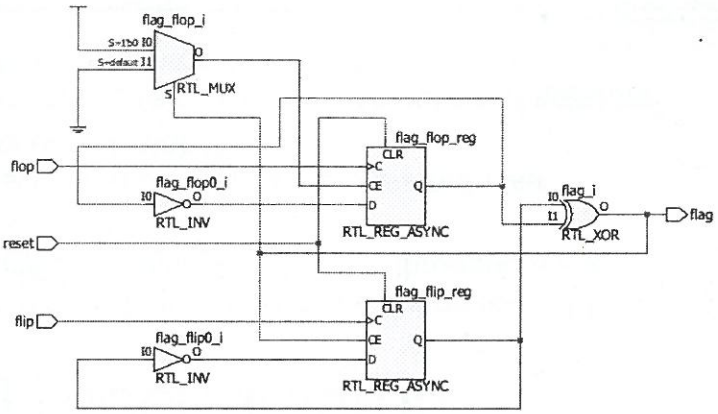
Toggling a Flag with Multiple Clocks

Standard D-type Flip-Flops do not support more than a single clonk. But in practice, there are cases where we need to change a flag using two independent clocks.

Example: handshaking mechanisms

```
1  module handshake (
2          input flip,
3          input flop,
4          input reset,
5          output flag
6      );
7
8      // initialize by flag = 0
9      reg flag_flip = 1'b0;
10     reg flag_flop = 1'b0;
11
12     assign flag = flag_flip ^ flag_flop;
13
14     always @(posedge flip or posedge reset) begin
15         if(reset)
16             flag_flip <= 0;
17         else if(~flag)
18             flag_flip <= ~flag_flip;
19     end
20
21     always @(posedge flop or posedge reset) begin
22         if(reset)
23             flag_flop <= 0;
24         else if(flag)
25             flag_flop <= ~flag_flop;
26     end
27 endmodule
```



Clock Speed Reduction

Apart from DCMs, various methods exist for clock speed reduction, including:

```
1  reg clockhalf = 1'b0;
2  always @(posedge clock)
3      clockhalf <= ~clockhalf;
4
5  always @(posedge clockhalf)begin
6      //...
7  end
```

Gated-Clock; not recommended nor supported on most FPGA devices

```
1  reg flag = 1'b0;
2  always @(posedge clock)
3      flag <= ~flag;
4
5  always @(posedge clock)
6      if(flag) begin
7          //...
8      end
```

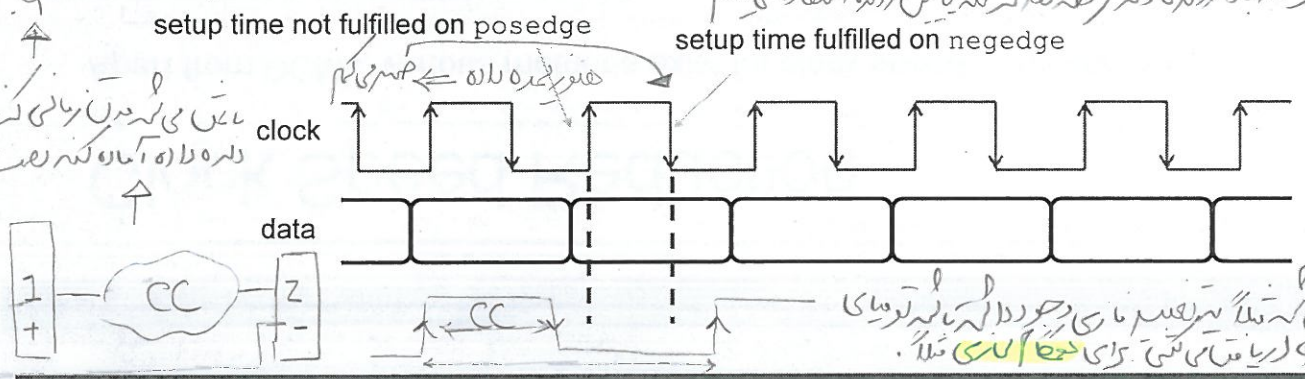
Standard method for clock halving using FF clock enable

```
1  parameter DIVIDE = 3'd6;
2  parameter DIVIDE_MINUS_ONE = MAX - 1'd1;
3  reg ce = 1'b0;
4  reg [2:0] counter = 3'd0;
5  always @(posedge clock)
6      if(counter == DIVIDE_MINUS_ONE)begin
7          counter <= 3'd0;
8          ce <= 1'b1;
9      end
10     else begin
11         counter <= counter + 3'd1;
12         ce <= 1'b0;
13     end
14
15 always @(posedge clock)
16     if(ce) begin
17         //...
18     end
```

Standard method for clock division using FF clock enable

Mixed Clock-Edge Design

- It is possible to use both positive and negative clock edges in a single design; but it should be avoided as much as possible
- Using mixed clock-edges does not double the clock rate; but it rather reduces the time for combination logic result settlements
- Utilization of mixed clock-edges should be confined to phase compensation between two signals when setup or hold-times are not fulfilled using a single edge (commonly at FPGA I/O)
- Example:



Standard Resetting Mechanisms

- Although both synchronous and asynchronous reset mechanisms are supported in FPGA designs, it is highly recommended to use a unified resetting mechanism throughout the entire design
- Synchronous resets with sufficient flip-flop synchronizer stages are preferred over asynchronous resets (due to lower probability of metastability)
- Even if the original reset command is asynchronous (e.g. using a push-button or software command), it is good practice to make an internal synchronous reset flag

```
always @(posedge clock or posedge reset)
    if(reset)begin // Asynchronous Reset
        //...
    end
    else begin
        //...
    end
```

Supported asynchronous reset mechanism

```
always @(posedge clock)
    if(reset)begin // Synchronous Reset
        //...
    end
    else begin
        //...
    end
```

Preferred synchronous reset mechanism

```
always @(posedge clock)begin // synchronizer Flip-Flops
    temp <= asynchreset;
    synchreset <= temp;
end

always @(posedge clock)
    if(synchreset)begin // Synchronous Reset
        //...
    end
    else begin
        //...
    end
```

Generating synchronous from asynchronous reset flag

Increasing Fan-out by HW Replication

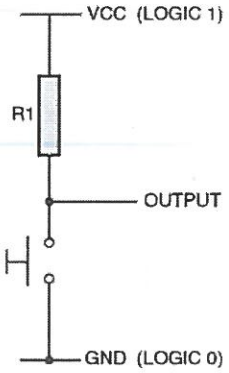
- The maximum fan-out of a logic circuit output is the maximum number of gate inputs it can drive without loading effects disturb its function (switching speed and voltage level)
- In contemporary FPGAs, Flip-Flop fan-outs are very high (several hundreds) and only the most frequently used signals (such as CLOCK, RESET, CE, …) may face fan-out issues
- The fan-out of a logic circuit may be increased by user constraints or hardware replication in HDL

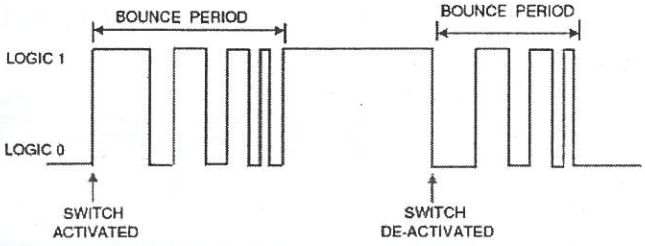
synthesis attributes to avoid register optimization and merging of ce1 and ce2

```
1  parameter DIVIDE = 3'd6;
2  parameter DIVIDE_MINUS_ONE = MAX - 1'd1;
3  (* dont_touch = "true" *) reg ce1 = 1'b0;
4  (* dont_touch = "true" *) reg ce2 = 1'b0;
5  reg [2:0] counter = 3'd0;
6  always @(posedge clock)
7      if(counter == DIVIDE_MINUS_ONE)begin
8          counter <= 3'd0;
9          ce1 <= 1'b1;
10         ce2 <= 1'b1;
11     end
12     else begin
13         counter <= counter + 3'd1;
14         ce1 <= 1'b0;
15         ce2 <= 1'b0;
16     end
17
18 always @(posedge clock) if(ce1) begin
19         // First instance of usage
20     end
21
22 always @(posedge clock) if(ce2) begin
23         // Second instance of usage
24     end
```


Debouncing

- In digital designs, bouncing (between 0 and 1) occurs during manual switch transitions
- The objective of debouncing is to avoid the mis-detection or multiple counting of events during switch transitions
- Debouncing can be implemented both in hardware (analog) and software (digital)

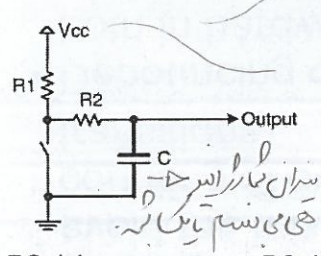




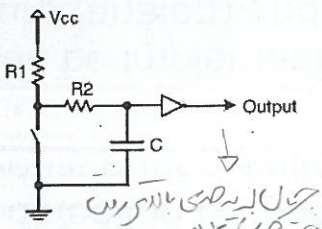
Reference: Arora, M. (2011). *The art of hardware architecture: Design methods and techniques for digital circuits*. Springer Science & Business Media, Chapter 8

Hardware Debouncing Techniques

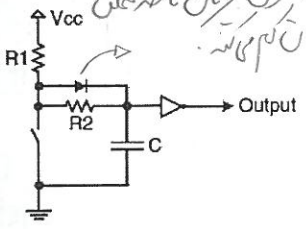
- Various hardware debouncing mechanisms:



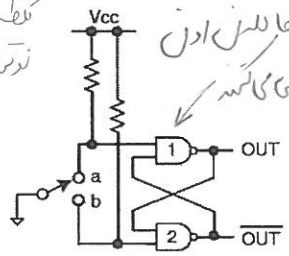
RC debouncer



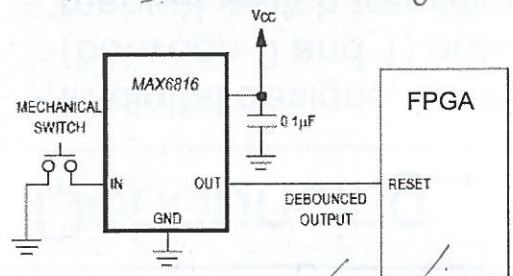
RC debouncer with digital logic



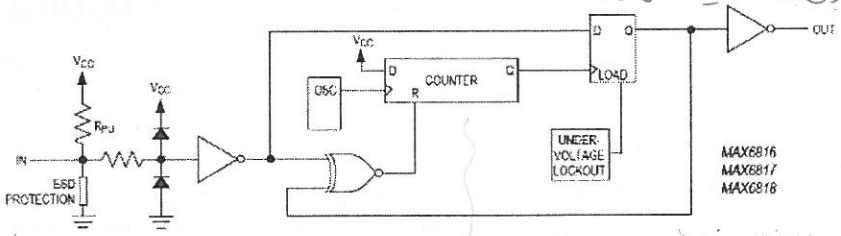
Robust RC debouncer with digital logic



SR debouncer



IC debouncer



MAX6816's internal mechanism

Software Debouncing Techniques

- Software debouncing mechanisms:

```
1  ; Intrupt Service Routine Pseudo-Code
2  DR: PUSH PSW ; SAVE PROGRAM STATUS WORD
3  LOOP: CALL DELAY ; WAIT A FIXED TIME PERIOD
4  IN SWITCH ; READ SWITCH
5  CMP ACTIVE ; IS IT STILL ACTIVATED?
6  JT LOOP ; IF TRUE, JUMP BACK
7  CALL DELAY ;
8  POP PSW ; RESTORE PROGRAM STATUS
9  EI ; RE-ENABLE INTERRUPTS
10 RETI ; RETURN BACK TO MAIN PROGRAM
```

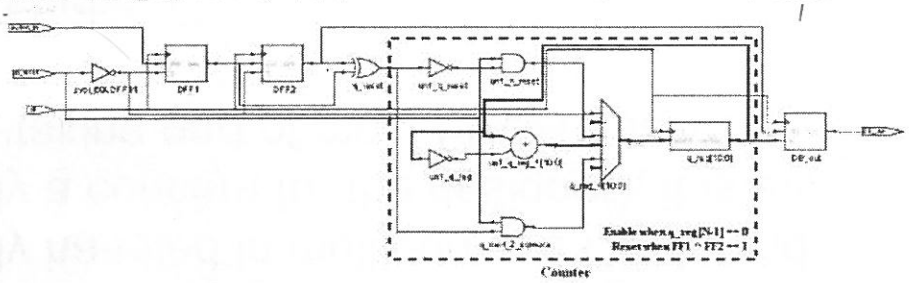
ISR assembly language debouncer pseudo-code

```
1  // Variable used to count:
2  unsigned char counter;
3  // Variable used as the minimum duration of a valid pulse:
4  unsigned char T_valid;
5  void main(){
6      P1 = 255;        // Initialise port 1 as input port
7      T_valid = 100;   /* Arbitrary number from 0 to 255 where
8                          the pulse if validated */
9      while(1){ // infinite loop
10         if (counter < 255){ // prevent the counter to roll
11             // back to 0
12             counter++;
13         }
14         if (P1_0 == 1){
15             counter = 0; // reset the counter back to 0
16         }
17         if (counter > T_valid){
18             // Code to be executed when a valid
19             // pulse is detected.
20         }
21         // Rest of you program goes here.
22     }
23 }
```

C language debouncer pseudo-code

HDL Debouncing Techniques (FSM)

```
1  module DeBounce(input clk, n_reset, button_in, output reg DB_out);
2      // internal constant
3      parameter N = 11;
4      reg  [N-1 : 0]  q_reg; // timing regs
5      reg  [N-1 : 0]  q_next;
6      reg DFF1, DFF2; // input flip-flops
7      wire q_add; // control flags
8      wire q_reset;
9      assign q_reset = (DFF1  ^ DFF2); // xor input FFs to look for level chage to reset counter
10     assign  q_add = ~(q_reg[N-1]); // add to counter when q_reg msb is equal to 0
11     // combo counter to manage q_next
12     always @ ( q_reset, q_add, q_reg)
13         case( {q_reset , q_add})
14             2'b00 : q_next <= q_reg;
15             2'b01 : q_next <= q_reg + 1;
16             default : q_next <= { N {1'b0} }
17         endcase
18     // Flip flop inputs and q_reg update
19     always @ ( posedge clk )
20         if(n_reset ==  1'b0) begin
21             DFF1 <= 1'b0;
22             DFF2 <= 1'b0;
23             q_reg <= { N {1'b0} };
24         end
25         else begin
26             DFF1 <= button_in;
27             DFF2 <= DFF1;
28             q_reg <= q_next;
29         end
30     // counter control
31     always @ ( posedge clk )
32         if(q_reg[N-1] == 1'b1)
33             DB_out <= DFF2;
34         else
35             DB_out <= DB_out;
36 endmodule
```



Ref: https://eewiki.net/pages/viewpage.action?pageId=13599139

PART III

Advanced Topics in Digital Design and Implementation

NUMBER REPRESENTATION

Number Representation in PLD Systems

- While number representation is fully standardized and rather automatically handled in multipurpose CPUs and GPUs (and is rarely a concern for the designer), it is an essential and time-taking part of most FPGA-based designs.
- In this section, we study:
  - The most common number representation standards
  - Fixed-point representation issues
  - Statistical analysis of truncation and rounding errors during data acquisition (using analog-to-digital converters) and calculations

An Overview of Binary Number Representation

- For many reasons radix-2 has remained the dominant number representation in digital hardware design:
  - In early technologies: the difficulty of generating high-speed switching logic circuits with more than two distinct and distinguishable levels of voltages.
  - In current technologies: besides the simplicity of radix-2, the huge body of literature, algorithms, codes, hardware (transistors, gates, etc.), and engineering experience and conventions, which already exist for radix-2 calculations makes it too expensive to migrate to higher radixes.

Binary Number Representation

Number representation can be studied from various aspects, including:

- Numbers of Interest:
  - Integers
  - Reals
- Sign Representation:
  - Unsigned
  - Signed
- Fractional Number Representation:
  - Fixed-point
  - Floating-point

Accuracy of Finite Length Binary Number Representations

Question: How accurate is it to represent numbers (integer or fractional) in radix-2 using finite number of bits?

Basis Representation Theorem: For a given base b, any integer $x \in \mathbb{Z}$ can be uniquely represented as follows:

$$x = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b^1 + a_0$$

where $a_j \in \{0, 1, \ldots, b-1\}$ and $a_k \neq 0$.

Dyadic Rationals Theorem: The dyadic rational set $\mathbb{P}$ (numbers which can be represented as an integer divided by a power of 2), is dense in the set of real numbers $\mathbb{R}$. This means that for any $x \in \mathbb{R}$, there exists a $y \in \mathbb{P}$ that is "as close as you like" to x.

Conclusion: Real numbers can be approximated in radix-2 with finite number of bits, up to a desired level of precision.

Signed Binary Number Representation Standards

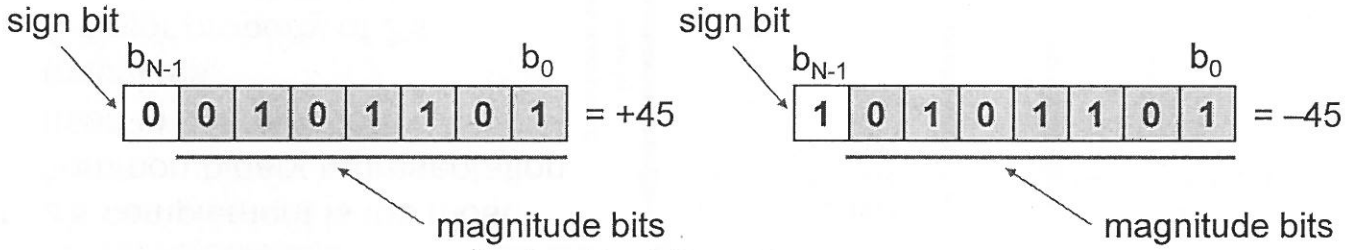
The most popular signed binary number representation standards are:

- Sign-Magnitude
- One's-Complement
- Two's-Complement
- Straight Offset Binary (SOB)
- Binary Coded Decimal (BCD)
- Canonical Signed Digit (CSD)

Sign-Magnitude Representation

The MSB is reserved for sign representation (0 for + and 1 for –). The remaining bits are used to represent the absolute magnitude. With N bits, it can code from $-(2^{N-1}-1)$ to $(2^{N-1}-1)$.

Decimal equivalent: $X_{10} = (-1)^{b_{N-1}}[b_{N-2}2^{N-2} + b_{N-3}2^{N-3} + \ldots + b_1 2 + b_0]$



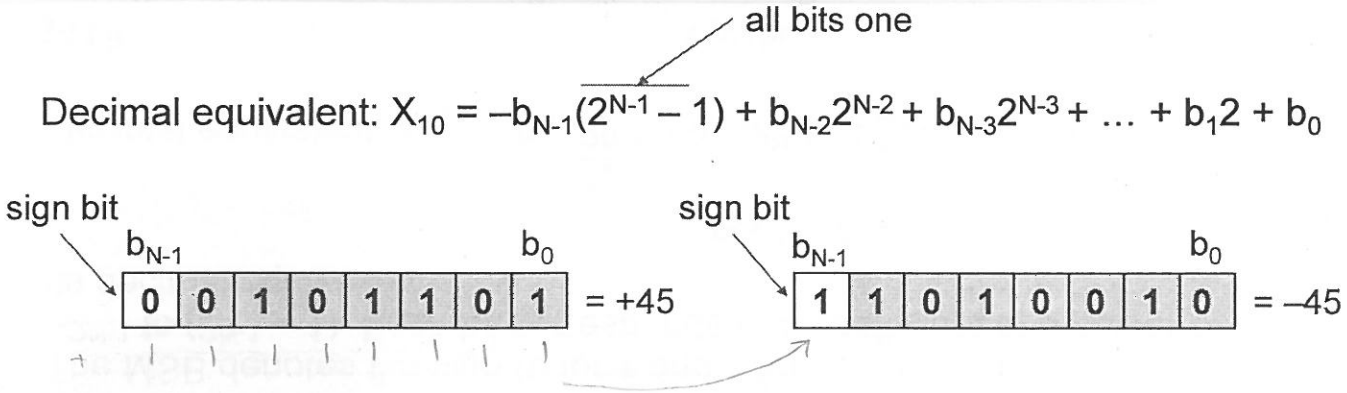
Advantage: Simple to generate and convert

Disadvantage: There are two zeros (+0 and -0); difficult to handle during arithmetic operations

One's Complement

The MSB denotes the sign (0 for + and 1 for –). With N bits, it can code from $-(2^{N-1}-1)$ to $(2^{N-1}-1)$. Each bit corresponds to a coefficient of a power of two in its decimal equivalent.

Decimal equivalent: $X_{10} = -b_{N-1}(2^{N-1}-1) + b_{N-2}2^{N-2} + b_{N-3}2^{N-3} + \ldots + b_1 2 + b_0$



Advantage: Simple to generate and convert

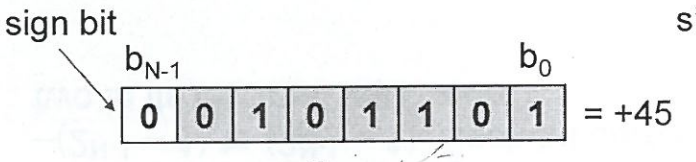
⊖ Disadvantage: There are two zeros (+0 and –0); difficult to handle during arithmetic operations

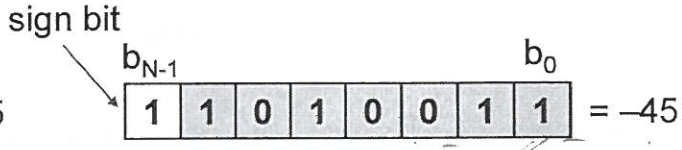
Two's Complement

The MSB denotes the sign (0 for + and 1 for –). With N bits, it can code from -2^{N-1} to $(2^{N-1} - 1)$. Each bit corresponds to a coefficient of a power of two in its decimal equivalent.

doesn't fit into N bits

Decimal equivalent: $X_{10} = -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + b_{N-3}2^{N-3} + \ldots + b_1 2 + b_0$



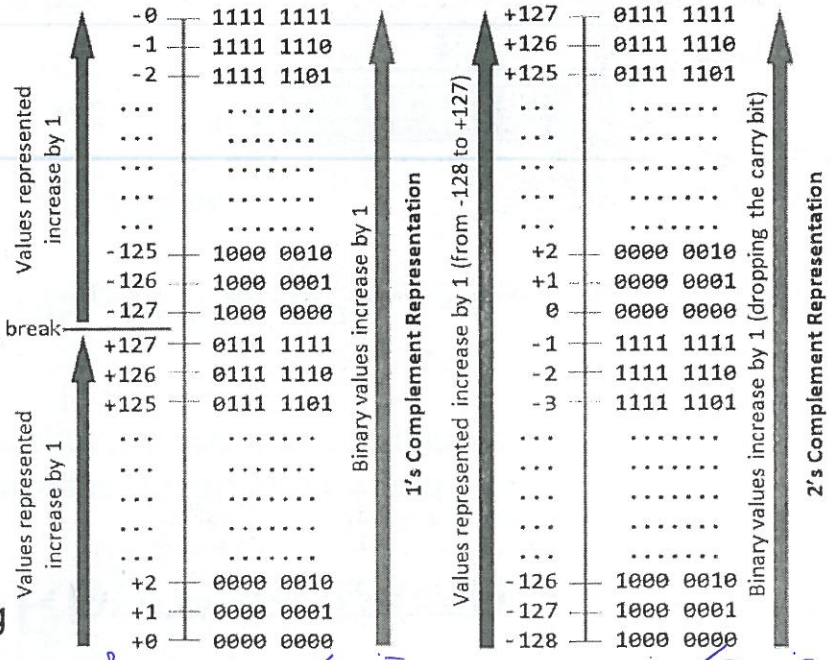


Advantage: No repeated zeros; can code -2^{N-1}; no sign control needed during arithmetic operations, and several other advantages (is the most popular signed number representation format)

Disadvantage: Slightly more difficult to read the decimal equivalent from the binary form (for human).

One's Complement vs. Two's Complement

- 2's complement is the most common binary representation used in computation machines.
- A major property of 2's complement is that the binary values are increased by one-by-one from the most negative to the most positive without a break (by discarding any carry values beyond the word length).
- The default implementation of arithmetic operations in Verilog (since Verilog 2001) is in this format.



Finding One and Two's Complements

1's Complement: Flip all the bits (0 to 1, and 1 to 0)

2's Complement:

- Method 1: Calculate the 1's complement, plus one
- Method 2: Subtract the number from 2^N (this is where the name 2's complement comes from)
- Method 3: Starting from the LSB, preserve all the bits as they are, up to (and including) the right most 1. Flip all the remaining bits up to the MSB

Note: The 2's complement of -2^{N-1} can not be represented in N bits. Therefore, during calculations, it's 2's complement overflows and becomes equal to itself (just like the 2's complement of zero)! This phenomenon can be mathematically explained by the orbit-stabilizer theorem.

Properties of Two's Complement

1. When fitting an N bit 2's complement number into M bits (M>N), the number should be sign extended, i.e., the left most M-N bits should be filled with the MSB (sign bit) of the original number:

```
wire signed [7:0] w;
wire signed [11:0] x;
assign x = {{4{w[7]}}, w};
```

2. In arithmetic right-shifts, the number should be filled by the sign bit from the left:

```
wire signed [7:0] w;

wire signed [7:0] xr = w >>> 3; // arithmetic shift-right (filled by signs from MSB)
wire signed [7:0] yr = w >> 3; // logic shift-right (filled by zeros from MSB)

wire signed [7:0] xl = w <<< 3; // arithmetic shift-left (filled by zeros from LSB)
wire signed [7:0] yl = w << 3; // logic shift-left (filled by zeros from LSB)
```

110 >>1 011 logical
110 >>>1 111 arithmetic

Properties of Two's Complement (continued)

3. No additional circuits are required for handling the signs during addition or subtraction (except for overflow checking). In fact, 2's complement numbers can be treated as unsigned numbers during such arithmetic operations.

4. Overflow check: If two numbers with the same sign are added, overflow occurs if and only if the result has an opposite sign.

   Example:

```
 0111    (carry)
  0111   (7)
+ 0011   (3)
======
  1010   (-6)   invalid!
```


Properties of Two's Complement (continued)

5. Two's Complement Intermediate Overflow Property: "In successive calculation using 2's complement arithmetic (allowing overflows instead of saturation), if it is guaranteed that the final result will fit in the assigned registers, then intermediate overflows are harmless and will not affect the final answer.

   Example (IIR Filter): $y_n = a.y_{n-1} + x_n$

Refs:

- Khan, S. A. (2011). *Digital design of signal processing systems: a practical approach*. John Wiley & Sons., Section 3.5.7
- Smith, J. O. (2007). *Introduction to digital filters: with audio applications* (Vol. 2). Julius Smith., P. 201

Note: Very interesting property; but I haven't seen a rigorous statement or proof for it, yet. Please let me know, if you find a good reference.

Straight Offset Binary (SOB)

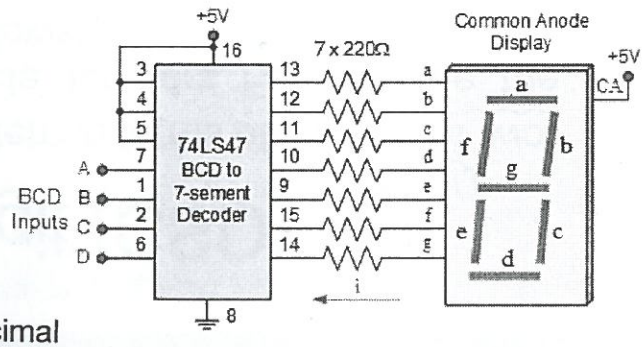
- Offset Binary is a binary code in which the code represents analog values between positive and negative Full-Scale
- Using N bits, starts assigns all-zeros to -2^{N-1} and increments one-by-one up to $2^{N-1} - 1$.
- Conversion to 2's complement: Flip the MSB to convert from SOB to 2's complement and vice versa.
- Application: SOB is most common in Flash Analog-to-Digital Converters (ADC) and Digital-to-Analog Converters (DAC) that use ladder comparators.

SOB	Decimal	2's Complement
1111	7	0111
1110	6	0110
1101	5	0101
1100	4	0100
1011	3	0011
1010	2	0010
1001	1	0001
1000	0	0000
0111	−1	1111
0110	−2	1110
0101	−3	1101
0100	−4	1100
0011	−5	1011
0010	−6	1010
0001	−7	1001
0000	−8	1000

Binary Coded Decimal (BCD)

- A class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits (usually four or eight).
- Special bit patterns are used for a sign or for other indications (e.g., error or overflow)
- Applications: whenever human interaction is needed; such as LCDs, 7–segments, etc.

Decimal digit	BCD Code		
	8 4 2 1	4 2 2 1	5 4 2 1
0	0 0 0 0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 0	0 0 1 0
3	0 0 1 1	0 0 1 1	0 0 1 1
4	0 1 0 0	1 0 0 0	0 1 0 0
5	0 1 0 1	0 0 1 1	1 0 0 1
6	0 1 1 0	1 1 0 0	1 0 1 0
7	0 1 1 1	1 1 0 1	1 0 1 1
8	1 0 0 0	1 1 1 0	1 1 0 0
9	1 0 0 1	1 1 1 1	



Ref: https://en.wikipedia.org/wiki/Binary-coded_decimal

Canonical Signed Digit (CSD)

- CCD is a three-symbol coding system in terms of powers of two.
- It uses a sequence of (+,0,–) to code numbers. For example, the integer 23 can be expanded as follows:

$$23 = + 2^5 - 2^3 - 2^0$$

In CCD, 23 is coded as (+0–00–), i.e.,

  - Positive powers of two are denoted by +
  - Negative powers of two are denoted by –
  - Missing powers of two are denoted by 0

- CCD is popular in some digital signal processors (DSP)

Note: CCD is a non-unique number representation

Note: Statistically, the probability of a digit being zero in CCD can be shown to be close to 66% (vs. 50% in 2's complement encoding). This property leads to more efficient hardware implementations of add/subtract networks and multiplication by constants.

Further Reading: Khan, S. A. (2011). *Digital design of signal processing systems: a practical approach*. John Wiley & Sons., Chapter 6

Fractional Number Representation

The most common binary representations of fractional numbers are:

- Floating-Point: Uses an exponential representation of a number; it is used in most CPUs and some DSP. In FPGA, floating point units (FPUs) are provided by some vendors as hard or soft IP
- Fixed-Point: Uses positive and negative powers of two expansion of a number with a fixed radix point; it is commonly used in fixed-point DSP and microcontrollers
- Mixed-Precision: Uses positive and negative powers of two expansion of a number with a different radix point (at each point of the computing system); it is commonly used in FPGA design

Floating-Point Number Representation

- The basic idea of floating point (FP) representation is to approximate a real number in terms of a fixed number of significant digits (significands or mantissa) scaled by an exponent of a fixed base (e.g., 2, 10, 16, etc.).
- For example:

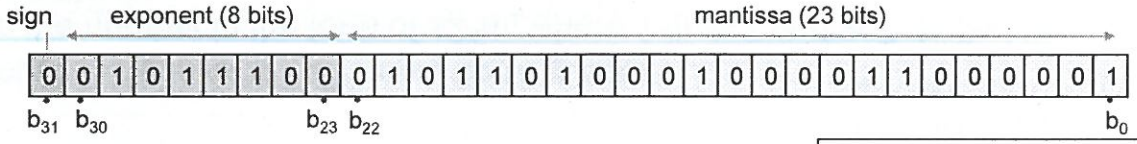
$$1.2345 = 12345 \times 10^{-4}$$

significand: 12345; base: 10; exponent: −4

- Apparently, not all real numbers can be represented in this format (using finite number of digits). However, FP provides an approximation with a fixed relative error throughout the real line (i.e., small errors for small numbers and larger errors for large numbers).

