REAL-TIME DIGITAL SYSTEMS DESIGN AND VERIFICATION WITH FPGAS
ECE 387 – LECTURE 11

PROF. DAVID ZARETSKY

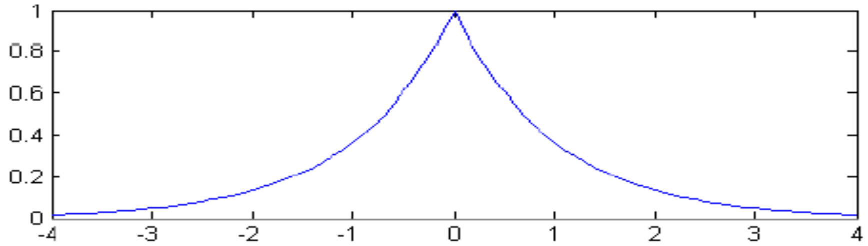
DAVID.ZARETSKY@NORTHWESTERN.EDU

AGENDA

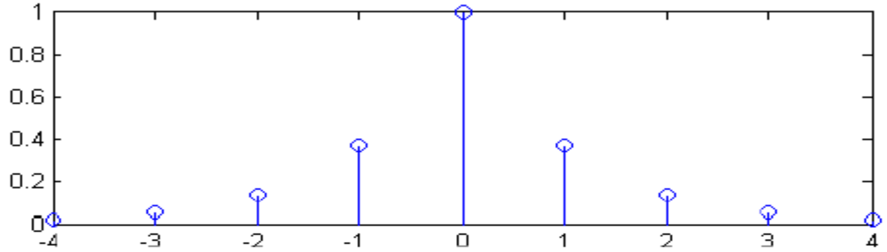
- Quantization

CONTINUOUS-TIME VS. DISCRETE-TIME

- Continuous-time signal: an analog signal defined by a function of a continuous-time variable.

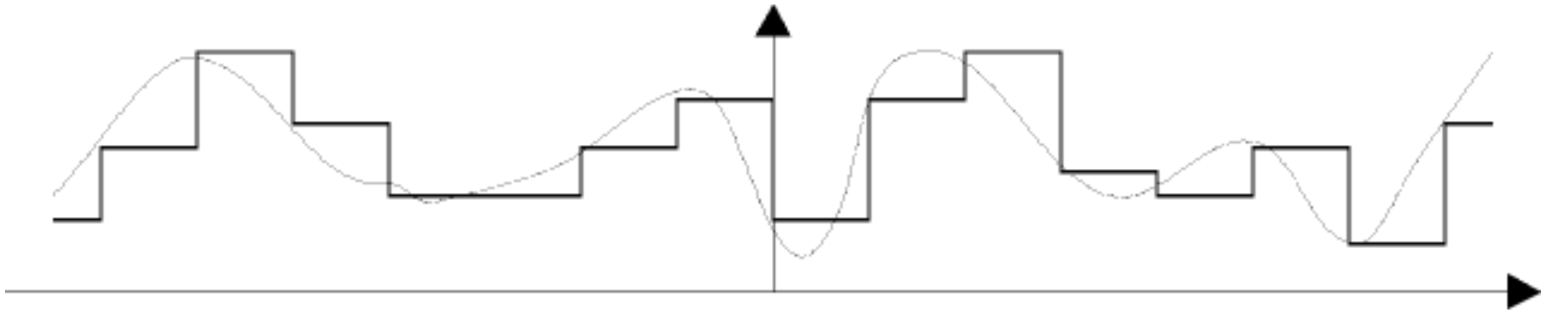


- Discrete-time signal: a signal defined by specifying the value of the signal only at discrete times, called sampling instants.



