ФГБОУ ВПО «Воронежский государственный
технический университет»

А.В. Строгонов

ЦИФРОВАЯ ОБРАБОТКА СИГНАЛОВ В БАЗИСЕ ПРОГРАММИРУЕМЫХ ЛОГИЧЕСКИХ ИНТЕГРАЛЬНЫХ СХЕМ

Утверждено Редакционно-издательским советом
университета в качестве учебного пособия

Воронеж 2015

УДК 621.3.049.77

Строгонов А.В. Цифровая обработка сигналов в базисе программируемых логических интегральных схем: учеб. пособие [Электронный ресурс]. – Электрон. текстовые и граф. данные (34,3 Мб) / А.В. Строгонов. – Воронеж: ФГБОУ ВПО «Воронежский государственный технический университет», 2015. – 1 электрон. опт. диск (DVD-ROM) : цв. – Систем. требования : ПК 500 и выше ; 256 Мб ОЗУ ; Windows XP ; SVGA с разрешением 1024x768 ; Adobe Acrobat; DVD-ROM дисковод ; мышь. – Загл. с экрана.

В учебном пособии рассматривается проектирование устройств цифровой обработки сигналов для реализации в базисе ПЛИС. Даются практические примеры проектирования цифровых фильтров с использованием высокоуровневого языка описания аппаратурных средств VHDL и мегафункций в САПР ПЛИС Altera Quartus II и Xilinx ISE Design Suite.

Издание соответствует требованиям Федерального государственного образовательного стандарта высшего профессионального образования по направлению 210100.68 «Электроника и наноэлектроника» (программа магистерской подготовки «Приборы и устройства в микро- и наноэлектронике»), дисциплинам «Цифровая обработка сигналов», «Архитектуры микропроцессорных вычислительных систем», «САПР БИС программируемой логики», «САПР системного уровня проектирования БИС».

Табл. 16. Ил. 199. Библиогр.: 25 назв.

Научный редактор д-р физ.-мат. наук, проф. С.И. Рембеза

Рецензенты: кафедра физики полупроводников и микроэлектроники Воронежского государственного университета (зав. кафедрой д-р физ.-мат. наук, проф. Е.Н. Бормонтов);
канд. техн. наук, доц. В.Б. Стешенко

ВВЕДЕНИЕ

ПЛИС – цифровые БИС высокой степени интеграции, имеющие программируемую пользователем внутреннюю структуру и предназначенные для реализации сложных цифровых устройств. Использование ПЛИС и САПР позволяет в сжатые сроки создавать конкурентоспособные устройства и системы, удовлетворяющие жестким требованиям по производительности, энергопотреблению, надежности, массо-габаритным параметрам, стоимости. Обработка сигналов может осуществляться с помощью различных технических средств. В последнее десятилетие лидирующее положение занимает цифровая обработка сигналов (ЦОС), которая по сравнению с аналоговой имеет следующие преимущества: малую чувствительность к параметрам окружающей среды, простоту перепрограммирования и переносимость алгоритмов. Одной из распространённых операций ЦОС является фильтрация. Вид импульсной характеристики цифрового фильтра (ЦФ) определяет их деление на ЦФ с конечной импульсной характеристикой (КИХ-фильтры) и ЦФ с бесконечной импульсной характеристикой (БИХ-фильтры).

Широкое применение цифровых КИХ-фильтров вызвано тем, что свойства их хорошо исследованы. Использование особенностей архитектуры ПЛИС позволяет проектировать компактные и быстрые КИХ-фильтры с использованием так называемой распределённой арифметики.

В первой главе рассматриваются основы двоичной арифметики и представление чисел со знаком, основное внимание уделено проектированию умножителей чисел со знаком, представленных в дополнительном коде. Приводятся сведения по программным умножителям в базисе ПЛИС. Дается практический пример по использованию учебного лабораторного стенда LESO2.1 (Лаборатории электронных средств обучения, ЛЭСО ГОУ ВПО «СибГУТИ») для отладки

проекта умножителя целых положительных чисел представленных в прямом коде размерностью 4х4 методом правого сдвига и сложения в базисе ПЛИС серии Cyclone.

Во второй главе рассматривается моделирование КИХ-фильтра в системе Matlab/Simulink (пакет Signal Processing, среда FDATool). Демонстрируются различные варианты реализации параллельных КИХ-фильтров с использованием перемножителей на мегафункциях САПР Quartus II компании Altera. Обсуждаются вопросы проектирования с учетом эффектов квантования возникающих при переходе от имитационной модели КИХ-фильтра разработанной в системе Matlab/Simulink к функциональной реализованной в САПР ПЛИС Quartus II компании Altera.

В главе 3 затрагиваются вопросы проектирования высокопроизводительных КИХ-фильтров на последовательной и параллельной распределенных арифметиках учитывающих архитектурные особенности ПЛИС.

В главе 4 рассматривается проектирование систолических КИХ-фильтров в базисе ПЛИС с использованием системы цифрового моделирования ModelSim-Altera. А также показано использование программы синтеза логики Synplicity совместно с САПР ISE 14.2 для проектирования фильтров в базисе ПЛИС фирмы Xilinx.

Уделено внимание методологии объектно-ориентированного проектирования с использованием System Generator IDS 14.4 — инструмента для разработки и отладки высокопроизводительных систем цифровой обработки сигналов в базисе ПЛИС фирмы Xilinx в системе визуально-имитационного моделирования Matlab/Simulink (версия 8.0.0.783 (R2012b)). Программный пакет обеспечивает высокоуровневое представление проекта, абстрагированное от конкретной аппаратной платформы, которое автоматически компилируется в ПЛИС Xilinx.

1. ПРОЕКТИРОВАНИЕ УМНОЖИТЕЛЕЙ В БАЗИСЕ ПЛИС

1.1. Двоичная арифметика

Положительные двоичные числа можно представить только одним способом, а отрицательные двоичные числа – тремя способами. В табл. 1.1 приведены в качестве примера десятичные числа со знаком и их эквивалентные представления в прямом, обратном и дополнительном двоичном коде.

Прямой код. Знак – старший значащий разряд (СЗР) указывает знак (0 – положительный, 1 - отрицательный). Остальные разряды отражают величину, представляющую положительное число:

```
Знак
СЗР    МЗР
 0  110   1  = + 13
 1  110   1  = - 13
```

Это представление чисел удобно для умножения и деления, но при операциях сложения и вычитания нецелесообразно и поэтому используется редко.

В ЭВМ положительные числа представляются в прямом коде, а отрицательные – в виде дополнений, т.е. путем сдвига по числовой оси исходного числа на некоторую константу. Если z – положительное число, то – z представляется в виде K-z, где K таково, что разрядность положительна. Обратный код отличается от дополнительного только выбором значения K.

Дополнение до единицы (обратный код) – отрицательные числа получаются путем инверсии всех разрядов их положительных эквивалентов. Старший значащий разряд указывает знак (0 – положительный, 1 - отрицательный).

Таблица 1.1

Представление чисел в прямом, обратном и дополнительном четырехразрядном двоичном коде

ДЧ со знаком	Прямой код	Обратный код* (инверсия $\lvert X_{10}\rvert$ и 1 в знаковый разряд)	Дополнительный код** (инверсия $\lvert X_{10}\rvert$, плюс 1 к МЗР и 1 в знаковый разряд)
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
0	0000 1000	0000 1111	0000
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

* при суммировании чисел циклический перенос к МЗР;

** при суммировании чисел перенос игнорируется

Пусть X_{10} – десятичное число со знаком, которое необходимо представить в обратном коде. Необходимо найти n-разрядное представление числа X_{10}, включая знак и часть абсолютной величины, которая считается (n-1)-разрядной. Если $X_{10} \geq 0$, то обратный код содержит 0 в старшем,

знаковом разряде и обычное двоичное представление X_{10} в остальных n-1 разрядах. Таким образом, для положительных чисел обратный код совпадает с прямым. Если же $X_{10} \le 0$, то знаковый разряд содержит 1, а остальные разряды содержат двоичное представление числа:

$$2^{n-1} - 1 - \left|X_{10}\right|.$$

Дополнение до единицы формируется очень просто, однако обладает некоторыми недостатками, среди которых двойное представление нуля (все единицы или нули).

Рассмотрим положительное число +13. Выбрав шестиразрядное представление, включая знак (n = 6), получим обратный код, равный 001101. Под абсолютную величину числа отводим пять разрядов. Рассмотрим отрицательное число -13_{10}, считая представление шестиразрядным, включая знак. В пятиразрядном представлении $\left|-13_{10}\right| = 13_{10} = 01101_2$ и $2^5 - 1_{10} = 31_{10} = 11111_2$ тогда

$$(2^{6-1} - 1 - 13)_{10} = (11111 - 01101)_2 = 10010_2.$$

Добавив шестой, знаковый, разряд, получим шестиразрядный код для -13_{10}, равный 110010.

Дополнение до двух (дополнительный код). Его труднее сформировать, чем дополнение до единицы, но использованием данного кода удается упростить операции сложения и вычитания. Дополнение до двух образуется путем инверсии каждого разряда положительного числа и последующего добавления единицы к МЗР:

Знак	МЗР	
0	110 1	= + 13
1	001 1	= - 13

Если $X_{10} \ge 0$, то так же, как для прямого и обратного кодов, имеем 0 в знаковом разряде и обычное двоичное представление числа X_{10} в остальных n-1 разрядах. Если же

$X_{10} < 0$, то имеем 1 в знаковом разряде, а в остальных n-1 разрядах двоичный эквивалент числа $2^{n-1} - |X_{10}|$

Рассмотрим схему сумматора, основанного на поразрядном процессе. Обозначим два складываемых числа через $A = a_{n-1}a_{n-2}\ldots a_1a_0$ и $B = b_{n-1}b_{n-2}\ldots b_1b_0$. При сложении двоичных чисел значения цифр в каждом двоичном разряде должны быть сложены между собой с переносом из предыдущего разряда. Если результат при этом превышает 1, то возникает перенос в следующий разряд.

Рассмотрим число -13_{10}. Представим его в шестиразрядном дополнительном коде. Так как $|-13_{10}| = 13_{10} = 01101_2$ и $2^5_{10} = 32_{10} = 100000_2$, то получим в пятиразрядном представлении

$$2^{n-1} - |X_{10}| = (2^{6-1} - 13)_{10} = (100000 - 01101)_2 = 10011_2 .$$

Добавляя шестой знаковый разряд, получаем дополнительный код числа -13_{10}, равный 110011. Ноль в дополнительном коде имеет единственное представление.

Сложение положительных чисел происходит непосредственно, но перенос в разряд знака нужно предотвратить и рассматривать как переполнение. Когда складываются два отрицательных числа или отрицательное число с положительным, то работа сумматора зависит от способа представления отрицательного числа. При представлении последних в дополнительном коде сложение осуществляется просто, но необходим дополнительный знаковый разряд, любой перенос за пределы положения знакового разряда просто игнорируется.

```
+14  01110        +7  00111        -4  11100
- 7  11001       -14  10010       -3  11101
-----------      -----------      -----------
+7   00111       -7   11001       -7   11001
```

Если используется дополнение до единицы, то перенос из знакового разряда должен использоваться как входной перенос к МЗР.

+14	01110		+7	00111		-4	11011	
-7	11000		-14	10001		-3	11100	
	00110		-7	11000			10111	
+	1					+	1	
+7	00111					-7	11000	

Рассмотрим такое понятие как “расширение знака”. Рассмотрим десятичное число -3_{10} в дополнительном, а число 3_{10} - прямом кодах в трехразрядном представлении:

$$-3 \quad 101 \quad (-2^2 + 2^0)$$
$$3 \quad 011 \quad (2^1 + 2^0)$$

,

в четырехразрядном представлении:

$$1101 \quad (-2^3 + 2^2 + 2^0)$$
$$0011 \quad (2^1 + 2^0)$$

Таким образом, добавление единиц для отрицательных чисел в дополнительном коде и нулей для положительных чисел старше знакового разряда (дублирование знакового разряда) не изменяет представление десятичного числа, этим свойством воспользуемся при проектировании накапливающего сумматора.

1.2. Представление чисел со знаком

Числа с фиксированной запятой характеризуются длиной слова в битах, положением двоичной точки (binary point) и могут быть беззнаковыми или знаковыми. Позиция двоичной точки определяет число разрядов в целой и дробной частях машинного слова. Для представления знаковых чисел (отрицательных и положительных) старший разряд двоичного слова отводится под знак числа (sign bit). При представлении

беззнаковых чисел с фиксированной точкой разряд знака отсутствует, и он становится значимым разрядом. Отрицательные числа представляются в дополнительном коде. Данные с фиксированной запятой могут быть следующих типов: целыми (integers); дробными (fractional); обобщёнными (generalize). Обобщённый тип не имеет возможности определить позицию двоичной запятой по умолчанию и требует явного указания её положения. Этот тип данных специфицируют ufix и sfix форматами.

На рис. 1.1 представлено двоичное число с фиксированной запятой обобщенного типа, где b_i - i-й разряд числа; n - длина двоичного слова в битах; b_{n-1} - старший значимый разряд (MSB); b_0 - младший значимый разряд (LSB); 2^i - вес i-го разряда числа. Двоичная запятая занимает четвёртую позицию от младшего (LSB) разряда числа. При этом длина дробной части числа m = 4.

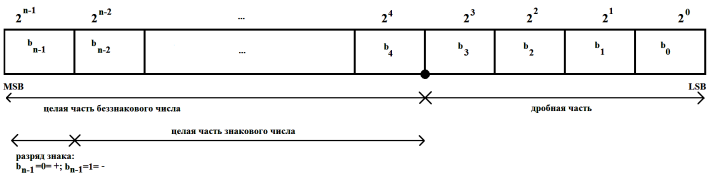


Рис. 1.1. Формат машинного слова

В файле помощи Fixed-Point Blockset системы Matlab для представления такого числа применяется следующая формула:

$$V=SQ+B,$$

где V - точное значение действительного десятичного числа; S - наклон; Q - квантованное (двоично-взвешенное) значение целого числа; B - смещение. Наклон S представляется следующим образом: $S=Fx2^E$, где F - наклон дробной части,

нормализованная величина $1 \le F < 2$; E - показатель степени E = -m. При проектировании устройств цифровой обработки сигналов принимают B = 0 и F = 1:

$$V \approx 2^{-m} x Q.$$

Квантованное значение Q приближённо представляет истинное значение действительного числа V в виде суммы произведений весовых коэффициентов b_i на веса 2^i соответствующих двоичных разрядов машинного слова; для беззнаковых чисел с фиксированной точкой определяется формулой

$$Q = \sum_{i=0}^{n-1} b_i \times 2^i .$$

Квантованное значение знаковых чисел определяется по формуле

$$Q = -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i .$$

Так как целые числа не имеют дробной части (m = 0), то выражение для V имеет вид

$$V = \sum_{i=0}^{n-1} b_i \times 2^i$$

и для знакового целого числа:

$$V = -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i .$$

В формате с фиксированной запятой без знака вещественное число V можно считать обозначением полинома

$$V = S * \left[\sum_{i=0}^{n-1} b_i 2^i \right].$$

Например, двоичное число в дополнительном коде 0011.0101 при длине машинного слова $n = 8$ и m = 4 представляет беззнаковое (MSB = 0) вещественное число 3.3125:

$$3.3125 = 2^{-4} \begin{pmatrix} 0*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + \\ + 1*2^2 + 0*2^1 + 1*2^0 \end{pmatrix}.$$

При MSB=1 будем иметь уже другое число -4.6875:

$$-4.6875 = 2^{-4}\left(-1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0\right).$$

1.3. Матричные умножители

Существует огромное число разновидностей матричных умножителей, превосходящих по скорости последовательностные умножители, основанные на методе сдвига и сложения. Известны и более сложные процедуры, например, с представлением суммирования в древовидном формате. В данном разделе не ставится цель разработать умножитель, превышающий по своим техническим характеристикам существующие матричные умножители, а необходимо показать читателю процедуру умножения чисел, представленных в дополнительном коде, методом правого сдвига и сложения, пригодную для реализации в базисе ПЛИС и черезвычайно популярную для реализации в базисе сигнальных процессоров.

В качестве сравнения рассмотрим один из хорошо известных умножителей чисел в дополнительном коде, так называемый умножитель Бо-Вули (Baugh-Wooley). Если A и B целые десятичные числа со знаком, то в дополнительном двоичном коде представляются в виде:

$$A = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \quad \text{и} \quad X = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i ,$$

то
$$A * X = a_{m-1}x_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i x_j 2^{i+j} - \sum_{i=0}^{m-2} x_{n-1}a_i 2^{i+n-1} - \sum_{j=0}^{n-2} a_{m-1}x_j 2^{j+m-1} .$$

На рис. 1.2 показан пример умножения чисел 5x5, представленных дополнительным кодом по формуле, а на рис. 1.3 - пример умножения чисел 5x5, представленных дополнительным кодом по схеме Бо-Вули. На рис. 1.4 показана схема преобразований, позволяющая перевести частичные произведения со знаком в беззнаковые величины. На рис. 1.5 показан матричный умножитель Бо-Вули размерностью 5x5 чисел, представленных в дополнительном коде. Наличие полных сумматоров (FA) в матричной структуре такого умножителя является главным достоинством для реализации в базисе заказных БИС, а недостатком - пониженное быстродействие за счет увеличения высоты столбца с 5 до 7.

Ниже показы примеры умножения для различных случаев по схеме Бо-Вули (рис. 1.6).

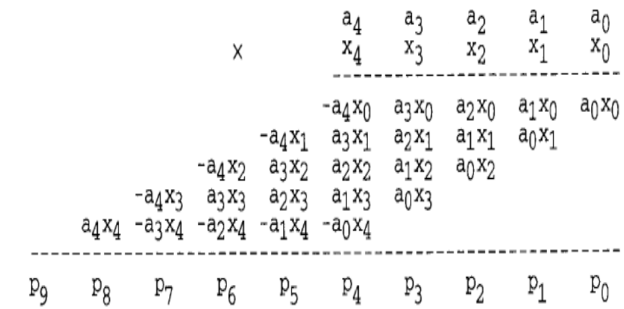


Рис. 1.2. Пример умножения чисел 5x5, представленных дополнительным кодом по формуле

					a_4	a_3	a_2	a_1	a_0
				x	x_4	x_3	x_2	x_1	x_0
					$a_4\overline{x}_0$	a_3x_0	a_2x_0	a_1x_0	a_0x_0
			+	$a_4\overline{x}_1$	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
			$a_4\overline{x}_2$	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
		$a_4\overline{x}_3$	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
	a_4x_4	$\overline{a}_3x_4$	$\overline{a}_2x_4$	$\overline{a}_1x_4$	$\overline{a}_0x_4$				
	$\overline{a}_4$				a_4				
1	$\overline{x}_4$				x_4				
$\mathbf{p_9}$	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0
$\mathbf{-2^9}$	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Рис. 1.3. Пример умножения чисел 5x5, представленных дополнительным кодом по схеме Бо-Вули

$$-a_4 x_i = a_4(1-x_i) - a_4 = a_4\bar{x}_i - a_4$$

$$-a_4 \to a_4 - 2a_4 \Rightarrow a_4 + (-a_4 \text{ перенос в следующий столбец})$$

$$\Rightarrow$$

$$-a_4 x_0 \Rightarrow a_4\bar{x}_0 + a_4 \qquad +(-a_4 \text{ перенос в следующий столбец})$$

$$-a_4 x_1 \Rightarrow a_4\bar{x}_1 + a_4 \quad -a_4 \quad +(-a_4 \text{ перенос в следующий столбец})$$

$$-a_4 x_2 \Rightarrow a_4\bar{x}_2 + a_4 \quad -a_4 \quad +(-a_4 \text{ перенос в следующий столбец})$$

$$-a_4 x_3 \Rightarrow a_4\bar{x}_3 + a_4 \quad -a_4 \quad +(-a_4 \text{ перенос в следующий столбец})$$

$$-a_4 x_4 \Rightarrow a_4\bar{x}_4 + a_4 \quad -a_4 \quad +(-a_4 \text{ перенос в следующий столбец})$$

$$-a_i x_4 = x_4(1-a_i) - x_4 = x_4\bar{a}_i - x_4$$

$$-x_4 \to x_4 - 2x_4 \Rightarrow x_4 + (-x_4 \text{ перенос в следующий столбец})$$

$$-x_4 a_0 \Rightarrow x_4\bar{a}_0 + x_4$$

$$-x_4 a_1 \Rightarrow x_4\bar{a}_1$$

$$-x_4 a_2 \Rightarrow x_4\bar{a}_2$$

$$-x_4 a_3 \Rightarrow x_4\bar{a}_3$$

$$-x_4 a_4 \Rightarrow x_4\bar{a}_4$$

$$-x_4 = \bar{x}_4 - 1$$

$$-a_4 x_0 \Rightarrow a_4\bar{x}_0 + a_4$$

$$-a_4 x_1 \Rightarrow a_4\bar{x}_1$$

$$-a_4 x_2 \Rightarrow a_4\bar{x}_2$$

$$-a_4 x_3 \Rightarrow a_4\bar{x}_3$$

$$-a_4 x_4 \Rightarrow a_4\bar{x}_4$$

$$-a_4 = \bar{a}_4 - 1$$

Рис. 1.4. Схема преобразований Бо-Вули

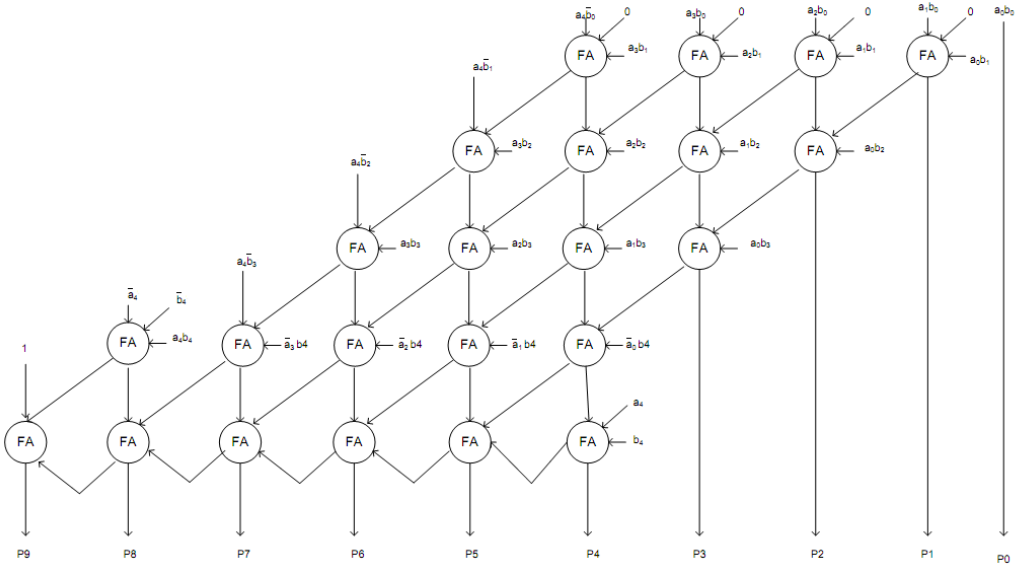
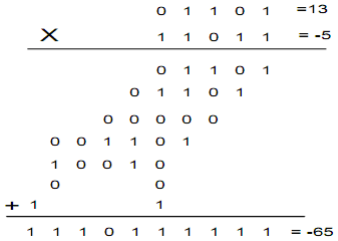
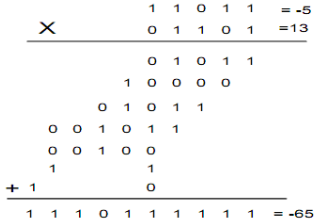


Рис. 1.5. Матричный умножитель Бо-Вули размерностью 5x5 чисел, представленных в дополнительном коде

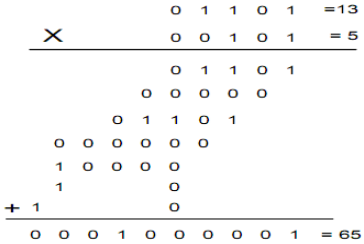
Случай 1.
Множимое - положительное число
Множитель - отрицательное число



Случай 2.
Множимое - отрицательное число
Множитель - положительное число



Случай 3.
Множимое и множитель - положительные числа



Случай 4.
Множимое и множитель - отрицательные числа

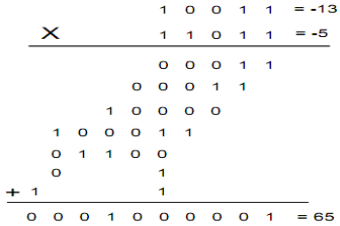


Рис. 1.6. Примеры умножения для различных случаев по схеме Бо-Вули

1.4. Проектирование умножителя методом правого сдвига и сложения с управляющим автоматом в базисе ПЛИС

Для проектирования КИХ-фильтров в базисе процессоров цифровой обработки сигналов (ЦОС-процессор) используется общепринятая методика умножения с накоплением с применением так называемых MAC-блоков из-за отсутствия встроенных комбинационных умножителей.

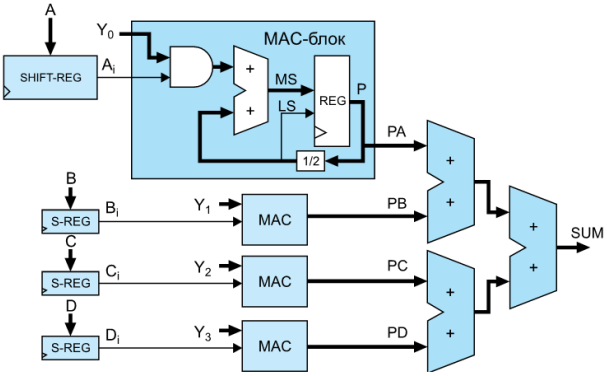
Использование данного метода для умножения чисел в базисе сигнальных процессоров является чрезвычайно популярным у разработчиков РЭА. На базе данного метода реализуются схемы быстрого умножения (например, кодирование по Буту, которое позволяет уменьшать число частичных произведений вдвое, умножение по основанию 4, модифицированное кодирование по Буту).

При реализации КИХ-фильтра на четыре отвода в базисе ЦОС-процессоров требуется четыре блока умножения с накоплением (рис. 1.7, а). Применение последовательной распределенной арифметики позволяет сократить число используемых ресурсов за счет использования составных частей MAC-блока. Это четыре блока логики генерации частичных произведений, получаемых применением булевой операции логическое И к множимому (многоразрядные константы, являющиеся коэффициентами фильтра) и битовому значению множителя с выходов линии задержки, а также единственный масштабирующий аккумулятор. При этом дерево многоразрядных сумматоров не сокращается (рис. 1.7, б). При проектировании фильтра в базисе ПЛИС (рис. 1.7, в) на распределенной последовательной арифметике логика генерации частичных произведений и их последующее суммирование с помощью дерева многоразрядных сумматоров

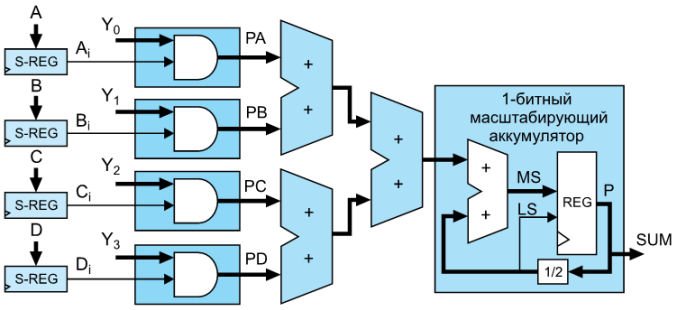
реализуются единственной таблицей перекодировки (LUT). Суммирование значений с выходов таблицы перекодировки осуществляется с использованием масштабирующего аккумулятора. Реализация КИХ-фильтров в базисе ПЛИС с использованием распределенной арифметики обеспечивает наивысшее быстродействие системы и наименьшее число используемых ресурсов проекта.

Рассмотрим проектирование MAC-блока с использованием управляющего автомата. Встраивание автомата в схему позволяет получить готовую функцию без использования дополнительных управляющих сигналов для умножения двух четырехразрядных чисел без знака. По входам MAC-блока потребуются всего лишь три сигнала: сигнал асинхронного сброса res, сигнал тактирования clk и сигнал разрешения загрузки числа X (множителя) в сдвиговый регистр load_PSC. Для корректной работы схемы необходимо обнулить все регистры умножителя (активный – высокий уровень сигнала res). Поскольку все регистры, в том числе и регистр для хранения состояний в управляющем автомате обнуляются лишь перед началом работы единожды, то для упрощения процесса разработки схемы можно воспользоваться асинхронным сбросом.

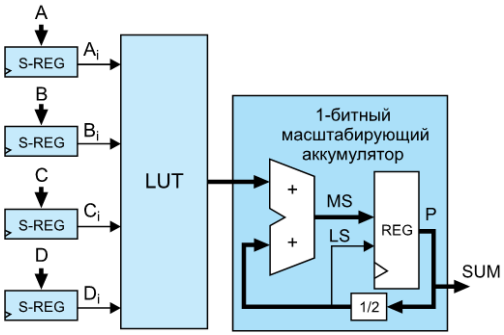
Принцип умножения методом правого сдвига и сложения показан на рис. 1.8. Идея схемы метода умножения методом правого сдвига с накоплением показана на рис. 1.9. На рис. 1.10 предлагается структурная схема метода умножения с использованием управляющего автомата.



а)



б)



в)

Рис. 1.7. Миграция проекта КИХ-фильтра на четыре отвода:
а) и б) реализация в базисе сигнальных процессоров;
в) в базисе ПЛИС

	Cout	2 тетрада	1 тетрада	2 тетрада	2 тетрада	1 и 2 тетрады
a 10 x 11			1 0 1 0 1 0 1 1			
$p^{(0)}$ $+x_0a$		0 0 0 0 1 0 1 0	0 0 0 0	0 a	0 10	
$2p^{(1)}$ $p^{(1)}$ $+x_1a$	0	1 0 1 0 0 1 0 1 1 0 1 0	0 0 0 0	a a/2 a	10 5 10	80
$2p^{(2)}$ $p^{(2)}$ $+x_2a$	0	1 1 1 1 0 1 1 1 0 0 0 0	0 1 0 0 0	a/2+a (a/2+a)/2 0	15 7 0	120
$2p^{(3)}$ $p^{(3)}$ $+x_3a$	0	0 1 1 1 0 0 1 1 1 0 1 0	1 0 1 1 0 0	(a/2+a)/2 ((a/2+a)/2)/2 a	7 3 10	60
$2p^{(4)}$ $p^{(4)}$	0	1 1 0 1 0 1 1 0	1 1 0 1 1 1 0	((a/2+a)/2)/2+a (((a/2+a)/2)/2+a)/2	13 6	110

Рис. 1.8. Принцип умножения методом правого сдвига и сложения. Умножение десятичного числа 10 на десятичное число 11

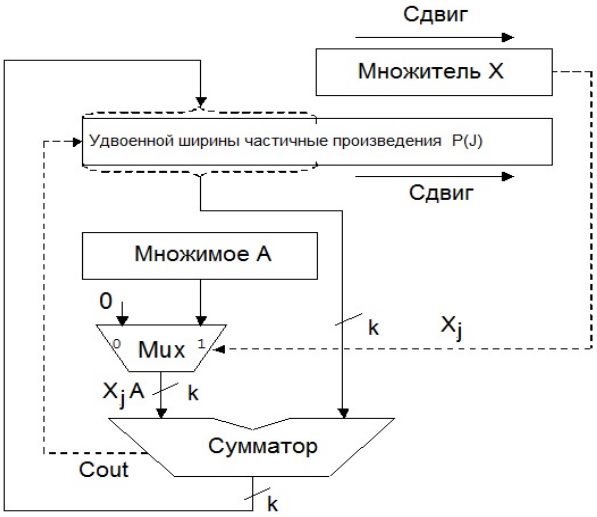


Рис. 1.9. Структурная схема умножителя методом правого сдвига и сложения

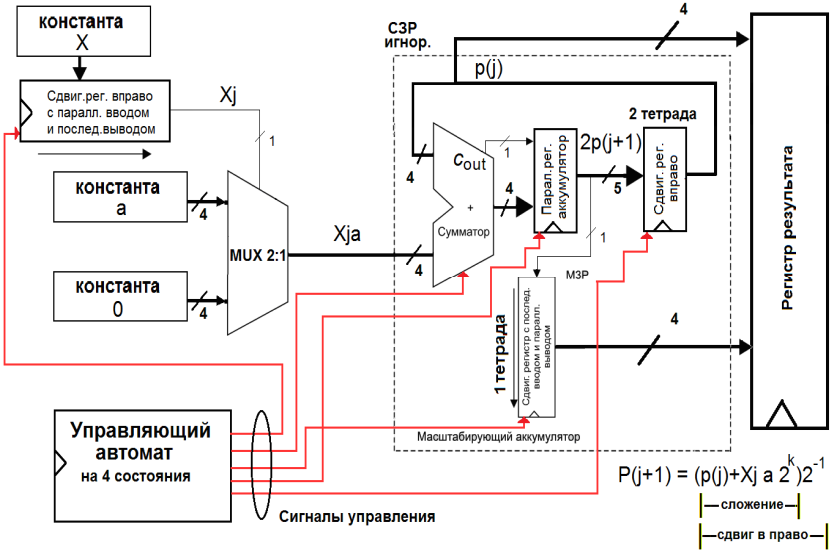


Рис. 1.10. Предлагаемая структурная схема умножителя с управляющим автоматом

Структурная схема умножителя двух 4-разрядных чисел, представленных в двоичном коде (целые, положительные числа), состоит из шинного мультиплексора 2 в 1, сдвигового регистра, цифрового автомата на четыре состояния и масштабирующего аккумулятора (рис. 1.11). Шинный мультиплексор 2 в 1 и сдвиговый регистр вправо реализуют логику генерации частичных произведений. В основе масштабирующего аккумулятора лежит синхронизируемый сумматор с сигналом разрешения тактирования (рис. 1.12).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY avtomat IS
PORT(
```

```
Res,clk : IN      STD_LOGIC;
Ena_Add, LoadPSC,ena_shift,Stop : OUT          STD_LOGIC;
Qa      : OUT STD_LOGIC_VECTOR(4 downto 0));
END avtomat;

ARCHITECTURE a OF avtomat IS
TYPE state_values IS (SA, SB, SC, SD);
signal state,next_state: state_values;
SIGNAL cnt: STD_LOGIC_VECTOR(4 downto 0);
BEGIN
statereg: process(clk,Res)
begin
          if (Res = '1') then state<=SA;
                    elsif (clk'event and clk='1') then
                                        state<=next_state;
          end if;
end process statereg;
process(state)
begin
case state is
when SA=> next_state<=SB;
when SB=> next_state<=SC;
when SC=>  next_state<=SD;
when SD=> next_state<=SA;
end case;
end process;
process (state)
begin
case state is
when SA=>Ena_Add<='0';
                              LoadPSC<='0'; ena_shift<='1';
when SB=>
                              Ena_Add<='1'; LoadPSC<='0';
                              ena_shift<='0';
when SC=>
                              Ena_Add<='0';
                              LoadPSC<='0';
```