IEEE 754 Single-Precision Binary Floating-Point Format

- According to IEEE 754 floating-point standard:



- The decimal equivalent is:

$$X_{10} = (-1)^{S} \times 2^{e-B} \times \left(1 + \sum_{i=1}^{M} b_{M-i} 2^{-i}\right)$$

The exponent is selected such that the left-most bit of the mantissa is always 1 (which isn't stored in the binary form), making the representation *unique*.

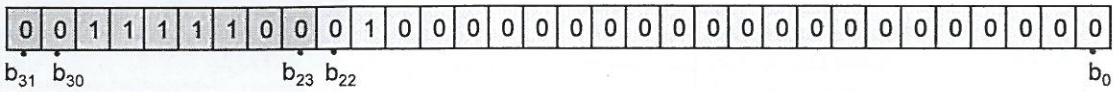
where:

- Total number of bits is 32 in single precision (binary32) and 64 in double precision (binary64)
- S is the sign bit (b_{31} in single precision and b_{63} in double precision)
- e is the exponent (8 bits in single precision and 11 bits in double precision)
- B is a constant bias (equal to 127 in single precision and 1023 in double precision)
- M is the fractional length (23 bits in single precision and 52 bits in double precision)

Single-Precision Binary Floating-Point Examples

Example 1: binary floating point to decimal

0x3E200000 = $(0011\ 1110\ 0010\ 0000\ 0000\ 0000\ 0000\ 0000)_2$



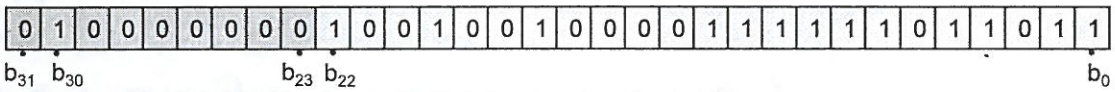
The decimal equivalent is $(-1)^0 \times 2^{124-127} \times (1 + 0.25) = 0.15625$

Example 2: decimal to hex/binary floating point

Scale the number in the form of $\pm 2^a m$, where $1 \leq |m| < 2$ and $a \epsilon Z$, to find the exponent and mantissa

π (3.14159265358979323846264338327950...) ≈ 3.14159274101257324218750

the most accurate 32-bit single-precision approximation for π



which is 0x40490FDB = $(0100\ 0000\ 0100\ 1001\ 0000\ 1111\ 1101\ 1011)_2$

A nice tool: http://www.binaryconvert.com

Floating-Point Arithmetic

Addition/Subtraction:

1. Make the smallest exponent equal to the biggest (by right-shifting the mantissa)
2. Add/subtract the mantissas (note that the smaller ones may vanish to 0 during the right-shifts)

Multiplication/Division:

1. Add/subtract the exponents
2. Multiply/Divide the mantissas
3. Scale and round the results

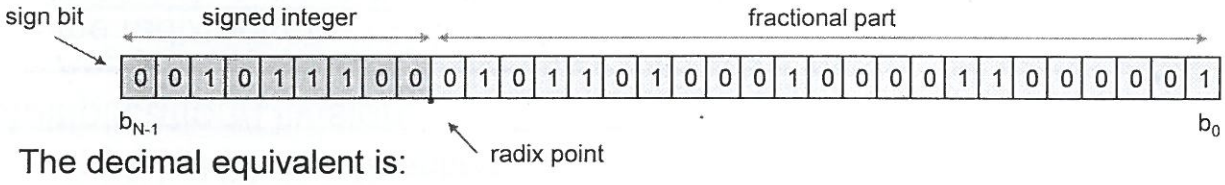
Special Values:

Floating-point representation has reserved codes for special values including: 0^+, 0^-, $+\infty$, $-\infty$, and Not-a-Number (NaN) such as 0/0, $+\infty/-\infty$, $0\times\infty$

Note: Due to the (implicit) leading 1 in front of the mantissa, zero needs to be defined as a special value (when all the bits of the exponent and mantissa are zero), which is different from epsilon ($\pm 2^{-127}$)

Fixed-Point Number Representation

- Fixed-point is basically the 2's complement representation with a fixed power-of-two scaling factor for changing the radix point to enable fractional number representations:



- The decimal equivalent is:

$$X_{10} = 2^{-M} \times (-b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i)$$

signed two's complement

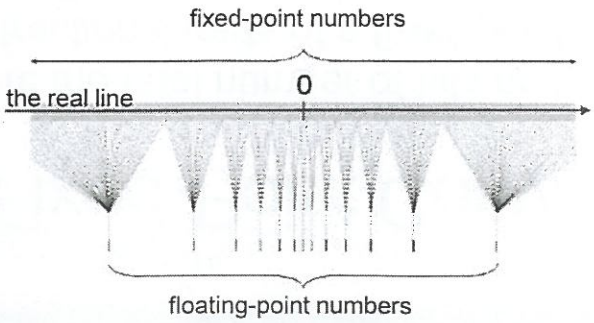
where:

- N is the total number of bits
- M is the fractional point

Note: In fixed-point systems the radix point location is assumed to be fixed throughout the entire system. That's where the name comes from.

Floating-Point vs. Fixed-Point

1. For the same number of bits, they can (almost) code the same number of real numbers.
2. Fixed-point uses all possible codes for number representation, while floating point reserves a few codes for special values. Floating-point has a larger dynamic range (the ratio of the largest to smallest number that are represented)
3. In fixed-point, the range of its MIN and MAX over the real line is quantized to equally spaced numbers (therefore the approximation error is uniform from MIN to MAX); in floating-point, the spacing of numbers is non-uniform (groups of numbers with a fixed intra-gaps but different inter-gaps)
4. Fixed-point hardware architectures are simpler than floating-point architectures; floating-point architectures have additional circuitry for handing special values.



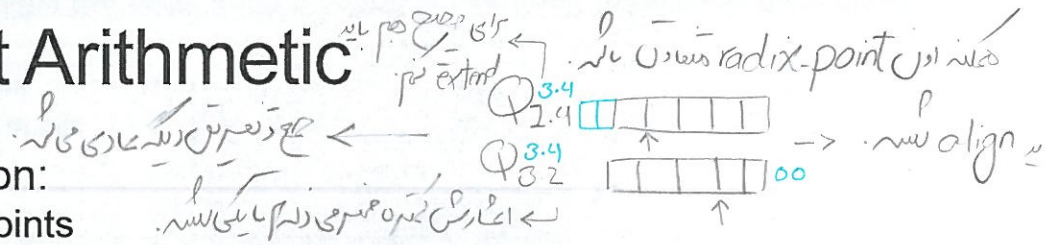
Inspired from: Izquierdo, Luis R. and Polhill, J. Gary (2006). 'Is Your Model Susceptible to Floating-Point Errors?'. Journal of Artificial Societies and Social Simulation 9(4)4 <http://jasss.soc.surrey.ac.uk/9/4/4.html>

The $Q_{m.n}$ Fixed-Point Convention

- In order to denote the total number of bits and the bits assigned to the integer and fractional parts of a fixed-point number, various conventions exist. For example,
  - Texas Instruments' Q_N format (or $Q_{1.N}$) assumes 1 bit (the sign bit) as the integer part and N bits for the fractional part.
  - Matlab's fixed-point toolbox takes the total number of bits and the fractional length to form an fi-object.

Throughout this course, we use the $Q_{m.n}$ convention, where:
- m is the number of bits assigned to the integer part
- n is the number of bits assigned to the fractional part
- N = m + n is the total number of bits (including the sign)
- The numbers are signed, therefore the MSB represents the sign

Fixed-Point Arithmetic



Addition/Subtraction:
1. Align the radix points
2. Zero pad the LSB of numbers with shorter fractional lengths
3. Sign extend the MSB of numbers with shorter integer lengths
4. Apply addition/subtraction

Multiplication/Division:
1. Apply multiplication/division as if they were integer valued (regardless of the radix point)
2. Find the appropriate radix point by adding/subtracting the radix points

Note: Bit-growth occurs during fixed-points arithmetic, which is handled by either:
1. increasing the number of bits,
2. truncation/rounding from the LSB or MSB (is discussed in details later), or
3. a combination of both 1 and 2

Bit-Growth in Fixed-Point Arithmetic

In order to guarantee that no overflow occurs during arithmetic operations, the number of output bits should be longer than the arithmetic operands:

1. $Q_{m_1.n_1} \pm Q_{m_2.n_2} = Q_{m.n}$
   where $m = \max(m_1,m_2)+1$ and $n = \max(n_1,n_2)$
2. $Q_{m_1.n_1} \times Q_{m_2.n_2} = Q_{m.n}$
   where $m = m_1+m_2$ and $n = n_1+n_2$

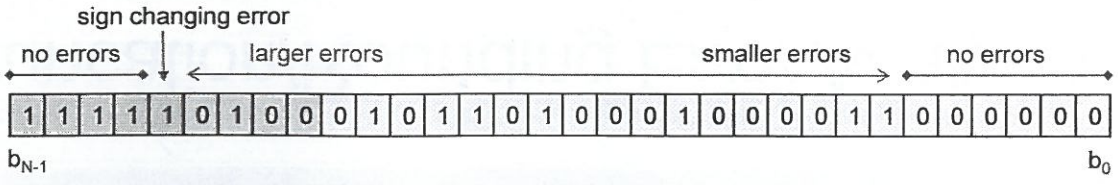
Note: During multiplication, $N = N_1 + N_2 - 1$ is generally enough. The only exception (requiring $N = N_1+N_2$) is for signed numbers when the two most negative numbers (-2^{N_1-1} and -2^{N_2-1}) are multiplied together, resulting in $+2^{(N_1+N_2-2)}$, which overflows in $N = N_1 + N_2 - 1$ bits and requires $N = N_1 + N_2$. This single bit can be saved by either:

1. Making sure that the two operands are never equal to the most negative numbers (this is possible when one of the operands is a known constant)
2. Approximating $2^{(N_1+N_2-2)}$ with $2^{(N_1+N_2-2)}-1$! Yes, this approximation is OK in many systems.

Controlling Bit-Growth in Fixed-Point Systems

It is impractical (and unnecessary) to increase the number of bits after successive arithmetic operations. Bit growth can be controlled by discarding either from the LSB or MSB of the arithmetic result.

- When to discard from the MSB?
  - Only possible when the full-length is not utilized or the arithmetic operation (mathematically) guarantees that no bit growths occur → results in no errors
  - If the full-length is utilized → causes large sign/amplitude errors
- When to discard from the LSB?
  - The right most LSB zeros can be discarded without any errors
  - Truncating/rounding non-zero LSB results in relatively small errors, depending on the number's magnitude
  - A stochastic framework is required to analyze the average truncation/rounding error effect.



Truncation/Rounding Error Analysis

- The truncation procedure can be modeled by an operator Q(•):

$$y_n = Q(x_n) = x_n + e_n$$

x: input sample (signal) y: truncated/rounded result

e: truncation/rounding error n: sample index

- The impact of truncation error depends on both the original sample (signal) and the truncated values' amplitudes.
- In continuous data streams, the most common approach for studying the truncation error impact is to measure the ratio of the average data power to the average noise power, known as the signal-to-noise ratio (SNR):

$$\mathrm{SNR_{dB}} = 10\log_{10}\left(\frac{E\{x_n^2\}}{E\{e_n^2\}}\right)$$

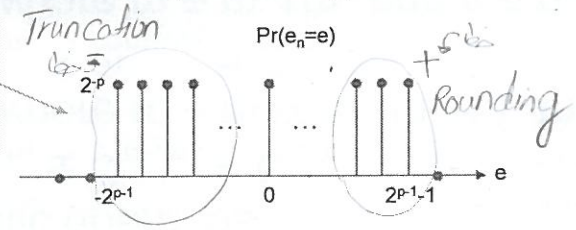
where $E\{\cdot\}$ denotes averaging (or stochastic expectation) over all ensembles.

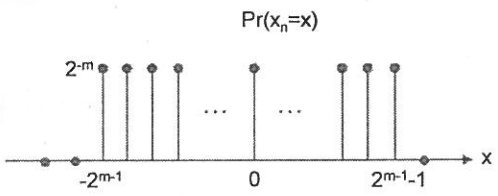
Note: The calculation of the SNR requires prior assumptions regarding the input stream and the truncation error distribution.

Truncation/Rounding Error SNR Calculation

Suppose that we have an m bit signed integer sequence x_n, for which we want to round the p LSB bits (to zero) and obtain y_n. Assuming a uniform distribution for x_n, the probability density functions (pdf) of x_n and the error sequence e_n are:

rounding instead of truncation (+ and – error values)





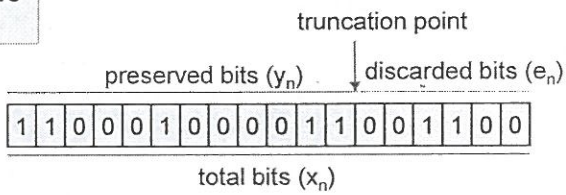
Error mean:

$$\bar{e} = E\{e_n\} = \sum_{i=-2^{p-1}}^{2^{p-1}-1} i \cdot \frac{1}{2^p} = -\frac{1}{2}$$

slightly biases towards negative numbers

Error variance:

$$\sigma_e^2 = E\{(e_n - \bar{e})^2\} = \sum_{i=-2^{p-1}}^{2^{p-1}-1} (i + \frac{1}{2})^2 \cdot \frac{1}{2^p} = \frac{2^{2p}-1}{12}$$



Truncation/Rounding Error SNR Calculation

(continued)

Similar results hold for the mean and variance of x_n. Therefore the SNR is:

$$\mathrm{SNR_{dB}} = 10\log_{10}\left(\frac{\sigma_x^2}{\sigma_e^2}\right) = 10\log_{10}\left(\frac{2^{2m}-1}{2^{2p}-1}\right)$$

which for large p can be approximated as:

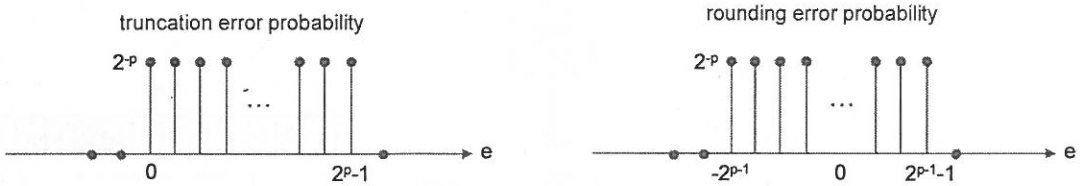
$$\mathrm{SNR_{dB}} \approx 10\log_{10}\left(\frac{2^{2m}}{2^{2p}}\right) = 20(\mathrm{m}-\mathrm{p})\log_{10}2 \approx 6.02(m-p)$$

Note: This is the 6dB per-bit rule of thumb: *truncating each bit reduces the SNR for about 6dB*. We will find a similar rule later for ADC performance with different signal and noise distributions.

Exercise: Derive the above equations (mean and variance of error) analytically. Do the results change if the number is in the $Q_{m.n}$ format?

Truncation vs. Rounding

- While truncation simply discards the unnecessary bits, rounding approximates with the closest number.
- Rounding is commonly preferred over truncation, as it is less-biased (the very small bias is due to the representation of -2^{p-1} in 2's complement).



Example: `round(3.7) = 4;  truncate(3.7) = 3;`

- Truncation versus rounding in Verilog:

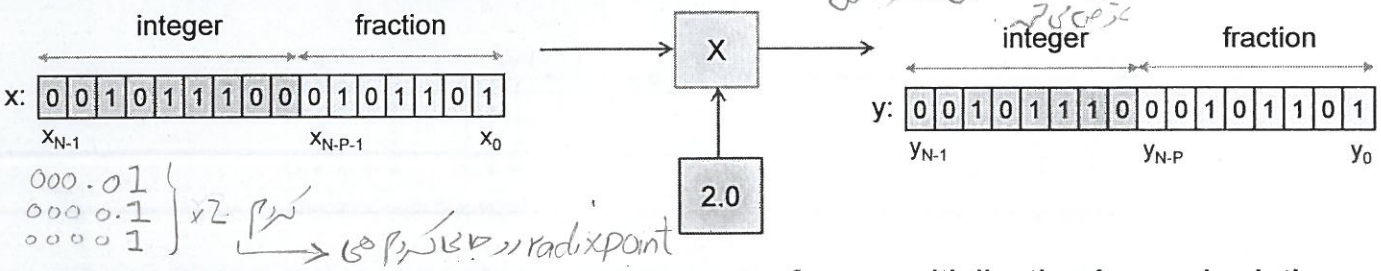
```
1 reg [11:0] a;
2 wire [9:0] a_round = a[11:2] + a[1];
3 wire [9:0] a_trunc = a[11:2];
```

Radix-10 equivalent trick:

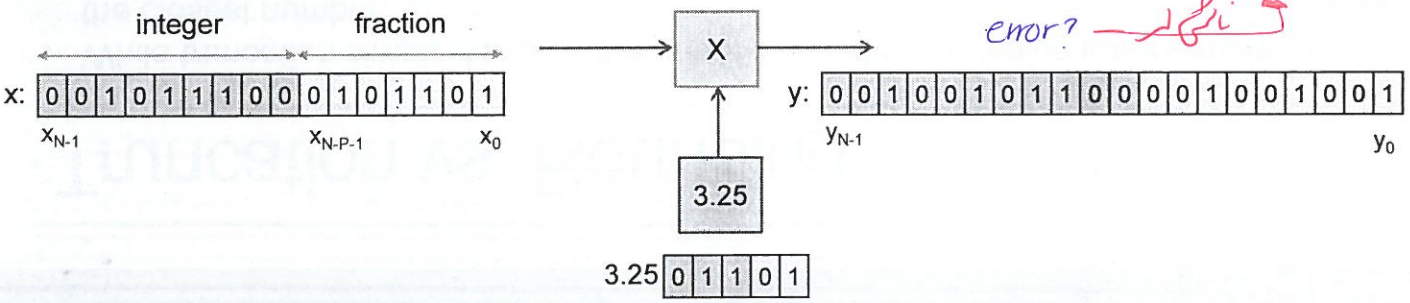
```
[3.7 + 0.5] = 4;
[3.2 + 0.5] = 3;
```

Mixed-Precision Multiplication Examples

Example 1: Multiplication by constant powers of two: no multiplication is required; only the radix point convention changes; no error increase

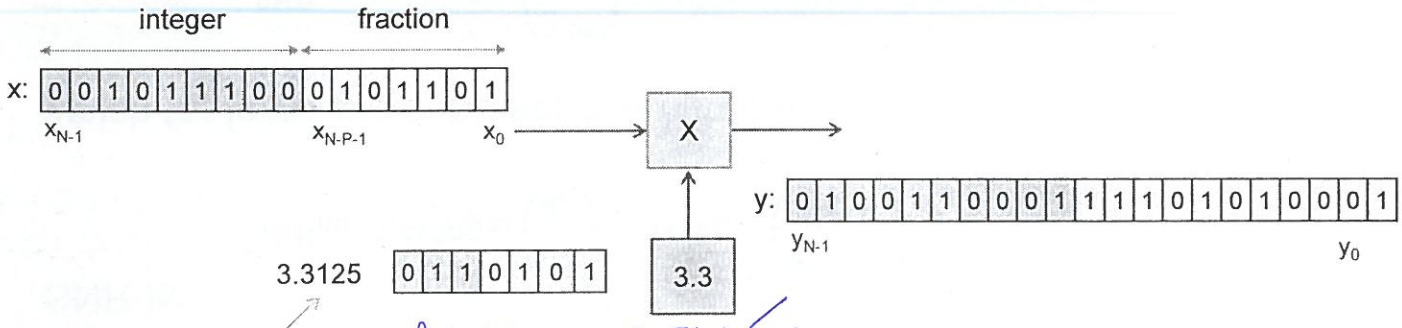


Example 2: Multiplication by constant non powers of two: multiplication is required; the radix point and register length may change; error might be added due to output truncation



Mixed-Precision Multiplication Examples

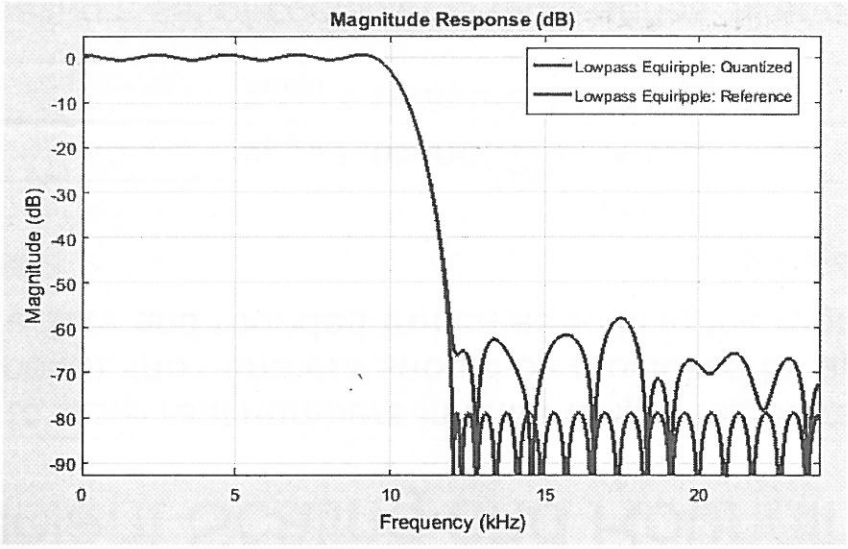
Example 3: Multiplication by fractional non powers of two that can not be represented by sum of powers of two: Unavoidable representation error, even before multiplication



The closest approximation of the constant in 7 bits

Mixed-Precision Multiplication Examples

Rounding/truncating the coefficients in data/signal processing systems can change the nominal performance of the system. For example, in filter design:

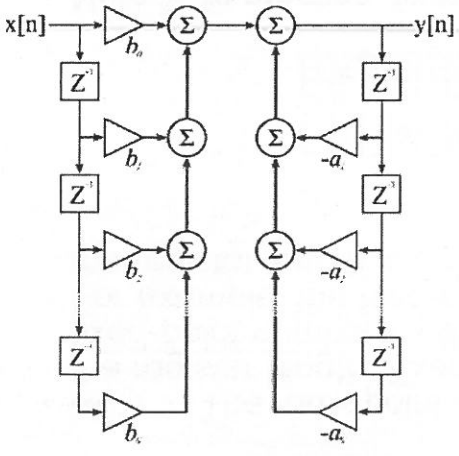


Sample lowpass filter designed in Matlab FDATool in double precision floating-point (blue) and after quantization with 12-bit fixed-point (red)

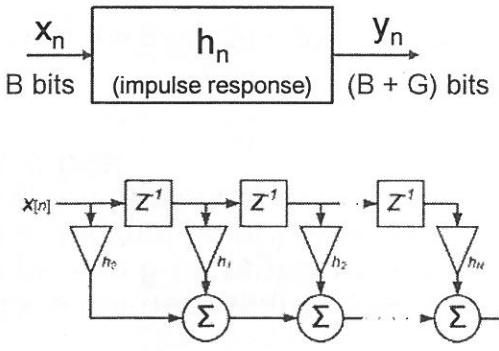
Mixed-Precision in Digital Filters

Example 4: Discrete-time convolution $y_n = x_n * h_n = \sum_m h_m x_{n-m}$: The maximum bit growth in the output is equal to the length of the filter coefficients L1–Norm:

$$G = \left\lceil \log_2 \left(\sum_m |h_m| \right) \right\rceil$$



IIR Filter



FIR Filter

Note: From Signals & Systems Theory we know that for a stable causal filter $\sum_m |h_m| = B < \infty$. Therefore "the output of a stable filter with a bounded input can always be stored in a register of finite length without overflow"

Further Notes on Fixed-Point and Mixed-Precision

- Note 1: The radix point does not necessarily need to be within the range of the register length. Example: An 8-bit register can be used to represent fixed-point numbers with a decimal point below the LSB or above the MSB. For example, the following are legitimate fixed-point numbers, even though the register length is only 8 bits:

implied but not stored with the number

0	1	1	0	1	0	1	0	$x = 2^{+15}.(2^1 + 2^3 + 2^5 + 2^6)$
1	0	1	0	1	0	1	1	$x = 2^{-12}.(2^0 + 2^1 + 2^3 + 2^5 + 2^6 - 2^7)$

- Note 2: In practice, fixed or floating-point numbers can have an arbitrary and implicit scaling factor, which is known to the designer; but is not coded or stored with the number. These scaling factors are only incorporated when numbers are mapped to their corresponding physical values (voltage, temperature, current, etc.) for user visualization or analysis. Example: Uniform analog-to-digital convertors map their input voltage to the output code with a constant scaling factor, which is known by the designer; but does not affect internal FPGA calculations.

Coefficient Scaling and Rounding

- In order to store real numbers in finite-length registers (fixed or floating-point), the numbers should be multiplied by appropriate scaling factors and rounded/truncated to fit in the registers.

Examples:

scaling factors

$y_{fixed} = \text{round}(2^{16} \times y_{real})$

$y_{fixed} = \text{round}(3.14 \times y_{real})$

- When scaling a set of coefficients (time-series, filter coefficients, etc.) to fit in N bits, the optimal performance (with minimum quantization error) is obtained when the maximum/minimum scaled values are equal to the maximum/minimum possible numbers (-2^{N-1} and $2^{N-1}-1$).
- Therefore, the optimal scaling factor is not necessarily a power of two (e.g., see Matlab FDAtool's quantization and scaling options)

Bit-Growth in Digital Filter Implementation* (optional)

In digital filter implementation, the L1–Norm bit growth $G = \lceil \log_2(\sum_m |h_m|) \rceil$ is the worst-case (most pessimistic), which does not make any assumptions on the input signal. This formula can be relaxed (approximated) in some cases.

1. Instantaneously narrow-band signals: For signals having a dominant frequency peak at each time instant:

$$x_n = A\cos(\omega_0 n + \theta) \rightarrow y_n \approx |H(e^{j\omega_0})|\, A\cos(\omega_0 n + \varphi_{\omega_0})$$

$$\text{Bit Growth } G_0 = \left\lceil \log_2 \left(\max_{-\pi \le \omega < \pi} |H(e^{j\omega})| \right) \right\rceil$$

2. Random input signals: Using Parseval's theorem, the output variance of a filter with a random input is related to its input variance as follows:

$$\sigma_y^2 = \sigma_x^2 \sum_m |h_m|^2$$

Therefore, with the following bit-growth, the probability of overflow at a filter's output is (almost) equal to the probability of input overflow:

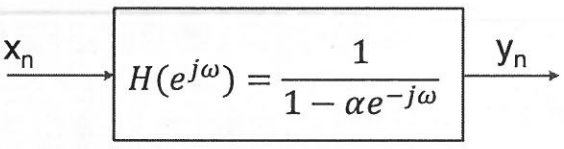
$$\text{Bit Growth } G_1 = \left\lceil \log_2 \left(\sqrt{\textstyle\sum_m |h_m|^2} \right) \right\rceil$$

Bit-Growth in Digital Filter Implementation* (optional)

Example: A first-order lowpass IIR filter: $y_n = \alpha y_{n-1} + x_n \ (0 < \alpha < 1)$

The impulse response is $h_n = \alpha^n u[n]$

Therefore $\sum_m |h_m| = \frac{1}{1-\alpha}$ and $\sum_m |h_m|^2 = \frac{1}{1-\alpha^2}$



Bit-growth analysis for $\alpha = 0.9$:

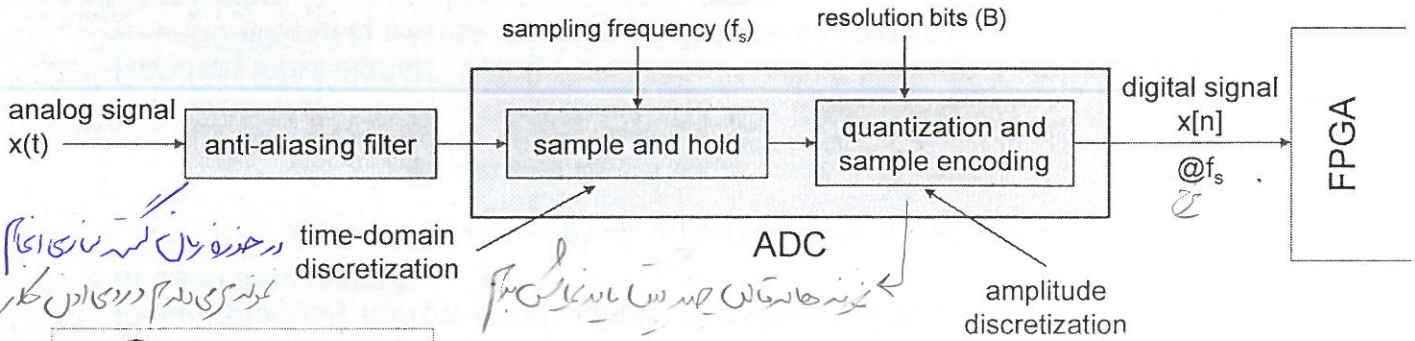
- L1-Norm: $G = \lceil \log_2(\sum_m |h_m|) \rceil = \left\lceil \log_2 \left(\frac{1}{1-\alpha} \right) \right\rceil = \lceil 3.3219 \rceil = 4$
- Narrow-band assumption: $G_0 = \left\lceil \log_2 \left(\frac{1}{|1-\alpha|} \right) \right\rceil = \lceil 3.3219 \rceil = 4$
- Parseval's theorem for stochastic inputs: $G_1 = \left\lceil \log_2 \left(\frac{1}{\sqrt{1-\alpha^2}} \right) \right\rceil = \lceil 1.198 \rceil = 2$

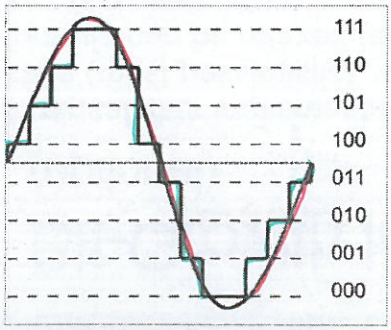
Result: In this example the L1-norm and narrow-band assumption, both demand 4 additional bits at the output y_n; but according to the output variance criterion if we are fine with occasional overflows, adding only 2 bits is statistically OK.

