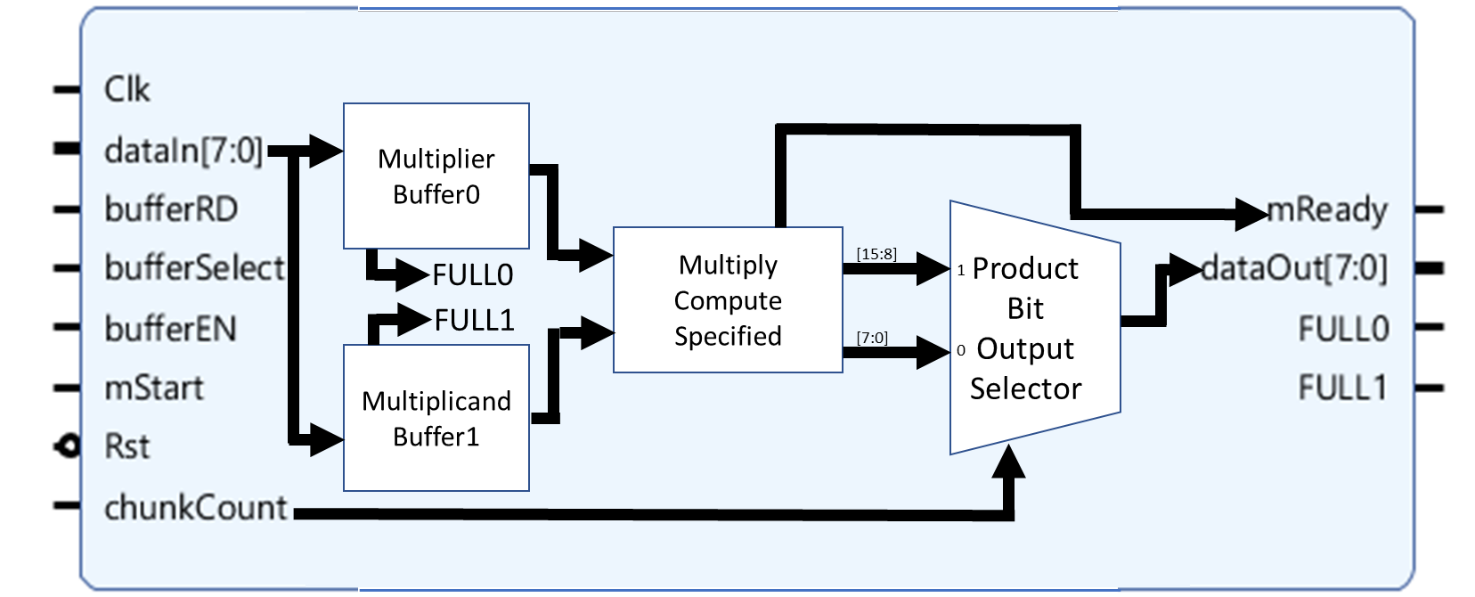
Multiplier Benchmark

# Overview

Internal Block Diagram:



* The only portion of the multiplier that changes between data type is the “Multiply Compute” block.
* The block can be adjusted to n-bit data input. This benchmark will demonstrate n-bit inputs of n = 8-16-32.
* To ensure all signals are zeroed and set properly, there will be a 1 clock period reset delay at the beginning of each simulation. The period after this delay is when cycles to complete will begin to be counted, up until the last period necessary for all needed values to be extracted.
* Input vectors used will be the same bit patterns, except for fixed point, but due to notation structure difference, they will represent different numbers.
* Error is not a focus of the simulation but is monitored. Calculated with the following formula. Error is not a focus of the simulation but is monitored. Calculated with the following formula.

# Data Structure and Example Data

## Integer Representation

### Data Structure

Simple base conversion from decimal to binary.

### Example Data

22 = (1\*16) + (0\*8) + (1\*4) + (1\*2) + (0\*1) = (1\*24) + (0\*23) + (1\*22) + (1\*21) + (1\*20) = 10110BINARY

## Floating Point Representation

### Data Structure:

|  |  |  |  |
| --- | --- | --- | --- |
| *N-Bit Count* | *Sign Bits* | *Bias Bits*  *(Bias Value)* | *Mantissa Bits* |
| 8 | 1 | 4  (-7) | 3 |
| 16 | 1 | 5  (-15) | 10 |
| 32 | 1 | 8  (-127) | 23 |