```
                        ena_shift<='0';
when SD=>
                        Ena_Add<='0';
                        LoadPSC<='1';
                        ena_shift<='1';
end case;
end process;
process (clk, res)
        begin
                if (res = '1') then
                        cnt <=(others=>'0');
                elsif (clk'event and clk = '1') then
                        if cnt = "10001" then Stop <= '1';
                        else cnt <= cnt+'1';
                        end if;
                end if;
end process;
        Qa <= cnt;
END a;
```

Пример. Код языка VHDL управляющего автомата на четыре состояния

Цифровой автомат представляет собой автомат Мура на четыре состояния (SA (первое состояние), SB (второе состояние), SC (третье состояние) и SD (четвертое состояние)) и формирует три управляющих сигнала ena_add_temp, load_acc и ena_shift_temp по срезу фронта синхроимпульса clk (пример).

Два из которых (ena_add_temp, load_acc) - не перекрывающиеся (рис. 1.13). По активному уровню сигнала асинхронного сброса res автомат попадает в первое состояние SA, по низкому уровню res осуществляется переход по состояниям. На рис. 1.13 над сигналами ena_shift_temp и load_acc нанесены номера состояний цифрового автомата. Так, сигнал ena_shift_temp активен в первом и четвертом состояниях.

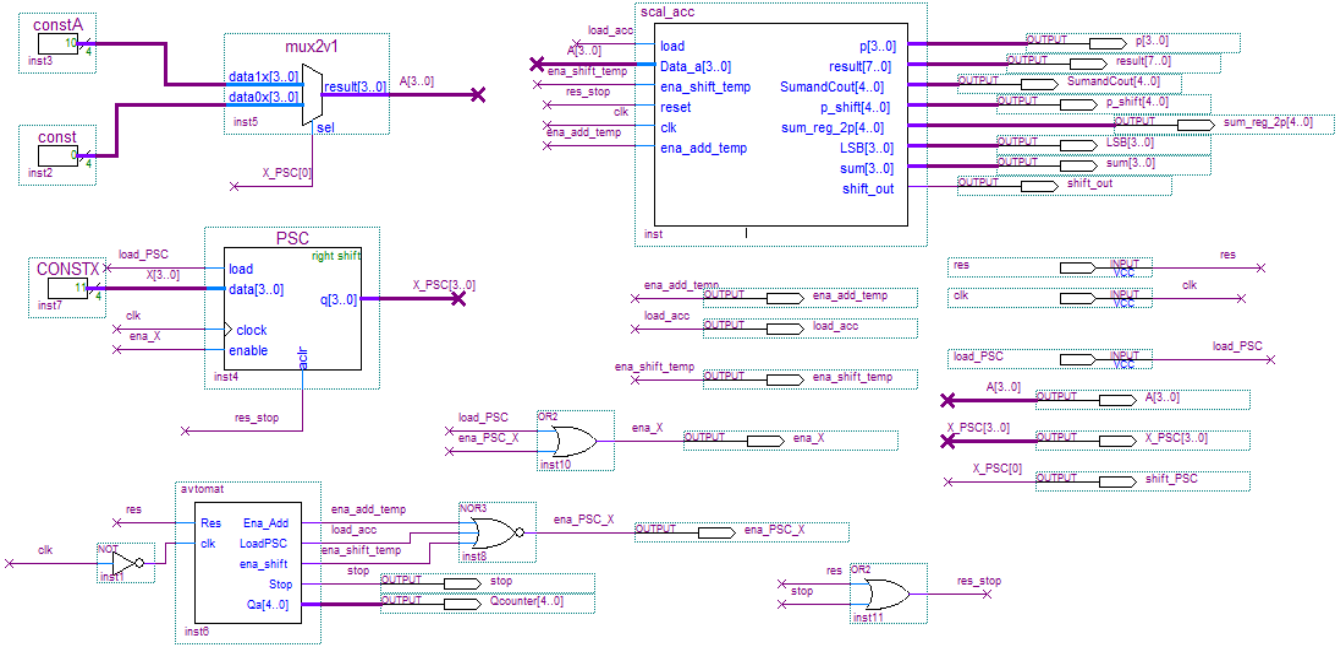


Рис. 1.11. Схема умножителя в САПР ПЛИС Quatus II

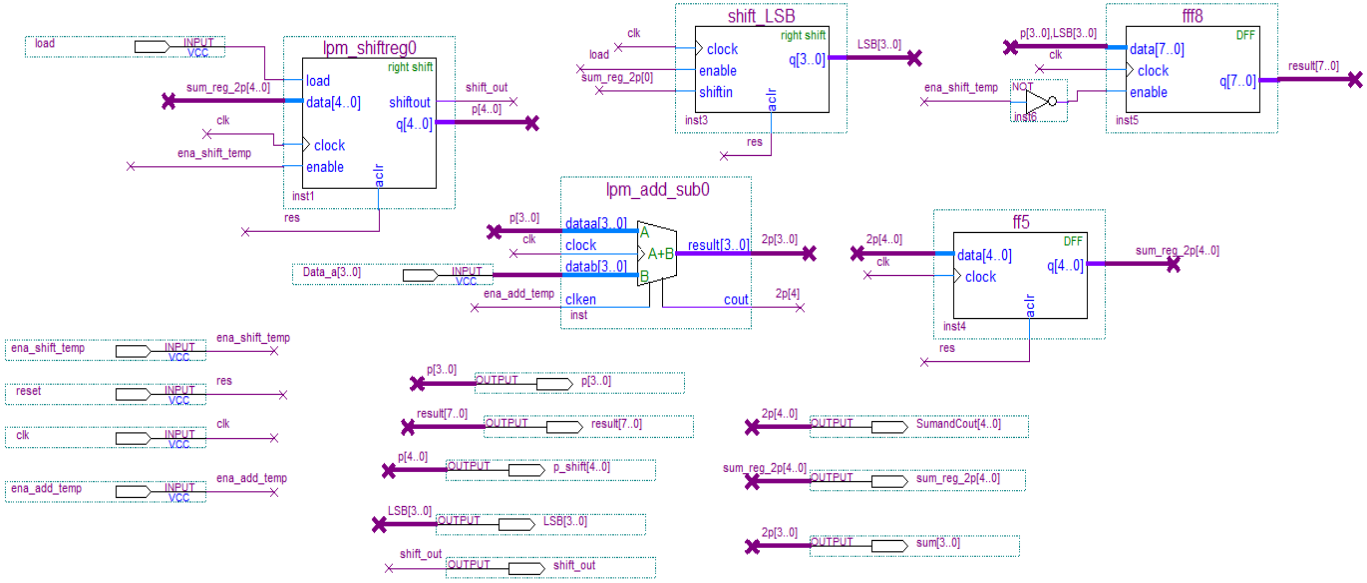


Рис. 1.12. Схема масштабирующего аккумулятора

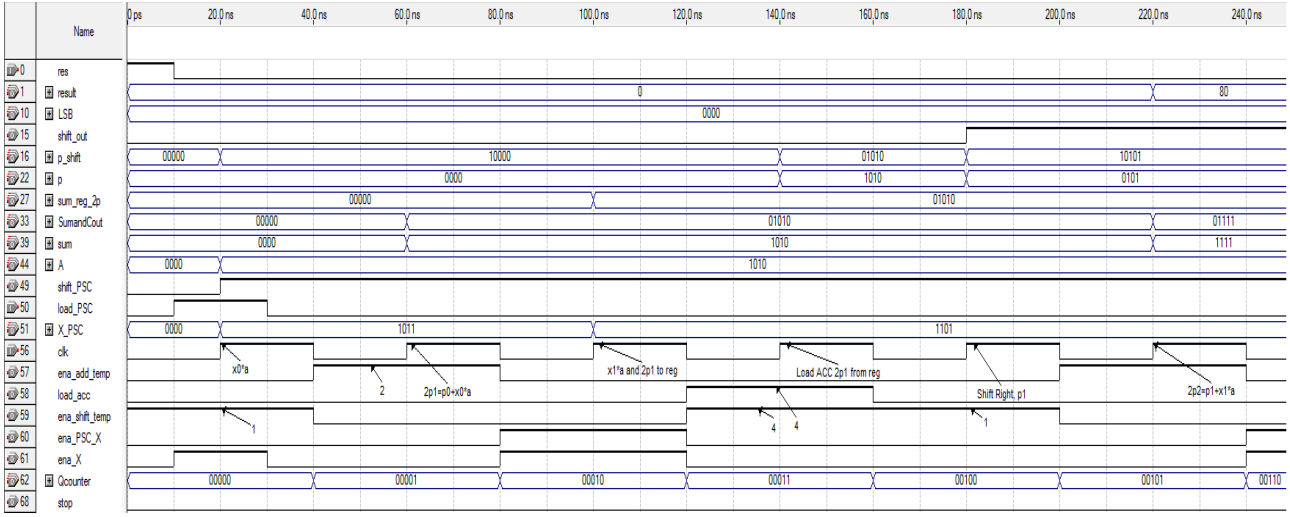


Рис. 1.13. Временные диаграммы процесса вычисления удвоенных первого 2P(1) и второго частичных произведений 2P(2)

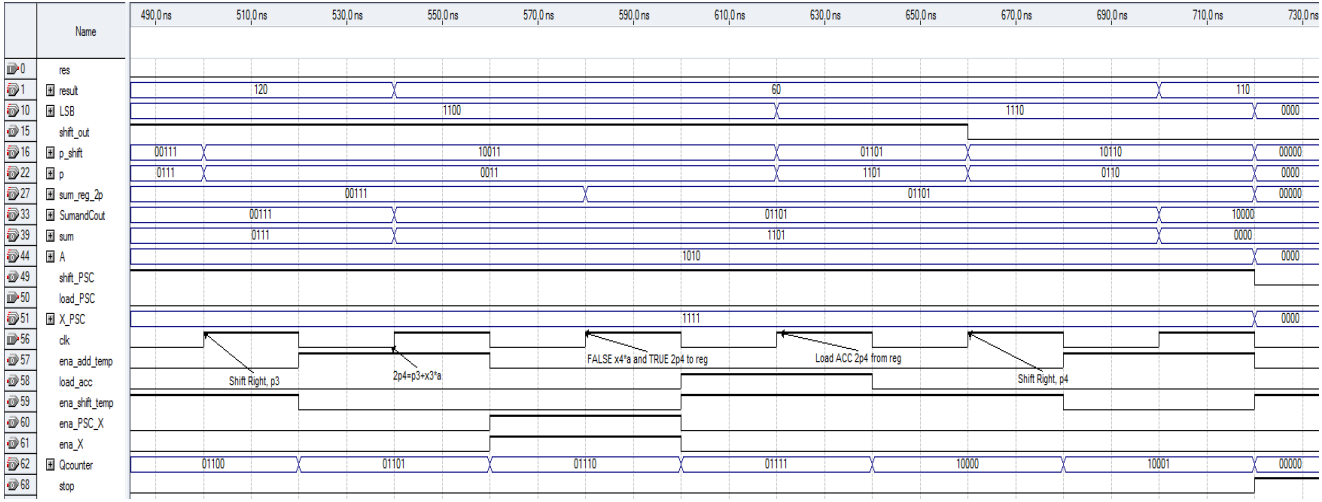


Рис. 1.14. Временные диаграммы процесса вычисления удвоенного четвертого частичного произведения 2P(4)

Сигнал ena_add_temp формируется, когда автомат находится во втором состоянии SB. Сигнал load_acc формируется в четвертом состоянии. Таким образом, удается обеспечить конвейерный режим работы масштабирующего аккумулятора, при этом формируемые сигналы ena_add_temp, load_acc и ena_shift_temp являются синхронными для всех регистров умножителя. Так, второй передний фронт синхроимпульса оказывается ровно над половиной высокого уровня сигнал ena_add_temp. Что обеспечивает корректную работу синхронного сумматора (рис. 1.13). Четвертый передний фронт также оказывается ровно над половиной высоких уровней сигналов load_acc и ena_shift_temp. Что обеспечивает загрузку числа с промежуточного регистра (аккумулятора) в сдвиговый регистр lpm_shiftreg0. Пятый фронт также оказывается ровно над серединой высокого уровня сигнала ena_shift_temp для выполнения операции деления на 2 (сдвиг вправо).

В управляющий автомат встроен суммирующий счетчик, который подсчитывает число синхроимпульсов. И при достижении 18 (отсчет ведется с нуля) синхроимпульса вырабатывается сигнал остановки работы умножителя, который сбрасывает все регистры умножителя в ноль, кроме регистра результата (fff8), запись информации в который осуществляется низким уровнем сигнала ena_shift_temp.

На информационные входы мультиплексора подключаются две константы. Множимое (число A) и логический ноль. На адресный вход мультиплексора подключается младший разряд (множителя) сдвигового регистра (мегафункция LPM_SHIFTREG) с параллельным вводом информации и последовательным выводом. Такой регистр будем называть конвертор (преобразователь параллельного кода в последовательный) и обозначать как PSC. Регистр настроен на сдвиг вправо и имеет синхронные сигналы загрузки load и разрешения тактирования enable активные высоким уровнем. При сдвиге вправо в старший

разряд заносится логическая 1, а младший разряд теряется. В сдвиговый регистр по высокому уровню сигнала load_PSC записывается число X (множитель). При этом сигнал ena_X на входе enable должен быть высокого уровня, который может быть получен объединением по ИЛИ сигналов load_PSC и ena_PSC_X. Сигнал ena_PSC_X является выходом цифрового автомата и получается применением операции 3ИЛИ-НЕ над сигналами ena_add_temp, load_acc и ena_shift_temp, т.е. он возникает только в том случае, когда сигналы ena_add_temp, load_acc и ena_shift_temp низкого уровня. В первом такте синхроимпульса сигнал ena_PSC_X не активен и не оказывает влияния на формирование сигнала ena_X и используется при последующем сдвиге множителя X вправо. Параллельная загрузка числа X происходит по принципу: если активен сигнал загрузки load_PSC, то значит должен быть активен и сигнал разрешения тактирования ena_X.

Недостатком такого решения является неразрешенный сдвиг вправо пятиразрядным сдвиговым регистром (lpm_shiftreg0) масштабируемого аккумулятора при первом фронте синхроимпульса, на выходах которого появляется 1 в старшем разряде при активном сигнале ena_shift_temp, что и демонстрирует рис. 1.13. Однако это не влияет на работу масштабирующего аккумулятора, т.к. перед сложением сдвиговый регистр перегружается правильным значением по четвертому фронту синхросигнала при активных сигналах load_acc и ena_shift_temp. Неразрешенный сдвиг будет проявляться и при последующих синхроимпульсах при наличии активного уровня сигнала ena_shift_temp. Для противодействия этому явлению старший значащий разряд p[4] регистра lpm_shiftreg0 после сдвига просто игнорируется, а на вход сумматора поступает уменьшенное на два значение частичного произведения p[3..0] с 4-битной точностью представления.

Рассмотрим работу масштабирующего аккумулятора. По первому фронту синхроимпульса clk при активных

сигналах load_PSC и ena_X происходит загрузка числа X в сдвиговый регистр. По второму фронту синхроимпульса управляющий автомат вырабатывает синхронный сигнал разрешения тактирования ena_add_temp для сумматора масштабирующего аккумулятора. На выходах сумматора формируется первое удвоенное частичное произведение 2P(1)=P(0)+X(0)*A (шина 2p[3..0]), равное 10 при этом P(0)=0. Выход переноса Cout и 2P[3..0] объединяются в пятиразрядную шину, значения которой сохраняются в промежуточном параллельном регистре-аккумуляторе (выход sum_reg_2p[4..0]) по третьему фронту синхроимпульса.

По четвертому фронту синхроимпульса при активных сигналах load_acc и ena_shift_temp происходит загрузка первого удвоенного частичного произведения равного десяти в сдвиговый регистр. По пятому фронту при активном сигнале ena_shift_temp происходит сдвиг вправо удвоенного частичного произведения 2P(1). При этом на выходах сдвигового регистра образуется число 5. По шестому фронту синхроимпульса при активном сигнале ena_add_temp произойдет сложение числа A (десятичное число 10) с числом 5 и на выходах сумматора сформируется второе частичное произведение 2P(2)=P(1)+X(1)*A равное числу 15. Что в точности соответствует принципу умножения, продемонстрированному на рис. 1.8.

На рис. 1.14 показан момент окончания счета. По 15-му фронту синхроимпульса (отсчет по тексту ведется с 1-го фронта синхроимпульса) произойдет неразрешенная загрузка четвертого бита в сдвиговый регистр PSC по высокому уровню сигнала ena_PSC_X и сформируется еще одна копия сигнала A т.е. X(4)*A. Однако это тоже не повлияет на результат, т.к. последующего сложения уже не будет. Потребуются еще два такта синхроимпульса для загрузки удвоенного произведения 2P(4) в аккумулятор и последующий сдвиг вправо для формирования P(4). И по 18-му такту

импульса результат умножения будет доступен в регистре результата fff8.

Было проведено тестирование разработанной схемы на предмет умножения двух 4-разрядных чисел без знака в диапазоне входных значений от 0 до 15. На рис. 1.15 показан принцип умножения десятичного числа 15 на 15, а на рис. 1.16 - представлены временные диаграммы процесса умножения.

	Cout	2 тетрада	1 тетрада	1 и 2 тетрады
a x		1 1 1 1 1 1 1 1		
$p^{(0)}$ $+x_0a$		0 0 0 0 1 1 1 1		
$2p^{(1)}$ $p^{(1)}$ $+x_1a$	0	1 1 1 1 0 1 1 1 1 1 1 1	1 0 0 0	120
$2p^{(2)}$ $p^{(2)}$ $+x_2a$	1	0 1 1 0 1 0 1 1 1 1 1 1	0 1 0 0	180
$2p^{(3)}$ $p^{(3)}$ $+x_3a$	1	1 0 1 0 1 1 0 1 1 1 1 1	0 0 1 0	210
$2p^{(4)}$ $p^{(4)}$	1	1 1 0 0 1 1 1 0	0 0 0 1	225

Рис. 1.15. Принцип умножения методом правого сдвига и сложения. Умножение десятичного числа 15 на десятичное число 15

Разработан MAC-блок для умножения двух 4-разрядных чисел без знака с использованием метода умножения и накопления. Управление блоком осуществляется с помощью цифрового автомата на четыре состояния. Предложенная схема реализации MAC-блока в базисе ПЛИС может быть использована при проектировании КИХ-фильтров.

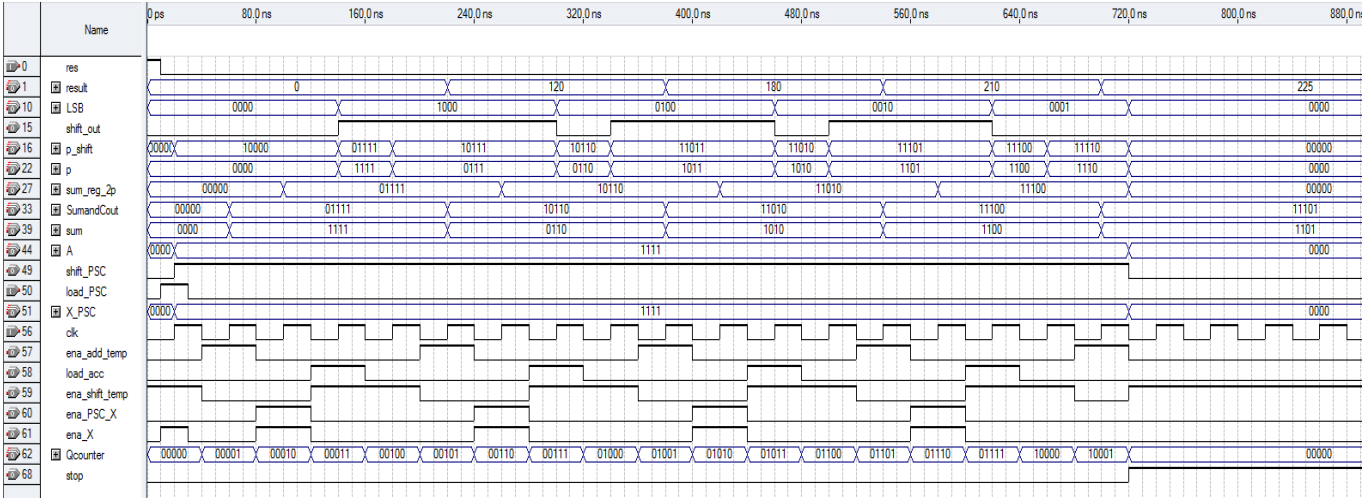


Рис. 1.16. Умножение числа десятичного числа 15 на десятичное число 15. Результат 225

1.5. Проектирование умножителя целых чисел со знаком методом правого сдвига и сложения в базисе ПЛИС

Рассмотрим пример проектирования последовательностного универсального умножителя целых чисел, представленных в дополнительном коде, методом правого сдвига и сложения (MAC-блок) в базисе ПЛИС.

В качестве базовой схемы разрабатываемого MAC-блока возьмем схему умножителя целых без знаковых чисел с управляющим автоматом на четыре состояния (раздел 1.3).

Рассмотрим умножение чисел со знаком в “столбик” (рис. 1.17). Дополнение до двух можно получить, если прибавить 1 к результату обращения. Обращение логически эквивалентно инверсии каждого бита в числе. Вентили Исключающее ИЛИ можно применить для избирательной инверсии в зависимости от значения управляющего сигнала. Прибавление 1 к результату обращения можно реализовать, задавая 1 на входе переноса сумматора.

На рис. 1.18 показан принцип умножения чисел представленных дополнительным кодом, на примере умножения -5x-3.

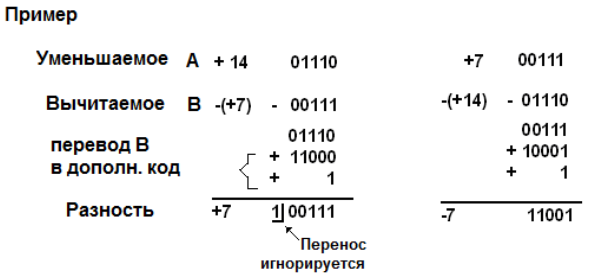


Рис. 1.17. Вычитание с использованием дополнительного кода (дополнение до двух). Осуществляются инвертирование вычитаемого, суммирование и перенос 1 в младший разряд с последующим сложением

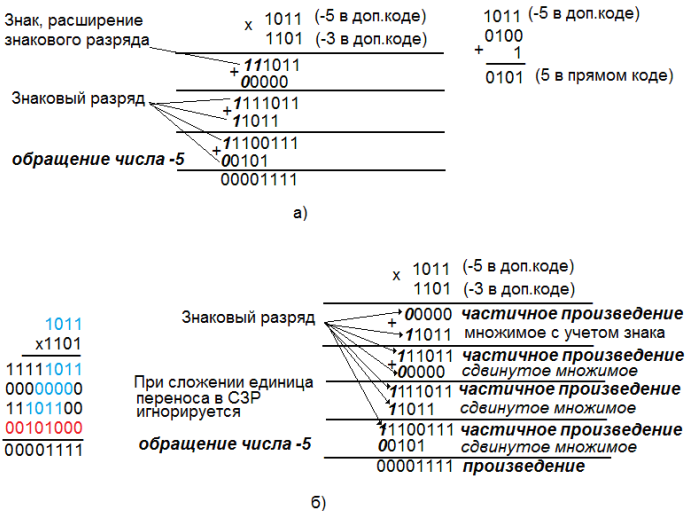


Рис. 1.18. Умножение в столбик (а); умножение методом сдвига множимого и последующего сложения с частичным произведением (умножение чисел -5x-3, представленных в дополнительном коде) (б)

Представление процесса умножения в точечной нотации, в которой под каждой точкой подразумевается логическая 1 или логический 0, позволяет получить рекуррентную формулу (рис. 1.19).

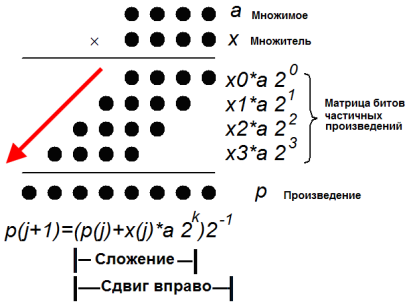


Рис. 1.19. Представление процесса умножения методом правого сдвига и сложения в точечной нотации

Ниже показаны примеры умножения, рассматриваемые при разработке схемы универсального умножителя.

Случай 1. Множимое - отрицательное число Множитель - положительное число -10x11	**Случай 2.** Множимое и множитель - отрицательные числа -10x-11
Случай 3. Множимое и множитель - четные отрицательные числа -4x-4	**Случай 4.** Множимое - положительное число Множитель - отрицательное число 5x-3

На рис. 1.20 показан принцип умножения методом правого сдвига и сложения для двух случаев в формате проектируемого MAC-блока. В первом случае осуществляется умножение числа со знаком -10 на число 11, а во втором - умножение -10 на -11. В первом случае множимое (-10) переводится в дополнительный код (дополнение до двух).

Множитель (11) – целое положительное число, расширенное знаковым разрядом 0, представлено в прямом коде. Для удвоенных частичных произведений 2p(1), 2p(2), 2p(3), 2p(4) и 2p(5) в поле MSB (название полей произвольное) необходимо добавить логическую 1. При формировании частичных произведений методом правого сдвига p(1), p(2), p(3), p(4) и p(5) логическая 1 из поля MSB попадает в старший разряд второй тетрады (название “тетрады” в данном случае не корректно, т.к. это поле уже 5-разрядное, но сохранено для приемлемости с принципом умножения без знаковых чисел).

При сложении p(3) с x3*a единица переноса в старший разряд, т.е. в поле MSB, игнорируется. Данная схема

вычислений справедлива только для случая, когда в младшем разряде множителя находится 1.

	MSB	2 тетрада	1 тетрада
a(-10) x(11)		1 0 1 1 0 0 1 0 1 1	
p(0) +x0*a		0 0 0 0 0 1 0 1 1 0	
2p(1) p(1) +x1*a	1	1 0 1 1 0 1 1 0 1 1 1 0 1 1 0	0
2p(2) p(2) +x2*a	1	1 0 0 0 1 1 1 0 0 0 0 0 0 0 0	0 1 0
2p(3) p(3) +x3(a)	1	1 1 0 0 0 1 1 1 0 0 1 0 1 1 0	1 0 0 1 0
2p(4) p(4) x4*a	1	1 0 0 1 0 1 1 0 0 1 0 0 0 0 0	0 1 0 0 0 1 0
2p5 p5	1	1 1 0 0 1 1 1 1 0 0	0 0 1 0 1 0 0 1 0

а)

	MSB	2 тетрада	1 тетрада
a(-10) x(-11)		1 0 1 1 0 1 0 1 0 1	
p(0) +x0*a		0 0 0 0 0 1 0 1 1 0	
2p(1) p(1) +x1*a	1	1 0 1 1 0 1 1 0 1 1 0 0 0 0 0	0
2p(2) p(2) +x2*a	1	1 1 0 1 1 1 1 1 0 1 1 0 1 1 0	0 1 0
2p(3) p(3) +x3(a)	1	1 0 0 1 1 1 1 0 0 1 0 0 0 0 0	1 0 1 1 0
2p(4) p(4) (-x4*a)	1	1 1 0 0 1 1 1 1 0 0 0 1 0 1 0	1 1 0 1 1 1 0
2p5 p5	0	0 0 1 1 0 0 0 0 1 1	1 1 1 0 0 1 1 1 0

б)

Рис. 1.20. Принцип умножения методом правого сдвига и сложения:
а) умножение -10x11; б) умножение -10x-11

При умножении двух отрицательных чисел, представленных дополнительным кодом (например, -10x-11), необходимо произвести два действия. Первое, необходимо учесть знак при представлении числа в дополнительном коде, что достигается обращением произведения старшего разряда множителя на множимое с последующим прибавлением 1 к младшему разряду. Дополнительный код произведения (–x4*a) при x4=1 есть число 10. Перевод в дополнительный код

произведения (–x4*a) должен быть осуществлен до операции сложения, т.е. до получения удвоенного частичного произведения 2p(5).

Второе, при формировании удвоенного частичного произведения 2p(5) необходимо произвести коррекцию, т.е. в поле MSB поставить логический ноль.

Рассмотрим умножение четных чисел со знаком (рис. 1.21). При умножении -4x-4 в поле MSB для удвоенных значений частичных произведений 2p(1) и 2p(2) должны стоять нули. А при умножении -8x-8 нуль в поле MSB должен быть еще 0 и для частичного произведения 2p(3). Далее, принцип умножения не отличается от умножения чисел, представленных дополнительным кодом (например, -10x-11).

	MSB	2 тетрада	1 тетрада
a (-4) x(-4)		1 1 1 0 0 1 1 1 0 0	
p(0) +x0*a		0 0 0 0 0 0 0 0 0 0	
2p(1) p(1) +x1*a	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0
2p(2) p(2) +x2*a	0	0 0 0 0 0 0 0 0 0 0 1 1 1 0 0	0 0 0
2p(3) p(3) +x3(a)	1	1 1 1 0 0 1 1 1 1 0 1 1 1 0 0	0 0 0 0 0
2p(4) p(4) (-x4*a)	1	1 1 0 1 0 1 1 1 0 1 0 0 1 0 0	0 0 0 0 0 0 0
2p5 p5	0	0 0 0 0 1 0 0 0 0 0	0 0 0 0 1 0 0 0 0

а)

	MSB	2 тетрада	1 тетрада
a(-8) x(-8)		1 1 0 0 0 1 1 0 0 0	
p(0) +x0*a		0 0 0 0 0 0 0 0 0 0	
2p(1) p(1) +x1*a	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0
2p(2) p(2) +x2*a	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0
2p(3) p(3) +x3(a)	0	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0	0 0 0 0 0
2p(4) p(4) (-x4*a)	1	1 1 0 0 0 1 1 1 0 0 0 1 0 0 0	0 0 0 0 0 0 0
2p5 p5	0	0 0 1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0 0

б)

Рис. 1.21. Принцип умножения методом правого сдвига и сложения: а) умножение -4x-4; б) умножение -8x-8

На рис. 1.22 показан принцип умножения методом правого сдвига и сложения в случае, когда множимое – положительное, а множитель - отрицательное числа.

	MSB	2 тетрада	1 тетрада
a (5) x (-3)		0 0 1 0 1 1 1 1 0 1	
p(0) +x0*a		0 0 0 0 0 0 0 1 0 1	
2p(1) p(1) +x1*a	0	0 0 1 0 1 0 0 0 1 0 0 0 0 0 0	1
2p(2) p(2) +x2*a	0	0 0 0 1 0 0 0 0 0 1 0 0 1 0 1	1 0 1
2p(3) p(3) +x3(a)	0	0 0 1 1 0 0 0 0 1 1 0 0 1 0 1	0 1 0 0 1
2p(4) p(4) (-x4*a)	0	0 1 0 0 0 0 0 1 0 0 1 1 0 1 1	0 0 1 0 0 0 1
2p5 p5	1	1 1 1 1 1 1 1 1 1 1	0 0 0 1 1 0 0 0 1

	MSB	2 тетрада	1 тетрада
a (2) x (-2)		0 0 0 1 0 1 1 1 1 0	
p(0) +x0*a		0 0 0 0 0 0 0 0 0 0	
2p(1) p(1) +x1*a	0	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	0
2p(2) p(2) +x2*a	0	0 0 0 1 0 0 0 0 0 1 0 0 0 1 0	0 0 0
2p(3) p(3) +x3(a)	0	0 0 0 1 1 0 0 0 0 1 0 0 0 1 0	0 0 1 0 0
2p(4) p(4) (-x4*a)	0	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0	1 0 0 1 1 0 0
2p5 p5	1	1 1 1 1 1 1 1 1 1 1	1 1 0 0 1 1 1 0 0

Рис. 1.22. Принцип умножения методом правого сдвига и сложения: а) умножение 5x-3; б) умножение 2x-2

Разработанный MAC-блок также способен умножать числа без знака. Для этого применяется принцип, показанный на рис. 1.23. Единица переноса при сложении в поле “2 тетрада” уже не игнорируется, а переносится в поле Cout (сигнал Cout является выходом переноса многоразрядного сумматора масштабирующего аккумулятора).

	Cout	2 тетрада	1 тетрада	1 и 2 тетрады
a x		1 1 1 1 1 1 1 1		
$p^{(0)}$ $+x_0a$		0 0 0 0 1 1 1 1		
$2p^{(1)}$ $p^{(1)}$ $+x_1a$	0	1 1 1 1 0 1 1 1 1 1 1 1	1 0 0 0	120
$2p^{(2)}$ $p^{(2)}$ $+x_2a$	1	0 1 1 0 1 0 1 1 1 1 1 1	0 1 0 0	180
$2p^{(3)}$ $p^{(3)}$ $+x_3a$	1	1 0 1 0 1 1 0 1 1 1 1 1	0 0 1 0	210
$2p^{(4)}$ $p^{(4)}$	1	1 1 0 0 1 1 1 0	0 0 0 1	225

Рис. 1.23. Принцип умножения методом правого сдвига и сложения. Умножение 15x15

Поскольку принципы умножения чисел со знаком и без знака отличаются, то необходимо откорректировать код языка VHDL цифрового автомата (пример). В коде содержатся шесть операторов process которые, выполняются параллельно.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY avtomat IS
PORT(
Res,clk : IN      STD_LOGIC;
X,A: IN STD_LOGIC_VECTOR(4 downto 0);
Ena_Add, LoadPSC,ena_shift,Stop,sub_add, sign: OUT   STD_LOGIC;
Q_sub_add       : OUT STD_LOGIC_VECTOR(4 downto 0);
Q_stop      : OUT STD_LOGIC_VECTOR(4 downto 0));
END avtomat;
ARCHITECTURE a OF avtomat IS
```

```
TYPE state_values IS (SA, SB, SC, SD);
signal state,next_state: state_values;
SIGNAL cnt_sub_add: STD_LOGIC_VECTOR(4 downto 0);
SIGNAL cnt_stop: STD_LOGIC_VECTOR(4 downto 0);
BEGIN
statereg: process(clk,Res)
begin
        if (Res = '1') then state<=SA;
                elsif (clk'event and clk='1') then
                                state<=next_state;
        end if;
end process statereg;
process(state)
begin
case state is
when SA=> next_state<=SB;
when SB=> next_state<=SC;
when SC=>  next_state<=SD;
when SD=> next_state<=SA;
end case;
end process;
process (state)
begin
case state is
when SA=>Ena_Add<='0';
            LoadPSC<='0';
            ena_shift<='1';
when SB=>
                Ena_Add<='1';
                LoadPSC<='0';
                ena_shift<='0';
when SC=>
                Ena_Add<='0';
                LoadPSC<='0';
                ena_shift<='0';
when SD=>
                Ena_Add<='0';
                LoadPSC<='1';
```

```
                    ena_shift<='1';
end case;
end process;
process (clk, res)
          begin
                    if (res = '1') then cnt_sub_add <=(others=>'0');
                    elsif (clk'event and clk = '0') then
                    cnt_sub_add <= cnt_sub_add+'1';
                    end if;
end process;
         Q_sub_add <= cnt_sub_add;
process(cnt_sub_add,X,A)
                    begin
                    -- Случай 4
                    if A(4)='0' and X(4)='1' then
                    sign<='1'; sub_add<='0';
                    case cnt_sub_add is
                    when "10001" =>
sign<='0'; sub_add<='1';
                    when "10010" =>
                    sign<='1'; sub_add<='1';
                    when others => sign<='1'; sub_add<='0';
                    end case;
                    -- Для случаев 1,2 и 3
elsif A(4)='1' then
case cnt_sub_add is
when "00010" =>
if (X(0)='0' ) then sign<='1'; sub_add<='0'; else
sign<='0'; sub_add<='0'; end if;
when "00110" =>
if (X(1)='0' and X(0)='0') then sign<='1'; sub_add<='0'; else sign<='0';
sub_add<='0'; end if;
when "01010" =>
if (X(2)='0' and X(1)='0' and X(0)='0') then sign<='1'; sub_add<='0'; else
sign<='0'; sub_add<='0'; end if;
when "01110" =>
if (X(3)='0' and X(2)='0' and X(1)='0' and X(0)='0') then sign<='1';
sub_add<='0'; else sign<='0'; sub_add<='0'; end if;
```

```
when "10001" =>
--2S complement else no 2S complement
if (X(4)='1') then sign<='1'; sub_add<='1'; else sub_add<='0';sign<='0';end
if;
when "10010" =>
----2S complement and 2p[5]=0 else no 2S complement and 2p[5]=1
if (X(4)='1')
or (X(4)='0' and X(3)='0' and X(2)='0' and X(1)='0' and X(0)='0')
then sign<='1'; sub_add<='0'; else sub_add<='0';sign<='0'; end if;
when others => sign<='0';
                    sub_add<='0';
end case;
end if;
end process;
process (clk, res)
        begin
                if (res = '1') then cnt_stop <=(others=>'0');
                elsif (clk'event and clk = '1') then
                if cnt_stop = "10101" then Stop<='1';
                else cnt_stop <= cnt_stop+'1';
                end if;
                end if;
end process;
       Q_stop <= cnt_stop;
END a;
```

Пример. Код языка VHDL-цифрового автомата умножителя методом правого сдвига и сложения

Первые три оператора process реализуют цифровой автомат Мура на четыре состояния с логикой переходов и логикой формирования выхода для управления процессом умножения. Память состояний (регистр состояний) автомата тактируется фронтом синхроимпульса. Надо принимать во внимание, что на синхровход автомата clk подключен инвертор. Автомат формирует три управляющих сигнала ena_add_temp (сигнал разрешения суммирования многоразрядному сумматору масштабирующего

аккумулятора), load_acc (сигнал разрешения загрузки в регистр со сдвигом вправо) и ena_shift_temp (сигнал разрешения сдвига) (рис. 1.24).

Четвертый оператор process представляет собой 5-разрядный суммирующий счетчик на сигнале cnt_sub_add (тактируется срезом синхроимпульса clk'event and clk = '0'). Таким образом удается обеспечить конвейерный режим работы масштабирующего аккумулятора, при этом формируемые сигналы ena_add_temp, load_acc и ena_shift_temp являются синхронными для всех регистров умножителя. Так, второй передний фронт синхроимпульса оказывается ровно над половиной высокого уровня сигнал ena_add_temp. Что обеспечивает корректную работу синхронного многоразрядного сумматора масштабирующего аккумулятора.

Согласно схемам на рис. 1.20-1.23 удвоенные частичные произведения 2p(1), 2p(2), 2p(3), 2p(4) и 2p(5) будут получены на 2, 6, 10 и 14 тактах сигнала cnt_sub_add.

Пятый оператор process, в списке чувствительности которого стоит сигнал cnt_sub_add, используется как дешифратор случаев 4 и 1, 2 и 3. На сигнале cnt_sub_add осуществляется подсчет синхроимпульсов.

Рассмотрим подробно случаи 1, 2 и 3 when "00010", "00110", "01010". Приведем примеры для множителя X: X=*XXXX*0 (например, 11110 BIN или -2D), X=*XXX*00 (например, 11100 BIN или -4D или же число со знаком -12, (10100 BIN), где четвертый разряд нулевой), X=*XX*000 (11000 BIN или -8D). *X*- или логическая 1 или логический 0. В случае обнаружения этих чисел цифровой автомат сформирует два сигнала sign<='1' и sub_add<='0', для того чтобы в поле MSB (2p[5]) установился логический 0. В этих случаях умножение идет согласно принципу, показанному на рис. 1.21.

Существует еще случай when "10001" когда будет подсчитан 17-й такт синхроимпульса, при достижении которого формируется двоичное дополнение. Это распространяется на умножение двух четных отрицательных

чисел и когда оба числа оказываются отрицательными, одно из которых может быть четное, а другое нечетное и на оборот.

Сигнал sub_add используется для подачи логической 1 на вход переноса многоразрядного сумматора Cin масштабирующего аккумулятора, в случае обнаружения 1 в старшем разряде X(4)=1, а также для селективной инверсии числа A при переводе его в обратный код.

Когда будет подсчитан 18-й такт синхросигнала (when "10010") цифровой автомат сформирует сигналы sign<='1' и sub_add<='0' и в поле MSB будет установлен логический 0.

Шестой оператор process реализует схему останова процесса умножения на суммирующем счетчике тактируемым срезом синхроимпульса. При достижение 21-го синхроимпульса вырабатывается сигнал останова Stop.

На рис. 1.24 и рис. 1.25 овалами под номерами один, два и три обведена дополнительная логика, обеспечивающая умножение чисел как со знаком, так и без. На рис. 1.24 показан верхний уровень иерархии проекта универсального умножителя двух чисел в дополнительном коде как со знаком, так и без в САПР ПЛИС Quatus II Web Edition 13.0.1sp1 (сборка 232). В отличие от схемы умножения беззнаковых чисел в схему введена дополнительная проверка на знак. Если сигнал gg = 1, то множимое A и множитель X - беззнаковые числа (выделен овалом под номером один). Блок complementer обеспечивает формирование обратного кода. Режим переключения между без- и знаковыми числами обеспечивает мультиплексор mux21 (выделен овалом под номером два), на адресный вход которого подается сигнал gg. В целом принцип работы масштабирующего аккумулятора не отличается от аккумулятора для беззнаковых чисел за исключением увеличения разрядности всех блоков на 1 бит.

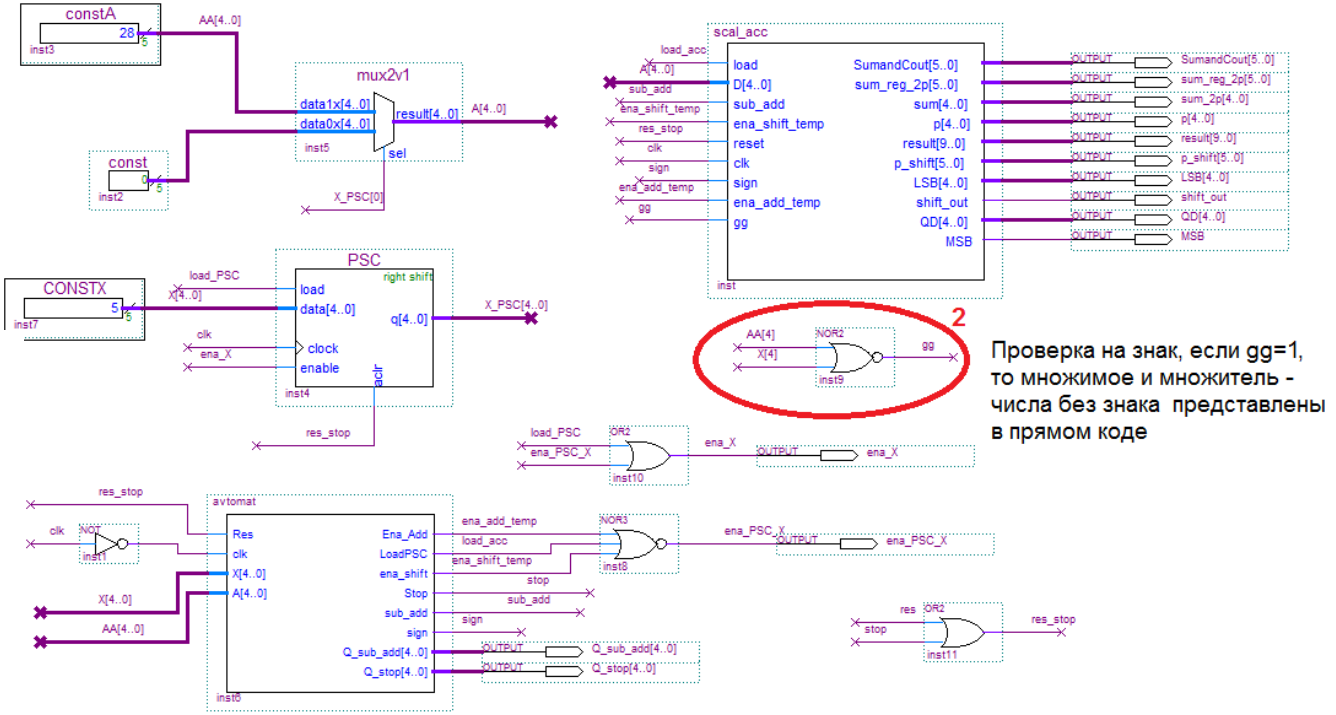
Поле MSB (рис. 1.25) формируется с помощью двух элементов “исключающее ИЛИ (XOR)”. Вспомогательная схема выделена на рис. 1.25 овалом под номером три. Один из входов элемента XOR (inst10) подключен к напряжению

питания. Это необходимо для заполнения поля MSB логической 1 или логическим 0, как требуют случаи 1-4.

На рис. 1.26 и 1.27 показаны случаи 1 и 2, а на рис. 1.28 случай 4. В случае 1 в поле MSB в процессе умножения сохраняется логическая 1. В случае 2 в поле MSB при вычислении удвоенного частичного произведения 2p(5) прописывается логический 0 по 18-му такту синхроимпульса. В случае 4 в поле MSB все происходит с точностью наоборот.

На рис. 1.29 показаны тестовые схемы для умножения -10x-11 для реализации в ПЛИС серии Cyclone II. На вход coeff_in[4..0] мегафункции ALTMEMMULT подключается константа -10 (22D). Такое же значение подключается и на вход AA[4..0] разработанного MAC-блока (рис. 1.29). На информационный вход data_in[4..0] мегафункции ALTMEMMULT и на вход X[4..0] MAC-блока подается число -11. На рис. 1.30 показан процесс умножения -10x-11 MAC-блоком и мегафункцией для случая, когда константа загружается с внешнего порта. В мегафункцию предварительно загружена константа (режим загрузки из блочной памяти ПЛИС) число 3.

Итак, в данном разделе разработан MAC-блок с использованием метода правого сдвига и сложения для умножения чисел, представленных в дополнительном коде как четных, так и нечетных, со знаком и без для реализации в базисе ПЛИС. Предложенная схема реализации MAC-блока умножает за 21 такт синхрочастоты и может быть использована при проектировании КИХ-фильтров.



Проверка на знак, если gg=1, то множимое и множитель - числа без знака представлены в прямом коде

Рис. 1.24. Схема умножителя в САПР ПЛИС Quatus II. Верхний уровень иерархии

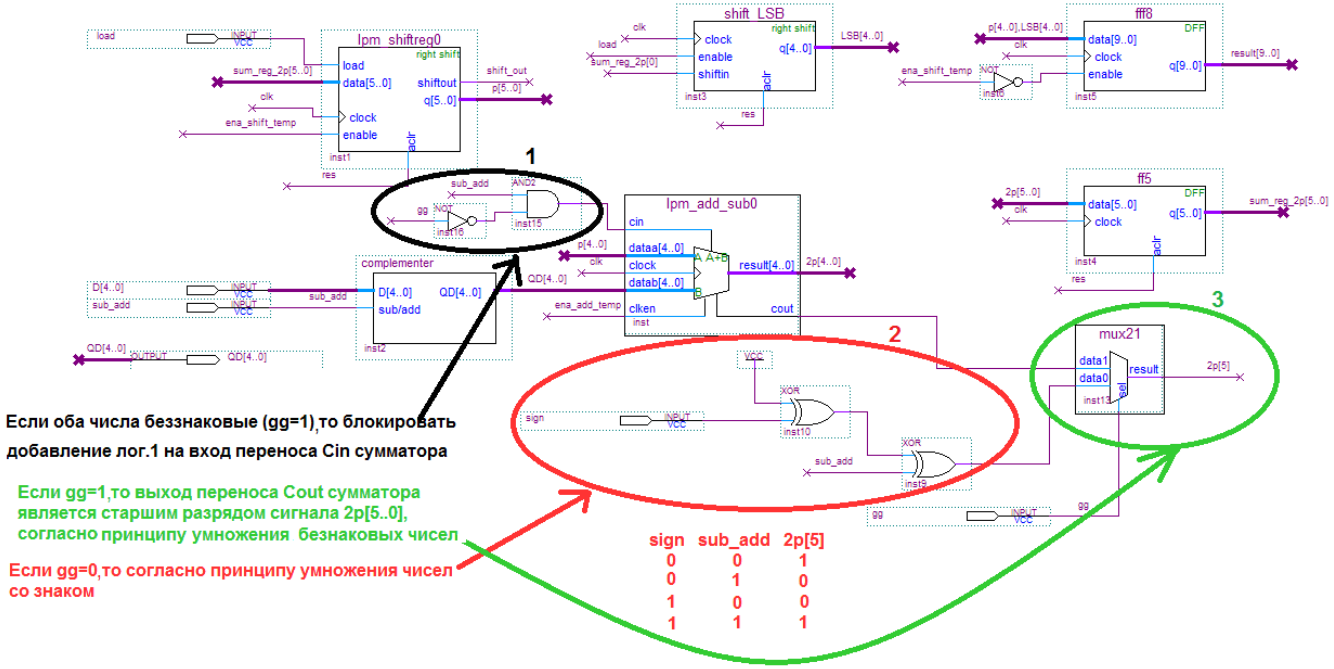


Рис. 1.25. Схема масштабирующего аккумулятора

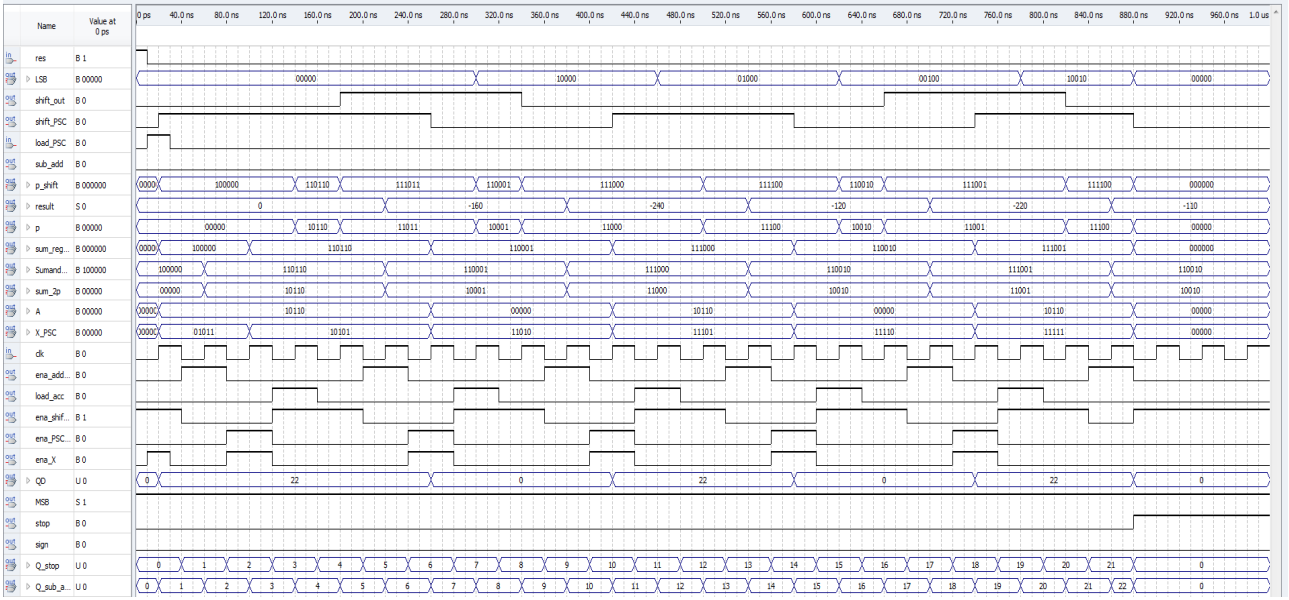


Рис. 1.26. Временные диаграммы процесса умножения -10x11. Результат -110

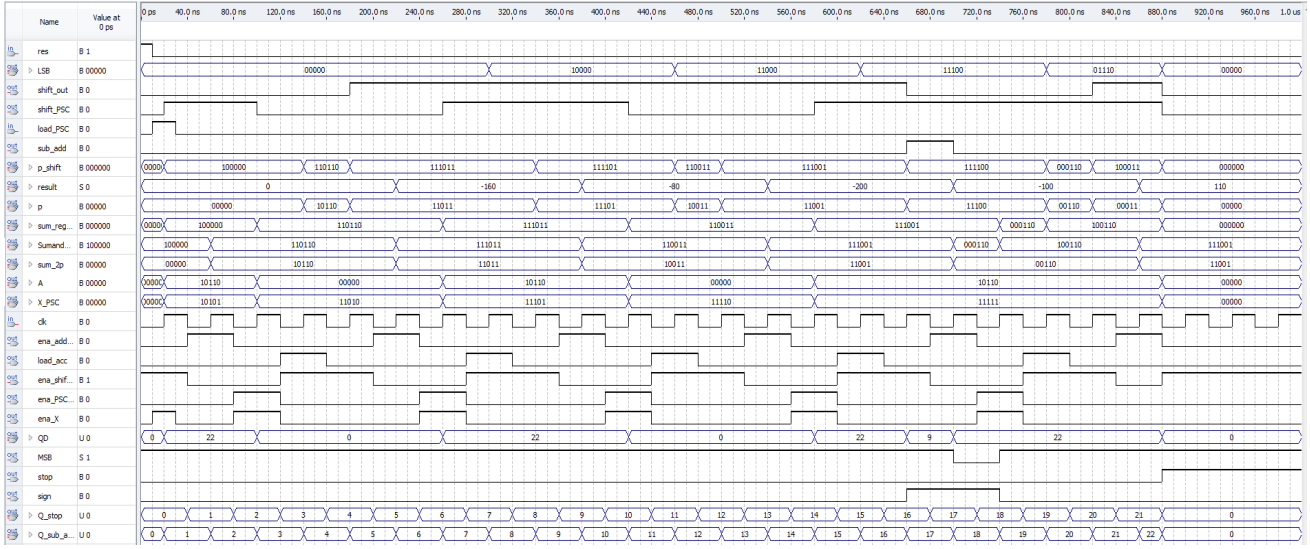


Рис. 1.27. Временные диаграммы процесса умножения -10x-11. Результат 110

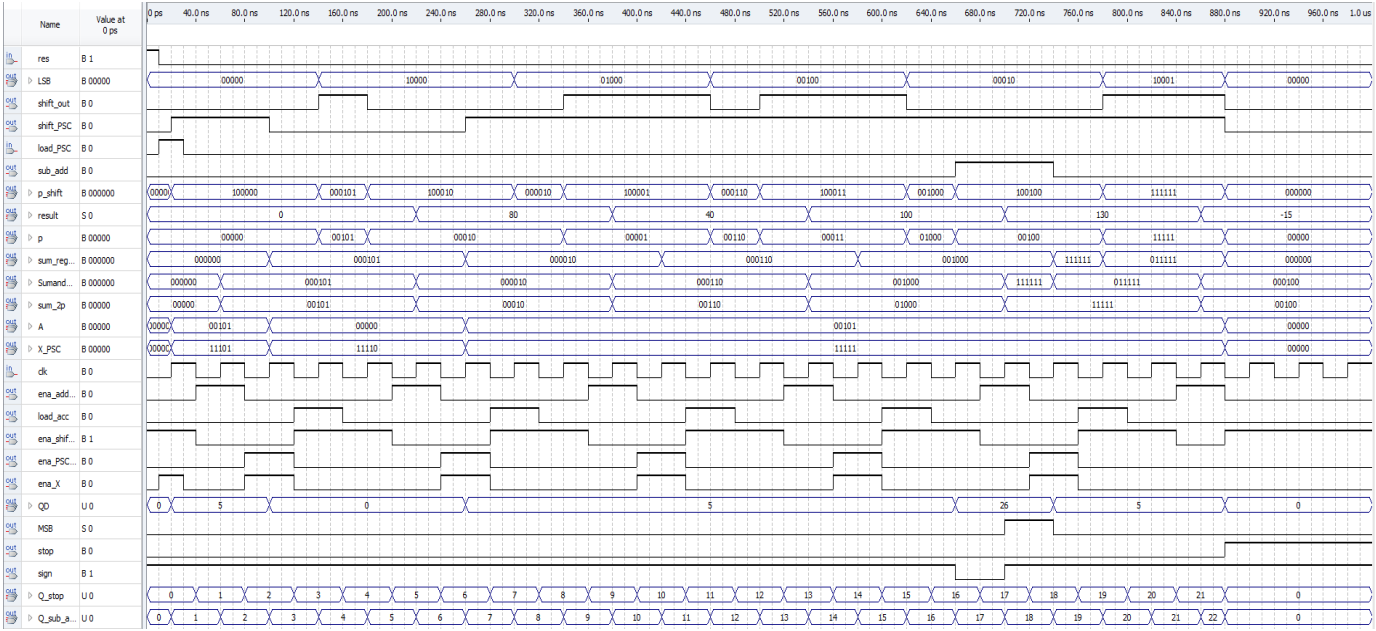


Рис. 1.28. Временные диаграммы процесса умножения 5х-3. Результат -15

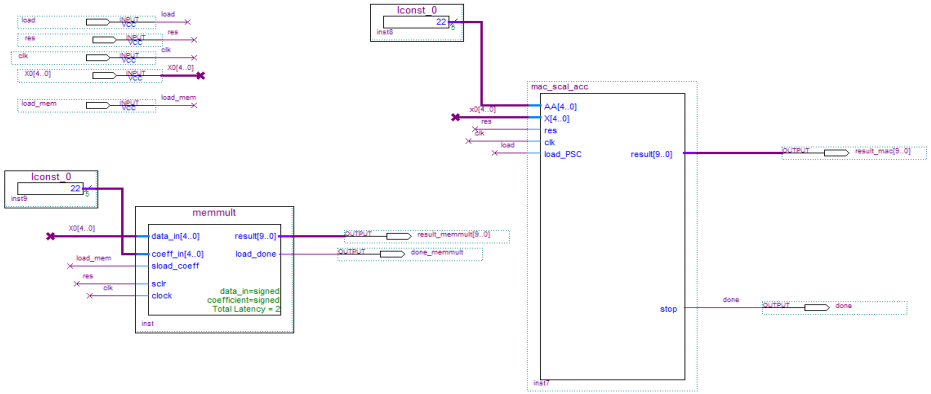


Рис. 1.29. Тестирование мегафункции ALTMEMMULT и разработанного MAC-блока

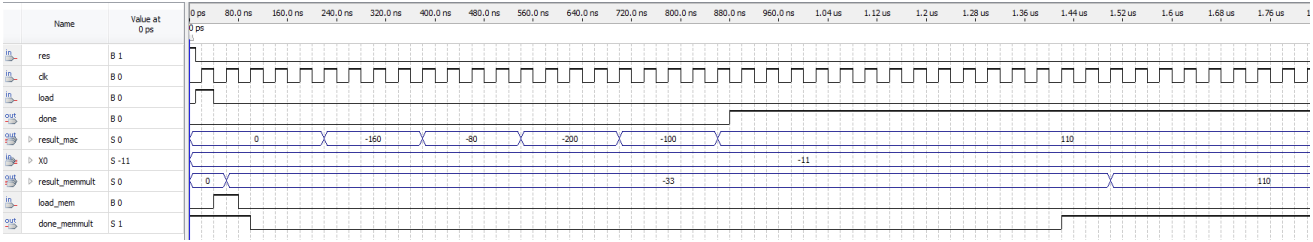


Рис. 1.30. Временные диаграммы процесса умножения -10х-11 MAC-блоком и мегафункцией ALTMEMMULT. Результат 110

1.6. Общие сведения по программным умножителям в базисе ПЛИС

В ПЛИС для повышения их функциональных возможностей встраивают, например, для серии Cyclone III фирмы Altera аппаратные умножители, которые могут быть сконфигурированы в виде одного умножителя 18x18 либо в виде двух умножителей 9x9. Так, ПЛИС EP3CLS200 содержит 396 аппаратных умножителей 18x18, а на оставшихся ресурсах может быть реализован 891 программный умножитель 16x16. В итоге суммарное число умножителей составляет 1287 без какого-либо значительного использования логических ресурсов.

Для устройств цифровой обработки сигналов себя хорошо зарекомендовали софт-умножители (программные умножители), которые не требуют ресурсов аппаратных умножителей, встроенных в базис ПЛИС. Повысить производительность устройств цифровой обработки сигналов позволяет также использование параллельного векторного умножителя и “безумножительных” схем умножения с использованием основ распределенной арифметики.

Рассмотрим параллельные программные умножители, способные вычислять произведение за один такт синхроимпульса, обеспечивая наивысшую производительность устройств цифровой обработки сигналов. Программные умножители БИС программируемой логики (БИС ПЛ) фирмы Actel серий Fusion, IGLOO и ProASIC3 реализуются на блочной памяти меньшей размерности, чем у ПЛИС фирмы Altera и их можно рассматривать как 8-входовые LUT или таблицы произведений. Таблица произведений множимого, записанная во фрагмент блочной памяти, и называется LUT. Табл. 1.2 показывает умножитель размерностью 3x3, реализованный с помощью 6-входовой LUT.

Таблица 1.2

Умножитель размерностью 3х3, реализованный с помощью 6-входовой LUT

Множитель \ Множимое	000	001	010	011	100	101	110	111
000	000000	000000	000000	000000	000000	000000	000000	000000
001	000000	000001	000010	000011	000100	000101	000110	000111
010	000000	000010	000100	000110	001000	001010	001100	001110
011	000000	000011	000110	001001	001100	001111	010010	010101
100	000000	000100	001000	001100	010000	010100	011000	011100
101	000000	000101	001010	001111	010100	011001	011110	100011
110	000000	000110	001100	010010	011000	011110	100100	101010
111	000000	000111	001110	010101	011100	100011	101010	110001

Например, у ПЛИС фирмы Actel используется ОЗУ емкостью 256 слов х 8 бит (256 8-разрядных слов), а у ПЛИС фирмы Altera может использоваться память M4K, которая может быть сконфигурирована, как 128 слов х 36 бит или 256 слов х 18 бит для серии Cyclone II. Такие умножители получили название RAM-LUT-умножители или LUT-based умножители.

На рис. 1.31 показан умножитель 4х4 на базе синхронного ОЗУ емкостью 256 8-разрядных слов. Множимое (младшие четыре разряда адресной шины) и множитель (старшие четыре разряда адресной шины), представленные 4-разрядным двоичным кодом, объединяются в 8-разрядную адресную шину, адресуя своим уникальным кодом содержимое конкретной строки ОЗУ (операнды), являющееся 8-ми разрядным произведением.

Недостатком такого умножителя является резкое возрастание требуемого объема блочной памяти в случае увеличения его разрядности. Для умножителя размерностью 8х8 требуется 65536 16-разрядных слов. Поэтому чтобы предотвратить рост требуемой памяти на практике используется умножитель на суммировании частичных

произведений в соответствии со своим весом (partial product multipliers).

На рис. 1.32 показан пример умножения десятичного числа 24 на 43. Например, произведению 2 на 4 приписывается вес 100, что равносильно сдвигу на две позиции в десятичной системе. В этом случае необходим умножитель размерностью 8x8. Однако согласно принципу умножения с использованием частичных произведений требуются четыре умножителя размерностью 4x4 для формирования четырех частичных произведений и три сумматора: (4x3+((2x3)x10))+((4x4)+((2x4)x10)x10)=1032. На рис. 1.33 показана структурная схема такого умножителя. Так же для сдвига на одну и две десятичные позиции потребуются три сдвиговых регистра на четыре разряда влево и дополнительные блоки, выполняющие операции расширения знака со значением старшего разряда.

