**Introduction**

This assignment was about applying what we’ve learned about Boolean logic and turning it into virtual circuits using the Digital Works software. This report outlines how I completed each task and my understanding of the concepts involved. Each task in the assignment is given a section in this document, and the solutions to each task are in “Assignment.dwm”. Alongside the main assignment solution file, I have included the parts of each circuit in the “parts/” folder, as well as flowcharts in “flowcharts/”. If the main solutions file (Assignment.dwm) is too slow to load, I’ve also put each part into separate files: “Part 1.dwm”, “Part 2.dwm”, etc.

**Part 1 – AND, OR, NOT, NAND, and XOR gates**

This task was about familiarising myself with the fundamentals of what nearly all complex computer circuits are built on top of.

**Part 2 – Half-adder**

A half-adder is essentially a one-bit adder that returns the sum of the two one-bit numbers on one wire, with a carry on another wire. A carry is the output of the operation that could not be fitted into the sum bits (in this case, 1 bit). This could also be considered an overflow.

The truth table below shows the addition of two bits **A** and **B**:

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **Sum** | **Carry** |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

From this we can see that to add two bits together we will need two outputs, a sum and carry. We can also see the logic gates that we will need to get the sum and carry: an **Exclusive OR** gate (**XOR**) for the sum, and an **AND** gate for the carry. The truth tables for both of these gates match exactly to our **Sum** and **Carry** outputs, respectively. This means that no other logic gates will be needed for a half-adder.

My design for the half-adder is in the file named “Half-adder (Part 2).dwm”.

**Part 3 – Full-adder and 2s complement generator**

**Full-adder**

A full-adder takes in three bits, adds them together, and outputs the result in two bits (a sum and carry). The last bit put into a full-adder is a carry from another circuit operation (or 0 if no other operation was before it). This makes it different from a half-adder which only takes in two-bits, and also outputs two-bits.

The function of a full-adder is to first add the two main input bits like in the half-adder circuit. Then we need to add the sum of this to the carry input wire. Therefore, we will need two half-adders in a full-adder. We still have a carry to output, however. If a carry occurred in either of the half-adder circuits, we should output this from the full-adder as well. That means we need an **OR** gate, connected to the carry outputs from each half-adder.

The circuit for my full-adder can be found in “parts/Full-adder.dwm”.

**4-bit 2s Complement Generator**

To convert a 4 bit number into 2s complement, we invert all the bits and then add one to the result. To get the inverse of the bits in a binary number, we use **NOT** gates in order to change all 0s to 1s, and all 1s to 0s. We then need to add one. Since one fits exactly into one bit, we can use a half-adder to add a one to the least significant inverted bit of the 4-bit input and pass the carry out (if there is one) down a chain of other half-adders. This also means that this 2s complement generator could be extended to any word size by simply adding more half-adders to the chain.

The same process can also be achieved using a chain of full-adders. However, this requires unnecessary gates be used to achieve the same results as a half-adder chain.

The circuit for my 2s complement generator can be found in “parts/2s Complement Generator (Part 3b).dwm”.

**Part 4 – Arithmetic Unit for Addition and Subtraction**

The way 4-bit addition works is by chaining four full-adder components together. They are chained together by passing the carry out of one adder into the carry in of the subsequent one.

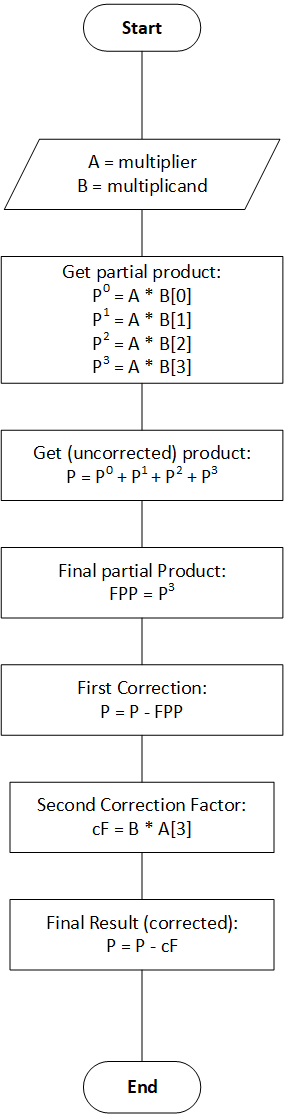
To add subtraction functionality, the second operand has to be converted into twos complement. For this process I made an 4 x 4 multiplexer, that takes in 4 bits as input and either leaves them as they are, or if the negate input is set it will convert the bits into 2s complement.

This multiplexer is wired up to the second operand of the subtractor, and to the addition/subtraction operand to allow the user to switch between addition or subtraction mode. Both operands, **A** and **B**, are then passed into the chain of four full-adders, which output the results, plus a carry. I also enabled an option to pass in a carry to the adder/subtractor unit, so that these can also be chained to other adders/subtractors.

