MyHDL Fixed-Point Object

Christopher L. Felton
cfelton@ieee.org

1 Introduction

This document outlines the design of a fixed-point object for MyHDL. Fixed-point calculations are common in many different applications and are commonly used in DSP hardware implementations. Python and MyHDL provide the frameworks to develop a fixed-point object that can be used for design, analysis, and conversion. Conversion in this context implies taking the MyHDL design and converting it to synthesizable Verilog or VHDL.

MyHDL has an *intbv* object which is used for integer representation. The *intbv* also can be used for standard bit-vectors but here we are interested in the constraint integer usage. The *intbv* is a powerful tool in HDL design see [1] and [2] for more information on the *intbv* object. The *intbv* will be used as the base class for the fixed point object, *fxintbv*.

Fixed-point is used to represent fractional numbers of finite-word length. Fixed-point is an interpretation of a 2's compliment number usually signed but not limited to sign representation. It extends our finite-word length from a finite set of integers to a finite set of rational real numbers [3]. A fixed-point representation of a number consists of integer and fractional components. The overall bit length is defined as $X_{Nbits} = X_{IntegerNbits} + X_{FractionNbits} + 1$.

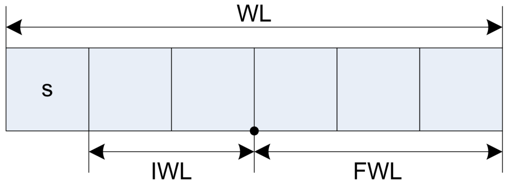

Figure 1: Fixed-Point Bit Vector Layout

Here **IWL** is the integer word length, **FWL** is the fraction word length, and **WL** is the overal word length. $WL = IWL + FWL + 1$. The notation used in this document, $Q0.15$ [8] often called Q-format, is used to represent a signed fixed point fractional number. The first number (left of the decimal point) indicates the number of integer bits (guard bits, **IWL**) and the second number (right of the decimal point, **FWL**) indicates the number of fractional bits. Although the fixed-point object will use the range and resolution to define a fixed-point number the Q-format is often used and will be used here as well. For a more formal introduction to fixed-point representation see [3], [7], and [8].

Now that fixed-point has been breifly explain, why is it useful? Many are familar with floating-point and fixed-point processors. Fixed-point processors are the processors that lack the hardware for floating-point calculations. Fixed-point processors can only do integer arithmetic. These types of processors are popular because the floating-point units can require a significant amount of hardware which in turn affects power consumption, maximum clock frequency, complexity, etc.

As mentioned floating-point units (hardware modules) can be expensive, the example used was for processors, where there is only one hardware module for the integer arithmetic and floating-point arithmetic. In a digital hardware signal processing design there can be many explict adders, subtractors, multipliers, etc. In these cases floating-point arithmetic is rarely used and fixed-point implementations dominate.

Table 1 is an example of fixed-point representations.

Binary	Hex	Integer	Floating Point Fraction	Fixed-Point Fraction	Actual
0100000000000000	4000	16384	0.50000000	0.50000000	1/2
0010000000000000	2000	8192	0.25000000	0.25000000	1/4
0001000000000000	1000	4096	0.12500000	0.12500000	1/8
0000100000000000	0800	2048	0.06250000	0.06250000	1/16
0000010000000000	0400	1024	0.03125000	0.03125000	1/32
0000001000000000	0200	512	0.01562500	0.01562500	1/64
0000000100000000	0100	256	0.00781250	0.00781250	1/128
0000000010000000	0080	128	0.00390625	0.00390625	1/256
0000000001000000	0040	64	0.00195312	0.00195312	1/512
0000000000100000	0020	32	0.00097656	0.00097656	1/1024
0000000000010000	0010	16	0.00048828	0.00048828	1/2048
0000000000001000	0008	8	0.00024414	0.00024414	1/4096
0000000000000100	0004	4	0.00012207	0.00012207	1/8192
0000000000000010	0002	2	0.00006104	0.00006104	1/16384
0000000000000001	0001	1	0.00003052	0.00003052	1/32768
0010101010101011	2AAB	10923	0.33333000	0.33334351	0.33333
0101101001111111	5A7F	23167	0.70700000	0.70700073	0.707
0000000000001010	000A	10	0.0003141592	0.00030518	0.0003141592
0000000000000011	0003	3	0.000086476908	0.00009155	0.000086476908