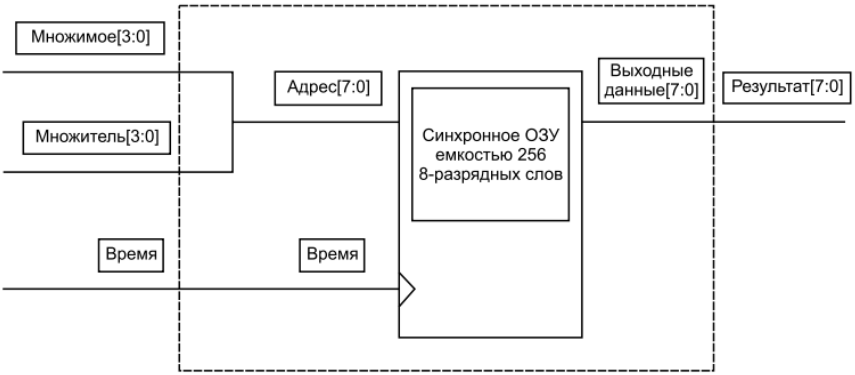


Рис. 1.31. Программный умножитель размерностью 4x4 на базе ОЗУ емкостью 256 8-разрядных слов (256x8) фирмы Actel

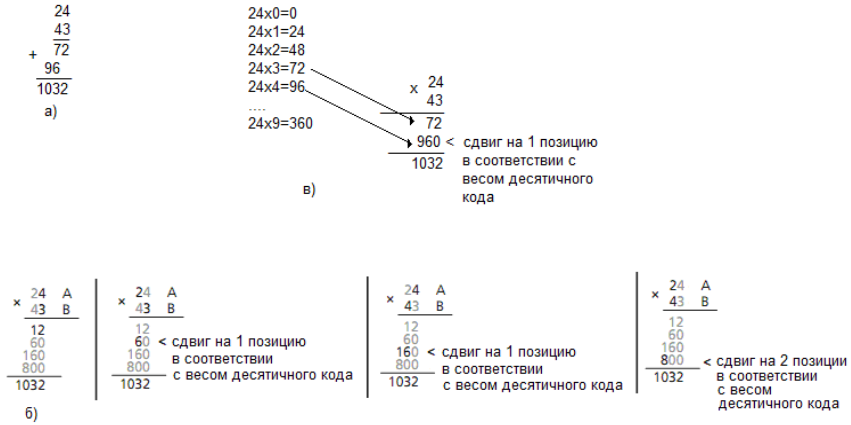


Рис. 1.32. Принцип умножения: а) “в столбик” по правилу умножения десятичных чисел; б) с использованием частичных произведений; в) умножение на константу

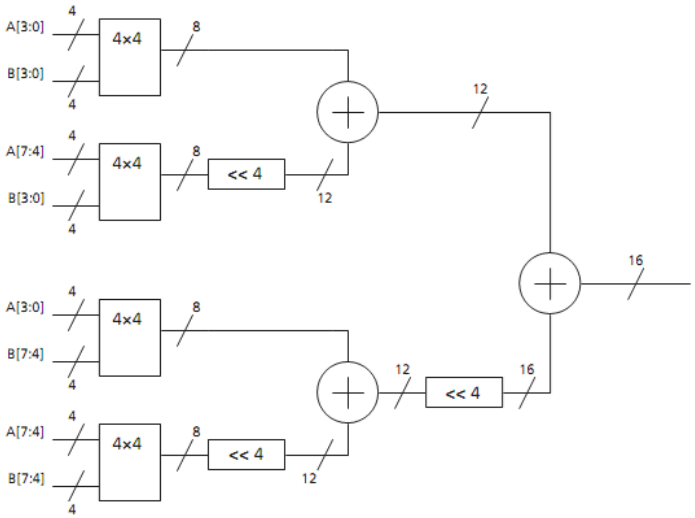


Рис. 1.33. Структурная схема умножителя размерностью 8x8 с использованием четырех умножителей размерностью 4x4

Рассмотрим программные умножители на константу. Одна из наиболее распространенных операций цифровой обработки сигналов - умножение числа на константу. Для

перемножения двух чисел достаточно иметь таблицу произведений множимого (константы) на весь ранг возможных цифр множителя (табл. 1.3) и осуществить корректное суммирование полученных частичных произведений (рис. 1.32, в).

Таблица 1.3

Умножение 4-разрядного числа на константу 24 (рис. 1.32, в)

Входы			X[3]	
X[2]	X[1]	X[0]	0	1
000			0	192
001			24	216
010			48	240
011			72	264
100			96	288
101			120	312
110			144	336
111			168	360

Программные умножители фирмы Actel реализуются на 8-входовых LUT. Множимое (константа) в этом случае предопределено. В этом случае необходимы два блока памяти емкостью 256 8-разрядных слов позволяющих организовать массив памяти 256 16-разрядных слов из двух блоков емкостью 256 8-разрядных слов, выходная шина которого и есть 16-разрядный результат умножения двух 8-разрядных чисел (рис. 1.34).

Программные умножители фирмы Altera. Наличие встроенной блочной памяти TriMatrix™ в ПЛИС фирмы Altera, например, типа M9K используемой в качестве LUT, в которых хранятся частичные произведения, позволяет реализовывать параллельные умножители, экономя при этом не только аппаратные умножители, но и ресурсы логических

блоков. Наличие программных и аппаратных умножителей приводит в целом к увеличению общего числа возможных умножителей.

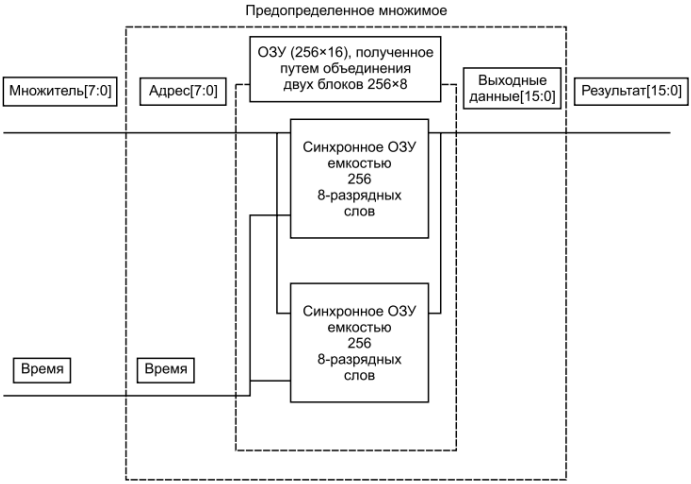


Рис. 1.34. Программный умножитель размерностью 8x8 числа на константу на базе ОЗУ емкостью 256 16-разрядных слов фирмы Actel

Использовать программные и аппаратные умножители в проекте пользователя возможно через мегафункции. Мегафункции lpm_mult, altmult_add и altmult_accum позволяют использовать аппаратные умножители. Рассмотрим мегафункцию ALTMEMMULT – программный умножитель. Мегафункция ALTMEMMULT позволяет осуществлять процесс умножения числа на константу С, при этом константа может храниться в блочной памяти ПЛИС либо загружается с внешнего порта.

Для ПЛИС серии Cyclone II возможно использовать только память M4K (128x36 бит) или режим Avto. Например, ПЛИС EP2C70 содержит 250 блоков M4K. На рис. 1.35 показана идея умножения числа на константу. В целом,

принцип умножения, показанный на рис. 1.35 не отличается от ранее рассмотренного. При этом число и константа могут быть как со знаком, так и без него. На рис. 1.36 показан принцип построения программного умножителя 16-разрядного числа на 10-разрядную константу (обозначена буквой С) с использованием блоков памяти типа M4K большего размера, чем у БИС ПЛ фирмы Actel для случая, когда константы хранятся в блочной памяти, т.е. отсутствует возможность их загрузки из вне (отсутствуют адресные порты для загрузки коэффициентов).

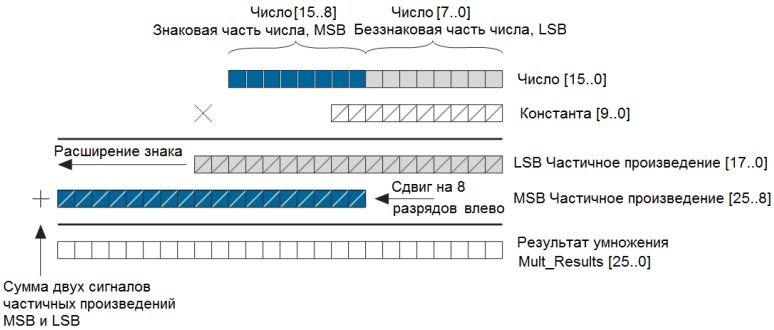


Рис. 1.35. Идея параллельного умножения 16-разрядного числа на 10-разрядную константу

Входной 16-разрядный сигнал разделяется на два 8-разрядных сигнала с именами LSB (младший значащий разряд) и MSB (старший значащий разряд). Сигнал LSB адресуется к блоку памяти M4K с одноименным названием LSB, а сигнал MSB адресуется к блоку памяти M4K с одноименным названием MSB. В блочной памяти LSB хранятся 256 предварительно вычисленных частичных произведений с именем “LSB Частичное произведение [17..0]” (младшее частичное произведение) разрядностью 18 бит с диапазоном от 0 до 255xC, а в памяти MSB с диапазоном от 0 до (-1)xC хранятся 256 предварительно вычисленных

частичных произведений с именем “MSB Частичное произведение [25..8]” (старшее частичное произведение). Далее старшее частичное произведение необходимо сдвинуть на восемь разрядов влево, а затем осуществить сквозное суммирование.

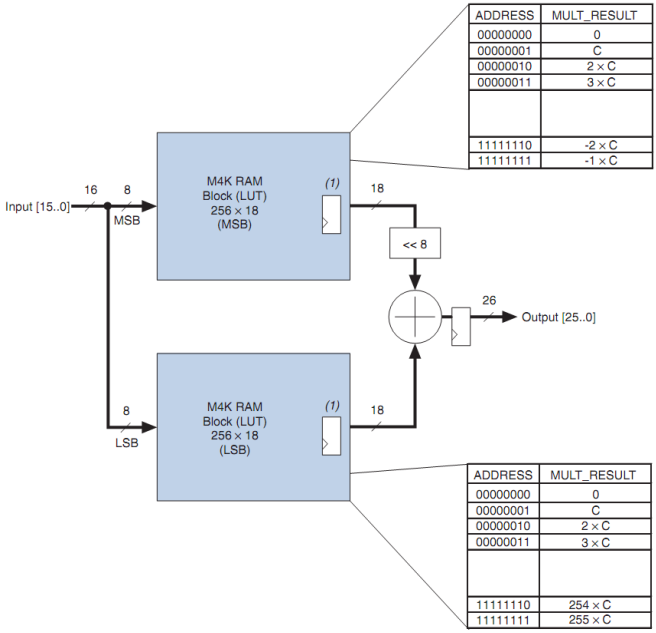


Рис. 1.36. Программный параллельный умножитель 16-разрядного числа на 10-разрядную константу размерностью 16x10 с использованием двух блоков памяти типа M4K в качестве LUT фирмы Altera

Умножение фактически осуществляется за один такт синхроимпульса, необходимый для загрузки входных значений сигналов LSB и MSB в адресные порты блоков памяти. Еще два такта требуются для конвейеризации задержки вычислений, т.к. выходные значения блоков памяти, представляющие собой частичные произведения, должны

быть еще просуммированы с соответствующими весами для получения 26-разрядного результата умножения.

В качестве примера на рис. 1.37 показаны настройки мегафункции ALTMEMMULT для умножения 4-разрядного числа, представленного дополнительным кодом, и 4-разрядной константы, загружаемой из внешнего порта. В этом случае требуется 20 LUT логических блоков плюс 1 блок памяти типа M4K и 20 триггеров (20 lut+1M4K+20 reg). В случае загрузки константы из блочной памяти требуется всего лишь 1M4K.

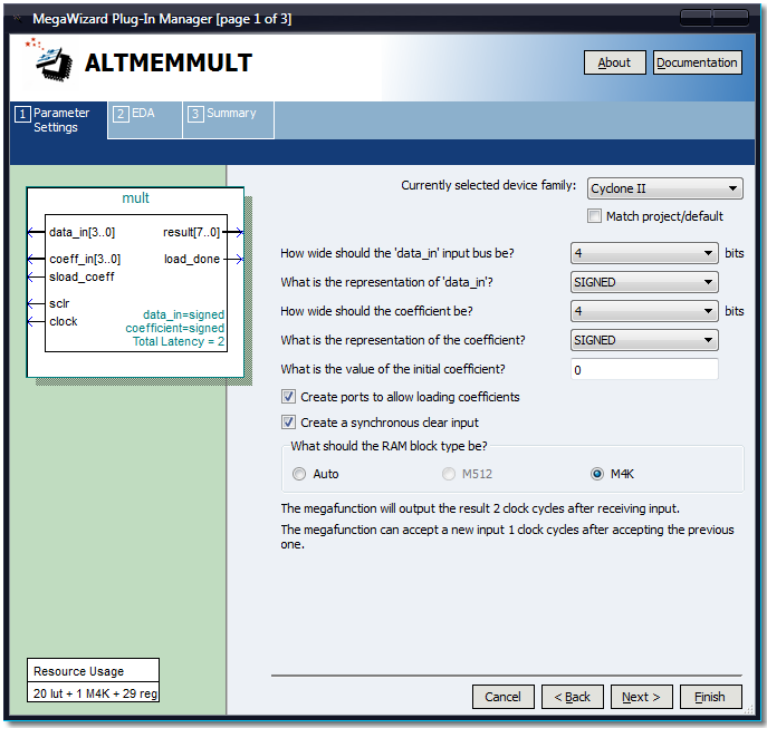
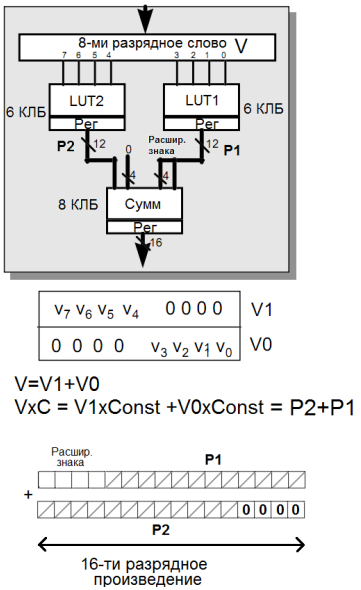


Рис. 1.37. Мегафункция ALTMEMMULT, настроенная для реализации программного умножителя 4x4

Принцип построения на рис. 1.36 не раскрывает все тонкости такого умножителя. В частности, не показана операция расширения знака числа. На рис. 1.38 показан принцип построения программного умножителя на константу размерностью 8x8 с использованием двух 4-входовых LUT ПЛИС серии XC4000. На рис. 1.38 обозначено: V - входной 8-разрядный сигнал; P1 и P2 - младшее и старшее частичные произведения; C - константа. В частности, показано, как на практике осуществляется сдвиг на четыре разряда влево. Для умножителя требуются 25 конфигурируемых логических блоков (КЛБ). Объединяя такие умножители в секции (одна секция на отвод фильтра), можно построить высокопроизводительный параллельный КИХ-фильтр, работающий на частотах 50-70 МГц.



ADRS		P2 LUT2	P1 LUT1
0	0000	0	0
1	0001	1 x Const	1 x Const
2	0010	2 x Const	2 x Const
3	0011	3 x Const	3 x Const
4	0100	4 x Const	4 x Const
5	0101	5 x Const	5 x Const
6	0110	6 x Const	6 x Const
7	0111	7 x Const	7 x Const
8	1000	-8 x Const	8 x Const
9	1001	-7 x Const	9 x Const
A	1010	-6 x Const	A x Const
B	1011	-5 x Const	B x Const
C	1100	-4 x Const	C x Const
D	1101	-3 x Const	D x Const
E	1110	-2 x Const	E x Const
F	1111	-1 x Const	F x Const
SIGN BIT		16 старших частичных произведений разрядностью 12 бит	16 младших частичных произведений разрядностью 12 бит

Рис. 1.38. Программный умножитель на константу размерностью 8x8 с использованием двух 4-входовых LUT ПЛИС серии XC4000

1.7. Разработка проекта умножителя размерностью 4x4 в базисе ПЛИС типа ППВМ серии Cyclone фирмы Altera с помощью учебного лабораторного стенда LESO2.1

В разделе 1.3 рассмотрено проектирование умножителя целых положительных чисел, представленных в прямом коде, размерностью 4x4 методом правого сдвига и сложения (MAC-блок), а в разделе 1.4 - проектирование умножителя целых чисел со знаком, представленных в дополнительном коде. В обоих случаях управляющие автоматы являлись оригинальными и были разработанны с использованием языка VHDL.

Рассмотрим проектирование цифрового автомата более простым способом - методом умножения в столбик. Управляющий автомат умножителя разработаем с помощью редактора состояний САПР Quartus II (State Machine Viewer). Далее реализуем умножитель размерностью 4x4 в базисе ПЛИС типа ППВМ серии Cyclone EP1C3T144C8N фирмы Altera с помощью учебного лабораторного стенда LESO2.1 (Лаборатории электронных средств обучения, ЛЭСО ГОУ ВПО «СибГУТИ») отечественной разработки. Учебный лабораторный стенд предназначен для обучения основам проектирования цифровой техники на основе ПЛИС.

Так как САПР Quartus II Web Edition version 13.0.1 сборка 232 не поддерживает ПЛИС серии Cyclone, то необходимо перейти на более раннюю версию Quartus II Web Edition version 9.1.

На рис. 1.39 и рис. 1.40 показаны верхний и нижний уровни иерархии проекта умножителя (P=B(множимое)*A(множитель)) размерностью 4x4. Сигнал A (множитель) следует рассматривать как число, а сигнал B - как

константу (множимое). Умножитель настроен на умножение двух чисел 10x10. Умножитель состоит из двух однотипных регистров ShiftN, сдвигающих влево или вправо в зависимости от сигнала DIR задающего направление сдвига (пример 1), детектора нуля AllZero, управляющего автомата avt на пять состояний (пример 2), 8-разрядного сумматора на мегафункции lpm_add_sub, шинного мультиплексора на мегафункции lpm_mux и 8-разрядного регистра на мегафункции lpm_dff, выполняющего роль аккумулятора. Один из регистров ShiftN (DIR=0), на вход которого подается число A, работает как преобразователь параллельного кода в последовательный, параллельный выход SRA[7..0] нужен лишь для детектирования нуля.

Рис. 1.41 демонстрирует принцип работы управляющего автомата. Автомат принимает пять состояний с именами Check_FS, Init_FS, Adder_FS, shift_FS, End_mult. В каждом из состояний активным является один из сигналов Init, Add, Shift и Done. Автомат разработан по “классической” схеме с использованием одного оператора Process (однопроцессный шаблон) для описания памяти состояний и логики переходов.

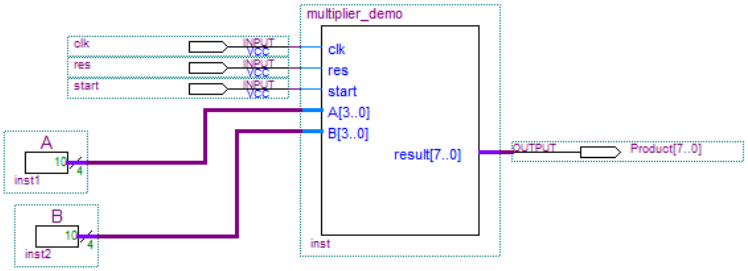


Рис. 1.39. Умножитель размерностью 4x4. Верхний уровень иерархии

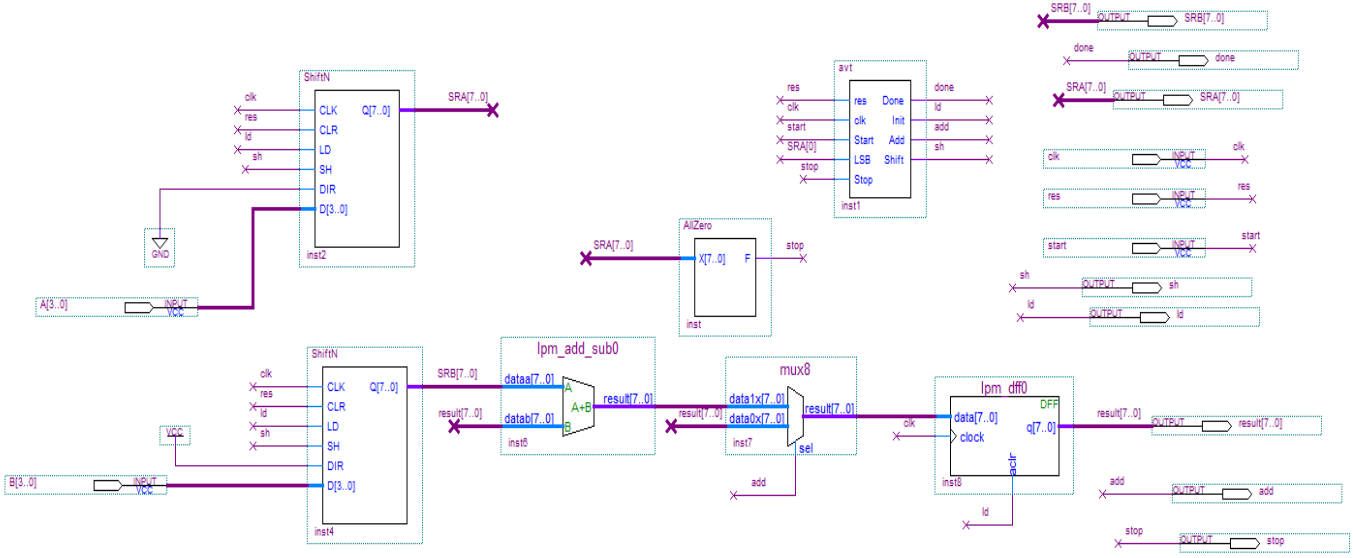


Рис. 1.40. Умножитель размерностью 4x4. Нижний уровень иерархии

Автомат инициализируется высоким уровнем сигнала Start синхронизируемого синхросигналом clk, переводящим выход Init в активное состояние. При высоком уровне сигнала Init происходит загрузка обоих сдвиговых регистров параллельным кодом. Если на вход LSB все время будет поступать логическая 1 (младший разряд SRA[0] 8-разрядного сигнала SRA[7..0]) с выхода сдвигового регистра ShiftN при DIR=0, то управляющий автомат будет вырабатывать сигналы “сложить” (Add) и “сдвинуть” (Shift). Это возможно, например, при загрузке числа 15D (1111BIN). На рис. 1.42 показан пример умножения чисел 10x10. Результат 100. По окончании процесса умножения вырабатывается сигнал готовности Done.

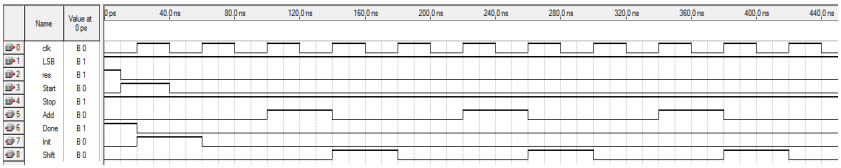


Рис. 1.41. Тест цифрового автомата

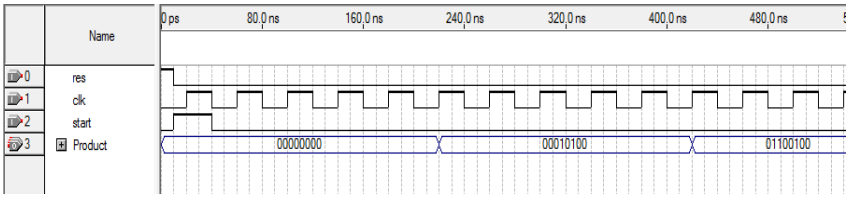


Рис. 1.42. Тестирование умножителя на примере умножения 10x10. Результат 100

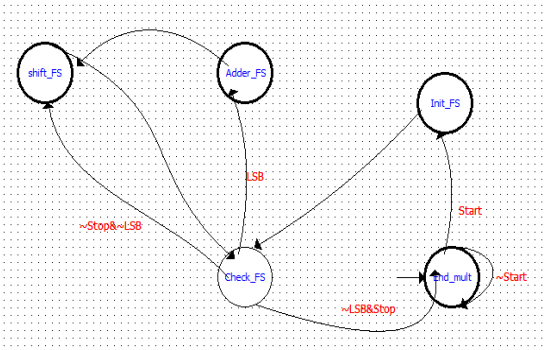
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
entity ShiftN is
port(CLK, CLR, LD, SH, DIR: in STD_LOGIC;
D: in std_logic_vector(3 downto 0);
Q: out std_logic_vector(7 downto 0));
end ShiftN;
architecture a of ShiftN is
  begin
  process (CLR, CLK)
  variable St: std_logic_vector(7 downto 0);
  subtype InB is natural range 3 downto 0;
      begin
          if CLR = '1' then
              St := (others => '0'); Q <= St;
          elsif CLK'EVENT and CLK='1' then
              if LD = '1' then
                  St:=(others=>'0');
                  St(InB) := D;
                  Q <= St;
              elsif SH = '1' then
                  case DIR is
                  when '0' => St := '0' & St(St'LEFT downto 1);
                  when '1' => St := St(St'LEFT-1 downto 0) &
'0';
                  end case;
                  Q <= St;
              end if;
          end if;
      end process;
end a;
```

Пример 1. Сдвиговый регистр на языке VHDL

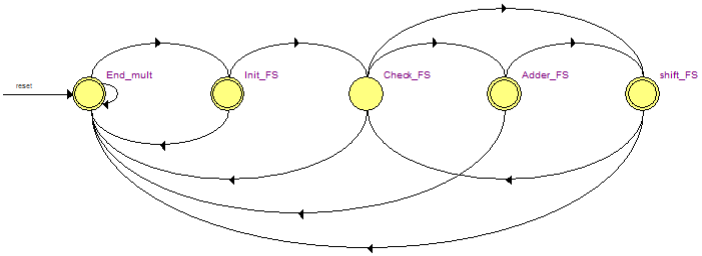
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY avt IS
   PORT (
      res : IN STD_LOGIC;
      clk : IN STD_LOGIC;
      Start : IN STD_LOGIC;
      LSB : IN STD_LOGIC;
      Stop : IN STD_LOGIC;
      Done : OUT STD_LOGIC;
      Init : OUT STD_LOGIC;
      Add : OUT STD_LOGIC;
      Shift : OUT STD_LOGIC);
END avt;
ARCHITECTURE BEHAVIOR OF avt IS
TYPE type_fstate IS (Check_FS,Init_FS,Adder_FS,shift_FS,End_mult);
   SIGNAL fstate : type_fstate;
   SIGNAL reg_fstate : type_fstate;
BEGIN
Init <='1' when reg_fstate = Init_FS else '0';
Add <='1' when reg_fstate = Adder_FS else '0';
Shift <='1' when reg_fstate = shift_FS else '0';
Done <='1' when reg_fstate = End_mult else '0';
process (clk, res) begin
if res = '1' then reg_fstate <= End_mult;
elsif clk'event and clk = '1' then
case reg_fstate is
when  Init_FS => reg_fstate <= Check_FS;
when Check_FS =>
                    if LSB = '1' then reg_fstate <= Adder_FS;
                    elsif Stop ='0' then reg_fstate <= shift_FS;
                    else reg_fstate <= End_mult;
                    end if;
when Adder_FS => reg_fstate <= shift_FS;
when shift_FS => reg_fstate <= Check_FS;
when End_mult => if Start = '1' then reg_fstate <= Init_FS; end if;
end case;
end if;
end process;
END BEHAVIOR;
```

Пример 2. Код языка VHDL управляющего автомата

Разработаем цифровой автомат с использованием встроенного редактора состояний конечного автомата (рис. 1.43) и извлечем код языка VHDL в автоматическом режиме. Используется двухпроцессный шаблон. Первый оператор Process описывает блок регистров (память состояний) для хранения состояний автомата. Второй оператор Process используется для описания логики переходов и логики формирования выхода (пример 3). Тестирование умножителя на примере умножения 5x5 показано на рис. 1.44. Общие сведения по числу задействованных ресурсов в проекте показаны в табл. 1.4.



а)



б)

Рис. 1.43. Граф-автомат, разработанный с помощью редактора состояний (а) и синтезированный граф-автомат (меню Netlist Viewers/State Machine Viewer)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY avt_flow IS
PORT (
reset : IN STD_LOGIC := '0';
clock : IN STD_LOGIC;
Start : IN STD_LOGIC := '0';
LSB : IN STD_LOGIC := '0';
Stop : IN STD_LOGIC := '0';
Done : OUT STD_LOGIC;
Init : OUT STD_LOGIC;
Add : OUT STD_LOGIC;
Shift : OUT STD_LOGIC
   );
END avt_flow;
ARCHITECTURE BEHAVIOR OF avt_flow IS
TYPE type_fstate IS (Check_FS,Init_FS,Adder_FS,shift_FS,End_mult);
   SIGNAL fstate : type_fstate;
   SIGNAL reg_fstate : type_fstate;
BEGIN
   PROCESS (clock,reg_fstate)
   BEGIN
      IF (clock='1' AND clock'event) THEN
         fstate <= reg_fstate;
      END IF;
   END PROCESS;
   PROCESS (fstate,reset,Start,LSB,Stop)
   BEGIN
      IF (reset='1') THEN
         reg_fstate <= End_mult;
         Done <= '0';
         Init <= '0';
         Add <= '0';
         Shift <= '0';
      ELSE
         Done <= '0';
         Init <= '0';
         Add <= '0';
         Shift <= '0';
```

```
CASE fstate IS
   WHEN Check_FS =>
      IF ((NOT((LSB = '1')) AND (Stop = '1'))) THEN
         reg_fstate <= End_mult;
      ELSIF ((LSB = '1')) THEN
         reg_fstate <= Adder_FS;
      ELSIF ((NOT((Stop = '1')) AND NOT((LSB = '1')))) THEN
         reg_fstate <= shift_FS;
      -- Inserting 'else' block to prevent latch inference
      ELSE
         reg_fstate <= Check_FS;
      END IF;
   WHEN Init_FS =>
      reg_fstate <= Check_FS;
      Init <= '1';
   WHEN Adder_FS =>
      reg_fstate <= shift_FS;
      Add <= '1';
   WHEN shift_FS =>
      reg_fstate <= Check_FS;
      Shift <= '1';
   WHEN End_mult =>
      IF ((Start = '1')) THEN
         reg_fstate <= Init_FS;
      ELSIF (NOT((Start = '1'))) THEN
         reg_fstate <= End_mult;
      -- Inserting 'else' block to prevent latch inference
      ELSE
         reg_fstate <= End_mult;
      END IF;
      Done <= '1';
```