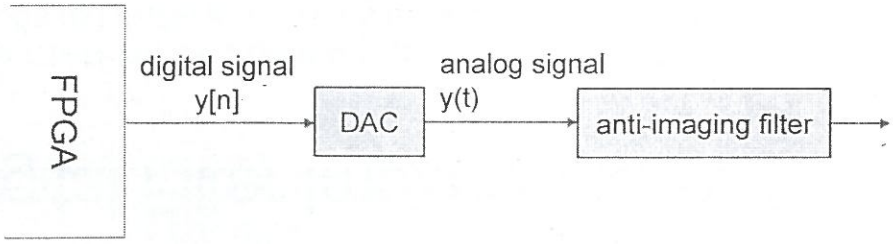
ANALOG TO DIGITAL CONVERTORS AND DIGITAL TO ANALOG CONVERTORS

Analog to Digital Convertor (ADC) vs. Digital to Analog Convertor (DAC)

ADC and DAC are integral parts of most FPGA-based signal processing systems

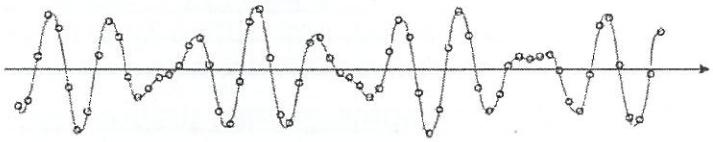




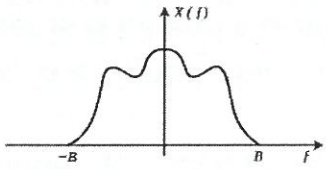


The Nyquist Rate

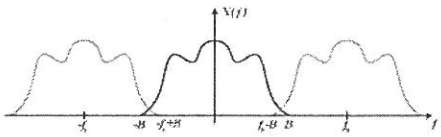
- The Nyquist sampling theorem defines the minimum number of samples acquired from a band-limited analog signal per unit time, in order to guarantee the reconstruction of the original signal from these samples. It requires: $f_s \geq 2B$



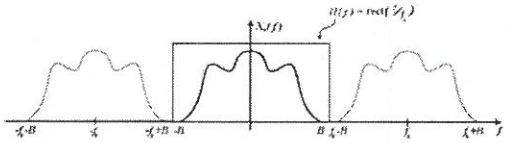
time-domain signal and its samples



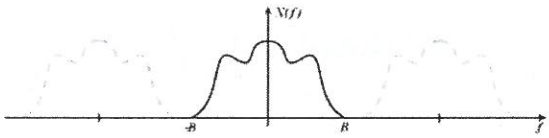
band-limited signal in the frequency domain



After impulse train sampling with $f_s < 2B$; Nyquist rate violated



After impulse train sampling with $f_s > 2B$; Nyquist rate fulfilled

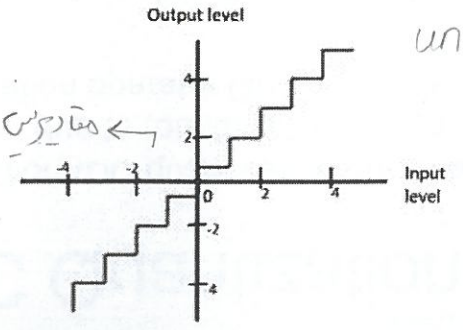


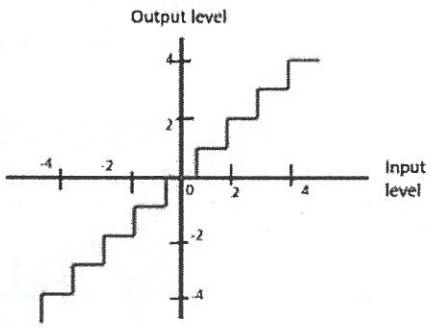
reconstructed signal

Ref: https://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem
Further Reading: Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems (2nd Ed.)*. Prentice-Hall, Inc., 1996

ADC Encoding Curve

- The mapping between the input voltage of an ADC and the output code can be described by an encoding curve.
- In a binary encoding ADC with B bits, the input voltage range $[V_{min}, V_{max}]$ is divided into 2^B segments and any input voltage within this range is approximated with one of the nearest voltages and represented by a code.
- The ADC encoding curve may be uniform or non-uniform.
- For example, the following are two uniform encoding curves, based on rounding (left) and truncation (right)





Question: How to quantify the performance of an ADC?

ADC Quantization Error Analysis

- The effect of ADC quantization error can be analyzed with a method similar to SNR calculation due to rounding/truncation. The quantization procedure can be modeled by a quantization operator Q(•):

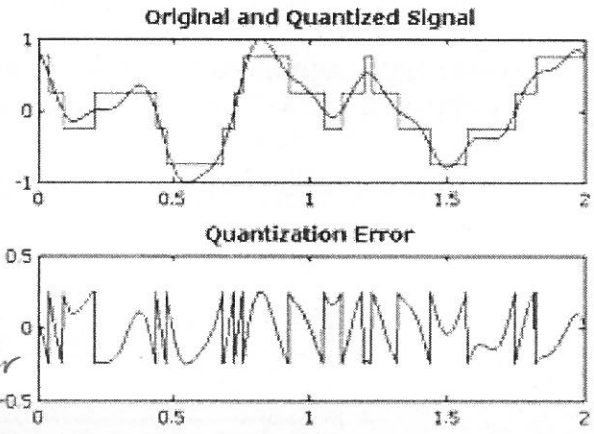
$$y_n = Q(x_n) = x_n + e_n$$

x_n: ADC input sample (after zero-order hold), y_n: quantized result, e_n: quantization error

- We again use the signal-to-noise ratio (SNR) as the performance measure:

$$\mathrm{SNR_{dB}} = 10\log_{10}\left(\frac{E\{x_n^2\}}{E\{e_n^2\}}\right)$$

- This analysis requires some assumptions regarding the input signal and the quantization error probability density functions

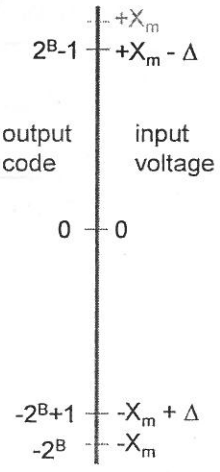


ADC Quantization Error Analysis (continued)

Quantization model: $y_n = Q(x_n) = x_n + e_n$

Assumptions:

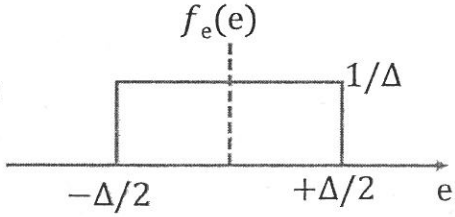
1. The signal x_n is a signed real value in $[-X_m, X_m)$
2. The quantizer is B bit and it divides $[-X_m, X_m)$ into 2^B equal segments of length $\Delta = 2X_m/2^B$
3. The signal x_n and the quantization error e_n are statistically independent (we will study the counter assumption later)
4. The quantization error samples e_n are independent identically distributed (*iid*) with a uniform distribution between $-\Delta/2$ and $\Delta/2$



Therefore: $\bar{e} = E\{e\} = \int_{-\infty}^{+\infty} e f_e(e)de = 0$

$$\sigma_e^2 = E\{(e-\bar{e})^2\} = \int_{-\infty}^{+\infty}(e-\bar{e})^2 f_e(e)de = \frac{\Delta^2}{12}$$

We have calculated the denominator of the SNR equation. In the sequel we consider three cases for the input signal: Sinusoidal (deterministic) signal, Gaussian distributed stochastic signal, Uniformly distributed stochastic signal



Quantization error probability density function

ADC Quantization SNR with Sinusoidal Input

- Sinusoidal input signals are the standard measurement method for calculating ADC SNR.
- Assuming $x_n=X_m\cos(\omega n)$, we have $E\{x_n\}=0$ and $E\{x_n^2\} = X_m^2/2$.
  Therefore:

$$\mathrm{SNR_{dB}} = 10\log_{10}\left(\frac{E\{x_n^2\}}{E\{e_n^2\}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{2}}{\frac{\Delta^2}{12}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{2}}{\frac{4X_m^2}{12\times 2^{2B}}}\right)$$

or

$$\mathrm{SNR_{dB}} \approx 6.02B + 1.76\mathrm{dB}$$

Note: This is the well-known 6dB per-bit rule, which should be memorized as a rule of thumb by any hardware engineer!

ADC Quantization SNR with Uniformly Distributed Input

- We next assume that the input signal is a stochastic random variable, uniformly distributed between $-X_m$ and X_m: $x_n \sim U(-X_m, X_m)$
- Therefore we have $E\{x_n\}=0$ and $E\{x_n^2\} = X_m^2/3$.
  Therefore:

$$\mathrm{SNR_{dB}} = 10\log_{10}\left(\frac{E\{x_n^2\}}{E\{e_n^2\}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{3}}{\frac{\Delta^2}{12}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{3}}{\frac{4X_m^2}{12\times 2^{2B}}}\right)$$

or

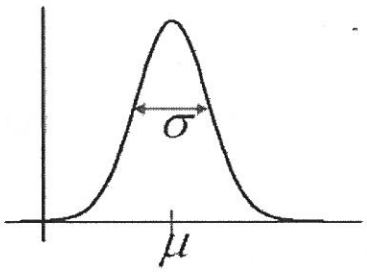
$$\mathrm{SNR_{dB}} \approx 6.02B$$

Note: The 1.76dB is no longer there, but we still see the 6dB per-bit property.

ADC Quantization SNR with Gaussian Distributed Input

- We finally assume that the input signal is a stochastic random variable, with a Gaussian distribution $x_n \sim N(0,\sigma_x^2)$.
- The Gaussian distribution has infinite tails and overflow at the ADC input is unavoidable. However, the probability of overflow is reduced by controlling the input variance σ_x^2 relative to the ADC reference voltages $-X_m$ and X_m.
- Let's assume $X_m=k\sigma_x$. According to the Gaussian curve, for k = 1, 2, 3, and 4, the probability of ADC input overflow is 31.73%, 4.55%, 0.26%, and 0.01%, respectively.
- Assuming k = 4, we have $E\{x_n\}=0$ and $E\{x_n^2\} = X_m^2/16$. Therefore:

$$\mathrm{SNR_{dB}} = 10\log_{10}\left(\frac{E\{x_n^2\}}{E\{e_n^2\}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{16}}{\frac{\Delta^2}{12}}\right) = 10\log_{10}\left(\frac{\frac{X_m^2}{16}}{\frac{4X_m^2}{12\times 2^{2B}}}\right)$$



or

$$\mathrm{SNR_{dB}} \approx 6.02B - 7.27\mathrm{dB}$$

Note: We still see the 6dB per-bit property.

Note: ADC ICs commonly have an out-of-range (OTR) pin for reporting input overflow per-sample

Non-ideal ADC

- Practical ADC circuitry are never ideal and do not reach their nominal performance (SNR=6.02B + 1.76dB).
- The standard approach to measure the true performance of an ADC is by giving it a sinusoidal input signal with an amplitude of 1dB below full-scale (to avoid overflow) and measuring the real SNR and the effective number of bits (ENOB):

True SNR measured by giving a full dynamic-range sinusoidal to the ADC and measuring the SNR of an acquired block of data

$$\mathrm{ENOB} = \frac{\mathrm{SNR_{dB}} - 1.76\mathrm{dB}}{6.02}$$

The effective number of bits; a real-value, always smaller than the nominal number of ADC bits (ENOB < B)

ENOB Examples

- AD9246 14-Bit, 80 MSPS/105 MSPS/125 MSPS, 1.8 V Analog-to-Digital Converter:

Parameter[1]	Temp	AD9246BCPZ-80 Min	Typ	Max	AD9246BCPZ-105 Min	Typ	Max	AD9246BCPZ-125 Min	Typ	Max	Unit
SIGNAL-TO-NOISE-RATIO (SNR)											
f_{IN} = 2.4 MHz	25°C		71.9			71.9			71.9		dBc
f_{IN} = 70 MHz	25°C		71.9			71.9			71.7		dBc
	Full	70.8			69.5			69.5			dBc
f_{IN} = 100 MHz	25°C		71.6			71.6			71.6		dBc
f_{IN} = 170 MHz	25°C		70.9			70.9			70.8		dBc
SIGNAL-TO-NOISE AND DISTORTION (SINAD)											
f_{IN} = 2.4 MHz	25°C		71.1			71.1			71.1		dBc
f_{IN} = 70 MHz	25°C		71.5			70.8			70.6		dBc
	Full	70.4			68.5			68.5			dBc
f_{IN} = 100 MHz	25°C		70.6			70.6			70.6		dBc
f_{IN} = 170 MHz	25°C		69.9			69.9			69.8		dBc
EFFECTIVE NUMBER OF BITS (ENOB)											
f_{IN} = 2.4 MHz	25°C		11.7			11.7			11.7		Bits
f_{IN} = 70 MHz	25°C		11.6			11.6			11.6		Bits
f_{IN} = 100 MHz	25°C		11.6			11.6			11.6		Bits
f_{IN} = 170 MHz	25°C		11.5			11.5			11.5		Bits

Non-uniform ADC Encoding Curves

- Intuitively, in ADC with non-uniformly distributed inputs, many bits are "wasted" for low-probability samples (there are profound theoretical proofs behind this intuition). One could use the bits more efficiently by:

1. Using non-uniform ADC encoding curves: Divide $-X_m$ and X_m into unequal segments (assign smaller segments to higher probabilities and larger segments to lower probability values). Example: A-law and μ-law companding algorithms used in old 8-bit PCM digital communication systems for better use of the dynamic range

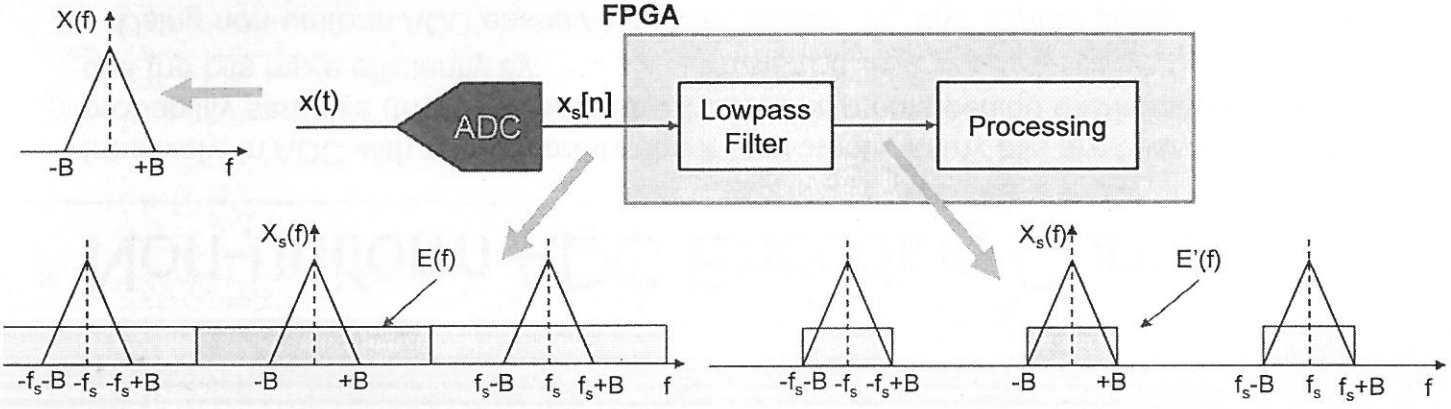
2. Making the input sequence distribution uniform: A useful theorem from random variables:

   If a random variable (RV) x with a probability density function (pdf) $f_X(x)$ and cumulative distribution function (CDF) $F_X(x)$ passes a nonlinear memoryless system with a characteristics $u = F_X(x)$, the output u is uniformly distributed. Also, if a uniformly distributed RV u is given to $y = F_X^{-1}(u)$, the output has a distribution $f_X(\cdot)$.

   Note: This property can be used to make arbitrary RVs from uniform distributions and vice versa in FPGA.

ADC SNR Improvement by Over-Sampling

- Looking back at the quantization model $y_n = Q(x_n) = x_n + e_n$, the quantization error samples e_n were assumed to be independent identically distributed (iid). Therefore, the quantization noise has a white spectrum and its total power $E\{e_n^2\}$ is equally distributed over the entire Nyquist-band $[0,f_s]$.
- If the signal is over-sampled beyond the Nyquist rate, the ADC SNR can be improved by lowpass filtering the ADC outputs (in the digital domain).
- In this case, we have: $\mathrm{SNR_{dB}} \approx 6.02\mathrm{B} + 1.76\mathrm{dB} + 10\log_{10}(\mathrm{OSR})$, where OSR is the over-sampling ration (f_s/2B)

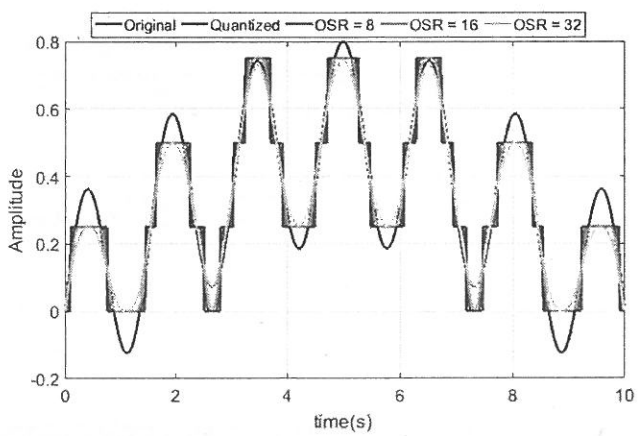


ADC SNR Improvement by Over-Sampling (continued)

- Over-sampling rule of thumb: "Each factor of two above the Nyquist rate, is equivalent to 3dB of SNR improvement (after low-pass filtering)". Therefore, SNR improvement by OSR is expensive!
- Question: OSR = 4 improves the SNR for 6dB, equivalent to 1 bit of higher resolution. Does this mean that we can have a mono-bit ADC that is equivalent to a 12-bit ADC?!

Answer: Yes (to some extent)!

```
7     % Signal simulation
8 -   N = 1000;
9 -   n = 0 : N-1;
10 -  fs = 100; % analog signal simulation
11 -  t = n/fs;
12 -  f1 = 0.05;        f2 = 0.65;
13 -  x = .5*sin(2*pi*f1*t) + 0.3*sin(2*pi*f2*t) + 1e-4*randn(size(t));
14    %(The Nyquist rate is twice the maximum frequency)
15
16    % Amplitude quantization
17 -  B = 3; % number of quantization bits
18 -  y = double(fi(x, 1, B));
19
20    % Signal recovery using over-sampling ratio (OSR)
21 -  OSR1 = 8;
22 -  OSR2 = 16;
23 -  OSR3 = 32;
24 -  y1_smoothed = filtfilt(ones(1, OSR1), OSR1, y);
25 -  y2_smoothed = filtfilt(ones(1, OSR2), OSR2, y);
26 -  y3_smoothed = filtfilt(ones(1, OSR3), OSR3, y);
```

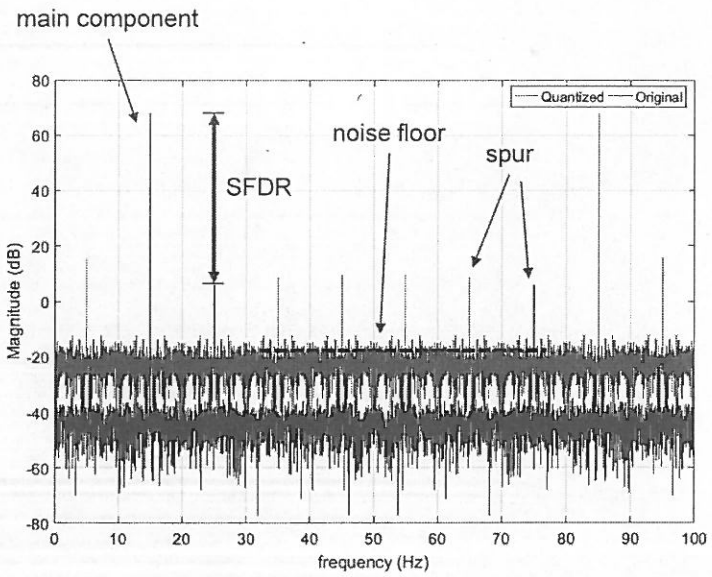


Spurious-Free Dynamic Range (SFDR)

- Looking back at the quantization model $y_n = Q(x_n) = x_n + e_n$, the quantization error e_n was assumed to be independent from x_n. However, this assumption is violated in low number of bits.

Spurs are notable components and spikes of noise within a signal's spectrum and above the noise floor, which do not correspond to the original signal; but are somehow correlated with it (they move in the spectrum as the sampling frequency changes or as the signal components move).

SFDR is the gap (in dB) between the original frequency component and the strongest spur



Spurious-Free Dynamic Range Improvement

Spurs are very important in practice, as they are commonly mistaken with the original signal components.

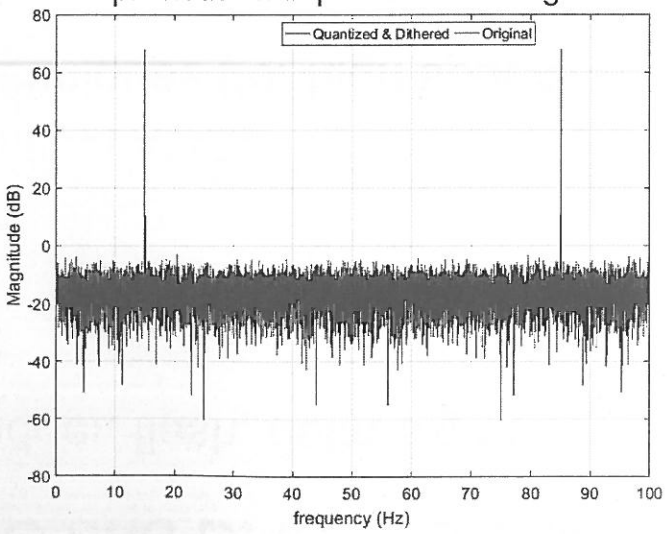
Note: Spurs can also occur during FPGA arithmetic truncation/rounding

How to improve the SFDR?

1. Increase the number of ADC (quantization) bits
2. Break the correlation between the signal and quantization (rounding/truncation) errors by adding dithers prior to quantization (rounding/truncation), e.g., by using high-thermal noise resistors in ADC inputs

Dither is a noise (at the level of the signal's LSB) intentionally added to the signal before quantization to de-correlate the signal and quantization noise

previous example after dithering



Note: Dithering improves the SFDR at a cost of decreasing the SNR (increasing the noise floor)
Note: Dithers can be generated in FPGA using linear-feedback shift registers (LFSR)

Further Reading on ADC and DAC* (Optional)

- ADC internal technologies: ladder, flash, delta-sigma modulation
- Integral nonlinearity (INL)
- Clock jitter
- DAC technologies
- Contemporary FPGAs with built-in ADCs
- Quadrature ADC sampling techniques (for high speed)
- Mono-bit technologies
- ADC/DAC tradeoffs

Further reading: refer to the references on ADC/DAC in the course's references folder

WORD LENGTH SELECTION IN FPGA-BASED ARITHMETIC

Background

- Real-world applications require the representation of real-valued data in floating-point or fixed-point formats
- Real numbers can be approximated in these formats using the necessary number of bits and by proper scaling

Question 1: How many bits should be used for internal calculations?

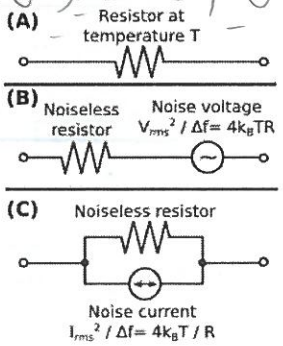
Answer: Considering that coefficient quantization and rounding/truncation introduce additional errors to the input data, the internal register lengths are selected to meet the minimum required SNR (selected by the designer)

Question 2: How to choose the minimum required SNR?

Answer: It is application-dependent

Word-Length Selection in FPGA Designs

- The most common sources of noise in analog and digital electronics systems are
  1. Thermal noise of electronic devices and elements
  2. Quantization errors in digital systems, due to number representation in finite-length registers and rounding/truncation
- In mixed analog digital designs (containing analog elements, ADC, DAC, FPGA, processors, etc.) the conventional standard is to keep the fixed-point computational errors at the same level or below the input analog noise level



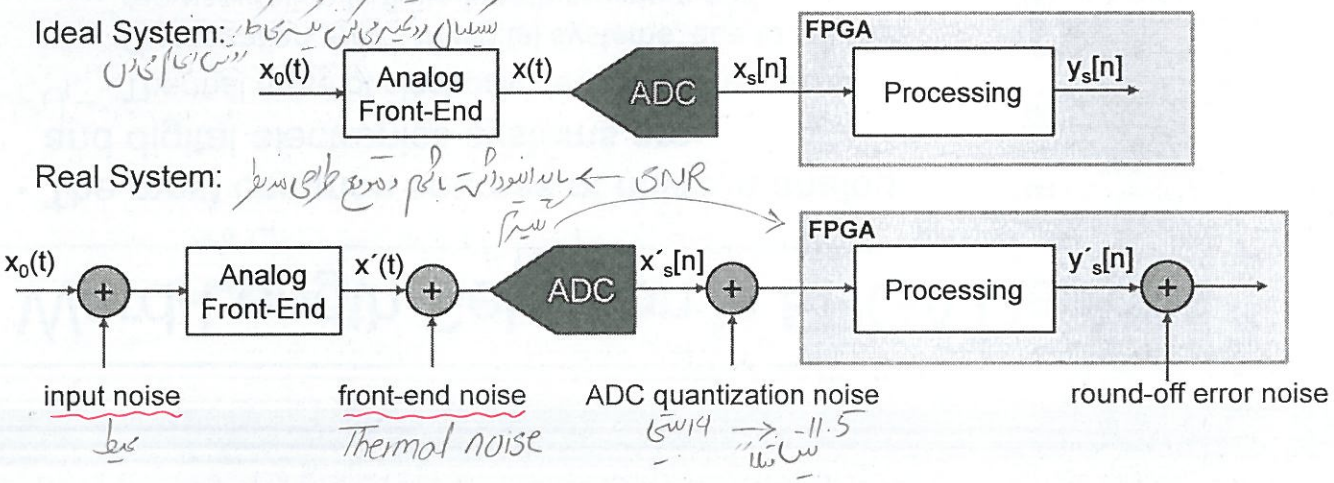
Thermal noise model of a resistor

Input Word-Length Selection Procedure

How to determine the input noise level and internal register lengths?

1. Thermal noise (noise figure) calculation of all analog elements, up to the digital units (beyond the scope of this course)
2. Calculating the ENOB of the ADC
3. Selecting the processing register lengths such that the internal FPGA quantization errors are below (or at the same level as) the above items

Note: For pure digital processing or when the input noise level is unknown for the digital designer, the noise level can be assumed to be half the input register LSB



Input Word-Length Selection Procedure (continued)

- Note: As far as the FPGA designer is concerned, the input noise and the analog front-end noise can usually be lumped in the ADC quantization noise (as factors that reduce the input ENOB)
- For example, with a 16-bit ADC, the 3 LSB may fluctuate due to the different noise factors (input noise, device thermal noise, ADC quantization error)

A typical analog front-end with various sources of noise

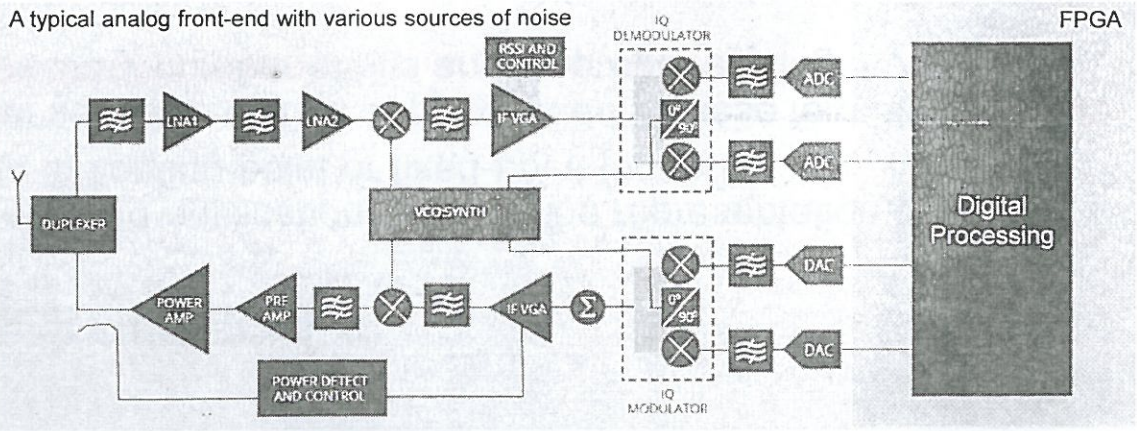
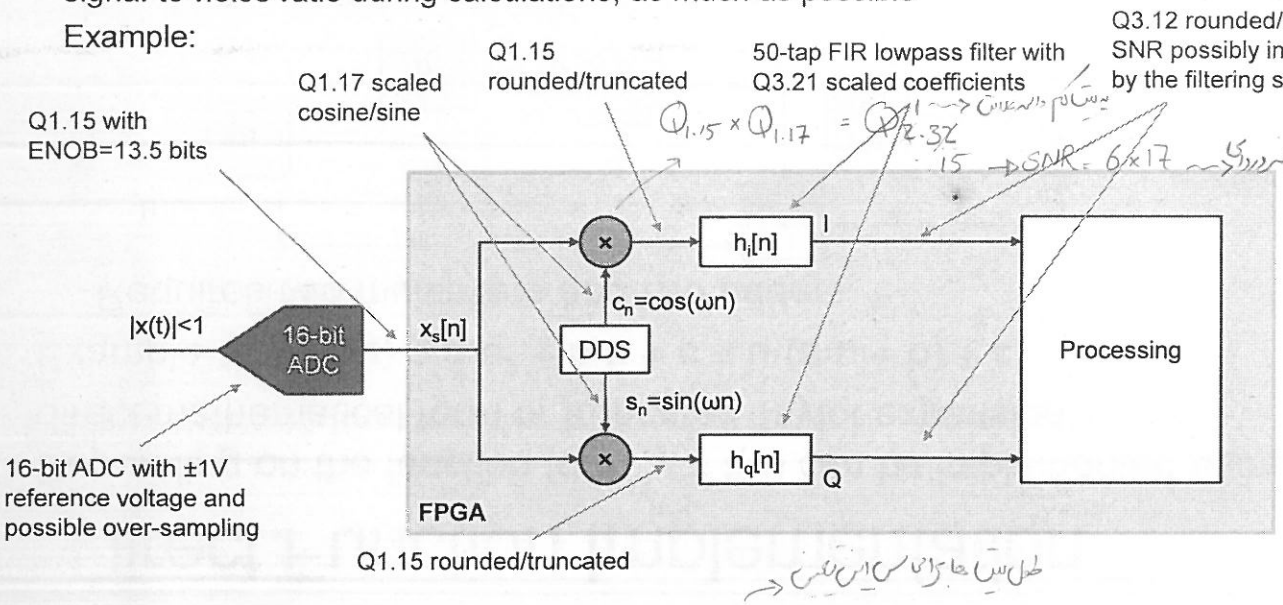


Image adapted from: http://www.azcom.it/index.php/services/rf-design/analog-front-end-afe/

Intermediate Word-Length Selection in FPGA Designs

Intermediate calculation word-length selection follows similar rules: "try to preserve the signal-to-noise ratio during calculations, as much as possible"

Example:



Note 1: The internal register lengths are selected according to the input noise level and ENOB, not the ADC number of bits
Note 2: The SNR can be increased due to the processing gain. For example, remember the SNR improvement due to over-sampling noted in the previous section

ARBITRARY WAVEFORM GENERATION

Waveform Generation

The calculation/generation of arbitrary functions/waveforms of the form y = f(x) is required in many computational and signal processing applications. We study several methods for this purpose:

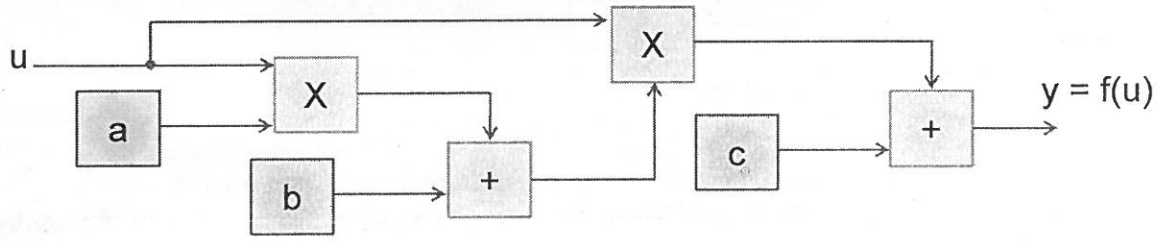
- Arbitrary functions:
  - Direct Implementations (functional calculation)
  - Lookup-Tables & Interpolated Lookup-Tables
- Special functions:
  - CORDIC machines
- Periodic functions:
  - NCO and Periodic Waveform Generators
  - Recursive Oscillators
- Random signal:
  - LFSR

Direct Function Implementation

Depending on the function form, y = f(u) can be implemented using its direct mathematical form or truncated Taylor expansion:

Example 1: $y = f(u) = a \cdot u^2 + b \cdot u + c = u \cdot (a \cdot u + b) + c$

Requires two multipliers and two adders



Example 2: $y = f(u) \approx f(a) + f'(a) \cdot (u - a)$

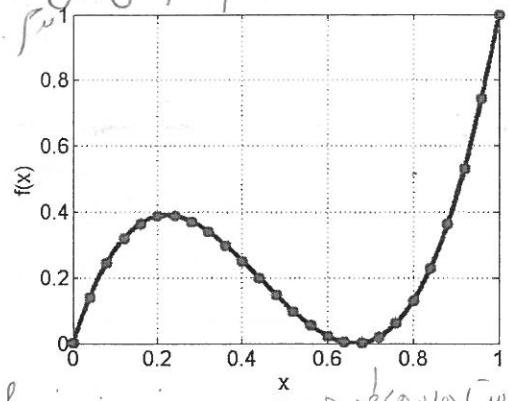
Requires a multiplier and two adders for a first-order approximation

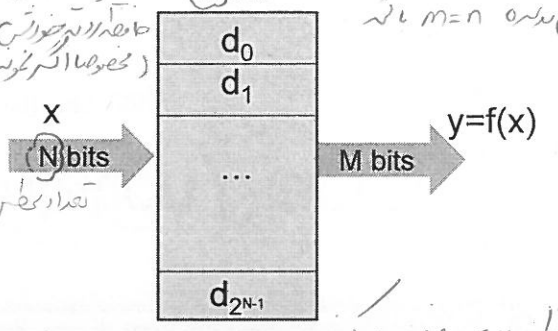
Note 1: The implementation of the direct form of a function on FPGA is simplified when the expansion coefficients are constants or powers of 2.

