Computer Systems, Spring 2019

Week 2 Lab

Numbers and Logic Gates

Solutions

Problems to work out on paper

1. 64 + 8 + 4 + 1 = 77
2. 0101 1001. Checking, this gives 64 + 16 + 8 + 1 = 89.
3. We are considering 1001 0110 two ways: as binary and as two's complement.
   (a) As binary: 128 + 16 + 4 + 2 = 150.
   (b) As two's complement: It's negative, so negate it: inverting gives 0110 1001; adding 1 gives 0110 1010, whose binary value is 64 + 32 + 8 + 2 = 106, so the original word has two's complement value of -106.
   (c) A representation, such as a number system, defines a meaning for each of the 2^k values that a k-bit word can have. These meanings are arbitrary, and they don't even have to represent numbers. To know what a word means, you need to know the bits in the word and also which representation is intended. This is analogous to ordinary words (i.e. strings of letters): to know what it means, you need to know both the letters and the language. The same sequence of letters could have one meaning if it's interpreted as English, and a different meaning if interpreted as French, and a third if German.
4. 0001 0111 + 0011 1011 = 0101 0010 which represents 64 + 16 + 2 = 82 in binary.
5. Recall that k-bit binary can represent x for $0 \leq x < 2^k$, and another way to write it is $0 \leq x \leq 2^k - 1$. Two's complement can represent x for $-2^{k-1} \leq x < 2^{k-1}$, and this can also be written as $-2^{k-1} \leq x \leq 2^{k-1} - 1$.
   (a) 4-bit binary can represent x where $0 \leq x \leq 15$, so 12 is ok.
   (b) 4-bit two's complement can represent x where $-8 \leq x \leq 7$ (an equivalent way to write it is $-8 \leq x < 8$). 12 isn't in this range, so the conversion is impossible.
   (c) 5-bit binary allows $0 \leq x \leq 31$ so it's impossible; 73 is too big.
   (d) Binary cannot represent negative numbers at all; this conversion is impossible.
   (e) 8-bit two's complement can represent x where $-128 \leq x \leq 127$, so this is ok.

(f) But 128 is too big, and the conversion is impossible. Notice that in k-bit two's complement, there are 2^{k-1} negative numbers; there is one representation of 0; and there are $2^{k-1}-1$ positive numbers. Thus the total number of numbers is $2^{k-1}+1+2^{k-1}-1 = 2\times 2^{k-1} = 2^k$ and thus all the 2^k distinct bit strings are being used. But the representation of 0 "uses up" one of the strings where the sign bit is 0—that is why the range of positive numbers is slightly smaller than the range of negatives.

6. Recall that we will ignore carry output from the most significant bit position.

   (a) Convert 95 and -30 to 8-bit two's complement numbers. 95 is represented in binary by 0101 1111 which is nonnegative, so that is also the two's complement representation. -30 is represented by 1110 0010 (convert 30 to binary and negate it).

   (b) Adding these two *two's complement* numbers using the binary addition algorithm, we get 0100 0001 (there is an overflow but ignore it).

   (c) Now, 0100 0001 is the two's complement representation of 65. That is the correct result of the original problem: $95 + (-30) = 65$.

   (d) You'll always get the right answer. You can do any addition of the form $x + y$, using the two's complement representations but doing binary addition, and the answer will be right. You can also do any subtraction of the form $x-y$ by using addition and negation: $x-y = x + (-y)$.

   (e) Hypothesis: The binary addition algorithm works not only for binary representations, but also for two's complement representations.

   (f) This hypothesis should be very surprising indeed! Here is an analogy based on words and natural languages. There is an algorithm to take a noun and make it plural: add an "s" at the end. But that algorithm works only for English—it gives completely wrong results if you try it on a German noun. Yet here we are doing *binary* addition on representations that are *not* binary, yet we *always get the right answer!*

   (g) The hypothesis is true. Actually, it is a theorem and can be proved mathematically.

   (h) This theorem has a major practical application: using it, we can make computers both faster and cheaper. Before the properties of two's complement were fully understood, computers needed separate circuits for working with binary numbers and signed integers, and they also needed separate adders and subtractors. Using this theorem, however, all four operations (bin+bin, bin-bin, tc+tc, tc-tc), as well as negation, can be performed by just one circuit: a binary adder. (There is such a thing a a subtraction circuit, but modern computers don't use it!)

7. A good way to remember the logic functions is that: inv means "not"; and2 means "all inputs are 1", or2 means "any input is 1", and xor2

means “an odd number of inputs are 1”. Also note that xor2 means “not equal”.

a	b	inv a	and2 a b	or2 a b	xor2 a b
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Problems using the computer

- switch-inv-probe — as you click the switch, you should see the output change. When the input is 0, the output should be 1; when the input is 1 the output should be 0.
- and2 — similar to the switch-inv-probe, but there will be two switches because there are two inputs. To simulate this circuit you need to try all possible input combinations: 00 01 10 11.
- HalfAdd — LogicWorks simulation. There are two inputs so the truth table has four lines. Compare your results with the truth table given in the lectures.
- FullAdd — LogicWorks simulation. Now there are three inputs, so there are $2^3 = 8$ lines in the truth table. Compare your results with the truth table given in the lectures. Notice that the size of the truth table grows exponentially in the number of inputs. It’s difficult to work with truth tables for circuits that have more than 4 or 5 inputs; Boolean algebra works better for larger circuits.
- mux1 — you need to draw the circuit in the canvas, and then simulate it to produce the truth table. The diagram should be reasonably neat and legible.