a general purpose register (condition registers). For zero detection, this means that the result of an addition
or subtraction is stored in a register, and the branch statement examines this register to see if it contains zero. This method is superficially less attractive than condition codes if early zero detection is available. The reason is that the zero condition is known a full cycle later than it could be, which has an adverse effect on ILP. We think that this objection can be overcome by forwarding the branch conditions during the cycle, as we will show later in this section.

3. The conditions can be evaluated within the branch statement itself. In this case, early branch condition resolution will allow more sophisticated compare and branch operations, an idea that we explore in section 5.3.2.

In some architectures, branch penalties are reduced by moving the execution phase of the branch statement back into the instruction decode phase. Branch statements become two-stage instruction: (1) fetch, and (2) decode, compute target address, and execute. For example, this technique is used in the IBM PowerPC. A data hazard is introduced if the branch depends on an instruction that immediately precedes it in the pipeline. Consider the following sequence:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1,R2,R3</td>
<td></td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>BEQZ R1,offset</td>
<td>IF</td>
<td>ID/EX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The meaning of this sequence is $R1 \leftarrow R2 + R3$, and then branch to PC plus offset if $R1 = 0$. The execution of the branch must stall for one cycle because it relies on the outcome of the execution of the ADD instruction. Forwarding cannot help because there is no result to forward until the end of the execution phase.
Early branch condition resolution can help here, because the zero condition can be detected early in the cycle, and this single bit can be forwarded to the branch instruction that is executing during the same cycle, thus resulting in no stalling.

5.3.2 New Instructions

Early branch condition resolution opens up some interesting opportunities for new instructions. As noted previously, compare and branch instructions inhibit ILP since they evaluate branch conditions a full cycle later than they could. No such problem occurs in a single instruction that adds, compares, and branches. The only difficulty with such an instruction is finding enough bits to specify the operands and branch offset. One solution to this difficulty is to reduce the generality of the add instruction.

For example, consider the instruction “increment and branch on zero”

IBZ R1,offset

The meaning of the instruction is to add one to R1, and branch to PC + offset when R1 becomes zero. Such an instruction would make a useful loop counter, and would save an instruction on every iteration as compared to the usual method.

Another possibility is to allow one of the operands to be overwritten. For example, the instruction “add and branch on zero”

ABZ R1,R2,offset

could mean to set $R1 \leftarrow R1 + R2$, and then branch to PC + offset when R1 becomes zero. This allows somewhat more sophisticated loop control, because the loop index could be incremented or decremented by an arbitrary value. The disadvantage is that the number of bits for the offset is decreased.
5.3.3 Speculative Execution

Speculative execution is the execution of some instructions before the outcome of the control flow is known. Speculative execution avoids control dependence penalties when the speculated sequence is correctly predicted. One method of implementing speculation used in current processors is to separate instruction execution from instruction completion or commitment during which results are written to registers or to memory [HP96]. Instructions could be executed speculatively out of order but are always committed, after speculation resolution, in order. A reorder buffer is used to hold the instructions that have executed but not committed. With early branch condition resolution, the control outcome is known sooner. Hence, the size of the reorder buffer and the associated control circuitry decreases.

A restrictive but widely used form of speculative execution uses conditional or predicated instructions. For example, the conditional move instruction, which moves the contents of one register to another conditionally, is part of the instruction set of many processors, including Alpha, MIPS, PowerPC, SPARC, and the P6. The popularity of this instruction is due to the fact that it avoids branch delays and increases instruction-level parallelism. The instruction essentially converts a control dependence into a data dependence as the example below shows. The data dependence can be resolved by forwarding, and early branch condition resolution is not necessary to prevent a stall in a single-issue pipelined processor. In multiple-issue pipelines, however, early branch condition resolution could permit the conditional move to be done
at the same time as the instruction that sets the condition. Consider the following instruction sequence:

\[
\begin{array}{c|c|c|c|c|c|c}
& ID & EX & MEM & WB \\
\hline
ADD & R1, R2, R3 & IF & & & \\
CMOVZ & R4, R5, R1 & IF & & & \\
\end{array}
\]

The ADD instruction adds R2 and R3, and puts the result in R1. Using early zero detection, we can tell whether or not R1 will be zero well before the add has completed. With suitable hardware, this single bit of information could be used by the CMOVZ instruction that executes at the same time.