```
            WHEN OTHERS =>
               Done <= 'X';
               Init <= 'X';
               Add <= 'X';
               Shift <= 'X';
               report "Reach undefined state";
         END CASE;
      END IF;
   END PROCESS;
END BEHAVIOR;
```

Пример 3. VHDL-код, извлеченный в автоматическом режиме из граф-автомата, созданного с помощью редактора состояний в САПР Quartus II

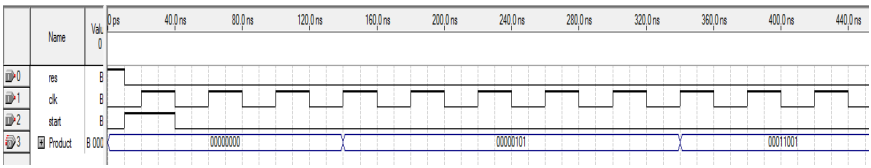


Рис. 1.44. Тестирование умножителя на примере умножения 5x5. Результат 25

Таблица 1.4

Общие сведения по числу задействованных ресурсов ПЛИС Cyclone EP1C3T144C8N

Логические элементы (Logic Cells, ЛЭ)	Триггеры логических элементов (LC Registers)	Таблицы перекодировок (LUT-only LC)	Рабочая частота в наихудшем случае Fmax, МГц
47	41	6	275

Стенд подключается к персональному компьютеру через порт USB. Для записи файла конфигурации в память ПЛИС через порт USB персонального компьютера требуется

преобразовать *.sof- файл в формат с расширением *.rbf. Загрузка конфигурационного файла в ПЛИС производится с помощью отдельной программы – загрузчика (l2flash.exe).

Входные и выходные контакты к внешним выводам ПЛИС подключены с помощью меню Assignments/Pins (рис. 1.45). Из-за того, что стенд имеет 8 переключателей S1-S8, пришлось отказаться от загрузки чисел с внешних портов (4-разрядные сигналы A и B) и от сигнала Done, так как доступно всего лишь 8 светодиодов. Умножаемые числа предварительно сохраняются в константах (мегафукция LPM_constant). Далее необходимо следовать рис. 1.44 и 1.45. Светодиоды LED1-LED8 отображают результат умножения (8-разрядный сигнал Product[7..0]). В проекте принято, что LED8 (pin 121) - младший значащий разряд.

Для подачи тактовых импульсов с помощью кнопки Bottom необходимо использовать фильтр (блок Antitinkling). Данный блок предназначен для подавления дребезга контактов. Из-за этого явления непосредственное подключение кнопки с механическим замыканием контактов к цифровой схеме не всегда допустимо. Суть дребезга заключается в многократном неконтролируемом замыкании и размыкании контактов в момент коммутации, в результате чего на цифровую схему подается множество импульсов вместо одного.

Частота тактового генератора в учебных стендах LESO2 равна 6 МГц, в стендах LESO2.1 и LESO2.3 – 50 МГц. Делитель частоты должен обеспечить интервал между импульсами больше, чем длительность дребезга и менее чем длительность нажатия кнопки. На рис. 1.46 показана схема подавителя дребезга с использованием суммирующего счетчика-делителя частоты. В нашем случае 19-разрядный счетчик обеспечивает коэффициент счета 524287 и выходной сигнал cout с пониженной частотой 95,37 Гц (100 Гц – период 10 мс). Время дребезга кнопки примерно составляет 2 мс.

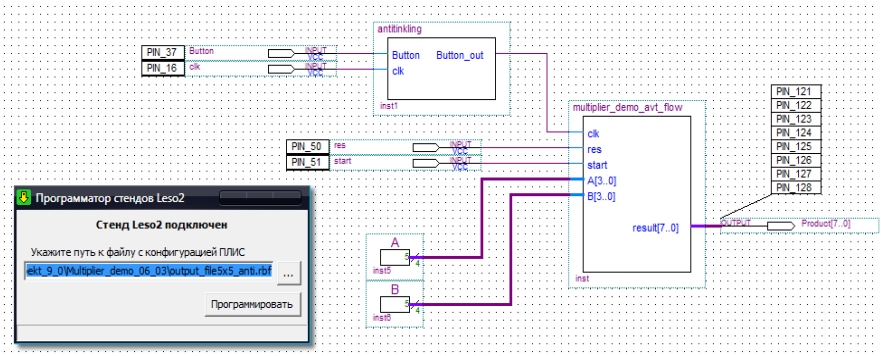
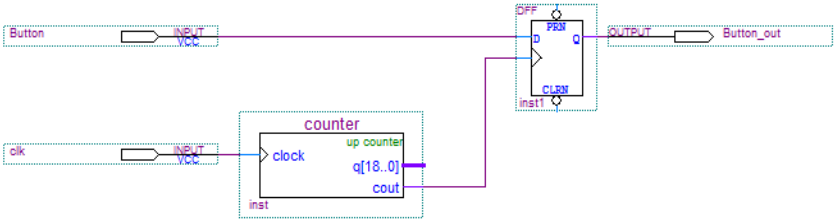
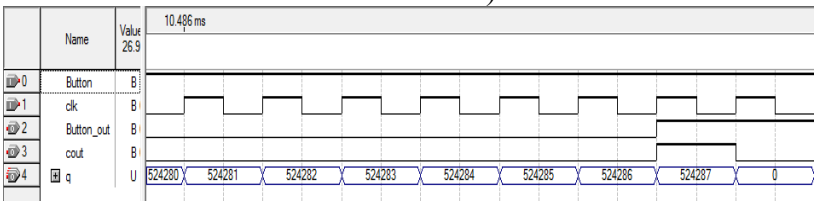


Рис. 1.45. Схема умножителя с подключенными внешними выводами



а)



б)

Рис. 1.46. Подавитель дребезга с использованием суммирующего счетчика-делителя частоты (а) и временные диаграммы его работы (б)

На рис. 1.47 и 1.48 показано тестирование умножителя на примере умножения 5x5. Тестирование осуществляется следующим образом. Согласно рис. 1.44 щелкаем переключателем S2 (pin 50), выполняющим роль

асинхронного сигнала res. Переводим в верхнее положение переключатель S3 (pin 51) – сигнал start, далее нажимаем на кнопку Button (pin 37) один раз, происходит загрузка чисел в умножитель. Переводим переключатель S3 в нижнее положение. Щелкаем три (рис. 1.47) и пять раз (рис. 1.48) кнопкой Button для имитации подачи синхросигнала. Итоговый результат умножения десятичное число 25, а процесс умножения осуществляется за 9 тактов синхрочастоты.

Запрограммировать ПЛИС возможно с помощью Altera USB Blaster без предварительного преобразования *.sof- файла в формат *.rbf (рис. 1.49). Программирование осуществляется непосредственно в САПР Quartus II (меню Tools/Programmer). В этом случае питание лабораторного стенда LESO2.1 осуществляется через USB-кабель а программирование осуществляется через JTAG-интерфейс.

Учебный лабораторный стенд LESO2.1 отечественной разработки содержит хороший функциональный набор для занятий по цифровой схемотехнике и может быть использован для изучения основ проектирования комбинационных и последовательностных устройств в базисе ПЛИС.

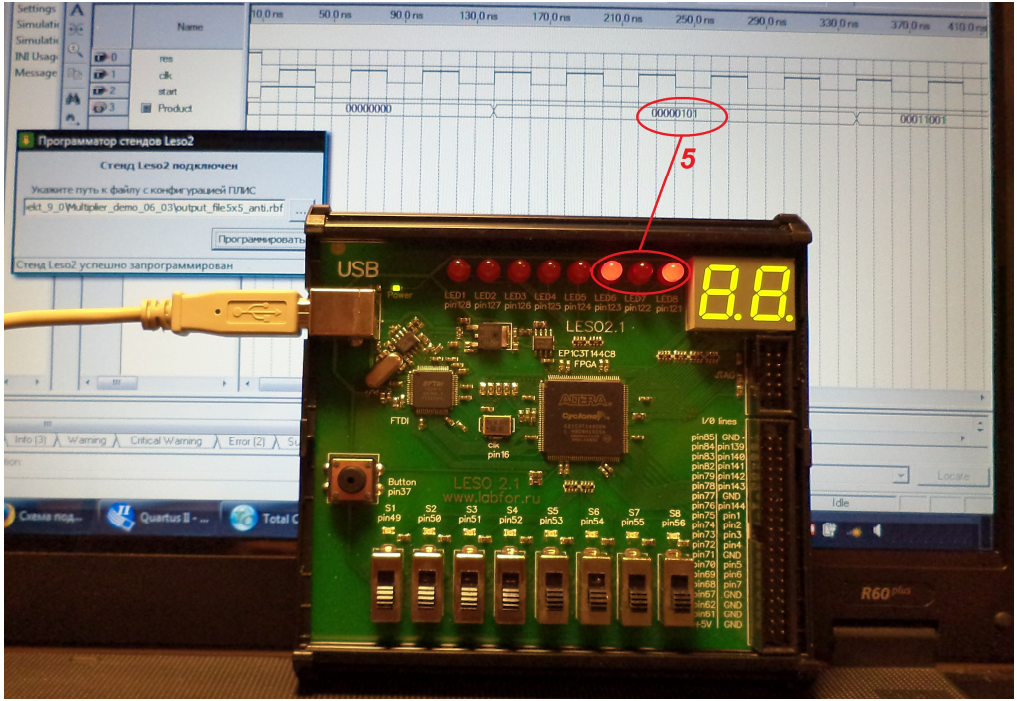


Рис. 1.47. Тестирование умножителя на примере умножения 5x5. Промежуточный результат 5

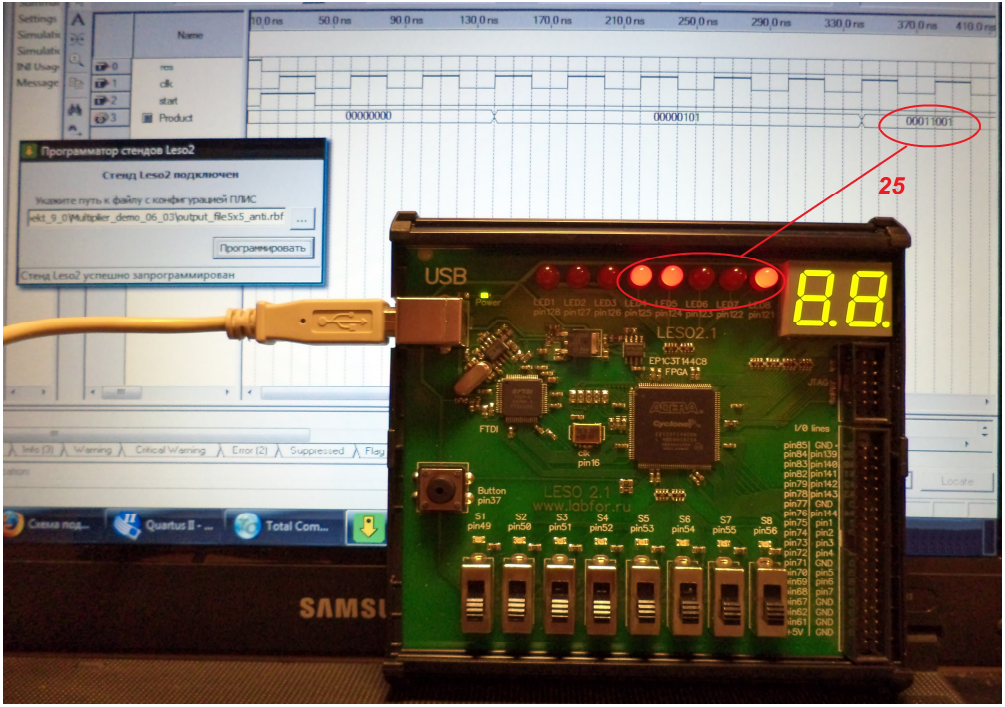


Рис. 1.48. Тестирование умножителя на примере умножения 5x5. Итоговый результат 25

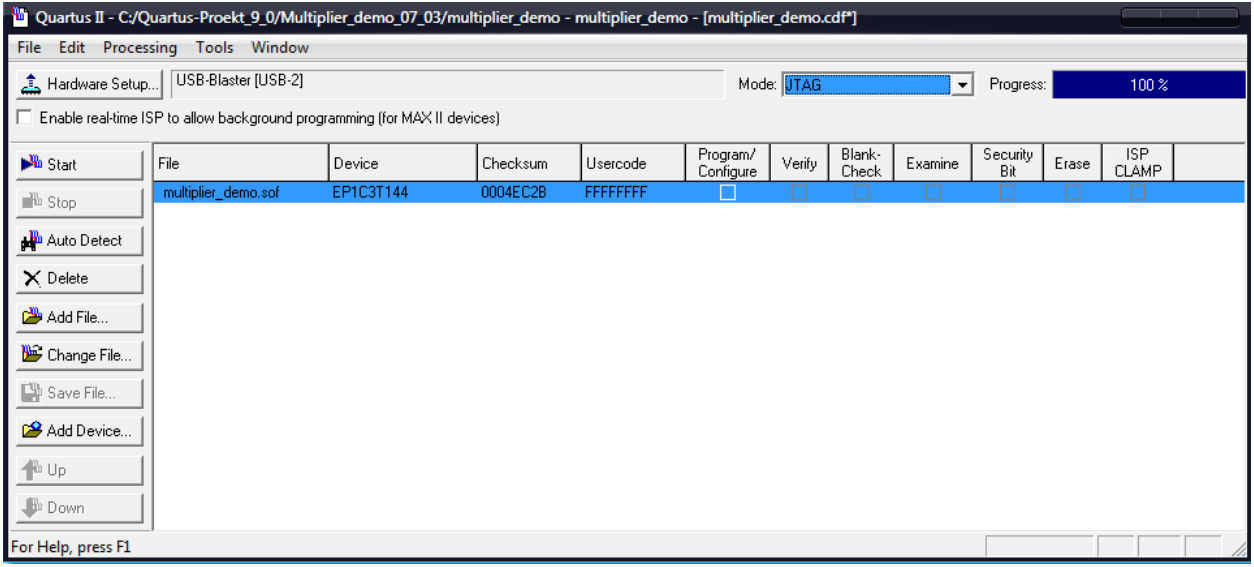


Рис. 1.49. Окно утилиты программирования ПЛИС

2. ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ ФИЛЬТРОВ В БАЗИСЕ ПЛИС

2.1. Проектирование КИХ-фильтров с использованием системы визуально-имитационного моделирования Matlab/Simulink

Рассмотрим особенности проектирования КИХ-фильтра в системе Matlab/Simulink (пакет Signal Processing, среда FDATool) и с применением мегафункции Mega Core FIR Compiler САПР ПЛИС Quartus II Altera.

Главным достоинством среды FDATool от других программ расчета КИХ-фильтров является возможность генерации кода языка VHDL с помощью приложения Simulink HDL Coder. Сгенерированный в автоматическом режиме код языка VHDL может быть использован в системе цифрового моделирования ModelSim (Mentor Graphics HDL simulator).

На рис. 2.1 показана амплитудно-частотная характеристика (АЧХ) КИХ-фильтра. Серые области на рис. 2.1 демонстрируют допуски, превышать границы которых АЧХ фильтра не должна. Исходные данные для расчета КИХ-фильтра: частота взятия отчетов F_S; выбор порядка фильтра n; граница полосы пропускания f_p; граница полосы задерживания (подавления) f_s; неравномерность АЧХ в полосе (полосах) пропускания δ_1 (R_p); минимальное затухание в полосе задерживания δ_2 (R_S).

На практике, как правило, вместо δ_1, δ_2 задают логарифмические величины R_p, R_s, заданные в децибелах:

$$A_p = 20 \lg \frac{1+\delta_1}{1-\delta_1}.$$
$$A_a = 20 \lg \delta_2$$

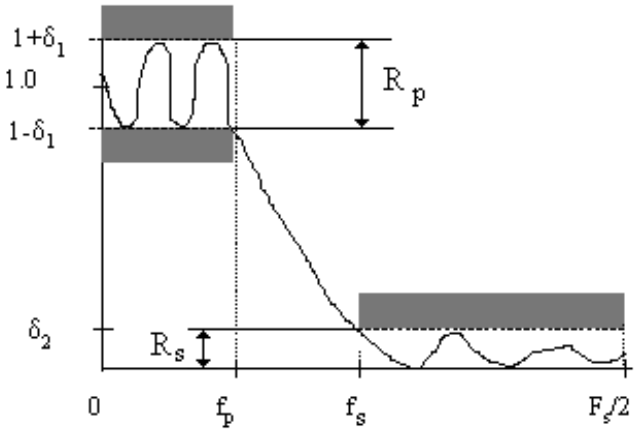


Рис. 2.1. Амплитудно-частотная характеристика фильтра нижних частот

Для построения специализированного устройства, реализующего алгоритм цифровой фильтрации, могут быть использованы регистры, умножители, сумматоры и т.д. – и соответствующее управляющее устройство для управления последовательностью операций. После расчета коэффициентов и выбора структуры фильтра решаются вопросы выбора кодирования чисел (прямой или дополнительный код), способов их представления (с фиксированной или плавающей запятой) и выбора элементной базы.

Исходные данные для расчета КИХ-фильтра нижних частот показаны в табл. 2.1. Пример расчета КИХ-фильтра в среде FDATool показан на рис. 2.2. Среда FDATool представляет графический интерфейс для расчета фильтров и просмотра их характеристик. На вкладке Design Filter зададим тип синтезируемой АЧХ - фильтр нижних частот, тип фильтра – нерекурсивный (FIR), метод синтеза – метод окон (синтез с использованием весовых функций).

Таблица 2.1

Исходные данные для расчета КИХ-фильтра нижних частот

Параметры фильтра	Значение
Фильтр нижних частот	Low Pass
Частота взятия отсчетов F_s, Гц	48000
Неравномерность АЧХ в полосе пропускания R_p, Дб	1
Минимальное затухание в полосе задерживания R_s, Дб	80
Переходная полоса, Гц	2400
Частота среза, F_c, Гц	9600
Тип окна	Blackman

Среда FDATool поддерживает больше методов синтеза, чем мегафункция Mega Core FIR. Преимущество мегафункции в том, что порядок проектируемого КИХ-фильтра (число отводов) оценивается автоматически, но синтез АЧХ осуществляеться методом окон.

Этот недостаток компенсируется возможностью загрузки коэффициентов проектируемого фильтра, полученных, например, с использованием среды FDATool. При проектировании КИХ-фильтра в среде FDATool используются следующие методы: Equiriple – синтез фильтров с равномерными пульсациями АЧХ методом Ремеза; Least-Squares – минимизация среднеквадратичного отклонения АЧХ от заданной и метод окон (Window). В разделе Filter Order зададим порядок КИХ-фильтра. Порядок КИХ-фильтра зададим тот, который рекомендует выбрать мегафункция Mega Core FIR. Мегафункция также предлагает и метод синтеза (окно Blackman - Блекмена). Расчет фильтра осуществляется нажатием кнопки Design Filter. На рис. 2.2 показана АЧХ, вычисленная с использованием формата с плавающей (штрих-

пунктирная линия) и формата с фиксированной запятой (непрерывная линия).

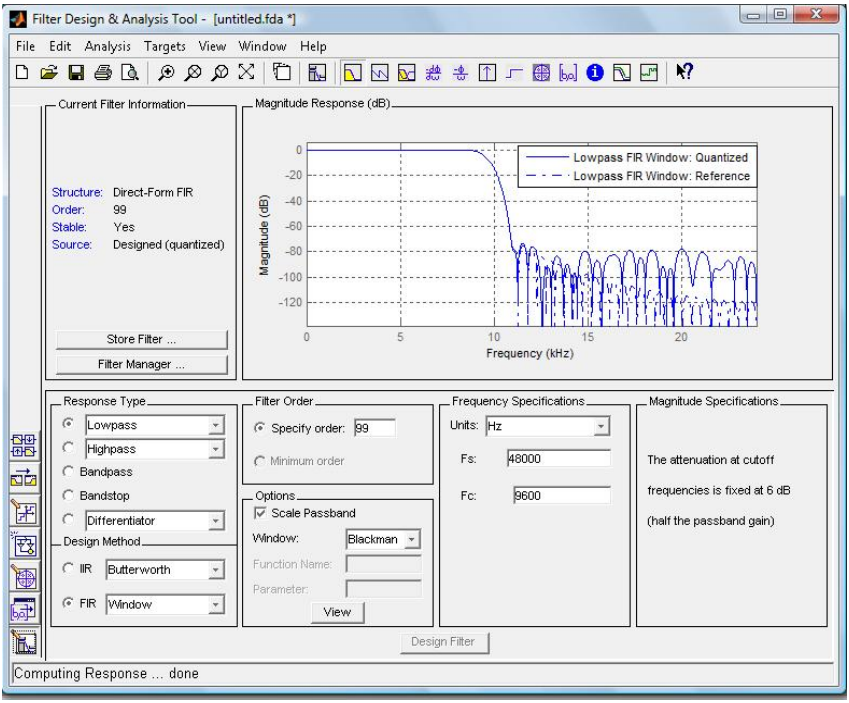


Рис. 2.2. Интерфейс среды FDATool. Пример расчета АЧХ КИХ-фильтра

На рис. 2.3 показана синтезируемая АЧХ (задается комплексный коэффициент передачи $|H(f)|$, определенный в диапазоне частот от нуля до $F_2/2$). Частота среза задается равной $F_c = 9600$ Гц. В мегафункции Mega Core FIR Compiler задается переходная полоса (Transition Bandwith) равная 2400 Гц и частота среза равная 9600 Гц (обозначается как cutoff freq (1)).

В методе окон $|H(f)|$ обратное преобразование Фурье этой характеристики дает бесконечную в обе стороны последовательность отсчетов импульсной характеристики. Для получения КИХ-фильтра заданного порядка эта последовательность усекается путем выбора центрального фрагмента нужной длины. Для ослабления паразитных эффектов в этом методе синтеза усеченная импульсная характеристика умножается на весовую функцию (окно), плавно спадающую к краям.

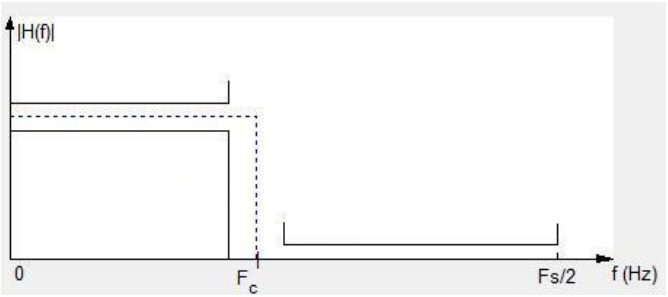


Рис. 2.3. Характеристики синтезируемой АЧХ (окно Blackman) КИХ-фильтра в среде FDATool

Вкладка Realize Model позволяет импортировать спроектированный КИХ-фильтр (модель) в Simulink (рис. 2.2). На рис. 2.4, *а* показана модель КИХ-фильтра (имя модели Filter simulink), построенная как с использованием базовых элементов (задержка, сумма, коэффициент усиления) цифровых фильтров, так и с использованием S-функции (модель КИХ-фильтра, построенная с использованием мегафункции Mega Core FIR Compiler). На рис. 2.4, *б* показан сигнал до фильтрации, а на рис. 2.4, *в* и *г* после. Меню Targets опция Generate HDL позволяют сгенерировать код фильтра на языке VHDL (рис. 2.5). Выберем параллельную архитектуру КИХ-фильтра, обладающего высокой производительностью.

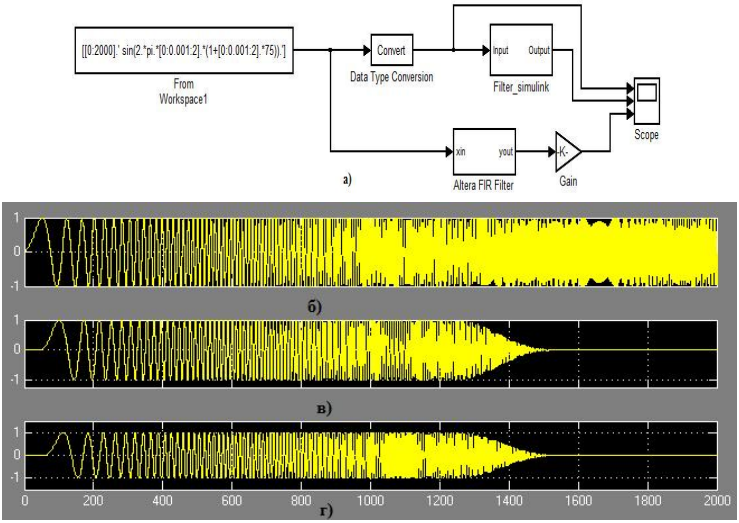


Рис. 2.4. Имитационная модель КИХ-фильтра в системе Matlab/Simulink (а) и сигнал до (б) и после фильтрации КИХ-фильтром нижних частот с использованием среды FDATool (в) и с использованием мегафункции Core FIR Compiler САПР ПЛИС Quartus

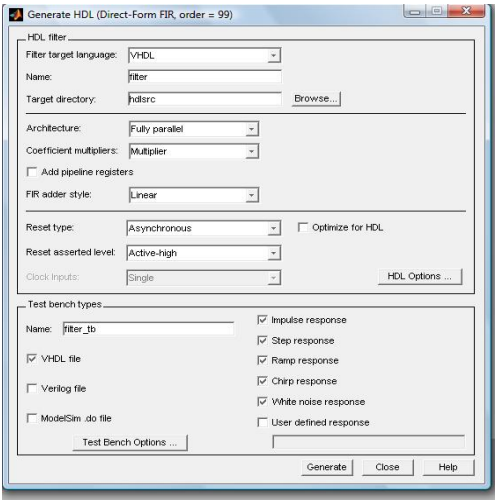


Рис. 2.5. Окно Simulink HDL Coder

2.2. Проектирование параллельных КИХ-фильтров в базисе ПЛИС

На рис. 2.6 показаны структуры фильтров, характерные для реализации в базисе сигнальных цифровых процессоров, а на рис. 2.7 показаны структуры фильтров, характерные для реализации в базисе ПЛИС. В качестве матричных умножителей могут быть использованы параллельные векторные умножители. На рис. 2.8 показан 2-разрядный векторный умножитель с использованием двух идентичных таблиц перекодировки LUT1 и LUT2 для формирования частичных произведений P1(n) и P2(n), которые необходимо сложить с учетом их веса. Каждая LUT образована из четырех LUT логических элементов (ЛЭ) ПЛИС. Результат вычисления P2(n) необходимо сдвинуть на один разряд влево. Такой умножитель может быть использован для структуры фильтра четыре отвода два бита при 2-разрядном представлении коэффициентов. В случае если число отводов останется постоянным (например четыре в случае симметрии коэффициентов фильтра), а разрядность входного сигнала, подлежащего фильтрации, и коэффициентов фильтра составит восемь бит, то уже потребуется восемь LUT, каждая из которых будет содержать восемь LUT ЛЭ. При этом увеличивается и число многоразрядных сумматоров и операций сдвига (рис. 2.9).

Параллельные КИХ-фильтры, реализованные в базисе ПЛИС, обладая наивысшим быстродействием, позволяют получать результат фильтрации, например, для КИХ-фильтра со структурой 120 отводов 12 бит уже после первого синхроимпульса, последовательные через 12, а фильтр в базисе ЦОС-процессоров через 120 синхроимпульсов.

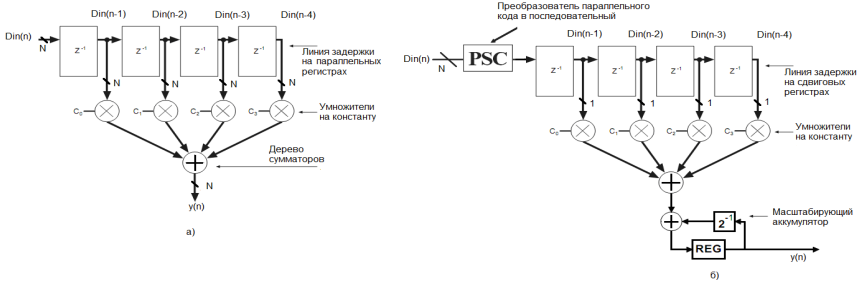


Рис. 2.6. Параллельный (а) и последовательный фильтры (б) на четыре отвода для реализации в базисе цифровых сигнальных процессоров

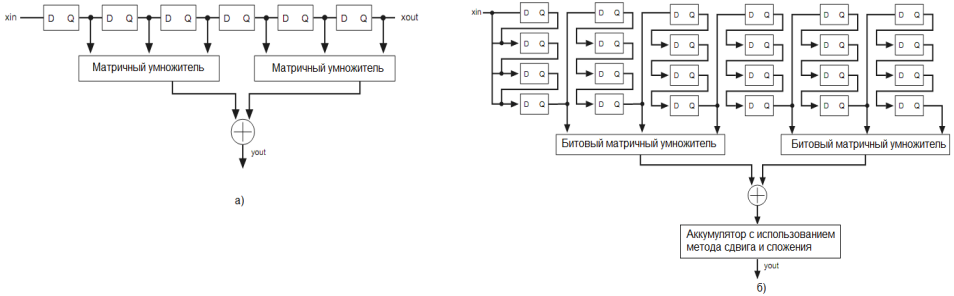


Рис. 2.7. Обобщенное представление структур КИХ-фильтров: а) параллельных; б) последовательных

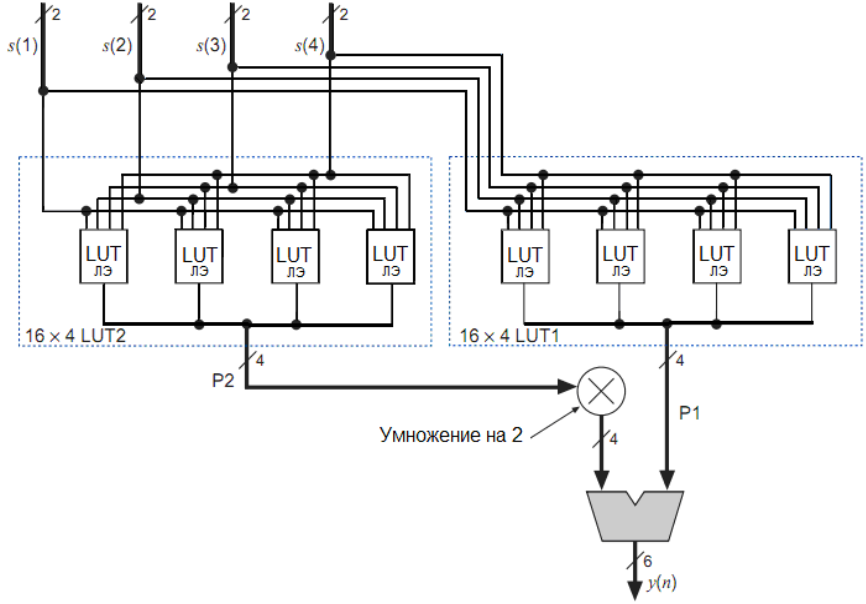


Рис. 2.8. Параллельный векторный умножитель четырех 2-разрядных сигналов на четыре 2-разрядные константы с использованием LUT ЛЭ в ПЛИС серии FLEX

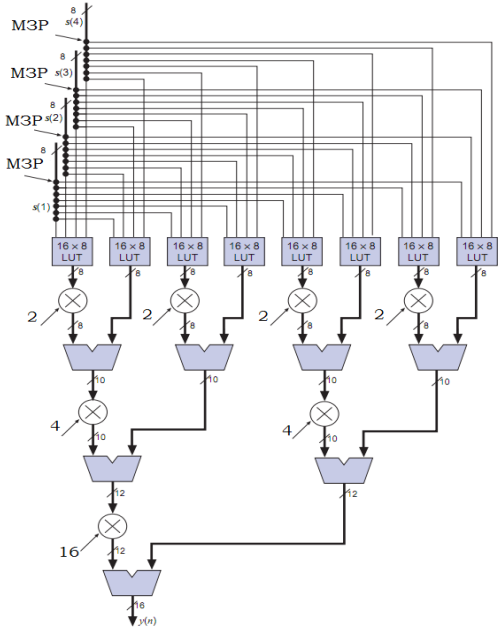


Рис. 2.9. Параллельный векторный умножитель четырех 8-разрядных сигналов на четыре 8-разрядные константы с использованием LUT ЛЭ в ПЛИС серии FLEX

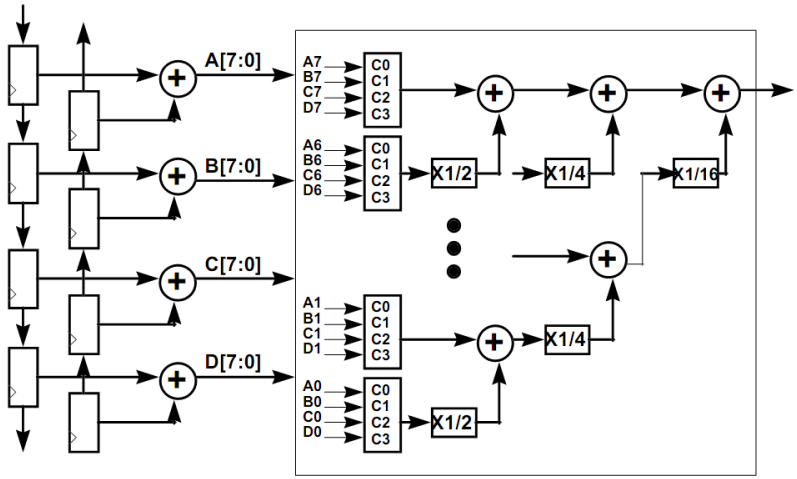
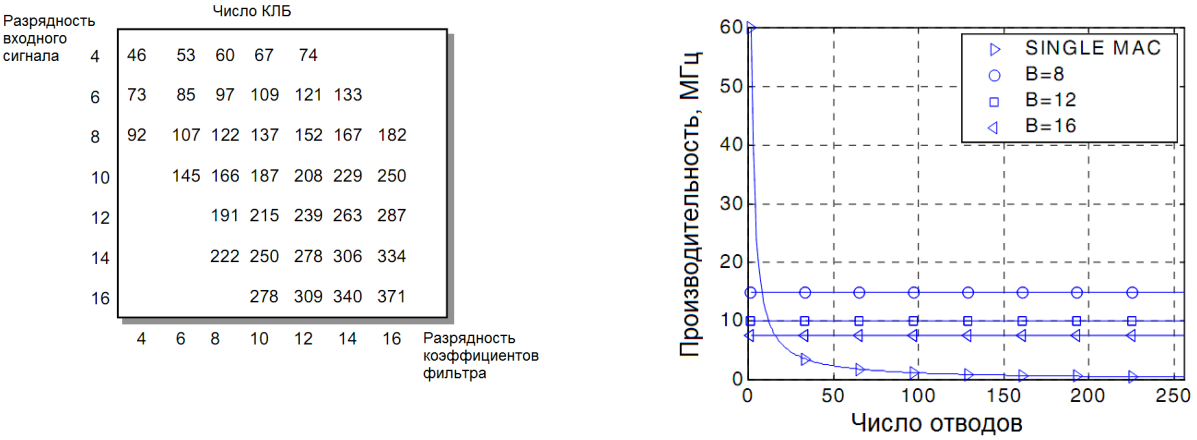


Рис. 2.10. Использование параллельного векторного умножителя четырех 8-разрядных сигналов на четыре 8-разрядные константы в составе КИХ-фильтра на восемь отводов с симметричными коэффициентами в базисе ПЛИС XC4000, построенного с использованием параллельной распределенной арифметики



а) б)

Рис. 2.11. Число задействованных ресурсов для реализации параллельного векторного умножителя в базисе ПЛИС XC4000 (а) и производительность фильтра в зависимости от числа отводов, при частоте тактирования 120 МГц

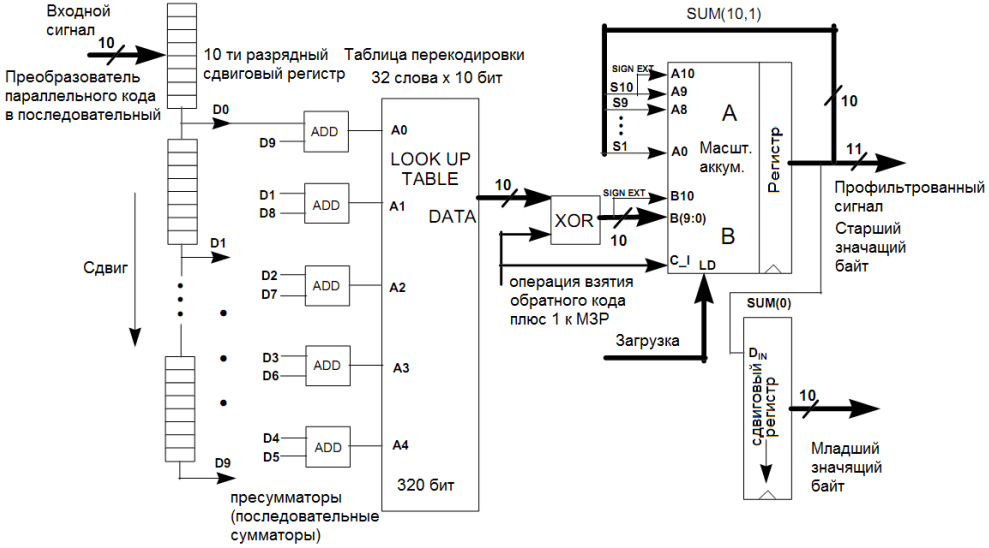


Рис. 2.12. Симметричный КИХ-фильтр со структурой десять отводов десять бит с точностью представления коэффициентов 10 бит с использованием последовательной распределенной арифметики

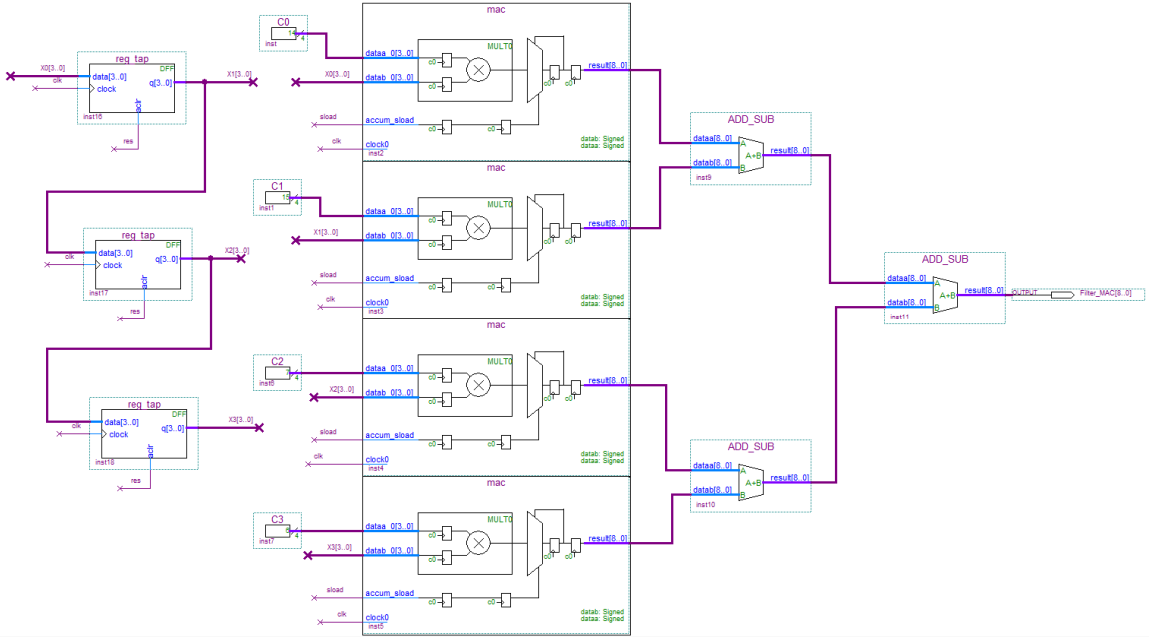


Рис. 2.13. Параллельная реализация КИХ-фильтра на четыре отвода с использованием четырех блоков в САПР ПЛИС Quartus II (мегафункция ALTMULT_ACCUM)

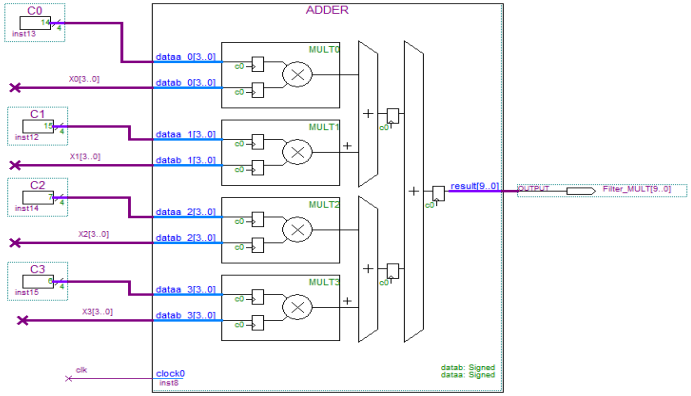


Рис. 2.14. Параллельная реализация КИХ-фильтра на четыре отвода с использованием четырех перемножителей в блоке (мегафункция ALTMULT_ADD, линия задержки такая же, как и на рис. 2.13)

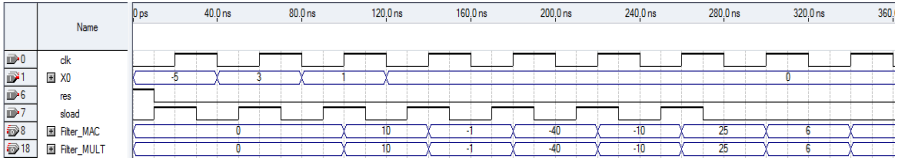


Рис. 2.15. Временные диаграммы работы параллельных фильтров на четыре отвода с использованием мегафункции ALTMULT_ACCUM и ALTMULT_ADD

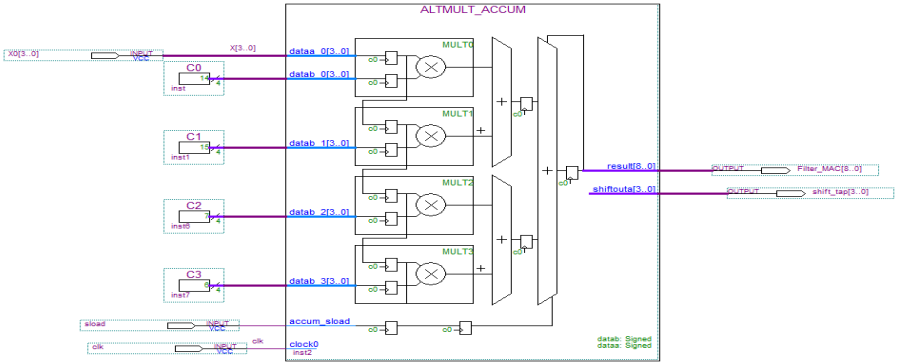


Рис. 2.16. Параллельная реализация КИХ-фильтра на четыре отвода с использованием четырех перемножителей в блоке (мегафункция ALTMULT_ACCUM, линия задержки построена на внутренних регистрах перемножителей MULT0-MULT3)

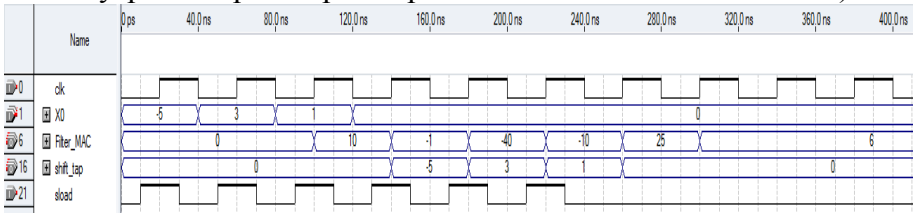


Рис. 2.17. Временные диаграммы работы фильтра на четыре отвода с использованием мегафункции ALTMULT_ACCUM

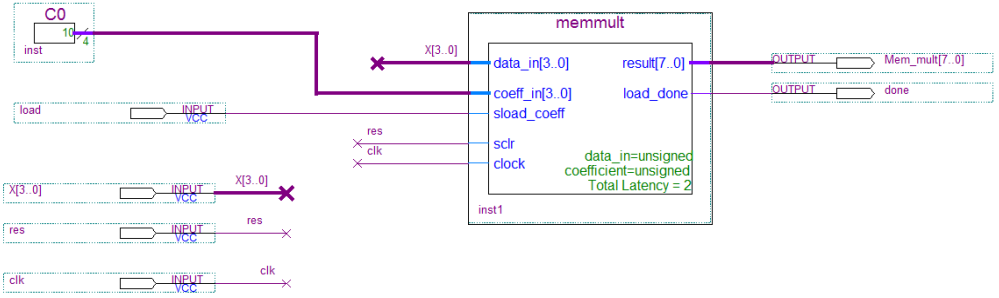


Рис. 2.18. Умножение 11 на 10 с помощью мегафункции ALTMEMMULT

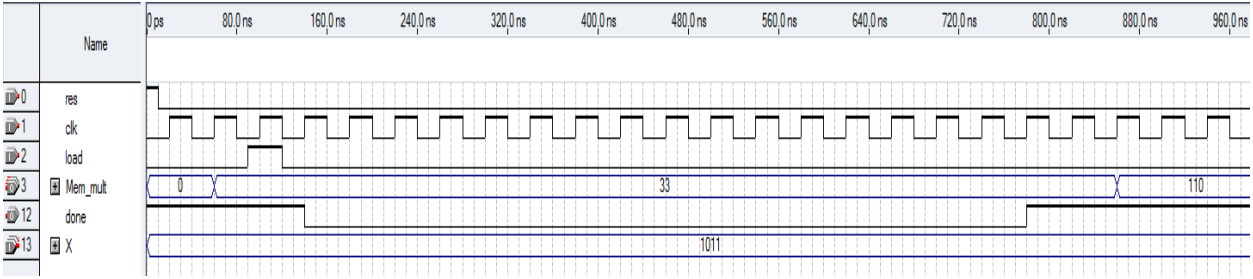


Рис. 2.19. Временные диаграммы умножения 11 на 10. По умолчанию в мегафункцию загружена константа 3. Результат 110

Рис. 2.10 показывает использование параллельного векторного умножителя четырех 8-разрядных сигналов на четыре 8-разрядные константы в составе КИХ-фильтра на 8 отводов с симметричными коэффициентами. Симметричность коэффициентов позволяет использовать пресумматоры на выходах линии задержки, что и обеспечит формирование четырех 8-разрядных сигналов. Рис. 2.11, *а* демонстрирует число задействованных конфигурируемых логических блоков (КЛБ) для реализации параллельного векторного умножителя в базисе ПЛИС серии XC4000. Для симметричного КИХ-фильтра со структурой 8 отводов 8 бит с точностью представления коэффициентов 8 бит потребуется 122 КЛБ ПЛИС серии XC4000 фирмы Xilinx, при этом обеспечивается быстродействие 50-70 MSPS.

В случае последовательной структуры (рис. 2.7, *б*) применяется одна единственная LUT для вычисления частичных произведений (рис. 2.12). Такой фильтр обрабатывает только один разряд входного сигнала в течение такта. Последовательно вычисляемые частичные произведения накапливаются в масштабирующем аккумуляторе. После N для несимметричного и N+1 тактов синхроимпульсов для симметричного фильтра на выходе появляется результат, где N разрядность входного сигнала подлежащего фильтрации. Для обеспечения правильной работы фильтра требуется управляющий автомат. Производительность фильтра определяется как fclk/N для несимметричного и как fclk/N+1 для симметричного фильтра. Рис. 2.11, *б* показывает, что с ростом числа отводов производительность фильтра остается постоянной, в то время как у фильтра на базе ЦОС-процессора с использованием MAC-блоков начинает резко падать при числе отводов более 16.

Перемножители сигналов играют ключевую роль в проектировании высокопроизводительных цифровых фильтров.

Покажем различные варианты реализации КИХ-фильтров с использованием перемножителей на мегафункциях

ALTMULT_ACCUM, ALTMULT_ADD и ALTMEMMULT САПР Quartus II компании Altera в базисе ПЛИС, а затем сосредоточим внимание на реализации умножения методом правого сдвига с накоплением, применяемого для разработки масштабирующего аккумулятора.

Рассмотрим уравнение КИХ-фильтра (нерекурсивного цифрового фильтра с конечно-импульсной характеристикой), которое представляется как арифметическая сумма произведений:

$$y = \sum_{k=0}^{K-1} C_k \cdot x_k \ , \qquad (2.1)$$

где $y-$ отклик цепи; $x_k -$ $k-$я входная переменная; $C_k -$весовой коэффициент $k-$й входной переменной, который является постоянным для всех n; K - число отводов фильтра.

В качестве простейшего примера рассмотрим три варианта проектирования параллельного КИХ-фильтра на четыре отвода: $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$ с использованием мегафункций САПР ПЛИС Quartus II компании Altera, объединенных общей идеей использования перемножителей цифровых сигналов и “дерева сумматоров”. Предположим, что коэффициенты фильтра целочисленные со знаком известны и равны $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$. На вход КИХ-фильтра поступают входные отсчеты -5, 3, 1 и 0. Правильные значения на выходе фильтра: 10, -1, -40, -10, 26, 6 и т.д., т.е. согласно формуле $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$.

Первый вариант. Параллельная реализация КИХ-фильтра на четыре отвода с использованием четырех блоков умножения с накоплением. В проекте используются четыре мегафункции ALTMULT_ACCUM (рис. 2.13). Каждый блок использует один перемножитель и один сумматор-аккумулятор. Для параллельной реализации фильтра на четыре отвода требуются четыре блока и три дополнительных однотипных многоразрядных сумматора, связанных по принципу “дерево сумматоров”. Для того чтобы фильтр

работал корректно, необходимо осуществлять синхронную загрузку каждого произведения в каждый сумматор-аккумулятор каждого блока, для этого используется дополнительный вход мегафункции accum_sload. На рис. 2.13 также показана внешняя линия задержки на четыре отвода из трех 4-разрядных регистров, тактируемых фронтом синхросигнала. Коэффициенты фильтра представляются в двоичном виде с учетом знака числа и загружаются с помощью мегафункции LPM_CONSTANT.

Второй вариант. Параллельная реализация КИХ-фильтра на четыре отвода с использованием четырех перемножителей в блоке на мегафункции ALTMULT_ADD (функция умножения и сложения) в САПР ПЛИС Quartus II показана на рис. 2.14. В мегафункции ALTMULT_ACCUM используется три встроенных сумматора. Профильтрованные значения показаны на рис. 2.15. Сравнивая временные диаграммы, видим, что профильтрованные значения на выходе у двух фильтров, построенных на разных мегафункциях, совпадают.

Значительно упростить разработку КИХ-фильтра позволяет иное использование мегафункции ALTMULT_ACCUM (модификация варианта 1). Фактически это одна мегафункция (блок с четырьмя перемножителями), в которой линия задержки организована на внутренних регистрах располагающихся на входах перемножителей. В мегафункции используются встроенные два сумматора и один сумматор-аккумулятор (рис. 2.16). Временные диаграммы работы фильтров, показанные на рис. 2.17, не отличаются от диаграмм на рис. 2.15.

Третий вариант. Рассмотрим умножение десятичного числа 11 на 10 на примере мегафункции ALTMEMMULT (рис. 2.18). Мегафункция ALTMEMMULT (программный умножитель) предназначена для умножения числа на константу, которая хранится в блочной памяти ПЛИС (M512, M4K, M9K и MLAB-блоки), обеспечивая наивысшее быстродействие, лимитируемое латентностью. Однако константу можно загрузить и из внешнего порта.

Считаем, что десятичное число 10 это константа и загружается из внешнего порта. По умолчанию, в мегафункцию загружена, например, константа 3. Латентность мегафункции - 2, т.е. доступность результата умножения числа на константу, если константа хранится в памяти мегафункции, возможно уже после 2 синхроимпульсов (высокий уровень сигнала done, соответствующий порту load_done). Число 3, загруженное в мегафункцию по умолчанию, умноженное на число 11, с входного порта data_in[3..0] дает результат 33. Далее, синхронный сигнал загрузки load (порт sload_coeff) разрешает загрузку числа 10 в перемножитель. Низкий уровень сигнала done в течение 16 тактов синхрочастоты говорит о том, что идет процесс умножения. И лишь спустя 2 синхроимпульса при высоком уровне сигнала done на выходе появляется требуемое число 110. Таким образом, процесс умножения составляет 20 синхроимпульсов от момента появления сигнала load (рис. 2.19).

Применяя мегафункцию ALTMEMMULT, разработаем параллельную реализацию КИХ-фильтра на четыре отвода с использованием четырех перемножителей (4 блока по 1 перемножителю в каждом) в САПР ПЛИС Quartus II и дерева сумматоров (рис. 2.20). Внешняя линия задержки состоит не из трех регистров, как в первых двух вариантах, а из четырех регистров. Дополнительно требуется, как и в первом варианте, три однотипных многоразрядных сумматора.

Латентность каждого умножителя равна двум. В каждый умножитель по умолчанию загружено число 0. Временные диаграммы работы фильтра на четыре отвода с использованием мегафункции ALTMEMMULT показаны на рис. 2.21. Коэффициенты фильтра $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$ загружаются из внешнего порта. Для этих целей используется мегафункция LPM_CONSTANT.

Рассмотрим вариант, когда коэффициенты фильтра загружаются из блочной памяти в ПЛИС. В мегафункции ALTMEMMULT коэффициенты представляются как целочисленные значения со знаком (рис. 2.22). Временные

диаграммы работы фильтра на четыре отвода с использованием мегафункции ALTMEMMULT показаны на рис. 2.23. Сравнивая рис. 2.21 и рис. 2.23 видим, что быстродействие фильтра в этом случае значительно увеличивается за счет хранения коэффициентов в блочной памяти. Фильтр на мегафункции ALTMULT_ACCUM (вариант 1) является самым затратным, т.к. требует 16 аппаратных перемножителей, три дополнительных сумматора и внешнюю линию задержки (табл. 2.2, АЛМ-адаптивный логический модуль).

Наиболее оптимальным по числу используемых ресурсов ПЛИС является модификация варианта 1 (рис. 2.16), которая позволяет построить параллельный КИХ-фильтр на четыре отвода с использованием всего лишь одного блока со встроенными перемножителями в количестве четырех штук, двумя сумматорами, сумматором-аккумулятором и линией задержки. Мегафункции ALTMULT_ADD (рис. 2.14) так же позволяет построить параллельный КИХ-фильтр с использованием всего лишь одного блока со встроенными перемножителями и сумматорами. Использование внешней линии задержки из трех регистров приводит к незначительному увеличению ресурсов и не сказывается на быстродействие. Экономия ресурсов ПЛИС в первом модифицированном и во втором вариантах достигается за счет использования встроенных четырех аппаратных перемножителей размерностью 18x18. Фильтр на мегафункции ALTMEMMULT с загрузкой коэффициентов из внешнего порта обладает пониженным быстродействием (рис. 2.20). Использование же блочной памяти (рис. 2.22) для хранения коэффициентов фильтра внутри ПЛИС значительно упрощает процесс разработки и не приводит к существенному увеличению ресурсов за счет использования внешней линии задержки на четырех регистрах, дополнительных трех однотипных многоразрядных сумматоров и не снижает быстродействие проекта.

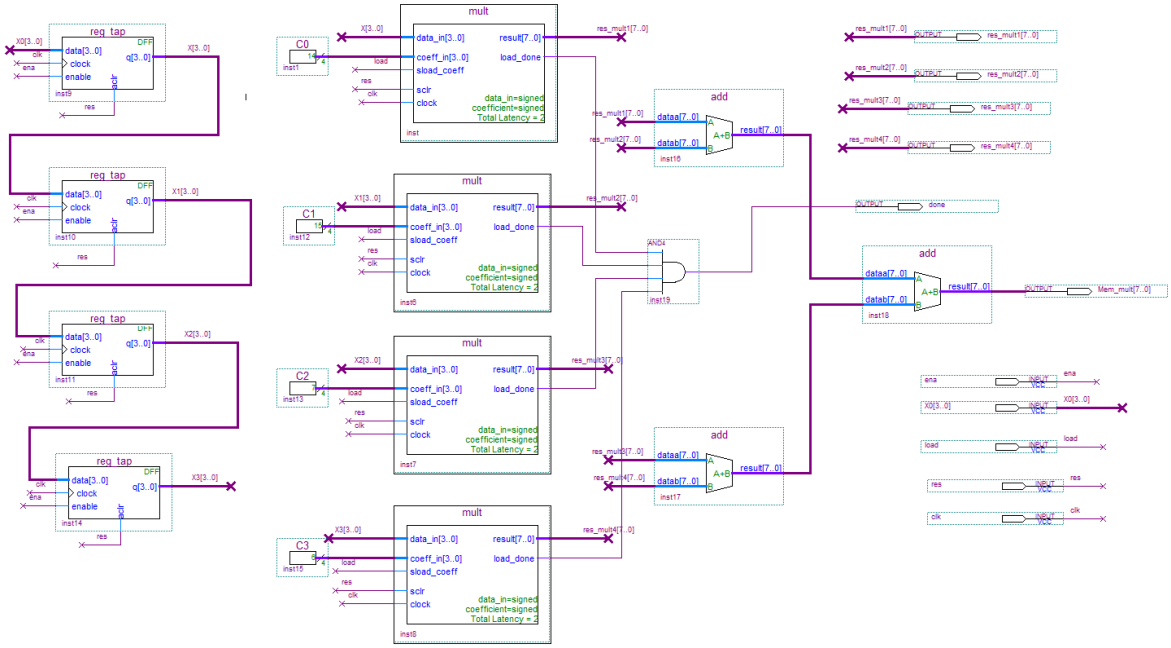


Рис. 2.20. Параллельная реализация КИХ-фильтра на четыре отвода с использованием четырех умножителей в САПР ПЛИС Quartus II (мегафункция ALTMEMMULT, линия задержки построена на четырех регистрах, коэффициенты фильтра загружаются из внешнего порта)

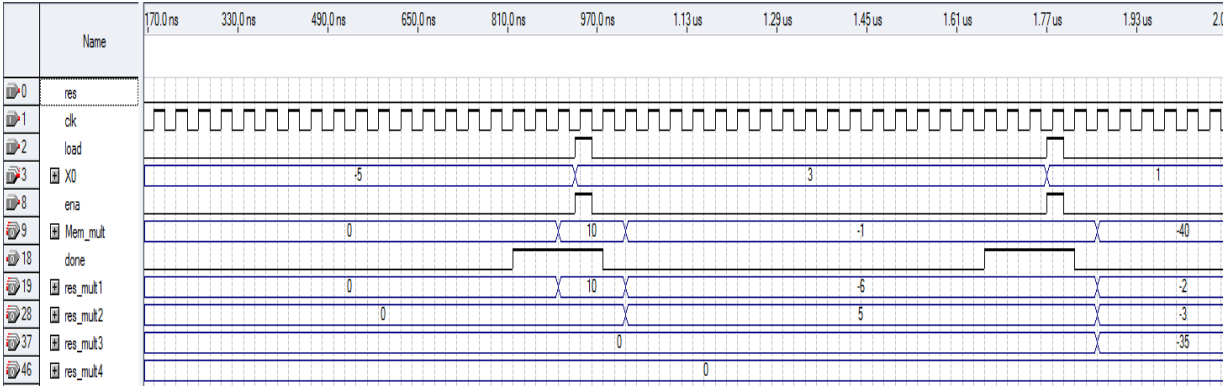


Рис. 2.21. Временные диаграммы работы фильтра на четыре отвода с использованием мегафункции ALTMEMMULT (коэффициенты фильтра загружаются из внешнего порта)

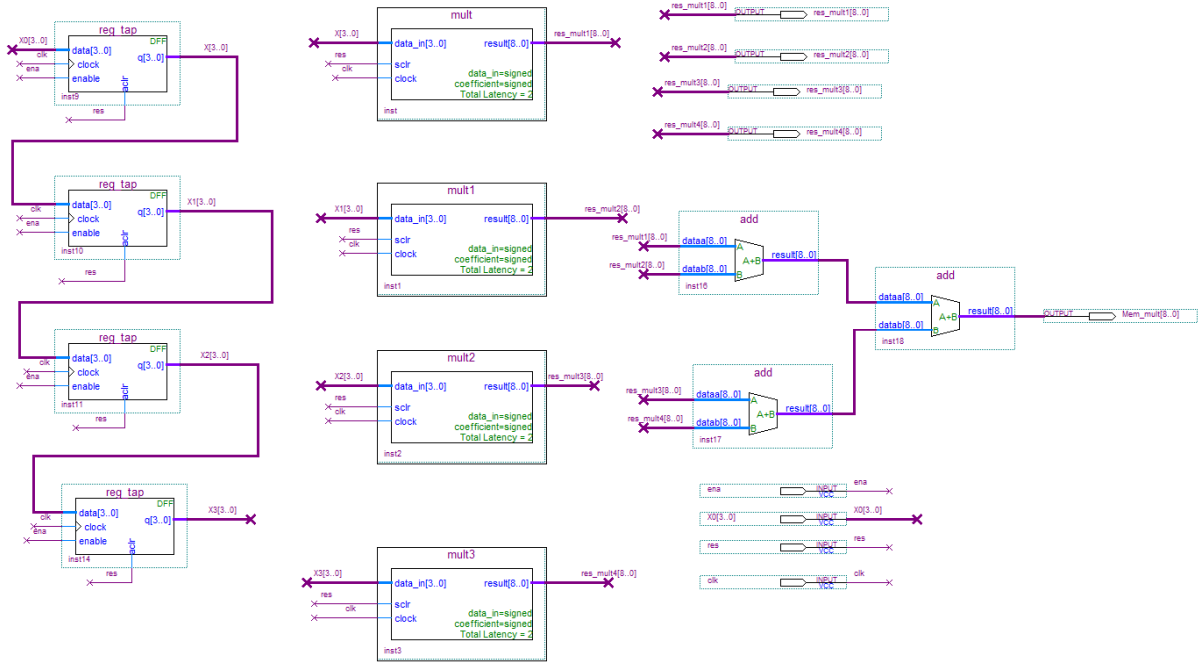


Рис. 2.22. Параллельная реализация КИХ-фильтра на четыре отвода с использованием четырех перемножителей в САПР ПЛИС Quartus II (мегафункция ALTMEMMULT, линия задержки построена на четырех регистрах, коэффициенты фильтра загружаются из блочной памяти)

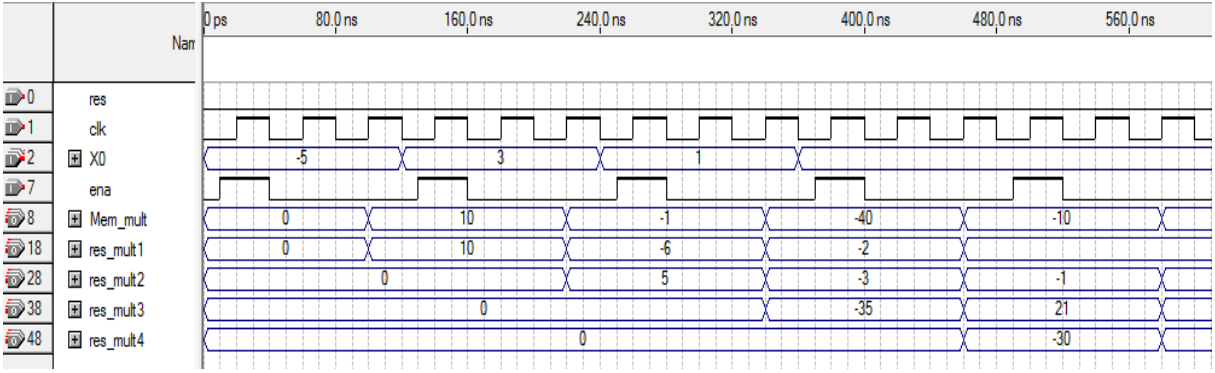


Рис. 2.23. Временные диаграммы работы фильтра на четыре отвода с использованием мегафункции ALTMEMMULT (коэффициенты фильтра загружаются из блочной памяти ПЛИС)

Таблица 2.2

Анализ задействованных ресурсов ПЛИС серии Stratix при реализации параллельных КИХ-фильтров на четыре отвода с использованием различных мегафункций

Ресурсы ПЛИС серии Stratix	Мегафункция ALTMULT_ACCUM. Четыре блока по одному перемножителю в каждом, внешняя линия задержки из трех регистров / четыре перемножителя в блоке, встроенная линия задержки		Мегафункция ALT-MULT_ADD. Четыре перемножителя в блоке, внешняя линия задержки из трех регистров	Мегафункция ALTMEM-MULT. Четыре блока по одному перемножителю в каждом (коэффициенты фильтра загружаются из внешнего порта)	Мегафункция ALTMEMMULT. Четыре блока по одному перемножителю в каждом (коэффициенты фильтра загружаются из блочной памяти)
	Вариант 1	Модифи-кация варианта 1	Вариант 2	Вариант 3	
1	2	3	4	5	6
Кол-во АЛМ для реализации комбинацион-ных функций	18	0	0	181	18

Продолжение табл. 2.2

1	2	3	4	5	6
Кол-во АЛМ с памятью	0	0	0	64	36
АЛМ	18	0	8	152	4
Кол-во выделенных регистров	12	0	12	248	68
Аппаратные перемножители (DSP 18x18)	16	4	4	0	0
Кол-во АЛМ для выполнения комбинационных функций без использования регистров	18	0	0	49	36
Кол-во АЛМ под регистерные ресурсы	12	0	12	50	196
Кол-во АЛМ под комбинационные и регистерные ресурсы	0	0	0	196	18
Рабочая частота в наихудшем случае, МГц	400	400	400	331	400

2.3. Проектирование КИХ-фильтра с использованием умножителя на методе правого сдвига и сложения

Рассмотрим уравнение КИХ-фильтра (нерекурсивного цифрового фильтра с конечно-импульсной характеристикой), которое представляется как арифметическая сумма произведений

$$y = \sum_{k=0}^{K-1} c_k \cdot x_k \,, \qquad (2.2)$$

где $y-$ отклик цепи; x_k- $k-$я входная переменная; c_k – весовой коэффициент $k-$й входной переменной, который является постоянным для всех n; K - число отводов фильтра.

На рис. 2.24 показана тестовая схема КИХ-фильтра на четыре отвода $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$ в САПР ПЛИС Quartus II для реализации в базисе ПЛИС серии Cyclone II, состоящая из линии задержки, четырех умножителей и дерева многоразрядных сумматоров. Предположим, что коэффициенты фильтра целочисленные со знаком, известны и равны $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$.

Числа (входные отсчеты) поступают с выходов линии задержки на регистрах reg_tap на входы data_in[3..0] мегафункции ALTMEMMULT. Константы C_0, C_1, C_2 и C_4 в которых хранятся значения коэффициентов в дополнительном коде (14D, 15D, 7D и 6D), подключены к входам coeff_in[3..0]. В каждой из четырех мегафункций ALTMEMMULT в блочной памяти ПЛИС типа M9K хранятся нулевые коэффициенты (могут быть и ненулевыми). Режим загрузки с внешнего порта или из блочной памяти определяется опцией ***create ports to allow loading coefficients***. Латентность мегафункции - 2 такта синхросигнала. Смоделируем прохождение дельта-функции по структуре фильтра. Для этого на вход линии задержки фильтра X0[3..0] в дополнительном коде подадим единичный импульс

лог.1. На выходе фильтра Mem_mult[7..0] видим коэффициенты фильтра (импульсную характеристику) (рис. 2.25).

Подадим на вход КИХ-фильтра входные отсчеты -5, 3, 1 и 0 (рис. 2.26). Правильные значения на выходе фильтра: 10, -1, -40, -10, 25, 6 и т.д., т.е. согласно формуле (2.1).

Рассмотрим КИХ-фильтр на 4 отвода на умножителях (MAC-блоках) с использованием метода правого сдвига и сложения. Дополнительную информацию о применяемом MAC-блоке можно получить в главе 1, разделы 1.4 и 1.5.

На рис. 2.27 показана линия задержки на двухтактных триггерах с использованием мегафункции LPM_FF. На рис. 2.28 показаны умножители (mac_scal_acc) с константами и дерево сумматоров, а на рис. 2.29 доработанная схема умножителя с учетом работы в составе КИХ-фильтра. Модификация MAC-блока выделена овалом. На рис. 2.30 показаны временные диаграммы работы КИХ-фильтра. Требуемые значения на выходе фильтра 10, -1, -40, -10, 25, 6 выделены.

В табл. 2.3 приведены технические характеристики КИХ-фильтров на четыре отвода реализованных в базисе ПЛИС EP2C5F256C6 (4608 логических элементов, 119808 бит встроенной памяти, 26 аппаратных умножителей 9x9) с использованием разработанного MAC-блока и программных умножителей на мегафункции ALTMEMMULT. Смена данных на выходе КИХ-фильтра в случае использования MAC-блока в качестве умножителя и умножителя на мегафункции ALTMEMMULT в случае загрузки коэффициентов из внешнего порта приблизительно одинаковая (22 MAC и 21 ALTMEMMULT такта синхроимпульса). Мегафункция ALTMULT_ADD так же позволяет построить параллельный КИХ-фильтр на четыре отвода за счет использования четырех встроенных аппаратных перемножителей размерностью операндов 18x18. Рабочая частота в наихудшем случае для ПЛИС серии Stratix III составила 400 МГц.

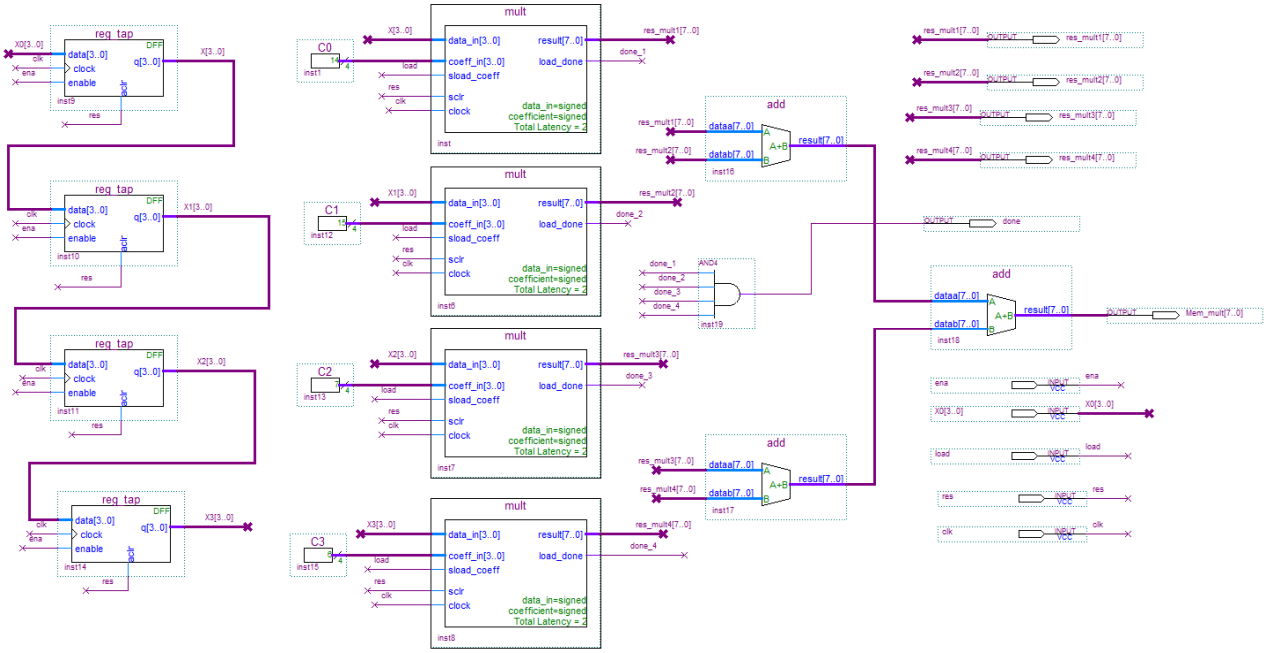


Рис. 2.24. КИХ-фильтр на четыре отвода с использованием мегафункции ALTMEMMULT

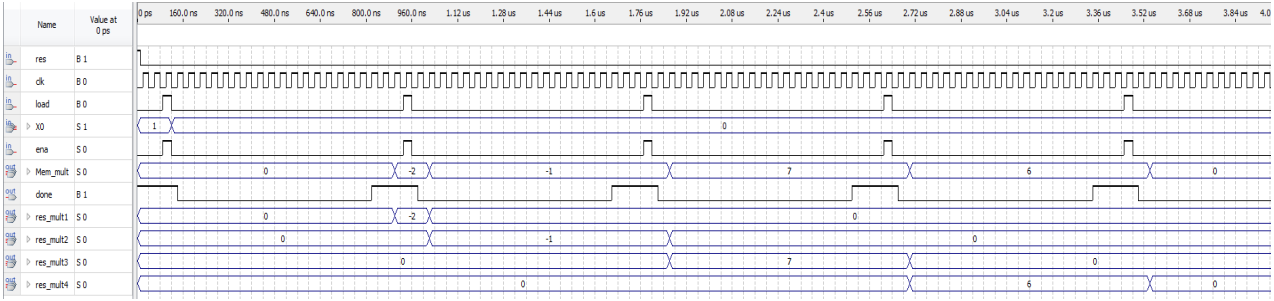


Рис. 2.25. Результат моделирования прохождения аналога дельта-функции по структуре КИХ-фильтра на четыре отвода

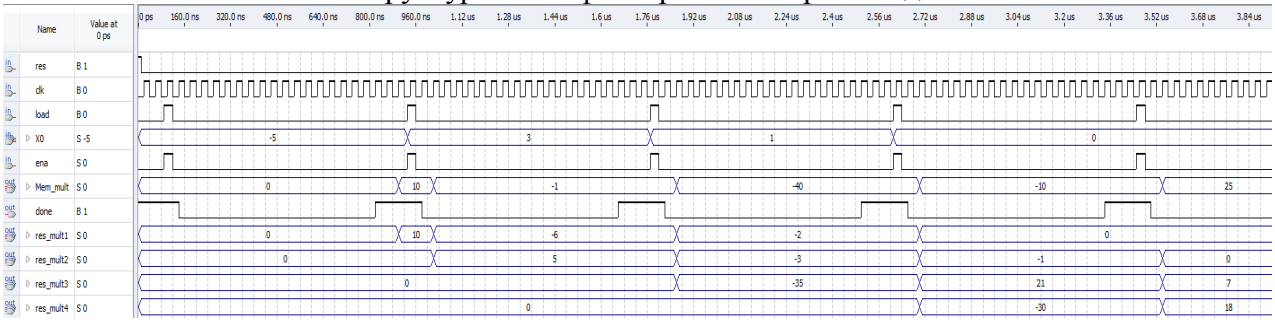


Рис. 2.26. Временные диаграммы работы КИХ-фильтра на четыре отвода при поступлении
входных отчетов -5, 3, 1 и 0. Результат: 10, -1, -40, -10, 25

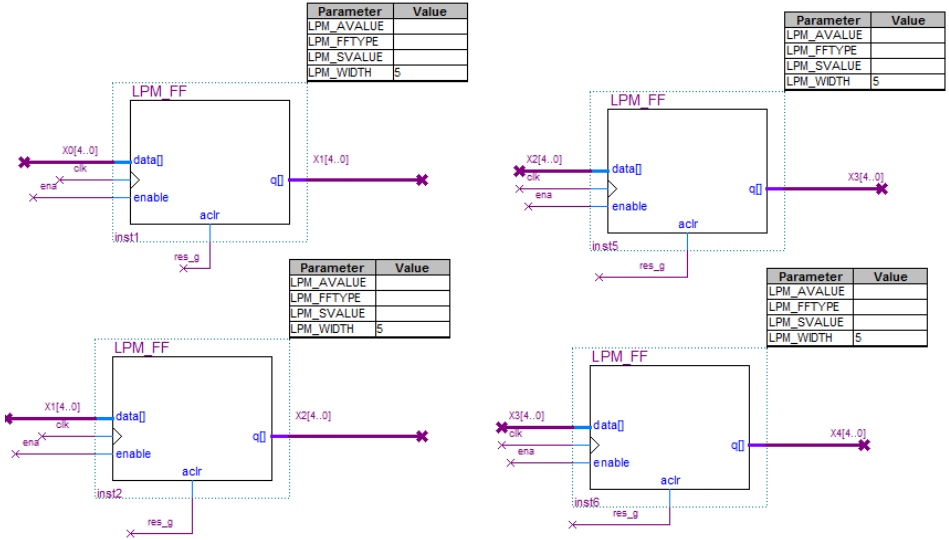


Рис. 2.27. Линия задержки КИХ-фильтра на четыре отвода на умножителях с использованием метода правого сдвига и сложения

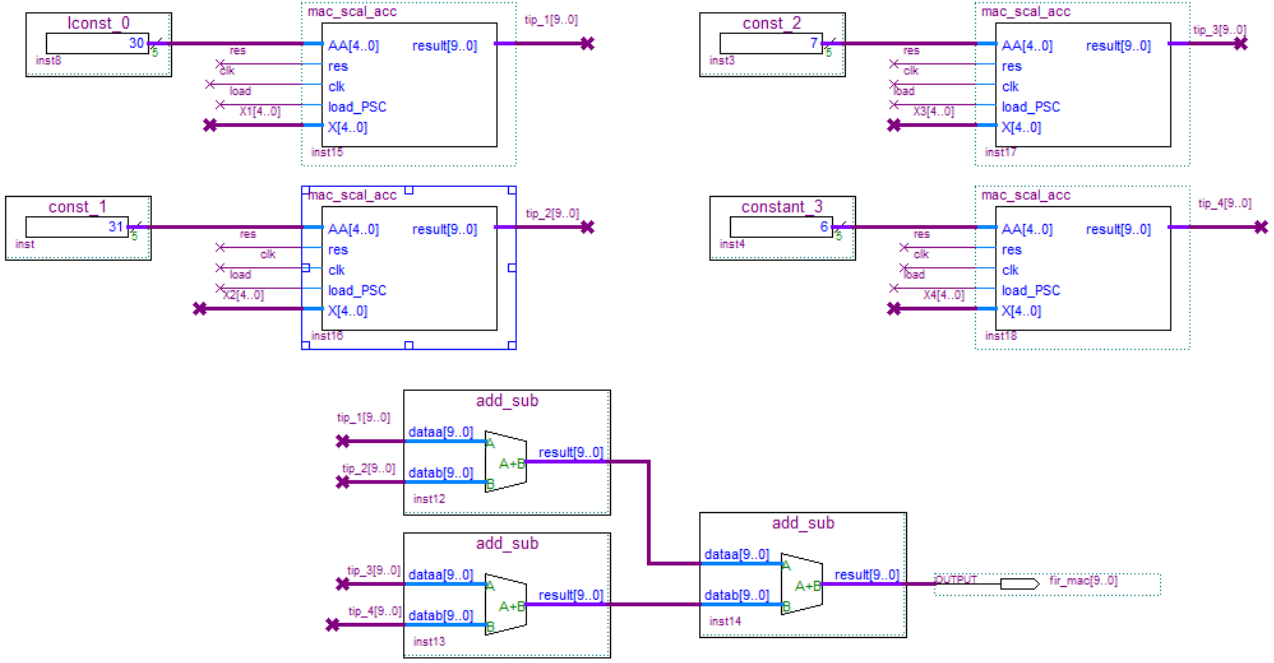


Рис. 2.28. Умножители с использованием метода правого сдвига и сложения и дерево сумматоров КИХ-фильтра на четыре отвода

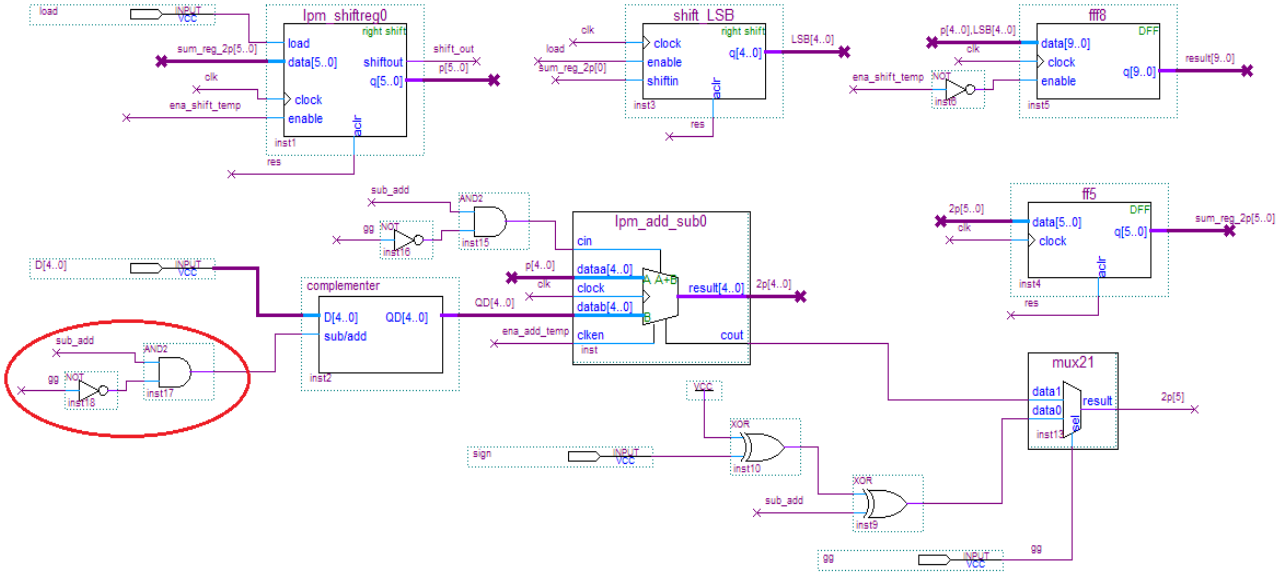


Рис. 2.29. Доработанная схема умножителя с использованием метода правого сдвига и сложения с учетом его работы в составе КИХ-фильтра

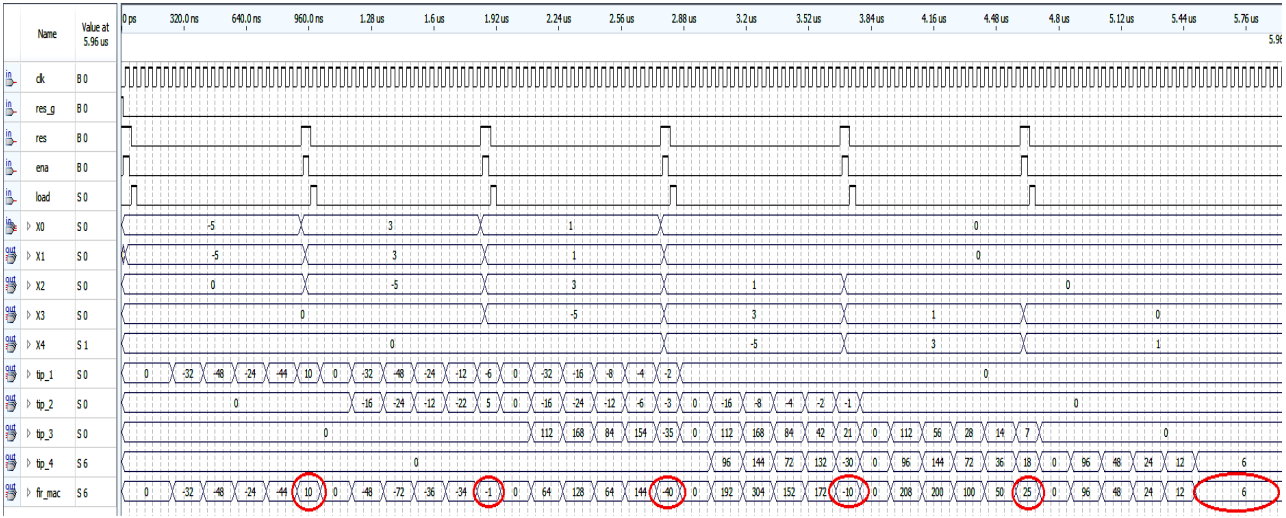


Рис. 2.30. Временные диаграммы работы КИХ-фильтра на четыре отвода на умножителях с использованием метода правого сдвига и сложения при поступлении входных отчетов -5, 3, 1 и 0. Результат: 10, -1, -40, -10, 25, 6

Таблица 2.3

КИХ-фильтр на четыре отвода, реализованный в базисе ПЛИС EP2C5F256C6, временная модель Slow-model

Технические характеристики	Разработанный MAC-блок	Программный умножитель на мегафункции ALTMEMMULT. Коэффициенты загружаются из внешнего порта, МГц	Программный умножитель на мегафункции ALTMEMMULT. Коэффициенты загружаются из памяти (умножитель на константу), МГц
Частота в наихудшем случае, Fmax	240	210	260

Повысить производительность КИХ-фильтра с использованием программного умножителя на мегафункции ALTMEMMULT позволяет вариант загрузки коэффициентов из блочной памяти ПЛИС (260 МГц).

Представленный в главе 1 (раздел 1.5) MAC-блок с использованием метода правого сдвига и сложения может быть использован для проектирования высокопроизводительных и компактных КИХ-фильтров небольшой разрядности. При этом на его реализацию в базисе ПЛИС потребуются незначительные логические ресурсы (менее 1 %) и будет наблюдаться экономия аппаратных и программных умножители. Недостатками такого MAC-блока являются трудоемкость его разработки и наличие своего интерфейса в отличие от унифицированных интерфейсов мегафункции ALTMEMMULT, что может оказаться неудобным в использовании.

2.4. Проектирование квантованных КИХ-фильтров

Рассмотрим два варианта извлечения кода языка VHDL из моделей расширения Simulink (среда моделирования систем непрерывного и дискретного времени) системы Matlab (существует еще вариант извлечения кода непосредственно из алгоритмов Matlab). Первый вариант извлечения кода из описания структуры фильтра базовыми элементами расширения Simulink. Второй вариант из описания структуры фильтра с помощью языка М-файлов Simulink.

На рис. 2.31 показана имитационная модель (верхний уровень иерархии) симметричного КИХ-фильтра на восемь отводов взятая из демонстрационного примера Simulink HDL Coder Examples Symmetric FIR Filters. На вход фильтра поступает сигнал, зашумленный шумом (2001 отсчет):

x_in = cos(2.*pi.*(0:0.001:2).*(1+(0:0.001:2).*75)).'.

Рассмотрим используемые параметры сигнала. Частота дискретизации сигнала $F_S = 1$ кГц ([0:1/fs:2] – вектор дискретных значений времени; 1+[0:1/fs:2].*75 – частота импульса). Шаг модельного времени (Sample time) – 1. Ниже показан пример расчета временной функции в системе Matlab.

