

System Design Flow and Fixed-point Arithmetic

Lecture 3

Dr. Shoab A. Khan

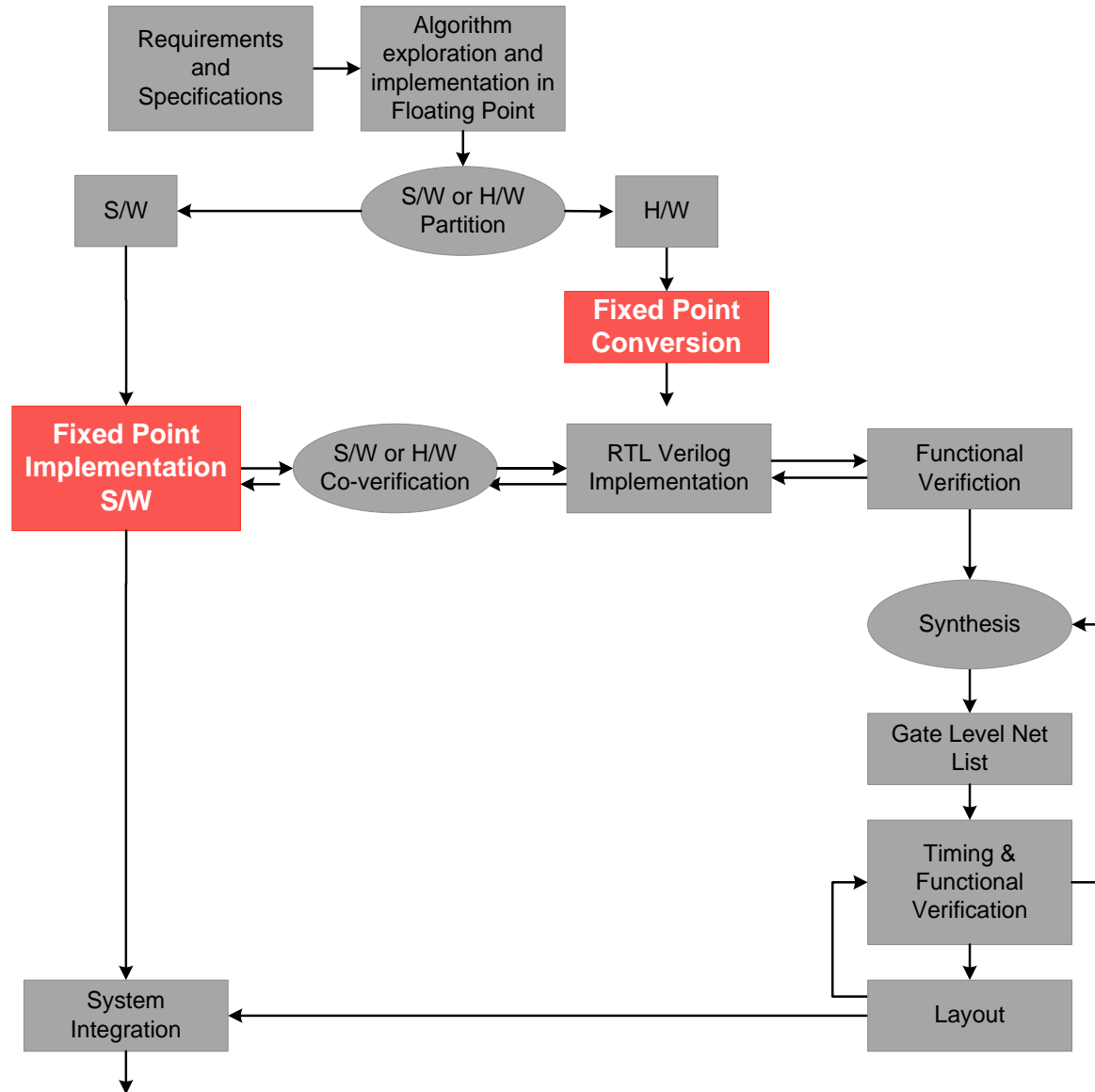
Contents

- Floating and Fixed Point Arithmetic
- System Design Flow
 - Requirements and Specifications (R&S)
 - Algorithmic Development in Matlab and Coding Guidelines
- 2's Complement Arithmetic
- Floating Point Format
- $Q_n.m$ format for Fixed Point Arithmetic
- Addition, Multiplication and Scaling in $Q_n.m$
- LTI systems and implementation in $Q_n.m$ format

Floating & Fixed Point Arithmetic

- Two Types of arithmetic
 - **Floating Point Arithmetic**
 - After each arithmetic operation numbers are normalized
 - Used where precision and dynamic range are important
 - Most algorithms are developed in FP
 - Ease of coding
 - More Cost (Area, Speed, Power)
 - **Fixed Point Arithmetic**
 - Place of decimal is fixed
 - Simpler HW, low power, less silicon
 - Converting FP simulation to Fixed-point simulation is time consuming
 - Multiplication doubles the number of bits
 - $N \times N$ multiplier produces $2N$ bits
 - The code is less readable, need to worry about overflow and scaling issues

System-level Design Flow and Fixed-point Arithmetic



System Design Flow

- The requirements and specifications of the application are captured
- The algorithms are then developed in double precision floating point format
 - Matlab or C/C++
- A signal processing system general consists of hybrid target technologies
 - DSPs, FPGAs, ASICs
- For mapping application developed in double precision is partitioned into
 - hardware & software
- Most of signal processing applications are mapped on Fixed-point Digital Signal Processors or HW in ASICs or FPGAs
- The HW and SW components of the application are **converted** into Fixed Point format for this mapping

1st Step: Requirement & Specification

- Gathering R&S is the first part of the system design
- System components and algorithms are then selected that meets the requirements
- Example R&S of a UHF radio are listed in the table

Characteristics	Specification
Output power	2W
Spurious emission	< 60 dB
Harmonic suppression	> 55 dB
Frequency stability	2 ppm or better
Reliability	> 10,000 hours MTBF minimum < 30 minutes MTTR
Handheld	12V DC nickel metal hydride, nickel cadmium or lithium-ion (rechargeable) battery pack

R&S of a UHF Radio (Cont)

Characteristic	Specification
Frequency range	420 MHz to 512 MHz
Data rate	Up to 512 kbps multi-channel non-line of sight
Channel	Multi-path with 15 μ s delay spread and 220 km/h relative speed between transmitter and receiver
Modulation	OFDM supporting BPSK, QPSK and QAM
FEC	Turbo codes, convolution, Reed–Solomon
Frequency hopping	> 600 hops/s, frequency hopping on full hopping band
Waveforms	Radio works as SDR and should be capable of accepting additional waveforms

Next Step: Algorithm Development and Mapping

- The R&S related to digital design are forwarded to algorithm developers and system designers
- Algorithms are coded in behavioral modeling tools like Matlab
- The Matlab code is then translated into a high level language, for example, C/C++
- System is designed based on R&S
 - System usually consists of hybrid technologies consisting of ASICs, DSPs, GPP, and FPGAs
- Partitioning of the application into HW/SW parts is performed
- The SW is then developed for the SW part and architectures are designed and implemented for the HW parts
- Integration and testing is performed throughout the design cycle

Guidelines for Coding Algorithms in Matlab

- Signal processing applications are mostly developed in Matlab
- As the Matlab code is to be mapped in HW and SW so adhering to coding guidelines is critical
 - The code must be designed to work for processing of data in chunks
 - The code should be structured in distinct components
 - Well defined interfaces in terms of input and output arguments and internal data storages
 - All variables and constants should be defined in data structures
 - User defined configurations in one structure
 - System design constants in another structure
 - Internal states for each block in another structure
 - Initialization in the start of simulation

Processing in Chunks

```
% BPSK = 1, QPSK = 2, 8PSK = 3, 16QAM = 4
% All-user defined parameters are set in structure USER_PARAMS
USER_PARAMS.MOD_SCH = 2; %select QPSK for current simulation
USER_PARAMS.CHUNK_SZ = 256; %set buffer size
USER_PARAMS.NO_CHUNKS = 100;% set no of chunks for simulation
% generate raw data for simulation
raw_data = randint(1, USER_PARAMS.NO_CHUNKS*USER_PARAMS.CHUNK_SZ)
% Initialize user defined, system defined parameters and states
PARAMS = MOD_Params_Init(USER_PARAMS);
STATES = MOD_States_Init(PARAMS);
mod_out = [];
% Code should be structured to process data on chunk-by-chunk basis
for iter = 0:USER_PARAMS.NO_CHUNKS-1
    in_data = raw_data
        (iter*USER_PARAMS.CHUNK_SZ+1:USER_PARAMS.CHUNK_SZ*(iter+1));
    [out_sig,STATES]= Modulator(in_data,PARAMS,STATES);
    mod_out = [mod_out out_sig];
end
```

Parameters & Initialization

```
% Initializing the user defined parameters and system design parameters in PARAMS
function PARAMS = MOD_Params_Init(USER_PARAMS)
% Structure for transmitter parameters
PARAMS.MOD_SCH = USER_PARAMS.MOD_SCH;
PARAMS.SPS = 4; % Sample per symbol
% Create a root raised cosine pulse-shaping filter
PARAMS.Nyquist_filter = rcosfir(.5 , 5, PARAMS.SPS, 1);
% Bits per symbol, in this case bits per symbols is same as mod scheme
PARAMS.BPS = USER_PARAMS.MOD_SCH;
% Lookup tables for BPSK, QPSK, 8-PSK and 16-QAM using gray coding
BPSK_Table = [(-1 + 0*j) (1 + 0*j)];
QPSK_Table = [(-.707 - .707*j) (-.707 + .707*j) (.707 - .707*j) (.707 + .707*j)];
PSK8_Table = [(1 + 0j) (.7071 + .7071i) (-.7071 + .7071i) (0 + i)...
              (-1 + 0i) (-.7071 - .7071i) (.7071 - .7071i) (0 - 1i)];
QAM_Table = [(-3 + -3*j) (-3 + -1*j) (-3 + 3*j) (-3 + 1*j) (-1 + -3*j)...
             (-1 + -1*j) (-1 + 3*j) (-1 + 1*j) (3 + -3*j) (3 + -1*j)...
             (3 + 3*j) (3 + 1*j) (1 + -3*j) (1 + -1*j) (1 + 3*j) (1 + 1*j)];
% Constellation selection according to bits per symbol
if(PARAMS.BPS == 1)
    PARAMS.const_Table = BPSK_Table;
elseif(PARAMS.BPS == 2)
    PARAMS.const_Table = QPSK_Table;
elseif(PARAMS.BPS == 3)
    PARAMS.const_Table = PSK8_Table;
elseif(PARAMS.BPS == 4)
    PARAMS.const_Table = QAM_Table;
else
    error('ERROR!!! This constellation size not supported')
end
```

States & Initializations

- `function STATES = MOD_States_Init(PARAMS)`
- `% Pulse shaping filter delayline`
- `STATES.filter_delayline =
zeros(1,length(PARAMS.Nyquist_filter)-1);`