Another form of speculative execution, nullification, is used in HP's PA-RISC. Nullification allows any instruction to be skipped based on a condition set by the previous instruction. For example, an addition can be performed with the "less than" condition. If the result is less than zero, then the next instruction is skipped. Early branch condition resolution will allow both instructions to execute concurrently in those cases in which there is no data dependency between the two instructions. This clearly increases ILP for superscalar implementations of PA-RISC. Early branch condition resolution may also result in a shorter cycle time, because currently the cycle has to accommodate an addition plus zero detection.

Architectures using condition codes increase the possibilities for conditional instructions. Since the condition code is global state, no condition register has to be specified in the instruction, and potentially any instruction could conditionally execute. For example, an instruction CADDZ R1, R2, R3 could set \( R1 \leftarrow R2 + R3 \) if the
condition code were zero. As with conditional move, early branch condition resolution allows the conditional instructions to execute in the same cycle as the instruction that sets the condition, thus increasing ILP.

5.4 Simulation

In this section, we measure some of the benefits of early branch resolution. In particular, we measure what happens when we avoid the stalls discussed in section 5.3.1, i.e., we allow a condition-setting instruction and a conditional branch that uses the result of that instruction to execute in the same cycle if the condition-setting instruction is an (integer) add, subtract, logical, or compare instruction.

5.4.1 Methodology

Our goal is to simulate a scalar machine in which the execution of conditional branches is moved back into the stage before the execution stage (see section 5.3 for a more complete description of the machine we want to model). The question we want to answer is how much improvement we will see on such a machine by using early branch condition resolution.

The tool we use for our simulations is DEC's ATOM [ES94, SE94] running on an Alpha 21064 processor. ATOM is a system that facilitates the construction of program analysis tools. Tools are constructed by creating an instrumentation program and an analysis program. The instrumentation program specifies where in the execution of a program the analysis routines should be called, what analysis routines to call, and what arguments to pass. For example, an analysis routine could be called at the
beginning of every basic block [ASU86]. The instrumentation and analysis programs are independent of the programs to be analyzed.

An important advantage to using ATOM is that it allows other researchers to reproduce our results. In most sciences, results are not accepted until they have been independently confirmed. In computer science, most experiments involve unique simulation programs and/or traces, and descriptions of the simulation usually do not give enough information to reproduce the simulation. Recently, there has been some recognition that computer architecture researchers must require higher standards of reproducibility [Mud96]. We believe that researchers with access to ATOM should be able to reproduce our simulations.

Some standard tools come prepackaged with ATOM, and one of these, the pipe tool, is very nearly what we needed for our simulation. The pipe tool provides a lower bound on the number of pipeline stalls that will occur when running a program. The tool only gives a lower bound because it does not account for cache misses, branch mispredictions, or interblock dependencies. What it does count is the number of cycles that must exist between the definition and use of any value within a basic block. Its output includes the number of instructions executed, the number of stalls, the number of instructions that could be dual issued, and the number of cycles required by the program.

The 21064 is a dual-issue superscalar machine with a seven-stage integer pipeline, and it does not move the execution of conditional branches back into the stage before the execution stage. Obviously some modifications must be made to the pipe tool in
order to simulate our scalar machine. Fortunately the 21064 does produce a stall if the condition cannot be separated from the branch. For our purposes, this stall is beneficial, in that it allows us to simulate what stall reduction would be like in the presence of an optimizer that is trying to avoid the stall.

Our modifications to the pipe tool and its output are easiest to understand when they are broken into three steps:

**step 1:** Convert the cycle count from superscalar to scalar.

**step 2:** Convert the cycle count in the scalar machine so it reflects a machine in which the execution of conditional branches is moved to the stage preceding the normal execution stage. This will be our base machine.

**step 3:** Compute the cycle count for the improved machine, which is the base machine with early branch condition resolution.