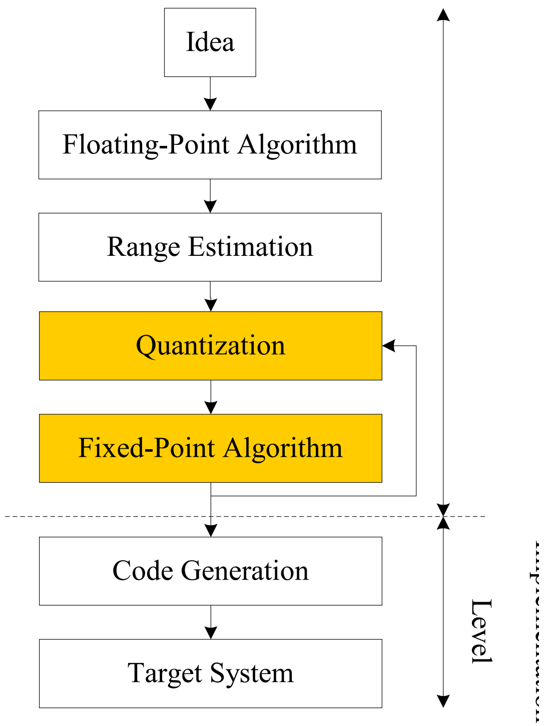
QUANTIZED SIGNALS

- A quantized signal is one whose values may assume only a countable number of values, or levels
- Changes from level to level may occur at any time.



FIXED-POINT DESIGN

- Digital signal processing algorithms
  - Often developed in floating point
  - Later mapped into fixed-point for digital hardware
- Fixed-point digital hardware
  - Lower area
  - Lower power
  - Lower per unit production cost
- Float-to-fixed point conversion
  - Required for ASIC and FPGA implementations
  - Avoid overflow
  - Minimize quantization effects
  - Find optimum wordlength

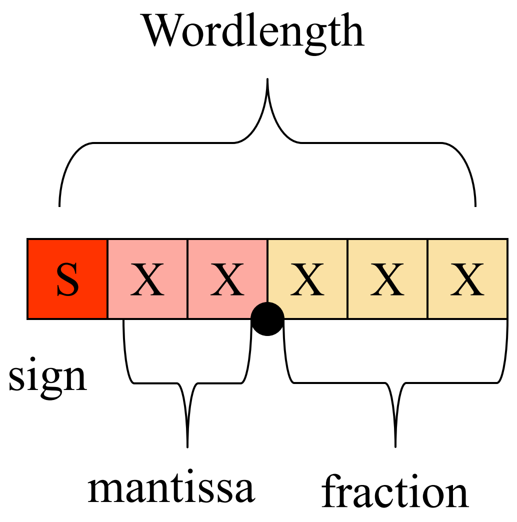


QUANTIZATION IN DSP APPLICATIONS

- Quantization is the mapping a large set of input values to a smaller set.
- Rounding and truncation are examples of quantization
- The difference between an input value and its quantized value (such as round-off error) is referred to as quantization error.
- Errors in digital signal applications
  - Quantization in A-D and A-D converters
  - Quantization of parameters
  - Round-off and overflow in addition, subtraction, multiplication, division, and other operations
- A-D and D-A converters often have poor resolution
  - A-D: 10–16 bits
  - D-A: 8–12 bits