```
>> fs=1e3; % частота дискретизации 1кГц
>> t=0:1/fs:2; % вектор дискретных значений
>> f0=1+[t*75]; % частота импульса
>> s1=cos(2.*pi.*t.*f0); зашумленный сигнал
>> plot(s1);
```

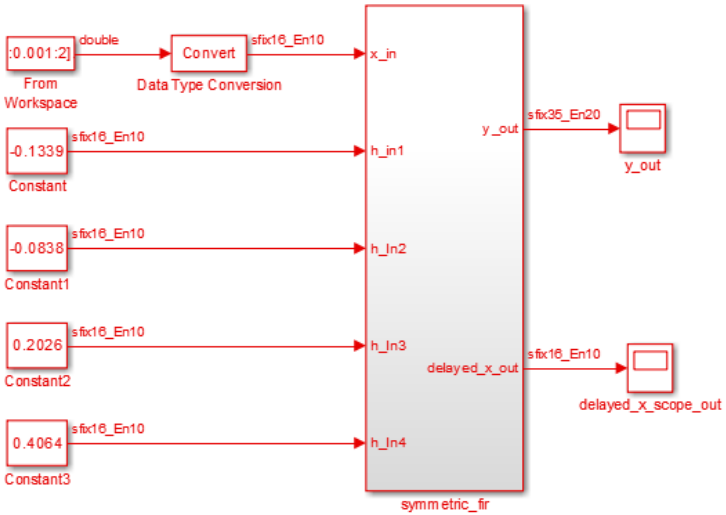
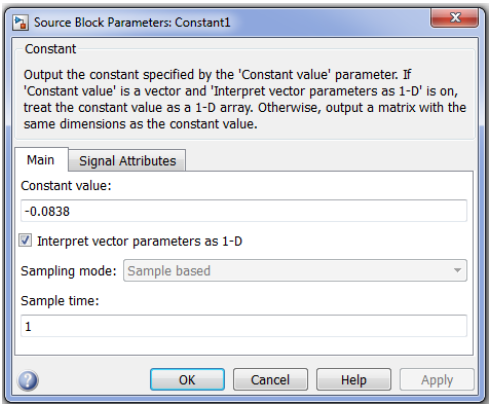


Рис. 2.31. Имитационная модель симметричного КИХ-фильтра на восемь отводов. Верхний уровень иерархии

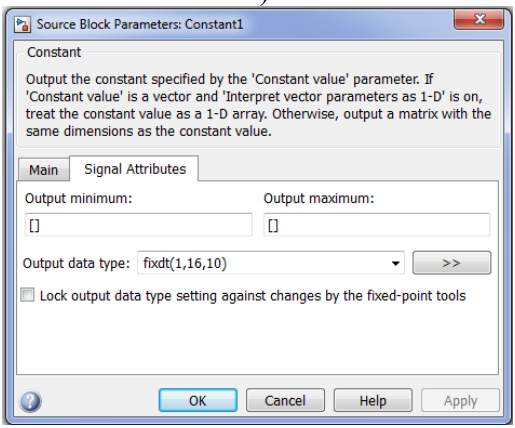
Предположим, что коэффициенты фильтра известны и хранятся в функциональных блоках Constant с одноименными именами Constant – Constant3. Выходные сигналы представляются в формате с фиксированной запятой fixdt(1,16,10): h1 = -0.1339; h2 = -0.0838; h3 = 0.2026; h4 = 0.4064. Преобразовывать значения из формата с плавающей запятой в формат с фиксированной запятой позволяют “автоматизированные мастера”, встроенные в функциональные блоки (рис. 2.32).

Коэффициенты фильтра и входной сигнал подлежащий фильтрации подвергаются масштабированию путем умножения на масштабный множитель 1024 (2^{10}) в соответствии с выбранным форматом 16.10 (sfix16_En10) и последующему округлению. Например, h1*1024= -137,1136 округляется до целого значения -137D или ff77 в дополнительном коде при длине машинного слова 16 бит). В шестнадцатеричной системе счисления с учетом знака числа они будут выглядеть следующим образом: ff77, ffaa, 00cf, 01a0.

При этом следует помнить, что формат квантования коэффициентов КИХ-фильтра влияет на его частотные и временные характеристики.



а)



б)

Рис. 2.32. Настройка функционального блока Constant с именем Constant1: а) задается вещественное число -0.083, являющееся одним из коэффициентов фильтра (закладка Main); б) выходной формат представления этого числа fixdt(1,16,10) (закладка Signal Attributes)

Значения входного сигнала, который необходимо

профильтровать, изменяются по амплитуде от -1 до +1. Они представлены в формате с фиксированной запятой (sfix16_En10). Функциональный блок Data Tupe Conversion осуществляет преобразование формата double в fixdt(1,16,10).

В дальнейшим необходимо предусмотреть деление коэффициентов и входных значений сигнала на 1024 или профильтрованных значений на 1048576. Например, начальные значения сигнала выглядят следующим образом: 0400, 03ff, 03ff, 03ff, 03ff, 03ff, 03fe. Для перевода в формат double их необходимо разделить на масштабный множитель 1024: 1, 0.9990234375 и т.д.

Рассмотрим имитационную модель КИХ-фильтра на рис. 2.33. После запуска модели над входным сигналом x_in, коэффициентами фильтра, выходными сигналами y_out и delayed_x_scope_out (выход линии задержки) указывается используемый формат.

При использовании арифметики с фиксированной запятой операции сложения не приводят к необходимости округления результатов – они могут лишь вызвать переполнение разрядной сетки. Поэтому выходы с пресумматоров Add-Add3 представляются в формате sfix17_En10 для учета переполнения. Выходы с умножителей Product –Product 3 представляются в формате sfix33_En20, т.к. умножение чисел с фиксированной запятой приводит к увеличению числа значащих цифр результата по сравнению с сомножителями, которые в данном случае 16- и 17-разрядные и следовательно - к необходимости округления. Выходы с сумматоров Add5-Add6 первого уровня дерева сумматоров представляются в формате sfix34_En20 а второго уровня - sfix35_En20. Следовательно, результат фильтрации (сигнал y_out) представляется в формате sfix35_En20 (рис. 2.33).

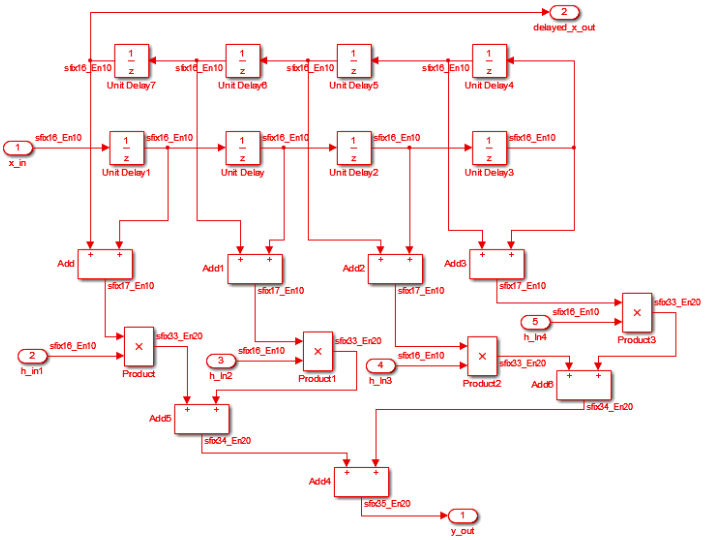


Рис. 2.33. Имитационная модель симметричного КИХ-фильтра на восемь отводов с указанием формата сигналов. Нижний уровень иерархии. Структура фильтра в элементах системы Matlab/Simulink

Далее с помощью приложения Simulink HDL Coder системы Matlab/Simulink извлечем в автоматическом режиме код языка VHDL КИХ-фильтра (пример 1). Запуск приложения осуществляется из меню Code/HDL Code/Generate HDL. Предварительно с помощью меню Code/HDL Code/Option необходимо заказать определенные опции и осуществить настройки.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ENTITY symmetric_fir IS
PORT( clk     :  IN    std_logic;
reset         :  IN    std_logic;
clk_enable    :  IN    std_logic;
```

```
x_in     :  IN   std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_in1    :  IN   std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_In2    :  IN   std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_In3    :  IN   std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_In4    :  IN   std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
ce_out   :  OUT  std_logic;
y_out    :  OUT  std_logic_vector(34 DOWNTO 0);  -- sfix35_En20
delayed_x_out: OUT std_logic_vector(15 DOWNTO 0)
 -- sfix16_En10);
END symmetric_fir;
ARCHITECTURE rtl OF symmetric_fir IS
-- Signals
 SIGNAL enb         : std_logic;
  SIGNAL x_in_signed  : signed(15 DOWNTO 0);  -- sfix16_En10
SIGNAL Unit_Delay1_out1 : signed(15 DOWNTO 0);
-- sfix16_En10
SIGNAL Unit_Delay_out1 : signed(15 DOWNTO 0);
 -- sfix16_En10
SIGNAL Unit_Delay2_out1 : signed(15 DOWNTO 0);
 -- sfix16_En10
SIGNAL Unit_Delay3_out1 : signed(15 DOWNTO 0);
-- sfix16_En10
SIGNAL Unit_Delay4_out1  : signed(15 DOWNTO 0);
 -- sfix16_En10
SIGNAL Unit_Delay5_out1  : signed(15 DOWNTO 0);
 -- sfix16_En10
SIGNAL Unit_Delay6_out1 : signed(15 DOWNTO 0);
 -- sfix16_En10
SIGNAL Unit_Delay7_out1 : signed(15 DOWNTO 0);
 -- sfix16_En10
SIGNAL Add_add_cast : signed(16 DOWNTO 0);  -- sfix17_En10
SIGNAL Add_add_cast_1 : signed(16 DOWNTO 0);
 -- sfix17_En10
SIGNAL Add_out1  : signed(16 DOWNTO 0);  -- sfix17_En10
SIGNAL Add1_add_cast: signed(16 DOWNTO 0);  -- sfix17_En10
SIGNAL Add1_add_cast_1 : signed(16 DOWNTO 0);
 …
…
```

```
SIGNAL h_In4_signed   signed(15 DOWNTO 0);  -- sfix16_En10
SIGNAL Product3_out1: signed(32 DOWNTO 0);  -- sfix33_En20
SIGNAL Add6_add_cast  : signed(33 DOWNTO 0);
 -- sfix34_En20
SIGNAL Add6_add_cast_1  : signed(33 DOWNTO 0);
 -- sfix34_En20
SIGNAL Add6_out1  : signed(33 DOWNTO 0);  -- sfix34_En20
SIGNAL Add4_add_cast: signed(34 DOWNTO 0);  -- sfix35_En20
SIGNAL Add4_add_cast_1: signed(34 DOWNTO 0);
 -- sfix35_En20
SIGNAL Add4_out1  : signed(34 DOWNTO 0);  -- sfix35_En20
          BEGIN
            x_in_signed <= signed(x_in);
            enb <= clk_enable;
            -- <S1>/Unit Delay1
            Unit_Delay1_process : PROCESS (clk, reset)
            BEGIN
              IF reset = '1' THEN
                Unit_Delay1_out1 <= to_signed(0, 16);
              ELSIF clk'EVENT AND clk = '1' THEN
                IF enb = '1' THEN
                  Unit_Delay1_out1 <= x_in_signed;
                END IF;
              END IF;
            END PROCESS Unit_Delay1_process;
            -- <S1>/Unit Delay
            Unit_Delay_process : PROCESS (clk, reset)
            BEGIN
              IF reset = '1' THEN
                Unit_Delay_out1 <= to_signed(0, 16);
              ELSIF clk'EVENT AND clk = '1' THEN
                IF enb = '1' THEN
                  Unit_Delay_out1 <= Unit_Delay1_out1;
                END IF;
              END IF;
            END PROCESS Unit_Delay_process;
            -- <S1>/Unit Delay2
            Unit_Delay2_process : PROCESS (clk, reset)
```

```
BEGIN
  IF reset = '1' THEN
    Unit_Delay2_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      Unit_Delay2_out1 <= Unit_Delay_out1;
    END IF;
  END IF;
END PROCESS Unit_Delay2_process;
-- <S1>/Unit Delay3
Unit_Delay3_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay3_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      Unit_Delay3_out1 <= Unit_Delay2_out1;
    END IF;
  END IF;
END PROCESS Unit_Delay3_process;
-- <S1>/Unit Delay4
Unit_Delay4_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay4_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      Unit_Delay4_out1 <= Unit_Delay3_out1;
    END IF;
  END IF;
END PROCESS Unit_Delay4_process;
-- <S1>/Unit Delay5
Unit_Delay5_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay5_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
```

```
      Unit_Delay5_out1 <= Unit_Delay4_out1;
    END IF;
  END IF;
END PROCESS Unit_Delay5_process;
-- <S1>/Unit Delay6
Unit_Delay6_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay6_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      Unit_Delay6_out1 <= Unit_Delay5_out1;
    END IF;
  END IF;
END PROCESS Unit_Delay6_process;
-- <S1>/Unit Delay7
Unit_Delay7_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    Unit_Delay7_out1 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      Unit_Delay7_out1 <= Unit_Delay6_out1;
    END IF;
  END IF;
END PROCESS Unit_Delay7_process;
-- <S1>/Add
Add_add_cast <= resize(Unit_Delay7_out1, 17);
Add_add_cast_1 <= resize(Unit_Delay1_out1, 17);
Add_out1 <= Add_add_cast + Add_add_cast_1;
-- <S1>/Add1
Add1_add_cast <= resize(Unit_Delay6_out1, 17);
Add1_add_cast_1 <= resize(Unit_Delay_out1, 17);
Add1_out1 <= Add1_add_cast + Add1_add_cast_1;
-- <S1>/Add2
Add2_add_cast <= resize(Unit_Delay5_out1, 17);
Add2_add_cast_1 <= resize(Unit_Delay2_out1, 17);
Add2_out1 <= Add2_add_cast + Add2_add_cast_1;
```

```
-- <S1>/Add3
Add3_add_cast <= resize(Unit_Delay4_out1, 17);
Add3_add_cast_1 <= resize(Unit_Delay3_out1, 17);
Add3_out1 <= Add3_add_cast + Add3_add_cast_1;
h_in1_signed <= signed(h_in1);
-- <S1>/Product
Product_out1 <= Add_out1 * h_in1_signed;
h_In2_signed <= signed(h_In2);
-- <S1>/Product1
Product1_out1 <= Add1_out1 * h_In2_signed;
-- <S1>/Add5
Add5_add_cast <= resize(Product_out1, 34);
Add5_add_cast_1 <= resize(Product1_out1, 34);
Add5_out1 <= Add5_add_cast + Add5_add_cast_1;
h_In3_signed <= signed(h_In3);
-- <S1>/Product2
Product2_out1 <= Add2_out1 * h_In3_signed;
h_In4_signed <= signed(h_In4);
-- <S1>/Product3
Product3_out1 <= Add3_out1 * h_In4_signed;
-- <S1>/Add6
Add6_add_cast <= resize(Product2_out1, 34);
Add6_add_cast_1 <= resize(Product3_out1, 34);
Add6_out1 <= Add6_add_cast + Add6_add_cast_1;
-- <S1>/Add4
Add4_add_cast <= resize(Add5_out1, 35);
Add4_add_cast_1 <= resize(Add6_out1, 35);
Add4_out1 <= Add4_add_cast + Add4_add_cast_1;
y_out <= std_logic_vector(Add4_out1);
delayed_x_out <= std_logic_vector(Unit_Delay7_out1);
ce_out <= clk_enable;
END rtl;
```

Пример 1. Фрагмент кода языка VHDL, извлеченный в автоматическом режиме с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов

Рассмотрим второй вариант. Разработаем имитационную модель симметричного КИХ-фильтра на восемь отводов в формате с фиксированной запятой с использованием fi-объектов и языка М-файлов системы Matlab (рис. 2.34 и рис. 2.35). Будем использовать следующий формат для представления десятичных чисел:

a = fi(v, s, w, f),

где v – десятичное число, s – знак (0 (false) – для чисел без знака и 1 (true) – для чисел со знаком), w - размер слова в битах (целая часть числа), f – дробная часть числа в битах. Это можно осуществить с использование следующего формата:

```
a = fi(v, s, w, f, fimath).
% HDL specific fimath
hdl_fm = fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
'SumMode', 'FullPrecision', 'SumWordLength', 32,...
'CastBeforeSum', true);
```

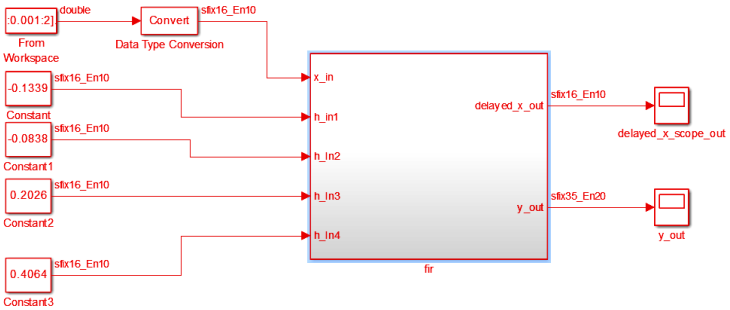


Рис. 2.34. Верхний уровень иерархии имитационной модели симметричного КИХ-фильтра на восемь отводов с использованием М-файла

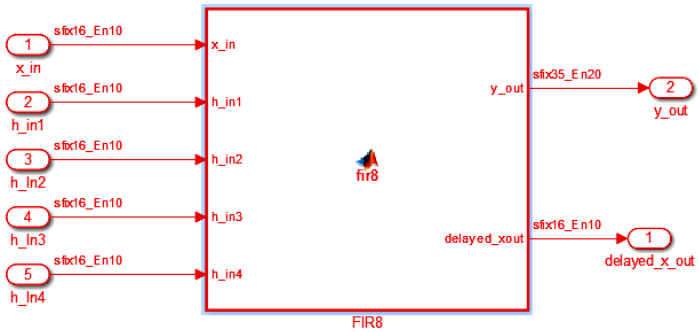


Рис. 2.35. Подсистема симметричного КИХ-фильтра на восемь отводов (подключение входных и выходных портов к М-файлу)

Данные настройки вычислений в формате с фиксированной запятой приняты в системе Simulink по умолчанию. Можно задать режим округления (Roundmode) – ‘floor’ – округление вниз; реакцию на переполнение (OverflowMode) – ‘wrap’ – перенос, при выходе значения v из допустимого диапазона, “лишние” старшие разряды игнорируются. При выполнении операций умножения (‘ProductMode’) и сложения (SumMode) для повышения точности вычислений (precision) используется машинное слово шириной в 32 бита.

Пример 2 показывает М-файл симметричного КИХ-фильтра на восемь отводов с использованием fi-объектов. На рис. 2.36 показан сигнал до и после фильтрации.

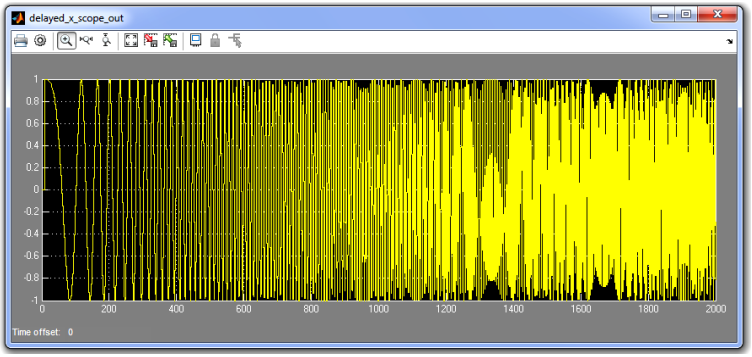
```
%#codegen
function [y_out, delayed_xout] = fir8(x_in, h_in1, h_in2, h_in3, h_in4)
% Symmetric FIR Filter
% HDL specific fimath
hdl_fm = fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
```

```
'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
'SumMode', 'FullPrecision', 'SumWordLength', 32,...
'CastBeforeSum', true);
% declare and initialize the delay registers
persistent ud1 ud2 ud3 ud4 ud5 ud6 ud7 ud8;
if isempty(ud1)
   ud1 = fi(0, 1, 16, 10, hdl_fm);
   ud2 = fi(0, 1, 16, 10, hdl_fm);
   ud3 = fi(0, 1, 16, 10, hdl_fm);
   ud4 = fi(0, 1, 16, 10, hdl_fm);
   ud5 = fi(0, 1, 16, 10, hdl_fm);
   ud6 = fi(0, 1, 16, 10, hdl_fm);
   ud7 = fi(0, 1, 16, 10, hdl_fm);
   ud8 = fi(0, 1, 16, 10, hdl_fm);
end
% access the previous value of states/registers
a1 = fi(ud1 + ud8, 1, 17, 10, hdl_fm);
a2 = fi(ud2 + ud7, 1, 17, 10, hdl_fm);
a3 = fi(ud3 + ud6, 1, 17, 10, hdl_fm);
a4 = fi(ud4 + ud5, 1, 17, 10, hdl_fm);
% multiplier chain
m1 = fi((h_in1*a1), 1, 33, 20, hdl_fm);
m2 = fi((h_in2*a2), 1, 33, 20, hdl_fm);
m3 = fi((h_in3*a3), 1, 33, 20, hdl_fm);
m4 = fi((h_in4*a4), 1, 33, 20, hdl_fm);
% adder chain
a5 = fi(m1 + m2, 1, 34, 20, hdl_fm);
a6 = fi(m3 + m4, 1, 34, 20, hdl_fm);
% filtered output
y_out = fi(a5 + a6, 1, 35, 20, hdl_fm);
% delayout input signal
delayed_xout = ud8;
% update the delay line
ud8 = ud7;  ud7 = ud6; ud6 = ud5;
ud5 = ud4;
ud4 = ud3;
ud3 = ud2;
ud2 = ud1;
```

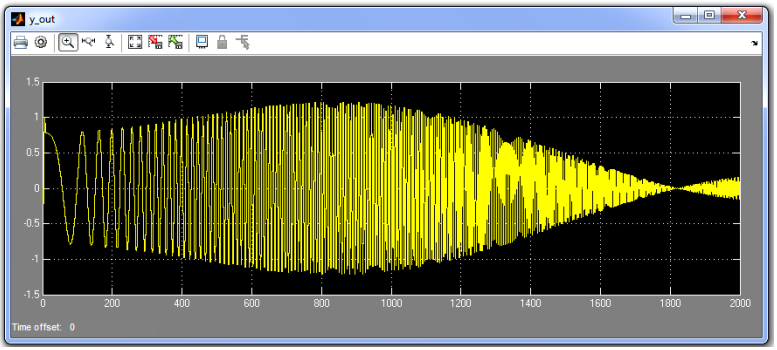
```
ud1 = fi(x_in, 1, 16, 10, hdl_fm);
end
```

Пример 2. М-файл симметричного КИХ-фильтра на восемь отводов с использованием fi-объектов

Пример 3 демонстрирует код языка VHDL, извлеченный в автоматическом режиме с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе М-файлов и fi-объектов.



