COMS12200 lab. worksheet: week #5

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-5.s.circ

supports such cases.

S1. There is a set of solutions available at

http://www.cs.bris.ac.uk/home/page/teaching/material/arch_old/sheet/exam-logic_minimise.s.pdf

S2. Since the same steps apply to each component, we address them one at a time:

a The truth table for a 2-input multiplexer is

MUX2			
c	x	y	r
0	0	**?**	0
0	1	**?**	1
1	**?**	0	0
1	**?**	1	1

where **?** denotes the don't care state, meaning the output can be described as

$$r = (\neg c \wedge x) \vee (c \wedge y)$$

This translates easily into a simulatable Logisim implementation because all gate types are available. Furthermore, we can immediately rewrite the expression as follows

$$\begin{array}{rcll} r & = & (\neg c \wedge x) \vee (c \wedge y) & \\ & \equiv & \neg\neg((\neg c \wedge x) \vee (c \wedge y)) & \text{(involution)} \\ & \equiv & \neg(\neg(\neg c \wedge x) \wedge \neg(c \wedge y)) & \text{(de Morgan)} \\ & \equiv & ((c \overline{\wedge} c) \overline{\wedge} x) \overline{\wedge} (c \overline{\wedge} y) & \end{array}$$

meaning a NAND-only implementation is therefore

$$\begin{array}{rcll} t_0 & = & c & \overline{\wedge}\ c \\ t_1 & = & t_0 & \overline{\wedge}\ x \\ t_2 & = & c & \overline{\wedge}\ y \\ t_3 & = & t_1 & \overline{\wedge}\ t_2 \end{array}$$

where $r \equiv t_3$.

b The truth table for a full-adder is

Full-Adder				
ci	x	y	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

meaning the outputs can be described as

$$\begin{array}{rcl} co & = & (x \wedge y) \vee ((x \oplus y) \wedge ci) \\ s & = & x \oplus y \oplus ci \end{array}$$

This translates easily into a simulatable Logisim implementation because all gate types are available. The worksheet in week #2 used the fact that

$$x \oplus y \equiv (x \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)) \mathbin{\overline{\wedge}} (y \mathbin{\overline{\wedge}} (x \mathbin{\overline{\wedge}} y)).$$

We can use this again, rewriting the carry-out expression into

$$\begin{array}{rcll} co & = & (x \wedge y) \vee ((x \oplus y) \wedge ci) & \\ & \equiv & \neg\neg((x \wedge y) \vee ((x \oplus y) \wedge ci)) & \text{(involution)} \\ & \equiv & \neg(\neg(x \wedge y) \wedge \neg((x \oplus y) \wedge ci)) & \text{(de Morgan)} \\ & \equiv & (x \mathbin{\overline{\wedge}} y) \mathbin{\overline{\wedge}} ((x \oplus y) \mathbin{\overline{\wedge}} ci) & \end{array}$$

and producing the NAND-only implementation

$$\begin{array}{rcl} t_0 & = & x \mathbin{\overline{\wedge}} y \\ t_1 & = & x \mathbin{\overline{\wedge}} t_0 \\ t_2 & = & y \mathbin{\overline{\wedge}} t_0 \\ t_3 & = & t_1 \mathbin{\overline{\wedge}} t_2 \\ t_4 & = & t_3 \mathbin{\overline{\wedge}} ci \\ t_5 & = & t_3 \mathbin{\overline{\wedge}} t_4 \\ t_6 & = & ci \mathbin{\overline{\wedge}} t_4 \\ t_7 & = & t_5 \mathbin{\overline{\wedge}} t_6 \\ t_8 & = & t_0 \mathbin{\overline{\wedge}} t_4 \end{array}$$

where $s \equiv t_7$ and $co \equiv t_8$.

S3. b Whereas a full-adder computes the carry-out and sum resulting from

$$x + y + ci,$$

with a full-subtractor we are instead interested in the borrow-out and difference from

$$x - y - bi.$$

As such, the associated truth table is

Full-Subtractor				
bi	x	y	bo	d
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

meaning the outputs can be described as

$$\begin{array}{rcl} bo & = & (\neg x \wedge y) \vee (\neg(x \oplus y) \wedge bi) \\ d & = & x \oplus y \oplus bi \end{array}$$

As with the full-adder, this translates easily into a simulatable Logisim implementation because all gate types are available. Although the question does not ask for a NAND-only implementation, notice that the same approach is possible as with the full-adder: the only difference is a couple of NOT gates.