Table 1: Fixed-Point Examples

Another illustration of the fixed-point bit-vector.

```
  b7    b6    b5    b4    b3     b2    b1    b0
   S     F     F     F     F      F     F     F
   0     1     0     1     1      0     0     1
 +-1    1/2   1/4   1/8   1/16   1/32  1/64  1/128

Value = + 1/2 + 1/8 + 1/16 + 1/128 which equals
      = + 0.5 + 0.125 + 0.0625 + 0.0078125
      = 0.6953125
```

1.1 Examples

The following are fixed-point examples for multiplication and addition. Fixed-point subtraction can be calculated the similar to a 2's complement subrtraction. The difference being the same as fixed-point addition. The points need to be aligned before the addition / subtraction. Multiplication is the same as 2's complement multiplication with additional "point" arthimetic. The "point" bookkeeping is usually calculated at design time and not run time.

Multiplication Fixed-point multiplication same as 2's compliment multiplication. This example illustrates $6.5625(Q3.4) * 4.25(Q5.2)$.

```
0110.1001  == 6.5625
000100.01  == 4.25

          01101001
        x 00010001
      ------------
          01101001
         00000000
        00000000
       00000000
      01101001
     00000000
    00000000
   00000000
 --------------------
  x000011011111001   ==  0000011011.111001  ==  27.890625
```

Addition Addition is a little more complicated because the points have to be aligned. Using the same numbers from the multiplication problem.

```
0110.1001  == 6.5625
000100.01  == 4.25

            0110.1001
        + 000100.01
      ------------
          001010.1101  ==  10.8125
```

A basic introduction to fixed-point has been provided with references to more complete coverage of the topic. The rest of this document will outline the design and examples of the fixed-point object.

1.2 Design Goals for a Fixed-Point Object

1. MyHDL intbv base class
2. Fixed-Point support
   (a) Floating-Point conversion
   (b) Rounding and Truncating
   (c) Bound checking
   (d) Display methods
   (e) Mathematical operations handled by *intbv*
   (f) "Point" alignment checking
3. Convertible
4. Follow Python and MyHDL design goals.

2 Fixed-Point Object

2.1 Constructor

- fxintbv(value, min=None, max=None, res=None, Q=(None,None), roundMode="round-even")
  - **value** : This is the initial value of the fixed-point number. This can be a float, int, or long.
  - **min** : This is the minimum value of the fixed-point number.
  - **max** : This is the maximum value of the fixed-point number. This is in the same form as all other python upper bounds. The actual max number is $max - res$.
  - **res** : This is the resolution of the fixed-point number. The resolution is the smallest non-zero magnitude [3].
  - **Q** : A tuple that defines the Q-format. This is an alternative to the min, max, res definition.
  - **roundMode** : The rounding method used, truncate, round, round-even.

Examples

```
>>> x = fxintbv(0, min=-1, max=1, res=2**-15)
>>> print x, hex(x), repr(x)
 0.000000e+00 0x0 <0 (0.000000) Q0.15>

>>> x = fxintbv(2.5, min=-8, max=8, res=1/32.)
>>> print x, hex(x), repr(x)
 2.500000e+00 0x50 <80 (2.500000) Q3.5>

>>> x = fxintbv(0.0333, min=-1, max=1, res=0.0001)
>>> print x, hex(x), repr(x)
 3.332520e-02 0x222 <546 (0.033325) Q0.14>
```

2.2 Base Object: myhdl.intbv

The MyHDL *intbv* object is the base class for the fixed-point object. Because fixed-point is basically an interpretation of a constrained integer it is natural and straight forward to extend the *intbv* object for a fixed-point representation. See [1] for more information on *intbv*.

```
class fxintbv(intbv):
```

2.3 Public Functions

Non Convertible Functions These functions cannot be used in the generators. These can only be used in the elaboration portion of the design.

- **Round(...)** : Returns the floating-point number as an integer.

