

The Half-Adder Form and Early Branch Condition Resolution

David R. Lutz
Bell Labs
6200 East Broad Street
Columbus, Ohio 43213
d.r.lutz@ieee.org

D. N. Jayasimha*
MS RN2-02
Intel Corporation
Santa Clara, CA 95052
djayasim@mipos2.intel.com

Abstract

We present efficient methods to determine the four usual branch conditions for a sum or difference, before the result of the addition or subtraction is available. The methods lead to the design of an early branch resolver which integrates well with a regular adder/subtractor, adding only a small amount of circuitry and almost no delay. The methods exploit the properties of half-adder form. Sums in half-adder form can be computed very quickly (with the delay of a half adder), yet they have enough structure so that many of the properties of the final sum can be easily detected. The reduced latency for evaluating branch conditions means that an addition or subtraction and a dependent conditional instruction can execute in the same cycle, with a consequent increase in instruction-level parallelism, and improved performance for both single-issue and superscalar processors.

Index terms: *Half-adder form, branch conditions, addition, subtraction, early zero detection, carry generation detection.*

1 Introduction

Conditional branches and other conditional instructions negatively impact computer performance. Reducing the latency of the evaluation of branch conditions allows us to reduce this impact. The conditions (zero, sign, overflow, carry, and their variants) are usually evaluated based on a preceding addition or subtraction operation. In this paper, we present efficient methods to determine the four usual branch conditions for a sum or difference, *before* the result of the addition or subtraction is available. These methods lead to the design of an early branch condition resolver which integrates well with a regular adder/subtractor, adding only a

small amount of circuitry and almost no delay. The methods exploit the properties of *half-adder form*, a new intermediate representation of sums, by which 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, with a consequent reduction in branch penalty. This reduction improves the performance of both single-issue and superscalar processors and also has positive architectural implications relating to instruction level parallelism, enhanced instruction sets, and speculative execution.

The organization of the paper is as follows: The rest of this section discusses previous work done in this area, and introduces our notation. Since the half-adder form is so critical to our method, Section 2 is devoted to its explanation. Section 3 presents a way to determine whether a sum or difference is zero before computing the sum or difference. The remaining conditions all require a fast method to compute whether a sum generates a carry, which we present in section 4. We then use this method to extend our early zero detector to compute the other conditions in section 5. The paper concludes with a discussion of the impact of the work and problems that merit further investigation.

Prior work on early branch condition resolution has focused primarily on early zero detection. This is not surprising since “branch on zero” and “branch on nonzero” are the most frequently executed conditional branch instructions for most architectures [3]. 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 [17]. The method is based on half adders, and results from simplifying the equations for

*This work was done while the author was with the Department of Computer and Information Science at the Ohio State University in Columbus, Ohio.

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 [5], 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 XNOR gates and an n -input AND.

MIPS processors make heavy use of this partial solution [1]. 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 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.

Putrino and Vassiliadis designed a branch condition resolver that operates in parallel with the adder for IBM mainframe computers [13] (they also discuss a zero detector based on the same idea in [16]). Their 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 they claim that it is approximately 35% faster than the adder. Phillips and Vassiliadis later extended this result to zero detection for 3-input ALUs [12].

More recently, several authors have independently discovered some closely related methods to compute the more general condition $X + Y = T$ for an arbitrary T [2, 15, 7, 11].

All of these approaches to early zero detection are different from the one taken in this paper. Our approach is faster, simpler, more closely integrated with the ALU, and can be more easily applied to existing (non-mainframe) instruction sets. Furthermore, our approach can be extended to include all of the usual branch conditions, including the conditions that require carry generation detection.

Notation

We use lower-case letters to represent numbers, upper-case letters to represent n -bit words, and subscripted lower-case letters to represent bits. The meaning of other symbols is given in Table 1.

