Snow64 BFloat16 Details

- General Notes
  - A *BFloat16* is simply the high 16-bit half of an IEEE binary32 float.
  - Note that this is different from the IEEE binary16 float format!
  - For Snow64's BFloat16 implementation, there are no NaN or infinities, and zero is the only subnormal. If the exponent of a BFloat16 is encoded as 0x00 or as 0xff, then the floating point number is considered to be zero.
  - Signed zero still exists.
  - The rounding mode used is "toward zero".
  - The implementation was tuned such that, with these conditions in mind, results will be identical to use of IEEE binary32 floats if the low 16 bits of the mantissa are treated as always zero.
- Implemented BFloat16 operations
  - Add and Subtract (3 + 1 cycles)
  - Multiply (2 + 1 cycles)
    - Of note is that this is the fastest of the traditional add, sub, mul, and div operations. This is perhaps due to
  - Divide
    - The division implementation is of particular interest because of how the underlying `uint16_t` by `uint8_t` division was implemented.
      - Note that this underlying `uint16_t` by `uint8_t` is used for dividing the fractions of the operands.
      - To use an integer divider for the implementation, it is necessary to understand that the significands are effectively fixed-point numbers, in this case of the format with one bit for the whole number part, and seven bits for the fractional part.
      - This means that a fixed-point divide must be performed instead of an integer divide, which also means that we cannot simply perform a `uint8_t` by `uint8_t` integer divide of the significands of the operands. We instead need a larger integer divider. This is why a `uint16_t` by `uint8_t` divider is used.
    - In a nutshell, hexadecimal (radix 16) long division was implemented, permitting computing 4 bits of quotient per cycle.
    - A basic way to speed up long division is to compute all of the necessary multiplications in parallel before performing the divide. In the case of hexadecimal (radix 16) long division, the multiplications to be performed are denominator * 0, denominator * 1, denominator * 2, denominator * 3, denominator * 4, ..., denominator * 15. The multiplication results are stored for fast access.
      - These multiplications are effectively 8-bit by 4-bit, with a 12-bit result, and because the 4-bit numbers are constants, they can simply be computed by use of shifts and adds. The FPGA synthesis tools almost certainly implement them this way, as it appears that none of my FPGA's more generic multipliers were used to implement these multiplications.
    - For this to fast enough, there needs to be a way to very quickly search through the multiplication results. My implementation uses binary search for that purpose. The binary search can be implemented with four greater than (or less than) compares (effectively implemented as subtractions) and four multiplexers. The number of compares and multiplexers for the binary search is equal to the number of bits computed per iteration of the divide, which is why in this case there are 4 of each.
  - Compare less than (1 + 1 cycles)
  - Cast Integer types (`uint8_t` , `int8_t` , `uint16_t` , `int16_t` , `uint32_t` , `int32_t` , `uint64_t` , and `int64_t`) to BFloat16.
  - Cast BFloat16 to Integer types (`uint8_t` , `int8_t` , `uint16_t` , `int16_t` , `uint32_t` , `int32_t` ,

`uint64_t`, and `int64_t`).

- Method of Testing
  - A software implementation of this form of BFloat16 was developed first.
    - The software implementation was written in C++ and was almost entirely exhaustively tested against my x86-64 laptop's IEEE binary32 format.
    - The one exception to this was (a) casting `int64_t` to BFloat16 and (b) casting `uint64_t` to BFloat16. These took too long to exhaustively test. Given more time, I would have instead attempted to formally verify the BFloat16 software implementation, but because of C++'s modern template metaprogramming (especially the standard library's `type_traits`), I believe that exhaustively testing the smaller integer types works fine.
  - The hardware implementation was compared to the software by using Verilator to convert the hardware implementation to cycle-accurate C++ emulation.
    - In most cases, this was done so as
    - The one exception to this was (a) casting `int64_t` to BFloat16 and (b) casting `uint64_t` to BFloat16. These were instead tested via large amounts of random testing, and due to relative simplicity of the hardware implementation, were deemed to be working.