- **Bit(...)** : Returns the binary representation as a string of 1's and 0's.
- **Range(...)** : Display the range of the fixed-point number
- **Resolution(...)** : Display the resolution of the fixed-point number
- **NewRep(...)** : This function will change the representation, Qa.b to Qan.bn. This function will allow the modification of the number of bits allocated for the integer and fraction. Returns a new fxintbv object.
- **ProductRep(...)** : Returns a new fxintbv that is the correct representation for a * b
- **AdditionRep(...)** : Returns a new fxintbv that is the correct representation for a + b

2.4 Properties

- **min** : Minimum value
- **max** : Maximum value
- **res** : Resolution of the fixed-point number.
- **value** : Integer value (non-fixed-point)
- **fValue** : Floating-point value
- **iwl** : Integer Word Width
- **fwl** : Fractional Word Width
- **wl** : Total bit-vector width
- **rep** : Fixed-point representation

The main goal with creating a fixed-point object is making it compatible with the MyHDL convertible subset. Most of this is automatically handled by the underlying *intbv* object. For addition and subtraction the "point" has to be aligned. For the *fxintbv* the result from a mathematcial operation isn't automatically promoted. The object does provide asserts when the result overflows and helper functions to determine the required bit-width from the different operations. See the examples at the end of this document for more information.

2.5 Private Functions

- **_calcWidth** : Calculate the bit width required for the intger and frantional portions.
- **_toFloat** : Convert the current value to a floating-point value.
- **_fromFloat** : Covert a floating-point to fixed-point using the specified rounding mode.

3 Conversion

3.1 Addition and Subtraction

The following is an example of the fixed-point object being converted. These are very basic examples to demonstrate the conversion ability of the *fxintbv* type.

Example MyHDL fixed-point behavioral statement.

```python
from myhdl import *
from fxintbv import *

def fx_add_test(a,b,c):

    @always_comb
    def rtl():
        c.next = a + b

    return rtl

if __name__ == '__main__':
    a = Signal(fxintbv(0, min=-1, max=1, res=2**-15))
    b = Signal(fxintbv(0, min=-1, max=1, res=2**-15))
    c = Signal(fxintbv(0, min=-1, max=1, res=2**-16))

    toVerilog(fx_add_test, a, b, c)
```

Converted to Verilog

```verilog
// File: fx_add_test.v
// Generated by MyHDL 0.6
// Date: Wed Apr 29 18:12:38 2009

‘timescale 1ns/10ps

module fx_add_test (
    a,
    b,
    c
);

input signed [15:0] a;
input signed [15:0] b;
output signed [16:0] c;
wire signed [16:0] c;


assign c = (a + b);

endmodule
```

3.2 Multiplication

The following is a simple example showing the multiplication conversion.

Example The MyHDL fixed-point behavioral statement.

```python
def fx_mult_test(a,b,c):

    @always_comb
    def rtl():
        c.next = a * b

    return rtl

if __name__ == '__main__':
    a     = Signal(fxintbv(0, min=-1, max=1, res=2**-15))
    b     = Signal(fxintbv(0, min=-1, max=1, res=2**-15))
    c_mult= Signal(fxintbv(0, min=-1, max=1, res=2**-31))

    toVerilog(fx_mult_test, a, b, c_mult)
```

The converted Verilog.

```verilog
// File: fx_mult_test.v
// Generated by MyHDL 0.6
// Date: Fri May 22 11:01:11 2009

'timescale 1ns/10ps

module fx_mult_test (
    a,
    b,
    c
);

input signed [15:0] a;
input signed [15:0] b;
output signed [32:0] c;
wire signed [32:0] c;


assign c = (a * b);

endmodule
```

4 Example 1: Sum of Squares

This example is similar to the example used in [3]. This is a good example for fixed-point because it uses many aspects of the fractional representation. For these examples a floating-point model is created and then a fixed-point model using the *fxint* (simulation only, auto promotion) fixed-point object. These models are compared and debuged from the algorithm point of view. Once the fixed-point resolution has been determined an HDL fixed-point model is developed and compared to the non-hardware fixed-point model. The results for this approach follow.

The sum of squares is defined as

$$y[n] = \frac{1}{N}\sum_{k=0}^{N-1} x^2[n-k] \tag{1}$$

The floating-point and fixed-point plots are shown in Figure 2 and 3 These plots show the floating-point versus fixed-point simulation for a $Q0.3$ input. The fixed-point simulation used the *fxint* object which isn't convertible but automatically handles the fixed-point rules.

