3. Finite Impulse Response (FIR) Digital Filters

3.1 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 $y = f * x$ 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:

$$y[n] = x[n] * f[n] = \sum_k x[k]f[n-k] = \sum_k f[k]x[n-k]. \tag{3.1}$$

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. An FIR design and implementation methodology is discussed in this chapter, while IIR filter issues are addressed in Chap. 4.

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.

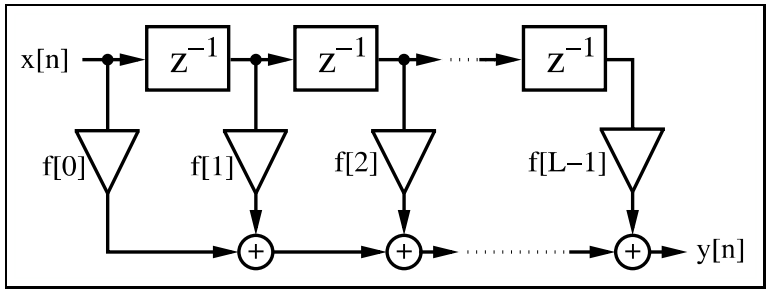


Fig. 3.1. Direct form FIR filter.

3.2 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 given in (3.1), namely:

$$y[n] = x[n] * f[n] = \sum_{k=0}^{L-1} f[k]x[n-k], \tag{3.2}$$

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 (3.2) in the z-domain with

$$Y(z) = F(z)X(z), \tag{3.3}$$

where $F(z)$ is the FIR's *transfer function* defined in the z-domain by

$$F(z) = \sum_{k=0}^{L-1} f[k]z^{-k}. \tag{3.4}$$

The L^{th}-order LTI FIR filter is graphically interpreted in Fig. 3.1. 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.

The *roots* of polynomial $F(z)$ in (3.4) define the zeros of the filter. The presence of only zeros is the reason that FIRs are sometimes called *all zero filters*. In Chap. 5 we will discuss an important class of FIR filters (called CIC filters) that are *recursive* but also FIR. This is possible because the poles produced by the recursive part are canceled by the nonrecursive part of the filter. The effective pole/zero plot also then has *only* zeros, i.e., is an *all-zero filter* or FIR. We note that nonrecursive filters are always FIR, but recursive filters can be either FIR or IIR. Figure 3.2 illustrates this dependence.

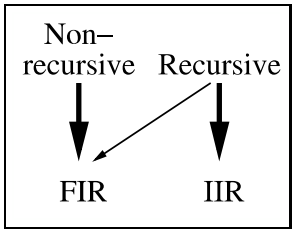


Fig. 3.2. Relation between structure and impulse length.

3.2.1 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. 3.1 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. 3.3 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.

Example 3.1: Programmable FIR Filter

We recall from the discussion of sum-of-product (SOP) computations using a PDSP (see Sect. 2.7, p. 114) that, for B_x 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 $9 + 9 + \log_2(4) - 1 = 19$.

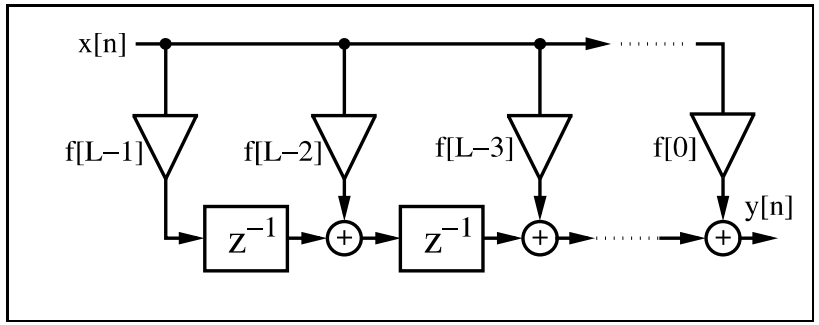


Fig. 3.3. FIR filter in the transposed structure.

The following VHDL code[2] 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
           );
  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

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);
```

[2] The equivalent Verilog code `fir_gen.v` for this example can be found in Appendix A on page 680. Synthesis results are shown in Appendix B on page 731.

```
    END LOOP;
  ELSE
    x <= x_in;           -- Get one data sample at a time
  END IF;
END PROCESS Load;


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 => x,
                 datab => c(I), 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(I)` 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., $|f[k]| \leq 1.0$). The design uses 184 LEs, 4 embedded multipliers, and has a 329.06 MHz `Registered Performance`.

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 model:

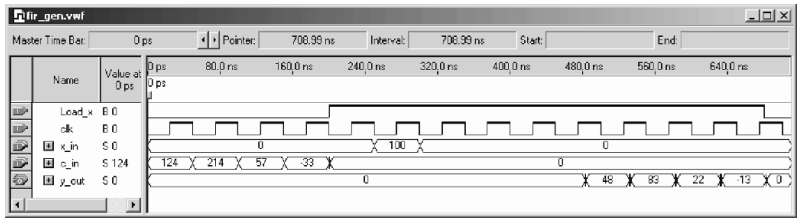


Fig. 3.4. Simulation of the 4-tap programmable FIR filter with Daubechies filter coefficient loaded.

$$G(z) = \left(124 + 214z^{-1} + 57z^{-2} - 33z^{-3}\right)/256$$
$$= \frac{124}{256} + \frac{214}{256}z^{-1} + \frac{57}{256}z^{-2} - \frac{33}{256}z^{-3}.$$

As can be seen from Fig. 3.4, 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.

3.1

3.2.2 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 0^{th} sample instant. Such filter descriptions are *a-causal* (centered notation). For an odd-length FIR, the a-causal filter model is given by:

$$F(z) = \sum_{k=-(L-1)/2}^{(L-1)/2} f[k]z^{-k}. \tag{3.5}$$

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:

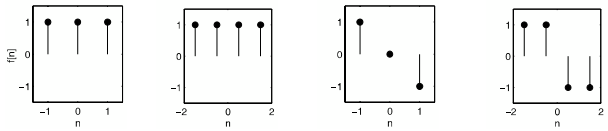
$$F(\omega) = F(e^{j\omega T}) = \sum_k f[k]e^{-j\omega kT}. \tag{3.6}$$

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

$$\phi(\omega) = \arctan\left(\frac{\Im(F(\omega))}{\Re(F(\omega))}\right). \tag{3.7}$$

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

Table 3.1. Four possible linear-phase FIR filters $F(z) = \sum_k f[k]z^{-k}$.

Symmetry L	$f[n] = f[-n]$ odd	$f[n] = f[-n]$ even	$f[n] = -f[-n]$ odd	$f[n] = -f[-n]$ even
Example				
Zeros at	$\pm 120°$	$\pm 90°, 180°$	$0°, 180°$	$0°, 2 \times 180°$

3.2.3 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:

$$\tau(\omega) = -\frac{\mathrm{d}\phi(\omega)}{\mathrm{d}\omega}. \tag{3.8}$$

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 (3.5). From (3.7) 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:

$$f[n] = f[-n] \quad \text{or} \quad f[n] = -f[-n]. \tag{3.9}$$

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

$$F(\omega) = f[0] + \sum_{k>0} f[k]\mathrm{e}^{-\mathrm{j}k\omega T} + f[-k]\mathrm{e}^{\mathrm{j}k\omega T} \tag{3.10}$$

$$= f[0] + 2\sum_{k>0} f[k]\cos(k\omega T), \tag{3.11}$$

which is seen to be a purely real function of frequency. Table 3.1 summarizes the four possible choices of symmetry, antisymmetry, even order and odd order. In addition, Table 3.1 graphically displays an example of each class of linear-phase FIR.

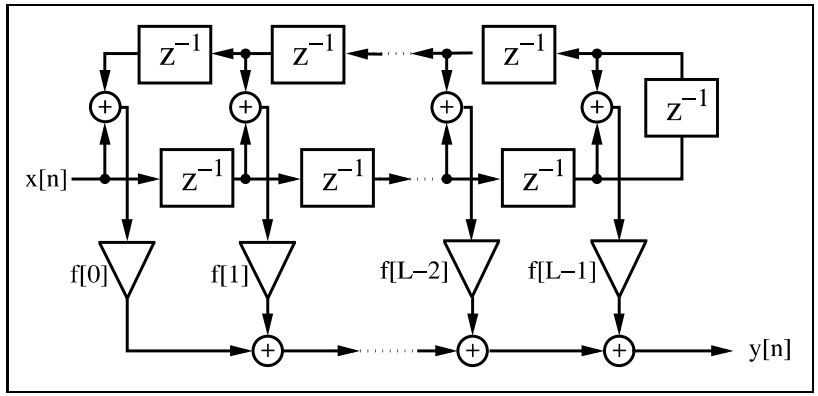


Fig. 3.5. Linear-phase filter with reduced number of multipliers.

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. 3.1. Consider the linear-phase FIR shown in Fig. 3.5 (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. 3.1 (L versus $L/2$) while the number of adders remains constant at $L - 1$.

3.3 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 Chap. 4.

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 $[0 \ldots \omega_\mathrm{p}]$, the transition band $[\omega_\mathrm{p} \ldots \omega_\mathrm{s}]$, and the stopband $[\omega_\mathrm{s} \ldots \pi]$ specification, where the sampling frequency is assumed to be 2π. 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:

$$f[n] = \mathrm{IDFT}(F[k]) = \sum_k F[k] \mathrm{e}^{\mathrm{j}2\pi kn/L}. \tag{3.12}$$

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. 3.6a, with the passband ripple shown in Fig. 3.6b. 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 $\pm 7\%$ over a wide range of filter orders. To illustrate this, consider the example filter with length 128, shown in Fig. 3.6c, with the passband ripple shown in Fig. 3.6d. 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. 3.7(upper). The deleterious effect on the transition band can also be seen. Other classic window functions are summarized in Table 3.2. 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:

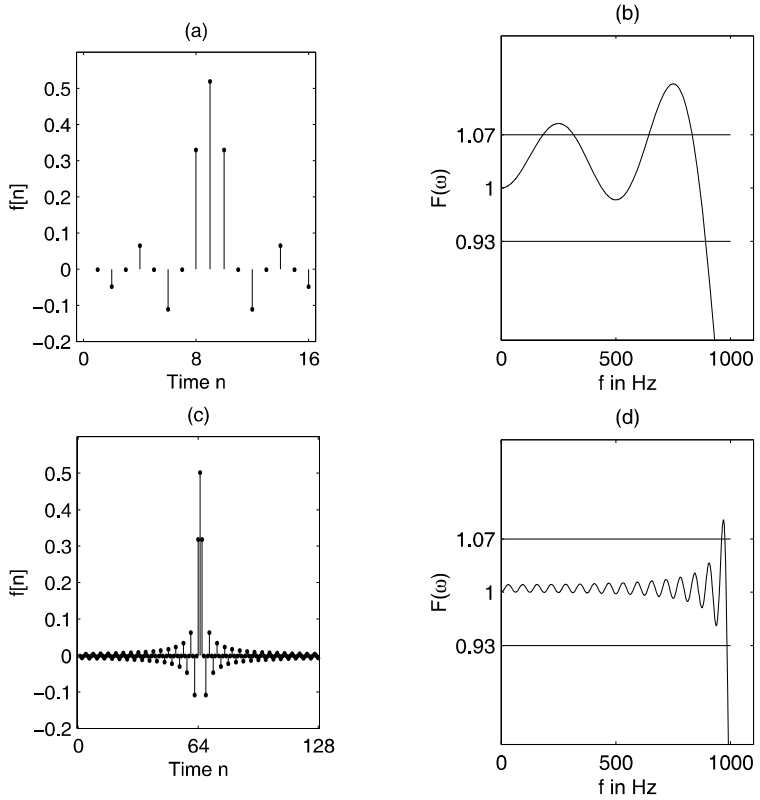


Fig. 3.6. Gibbs phenomenon. **(a)** Impulse response of FIR lowpass with $L = 16$. **(b)** Passband of transfer function $L = 16$. **(c)** Impulse response of FIR lowpass with $L = 128$. **(d)** Passband of transfer function $L = 128$.