**Part 5 – Multiplication and Division**

**Multiplication**

Multiplication is performed using the long division algorithm. This algorithm is what people learn in school to multiply large numbers together quickly and efficiently. It can also be applied to binary numbers.

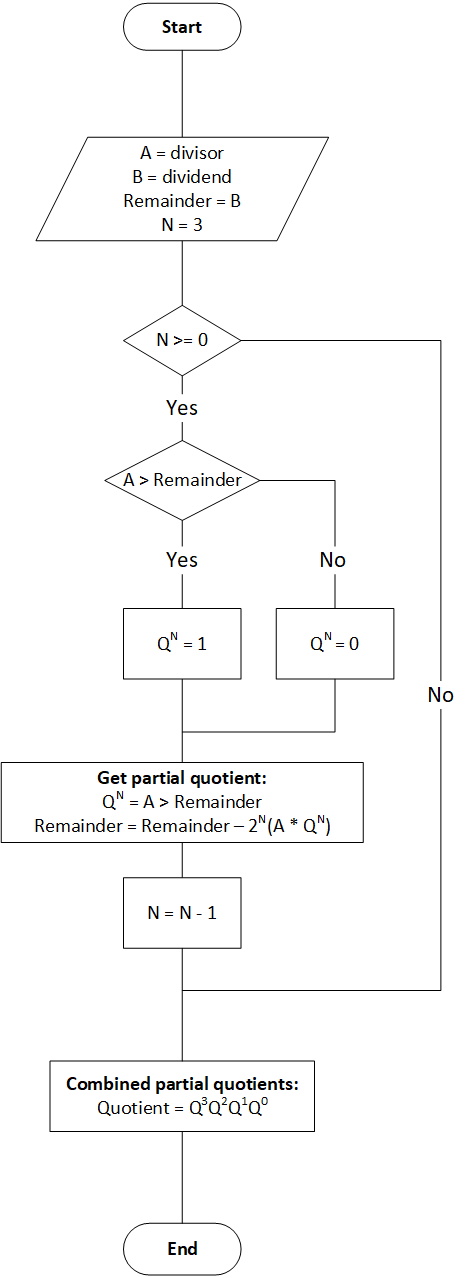


The process of multiplying binary numbers together is somewhat simpler than with decimal. First the partial products are computed using 1-bit binary multiplication, or a series of **AND** gates, which give either the multiplier or 0. Each partial product is the multiplier multiplied by the nth bit in the multiplicand. These are then added together, and if either or both of the numbers were negative (2s complement), this outputs an uncorrected result.

The reason for it being uncorrected is that we double the working length, otherwise we would lose data when multiplying (we’d only get the bottom half of the result of the calculation). When we double the working length, the highest power is now double what it was before, so we have to take the twos complement using the new most significant bit and take away the result (AB) from it.

For this circuit, I realised that I only needed to apply a correction factor to the four most significant bits of the result, as in each case the correction factor is just the 2s complement of the four least significant bits, and then scaled up by the new working length by multiplying it by 2n, where **n** is the number of bits in the new working length.

**Division**



This division circuit took me a fair amount of time to work out. I eventually found out that all I needed to was implement the long division algorithm that is mainly used with decimal numbers but works nicely with binary numbers as each partial quotient is either the divisor or 0. There are a lot of parallels between long division and long multiplication: in the first case we’re reducing the remainder by repeatedly subtracting the divisor from the dividend, while in the last case we’re repeatedly adding the multiplier onto the result.

Before any division happens, if **A** or **B** or both are negative, I convert them to positive 2s complement representation, then perform the division. For example, -5 / 3 becomes 5 / 3, or 5 / -3 becomes 5 / 3, or -5 / -3 becomes 5 / 3. This is so that the partial products are calculated correctly. After the division is completed I convert the result to negative if either of the operands were exclusively negative (**XOR**).

To get each partial quotient we check if the divisor fits into the remainder from the previous operations. If it does, then the partial quotient is 1 (**QN**), otherwise it is 0. When all of the partial quotients have been calculated, they are combined to form the final quotient, like so: **Q3Q2Q1Q0**. The other output is the remainder, which occurs when the answer is not an integer (7 / 3 = 2.5, for example).

Although the flowchart shows that the process is done inside a loop, in the divider circuit the partial quotient circuits are chained together. Despite this, it would also be possible to use a loop, and given more time I would implement a counter and buffer to allow each partial quotient to be calculated on each clock oscillation.

Another thing to note is the N = 3 in the flowchart. If N was higher, the precision of the remainder would be increased, at the expense of computational time.

Finally, there is one problem with the divider in that it can’t divide -8 / -8, because when -8 is converted to positive 2s complement representation, the word length of 4 is too small. One solution to this would be to check if both operands are equal to -8 and make the result 1. This solution does not seem very neat to me, however.

**Summary**

I had a lot of fun working on this assignment, despite it taking a long time to implement, especially the multiplication and division. I learned a lot about logic, circuits, arithmetic, algorithms, and other fundamental computing concepts during this assignment, however.