а)



б)

Рис. 2.36. Результаты имитационного моделирования: а – исходный сигнал; б - профильтрованный сигнал

```
-- File Name: C:\fir\fir8_vhdl\FIR8.vhd
-- Created: 2014-03-04 15:01:23
-- Generated by MATLAB 8.0 and HDL Coder 3.1
-- Module: FIR8
-- Source Path: fir_new/fir/FIR8
-- Hierarchy Level: 1
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ENTITY FIR8 IS
  PORT( clk               :   IN    std_logic;
reset                     :   IN    std_logic;
enb                       :   IN    std_logic;
x_in   :   IN    std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_in1  :  IN    std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_in2 :   IN    std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_in3 :   IN    std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
h_in4 :   IN    std_logic_vector(15 DOWNTO 0);  -- sfix16_En10
y_out :   OUT   std_logic_vector(34 DOWNTO 0);  -- sfix35_En20
delayed_xout:   OUT   std_logic_vector(15 DOWNTO 0)
 -- sfix16_En10);
END FIR8;
ARCHITECTURE rtl OF FIR8 IS
  -- Signals
  SIGNAL x_in_signed : signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL h_in1_signed: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL h_in2_signed: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL h_in3_signed: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL h_in4_signed: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL y_out_tmp: signed(34 DOWNTO 0);  -- sfix35_En20
  SIGNAL delayed_xout_tmp: signed(15 DOWNTO 0);
 -- sfix16_En10
  SIGNAL ud1: signed(15 DOWNTO 0);  -- sfix16
  SIGNAL ud2 : signed(15 DOWNTO 0);  -- sfix16
  SIGNAL ud3 : signed(15 DOWNTO 0);  -- sfix16
  SIGNAL ud4: signed(15 DOWNTO 0);  -- sfix16
  SIGNAL ud5: signed(15 DOWNTO 0);  -- sfix16
  SIGNAL ud6: signed(15 DOWNTO 0);  -- sfix16
```

```
  SIGNAL ud7: signed(15 DOWNTO 0);  -- sfix16
  SIGNAL ud8: signed(15 DOWNTO 0);  -- sfix16
  SIGNAL ud2_next: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL ud3_next: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL ud4_next: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL ud5_next: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL ud6_next : signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL ud7_next: signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL ud8_next : signed(15 DOWNTO 0);  -- sfix16_En10
  SIGNAL a1: signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL a2: signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL a3: signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL a4: signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL m1: signed(32 DOWNTO 0);  -- sfix33_En20
  SIGNAL m2: signed(32 DOWNTO 0);  -- sfix33_En20
  SIGNAL m3: signed(32 DOWNTO 0);  -- sfix33_En20
  SIGNAL m4  : signed(32 DOWNTO 0);  -- sfix33_En20
  SIGNAL a5: signed(33 DOWNTO 0);  -- sfix34_En20
  SIGNAL a6: signed(33 DOWNTO 0);  -- sfix34_En20
  SIGNAL add_cast: signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_1  : signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_2: signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_3 : signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_4 : signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_5 : signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_6 : signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_7  : signed(16 DOWNTO 0);  -- sfix17_En10
  SIGNAL add_cast_8 : signed(33 DOWNTO 0);  -- sfix34_En20
  SIGNAL add_cast_9: signed(33 DOWNTO 0);  -- sfix34_En20
  SIGNAL add_cast_10: signed(33 DOWNTO 0);  -- sfix34_En20
  SIGNAL add_cast_11 : signed(33 DOWNTO 0);  -- sfix34_En20
  SIGNAL add_cast_12 : signed(34 DOWNTO 0);  -- sfix35_En20
  SIGNAL add_cast_13: signed(34 DOWNTO 0);  -- sfix35_En20
BEGIN
  x_in_signed <= signed(x_in);
  h_in1_signed <= signed(h_in1);
  h_in2_signed <= signed(h_in2);
  h_in3_signed <= signed(h_in3);
```

```
h_in4_signed <= signed(h_in4);
FIR8_1_process : PROCESS (clk, reset)
BEGIN
  IF reset = '1' THEN
    ud1 <= to_signed(0, 16);
    ud2 <= to_signed(0, 16);
    ud3 <= to_signed(0, 16);
    ud4 <= to_signed(0, 16);
    ud5 <= to_signed(0, 16);
    ud6 <= to_signed(0, 16);
    ud7 <= to_signed(0, 16);
    ud8 <= to_signed(0, 16);
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      ud2 <= ud2_next;
      ud3 <= ud3_next;
      ud4 <= ud4_next;
      ud5 <= ud5_next;
      ud6 <= ud6_next;
      ud7 <= ud7_next;
      ud8 <= ud8_next;
      ud1 <= x_in_signed;
    END IF;
  END IF;
END PROCESS FIR8_1_process;
--MATLAB Function 'fir/FIR8': '<S2>:1'
-- Symmetric FIR Filter
-- HDL specific fimath
-- declare and initialize the delay registers
-- access the previous value of states/registers
--'<S2>:1:27'
add_cast <= resize(ud1, 17);
add_cast_1 <= resize(ud8, 17);
a1 <= add_cast + add_cast_1;
--'<S2>:1:28'
add_cast_2 <= resize(ud2, 17);
add_cast_3 <= resize(ud7, 17);
a2 <= add_cast_2 + add_cast_3;
```

```
--'<S2>:1:29'
add_cast_4 <= resize(ud3, 17);
add_cast_5 <= resize(ud6, 17);
a3 <= add_cast_4 + add_cast_5;
--'<S2>:1:30'
add_cast_6 <= resize(ud4, 17);
add_cast_7 <= resize(ud5, 17);
a4 <= add_cast_6 + add_cast_7;
-- multiplier chain
--'<S2>:1:33'
m1 <= h_in1_signed * a1;
--'<S2>:1:34'
m2 <= h_in2_signed * a2;
--'<S2>:1:35'
m3 <= h_in3_signed * a3;
--'<S2>:1:36'
m4 <= h_in4_signed * a4;
-- adder chain
--'<S2>:1:39'
add_cast_8 <= resize(m1, 34);
add_cast_9 <= resize(m2, 34);
a5 <= add_cast_8 + add_cast_9;
--'<S2>:1:40'
add_cast_10 <= resize(m3, 34);
add_cast_11 <= resize(m4, 34);
a6 <= add_cast_10 + add_cast_11;
-- filtered output
--'<S2>:1:43'
add_cast_12 <= resize(a5, 35);
add_cast_13 <= resize(a6, 35);
y_out_tmp <= add_cast_12 + add_cast_13;
-- delayout input signal
--'<S2>:1:46'
delayed_xout_tmp <= ud8;
-- update the delay line
--'<S2>:1:49'
ud8_next <= ud7;
--'<S2>:1:50'
```

```
  ud7_next <= ud6;
  --'<S2>:1:51'
  ud6_next <= ud5;
  --'<S2>:1:52'
  ud5_next <= ud4;
  --'<S2>:1:53'
  ud4_next <= ud3;
  --'<S2>:1:54'
  ud3_next <= ud2;
  --'<S2>:1:55'
  ud2_next <= ud1;
  --'<S2>:1:56'
  y_out <= std_logic_vector(y_out_tmp);
  delayed_xout <= std_logic_vector(delayed_xout_tmp);
END rtl;
```

Пример 3. Код языка VHDL извлеченный в автоматическом режиме с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе М-файлов и fi-объектов

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.numeric_std.ALL;
USE work.fir_tb_pkg.ALL;
PACKAGE fir_tb_data IS
CONSTANT Data_Type_Conversion_out1_force :
Data_Type_Conversion_out1_type;
CONSTANT Constant_out1_force : std_logic_vector(15 DOWNTO 0);
CONSTANT Constant1_out1_force : std_logic_vector(15 DOWNTO 0);
CONSTANT Constant2_out1_force : std_logic_vector(15 DOWNTO 0);
CONSTANT Constant3_out1_force : std_logic_vector(15 DOWNTO 0);
CONSTANT delayed_x_out_expected :
Data_Type_Conversion_out1_type;
CONSTANT y_out_expected : y_out_type;
END fir_tb_data;
PACKAGE BODY fir_tb_data IS
        -- Входной сигнал
CONSTANT                Data_Type_Conversion_out1_force                :
```

```
Data_Type_Conversion_out1_type :=
            (
                X"0400",
                X"03ff",
                X"03ff",
                X"03ff",
                X"03ff",
                X"03ff",
                X"03fe",
                X"03fd",
                X"03fc",
                X"03fb",
                X"03f9",
                X"03f7",
                X"03f5",
                X"03f2",
                X"03ef",
                X"03eb",
        -- Коэффициенты фильтра
CONSTANT Constant_out1_force : std_logic_vector(15 DOWNTO 0)
:=( X"ff77");
CONSTANT Constant1_out1_force : std_logic_vector(15 DOWNTO 0)
:=(X"ffaa");
CONSTANT Constant2_out1_force : std_logic_vector(15 DOWNTO 0)
:=(X"00cf");
CONSTANT Constant3_out1_force : std_logic_vector(15 DOWNTO 0)
:=(X"01a0");
CONSTANT delayed_x_out_expected :
Data_Type_Conversion_out1_type :=
            (  X"0000",
               X"0000",
                X"0000",
                X"0000",
                X"0000",
                X"0000",
                X"0000",
                X"0000",
                X"0400",
```

```
        X"03ff",
        X"03ff",
        X"03ff",
-- Отклик фильтра, профильтрованные значения
CONSTANT y_out_expected : y_out_type :=
   (
        SLICE(X"000000000",35),
        SLICE(X"7fffddc00",35),
        SLICE(X"7fffc8489",35),
        SLICE(X"7ffffc0df",35),
        SLICE(X"000064010",35),
        SLICE(X"0000cbe70",35),
        SLICE(X"0000ff8d0",35),
        SLICE(X"0000ea08a",35),
        SLICE(X"0000c7dbf",35),
        SLICE(X"0000c7e58",35),
        SLICE(X"0000c7cc8",35),
……
```

Пример 4. Фрагмент тестбенча полученный с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе М-файлов и fi-объектов

Рассмотрим функциональное моделирование КИХ-фильтра с использованием симулятора ModelSim-Altera STARTER EDITION. На рис. 2.37 показано моделирование фильтра с использованием тестбенча (пример 4), полученного с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе М-файлов и fi-объектов.

Используя полученный код (пример 3), разработаем функциональную модель в САПР ПЛИС Quartus II (рис. 2.38). Входной сигнал сформируем с помощью векторного редактора (рис. 2.39) в соответствии с примером 4. Проект размещен в ПЛИС EP2C5AF256A7 серии Cyclone II. На рис. 2.40 показано распределение задействованных ресурсов

проекта по кристаллу ПЛИС EP2C5AF256A7. Используются восемь аппаратных умножителей размерностью операндов 9x9. Что равносильно использованию четырех умножителей размерностью операндов 18x18 (табл. 2.4).

Таблица 2.4

Общие сведения по числу задействованных ресурсов ПЛИС Cyclone II EP2C5AF256A7

Логические элементы (Logic Cells, LC)	Триггеры логических элементов (Dedicated logic registers)	Аппаратные умножители размерностью операндов 9x9 (Embedded Multiplier 9-bit elements)	Рабочая частота в наихудшем случае Fmax, МГц
238/4608 (5 %)	128/4608 (3 %)	8/26 (31 %) или 4 умножителя размерностью 18x18	380

Сравнивая рис. 2.37 и рис. 2.39, видим, что функциональная модель КИХ-фильтра на восемь отводов, построенная с использованием кода языка VHDL, извлеченного в автоматическом режиме с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе M-файлов и fi-объектов в САПР ПЛИС Quartus II версии 13.0, работает корректно.

В состав HDL Coder входит инструмент под названием HDL Workflow Advisor (помощник по работе с HDL), который автоматизирует программирование ПЛИС фирм Xilinx и Altera. Можно контролировать HDL-архитектуру и реализацию, выделять критические пути и генерировать отчеты об использовании аппаратных ресурсов.

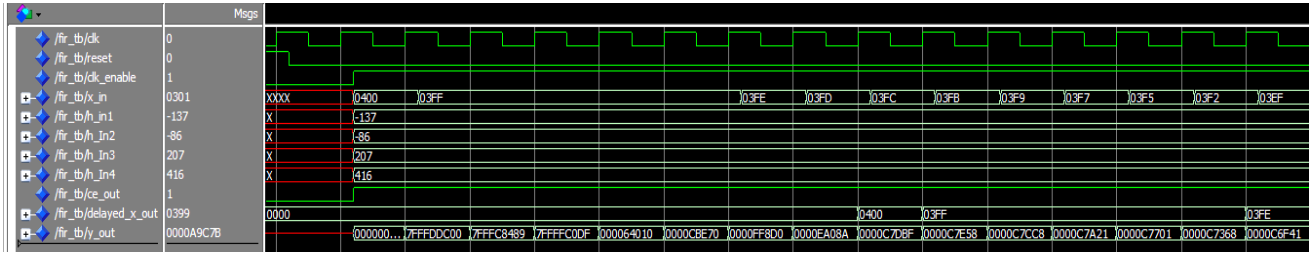


Рис. 2.37. Функциональное моделирование фильтра с использованием тестбенча, полученного с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе М-файлов и fi-объектов

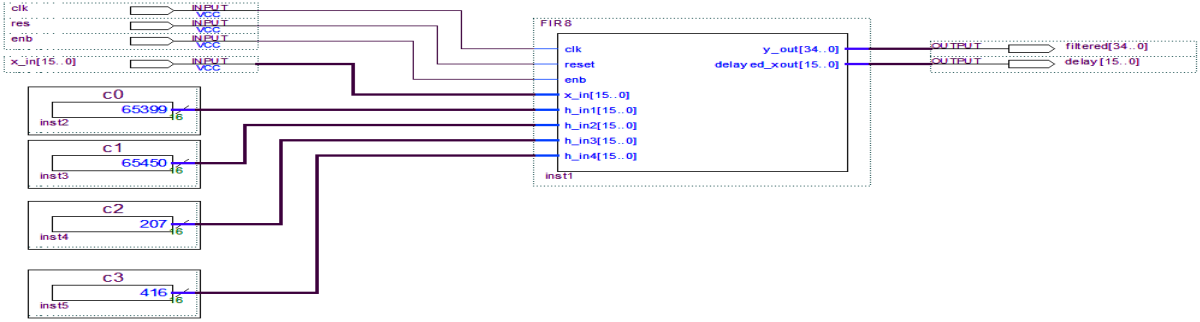


Рис. 2.38. Функциональная модель КИХ-фильтра на восемь отводов, построенная с использованием кода языка VHDL, извлеченного в автоматическом режиме с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе М-файлов и fi-объектов в САПР ПЛИС Quartus II версии 13.0

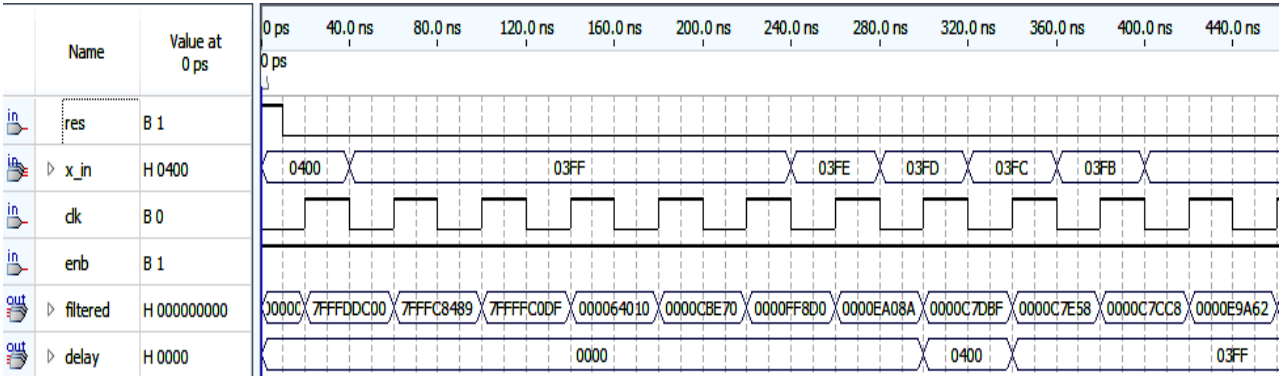


Рис. 2.39. Функциональное моделирование КИХ-фильтра с использованием кода языка VHDL, извлеченного в автоматическом режиме с помощью приложения Simulink HDL Coder из имитационной модели симметричного КИХ-фильтра на восемь отводов на основе M-файлов и fi-объектов

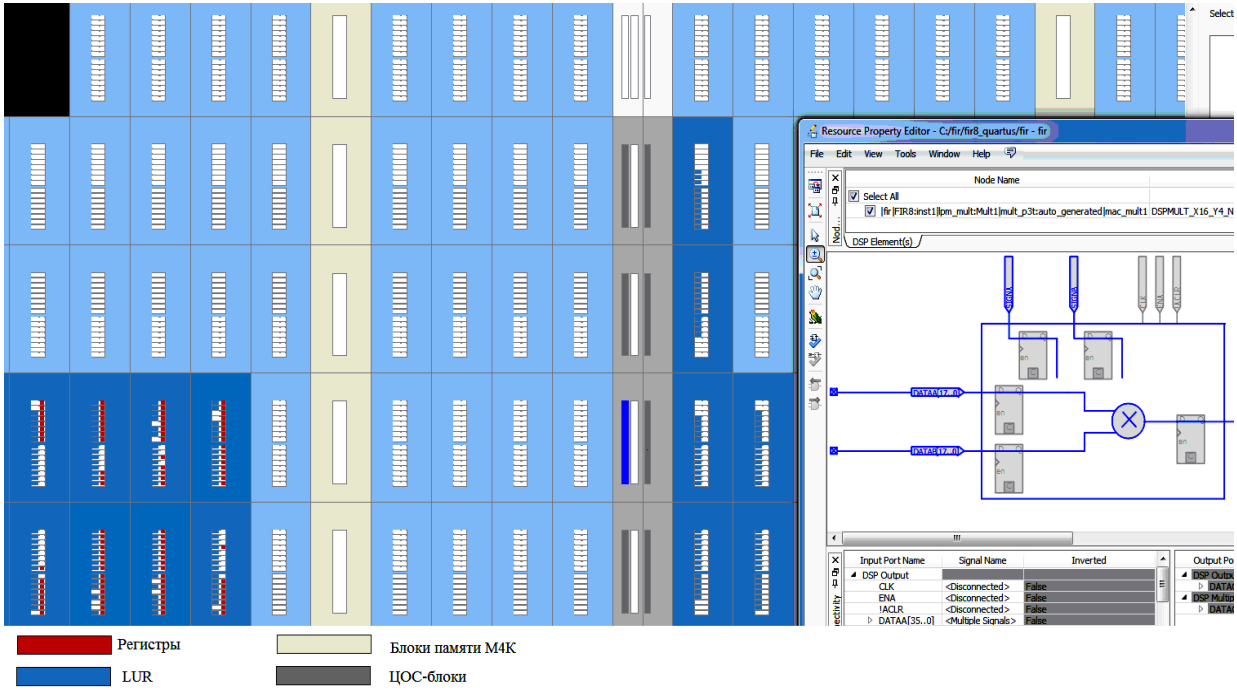


Рис. 2.40. Распределение задействованных ресурсов проекта
по кристаллу ПЛИС EP2C5AF256A7

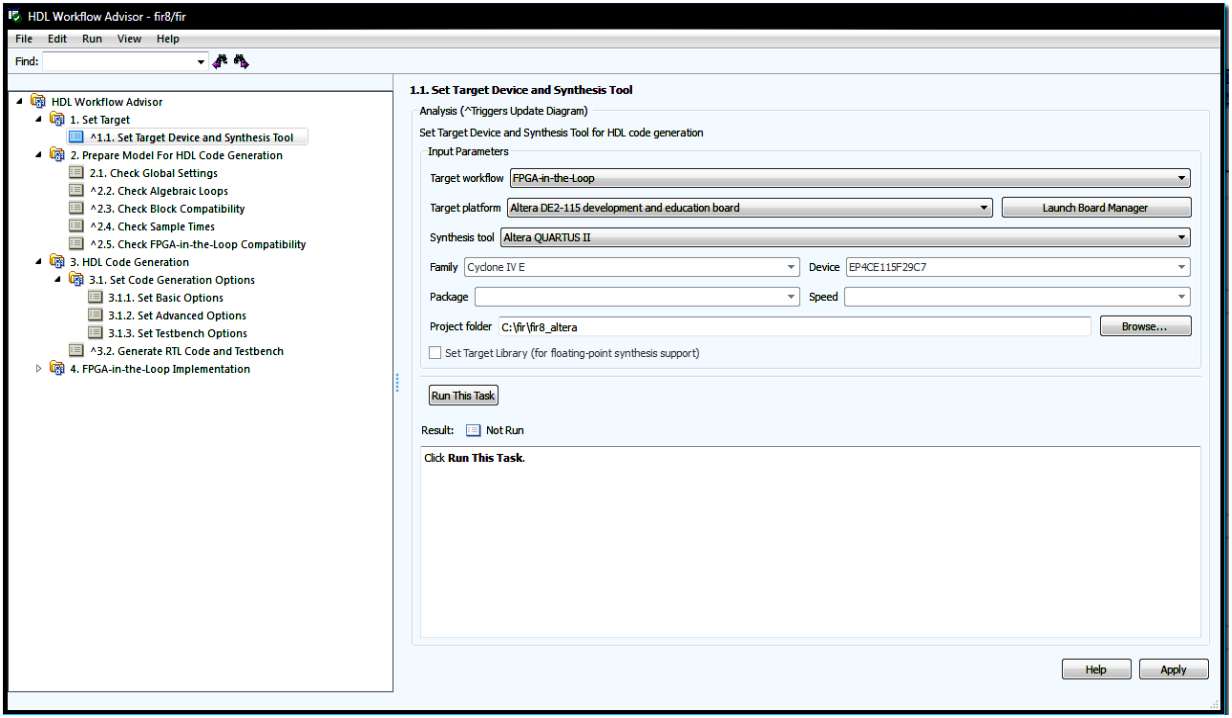


Рис. 2.41. Программный инструмент Workflow Advisor расширения Simulink

На рис. 2.41 показан программный инструмент HDL Workflow Advisor расширения Simulink. Более подробную информацию можно получить на сайте matlab.ru. Программный инструмент HDL Workflow Advisor дублирует пункты меню Code/HDL Code/Generate HDL и Code/HDL Code/Option, но позволяет работать на более качественном уровне.

Генерация кода происходит в несколько этапов. Первый этап заключается в выборе стиля проектирования – реализация проекта в базисе заказных БИС/ПЛИС без привязки к конкретному производителю или реализация проекта с выбором отладочной платы и серии ПЛИС фирм Xilinx или Altera. Для проектов в базисе ПЛИС Workflow Advisor обеспечивает такие возможности оптимизации, как площадь-скорость, распределение памяти RAM, конвейеризация, совместное использование ресурсов и развертывание циклов.

Например, в разделе Set Target выбирается отладочная плата Altera DE2-115 Development and Education Board на ПЛИС серии Cyclone IV EP4CE115. В случае выбора стиля проектирования FPGA-in-the-Loop (замкнутый цикл) отлаживать проект возможно непосредственно из Simulink. Проверка на поддержку отладочной платы Workflow Advisor можно осуществить на втором этапе подготовки модели для генерации кода Prepare Model For HDL Code Generation в разделе Chek FPGA-in-the-Loop Compatibility. На третьем этапе осуществляется генерация кода.

В данном разделе рассмотрены расчет спецификации КИХ-фильтров в системе Matlab/Simulink с применением пакета Signal Processing среды FDATool; различные варианты проектирования параллельных КИХ-фильтров с использованием мегафункций САПР ПЛИС Quartus II компании Altera и др.

Показано, что КИХ-фильтр на мегафункции ALTMULT_ACCUM с использованием четырех блоков

умножения с накоплением является самым затратным. Наиболее оптимальным по числу используемых ресурсов ПЛИС является его модификация, позволяющая построить параллельный КИХ-фильтр на 4 отвода с использованием всего лишь одного блока со встроенными перемножителями в количестве четырех штук, двумя сумматорами, сумматором-аккумулятором и линией задержки.

Мегафункции ALTMULT_ADD так же позволяет построить параллельный КИХ-фильтр с использованием всего лишь одного блока со встроенными перемножителями и сумматорами выполняющего функцию умножения и сложения. Использование внешней линии задержки из трех регистров приводит к незначительному увеличению ресурсов и не сказывается на быстродействие. Экономия ресурсов ПЛИС в первом модифицированном и во втором вариантах достигается за счет использования встроенных четырех аппаратных умножителей размерностью 18x18.

Фильтр на мегафункции ALTMEMMULT с загрузкой коэффициентов из внешнего порта обладает пониженным быстродействием. Использование же блочной памяти (рис. 2.22) для хранения коэффициентов фильтра внутри ПЛИС значительно упрощает процесс разработки и не приводит к существенному увеличению ресурсов за счет использования внешней линии задержки на четырех регистрах, дополнительных трех однотипных многоразрядных сумматоров и не снижает быстродействие проекта.

Система Matlab/simulink с приложением Simulink HDL Coder может быть эффективно использована для ускорения процесса разработки квантованных КИХ-фильтров в базисе ПЛИС. Автоматически извлеченный VHDL-код фильтра из описания Simulink-модели приводит к использованию встроенных ЦОС-блоков в ПЛИС серии Cyclone II, обеспечивая разработку высокопроизводительных КИХ-фильтров.

3. ПРОЕКТИРОВАНИЕ КИХ-ФИЛЬТРОВ НА РАСПРЕДЕЛЕННОЙ АРИФМЕТИКЕ

3.1. КИХ-фильтры на последовательной распределенной арифметике

Распределенная арифметика широко используется при проектировании высокопроизводительных КИХ- и БИХ-фильтров, адаптивных фильтров, специальных вычислителей например, с применением быстрого преобразования Фурье, дискретного вейвлет-преобразования и др., для реализации мультимедиа систем в базисе ПЛИС. Поэтому представляет определенный интерес рассмотреть основы такой арифметики на примере проектирования КИХ-фильтра на четыре отвода.

В ЦОС-приложениях коэффициенты фильтра могут быть представлены как положительными, так и отрицательными числами (целочисленными значениями со знаком), в свою очередь, информационные сигналы, поступающие на вход фильтра также могут быть представлены как положительными, так и отрицательными числами. Поэтому важно рассмотреть такие понятия, как дополнение до единицы и дополнение до двух, т.е. обратный и дополнительный коды, а также понятие “расширение знака”. Дополнение до двух наиболее эффективно в операциях умножения и накопления чисел со знаком.

Уравнение КИХ-фильтра (нерекурсивного цифрового фильтра с конечно-импульсной характеристикой) представляется как арифметическая сумма произведений:

$$y = \sum_{k=0}^{K-1} C_k \cdot x_k \,, \qquad (3.1)$$

где $y-$ отклик цепи; $x_k -$ $k-$я входная переменная; $C_k -$весовой коэффициент $k-$й входной переменной,

который является постоянным для всех n; K - число отводов фильтра.

На рис. 3.1 показана прямая реализация КИХ-фильтра по формуле $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$ с использованием сдвиговых регистров для организации линии задержки на четыре отвода (такой функциональный узел называют многоразрядный сдвиговый регистр), перемножителей для умножения сигналов на константы и одного сумматора с внутренней организацией “дерево сумматоров”. На рис. 3.2 показана общепринятая методика умножения с накоплением (MAC), характерная для реализации в базисе сигнальных процессоров, используемая для построения КИХ-фильтра на четыре отвода из-за отсутствия встроенных умножителей.

На рис. 3.3 показаны один из вариантов построения блока умножения с накоплением и алгоритм реализации умножения методом сдвига и сложения с накоплением. Демонстрируется аппаратная реализация умножения числа x (0101, D5) на константу C (1011, D11) с использованием многоразрядного мультиплексора 2 в 1, на один из входов которого подается константа D11, а на другой – ноль, и масштабирующего аккумулятора (сумматора) для суммирования частичных произведений с соответствующими весами. На адресный вход мультиплексора с помощью сдвигового регистра подается число x (D5) младшими разрядами вперед. Результатом умножения является десятичное число 55. Рис. 3.4 показывает умножение десятичного числа 11 на 5 методом правого сдвига с накоплением по рекуррентной формуле, показанной на рис. 3.3.

Рассмотрим проектирование КИХ-фильтров в базисе ПЛИС с использованием распределенной арифметики. Преимущество последовательной распределенной арифметики, реализованной в базисе ПЛИС, заключается в снижении объема задействованных ресурсов за счет отказа от использования встроенных умножителей. Структура КИХ-фильтра на четыре отвода будет состоять из одной LUT-

таблицы, содержащей комбинацию сумм коэффициентов, являющихся константами, всех возможных вариантов на ее адресных входах, накапливающего (масштабирующего) сумматора и многоразрядного сдвигового регистра (рис. 3.5).

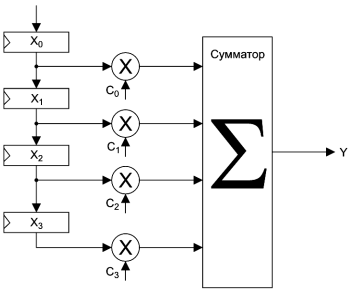


Рис. 3.1. Прямая реализация КИХ-фильтра на четыре отвода

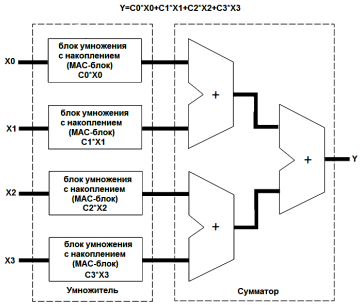


Рис. 3.2. Параллельный алгоритм реализации уравнения КИХ-фильтра на четыре отвода с использованием четырех блоков умножения с накоплением

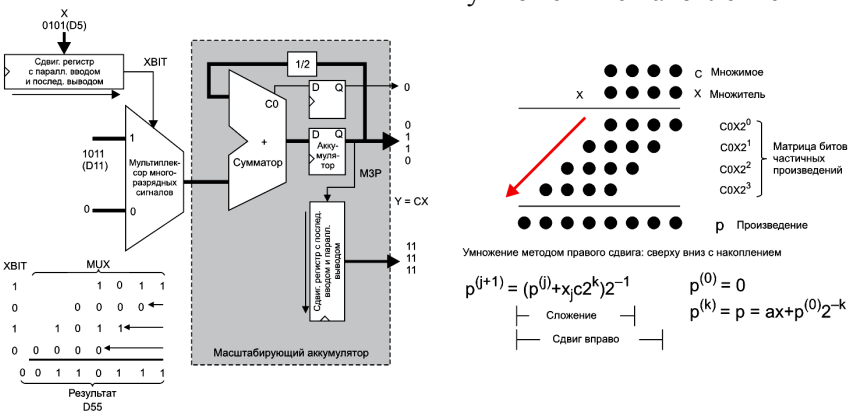


Рис. 3.3. Блок умножения с накоплением и алгоритм реализации умножения методом сдвига и сложения с накоплением

C	1 0 1 1	
X	0 1 0 1	
$p^{(0)}$	0 0 0 0	
$+X_0C$	1 0 1 1	
$2p^{(1)}$	0 1 0 1 1	
$p^{(1)}$	0 1 0 1	1
$+X_1C$	0 0 0 0	
$2p^{(2)}$	0 0 1 0 1	1
$p^{(2)}$	0 0 1 0	1 1
$+X_2C$	1 0 1 1	
$2p^{(3)}$	0 1 1 0 1	1 1
$p^{(3)}$	0 1 1 0	1 1 1
$+X_3C$	0 0 0 0	
$2p^{(4)}$	0 0 1 1 0	1 1 1
$p^{(4)}$	0 0 1 1	0 1 1 1

Рис. 3.4. Умножение десятичного числа 11 на 5 методом правого сдвига с накоплением

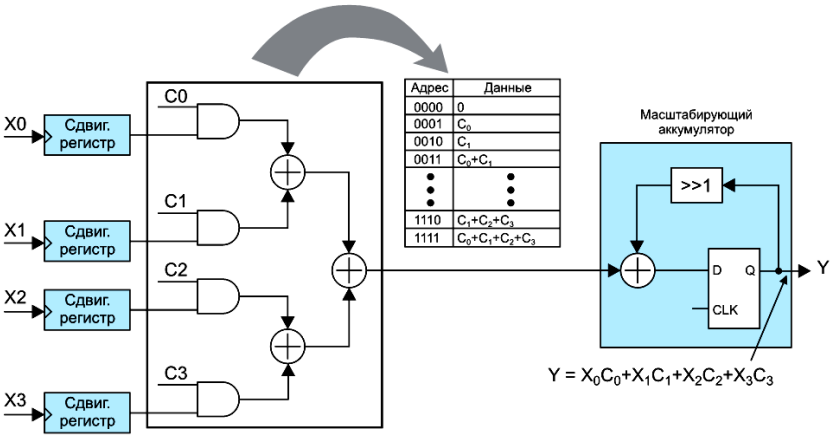


Рис. 3.5. Идея использования распределенной арифметики на примере КИХ-фильтра на четыре отвода

Если рассматривать входные переменные x_k как целые десятичные числа со знаком в дополнительном двоичном коде, то

$$x_k = -x_{k(\text{B}-1)} 2^{B-1} + \sum_{b=0}^{B-2} x_{kb} \cdot 2^b \,, \tag{3.2}$$

где B – разрядность кода. Подставим выражение (3.2) в (3.1), получим:

$$\begin{aligned} y &= \sum_{k=0}^{K-1} C_k \cdot \left[- x_{k(\text{B}-1)} 2^{B-1} + \sum_{b=0}^{B-2} x_{kb} \cdot 2^b \right] = \\ &= -\sum_{k=0}^{K-1} x_{k(B-1)} 2^{B-1} C_k + \sum_{k=0}^{K-1} \sum_{b=0}^{B-2} x_{kb} 2^b C_k \end{aligned} \,. \tag{3.3}$$

Раскроем все суммы в выражении (3.3) и сгруппируем числа по степеням B:

$$\begin{aligned} y &= \left[x_{00} \cdot C_0 + x_{10} \cdot C_1 + x_{20} \cdot C_2 + \ldots + x_{(K\text{-}1)0} \cdot C_{K-1} \right] \\ &+ \left[x_{01} \cdot C_0 + x_{11} \cdot C_1 + x_{21} \cdot C_2 + \ldots + x_{(K-1)1} \cdot C_{K-1} \right] \cdot 2^1 \\ &+ \left[x_{02} \cdot C_0 + x_{12} \cdot C_1 + x_{22} \cdot C_2 + \ldots + x_{(K-1)2} \cdot C_{K-1} \right] \cdot 2^2 \\ &\ldots. \\ &\ldots. \\ &+ \left[x_{0(B-2)} \cdot C_0 + x_{1(B-2)} \cdot C_1 + x_{2(B-2)} \cdot C_2 + \ldots + x_{(K-1)(B-2)} \cdot C_{K-1} \right] \cdot 2^{B-2} \\ &- \left[x_{0(B-1)} \cdot C_0 + x_{1(B-1)} \cdot C_1 + x_{2(B-1)} \cdot C_2 + \ldots + x_{(K-1)(B-1)} \cdot C_{K-1} \right] \cdot 2^{B-1} \end{aligned} \tag{3.4}$$

Выражение (3.4) для КИХ-фильтра на четыре отвода ($K = 4$), в котором входная переменная $x_k(n)$ 4-разрядная ($B = 4$), запишем в виде:

$$y = P_0 + P_1 \cdot 2^1 + P_2 \cdot 2^2 - P_3 \cdot 2^3 \,.$$

$$
\begin{aligned}
P_0 &= X_{00}C_0 + X_{10}C_1 + X_{20}C_2 + X_{30}C_3 \\
P_1 &= X_{01}C_0 + X_{11}C_1 + X_{21}C_2 + X_{31}C_3 \\
P_2 &= X_{02}C_0 + X_{12}C_1 + X_{22}C_2 + X_{32}C_3 \\
P_3 &= X_{03}C_0 + X_{13}C_1 + X_{23}C_2 + X_{33}C_3
\end{aligned}, \qquad (3.5)
$$

где $P_0, P_1, P_2, P_3 -$ частичные произведения.

Вычисление результата $y(n)$ начинается путем адресации всеми битами младшего значащего разряда всех k входных переменных LUT-таблицы, содержащей комбинацию сумм коэффициентов фильтра. Выходное значение просмотровой таблицы сохраняется в масштабирующем аккумуляторе. После этого LUT-таблица адресуется следующими битами от младшего значащего всех входных переменных, результат умножается на 2^1 (путём сдвига слова влево) и добавляется в аккумулятор. Данная операция выполняется над всеми значащими битами, кроме знакового, выходное значение LUT-таблицы, адресуемой старшими битами входных переменных, вычитается из аккумулятора (рис. 3.6). Одна четырех входовая LUT-таблица обеспечивает 16 частичных произведений, которые являются комбинациями сумм коэффициентов КИХ-фильтров.

Еще раз обратим внимание на то, что дополнение до двух можно получить, если прибавить 1 к результату обращения. Обращение логически эквивалентно инверсии каждого бита в числе. Вентили Исключающее ИЛИ можно применить для избирательной инверсии в зависимости от значения управляющего сигнала. Прибавление 1 к результату обращения можно реализовать, задавая 1 на входе переноса c_0 (рис. 3.6).

Пример

Уменьшаемое	**A**	**+ 14**	**01110**		**+7**	**00111**
Вычитаемое	**B**	**-(+7)**	**- 00111**		**-(+14)**	**- 01110**
перевод B в дополн. код			**01110** **+ 11000** **+ 1**			**00111** **+ 10001** **+ 1**
Разность		**+7**	**1 00111**		**-7**	**11001**

Перенос игнорируется

Пример 1. Вычитание с использованием дополнительного кода (дополнение до двух). Осуществляются инвертирование вычитаемого и суммирование и перенос 1 в младший значащий разряд с последующим сложением

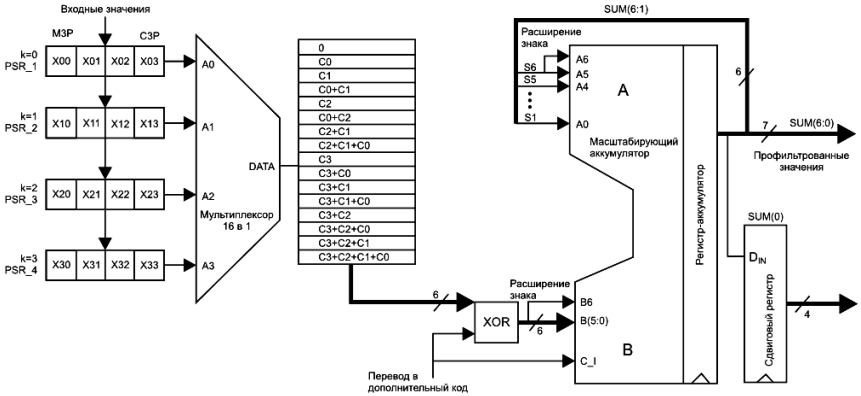


Рис. 3.6. Упрощенная схема КИХ-фильтра на распределенной арифметике

Рассмотрим процесс вычисления более подробно. Предположим что коэффициенты фильтра целочисленные со знаком известны и равны $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$. В противном случае можно было воспользоваться инструментом FDATool (Filter Design & Analysis Tool) системы Matlab/Simulink.

В момент времени $n = 0$ на вход КИХ-фильтра подается входная переменная $x_0(n)$ (отсчет, например, число десятичное число равное -5, представленное в дополнительном четырехзначном двоичном коде как 1011), которое сохраняется в регистре PSR_1 (перед вычислением регистры PSR1-PSR4 обнуляются).

Первый цикл обработки состоит в адресации всеми битами младшего значащего разряда всех $K = 4$ входных переменных 0001 LUT-таблицы (рис. 3.7). Из LUT-таблицы извлекается коэффициент C_0, представленный в дополнительном коде 111110 с расширением знака на два разряда и поступает в масштабирующий аккумулятор, где происходит его сложение с нулем. Операция расширения знака для чисел, представленных в дополнительном коде показана на рис. 3.6.

Полученный результат без учета МЗР сохраняется в регистре Reg 1, а в сдвиговый регистр Shif Reg 2 сохраняется МЗР. Расширение знака для чисел, поступающих на вход масштабируемого аккумулятора перед сложением и последующее отбрасывание МЗР у полученного результата обеспечивают эквивалент операции масштабирования.

Второй цикл обработки (рис. 3.8) начинается с адресации всеми битами более старшего младшего значащего разряда всех k входных переменных LUT-таблицы. Из LUT-таблицы опять извлекается коэффициент C_0, представленный в дополнительном коде 111110 с расширением знака на два разряда, который поступает в масштабирующий аккумулятор, где происходит его сложение с ранее полученным результатом, сохраненным в регистре Reg 1 с расширением знака до 6 разрядов. Полученный МЗР сохраняется в регистре Shif Reg 2, а СЗР игнорируется.

Третий цикл обработки позволяет накопить в регистре Shif Reg 2 число 010 (рис. 3.9). Четвертый цикл обработки

заканчивается вычитанием всех битов знакового разряда всех k входных переменных LUT-таблицы (рис. 3.10).

Извлеченное из LUT-таблицы число переводится в дополнительный код с помощью операции взятия обратного кода (инверсия всех битов) и прибавления 1 к МЗР входа B масштабирующего аккумулятора. В результате таких манипуляций в регистре Shif Reg 2 накапливается число 1010 (десятичное число 10), что соответствует формуле 1: $y(n) = C_0 x_0$. А в регистре Reg 3 будет накоплено двоичное десятиразрядное число 0000001010.

Предположим, что на вход КИХ-фильтра подается, например, десятичное число равное 3, представленное в дополнительном четырехзначном двоичном коде как 0011, то $y = C_0 x_0 + C_1 x_1 = -2 * 3 + (-1) * (-5) = -6 + 5 = -1$.

Старое значение регистра PSR_1 (-5) сохраняется в регистр PSR_2 и замещается новым числом 3. Получим новые значения адресации LUT-таблицы 0011, 0011, 0000 и 0010. Осуществив четыре цикла обработки, получим в регистре Reg 3 двоичное десятиразрядное число 1111111111 (-1 в дополнительном коде в десятиразрядном представлении).

Электрическая схема КИХ-фильтра на четыре отвода с использованием последовательной распределенной арифметики в САПР ПЛИС Quartus II компании Altera показана на рис. 3.11.

Для хранения комбинации сумм коэффициентов КИХ-фильтра (LUT-таблица) используется мультиплексор 16 в 1. На информационных входах мультиплексора в шестиразрядном представлении с использованием дополнительного кода хранится булева функция $f = x_0 C_0 + x_1 C_1 + x_2 C_2 + x_3 C_3$.

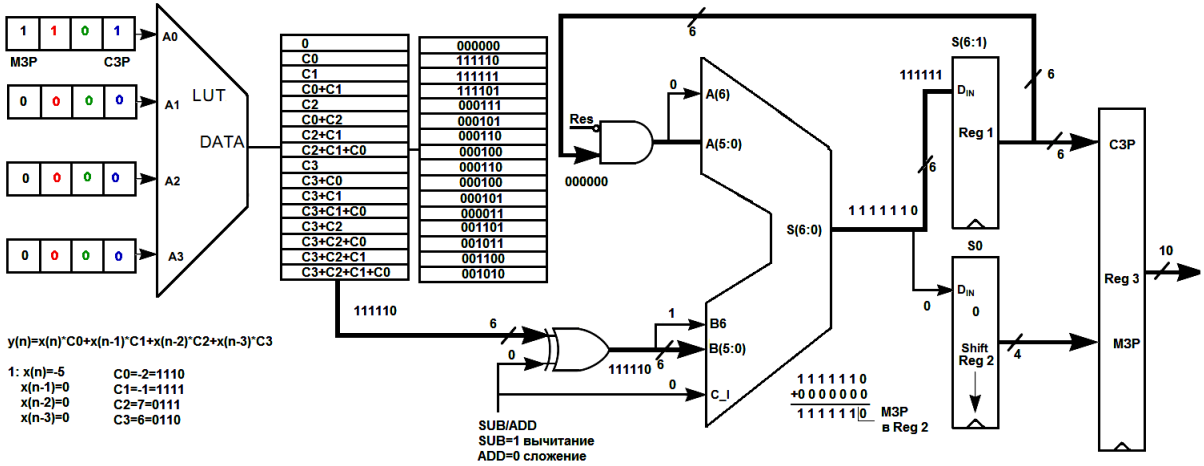


Рис. 3.7. На вход КИХ-фильтра подается число десятичное число равное -5, представленное в дополнительном четырехзначном двоичном коде как 1011, первый цикл обработки (адресация 0001). Суммирование частичного произведения P_0 с весом 2^0 с 0

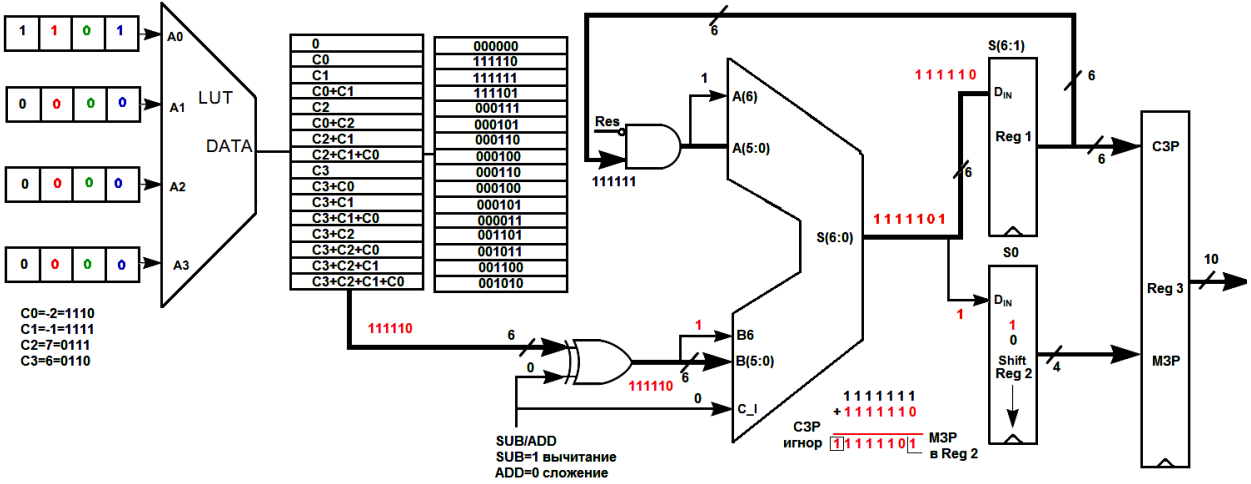


Рис. 3.8. Второй цикл обработки (адресация 0001). Суммирование частичного произведения P_1 с весом 2^1 с частичным произведением P_0 с весом 2^0

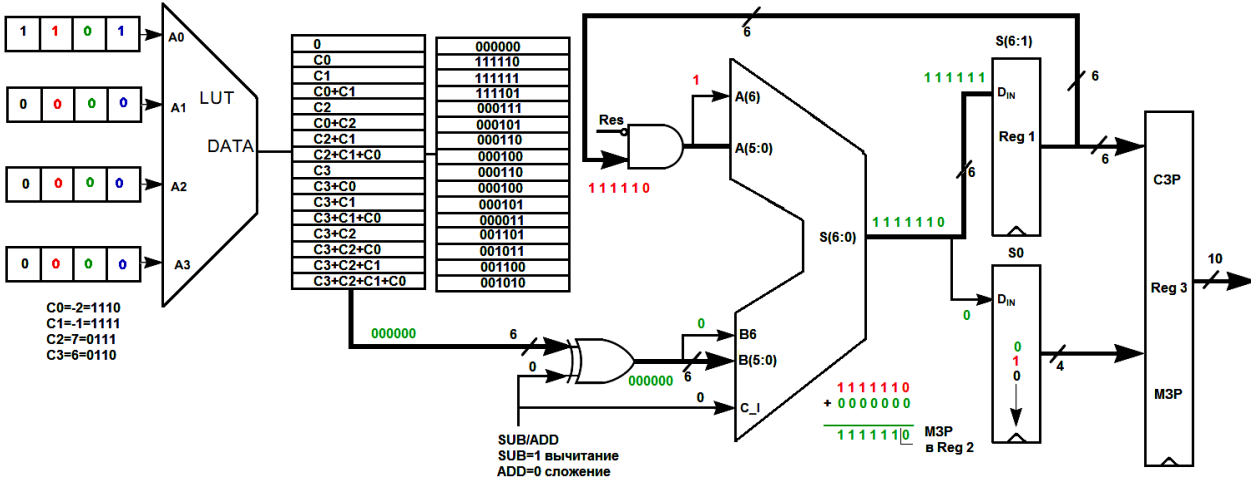


Рис. 3.9. Третий цикл обработки (адресация 0000). Суммирование частичного произведения P_2 с весом 2^2 с суммой частичных произведений P_0 с весом 2^0 и P_1 с весом 2^1

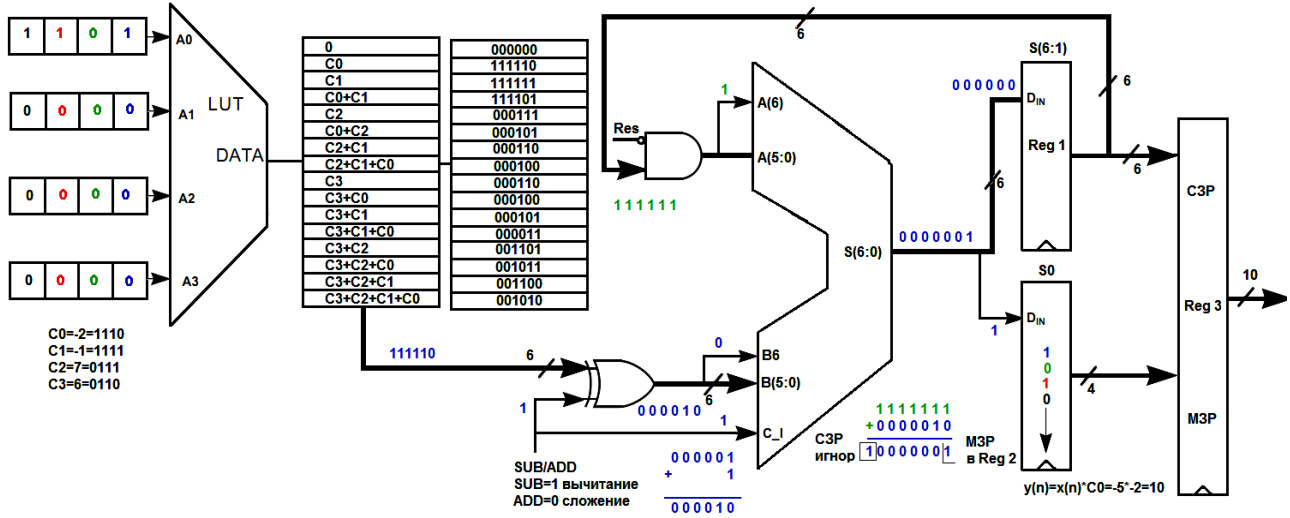


Рис. 3.10. Четвертый цикл обработки (адресация 0001). Суммирование (вычитание) частичного произведения P_3 с весом -2^3, представленного в дополнительном коде с суммой частичных произведений P_2 с весом 2^2, P_1 с весом 2^1 и P_0 с весом 2^0

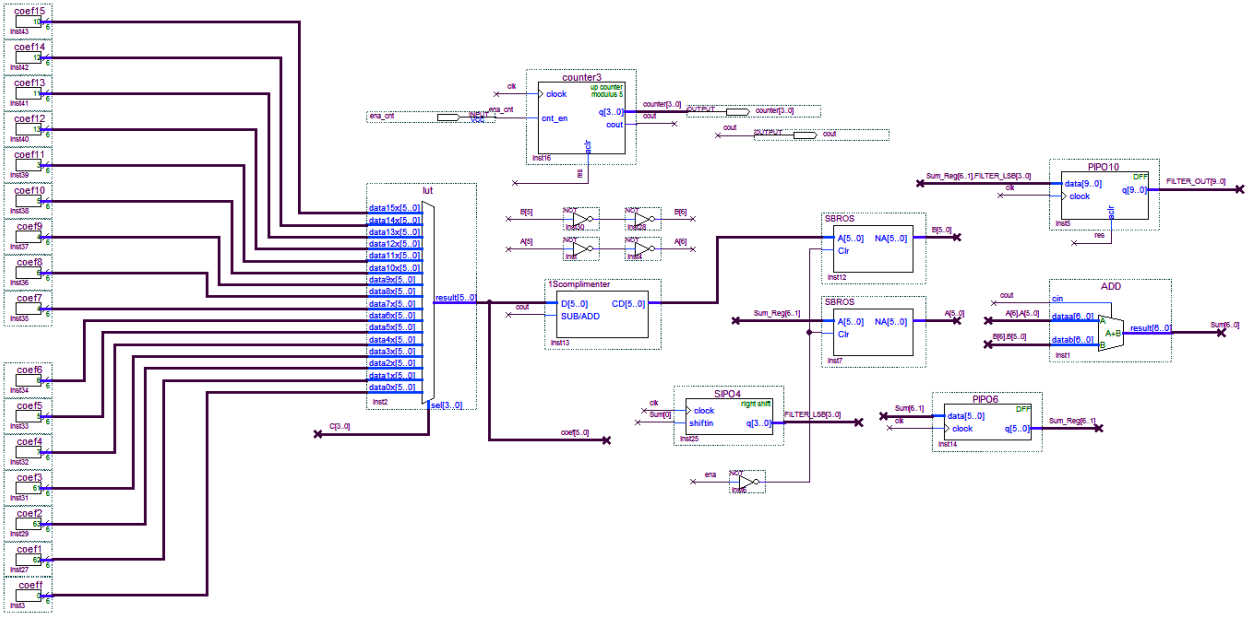


Рис. 3.11. Фрагмент схемы КИХ-фильтра на четыре отвода. Показаны мультиплексор 16 в 1 для хранения комбинации сумм коэффициентов, блок вычисления обратного кода, два блока очистки данных на входах сумматора, счетчик с модулем счета 5 и вспомогательные регистры

На адресные входы мультиплексора поступают биты младшего значащего разряда всех k входных переменных LUT-таблицы. Перед началом работы регистры линии задержки (рис. 3.12) и счетчик обнуляются. Входной отсчет (X0) первоначально сохраняется в 4-разрядном регистре PSR4 со сдвигом вправо с параллельным входом и последовательным выходом (для отладки системы, добавляется параллельный выход). При сдвиге вправо в старший значащий разряд регистра PSR4 добавляется 1. За четыре такта синхронизации входной отсчет X0 окажется в сдвиговом регистре SISO4 с последовательным входом и последовательным выходом, за следующие четыре такта в другом регистре SISO4 и так далее. Каждый регистр SISO4 осуществляет сдвиг вправо. Регистр PSR4 и три регистра SISO4 играют роль линии задержки КИХ-фильтра (многоразрядный сдвиговый регистр).

В качестве простейшего управляющего автомата используется суммирующий счетчик с модулем счета 5. Его задача отсчитать три значения (частичные произведения), поступающие с выхода мультиплексора, и вычесть четвертое из накопленной суммы. Так как операция вычитания заменяется взятием обратного кода и прибавлением 1 к МЗР, можно использовать обычный 7-разрядный сумматор со входом переноса Cin. В регистре SIPO4 сохраняется МЗР полученной суммы, а в регистре PIPO6 результат суммирования без учета этого МЗР. Расширение знака на входах сумматора осуществляется с помощью простого копирования СЗР полученной суммы. Регистр PIPO6 и сумматор ADD со схемами расширения знака представляют масштабируемый аккумулятор. Для корректной работы необходимо после обработки каждого входного отсчета сбрасывать входы сумматора в ноль. Это обеспечивают блоки SBROS, представляющие набор элементов 2И. Полученный результат (профильтрованные

входные отсчеты), представляемый в дополнительном коде, сохраняется в регистре PIPO10 с десятибитной точностью.

На рис. 3.13 показаны временные диаграммы работы КИХ-фильтра на распределенной арифметике. На вход КИХ-фильтра подаются входные отсчеты -5 (не показан), 3, 1, 0 в дополнительном коде (представляются как целые числа со знаком). Профильтрованные значения на выходе фильтра (подсвечены оранжевым цветом): 10, -1, -40, 25.

Интересно сравнить временные диаграммы работы КИХ-фильтра на четыре отвода, построенного с помощью мегафукнкции FIR Compiler САПР ПЛИС Quartus II.

Использование Mega Core Fir Compiler позволяет быстро спроектировать цифровой КИХ-фильтр исходя из заданных параметров. Быстрота и малая трудоемкость расчетов делают данное программное обеспечение незаменимым при проектировании КИХ-фильтров в базисе ПЛИС фирмы Altera.

На рис. 3.14 показаны настройки мегаядра FIR Compiler САПР ПЛИС Quartus II и импульсная характеристика проектируемого фильтра. Целочисленные коэффициенты фильтра загружаются из файла, не масштабируются, имеют представление в формате с фиксированной запятой. Выбираются структура КИХ-фильтра на последовательной распределенной арифметике и ПЛИС серии Stratix III. Галочкой отмечается, что фильтр имеет сильно несимметричную структуру коэффициентов. Для хранения коэффициентов фильтра и отсчетов используются логические ячейки адаптивных логических модулей. Задаются также входная и выходная спецификации фильтра (разрядность представления входных и выходных данных). На рис. 3.15 показана тестовая схема КИХ-фильтра с использованием мегаядра FIR Compiler, а на рис. 3.16 - временные диаграммы работы. Входные отсчеты -5, 3, 1, 0. Профильтрованные значения на выходе: 10, -1, -40, 25.

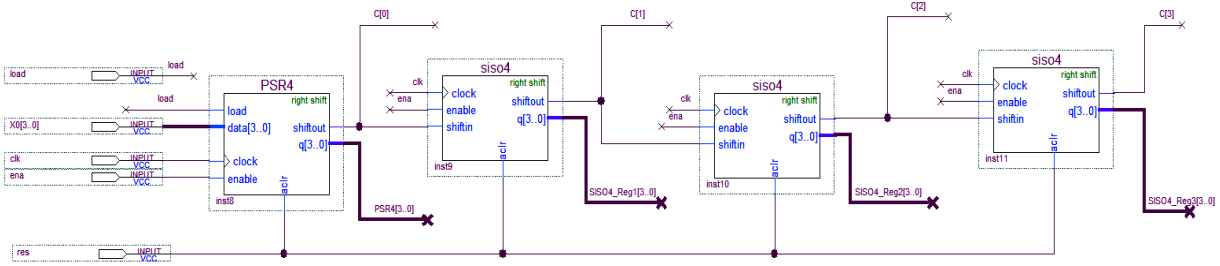


Рис.3.12. Фрагмент схемы КИХ-фильтра. Линия задержки

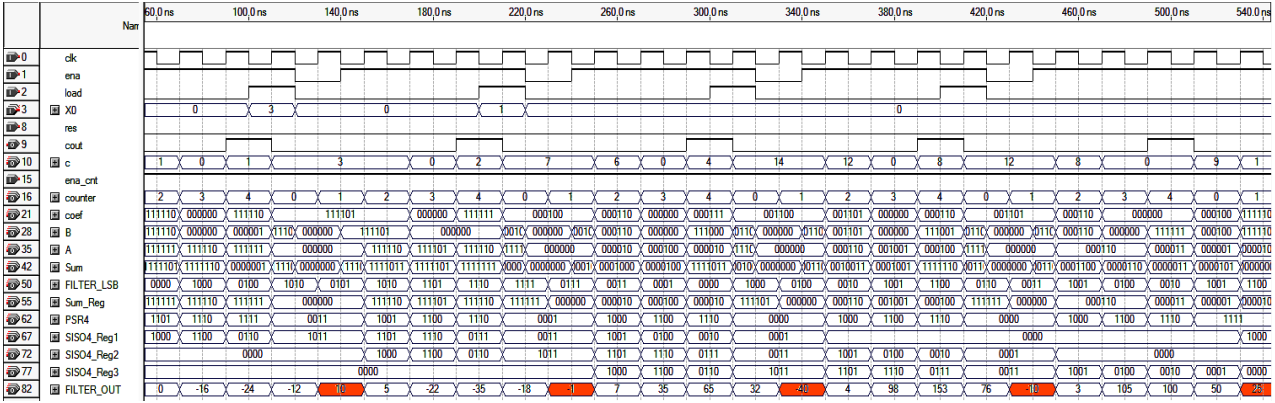


Рис. 3.13. Временные диаграммы работы КИХ-фильтра на распределенной арифметике

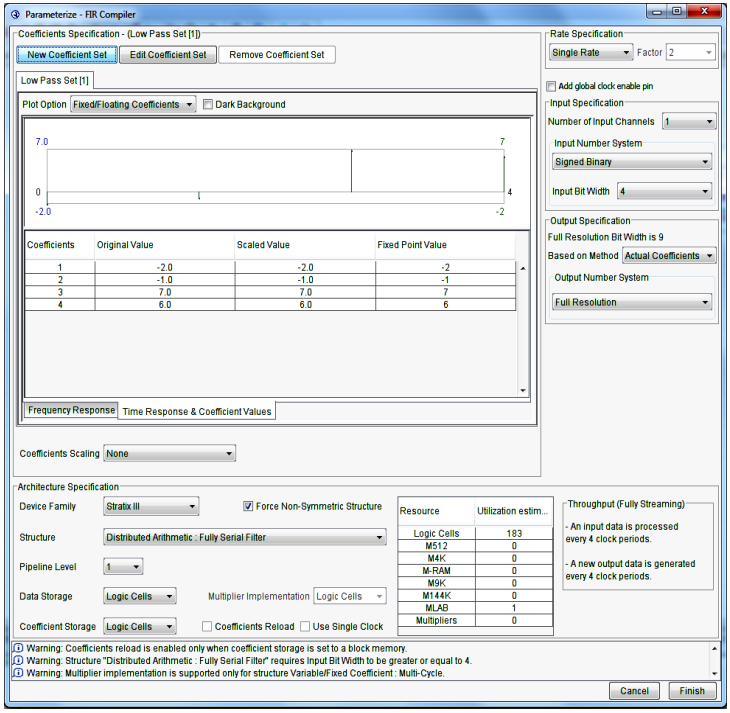


Рис. 3.14. Настройки мегаядра FIR Compiler САПР ПЛИС Quartus II

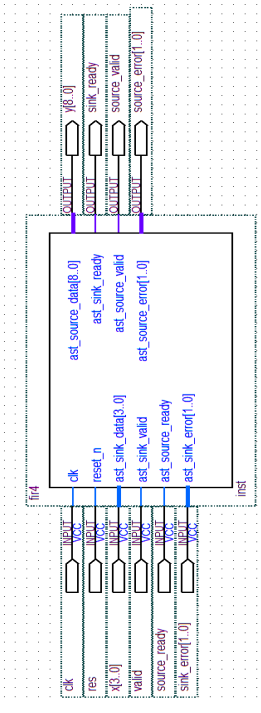


Рис. 3.15.Тестовая схема КИХ-фильтра с использованием мегаядра FIR Compiler

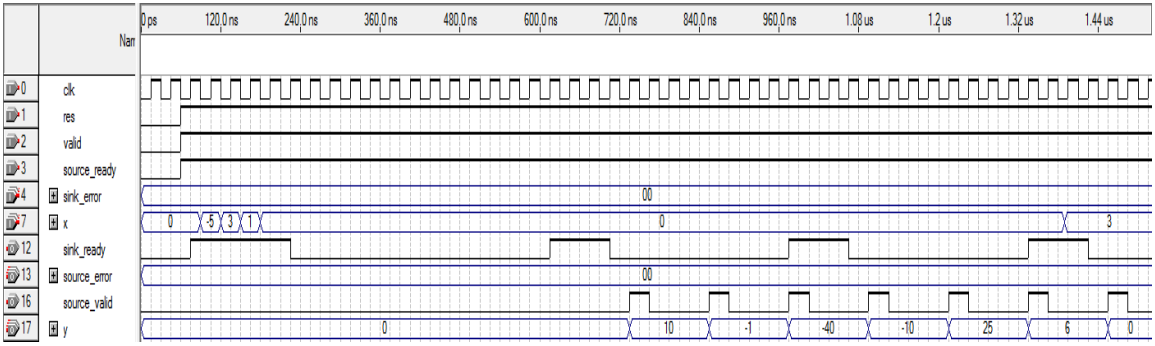


Рис. 3.16. Временные диаграммы работы КИХ-фильтра на мегаядре FIR Compiler

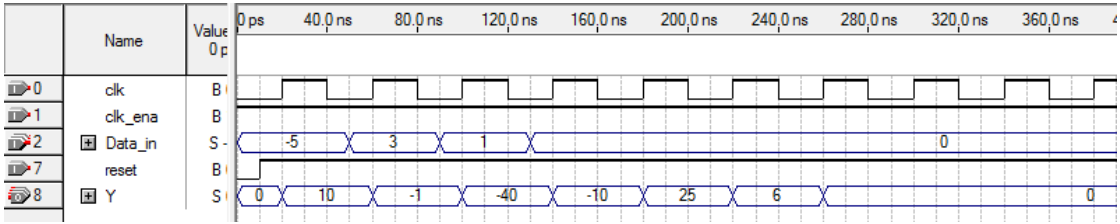


Рис. 3.17. Временные диаграммы работы КИХ-фильтра на четыре отвода по коду языка VHDL (пример 2)

Разработаем код высокоуровневого языка описания аппаратных средств VHDL КИХ-фильтра (пример 2 и пример 3) с использованием прямой реализации по формуле $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$ и посмотрим временные диаграммы (рис. 3.17). Сравнивая временные диаграммы (рис. 3.13 и рис. 3.16 с рис. 3.17), видим, что профильтрованные значения на выходе у трех фильтров совпадают, однако у фильтров на распределенной арифметике “нужные” выходные значения обновляются через каждые 4 такта. Существенным отличием является наличие у разных фильтров различных вспомогательных сигналов. Дополнительно мегафункция FIR Compiler имеет встроенный интерфейс, облегчающий взаимодействие с периферийными устройствами.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
package coeffs is
type coef_arr is array (0 to 3) of signed(3 downto 0);
constant coefs: coef_arr:= coef_arr'("1110", "1111", "0111", "0110");
  end coeffs;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.coeffs.all;
entity fir_var is
 port (clk, reset, clk_ena: in std_logic;
date: in signed (3 downto 0);
q_reg: out signed (9 downto 0));
end fir_var;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity filter_4 is
port (din : in std_logic_vector(3 downto 0);
reset, clk : in std_logic;
Sout : out std_logic_vector(7 downto 0));
end filter_4;
ARCHITECTURE a OF filter_4 IS
constant C0: std_logic_vector(3 downto 0) :="1110";
constant C1: std_logic_vector(3 downto 0) :="1111";
constant C2: std_logic_vector(3 downto 0) :="0111";
constant C3: std_logic_vector(3 downto 0) :="0110";
signal x0,x1,x2,x3:std_logic_vector(3 downto 0);
signal m0,m1,m2,m3:std_logic_vector(7 downto 0);
```

```
architecture beh of fir_var is
begin
process(clk,reset)
type shift_arr is array (3 downto 0)
     of signed (3 downto 0);
variable shift: shift_arr;
variable tmp: signed (3 downto 0);
variable pro: signed (7 downto 0);
variable acc: signed (9 downto 0);
begin
if reset='0' then
for i in 0 to 3 loop
shift(i):= (others => '0');
end loop;
q_reg<= (others => '0');
elsif(clk'event and clk = '1') then
if clk_ena='1' then
shift(0):=date;
pro := shift(0) * coefs(0);
acc := conv_signed(pro, 10);
for i in 2 downto 0 loop
pro := shift(i+1) * coefs(i+1);
acc := acc + conv_signed(pro, 10);
shift(i+1):= shift(i);
end loop;
end if; end if;
q_reg<=acc;
end process; end beh;
```

