Ch. 10
Finite Impulse Response (FIR) Digital Filters

- Digital Filters
- FIR Theory
- Designing FIR Filters
- Constant Coefficient FIR Design
10.1 Digital Filters

- Digital filters are typically used to modify or alter the attributes of a signal in the time or frequency domain. The most common digital filter is the linear time-invariant (LTI) filter. An LTI interacts with its input signal through a process called linear convolution, denoted by \( y = f \ast x \) where \( f \) is the filter’s impulse response, \( x \) is the input signal, and \( y \) is the convolved output.

The linear convolution process is formally defined by:

\[
y[n] = x[n] \ast f[n] = \sum_{k} x[k] f[n-k] = \sum_{k} f[k] x[n-k]. \tag{10.1}
\]

- LTI digital filters are generally classified as being finite impulse response (i.e., FIR), or infinite impulse response (i.e., IIR). As the name implies, an FIR filter consists of a finite number of sample values, reducing the above convolution sum to a finite sum per output sample instant. An IIR filter, however, requires that an infinite sum be performed.

- The motivation for studying digital filters is found in their growing popularity as a primary DSP operation.

- Digital filters are rapidly replacing classic analog filters, which were implemented using RLC components and operational amplifiers. Analog filters were mathematically modeled using ordinary differential equations of Laplace transforms. They were analyzed in the time or \( s \) (also known as Laplace) domain. Analog prototypes are now only used in IIR design, while FIR are typically designed using direct computer specifications and algorithms.

- In this chapter it is assumed that a digital filter, an FIR in particular, has been designed and selected for implementation. The FIR design process will be briefly reviewed, followed by a discussion of FPGA implementation variations.
10.2 FIR Theory

- An FIR with constant coefficients is an LTI digital filter. The output of an FIR of order or length \( L \), to an input time-series \( x[n] \), is given by a finite version of the convolution sum, namely:

\[
y[n] = x[n] * f[n] = \sum_{k=0}^{L-1} f[k] x[n-k],
\]

where \( f[0] \neq 0 \) through \( f[L-1] \neq 0 \) are the filter’s \( L \) coefficients. They also correspond to the FIR’s impulse response.

- For LTI systems it is sometimes more convenient to express (10.2) in the \( z \)-domain with

\[
Y(z) = F(z)X(z),  \quad (10.3)
\]

where \( F(z) \) is the FIR’s transfer function defined in the \( z \)-domain by

\[
F(z) = \sum_{k=0}^{L-1} f[k] z^{-k}.  \quad (10.4)
\]

The \( L \)th-order LTI FIR filter is graphically interpreted in Fig. 10.1. It can be seen to consist of a collection of a “tapped delay line,” adders, and multipliers. One of the operands presented to each multiplier is an FIR coefficient, often referred to as a “tap weight” for obvious reasons. Historically, the FIR filter is also known by the name “transversal filter,” suggesting its “tapped delay line” structure.
10.2.1 FIR Filter with Transposed Structure

- A variation of the direct FIR model is called the transposed FIR filter. It can be constructed from the FIR filter in Fig. 10.1 by:
  - Exchanging the input and output
  - Inverting the direction of signal flow
  - Substituting an adder by a fork, and vice versa

- A transposed FIR filter is shown in Fig. 10.3 and is, in general, the preferred implementation of an FIR filter.

- The benefit of this filter is that we do not need an extra shift register for $x[n]$, and there is no need for an extra pipeline stage for the adder (tree) of the products to achieve high throughput.

- The following examples show a direct implementation of the transposed filter.

---

**Fig. 10.3. FIR filter in the transposed structure.**

*Ex: FIR Filter with $L = 4$:*
We recall from the discussion of sum-of-product (SOP) computations using a PDSP that, for $Bx$ data/coefficient bit width and filter length $L$, additional $\log_2(L)$ bits for unsigned SOP and $\log_2(L) - 1$ guard bits for signed arithmetic must be provided. For a 9-bit signed data/coefficient and $L = 4$, the adder width must be $9 + 9 + \log_2(4) - 1 = 19$. 
Example 10.1: Programmable FIR Filter (1/4)

The following VHDL code shows the generic specification for an implementation for a length-4 filter.
-- This is a generic FIR filter generator
-- It uses W1 bit data/coefficients bits
LIBRARY lpm; -- Using predefined packages
USE lpm.lpm_components.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY fir_gen IS ------> Interface
GENERIC (W1 : INTEGER := 9; -- Input bit width
W2 : INTEGER := 18;-- Multiplier bit width 2*W1
W3 : INTEGER := 19;-- Adder width = W2+log2(L)-1
W4 : INTEGER := 11;-- Output bit width
L : INTEGER := 4; -- Filter length
Mpipe : INTEGER := 3-- Pipeline steps of multiplier
);

Example 10.1: Programmable FIR Filter (2/4)

PORT ( clk : IN STD_LOGIC;
Load_x : IN STD_LOGIC;
x_in : IN STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
c_in : IN STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
y_out : OUT STD_LOGIC_VECTOR(W4-1 DOWNTO 0));
END fir_gen;
ARCHITECTURE fpga OF fir_gen IS
SUBTYPE N1BIT IS STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
SUBTYPE N2BIT IS STD_LOGIC_VECTOR(W2-1 DOWNTO 0);
SUBTYPE N3BIT IS STD_LOGIC_VECTOR(W3-1 DOWNTO 0);
TYPE ARRAY_N1BIT IS ARRAY (0 TO L-1) OF N1BIT;
TYPE ARRAY_N2BIT IS ARRAY (0 TO L-1) OF N2BIT;
TYPE ARRAY_N3BIT IS ARRAY (0 TO L-1) OF N3BIT;
SIGNAL x : N1BIT;
SIGNAL y : N3BIT;
SIGNAL c : ARRAY_N1BIT; -- Coefficient array
SIGNAL p : ARRAY_N2BIT; -- Product array
SIGNAL a : ARRAY_N3BIT; -- Adder array
Example 10.1: Programmable FIR Filter (3/4)

