Digital Hardware: Carry Lookahead Adder

Mike Spivey, Hilary Term 2004

1 2^k-input AND gate

Let's begin with two easier circuits. To build an AND gate with 2^k inputs from 2-input gates, the best plan is to use a tree of gates as shown in the diagram (Figure 1). We can describe this circuit by a purely functional program:

$$
\begin{aligned}
&\mathit{annd} :: [\mathit{Bool}] \to \mathit{Bool} \\
&\mathit{annd}\ [x] = x \\
&\mathit{annd}\ xs = \\
&\quad \mathit{annd}\ (\mathit{take}\ (n/2)\ xs) \wedge \mathit{annd}\ (\mathit{drop}\ (n/2)\ xs) \\
&\quad \textbf{where}\ n = \mathit{length}\ xs
\end{aligned}
$$

Alternatively, we could replace the second equation with

$$\mathit{annd}\ xs = \mathit{annd}\ (\mathit{zipWith}\ (\wedge)\ (\mathit{evens}\ xs)\ (\mathit{odds}\ xs))$$

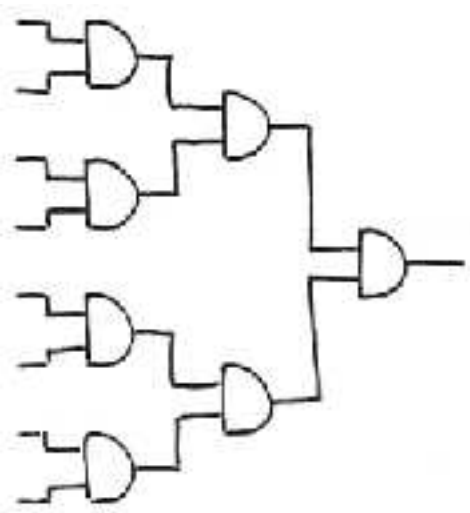

Figure 1: *A tree of* AND *gates*

which describes the tree 'bottom-up' instead of 'top-down'.

Although these are recursive functional programs, if we fix the length of the input list, we can unwind the recursion and get a flat expression for the result – effectively a circuit:

$$\begin{aligned}
&\mathit{annd}\,[x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7] \\
&\quad = \mathit{annd}\,[x_0 \wedge x_1, x_2 \wedge x_3, x_4 \wedge x_5, x_6 \wedge x_7] \\
&\quad = \mathit{annd}\,[(x_0 \wedge x_1) \wedge (x_2 \wedge x_3), (x_4 \wedge x_5) \wedge (x_6 \wedge x_7)] \\
&\quad = \mathit{annd}\,[((x_0 \wedge x_1) \wedge (x_2 \wedge x_3)) \wedge ((x_4 \wedge x_5) \wedge (x_6 \wedge x_7))] \\
&\quad = ((x_0 \wedge x_1) \wedge (x_2 \wedge x_3)) \wedge ((x_4 \wedge x_5) \wedge (x_6 \wedge x_7))
\end{aligned}$$

2 2^k-bit comparator

Let's design a circuit that takes two binary numbers of 2^k bits each and computes one of the values *Lt*, *Eq*, *Gt* to indicate the result of comparing them:

$$\mathit{compare} :: [\mathit{Bit}] \to [\mathit{Bit}] \to \textsc{Leg}$$

where

$$\textbf{data}\ \textsc{Leg} = \mathit{Lt} \mid \mathit{Eq} \mid \mathit{Gt}$$

We'll decide later how to represent *Leg* values as bits. Obviously,

$$\mathit{compare}\,[\,]\,[\,] = \mathit{Eq}$$

Also (remembering that we represent numbers LSB first)

$$\begin{aligned}
&\mathit{compare}\,(\mathit{as} +\!\!+ [a])\,(\mathit{bs} +\!\!+ [b]) = \\
&\quad \textbf{if}\ a < b\ \textbf{then}\ \mathit{Lt} \\
&\quad \textbf{else if}\ a > b\ \textbf{then}\ \mathit{Gt} \\
&\quad \textbf{else}\ \mathit{compare\ as\ bs}
\end{aligned}$$

