ALU Simulator

1. Introduction

This mini project aims to investigate how arithmetic is implemented on hardware at the logic gate level. Specifically, the current version studies three different number types, i.e., signed int32, unsigned int32 and float32, on four different operations, i.e., addition (add), subtraction (sub), multiplication (mul) and division (div).

To achieve this goal, I write C code to simulate how the logic circuits manipulate the bits to realize the operations mentioned above, then test if the simulation results are consistent with the results computed in C. As the goal is to simulate at logic gate level, in most cases only bitwise operations are used in the simulation code.

The remaining sections are organized as follows. Section 2 investigates add and sub for integer, while section 3 looks into mul and div for integer. Representation and arithmetic of floating point number are discussed in Section 4. Section 5 highlights some aspects which are very important for ALU design but not covered in this mini project. I introduce the code structure in Section 6 and conclude in Section 7.

2. Integer Addition and Subtraction

In this section, the implementation order is first signed int add, then signed int sub, finally unsigned int add / sub.

For addition, the basic components are half adder, which takes two input bits and output one sum bit and one carry out bit, and full adder, which takes two input bits, one carry in bit, and output one sum bit and one carry out bit. Then we can concatenate the adder for each bit to compute the sum of two signed int32 number.

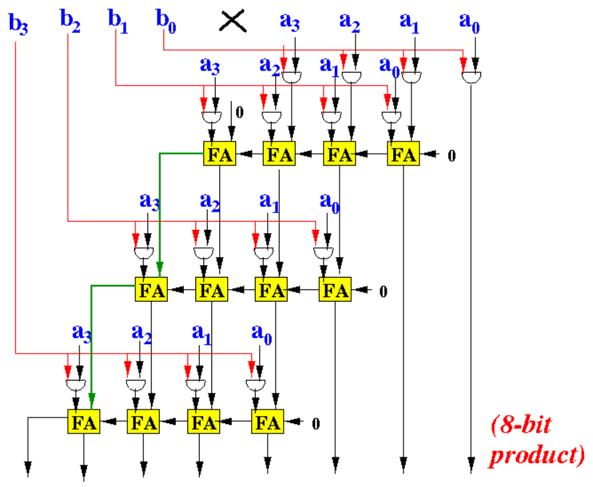
For subtraction, with 2-complement representation, we have that $A - B = A + (-B) = A + (\sim B) + 1$, where $\sim B$ is the bitwise not of B. Thus, we can implement subtraction by first passing B through bitwise not gate, then computing its sum with A using the addition circuit that has been implemented (a trick here is to set the least significant bit carry in signal to 1).

For unsigned int32, the problem here is that we don't have the 2-complement form for an unsigned int, thus can not directly follow the trick for signed int32 as before. Intuitively, we can transform an unsigned int32 number to a signed "int33" number by inserting a virtual "0" bit at its most significant bit, then we can compute their sum and subtraction with the same circuit designed for signed int32. Although we introduce an additional virtual bit, it will not influence the computation process, as the final result is only dependent on the least significant 32 bits in the output.

3. Integer Multiplication and Division

For mul and div, it's easier to start with unsigned int as we don't need to consider the effect of sign for it.

To compute $C = A \times B$, we have that $C = \sum_{i=0}^{31} B_i \cdot (A \ll i)$, where B_i is the i^{th} least significant bit, $\ll$ is left shift. Left shift is a basic component in circuits, and the product term is also very easy to compute as bitwise and. The main computation bottleneck is the summation term here. I think an important issue here is the tradeoff between circuit complexity and time complexity. For example, we can design a specified multiplier which uses $N(N-1)$ full adders for fast computation as shown in the figure below [5]. We can also design it in a more "software" way as a sequence of summation, which only needs N adders but may require more clock cycles for computation.



To compute $C = A/B$, we have that $A = \sum_{i=0}^{31} (C_i \ll i) \cdot B$. I implement it with the long division algorithm [7] for simulation.

For signed int32 multiplication, a natural idea is to consider the sign and value independently, i.e., $A \times B = \big(sign(A) \cdot sign(B)\big) \cdot (|A| \cdot |B|)$. However, this method introduces some additional cost to compute the negative values. Another way is to see the multiplication rule as $C = -B_{31} \cdot (A \ll 31) + \sum_{i=0}^{30} B_i \cdot (A \ll i)$. The additional cost here is that we need to extend the MSB of all the partial products to make them aligned with the MSB of the summation. The Baugh-Wooley algorithm [4] uses a cunning trick to reduce the cost to the same level as unsigned int multiplication.

For signed int32 division, I follow the idea of $\frac{A}{B} = XOR\big(sign(A), sign(B)\big) \cdot \frac{|A|}{|B|}$, where $\frac{|A|}{|B|}$ has been implemented for unsigned int. The additional cost here is that we need to include some negative operations.