BEGIN
Load: PROCESS ------> Load data or coefficient
BEGIN
WAIT UNTIL clk = '1';
IF (Load_x = '0') THEN
   c(L-1) <= c_in; -- Store coefficient in register
   FOR I IN L-2 DOWNTO 0 LOOP -- Coefficients shift one
      c(I) <= c(I+1);
   END LOOP;
ELSE
   x <= x_in; -- Get one data sample at a time
END IF;
END PROCESS Load;

Example 10.1: Programmable FIR Filter (4/4)

SOP: PROCESS (clk) ------> Compute sum-of-products
BEGIN
IF clk'event and (clk = '1') THEN
   FOR I IN 0 TO L-2 LOOP -- Compute the transposed
      a(I) <= (p(I)(W2-1) & p(I)) + a(I+1); -- filter adds
   END LOOP;
a(L-1) <= p(L-1)(W2-1) & p(L-1); -- First TAP has
END IF; -- only a register
y <= a(0);
END PROCESS SOP;
-- Instantiate L pipelined multiplier
MulGen: FOR I IN 0 TO L-1 GENERATE
   Muls: lpm_mult -- Multiply p(i) = c(i) * x;
   GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W1,
                LPM_PIPELINE => Mpipe,
                LPM_REPRESENTATION => "SIGNED",
                LPM_WIDTHP => W2,
                LPM_WIDTHS => W2)
   PORT MAP ( clock => clk, dataa => c(I),
               datab => x, result => p(I));
END GENERATE;
y_out <= y(W3-1 DOWNTO W3-W4);
END fpga;
• The first process, Load, is used to load the coefficient in a tapped delay line if Load_x=0. Otherwise, a data word is loaded into the x register.

• The second process, called SOP, implements the sum-of-products computation. The products p(l) are sign-extended by one bit and added to the previous partial SOP. Note also that all multipliers are instantiated by a generate statement, which allows the assignment of extra pipeline stages.

• Finally, the output y_out is assigned the value of the SOP divided by 256, because the coefficients are all assumed to be fractional (i.e., |f[k]| ≤ 1.0). The design uses 184 LEs, 4 embedded multipliers, and has a 329.06MHz Registered Performance.

---

Length-4 filter with Daubechies DB4 filter coefficients

• To simulate this length-4 filter consider a Daubechies DB4 filter coefficient with

\[
G(z) = \frac{1 + \sqrt{3} + (3 + \sqrt{3})z^{-1} + (3 - \sqrt{3})z^{-2} + (1 - \sqrt{3})z^{-3}}{4\sqrt{2}}.
\]

\[
G(z) = 0.48301 + 0.8365z^{-1} + 0.2241z^{-2} - 0.1294z^{-3}.
\]

• Quantizing the coefficients to eight bits (plus a sign bit) of precision results in the following

\[
G(z) = \frac{124 + 214z^{-1} + 57z^{-2} - 33z^{-3}}{256}
\]

\[
= \frac{124}{256} + \frac{214}{256}z^{-1} + \frac{57}{256}z^{-2} - \frac{33}{256}z^{-3}.
\]

• As can be seen from Fig. 10.4, in the first four steps we load the coefficients \{124, 214, 57, -33\} into the tapped delay line. Note that Quartus II can also display signed numbers. As unsigned data the value -33 will be displayed as \(512 - 33 = 479\). Then we check the impulse response of the filter by loading 100 into the x register. The first valid output is then available after 450 ns.
10.2.2 Symmetry in FIR Filters

- The center of an FIR’s impulse response is an important point of symmetry. It is sometimes convenient to define this point as the 0th sample instant. Such filter descriptions are a-causal (centered notation). For an odd-length FIR, the a-causal filter model is given by:

\[ F(z) = \sum_{k=-\frac{L-1}{2}}^{\frac{L-1}{2}} f[k]z^{-k}. \]  

(10.5)

- The FIR’s frequency response can be computed by evaluating the filter’s transfer function about the periphery of the unity circle, by setting \( z = e^{j\omega T} \). It then follows that:

\[ F(\omega) = F(e^{j\omega T}) = \sum_{k} f[k]e^{-j\omega kT}. \]  

(10.6)

- We then denote with \(|F(\omega)|\) the filter’s magnitude frequency response and \( \phi(\omega) \) denotes the phase response, and satisfies:

\[ \phi(\omega) = \arctan \left( \frac{\Im(F(\omega))}{\Re(F(\omega))} \right). \]  

(10.7)

- Digital filters are more often characterized by phase and magnitude than by the z-domain transfer function or the complex frequency transform.
10.2.3 Linear-phase FIR Filters

- Maintaining phase integrity across a range of frequencies is a desired system attribute in many applications such as communications and image processing. As a result, designing filters that establish linear-phase versus frequency is often mandatory. The standard measure of the phase linearity of a system is the “group delay” defined by:

$$\tau(\omega) = -\frac{d\phi(\omega)}{d\omega}. \quad (10.8)$$

- A perfectly linear-phase filter has a group delay that is constant over a range of frequencies. It can be shown that linear-phase is achieved if the filter is symmetric or antisymmetric, and it is therefore preferable to use the a-causal framework of (10.5). From (10.7) it can be seen that a constant group delay can only be achieved if the frequency response $F(\omega)$ is a purely real or imaginary function. This implies that the filter’s impulse response possesses even or odd symmetry. That is:

$$f[n] = f[-n] \quad \text{or} \quad f[n] = -f[-n]. \quad (10.9)$$

Linear-phase FIR Filters (cont’)

- An odd-order even-symmetry FIR filter would, for example, have a frequency response given by:

$$F(\omega) = f[0] + \sum_{k>0} f[k] e^{-jk\omega T} + f[-k] e^{jk\omega T} \quad (10.10)$$

$$= f[0] + 2 \sum_{k>0} f[k] \cos(k\omega T). \quad (10.11)$$

which is seen to be a purely real function of frequency. Table 10.1 summarizes the four possible choices of symmetry, antisymmetry, even order and odd order. In addition, Table 10.1 graphically displays an example of each class of linear-phase FIR.
Linear-phase FIR Filters (cont’)

- The symmetry properties intrinsic to a linear-phase FIR can also be used to reduce the necessary number of multipliers $L$, as shown in Fig. 10.1. Consider the linear-phase FIR shown in Fig. 10.5 (even symmetry assumed), which fully exploits coefficient symmetry. Observe that the “symmetric” architecture has a multiplier budget per filter cycle exactly half of that found in the direct architecture shown in Fig. 10.1 ($L$ versus $L/2$) while the number of adders remains constant at $L - 1$.