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,

Symbol	Meaning
\wedge	bitwise and
\vee	bitwise or
\oplus	bitwise exclusive or
$\overline{x_i}$	not x_i
\overline{X}	one's complement of X

Table 1. Notation

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.

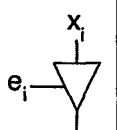
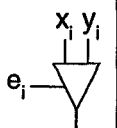
Symbol	Name	Meaning						
	tristate buffer	<table><tr><th>e_i</th><th>output</th></tr><tr><td>0</td><td>Z</td></tr><tr><td>1</td><td>x_i</td></tr></table>	e_i	output	0	Z	1	x_i
e_i	output							
0	Z							
1	x_i							
	two-input multiplexer	<table><tr><th>e_i</th><th>output</th></tr><tr><td>0</td><td>x_i</td></tr><tr><td>1</td><td>y_i</td></tr></table>	e_i	output	0	x_i	1	y_i
e_i	output							
0	x_i							
1	y_i							

Table 2. Representation of tristate buffers and 2-input multiplexers

We measure 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. Of course this model does not provide precise information about delay, but it does provide a useful starting point for comparing the speed of various algorithms. A more complete description of and justification for the model is given in [8].

2 Background

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 $X = x_{n-1} \dots x_1 x_0$, and $Y = y_{n-1} \dots y_1 y_0$ be n -bit words with low order bits x_0 and y_0 . An n -bit half adder produces a carry word $C = c_{n-1} \dots c_1 c_0$ and a sum word $S = s_{n-1} \dots s_1 s_0$ such that

$$c_i = x_{i-1} \wedge y_{i-1} \quad (1)$$

$$s_i = x_i \oplus y_i \quad (2)$$

Note that c_0 is always 0, and that $C + S = X + Y$ (modulo 2^n). The high order carry bit c_n is not part of C , but is sometimes useful as part of a larger calculation.

Definition: (C, S) is in *half-adder form* (or *h-a form*) if there exist X and Y satisfying equations 1 and 2. We write $(C, S) = ha(X, Y)$.

Numbers in h-a form have a rich set of properties, some of which we have described previously [6, 8, 9, 10]. One of the most basic of these properties is the following:

Theorem 1 *Let (C, S) be a number in h-a form. Then $C + S = -1 - S = -1$.*

Proof:

[\Rightarrow] (C, S) is in h-a form, so there exist X and Y such that $X + Y = -1$ and $(C, S) = ha(X, Y)$. By the definition of a two's complement number, $X + Y = -1 - Y = \overline{X}$. Then by equation 2, $S = X \oplus \overline{X} = -1$.

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

Theorem 1 says that we can detect whether a sum is -1 with the delay of a half adder and an n-input AND. Unfortunately, the same technique does not work when trying to determine whether $X + Y = 0$: if $S = 0$, then this tells us nothing about C , and almost nothing about $C + S$. Fortunately, there is a duality to two's complement numbers that we can use to our advantage. The trick, which we present in the next section, is to convert a problem involving zero detection to an equivalent (or nearly equivalent) problem involving -1 detection.

3 Early Zero Detection

Observe that for two's complement numbers, $X + Y = 0 - \overline{X + Y} = -1$, and of course, $X - Y = 0 - \overline{X - Y} = -1$. 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., $\overline{X + Y}$ or $\overline{X - Y}$) can be expressed in h-a form, we can use theorem 1 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 be 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 $\overline{X + Y}$ using just these "fast" operations.