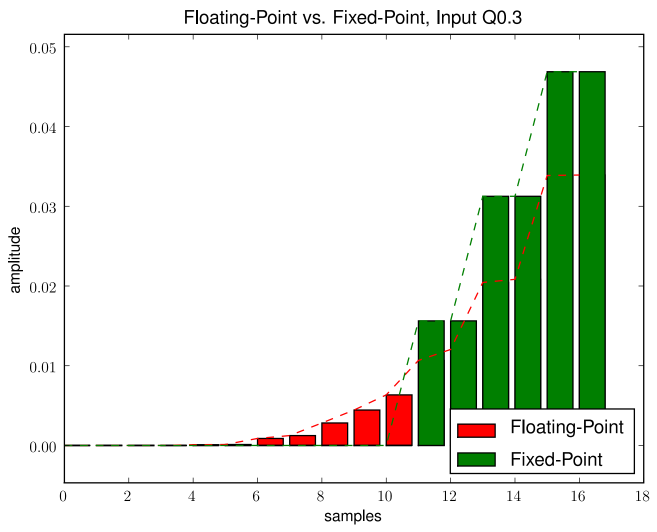

Figure 2: Floating-point (red) and Fixed-point (green)

There is an interesting side effect for the summation in the sum of squares. The summation isn't automatically handled, correctly, by the auto promotion. The auto promotion, handled by the *fxint*, will promote the individual adds. Based on the addition rules for fixed-point the sum will have more bits than it needs. For a summation only $log_2(N)$ additional bits are required. The sum representation is manually adjusted. See the code example below.

A sumation of a collection of fixed-point objects all with the same size, same Q representation, will require $IWL = IWL + log_2 N$. Where N is the number of fixed-point objects being sumed. This cause a problem for the fixed-point object because it cannot evaluate the expression. In this example we need to adjust the representation of the sumation result. Otherwise more bits are used than needed. The following listing shows the bit-widths being modified for the fixed-point model. For the actual convertible module a fixed-point summation component is used.

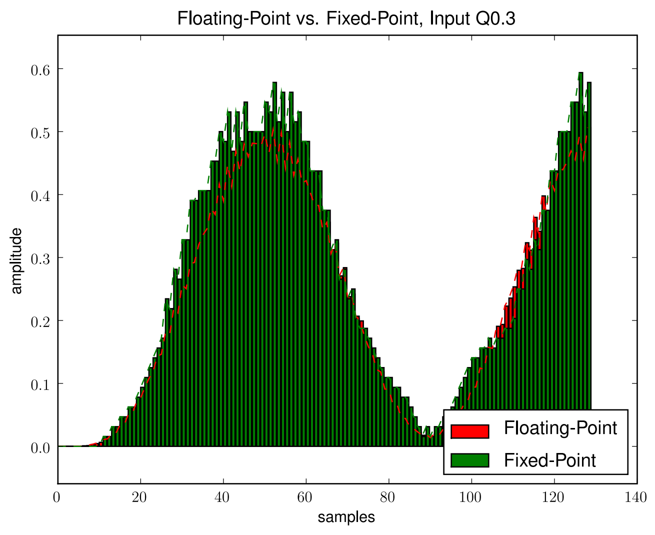

Figure 3: Floating-point (red) and Fixed-point (green)

```python
    for ii in range(1, self.N):
        # Note power operator not implemented
        sqr    = self.xbuf[ii] * self.xbuf[ii]
        self.y = self.y + sqr

    # Manually have to adjust the summation bit result.  Because the basic
    # fixed-point rule is only true for adding 2 fixed-point numbers,
    # when summing a bunch of the same sized fixed-point numbers the
    # growth is Q( (iwl+log2(N)) . fwl)
    aiw = log(self.N,2)
    self.y.NewRep(aiw, self.y.fwl)
```

Figure 4 is a diagram of the simulation. As mentioned a floating-point, fixed-point, and HDL fixed-point models were all used.