For step 1, computing the cycle count for a scalar machine, recall that the original pipe tool reports the number of instructions that were executed, the number of instructions that could be dual issued, and the number of stalls. The number of cycles is reported as the number of instructions that could be dual issued plus the number of stalls. For our scalar simulation, we report the number of cycles as the number of instructions executed plus the number of stalls. This is almost certainly an overestimate of the number of cycles that are actually required in a scalar implementation, because some of the stalls will be resolved by the serialization of instructions that were formerly issued in parallel. A way to reduce some of this overestimate is to
eliminate the one-cycle stall that is present if a conditional branch depends on the result of a dual-issued instruction when the accompanying instruction is an integer add, subtract, compare, or logical instruction. This can easily be accomplished by changing a few lines in the instrumentation program pipe.inst.c. We also add code to both the instrumentation and analysis routines to compute the number of conditional branches, and the number of 'improved' conditional branches, i.e., those that are immediately preceded by a defining integer add, subtract, compare, or logical instruction. We label these branches as 'improved', because they are the branches that will be improved by early branch condition resolution. As a result of these changes, our modified pipe tool produces four output values, which are listed in table 8.

Table 8: Description of raw simulation data

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instructions</td>
<td>the number of instructions executed</td>
</tr>
<tr>
<td>oldstalls</td>
<td>the number of stalls</td>
</tr>
<tr>
<td>branches</td>
<td>the number of conditional branches</td>
</tr>
<tr>
<td>improved</td>
<td>the number of improved conditional branches</td>
</tr>
</tbody>
</table>

For step 2, computing the number of cycles in the base machine, we need to consider the improvement in stalls due to early execution of conditional branches. Moving the execution of these branches back by one stage reduces the number of stalls by one for each branch executed, but introduces a stall for each of the 'improved' branches. Table 9 shows how to compute the number of stalls in the base machine given the four output values from the simulation. It also gives formulas for the number
of cycles required by the base machine, the percentage of stalls in the base machine, and the CPI for the base machine.

Table 9: Formulas for simulation analysis

<table>
<thead>
<tr>
<th>Name</th>
<th>Column (table 12)</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>basestalls</td>
<td></td>
<td>oldstalls - branches + improved instructions + basestalls</td>
</tr>
<tr>
<td>basecycles</td>
<td></td>
<td>basestalls - improved</td>
</tr>
<tr>
<td>newstalls</td>
<td></td>
<td>basecycles - improved</td>
</tr>
<tr>
<td>newcycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>basepstalls</td>
<td></td>
<td>% stalls (base)</td>
</tr>
<tr>
<td>newpstalls</td>
<td></td>
<td>% stalls (new)</td>
</tr>
<tr>
<td>stallimprovep</td>
<td></td>
<td>100 * basestalls/basecycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 * newstalls/newcycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 * (basestalls - newstalls)/newstalls</td>
</tr>
<tr>
<td>pbranches</td>
<td>cond. branches %</td>
<td>(100 * branches)/instructions</td>
</tr>
<tr>
<td>pimproved</td>
<td>cond. branches %</td>
<td>(100 * improved)/instructions</td>
</tr>
<tr>
<td>basecpi</td>
<td>cpi (base)</td>
<td>basecycles/instructions</td>
</tr>
<tr>
<td>newcpi</td>
<td>cpi (new)</td>
<td>newcycles/instructions</td>
</tr>
<tr>
<td>speedup</td>
<td>speedup %</td>
<td>100 * (basecycles - newcycles)/newcycles</td>
</tr>
</tbody>
</table>

For step 3, computing the number of cycles in the improved machine, we subtract one stall for each improved conditional branch. The formulas for the new number of cycles, percentage of stalls, and CPI are given in table 9.