### Example Data:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *N-Bit Count* | *Decimal Value* | *Bit Representation* | | | *Expanded Form* |
| *Sign* | *Bias* | *Mantissa* |
| 8-bit | +48.0 | 0 | 1100 | 100 | +212-7 \* (1 + ) |
| 16-bit | -48.0 | 1 | 10100 | 10000000000 | -220-15 \* (1 + ) |
| 32-bit | +48.0 | 0 | 10000100 | 10000000000000000000000 | +2132-127 \* (1 + ) |

## Fixed Point Representation

### Data Structure:

|  |  |  |
| --- | --- | --- |
| *N-Bit Count* | *Integer Bits* | *Fractional Bits* |
| 8 | 4 | 4 |
| 16 | 8 | 8 |
| 32 | 16 | 16 |

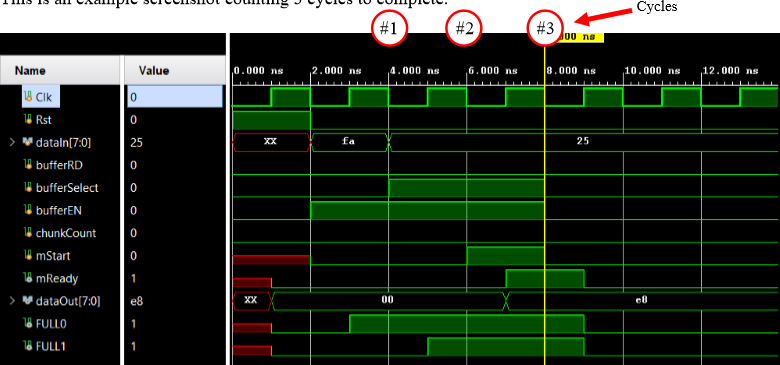
### Example Data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N-Bit Count* | *Decimal Value* | *Bit Representation* | | *Expanded Form* |
| *Integer* | *Fractional* |
| 8-bit | 12.875 | 1100 | 1110 | 23 + 22 + 2-1 + 2-2 + 2-3 |
| 16-bit | 100.00390625 | 1100100 | 00000001 | 26 + 25 + 22 + 2-8 |
| 32-bit | 4100.250244 | 1000000000100 | 0010000000001 | 212 + 22 + -2 + 2-12 |

# Benchmark Simulation Generation and Analysis Steps

The exact same code structure is used for all three simulation objects. These are the steps to setup, generate, and demonstrate the analysis of the data for the following tables.

1. Set input parameters by adjusting the values in “definitions.h”. This is a breakdown of input parameters with corresponding purpose.
   1. **inputWidth:** Defines the width of the input vector. i.e. N-Bit Count.
   2. **myTimeScale:** Defines the unit time and precision for simulation.
   3. **clkPerid:** Defines the period of clock used for simulation
   4. **dataIn1:** Defines the input multiplier value. Note that this is dependent on the width of the input vector.
   5. **dataIn2:** Defines the input multiplicand value. Note that this is dependent on the width of the input vector.
   6. **multiplyIndex:** Defines the width of the product for integer multiplication.
   7. **mantissaIndex:** Defines the highest index of the mantissa for floating point multiplication. Simply, this should be (Mantissa Bits – 1).
   8. **expBias:** Defines the bias to remove from added exponents for floating point multiplication.
   9. **expWide:** Define the width of the exponent bias.
   10. **fracBitCount:** Defines the number of bits counted as fractional for fixed point multiplication.
2. Click the “Run Implementation” option. The pop-up will ask to complete synthesis first, click OK. Once this completes, press OK to open the implemented design.
3. Click “Run Simulation, then select “Run Post-Implementation Timing Simulation”
4. As stated before, the first clock period will not be counted when analyzing simulation. This is an example screenshot counting 3 cycles to complete.



# Integer Multiplier

* Design Source File Name: “integercomputeBlock.v”
* Test Bench File Name: “computeBlock\_tb.v”

### Simulation Data:

*(Input)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N-Bit Count* | *Multiplier*  *Hex Representation* | *Multiplier*  *Decimal Notation* | *Multiplicand*  *Hex Representation* | *Multiplicand*  *Decimal Representation* |
| 8 | 0xfa | 250 | 0x25 | 37 |
| 16 | 0xfafa | 64250 | 0x25ff | 9727 |
| 32 | 0xfafafafa | 4210752250 | 0xa925ff | 11085311 |