Note 2: The approximated Taylor expansion is only accurate for smooth functions

Functional Implementation by Lookup Tables (LUT)

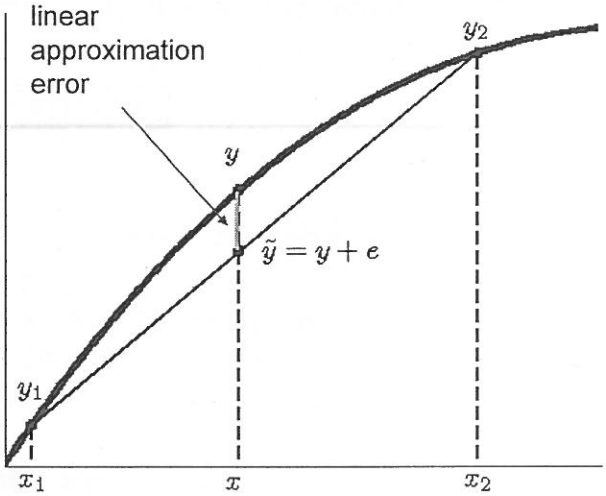
- In order to implement y = f(x) over a finite domain, one may pre-calculate and store the values of y over the entire domain of x in a memory. The values of x can next be used as the address bus of the memory during runtime.
- LUT-based implementation of functions is applicable for arbitrary functions (not necessarily smooth); but requires a lot of memory when x has many bits.
- The accuracy of this method depends on the function form, and the number of bits assigned to x (N) and y (M)





Functional Implementation by Interpolated LUT

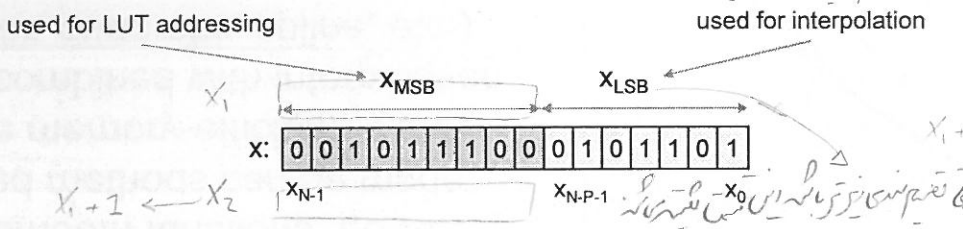
- For smooth functions, LUT-based methods can be made more memory-efficient, if they are combined with interpolation (linear, quadratic, spline, etc.)
- For example, in linear interpolation, we interpolate between successive values of the LUT with appropriate weights:



$$y \approx \frac{(x - x_1)y_2 + (x_2 - x)y_1}{(x_2 - x_1)} = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1)$$

Interpolated LUT Implementation

- Linear interpolated LUTs can be implemented very efficiently using a single or dual-port LUT and minor computations.
- Idea: Suppose that x has N bits, which means that an LUT of length 2^N is required for its complete implementation. However, if one uses the P MBS bits of x (P < N) for addressing a 2^P points LUT, the N-P LSB bits of x could be used for linear interpolating between two successive samples of the P-point LUT.



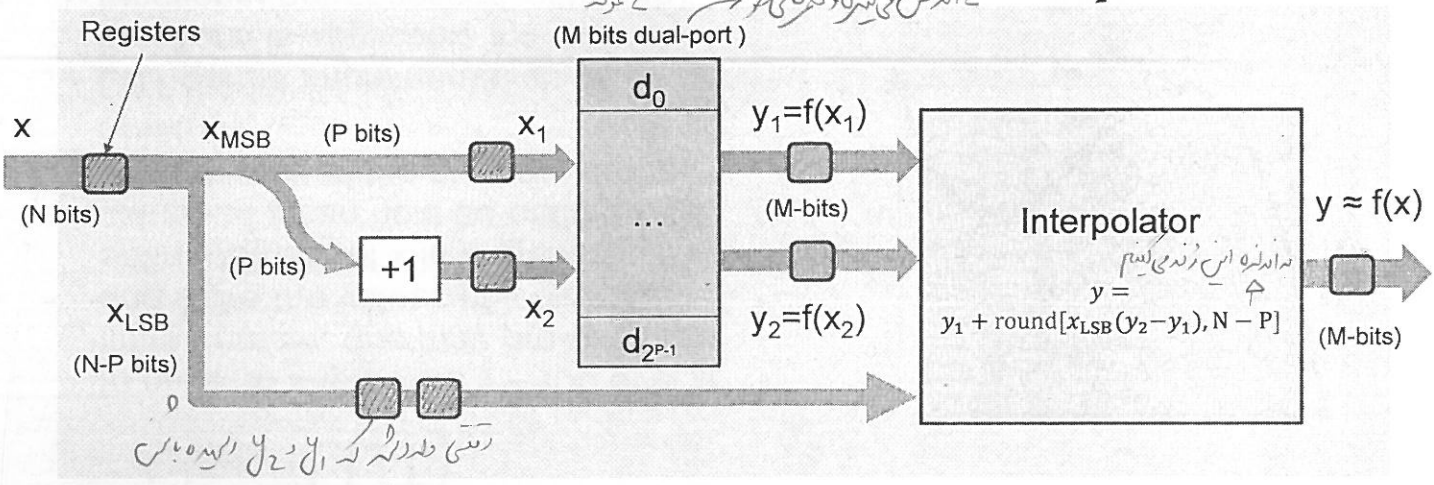
- Therefore, we can write $y_1 = f(x_{MSB})$, $y_2 = f(x_{MSB} + 1)$ and calculate the first-order interpolation as follows:

$$y = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1) = y_1 + \frac{x_{LSB} \times (y_2 - y_1)}{2^{N-P}}$$

This division requires only a shift (rounding)

Interpolated LUT Implementation Diagram

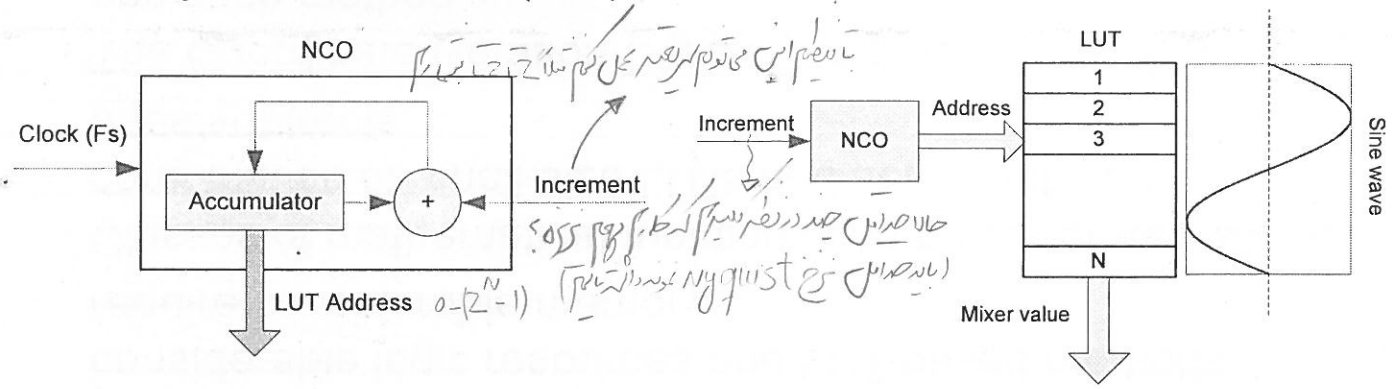
The overall block-diagram of an interpolated LUT of length N using a two-port LUT of length P for pipelined FPGA-based implementation is as follows:



Note: Similar ideas can be implemented using quadratic and spline interpolations. See the following reference for further ideas and general LUT-based methods: Behrooz, P. (2000). Computer arithmetic: Algorithms and hardware designs. *Oxford University Press*, Chapter 24

Periodic Signal Generators

An efficient method for generating periodic signals is to combine an LUT with a numerically controlled oscillator (NCO)



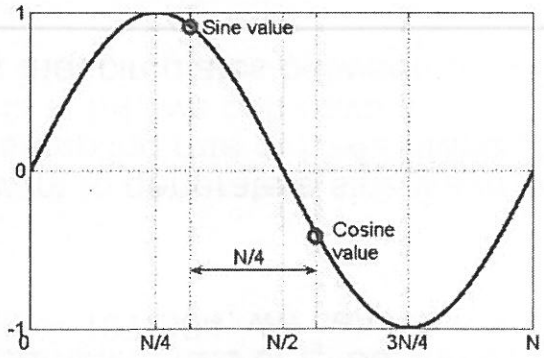
Example: In order to generate a sinusoidal signal with frequency f_0 in a sampling rate f_s, using an LUT of length N, the NCO increment can be found as follows:

$$* \text{inc} = \frac{N f_0}{f_s}$$

Note: As a sinusoidal signal, inc should be smaller than $N/2$ to fulfill the Nyquist sampling rate.

Notes on Periodic Signal Generators

1. Sine and Cosines can be produced using a single two-port LUT with ¼ of initial address offset between the two ports.



2. Sine/cosine generation is precise (with no phase errors), if the desired frequency (f_0), sampling frequency (fs), LUT length (N) and LUT address increment (inc) satisfy:

$$\frac{f_0}{f_s} = \frac{\text{inc}}{N}$$

Sine Wave Generator Examples

- Example 1: We want to generate a sine wave with frequency f_0=10.7MHz at a sampling rate of f_s=38.4MHz. Noting that 10.7MHz/38.4MHz = 107/384, we can have a 384-point LUT with inc=107.
- Example 2: We want to generate a sine wave with frequency f_0=10.7MHz at a sampling rate of f_s=42.8MHz. Noting that 10.7MHz/42.8MHz = 1/4, we can have a 4-point LUT, which is basically a 4-state selector that circulates between 0,+1,0, and -1 (no LUT needed).
- Example 3: We want to make a direct digital synthesizer (DDS) for generating sine waves at a sampling frequency of f_s=100MHz. The DDS should be able to synthesize frequency from DC to 50MHz (Nyquist rate), with frequency steps of Δf=100kHz. A LUT of length N=1000 is required.

CORDIC Machines

- The direct implementation of arbitrary functions requires considerable logic resources and LUT-based methods require considerable memory.
- Classes of mathematical functions can be generated with a combination of small-size LUTs and set of shifts and adds/subtracts.
- The Coordinate Rotation Digital Computer (CORDIC) is one such method
- The CORDIC machine was invented in 1956 by Jack E. Volder to be used in B58 bomber's navigation system for accurate real-time digital calculations

Volder's CORDIC Algorithm

Volder's original algorithm is a set of recursive multiplier-free equations:

$$\begin{cases} x_{n+1} = x_n - d_n y_n 2^{-n} \\ y_{n+1} = y_n + d_n x_n 2^{-n} \\ z_{n+1} = z_n - d_n \arctan 2^{-n} \end{cases}$$

where

- $\arctan 2^{-n}$ are pre-calculated and stored in a LUT
- $d_n = \text{sign}(z_n)$ (+1 if $z_n \geq 0$ and −1 if $z_n < 0$)

If $|z_n| < \theta_{max} = \sum_{n=0}^{\infty} \arctan 2^{-n} = 1.7432866\ldots$, it can be shown that:

$$\lim_{n\to\infty} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = K \times \begin{pmatrix} x_0 \cos z_0 - y_0 \sin z_0 \\ x_0 \sin z_0 + y_0 \cos z_0 \\ 0 \end{pmatrix}$$

where $K = \prod_{n=0}^{\infty} \sqrt{1 + 2^{-2n}} = 1.6467603\ldots$

CORDIC Machine Principles

- The non-restoring decomposition of an arbitrary angle:

$$\theta = \sum_{k=0}^{\infty} d_k w_k \,, d_k = \pm 1, w_k = \tan^{-1}(2^{-k})$$

The nonrestoring algorithm:

The following algorithm converges to θ:

$$t_0 = 0$$

$$t_{n+1} = t_n + d_n w_n$$

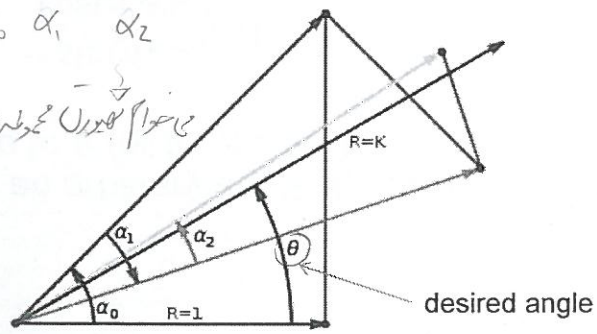
$$d_n = \begin{cases} 1 \text{ if } t_n \leq \theta \\ -1 \text{ otherwise} \end{cases}$$

Or in the reverse direction:

$$t_0 = \theta$$

$$t_{n+1} = t_n - d_n w_n$$

$$d_n = \begin{cases} 1 \text{ if } t_n \geq 0 \\ -1 \text{ otherwise} \end{cases}$$



The CORDIC Algorithm in Circular Rotation Mode

- According to the restoring algorithm, for an arbitrary angle θ, successive rotations can be used to rotate from zero to θ (or from θ to 0):

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} \cos(d_n w_n) & -\sin(d_n w_n) \\ \sin(d_n w_n) & \cos(d_n w_n) \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$

or

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \cos(w_n) \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}$$

- The term $\cos(w_n) = 1/\sqrt{1+2^{-2n}}$ is the only required multiplication, which can be omitted, as it does not alter the rotation angles and only changes the vector magnitudes.
- Alternatively, depending on the number of iterations P, $A = 1/\prod_{n=0}^{P}\sqrt{1+2^{-2n}}$ can be compensated as a constant multiplier.

Alternative Forms of the CORDIC Algorithm

- Alternative modes of the CORDIC algorithm include:

Type	m	w_k	$d_n = \text{sign} z_n$ *(Rotation Mode)	$d_n = -\text{sign} y_n$ *(Vectoring Mode)
circular	1	$\arctan 2^{-k}$	$x_n \to K(x_0 \cos z_0 - y_0 \sin z_0)$ $y_n \to K(y_0 \cos z_0 + x_0 \sin z_0)$ $z_n \to 0$	$x_n \to K\sqrt{x_0^2 + y_0^2}$ $y_n \to 0$ $z_n \to z_0 + \arctan \frac{y_0}{x_0}$
linear	0	2^{-k}	$x_n \to x_0$ $y_n \to y_0 + x_0 z_0$ $z_n \to 0$	$x_n \to x_0$ $y_n \to 0$ $z_n \to z_0 + \frac{y_0}{x_0}$
hyperbolic	-1	$\tanh^{-1} 2^{-k}$	$x_n \to K'(x_1 \cosh z_1 + y_1 \sinh z_1)$ $y_n \to K'(y_1 \cosh z_1 + x_1 \sinh z_1)$ $z_n \to 0$	$x_n \to K'\sqrt{x_1^2 - y_1^2}$ $y_n \to 0$ $z_n \to z_1 + \tanh^{-1} \frac{y_1}{x_1}$

$$\begin{cases} x_{n+1} = x_n - m d_n y_n 2^{-\sigma(n)} \\ y_{n+1} = y_n + d_n x_n 2^{-\sigma(n)} \\ z_{n+1} = z_n - d_n w_{\sigma(n)}, \end{cases}$$

Circular ($m = 1$)	$\sigma(n) = n$
Linear ($m = 0$)	$\sigma(n) = n$
Hyperbolic ($m = -1$)	$\sigma(n) = n - k$ where k is the largest integer such that $3^{k+1} + 2k - 1 \le 2n$

Note: The implementation of CORDIC on FPGA requires attention in word length selection and number representation

Reference and further reading: Muller, Jean-Michel. *Elementary functions*. Birkhäuser Boston, 2006. Chapter 7

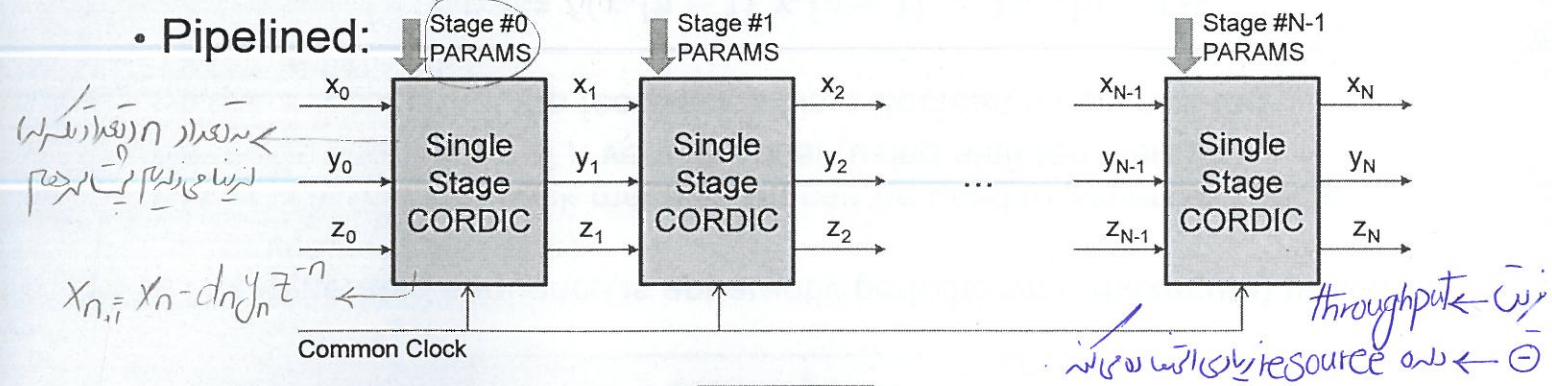
CORDIC Implementation on FPGA

- Before implementation, the CORDIC parameters need to be set:

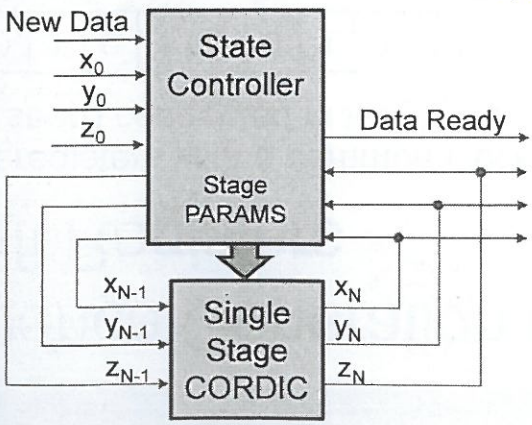
1. Choose the CORDIC mode
2. Set the input and output lengths and $Q_{m.n}$ data format
3. Find the required number of CORDIC iterations by simulation, such that the calculation error is smaller than the LSB of the selected word lengths
4. Implement the CORDIC machine using pipelining or resource sharing (or a combination of both)

CORDIC Implementation on FPGA (continued)

- Pipelined:

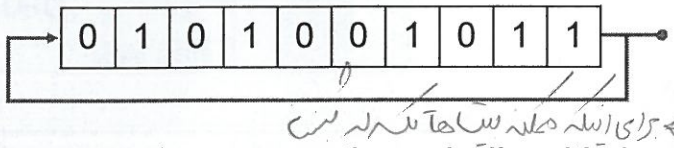


- Resource Shared:



Periodic Sequence Generation using Feedback Shift Registers

- Consider a chain of N registers with a common clock and arbitrary initial values (known as the seed) connected in feedback:

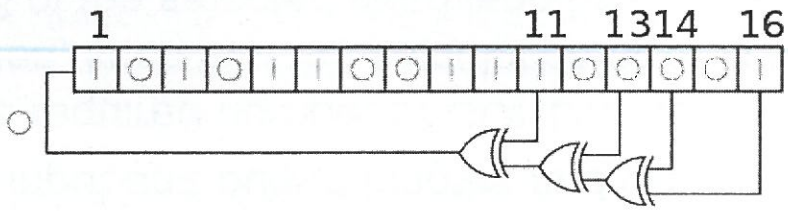


- The generated sequence is apparently periodic with (maximum) period N samples (N/f_s seconds)
- In FPGA, this feedback mechanism can be used to generate special periodic sequences at a very low cost (using shift registers)
- Next, suppose that the feedback bit is a Boolean function of the intermediate bits:

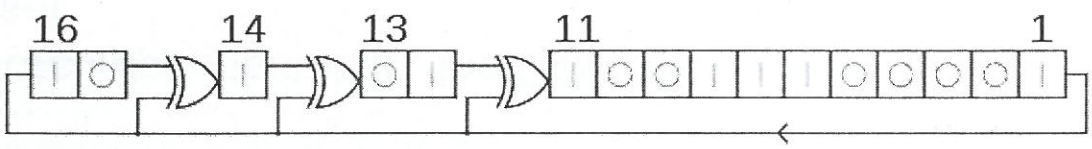
$$\begin{cases} x_0[n] = f(x_0[n-1], x_1[n-1], \dots, x_{N-1}[n-1]) \\ x_1[n] = x_0[n-1] \\ \dots \\ x_{N-1}[n] = x_{N-2}[n-1] \end{cases}$$

Periodic Sequence Generation using Feedback Shift Registers (continued)

Examples:



A 16-bit Fibonacci LFSR



A 16-bit Galois LFSR

Pseudo Random Number Generation using LFSR

- Linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function (e.g. XOR, XNOR, etc.) of its previous state
- The initial value of the LFSR is called the seed
- LFSRs are deterministic FSM, as the output stream is completely determined by its initial state and the linear function
- Since the register has a finite number of states, LFSR has a periodic cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that are pseudo-random (have a very long period).
- An N-bit LFSR is called maximum-length, if it cycles over all 2^N possible states except 0 (from which it would not exit from)

Ref: https://en.wikipedia.org/wiki/Linear-feedback_shift_register

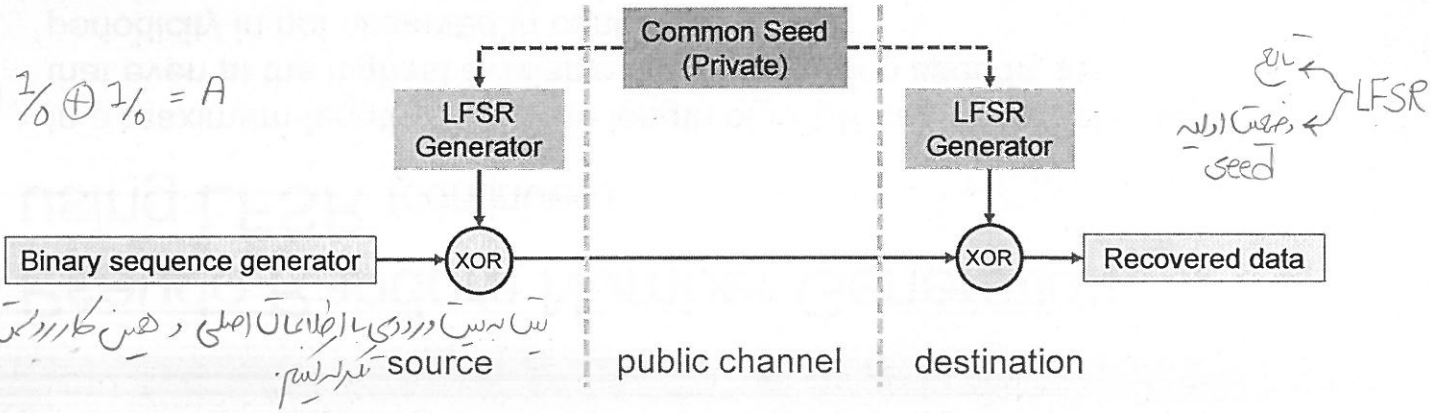
Pseudo Random Number Generation using LFSR (continued)

- In a maximum-length LFSR The length of LFSR can be selected such that even at the highest available flip-flop clocking speeds, the periodicity in not observed in centuries!
- Example: A maximum-length LFSR of length 64 clocked at 1GHz, takes $(2^{64}-1)/1\text{GHz} \approx 585$ years to repeat itself!
- Moreover, with an appropriate choice of the LFSR length and the feedback function (also known as the LFSR polynomial), the generated sequence resembles a fully stochastic sequence, which passes all the statistical tests of stochastic white noise.
- In this case, the periodic sequence may only be repeated by having the initial seed.
- LFSR have profound mathematical bases with numerous applications in coding, security, numeric computation, etc.

Ref: See the following for a nice introduction on the mathematics behind LFSR (Galois Fields): http://inst.eecs.berkeley.edu/~cs150/sp03/handouts/15/LectureA/lec27-6up

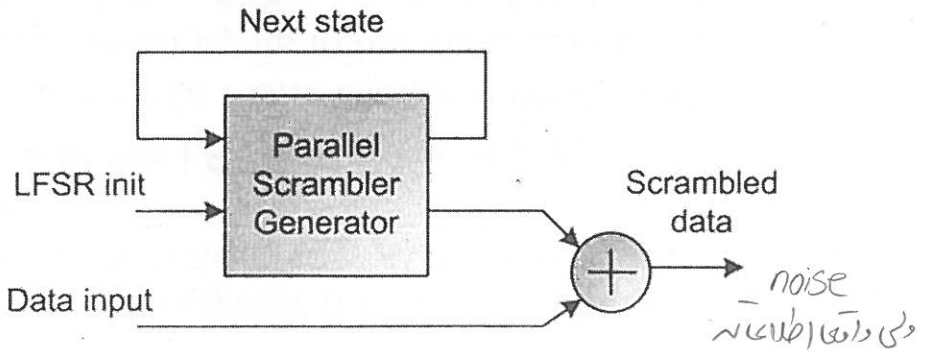
Other Applications of LFSR

1. Counters: LFSR can be used as extremely efficient counters (only requiring shift-registers and a few XOR), when the counting order is not important. For example for FSM encoding and micro-codes
2. Cyclic Redundancy Check (CRC): LFSR can be used to generate CRC for error detection and correction
3. Data Encryption/Decryption: LFSRs can be used for encryption of data transmitted over public channels



Other Applications of LFSR (continued)

4. Scramblers: Scramblers are used in many communication and storage protocols to randomize the transmitted data in order to remove long sequences of logic zeros and ones.

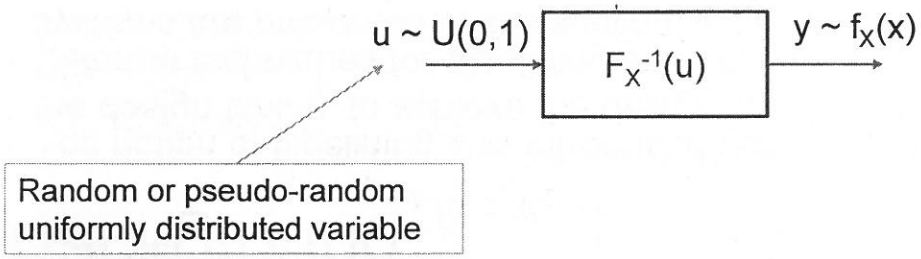


Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

Pseudo Random Numbers with Arbitrary Distributions

As noted before:

If a random variable (RV) x with a probability density function (pdf) $f_X(x)$ and cumulative distribution function (CDF) $F_X(x)$ passes a nonlinear memoryless system with a characteristics $u = F_X(x)$, the output u is uniformly distributed. Also, if a uniformly distributed RV u is given to $y = F_X^{-1}(u)$, the output has a distribution $f_X(\bullet)$.



Pseudo Random Signals with Arbitrary Spectral Color*(optional)

Alternative methods for generating signal/noise with arbitrary spectra include:

- Frequency modulation using fast frequency sweeps (e.g. using a Chirp signal)
- Bandpass filtering pseudo-random white noise
- Superposition of synthetic signals and noise

PIPELINING & DESIGN TIMING IMPROVEMENT TECHNIQUES

Background

→ throughput

- The notion of pipelining was introduced before, as a means of improving the design timing, to achieve the design constraints (clock speed)
- Different techniques for pipelining and timing improvement in FPGA systems are presented in this section, including:
  - Retiming
  - Re-pipelining
  - Cut-set retiming
  - C-slow retiming
  - Pipelining in feedback systems

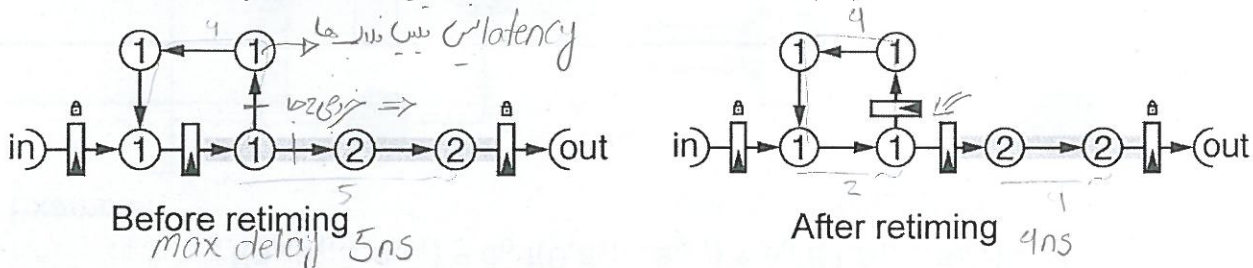
References:

- Hauck, Scott, and Andre DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Vol. 1. Elsevier, 2010, Chapter 18
- Khan, Shoab Ahmed. *Digital design of signal processing systems: a practical approach*. John Wiley & Sons, 2011, Chapter 7

Retiming

- Retiming consists of reducing the critical path (increasing the clock speed) by moving the pipeline registers to an "optimal position".