Пример 2. Код языка VHDL КИХ-фильтра на четыре отвода

```
BEGIN
m0<=(signed(x0)*signed(C0));
m1<=(signed(x1)*signed(C1));
m2<=(signed(x2)*signed(C2));
m3<=(signed(x3)*signed(C3));

Sout<=(signed(m0)+signed(m1)+
signed(m2)+signed(m3));
process(clk,reset)
begin
if reset='1' then
x0<=(others=>'0');
x1<=(others=>'0');
x2<=(others=>'0');
x3<=(others=>'0');
elsif (clk'event and clk='1') then
x0(3 downto 0) <=din(3 downto 0);
x1(3 downto 0) <=x0(3 downto 0);
x2(3 downto 0) <=x1(3 downto 0);
x3(3 downto 0) <=x2(3 downto 0);
end if;
end process;
END a;
```

Пример 3. Код языка VHDL КИХ-фильтра на четыре отвода (вариант)

В данном подразделе на примере проектирования простейшего КИХ-фильтра на четыре отвода показаны основы распределенной арифметики, широко используемой для разработки высокопроизводительных цифровых устройств цифровой обработки сигналов.

3.2. КИХ-фильтры на параллельной распределенной арифметике

Цель данного раздела показать, что основой КИХ-фильтра на параллельной распределенной арифметике является параллельный векторный умножитель, реализация которого в базисе ПЛИС позволяет получить максимальный выигрыш по быстродействию.

Уравнение КИХ-фильтра (нерекурсивного цифрового фильтра с конечно-импульсной характеристикой) представляется как арифметическая сумма произведений:

$$y = \sum_{k=1}^{K} C_k \cdot x_k , \tag{3.6}$$

где $y-$ отклик цепи; $x_k -$ $k-$я входная переменная; $C_k -$весовой коэффициент $k-$й входной переменной, который является постоянным для всех n; K - число отводов фильтра.

Если рассматривать входные переменные x_k как целые десятичные числа со знаком в дополнительном двоичном коде, то

$$x_k = -x_{k(\text{B}-1)} 2^{B-1} + \sum_{b=0}^{B-2} x_{kb} \cdot 2^b , \tag{3.7}$$

где $B-$разрядность кода; $x_{k(\text{B}-1)} 2^{B-1} -$знаковый разряд. Подставим выражение (3.7) в (3.6), получим:

$$\begin{aligned} y &= \sum_{k=1}^{K} C_k \cdot \left[- x_{k(\text{B}-1)} 2^{B-1} + \sum_{b=0}^{B-2} x_{kb} \cdot 2^b \right] = \\ &= -\sum_{k=1}^{K} x_{k(B-1)} 2^{B-1} C_k + \sum_{k=1}^{K} \sum_{b=0}^{B-2} x_{kb} 2^b C_k \end{aligned} \tag{3.8}$$

Раскроем все суммы в выражении (3.8) и сгруппируем числа по степеням B:

$$
\begin{aligned}
y = &\left[x_{10}\cdot C_1 + x_{20}\cdot C_2 + x_{30}\cdot C_3 + ... + x_{K0}\cdot C_K\right]\cdot 2^0 \\
&+\left[x_{11}\cdot C_1 + x_{21}\cdot C_2 + x_{31}\cdot C_3 + ... + x_{K1}\cdot C_K\right]\cdot 2^1 \\
&+\left[x_{12}\cdot C_1 + x_{22}\cdot C_2 + x_{32}\cdot C_3 + ... + x_{K2}\cdot C_K\right]\cdot 2^2 \\
&.... \\
&.... \\
&+\left[x_{1(B-2)}\cdot C_1 + x_{2(B-2)}\cdot C_2 + x_{3(B-2)}\cdot C_3 + ... + x_{K(B-2)}\cdot C_K\right]\cdot 2^{B-2} \\
&-\left[x_{1(B-1)}\cdot C_1 + x_{2(B-1)}\cdot C_2 + x_{3(B-1)}\cdot C_3 + ... + x_{K(B-1)}\cdot C_K\right]\cdot 2^{B-1}
\end{aligned}
\qquad (3.9)
$$

Каждое выражение в квадратной скобке представляет собой сумму операций И над одним разрядом входной переменной x_k, определяемую весовым фактором B всех k входных переменных и всеми битами весовых коэффициентов C_k.

Для структуры КИХ-фильтра с восемью отводами на распределенной арифметике с несимметричными коэффициентами выражение в квадратной скобке для индекса b может быть записано в виде

$$
\begin{aligned}
[sumb] = &x_{1b}\cdot C_1 + x_{2b}\cdot C_2 + x_{3b}\cdot C_3 + x_{4b}\cdot C_4 + \\
&+ x_{5b}C_5 + x_{6b}C_6 + x_{7b}C_7 + x_{8b}C_8
\end{aligned},
$$

и для фильтра с симметричными коэффициентами:

$$
\begin{aligned}
[sumb] = &(x_{1b} + x_{8b})\cdot C_1 + (x_{2b} + x_{7b})\cdot C_2 \\
&+ (x_{3b} + x_{6b})\cdot C_3 + (x_{4b} + x_{5b})\cdot C_4
\end{aligned}.
$$

Рассмотрим построение КИХ-фильтра на основе параллельной распределённой арифметики на примере структуры восемь отводов восемь бит. Перепишем уравнение (3.9) в следующем виде:

$$
\begin{aligned}
y = &[sum0] + [sum1]2^1 + [sum2]\cdot 2^2 + ... + \\
&+ [sum6]\cdot 2^6 - [sum7]\cdot 2^7
\end{aligned}
\qquad (3.10)
$$

при этом под обозначениями sum0, sum1 и т.д. подразумеваются выражения, заключённые соответственно в первых квадратных скобках, во вторых и т.д.

Преобразуем содержимое формулы (3.10) в массив подобных сумм на основе двухвходовых сумматоров:

$$y = \{[sum0]+[sum1]2^1\}+\{[sum2]+[sum3]2^1\}2^2+\{[sum4]+[sum5]2^1\}2^4 + \{[sum6]-[sum7]2^1\}2^6 \quad (3.11)$$

или

$$y = \{\{[sum0]+[sum1]2^1\}+\{[sum2]+[sum3]2^1\}2^2\}+ \{\{[sum4]+[sum5]2^1\}+\{[sum6]-[sum7]2^1\}2^2\}2^4 \quad (3.12)$$

Разница между принципами параллельной и последовательной распределённой арифметики состоит в числе последовательных масштабирующих суммирований значений квадратных скобок за период изменения входного отсчёта. В случае параллельной распределённой арифметики необходимо иметь B идентичных массивов памяти, параллельно адресуемых всеми битами всех входных переменных, и дерево масштабирующих сумматоров, осуществляющих соответствующее суммирование всех значений квадратных скобок. В данном случае результат формируется за один такт и тем самым достигается наибольшее быстродействие структуры.

Выше приведенные уравнения (3.11) и (3.12) полностью эквивалентны по своему значению, однако в последнем случае имеется возможность построения свёртывающего иерархического дерева многоразрядных сумматоров, что намного упрощает введение конвейера и достижение максимального быстродействия. В итоге всё дерево будет включать в себя восемь многоразрядных сумматоров. Данная структура на основе параллельной распределённой арифметики обеспечивает практически

предельное быстродействие при значительном объёме задействованных ресурсов.

Если рассматривать входные переменные x_k в формате с фиксированной запятой (x_k – дробные значения, $|x_k|<1$, x_{k0} – знаковый разряд), то

$$x_k = -x_{k0} + \sum_{b=1}^{B-1} x_{kb} \cdot 2^{-b},$$

$$y(n) = \sum_{k=1}^{K} C_k \cdot \left[-x_{k0} + \sum_{b=1}^{B-1} x_{kb} \cdot 2^{-b} \right] = \\ = -\sum_{k=1}^{K} x_{k0} \cdot C_k + \sum_{k=1}^{K} \sum_{b=1}^{B-1} x_{kb} \cdot C_k \cdot 2^{-b} \quad . \tag{3.13}$$

Аналогично выражению (3.9) уравнение КИХ-фильтра для формата с фиксированной запятой будет иметь вид

$$\begin{aligned} y(n) = &-\left[x_{10} \cdot C_1 + x_{20} \cdot C_2 + x_{30} \cdot C_3 + ... + x_{K0} \cdot C_K\right] \cdot 2^0 + \\ &+ \left[x_{11} \cdot C_1 + x_{21} \cdot C_2 + x_{31} \cdot C_3 + ... + x_{K1} \cdot C_K\right] \cdot 2^{-1} + \\ &+ \left[x_{12} \cdot C_1 + x_{22} \cdot C_2 + x_{32} \cdot C_3 + ... + x_{K2} \cdot C_K\right] \cdot 2^{-2} + \\ &.... \\ &.... \\ &+ \left[x_{1(B-2)} \cdot C_1 + x_{2(B-2)} \cdot C_2 + x_{3(B-2)} \cdot C_3 + ... + x_{K(B-2)} \cdot C_K\right] \cdot 2^{-(B-2)} + \\ &+ \left[x_{1(B-1)} \cdot C_1 + x_{2(B-1)} \cdot C_2 + x_{3(B-1)} \cdot C_3 + ... + x_{K(B-1)} \cdot C_K\right] \cdot 2^{-(B-1)}. \end{aligned} \tag{3.14}$$

Переформулируем выражение (3.14) и представим его в следующем виде и сравним с уравнением (3.13):

$$y(n) = -\sum_{k=1}^{K} C_k x_{k0} + \sum_{b=1}^{B-1} \left[\sum_{k=1}^{K} C_k x_{kb} \right] 2^{-b} = \\ = -P_0 + \left[\sum_{b=1}^{B-1} 2^{-b} P_b \right] \quad , \tag{3.15}$$

где $P-$ частичные произведения (значения в квадратных скобках выражения (3.14), представляющие комбинации сумм коэффициентов фильтра, которые предварительно вычисляются), а масштабирование частичных произведений может быть параллельным или последовательным.

Видим, что изменение в формулах приводит к перестройке аппаратных ресурсов фильтра (рис. 3.18). По формуле (3.13) получается фильтр с использованием операций умножения с накоплением, а по формуле (3.15) - фильтр на распределенной арифметике без операций явного умножения. А выражения для КИХ-фильтра, представленные целочисленными и дробными значениями, отличаются вычислениями знакового разряда и весовых коэффициентов. Например, для КИХ-фильтра на восемь отводов с целочисленными значениями старший знаковый разряд - это выражение $-[sum7]$ с весом 2^7, а для фильтра с дробными значениями это $-[sum0]$ с весом 2^0.

Дополнительный код, целочисленные значения

$$y(n) = \left[\sum_{b=0}^{B-2} 2^b P_b \right] - 2^{B-1} P_{B-1} \qquad (3.16)$$

Дополнительный код, дробные значения

$$y(n) = -P_0 + \left[\sum_{b=1}^{B-1} 2^{-b} P_b \right] \qquad (3.17)$$

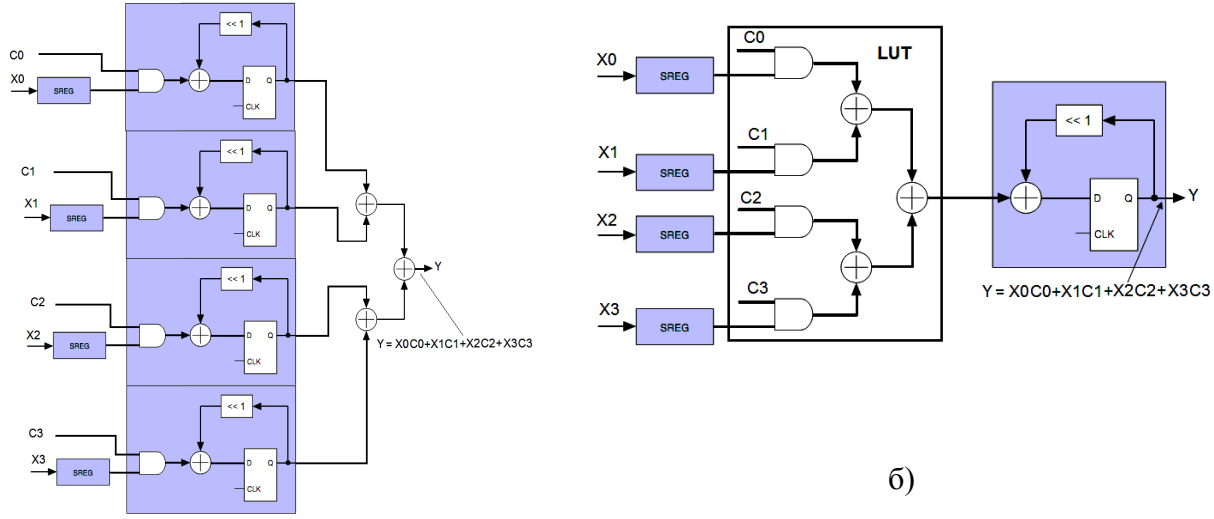


Рис. 3.18. КИХ-фильтр на четыре отвода: а) аппаратная реализация фильтра по формуле (3.13); б) по формуле (3.15) с использованием LUT

Рассмотрим КИХ-фильтр (рис. 3.19) с симметричными коэффициентами (выбираются симметрично относительно центральной величины). В основе стуктуры КИХ-фильтра лежит параллельный векторный перемножитель, в качестве которого из-за постоянства коэффициентов используют таблицу перекодировок (LUT). Рассмотрим простейший параллельный векторный перемножитель четырех 2-разрядных сигналов на четыре 2-разрядные константы (4-разрядный векторный перемножитель) в предположении, что все величины целочисленные и положительные, представлены в прямом коде (рис. 3.20). Умножение и сложение происходят параллельно с использованием LUT (табл. 3.1).

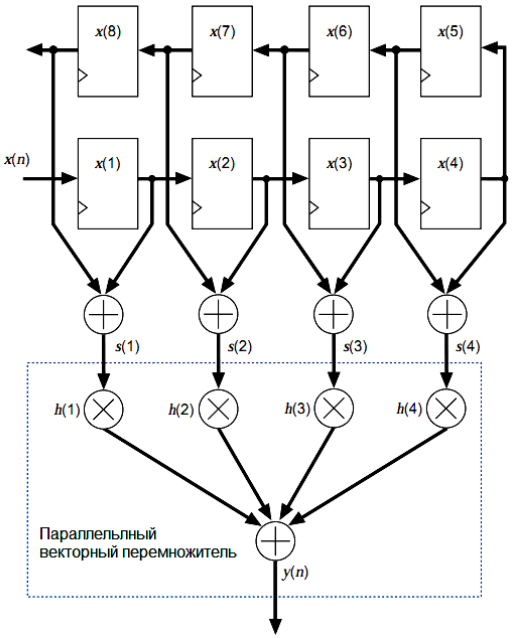


Рис. 3.19. Структура КИХ-фильтра на восемь отводов с симметричными коэффициентами, в основе которой лежит параллельный векторный перемножитель

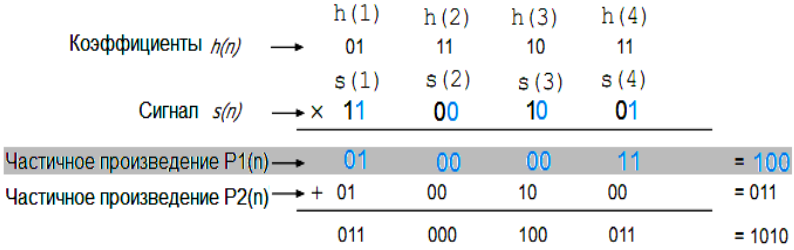


Рис. 3.20. Принцип параллельного векторного перемножения

Принцип формирования частичного произведения P1(n) показан на рис. 3.20. Булева функция $y = [s(1)h(1)] + [s(2)h(2)] + [s(3)h(3)] + [s(4)h(4)]$ для формирования P1(n) реализуется таблицей истинности, которая хранится в LUT. Идентичная таблица используется и для формирования P2(n).

Для завершения формирования частичного произведения P2(n) результат необходимо сдвинуть на один разряд влево, что равносильно умножению на 2. Это легко реализовать с помощью сдвиговых регистров. Далее частичные произведения P1(n) и P2(n) необходимо сложить с учетом возможного переполнения. На рис. 3.21 показан параллельный векторный перемножитель четырех 2-разрядных сигналов. Таким образом, требуются две идентичные таблицы, двоичный сдвиг влево и операция суммирования.

На рис. 3.22 показана структура КИХ-фильтра восемь отводов восемь бит на распределенной арифметике с несимметричными и с симметричными коэффициентами, обеспечивающими точность вычислений от 8 до 19 бит (полная точность), в основе которой лежит параллельный векторный перемножитель.

Таблица 3.1

Формирование частичного произведения P1(n)

s(n)	P1	y=[s(1)*h(1)]+[s(2)h(2)] +[s(3)h(3)]+[s(4)h(4)]
0000	0	00 + 00 + 00 + 00 = 0000
0001	h(1)	00 + 00 + 00 + 01 = 0001
0010	h(2)	00 + 00 + 11 + 00 = 0011
0011	h(2) + h(1)	00 + 00 + 11 + 01 = 0100
0100	h(3)	00 + 10 + 00 + 00 = 0010
0101	h(3) + h(1)	00 + 10 + 00 + 01 = 0011
0110	h(3) + h(2)	00 + 10 + 11 + 00 = 0101
0111	h(3) + h(2) + h(1)	00 + 10 + 11 + 01 = 0110
1000	h(4)	11 + 00 + 00 + 00 = 0011
1001	h(4) + h(1)	11 + 00 + 00 + 01 = 0100
1010	h(4) + h(2)	11 + 00 + 11 + 00 = 0110
1011	h(4) + h(2) + h(1)	11 + 00 + 11 + 01 = 0111
1100	h(4) + h(3)	11 + 10 + 00 + 00 = 0101
1101	h(4) + h(3) + h(1)	11 + 10 + 00 + 01 = 0110
1110	h(4) + h(3) + h(2)	11 + 10 + 11 + 00 = 1000
1111	h(4) + h(3) + h(2) + h(1)	11 + 10 + 11 + 01 = 1001

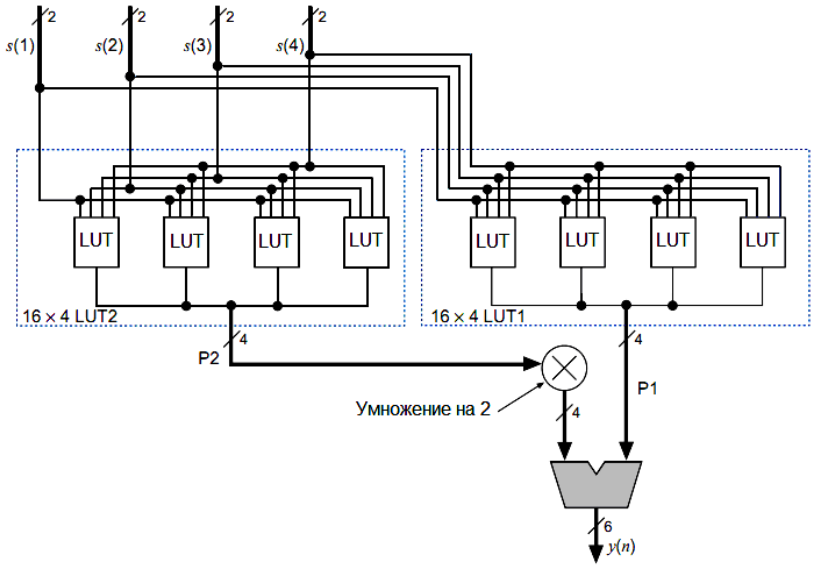
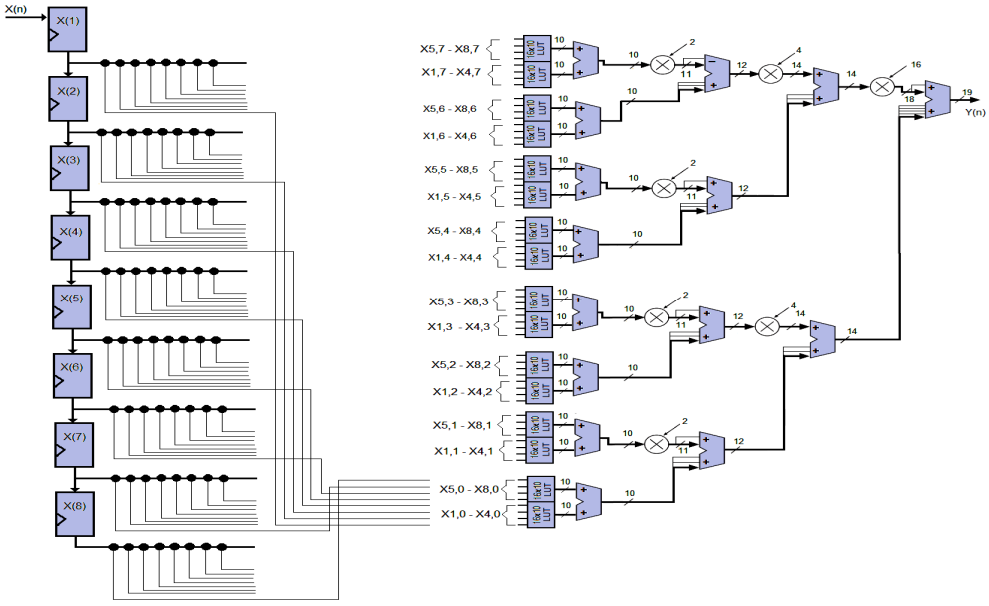
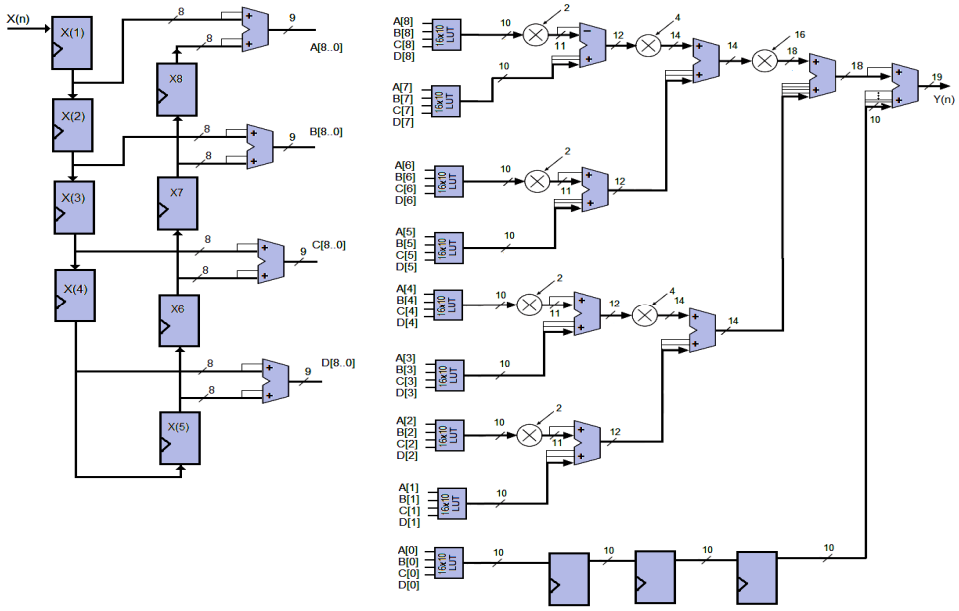


Рис. 3.21. Параллельный векторный перемножитель четырех 2-разрядных сигналов на четыре 2-разрядные константы



а)

Рис. 3.22. Структура КИХ-фильтра восемь отводов восемь бит на распределенной параллельной арифметике: а) с несимметричными коэффициентами; б) с симметричными



б)

Рис. 3.22. Структура КИХ-фильтра восемь отводов восемь бит на распределенной параллельной арифметике: а) с несимметричными коэффициентами; б) с симметричными (продолжение)

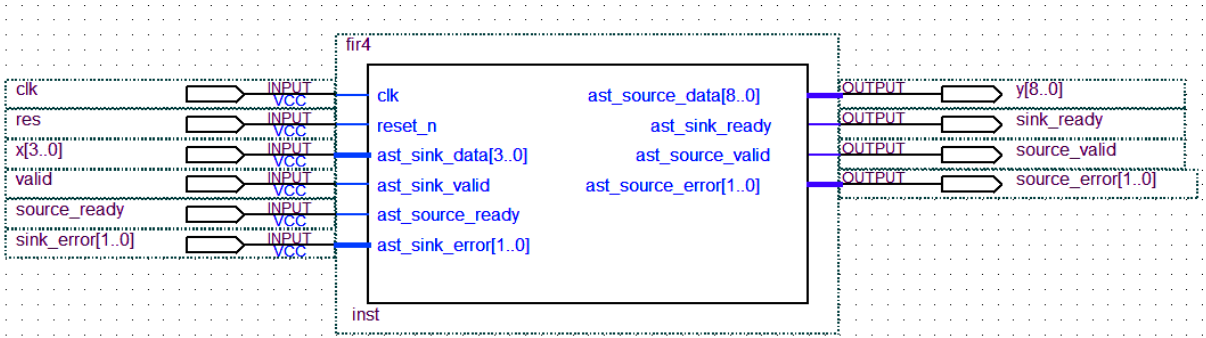


Рис. 3.23.Тестовая схема КИХ-фильтра с использованием мегаядра FIR Compiler на последовательной и параллельной распределенной арифметике

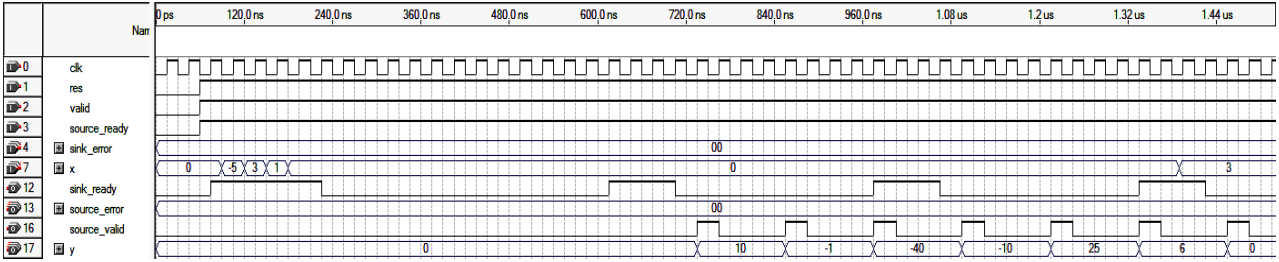


Рис. 3.24. Временные диаграммы работы КИХ-фильтра с использованием последовательной распределенной арифметики на мегаядре FIR Compiler

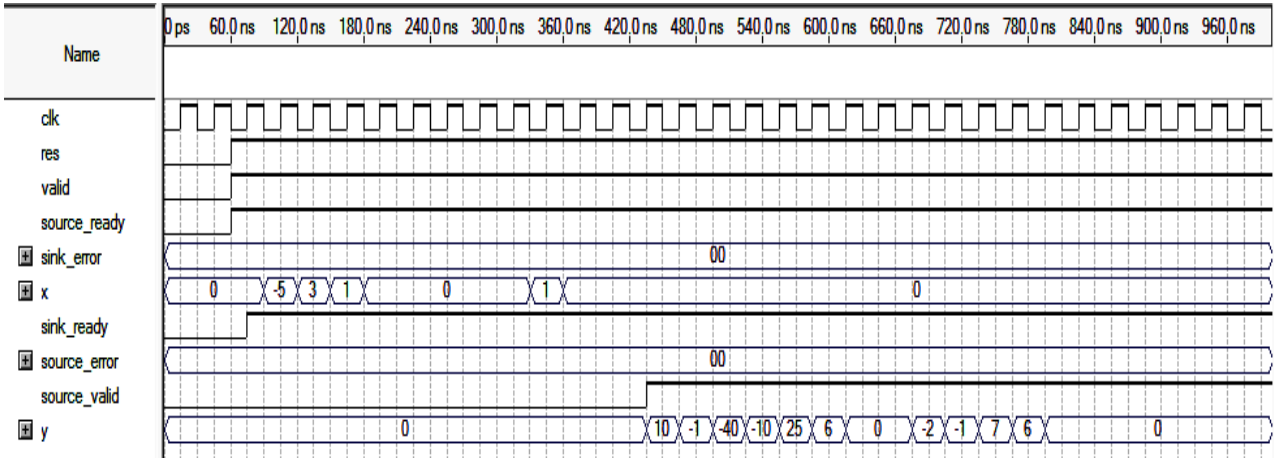


Рис. 3.25. Временные диаграммы работы КИХ-фильтра с использованием параллельной распределенной арифметики на мегаядре FIR Compiler

Входные данные на линии задержки представлены с 8-битной точностью параллельного кода. Для фильтра с симметричными коэффициентами требуются на выходах линии задержки 4 параллельных сумматора, которые своими выходами непосредственно адресуют 4-входовые LUT. Для того, чтобы переполнение гарантированно не произошло, необходимы 9-разрядные сумматоры, что обеспечивается расширением знакового разряда на входах. Это приводит к увеличению числа LUT с 8 до 9. В случае фильтра с несимметричными коэффициентами восемь отводов линии задержки уже адресуют 16 таких таблиц (число адресных линий равно числу элементов в векторе размерностью K, т.е. вместо использования 8 LUT с восьмью входами можно использовать 16 LUT с четырьмя входами).

Частичные произведения, представляющие комбинацию сумм 8-разрядных коэффициентов, хранящиеся в LUT представлены с 10-битной точность с запасом в два разряда. Поэтому в случае фильтра с несимметричными коэффициентами для суммирований значений с выходов LUT используются восемь 10 разрядных сумматоров. На входах последующих 12, 14 и 19-разрядных сумматоров требуется коррекция разрядности. Для фильтра с симметричными коэффициентами необходимы 12, 14, 18 и 19-разрядные сумматоры с соответствующей коррекцией на входах, а для получения 19-битной точности дополнительно требуется конвейер из трех регистров для суммирования значений выхода самой младшей LUT.

Для ускорения процесса разработки целесообразно воспользоваться мегаядрами. На рис. 3.23 приведена тестовая схема КИХ-фильтра с использованием мегаядра FIR Compiler САПР Quartus II компании Altera на последовательной и параллельной распределенной арифметике.

Предположим, что коэффициенты фильтра целочисленные со знаком известны и равны $C_0 = -2$, $C_1 = -1$,

$C_2 = 7$ и $C_3 = 6$. На вход КИХ-фильтра поступают входные отсчеты -5, 3, 1 и 0. Правильные значения на выходе фильтра: 10, -1, -40, -10, 26, 6 и т.д., т.е. согласно формуле $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$.

На рис. 3.24 и рис. 3.25 показаны временные диаграммы работы КИХ-фильтра с использованием последовательной и параллельной распределенной арифметики. Анализ задействованных ресурсов ПЛИС серии Stratix III при реализации КИХ-фильтров на четыре отвода с использованием мегаядра FIR Compiler показан в табл. 3.2.

Анализ табл. 3.2 показывает, что при числе отводов равным четырем существенной разницы между последовательной и параллельной арифметиками нет, т.к. для фильтра на последовательной арифметики требуется еще и управляющий автомат, который по числу задействованных триггеров может перекрыть число используемых АЛМ для выполнения комбинационных функций.

В структуре КИХ-фильтра на параллельной распределенной арифметике используется параллельный векторный перемножитель.

Несимметричность коэффициентов КИХ-фильтра на параллельной распределенной арифметики ведет к увеличению числа LUT.

Основным достоинством КИХ-фильтров на параллельной распределенной арифметике является повышенное быстродействие при возрастании числа задействованных ресурсов. Значительно снизить число используемых LUT позволяет последовательная распределенная арифметика.

Таблица 3.2

Анализ задействованных ресурсов ПЛИС серии Stratix III при реализации КИХ-фильтров на четыре отвода с использованием мегаядра FIR Compiler

Ресурсы ПЛИС серии Stratix III	Последовательная распределенная арифметика	Параллельная распределенная арифметика
1	2	3
Кол-во АЛМ для выполнения комбинационных функций	74	87
Кол-во АЛМ с памятью	4	4
АЛМ	79	79
Кол-во выделенных регистров	136	134
Аппаратные перемножители		
Кол-во АЛМ для выполнения комбинационных функций без использования регистров	13	17
Кол-во АЛМ под регистерные ресурсы	71	60
Кол-во АЛМ под комбинационные и регистерные ресурсы	65	74
Рабочая частота в наихудшем случае, МГц	400	400

3.3. Пример реализации КИХ-фильтра на параллельной распределенной арифметике

Уравнение КИХ-фильтра со структурой четыре отвода четыре бита (прямая реализация) представляется как арифметическая сумма произведений: $P_{out} = C_1 d_1 + C_2 d_2 + C_3 d_3 + C_4 d_4$. В случае параллельной распределенной арифметики уравнение КИХ-фильтра на четыре отвода записывается виде

$$P_{out} = 2^0 * P_0 + 2^1 * P_1 + 2^2 * P_2 - 2^3 * P_3 \qquad (3.18)$$

Частичные произведения P_0, P_1, P_2 и P_3:

$$P_0 = C_1 d_1(0) + C_2 d_2(0) + C_3 d_3(0) + C_4 d_4(0) = \sum_{n=1}^{4} C_n d_n(0);$$

(3.19)

$$P_1 = C_1 d_1(1) + C_2 d_2(1) + C_3 d_3(1) + C_4 d_4(1) = \sum_{n=1}^{4} C_n d_n(1); \qquad (3.20)$$

$$P_2 = C_1 d_1(2) + C_2 d_2(2) + C_3 d_3(2) + C_4 d_4(2) = \sum_{n=1}^{4} C_n d_n(2); \quad (3.21)$$

$$P_3 = C_1 d_1(3) + C_2 d_2(3) + C_3 d_3(3) + C_4 d_4(3) = \sum_{n=1}^{4} C_n d_n(3). \quad (3.22)$$

В случае параллельной распределённой арифметики для КИХ-фильтра на четыре отвода необходимо иметь четыре идентичных массивов памяти, параллельно адресуемых всеми битами всех входных переменных и свертывающееся иерархического дерево многоразрядных сумматоров, осуществляющих соответствующее суммирование частичных произведений P_0, P_1, P_2 и P_3. В данном случае результат формируется за один такт и тем самым достигается наибольшее быстродействие структуры.

На рис. 3.26 показана линия задержки КИХ-фильтра, а на рис. 3.27 - принцип подключения выходов линии задержки

КИХ-фильтра на четыре отвода к 4-входовым LUT. Разрядность входной шины данных $N = 4$. Входные данные на линии задержки представлены с 4-битной точностью параллельного кода. 4-входовая LUT обеспечивает 16 частичных произведений (на примере P_0, представляющих собой комбинации сумм коэффициентов фильтра представленные с 8-битной точностью).

Пример заполнения 4-входовой LUT показан в табл. 3.3, а описание на языке VHDL представлено ниже. На рис. 3.28 показана структура КИХ-фильтра на четыре отвода четыре бита на распределенной параллельной арифметике. Фильтр состоит из четырех однотипных LUT, формирующих частичные произведения P_0, P_1, P_2 и P_3 согласно формулам 3.19-3.22. Для суммирований значений с выходов LUT в соответствии с их весом (формула 3.18) используются два 12- и один 14-разрядные сумматоры с соответствующей коррекцией разрядности на входах для того, чтобы гарантировано предотвратить переполнение.