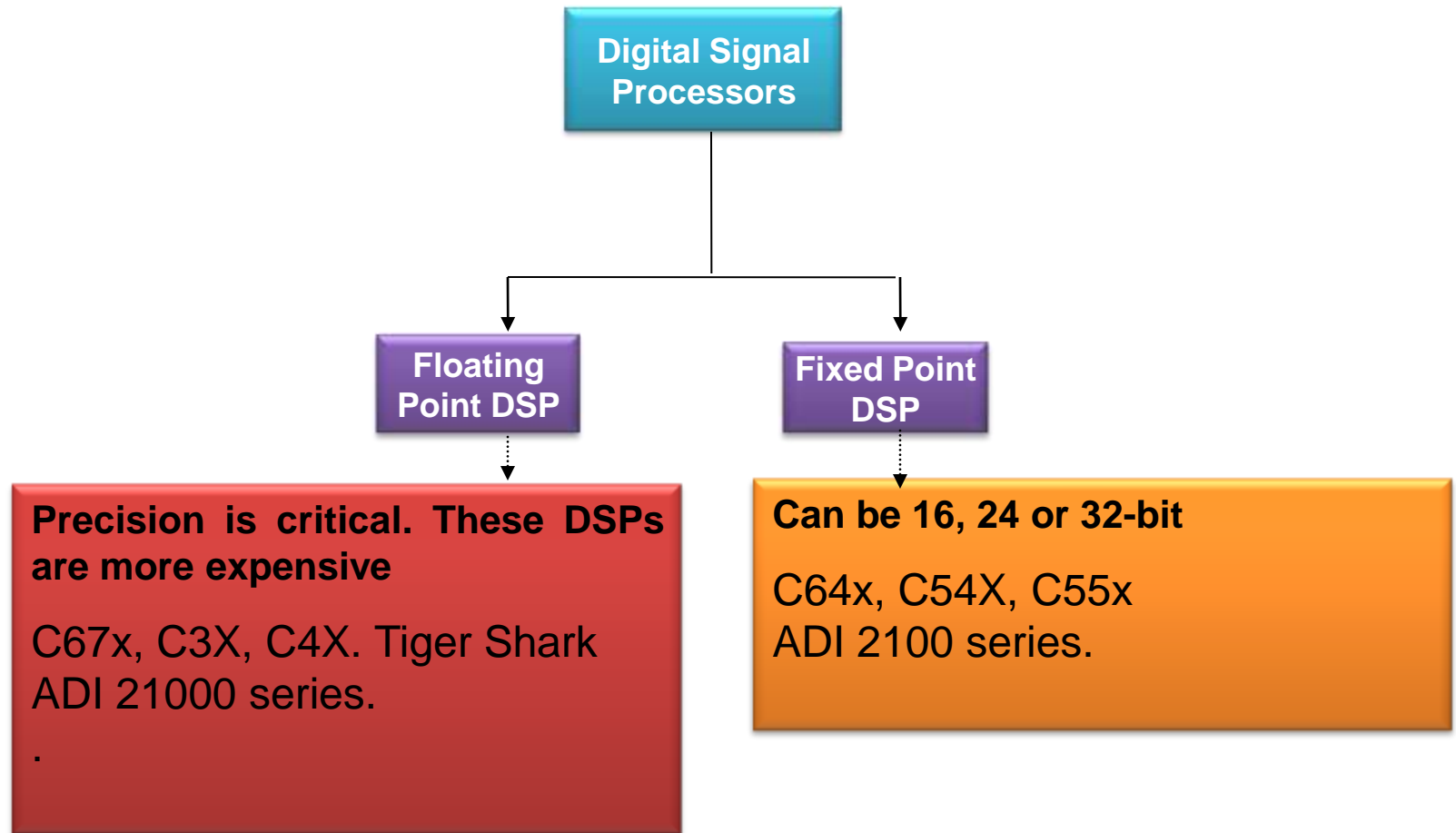
Chunk by chunk processing

```
function [out_data, STATES] = Modulator(in_data, PARAMS, STATES);  
% Bits to symbols conversion  
sym = reshape(in_data,PARAMS.BPS,length(in_data)/PARAMS.BPS)';  
% Binary to decimal conversion  
sym_decimal = bi2de(sym);  
% Bit to symbol mapping  
const_sym = PARAMS.const_Table(sym_decimal+1);  
% Zero padding for up-sampling  
up_sym = upsample(const_sym,PARAMS.SPS);  
% Zero padded signal passed through Nyquist filter  
[out_data, STATES.filter_delayline] =  
filter(PARAMS.Nyquist_filter,1,up_sym, STATES.filter_delayline);
```

Fixed-point v/s Floating-point Hardware

- Algorithms are developed in floating point format using tools like Matlab
- Floating point processors and HW are expensive
- Fixed-point processors and HW are used in embedded systems
- After algorithms are designed and tested then they are converted into fixed-point implementation
- The algorithms are ported on Fixed-point processor or application specific hardware

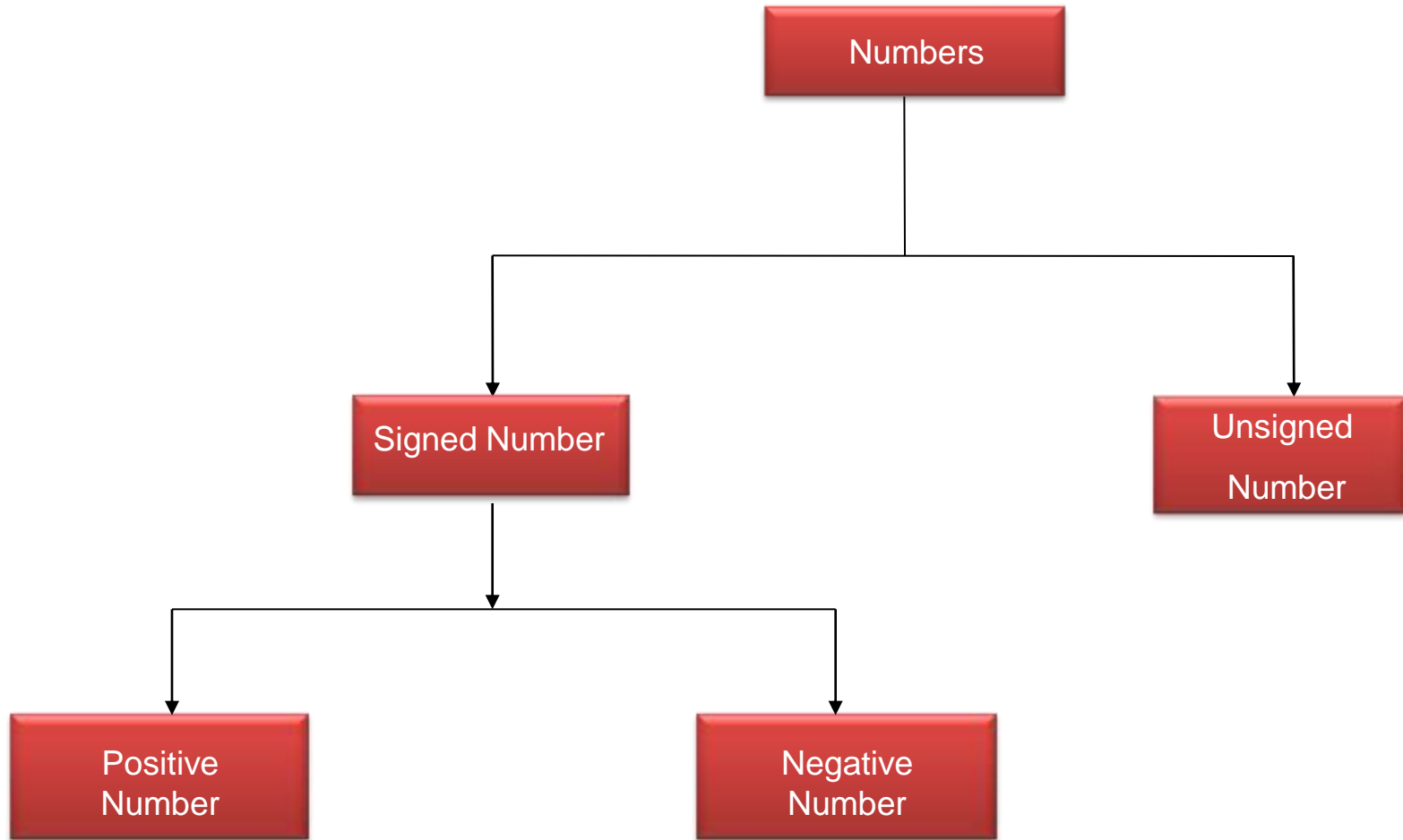
Digital Signal Processors



Representation of Numbers

- In a digital design fixed or floating point numbers are represented in binary format
- Types of Representation
 - one's complement
 - sign magnitude
 - canonic sign digit (CSD)
 - two's complement
- In digital system design for fixed point implementation the canonic sign digit (CSD), and two's complement are normally used

2's Complement Arithmetic



2's Complement Arithmetic

- MSB has negative weight,
 - Positive number $a_{N-1} = 0$
 - Negative number $a_{N-1} = 1$

Example

1 0 1 1 (Negative number as MSB = 1)

2^3

2^2

2^1

2^0

$$-8 + 2 + 1 = -5$$

Equivalent Representation

Many design tools do not display numbers as 2's complement signed numbers

A signed number is represented as an equivalent unsigned number

Equivalent unsigned value of an N-bit negative number is

$$2^N - |a|$$

Example

for -5 = 1011

$$N=4$$

$$a=-5$$

$$2^4 - |-5| = 16 - 5 = +11$$

In binary it is equivalent rep is 1011

Four-bit representation of two's complement and equivalent unsigned numbers

Decimal number	Two's complement representation				Equivalent unsigned number
	-2^3	2^2	2^1	2^0	
0	0	0	0	0	0
+1	0	0	0	1	1
+2	0	0	1	0	2
+3	0	0	1	1	3
+4	0	1	0	0	4
+5	0	1	0	1	5
+6	0	1	1	0	6
+7	0	1	1	1	7
-8	1	0	0	0	8
-7	1	0	0	1	9
-6	1	0	1	0	10
-5	1	0	1	1	11
-4	1	1	0	0	12
-3	1	1	0	1	13
-2	1	1	1	0	14
-1	1	1	1	1	15

Computing Two's Complement of a Signed Number

- Refers to the negative of a number
- Computes by inverting all bits and adding 1 to the least significant bit (LSB) of the number
- This is equivalent to inverting all the bits while moving from LSB to MSB and leaving the least significant 1 as it is
- From the hardware perspective, adding 1 requires an adder in the HW and is an expensive proposition

Scaling and sign extension

- Scaling to Avoid Overflows
 - Overflow adds an error that equals to the complete dynamic range of the number
 - To avoid overflow, numbers are scaled down before mathematical operations
- Sign Extension
 - Numbers may also need to be sign extended

Sign Extension

- An N bit number is extended to an M bit number $M > N$, by replicating M-N sign bits to the most significant bit positions
 - Positive number: M-N extended bits are filled with 0s
 - The number unsigned value remains the same
 - Negative number: M-N extended bits are filled with 1s,
 - Signed value remains the same
 - Equivalent unsigned value is changed

4'b1000 2's complement sign number is sign extend to 8'b1111_1000

Dropping Redundant Sign bits

- When a number has redundant sign bits, these redundant bits can be dropped
- This dropping of bits does not affect the value of the number

Example

$8'b1111_1000 = -8$

Is same as

$4'b1000 = -8$

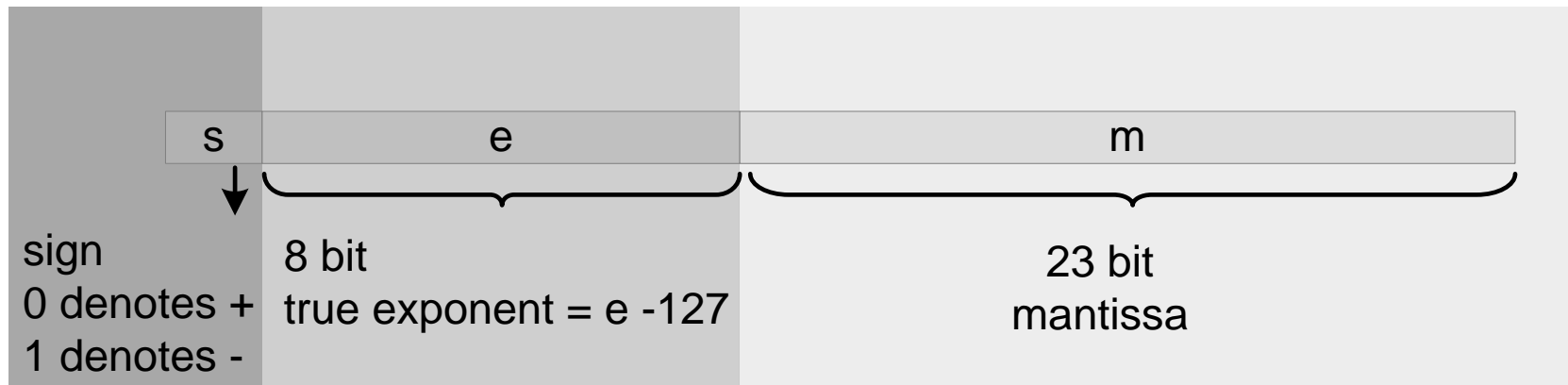
Floating Point Format

- Floating point arithmetic is appropriate for high precision applications
- Applications that deals with number with wider dynamic range
- A floating point number is represented as

$$x = (-1)^s \times 1 \times m \times 2^{e-b}$$

- s represents sign of the number
- m is a fraction number >1 and < 2
- e is a biased exponent, always positive
 - The bias \mathbf{b} is subtracted to get the actual exponent

IEEE floating point format for single precision 32-bit floating point number



Example Floating Point Representation

32-bit IEEE Floating point number is

0_10000010_11010000_00000000_00000000

Sign bit	Exponent	Mantissa
0	10000010	11010000_00000000_00000000
$(-1)^0$	$\times 2^{(130-127)}$	$\times (1.1101)_2$
1	$\times 2^{(3)}$	$\times (1 + 0.5 + .25 + 0 + 0.0625)_{10}$
$(+1)$	$\times 2^{(3)}$	$\times (1.8125)_{10}$

Floating-point Arithmetic Addition

S0: Append the implied 1 of the mantissa

S1: Shift the mantissa from S0 with smaller exponent e_s to the right by $e_l - e_s$,

where e_l is the larger of the two exponents

S2: For negative operand take two's complement and then add the two mantissas.

If the result is negative, again takes two's complement of the result for storing in IEEE format

S3: Normalize the sum back to IEEE format by adjusting the mantissa and appropriately changing the value of the exponent e_l

S4: Round or truncate the resultant mantissa to fit in IEEE format

Example: Floating-point Addition

Add two floating point numbers in 10-bit precision

4 bits for exponent and 5 bits for the mantissa and 1 bit for the sign, bias value is 7.