  Example: In the following, each circle denotes combination logic, with the number representing the combinational latency



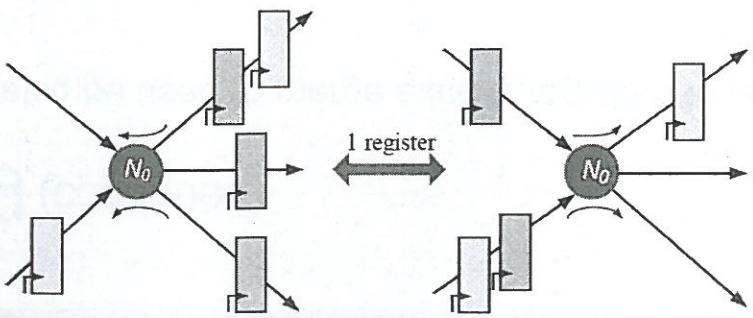
Before retiming

After retiming

- The objective of retiming is to automate this procedure in a systematic manner with concise algorithms, which 1) guarantee that the circuit's I/O transfer function is not changed and 2) can be implemented in CAD tools (for instance during the synthesis or technology mapping stages)
- Limitation: Retiming cannot improve the design clock speed beyond the optimal register placement

Retiming (continued)

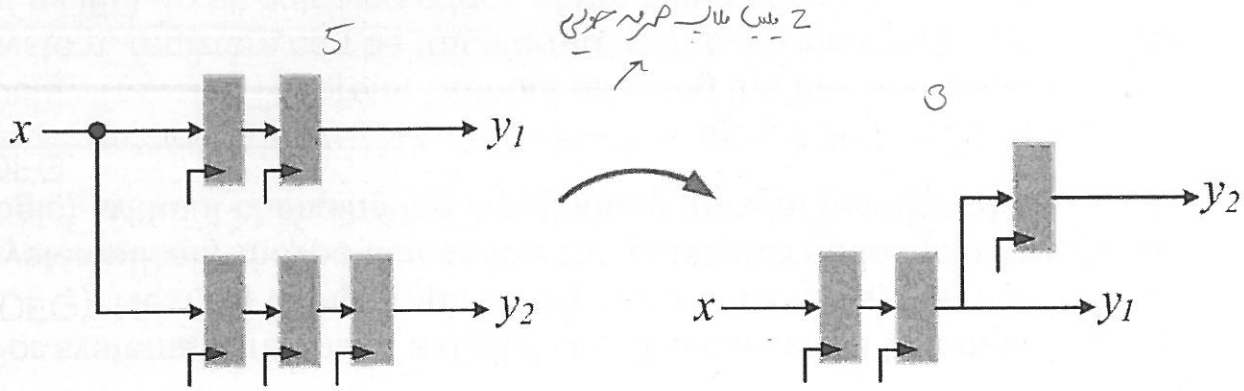
- For systematic retiming, a digital circuit is converted to a data flow graph (DFG). Next, by using graph theory based theorems, the registers are systematically moved across the computational nodes (combinational logic), without changing the input/output transfer function of the original DFG.
- Delay Transfer Theorem: "without affecting the transfer function of the system, registers can be transferred from each incoming edge of a node of a DFG, to all outgoing edges of the same node or vice versa" [Khan, 2011, p. 304].



Retiming (continued)

- Retiming can also be used to merge excess registers to reduce the area utilization.

  Example:

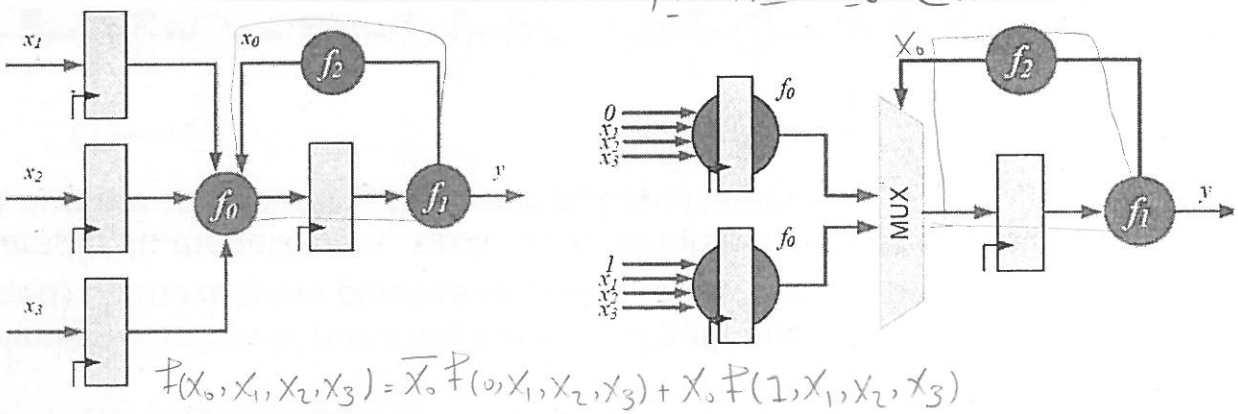


Shannon Decomposition Retiming

- The Shannon decomposition can be used to improve the timing of Boolean functions. Accordingly:

$$f(a_0,a_1,\ldots a_{N-1}) = \bar{a}_0 \cdot f(0,a_1,\ldots a_{N-1}) + a_0 \cdot f(1,a_1,\ldots a_{N-1})$$

  Example:



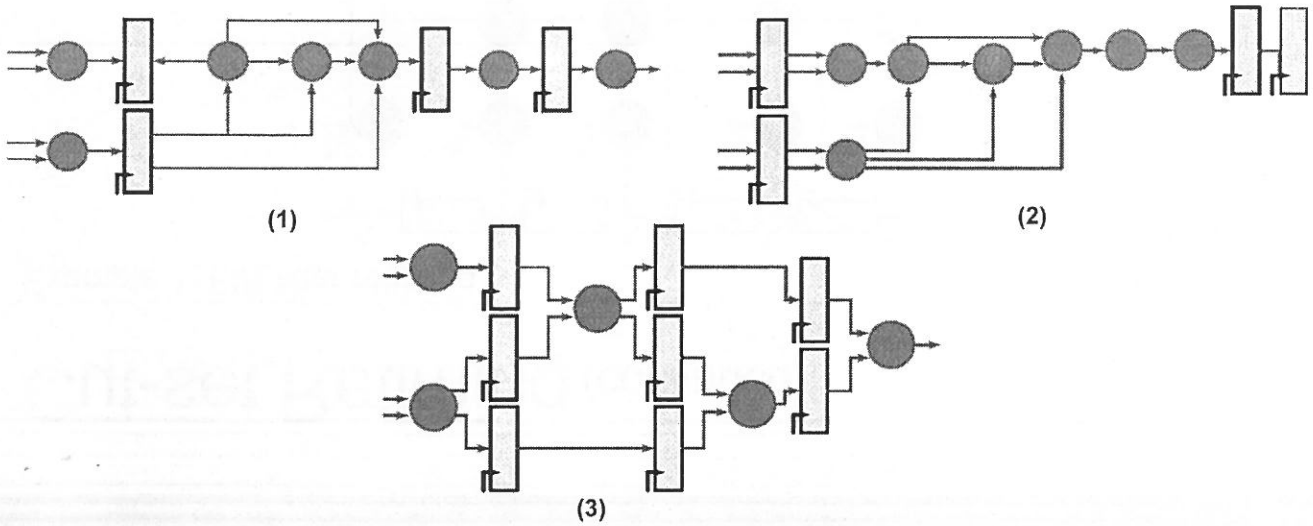
$$F(X_0,X_1,X_2,X_3) = \bar{X}_0 F(0,X_1,X_2,X_3) + X_0 F(1,X_1,X_2,X_3)$$

Note: The Shannon decomposition is specifically useful for FPGA-based designs, which are implemented on fixed-input LUTs

Peripheral Retiming

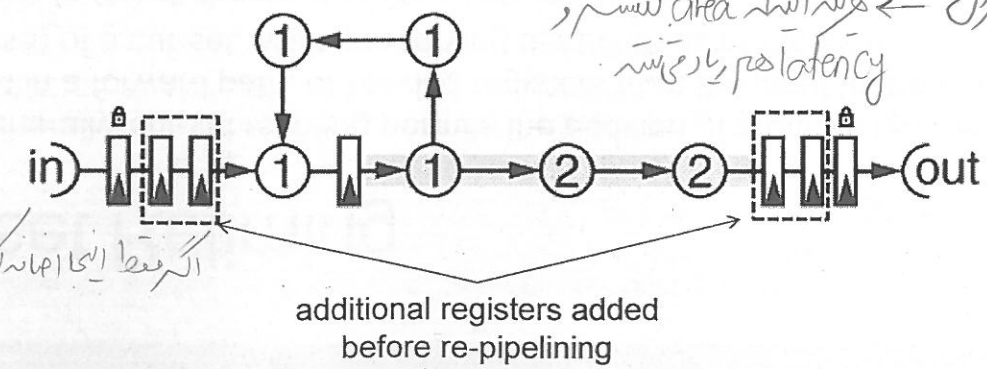
- In this technique: 1) all the internal registers are shifted to the input or output of the design; 2) the combinational logic is simplified; finally 3) the registers are pushed to their optimal position by conventional retiming.

  Example:



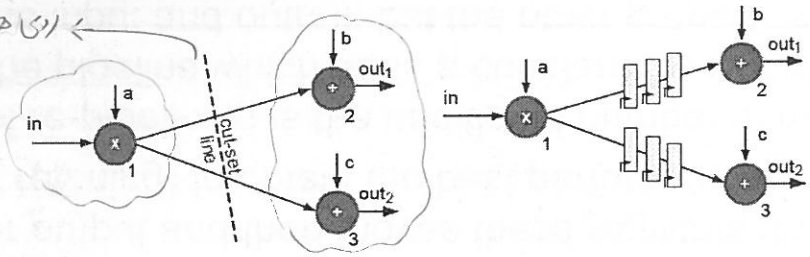
Re-pipelining

- In feed-forward designs, re-pipelining adds additional registers at the input or output and then moves these registers across the design (by retiming) to obtain the best performance.
- The cost of re-pipelining is the additional number of registers added to the pipeline which adds a constant clock latency between the input and output; but the other properties of the design are preserved.

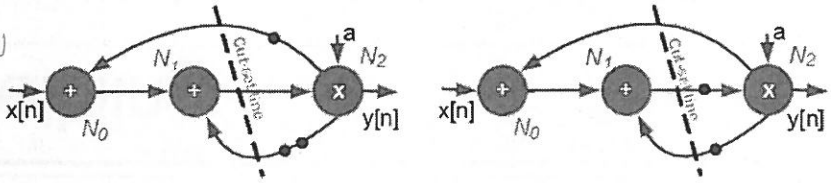


Cut-set Retiming

- More generally, cut-set retiming permits the addition of arbitrary number of registers in a forward path, or moving registers from the input to the output (or vise versa) of a cut-set, while preserving the I/O transfer function.
- Reminder: In Graph theory, a cut is a virtual partitioning of the edges of a graph into two disjoint subsets, known as cut-sets.



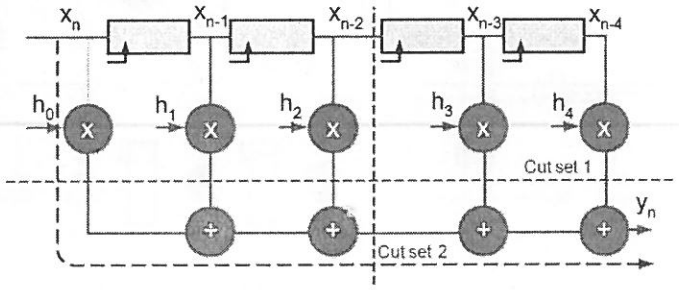
adding registers in feed-forward cut-sets



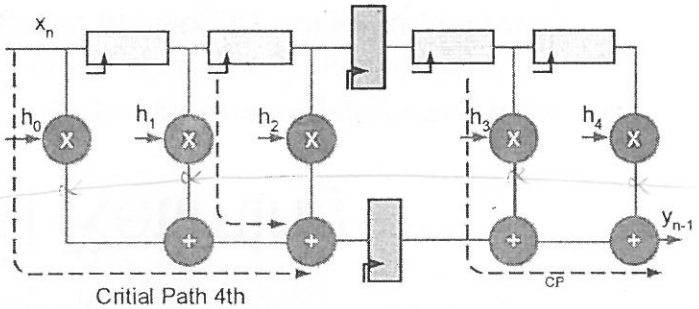
moving registers from cut-set output to cut-set inputs

Cut-set Retiming (continued)

Example 1: FIR filter retiming



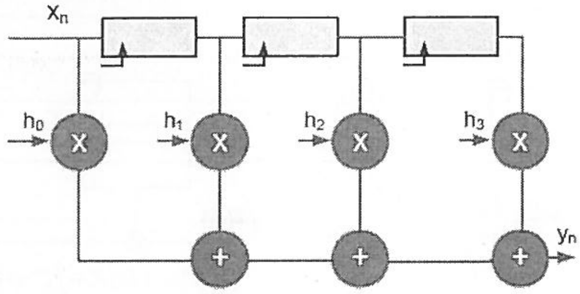
Two possible cut-sets

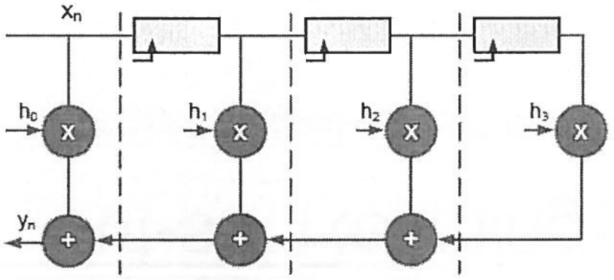


Re-pipelining across feed-forward cut-set 2

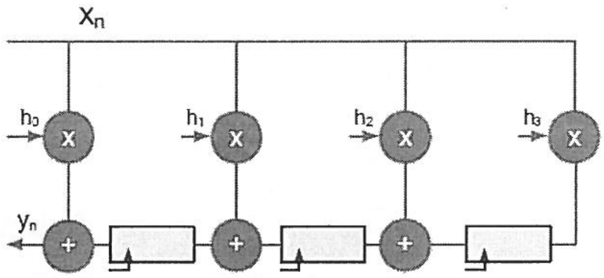
Cut-set Retiming (continued)

Example 2: FIR filter retiming, second approach: multiple cut-set retiming





Three cut-sets with feedback paths

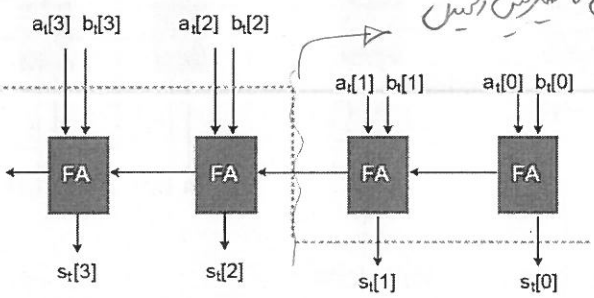


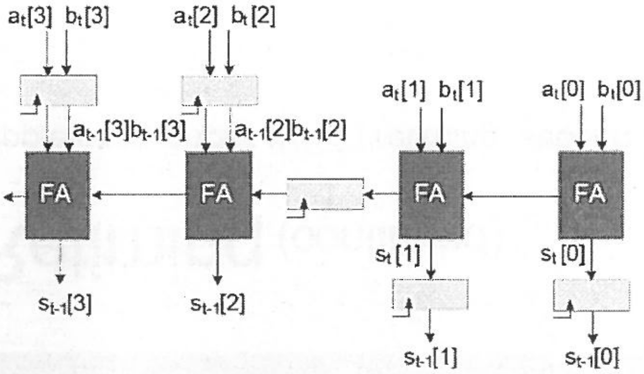
After cut-set retiming

Cut-set Retiming (continued)

Example 3: 4-bit ripple carry adder (RCA) retiming

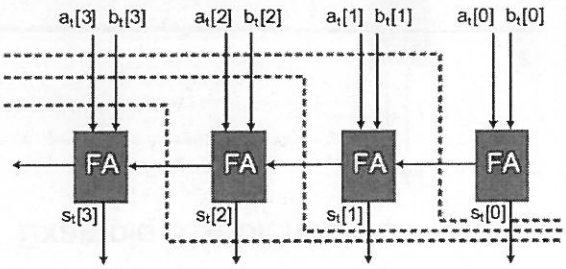
چرا اینجا بگیر شد؟ چون تأخیرش زیاده
جدا می‌افته.

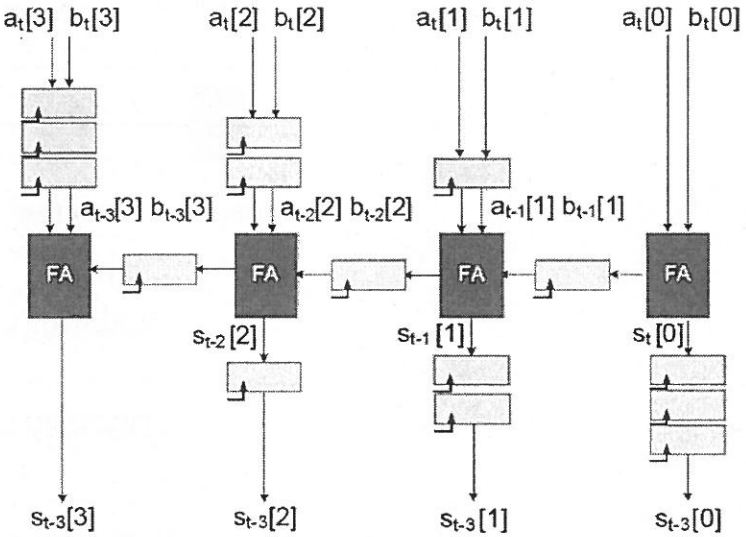




Cut-set Retiming (continued)

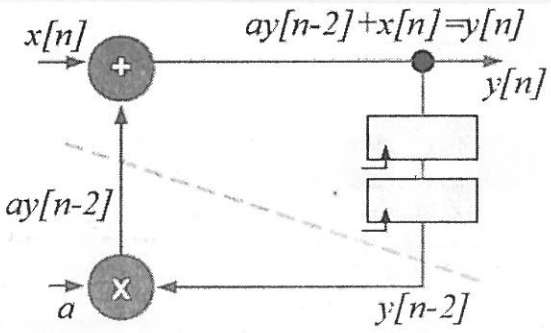
Example 4: 4-bit ripple carry adder (RCA) retiming; second approach

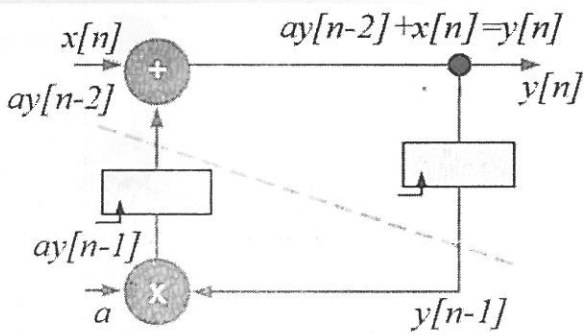




Cut-set Retiming (continued)

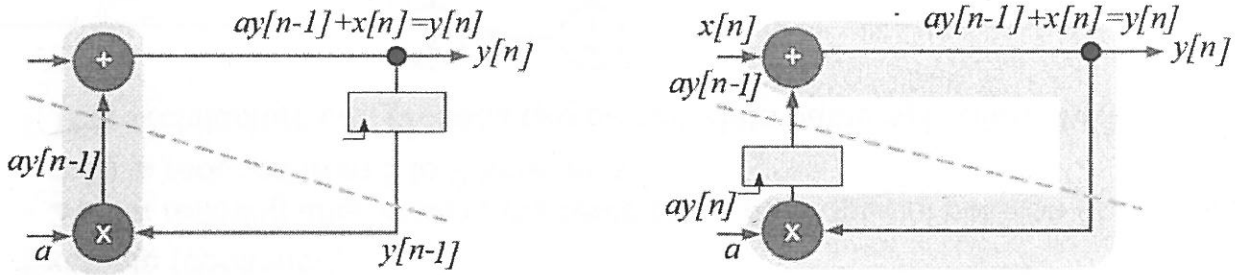
Example 5: Second-order IIR filter





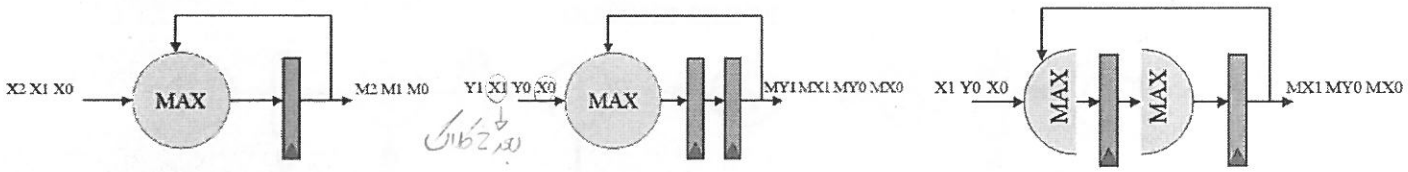
Cut-set Retiming (continued)

- Cut-set retiming does not always result in an improved timing.
- Example: In a first-order IIR filter, the critical path is not changed by cut-set retiming of the feedback loop.



C-Slow Retiming

- C-slow retiming consists of replicating all the registers of a synchronous design C times, followed by moving the registers (conventional retiming), or by splitting the circuit into C distinct parallel paths which multiplex and switch between the input data and results.



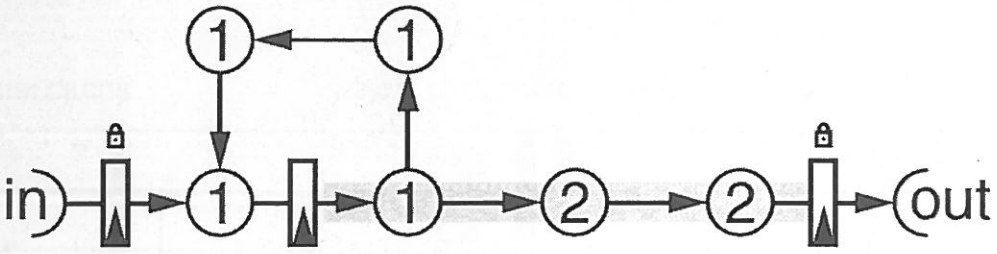
original circuit | 2-slow counterpart circuit | 2-slow circuit after retiming

Note: The design interleaves between two computations (2-slow): on the first clock cycle, it accepts the first input for the first data stream; on the second clock cycle, it accepts the first input for the second stream, and on the third it accepts the second input for the first stream. Due to the interleaved nature of the design, *the two streams of execution will never interfere* (on odd clock cycles, the first stream of execution accepts input; on even clock cycles, the second stream accepts input).

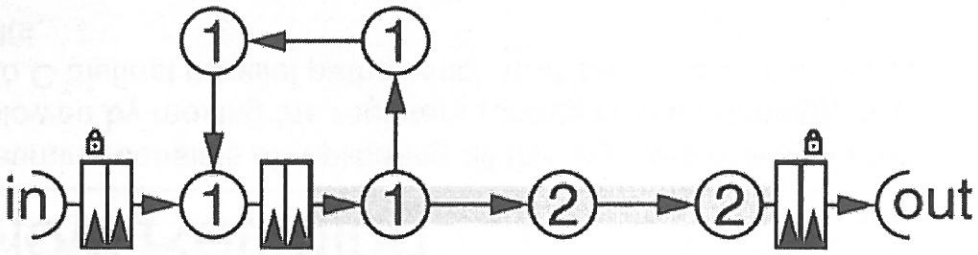
stream → interleaving

C-Slow Retiming (continued)

Example:



original circuit

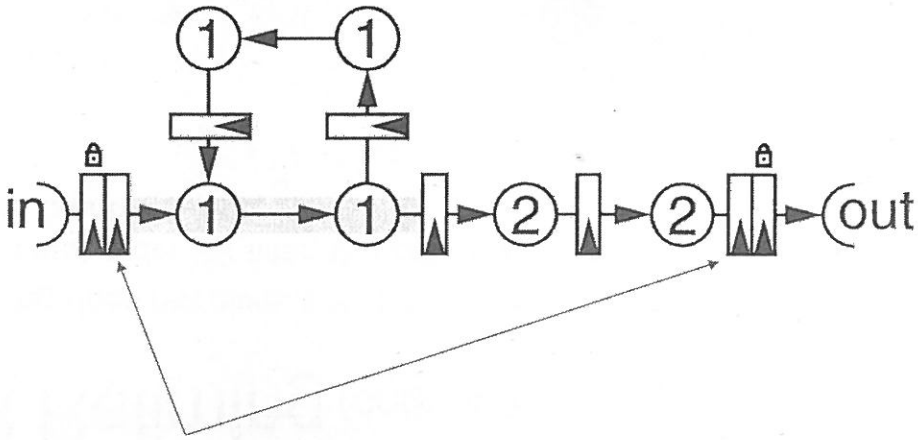


2-slow counterpart circuit

C-Slow Retiming (continued)

Example (continued):

- 2-slow retiming after moving the registers to their optimal position (the critical path is reduced from 5 to 2 time units):
- This architecture can process two parallel data paths with interleaved data



excess feed-forward registers can be eliminated after retiming

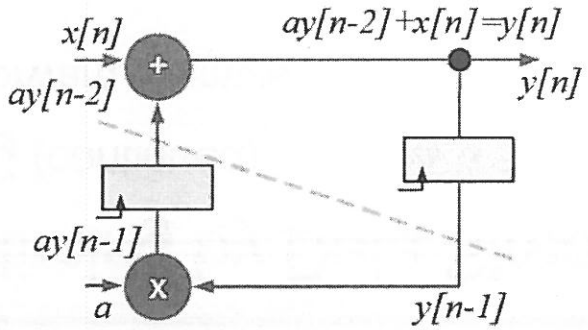
C-Slow Retiming (continued)

- Example: A single 2-slow retimed IIR filter architecture can be used to process the real and imaginary parts of a complex-valued digital filter by interleaving the real and imaginary parts of the input:

$$y_r[n] + j\, y_i[n] = h[n]*(x_r[n] + j\, x_i[n]) = h[n]*x_r[n] + j\, h[n]x_i[n]$$

$x_r[0]$	$x_i[0]$	$x_r[1]$	$x_i[1]$	$x_r[2]$	$x_i[2]$	$x_r[3]$	...

$y_r[0]$	$y_i[0]$	$y_r[1]$	$y_i[1]$	$y_r[2]$	$y_i[2]$	$y_r[3]$	...



C-Slow Retiming by Data Stream Interleaving

- The disjoint data stream property of C-slow retiming can be used to obtain parallel hardware threads, which interleave the input data stream between C identical circuits, each working at 1/C of the input clock rate and finally multiplexing the results back together. This method is referred to as unfolding in some textbooks.
- The idea is related to loop unrolling used for optimizing for-loops in multicore processors and GPUs
- The complementary method is hardware folding (hardware reuse), which uses a single hardware and a scheduler (FSM controller) to reduce the hardware size.
- Note: Systematic and *ad hoc* retiming and resource sharing may additionally be used to improve the area and timing performance of the design.

C-Slow Retiming by Data Stream Interleaving (continued)

Algorithm: Any DFG can be unfolded by an unfolding factor J using the following two steps:

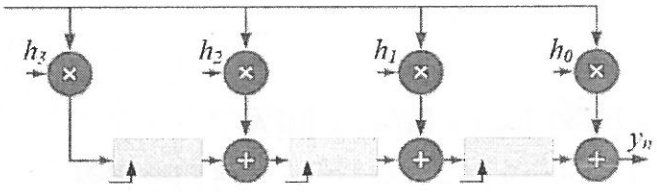
S_0) To unfold the graph, each node U of the original DFG is replicated J times as $U_0,\ldots, U_{J-1}$ in the unfolded DFG.

S_1) For two connected nodes U and V in the original DFG with w delays, draw J edges such that each edge j (= 0,…, J-1) connects node U_j to node $V_{(j+w)\%J}$ with floor[(j+w)/J] delays.

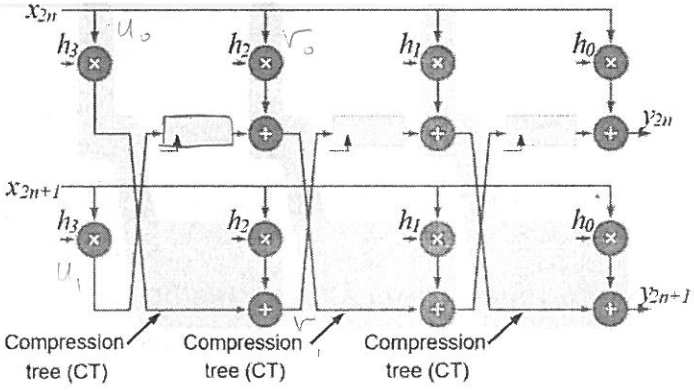
Ref: [Khan, 2011] p. 349

C-Slow Retiming by Data Stream Interleaving (continued)

Example: Feed-forward example



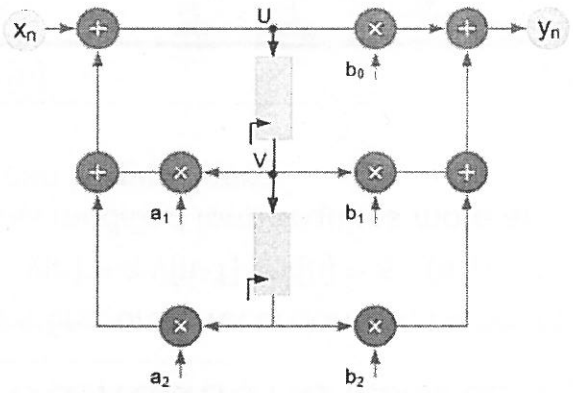
Original circuit



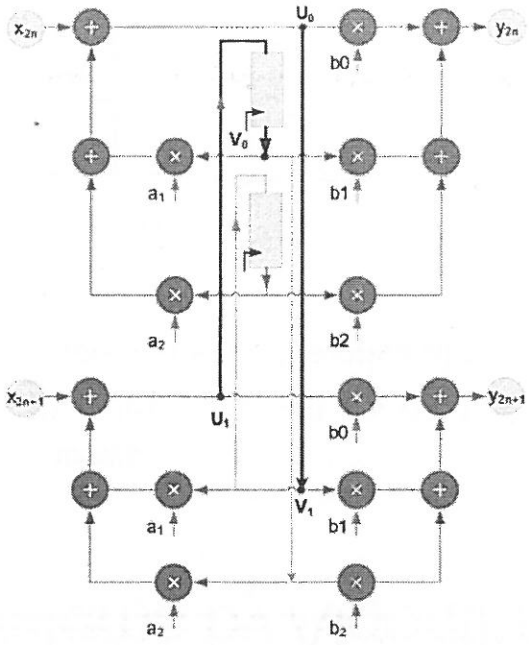
Unfolded system (2-fold)

C-Slow Retiming by Data Stream Interleaving (continued)

Example: Feedback systems



Original circuit

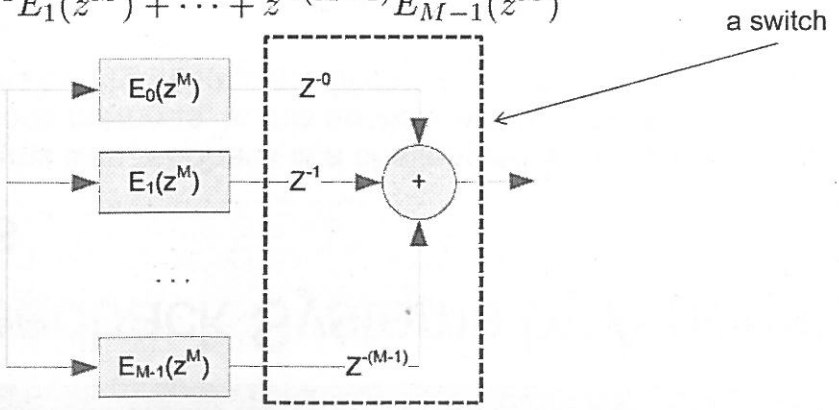


Unfolded system (2-fold)

C-Slow Retiming by Data Stream Interleaving Example* (optional)

- Example: Polyphase filter Implementation

$$
\begin{aligned}
H(z) &= \sum_{n=0}^{\infty} h[n]z^{-n} = h_0 + h_1 z^{-1} + h_2 z^{-2} + \cdots \\
&= (h_0 + h_M z^{-M} + h_{2M} z^{-2M} + \cdots) \\
&+ z^{-1}(h_1 + h_{M+1} z^{-M} + h_{2M+1} z^{-2M} + \cdots) + \cdots \\
&+ z^{-(M-1)}(h_{M-1} + h_{2M-1} z^{-M} + h_{3M-1} z^{-2M} + \cdots) \\
&= E_0(z^M) + z^{-1} E_1(z^M) + \cdots + z^{-(M-1)} E_{M-1}(z^M)
\end{aligned}
$$

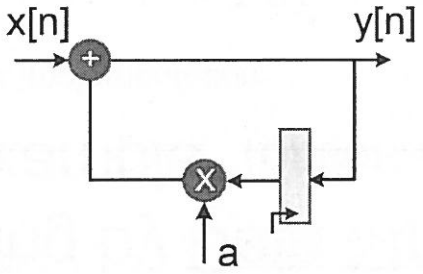


Pipelining Feedback Systems by Algorithmic Modifications

- Pipelining digital systems with feedback is a challenging issue and is not always solved using the previous methods. In this section, we study a few techniques for pipelining such systems by algorithmic modifications, using a simple case study.