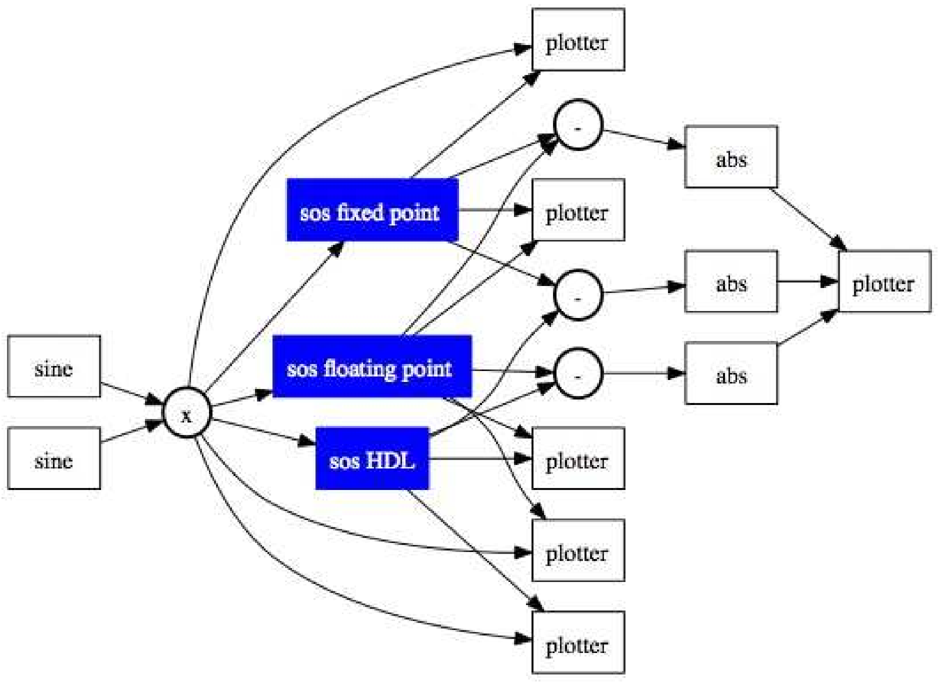

Figure 4: Simulation Configuration

Error plots were created for the implementations figure 5 shows that there is no difference between the HDL and non-HDL fixed-point models (which is good). Using a $Q.03$ it can be seen how much error is introduced. For this implementation even though the input was quantized to $Q0.3$ the rest of the calculations were full precision.

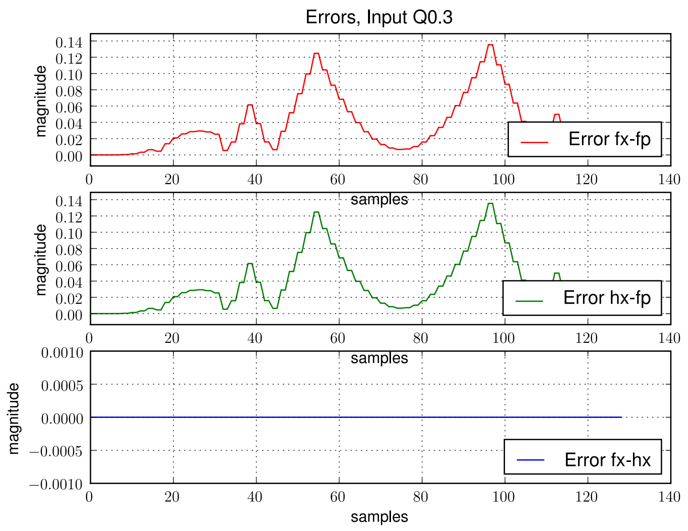

Figure 5: Fixed-Point Error for Q0.3

The following plots are different bit-widths.