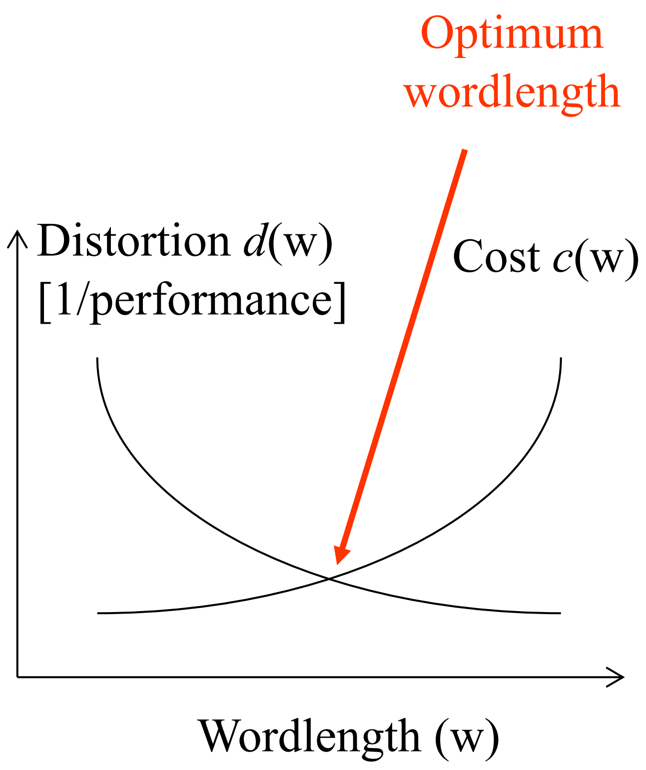
FIXED-POINT REPRESENTATION

- Fixed point Wordlength
  - Mantissa
  - Fraction
  - Sign
- Quantization modes
  - Round
  - Truncation
- Overflow modes
  - Saturation
  - Saturation to zero
  - Wrap-around



OPTIMUM WORDLENGTH

- Longer wordlength
  - May improve application performance
  - Increases hardware cost
- Shorter wordlength
  - May increase quantization errors and overflows
  - Reduces hardware cost
- Optimum wordlength
  - Maximize application performance or minimize quantization error
  - Minimize hardware cost

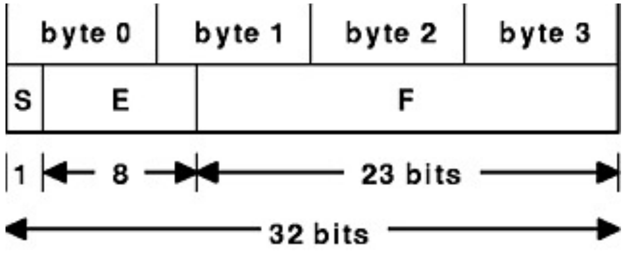


WORD LENGTH OPTIMIZATION APPROACH

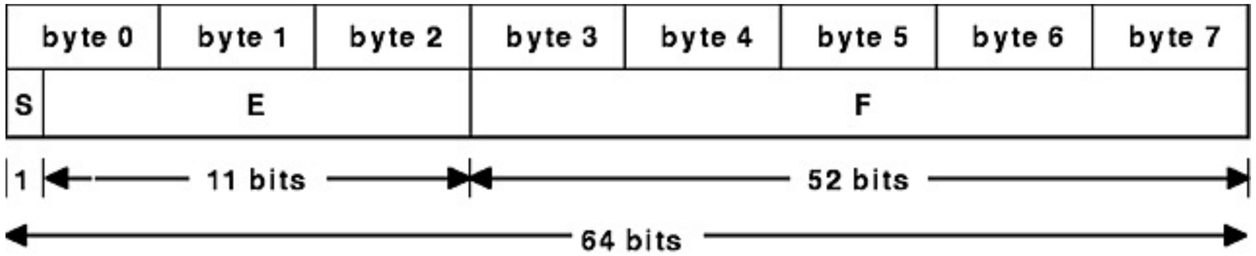
- Analytical approach
  - Quantization error model
  - For feedback systems, instability and limit cycles can occur
  - Difficult to develop analytical quantization error model of adaptive or non-linear systems
- Simulation-based approach
  - Word Lengths chosen while observing error criteria
  - Repeated until word lengths converge
  - Long simulation time

IEEE FLOATING POINT STANDARD

- Single precision (Java/C float):
  - 32-bit word divided into
    - 1 sign bit
    - 8-bit biased exponent
    - 23-bit mantissa (7 decimal digits)
  - Range: $2^{-126} - 2^{128}$
- Double precision (Java/C double):
  - 64-bit word divided into
    - 1 sign bit
    - 11-bit biased exponent
    - 52-bit mantissa (15 decimal digits)
  - Range: $2^{-1022} - 2^{1024}$



Single-Precision Floating Point



Double-Precision Floating Point

FLOATING POINT EXAMPLE

- What's the result of this function?

```
void main()
{
     float a[] = { 10000.0, 1.0, 10000.0 };
     float b[] = { 10000.0, 1.0, -10000.0 };
     float sum = 0.0;
     for (int i=0; i<3; i++)
          sum += a[i] * b[i];
     printf("sum = %f\n", sum)
}
```