![Fig. 3.5. Linear-phase filter with reduced number of multipliers.](image)
10.3 Designing FIR Filters

- Modern digital FIR filters are designed using computer-aided engineering (CAE) tools. The filters used in this chapter are designed using the MatLab Signal Processing toolbox. The toolbox includes an “Interactive Lowpass Filter Design” demo example that covers many typical digital filter designs, including:
  - **Equiripple (also known as minimax) FIR design**, which uses the Parks–McClellan and Remez exchange methods for designing a linear-phase (symmetric) equiripple FIR. This equiripple design may also be used to design a differentiator or Hilbert transformer.
  - **Kaiser window design** using the inverse DFT method weighted by a Kaiser window.
  - **Least square FIR method**. This filter design also has ripple in the passband and stopband, but the mean least square error is minimized.
  - **Four IIR filter design** methods (Butterworth, Chebyshev I and II, and elliptic) which will be discussed in next chapter.

- The FIR methods are individually developed in this section.
- Most often we already know the transfer function (i.e., magnitude of the frequency response) of the desired filter. Such a lowpass specification typically consists of the passband $[0 \ldots \omega_p]$, the transition band $[\omega_p \ldots \omega_s]$, and the stopband $[\omega_s \ldots \pi]$ specification, where the sampling frequency is assumed to be $2\pi$.
- To compute the filter coefficients we may therefore apply the direct frequency method discussed next.
3.3.1 Direct Window Design Method

- The discrete Fourier transform (DFT) establishes a direct connection between the frequency and time domains. Since the frequency domain is the domain of filter definition, the DFT can be used to calculate a set of FIR filter coefficients that produce a filter that approximates the frequency response of the target filter. A filter designed in this manner is called a direct FIR filter.

- A direct FIR filter is defined by:

\[
f[n] = \text{IDFT}(F[k]) = \sum_{k} F[k] e^{j2\pi kn/L}, \quad (10.12)
\]

- From basic signals and systems theory, it is known that the spectrum of a real signal is Hermitian. That is, the real spectrum has even symmetry and the imaginary spectrum has odd symmetry. If the synthesized filter should have only real coefficients, the target DFT design spectrum must therefore be Hermitian or \( F[k] = F^*[−k] \), where the * denotes conjugate complex.

- Consider a length-16 direct FIR filter design with a rectangular window, shown in Fig. 10.6a, with the passband ripple shown in Fig. 10.6b. Note that the filter provides a reasonable approximation to the ideal lowpass filter with the greatest mismatch occurring at the edges of the transition band. The observed “ringing” is due to the Gibbs phenomenon, which relates to the inability of a finite Fourier spectrum to reproduce sharp edges.

- The Gibbs ringing is implicit in the direct inverse DFT method and can be expected to be about ±7% over a wide range of filter orders. To illustrate this, consider the example filter with length 128, shown in Fig. 10.6c, with the passband ripple shown in Fig. 10.6d. Although the filter length is essentially increased (from 16 to 128) the ringing at the edge still has about the same quantity.

- The effects of ringing can only be suppressed with the use of a data “window” that tapers smoothly to zero on both sides. Data windows overlay the FIR’s impulse response, resulting in a “smoother” magnitude frequency response with an attendant widening of the transition band. If, for instance, a Kaiser window is applied to the FIR, the Gibbs ringing can be reduced as shown in Fig. 10.7(upper). The deleterious effect on the transition band can also be seen. Other classic window functions are summarized in Table 10.2.
The deleterious effect on the transition band can also be seen. Other classic window functions are summarized in Table 10.2. They differ in terms of their ability to make tradeoffs between “ringing” and transition bandwidth extension. The number of recognized and published window functions is large. The most common windows, denoted \( w[n] \), are:

- Rectangular: \( w[n] = 1 \)
- Bartlett (triangular): \( w[n] = 2n/N \)
- Hanning: \( w[n] = 0.5(1 - \cos(2\pi n/L)) \)
- Hamming: \( w[n] = 0.54 - 0.46\cos(2\pi n/L) \)
- Blackman: \( w[n] = 0.42 - 0.5\cos(2\pi n/L) + 0.08\cos(4\pi n/L) \)
- Kaiser: \( w[n] = I_0 \left( \beta \sqrt{1 - (n - L/2)^2/(L/2)^2} \right) \)

Table 10.2 shows the most important parameters of these windows.
Table 10.2. Parameters of commonly used window functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>3-dB bandwidth</th>
<th>First zero</th>
<th>Maximum sidelobe decrease per octave</th>
<th>Equivalent Kaiser $\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangular</td>
<td>$0.89/T$</td>
<td>$1/T$</td>
<td>$-13$ dB</td>
<td>0</td>
</tr>
<tr>
<td>Bartlett</td>
<td>$1.28/T$</td>
<td>$2/T$</td>
<td>$-27$ dB</td>
<td>1.33</td>
</tr>
<tr>
<td>Hanning</td>
<td>$1.44/T$</td>
<td>$2/T$</td>
<td>$-32$ dB</td>
<td>3.86</td>
</tr>
<tr>
<td>Hamming</td>
<td>$1.33/T$</td>
<td>$2/T$</td>
<td>$-42$ dB</td>
<td>4.86</td>
</tr>
<tr>
<td>Blackman</td>
<td>$1.79/T$</td>
<td>$3/T$</td>
<td>$-74$ dB</td>
<td>7.04</td>
</tr>
<tr>
<td>Kaiser</td>
<td>$1.44/T$</td>
<td>$2/T$</td>
<td>$-38$ dB</td>
<td>3</td>
</tr>
</tbody>
</table>