0_1010_00101

0_1001_00101

Taking the bias +7 off from the exponents and appending the implied 1, the numbers are

$1.00101_b \times 2^3$ and $1.00101_b \times 2^2$

S0: Align two exponents by shifting the mantissa of the number with smaller exponent accordingly

$1.00101_b \times 2^3 \Rightarrow 1.001010_b \times 2^3$

$1.00101_b \times 2^2 \Rightarrow 0.100101_b \times 2^3$

S1: Add the mantissas

$1.001010_b + 0.100101_b = 1.101111_b$

S2: Drop the guard bit $1.10111_b \times 2^3$

S3: The final answer is $1.10111_b \times 2^3$ which in 10-bit format of the operands is

0_1010_10111

Floating-point Multiplication

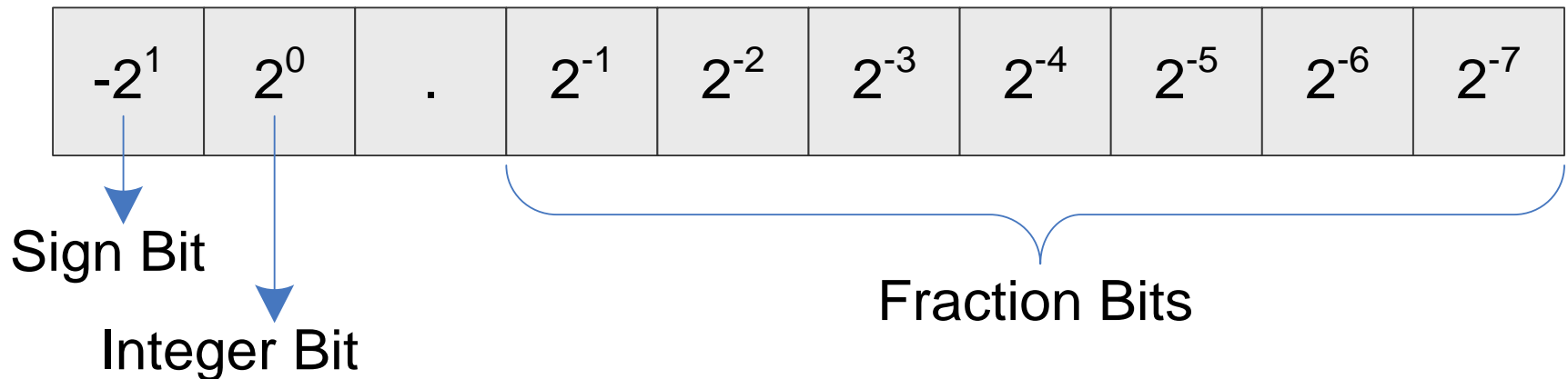
- S0:** Add the two exponents e_1 and e_2 and subtract the bias once
- S1:** Multiply the mantissas as unsigned numbers to get the product, and XOR the two sign bits to get the sign of the product
- S2:** Normalize the product if required
- S3:** Round or truncate the mantissa

Fixed-point format

Algorithms in double precision Floating-point format are converted into Fixed-Point format for mapping on low cost DSPs and Application Specific HW designs

Qn.m Format for Fixed-point Arithmetic

- Qn.m format is a fixed positional number system for representing floating-point numbers
- A Qn.m format N-bit binary number assumes n bits to the left and m bits to the right of the binary point



Qn.m Positive Number

- The MSB is the sign bit
- For a positive fixed-point number, MSB is 0

$$b = 0b_{n-2} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-m}$$

- Equivalent floating point value of the positive number is

$$b = b_{n-2} 2^{n-2} + \dots + b_1 2^1 + b_0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} + \dots + b_{-m} 2^{-m}$$

- For Negative numbers, MSB has negative weight and equivalent value is

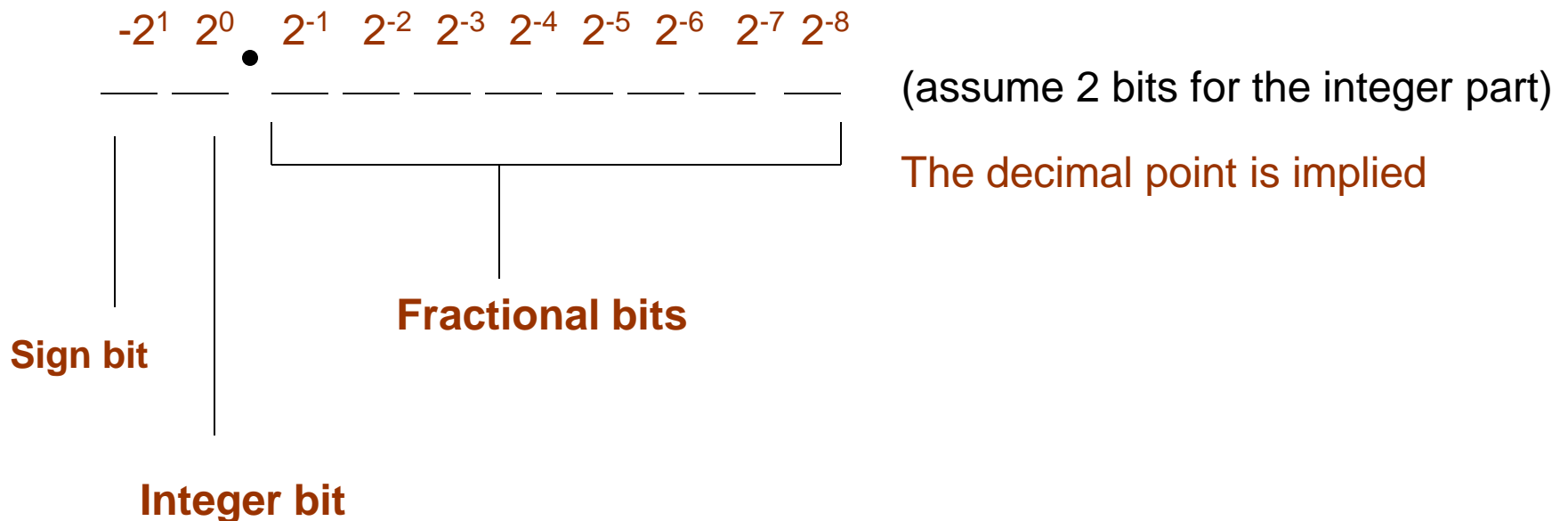
$$b = -b_{n-1} 2^{n-1} + b_{n-2} 2^{n-2} + \dots + b_1 2^1 + b_0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} + \dots + b_{-m} 2^{-m}$$

Conversion to Qn.m

1. Define the total number. of bits to represent a Qn.m number

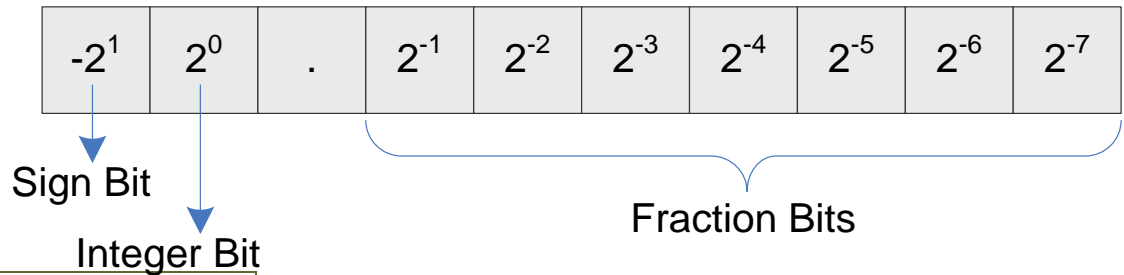
_____ (assume 10 bits)

2. Fix location of decimal based on the value of the number



Example

01 1101 0000



In Q2.8 format, the number is :

$$1 + 1/2 + 1/2^2 + 1/2^4 = 1.8125$$

Equivalent fixed-point integer value is

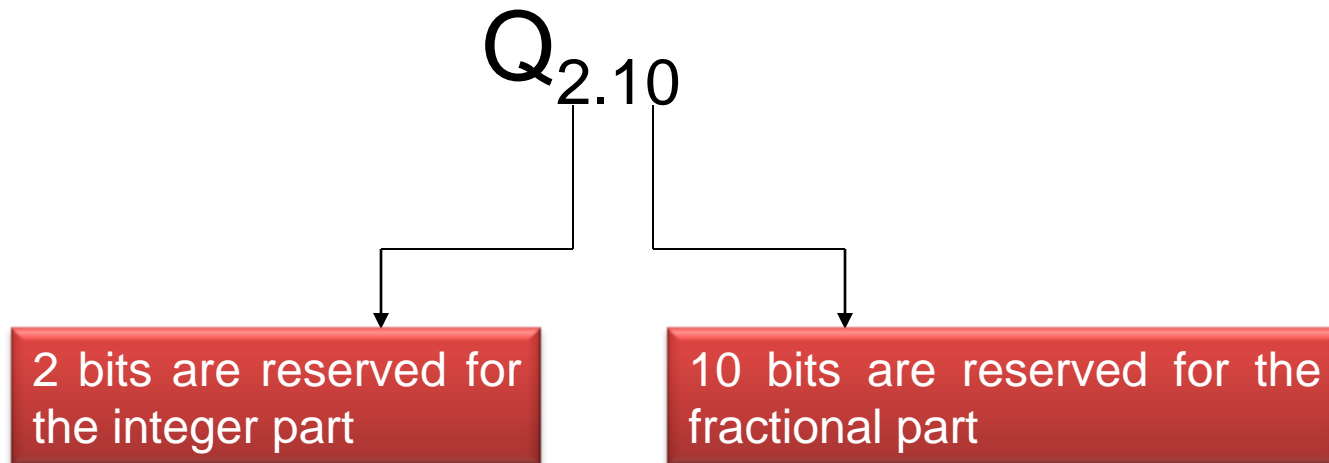
0x1D0

- 2 bits for the integer and remaining 8 bits keeps the fraction part
- A 10 bit Q2.8 signed number covers -2 to +1.9922.
- Increasing the fractional bits increases the precision.

In $Q_{n.m}$ format,

n entirely depends upon the range of integer

m defines the precision of the fractional part



Fixed-point numbers in COTS DSPs

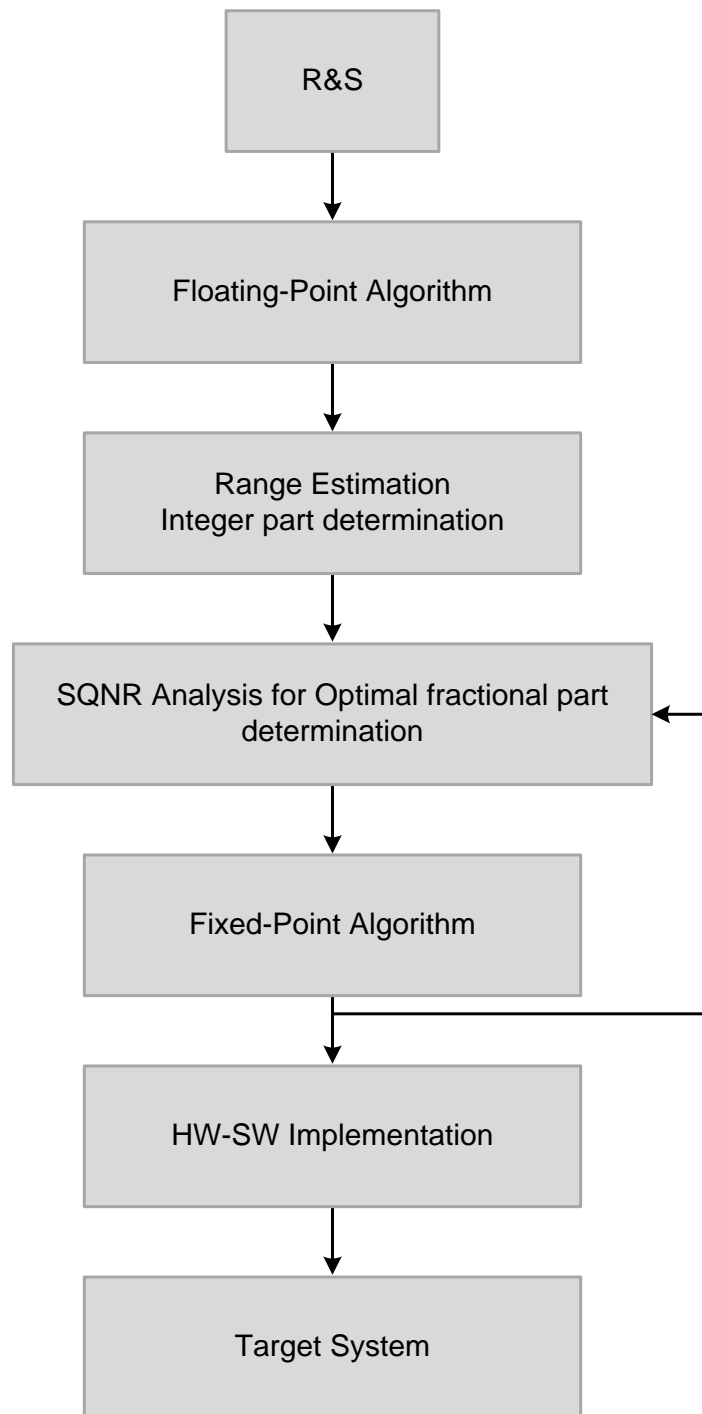
- Commercially available off the shelf processors usually have 16 bits to represent a number
- In C, a programmer can define 8, 16 or 32 bit numbers as **char**, **short** and **int/long** respectively
- In case a variable requires different precision than what can be defined in C, the number is defined in higher precision
- For example an 18-bit number should be defined as a 32-bit integer
- High precision arithmetic is performed using low precision arithmetic operations
 - 32-bit addition requires two 16-bit addition
 - 32-bit multiplication requires four 16-bit multiplications

