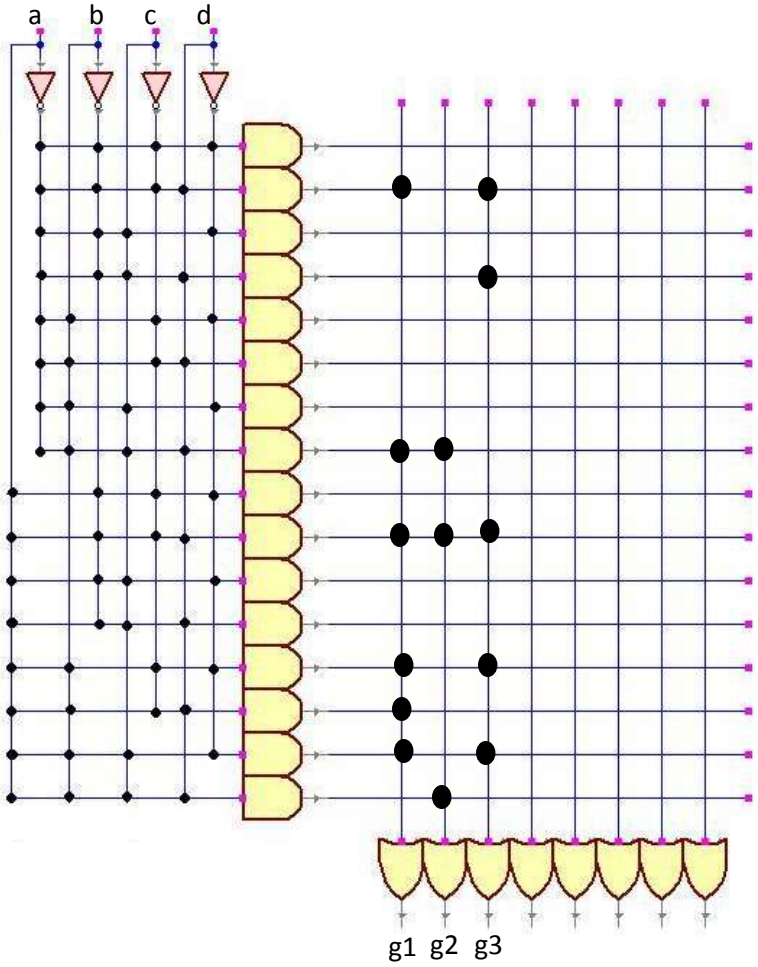
CSE370 HW5 Solutions (Winter 2010)

1. **ROM/PAL/PLA**

   We have the following equations for g1, g2, and g3:

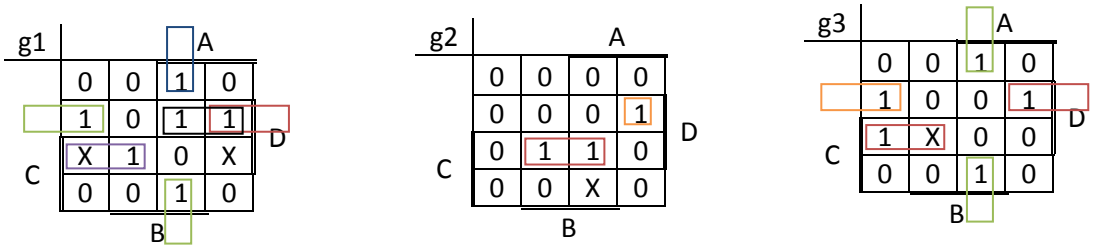
   g1(a, b, c, d) = ∑ m(1, 7, 9, 12, 13, 14) + d(3, 11)
   g2(a, b, c, d) = ∑ m(7, 9, 15) + d(14)
   g3(a, b, c, d) = ∑ m(1, 3, 9, 12, 14) + d(7)

   We can directly implement this using the ROM by hooking up the rows for each of the minterms to each output:



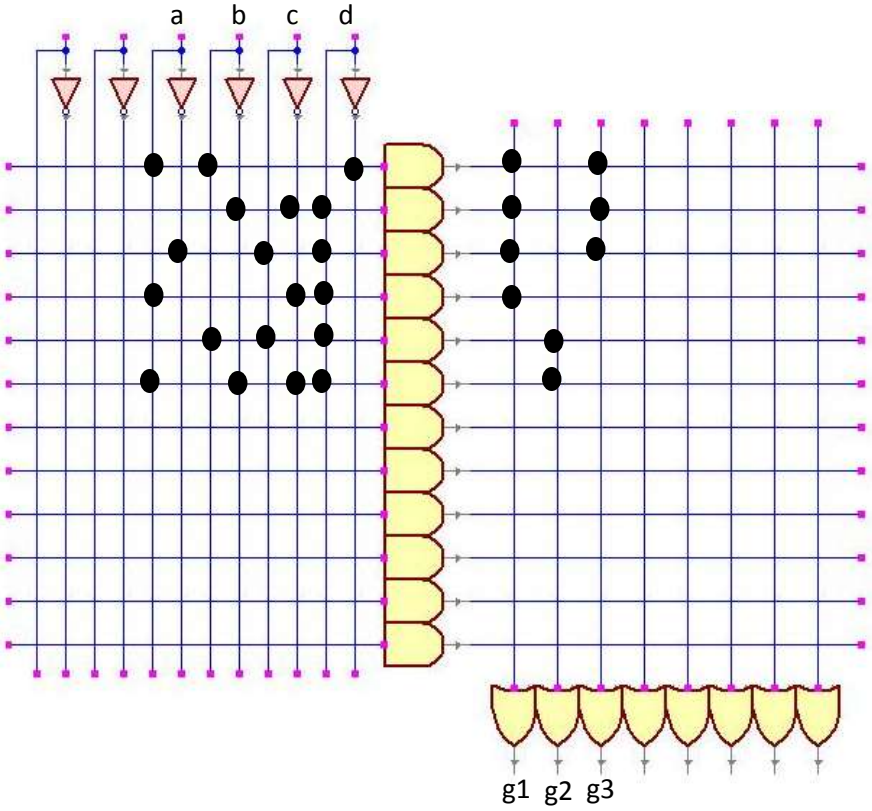
   Now before we implement for the PLA we will first try to minimize the number of AND gates we'd need to use. The easiest technique for doing this is to look at the K-maps for each of the three functions we want to generate. Then look for groupings that are the same in multiple K-maps. These will result in common terms that can be shared.

   So let us first look at the K-maps:

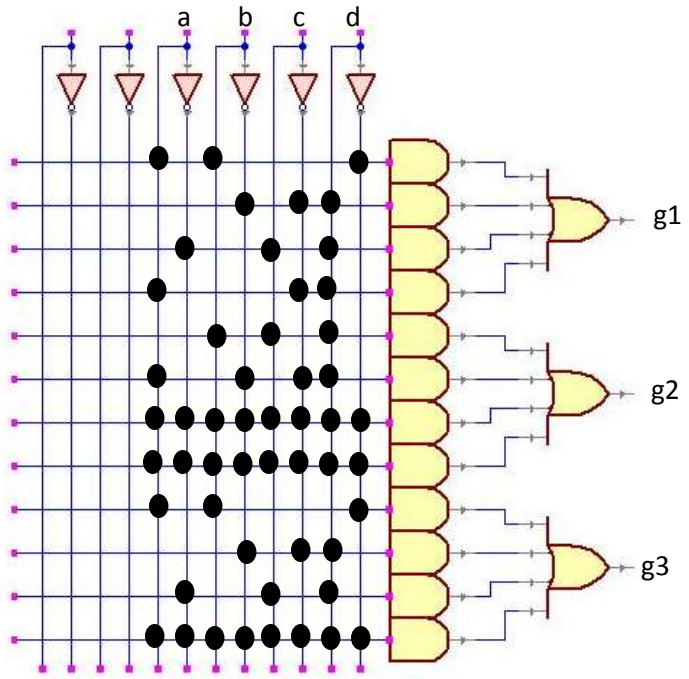


Notice that I was able to cover every 1 with only a total of 6 terms, giving the equations:
g1 = ABD' + B'C'D + A'CD + AC'D
g2 = BCD + AB'C'D
g3 = ABD' + B'C'D + A'CD

Notice that g3 shares all three terms with g1. Also, this is not the only way to reduce it to 6 terms. We can now implement the functions using these terms in our PLA:



Also, the same terms as before can be used in our PAL, giving the following implementation:



Note that we need to set the unused AND gates to always output 0 in the PAL.