S4. As the question suggests, both components need a little thought as to design but use of Karnaugh maps is central throughout.

a First, notice that rotation (or shift-type operations generally) by a fixed distance requires no actual computation: it just permutes bits in the input to form the output. For example, rotation of a 28-bit x by a distance of 1 bit means

$$t_{i+1 \pmod{28}} = x_i$$

for $0 \leq i < 28$. Put another way, we just connect the i-th bit of x to the $(i+1 \pmod{28})$-th bit of the rotation output t.

Armed with this knowledge, we can form two 28-bit values

$$\begin{aligned} p &= x \lll 1 \\ q &= p \lll 1 \end{aligned}$$

where both apply the strategy above so are basically just permutations of x. Having generated both options, we can then select between them (based on y) to form the output. To achieve this we use a 2-input, 28-bit multiplexer whose control signal is

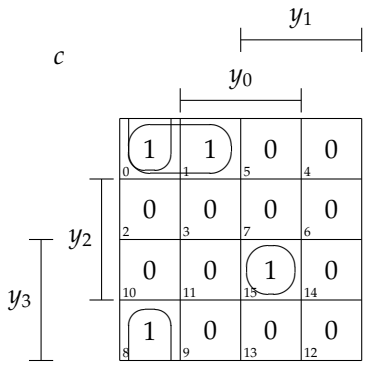
$$c = f(y) = \begin{cases} 1 & \text{if } y \in \{0, 1, 8, 15\}, \text{ meaning } p \text{ should be selected} \\ 0 & \text{if } y \notin \{0, 1, 8, 15\}, \text{ meaning } q \text{ should be selected} \end{cases}$$

and whose output is connected to r: if x should be rotated by 1 bit $c = 1$ so the multiplexer sets $r = p = x \lll 1$, but if x should be rotated by 2 bits $c = 0$ so the multiplexer sets $r = q = p \lll 1 = (x \lll 1) \lll 1 = x \lll 2$.

All we need to do is generate c based on y: by writing the truth table

y_3	y_2	y_1	y_1	$c = f(y)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

and translating it into the Karnaugh map



we can do this via the Boolean expression

$$\begin{aligned} c = (& \quad \neg x_2 \wedge \neg x_1 \wedge \neg x_0) \vee \\ (& \neg x_3 \wedge \neg x_2 \wedge \neg x_1) \vee \\ (& \quad x_3 \wedge x_2 \wedge x_1 \wedge x_0) \end{aligned}$$

or even more simply as

$$c = (a_0 \wedge b) \vee (a_3 \wedge b) \vee (x_3 \wedge x_2 \wedge x_1 \wedge x_0)$$

where

$$\begin{aligned} a_0 &= \neg x_0 \\ a_1 &= \neg x_1 \\ a_2 &= \neg x_2 \\ a_3 &= \neg x_3 \\ b &= a_2 \wedge a_1 \end{aligned}$$

As an aside, implementing this approach in Logisim is slightly complicated by the built-in component for rotation: although using it reduces the amount of work involved, it rotates the input by a variable rather than fixed distance. So even if we hard-code the distance (e.g., by fixing bits to a constant, or even *GND* and/or V_{dd}) to make it do what we want, this is wasteful wrt. the total number of logic gates since we *know* the result required can be generated with none at all. Put another way, using a hand-specified component for rotation by a fixed distance of 1 bit will be much more efficient wrt. area.