- 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: $w[n] = \mathrm{I}_0\left(\beta\sqrt{1 - (n - L/2)^2/(L/2)^2}\right)$

Table 3.2 shows the most important parameters of these windows.

The 3-dB bandwidth shown in Table 3.2 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 0^{th} harmonic. De-

Table 3.2. Parameters of commonly used window functions.

Name	3-dB bandwidth	First zero	Maximum sidelobe	Sidelobe decrease per octave	Equivalent Kaiser β
Rectangular	$0.89/T$	$1/T$	−13 dB	−6 dB	0
Bartlett	$1.28/T$	$2/T$	−27 dB	−12 dB	1.33
Hanning	$1.44/T$	$2/T$	−32 dB	−18 dB	3.86
Hamming	$1.33/T$	$2/T$	−42 dB	−6 dB	4.86
Blackman	$1.79\ /T$	$3/T$	−74 dB	−6 dB	7.04
Kaiser	$1.44/T$	$2/T$	−38 dB	−18 dB	3

pending on the smoothness of the window, the third column in Table 3.2 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 0^{th} harmonic value. The fifth column describes the asymptotic decrease of the window per octave. Finally, the last column describes the value β for a Kaiser window that emulates the corresponding window properties. 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 β, which determines the ringing of the filter. This can be seen from the following equation credited to Kaiser.

$$\beta = \begin{cases} 0.1102(A-8.7) & A > 50, \\ 0.5842(A-21)^{0.4} + 0.07886(A-21) & 21 \le A \le 50, \\ 0 & A < 21, \end{cases} \tag{3.13}$$

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

$$L = \frac{A-8}{2.285(\omega_{\text{s}} - \omega_{\text{p}})} + 1. \tag{3.14}$$

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 ω_{p} and stopband ω_{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:

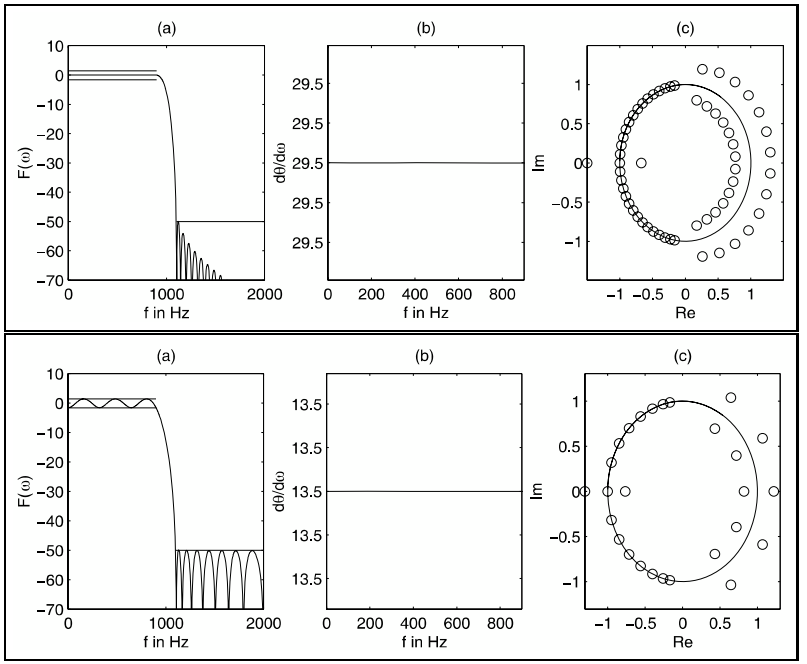


Fig. 3.7. (upper) Kaiser window design with $L = 59$. **(lower)** Parks-McClellan design with $L = 27$.
(a) Transfer function. **(b)** Group delay of passband. **(c)** Zero plot.

- Lowpass filter design (in MatLab[3] use `firpm(L,F,A,W)`), with tolerance scheme as shown in Fig. 3.8a
- 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 ω (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. 3.8a, and Fig. 3.8b shows a polynomial that fulfills this tolerance scheme. The length

[3] In previous MatLab versions the function `remez` had to be used.

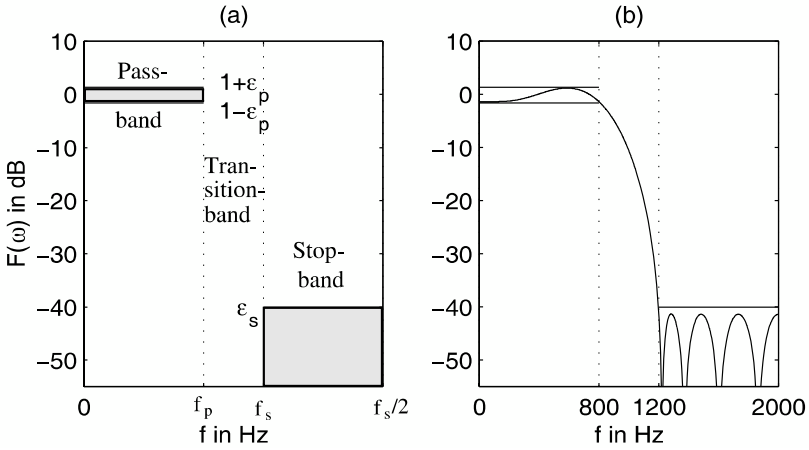


Fig. 3.8. Parameters for the filter design. **(a)** Tolerance scheme **(b)** Example function, which fulfills the scheme.

of the polynomial, and therefore the filter, can be estimated for a lowpass with

$$L = \frac{-10\log_{10}(\varepsilon_p\varepsilon_s) - 13}{2.324(\omega_s - \omega_p)} + 1, \tag{3.15}$$

where ε_p is the passband and ε_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, see [79, p. 478]. 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 stopband 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. 3.7(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.

3.4 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 3.1 (p. 167). 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.

3.4.1 Direct FIR Design

The direct FIR filter shown in Fig. 3.1 (p. 166) 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

$$f[k] = \{-1.0, 3.75, 3.75, -1.0\}. \tag{3.16}$$

These coefficients can be directly encoded into a 5-bit fractional number. For example, 3.75_{10} would have a 5-bit binary representation 011.11_2 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 3.4 discussed later, p. 183.

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:

$$G \leq \log_2 \left(\sum_{k=0}^{L-1} |f[k]| \right). \tag{3.17}$$

The total bit width is then the sum of the input bit width and the bit growth G. For the above filter for (3.16) 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 $\pm 128/9.5 = \pm 13$.

Example 3.2: Four-tap Direct FIR Filter

The VHDL design[4] for a filter with coefficients $\{-1, 3.75, 3.75, -1\}$ is shown in the following listing.

```
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].
```

[4] The equivalent Verilog code `fir_srg.v` for this example can be found in Appendix A on page 682. Synthesis results are shown in Appendix B on page 731.

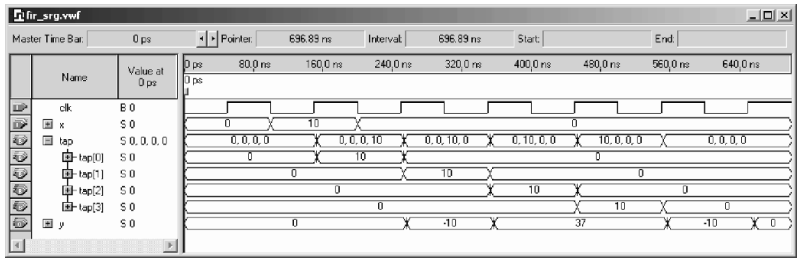


Fig. 3.9. VHDL simulation results of the FIR filter with impulse input 10.

```
  -- 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. 3.1 (p. 166). 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. 3.9. 3.2

There are three obvious actions that can improve this design:

1) Realize each filter coefficient with an optimized CSD code (see Chap. 2, Example 2.1, p. 58).
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. 3.5 (p. 172).

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.

Table 3.3. Improved FIR filter.

Symmetry	no	yes	no	no	yes	yes
CSD	no	no	yes	no	yes	yes
Tree	no	no	no	yes	no	yes
Speed/MHz	99.17	178.83	123.59	270.20	161.79	277.24
Size/LEs	114	99	65	139	57	81

Example 3.3: 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 3.3 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.

```
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
...
```

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