Our test programs came from a public collection of benchmarks maintained by Al Aburto of the Naval Command, Control and Ocean Surveillance Center (NC-COSC) RDT&E Division (NRaD) in San Diego, CA. The benchmarks are available by anonymous ftp to 'ftp.nosc.mil' in directory 'pub/aburto'. A brief description of the benchmarks is given in table 10.
Table 10: Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dhrystones 1.1</td>
<td>integer</td>
<td>synthetic instruction mix</td>
</tr>
<tr>
<td>dhrystones 2.1</td>
<td>integer</td>
<td>synthetic instruction mix</td>
</tr>
<tr>
<td>fhourstones</td>
<td>integer</td>
<td>finds solutions to the ‘connect-4’ game</td>
</tr>
<tr>
<td>hanoi</td>
<td>integer</td>
<td>solves ‘Tower of Hanoi’ puzzle recursively</td>
</tr>
<tr>
<td>heapsort</td>
<td>integer</td>
<td>sorts random arrays of long integers</td>
</tr>
<tr>
<td>nsieve</td>
<td>integer</td>
<td>finds primes using sieves of various sizes</td>
</tr>
<tr>
<td>queens</td>
<td>integer</td>
<td>solves ‘12-Queens’ problem</td>
</tr>
<tr>
<td>sim</td>
<td>integer</td>
<td>compares DNA segments for similarity</td>
</tr>
<tr>
<td>clinpack</td>
<td>floating point</td>
<td>Linpack converted to C</td>
</tr>
<tr>
<td>fft</td>
<td>floating point</td>
<td>solves a 131072-point FFT</td>
</tr>
<tr>
<td>tftdp</td>
<td>floating point</td>
<td>solves FFTs using Duhamel-Hollman method</td>
</tr>
</tbody>
</table>

5.4.2 Results and Discussion

Our results are shown in tables 11 and 12. The meaning of each of the columns is described in the previous section. Each program was compiled and run at optimization levels O1, O2, and O3. The programs are sorted according to their speedup at optimization level O3.

The geometric mean speedup at optimization level O3 for the integer benchmarks is 5.1%, as opposed to 1.4% for the floating point benchmarks. One reason for this is that both the percentage of instructions that are conditional branches and the percentage of conditional branches that can be improved are lower for the floating point programs. For example, consider the queens and clinpack benchmarks at optimization level O3. Queens has 17.9% conditional branches, 58.7% of which can be improved, which means that about 10.5% of all instructions are improved. Clinpack has 6.3% conditional branches, 22.1% of which can be improved, which means that
Table 11: Raw simulation data

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Opt.</th>
<th>Instructions</th>
<th>Oldstalls</th>
<th>Conditional branches improved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>total</td>
<td></td>
<td>total</td>
</tr>
<tr>
<td>queens</td>
<td>O1</td>
<td>179477524</td>
<td>55376300</td>
<td>26516308</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>148293436</td>
<td>48131925</td>
<td>26516084</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>148321789</td>
<td>48146096</td>
<td>26516080</td>
</tr>
<tr>
<td>sim</td>
<td>O1</td>
<td>3357118351</td>
<td>19040689561</td>
