Digital Electronics: Combinational Logic

Binary Adders

Introduction

• We will now look at how binary addition may be implemented using combinational logic circuits. We will consider:
  – Half adder
  – Full adder
  – Ripple carry adder
Half Adder

• Adds together two, single bit binary numbers $a$ and $b$ (note: no carry input)

• Has the following truth table:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c_{out}$</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• By inspection:

$$sum = a \cdot b + a \cdot \overline{b} = a \oplus b$$

$$c_{out} = a \cdot b$$

Full Adder

• Adds together two, single bit binary numbers $a$ and $b$ (note: with a carry input)

• Has the following truth table:
Full Adder

<table>
<thead>
<tr>
<th>$c_{in}$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c_{out}$</th>
<th>$sum$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$sum = c_{in}.a.b + c_{in}.a.b + c_{in}.a.b + c_{in}.a.b$

$sum = c_{in}.(a.b + a.b) + c_{in}.(a.b + a.b)$

From DeMorgan

$a.b + a.b = (a + b). (a + b)$

$= (a.a + a.b + b.a + b.b)$

$= (a.b + b.a)$

So,

$sum = \overline{c}_{in}.(\overline{a}.b + a.\overline{b}) + c_{in}.(\overline{a}.b + a.\overline{b})$

$sum = c_{in}.x + c_{in}.\overline{x} = c_{in} \oplus x = c_{in} \oplus a \oplus b$

Full Adder

<table>
<thead>
<tr>
<th>$c_{in}$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c_{out}$</th>
<th>$sum$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$c_{out} = c_{in}.a.b + c_{in}.a.b + c_{in}.a.b + c_{in}.a.b$

$c_{out} = a.b.(c_{in} + c_{in}) + c_{in}.a.b + c_{in}.a.b$

$c_{out} = a.b + c_{in}.\overline{a}.b + c_{in}.a.\overline{b}$

$c_{out} = a.(b + c_{in}.\overline{b}) + c_{in}.a.b$

$c_{out} = a.(b + c_{in}).(b + \overline{b}) + c_{in}.a.b$

$c_{out} = b.(a + c_{in}.\overline{a}) + a.c_{in} = b.(a + c_{in}).(a + \overline{a}) + a.c_{in}$

$c_{out} = b.a + b.c_{in} + a.c_{in}$

$c_{out} = b.a + c_{in}.(b + a)$
Full Adder

- Alternatively,

$$c_{out} = \bar{c}_{in}.a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b} + c_{in}.a.b$$

<table>
<thead>
<tr>
<th>$c_{in}$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c_{out}$</th>
<th>$\text{sum}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0</td>
<td>1</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0</td>
<td>1</td>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
<td>0</td>
<td>1 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
<td>1</td>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
<td>1</td>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Which is similar to previous expression except with the OR replaced by XOR

Ripple Carry Adder

- We have seen how we can implement a logic to add two, one bit binary numbers (inc. carry-in).
- However, in general we need to add together two, $n$ bit binary numbers.
- One possible solution is known as the Ripple Carry Adder
  – This is simply $n$, full adders cascaded together
Ripple Carry Adder

- Example, 4 bit adder

- Note: If we complement \(a\) and set \(c_0\) to one we have implemented \(s = b - a\)

To Speed up Ripple Carry Adder

- Abandon compositional approach to the adder design, i.e., do not build the design up from full-adders, but instead design the adder as a block of 2-level combinational logic with \(2n\) inputs (+1 for carry in) and \(n\) outputs (+1 for carry out).