```
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;
...
```

3.3

Exercise 3.7 (p. 210) discusses the implementation of the filter in more detail.

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

$$f[n] = z^d f[n] z^{-d} \tag{3.18}$$

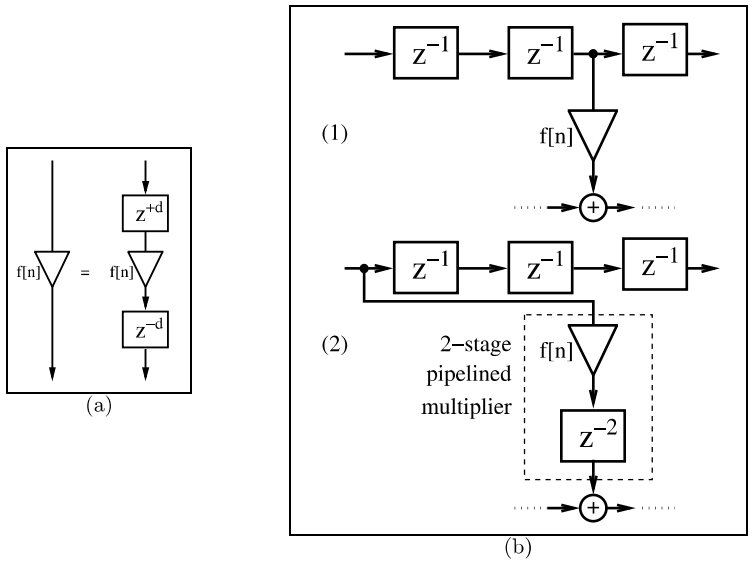


Fig. 3.10. Rephasing FIR filter. **(a)** Principle. **(b)** Rephasing a multiplier. **(1)** Without pipelining. **(2)** With two-stage pipelining.

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 d position previous register.

This principle is shown in Fig. 3.10a. Figure 3.10b shows an example of rephasing a pipelined multiplier that has two delays.

3.4.2 FIR Filter with Transposed Structure

A variation of the direct FIR filter is called the *transposed filter* and has been discussed in Sect. 3.2.1 (p. 167). 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 [31, 32, 33, 34]
- 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. The pipeline adder principle has been discussed in Chap. 2 and here we will focus on the RAG algorithm.

In Chap. 2 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 3×31 only requires two adders, see Fig. 2.3 (p. 61). 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 [33].

Algorithm 3.4: Reduced Adder Graph

1) Reduce all coefficients in the input set to positive odd fundamentals (OF).
2) Evaluate the single-coefficient adder cost of each coefficient using the MAG Table 2.3, p. 64.
3) Remove from the input set all power-of-two values and repeated fundamentals.
4) Create a graph set of all coefficients that can be built with one adder. Remove these coefficients from the input set.
5) Check if a pair of fundamentals in the graph set can be used to generate a coefficient in the input set by using a single adder.
6) Repeat step 5 until no further coefficients are added to the graph set.

This completes the optimal part of the algorithm. Next follows the heuristic part of the algorithm:

7) Add the smallest coefficient requiring two adders (if found) from the input set and its smallest NOF. The OF and one NOF (i.e., auxiliary coefficient) requires two adders using the fundamentals in the graph set.
8) Go to step 5 since the two new fundamentals from step 7 can be used to build other coefficients from the input set.
9) Add the smallest adder cost-3 or higher OF to the graph set and use the minimum NOF sum for this coefficient.
10) Go to step 5 until all coefficients are synthesized.

Steps 1–6 are straightforward, but steps 7–10 are potentially complex since the number of theoretical graphs increases exponentially. To simplify the process it is helpful to use the MAG coding data shown in Table 2.3 (p. 64). Let us briefly review some of the RAG steps that are not so obvious at first glance.

In step 1 all coefficients are reduced to positive odd fundamentals (i.e., power-of-two factors are removed from each coefficient), since this maximizes the number of partial sums, and the negative signs of the coefficients are implemented in the output adder TAPs of the filter. The two coefficient -7 and $28 = 4 \times 7$ would be merged. This works fine except for the unlikely case

when all coefficients are negative. Then a sign complement operation has to be added to the filter output.

In step 5 all sums of two extended fundamentals are considered. It may happen that a final division is also required, i.e., $g = (2^u f_1 \pm 2^v f_2)/2^w$. Note that multiplication or division by two can be implemented by left and right shift, respectively, i.e., they do not require hardware resources. For instance the coefficient set {7,105,53} MAG coding required one, two, and three adders, respectively. In RAG the set is synthesized as $7 = 8 - 1; 105 = 7 \times 15; 53 = (105 + 1)/2$, requiring only three adders but also a divide/right shift operation.

In step 7 an adder cost-2 coefficient is added and the algorithm selects the auxiliary coefficient, called the non-output fundamental (NOF), with the smallest values. This is motivated by the fact that an additional small NOF will generate more additional coefficients than a larger NOF. For instance, let us assume that the coefficient 45 needs to be added and we must decide which NOF value has to be used. The NOF LUTs lists possible NOFs as 3, 5, 9, or 15. It can now be argued that, if 3 is selected, more coefficients are generated than if any other NOF is used, since $3, 6, 12, 24, 48, \ldots$ can be generated without additional effort from NOF 3. If 15 is used, for instance, as the NOF the coefficients $15, 30, 45, \ldots$, are generated, which produces significantly fewer coefficients than NOF 3.

To illustrate the RAG algorithm, consider coding the coefficients defining the F6 half-band FIR filter of Goodman and Carey [80].

Example 3.5: Reduced Adder Graph for an F6 Half-band Filter

The half-band filter F6 has four nonzero coefficients, namely $f[0], f[1], f[3]$, and $f[5]$, which are 346, 208, -44, and 9. For a first cost estimation we convert the decimal values (index 10) into binary representations (index 2) and look up the cost for the coefficients in Table 2.3 (p. 64):

		$f[k]$		Cost
$f[0] = 346_{10}$	$=$	2×173	$= 101011010_2$	4
$f[1] = 208_{10}$	$=$	$2^4 \times 13$	$= 11010000_2$	2
$f[3] = -44_{10}$	$=$	$-2^2 \times 11$	$= -101100_2$	2
$f[5] = 9_{10}$	$=$	3^2	$= 1001_2$	1
		Total		9

For the direct CSD code realization, nine adders are required. The RAG algorithms proceeds as follows:

Step	To be realized	Already realized	Action
0)	{346, 208, −44, 9}	{ − }	Initialization
1a)	{346,208,44,9}	{ − }	No negative coefficients
1b)	{173,13,11,9}	{ − }	Remove 2^k factors
2)	{173,13,11,9}	{ − }	Look-up coefficients costs: {3,2,2,1}
3)	{173,13,11,9}	{ − }	Remove cost-0 coefficients from set
4)	{173,13,11}	{ 9 }	Realize cost-1 coefficients: $9 = 8 + 1$
5)	{173,13,11}	{9,11,13}	Build $11 = 9 + 2$ and $13 = 9 + 4$

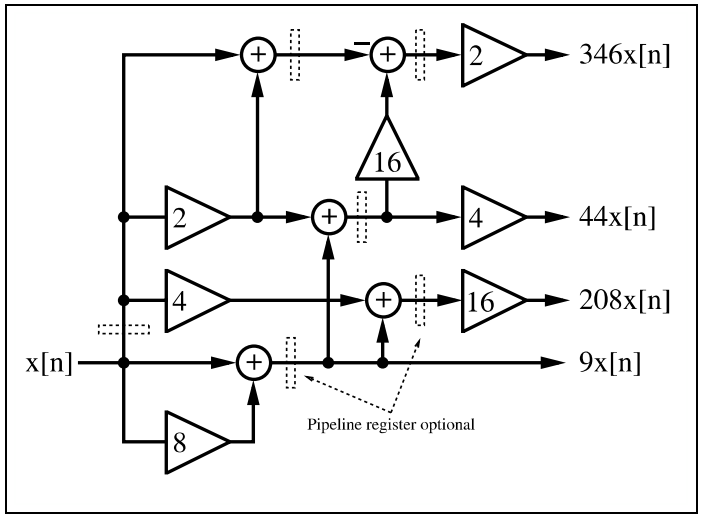


Fig. 3.11. Realization of F6 using RAG algorithm.

Apply the heuristic to the remaining coefficients, starting with the coefficient with the lowest cost and smallest value. It follows that:

Step	Realize	Already realized	Action Find representation
7)	$\{-\}$	{9,11,13,173}	Add NOF 3: $173 = 11 \times 16 - 3$

Figure 3.11 shows the resulting reduced adder graph. The number of adders is reduced from 9 to 5. The adder path delay is also reduced from 4 to 3. 3.5

A program `ragopt.exe` that implements the optimal part of the algorithms can be found in the book CD under `book3e/util`. Compared with the original algorithm only some minor improvements have been reported over the years [81].

- The MAG LUT table used has been extended to 14 bits (Gustafsson et al. [82] have actually extended the cost table to 19 bits but do not keep the fundamental table) and all 32 MAG adder cost-4 graph are now considered when computing the minimum NOF sum. Within 14 bits only two coefficients (i.e., 14 709, 15 573) are of cost 5 and, as long as these coefficients are not used, the computed minimum NOF sum list will be optimal in the RAG-95 sense.
- In step 7 all adder cost-2 graph are now considered. There are three such graphs, i.e., a single fundamental followed by an adder cost-2 factor, a sum of two fundamentals, and an adder cost-1 factor or a sum of three fundamentals.

- The last improvement is based on the adder cost-2 selection, which sometimes produced suboptimal results in the RAG-95 algorithm when multiple adder cost-2 coefficients have to be implemented. This can be explained as follows. While the selection of the smallest NOF is motivated by the statistical observation this may lead to suboptional results. For instance, for the coefficient set $\{13, 59, 479\}$ the minimum NOFs values used by RAG-95 are $\{3, 5, 7\}$ because $13 = 4 \times 3 + 1; 59 = 64 - 5; 479 = 59 \times 8 + 7$, resulting in a six-adder graph. If the NOF $\{15\}$ is chosen instead, then all coefficients $(13 = 15 - 2; 59 = 15 \times 4 - 1; 479 = 15 \times 32 - 1)$ benefit and RAG-05 only requires four adders, a 30% improvement. Therefore, instead of selecting the smallest NOF for the smallest adder cost-2 coefficient, a search for the best NOF is done over all adder cost-2 coefficients.

These modifications have been implemented in the RAG-05 algorithm, while the original RAG will be called RAG-95 based on the year of publishing the algorithms.

Although the RAG algorithm has been in use for quite some time, a large set of reliable benchmark data that can be verified and reproduced by anyone was not produced until recently [81]. In a recent paper by Wang and Roy [83], for instance, 60% of the comparison RAG data were declared "unknown." A benchmark should cover filters used in practical applications that are widely published or can easily be computed – a generation of random number filter coefficients that: (a) cannot be verified by a third party, and (b) are of no practical relevance (although used in many publications) are less useful. The problem with a RAG benchmark is that the heuristic part may give different results depending on the exact software implementation or the NOF table used. In addition, since some filters are rather long, a benchmark that lists the whole RAG is not practical in most cases. It is therefore suggested to use a benchmark based on the following equivalence transformation (remembering that the number of output fundamentals is equivalent to the number of adders required):

Theorem 3.6: RAG Equivalent Transformation

Let $\mathbb{S}_1$ be a coefficient set that can be synthesized by the RAG algorithm with a set of $\mathbb{F}_1$ output fundamentals and $\mathbb{N}_1$ non-output fundamentals, (i.e., internal auxiliary coefficients). A congruent RAG is synthesized if a coefficient set $\mathbb{S}_2$ is used that contains as fundamentals both output and non-output fundamentals from the first set $\mathbb{S}_2 = \mathbb{F}_1 \cup \mathbb{N}_1$.

Proof: Assume that $\mathbb{S}_2$ is synthesized via the RAG algorithm. Now all fundamentals can be synthesized with exactly one adder, since all fundamentals are synthesized in the optimal part of the algorithm. As a result a minimum number $C_2 = \#\mathbb{F}_1 + \#\mathbb{N}_1$ of adders for this fundamental set is used. If now set $\mathbb{S}_1$ is synthesized and generates the same fundamentals (output and non-output) as set $\mathbb{S}_2$, the resulting RAG also uses the minimum number of adders. Since both use the minimum number of adders they must be congruent. q.e.d.

A corollary of Theorem 3.6 is that graphs can now be classified as (guaranteed) optimal and heuristic graphs. An optimal graph has no more than one NOF, while a heuristic graph has more than one NOF. It is only required to provide a list of the NOFs to describe a unique OF graph. If this NOF is added to the coefficient set, all OFs are synthesized via the optimal part of the algorithm, which can easily be programmed. The program `ragopt.exe` that implements the optimal part of the algorithms is in fact available on the book CD under `book3e/util`. Some example benchmarks are given in Table 3.4. The first column shows the filter name, followed by the filter length L, and the bitwidth of the largest coefficient B. Then the reference adder data for CSD coding and CSE coding follows. The idea of the CSE coding is studied in Exercises 3.4 and 3.5 (p. 209) Note that the number of CSD adders given already takes advantage of coefficient symmetry, i.e., $f(k) = f(L-k)$. Common subexpression (CSE) required adder data are used from [83]. For the RAG algorithm the output fundamental (OF) and non-output fundamental (NOF) for RAG-2005 are listed. Note that the number of OFs is already much smaller than the filter length L. We then list in column 8 the adders required in the improved RAG-2005 version. Finally in the last column we list the NOF values that are required to synthesize the RAG filter via the optimal part of the RAG algorithms that is the basis for the program `ragopt.exe`[5] on the book CD under `book3e/util`. `ragopt.exe` uses a MAG LUT `mag14.dat` to determine the MAG costs, and produces two output files: `firXX.dat` that contains the filter data, and a file `ragopt.pro` that has the RAG-n coefficient equations. A `grep` command for lines that start with `Build` yields the equations necessary to construct the RAG-n graph.

It can be seen that the examples from Samueli [84] and Lim and Parker [85] all produce optimal RAG results, i.e., have a maximum of one NOF. Notice particularly for long filters the improvement of RAG compared to CSD and CSE adders. Filters F5-F9 are from the Goodman and Carey set of half-band filters (see Table 5.3, p. 274) and give better results using RAG-05 than RAG-95. The benchmark data from Samueli, and Lim andParker work very well for RAG since the filters are lowpass and therefore taper smoothly to zero at both sides, improving the likelihood of cost-1 output fundamentals. A more-challenging RAG design for DFT coefficients will be discussed in Chap. 6.

Pipelined RAG FIR Filter

Due to the logic delay in the RAG running through several adders, the resulting register performance of the design is not very high even for a small graph. To improve the register performance one can take advantage of the register embedded in each LE that would not otherwise be used. A single register

[5] You need to copy the program to your hard drive first; you can not start it from the CD directly.

Table 3.4. Required number of adders for the CSD, CSE, and RAG algorithms for lowpass filters. Prototype filters are from Goodman and Carey [80], Samueli [84], and Lim and Parker [85].

Filter name	L	B	CSD adder	CSE adder	#OF	#NOF	RAG-05 adder	NOF values
F5	11	8	6	-	3	0	3	-
F6	11	9	9	-	4	1	5	3
F7	11	9	7	-	3	1	4	23
F8	15	10	10	-	5	2	7	11, 17
F9	19	13	14	-	5	2	7	13, 1261
S1	25	9	11	6	6	0	6	-
S2	60	14	57	29	26	0	26	-
L1	121	17	145	57	51	1	52	49
L2	63	13	49	23	22	0	22	-
L3	36	11	16	5	5	0	5	-

placed at the output of an adder does therefore not require any additional logic resource. However, power-of-two coefficients that are implemented by shifting the register input word require an additional register not included in the zero-pipeline design. This design with one pipeline stage already enjoys a speed improvement of 50% compared with the non-pipelined design, see Table 3.5(Pipeline stages=1). For the fully pipelined design we need to have the same delay for each incoming path of the adders. For the F6 design one needs to build:

$$\begin{aligned} x9 &<= 8 \times x + x; \quad \text{has delay 1} \\ x11 &<= x9 + 2 \times x \times z^{-1}; \quad \text{has delay 2} \\ x13 &<= x9 + 4 \times x \times z^{-1}; \quad \text{has delay 2} \\ x3 &<= xz^{-1} + 2 \times x \times z^{-1}; \quad \text{has delay 2} \\ x173 &<= 16 \times x11 - x3; \quad \text{has delay 3} \end{aligned}$$

i.e., one extra pipeline register is used for input x, and a maximum delay of three pipeline stage is needed. The pipelined graph is shown in Fig. 3.11 with the dashed register active. Now the coefficients in the RAG are all fully pipelined. Now we need to take care of the different delays of the coefficients. We basically have two options: we can add to the output of *all* coefficients an additional delay, that we achieve the same delay for all coefficients (three in the case of the F6 filter) and then do not need to change the output tap delay line structure; alternative we can use pipeline retiming, i.e., the multiplier outputs need to be aligned in the tap delay line according to their pipeline stages. This is a similar approach to that used in the direct FIR (see Fig. 3.10, p. 182) by aligning the coefficient adder location according to the

Table 3.5. F6 pipeline options for the RAG algorithm.

Pipeline stages	LEs	Fmax (MHz)	Cost LEs/Fmax
0	225	165.95	1.36
1	234	223.61	1.05
max	252	353.86	0.71
Gain% 0/max	-11	114	92

delay, and is shown in Fig. 3.12. Note in order to build only two input adder, we had to use an additional register to delay the `x13` coefficient.

For this half-band filter design the pipeline retiming synthesis results shown in Table 3.5 reveal that the design now runs about twice as fast with a moderate (11%) increase in LEs when compared with the unpipelined design. Since the overall cost measured by LEs/Fmax is improved, fully pipelined designs should be preferred.

3.4.3 FIR Filters Using Distributed Arithmetic

A completely different FIR architecture is based on the distributed arithmetic (DA) concept introduced in Sect. 2.7.1 (p. 115). 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 $\{2, 3, 1\}$ found in Example 2.24 (p. 117).

Example 3.7: Distributed Arithmetic Filter as State Machine

A distributed arithmetic filter can be built in VHDL code[6] using the following state machine description:

[6] The equivalent Verilog code `dafsm.v` for this example can be found in Appendix A on page 683. Synthesis results are shown in Appendix B on page 731.

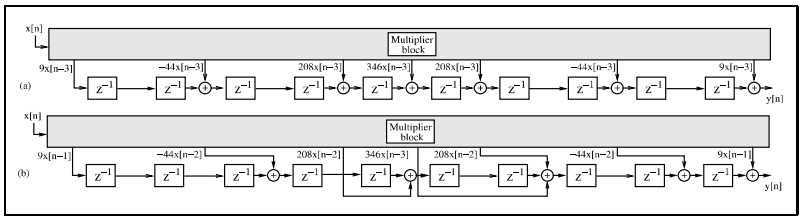


Fig. 3.12. F6 RAG filter with pipeline retiming.