Example: Consider a first-order recursion $y[n] = a \cdot y[n-1] + x[n]$.

- Such equations appear in many applications, e.g., infinite-impulse response (IIR) filters in signal processing
- The multiplication is problematic for pipelining, since the result of $a \cdot y[n-1]$ is needed for calculating $y[n]$ before the next clock edge arrives



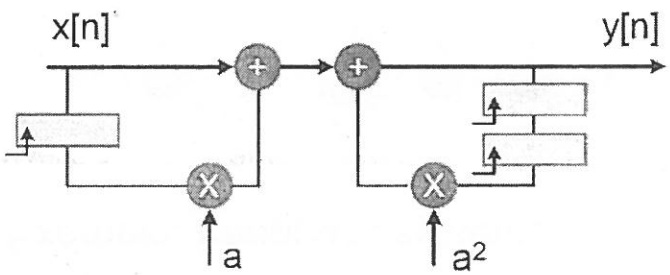
Solution?

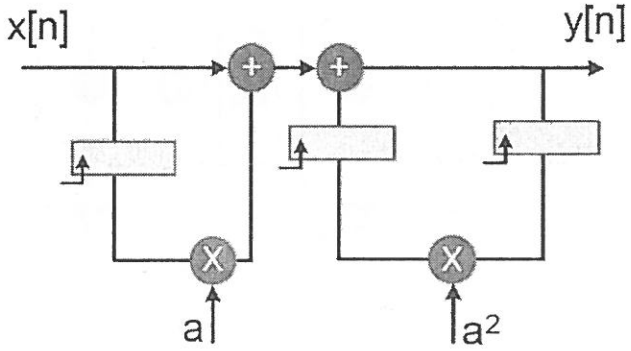
Pipelining Feedback Systems by Algorithmic Modifications (continued)

- The first-order recursion can be rewritten as follows:

$$y[n] = a \cdot y[n-1] + x[n] = a \cdot (a \cdot y[n-2] + x[n-1]) + x[n] = a^2 \cdot y[n-2] + a \cdot x[n-1] + x[n]$$

- This modified form requires more architecture (compared to the original form); but it can be pipelined:



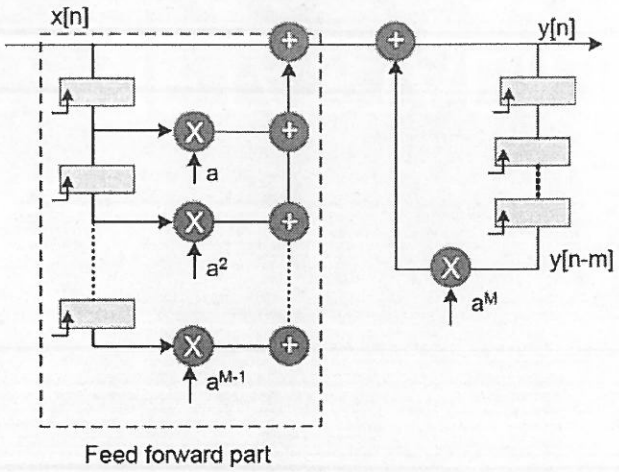


Pipelining Feedback Systems by Algorithmic Modifications (continued)

- More generally:

$$y[n] = a \cdot y[n-1] + x[n] = a^{M} \cdot y[n-M] + (x[n] + a \cdot x[n-1] + \ldots + a^{M-1} \cdot x[n-M+1])$$

- This form can be pipelined as follows:



Note*: From the signal processing viewpoint, we are using the following property of the z-transform of the system response:

$$H(z) = 1/(1-az^{-1}) = (1 + az^{-1} + \ldots + a^{M-1}z^{-M+1})/(1- a^{M}z^{-M})$$

In other words, we are adding overlapping zeros and poles to the transfer function, in favor of pipelining

- This method is known as look ahead transformation in the literature.

Architectural Improvements by Algorithmic Modifications* (optional)

- Replacing a system with it's algorithmically equivalent counterpart (in favor of architectural improvement) is very common in digital implementations.
- Example: Consider a moving average filter (used for lowpass filtering) defined by the input-output recursion: $y[n]=x[n]+x[n-1]+\ldots x[n-N+1]$

  Accordingly the impulse response and transfer functions of the system are:

$$h[n]=\delta[n]+ \delta[n-1]+\ldots \delta[n-N+1] \quad \text{or} \quad H(z)=1+z^{-1}+\ldots z^{-N+1}$$

  The FPGA implementation of this system requires N-input adders, which can cause huge combinational delays for large N.

  A method for improving this limitation is by using pipelined adder-trees. Alternatively, one may use the equivalent system: $y[n] = y[n-1] + x[n] - x[n-N]$

  We have used the fact that:

$$H(z) = (1 + z^{-1} + \ldots + z^{-N+1}) = (1- z^{-N})/(1- z^{-1})$$

- Cascaded Integrator Comb (CIC) also known as Hogenauer filters, which are very common in FPGA-based designs due to their multiplier-free property, are based on this method.

Further Reading

- Further reading on pipelining, folding and unfolding techniques for feed-forward and feedback systems:

1. Khan, Shoab Ahmed. Digital design of signal processing systems: a practical approach. John Wiley & Sons, 2011, Chapter 7.
2. Meyer-Baese, Uwe, and U. Meyer-Baese. Digital signal processing with field programmable gate arrays. Vol. 2. Berlin: Springer, 2004, Chapter 4.
3. Hauck, Scott, and Andre DeHon. Reconfigurable computing: the theory and practice of FPGA-based computation. Vol. 1. Elsevier, 2010, Chapter 18

METASTABILITY & MULTIPLE CLOCK DOMAINS

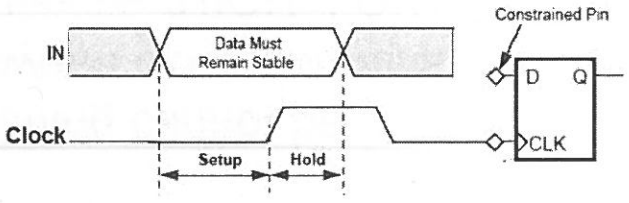
Introduction

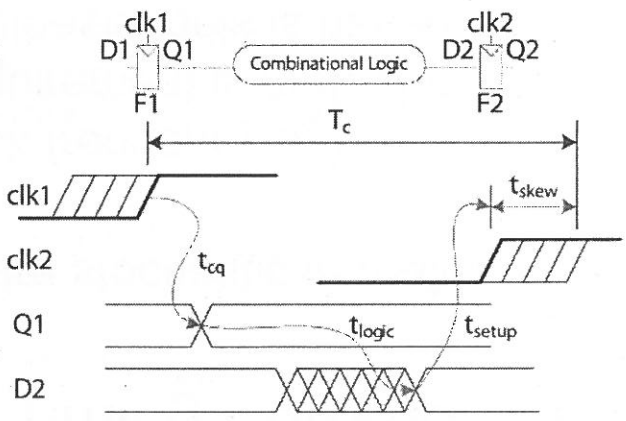
- Up to now, we have considered flip-flops and other logic devices as fully deterministic elements.
- However, in reality, no two flip-flops are "exactly" the same. The (minor) deviations between the electronic aspects and fabrication indeterminacies of these elements result in stochastic behaviors.
- Although current FPGA vendors guarantee extremely robust behaviors and extremely low probabilities of device failures, the consideration of the stochastic aspects are inevitable in certain cases, including multiple clock domain applications, which may result in metastability.
- In this section, we study some of the stochastic aspects of digital elements, such as flip-flops and robust design methods that reduce the probability of metastability and failure of digital systems.

Reference: M. Arora. The art of hardware architecture: Design methods and techniques for digital circuits. Springer Science & Business Media, 2011.

Review of Logic Circuits Timing Parameters

- Clock period (t_C): clock edge-to-edge time; inverse of clock frequency (f_C)
- Clock Skew (t_{skew}): indeterminacy of the clock edge arrival time
- Setup Time (t_{setup}): data should be stable before clock edge
- Hold Time (t_{hold}): data should be stable after clock edge
- Propagation Delay (t_{CQ}): clock edge to stable output
- Combinational delay (t_{logic}): combinational logic circuit settling time
- Setup Slack (t_{slack}): minimum data required time minus data arrival time:
  - Positive: timing met
  - Negative: timing violated
- We want: $t_C \geq t_{CQ} + t_{logic} + t_{logic} + t_{skew}$





Note: HIGH-to-LOW and LOW-to-HIGH times are not necessarily the same

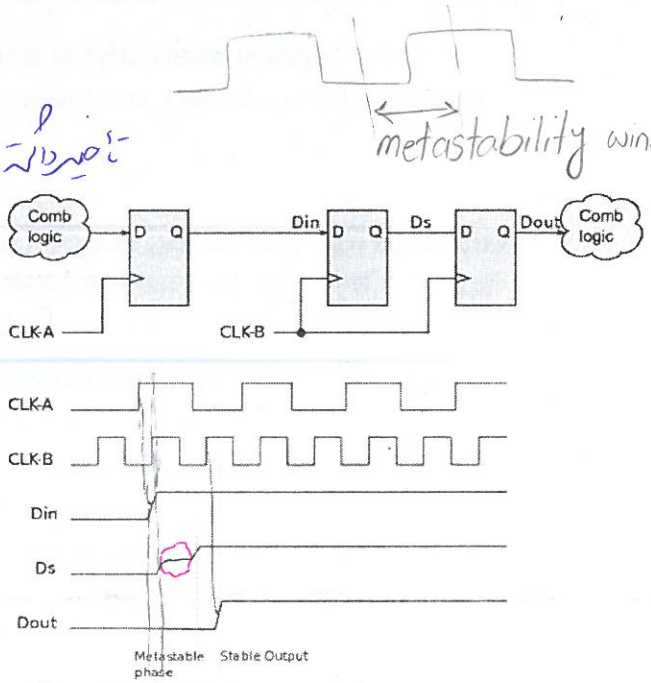
Review of Logic Circuits Timing Parameters (continued)

- Note: All the listed parameters are stochastic in reality (vary over time and space)
- In single clock designs, the clock frequency (f_C) is selected such that the slack requirement is met. The maximum clock reported by synthesis tools is based on such calculations
- In multiple clock designs, the timing cannot be guaranteed when crossing between clock domains
- Result: The output logic is not known (HIGH, LOW, or even a voltage in between). This is known as metastability

Metastability

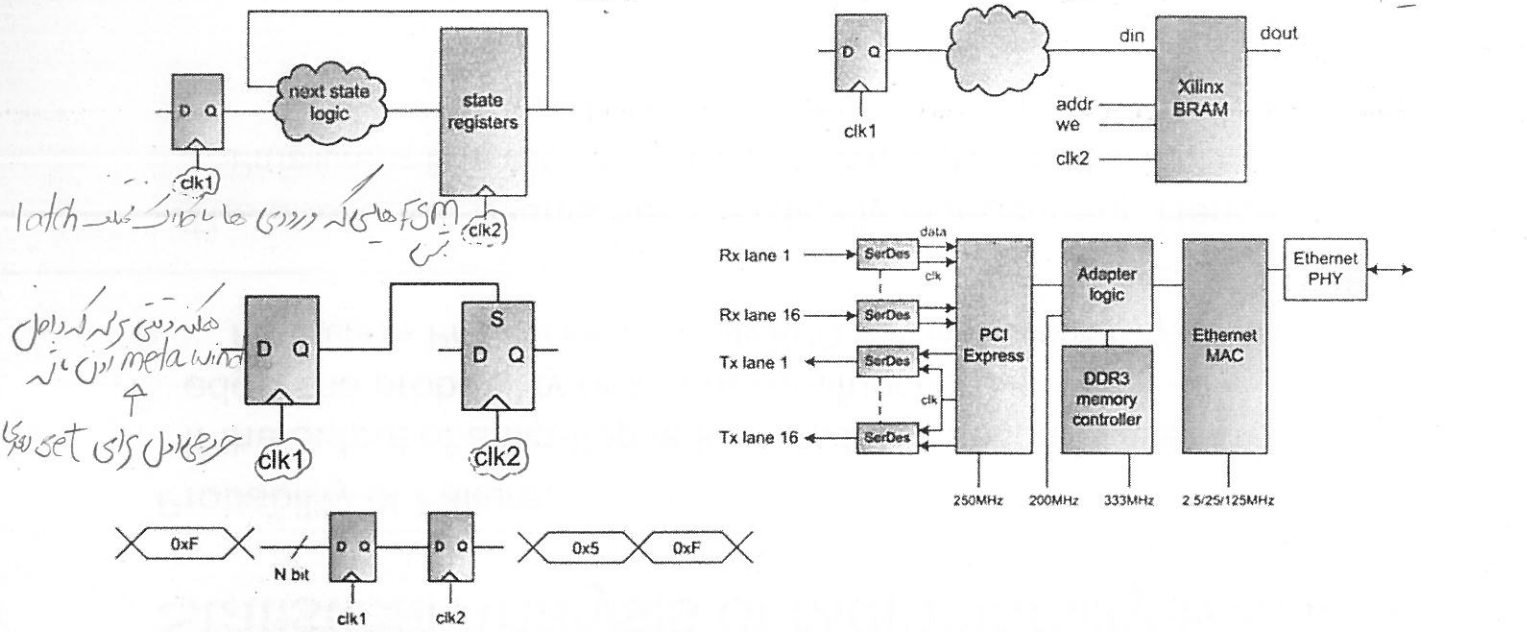
Metastability can occur when:

1. A flip-flop's slack timing is violated (high clock rate)
2. The data input to a flip-flop is asynchronous to the clock (leading to setup or hold-time violations)
3. When using multiple un-synchronized clock domains.



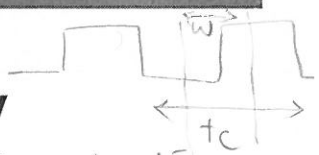
- During metastability t_{CQ} becomes longer than its nominal value.
- The additional time beyond t_{CQ}, which a metastable circuit requires to become stable is called the settling time (t_{MET})

Metastability Examples



Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

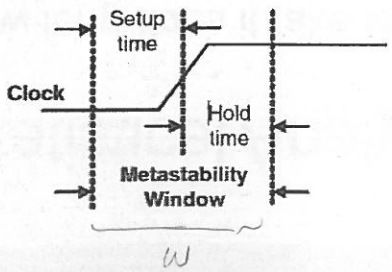
Statistical Analysis of Metastability

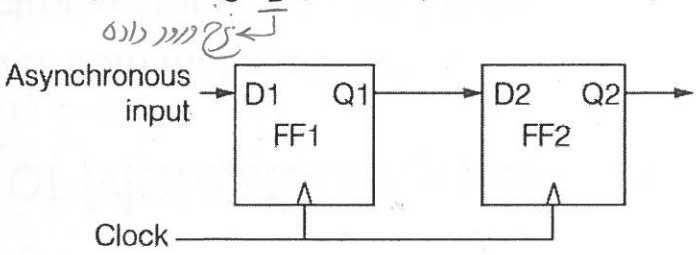


* How often does metastability occur?

  Considering t_C as the FF clock period (inverse of f_C), t_D as the asynchronous data period (inverse of f_D), and w as the metastability window length:

  - Considering the data transition probability to be uniform over the entire clock period and independent of the clock, the probability of data transition during a metastable window is $w/t_C = w \cdot f_C$
  - Therefore, the rate of metastability is $w \cdot f_C \cdot f_D$ (times per seconds)





Statistical Analysis of Metastability (continued)

How long does it take to recover from metastability?

- It can be shown that the electronic properties of flip-flops eventually take it back a stable state (0 or 1)
- Assuming that a flip-flop becomes metastable at t=0, the probability of remaining in metastability after t_{MET} seconds has been shown to be (approximately) exponentially decaying over time, i.e.:

$$\Pr(\text{staying metastable} \geq t_{MET}) = e^{\frac{-t_{MET}}{\tau}}$$

where τ is a device and technology dependent parameter.

- Reference: *Ginosar, Ran. "Metastability and synchronizers: A tutorial." IEEE Design & Test of Computers 28.5 (2011): 23-35.*

Statistical Analysis of Metastability (continued)

Probability of Failure:

- If the output of a flip-flop is sampled t_{MET} seconds after the clock edge, the probability of failure (malfunction) is

  Pr(failure) = Pr(enter metastability AND stay metastable t_{MET} or longer)

- The above two events are statistically independent. Hence:

  Pr(failure) = Pr(enter metastability)·Pr(stay metastable tMET or longer)

Mean Time Between Failures (MTBF) for Metastable Flip-Flops

The industrial standard formula for Failure Rate and Mean Time Between Failures (MTBF) of a single stage metastable flip-flop is:

$$\text{Failure Rate} = \frac{1}{\text{MTBF}} = f_D . \Pr(\text{failure}) = f_D . \underbrace{W . f_C}_{\text{Metastable window probability (how often we are in a metastable window)}} \times \underbrace{e^{\frac{-t_{MET}}{\tau}}}_{\text{The probability of remaining in metastability for } t_{MET} \text{ seconds}}$$

where:

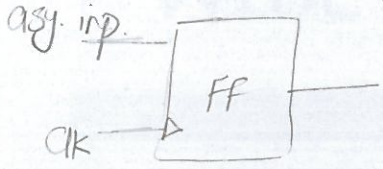
- f_C: system clock rate (Flip-Flop clock)
- f_D: (asynchronous) input data clock rate
- W: metastability window length constant
- τ: metastability time constant
- t_{MET}: time delay for the metastability to resolve itself

Note: W and τ are constants depending on the setup-time and hold-time of the device (vendor and technology dependent)

MTBF Calculation

Example 1: Consider a 28nm ASIC high-performance CMOS with W=20ps and τ=10ps (typical values for this process technology). Assuming f_C=1GHz and f_D=100MHz, we find MTBF=4x10^{29} years for a single-stage synchronizer (the universe is estimated to be 10^{10} years old).

setup + hold = 20ps
τ = 10ps
f_C = 1GHz
f_D = 100MHz
MTBF = 4×10^{29} years

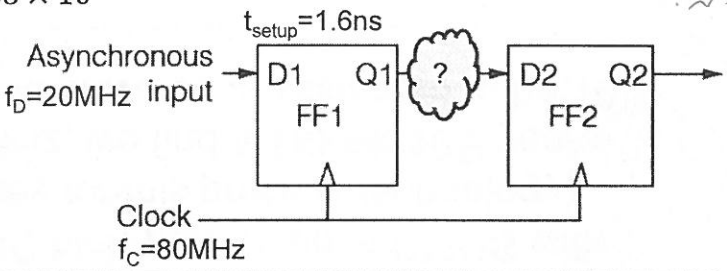


MTBF Calculation (continued)

Example 2: Suppose we want to guarantee a 1year MTBF (approximately 3×10^7s) on an Altera FLEX 10K CPLD. The MTBF constants of this family of Altera devices can be seen in the table below. In certain devices of this family t_{setup}= 1.6ns. For a data frequency f_D = 20MHz and clock frequency f_C = 80MHz we have:

$$t_{MET} = \frac{\ln(3\times10^7) + \ln[(80\times10^6)(20\times10^6)(1.01\times10^{-13})]}{1.268\times10^{10}} = 1.76\text{ns}$$

In this example the combination circuit shown in the figure can have the following maximum combinational delay to fulfil the required MTBF:
$t_{logic} \le 12.5\text{ns} - 1.76\text{ns} - 1.6\text{ns} = 9.14\text{ns}$



Note: Due to the logarithmic form of the equation, increasing t_{MET} to 2.12ns increases the MTBF to 100 years.

Device	W	1/τ
FLEX 10K	1.01×10^{-13}	1.268×10^{10}
FLEX 8000	1.01×10^{-13}	1.268×10^{10}
FLEX 6000	1.01×10^{-13}	1.268×10^{10}
MAX 9000	2.98×10^{-17}	5.023×10^{9}
MAX 7000	2.98×10^{-17}	5.023×10^{9}

Ref: Metastability in Altera Devices (May 1999, Available: ftp://ftp.altera.com/pub/lit_req/document/an/an042.pdf)

MTBF of Multistage Synchronizers

For multistage synchronizers:

$$\text{MTBF} = \frac{1}{W.f_C.f_D} \times e^{\frac{t_{MET1}}{\tau}} \times e^{\frac{t_{MET2}}{\tau}} \times \cdots$$

where t_{MET1}, t_{MET2}, etc. are the time delay for the metastability to resolve itself in each synchronizer stage.

- How many synchronizer stages are required? The parameters W and τ are commonly provided by IC manufacturers; f_C and f_D are also known by-design. The designer can define a desired MTBF, calculate t_{MET} and decide about the number of required stages to fulfil the required MTBF.

Metastability Guidelines

Avoiding metastability (by design):

1. Avoiding real-time data transfer between different clock domains
2. Using a single global clock instead of multiple clock domains
3. Avoiding gated clocks and using standard clock decreasing techniques (using clock enable)

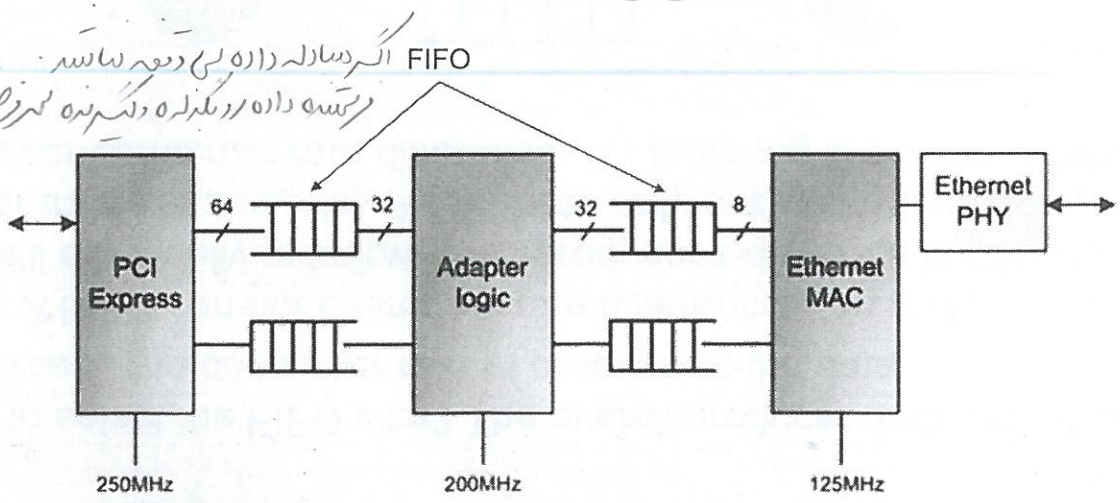
Solving metastability (by implementation):

1. Clock synchronization using DCMs
2. Using synchronizers (register chains and asynchronous FIFOs) to reduce the probability of metastability

Note: These methods only resolve metastability; but do not solve other rate mismatch issues, when transferring data between different clock domains. For example, sampling a data that changes with f_D=80MHz, at a clock rate of f_C=100MHz, results in regular repeated samples and sampling it at f_C=60MHz results in regular data loss (even without metastability).

Metastability Guidelines (continued)

Example: Using FIFOs while crossing different clock domains

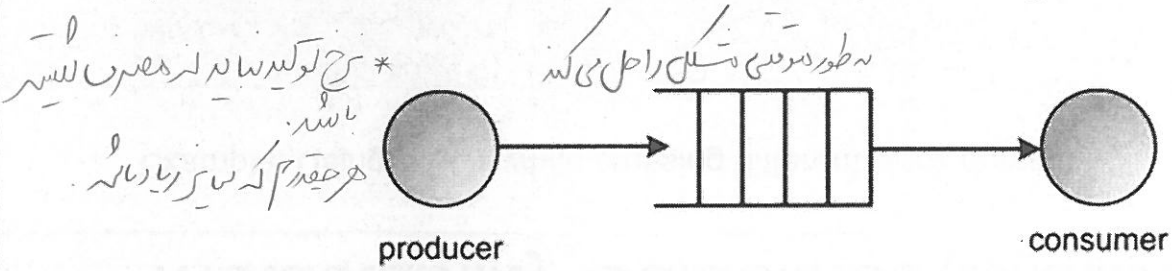


Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

FIFO Size Selection

How to select the FIFO size? The overall producer data rate should not exceed the consumer rate of processing the data.

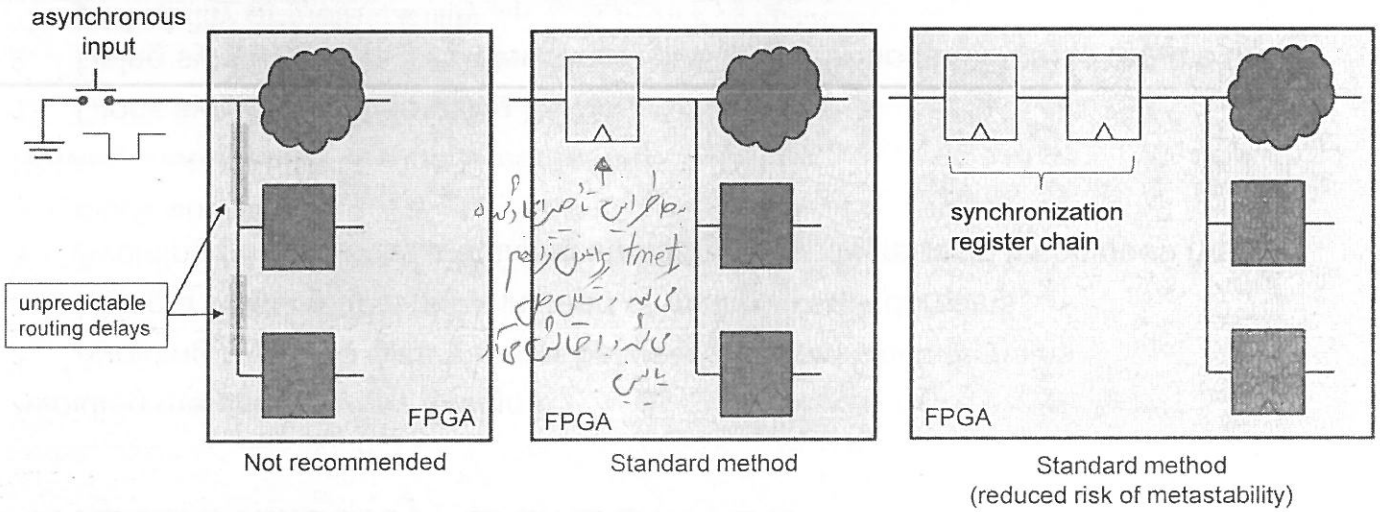
Note: A FIFO can not overcome rate differences (no matter how deep it is, it'll eventually overflow if the producer's data rate is consistently higher than consumer's). A FIFO can only overcome temporary producer-consumer rate differences by buffering the excess data.



Ref: Stavinov, Evgeni. *100 power tips for FPGA designers*. Evgeni Stavinov, 2011

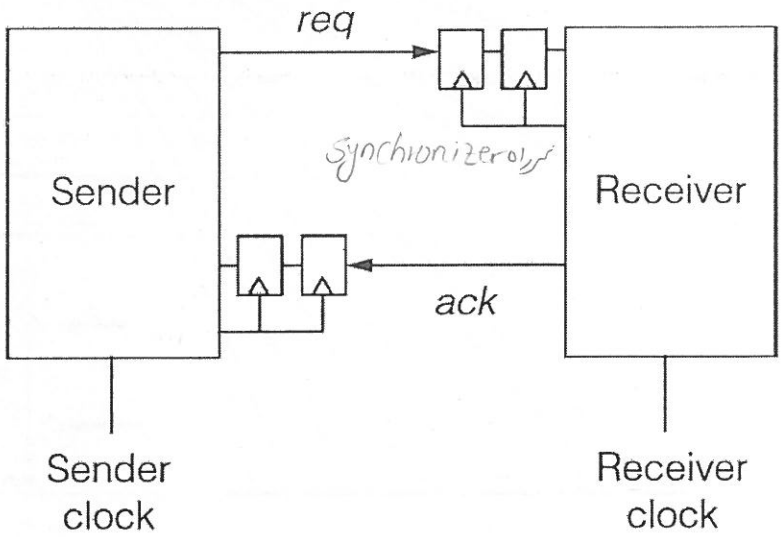
Applications: Metastability due to Top-Module Asynchronous Inputs

The standard procedure for working with top-module (asynchronous) inputs is to pass them through one or more layers of flop-flops before any internal usage.



Note: The probability of metastability decreases by increasing the number of FF layers
Question: How to handle asynchronous input buses (group of asynchronous inputs)?
Answer: By placing user defined constraints over the bus routing length.