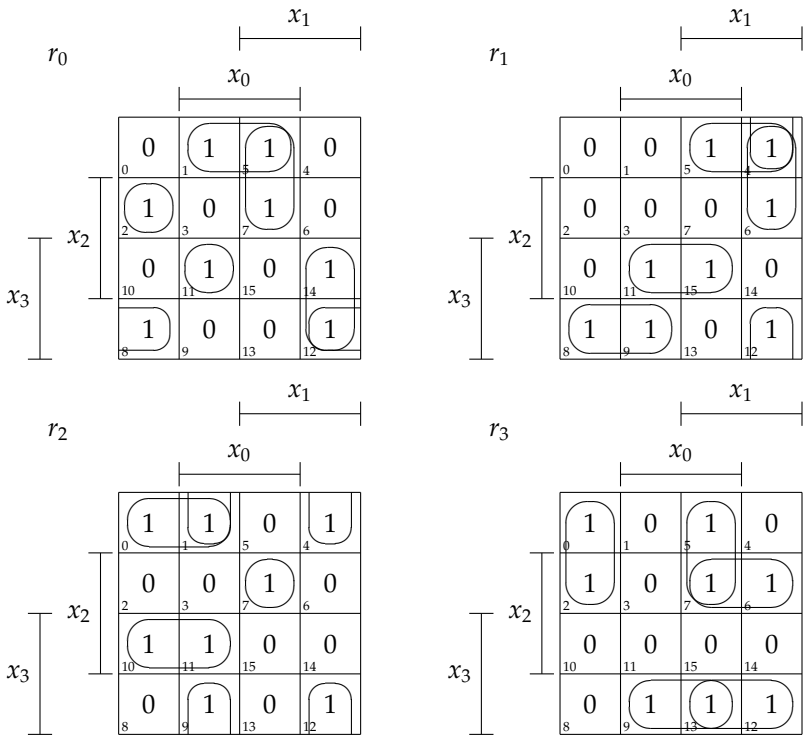
b The first step is to notice that we can rewrite the truth table in binary so it reads

x	$r = f(x)$
$0000_{(2)}$	$1100_{(2)}$
$0001_{(2)}$	$0101_{(2)}$
$0010_{(2)}$	$0110_{(2)}$
$0011_{(2)}$	$1011_{(2)}$
$0100_{(2)}$	$1001_{(2)}$
$0101_{(2)}$	$0000_{(2)}$
$0110_{(2)}$	$1010_{(2)}$
$0111_{(2)}$	$1101_{(2)}$

x	$r = f(x)$
$1000_{(2)}$	$0011_{(2)}$
$1001_{(2)}$	$1110_{(2)}$
$1010_{(2)}$	$1111_{(2)}$
$1011_{(2)}$	$1000_{(2)}$
$1100_{(2)}$	$0100_{(2)}$
$1101_{(2)}$	$0111_{(2)}$
$1110_{(2)}$	$0001_{(2)}$
$1111_{(2)}$	$0010_{(2)}$

This highlights the fact we have a 4-bit output $r = f(x)$ and a 4-bit input x; we can implement the behaviour required via a set of Boolean expressions, each of which produces the i-th bit of said result given x (i.e., the bits x_0, x_1, x_2 and x_3) as input. Put another way, we need to write four separate expressions f_i st. $r_i = f_i(x)$ for $0 \leq i < 4$.

To do this, we first translate the truth table into four Karnaugh maps, one for each r_i



and then further into Boolean expressions:

$$
\begin{array}{rcccccccccccc}
r_0 & = & (& \neg x_3 & \wedge & \neg x_2 & & & \wedge & x_0 &) & \vee \\
 & & (& \neg x_3 & & & \wedge & x_1 & \wedge & x_0 &) & \vee \\
 & & (& x_3 & & & \wedge & x_1 & \wedge & \neg x_0 &) & \vee \\
 & & (& x_3 & \wedge & \neg x_2 & & & \wedge & \neg x_0 &) & \vee \\
 & & (& \neg x_3 & \wedge & x_2 & \wedge & \neg x_1 & \wedge & \neg x_0 &) & \vee \\
 & & (& x_3 & \wedge & x_2 & \wedge & \neg x_1 & \wedge & x_0 &) & \\
 & & & & & & & & & & & \\
r_1 & = & (& \neg x_3 & \wedge & \neg x_2 & \wedge & x_1 & & &) & \vee \\
 & & (& \neg x_3 & & & \wedge & x_1 & \wedge & \neg x_0 &) & \vee \\
 & & (& x_3 & \wedge & x_2 & \wedge & & & x_0 &) & \vee \\
 & & (& x_3 & \wedge & \neg x_2 & \wedge & \neg x_1 & & &) & \vee \\
 & & (& & & \neg x_2 & \wedge & x_1 & \wedge & \neg x_0 &) & \\
 & & & & & & & & & & & \\
r_2 & = & (& & & \neg x_2 & \wedge & x_1 & \wedge & \neg x_0 &) & \vee \\
 & & (& & & \neg x_2 & \wedge & \neg x_1 & \wedge & x_0 &) & \vee \\
 & & (& \neg x_3 & \wedge & x_2 & \wedge & x_1 & \wedge & x_0 &) & \vee \\
 & & (& \neg x_3 & \wedge & \neg x_2 & \wedge & \neg x_1 & & &) & \vee \\
 & & (& x_3 & \wedge & x_2 & \wedge & \neg x_1 & & &) & \\
 & & & & & & & & & & & \\
r_3 & = & (& \neg x_3 & & & \wedge & \neg x_1 & \wedge & \neg x_0 &) & \vee \\
 & & (& \neg x_3 & & & \wedge & x_1 & \wedge & x_0 &) & \vee \\
 & & (& x_3 & \wedge & \neg x_2 & & & \wedge & x_0 &) & \vee \\
 & & (& \neg x_3 & \wedge & x_2 & \wedge & x_1 & & &) & \vee \\
 & & (& x_3 & \wedge & \neg x_2 & \wedge & x_1 & & &) &
\end{array}
$$

Various next steps to further simplify these equations can be made. The best known solution uses only 14 gates [1], most of which are XOR. This presents a fairly subtle problem, in the sense that a naive gate count will ignore the cost of any derived gates (an XOR might internally require two AND gates, an OR and a NOT for instance). One way to resolve this would be to count a single gate type (e.g., NAND or NOR), or even the number of underlying transistors instead.

This level of detail clearly *is* important; without it, we risk producing an unfair comparison and hence sub-optimal result. However, it is also a little beyond the scope of the worksheet. As such, having produced a set of common sub-terms

$$
\begin{array}{rcl}
a_0 & = & \neg x_0 \\
a_1 & = & \neg x_1 \\
a_2 & = & \neg x_2 \\
a_3 & = & \neg x_3 \\
b_0 & = & a_1 \wedge a_0 \\
b_1 & = & a_1 \wedge x_0 \\
b_2 & = & x_1 \wedge a_0 \\
b_3 & = & x_1 \wedge x_0 \\
c_0 & = & a_3 \wedge a_2 \\
c_1 & = & a_3 \wedge x_2 \\
c_2 & = & x_3 \wedge a_2 \\
c_3 & = & x_3 \wedge x_2 \\
d_0 & = & a_2 \wedge b_2 \\
d_1 & = & a_3 \wedge b_3
\end{array}
$$

[1] http://eprint.iacr.org/2011/475

we can at least simplify the expressions somewhat by rewriting them as follows:

$$
\begin{array}{lcllllll}
r_0 & = & (c_0 \wedge x_0) & \vee & (x_3 \wedge b_2) & \vee & \\
 & & (c_2 \wedge a_0) & \vee & (c_1 \wedge b_0) & \vee & \\
 & & (c_3 \wedge b_1) & & & \vee & d_1 \\
 & & & & & & \\
r_1 & = & (c_0 \wedge x_1) & \vee & (a_3 \wedge b_2) & \vee & \\
 & & (c_3 \wedge x_0) & \vee & (c_2 \wedge a_1) & \vee & d_0 \\
 & & & & & & \\
r_2 & = & (a_2 \wedge b_1) & \vee & (c_1 \wedge b_3) & \vee & \\
 & & (a_1 \wedge (c_0 \vee c_3)) & & & \vee & d_0 \\
 & & & & & & \\
r_3 & = & (a_3 \wedge b_0) & \vee & (c_2 \wedge x_0) & \vee & \\
 & & (x_1 \wedge (c_1 \vee c_2)) & & & \vee & d_1
\end{array}
$$