The 3-dB bandwidth shown in Table 10.2 is the bandwidth where the transfer function is decreased from DC by 3 dB or $\approx 1/\sqrt{2}$. Data windows also generate sidelobes, to various degrees, away from the 0th harmonic. Depending on the smoothness of the window, the third column in Table 10.2 shows that some windows do not have a zero at the first or second zero DFT frequency $1/T$. The maximum sidelobe gain is measured relative to the 0th harmonic value. The fifth column describes the asymptotic decrease of the window per octave. Finally, the last column describes the value $\beta$ for a Kaiser window that emulates the corresponding window properties.

Fig. 3.7. (upper) Kaiser window design with $L = 59$. (lower) Parks-McClellan design with $L = 27$. (a) Transfer function. (b) Group delay of passband. (c) Zero plot.
• The Kaiser window, based on the first-order Bessel function $I_0$, is special in two respects. It is nearly optimal in terms of the relationship between “ringing” suppression and transition width, and second, it can be tuned by $\beta$, which determines the ringing of the filter. This can be seen from the following equation credited to Kaiser.

$$\beta = \begin{cases} 0.1102(A - 8.7) & A > 50, \\ 0.0842(A - 21)^0.4 + 0.07886(A - 21) & 21 \leq A \leq 50, \\ 0 & A < 21, \end{cases}$$  \tag{10.13}

• where $A = 20\log_{10} \epsilon_r$ is both stopband attenuation and the passband ripple in dB. The Kaiser window length to achieve a desired level of suppression can be estimated:

$$L = \frac{A - 8}{2.285(\omega_s - \omega_p)} + 1. \tag{10.14}$$

• The length is generally correct within an error of ±2 taps.

### 3.3.2 Equiripple Design Method

• A typical filter specification not only includes the specification of passband $\omega_p$ and stopband $\omega_s$ frequencies and ideal gains, but also the allowed deviation (or ripple) from the desired transfer function. The transition band is most often assumed to be arbitrary in terms of ripples. A special class of FIR filter that is particularly effective in meeting such specifications is called the equiripple FIR. An equiripple design protocol minimizes the maximal deviations (ripple error) from the ideal transfer function. The equiripple algorithm applies to a number of FIR design instances. The most popular are:

  – **Lowpass filter design** (in MatLab use firpm(L,F,A,W)), with tolerance scheme as shown in Fig. 10.8a
  
  – **Hilbert filter**, i.e., a unit magnitude filter that produces a 90° phase shift for all frequencies in the passband (in MatLab use firpm(L,F,A,’Hilbert’))
  
  – **Differentiator filter** that has a linear increasing frequency magnitude proportional to $\omega$ (in MatLab use firpm(L,F,A,’differentiator’))
The equiripple or minimum-maximum algorithm is normally implemented using the Parks–McClellan iterative method. The Parks–McClellan method is used to produce a equiripple or minimax data fit in the frequency domain. It is based on the “alternation theorem” that says that there is exactly one polynomial, a Chebyshev polynomial with minimum length, that fits into a given tolerance scheme. Such a tolerance scheme is shown in Fig. 10.8a, and Fig. 10.8b shows a polynomial that fulfills this tolerance scheme. The length of the polynomial, and therefore the filter, can be estimated for a lowpass with

$$L = \frac{-10 \log_{10}(\varepsilon_p \varepsilon_s) - 13}{2.324(\omega_s - \omega_p)} + 1, \quad (10.15)$$

where $\varepsilon_p$ is the passband and $\varepsilon_s$ the stopband ripple.
• The algorithm iteratively finds the location of locally maximum errors that deviate from a nominal value, reducing the size of the maximal error per iteration, until all deviation errors have the same value. Most often, the Remez method is used to select the new frequencies by selecting the frequency set with the largest peaks of the error curve between two iterations. This is why the MatLab equiripple function was called remez in the past (now renamed to firpm for Parks-McClellan).

• Compared to the direct frequency method, with or without data windows, the advantage of the equiripple design method is that passband and topband deviations can be specified differently. This may, for instance, be useful in audio applications where the ripple in the passband may be specified to be higher, because the ear only perceives differences larger than 3 dB. We note from Fig. 10.7(lower) that the equiripple design having the same tolerance requirements as the Kaiser window design enjoys a considerably reduced filter order, i.e., 27 compared with 59.

10.4 Constant Coefficient FIR Design

• There are only a few applications (e.g., adaptive filters) where we need a general programmable filter architecture like the one shown in Example 10.1

• In many applications, the filters are LTI (i.e., linear time invariant) and the coefficients do not change over time. In this case, the hardware effort can essentially be reduced by exploiting the multiplier and adder (trees) needed to implement the FIR filter arithmetic.

• With available digital filter design software the production of FIR coefficients is a straightforward process. The challenge remains to map the FIR design into a suitable architecture.

• The direct or transposed forms are preferred for maximum speed and lowest resource utilization. Lattice filters are used in adaptive filters because the filter can be enlarged by one section, without the need for recomputation of the previous lattice sections. But this feature only applies to PDSPs and is less applicable to FPGAs. We will therefore focus our attention on the direct and transposed implementations.

• We will start with possible improvements to the direct form and will then move on to the transposed form. At the end of the section we will discuss an alternative design approach using distributed arithmetic.
10.4.1 Direct FIR Design

- The direct FIR filter shown in Fig. 10.1 can be implemented in VHDL using (sequential) PROCESS statements or by “component instantiations” of the adders and multipliers. A PROCESS design provides more freedom to the synthesizer, while component instantiation gives full control to the designer.