Applications: Metastability in Two-Way Control/Acknowledge Systems



- Reference: *Ginosar, Ran. "Metastability and synchronizers: A tutorial." IEEE Design & Test of Computers 28.5 (2011): 23-35.*

Flip-Flop MTBF in Xilinx FPGA

Table legend:

$$MTBF = \frac{e^{K2 * \tau}}{F1 * F2 * K1}$$

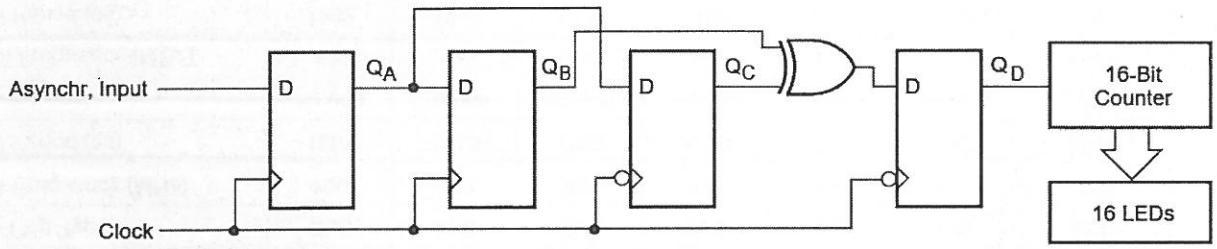
Example: Xilinx Virtex II, metastability datasheet

Device	XC2VP4			XC2VP4			XC4005E –3
V_{CC} (V)	1.5	1.35	1.65	1.5	1.35	1.65	5.0
Flip-Flop Type	CLB	CLB	CLB	IOB	IOB	IOB	CLB
Low Frequency (MHz)	300	310	390	310	300	420	109
Half Period (ps)	1667	1613	1283	1613	1667	1190	4587
MTBF1 (ms)	60,000	20,000	60,000	30,000	30,000	30,000	1,000
High Frequency (MHz)	390	420	490	420	430	500	124.4
Half Period (ps)	1282	1190	1020	1190	1163	1000	4019
MTBF2 (ms)	1.69	1.046	5.16	0.987	1.84	6.96	0.016
Half Period Difference (ps)	385	423	262	423	504	190	568
Ln (MTBF1 / MTBF2)	10.478	9.86	9.36	10.322	9.70	8.37	11.09
K2 (per ns)	27.2	23.3	35.7	24.4	19.24	44.05	19.52
1 / K2 = tau (ps)	36.8	42.9	28.0	41.0	52.0	22.7	51.2
MTBF multiply / 100 (ps)	15.2	10.3	35.6	11.5	6.85	81.8	7.04

Ref: https://china.xilinx.com/support/documentation/application_notes/xapp094.pdf

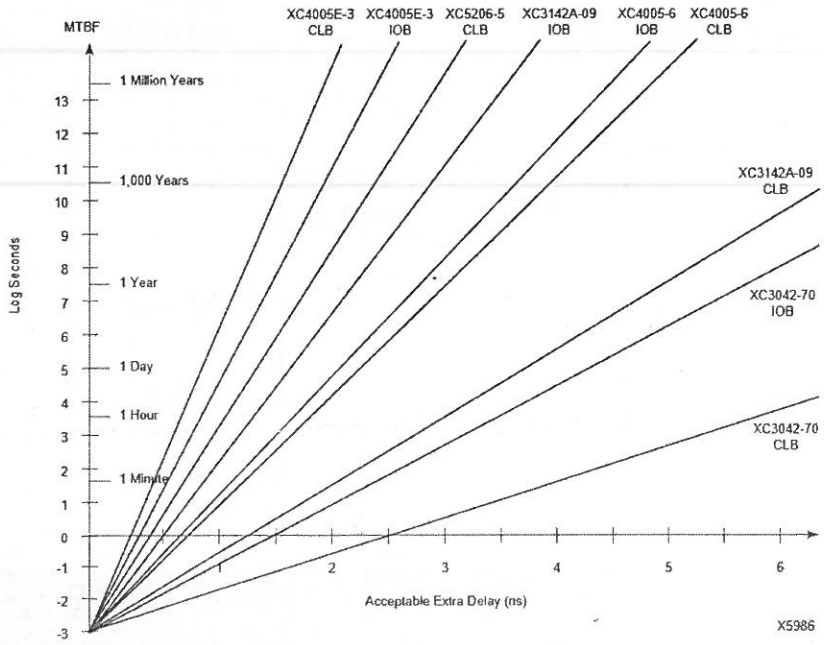
Note: Xilinx doesn't seem to list the FF MTBF of its newer devices; but it reports them in Vivado® during implementation.

Xilinx's Metastability Test Circuit



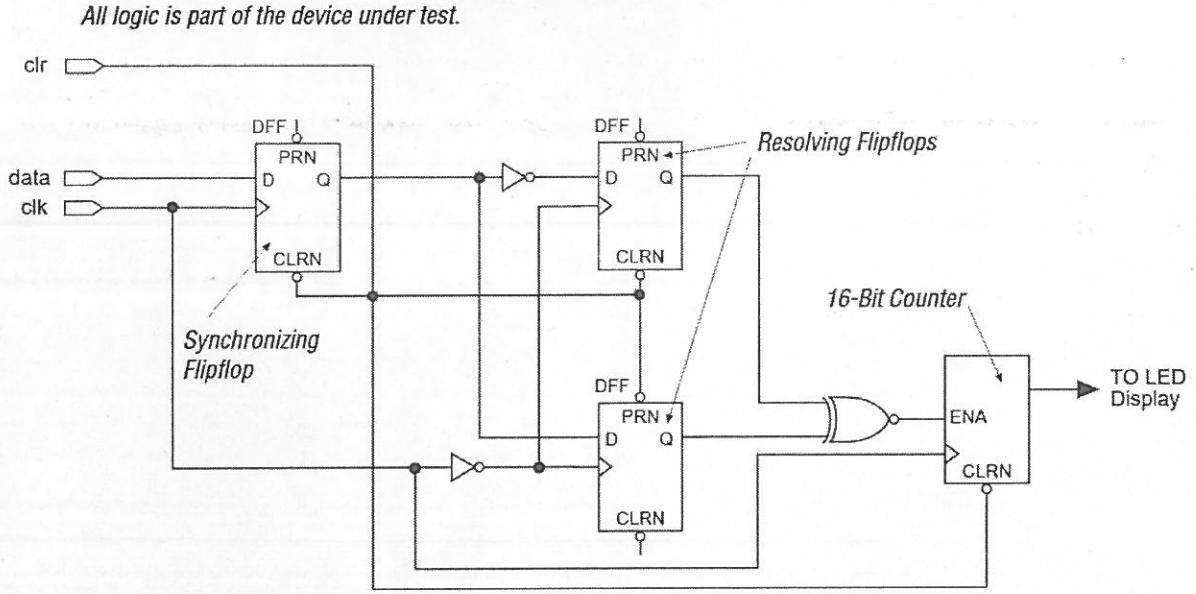
Ref: Xilinx Metastability Considerations (XAPP077.pdf January 1997, Available: http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf)

Xilinx's Metastability Test Results



Ref: Xilinx Metastability Considerations (XAPP077.pdf January 1997, Available: http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf)

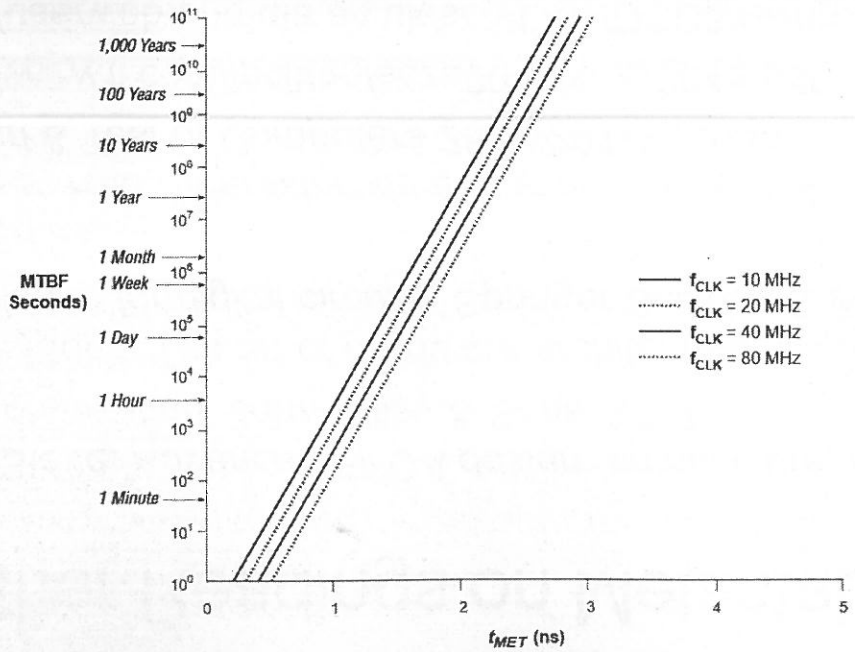
Altera's Metastability Test Circuit



Ref: ftp://ftp.altera.com/pub/lit_req/document/an/an042.pdf

Altera's Metastability Test Results

Figure 6. FLEX 10K, FLEX 8000 & FLEX 6000 MTBF Values



Ref: Metastability in Altera Devices (May 1999, Available: ftp://ftp.altera.com/pub/lit_req/document/an/an042.pdf)

Further Readings on Metastability

- Kilts, Steve. *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons, 2007.
- Arora, Mohit. *The art of hardware architecture: Design methods and techniques for digital circuits*. Springer Science & *Business Media, 2011.*
- *Ginosar, Ran. "Metastability and synchronizers: A tutorial." IEEE Design & Test of Computers 28.5 (2011): 23-35.*
- http://www.ti.com/jp/lit/an/scza004a/scza004a.pdf
- http://userweb.eng.gla.ac.uk/scott.roy/DCD3/technotes.pdf
- https://www.altera.com/en_US/pdfs/literature/wp/wp-01082-quartus-ii-metastability.pdf

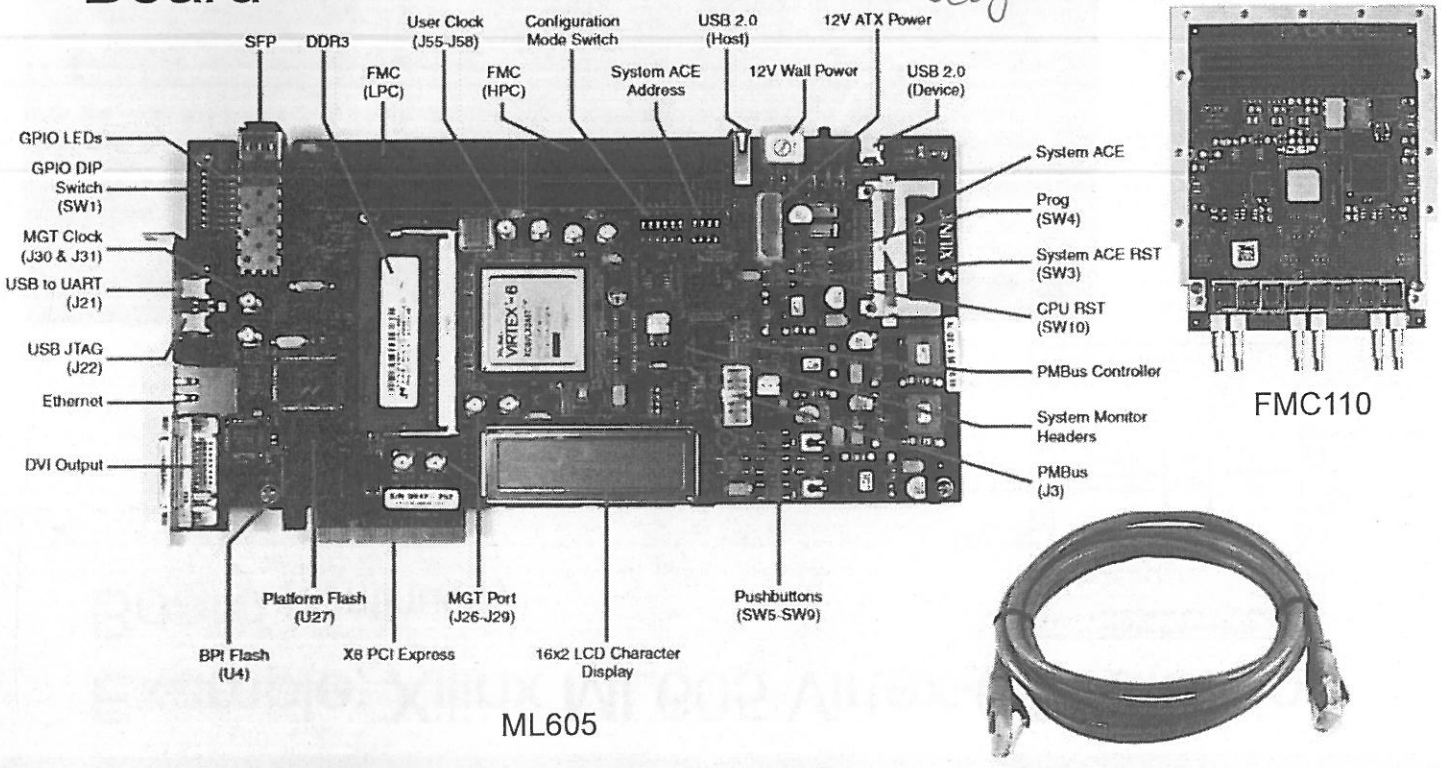
MEMORY-MAP DESIGN IN FPGA-BASED SYSTEMS

Introduction**

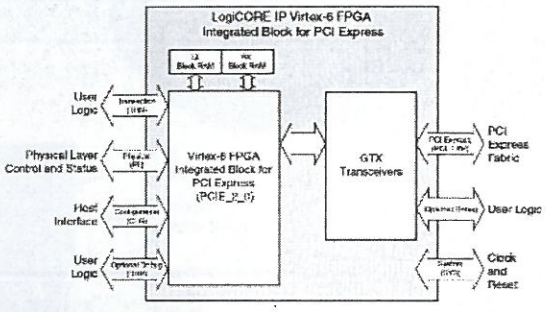
- Complex FPGA-based systems can contain multiple units (modules), each having multiple operation modes that are selected by appropriate control pins (or control bus) and give output messages in different occasions (handshakes, error codes, overflow flags, etc.)
- Each element of a design should have a unique address in the system's memory map, which can be accessed via proper commands
- In mixed CPU-FPGA systems, the internal memory map of the FPGA is commonly accessible by the software units
- The design of a memory map is discussed in this section by examples

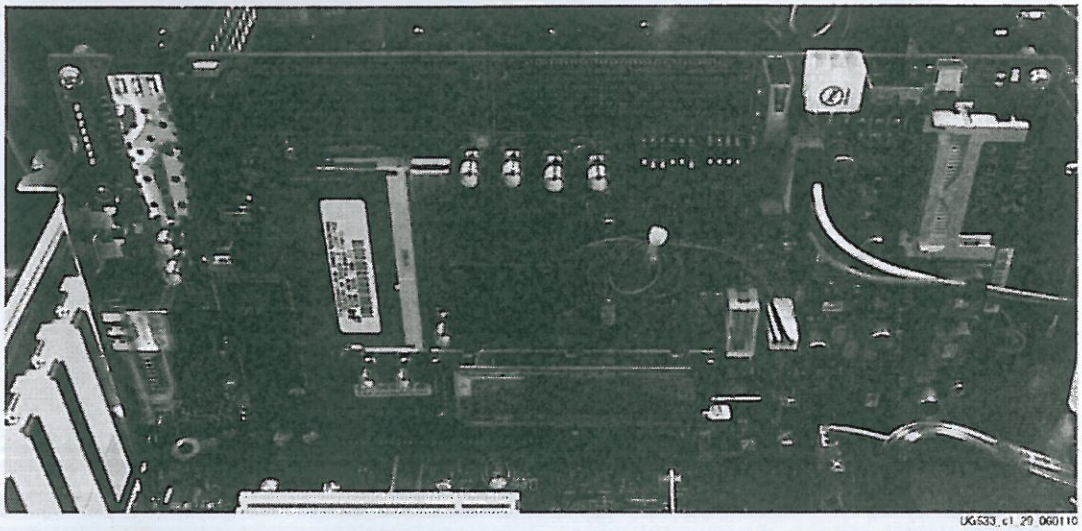
**This section is presented from industrial project source codes

Example: Xilinx ML605 Virtex-6 Evaluation Board

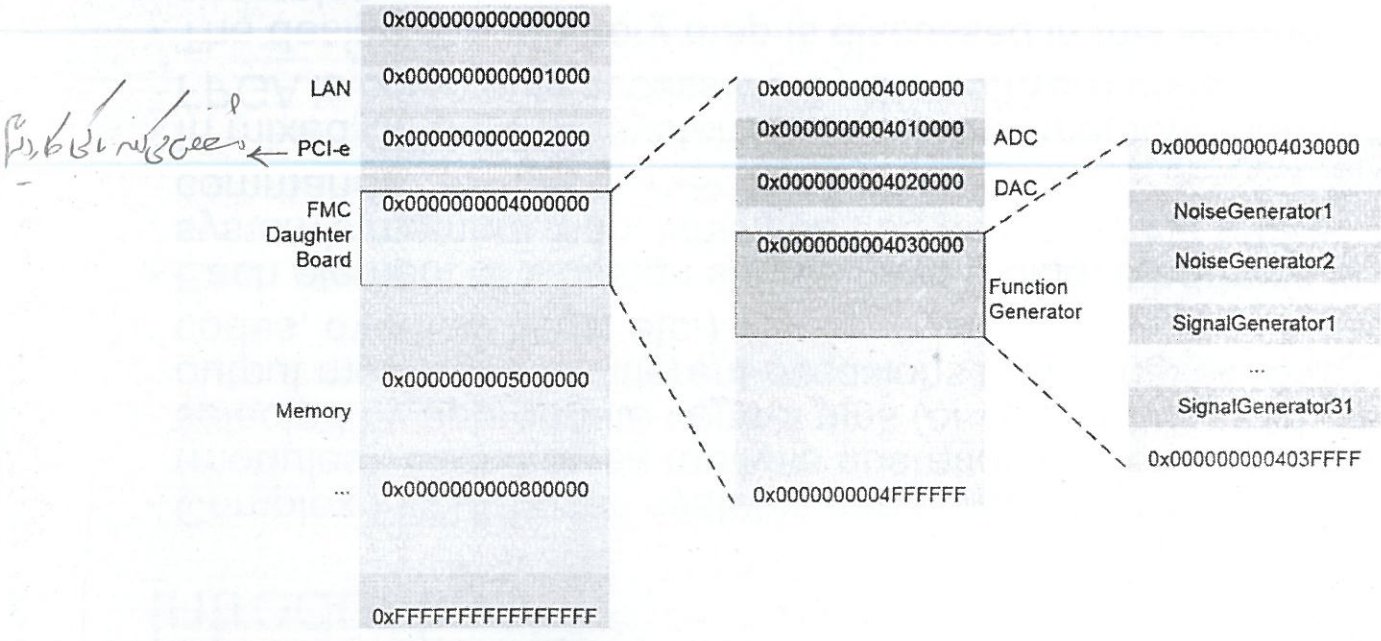


Example: Xilinx ML605 Virtex-6 Evaluation Board (continued)





Memory Map



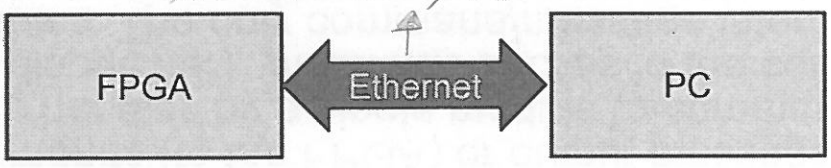
Memory map conceptual illustration

Accessing the Memory Map

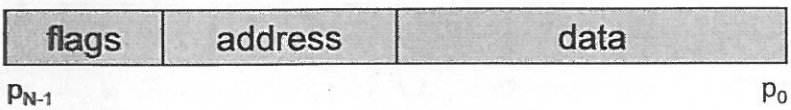
- The internal memory map (MMap) of the FPGA and the protocol for accessing the MM is designed and implemented by the FPGA designer
- The MMap can be accessed through any of the I/O ports of the FPGA board. For example:
  - Ethernet
  - PCI-e
  - JTAG
  - USB
  - Etc.

Accessing the Memory Map (continued)

- Example: Suppose that we use the Ethernet as the access port of the ML605 FPGA board. The FPGA board can send and receive Ethernet packets, which can have an arbitrary format after decoding:



Decoded Ethernet packets (arbitrary format defined by the designer):

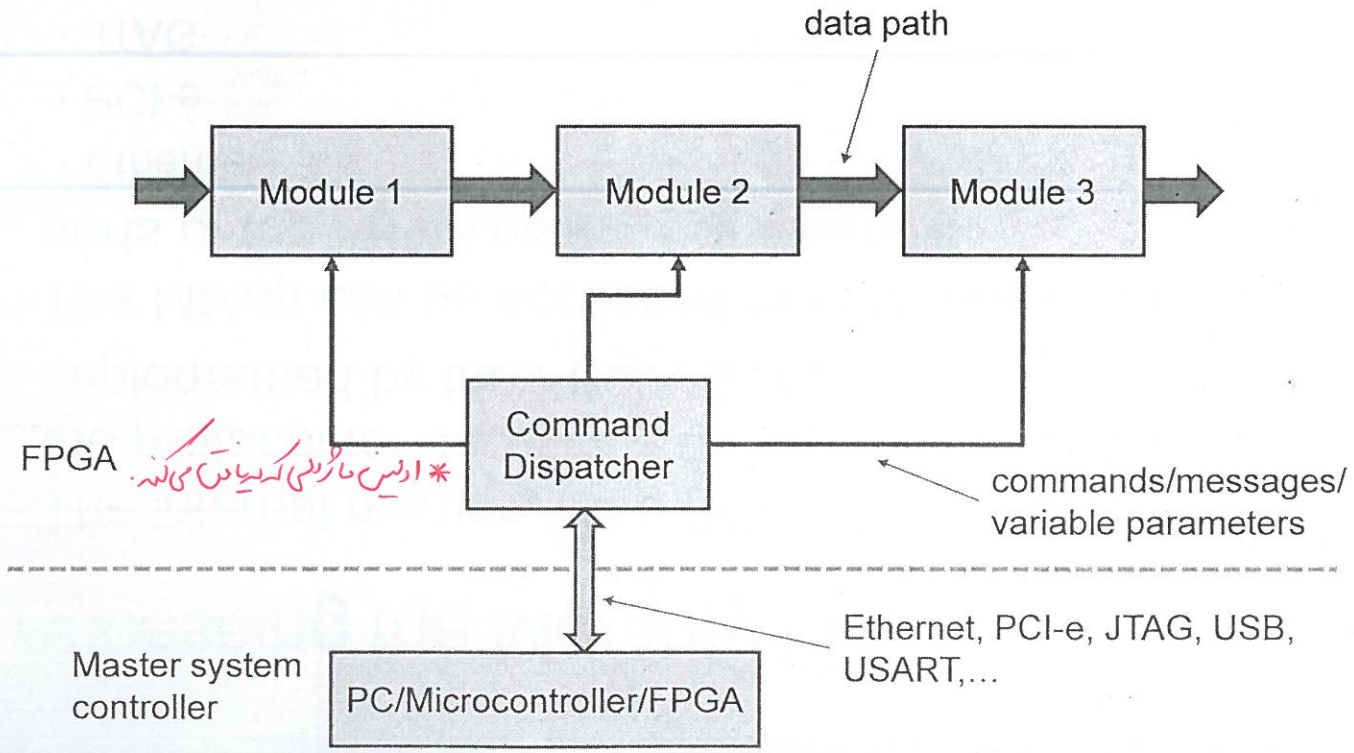


Memory Map Implementation Techniques

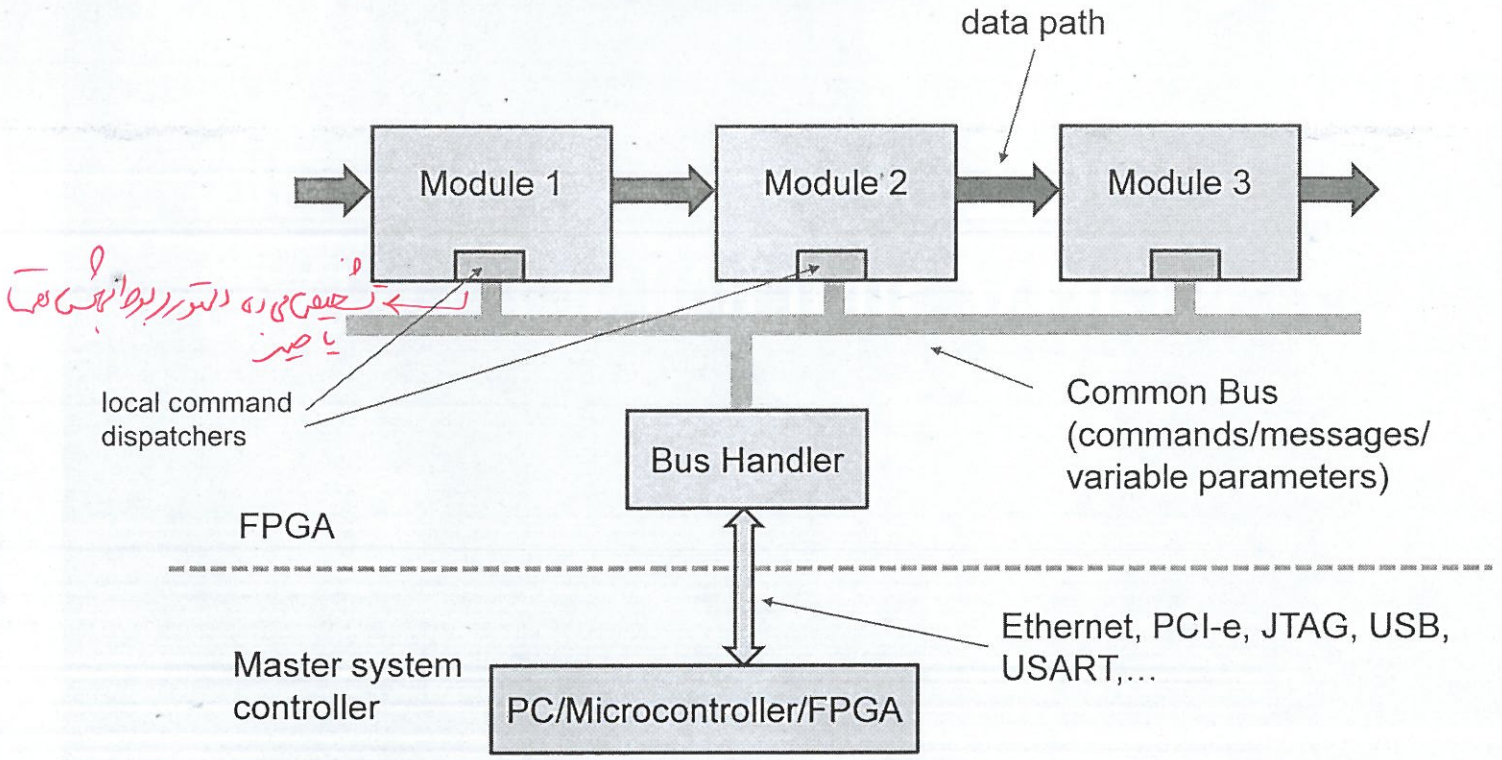
- Centralized: All modules have a set of input ports for commands and output ports for handshaking and messages. All input commands (to the FPGA) or output messages (from the FPGA) are handles by a single module (a command or message dispatcher), which has access to the command ports of all modules. The only command/message interface of the FPGA to the output world is this module.

- Distributed: There are no centralized command/message dispatchers. A common command bus is shared between all modules. Each module has a unique address (or address offset with respect to the top-module, for nested modules) in the memory map of the system. The commands/messages are handled locally by each module.