<td>3490837650</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>23780095970</td>
<td>7931175443</td>
<td>3111845392</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>22842010849</td>
<td>7503483111</td>
<td>3111845400</td>
</tr>
<tr>
<td>heapsort</td>
<td>O1</td>
<td>11143059511</td>
<td>637356935</td>
<td>135544723</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>1652769518</td>
<td>902036632</td>
<td>221809502</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>1652768292</td>
<td>902036731</td>
<td>221809512</td>
</tr>
<tr>
<td>fhourstones</td>
<td>O1</td>
<td>2642642464</td>
<td>1440025632</td>
<td>211198833</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>1984617177</td>
<td>925903518</td>
<td>178620445</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>1928235663</td>
<td>954722919</td>
<td>178620442</td>
</tr>
<tr>
<td>dhry11</td>
<td>O1</td>
<td>729019340</td>
<td>288005438</td>
<td>43002516</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>507019334</td>
<td>124005363</td>
<td>34002514</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>349019403</td>
<td>123005484</td>
<td>33002520</td>
</tr>
<tr>
<td>dhry21</td>
<td>O1</td>
<td>726068409</td>
<td>268013807</td>
<td>42009565</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>598068401</td>
<td>167013765</td>
<td>40009561</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>434068331</td>
<td>153013749</td>
<td>40009554</td>
</tr>
<tr>
<td>hanoi</td>
<td>O1</td>
<td>743702511</td>
<td>350955915</td>
<td>25071239</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>735346464</td>
<td>100280647</td>
<td>25071239</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>735346272</td>
<td>100280568</td>
<td>25071236</td>
</tr>
<tr>
<td>nsieve</td>
<td>O1</td>
<td>948583398</td>
<td>247570774</td>
<td>143663523</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>878848636</td>
<td>220335064</td>
<td>123240836</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>878847990</td>
<td>220334903</td>
<td>123240830</td>
</tr>
<tr>
<td>fft</td>
<td>O1</td>
<td>242546583</td>
<td>166666404</td>
<td>739709</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>104639749</td>
<td>36806903</td>
<td>6749684</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>104639528</td>
<td>39395602</td>
<td>6749688</td>
</tr>
<tr>
<td>tfftdp</td>
<td>O1</td>
<td>411630688</td>
<td>304123145</td>
<td>14627122</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>278333026</td>
<td>180483104</td>
<td>14627096</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>278332198</td>
<td>197609353</td>
<td>14627092</td>
</tr>
<tr>
<td>clinpack</td>
<td>O1</td>
<td>1280045407</td>
<td>1578587707</td>
<td>86671847</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>564997755</td>
<td>818596849</td>
<td>35248894</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>560071587</td>
<td>757662002</td>
<td>35208083</td>
</tr>
</tbody>
</table>
Table 12: Simulation analysis