```
LIBRARY ieee;               -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY dafsm IS                          ------> Interface
       PORT (clk, reset : IN STD_LOGIC;
             x0_in, x1_in, x2_in :
                          IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
             lut  : OUT INTEGER RANGE 0 TO 7;
             y    : OUT INTEGER RANGE 0 TO 63);
END dafsm;

ARCHITECTURE fpga OF dafsm IS

  COMPONENT case3  -- User-defined component
    PORT ( table_in   : IN   STD_LOGIC_VECTOR(2 DOWNTO 0);
           table_out  : OUT  INTEGER RANGE 0 TO 6);
  END COMPONENT;

  TYPE STATE_TYPE IS (s0, s1);
  SIGNAL state    : STATE_TYPE;
  SIGNAL x0, x1, x2, table_in
                          : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL table_out : INTEGER RANGE 0 TO 7;
BEGIN

  table_in(0) <= x0(0);
  table_in(1) <= x1(0);
  table_in(2) <= x2(0);

  PROCESS (reset, clk)       ------> DA in behavioral style
    VARIABLE p    : INTEGER RANGE 0 TO 63;-- temp. register
    VARIABLE count : INTEGER RANGE 0 TO 3; -- counts shifts
  BEGIN
    IF reset = '1' THEN               -- asynchronous reset
      state <= s0;
    ELSIF rising_edge(clk) THEN
    CASE state IS
      WHEN s0 =>        -- Initialization step
        state <= s1;
        count := 0;
        p := 0;
        x0 <= x0_in;
        x1 <= x1_in;
        x2 <= x2_in;
      WHEN s1 =>             -- Processing step
        IF count = 3 THEN    -- Is sum of product done ?
          y <= p;            -- Output of result to y and
          state <= s0;       -- start next sum of product
        ELSE
          p := p / 2 + table_out * 4;
          x0(0) <= x0(1);
          x0(1) <= x0(2);
```

```
            x1(0) <= x1(1);
            x1(1) <= x1(2);
            x2(0) <= x2(1);
            x2(1) <= x2(2);
            count := count + 1;
            state <= s1;
          END IF;
      END CASE;
      END IF;
    END PROCESS;

    LC_Table0: case3
      PORT MAP(table_in => table_in, table_out => table_out);
    lut <= table_out; -- Extra test signal

  END fpga;
```

The LE table[7] defined as `CASE` components was generated with the utility program `dagen3e.exe`. The output is show below.

```
  LIBRARY ieee;
  USE ieee.std_logic_1164.ALL;
  USE ieee.std_logic_arith.ALL;

  ENTITY case3 IS
         PORT ( table_in  : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
                table_out : OUT INTEGER RANGE 0 TO 6);
  END case3;

  ARCHITECTURE LEs OF case3 IS
  BEGIN

  -- This is the DA CASE table for
  -- the 3 coefficients: 2, 3, 1
  -- automatically generated with dagen.exe -- DO NOT EDIT!

    PROCESS (table_in)
    BEGIN
      CASE table_in IS
        WHEN  "000" =>    table_out <=  0;
        WHEN  "001" =>    table_out <=  2;
        WHEN  "010" =>    table_out <=  3;
        WHEN  "011" =>    table_out <=  5;
        WHEN  "100" =>    table_out <=  1;
        WHEN  "101" =>    table_out <=  3;
        WHEN  "110" =>    table_out <=  4;
        WHEN  "111" =>    table_out <=  6;
        WHEN   OTHERS  =>    table_out <=  0;
      END CASE;
    END PROCESS;
  END LEs;
```

[7] The equivalent Verilog code `case3.v` for this example can be found in Appendix A on page 684. Synthesis results are shown in Appendix B on page 731.

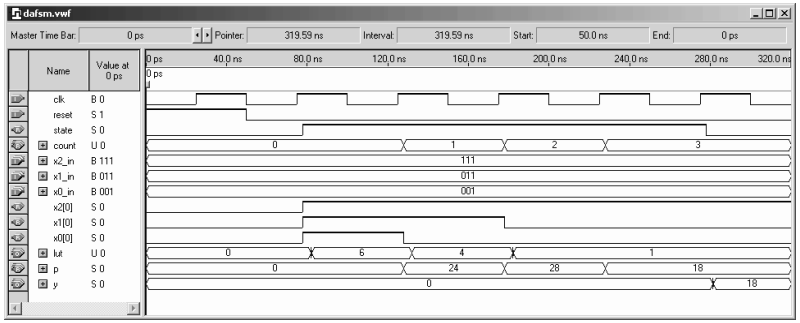


Fig. 3.13. Simulation of the 3-tap FIR filter with input $\{1, 3, 7\}$.

As suggested in Chap. 2, a shift/accumulator is used, which shifts only one position to the right for each step, instead of shifting k positions to the left. The simulation results, shown in Fig. 3.13, report the correct result ($y = 18$) for an input sequence $\{1, 3, 7\}$. 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 $y = 18 = 6 + 4 \times 2 + 1 \times 4$. The design uses 32 LEs, no embedded multiplier, no M4K block, and has a 420.17 MHz `Registered Performance`. 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 (as discussed in Chap. 1) can be configured as $2^9 \times 9, 2^{10} \times 4, 2^{11} \times 2$ or $2^{12} \times 1$ tables. These design paths are discussed in more detail in the following.

Distributed Arithmetic Using Logic Cells

The DA implementation of an FIR filter is particularly attractive for low-order 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, as shown in Fig. 2.37 (p. 122). 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 105 M4Ks. Also, M4Ks can be used to efficiently implement RAMs and FIFOs

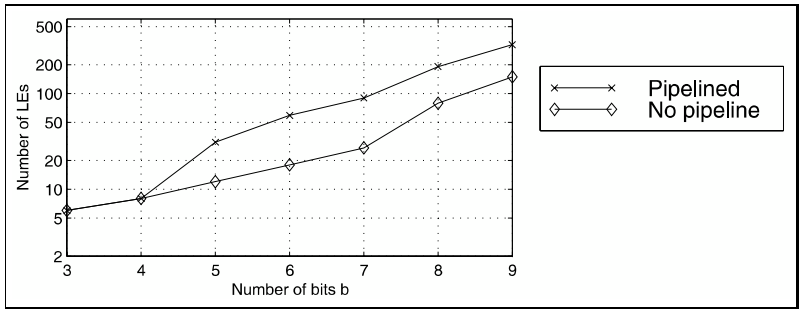


Fig. 3.14. Size comparison of synthesis results for different coding using the `CASE` statement with b input and outputs.

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 $2^9 \times 9$ table implemented with one VHDL `CASE` statement only, for example, required over 100 LEs. Figure 3.14 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`.

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[8] compared to the minimization of the one large LUT. The following example illustrates the structure of a 5-input table.

Example 3.8: 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[9] in the following listing:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY case5p IS
       PORT ( clk       : IN  STD_LOGIC;
              table_in  : IN  STD_LOGIC_VECTOR(4 DOWNTO 0);
```

[8] A 16:1 multiplexer and is reported with 11 LEs while we need 15 LEs or 2:1 MUX in a tree structure, see Cyclone II Device Handbook p. 5-15 [21].