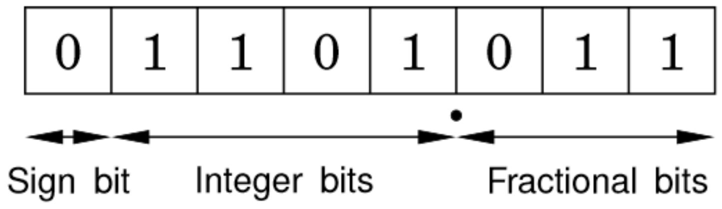
- Remarks:
  - The result depends on the order of the operations
  - Finite-wordlength operations are neither associative nor distributive

FLOATING POINT IN FPGAS

- FPGAs have limited support for floating-point arithmetic
  - Hardware floating point libraries – fast, costly, large area
  - Software emulation of floating-point arithmetic – slow, large area
  - Fixed-point arithmetic – fast, compact
- Challenges in Fixed-Point conversion
  - Must select data types to get sufficient numerical precision
  - Must know (or estimate) the minimum and maximum value of every variable in order to select appropriate scaling factors
  - Must keep track of the scaling factors in all arithmetic operations
  - Must handle potential arithmetic overflow

FIXED-POINT ARITHMETIC

- Fixed Point: represent all numbers using integers
- Use binary scaling to make all numbers fit into one of the integer data types
  - 8 bits (char): [−128, 127]
  - 16 bits (short): [−32768, 32767]
  - 32 bits (long): [−2147483648, 2147483647]
- In fixed-point representation, a real number x is represented by an integer X with N = m+n+1 bits, where
  - N is the wordlength
  - m is the number of integer bits (excluding the sign bit)
  - n is the number of fractional bits

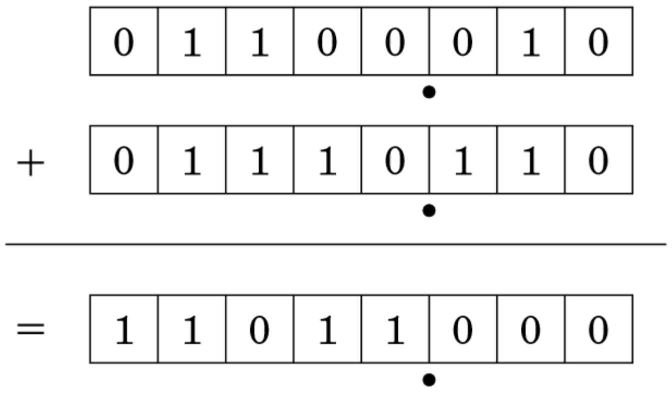


CONVERSION TO/FROM FIXED POINT

- Conversion from real to fixed-point number:
- $X := \mathrm{round}(x \cdot 2^n)$
- Conversion from fixed-point to real number:
- $x := X \cdot 2^{-n}$
- Example: Represent x = 13.4 using Q4.3 format
- $X = \mathrm{round}\ (13.4 \cdot 2^3) = 107\ (= 01101011_2\ )$