- To illustrate this, a length-4 FIR will be presented as a PROCESS design. Although a length-4 FIR is far too short for most practical applications, it is easily extended to higher orders and has the advantage of a short compiling time.

- The linear-phase (therefore symmetric) FIR’s impulse response is assumed to be given by

\[ f[k] = \{-1.0, 3.75, 3.75, -1.0\}. \tag{10.16} \]

- These coefficients can be directly encoded into a 5-bit fractional number. For example, 3.75 would have a 5-bit binary representation 011.11 where “,” denotes the location of the binary point. Note that it is, in general, more efficient to implement only positive CSD coefficients, because positive CSD coefficients have fewer nonzero terms and we can take the sign of the coefficient into account when the summation of the products is computed. See also the first step in the RAG algorithm discussed later.

- In a practical situation, the FIR coefficients are obtained from a computer design tool and presented to the designer as floating-point numbers. The performance of a fixed-point FIR, based on floating-point coefficients, needs to be verified using simulation or algebraic analysis to ensure that design specifications remain satisfied. In the above example, the floating-point numbers are 3.75 and 1.0, which can be represented exactly with fixed-point numbers, and the check can be skipped.

- Another issue that must be addressed when working with fixed-point designs is protecting the system from dynamic range overflow. Fortunately, the worst-case dynamic range growth \( G \) of an \( L \)th-order FIR is easy to compute and it is:

\[ G \leq \log_2 \left( \sum_{k=0}^{L-1} |f[k]| \right). \tag{10.17} \]

- The total bit width is then the sum of the input bit width and the bit growth \( G \). For the above filter for (10.16) we have \( G = \log_2(9.5) < 4 \), which states that the system’s internal data registers need to have at least four more integer bits than the input data to insure no overflow. If 8-bit internal arithmetic is used the input data should be bounded by ±128/9.5 = ±13.
Example 10.2: Four-tap Direct FIR Filter (1/2)

- The VHDL design for a filter with coefficients \((-1, 3.75, 3.75, -1)\) is shown in the following listing.

```vhdl
PACKAGE eight_bit_int IS -- User-defined types
  SUBTYPE BYTE IS INTEGER RANGE -128 TO 127;
  TYPE ARRAY_BYTE IS ARRAY (0 TO 3) OF BYTE;
END eight_bit_int;
LIBRARY work;
USE work.eight_bit_int.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY fir_srg IS ------> Interface
  PORT ( clk : IN STD_LOGIC;
         x : IN BYTE;
         y : OUT BYTE);
END fir_srg;
ARCHITECTURE flex OF fir_srg IS
  SIGNAL tap : ARRAY_BYTE := (0,0,0,0); -- Tapped delay line of bytes
BEGIN
  p1: PROCESS ------> Behavioral style
  BEGIN
    WAIT UNTIL clk = '1'; -- Compute output y with the filter coefficients weight.
    -- The coefficients are \([-1, 3.75, 3.75, -1]\).
    -- Division for Altera VHDL is only allowed for powers-of-two values!
    y <= 2 * tap(1) + tap(1) + tap(1) / 2 + tap(1) / 4
        + 2 * tap(2) + tap(2) + tap(2) / 2 + tap(2) / 4
        - tap(3) + tap(0);
    FOR I IN 3 DOWNTO 1 LOOP
      tap(I) <= tap(I-1); -- Tapped delay line: shift one
    END LOOP;
    tap(0) <= x; -- Input in register 0
  END PROCESS;
END flex;
```

Example 10.2: Four-tap Direct FIR Filter (2/2)

```vhdl
ARCHITECTURE flex OF fir_srg IS
  SIGNAL tap : ARRAY_BYTE := (0,0,0,0); -- Tapped delay line of bytes
BEGIN
  p1: PROCESS ------> Behavioral style
  BEGIN
    WAIT UNTIL clk = '1'; -- Compute output y with the filter coefficients weight.
    -- The coefficients are \([-1, 3.75, 3.75, -1]\).
    -- Division for Altera VHDL is only allowed for powers-of-two values!
    y <= 2 * tap(1) + tap(1) + tap(1) / 2 + tap(1) / 4
        + 2 * tap(2) + tap(2) + tap(2) / 2 + tap(2) / 4
        - tap(3) + tap(0);
    FOR I IN 3 DOWNTO 1 LOOP
      tap(I) <= tap(I-1); -- Tapped delay line: shift one
    END LOOP;
    tap(0) <= x; -- Input in register 0
  END PROCESS;
END flex;
```
The design is a literal interpretation of the direct FIR architecture found in Fig. 10.1. The design is applicable to both symmetric and asymmetric filters. The output of each tap of the tapped delay line is multiplied by the appropriately weighted binary value and the results are added. The impulse response $y$ of the filter to an impulse $10$ is shown in Fig. 10.9.

There are three obvious actions that can improve this design:

1) **Realize each filter coefficient with an optimized CSD code**

2) **Increase effective multiplier speed by pipelining.** The output **adder** should be arranged in a pipelined balance tree. If the coefficients are coded as “powers-of-two,” the pipelined multiplier and the adder tree can be merged. Pipelining has low overhead due to the fact that the LE registers are otherwise often unused. A few additional pipeline registers may be necessary if the number of terms in the tree to be added is not a power of two.

3) **For symmetric coefficients, the multiplication complexity can be reduced** as shown in Fig. 10.5.