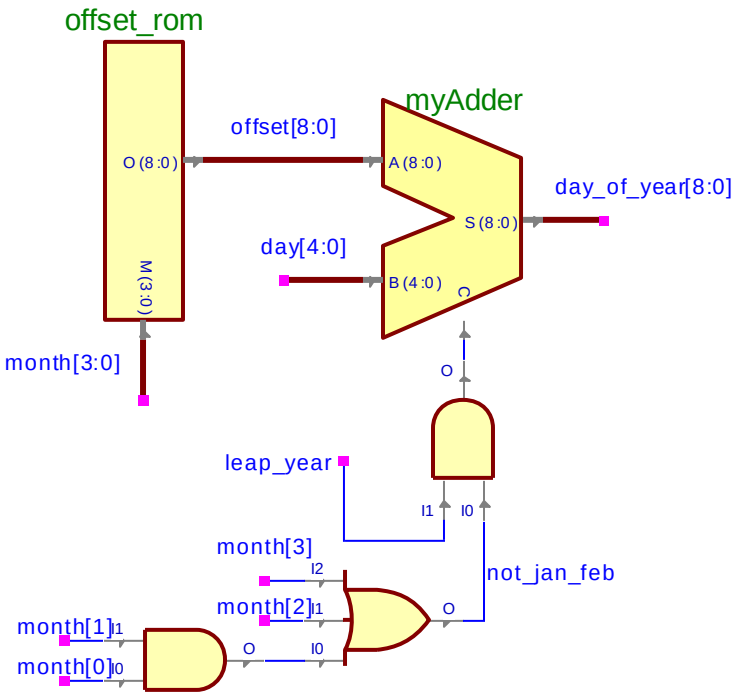
2. **CLD2e, 5.4**

For **part (a)**, we need to build some logic to generate our month offset. We can do this easily with a ROM (or any other of the logic elements) so long as the behavior of our component follows this truth table:

m_3	m_2	m_1	m_0	o_8	o_7	o_6	o_5	o_4	o_3	o_2	o_1	o_0	Offset
0	0	0	0	x	x	x	x	x	x	x	X	x	x
0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1	1	1	1	31
0	0	1	1	0	0	0	1	1	1	0	1	1	59
0	1	0	0	0	0	1	0	1	1	0	1	0	90
0	1	0	1	0	0	1	1	1	1	0	0	0	120
0	1	1	0	0	1	0	0	1	0	1	1	1	151
0	1	1	1	0	1	0	1	1	0	1	0	1	181
1	0	0	0	0	1	1	0	1	0	1	0	0	212
1	0	0	1	0	1	1	1	1	0	0	1	1	243
1	0	1	0	1	0	0	0	1	0	0	0	1	273
1	0	1	1	1	0	0	1	1	0	0	0	0	304
1	1	0	0	1	0	1	0	0	1	1	1	0	334
1	1	0	1	x	x	x	x	x	x	x	X	x	x
1	1	1	0	x	x	x	x	x	x	x	X	x	x
1	1	1	1	x	x	x	x	x	x	x	X	x	x

For **part (b)**, we need to combine our ROM from part (a) with an adder to get the output.

Some people used two adders, one to add in the leap year and one to add the day to the offset. We can actually do it with a single adder if we route the leap year flag into the adder's carry-in. One caveat though is that we only add in the leap year if the month is not January or February. So, we add a few gates (or you could use a 4:16 decoder) to detect this. The resulting completed circuit is:



For **part (c)**, the idea was to try to get you thinking about how your adder + logic would get synthesized and implemented on the FPGA. Since the FPGA only has lookup tables (actually there are adders built in, but let's assume not) we are trying to get you to think about how many of these lookup tables would get used for your circuit.

We know that we have some logic to decide whether to add the leap-year or not, so let's estimate about 1 LUT for that.

For the adder, a good rule of thumb is that each bit of the output will take at least 1 LUT. So this can require no less than 9 LUTs. We also know that we can use 2 LUT's per bit to implement the standard ripple-carry adder (one for sum, one for carry). So there should be no more than 18 LUTs.

So we can estimate that we will approximately use 10-19 LUTs. For grading this I accepted anything near this range.

Note: Since our adder is actually only adding a 5-bit number to a 9-bit number, some of the logic would actually get reduced. So for the ripple-carry adder, it would probably take no more than 14 LUTs [(2 LUTs x 5-bits) + (1 LUT x 4-bits)].