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Opt.</th>
<th>Cond. branches</th>
<th>% stalls base</th>
<th>Stall imp. %</th>
<th>cpi base</th>
<th>Speed-up %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>% inst.</td>
<td>% imp.</td>
<td>% stalls</td>
<td>new</td>
<td></td>
</tr>
<tr>
<td>queens</td>
<td>O1</td>
<td>14.8</td>
<td>96.8</td>
<td>23.3</td>
<td>13.9</td>
<td>89.0</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>17.9</td>
<td>58.7</td>
<td>20.0</td>
<td>12.7</td>
<td>72.0</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>17.9</td>
<td>58.7</td>
<td>20.0</td>
<td>12.7</td>
<td>71.9</td>
</tr>
<tr>
<td>sim</td>
<td>O1</td>
<td>10.4</td>
<td>70.9</td>
<td>34.6</td>
<td>31.7</td>
<td>15.9</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>13.1</td>
<td>60.5</td>
<td>22.0</td>
<td>16.9</td>
<td>39.0</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>13.4</td>
<td>60.5</td>
<td>20.9</td>
<td>15.3</td>
<td>45.5</td>
</tr>
<tr>
<td>heapsort</td>
<td>O1</td>
<td>12.2</td>
<td>64.8</td>
<td>34.6</td>
<td>31.7</td>
<td>17.5</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>13.4</td>
<td>60.1</td>
<td>33.0</td>
<td>29.2</td>
<td>19.6</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>13.4</td>
<td>60.1</td>
<td>33.0</td>
<td>29.2</td>
<td>19.6</td>
</tr>
<tr>
<td>fhourstones</td>
<td>O1</td>
<td>8.0</td>
<td>71.1</td>
<td>34.3</td>
<td>31.7</td>
<td>12.2</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>9.0</td>
<td>72.0</td>
<td>30.6</td>
<td>27.4</td>
<td>17.2</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>9.3</td>
<td>76.9</td>
<td>32.1</td>
<td>28.7</td>
<td>17.7</td>
</tr>
<tr>
<td>dhry11</td>
<td>O1</td>
<td>5.9</td>
<td>65.1</td>
<td>27.2</td>
<td>25.2</td>
<td>11.4</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>6.7</td>
<td>55.9</td>
<td>17.7</td>
<td>15.1</td>
<td>21.1</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>9.5</td>
<td>55.0</td>
<td>23.6</td>
<td>20.5</td>
<td>20.0</td>
</tr>
<tr>
<td>dhry21</td>
<td>O1</td>
<td>5.8</td>
<td>71.4</td>
<td>26.1</td>
<td>23.7</td>
<td>13.3</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>6.7</td>
<td>55.0</td>
<td>19.9</td>
<td>17.5</td>
<td>17.3</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>9.2</td>
<td>55.0</td>
<td>23.6</td>
<td>20.5</td>
<td>20.0</td>
</tr>
<tr>
<td>hanoi</td>
<td>O1</td>
<td>3.4</td>
<td>100.0</td>
<td>32.1</td>
<td>30.5</td>
<td>7.7</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>3.4</td>
<td>100.0</td>
<td>12.0</td>
<td>9.3</td>
<td>33.3</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>3.4</td>
<td>100.0</td>
<td>12.0</td>
<td>9.3</td>
<td>33.3</td>
</tr>
<tr>
<td>nsieve</td>
<td>O1</td>
<td>15.1</td>
<td>37.9</td>
<td>14.3</td>
<td>9.9</td>
<td>52.4</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>14.0</td>
<td>22.1</td>
<td>12.4</td>
<td>9.9</td>
<td>28.1</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>14.0</td>
<td>22.1</td>
<td>12.4</td>
<td>9.9</td>
<td>28.1</td>
</tr>
<tr>
<td>fft</td>
<td>O1</td>
<td>3.0</td>
<td>54.2</td>
<td>40.2</td>
<td>39.6</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>6.5</td>
<td>54.4</td>
<td>24.4</td>
<td>22.3</td>
<td>12.2</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>6.5</td>
<td>54.4</td>
<td>25.8</td>
<td>23.8</td>
<td>11.2</td>
</tr>
<tr>
<td>tfftdp</td>
<td>O1</td>
<td>3.6</td>
<td>47.8</td>
<td>41.9</td>
<td>41.3</td>
<td>2.4</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>5.3</td>
<td>30.5</td>
<td>38.0</td>
<td>37.3</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>5.3</td>
<td>30.5</td>
<td>40.2</td>
<td>39.7</td>
<td>2.4</td>
</tr>
<tr>
<td>clinpack</td>
<td>O1</td>
<td>6.8</td>
<td>11.0</td>
<td>54.0</td>
<td>53.8</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>6.2</td>
<td>19.9</td>
<td>58.3</td>
<td>58.1</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>6.3</td>
<td>22.1</td>
<td>56.6</td>
<td>56.3</td>
<td>1.1</td>
</tr>
</tbody>
</table>
only about 1.4% of all instructions are improved. To make the difference even worse, floating point programs tend to have more stalls for other reasons, such as longer latency instructions and more demands on memory. This is reflected in the higher percentage of stalls in the floating point programs (20% of all cycles for queens on the base machine, as opposed to 56.6% of all cycles for clinpack), and ultimately in the higher CPI for floating point programs.