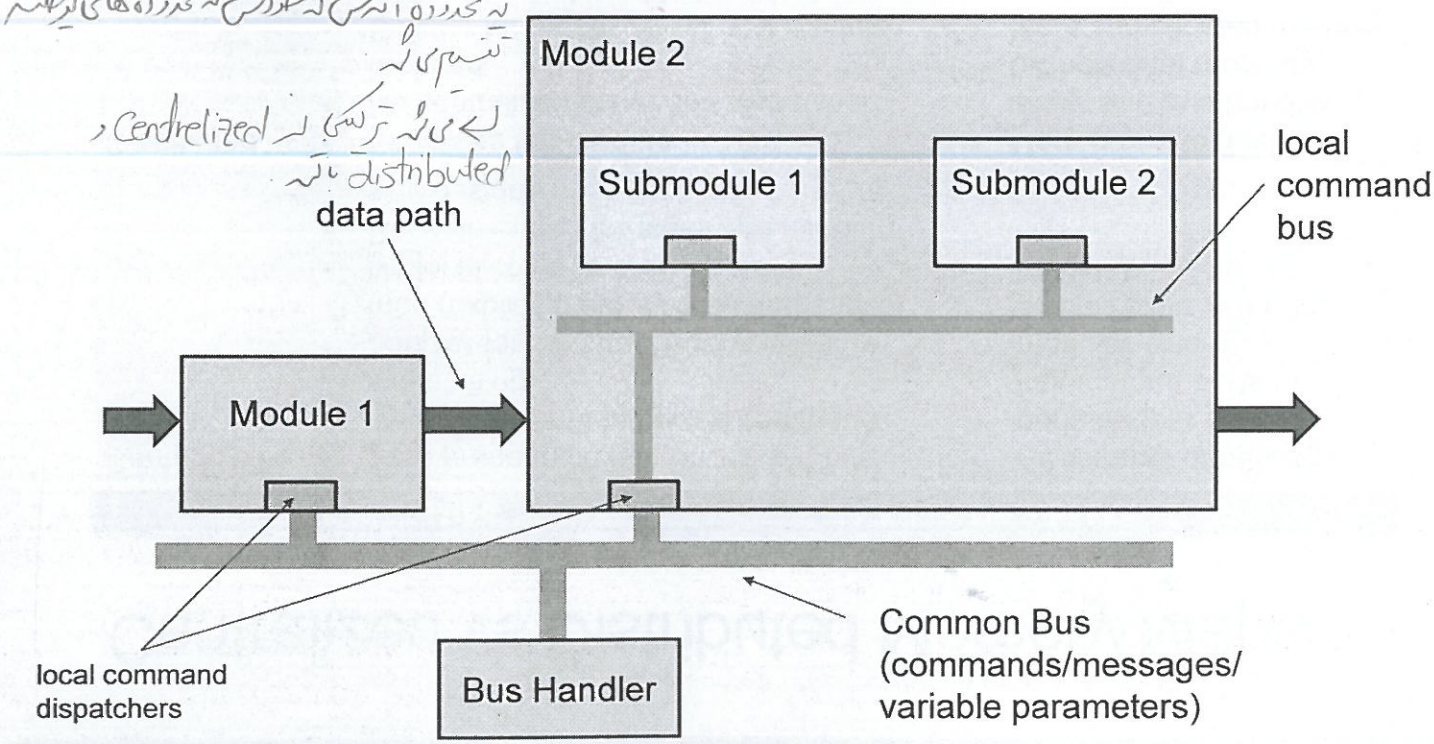
Centralized Memory Map Design



Distributed Memory Map Design



Nested Memory Maps



Centralized vs Distributed Memory Maps

	Advantages	Drawbacks
Centralized	• Less prone to design errors and bus write conflicts (centralized command dispatching) • Simpler for constructing the memory map (explicit memory map addresses) • No local command dispatchers ✓ Recommended for small and medium size designs	• All command/message ports appear as input/output ports of modules (more complication in the top-module)
Distributed	• Simplified top-module • No centralized command dispatchers required • Simpler for extension (similar module instances can be added to the design in a "plug-and-play" like manner) ✓ Recommended for complicated designs with possible future extensions	• More prone to design errors and bus handling by individual modules • More complicated memory map encoding/decoding • Each module requires a command dispatcher

DATA COMMUNICATION METHODS & PROTOCOLS

Introduction

- As with other aspects of FPGA designs, data transfer inside FPGA and between FPGA systems can be fully customized.
- In this section we review the most common techniques used for data transfer in FPGA designs
- The two classes of data transfer methods that we study are:
  - Stream Transfer
  - Packet Transfer

Continuous Stream Data Transfer

- Stream Transfer: used for continuous and synchronous data transfer between modules
- Usage: ADC, DAC, continuous data streams

- Advantage: no handshaking overheads; can use the maximum possible throughput between two endpoints
- Disadvantage: requires synchronization; even minor asynchrony between the sender and receiver clocks results in metastability, data replication or data loss
- Note: depending on the processing algorithm, continuous data streams can be up-sampled or down-sampled throughout processing

Packet (Block) Data Transfer

- Packet (Block) Transfer: used for discrete data transfer between modules
- Usage: data/message communication between asynchronous modules
- Advantage: enables data transfer between different clock domains; robust to minor sender/receiver clock frequency mismatch (depending on the block size)
- Disadvantage: requires handshaking, packing overhead (start/stop/CRC words), reduced bandwidth and packing/unpacking hardware overheads

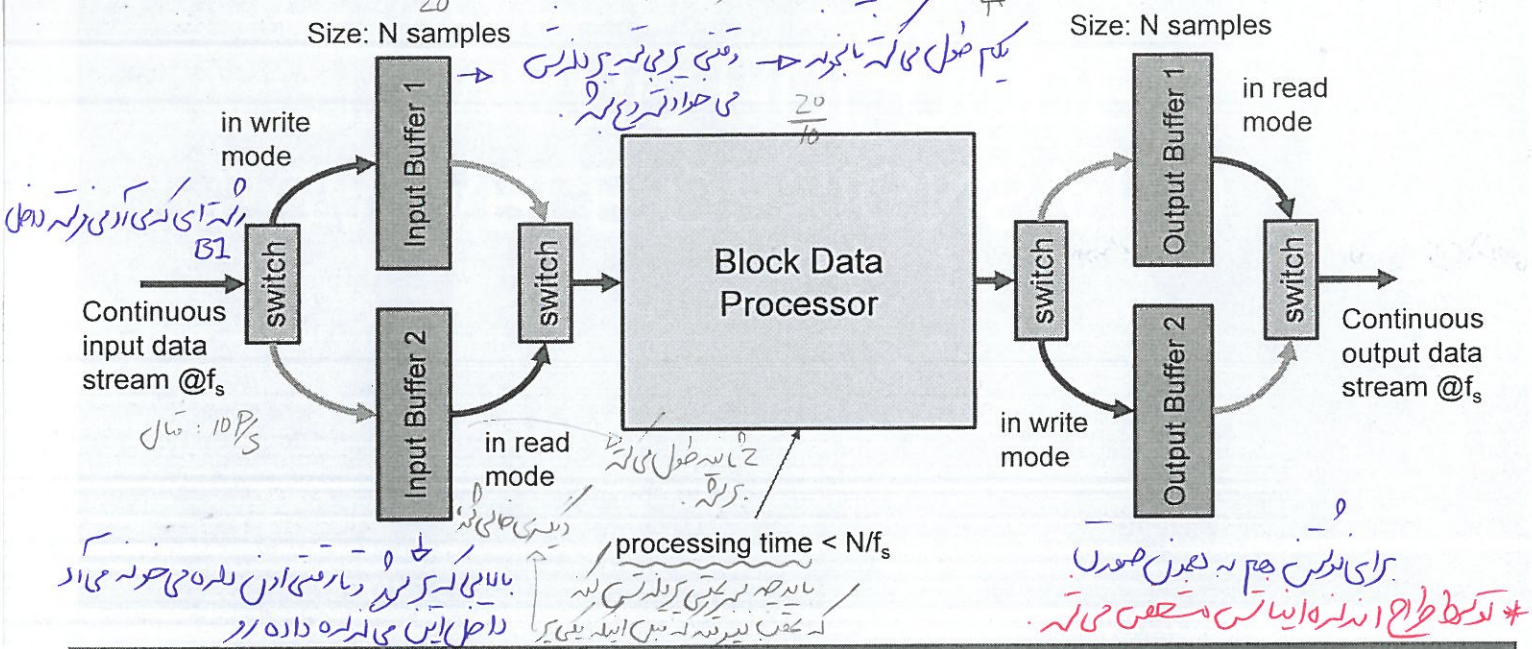
Block Processing of Streamed Data

- A common requirement in many data processing systems is the block-wise processing of continuous data streams. Examples include: DFT filtering, Reed-Solomon encoding, H.264 encoding, etc.
- The standard technique for implementing such algorithms is to use a dual-buffer at the interface between the continuous data stream and the block processor.
- As a rule if the block-wise algorithm processes a block of data faster than the data stream is accumulated in the input buffer (and read from the output buffer), no data loss occurs in the input (or output) and the block processing is masked from the outer world.

Block Processing of Streamed Data

(continued)

Dual-buffer implementation: When input is streamed in InBuff1, the block processor is working on previous data written in InBuff2. When the block processor is downloading its results in OutBuff2, the previous results are streamed from OutBuff1 to the output, etc.



The ARM Advanced Microcontroller Bus Architecture (AMBA)

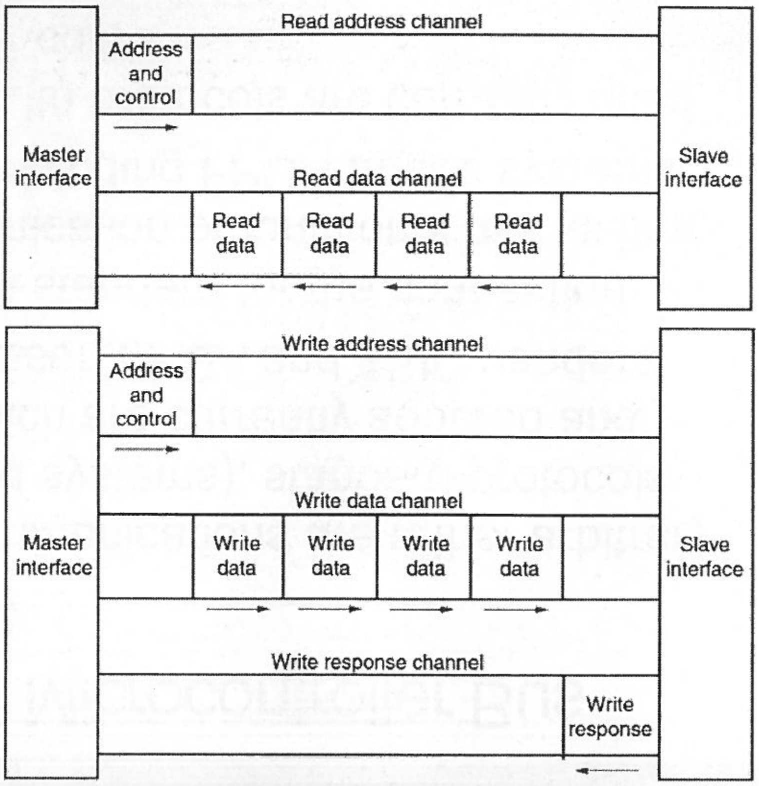
- Although on-chip data communications are rather arbitrary (especially in FPGA-based systems), standard protocols have been developed, which are currently adopted and supported by many processor, FPGA and ASIC vendors.
- The Arm AMBA is an open standard for the connection, management and communication of functional blocks in a system-on-a-chip (SoC), including FPGA-based systems.
- The AMBA AXI4 and AXI-Lite protocols are currently used in many Xilinx tools and IP cores
- AMBA AXI uses READY/VALID handshaking mechanisms

AMBA AXI4 and AXI-Lite Interfaces

- AXI4 and AXI-Lite interfaces consist of five different channels:
  - Read Address Channel
  - Write Address Channel
  - Read Data Channel
  - Write Data Channel
  - Write Response Channel

References and further reading on AXI interface protocols:

- AXI Reference Guide, https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf
- AMBA® AXI™ and ACE™ Protocol Specification, https://www.arm.com/products/system-ip/amba-specifications
- AXI4-StreamingtoStellarIP Interface, http://www.4dsp.com/pdf/AN001_KC705_FMC104_AXI_FFTcore_tutorial.pdf



SCALABLE DESIGNS AND AUTOMATIC HDL CODE GENERATION

ابزار بسازیم که خودش کد تولید کنی.

Scalable Design and Automatic HDL Code Generation

- Verilog and VHDL have limited features for scalable and parametric designs (such as genvar, generate, etc.)
- In this section, we will learn how to write scripts in other languages (C, Java, Python, Matlab, etc.) to generate synthesizable HDL codes
- These methods can be used to generate user defined HDL libraries, Netlists and EDIF files.
- The basic idea is to open a .v or .vhd file in another language and start writing in it with Verilog or VHDL supported syntax, while using the flexibilities and features of the higher level language.

Scalable Design and Automatic HDL Code Generation (continued)

Example 1: Matlab script for generating Running DFT Verilog code

```
clear;
close all;

fname = 'RunningDFT8';

INLEN = 16; % input data width
NDFTIN = 8; % Number of temporal DFT samples
NDFTINBITS = ceil(log2(NDFTIN)); % input data number of bits
OUTLEN = INLEN + NDFTINBITS; % output data width
DFTSamples = 0 : NDFTIN-1; % The frequency indexes
NDFTOUT = length(DFTSamples); % Number of temporal DFT samples
NDFTOUTBITS = ceil(log2(NDFTOUT)); % output data number of bits

theta = 2*pi*DFTSamples/NDFTIN;
phase = round(exp(1j*theta)*(2^(INLEN-1)-1));
tm = clock; tm = floor(tm(4:6));
fid = fopen([fname '.v'],'w');
fprintf(fid,['`timescale 1ns / 1ps\n',...
    '////////////////////////////////////////////////////////////////////////////////\n',...
    '// Company:    Signal Processing Center (SPC), Shiraz University\n',...
    '// Engineer:    Reza Sameni\n',...
    '// Web:    www.sameni.info\n',...
    '// \n',...
    '// Create Date:    %d:%d:%d %s \n',...
    '// Design Name:    Running Discrete Fourier Transform (DFT)\n',...
    '// Module Name:    %s \n',...
    '// Project Name:    Real Time Spectrum Analysis\n',...
    '// Target Devices:    Virtex-5\n',...
    '// Tool versions: \n',...
    '// Description: This module calculates the %d-point running Discrete Fourier \n//\t\t\t\t\tTransform (DFT)
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 1 (continued): Output Verilog file

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:    Signal Processing Center (SPC), Shiraz University
4  // Engineer:    Reza Sameni
5  // Web:    www.sameni.info
6  //
7  // Create Date:    13:38:50 01-Jun-2018
8  // Design Name:    Running Discrete Fourier Transform (DFT)
9  // Module Name:    RunningDFT8
10 // Project Name:    Real Time Spectrum Analysis
11 // Target Devices:    Virtex-5
12 // Tool versions:
13 // Description: This module calculates the 8-point running Discrete Fourier
14 //              Transform (DFT) of a stream of complex valued signal over 8
15 //              frequency samples.
16 //              The module has been automatically generated by RunningDFTGenerator.m.
17 //
18 // Dependencies:    Xilinx Core Generator Complex Multiplier
19 //
20 // Revision:
21 // Revision 0.01 - File Created
22 // Additional Comments:   www.spc.shirazu.ac.ir
23 //
24 //////////////////////////////////////////////////////////////////////////////////
25 module RunningDFT8(clock, ce, reset, logresult, xr, xi, loggedaddress, xxrlogged, xxilogged);
26
27 input clock;
28 input ce;
29 input reset;
30 input logresult;
31 input [15:0] xr;
32 input [15:0] xi;
33 input [2:0]loggedaddress;
```

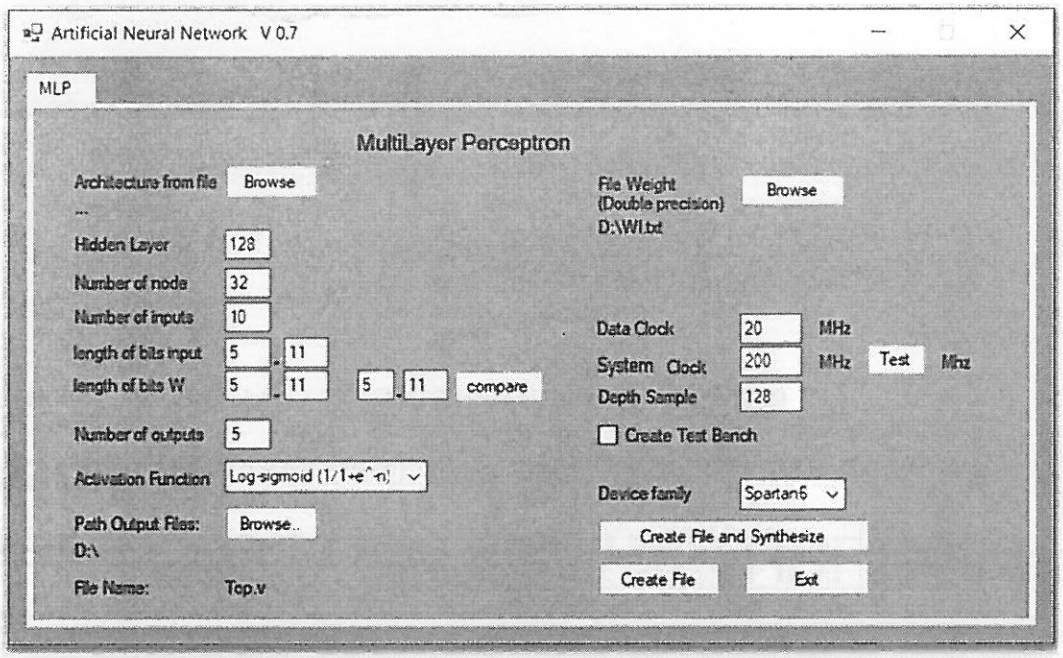

Scalable Design and Automatic HDL Code Generation (continued)

Example 1 (continued): Output Verilog file continued

```
34 output reg [15:0]xxrlogged;
35 output reg [15:0]xxilogged;
36
37 reg [2:0]oldestpointer;
38 reg [15:0]fifor[7:0];
39 reg [15:0]fifoi[7:0];
40 reg [15:0]deltar;
41 reg [15:0]deltai;
42 reg [15:0]xxrloggedarray[7:0];
43 reg [15:0]xxiloggedarray[7:0];
44
45 // for simulation only
46 integer i;
47 initial begin
48     for(i = 0 ; i < 8 ; i=i+1)begin
49         fifor[i] = 0;
50         fifoi[i] = 0;
51     end
52     for(i = 0 ; i < 8 ; i=i+1)begin
53         xxrloggedarray[i] = 0;
54         xxiloggedarray[i] = 0;
55     end
56     oldestpointer = 0;
57     deltar = 0;
58     deltai = 0;
59 end
60 // end simulation initialization
61
62 always @(posedge clock)if(ce)begin
63      if(reset)begin
64          oldestpointer <= 0;
65          deltar <= 0;
66          deltai <= 0;
67      end
68      else begin
69          deltar <= xr - fifor[oldestpointer];
70          deltai <= xi - fifoi[oldestpointer];
71          fifor[oldestpointer] <= xr;
72          fifoi[oldestpointer] <= xi;
73          oldestpointer <= oldestpointer + 1;
74      end
75 end
76
77 wire [18:0]xxr0;
78 wire [18:0]xxi0;
79 MultiplyAxB multiplier0 (
80     .ar(xxr0 + deltar),
81     .ai(xxi0 + deltai),
82     .br(16'd32767),
83     .bi(16'd0),
84     .clk(clock),
85     .ce(ce),
86     .sclr(reset),
87     .pr(xxr0),
88     .pi(xxi0)) /* synthesis syn_noprune =1 syn_preserve = 1 */;
89
90 wire [18:0]xxr1;
91 wire [18:0]xxi1;
92 MultiplyAxB multiplier1 (
93     .ar(xxr1 + deltar),
94     .ai(xxi1 + deltai),
95     .br(16'd23170),
96     .bi(16'd23170),
97     .clk(clock),
98     .ce(ce),
99     .sclr(reset),
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 2: Generating Multilayer Perceptron Artificial Neural Networks RTL codes in C# (By Pejman Torabi, Shiraz University)



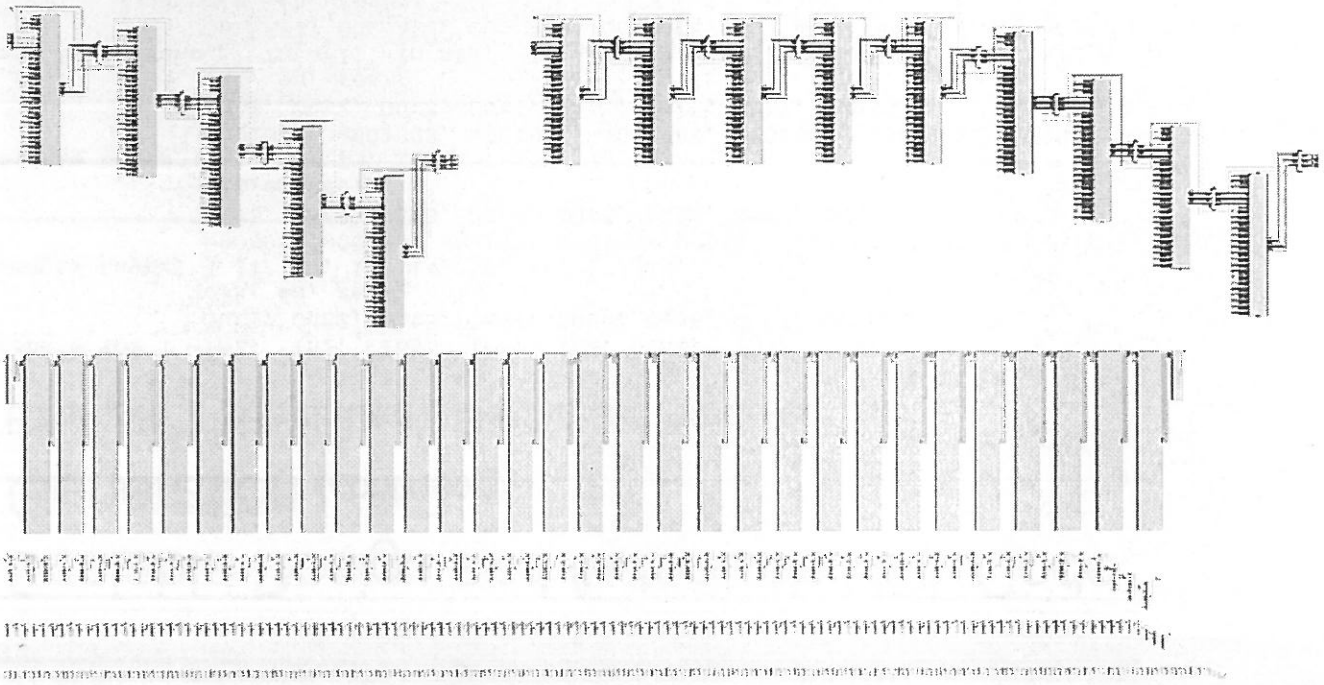
Scalable Design and Automatic HDL Code Generation (continued)

Example 2 (continued): Generated modules

```
module TOP (  In1,  In2,  In3,  In4,  In5,  In6,  In7,  In8,
              Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
              clk, en, res);
module Layer1 ( i1 ,i2, i3, i4, i5, i6, i7, i8,
              w001001,w001002,w001003,w001004,w001005,w001006,w001007,w001008, B001, …
              Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
              clk,en,res);
module Layer ( i1, i2, i3, i4, i5, i6, i7, i8,
              w001001, w001002, w001003, w001004, w001005, w001006, w001007,w001008, B001,…
              Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
              clk, en, res);
module ActFunc (  In_AF1,  In_AF2,  In_AF3,  In_AF4,  In_AF5,  In_AF6,  In_AF7,  In_AF8,
              Out_AF1, Out_AF2, Out_AF3, Out_AF4, Out_AF5, Out_AF6, Out_AF7, Out_AF8,
              clk, en, res);
module Function_Interpolation (inputVal, outputVal, clk);
module mult (a, b, z, clk);
module Layer_End ( i1,i2,i3,i4,i5,i6,i7,i8,
              w001001, w001002, w001003, w001004, w001005, w001006, w001007, w001008, B001,
…
              Out1, Out2, Out3, Out4, Out5, Out6, Out7, Out8,
              clk, en, res);
module ActFunc_End (In_AF1, In_AF2, In_AF3, In_AF4, In_AF5,  In_AF6,  In_AF7, In_AF8,
              Out_AF1, Out_AF2, Out_AF3, Out_AF4, Out_AF5, Out_AF6, Out_AF7, Out_AF8,
              clk,en,res);
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 2 (continued): RTL schematic of the generated codes



Scalable Design and Automatic HDL Code Generation (continued)

Example 3: Xilinx HEX file generation in Matlab

```
clear;
close all;

N = 340;
n = (0:N-1);
nbits = 16;

sine = sin(2*pi*n/N);
cosine = cos(2*pi*n/N);

figure
hold on
stem(cosine);
stem(sine,'r');
grid

fid1 = fopen('sine_init_vals.hex','w');
fid2 = fopen('cosine_init_vals.hex','w');
for i = 1:N
    fprintf(fid1,'%s\n',hex(fi(sine(i),1,nbits,nbits-1)));
    fprintf(fid2,'%s\n',hex(fi(cosine(i),1,nbits,nbits-1)));
end
fclose(fid1);
fclose(fid2);
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 3 (continued): Output HEX file

1	0000	21	2e3d	41	563c	61	7295	81	7f74						
2	025e	22	3070	42	57f7	62	739e	82	7fa7						
3	04bb	23	329e	43	59ab	63	749c	83	7fce						
4	0718	24	34c8	44	5b58	64	7591	84	7fea						
5	0974	25	36ed	45	5cfc	65	767b	85	7ffa						
6	0bcf	26	390e	46	5e98	66	775b	86	7fff						
7	0e2a	27	3b29	47	602c	67	7831	87	7ffa						
8	1083	28	3d40	48	61b7	68	78fc	88	7fea						
9	12db	29	3f51	49	633a	69	79bc	89	7fce						
10	1531	30	415c	50	64b4	70	7a72	90	7fa7						
11	1785	31	4362	51	6625	71	7b1d	91	7f74						
12	19d7	32	4562	52	678e	72	7bbd	92	7f37						
13	1c27	33	475c	53	68ed	73	7c53	93	7eee						
14	1e75	34	494f	54	6a43	74	7cde	94	7e9b						
15	20bf	35	4b3d	55	6b90	75	7d5d	95	7e3c						
16	2307	36	4d23	56	6cd4	76	7dd2	96	7dd2						
17	254c	37	4f03	57	6e0e	77	7e3c	97	7d5d						
18	278e	38	50dc	58	6f3e	78	7e9b	98	7cde						
19	29cc	39	52ae	59	7065	79	7eee	99	7c53						
20	2c07	40	5478	60	7182	80	7f37	100	7bbd						

...

Scalable Design and Automatic HDL Code Generation (continued)

Example 4: Xilinx coefficient file generation in C

```
#include<stdio.h>
#include<math.h>
#define PI (4*atan(1))
void main()
{
    int N = 400;
    int Radix = 16;
    int Coefficient_Width = 16;
    int MaxNum = (1<<(Coefficient_Width-1)) - 1;
    int AllOnes = (1<<Coefficient_Width) - 1;
    unsigned *CoefData;
    CoefData = new unsigned[N];
    FILE* fil;
    fil = fopen("Sine400.coe", "w");
    fprintf(fil,";\n; XILINX CORE Generator(tm) Look-Up-Table (.COE) File\n;
    Generated by Reza Sameni\n;\n; Generated on: 7-May-2012 12:52:00\n;\n");
    fprintf(fil,"memory_initialization_radix = %d;\n", Radix);
    ///fprintf(fil,"Coefficient_Width = %d;\n", Coefficient_Width);
    fprintf(fil,"memory_initialization_vector = ");
    for(int i = 0 ; i < N ; i++)
    {
        CoefData[i] = int(MaxNum*sin(i*2.*PI/N)) & AllOnes;
        if(i < N-1)
            fprintf(fil,"%x,", CoefData[i]);
        else
            fprintf(fil,"%x;\n", CoefData[i]);

    }
    fclose(fil);
}
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 4 (continued): Output COE file

```
;
; XILINX CORE Generator(tm) Look-Up-Table (.COE) File
; Generated by Reza Sameni
;
; Generated on: 7-May-2012 12:52:00
;
memory_initialization_radix = 16;
memory_initialization_vector =
0,202,405,607,809,a0a,c0b,e0b,100a,1208,1405,1601,17fb,19f4,1beb,1de1,1fd4,21c6,23b5,25a2,278d,2975
,2b5b,2d3e,2f1e,30fb,32d5,34ac,367f,384f,3a1b,3be4,3da9,3f6a,4127,42e0,4495,4645,47f1,4999,4b3b,4cd
9,4e73,5007,5196,5320,54a5,5624,579e,5912,5a81,5bea,5d4e,5eab,6002,6154,629f,63e4,6523,665b,678d,68
b8,69dc,6afa,6c12,6d22,6e2b,6f2e,7029,711e,720b,72f1,73d0,74a7,7578,7640,7701,77bb,786d,7918,79bb,7
a56,7ae9,7b75,7bf9,7c75,7ce9,7d56,7dba,7e17,7e6b,7eb8,7efc,7f39,7f6d,7f99,7fbe,7fda,7fee,7ffa,7fff,
7ffa,7fee,7fda,7fbe,7f99,7f6d,7f39,7efc,7eb8,7e6b,7e17,7dba,7d56,7ce9,7c75,7bf9,7b75,7ae9,7a56,79bb
,7918,786d,77bb,7701,7640,7578,74a7,73d0,72f1,720b,711e,7029,6f2e,6e2b,6d22,6c12,6afa,69dc,68b8,678
d,665b,6523,63e4,629f,6154,6002,5eab,5d4e,5bea,5a81,5912,579e,5624,54a5,5320,5196,5007,4e73,4cd9,4b
3b,4999,47f1,4645,4495,42e0,4127,3f6a,3da9,3be4,3a1b,384f,367f,34ac,32d5,30fb,2f1e,2d3e,2b5b,2975,2
78d,25a2,23b5,21c6,1fd4,1de1,1beb,19f4,17fb,1601,1405,1208,100a,e0b,c0b,a0a,809,607,405,202,0,fdfe,
fbfb,f9f9,f7f7,f5f6,f3f5,f1f5,eff6,edf8,ebfb,e9ff,e805,e60c,e415,e21f,e02c,de3a,dc4b,da5e,d873,d68b
,d4a5,d2c2,d0e2,cf05,cd2b,cb54,c981,c7b1,c5e5,c41c,c257,c096,bed9,bd20,bb6b,b9bb,b80f,b667,b4c5,b32
7,b18d,aff9,ae6a,ace0,ab5b,a9dc,a862,a6ee,a57f,a416,a2b2,a155,9ffe,9eac,9d61,9c1c,9add,99a5,9873,97
48,9624,9506,93ee,92de,91d5,90d2,8fd7,8ee2,8df5,8d0f,8c30,8b59,8a88,89c0,88ff,8845,8793,86e8,8645,8
5aa,8517,848b,8407,838b,8317,82aa,8246,81e9,8195,8148,8104,80c7,8093,8067,8042,8026,8012,8006,8001,
8006,8012,8026,8042,8067,8093,80c7,8104,8148,8195,81e9,8246,82aa,8317,838b,8407,848b,8517,85aa,8645
,86e8,8793,8845,88ff,89c0,8a88,8b59,8c30,8d0f,8df5,8ee2,8fd7,90d2,91d5,92de,93ee,9506,9624,9748,987
3,99a5,9add,9c1c,9d61,9eac,9ffe,a155,a2b2,a416,a57f,a6ee,a862,a9dc,ab5b,ace0,ae6a,aff9,b18d,b327,b4
c5,b667,b80f,b9bb,bb6b,bd20,bed9,c096,c257,c41c,c5e5,c7b1,c981,cb54,cd2b,cf05,d0e2,d2c2,d4a5,d68b,d
873,da5e,dc4b,de3a,e02c,e21f,e415,e60c,e805,e9ff,ebfb,edf8,eff6,f1f5,f3f5,f5f6,f7f7,f9f9,fbfb,fdfe;
```


Scalable Design and Automatic HDL Code Generation (continued)

Example 5: Automatic listing generation for LaTeX reports. Project reports (specifically in LaTeX) can be automatically updated with the latest version of the source codes

```
clear;
close all;
clc;
d = dir('*.v');
fid = fopen('VerilogSourceCodeListing.tex','w');
for i = 1 : length(d)
    I = strfind(d(i).name,'_');
    lbl = d(i).name;
    if(~isempty(I))
        dd = d(i).name;
        for j = 1: length(I)
            %dd = [dd(1:(I(j)-1 + (j-1))) '\' dd( (I(j) + (j-1)):end)];
            dd = [dd(1:(I(j)-1 - (j-1))) '\' dd( (I(j+1) - (j-1)):end)];
        end
        lbl = dd;
    end
    fprintf(fid, '\\lstinputlisting[basicstyle = \\footnotesize, language = Verilog, label = %s, caption
    = %s]{SourceCodes/Verilog/%s}\n',d(i).name, d(i).name, d(i).name);
end
fclose(fid);

d = dir('ipcore_dir\\*.xco');
fid = fopen('XilinxCoreGeneratorCodeListing.tex','w');
for i = 1 : length(d)
    I = strfind(d(i).name,'_');
    lbl = d(i).name;
    if(~isempty(I))
        dd = d(i).name;
        for j = 1: length(I)
            %dd = [dd(1:(I(j)-1 + (j-1))) '\' dd( (I(j) + (j-1)):end)];
            dd = [dd(1:(I(j)-1 - (j-1))) dd( (I(j) + 1 - (j-1)):end)];
        end
        lbl = dd;
    end
    fprintf(fid, '\\lstinputlisting[basicstyle = \\footnotesize, language = Verilog, label = %s, caption
    = %s]{SourceCodes/Verilog/ipcore_dir/%s}\n',lbl, lbl, d(i).name);
end
fclose(fid);
```

Scalable Design and Automatic HDL Code Generation (continued)

Example 5: Output LaTeX listing

```
1  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   AnalyticSignal.v, caption = AnalyticSignal.v]{SourceCodes/Verilog/
   AnalyticSignal.v}
2  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   CICFilter.v, caption = CICFilter.v]{SourceCodes/Verilog/CICFilter.v}
3  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   CICFilter2Channel.v, caption = CICFilter2Channel.v]{SourceCodes/Verilog/
   CICFilter2Channel.v}
4  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   CICFilter6Channel.v, caption = CICFilter6Channel.v]{SourceCodes/Verilog/
   CICFilter6Channel.v}
5  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   ClockDecimator.v, caption = ClockDecimator.v]{SourceCodes/Verilog/
   ClockDecimator.v}
6  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   DFTCore.v, caption = DFTCore.v]{SourceCodes/Verilog/DFTCore.v}
7  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   DFTCoreTop.v, caption = DFTCoreTop.v]{SourceCodes/Verilog/DFTCoreTop.v}
8  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   DFTCoreTopTB.v, caption = DFTCoreTopTB.v]{SourceCodes/Verilog/
   DFTCoreTopTB.v}
9  \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   DIFMArbitrage.v, caption = DIFMArbitrage.v]{SourceCodes/Verilog/
   DIFMArbitrage.v}
10 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   DIFMCorrelator.v, caption = DIFMCorrelator.v]{SourceCodes/Verilog/
   DIFMCorrelator.v}
11 \lstinputlisting[basicstyle = \footnotesize, language = Verilog, label =
   DIFMCorrelator2.v, caption = DIFMCorrelator2.v]{SourceCodes/Verilog/
   DIFMCorrelator2.v}
```