- Features
  - Low delay (2 gate delays)
  - Need some gates with large numbers of inputs (which are not available)
  - Very complex to design and implement (imagine the truth table!)
To Speed up Ripple Carry Adder

- Clearly the 2-level approach is not feasible
- One possible approach is to make use of the full-adder blocks, but to generate the carry signals independently, using fast carry generation logic
- Now we do not have to wait for the carry signals to ripple from full-adder to full-adder before output becomes valid

Fast Carry Generation

```
Conventional RCA

Fast Carry Adder
```

```
c_0 a_0 b_0 a_1 b_1 a_2 b_2 a_3 b_3

\[ \text{Fast Carry Generation} \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_0</td>
<td>c_1</td>
</tr>
<tr>
<td>c_2</td>
<td>c_3</td>
</tr>
<tr>
<td>c_4</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{sum} \]

\[ \text{c_in} \]

\[ \text{c_out} \]
```
Fast Carry Generation

• We will now determine the Boolean equations required to generate the fast carry signals

• To do this we will consider the carry out signal, $c_{\text{out}}$, generated by a full-adder stage (say $i$), which conventionally gives rise to the carry in ($c_{\text{in}}$) to the next stage, i.e., $c_{i+1}$.

<table>
<thead>
<tr>
<th>$c_i$</th>
<th>$a$</th>
<th>$b$</th>
<th>$s_i$</th>
<th>$c_{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>0 1</td>
<td>0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>0 1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Carry out always zero. Call this **carry kill**
  - $k_i = \overline{a_i} \overline{b_i}$

- Carry out same as carry in. Call this **carry propagate**
  - $p_i = a_i \oplus b_i$

- Carry out generated independently of carry in. Call this **carry generate**
  - $g_i = a_i \cdot b_i$

Also (from before), $s_i = a_i \oplus b_i \oplus c_i$
Fast Carry Generation

- Also from before we have,
\[ c_{i+1} = a_i b_i + c_i (a_i + b_i) \] or alternatively,
\[ c_{i+1} = a_i b_i + c_i (a_i \oplus b_i) \]
Using previous expressions gives,
\[ c_{i+1} = g_i + c_i P_i \]
So,
\[ c_{i+2} = g_{i+1} + c_{i+1} P_{i+1} \]
\[ c_{i+2} = g_{i+1} + P_{i+1} (g_i + c_i P_i) \]
\[ c_{i+2} = g_{i+1} + P_{i+1} g_i + P_{i+1} P_i c_i \]

Fast Carry Generation

Similarly,
\[ c_{i+3} = g_{i+2} + c_{i+2} P_{i+2} \]
\[ c_{i+3} = g_{i+2} + P_{i+2} (g_{i+1} + P_{i+1} (g_i + c_i P_i)) \]
\[ c_{i+3} = g_{i+2} + P_{i+2} (g_{i+1} + P_{i+1} g_i) + P_{i+2} P_{i+1} P_i c_i \]
and
\[ c_{i+4} = g_{i+3} + c_{i+3} P_{i+3} \]
\[ c_{i+4} = g_{i+3} + P_{i+3} (g_{i+2} + P_{i+2} (g_{i+1} + P_{i+1} g_i) + P_{i+2} P_{i+1} P_i c_i) \]
\[ c_{i+4} = g_{i+3} + P_{i+3} (g_{i+2} + P_{i+2} (g_{i+1} + P_{i+1} g_i)) + P_{i+3} P_{i+2} P_{i+1} P_i c_i \]
Fast Carry Generation

• So for example to generate $c_4$, i.e., $i = 0$,

\[
c_4 = g_3 + p_3 \cdot (g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0)) + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\]

\[
c_4 = G + Pc_0
\]

where,

\[
G = g_3 + p_3 \cdot (g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0))
\]

\[
P = p_3 \cdot p_2 \cdot p_1 \cdot p_0
\]

• See it is quick to evaluate this function

Fast Carry Generation

• We could generate all the carries within an adder block using the previous equations

• However, in order to reduce complexity, a suitable approach is to implement say 4-bit adder blocks with only $c_4$ generated using fast generation.
  – This is used as the carry-in to the next 4-bit adder block
  – Within each 4-bit adder block, conventional RCA is used
Fast Carry Generation

- Conventional ripple carry within 4-bit blocks
- Fast carry generation between 4-bit blocks
- Trade-off between complexity and speed