Theorem 2 *For two's complement numbers, $\overline{X + Y} = \overline{X} + \overline{Y} + 1$.*

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

$$\begin{aligned}\overline{X + Y} &= -(X + Y) - 1 \\ &= -X - 1 - Y - 1 + 1 \\ &= \overline{X} + \overline{Y} + 1 \quad \square\end{aligned}$$

The complete procedure for addition and early zero detection using the method of theorem 2 is as follows:

Algorithm 1 *Given X and Y , compute $X + Y$ and determine whether $X + Y = 0$.*

1. Compute $(C, S) = ha(\overline{X}, \overline{Y})$, and set $c_0 = 1$.
2. Do the following in parallel:
 - Compute $S' = C \oplus S$, and $m_s = s'_0 \wedge s'_1 \wedge \dots \wedge s'_{n-1}$.
 - 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 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 1 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:

$$\begin{aligned}63 &= 00111111 \\ -65 &= 10111111\end{aligned}$$

$$\begin{aligned}C &= 01111110 \\ S &= 10000000\end{aligned}$$

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

Proof:

$$\begin{aligned}\overline{X - Y} &= -(X - Y) - 1 \\ &= \overline{X} + Y \quad \square\end{aligned}$$

The subtraction procedure is as follows:

Algorithm 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 $m_s = s_0 \wedge s_1 \wedge \dots \wedge s_{n-1}$.
 - Compute the sum $C + S$
3. If the $m_s = 1$, then $X - Y = 0$. The difference is $X - Y = \overline{C + S}$.

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 1. 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 \overline{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 $c_0 = add$.

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 1 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.

While figure 1 looks like it contains a considerable amount of hardware above and beyond the adder, we will

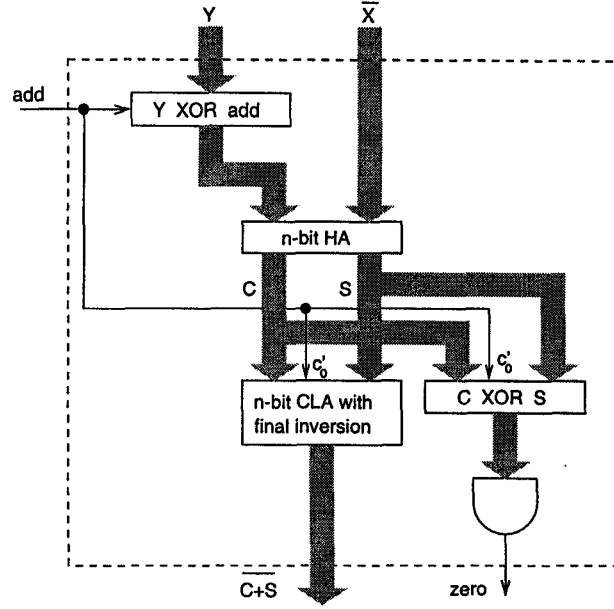


Figure 1. Combined adder/subtractor/zero-detector

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 $g_{i,i} = x_i \wedge y_i$. 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 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. 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 1 is equal to the delay of 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 the relatively small extra delay can be tolerated, the branch can also use the output of our zero detector.

4 Carry Generation Detection

Early branch condition resolution for the other conditions requires a fast method for determining whether a sum in h-a form (with a carry-in bit) generates a carry. In this section, we present a method that requires only $\log n + 2$ logic levels, which is asymptotically almost twice as fast as previous implementations.

This surprising result is based on the fact 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 that bit c_i determines whether or not $C + S$ generates a carry: the rest of C can be ignored. **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 $s_{n-1} = s_{n-2} = \dots = s_i = 1$.

Example: If $S = 111011$, then $A = 111000$.

Theorem 4 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 $c_i = 1$ — 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 $s_{n-1} = s_{n-2} = \dots = s_i = 1$. Since $c_i = 1$, and since the carry propagates through $s_i, s_{i+1}, \dots, s_{n-1}$, then there is a carry-out of $C + S$.