4. Floating-point Arithmetic

4.1 IEEE 754 Format

IEEE 754 is the standard format of floating-point number in modern computers. It consists of 1 sign bit, 8 biased exponent bits, and 23 normalized fractions (significand) bits. The exponent bits are biased so that we can use 00000000 and 11111111 to represent some special cases (00000000 for 0.0, and 11111111 for infinity). The normalized fractions are in the form of 1.xxxx, where we can always omit the leading 1 and only store the remaining bits.

4.2 FP Multiplication and Division

As the sign and absolute value of FP number are represented separately, it makes multiplication and division easier to compute than addition and subtraction. Intuitively, the output sign is the XOR of the two inputs' sign, the output fractions is the product / quotient of the two inputs' fractions, and the output exponent is the sum / diff of the two inputs' exponents. However, there are still many issues we need to tackle here, which makes FP computation much more complicated.

First, as the exponents of the two inputs are both biased, we need to additionally subtract / add the biased term for multiplication / division respectively.

Second, the position of the most significant significand may change after multiplication / division. For example, if we multiply 1.1 with 1.1, the result will be 10.01, and we need to right shift it by 1 bit to fit into the standard format of 1.001, and change the result exponent correspondingly. For multiplication, we only need to check if right shift by 1 bit is required. However, for division, we need to left shift until we encounter the first bit 1 in the result fractions.

Third, we need to consider rounding in FP computation. Consider the same example of multiplying 1.1 with 1.1, if the fractions only have 2 bits, then we need to discard the least significant bit and check if we need to add an additional 1 to the truncated fractions according to the value of the discarded part.

4.3 FP Addition and Subtraction

For FP add and sub, the main idea is to first right shift the fractions of the operand with smaller exponent to make the two operands aligned. The result's fractions equal to the sum / diff of the two operands' (shifted) fractions. The result's exponent equals to the larger exponent of the two operands. However, the situation becomes more complicated when the sign is taken into consideration.

So we first start with the simplest case of adding two positive FP numbers. It just follows the procedure as mentioned in the last paragraph. And we also need to consider rounding and position change of MSB here. First, we will discard some bits when right shifting the fractions of one operand, thus we need to check if an additional carry in bit is needed here for rounding. Second, we may produce a carry out bit when computing the fractions' sum, thus need to check if right shift is required after addition. Sum of two negative number follow the same pipeline except that the result's sign bit is 1.

When we need to sum one positive and one negative FP number, we can use the same trick of adding an additional bit at the MSB of the fractions to transform it to two's complement format. If the number is positive, we add a 0 here. If the number is negative, we add a 1 here, invert the remaining bits, and add 1. Then we can treat the sum of the fractions as 24-bit integer add, which has been implemented in integer part. If the MSB of the result fractions is 1, it means that the result is negative and we need to transform it back to the IEEE 754 format.

Finally, to compute $A - B$, we can simply invert the sign bit of B to get $-B$, and then compute $A + (-B)$ using the same method as above.

4.4 Remaining Issues

Some test cases for FP fail due to the two reasons below.

First, computations involving 0 all fail because 0 requires some special encoding in IEEE 754, and the corresponding logics are not implemented in the current version.

Second, some tests fail due to a difference in the LSB, which is very likely caused by some difference in the detailed logics of the rounding process, which remains to be investigated in more details.

5. Limitations

First, this project mainly focuses on understanding the principle of how arithmetic is implemented on hardware, but does not cover many other important designs in ALU such as the control circuit and flag signals. Control circuits are important to reuse several basic modules for different computations, which can make the ALU design more compact. The flag signals [2] can generally be acquired with simple logic circuits, and are crucial for many downstreaming tasks.

Second, there are many more hardware-level optimization which are crucial to improve the efficiency of ALU, which is also not covered here. For example, my current simulation implements the adder as a cascade, which means that the i^{th} full adder always needs to wait for the carry out signal from its previous unit to perform its computation. There are many advanced techniques like carry look ahead [3] to reduce the latency caused by this data dependence.

6. Code Structure

"utils" defines some useful functions, like the transition between int and bit array, half adder and full adder.

"int_operation" defines the operations on signed int32, and "int_test" implements the test functions for these operations. There are also corresponding code files for unsigned int32 and float32.

"main" call the above test functions to check correctness.

References

1. https://www.coursera.org/learn/build-a-computer
2. http://www.csc.villanova.edu/~mdamian/Past/csc2400fa13/assign/ALU.html
3. https://www.electronicshub.org/carry-look-ahead-adder/
4. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c8/c8s2/c8s2v2/
5. http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/1-circuits/multiply.html
6. http://people.ee.duke.edu/~sorin/prior-courses/ece152-spring2009/lectures/
7. https://en.wikipedia.org/wiki/Division_algorithm