Lecture 4 In-Class Worksheet

1 Learning Objectives

This worksheet is based on Lumetta course notes section 1.3–1.4 and Patt & Patel textbook sections 2.5–2.6. After completing this lesson, you will know how to:

- [S04-1] Determine when an arithmetic operation overflows.
- [S04-2] Write a truth table for a Boolean formula.
- [S04-3] Express a Boolean function using a truth table.
- [S04-4] Show that a particular set of logic function or operators is complete.

2 Overflow

We say that an arithmetic operation *overflows* when the result cannot be represented using the intended data type. For example, if we are working with 4-bit unsigned integers, the the result of adding $4+13$ cannot be represented using 4-bit unsigned representation, because the result (17) is outside the range $[0,15]$ representable using 4-bit unsigned representation. Similarly, working with two's complement representation, the sum $-4+-8$ cannot be represented using 4-bit two's complement representation, because the result (-12) is outside the range $[-8,7]$ representable using 4-bit two's complement. Thus, to determine whether or not overflow will occur, you need to know both the number of bits used to represent the result, and whether or not these bits are intended to represent an unsigned integer or a signed integer using two's complement representation.

In many programming languages, including C, which we will study in this class, it is the programmer's responsibility to make sure that the numbers used in a calculation are within the representable range of the data types used to represent them. Whether or not adding or subtracting a particular set of bits results in overflow depends on the intended data type of the result. For example, adding `0111 + 0001` using a 4-bit add unit results in `1000`, which represents 8 in unsigned binary representation and -8 in two's complement representation. In the first case, the result is correct (7+1=8), in the second case, we say the result overflowed (and is not correct for the representation). It is important to note, however, that the result is always correct modulo 16 when working with a 4-bit add unit. In those cases when we intended to do arithmetic modulo 16, rather than using modular arithmetic as a stand-in for integer arithmetic, the result is *always* correct (no overflow can occur).

The arithmetic unit in most modern microprocessors indicates when the result is within the representable range for unsigned representation and when the result is within the representable range of two's complement representation.[1] Section 1.3.2 of the course notes

[1] The LC-3 computer we will study in this class does not do this.

describes precisely how to tell when overflow has occurred in unsigned addition, and is summarized below.

Unsigned overflow occurs if and only if

- Carry out is 1

Section 1.4.3 of the course notes explain how to tell when overflow will occurs for two's complement addition, and is summarized below.

Two's complement overflow occurs if and only if

- Both addends are negative but the result is non-negative, or
- Both addends are non-negative but the result is negative.

Q1. Determine whether the following operations result overflow in 4-bit unsigned representation and in 4-bit two's complement representation:

- `0010` + `1100`, No overflow
- `1100` + `1100`, Unsigned overflow
- `0110` + `0110`, Two's complement overflow
- `1000` + `1000`. Both forms of overflow

Working in binary (without no limit on the size of the result), adding $10_2 + 1100_2$ gives 1110_2, which fits in 4 bits, there is no carry. Therefore, no unsigned overflow occurs. Because in 4-bit two's complement representation, `0010` represents a positive number and `1100` represents a negative number, no overflow occurred.

Adding $1100_2 + 1100_2$ gives 11000_2, which requires 5 bits to represent correctly, resulting in overflow. We can also see that 4 of 11000_2 is the carry bit and is 1, which signals unsigned overflow. The 4-bit result of addition is `1000`. Because both addends are negative and the result is negative, no two's complement overflow has occurred. We can confirm this in decimal: `1100` is the 4-bit two's complement representation of -4. Two sum $-4 + -4 = -8$ is within the representable range of 4-bit two's complement.

Adding $110_2 + 110_2$ gives 1100_2, which does not produce a carry and is representable using 4-bit unsigned representation. Therefore, no unsigned overflow occurred. Two 4-bit result `1100` is a negative number in two's complement representation, while `0110` is positive. Adding two positive values should not result in a negative value; therefore, two's complement overflow has occurred.

Adding $1000_2 + 1000_2$ gives 10000_2, which generates a carry and is not representable using 4-bit unsigned representation. Unsigned overflow has occurred. The 4-bit two's complement values `1000` represent negative numbers, while the result `0000` is not negative. This means overflow has occurred.