3. **CLD2e, 6.12**

These problems can be a bit tricky. Here technique that I used to try to determine which of the given memory elements each circuit represented.

i. **R-S, D, or J-K?:** First I looked to determine what the behavior of the inputs was. For the R-S type circuits there is going to be two inputs. One should cause the value to be set, the other reset. For the D-type circuits, there is a single data input that when high will store a 1 gets stored and when low will store a 0. J-K might look like R-S, but the feedback loop is good evidence that it is J-K.
ii. **Latch or Flip-flop?:** A latch is pretty easy to detect since it has the behavior that when the "enable" is valid the input passed straight through to the output. Flip-flops, on the other hand, hold the input internally until a clock edge.
iii. **Master-slave?:** If the circuit is a flip-flop, then you can look to see if it has a master-slave design. There should clearly be two-stages if it is of this design.
iv. **Positive or negative triggered?:** There are two clues that you can use for this. First, if the clock gets inverted only before the slave stage of a master-slave flip-flop then that means the output will change on the negative edge. For non-master slave flip-flops, you can trace the clock signal through the circuit to see whether it is enabled on the positive or negative edge.

Now, looking at the diagrams we can examine each one-by-one:

a. This looks exactly like the standard Negative edge triggered R/S master-slave flip-flop (**#3** or **#5**).

b. This was a very tricky one. The circuit looks like we have the feedback like we get in a J/K flip-flop, but notice that the J/K inputs are tied together with an inverter. This effectively forces the inputs to the J/K flip-flop to be 0/1 or 1/0. We can see from the J-K truth table (Figure 6.23) that this means when the input is 0 the output goes to 0 and when the input is 1 the output goes to 1...just like a D flip-flop! So in fact this is a negative edge-triggered master-slave D flip-flop (**#6** or **#8**)

c. This was another very tricky one. We see the same sort of feedback that we get from a J/K flip-flop...however, the J/K inputs are tied together...this means that this is a J-K master-slave negative edge triggered flip-flip that can only have the inputs for J/K 0/0 or 1/1. Looking at the truth table for the J-K (Figure 6.23) we notice that 0/0 is Hold, which 1/1 is toggle. This is a special type of flip-flop called a T flip-flop can hold its value or flip its value only. Since T flip-flop wasn't a choice and in this case we implemented it with a J-K flip flop so I will accept that as the answer (**#9** or **#11**)

   Note: From b) and c) we can see why J-K flip-flops were a popular flip-flop. They can be used to build D flip flops as well as this new T (toggle) flip-flop...or they can just act sort of like a normal R-S flip flop!

d. We can tell that this is a clocked latch because of the initial stage taking what looks like an enable/clock signal. It is clearly not a flip-flop. Looking at the behavior of the two remaining inputs, one will set the latch and the other will reset it...hence this must be a clocked R-S latch (**#1**)

e. This one is tricky to see, but if you look at the circuit diagram it is analogous to the flip-flop in 6.24 except the NOR gates are NAND gates, this effectively inverts all the inputs and so this becomes a positive edge triggered D flip-flop. Note that this is not a master-slave flip-flop because there are not distinct master and slave stages (**#7**)

4. CLD2e, 6.19

It is best to think about timing by considering each component of the delay:

T_{su} = setup time = this is the time before the clock edge for which the input to the flip-flop (FF) must be stable in order to ensure that the FF correctly saves the value on its input.

T_{prop} = FF propagation delay = this is the time that it takes for the FF's output to change to reflect new input when the clock triggers.

T_h = hold time = the hold time is the time that the input needs to be stable after the clock edge to ensure that output actually gets the input value.

T_{combo} = combinational logic delay = the time it takes for a change to go through some combinational logic.

In general, T_{prop} and T_h overlap because they both start when the clock edge triggers the flip-flop. T_{prop} should always be greater than T_h so we don't have to worry about T_h. By waiting 13ns for T_{prop}, we are waiting more than the 5ns required by T_h.

i. Part a

Now from looking at circuit 6.19 we see that the output of the FF is feeding back into our combinational circuit, this means that we need the output of the FF to be stable (T_{prop} = 13ns) before we can wait for the logic delay (which in part a: T_{combo} = 0ns), and then the output of the logic must be stable for the length of the setup time (T_{su} = 20ns). Therefore the total maximum period is $T_{prop} + T_{combo} + T_{su} = 13 + 0 + 20 = 33$ns.

Frequency = 1 cycle / 33ns
= 0.0303 GHz
= 30.3 MHz

ii. Part b

For this part we have the same analysis as for part a, but T_{combo} is now 100ns (since we need to take the worst case scenario). This gives a clock period of $T_{prop} + T_{combo} + T_{su} = 13 + 100 + 20 = 133$ns.

Frequency = 1 cycle / 133ns
= 0.0075 GHz
= 7.5 MHz

Note: Why do we care about the hold time? For example, if the circuit had inputs that were changing based on a different/skewed clock or changing asynchronously, then those changes might end up violating the T_h.