Floating Point to Fixed Point Conversion

- Serialize the floating-point code to separate all atomic computations and assignments to variables
- Insert range directives after each serial floating-point computation
- Runs the serialized implementation with range directives for all set of possible inputs
- Convert all floating point variables to fixed point format using the maximum and minimum values for each variable
- The integer part for var_i is defined as

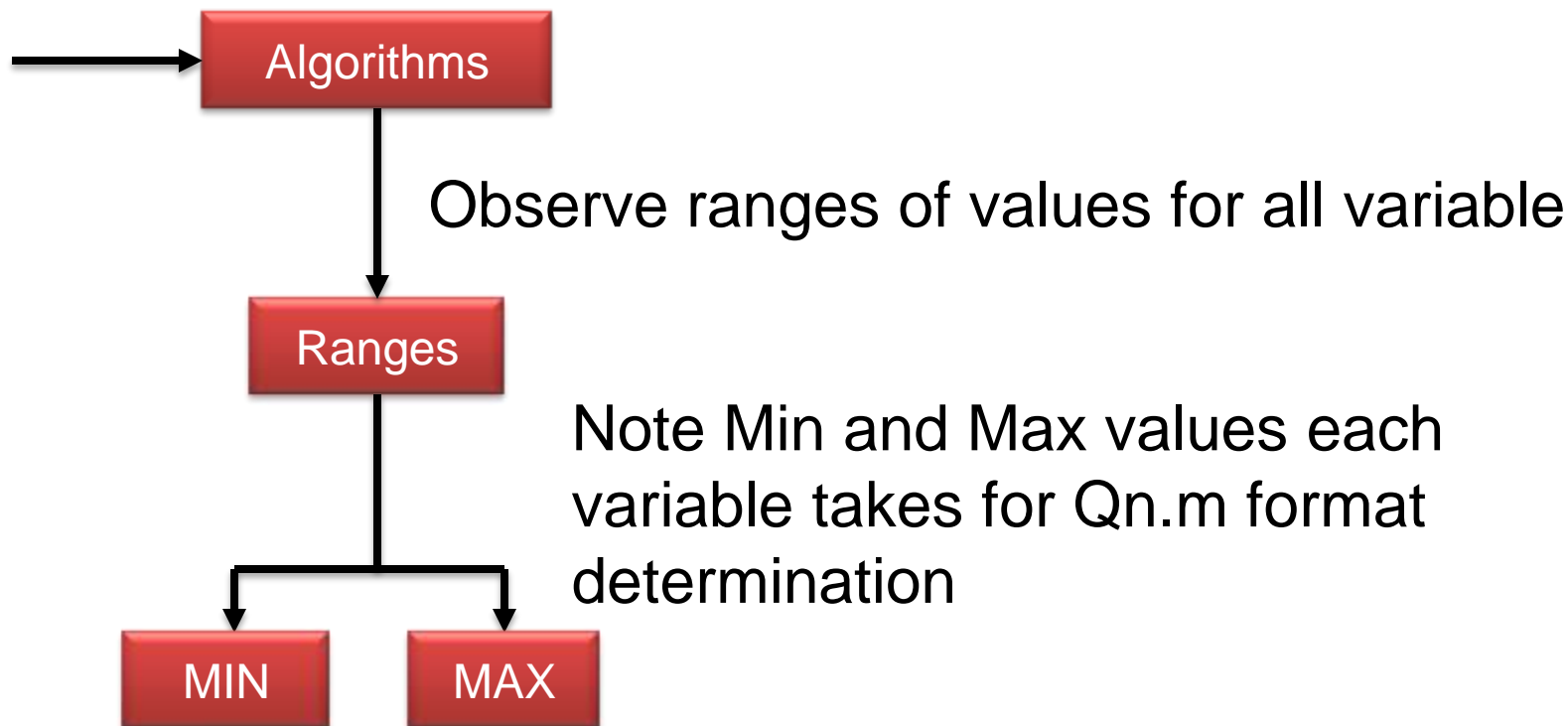
$$n_{i=\log_2} (\max|\max_val_i, \min_val_i|) + 1$$

- The fractional part m requires detail analysis of Signal to Quantization Noise analysis
 - Active area of research
 - Model based, exhaustive search based or optimization based techniques



Range Determination for Qn.m Format

Run simulations for all set of inputs



Floating-point to Fixed-point Conversion

- Shift left m fractional bits to the integer part and truncate or round the results
 - In Matlab, the conversion is done as

num_fixed = fix(num_float * 2^m) or

num_fixed = round(num_float * 2^m)

- Saturate if number is greater than the maximum positive number or less than the minimum negative number.

**if (num_fixed > 2^{N-1} - 1)
 num_fixed = 2^{N-1} - 1
else if (num_fixed < -2^{N-1})
 num_fixed = -2^{N-1}**

Matlab Support for Fixed-Point Arithmetic

```
>> PI = fi(pi, 1, 3+5, 5);  
>> bin(PI)  
01100101  
>> double(PI)  
    3.1563
```

Matlab Supports Fixed Point Arithmetic

- All the attributes of fixed point variable can be set for fixed point arithmetic
- These attributes are shown here

```
PI =  
    3.1563  
      DataType: Fixed  
      Scaling: Binary Point  
      Signed: true  
      WordLength: 8  
      FractionLength: 5  
      RoundMode: round  
      OverflowMode: saturate  
      ProductMode: Full Precision  
      MaxProductWordLength: 128  
      SumMode: Full Precision  
      MaxSumWordLength: 128  
      CastBeforeSum: true
```

Conversion using C Program

```
num_fixed_long = (long)(num_float * 215 )
```

```
if (num_fixed_long > 0x7fff)
```

```
    num_fixed_long = 0x7fff
```

```
elseif (num_fixed_long < 0xffff8000)
```

```
    num_fixed_long = 0xffff8000
```

```
num_fixed_Q15 = (short)(num_fixed_long & 0xffff)
```

Using this logic, the following lists a few floating-point numbers (num_float) and their

representation in Q1.15 fixed-point format (num_fixed_Q15):

0.5 → 0x4000

-0.5 → 0xC000

0.9997 → 0x7FFF

0.213 → 0x1B44

-1.0 → 0x8000

SystemC supports Fixed-point

- SystemC provides a very convenient tool to convert a floating point implementation to fixed point format
- Uses floating point implementation with redefined floating point variables

SystemC supports Fixed-point

```
int sc_main (int argc , char *argv[])
{
    sc_fixed <16,10, SC_TRN, SC_SAT> fx_value;
    double fp_value;
    int i, size;
    ofstream fout("fx_file.txt");
    ifstream fin("fp_file.txt");
    if (fin.is_open())
        fin >> size;
    else
        cout << "Error opening input file!\n";
    cout << "size = " << size << endl;
    for (i=0; i<size; i++)
    {
        if (!fin.eof())
        {
            fin >> fp_value;
            fx_value = fp_value;
            cout << "double = " << fp_value"\t fixpt =
"<<fx_value<<endl;
            fout << fx_value<< endl;
        }
    }
}
```

Arithmetic: Addition in Q Format

Addition of two fixed-point numbers a and b of $Q_{n_1.m_1}$ and $Q_{n_2.m_2}$ formats, respectively, results in a $Q_{n.m}$ format number, where n is the larger of n_1 and n_2 and m is the larger of m_1 and m_2 .

Example

implied decimal

$Q_{n_1.m_1}$	1	1	1	1	•	1	0			$= Q_{4.2} = -2+1+0.5 = -0.5$
$Q_{n_2.m_2}$	0	1	1	1	0	1	1	0		$= Q_{4.4} = 1+2+4+0.25+0.125 = 7.375$
$Q_{n.m}$	0	1	1	0	1	1	1	0		$= Q_{4.4} = 2+4+0.5+0.25+0.125 = 6.875$

Multiplication in Q Format

- Multiplications of two numbers in $Q_{n1.m1}$ and $Q_{n2.m2}$ formats generates product in $Q_{(n1 + n2).(m1 + m2)}$ format
- For signed x signed multiplication the MSB is a redundant sign bit
 - First Drop this bit by shifting the result to the left
 - Then Drop less precision bits to bring the number back to less number of bits

Multiplication in Q -Format

$$Q_{n1.m1} \times Q_{n2.m2} = Q_{(n1+n2).(m1+m2)}$$

Four types of Fractional Multiplication:

Unsigned Unsigned

Unsigned Signed

Signed Unsigned

Signed Signed

Signed x Signed multiplication, results in a redundant sign bit

Unsigned by Unsigned

- The partial products are added without any sign extension logic

$$\begin{array}{r}
 1\ 1\ 0\ 1 = 11.01 \text{ in } Q2.2 = 3.25 \\
 1\ 0\ 1\ 1 = 10.11 \text{ in } Q2.2 = 2.75 \\
 \hline
 1\ 1\ 0\ 1 \\
 1\ 1\ 0\ 1\ X \\
 0\ 0\ 0\ 0\ X\ X \\
 1\ 1\ 0\ 1\ X\ X\ X \\
 \hline
 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 = 1000.1111 \text{ in } Q4.4 \text{ i.e. } 8.9375
 \end{array}$$

Signed by Unsigned

- Sign extension of each partial product is necessary in signed-unsigned multiplication.
- The partial products are first sign-extended and then added

$$1\ 1\ 0\ 1 = 11.01 \text{ in } Q2.2 = -0.75$$

$$0\ 1\ 0\ 1 = 01.01 \text{ in } Q2.2 = 1.25$$

1 1 1 1 1 1 0 1 extended sign bits shown in bold

0 0 0 0 0 0 0 X

1 1 1 1 0 1 X X

0 0 0 0 0 X X X

$$1\ 1\ 1\ 1\ 0\ 0\ 0\ 1 = 1111.0001 \text{ in } Q4.4 \text{ i.e. } -0.9375$$

Unsigned by Signed

- The unsigned multiplicand is changed to a signed positive number
 - its 2's complement is computed if the signed number is negative
- The last product is a signed number

$$1\ 0\ 0\ 1 = 10.01 \text{ in } Q2.2 = 2.25 \text{ (unsigned)}$$

$$1\ 1\ 0\ 1 = 11.01 \text{ in } Q2.2 = -0.75 \text{ (signed)}$$

$$1\ 0\ 0\ 1$$

$$0\ 0\ 0\ 0\ X$$

$$1\ 0\ 0\ 1\ X\ X$$

$$1\ 0\ 1\ 1\ 1\ X\ X\ X$$

2's complement of the positive multiplicand 01001

$$1\ 1\ 1\ 0\ 0\ 1\ 0\ 1 = 1110.0101 \text{ in } Q4.4 \text{ i.e. } -1.6875$$

Signed by Signed

- Sign extend all partial products
- Takes 2's complement of the last partial product if multiplier is a negative number.
- The MSB of the product is a redundant sign bit
 - Removed the bit by shifting the product to left, the product is in

$$Q_{(n_1 + n_2 - 1) . (m_1 + m_2 + 1)}$$

$$\begin{array}{r} 1\ 1\ 0 \\ 0\ 1\ 0 \end{array} = Q_{1.2} = -0.5 \text{ (signed)}$$

$$= Q_{1.2} = 0.5 \text{ (signed)}$$

$$\begin{array}{r} 0\ 0\ 0\ 0\ 0\ 0 \\ 1\ 1\ 1\ 1\ 0\ X \\ 0\ 0\ 0\ 0\ X\ X \end{array}$$

$$1\ 1\ 1\ 1\ 0\ 0 = Q_{1.5} \text{ format } 1_11000 = -0.25$$

Example: Signed x Signed

1 1. 0 1 = -0.75 in Q2.2 format

1. 1 0 1 = -0.375 in Q1.3 format

1	1	1	1	1	1	0	1
0	0	0	0	0	0	0	X
1	1	1	1	0	1	X	X
0	0	0	1	1	X	X	X

0 0 0 0 1 0 0 1 = shifting left by one 00.010010 in Q2.6 format is 0.2815

Corner Case:

Signed-Signed Fractional Multiplication

- $-1 \times -1 = 1$ in Q fractional format is a corner case, it should be checked and result should be saturated as max positive number

$$1\ 0\ 0 = Q1.2 = -1$$

$$1\ 0\ 0 = Q1.2 = -1$$

0	0	0	0	0	0	0	
0	0	0	0	0	0	X	
0	1	0	0	X	X		2's Compliment
0	1	0	0	0	0	0	Dropping redundant sign bit results 1000000 = -1 in Q1.5

Fixed Point Multiplication

```
Word32 L_mult(Word16 var1,Word16 var2)
{
    Word32 L_var_out;

    L_var_out = (Word32)var1 * (Word32)var2;
    if (L_var_out != (Word32)0x40000000L)    // 0x8000 x 0x8000 =
0x40000000
    {
        L_var_out *= 2;                //remove the redundant bit
    }
    else
    {
        Overflow = 1;
        L_var_out = 0x7fffffff; //if overflow then clamp to max +ve value
    }

    return(L_var_out);
}
```

Fractional and Integer Arithmetic and truncation

4-bit Integer Arithmetic

$$1001 = -7_{10}$$

$$0111 = +7_{10}$$

11111001

1111001X

111001XX

$$11001111 = -49_{10}$$

$$\begin{aligned} 11001111 &\Rightarrow 1111 \\ &\Rightarrow -1(\text{overflow}) \end{aligned}$$

Q1.3 Fractional Arithmetic

$$1.001 = -0.875_{10}$$

$$0.111 = 0.875_{10}$$

11.111001

11.11001X

11.1001XX

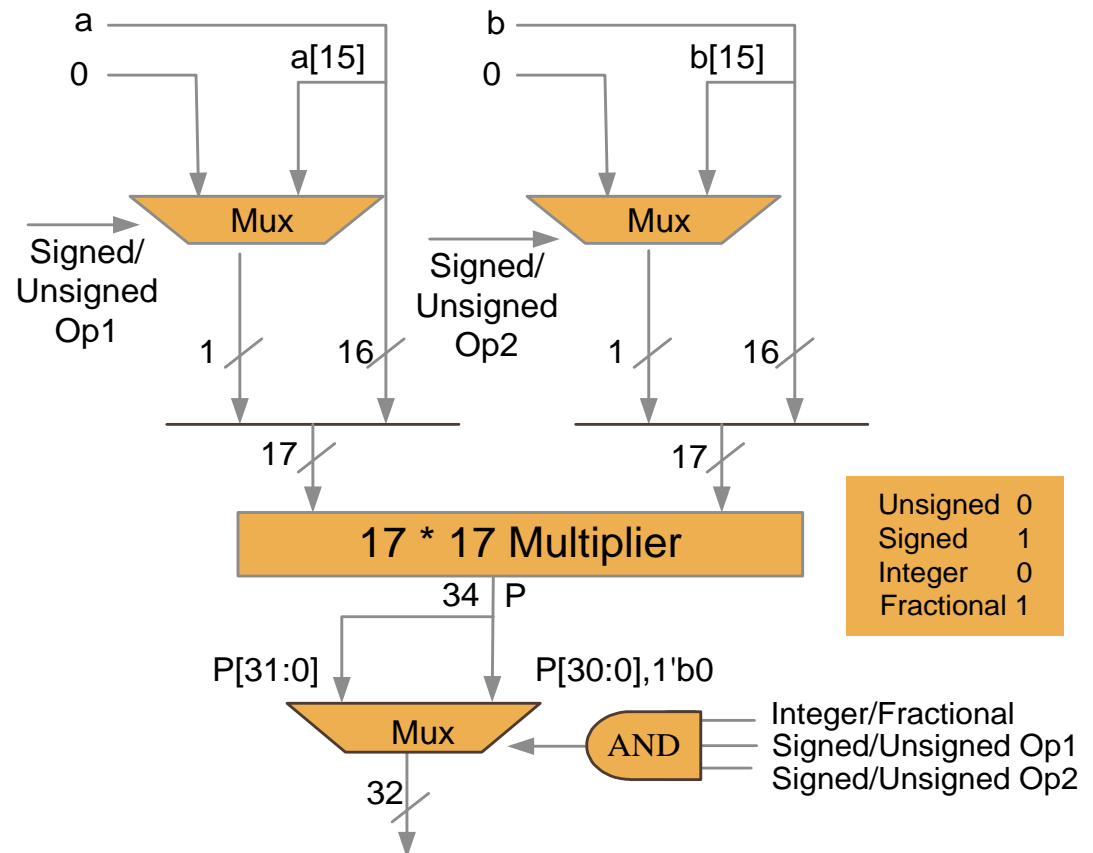
$$11.001111 = -0.765625_{10}$$

$$\begin{aligned} 11.001111 &\Rightarrow \text{Q1.3 format} \\ &\Rightarrow 1.001 = -0.875_{10} \end{aligned}$$

- 4x4 bit multiplication can easily be trimmed to get in 4 bit precision for fractional multiplication but for integer it may cause overflow

Unified Multiplier

- The multiplier supports integer and fractional multiplication
- All types of operands
 - SxS, SxU, UxS, UxU
- Checks the corner case of fractional multiplication



Bit Growth in Fixed-point Arithmetic

- Bit growth is one of the most critical issues in fixed-point arithmetic.
- Multiplication of an N-bit number with an M-bit number results in an (N+M)-bit number
 - In recursive system each iteration increases the width

$$y[n] = ay[n-1] + x[n]$$

Bit Growth in Q-format arithmetic

- Multiplication of two N_1 and N_2 bit numbers in $Q_{n_1.m_1}$ and $Q_{n_2.m_2}$ results in N_1+N_2 bit number
 - $Q_{(n_1+n_2-1).(m_1+m_2+1)}$ for SxS and $Q_{(n_1+n_2).(m_1+m_2)}$ for the rest
- Where as addition results is

$$Q_{n.m} = Q_{\max(n_1, n_2). \max(m_1, m_2)}$$
- For LTI system Schwarz's inequality is used for estimating the bit growth

$$|y_n| \leq \sqrt{\sum_{n=-\infty}^{\infty} h^2} \left| \sum_{n=-\infty}^{\infty} x^2 \right|$$

Truncation

- In multiplication of two Q format numbers as the number of bits in the product increases
- We sacrifice precision by throwing some low precision bits of the product
- $Q_{n1.m1}$ is truncated to $Q_{n1.m2}$ where $m2 < m1$

Let the product is

$8'b01110110$ ($Q_{4.4}$)

$$4 + 2 + 1 + 0.25 + 0.125 = 7.375$$

Truncate it to $Q_{4.2}$ results

$$6'b011101 = 7.25$$

Equivalent Q formats

- In many cases Q format of a number is to be changed

Convert $Q_{n1.m1}$ to $Q_{n2.m2}$

- If $n2 > n1$, we simply append **sign bits = $n2 - n1$** to the MSB location of **$n1$**
- If $m2 > m1$ we simply append zeros to the LSB locations of the Fractional part of **$m1$**

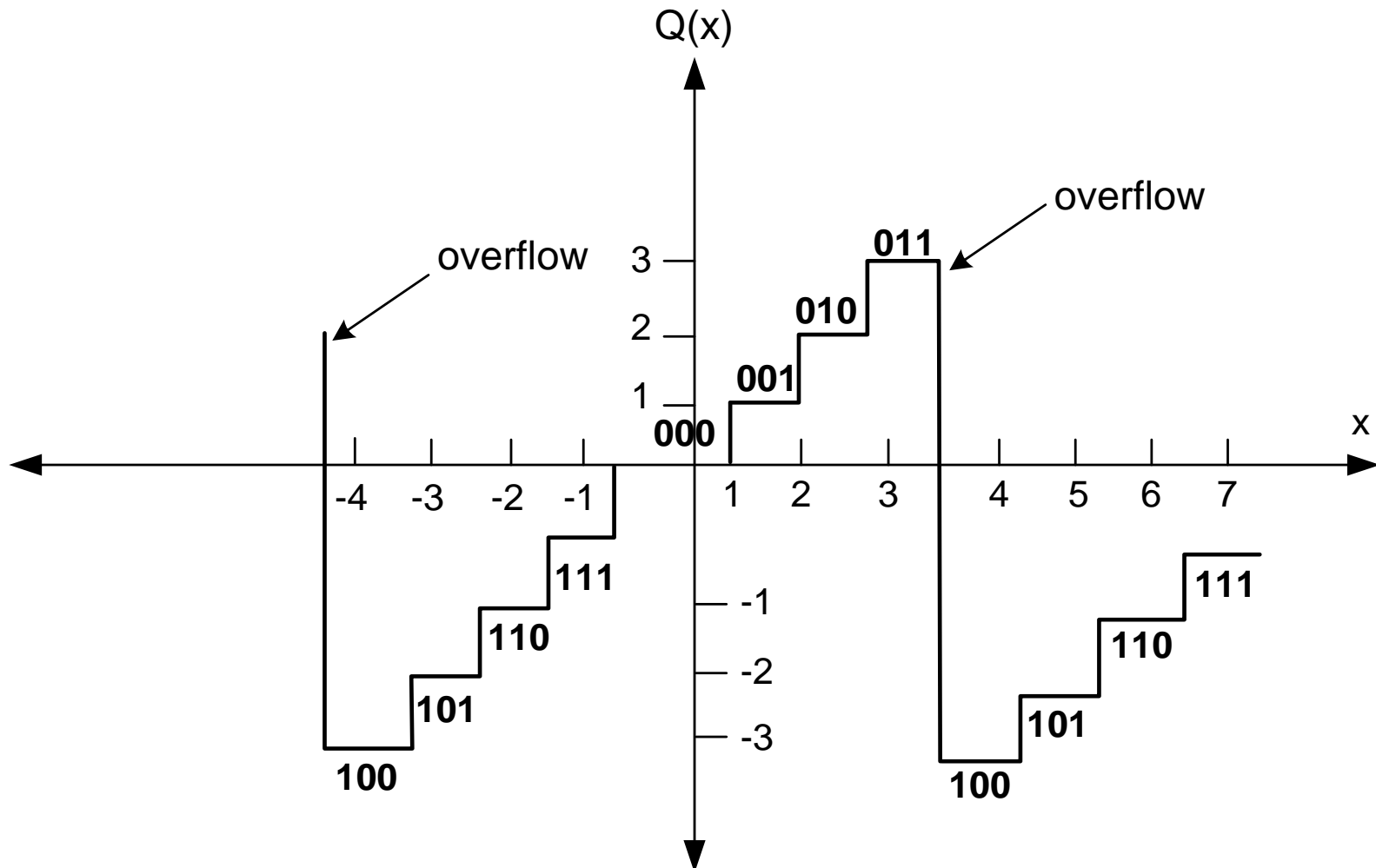
Example

- Let $a=11.101$ ($Q_{2.3}$)

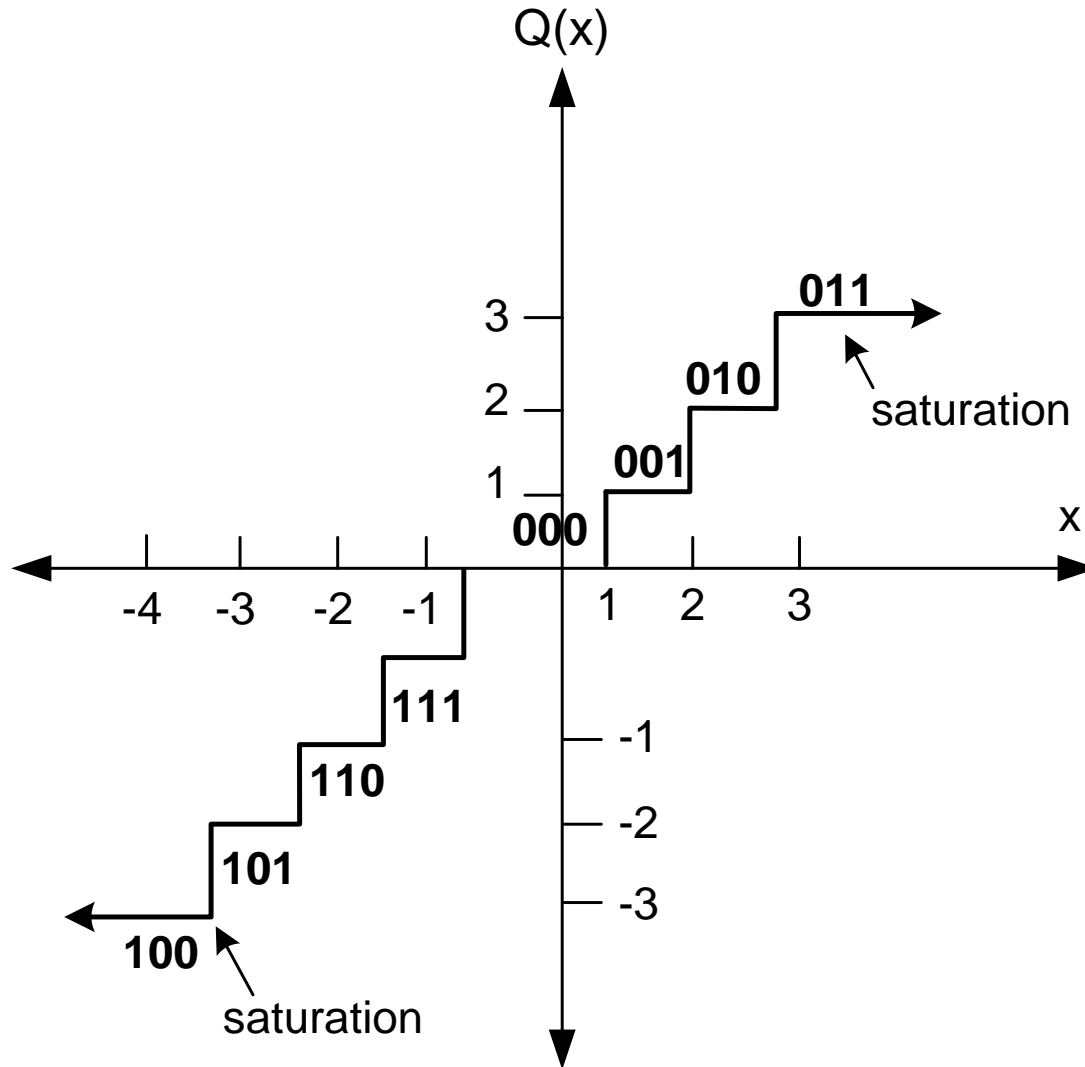
is supposed to be added to a number b of $Q_{7.7}$ format.
So extending a will result in:

1111111.1010000 ($Q_{7.7}$)

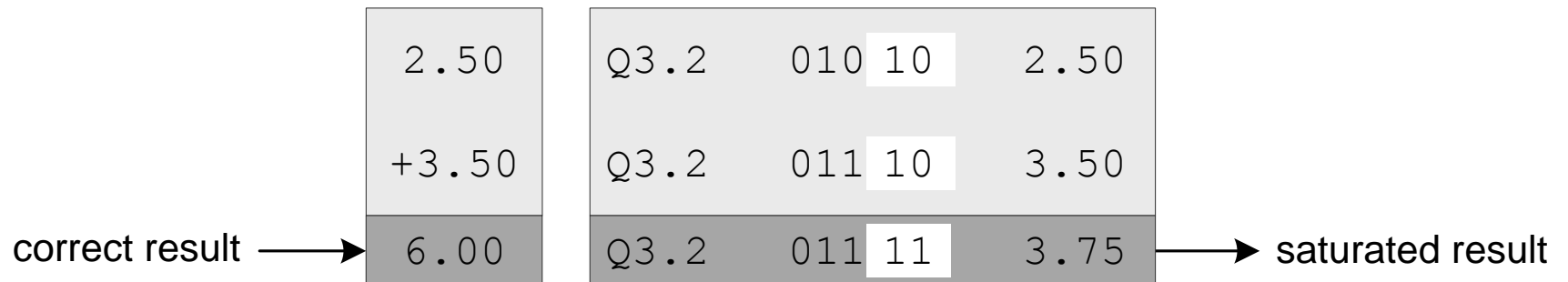
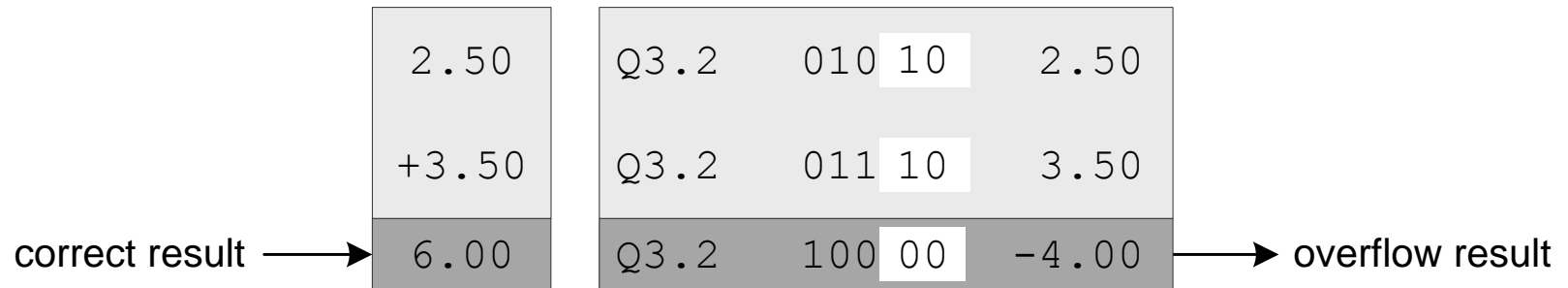
Overflow introduces an error equal to the dynamic range of the number



Saturation clamps the value to a maximum positive or minimum negative level



Overflow and Saturation



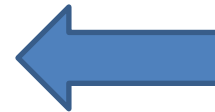
2's Complement Intermediate Overflow Property

In an iterative calculation using 2's complement arithmetic if it is guaranteed that the final result will be within precision bound of assigned fixed-point format then any number of intermediate overflows will not affect the final answer.

1.75
+1.25
3.00

Q2.2	0111	2.50
Q2.2	0101	3.50
Q2.2	1100	-1.00

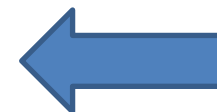
intermediate overflow



3.00
-1.25
1.75

Q2.2	1100	-1.00
Q2.2	1011	-1.25
Q2.2	0111	1.75

correct final answer



Adv Digital design

Example use of Intermediate Overflow property: CIC Filter

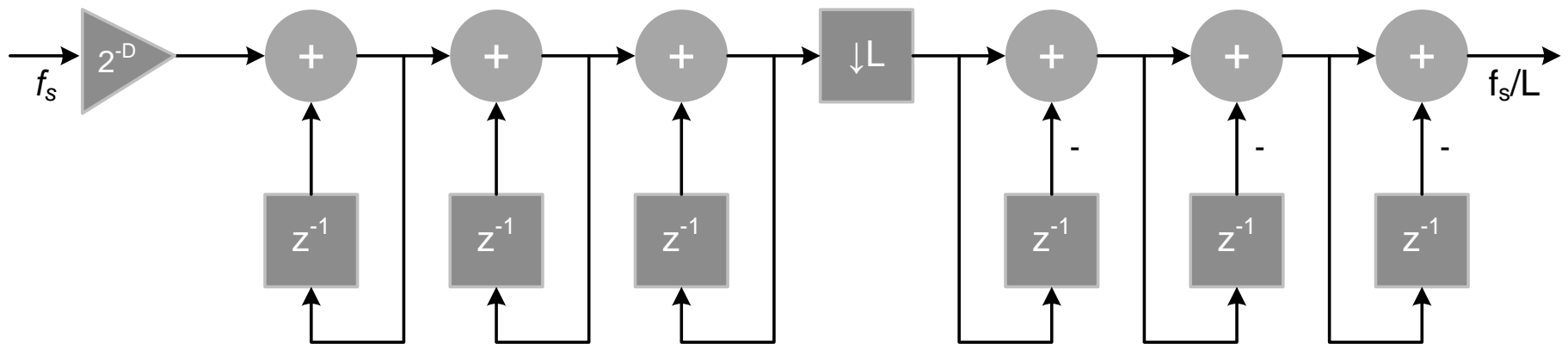
- The transfer function of a CIC filter is:

$$H(z) = 2^{-D} \left[\frac{1 - z^{-L}}{1 - z^{-L}} \right]^M$$

- For signed Qn.m input, the output fits in Qn.m format provided each input sample is scaled by D, where:

$$2^{-D} \leq \left(\frac{1}{L} \right)^M$$

An Mth-order CIC filter for decimating a signal by a factor of L (here, M=3)



Bit Growth in Q-format arithmetic

- For LTI system Schwarz's inequality is used for estimating the bit growth

$$\|y_n\| \leq \sqrt{\sum_{n=-\infty}^{\infty} h^2} \|x_n\|$$

FIR Filter Example

- For Q1.m format inputs the outputs will mostly fit in Q1.15 if the design can guarantee that the sum of all the coefficients of an FIR filter is 1; that is:

$$\sum_{n=0}^{L-1} h_n = 1$$

- All filters design in Matlab observe this property

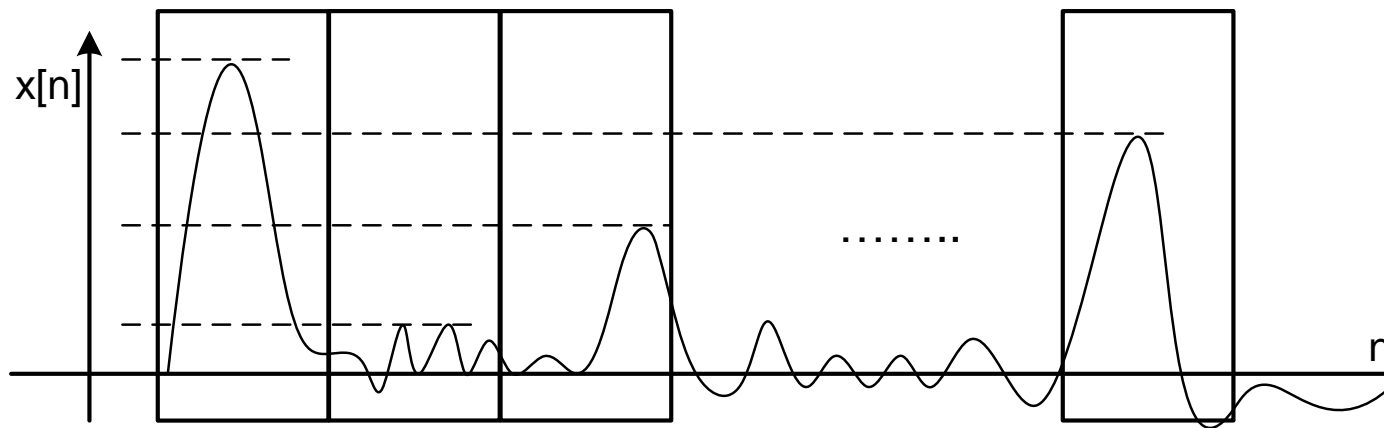
```
>> sum(fir1(20,.1))
```

```
ans = 1.0000
```

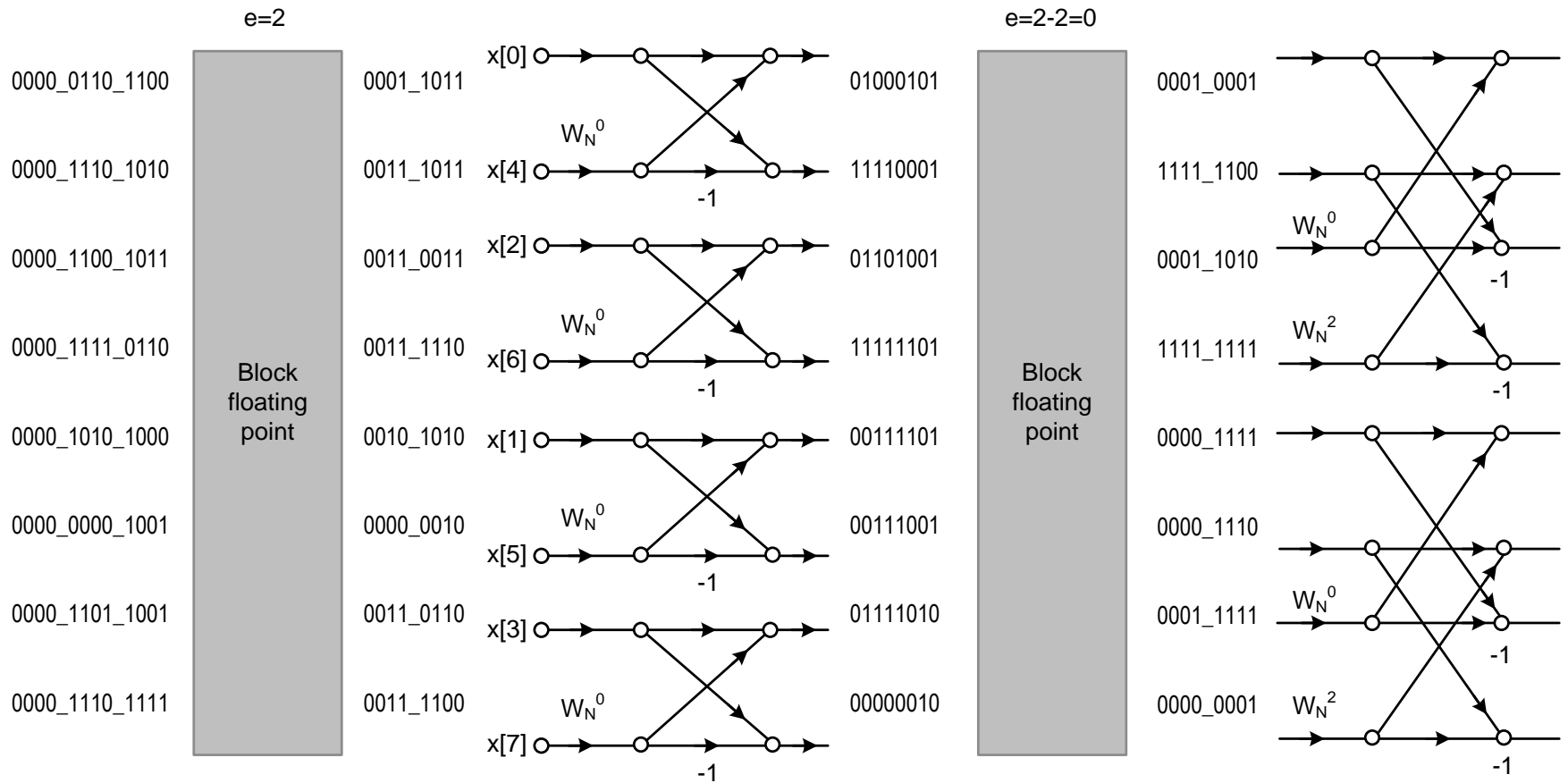
Block Floating Point

- Block floating-point format improves the dynamic range of numbers
- Useful in design where block of data goes through different stages of computation
- A block of data is represented with a common exponent
- All the values in the block are in fixed-point format.
- Each stage computes the number of redundant sign bits of all the values in the block