[\Leftarrow] By the definition of A , $s_i = s_{i+1} = \dots = s_{n-1} = 1$. By the definition of half-adder form, only one of s_k and c_{k+1} can be set, so $c_{i+1} = c_{i+2} = \dots = c_{n-1} = 0$, and no carry is generated at positions $i + 1, i + 2, \dots, n - 1$. Since $s_{i-1} = c_0 = 0$, and since only one member of each pair of bits (c_j, s_{j-1}) for $j = 1, 2, \dots, i - 1$ can be set, there is no carry into position i . Since there is a carry-out of position $n - 1$, we must have $c_i = 1$. \square

Example: Given X and Y as in equations 3 and 4, then C, S , and A are given in equations 5, 6, and 7. 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 .

$$X = 1101101100111001 \quad (3)$$

$$Y = 0010010011001111 \quad (4)$$

$$C = 0000000000010010 \quad (5)$$

$$S = 1111111111101110 \quad (6)$$

$$A = 111111111110000 \quad (7)$$

Given n -bit X and Y , and a carry-in bit c_{in} , the *carry-generation detection problem* is to determine whether or not there is a carry out from the n -bit sum $X + Y + c_{in}$. A straightforward application of theorem 4 leads to the following algorithm to solve this problem.

Algorithm 3 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 $1, 2, \dots, n - 1$ do in parallel
if $a_i \wedge \overline{a_{i-1}} = 1$ 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[4]. Step 3 can be implemented in two levels, the first of which computes the logical function $a_i \wedge \overline{a_{i-1}}$, 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, 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 2 shows an 8-bit circuit based on this method. To simplify the diagram, the computation of C is not shown. 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

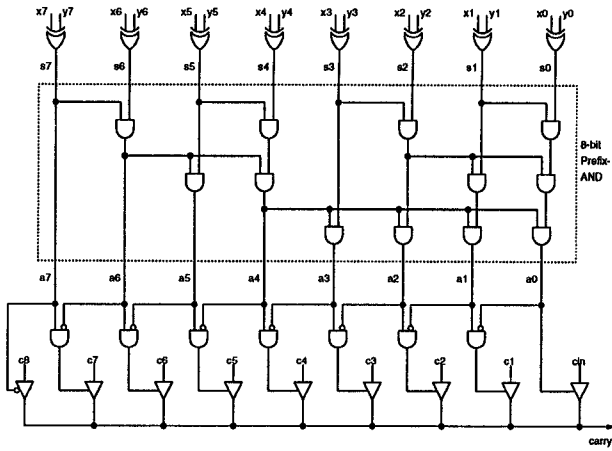


Figure 2. 8-bit prefix-and carry generation detection

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.

The prefix-and method as given is not suitable for large n , but it can be easily adapted to compute subblocks according to the usual propagate and generate scheme for computing carries. Details on this *partial prefix-and* method, together with comparisons to other methods of carry generation detection, can be found in [6, 8]. These papers also discuss fanout and several other implementation issues. The partial prefix-and method also has delay of only $\log n + 2$ logic levels to determine whether a sum in h-a form will generate a carry.

5 Sign, Carry, and Overflow Detection

Given fast carry-generation detection, we are now ready to compute the remaining conditions. 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 the complemented problems we need for early zero detection.

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

Let (C, S) be in n -bit h-a form, as shown in figure 3. Since we are discussing the carry and overflow conditions, we need to retain bit c_n , the AND output of the high order half-adder. In order to represent both addition and subtraction, we also consider a carry-in bit c_{in} . For addition, $(C, S) = ha(X, Y)$, $c_{in} = 0$, and $C + S = X + Y$. For subtraction, $(C, S) = ha(X, \bar{Y})$, $c_{in} = 1$, and $C + S = X - Y$.

Note that c_n and c_{in} are not part of C , but are just extra bits needed for these computations. Since $c_0 = 0$, it is convenient to replace c_0 with c_{in} when using C as an input to a sum.

$$\begin{array}{rcl} C & = & c_n \quad c_{n-1} \quad c_{n-2} \quad \dots \quad c_2 \quad c_1 \quad c_0 \\ S & = & \quad \quad s_{n-1} \quad s_{n-2} \quad \dots \quad s_2 \quad s_1 \quad s_0 \end{array}$$

Figure 3. 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, we can compute c_{out} in $\log n + 2$ steps. We now show 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 *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} . \square

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

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 $c_n = s_{n-1} = 0$. \square

5.2 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 the carry and overflow bits is not quite as obvious.