[9] The equivalent Verilog code `case5p.v` for this example can be found in Appendix A on page 685. Synthesis results are shown in Appendix B on page 731.


```
              table_out : OUT INTEGER RANGE 0 TO 25);
END case5p;

ARCHITECTURE LEs OF case5p IS

  SIGNAL lsbs : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL msbs0 : STD_LOGIC_VECTOR(1 DOWNTO 0);
  SIGNAL table0out00, table0out01 : INTEGER RANGE 0 TO 25;

BEGIN

-- These are the distributed arithmetic CASE tables for
-- the 5 coefficients: 1, 3, 5, 7, 9
-- automatically generated with dagen.exe -- DO NOT EDIT!

  PROCESS
  BEGIN
    WAIT UNTIL clk = '1';
    lsbs(0) <= table_in(0);
    lsbs(1) <= table_in(1);
    lsbs(2) <= table_in(2);
    lsbs(3) <= table_in(3);
    msbs0(0) <= table_in(4);
    msbs0(1) <= msbs0(0);
  END PROCESS;

  PROCESS     -- This is the final DA MPX stage.
  BEGIN       -- Automatically generated with dagen.exe
    WAIT UNTIL clk = '1';
    CASE msbs0(1) IS
      WHEN  '0' =>    table_out <=  table0out00;
      WHEN  '1' =>    table_out <=  table0out01;
      WHEN   OTHERS  =>    table_out <=  0;
    END CASE;
  END PROCESS;

  PROCESS     -- This is the DA CASE table 00 out of 1.
  BEGIN       -- Automatically generated with dagen.exe
    WAIT UNTIL clk = '1';
    CASE lsbs IS
      WHEN  "0000" =>    table0out00 <=  0;
      WHEN  "0001" =>    table0out00 <=  1;
      WHEN  "0010" =>    table0out00 <=  3;
      WHEN  "0011" =>    table0out00 <=  4;
      WHEN  "0100" =>    table0out00 <=  5;
      WHEN  "0101" =>    table0out00 <=  6;
      WHEN  "0110" =>    table0out00 <=  8;
      WHEN  "0111" =>    table0out00 <=  9;
      WHEN  "1000" =>    table0out00 <=  7;
      WHEN  "1001" =>    table0out00 <=  8;
      WHEN  "1010" =>    table0out00 <=  10;
      WHEN  "1011" =>    table0out00 <=  11;
      WHEN  "1100" =>    table0out00 <=  12;
```

```
      WHEN  "1101" =>    table0out00 <=  13;
      WHEN  "1110" =>    table0out00 <=  15;
      WHEN  "1111" =>    table0out00 <=  16;
      WHEN   OTHERS  =>    table0out00 <=  0;
    END CASE;
  END PROCESS;

  PROCESS     -- This is the DA CASE table 01 out of 1.
  BEGIN       -- Automatically generated with dagen.exe
    WAIT UNTIL clk = '1';
    CASE lsbs IS
      WHEN  "0000" =>    table0out01 <=  9;
      WHEN  "0001" =>    table0out01 <=  10;
      WHEN  "0010" =>    table0out01 <=  12;
      WHEN  "0011" =>    table0out01 <=  13;
      WHEN  "0100" =>    table0out01 <=  14;
      WHEN  "0101" =>    table0out01 <=  15;
      WHEN  "0110" =>    table0out01 <=  17;
      WHEN  "0111" =>    table0out01 <=  18;
      WHEN  "1000" =>    table0out01 <=  16;
      WHEN  "1001" =>    table0out01 <=  17;
      WHEN  "1010" =>    table0out01 <=  19;
      WHEN  "1011" =>    table0out01 <=  20;
      WHEN  "1100" =>    table0out01 <=  21;
      WHEN  "1101" =>    table0out01 <=  22;
      WHEN  "1110" =>    table0out01 <=  24;
      WHEN  "1111" =>    table0out01 <=  25;
      WHEN   OTHERS  =>    table0out01 <=  0;
    END CASE;
  END PROCESS;
END LEs;
```

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. 3.8

Referring to Fig. 3.14, it can be seen that the structured VHDL code improves on the number of required LEs. Figure 3.15 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 464 MHz 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 464 MHz the design may now be too large for some applications. We may also consider the partitioning technique

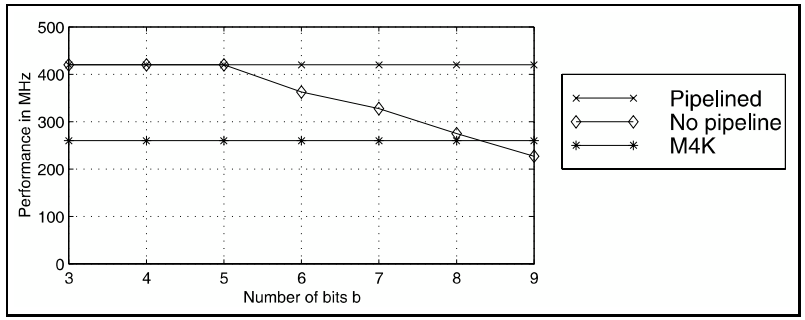


Fig. 3.15. Speed comparison for different coding styles using the `CASE` statement.

(Exercise 3.6, p. 210), shown in Fig. 2.36 (p. 121), or implementation with an M4K, discussed next.

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 260 MHz, and an LE table implementation may be faster. The following example shows the DA implementation using a component instantiation of the M4K.

Example 3.9: 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[10] for the DA state machine design is shown below:

```
LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;                -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;  -- Contains conversion
                                  -- VECTOR -> INTEGER
ENTITY darom IS                     ------> Interface
  PORT (clk, reset  : IN STD_LOGIC;
        x_in0, x_in1, x_in2
```

[10] The equivalent Verilog code `darom.v` for this example can be found in Appendix A on page 687. Synthesis results are shown in Appendix B on page 731.

```
                        : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
       lut  : OUT INTEGER RANGE 0 TO 7;
       y    : OUT INTEGER RANGE 0 TO 63);
END darom;

ARCHITECTURE fpga OF darom IS
  TYPE STATE_TYPE IS (s0, s1);
  SIGNAL state                     : STATE_TYPE;
  SIGNAL x0, x1, x2, table_in, mem
                          : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL table_out                 : INTEGER RANGE 0 TO 7;
BEGIN

  table_in(0) <= x0(0);
  table_in(1) <= x1(0);
  table_in(2) <= x2(0);

  PROCESS (reset, clk)       ------> DA in behavioral style
    VARIABLE  p   : INTEGER RANGE 0 TO 63; --Temp. register
    VARIABLE count : INTEGER RANGE 0 TO 3;
  BEGIN                                -- Counts the shifts
    IF reset = '1' THEN               -- Asynchronous reset
      state <= s0;
    ELSIF rising_edge(clk) THEN
    CASE state IS
      WHEN s0 =>           -- Initialization step
        state <= s1;
        count := 0;
        p := 0;
        x0 <= x_in0;
        x1 <= x_in1;
        x2 <= x_in2;
      WHEN s1 =>            -- Processing step
        IF count = 3 THEN   -- Is sum of product done ?
          y <= p / 2 + table_out * 4; -- Output of result
          state <= s0;              -- to y andstart next
        ELSE                        -- sum of product
          p := p / 2 + table_out * 4;
          x0(0) <= x0(1);
          x0(1) <= x0(2);
          x1(0) <= x1(1);
          x1(1) <= x1(2);
          x2(0) <= x2(1);
          x2(1) <= x2(2);
          count := count + 1;
          state <= s1;
        END IF;
    END CASE;
    END IF;
  END PROCESS;

  rom_1: lpm_rom
    GENERIC MAP ( LPM_WIDTH => 3,
```



```
                  LPM_WIDTHAD => 3,
                  LPM_OUTDATA => "REGISTERED",
                  LPM_ADDRESS_CONTROL => "UNREGISTERED",
                  LPM_FILE => "darom3.mif")
      PORT MAP(outclock => clk,address => table_in,q => mem);

    table_out <= CONV_INTEGER(mem);
    lut <= table_out;

  END fpga;
```

Compared with Example 3.7 (p. 189), 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 include file `darom3.mif` was generated with the program `dagen3e.exe`. The file has the following contents:

```
--  This is the DA MIF table for the 3 coefficients: 2, 3, 1
--  automatically generated with dagen3e.exe
-- DO NOT EDIT!
WIDTH = 3; DEPTH = 8; ADDRESS_RADIX = uns; DATA_RADIX = uns;
CONTENT BEGIN
     0     :   0;
     1     :   2;
     2     :   3;
     3     :   5;
     4     :   1;
     5     :   3;
     6     :   4;
     7     :   6;
END;
```

The design runs at 218.29 MHz and uses 27 LEs, and one M4K memory block (more precisely, 24 bits of an M4K).
The simulation results, shown in Fig. 3.16, are very similar to the `dafsm` simulation shown in Fig. 3.13 (p, 3.13). 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 $y = 18 = 6 + 4 \times 2 + 1 \times 4$. 3.9

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

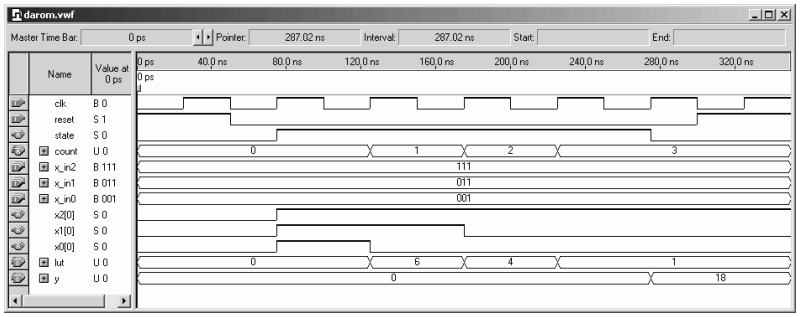


Fig. 3.16. Simulation of the 3-tap FIR M4K-based DA filter with input $\{1, 3, 7\}$.

Signed DA FIR Filter

A signed DA filter will require a signed accumulator. The following example shows the VHDL code for the previously studied three-coefficient example, 2.25 from Chap. 2 (p. 119).