3 Boolean Algebra

A *Boolean value* (or *logic value*) is the value 0 or 1. A *Boolean variable* (or *logic variable*) is a Boolean-valued variable, that is, it takes on the values 0 or 1. Boolean algebra is an algebra of Boolean variables and values. The main operations in Boolean algebra are **AND**, **OR**, and **NOT**, described in Section 1.4.2 of the course notes, and summarized below.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

NOT	
0	1
1	0

The **AND** operation uses multiplication notation: the operator is written as $\cdot$ or omitted, and has precedence over **OR**. The **OR** operation uses addition notation: the operator is written as $+$ and has lower precedence than **AND**. The **NOT** operation is written by drawing a line over the expression. For example, $\overline{AB}+C$ means evaluate **AND** of A and B, then evaluate **NOT** of the result, and finally evaluate **OR** of the result and C. To evaluate a Boolean formula for a given assignment of values to variables, we substitute the variables for their value in the formula, and evaluate using the rules above. For example, to evaluate $\overline{AB}+C$ when $A=0$, $B=1$, and $C=0$, we would replace the variables with the values and then proceed to perform the Boolean operations, respecting operator precedence (**AND** before **OR**):

$$\begin{aligned}\overline{0\cdot 1}+0 &= \overline{0}+0 && \text{AND}\\ &= 1+0 && \text{NOT}\\ &= 1 && \text{OR}\end{aligned}$$

Q2. Evaluate the Boolean formula $\bar{A}B+AC$ at $A=0$, $B=1$, $C=1$.

$$\overline{0}\cdot 0+0\cdot 1=1\cdot 1+0\cdot 1=1+0=1.$$

4 Truth Tables

Boolean algebra gives us one way to define Boolean functions (functions that take one or more Boolean values as input and produce a Boolean value as a result). Because each input to a Boolean function can have only two possible values, a function on n variables has only 2^n possible unique inputs. It is possible to completely describe the function by giving its value at all possible inputs. Such a listing is called

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

a *truth table*. For example, the truth table for the Boolean function $F(A,B,C) = \overline{AB} + C$ is given on the right. From the truth table, we can see that the function has the value 0 only when $A = 1$, $B = 1$, and $C = 0$.

Q3. Give the truth table for $G = A\bar{C} + BC$.

A	B	C	G
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

In many cases, a function is not given using a Boolean formula but in some other way, for example, as an English language description. If the description is well-written, we should be able to write down the truth table for the function.

Q4. Give the truth table for the following function of three variables S, A, and B. When $S = 0$, the function equals A, when $S = 1$, the function equals B.

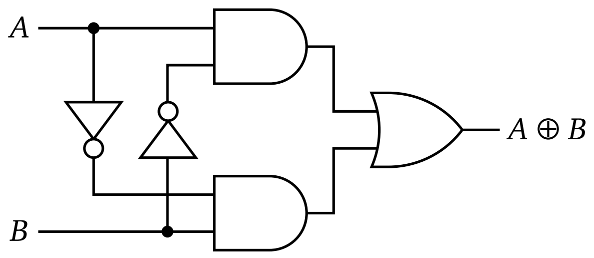
The truth table for this function, call it H, is given below.

S	A	B	H
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Note that the truth table above is the same as the truth table of function G above if we rename the variable S to C. That is, $H(S,A,B) = G(A,B,C)$, equivalently, $F(A,B,C) = H(C,B,A)$. This function describes a device called a *2-to-1 multiplexer*, something we will use extensively in this course.

[1]

5 Logical Completeness



In Section 1.4.4 of the course notes, you saw that the functions **AND**, **OR**, and **NOT** are *complete*, meaning that any Boolean function of one or more variables can be built from the functions **AND**, **OR**, and **NOT**. For example, the **XOR** operator (symbol ⊕) can be written using only **AND**, **OR**, and **NOT**: $A \oplus B = \bar{A}B + A\bar{B}$. This is shown schematically in the figure above.

We can use the fact that the operators **AND**, **OR**, and **NOT** together are logically complete to show that other combinations of operators are complete also. We do this by showing how we can implement **AND**, **OR**, and **NOT** using our new set of operators or functions. Because we can implement any Boolean function using **AND**, **OR**, and **NOT**, and we can implement those using our new set of operators, it follows that our new set of operators can implement any function.