The relationship between stalls, CPI, and speedup is straightforward. It is clear that reducing the number of stalls reduces the number of cycles required by a program, which in turn reduces the CPI. The speedup is also directly related to the stall improvement percent and the percent of instructions that are stalls. For example, the dhrystones1.1 benchmark at optimization level O3 has new percent stalls of 20.5%, stall improvement of 20%, and a speedup of $0.205 \times 0.20 = 4.1\%$. It is easy to verify that this relationship holds in general by using the formulas in table 9.

The speedups for our method are generally better at higher optimization levels, because a more optimized program provides fewer locations where the compiler can insert an instruction between a condition-setting instruction and a branch. In fact, of the eleven benchmark programs, only queens and nsieve have their best speedups at optimization level O1. The other nine benchmarks all have their best speedups at optimization level O3.

For a real implementation of early branch prediction, the speedups are likely to be different than those presented in table 12. The factors that will affect the true
speedups are branch misprediction and cache miss penalties, interblock dependencies, pipeline depth, increased functional units, superscalar processing, and compiler changes. As noted previously, the pipe tool does not take into account branch misprediction, interblock dependencies, or cache miss penalties. In a real program these would increase the total number of cycles used, hence increasing the denominator in the speedup calculation, and lowering the speedup.

The other factors should lead to better speedups. The Alpha has a seven-stage pipeline, and many of the pipeline stalls are due to the Alpha's high load and store latency. Reducing both this latency and the latency of other data hazards, as would happen in a more typical five-stage pipeline, would greatly decrease the denominator in the speedup computations, resulting in better speedups. Newer processors are becoming more superscalar, which again shrinks the denominator without shrinking the numerator in the speedup computation. The reason that the numerator does not shrink is that the number of branches does not change in a superscalar simulation. In fact, the number of improved branches is likely to increase, because there will be fewer occasions when instructions can be inserted between the condition and the branch when more instructions are executed in each cycle. For superscalar architectures, stalls due to condition-setting instructions issuing at the same time as dependent conditional move instructions could also be avoided. Finally, if compilers do not have to try to insert instructions between comparisons and branches, they may be better able to perform other optimizations.
In summary, we believe that the speedups given here are likely to be lower than what can be achieved in a modern superscalar machine. The speedups come without any appreciable hardware cost, and with no need to modify instructions sets. Given the low cost, and considering that early zero detection may also allow faster cycle time, the argument for early branch condition resolution is compelling.
CHAPTER VI

Conclusion

In this thesis we have introduced the half-adder form, and have provided new algorithms based on this form that have the potential to dramatically increase computer performance. Our contributions are in three main areas:

1. We have developed the theory of the half-adder form. Sums or differences can be converted to their half-adder form $(C, S)$ with only a small constant delay, and useful information about $C + S$ can be extracted from $(C, S)$ before $C + S$ has been computed.

2. We have developed new algorithms and circuit designs for some of the most fundamental operations in computers. Using our model, the new upper bounds for these problems are given in table 13. Latency for many of these operations is only about half of what it is in current implementations, and our designs are much simpler than the slower designs they replace.

3. We have suggested some easy architectural changes for making use of the early availability of branch conditions, and quantified some of the benefits to be obtained.
Table 13: New upper bounds

<table>
<thead>
<tr>
<th>Problem</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>programmable modulo-$k$ counting</td>
<td>3</td>
</tr>
<tr>
<td>early zero detection for addition or subtraction</td>
<td>$\log n + 3$</td>
</tr>
<tr>
<td>carry generation detection, with carry in comparison, $&gt;$</td>
<td>$\log n + 3$</td>
</tr>
<tr>
<td>addition, with carry in and carry out</td>
<td>$\log n + 4$</td>
</tr>
<tr>
<td>sign, carry, and overflow detection for addition or subtraction</td>
<td>$\log n + 5$</td>
</tr>
</tbody>
</table>

Many new areas of research are possible as a result of this work.

Perhaps most interesting is to investigate how computers will change as a result of faster addition. Addition is the fundamental operation in CPUs and a potential doubling of addition speed translates to a possible doubling of throughput in CPU designs whose clock period is governed by the time to add two numbers. While computer manufacturers seem reluctant to discuss where the critical path is in their pipelines, there is at least some evidence that addition is on that critical path for the fastest processors. For example, the architects at DEC say that one of the hard problems in designing the 21064 was creating an adder that could complete in the required time [D+92]. At a talk at Berkeley in November 1995, Bob Colwell of Intel said that one of the reasons that the P6 has so many pipeline stages is that they were able to increase the speed of the integer ALU to a greater extent than they expected [Asa95]. Since the P6 has a 14-stage pipeline, it is interesting to consider how many pipeline stages might be useful given even faster addition.

Whatever the actual number of pipeline stages turns out to be, it is clear that the number is going to be larger than it is today. Deeper pipelines will in turn affect
the detection and elimination of load, data, and control hazards. For example, the
design of forwarding hardware and schemes for branch prediction will have to be more sophisticated. Detecting and eliminating hazards through compiler optimizations, pipeline scheduling, extra hardware and pipeline interlocks will prove to be challenging research areas. The design of superscalar and VLIW architectures will also be affected: a deeper pipeline coupled with a faster cycle time will lead to a need for more sophisticated management of functional units.