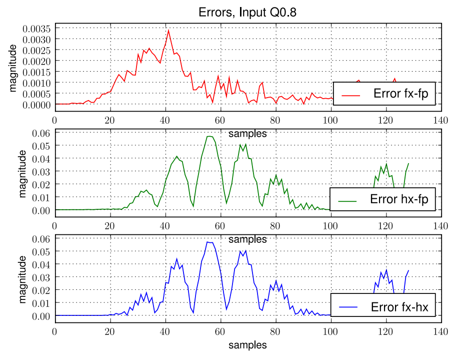

Figure 6: Fixed-Point Error for Q0.8

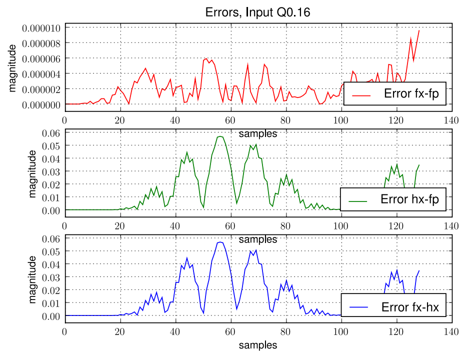

Figure 7: Fixed-Point Error for Q0.16

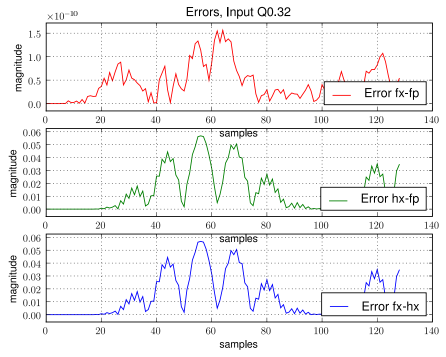

Figure 8: Fixed-Point Error for Q0.32

4.1 MyHDL Implementation

The following is the SOS MyHDL implementation and conversion.

```
def sos_hdl(
    clk,     # System clock
    rst,     # System sync reset
    x,       # Input
    fs,      # Sample strobe, new sample input (dvi)
    dv,      # Sample output valid (dvo)
    y,       # Output
    Q = 16,  # Quantization
    N = 16   # NTaps, delay
    ):

    # Different fixed-point types (sizes) used
    fx_t, fxmul_t, fxsum_t = getFxTypes(Q,N)

    nmul = Signal(intbv(0, min=0, max=N))
    taps = [Signal(copy(fxmul_t)) for ii in range(N)]
    ptap = Signal(copy(fxmul_t))

    fxm   = Signal(copy(fx_t))     # Multiply input
    fxmul = Signal(copy(fxmul_t))  # Multiply output
    fxsum = Signal(copy(fxsum_t))  # Summer output

    done = Signal(True)

    # Multiply the input by itself to pre-compute the squares
    mul = fxlib.multiply(x, x, fxmul)

    @always(clk.posedge)
    def rtl1():
        if rst:
            for ii in range(N):
                taps[N].next = 0
        else:
            if fs:
                for ii in range(N-1,0,-1):
                    taps[ii].next = taps[ii-1]
            else:
                # calculate x*x, 1 clock delay
                taps[0].next = fxmul


    # generate indexes for the summation calculation.  The SOS will only use
    # one adder and one multiply.  It will required N clock cycles to complete
    # the calculations.
    # There is also some setup
    #   clock cycle 1  -- transfer x to fxm (input to multiplier registered)
    #                  -- transfer all
    #   clock cycle 2  -- transfer fxmul (output of multiply) to tap[0]
    @always(clk.posedge)
    def rtl2():
        if rst:
            nmul.next = 0
            done.next = True
        else:
            if fs:
                # Shift happens at this clock, (i.e 14 --> 15).  Grab the
                # current N-2 which will be the N-1 after the shift.  This
                # means N-2 is grabbed twice.
                done.next  = False
                nmul.next  = N-2
                fxsum.next = taps[N-2]  # N-1 before shift
            elif nmul > 0 and nmul < N-1:
                done.next = False
                nmul.next -= 1
                fxsum.next = fxsum + ptap
            elif nmul == 0:
                done.next  = True
                nmul.next  = N-1
                fxsum.next = fxsum + ptap
            else:
                done.next  = True
                nmul.next  = N-1


    @always_comb
    def rtl3():
        ptap.next = taps[int(nmul)]


    # register the inputs and outputs
    @always(clk.posedge)
    def rtl4():
        assert isinstance(x.val,fxintbv), "x should be of type fixed_point.fxintbv and not %s" % (typ
        if rst:
            fxm.next = 0
            y.next   = 0
        else:
            fxm.next = x
            if done and not fs:
                y.next  = fxsum
                dv.next = True
            else:
                dv.next = False

    return mul, rtl1, rtl2, rtl3, rtl4
```

References

[1] Declawe, Jan These integers are made for counting ... www.jandecaluwe.com/hdldesign/counting.html

[2] Declawe, Jan MyHDL Manual www.myhdl.org/manual

[3] Yates, Randy Fixed-Point Arithmetic: An Introduction www.digitalsignallabs.com/fp.pdf

[4] Yates, Randy Practical Considerations in Fixed-Point Arithmetic: FIR Filter Implementations www.digitalsignallabs.com/fir.pdf

[5] K.B. Cullen, G.C.M. Silvestre, and .J. Hurley Simulation Tools for Fixed Point DSP Algroithms and Architectures

[6] Dillon Engineering, Guenter deModel www.dilloneng.com/documents/downloads/demodel

[7] Ercegovac, Milos Lang, Thomas Digital Arithmetic

[8] Due, Lee, Tian Real-Time Digital Signal Processing