Example 3.10: Signed DA FIR Filter

For the signed DA filter, an additional state is required. See the variable `count`[11] to process the sign bit.

```
LIBRARY ieee;                -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY dasign IS                        ------> Interface
       PORT (clk, reset : IN STD_LOGIC;
             x_in0, x_in1, x_in2
                        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
             lut  : out INTEGER RANGE -2 TO 4;
             y    : OUT INTEGER RANGE -64 TO 63);
END dasign;

ARCHITECTURE fpga OF dasign IS

  COMPONENT case3s      -- User-defined components
    PORT ( table_in : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
           table_out : OUT INTEGER RANGE -2 TO 4);
  END COMPONENT;

  TYPE STATE_TYPE IS (s0, s1);
  SIGNAL state      : STATE_TYPE;
  SIGNAL table_in   : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL x0, x1, x2 : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL table_out  : INTEGER RANGE -2 TO 4;
```

[11] The equivalent Verilog code `case3s.v` for this example can be found in Appendix A on page 688. Synthesis results are shown in Appendix B on page 731.


```
BEGIN

  table_in(0) <= x0(0);
  table_in(1) <= x1(0);
  table_in(2) <= x2(0);

  PROCESS (reset, clk)       ------> DA in behavioral style
    VARIABLE  p : INTEGER RANGE -64 TO 63:= 0; -- Temp. reg.
    VARIABLE count : INTEGER RANGE 0 TO 4; -- Counts the
  BEGIN                                          -- shifts
    IF reset = '1' THEN               -- asynchronous reset
      state <= s0;
    ELSIF rising_edge(clk) THEN
    CASE state IS
      WHEN s0 =>        -- Initialization step
        state <= s1;
        count := 0;
        p := 0;
        x0 <= x_in0;
        x1 <= x_in1;
        x2 <= x_in2;
      WHEN s1 =>          -- Processing step
        IF count = 4 THEN -- Is sum of product done?
          y <= p;       -- Output of result to y and
          state <= s0; -- start next sum of product
        ELSE
          IF count = 3 THEN           -- Subtract for last
          p := p / 2 - table_out * 8; -- accumulator step
          ELSE
          p := p / 2 + table_out * 8;  -- Accumulation for
          END IF;                       -- all other steps
            FOR k IN 0 TO 2 LOOP    -- Shift bits
              x0(k) <= x0(k+1);
              x1(k) <= x1(k+1);
              x2(k) <= x2(k+1);
            END LOOP;
          count := count + 1;
          state <= s1;
        END IF;
    END CASE;
    END IF;
  END PROCESS;

  LC_Table0: case3s
    PORT MAP(table_in => table_in, table_out => table_out);
  lut <= table_out; -- Extra test signal

END fpga;
```

The LE table (component `case3s.vhd`) was generated using the program `dagen3e.exe`. The VHDL code[12] is shown below:

[12] The equivalent Verilog code `case3s.v` for this example can be found in Appendix A on page 690. Synthesis results are shown in Appendix B on page 731.

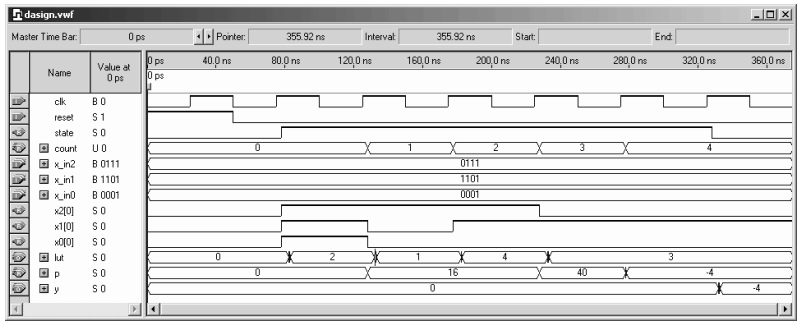


Fig. 3.17. Simulation of the 3-tap signed FIR filter with input $\{1, -3, 7\}$.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY case3s IS
  PORT ( table_in  : IN   STD_LOGIC_VECTOR(2 DOWNTO 0);
         table_out : OUT  INTEGER RANGE -2 TO 4);
END case3s;

ARCHITECTURE LEs OF case3s IS
BEGIN

-- This is the DA CASE table for
-- the 3 coefficients: -2, 3, 1
-- automatically generated with dagen.exe -- DO NOT EDIT!

  PROCESS (table_in)
  BEGIN
    CASE table_in IS
      WHEN  "000" =>    table_out <=  0;
      WHEN  "001" =>    table_out <=  -2;
      WHEN  "010" =>    table_out <=  3;
      WHEN  "011" =>    table_out <=  1;
      WHEN  "100" =>    table_out <=  1;
      WHEN  "101" =>    table_out <=  -1;
      WHEN  "110" =>    table_out <=  4;
      WHEN  "111" =>    table_out <=  2;
      WHEN   OTHERS  =>    table_out <=  0;
    END CASE;
  END PROCESS;
END LEs;
```

Figure 3.17 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 236.91 MHz `Registered Performance`. 3.10

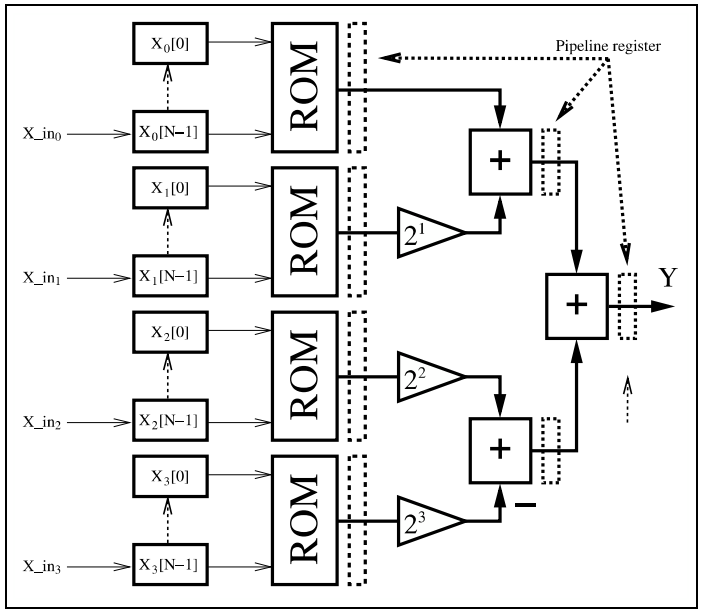


Fig. 3.18. Parallel implementation of a distributed arithmetic FIR filter.

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 3.11: Loop Unrolling for DA FIR Filter

In a typical FIR application, the input values are processed in word parallel form (i.e., see Fig. 3.18). The following VHDL code[3] illustrates the unrolled DA code, according to Fig. 3.18.

```
LIBRARY ieee;                  -- Using predefined packages
USE ieee.std_logic_1164.ALL;
```

[3] The equivalent Verilog code `dapara.v` for this example can be found in Appendix A on page 691. Synthesis results are shown in Appendix B on page 731.

```
USE ieee.std_logic_arith.ALL;

ENTITY dapara IS                          ------> Interface
       PORT (clk  : IN STD_LOGIC;
             x_in : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
             y    : OUT INTEGER RANGE -46 TO 44);
END dapara;

ARCHITECTURE fpga OF dapara IS
  TYPE ARRAY4x3 IS ARRAY (0 TO 3)
                           OF STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL x : ARRAY4x3;
  TYPE IARRAY IS ARRAY (0 TO 3) OF INTEGER RANGE -2 TO 4;
  SIGNAL h : IARRAY;
  SIGNAL s0 : INTEGER RANGE -6 TO 12;
  SIGNAL s1 : INTEGER RANGE -10 TO 8;
  SIGNAL t0, t1, t2, t3 : INTEGER RANGE -2 TO 4;
  COMPONENT case3s
    PORT ( table_in   : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
           table_out  : OUT INTEGER RANGE -2 TO 4);
  END COMPONENT;

BEGIN

  PROCESS                     ------> DA in behavioral style
  BEGIN
    WAIT UNTIL clk = '1';
    FOR l IN 0 TO 3 LOOP  -- For all four vectors
      FOR k IN 0 TO 1 LOOP  -- shift all bits
        x(l)(k) <= x(l)(k+1);
      END LOOP;
    END LOOP;
    FOR k IN 0 TO 3 LOOP  -- Load x_in in the
      x(k)(2) <= x_in(k); -- MSBs of the registers
    END LOOP;
    y <= h(0) + 2 * h(1) + 4 * h(2) - 8 * h(3);
-- Pipeline register and adder tree
--  t0 <= h(0); t1 <= h(1); t2 <= h(2); t3 <= h(3);
--  s0 <= t0 + 2 * t1; s1 <= t2 - 2 * t3;
--  y <= s0 + 4 * s1;
  END PROCESS;

  LC_Tables: FOR k IN 0 TO 3 GENERATE -- One table for each
  LC_Table: case3s                    -- bit in x_in
            PORT MAP(table_in => x(k), table_out => h(k));
  END GENERATE;

END fpga;
```

The design uses four tables of size $2^3 \times 4$ and all tables have the *same* content as the table in Example 3.10 (p. 199). Figure 3.19 shows the simulation for the input sequence $\{1, -3, 7\}$. Because the input is applied serially (and bit-parallel) the expected result $-4_{10} = 1111100_{2C}$ is computed at the 400-ns interval. 3.11

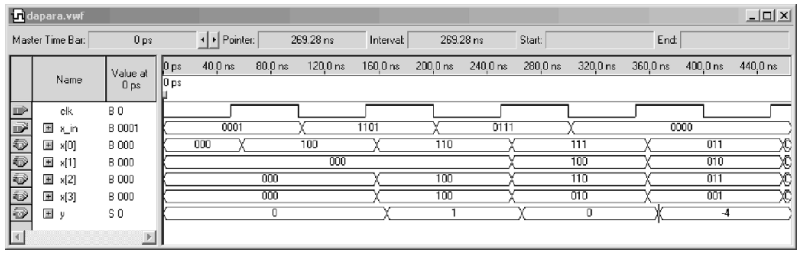


Fig. 3.19. Simulation results for the parallel distributed arithmetic FIR filter.

The previous design requires no embedded multiplier, 33 LEs, no M4K memory block, and runs at 214.96 MHz. 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 368.60 MHz, 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;
```

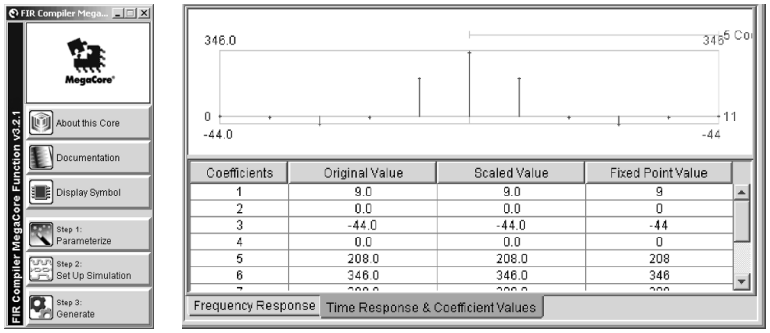
The size of the design increases to 47 LEs, because the registers of the LE that hold the `case` tables can no longer be used for the `x` input shift register. But the `Registered Performance` increases from 214.96 MHz to 420 MHz.

3.4.4 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. For an introduction to IP blocks see Sect. 1.4.4, p. 35.

FPGA vendors in general prefer distributed arithmetic (DA)-based FIR filter generators since these designs are characterized by:

- fully pipelined architecture
- short compile time



(a) (b)

Fig. 3.20. IP design of FIR **(a)** IP toolbench. **(b)** Coefficient specification.