The first two actions are applicable to all FIR filters, while the third applies only to linear-phase (symmetric) filters. These ideas will be illustrated by example designs.
Example 10.3: Improved Four-tap Direct FIR Filter

- The design from the previous example can be improved using a CSD code for the coefficients $3.75 = 2^2 - 2^{-2}$. In addition, symmetry and pipelining can also be employed to enhance the filter’s performance. Table 10.3 shows the maximum throughput that can be expected for each different design. CSD coding and symmetry result in smaller, more compact designs. Improvements in Registered Performance are obtained by pipelining the multiplier and providing an adder tree for the output accumulation.
- Two additional pipeline registers (i.e., 16 LEs) are necessary, however. The most compact design is expected using symmetry and CSD coding without the use of an adder tree.
- The partial VHDL code for producing the filter output $y$ is shown below.

```vhdl
  t1 <= tap(1) + tap(2); -- Using symmetry
  t2 <= tap(0) + tap(3);
  IF rising_edge(clk) THEN
    y <= 4 * t1 - t1 / 4 - t2; Apply CSD code and add
  ...  
```

Example 10.3: Improved Four-tap Direct FIR Filter

- The fastest design is obtained when all three enhancements are used. The partial VHDL code, in this case, becomes:

```vhdl
  WAIT UNTIL clk = '1'; -- Pipelined all operations
  t1 <= tap(1) + tap(2); -- Use symmetry of coefficients
  t2 <= tap(0) + tap(3); -- and pipeline adder
  t3 <= 4 * t1 - t1 / 4; -- Pipelined CSD multiplier
  t4 <= -t2; -- Build a binary tree and add delay
  y <= t3 + t4;
  ...
```
Table 10.3. Improved FIR filter

<table>
<thead>
<tr>
<th></th>
<th>Symmetry</th>
<th>CSD</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Speed/MHz</td>
<td>99.17</td>
<td>178.83</td>
<td>123.59</td>
</tr>
<tr>
<td>Size/LEs</td>
<td>114</td>
<td>99</td>
<td>65</td>
</tr>
</tbody>
</table>

Direct Form Pipelined FIR Filter

- Sometimes a single coefficient has more pipeline delay than all the other coefficients. We can model this delay by \( f[n]z^{-d} \). If we now add a positive delay with
  \[
  f[n] = z^d f[n] z^{-d}
  \]  
  (10.18)
  - the two delays are eliminated. Translating this into hardware means that for the direct form FIR filter we have to use the output of the previous register.
  - This principle is shown in Fig. 10.10a. Figure 10.10b shows an example of rephasing a pipelined multiplier that has two delays.
10.4.2 FIR Filter with Transposed Structure

- A variation of the direct FIR filter is called the transposed filter and has been discussed in Sect. 10.2.1. The transposed filter enjoys, in the case of a constant coefficient filter, the following two additional improvements compared with the direct FIR:
  - Multiple use of the repeated coefficients using the reduced adder graph (RAG) algorithm
  - Pipeline adders using a carry-save adder

- The pipeline adder increases the speed, at additional adder and register costs, while the RAG principle will reduce the size (i.e., number of LEs) of the filter and sometimes also increase the speed.
• it was noted that it can sometimes be advantageous to implement the factors of a constant coefficient, rather than implement the CSD code directly. For example, the CSD code realization of the constant multiplier coefficient 93 requires three adders, while the factors $3 \times 31$ only requires two adders.

• For a transposed FIR filter, the probability is high that all the coefficients will have several factors in common. For instance, the coefficients 9 and 11 can be built using $8 + 1 = 9$ for the first and $11 = 9 + 2$ for the second. This reduces the total effort by one adder.

• In general, however, finding the optimal reduced adder graph (RAG) is an NP-hard problem. As a result, heuristics must be used. The RAG algorithm first suggested by Dempster and Macleod is described next.
10.4.3 FIR Filters Using Distributed Arithmetic

- A completely different FIR architecture is based on the distributed arithmetic (DA) concept. In contrast to a conventional sum-of-products architecture, in distributed arithmetic we always compute the sum of products of a specific bit \( b \) over all coefficients in one step. This is computed using a small table and an accumulator with a shifter. To illustrate, consider the three-coefficient FIR with coefficients \( \{2, 3, 1\} \).

Ex 10.7: Distributed Arithmetic Filter as State Machine

- A distributed arithmetic filter can be built in VHDL code using the following state machine description:

- Codes: See files: dasfm.vhd and case3.vhd

Fig. 10.13. Simulation of the 3-tap FIR filter with input \( \{1, 3, 7\} \).
• By defining the distributed arithmetic table with a CASE statement, the synthesizer will use logic cells to implement the LUT. This will result in a fast and efficient design only if the tables are small.

• For large tables, alternative means must be found. In this case, we may use the 4-kbit embedded memory blocks (M4Ks), which can be configured as $2^9 \times 9$, $2^{10} \times 4$, $2^{11} \times 2$ or $2^{12} \times 1$ tables.

Distributed Arithmetic Using Logic Cells

• The DA implementation of an FIR filter is particularly attractive for loworder cases due to LUT address space limitations (e.g., $L \leq 4$). It should be remembered, however, that FIR filters are linear filters. This implies that the outputs of a collection of low-order filters can be added together to define the output of a high-order FIR.

• Based on the LEs found in a Cyclone II device, namely $2^4 \times 1$-bit tables, a DA table for four coefficients can be implemented. The number of necessary LEs increases exponentially with order. Typically, the number of LEs is much higher than the number of M4Ks. For example, an EP2C35 contains 35K LEs but only 105M4Ks. Also, M4Ks can be used to efficiently implement RAMs and FIFOs and other high-valued functions. It is therefore sometimes desirable to use M4Ks economically.
• On the other side if the design is implemented using larger tables with a $2^b \times b$ CASE statement, inefficient designs can result. The pipelined $2^3 \times 9$ table implemented with one VHDL CASE statement only, for example, required over 100 LEs.

• Figure 10.14 shows the number of Les necessary for tables having three to nine bits inputs and outputs using the CASE statement generated with utility program dagen3e.exe.

Fig. 10.14. Size comparison of synthesis results for different coding using the CASE statement with $b$ input and outputs.

• Another alternative is the design using 4-input LUT only via a CASE statements, and implementing table with more than 4 inputs with an additional (binary tree) multiplexer using $2 \rightarrow 1$ multiplexer only. In this model it is straightforward to add additional pipeline registers to the modular design.