The effect of half-adder form is going to be even more far-reaching than what has been developed here, with new possibilities opening up in computer arithmetic, computer architecture, and circuit design. As an example of work that remains to be done, consider the following research problems:

1. Multiplication. Multiplication is performed by generating and summing some number of product words. The product words are typically added in redundant form, and then a final carry-propagate addition completes the multiplication. The last step can easily be improved using a PPA adder, but it is interesting to consider whether other improvements can be made.

One possibility is to use Karatsuba's multiplication algorithm [Knu69]. Let $U = U_1 U_0$ and $V = V_1 V_0$, where $U_1$ and $V_1$ are the most significant $n/2$ bits of $U$ and $V$, and $U_0$ and $V_0$ are the least significant $n/2$ bits of $U$ and $V$. Karatsuba's algorithm reduces the $n$ bit multiplication $UV$ to the three $n/2$ bit multiplications $U_1 V_1$, $(U_1 - U_0)(V_0 - V_1)$, and $U_0 V_0$, plus an addition to create the final product. This method would be clearly superior to other hardware
methods if it were not for the second product, which requires the delay of a carry-propagate adder before the multiplication can begin. Of course we could compute the differences in h-a form, but the interesting research question is whether we can efficiently multiply numbers in h-a form. By corollary 2.1, the number of “1” bits for a number in \( n/2 \)-bit h-a form is at most \( n/2 \). Since only “1” bits generate product terms that have to be added, it is reasonable to suppose that such an efficient algorithm may exist.

2. Floating point addition. Floating point addition involves three separate integer additions, as well as a normalization step that requires locating the high order one in a word. Using PPA adders, and something like a prefix-or (analogous to the prefix-and discussed in chapter IV) to find the high order one, it should be possible to speed this up.

3. Division. Division is the slowest of the elementary arithmetic operations. Most algorithms are inherently sequential: guess a quotient digit, subtract an appropriate amount from the dividend, and see if the result is positive or negative. By maintaining the dividend in half-adder form, and using the fast comparison technique from section 4.2, these algorithms should be substantially improved.

4. Early iterative loop resolution. Early branch resolution can be made even faster for iterative loops by applying constant-time modulo \( k \) counting. The IBM PowerPC [WS94] already has a dedicated loop counter. Constructing such a loop counter as a constant-time modulo \( k \) counter would result in lower latency for
determining loop termination, and would also require considerably less hardware than the fast adder that is currently used. The latency for such a loop counter would be only a small constant number of gate delays, which is even faster than the early zero detection proposed in this thesis. With such a low latency, and no need to fetch arguments, the execution of this instruction could be moved very early in the pipeline. Such performance enhancements will be important with the longer pipelines that are coming.

5. Priority Encoder. A priority encoder encodes the n-bit address of the highest input bit that is set to one. A priority encoder can be constructed as a prefix-or followed by a regular encoder. An interesting research problem is whether the two parts can be executed concurrently, i.e., whether the encoding process can proceed simultaneously with the leading one detection.

6. Dot products and sums with no cumulative roundoff error. A major problem in floating point addition is cumulative roundoff error. Floating point addition is not associative, a fact that greatly complicates numerical programming and numerical analysis. A solution proposed by Bohlender and Kulisch [BK83] is to compute dot products exactly. Of course this requires very large adders: 279 bits for single precision, and 2100 bits for double precision. A systolic implementation of this idea was presented by Cappello and Miranker [CM88, CM92], but their unit has an unacceptably long latency. A low latency solution is made much more practical by using the ideas in this thesis. In particular, the numbers can be added in carry-save form, then converted to half-adder form.
to facilitate the conversion back to floating point. Once the sum is available in h-a form, the leading digit can be found using a prefix-and or prefix-or circuit, and the final sum can be obtained using a PPA adder. In theory, the latency of such a unit would not be much worse than the latency for a normal floating point addition.

The preceding list is not intended to be comprehensive, but only includes the problems we have started to consider. We expect that the tools in this thesis will be applicable to many other problems in arithmetic and computer architecture. In particular, we believe that the half-adder form will lead to decreased latency for all of the remaining elementary arithmetic operations, in both their fixed and floating point implementations. Together with the results in this thesis, these new designs will provide a significant increase in computer performance.
Bibliography