Block Floating Point



Example: Block Floating Point in FFT



Filter Structures: DF-I and DF-II

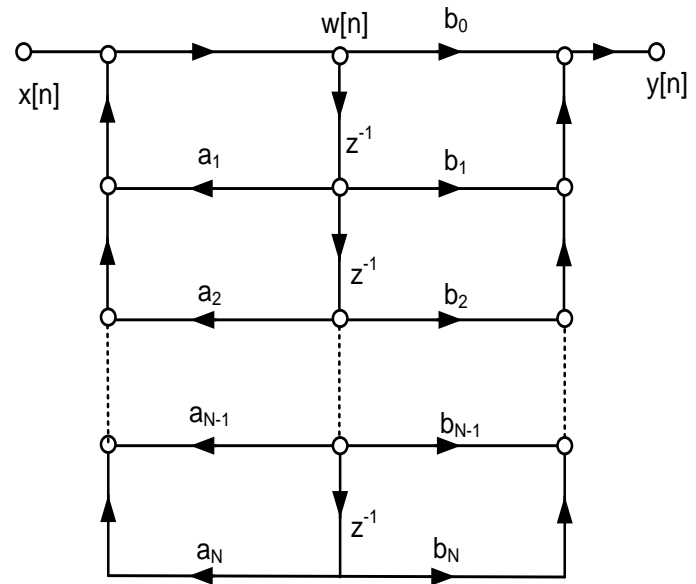
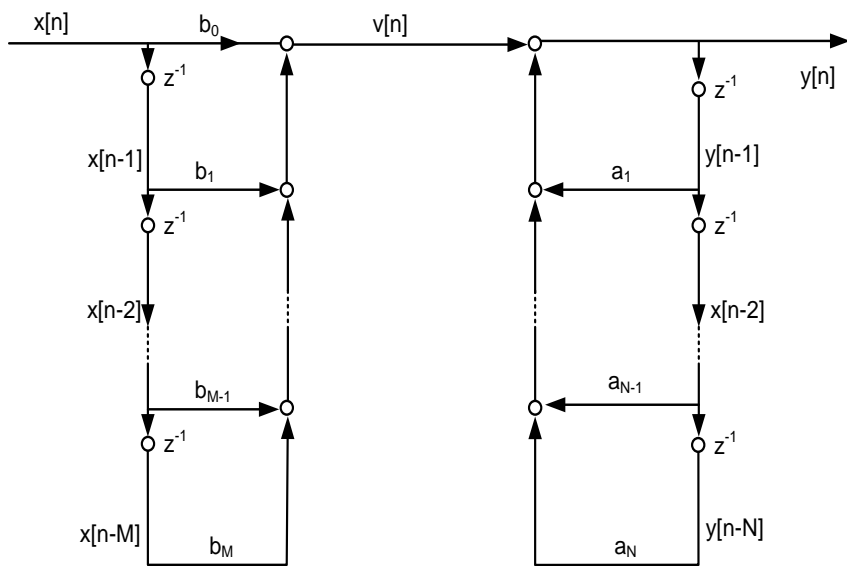
- Many ways to implement an IIR filter with

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^N b_k z^{-k}}{1 + \sum_{k=1}^M a_k z^{-k}}$$

$$y[n] = \sum_{k=0}^N b_k x[n-k] - \sum_{k=1}^M a_k y[n-k]$$

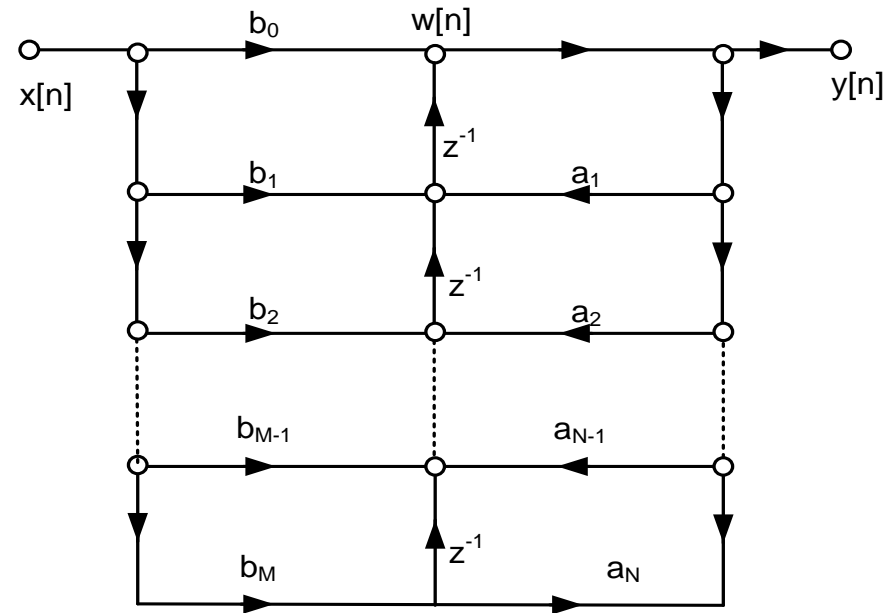
- All are analytical equivalent
- Direct Form-I and Direct form II or two implementations

Filter Structures: DF-I and DF-II



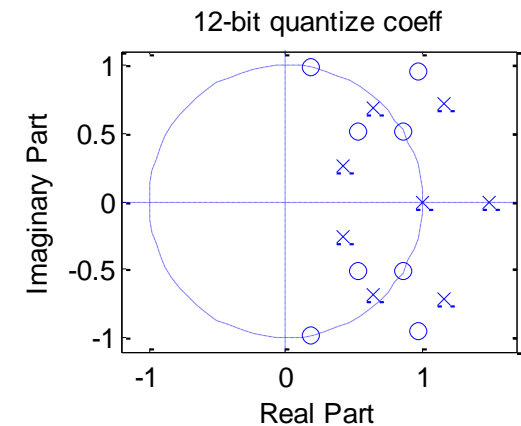
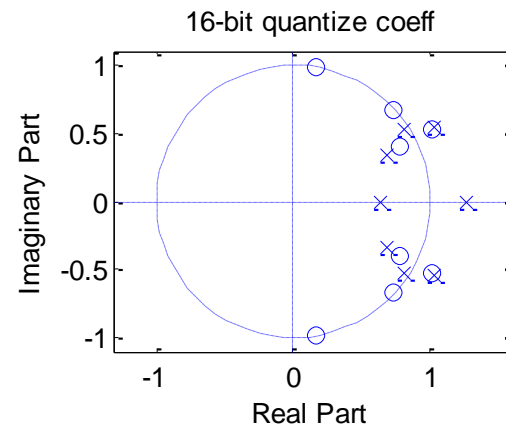
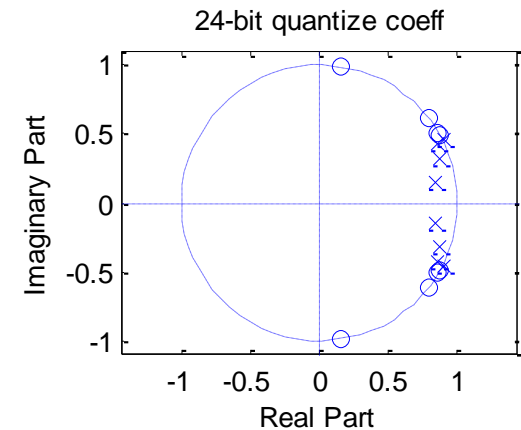
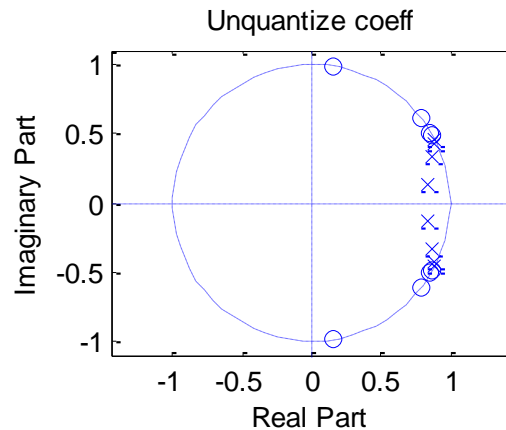
TDF-II

- Transposed Direct Form-II is another implementation that reduces the number of delays
- DF-I, DF-II, TDF-I and TDF-II all suffers from coefficient quantization
 - A filter designed using double precision may get unstable after quantization



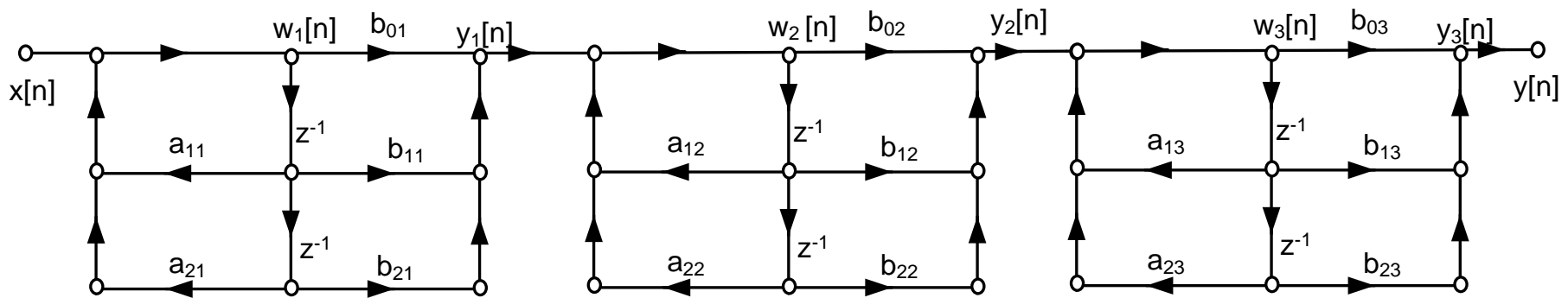
Direct Fixed-point Implementation

- A double precision filter gets unstable once quantized to even 24-bit precision
- Each coefficient quantization effects all poles
- Some of the poles may move outside the unit circle