Let's define a function *cf* and an operator $\odot$ by

$$\begin{aligned}
&\mathit{cf} :: \mathit{Bit} \to \mathit{Bit} \to \textsc{Leg} \\
&\mathit{cf}\ 0\ 0 = \mathit{Eq};\ \mathit{cf}\ 0\ 1 = \mathit{Lt};\ \mathit{cf}\ 1\ 0 = \mathit{Gt};\ \mathit{cf}\ 1\ 1 = \mathit{Eq}
\end{aligned}$$

$$\begin{aligned}
&(\odot) :: \textsc{Leg} \to \textsc{Leg} \to \textsc{Leg} \\
&z \odot \mathit{Lt} = \mathit{Lt} \\
&z \odot \mathit{Eq} = z \\
&z \odot \mathit{Gt} = \mathit{Gt}
\end{aligned}$$

The we find

$$\begin{aligned}
&\mathit{compare}\,(\mathit{as} +\!\!+ [a])\,(\mathit{bs} +\!\!+ [b]) \\
&\quad = \mathit{compare\ as\ bs} \odot \mathit{cf}\ a\ b
\end{aligned}$$

This makes *compare* fit into a well-known pattern of recursion:

$$\mathit{compare\ as\ bs} = \mathit{foldl}\,(\odot)\,\mathit{Eq}\,(\mathit{zipWith\ cf\ as\ bs})$$

But *dot* is associative, so we can play the same trick as with AND:

$$\mathit{compare\ as\ bs} = \mathit{pfold}\,(\odot)\,(\mathit{zipWith\ cf\ as\ bs})$$

where

$$\begin{aligned}
&\mathit{pfold}\,(\oplus)\,[x] = x \\
&\mathit{pfold}\,(\oplus)\,\mathit{xs} = \\
&\quad \mathit{pfold}\,(\oplus)\,(\mathit{zipWith}\,(\oplus)\,(\mathit{evens\ xs})\,(\mathit{odds\ xs}))
\end{aligned}$$

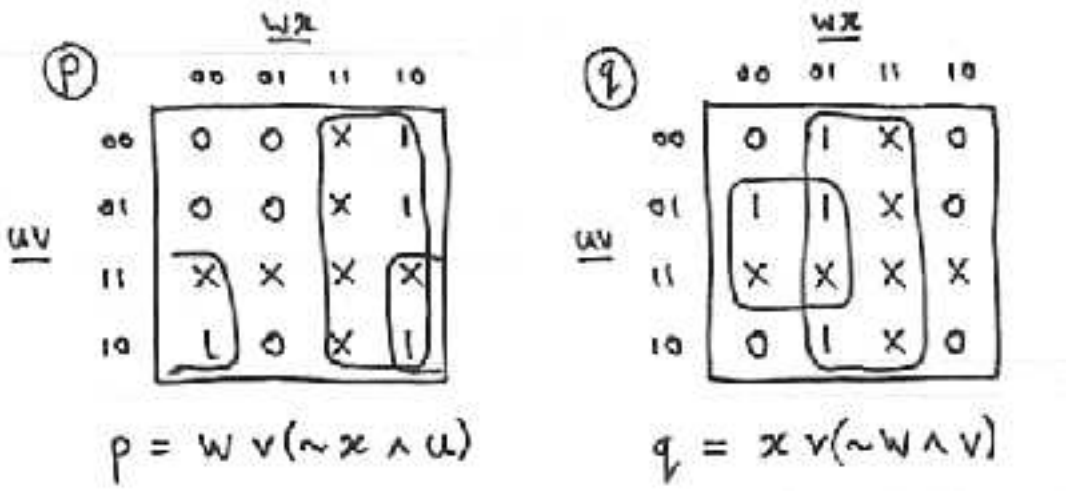

Figure 2: *Karnaugh maps for p and q*

This comparison circuit has size $O(2^k)$ but depth $O(k)$, because the comparison indicators are combined by a binary tree of depth n.

If we represent *Lt*, *Eq*, *Gt* by the bit patterns (0, 1), (0, 0), (1, 0), then the *cf* function becomes $cf\ a\ b = (a \wedge \neg b, b \wedge \neg a)$. And if $(p, q) = (u, v) \oplus (w, x)$, then we get the Karnaugh maps for p and q shown in Figure 2:[1] From these maps we read off $p = w \vee (\neg x \wedge u)$ and $q = x \vee (\neg w \wedge v)$.