• For maximum speed, a register must be introduced behind each LUT and $2 \rightarrow 1$ multiplexer. This will, most likely, yield a higher LE count compared to the minimization of the one large LUT. The following example illustrates the structure of a 5-input table.
Example 10.8: Five-input DA Table

- The utility program dagen3e.exe accepts filter length and coefficients, and returns the necessary PROCESS statements for the 4-input CASE table followed by a multiplexer. The VHDL output for an arbitrary set of coefficients, namely \{1, 3, 5, 7, 9\}, is given in the following listing:
  - Files: case5p.vhd
- The five inputs produce two CASE tables and a \(2 \rightarrow 1\) bus multiplexer. The multiplexer may also be realized with a component instantiation using the LPM function busmux. The program dagen3e.exe writes a VHDL file with the name caseX.vhd, where X is the filter length that is also the input bit width. The file caseXp.vhd is the same table, except with additional pipeline registers. The component can be used directly in a state machine design or in an unrolled filter structure.

- Figure 10.15 compares the different design methods in terms of speed. We notice that the busmux generated VHDL code allows to run all pipelined designs with the maximum speed of 464MHz outperforming the M4Ks by nearly a factor two. Without pipeline stages the synthesis tools is capable to reduce the LE count essentially, but Registered Performance is also reduced. Note that still a busmux design is used.
- The synthesis tool is not able to optimize one (large) case statement in the same way. Although we get a high Registered Performance using eight pipeline stages for a \(2^9 \times 9\) table with 464MHz the design may now be too large for some applications.
DA Using Embedded Array Blocks

- As mentioned in the last section, it is not economical to use the 4-kbit M4Ks for a short FIR filter, mainly because the number of available M4Ks is limited.
- Also, the maximum registered speed of an M4K is 260MHz, and an LE table implementation may be faster. The following example shows the DA implementation using a component instantiation of the M4K.
Example 10.9: Distributed Arithmetic Filter using M4Ks

- The CASE table from the last example can be replaced by a M4K ROM. The ROM table is defined by file darom3.mif. The default input and output configuration of the M4K is given by "REGISTERED." If it is not desirable to have a registered configuration, set LPM ADDRESS CONTROL => "UNREGISTERED" or LPM OUTDATA => "UNREGISTERED."

- Note that in Cyclone II at least one input must be registered. With Flex devices we can also build asynchronous, i.e., non registered M2K ROMs. The VHDL code for the DA state machine design is shown below: (darom.vhd and darom3.mif)

- Compared with Example 10.7, we now have a component instantiation of the LPM_ROM. Because there is a need to convert between the STD_LOGIC_VECTOR output of the ROM and the integer, we have used the package std_logic_unsigned from the library ieee. The latter contains the CONV_INTEGER function for unsigned STD_LOGIC_VECTOR.

- The design runs at 218.29MHz and uses 27 LEs, and one M4K memory block (more precisely, 24 bits of an M4K).

- The simulation results, shown in Fig. 10.16, are very similar to the dafsm simulation shown in Fig. 10.13. Due to the mandatory 1 clock cycle delay of the synchronous M4K memory block we notice a delay by one clock cycle in the lut output signal; the result \( y = 18 \) for the input sequence \( \{1, 3, 7\} \), however, is still correct. The simulation shows the clk, reset, state, and count signals followed by the three input signals. Next the three bits selected from the input word to address the prestored DA LUT are shown. The LUT output values \( \{6, 4, 1\} \) are then weighted and accumulated to generate the final output value \( y = 18 = 6 + 4 \times 2 + 1 \times 4 \).
But M4Ks have only a single address decoder and if we implement a $2^3 \times 3$ table, a complete M4K would be consumed unnecessarily, and it can not be used elsewhere. For longer filters, however, the use of M4Ks is attractive because:

- M4Ks have registered throughput at a constant 260 MHz, and
- Routing effort is reduced

**Signed DA FIR Filter**

- A signed DA filter will require a signed accumulator. The following example shows the VHDL code for the three-coefficient FIR filter example,

- **Example 10.10: Signed DA FIR Filter**
  
  - For the signed DA filter, an additional state is required. See the variable count to process the sign bit.
  
  - Files: dasign.vhd

- Figure 10.17 shows the simulation for the input sequence \{1, -3, 7\}. *The simulation* shows the clk, reset, state, and count signals followed by the four input signals. Next the three bits selected from the input word to address the prestored DA LUT are shown. The LUT output values \{2, 1, 4, 3\} are then weighted and accumulated to generate the final output value $y = 2 + 1 \times 2 + 4 \times 4 - 3 \times 8 = -4$. *The design uses 56 LEs, no embedded multiplier*, and has a 236.91MHz Registered Performance.
Fig. 10.17. Simulation of the 3-tap signed FIR filter with input \(\{1, -3, 7\}\).

To accelerate a DA filter, unrolled loops can be used. The input is applied sample by sample (one word at a time), in a bit-parallel form. In this case, for each bit of input a separate table is required. While the table size varies (input bit width equals number of filter taps), the contents of the tables are the same. The obvious advantage is a reduction of VHDL code size, if we use a component definition for the LE tables, as previously presented. To demonstrate, the unrolling of the 3-coefficients, 4-bit input example, previously considered, is developed below.

**Example 10.11: Loop Unrolling for DA FIR Filter**

- In a typical FIR application, the input values are processed in word parallel form (i.e., see Fig. 10.18). The following VHDL code illustrates the unrolled DA code, according to Fig. 10.18.

  -- File: dapara.vhd

Fig. 10.19. Simulation results for the parallel distributed arithmetic FIR filter.
Fig. 10.18. Parallel implementation of a distributed arithmetic FIR filter.