- 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 [86].

Let us have a look at the FIR filter generation of an F6 filter from Goodman and Carey [80] that we had discussed before, see Example 3.5, p. 184. But this time we use the Altera FIR compiler [86] to build the filter. 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 (see Fig. 1.23, p. 39) 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. 3.20b 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. 3.21. 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 3.6. 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 3.22. We see that two additional control signals `rdy_to_ld` and `done` have been synthesized, although we did not ask for them. 3.12

The design from the Example 3.12 requires 426 LEs and runs at 362.84 MHz. Without the wrapper file the LE count (404 LEs) is still slightly higher than the estimation of 312 LEs. The overall cost metric measured as the quotient LEs/Fmax is 1.17 and is better than RAG without pipelining, since the DA is fully pipelined, as you can see from the large initial delay of the impulse response. For an appropriate comparison we should compare the DA-based design with the fully pipelined RAG design. The cost of the DA design is higher than the fully pipelined RAG design, see Table 3.5, p. 189. But the `Registered Performance` of the DA-based IP core is slightly higher than the fully pipelined RAG design.

3.4.5 Comparison of DA- and RAG-Based FIR Filters

In the last section we followed a detailed case study of the F6 half-band RAG- and DA-based FIR filter designs. The question now would be whether the results were just one single (atypical) example or if the results in terms of speed/size/cost are typical. In order to answer this question a set of larger filters has been designed using VHDL for fully pipelined RAG (see Exercises 3.13-3.29, p. 212) and should be compared with the synthesis data using the FIR core compiler from Altera that implements a fully parallel DA filter [86]. Table 3.7 shows the results for three half-band filters (F6, F8, and F9) from Goodman and Carey [80], two from Samueli [84], and two from Lim and Parker [85]. The first column shows the filter name, followed by the pipeline stages used. No-pipeline and fully pipelined data are reported, but no one-pipeline design data, as in Table 3.5 (p. 189). The third column shows the

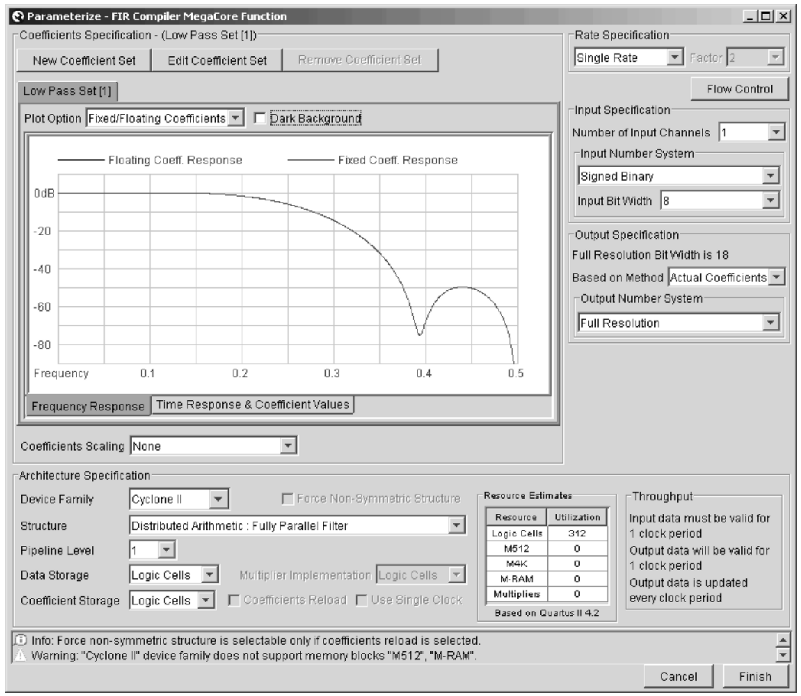


Fig. 3.21. IP parametrization of FIR core according to the F6 Example 3.5, p. 184.

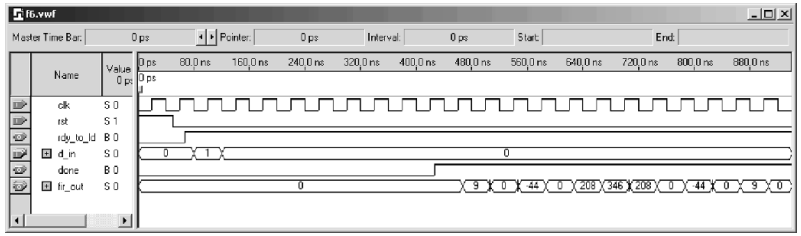


Fig. 3.22. FIR core timing simulation result.

filter length L. The next three columns show the synthesis data for the RAG-based filters, namely LEs, `Registered Performance` and the cost (area × delay) measured as the quotient LEs/Fmax. Columns 7-9 show the same three values for the DA-based designs generated with Altera's FIR compiler. For each filter two rows are used to show the data for zero/no and fully pipelined designs. Finally in the last rows the average value for zero- and fully pipelined designs are given and, at the very end, a comparison of the gain/loss of RAG

Table 3.6. IP files generation for FIR core.

File	Description
`f6_core.vhd`	A MegaCore function variation file, which defines a top-level VHDL description of the custom MegaCore function
`f6_core_inst.vhd`	VHDL sample instantiation file
`f6_core.cmp`	A VHDL component declaration for the MegaCore function variation
`f6_core.inc`	An AHDL include declaration file for the MegaCore function variation function
`f6_core_bb.v`	Verilog HDL black-box file for the MegaCore function variation
`f6_core.bsf`	Quartus II symbol file to be used in the Quartus II block diagram editor
`f6_core_st.v`	Generated FIR filter netlist
`f6_core` `_constraints.tcl`	This file contains the necessary constraints to achieve FIR filter size and speed
`f6_core_mlab.m`	This file provides a MATLAB simulation model for the customized FIR filter
`f6_core_tb.m`	This file provides a MATLAB testbench for the customized FIR filter
`f6_core.vec`	This file provides simulation test vectors to be used simulating the customized FIR filter with the Quartus II software
`f6_core.html`	The MegaCore function report file

zero- and fully pipelined and fully pipelined RAG compared with DA-based designs are given.

It can be seen from Table 3.7, that

- Fully pipelined RAG filters enjoy size reductions averaging 71% compared with DA-based designs.
- The fully pipelined RAG filter requires on average only 6% more LEs than the RAG design without pipelining.
- The `Register Performance` of the DA-based FIR filters is on average 8% higher than fully pipelined RAG designs.
- The overall cost, measured as LEs/Fmax, is on average 56% better for RAG-based compared with DA-based designs when a fully pipeline approach is used.

It can also be seen from Table 3.7 that, without pipelining (pipe=0), the DA-based approach gives better results. With a 6% increase in area, the cost for RAG pipelining is quite reasonable.

Table 3.7. Size, speed and cost comparison of DA and RAG algorithm.

Filter name	Pipe stages	L	RAG LEs	RAG Fmax (MHz)	RAG Cost $\frac{\text{LEs}}{\text{Fmax}}$	DA LEs	DA Fmax (MHz)	DA Cost $\frac{\text{LEs}}{\text{Fmax}}$
F6	0	11	225	165.95	1.36			
	max		234	353.86	0.71	396	332.34	1.19
F8	0	15	326	135.85	2.40			
	max		360	323.42	1.11	570	340.72	1.67
F9	0	19	461	97.26	4.74			
	max		534	304.04	1.76	717	326.16	2.20
S1	0	25	460	130.63	3.52			
	max		492	296.65	1.66	985	356.51	2.76
L3	0	36	651	205.3	3.17			
	max		671	310.37	2.16	1406	321.3	4.38
S2	0	60	1672	129.97	12.86			
	max		1745	252.91	6.90	2834	289.02	9.81
L2	0	63	1446	134.95	10.72			
	max		1531	265.53	5.77	2590	282.41	9.17
Mean	0		745	140.34	5.53			
Mean	max		793	296.60	2.86	1357	321.21	4.45
Gain%			RAG-0/RAG-max			RAG-max/DA		
			−6	111	93	71	−8	56

Exercises

Note: If you have no prior experience with the Quartus II software, refer to the case study found in Sect. 1.4.3, p. 29. If not otherwise noted use the EP2C35F672C6 from the Cyclone II family for the Quartus II synthesis evaluations.

3.1: A filter has the following specification: sampling frequency 2 kHz; passband 0–0.4 kHz, stopband 0.5–1 kHz; passband ripple, 3 dB, and stopband ripple, 48 dB. Use the MatLab software and the "Interactive Lowpass Filter Design" demo from the Signal Processing Toolbox for the filter design.
(a1) Design a direct filter with a Kaiser window.
(a2) Determine the filter length and the absolute ripple in the passband.
(b1) Design an equiripple filter (use the functions `remex` or `firpm`).
(b2) Determine the filter length and the absolute ripple in the passband.

3.2: **(a)** Compute the RAG for a length-11 half-band filter F5 that has the nonzero coefficients $f[0] = 256, f[\pm 1] = 150, f[\pm 3] = -25, f[\pm 5] = 3$.
(b) What is the minimum output bit width of the filter, if the input bit width is 8 bits?
(c1) Write and compile (with the Quartus II compiler) the HDL code for the filter.
(c2) Simulate the filter with impulse and step responses.
(d) Write the VHDL code for the filter in distributed arithmetic, using the state machine approach with the table realized as `LPM_ROM`.

3.3: **(a)** Compute the RAG for length-11 half-band filter F7 that has the nonzero coefficients $f[0] = 512, f[\pm 1] = 302, f[\pm 3] = -53, f[\pm 5] = 7$.
(b) What is the minimum output bit width of the filter, if the input bit width is 8 bits?
(c1) Write and compile (with the Quartus II compiler) the VHDL code for the filter.
(c2) Simulate the filter with impulse and step response.

3.4: Hartley [87] has introduced a concept to implement constant coefficient filters, by exploiting common subexpressions across coefficients. For instance, the filter

$$y[n] = \sum_{k=0}^{L-1} a[k]x[n-k], \tag{3.19}$$

with three coefficients $a[k] = \{480, -302, 31\}$. The CSD code of these three coefficients is given by

	512	256	128	64	32	16	8	4	2	1
480 :	1	0	0	0	−1	0	0	0	0	0
−302 :	0	−1	0	−1	0	1	0	0	1	0
31 :	0	0	0	0	1	0	0	0	0	−1

From the table we note that the pattern $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ can be found four times. If we therefore build the temporary variable $h[n] = 2x[n] - x[n-1]$, we can compute the filter output with

$$y[n] = 256h[n] - 16h[n] - 32h[n-1] + h[n-1]. \tag{3.20}$$

(a) Verify (3.20) by substituting $h[n] = 2x[n] - x[n-1]$.
(b) How many adders are required to yield the direct CSD implementation of (3.19) and the implementation with subexpression sharing?
(c1) Implement the filter with subexpression sharing with Quartus II for 8-bit inputs.
(c2) Simulate the impulse response of the filter.
(c3) Determine the `Registered Performance` and the used resources (LEs, multipliers, and M4Ks).