Theorem 7 *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 $\overline{X} + \overline{Y}$ for any $k \in \{1, 2, \dots, n\}$. 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 $\overline{X} - \overline{Y}$ for any $k \in \{1, 2, \dots, n\}$.*

Proof: For the computation of $X + Y$, let $(C, S) = ha(X, Y)$, with $c_{in} = 0$. By theorem 2, the computation of the complemented problem is given by $(C', S') = ha(\overline{X}, \overline{Y})$, 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 $\overline{X} + \overline{Y} + c'_{in}$. By theorem 2, $t_k = \overline{t'_k}$. 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 $c_n = 0 = \overline{c'_n}$, or $x_{n-1} = y_{n-1} = 1$, in which case $c_n = 1 = \overline{c'_n}$. 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 $(C, S) = ha(X, \overline{Y})$, with $c_{in} = 1$. By theorem 3, the computation of the complemented problem is given by $(C', S') = ha(\overline{X}, Y)$, 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 $X + \overline{Y} + c_{in}$, 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 3, $t_k = \overline{t'_k}$. 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 must be inverted to apply to the original problem.

Theorem 8 *The computation of $X + Y$ overflows – the computation of $\overline{X} + \overline{Y}$ overflows. Similarly, the computation of $X - Y$ overflows – the computation of $\overline{X} - \overline{Y}$ 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 ($\overline{X} + \overline{Y}$ or $\overline{X} - \overline{Y}$). Recall that overflow is true exactly

when $c_{in} \neq c_{out}$. By theorem 7, $c_{in} \neq c'_{in}$ and $c_{out} \neq c'_{out}$, so $c_{in} \neq c_{out} - c'_{in} \neq c'_{out}$. □

Applying theorems 5, 6, 7, and 8 to figure 1, we get figure 4. The $PG(n-1)$ box computes c_{out} using one of the prefix-and methods. Note that the sign and carry bits are inverted, and that the zero and overflow bits are not.

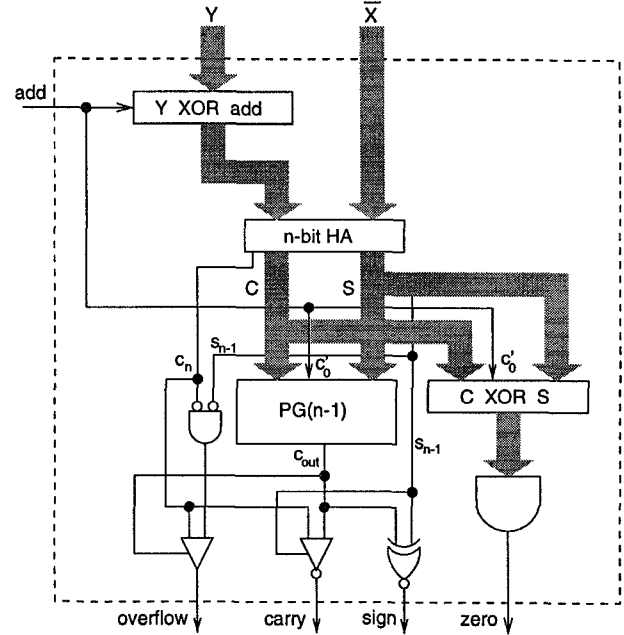


Figure 4. Early branch condition detector

The delay for zero detection is $\log n + 3$ logic levels, which is unchanged from figure 1. The delay for our new conditions is $\log n + 5$ logic levels, only two levels more than are required for zero detection.

6 Conclusion

In this paper, we have presented new methods for performing early branch condition resolution. We have shown that the methods can be incorporated into existing adders with very little additional cost and without the need to modify instruction sets.