3 Carry-lookahead adder

Now, at last, we can look at the carry-lookahead adder. We start from the ripple-carry adder:

$$adder\ as\ bs = ripple\ fulladd\ 0\ (zip\ as\ bs)$$

where $fulladd :: Bit \to (Bit, Bit) \to (Bit, Bit)$ models a full adder, and

$$ripple :: (\alpha \to \beta \to (\gamma, \alpha)) \to \alpha \to [\beta] \to [\gamma]$$

is defined so that

$$\begin{aligned} &ripple\ f\ c_0\ [x_0, x_1, \ldots, x_{n-1}] = [y_0, y_1, \ldots, y_{n-1}] \\ &\quad \textbf{where} \\ &\qquad (y_0, c_1) = f\ c_0\ x_0 \\ &\qquad (y_1, c_2) = f\ c_1\ x_1 \\ &\qquad\quad \ldots \\ &\qquad (y_{n-1}, c_n) = f\ c_n\ x_n \end{aligned}$$

Now let's concentrate on the carry bits:

[1] In the Karnaugh maps, I've marked with an X those entries that we don't care about, because the inputs $(u, v) = (1, 1)$ or $(w, x) = (1, 1)$ can never occur. The circuit will work properly whatever output the $\odot$ box produces for these inputs, so we are free to choose each X as 0 or 1, whichever gives the fewest, largest rectangles in the Karnaugh map.

$$
\begin{aligned}
&\mathit{fulladd}\ c\ (0,0) = (?, 0)\\
&\mathit{fulladd}\ c\ (0,1) = (?, c)\\
&\mathit{fulladd}\ c\ (1,0) = (?, c)\\
&\mathit{fulladd}\ c\ (1,1) = (?, 1)
\end{aligned}
$$

So the carry bit that is output by the full adder is one of these three things: always 0, the same as the carry input c, or always 1, depending on the two input bits (a, b). Let's call these three possibilities kill (K), propagate (P) and generate (G).

We can define a function *kpg* that returns these three values as appropriate:

$$
\begin{aligned}
&\textbf{data}\ \mathit{KPG} = K \mid P \mid G\\
&\\
&\mathit{kpg} :: (\mathit{Bit}, \mathit{Bit}) \to \mathit{KPG}\\
&\mathit{kpg}\ (0,0) = K;\ \mathit{kpg}\ (0,1) = P;\ \mathit{kpg}\ (1,0) = P;\ \mathit{kpg}\ (1,1) = G
\end{aligned}
$$

Now let's agree to replace the carry input and output of each full adder by either K (if it was 0 before) or G (if it was 1). This gives us a new version of the full adder:

$$
\begin{aligned}
&\mathit{kpg\_add} :: \mathit{KPG} \to (\mathit{Bit}, \mathit{Bit}) \to (\mathit{Bit}, \mathit{KPG})\\
&\mathit{kpg\_add}\ w\ (a,b) = (\mathit{sumbit}\ w\ (a,b), w \otimes \mathit{kpg}\ (a,b))
\end{aligned}
$$

Where $\mathit{sumbit} :: \mathit{KPG} \to (\mathit{Bit}, \mathit{Bit}) \to \mathit{Bit}$ is an appropriate function, and $\otimes$ is the binary operation defined by

$$
\begin{aligned}
&(\otimes) :: \mathit{KPG} \to \mathit{KPG} \to \mathit{KPG}\\
&z \otimes K = K\\
&z \otimes P = z\\
&z \otimes G = G
\end{aligned}
$$

This gives us a new version of the ripple-carry adder:

$$
\mathit{rip\_kpg}\ \mathit{as}\ \mathit{bs} = \mathit{ripple}\ \mathit{kpg\_add}\ K\ (\mathit{zip}\ \mathit{as}\ \mathit{bs})
$$