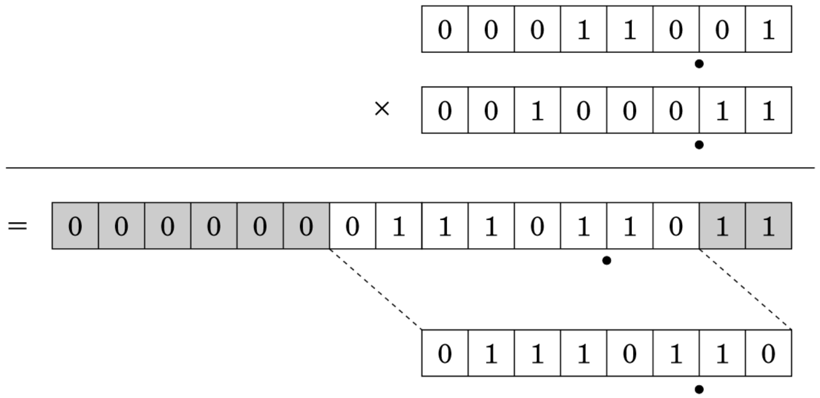
FIXED POINT ADDITION & SUBTRACTION

- Two fixed-point numbers in the same Qm.n format can be added or subtracted directly
- The result will have the same number of fractional bits
  - $z = x + y \quad \Leftrightarrow Z = X + Y$
  - $z = x - y \quad \Leftrightarrow Z = X - Y$
- The result will in general require N + 1 bits; risk of overflow
- Example: Addition with Overflow
  - Two numbers in Q4.3 format are added:
    - x = 12.25 => X = 98
    - y = 14.75 => Y = 118
    - Z = X + Y = 216 (11011000_2)
  - This number is however out of range and will be interpreted as
  - 216 - 256 = -40 => z = -5 (11111011_2)



FIXED POINT MULTIPLICATION & DIVISION

- If the operands and result have same Q-format, then
  - $z = x \cdot y \;\Leftrightarrow\; Z = (X \cdot Y) / 2^n$
  - $z = x / y \Leftrightarrow Z = (X \cdot 2^n) / Y$
- Double word length is needed for the intermediate result
- Unsigned Division by 2^n is implemented as a right-shift by n-bits
- Multiplication by 2^n is implemented as a left-shift by n-bits
- The lowest bits in the result are truncated (round-off noise); Risk of overflow
- Example: Two numbers in Q5.2 format are multiplied:
  - x = 6.25 => X = 25
  - y = 4.75 => Y = 19
  - Z = X * Y = 25 * 19 = 475
  - Z = 475 / 2^2 = 118 => z = 29.5
    (exact result is 29.6875)



MULTIPLICATION EXAMPLE

- Truncation

```
#define n 3              /* number of fractional bits */
char X, Y, Z;            /* Q4.3 operands and result */
short temp;              /* Q9.6 intermediate result */
temp = (short)X * Y;     /* cast operands to 16 bits and multiply */
temp = temp >> n;        /* divide by 2^n */
Z = temp;                /* truncate and assign result *
```

- Rounding & Saturation

```
temp = (short)X * Y;        /* cast operands to 16 bits & multiply */
temp = temp + (1 << n-1);   /* add 1/2 to give correct rounding */
temp = temp >> n;           /* divide by 2^n */
if (temp > INT8_MAX)        /* saturate result before assignment */
   Z = INT8_MAX;
else if (temp < INT8_MIN)
   Z = INT8_MIN;
else
   Z = temp;
```

DIVISION EXAMPLE

```
#define n 3        /* number of fractional bits */
char X, Y, Z;      /* Q4.3 operands and result */
short temp;        /* Q9.6 intermediate result */
...
temp = (short)X << n;     /* cast operand to 16 bits and shift */
temp = temp + (Y >> 1);   /* Add Y/2 to give correct rounding */
temp = temp / Y;          /* Perform the division (expensive!) */
Z = temp;                 /* Truncate and assign result *
```

QUANTIZATION IN SOFTWARE

```
#define BITS              10
#define QUANT_VAL         (1 << BITS)
#define QUANTIZE_F(f)     (int)(((float)(f) * (float)QUANT_VAL))
#define QUANTIZE_I(i)     (int)((int)(i) * (int)QUANT_VAL)
#define DEQUANTIZE(i)     (int)((int)(i) / (int)QUANT_VAL)

const int quad = QUANTIZE_F( PI / 4.0 );
int z = QUANTIZE_I( x ) / y;
int v = DEQUANTIZE( z * quad );
```

EXAMPLE QUANTIZED FUNCTION

- If we two numbers x and y in 8.8 format, then multiplying them will yield a 16.16 result.
- So, if we want a 8.8 result, we need to shift it right 8 bits.
- DEQUANTIZE ensures that the bits do not overflow when multiplying large numbers.

```
#define DEQUANTIZE(i)   (int)((int)(i) / (int)QUANT_VAL)

void multiply_n( int *x_in, int *y_in, const int n_samples, int *output )
{
    for ( int i = 0; i < n_samples; i++ )
    {
        output[i] = DEQUANTIZE( x_in[i] * y_in[i] );
    }
}
```

NEXT…

- HW #6: Cordic