To our knowledge, these are (at least algorithmically) the fastest methods for determining the standard branch conditions. For branch conditions requiring carry generation detection, our method requires only about half of the number of two-input logic levels as current methods. For early zero detection, fanin requirements show that our method is within one logic level of optimal. For technologies permitting higher fanin, our algorithms (which are dominated by n -input AND and prefix-and functions) should scale at least as well as competing algorithms. Of course determining

exact speedups will require implementations.

Since conditional instructions figure so prominently in CPU performance, faster resolution of branch conditions has a major architectural impact which could result in:

- Increased instruction-level parallelism.
- Reduced cycle time for architectures that set condition codes.
- The possibility of more powerful instructions, e.g., “increment and branch on zero”.
- Reduced hardware requirements for speculative execution.

For a detailed discussion of these architectural issues, the reader is referred to [9].

We have quantified some of the architectural benefits of early branch resolution through simulation [6] using DEC's ATOM [14]. Conservative measurements of the speedups range from three to nine percent for optimized integer benchmarks on a single-issue processor. These speedups should be regarded as preliminary, as they do not account for branch prediction (which would tend to decrease the impact) or superscalar processors (which would tend to increase the impact). We are currently working on extending our simulations to account for these conditions.

Early branch condition resolution is one of a growing set of applications of the half-adder form. The half-adder form has also been used to reduce the latency of comparison [8], modulo-k counting [10], and addition [6]. We are currently exploring applications to multiplication, division, floating point addition, floating point dot products, and early iterative loop resolution.

References

- [1] P. Chow, editor. *The MIPS-X RISC Microprocessor*. Kluwer Academic Publishers, Norwell, Massachusetts, 1989.
- [2] Jordi Cortadella and Jose M. Llaberia. Evaluation of $A + B = K$ conditions without carry propagation. *IEEE Transactions on Computers*, 41(11):1484–1488, November 1992.
- [3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.
- [4] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, October 1980.
- [5] J. J. Losq and G. S. Rao. Zero condition code detection for early resolution of bcs and bcrrs. *IBM Technical Disclosure Bulletin*, 25:130–133, 1982.
- [6] David R. Lutz. *The Power of the Half-Adder Form*. PhD thesis, The Ohio State University, 1996.
- [7] David R. Lutz and D. N. Jayasimha. The power of carry-save addition. Technical Report 15, Department of Computer and Information Science, The Ohio State University, March 1994.
- [8] David R. Lutz and D. N. Jayasimha. Comparison of two's complement numbers. *International Journal of Electronics*, 80(4):513–523, April 1996.
- [9] David R. Lutz and D. N. Jayasimha. Early zero detection. In *International Conference on Computer Design*, pages 545–550, October 1996.
- [10] David R. Lutz and D. N. Jayasimha. Programmable modulo-k counters. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 43(11):939–941, November 1996.
- [11] Behrooz Parhami. Comments on “Evaluation of $A + B = K$ conditions without carry propagation”. *IEEE Transactions on Computers*, 43:381, April 1994.
- [12] James Phillips and Stamatis Vassiliadis. Result equal zero predictor for 3-1 interlock collapsing ALUs. *International Journal of Electronics*, 75(3):379–392, March 1993.
- [13] M. Putrino and Stamatis Vassiliadis. Resolution of branching with prediction. *International Journal of Electronics*, 66(2):163–172, February 1989.
- [14] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.
- [15] Stamatis Vassiliadis, James Phillips, and Bart Blanner. Interlock collapsing ALU's. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.
- [16] Stamatis Vassiliadis and M. Putrino. Condition code predictor for fixed-point arithmetic units. *International Journal of Electronics*, 66(6):887–890, June 1989.
- [17] Arnold Weinberger. High-speed zero sum detection. In *4th IEEE Symposium on Computer Arithmetic*, pages 200–207, 1975.