We have not yet made any improvement – the depth of this circuit is still $O(n)$ for adding n-bit numbers. But now we calculate as follows:

$$
\begin{aligned}
&\mathit{rip\_kpg}\ [a_0, \ldots, a_{n-1}]\ [b_0, \ldots, b_{n-1}] = [s_0, \ldots, s_{n-1}]\\
&\quad \textbf{where}\\
&\qquad (s_0, w_1) = \mathit{kpg\_add}\ K\ (a_0, b_0)\\
&\qquad (s_1, w_2) = \mathit{kpg\_add}\ w_1\ (a_1, b_1)\\
&\qquad\quad \ldots\\
&\qquad (s_{n-1}, w_n) = \mathit{kpg\_add}\ w_{n-1}\ (a_{n-1}, b_{n-1})
\end{aligned}
$$

So for each i,

$$
\begin{aligned}
&(s_i, w_{i+1}) = \mathit{kpg\_add}\ w_i\ (a_i, b_i)\\
&\quad = (\mathit{sumbit}\ w_i\ (a_i, b_i), w_i \otimes \mathit{kpg}\ (a_{i-1}, b_{i-1})
\end{aligned}
$$

Thus

$$
\begin{aligned}
&[s_0, \ldots, s_{n-1}]\\
&\quad = \mathit{zipWith}\ \mathit{sumbit}\ [w_0, \ldots, w_{n-1}]\ [(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})]
\end{aligned}
$$

with $w_0 = K$ and $w_{i+1} = w_i \otimes kpg\,(a_i, b_i)$. We see that $w_i = K \otimes kpg\,(a_0, b_0) \otimes \cdots \otimes kpg\,(a_{i-1}, b_{i-1})$. This gives us a new adder:

```
cla as bs =
  zipWith sumbit carries inputs
  where
    carries = par_prefix (⊗) K (map kpg inputs)
    inputs = zip as bs
```

This uses a function *par_prefix*, defined so that

$$\begin{aligned}&par\_prefix\ (\%)\ a\ [x_0, \ldots, x_{n-1}]\\ &\quad = [a,\ a \,\%\, x_0,\ a \,\%\, x_0 \,\%\, x_1, \ldots,\ a \,\%\, x_0 \,\%\, \cdots \,\%\, x_{n-2}]\end{aligned}$$

Now for the really clever bit: we can define *par_prefix* in such a way that the n results can be evaluated in parallel, provided $\%$ is associative and n is a power of 2. Observe that if $xs = [x_0, ..., x_{2k-1}]$ then

$$\begin{aligned}&par\_prefix\ (\%)\ a\ xs\\ &\quad = [a,\ a \,\%\, x_0,\ a \,\%\, x_0 \,\%\, x_1,\ a \,\%\, x_0 \,\%\, x_1 \,\%\, x_2, \ldots,\\ &\qquad\quad a \,\%\, x_0 \,\%\, x_1 \,\%\, \cdots \,\%\, x_{2k-3},\\ &\qquad\quad a \,\%\, x_0 \,\%\, x_1 \,\%\, \cdots \,\%\, x_{2k-3} \,\%\, x_{2k-2}]\\ &\quad = [y_0,\ y_0 \,\%\, x_1,\ y_1,\ y_1 \,\%\, x_2, \ldots,\ y_{k-1},\ y_{k-1} \,\%\, x_{2k-2}]\end{aligned}$$

where

$$\begin{aligned}&[y_0, y_1, \ldots, y_{k-1}]\\ &\quad = [a,\ a \,\%\, (x_0 \,\%\, x_1), \ldots,\ a \,\%\, (x_0 \,\%\, x_1) \,\%\, \cdots \,\%\, (x_{2k-4} \,\%\, x_{2k-3})]\\ &\quad = par\_prefix\ (\%)\ a\ [x_0 \,\%\, x_1,\ x_2 \,\%\, x_3, \ldots,\ x_{2k-4} \,\%\, x_{2k-3}]\\ &\quad = par\_prefix\ (\%)\ a\ (zipWith\ (\%)\ (evens\ xs)\ (odds\ xs))\end{aligned}$$

So we can define *par_prefix* as follows:

```
par_prefix (%) a [x] = [a]
par_prefix (%) a xs =
  concat (zipWith g es (par_prefix (%) a (zipWith (%) es os)))
  where
    es = evens xs; os = odds xs
    g x y = [y, y % x]
```

The size and depth of this prefix circuit satisfy

$$\begin{aligned}&S(2^k) = O(2^{k-1}) + S(2^{k-1}) + O(2^{k-1}),\\ &D(2^k) = O(1) + D(2^{k-1}) + O(1).\end{aligned}$$

So $S(2^k) = O(2^k)$ and $D(2^k) = O(k)$. The resulting adder circuit is sketched in Figure 3.