- The previous design requires no embedded multiplier, 33LEs, no M4K memory block, and runs at 214.96MHz. An important advantage of the DA concept, compared with the general-purpose MAC design, is that pipelining is easy achieved. We can add additional pipeline registers to the table output and at the adder-tree output with no cost. To compute \( y \), we replace the line

  \[
  y <= h(0) + 2 * h(1) + 4 * h(2) - 8 * h(3);
  \]

- In a first step we only pipeline the adders. We use the signals \( s0 \) and \( s1 \) for the pipelined adder within the PROCESS statement, i.e.,

  \[
  s0 <= h(0) + 2 * h(1); s1 <= h(2) - 2 * h(3);
  \]
  \[
  y <= s0 + 4 * s1;
  \]

- and the Registered Performance increase to 368.60MHz, and about the same number of LEs are used. For a fully pipeline version we also need to store the case LUT output in registers; the partial VHDL code then becomes:

  \[
  t0 <= h(0); t1 <= h(1); t2 <= h(2); t3 <= h(3);
  \]
  \[
  s0 <= t0 + 2 * t1; s1 <= t2 - 2 * t3;
  \]
  \[
  y <= s0 + 4 * s1;
  \]

- The size of the design increases to 47LEs, because the registers of the LE that hold the case tables can no longer be used for the \( x \) input shift register. But the Registered Performance increases from 214.96MHz to 420MHz.
10.4.4 IP Core FIR Filter Design

- Altera and Xilinx usually also offer with the full subscription an FIR filter generator, since this is one of the most often used intellectual property (IP) blocks.

- FPGA vendors in general prefer distributed arithmetic (DA)-based FIR filter generators since these designs are characterized by:
  - **fully pipelined architecture**
  - **short compile time**
  - good resource estimation
  - *area results independent from the coefficient values, in contrast to the RAG algorithm*

- DA-based filters do not require any coefficient optimization or the computation of a RAG graph, which may be time consuming when the coefficient set is large. DA-based code generation including all VHDL code and testbenches is done in a few seconds using the vendor’s FIR compilers.

- The Altera FIR compiler MegaCore function generates FIR filters optimized for Altera devices. Stratix and Cyclone II devices are supported but no mature devices from the APEX or Flex family. You can use the IP toolbench MegaWizard design environment to specify a variety of filter architectures, including fixed-coefficient, multicycle variable, and multirate filters.

- The FIR compiler includes a coefficient generator, but can also load and use predefined (for instance computed via MatLab) coefficients from a file.
Example 3.12: F6 Half-band Filter IP Generation

- To start the Altera FIR compiler we select the MegaWizard Plug-In Manager under the Tools menu and the library selection window will pop up. The FIR compiler can be found under DSP→Filters. You need to specify a design name for the core and then proceed to the ToolBench.

- We first parameterize the filter and, since we want to use the F6 coefficients, we select Edit Coefficient Set and load the coefficient filter by selecting Imported Coefficient Set. The coefficient file is a simple text file with each line listing a single coefficient, starting with the first coefficient in the first line. The coefficients can be integer or floating-point numbers, which will then be quantized by the tool since only integer-coefficient filters can be generated with the FIR compiler. The coefficients are shown in the impulse response window as shown in Fig. 10.20b and can be modified if needed.

- After loading the coefficients we can then select the Structure to be fully parallel, fully serial, multi-bit serial, or multicycle. We select Distributed Arithmetic: Fully Parallel Filter. We set the input coefficient width to 8 bit and let the tool compute the output bitwidth based on the method Actual Coefficients. We select Coefficient Scaling as None since our integer coefficients should not be further quantized. The transfer function in integer and floating-point should therefore be seen as matching lines, see Fig. 10.21. The FIR compiler reports an estimated size of 312 LEs.

- We skip step 2 from the toolbench since the design is small and we will use the compiled data to verify and simulate the design. We proceed with step 3 and the generation of the VHDL code and all supporting files follows. These files are listed in Table 10.6. We see that not only are the VHDL and Verilog files generated along with their component files, but MatLab (bit accurate) and Quartus II (cycle accurate) test vectors are also provided to enable an easy verification path.
• We then instantiate the FIR core in a wrapper file that also includes registers for the input and output values. We then compile the HDL code of the filter to enable a timing simulation and provide precise resource data. The impulse response simulation of the F6 filter is shown in Figure 10.22. We see that two additional control signals rdy_to_ld and done have been synthesized, although we did not ask for them.

Table 10.6. IP files generation for FIR core.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f6_core.vhd</td>
<td>A MegaCore function variation file, which defines a top-level VHDL description of the custom MegaCore function</td>
</tr>
<tr>
<td>f6_core_inst.vhd</td>
<td>VHDL sample instantiation file</td>
</tr>
<tr>
<td>f6_core.vhd</td>
<td>A VHDL component declaration for the MegaCore function variation</td>
</tr>
<tr>
<td>f6_core.inc</td>
<td>An HDL include declaration file for the MegaCore function variation function</td>
</tr>
<tr>
<td>f6_core.bb.v</td>
<td>Verilog HDL block-box file for the MegaCore function variation</td>
</tr>
<tr>
<td>f6_core.bsf</td>
<td>Quartus II symbol file to be used in the Quartus II block diagram editor</td>
</tr>
<tr>
<td>f6_core_st.v</td>
<td>Generated FIR filter netlist</td>
</tr>
<tr>
<td>f6_core</td>
<td>This file contains the necessary constraints to achieve FIR filter size and speed</td>
</tr>
<tr>
<td>_constraints.tcl</td>
<td></td>
</tr>
<tr>
<td>f6_core_matlab.m</td>
<td>This file provides a MATLAB simulation model for the customized FIR filter</td>
</tr>
<tr>
<td>f6_core_tb.m</td>
<td>This file provides a MATLAB testbench for the customized FIR filter</td>
</tr>
<tr>
<td>f6_core_vec.m</td>
<td>This file provides simulation test vectors to be used simulating the customized FIR filter with the Quartus II software</td>
</tr>
<tr>
<td>f6_core.html</td>
<td>The MegaCore function report file</td>
</tr>
</tbody>
</table>