Using Prefix Computation to Add
Notes on UNITY: 26-90

Jayadev Misra*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

A circuit for adding two \( n \)-bit numbers, proposed in [?], uses prefix-computation to generate the carries. The role of prefix-computation is not made explicit there; nor does the proposed circuit employ the optimal prefix-computation scheme of Ladner and Fischer [?]. We address these issues in this note; this note is self-contained.

Let \( a_{n-1} \ldots a_0 \) and \( b_{n-1} \ldots b_0 \) be two \( n \)-bit numbers that are to be added; here, \( a_0, b_0 \) are the lowest bits. Let \( c_i \) denote the carry into the \( i^{th} \) bit, \( 0 \leq i \leq n \).

\[
\begin{align*}
  c_0 &= 0 \\
  0 \leq i < n : c_{i+1} &= \begin{cases} 0 & \text{if } a_i + b_i + c_i \leq 1 \\ 1 & \text{if } a_i + b_i + c_i > 1 \end{cases}
\end{align*}
\]

Given the carries the sum of the two numbers can be computed in one parallel step. Therefore, we consider the problem of computing the carries.

A Formula for the Carries

Define a three-valued variable \( m_i \) for every \( i, 0 \leq i < n \), as follows. The values assumed by \( m_i \) are 0, 1 or \( U \).

\[
m_i = \begin{cases} a_i & \text{if } a_i = b_i \\ U & \text{if } a_i \neq b_i \end{cases}
\]

We will express the carries, \( c_i \)'s, in terms of the \( m_i \)'s.

**Theorem 1:** For each \( i, 0 \leq i < n \),

\[
c_{i+1} = \begin{cases} m_i & \text{if } m_i \neq U \\ c_i & \text{if } m_i = U \end{cases}
\]

---

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658-065 and by the Office of Naval Research Contract N00014-90-J-1640.
Proof:
\[ m_i = 0 \]
\[ \Rightarrow \{ \text{From (3): } a_i = 0, b_i = 0 \} \]
\[ a_i + b_i + c_i \leq 1 \]
\[ \Rightarrow \{ \text{Using (2)} \} \]
\[ c_{i+1} = 0 \]
\[ \Rightarrow \{ m_i = 0 \} \]
\[ c_{i+1} = m_i \]

Similarly, \( m_i = 1 \) \( \Rightarrow \) \( c_{i+1} = m_i \)

Now,
\[ m_i = U \]
\[ \Rightarrow \{ a_i \neq b_i. \text{ Hence } a_i + b_i = 1 \} \]
\[ a_i + b_i + c_i \leq 1 \equiv c_i = 0 \]
\[ \Rightarrow \{ \text{Using (2)} \} \]
\[ c_{i+1} = 0 \equiv c_i = 0 \]
\[ \Rightarrow \{ c_i, c_{i+1} \text{ take one two possible values} \} \]
\[ c_{i+1} = c_i \]

The above theorem suggests defining a binary operator, \(*\), on three-valued variables. Let,

\[ x * y = \begin{cases} y & \text{if } y \neq U \\ x & \text{if } y = U \end{cases} \]

Then, \( c_{i+1} = c_i * m_i \)

Observation: \(*\) is associative.

Proof: Left to the reader.

Theorem 2:
\[ c_i = 0 * m_0 * \ldots * m_{i-1}, \ 0 \leq i \leq n \]

Proof: By induction on \( i \).

\( i = 0: \) Both sides in the above equation evaluate to 0.
\( i + 1, i \geq 0: \) From Theorem 1, \( c_{i+1} = c_i * m_i \). Using the induction hypothesis to replace \( c_i \), we get the desired result.

An Addition Circuit

We propose an addition circuit consisting of three stages that computes (in sequence) (1) the \( m_i \)'s, (2) the \( c_i \)'s, and (3) the sum.

We adopt the following encoding for the three values—0, 1, \( U \)—using two booleans (henceforth \( F \) stands for “False” and \( T \) for “True”): 0 by \( FF \), 1 by \( FT \) and \( U \) by \( TF \). Thus, the first boolean bit shows if the value differs from \( U \) and the second boolean encodes the values 0, 1 by \( F, T \) respectively.

Stage 1, computing the \( m_i \)'s from the \( a_i \)'s and \( b_i \)'s, can be done in one parallel step. With the proposed encoding, from (3),
the first bit of $m_i = a_i \not\equiv b_i$ and,
the second bit of $m_i = a_i \land b_i$

(where 0,1—the values of $a_i$, $b_i$—are encoded by $F$, $T$ respectively).

In Stage 2, we compute prefixes of the form $0 \ast m_0 \ldots \ast m_{i-1}$ for all $i$, $0 \leq i \leq n$. Since $\ast$ is associative, all these prefix-computations can be completed, using Ladner-Fischer scheme, in $O(\log n)$ parallel steps employing $O(n)$ processors (circuit-elements); moreover only $O(n)$ scalar operations need be performed. The application of $\ast$ can be performed by a switching element that directs the appropriate input to the output. Specifically, let $z = x \ast y$. In the following $x'$. $x$ denote the first and the second boolean bits of $x$ (similarly for $y, z$).

\[
\begin{align*}
'z &= x \land 'y \\
'z &= y \lor (\neg y \land 'x)
\end{align*}
\]

In Stage 3, the sum is computed from the original inputs and the carries in one parallel step. As a minor optimization, the $i^{th}$ bit of the sum, $s_i$, is given by

\[
s_i = \begin{cases} 
  c_i & \text{if } m_i \neq U \\
  \neg c_i & \text{if } m_i = U 
\end{cases}
\]

(To see this, note that $s_i = (a_i + b_i + c_i) \mod 2$, i.e., $s_i = ((a_i + b_i) \mod 2 + c_i) \mod 2$. And, $m_i = U \equiv (a_i + b_i) \mod 2 = 1$.) Using our encoding (note that $s_i$ is a single boolean value whereas $m_i, c_i$ are encoded by two boolean values)

\[
s_i = \neg m_i \not\equiv c_i
\]