*(Output)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *N-Bit Count* | *Calculated Output*  *Hex Representation* | *Calculated Output*  *Decimal Representation* | *Cycles to Complete* | *Expected Output*  *Decimal Representation* | *Error*  *(%)* |
| 8 | 0x2422 | 9250 | 4 | 9250 | 0.00 |
| 16 | 0x25402106 | 624959750 | 4 | 624959750 | 0.00 |
| 32 | 0x00a5d4eff5502106 | 4.66774e16 | 4 | 4.66774e16 | 0.00 |

# Floating Point Multiplier, Custom Multiply Operator

* Design Source File Name: “floatcomputeBlock.v”
* Test Bench File Name: “floatcomputeBlock\_tb.v”

### Simulation Data:

*(Input)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N-Bit Count* | *Multiplier*  *Hex Representation* | *Multiplier*  *Decimal Notation* | *Multiplicand*  *Hex Representation* | *Multiplicand*  *Decimal Representation* |
| 8 | 0xfa | -320 | 0x25 | +0.203125 |
| 16 | 0xfafa | -5.715e+4 | 0x25ff | 2.342e-2 |
| 32 | 0xfafafafa | -6.51582312038e+35 | 0x00a925ff | +1.55338292809e-38 |

*(Output)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *N-Bit Count* | *Calculated Output*  *Hex Representation* | *Calculated Output*  *Decimal Representation* | *Cycles to Complete* | *Expected Output*  *Decimal Representation* | *Error*  *(%)* |
| 8 | 0xe8 | -64 | 3 | -65 | 1.54 |
| 16 | 0xe66f | -1647 | 3 | -1339 | 18.70 |
| 32 | 0xbc4ba9d4 | -1.2430627e-2 | 3 | -1.012156839e-2 | 22.81 |

# Floating Point Multiplier, Built-In Multiply Operator

* Design Source File Name: “floatcomputeBlockPynq.v”
* Test Bench File Name: “floatComputeBlockPynq\_tb.v”

### Simulation Data:

*(Input)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N-Bit Count* | *Multiplier*  *Hex Representation* | *Multiplier*  *Decimal Notation* | *Multiplicand*  *Hex Representation* | *Multiplicand*  *Decimal Representation* |
| 8 | 0xfa | -320 | 0x25 | +0.203125 |
| 16 | 0xfafa | -5.715e+4 | 0x25ff | 2.342e-2 |
| 32 | 0xfafafafa | -6.51582312038e+35 | 0x00a925ff | +1.55338292809e-38 |

*(Output)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *N-Bit Count* | *Calculated Output*  *Hex Representation* | *Calculated Output*  *Decimal Representation* | *Cycles to Complete* | *Expected Output*  *Decimal Representation* | *Error*  *(%)* |
| 8 | 0xe8 | -64 | 3 | -65 | 1.54 |
| 16 | 0xe53a | -1338 | 3 | -1339 | 0.00 |
| 32 | 0xbc25d4ef | -1.01215681061e-2 | 3 | -1.012156839e-2 | 0.00 |

# Fixed Point Multiplier

* Design Source File Name: “fixedcomputeBlock.v”
* Test Bench File Name: “fixedcomputeBlock\_tb.v”

### Simulation Data:

(Overflow occurred with the previous bit values, so they were substituted.)

*(Input)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *N-Bit Count* | *Multiplier*  *Hex Representation* | *Multiplier*  *Decimal Notation* | *Multiplicand*  *Hex Representation* | *Multiplicand*  *Decimal Representation* |
| 8 | 0x29 | 2.5625 | 0x44 | 4.25 |
| 16 | 0x2929 | 41.16015625 | 0x051f | 5.12109375 |
| 32 | 0x027d1100 | 637.06640625 | 0x003d1100 | 61.06640625 |

*(Output)*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *N-Bit Count* | *Calculated Output*  *Hex Representation* | *Calculated Output*  *Decimal Representation* | *Cycles to Complete* | *Expected Output*  *Decimal Representation* | *Error*  *(%)* |
| 8 | 0xae | 10.875 | 3 | 10.890625 | 0.14 |
| 16 | 0xd2c8 | 210.78125 | 3 | 210.7850189 | 0.00 |
| 32 | 0x97c9 | 38903.78515625 | 3 | 68903.35597 | 0.00 |

# Notes

* The built in multiplier and my custom implementation perform very different. While my implementation minimizes buffer usage, proves inaccurate at higher bit counts. Both implementations were measured to use 3 cycles to complete. The first two cycles are loading the input data to buffers, then the third cycle is utilized for performing the computation.
* Previous testing methods were incorrectly adding a cycle to each measurement.
* Need to implement these multipliers into a crossbar structure.