COMS12200 lab. worksheet: week #7

Although some questions have a written solutions below, for others it makes more sense to experiment in a hands-on manner: the Logisim project

http://www.cs.bris.ac.uk/home/page/teaching/material/arch_new/sheet/lab-7.s.circ

supports such cases.

Input: Two unsigned, n-bit, base-2 integers x and y
Output: An unsigned, $2n$-bit, base-2 integer $r = y \cdot x$

```
1 t ← 0
2 for i = n − 1 downto 0 step −1 do
3 |   t ← 2 · t
4 |   if y_i = 1 then
5 |   |   t ← t + x
6 |   end
7 end
8 return t
```

Algorithm 1: An algorithm for left-to-right, bit-serial multiplication.

S2. a Per the description in the lecture(s), the combinatorial design is relatively simple: although it potentially needs a *lot* of logic (for larger n), it is combinatorial meaning a) has a lower latency, and b) is simpler to implement and test. The implementation directly mirrors the design on paper, in that it consists of three layers:

- The left layer is comprised of n groups of n AND gates: the i-th group computes $x_j \wedge y_i$ for $0 \leq j < n$, meaning it outputs either 0 if $y_i = 0$ or x if $y_i = 1$.
- The middle layer is comprised of n left-shift components. The i-th component shifts by a fixed distance of i bits, meaning the output is either 0 if $y_i = 0$, or $x \cdot 2^i$ if $y_i = 1$.
- The right layer is a balanced, binary tree of adder components: these accumulate the partial products resulting from the middle layer, meaning the output is

$$r = \sum_{i=0}^{n-1} y_i \cdot x \cdot 2^i = y \cdot x$$

as required.

Imagine we want to compute $r = x \cdot y$ for $x = 8_{(10)}$ and $y = 14_{(10)} = 00001110_{(2)}$. Abusing notation a little wrt. AND, we find that the first two layers combine to compute

$$\begin{array}{lcccl}
p_0 & = & (x \wedge y_0) \ll 0 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 0 = 00000000_{(2)} \\
p_1 & = & (x \wedge y_1) \ll 1 & = & (00001000_{(2)} \wedge 11111111_{(2)}) \ll 1 = 00010000_{(2)} \\
p_2 & = & (x \wedge y_2) \ll 2 & = & (00001000_{(2)} \wedge 11111111_{(2)}) \ll 2 = 00100000_{(2)} \\
p_3 & = & (x \wedge y_3) \ll 3 & = & (00001000_{(2)} \wedge 11111111_{(2)}) \ll 3 = 01000000_{(2)} \\
p_4 & = & (x \wedge y_4) \ll 4 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 4 = 00000000_{(2)} \\
p_5 & = & (x \wedge y_5) \ll 5 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 5 = 00000000_{(2)} \\
p_6 & = & (x \wedge y_6) \ll 6 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 6 = 00000000_{(2)} \\
p_7 & = & (x \wedge y_7) \ll 7 & = & (00001000_{(2)} \wedge 00000000_{(2)}) \ll 7 = 00000000_{(2)}
\end{array}$$

after which the final layer computes (with parentheses denoting the adder instances)

$$\begin{array}{rcl}
r & = & ((p_0 + p_1) + (p_2 + p_3)) + \\
 & & ((p_4 + p_5) + (p_6 + p_7)) \\
 & = & ((00000000_{(2)} + 00010000_{(2)}) + (00000000_{(2)} + 00000000_{(2)})) + \\
 & & ((00000000_{(2)} + 00000000_{(2)}) + (00000000_{(2)} + 00000000_{(2)})) \\
 & = & ((0_{(10)} + 16_{(10)}) + (32_{(10)} + 64_{(10)})) + \\
 & & ((0_{(10)} + 0_{(10)}) + (0_{(10)} + 0_{(10)})) \\
 & = & 112_{(10)}
\end{array}$$

b Algorithm 1 is for left-to-right bit-serial multiplication; note that it processes bits within the multiplier y from most- to least-significant. This is essentially the algorithm we need to implement. The counter component from the previous question was developed in such a way that it can control the loop. Since the loop iterates downward in this case, it seems we may need to alter the counter. It *is* of course possible to do so, but the only place we use i in the i-th iteration is to inspect the y_i: provided we inspect the correct y_i in each iteration and perform n iterations overall, we can actually leave the counter as it is.

This is achieved by redefining the problem: instead of inspecting y_i based on i, we instead move the i-th bit of y into a fixed index. More concretely we update y via $y' \leftarrow y \ll 1$, i.e., left-shift it, after each iteration; this means we can just inspect the 7-th bit of y in *every* iteration, because in each successive i-th iteration it will hold the i-th bit.

Overall, and in addition to the counter, we need a data-path that includes

- combinatorial logic to compute
  - $t' \leftarrow 2 \cdot t + y_i \cdot x$, and
  - $y' \leftarrow y \ll 1$,

  and
- a pair of input and output registers to store t and y, plus an input register to store x (which is not updated, so requires no output register).

Crucially, as discussed in the lecture(s),

- $2 \cdot t$ can be realised using a left-shift (i.e., $2 \cdot t \equiv t \ll 1$) and
- $y_i \cdot x$ can be realised using a multiplexer (since $y_i \in \{0, 1\}$, $y_i \cdot x$ evaluates to 0 or x).

Having first implemented said data-path and connected it to the control-path (i.e., the counter), imagine we want to compute $r = x \cdot y$ for $x = 8_{(10)}$ and $y = 14_{(10)} = 00001110_{(2)}$. We initiate the computation by first setting x and y, then $req = 1$ to signal a request. The multiplier steps through the following

i	t	y	y_7	t'	y'	
	0					
0	0	$00001110_{(2)}$	0	0	$0001110_{(2)}$	$t' \leftarrow 2 \cdot t$
1	0	$00011100_{(2)}$	0	0	$0011100_{(2)}$	$t' \leftarrow 2 \cdot t$
2	0	$00111000_{(2)}$	0	0	$0111000_{(2)}$	$t' \leftarrow 2 \cdot t$
3	0	$01110000_{(2)}$	0	0	$1110000_{(2)}$	$t' \leftarrow 2 \cdot t$
4	0	$11100000_{(2)}$	1	8	$1100000_{(2)}$	$t' \leftarrow 2 \cdot t + x$
5	8	$11000000_{(2)}$	1	24	$1000000_{(2)}$	$t' \leftarrow 2 \cdot t + x$
6	24	$10000000_{(2)}$	1	56	$0000000_{(2)}$	$t' \leftarrow 2 \cdot t + x$
7	56	$00000000_{(2)}$	0	112	$0000000_{(2)}$	$t' \leftarrow 2 \cdot t$
	112					

at which point it sets $ack = 1$ to signal computation is complete; once we have the result $r = 112_{(10)}$, we can set $req = 1$ at which point the multiplier sets $ack = 0$ and is ready for any subsequent computation.