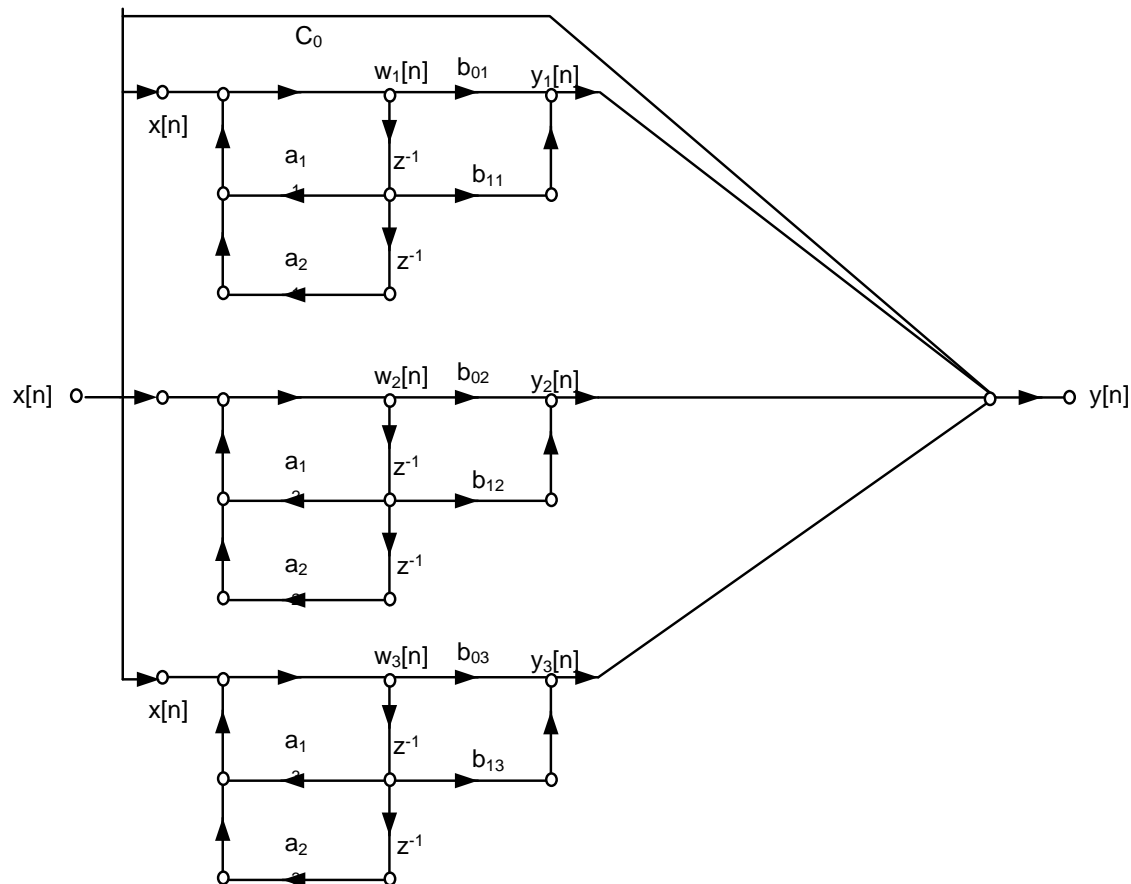
Second Order Cascaded Sections

- Conversion to Second Order Sections before quantization is highly desired
- Quantization of coefficient effects only the conjugate pole pair



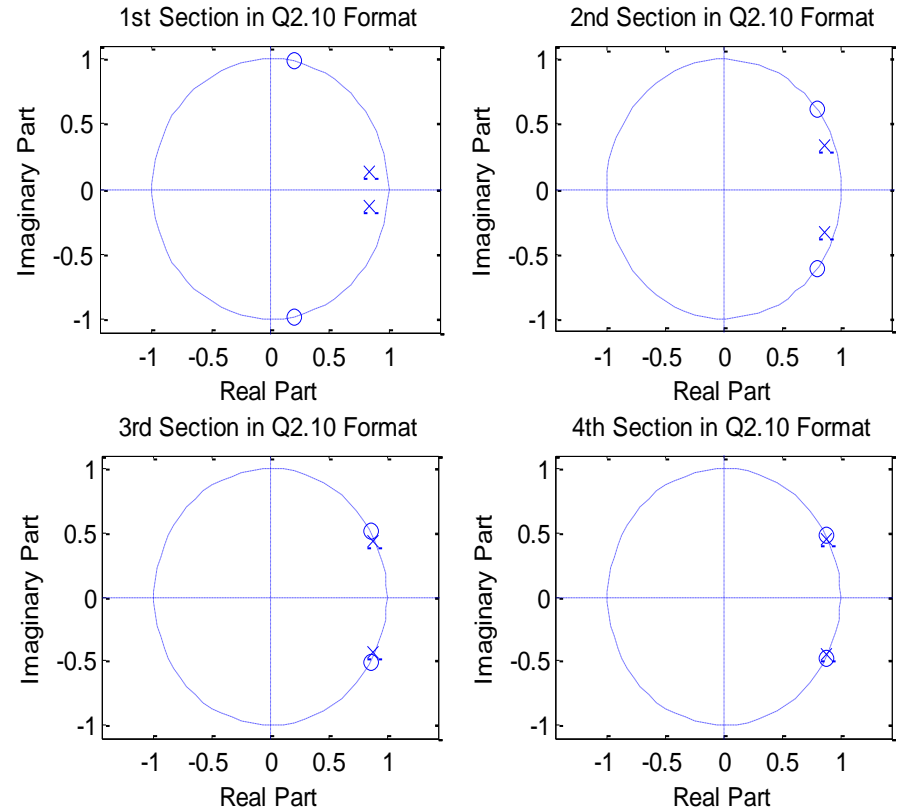
Parallel Implementation

- A parallel form also provides same benefits relating to stability



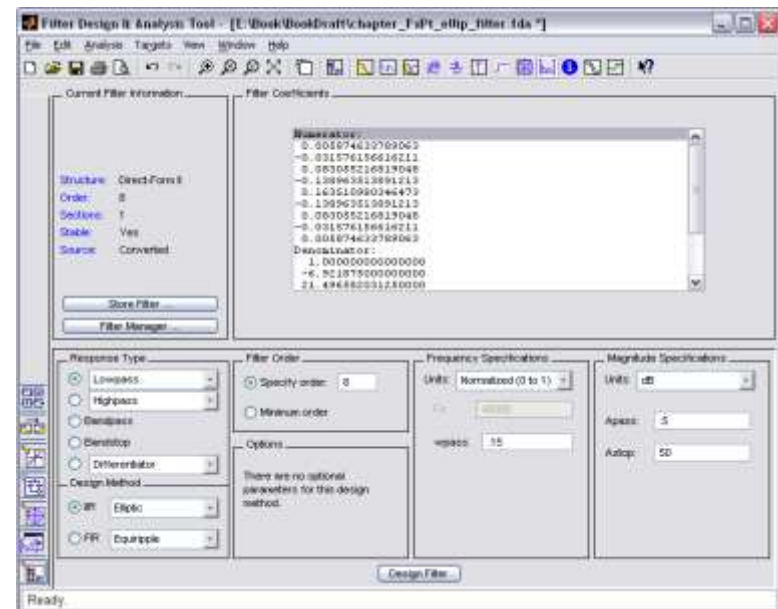
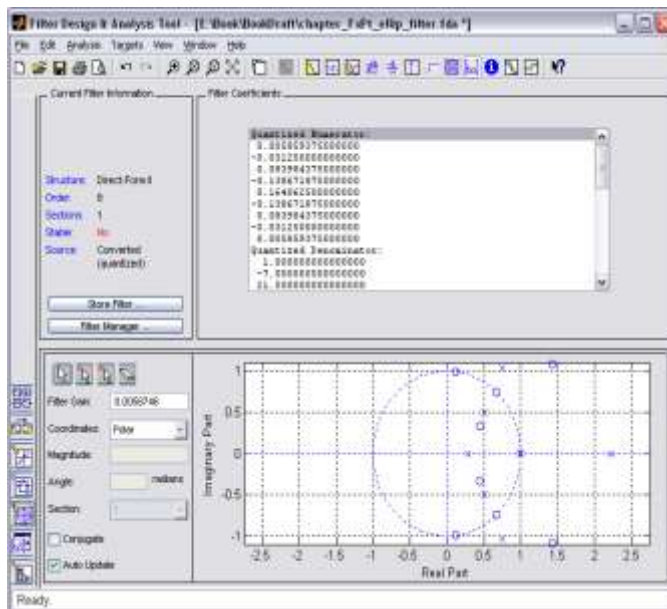
Second Order Sections

- The 8th order IIR filter is designed using 2nd order section
- The direct form results in instability at 24-bit precision
- The 2nd order is stable even at 8-bit precision



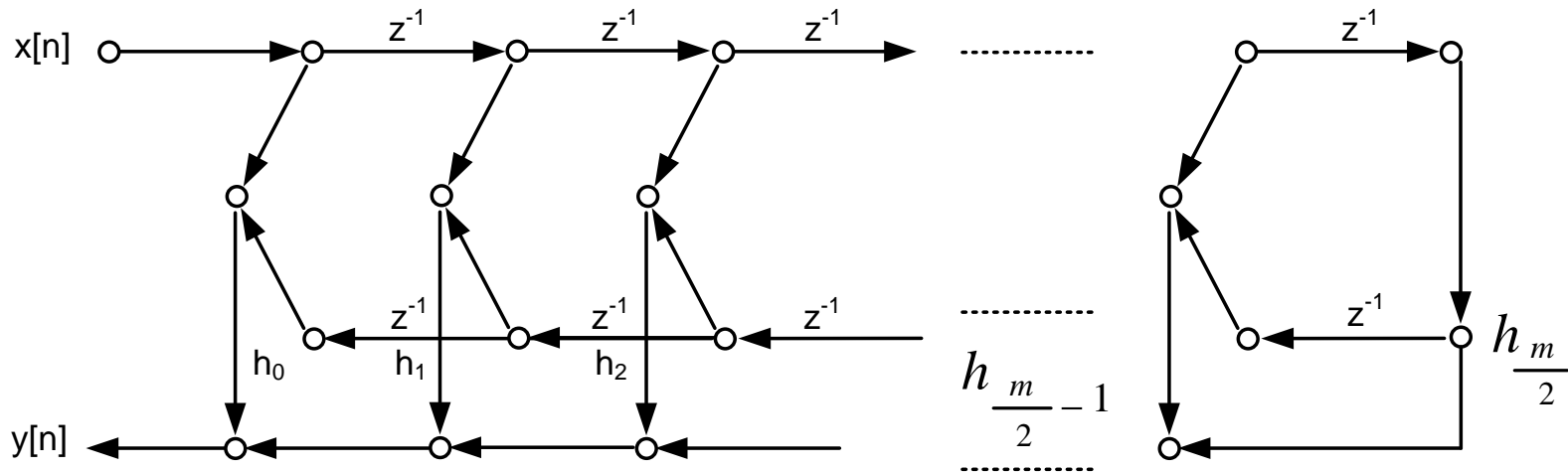
Matlab FDA Tool Box

- Matlab FDA tool box can be used to design and analyze floating and fixed point filter
- The tool can be used to port the coefficients in Matlab workspace, C/C++ header file or Verilog file
- The tool also generates fully parallel implementation in Verilog

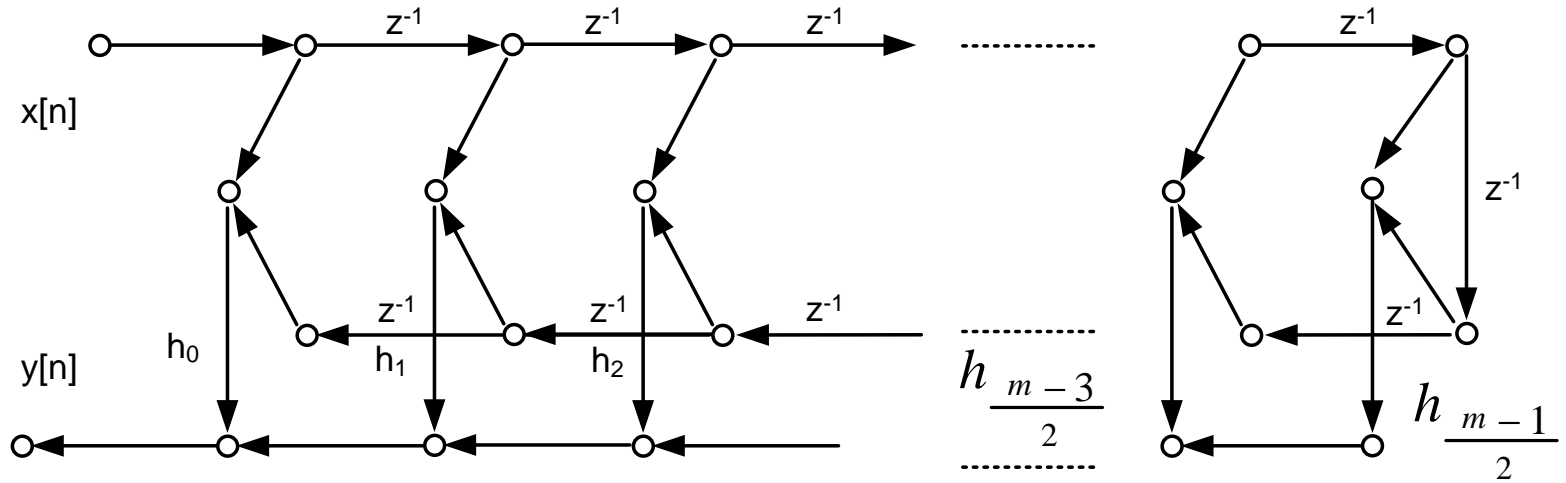


Folded Structures

- Linear phase FIR filter observes symmetry or anti-symmetry
- All LP filters can be folded for reducing the number of multipliers



Contd...



Summary

- Capturing Requirements & Specifications is the first step in system design
- A digital system usually consists of hybrid technologies and uses GPP, DSPs, FPGAs and ASICs
- The algorithms are developed in double precision floating point format
- Floating point HW and DSPs are expensive, for DSP applications Fixed point arithmetic is preferred
- The floating point code is converted to fixed point format and all variables are defined as Qn.m format numbers
- The fixed point arithmetic results in bit growth, the results from arithmetic computations are rounded and truncated
- In IIR filter implementation 2nd order and Parallel forms minimizes coefficient quantization
- Implementation of LP FIR should use symmetry and anti-symmetry of coefficients
- Block floating point can improve the precision for block processing of data