3.5: Use the subexpression method from Exercise 3.4 to implement a 4-tap filter with the coefficients $a[k] = \{-1406, -1109, -894, 2072\}$.
(a) Find the CSD code and the subexpression representation for the most frequent pattern.
(b) Substitute for the subexpression a 2 or −2, respectively. Apply the subexpression sharing one more time to the reduced set.
(c) Determine the temporary equations and check by substitution back into (3.19).
(d) How many adders are required to yield the direct CSD implementation of (3.19) and the implementation with subexpression sharing?
(e1) Implement the filter with subexpression sharing with Quartus II for 8-bit inputs.
(e2) Simulate the impulse response of the filter.
(e3) Determine the `Registered Performance` and the used resources (LEs, multipliers, and M4Ks).

3.6: **(a1)** Use the program `dagen3e.exe` to compile a DA table for the coefficients {20, 24, 21, 100, 13, 11, 19, 7} using multiple `CASE` statements.

Synthesize the design for maximum speed and determine the resources (LEs, multipliers, and M4Ks) and `Registered Performance`.
(a2) Simulate the design using power-of-two $2^k; 0 \leq k \leq 7$ input values.
(b) Use the partitioning technique to implement the same table using two sets, namely {20, 24, 21, 100} and {13, 11, 19, 7}, and an additional adder. Synthesize the design for maximum speed and determine the size and `Registered Performance`.
(b2) Simulate the design using power-of-two $2^k; 0 \leq k \leq 7$ input values.
(c) Compare the designs from (a) and (b).

3.7: Implement 8-bit input/output improved 4-tap {-1, 3.75, 3.75, -1} filter designs according to the listing in Table 3.3, p. 181. For each filter write the HDL code and determine the resources (LEs, multipliers, and M4Ks) and `Registered Performance`.
(a) Synthesize `fir_sym.vhd` as the filter using symmetry.
(b) Synthesize `fir_csd.vhd` as the filter using CSD coding.
(c) Synthesize `fir_tree.vhd` as the filter using an adder tree.
(d) Synthesize `fir_csd_sym.vhd` as the filter using CSD coding and symmetry.
(e) Synthesize `fir_csd_sym_tree.vhd` as the filter using all three improvements.

3.8: **(a)** Write a short MATLAB program that plots the
(a1) impulse response,
(a2) frequency response, and
(a2) the pole/zero plot for the half-band filter F3, see Table 5.3, p. 274.
Hint: Use the MATLAB functions: `filter, stem, freqz, zplane.`
(b) What is the bit growth of the F3 filter? What is the total required output bit width for an 8-bit input?
(c) Use the `csd3e.exe` program from the CD to determine the CSD code for the coefficients.
(c) Use the `ragopt.exe` program from the CD to determine the reduced adder graph (RAG) of the filter coefficients.

3.9: Repeat Exercise 3.8 for the CFIR filter of the GC4114 communication IC. Try the WWW to download a datasheet if possible. The 31 filter coefficients are: $-23, -3, 103, 137, -21, -230, -387, -235, 802, 1851, 81, -4372, -4774, 5134, 20\,605, 28\,216, 20\,605, 5134, -4774, -4372, 81, 1851, 802, -235, -387, -230, -21, 137, 103, -3, -23$.

3.10: Download the datasheet for the GC4114 from the WWW. Use the results from Exercise 3.9.
(a) Design the 31-tap symmetric CFIR compensation filter as CSD FIR filter in transposed form (see Fig. 3.3, p. 167) for 8-bit input and an asynchronous reset. Try to match the simulation shown in Fig. 3.23.
(b) For the device EP2C35F672C6 from the Cyclone II family determine the resources (LEs, multipliers, and M4Ks) and the `Registered Performance`.

3.11: Download the datasheet for the GC4114 from the WWW. Use the results from Exercise 3.9.
(a) Design the 31-tap symmetric CFIR compensation filter using distributed arithmetic. Use the `dagen3e.exe` program from the CD to generate the HDL code for the coefficients. Note you should use always groups of four coefficients each and add the results in an adder tree.
(b) Design the DA FIR filter in the full parallel form (see Fig. 3.18, p. 202) for 8-bit input and an asynchronous reset. Take advantage of the coefficient symmetry. Try to match the simulation shown in Fig. 3.24.

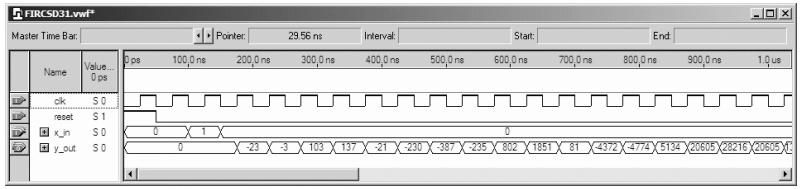


Fig. 3.23. Testbench for the CSD FIR filter in Exercise 3.10.

(c) For the device EP2C35F672C6 from the Cyclone II family determine the resources (LEs, multipliers, and M4Ks) and the `Registered Performance`.

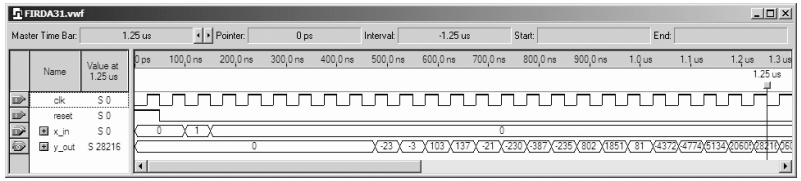


Fig. 3.24. Testbench for the DA-based FIR filter in Exercise 3.11.

3.12: Repeat Exercise 3.8 for the half-band filter F4, see Table 5.3, p. 274.

3.13: Repeat Exercise 3.8 for the half-band filter F5, see Table 5.3, p. 274.

3.14: Use the results from Exercise 3.13 and report the HDL code, resources (LEs, multipliers, and M4Ks) and `Registered Performance` for the HDL design of the F5 half-band HDL FIR filter as:
(a) an RAG filter without pipelining
(b) a fully pipelined RAG filter
(c) a DA fully pipelined filter using an FIR core generator

3.15: Repeat Exercise 3.8 for the half-band filter F6, see Table 5.3, p. 274.

3.16: Use the results from Exercise 3.15 and report the HDL code, resources (LEs, multipliers, and M4Ks) and `Registered Performance` for the HDL design of the F6 half-band HDL FIR filter as:
(a) an RAG filter without pipelining
(b) a fully pipelined RAG filter
(c) a DA fully pipelined filter using an FIR core generator

3.17: Repeat Exercise 3.8 for the half-band filter F7, see Table 5.3, p. 274.

3.18: Repeat Exercise 3.8 for the half-band filter F8, see Table 5.3, p. 274.

3.19: Use the results from Exercise 3.18 and report the HDL code, resources (LEs, multipliers, and M4Ks) and `Registered Performance` for the HDL design of the F8 half-band HDL FIR filter as:

(a) an RAG filter without pipelining
(b) a fully pipelined RAG filter
(c) a DA fully pipelined filter using an FIR core generator

3.20: FIR features design. In this problem we want to compare the influence of additional features like reset and enable for different device families. Use the results from Exercise 3.18 for the CSD code. For all following HDL F8 CSD designs with 8-bit input determine the resources (LEs, multipliers, and M2Ks/M4Ks) and `Registered Performance`. As the device use the EP2C35F672C6 from the Cyclone II family and the EPF10K70RC240-4 from the Flex 10K family.
(a) Design the F8 CSD FIR filter in direct form (see Fig. 3.1, p. 166).
(b) Design the F8 CSD FIR filter in transposed form (see Fig. 3.3, p. 167).
(c) Add a synchronous reset to the transposed FIR from (b).
(d) Add an asynchronous reset to the transposed FIR from (b).
(e) Add a synchronous reset and enable to the transposed FIR from (b).
(f) Add an asynchronous reset and enable to the transposed FIR from (b).
(g) Tabulate your resources (LEs, multipliers, and M2Ks/M4Ks) and `Registered Performance` results from (a)-(g). What conclusions can be drawn for Flex and Cyclone II devices from the measurements?

3.21: Repeat Exercise 3.8 for the half-band filter F9, see Table 5.3, p. 274.

3.22: Use the results from Exercise 3.21 and report the HDL code, resources (LEs, multipliers, and M4Ks) and `Registered Performance` for the HDL design of the F9 half-band HDL FIR filter as:
(a) an RAG filter without pipelining
(b) a fully pipelined RAG filter
(c) a DA fully pipelined filter using an FIR core generator

3.23: Repeat Exercise 3.8 for the Samueli filter S1 [84]. The 25 filter coefficients are: $1, 3, 1, 8, 7, 10, 20, 1, 40, 34, 56, 184, 246, 184, 56, 34, 40, 1, 20, 10, 7, 8, 1, 3, 1$.

3.24: Use the results from Exercise 3.23 and report the HDL code, resources (LEs, multipliers, and M4Ks) and `Registered Performance` for the HDL design of the Samueli filter S1 FIR filter as:
(a) an RAG filter without pipelining
(b) a fully pipelined RAG filter
(c) a DA fully pipelined filter using an FIR core generator

3.25: Repeat Exercise 3.8 for the Samueli filter S2 [84]. The 60 filter coefficients are: $31, 28, 29, 22, 8, -17, -59, -116, -188, -268, -352, -432, -500, -532, -529, -464, -336, -129, 158, 526, 964, 1472, 2008, 2576, 3136, 3648, 4110, 4478, 4737, 4868, 4868, 4737, 4478, 4110, 3648, 3136, 2576, 2008, 1472, 964, 526, 158, -129, -336, -464, -529, -532, -500, -432, -352, -268, -188, -116, -59, -17, 8, 22, 29, 28, 31$.

3.26: Use the results from Exercise 3.25 and report the HDL code, resources (LEs, multipliers, and M2Ks/M4Ks) and `Registered Performance` for the HDL design of the Samueli filter S2 FIR filter as:
(a) an RAG filter without pipelining.
(b) a fully pipelined RAG filter.
(c) a DA fully pipelined filter using an FIR core generator.

3.27: Repeat Exercise 3.8 for the Lim and Parker L2 filter [85]. The 63 filter coefficients are: $3, 6, 8, 7, 1, -9, -19, -24, -20, -5, 15, 31, 33, 16, -15, -46, -59, -42, 4, 61, 99, 92, 29, -71, -164, -195, -119, 74, 351, 642, 862, 944, 862, 642, 351, 74, -119, -195, -164, -71, 29, 92, 99, 61, 4, -42, -59, -46, -15, 16, 33, 31, 15, -5, -20, -24, -19, -9, 1, 7, 8, 6, 3$.

3.28: Use the results from Exercise 3.27 and report the HDL code, resources (LEs, multipliers, and M4Ks) and `Registered Performance` for the HDL design of the Lim and Parker L2 filter FIR filter as:
(a) an RAG filter without pipelining
(b) a fully pipelined RAG filter
(c) a DA fully pipelined filter using an FIR core generator

3.29: Repeat Exercise 3.8 for the Lim and Parker L3 filter [85]. The 36 filter coefficients are: $10, 1, -8, -14, -14, -3, 10, 20, 24, 9, -18, -40, -48, -20, 36, 120, 192, 240, 240, 192, 120, 36, -20, -48, -40, -18, 9, 24, 20, 10, -3, -14, -14, -8, 1, 10$.

3.30: Use the results from Exercise 3.29 and report the HDL code, resources (LEs, multipliers, and M4Ks) and `Registered Performance` for the HDL design of the Lim and Parker filter L3 FIR filter as:
(a) an RAG filter without pipelining
(b) a fully pipelined RAG filter
(c) a DA fully pipelined filter using an FIR core generator