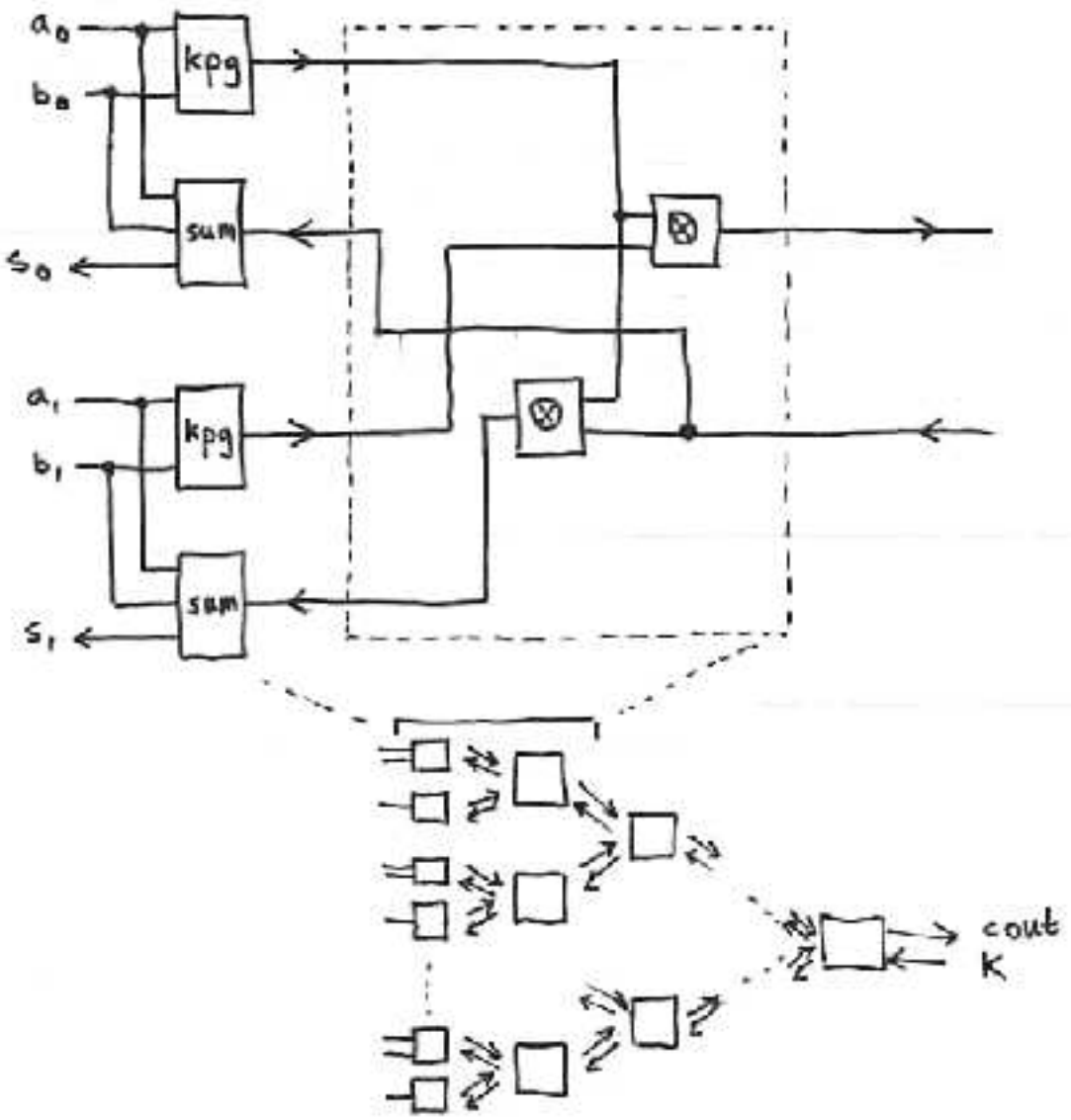

Figure 3: *Carry lookahead adder*