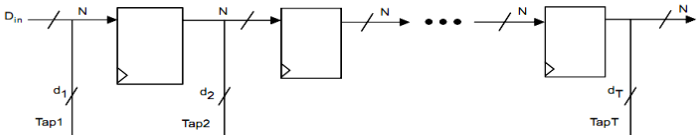


Рис. 3.26. Линия задержки КИХ-фильтра на регистрах

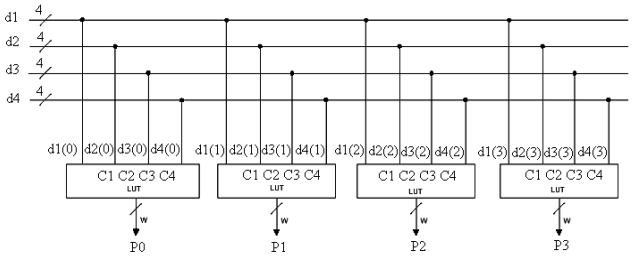


Рис. 3.27. Принцип подключения выходов линии задержки КИХ-фильтра на четыре отвода к 4-входовым LUT

Таблица 3.3

Вариант заполнения 4-входовой LUT на примере частичного произведения P_0

d4(0),d3(0),d2(0),d1(0)	Выход LUT-таблицы, P0
0000	0
0001	C1
0010	C2
0011	C2+C1
0100	C3
….	….
1111	C4+C3+C2+C1

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
Entity PartialProd is
Port (D : in std_logic_vector(3 downto 0);
      P : out std_logic_vector(9 downto 0));
-- 4 * 8 bit coeff => 10 bit product
End PartialProd;
Architecture Behave of PartialProd is
constant c1 : std_logic_vector := "11111110"; -- coefficient for tap 1
-- -2D
constant c2 : std_logic_vector := "11111111"; -- coefficient for tap 2
-- -1D
constant c3 : std_logic_vector := "00000111"; -- coefficient for tap 3
-- 7D
constant c4 : std_logic_vector := "00000110"; -- coefficient for tap 4
-- 6D
-- Compute all the partial products and store them as constants.
constant v0 : std_logic_vector := sxt("0", 10);
constant v1 : std_logic_vector := sxt(c1, 10);
constant v2 : std_logic_vector := sxt(c2, 10);
constant v3 : std_logic_vector := v1 + v2;
```

```
constant v4 : std_logic_vector := sxt(c3, 10);
constant v5 : std_logic_vector := v4 + v1;
constant v6 : std_logic_vector := v4 + v2;
constant v7 : std_logic_vector := v4 + v3;
constant v8 : std_logic_vector := sxt(c4, 10);
constant v9 : std_logic_vector := v8 + v1;
constant v10 : std_logic_vector := v8 + v2;
constant v11 : std_logic_vector := v8 + v3;
constant v12 : std_logic_vector := v8 + v4;
constant v13 : std_logic_vector := v8 + v5;
constant v14 : std_logic_vector := v8 + v6;
constant v15 : std_logic_vector := v8 + v7;
Begin
prodeval: process (D)
begin
case(d) is
when "0000" => P <= v0;
when "0001" => P <= v1;
when "0010" => P <= v2;
when "0011" => P <= v3;
when "0100" => P <= v4;
when "0101" => P <= v5;
when "0110" => P <= v6;
when "0111" => P <= v7;
when "1000" => P <= v8;
when "1001" => P <= v9;
when "1010" => P <= v10;
when "1011" => P <= v11;
when "1100" => P <= v12;
when "1101" => P <= v13;
when "1110" => P <= v14;
when "1111" => P <= v15;
end case;
end process;
end behave;
```

Пример описания заполнения LUT на языке VHDL для КИХ-фильтра на четыре отвода

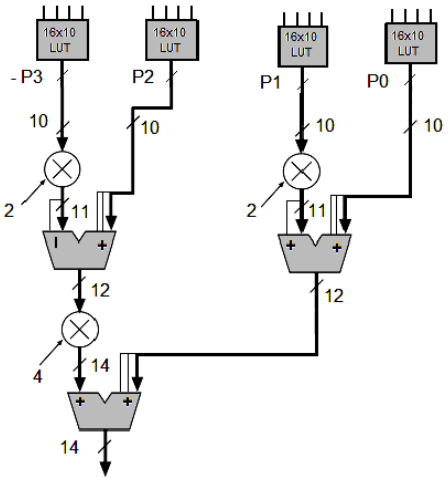


Рис. 3.28. Структура КИХ-фильтра на четыре отвода четыре бита на распределенной параллельной арифметике с указанием операции расширения знака

На рис. 3.29 показана функциональная модель КИХ-фильтра на четыре отвода четыре бита на распределенной параллельной арифметике в САПР ПЛИС Quartus II с использованием линии задержки на регистрах (рис. 3.29, *а*) и линии задержки на базе встроенных блоков ОЗУ (рис. 3.29, *б*). Мегафункция ALTSHIFT_TAPS, используемая в качестве линии задержки, представляет собой двухпортовое ОЗУ. Свертывающее иерархическое дерево многоразрядных сумматоров выполнено на мегафункциях LPM_ADD_SUB. Для знакового разряда (P_3) необходимо сформировать дополнение до двух путем обращения P_3 с последующим прибавлением 1 к младшему разряду.

Обращение логически эквивалентно инверсии каждого бита в числе. Вентили Исключающее ИЛИ можно применить для избирательной инверсии в зависимости от значения управляющего сигнала. Прибавление 1 можно организовать

подключением входа переноса C_{in} одного из 12-разрядного сумматора к шине питания.

Прохождение единичного импульса по структуре КИХ-фильтра в случае использования линии задержки на регистрах показано на рис. 3.30. На выходе фильтра видим коэффициенты фильтра $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$. На рис. 3.31 показаны временные диаграммы работы фильтра для случая, когда на вход КИХ-фильтра поступают входные отсчеты -5, 3, 1 и 0. Правильные значения на выходе фильтра: 10, -1, -40, -10, 26, 6 и т.д. На рис. 3.32 и 3.33 показаны временные диаграммы в случае использования линии задержки на базе встроенных блоков ОЗУ.

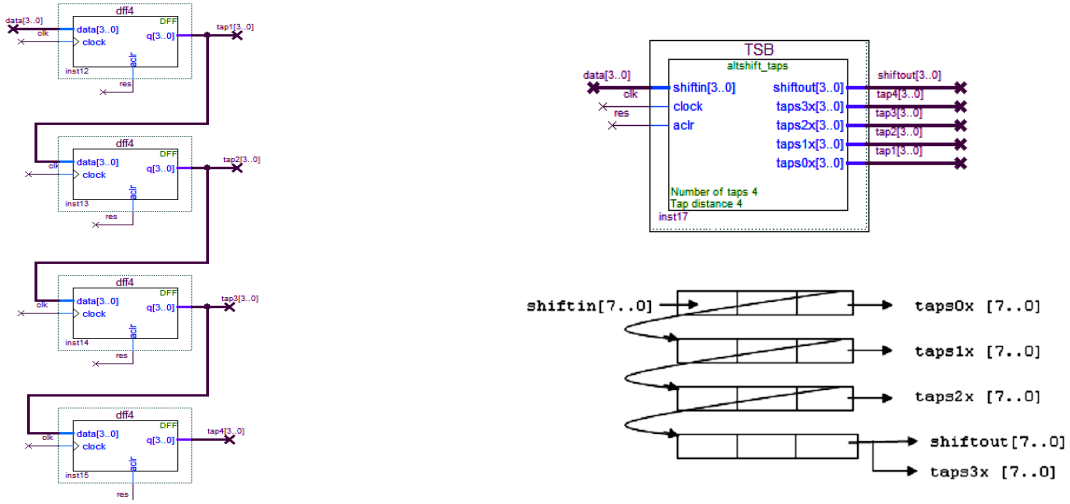
В тестировании принимают участие следующие структуры фильтров: классическая параллельная с использованием аппаратных ЦОС-боков (собран на мегафункции ALTMULT_ADD); систолическая структура; поведенческое описание на языке VHDL с использованием оператора цикла без привязки к какой-либо структуре, с использованием программных умножителей (мегафункция ALTMEMMULT) и фильтр, на параллельной распределенной арифметике, реализованный с помощью мегаядра MegaCore FIR Compiler.

Рассмотренные основные структурные схемы параллельных КИХ-фильтров позволяют сделать вывод, что использование КИХ-фильтров на параллельной распределенной арифметике позволяют для ПЛИС серии Cyclone III получить рекордное быстродействие за счет использования “безумножительных” схем умножения (табл. 3.4), не снижаемое при увеличении числа отводов фильтра и точности представления входных данных. Это особенно актуально для проектов, использующих низкопроизводительные (бюджетные) серии ПЛИС. А также в том случае, если пользователь откажется от применения в проекте мегафункций умножителей опционально

использующих либо аппаратные умножители, встроенные в ЦОС-блоки (ALTMULT_ADD, ALTMULT_ACCUM), либо программные (ALTMEMMULT).

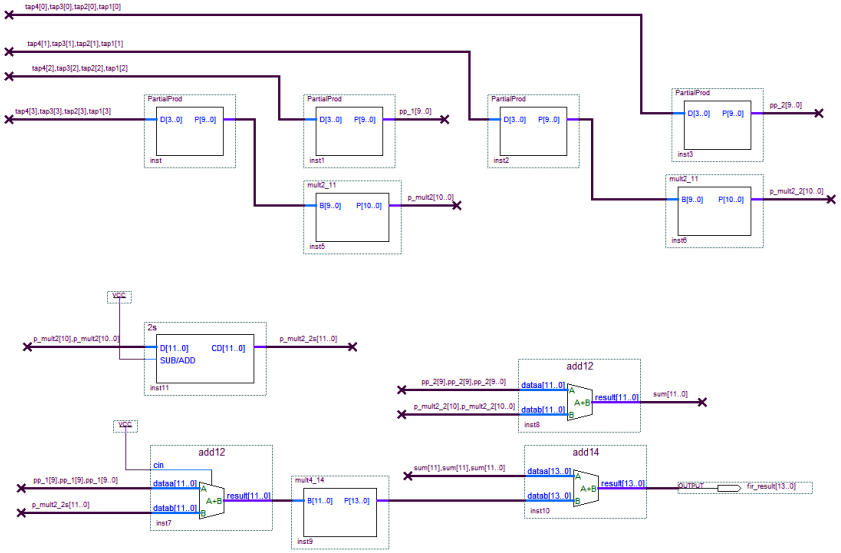
Использование кода языка VHDL и мегаядра FIR Compiler (структура фильтра задается пользователем опционально) в отличие от фильтров, структура которых разрабатывается “вручную”, приводит к пониженному быстродействию, однако при этом необходимо учитывать, что рассматриваемые примеры сильно упрощены. Для точной оценки производительности фильтров необходимо пользоваться сравнительными таблицами из официальных документов производителей ПЛИС.

Для КИХ-фильтров с большим числом отводов и точностью представления коэффициентов характерно возрастание числа задействованных ресурсов ПЛИС (табл. 3.5), поэтому необходимо по возможности использовать симметричные фильтры. Использование линии задержки на блочной памяти ПЛИС, также позволяет сократить число используемых логических элементов. Последовательно распределенная арифметика снижает объем задействованных ресурсов ПЛИС, но ухудшает быстродействие и производительность фильтров.



а) б)

Рис. 3.29. КИХ-фильтр на четыре отвода четыре бита на распределенной параллельной арифметике в САПР ПЛИС Quartus II: а) линия задержки на регистрах; б) линия задержки на базе встроенных блоков ОЗУ; в) частичные произведения на базе LUT и свертывающееся иерархического дерево многоразрядных сумматоров



в)

Рис. 3.29. КИХ-фильтр на четыре отвода четыре бита на распределенной параллельной арифметике в САПР ПЛИС Quartus II: а) линия задержки на регистрах; б) линия задержки на базе встроенных блоков ОЗУ; в) частичные произведения на базе LUT и свертывающееся иерархического дерево многоразрядных сумматоров (продолжение)

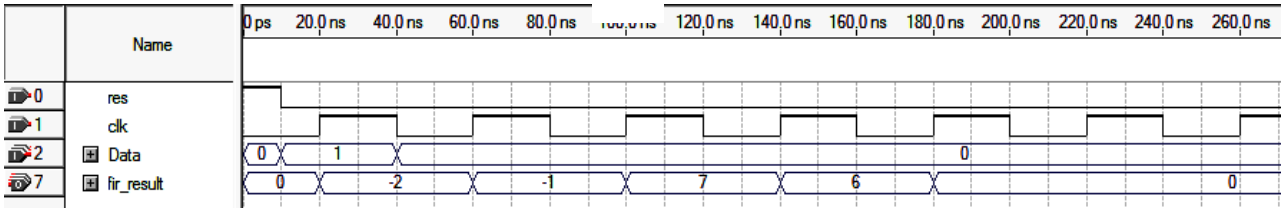


Рис. 3.30. Прохождение единичного импульса по структуре КИХ-фильтра в случае использования линии задержки на регистрах

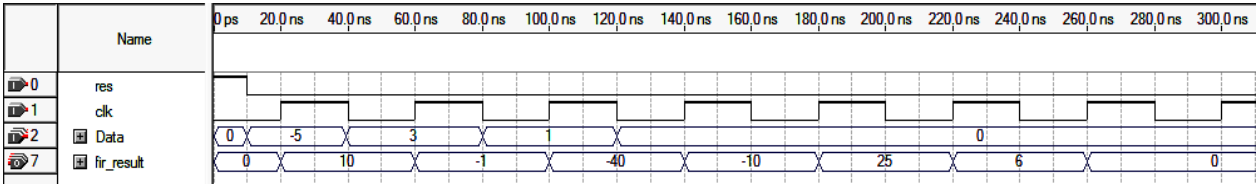


Рис. 3.31. Временные диаграммы работы фильтра в случае использования линии задержки на регистрах. На вход КИХ-фильтра поступают входные отсчеты -5, 3, 1 и 0. Правильные значения на выходе фильтра: 10, -1, -40, -10, 26, 6 и т.д.

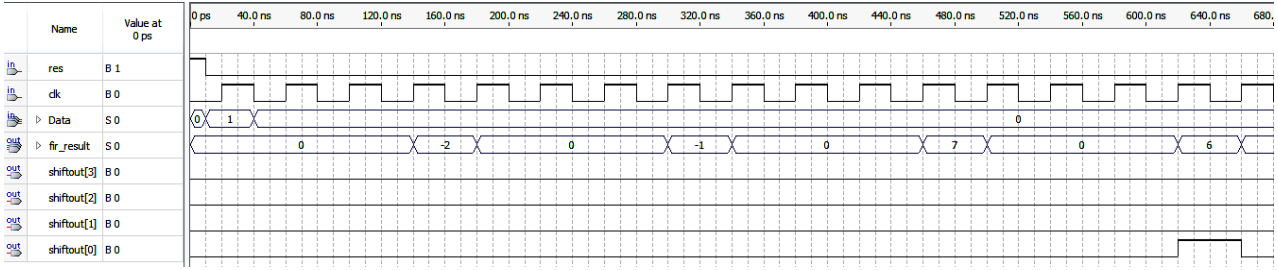


Рис. 3.32. Прохождение единичного импульса по структуре КИХ-фильтра в случае использования линии задержки на базе встроенных блоков ОЗУ

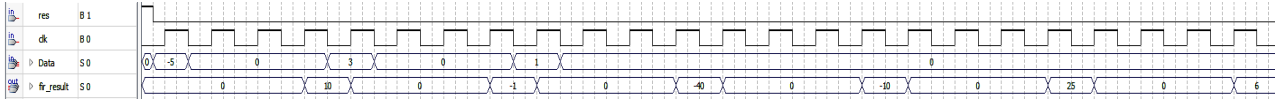


Рис. 3.33. Временные диаграммы работы фильтра в случае использования линии задержки на базе встроенных блоков ОЗУ

Таблица 3.4

Реализация параллельных КИХ-фильтров на 4 отвода в базисе ПЛИС Cyclone III EP3C5E144C7

Ресурсы ПЛИС	Параллельная распределенная арифметика, рис. 3.29. Линия задержки на		Параллельная распределенная арифметика, мегаядро FIR Compiler	Мегафункция ALTMULT_ADD. четыре умножителя и дерево сумматоров в блоке, внешняя линия задержки из 3-регистров	Систолический КИХ-фильтр с однотипным и процессорными элементами	КИХ-фильтр на языке VHDL с использованием оператора цикла loop	Мегафункция ALTMEMMULT, программный умножитель на константу, четыре блока по одному умножителю в каждом, коэффициенты загружаются из внешнего порта
	Регистрах	ОЗУ					
1	2	3	4	5	6	7	8
Логических элементов	56	44	382	41	165	54	166
LUT для реализации комбинационных функций	40	44	342	29	132	42	146

Продолжение табл. 3.4

1	2	3	4	5	6	7	8
Регистров для реализации последовательностной логики	16	4	253	22	65	22	116
Аппаратные ЦОС-блоки, встроенные в базис ПЛИС	0 (без использования мегафункций)	0	0	4 (умножитель 9x9)	0 (мегафункция LPM_MULT, умножители реализованы на LUT, 1 умножитель размерностью 4x4 занимает 24 LUT)	0	0 (режим auto, 1 умножитель размерностью 4x4 занимает 20 lut+256 bits(Auto)+29 reg)
Максимальная частота/частота в наихудшем случае, Fmax, МГц (временная модель Slow 1200mV 85C Model)	807/250	305/ 205	89/24	233/233	206/206	182/182	300/250

Таблица 3.5

Тестирование КИХ-фильтра со структурой 97 отводов восемь бит с точностью представления коэффициентов 14 бит, реализованного с помощью мегаядра FIR Compiler на ПЛИС серии Cyclone III

LUT для реализации комбинационных функций	Регистры ЛЭ для последовательностной логики	Память		Умножители 9x9	Fmax, МГц	Быстродействие, MSPS
		Регистры ЛЭ, биты	Блоки памяти M9K			
Параллельная распределенная арифметика с использованием ресурсов логических элементов, уровень конвейеризации 1, EP3C10F256C6						
3416	3715	208	3	-	288	288
Параллельная распределенная арифметика с использованием ресурсов встроенных блоков памяти M9K, уровень конвейеризации 1, EP3C40F780C6						
1948	2155	120030	48	-	283	283
Последовательная распределенная арифметика с использованием ресурсов встроенных блоков памяти M9K, уровень конвейеризации 1, EP3C10F256C6						
327	462	14167	8	-	323	36

4. СИСТОЛИЧЕСКИЕ КИХ-ФИЛЬТРЫ В БАЗИСЕ ПЛИС

4.1. Проектирование систолических КИХ-фильтров в базисе ПЛИС с использованием САПР Quartus II

Систолический КИХ-фильтр считается оптимальным решением для параллельных архитектур цифровых фильтров. В настоящее время входит в состав мегафункции (ALTMULT_ADD) САПР Quartus II начиная с версии 11, 12 и 13 для работы с ЦОС-блоками серий Cyclon V, Arria V и Stratix V, выполненных по 28 нм технологическому процессу и функции XtremeDSP™ Digital Signal Processing для ЦОС-блока DSP48 ПЛИС серии Virtex-4 Xilinx.

В фон-неймановских машинах данные, считанные из памяти, однократно обрабатываются в процессорном элементе (ПЭ), после чего снова возвращаются в память (рис. 4.1, *а*). Авторы идеи систолической матрицы Кунг и Лейзерсон предложили организовать вычисления так, чтобы данные на своем пути от считывания из памяти до возвращения обратно пропускались через как можно большее число ПЭ (рис. 4.1, *б*).

Если сравнить положение памяти в вычислительных системах со структурой живого организма, то по аналогии ей можно отвести роль сердца, множеству ПЭ – роль тканей, а поток данных рассматривать как циркулирующую кровь. Отсюда и происходит название систолическая матрица (систола - сокращение предсердий и желудочков сердца, при котором кровь нагнетается в артерии). Систолическая структура — это однородная вычислительная среда из процессорных элементов, совмещающая в себе свойства конвейерной и матричной обработки. Систолические структуры эффективны при выполнении матричных вычислений, обработке сигналов, сортировке данных и т.д.

Рассмотрим структуру систолического КИХ-фильтра на четыре отвода (рис. 4.2). В основе структуры лежит

ритмическое прохождение двух потоков данных x_{in} и y_{in} навстречу друг другу.

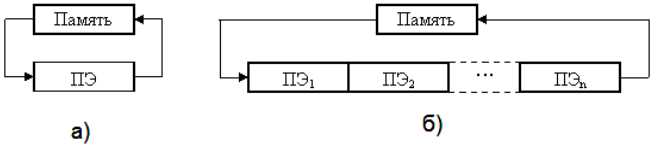


Рис. 4.1. Обработка данных в вычислительных системах:
а) фон-неймановского типа; б) – систолической структуры

Последовательные элементы каждого потока разделены одним тактовым периодом, чтобы любой из них мог встретиться с любым элементом встречного потока. Вычисления выполняются параллельно в процессорных элементах, каждый из которых реализует один шаг в операции вычисления скалярного произведения (рис. 4.2). Значение y_{in}, поступающее на вход ПЭ, суммируется с произведением входных значений x_{in} и c_k. Результат выходит из ПЭ как $y_{out} = y_{in} + c_k * x_{in}$. Значение x_{in}, кроме того, для возможного последующего использования остальной частью массива транслируется через ПЭ без изменений и покидает его в виде x_{out}. Таким образом, на каждый отвод фильтра приходится один ПЭ. На рис. 4.2 показана структура систолического КИХ-фильтра на четыре отвода. Потоки сигналов y_{in}, представляющего накопленный результат вычисления скалярного произведения и входного x_{in} распространяются слева на право. На входах x_{in} и выходах суммы каждого ПЭ добавлены регистерные элементы, что позволяет накопленному результату и входному значению оставаться в синхронизации.

Для получения конечного результата используется сеть сумматоров, которая последовательно суммирует скалярные

произведения. Фильтр состоит из двух однотипных процессорных элементов ПЭ1 и ПЭ2. На вход y_{in} ПЭ1 необходимо подать сигнал логического нуля, а на входной линии x_{in} используется один регистр, а не два, как у ПЭ2.

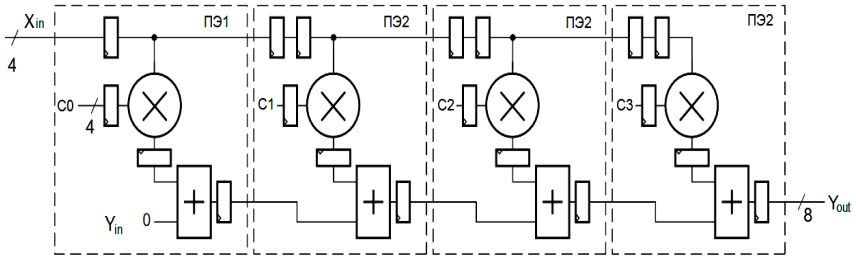


Рис. 4.2. Систолический КИХ-фильтр на четыре отвода составленный из четырех секций DSP48 ПЛИС Virtex-4 фирмы Xilinx

Используя представленные выше соображения разработаем модель систолического фильтра на четыре отвода $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$ в САПР ПЛИС Quartus II в базисе ПЛИС серии Stratix III. Предположим, что коэффициенты фильтра целочисленные со знаком известны и равны $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$. На вход КИХ-фильтра поступают входные отсчеты -5, 3, 1 и 0. Правильные значения на выходе фильтра: 10, -1, -40, -10, 26, 6 и т.д., т.е. согласно модели. В качестве перемножителей используем мегафункцию LPM_MULT. Регистры выполнены на мегафункциях LPM_FF. Систолический КИХ-фильтр на четыре отвода с процессорными элементами ПЭ1 и ПЭ2 в САПР ПЛИС Quartus II показан на рис. 4.3. На рис. 4.4 и рис. 4.5 показаны схемы процессорных элементов ПЭ1 и ПЭ2, а на рис. 4.6 - временные диаграммы работы систолического КИХ-фильтра на четыре отвода.

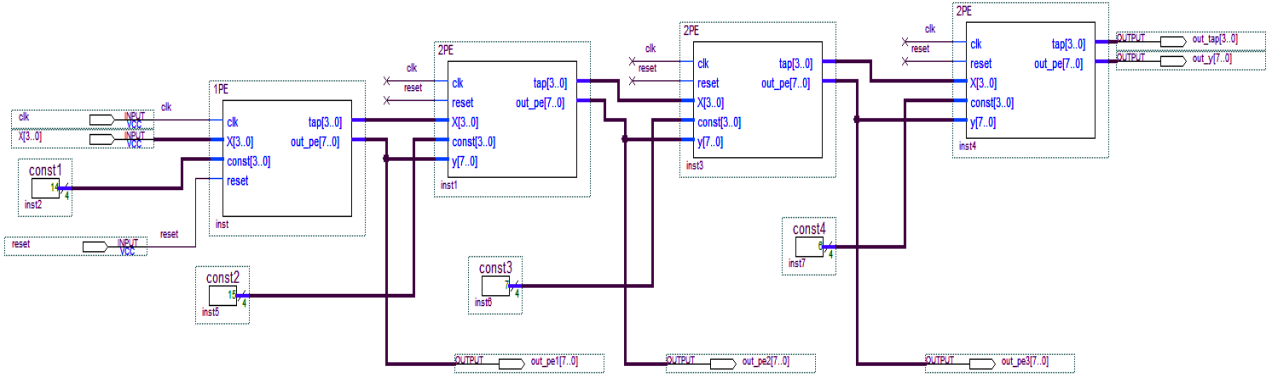


Рис. 4.3. Систолический КИХ-фильтр на четыре отвода в САПР ПЛИС Quartus II с процессорными элементами ПЭ1 и ПЭ2

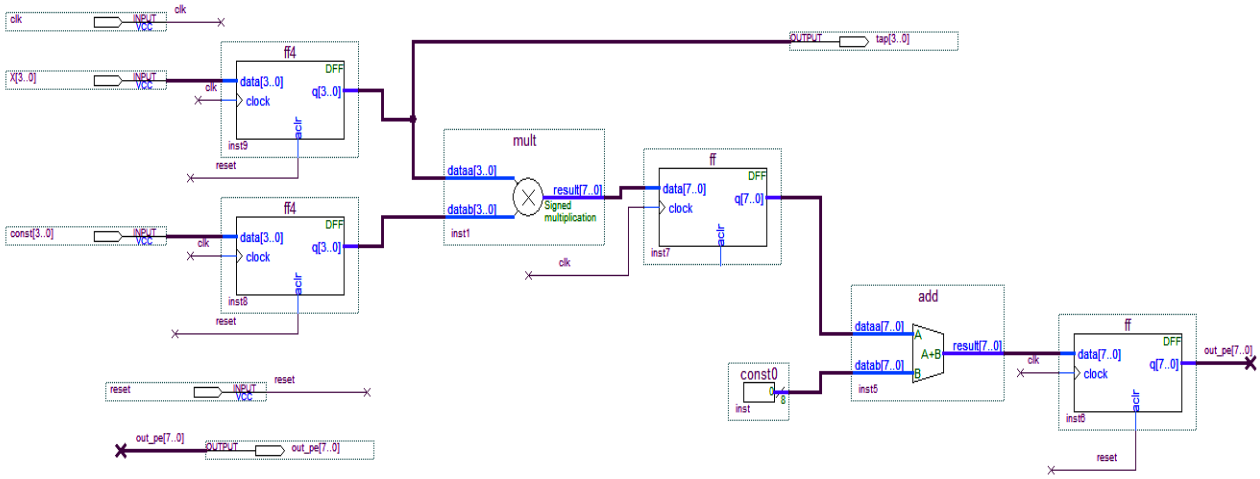


Рис. 4.4. Первый процессорный элемент ПЭ1

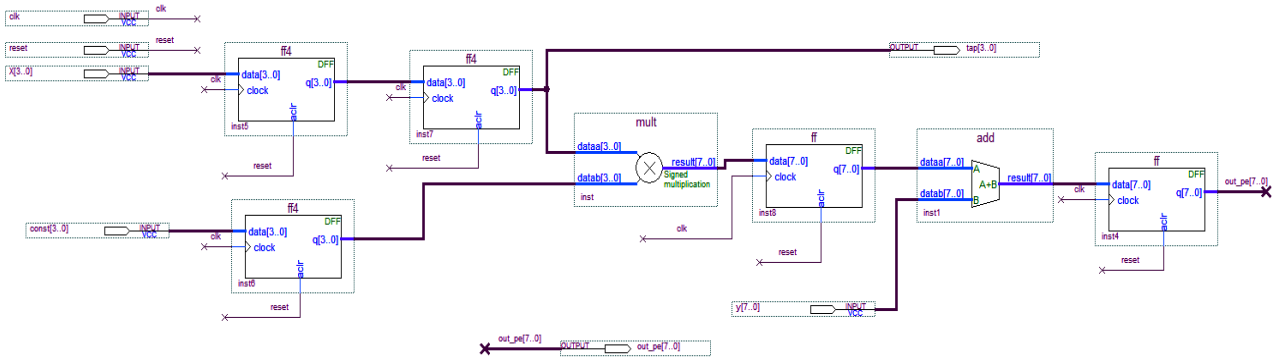


Рис. 4.5. Второй процессорный элемент ПЭ2

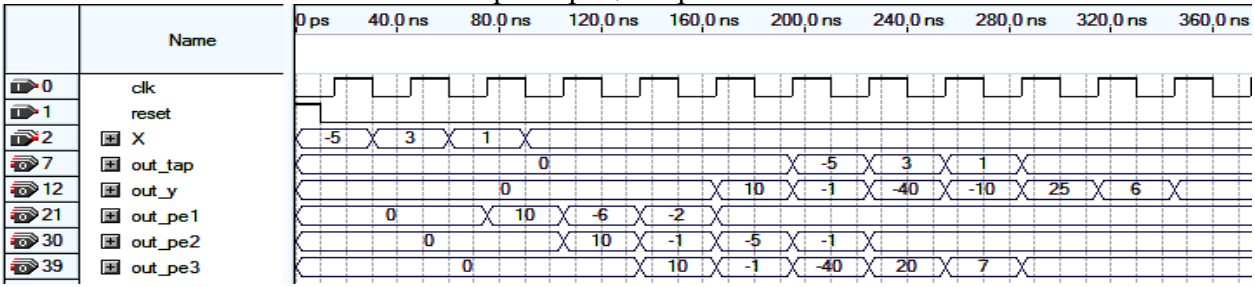


Рис. 4.6. Временные диаграммы работы систолического КИХ-фильтра на четыре отвода с процессорными элементами ПЭ1 и ПЭ2

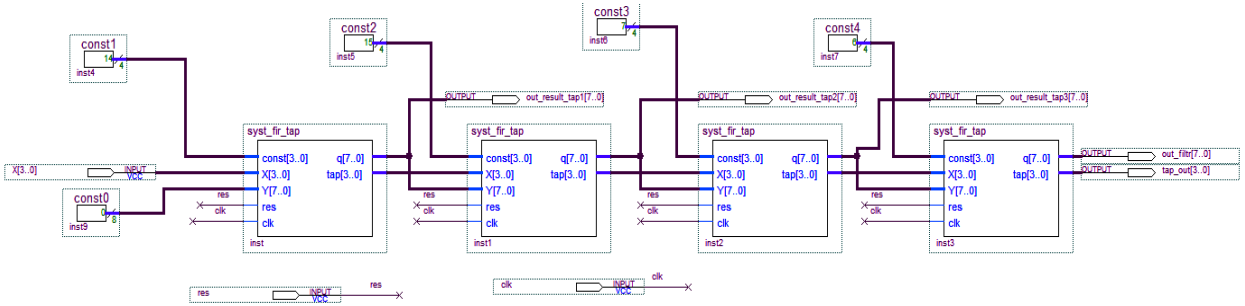


Рис. 4.7. Систолический КИХ-фильтр на четыре отвода в САПР ПЛИС Quartus II с однотипными процессорными элементами

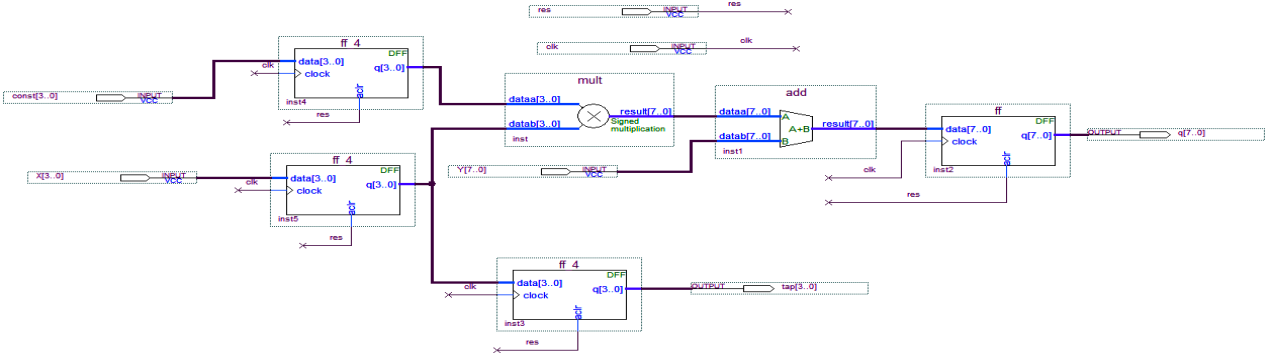


Рис. 4.8. Однотипный процессорный элемент

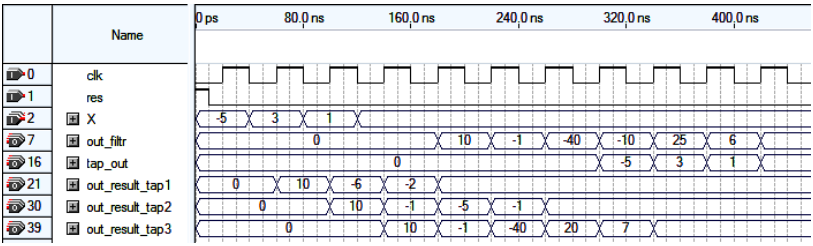


Рис. 4.9. Временные диаграммы работы систолического КИХ-фильтра на четыре отвода с однотипными процессорными элементами

На входах перемножителя сигналов, выполненных на мегафункции ALT_MULT в процессорном элементе 1PE, необходимы по одному 4-разрядному регистру для процесса согласования вычислений. А на одном из входов перемножителя сигналов в процессорном элементе 2PE необходимы два 4-разрядных регистра, первый из которых играет роль линии задержки.

Упростить структуру систолического КИХ-фильтра позволяет использование однотипного процессорного элемента (рис. 4.7 и рис. 4.8). Правильность функционирования такого решения подтверждает диаграмма на рис. 4.9.

Пример 1 демонстрирует код языка VHDL КИХ-фильтра с использованием прямой реализации по формуле $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$. По коду были получены временные диаграммы, показанные на рис. 4.10. Сравнивая временные диаграммы на рис. 4.6, 4.9 и рис. 4.10, видим, что фильтры работают корректно.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
package coeffs is
  type coef_arr is array (0 to 3) of signed(3 downto 0);
```

```
  constant coefs: coef_arr:= coef_arr'("1110", "1111", "0111", "0110");
  end coeffs;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.coeffs.all;
entity fir_var is
 port (clk, reset, clk_ena: in std_logic;
       date: in signed (3 downto 0);
       q_reg: out signed (9 downto 0));
end fir_var;
architecture beh of fir_var is
begin
process(clk,reset)
      type shift_arr is array (3 downto 0)
      of signed (3 downto 0);
      variable shift: shift_arr;
      variable tmp: signed (3 downto 0);
      variable pro: signed (7 downto 0);
      variable acc: signed (9 downto 0);
   begin
if reset='0' then
for i in 0 to 3 loop
shift(i):= (others => '0');
end loop;
q_reg<= (others => '0');
elsif(clk'event and clk = '1') then
if clk_ena='1' then
                        shift(0):=date;
                        pro := shift(0) * coefs(0);
                        acc := conv_signed(pro, 10);
                         for i in 2 downto 0 loop
                        pro := shift(i+1) * coefs(i+1);
                        acc := acc + conv_signed(pro, 10);
                        shift(i+1):= shift(i);
                        end loop;
end if;
```

```
end if;
        q_reg<=acc;
end process;
          end beh;
```

Пример 1. Код языка VHDL КИХ-фильтра на четыре отвода

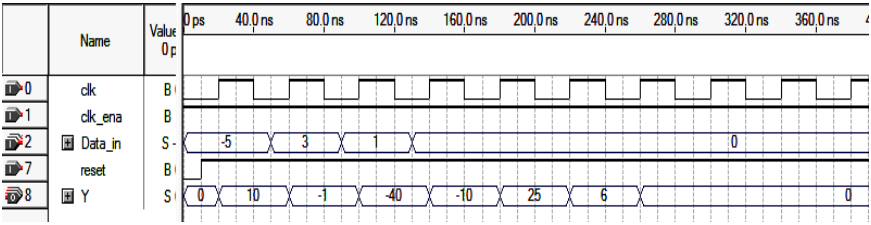


Рис. 4.10. Временные диаграммы работы КИХ-фильтра на четыре отвода по коду языка VHDL

В качестве примера рассмотрим, как обстоит дело с индустриальными ПЛИС последнего поколения фирмы Altera серии Cyclon V и Stratix V. В указанных сериях появились ЦОС-блоки с переменной точностью, одна из особенностей которых это реализация систолических КИХ-фильтров. Каждый ЦОС-блок серии Stratix V позволяет реализовать два перемножителя двух 18-разрядных сигналов или один перемножитель двух 32-разрядных сигналов, причем встроенные в выходные цепи ЦОС-блоков многоразрядные сумматоры позволяют организовать первый и второй уровень в дереве многоразрядных сумматоров в случае проектирования параллельных КИХ-фильтров.

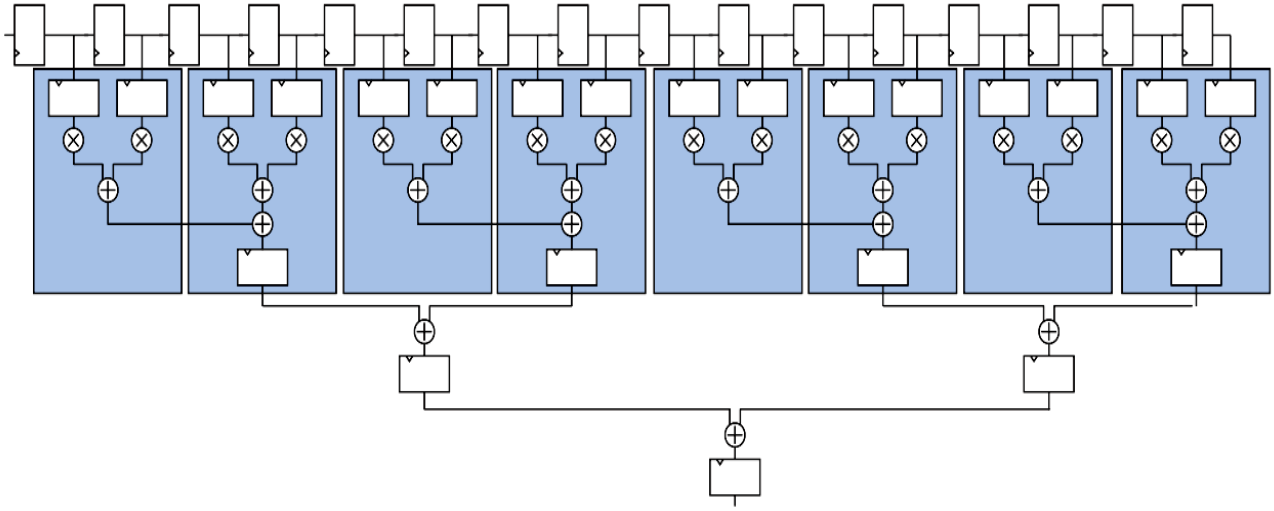
На рис. 4.11 показана прямая форма КИХ-фильтра с несимметричными коэффициентами на 16 отводов, коэффициенты представлены с 18-битной точностью, с использованием ЦОС-блоков ПЛИС серии Stratix V. В ЦОС-блоках используются два уровня суммирования по отношению к внешнему дереву многоразрядных сумматоров, каскадирующая шина и, как вариант, линия задержки может быть сконфигурирована из внутренних входных регистров.

На рис. 4.12 показана прямая форма КИХ-фильтра на 8 отводов с несимметричными коэффициентами с использованием ЦОС-блоков серии Stratix V. Для реализации фильтров с большим числом отводов необходимы внешние линия задержки и дерево многоразрядных сумматоров по отношению к ЦОС-блокам, при этом в самом ЦОС-блоке осуществляется первый уровень суммирования значений с отводов фильтра, предварительно умноженных на его коэффициенты.

Использование ЦОС-блоков при проектировании КИХ-фильтра прямой формы позволяет вдвое сократить число многоразрядных сумматоров, но оставшаяся часть дерева сумматоров реализуется на внутренних ресурсах ПЛИС, что отрицательно сказывается на общем числе задействованных ресурсах и, как следствие, снижается быстродействие фильтра за счет увеличения задержек в трассировочных каналах самой ПЛИС.

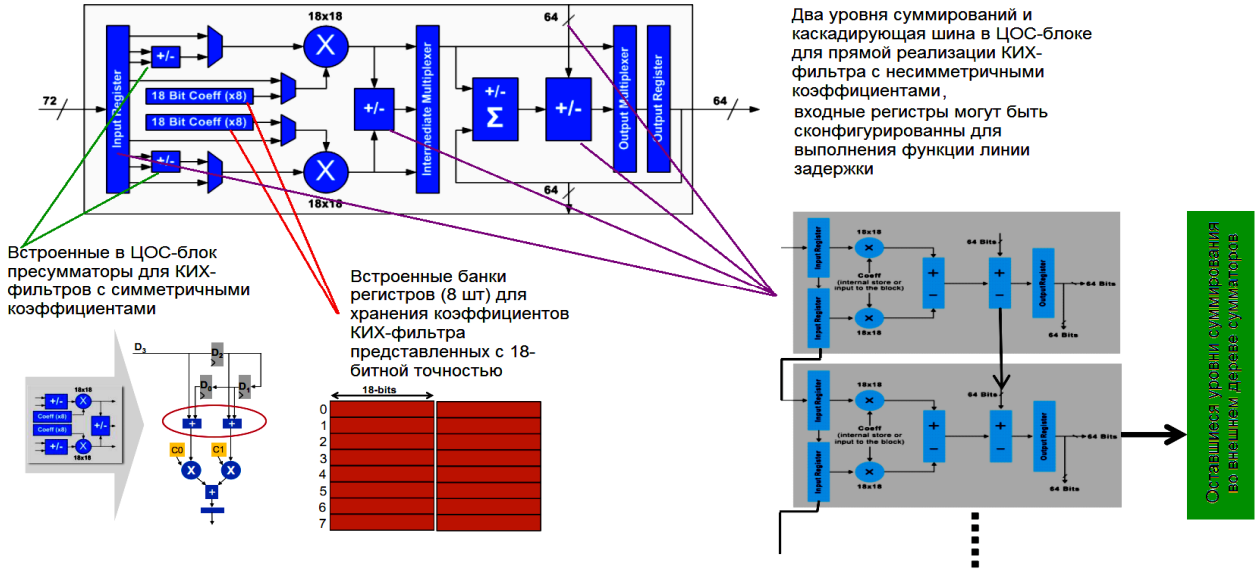
Существенно уменьшить число используемых ресурсов позволяет систолический КИХ-фильтр, в ЦОС-блоках которого реализуются операции умножения и сложения с конвеиеризацией, т.е. отпадает необходимость в дереве многоразрядных сумматоров, однако он может иметь большую латентность по сравнению с прямой реализацией фильтра (рис. 4.13). Мегафункция ALTMULT_ADD САПР Quartus II начиная с версии 11.0 для ПЛИС серий Arria V, Cyclone V и Stratix V позволяет конфигурировать ЦОС-блоки для организации систолических фильтров.

На рис. 4.14 представлена структура ЦОС-блока с переменной точностью ПЛИС серии Cyclon V, на которой, выделен процессорный элемент, а на рис. 4.15 показано включение систолического регистра в мегафункции ALTMULT_ADD для ПЛИС серии Cyclon V.



а)

Рис. 4.11. а) Прямая форма КИХ-фильтра с несимметричными коэффициентами на 16 отводов, коэффициенты представлены с 18-битной точностью с использованием ЦОС-блоков ПЛИС серии Stratix V; б) – режимы ЦОС-блока для прямой реализации КИХ-фильтра с симметричными и несимметричными коэффициентами



б)

Рис. 4.11. а) Прямая форма КИХ-фильтра с несимметричными коэффициентами на 16 отводов, коэффициенты представлены с 18-битной точностью с использованием ЦОС-блоков ПЛИС серии Stratix V; б) – режимы ЦОС-блока для прямой реализации КИХ-фильтра с симметричными и несимметричными коэффициентами (продолжение)

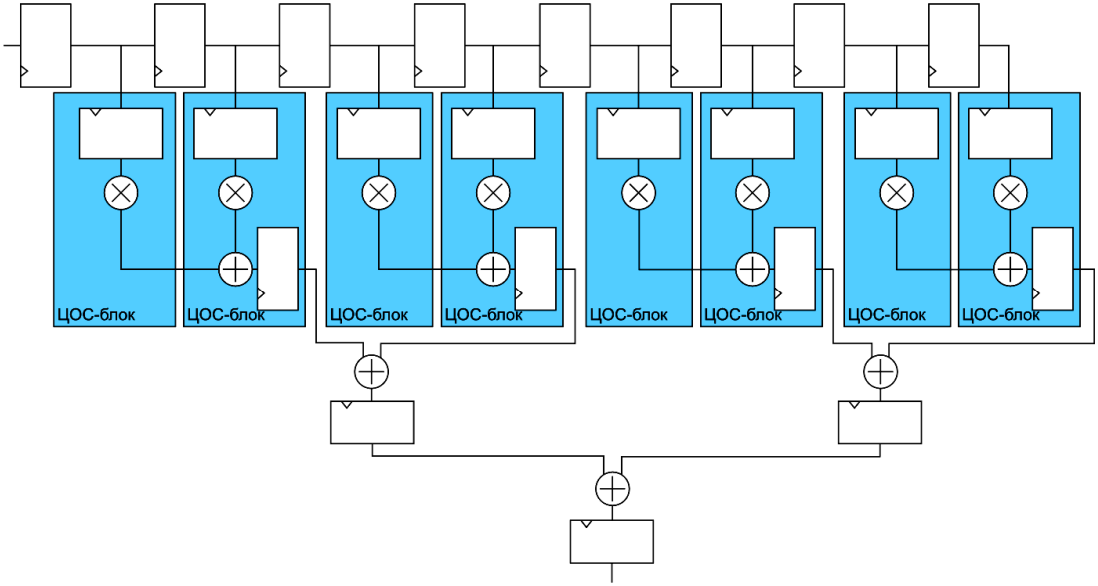


Рис. 4.12. Прямая реализация КИХ-фильтра с несимметричными коэффициентами на восемь отводов, коэффициенты представлены с 27-битной точностью (высокоточный режим) с использованием ЦОС-блоков ПЛИС серии Stratix V. В ЦОС-блоках используется один уровень суммирования по отношению к внешнему дереву многоразрядных сумматоров

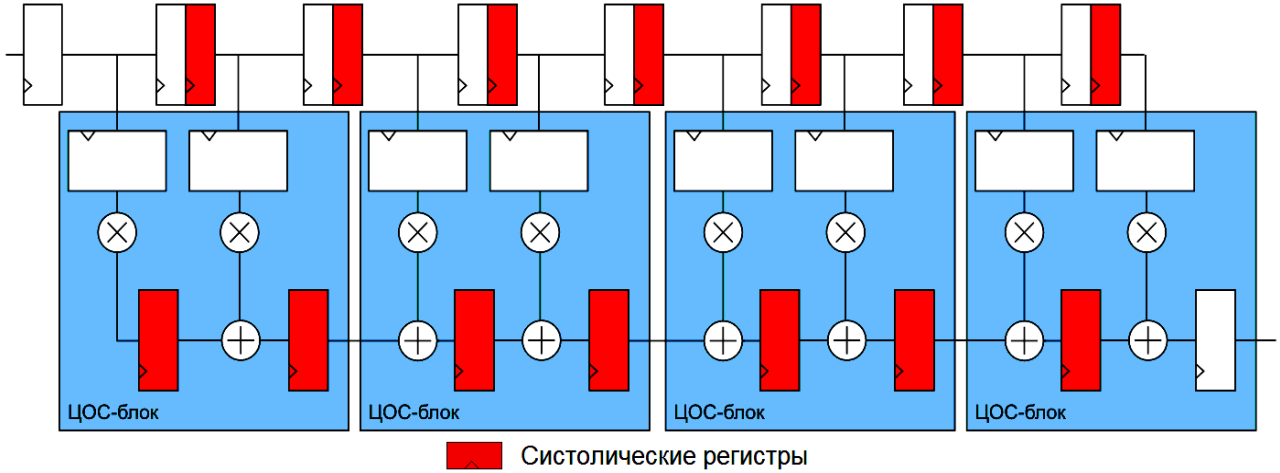


Рис. 4.13. Систолический КИХ-фильтр на восемь отводов с 18-битной точностью представления коэффициентов с использованием ЦОС-блоков ПЛИС серии Stratix V

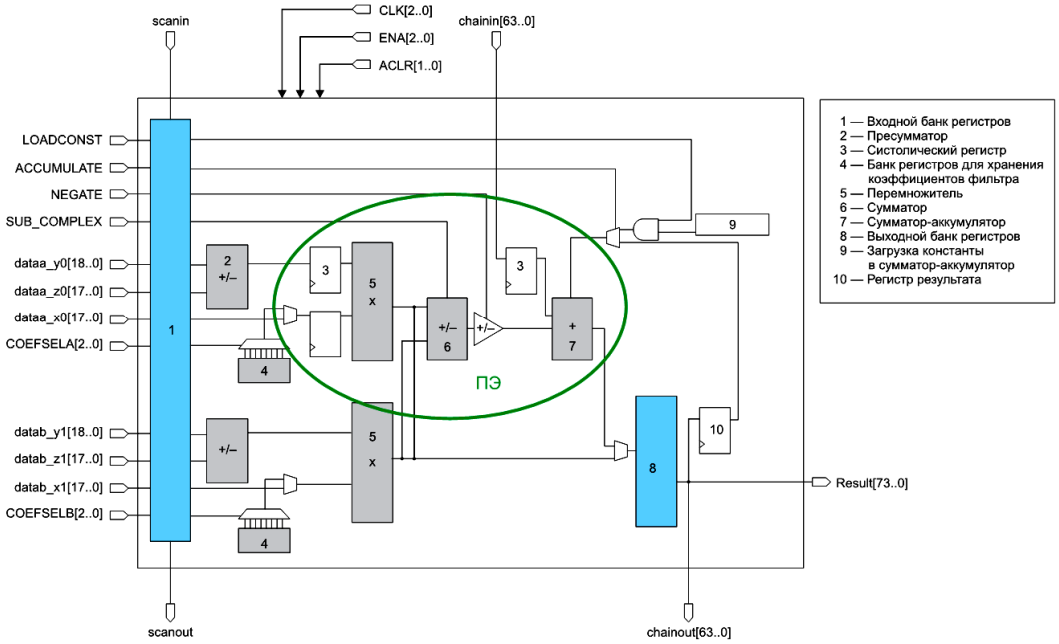


Рис. 4.14. Систолические регистры, подключаемые опционально с помощью мегафункции ALTMULT_ADD в ЦОС-блоке с переменной точность ПЛИС серии Cyclon V

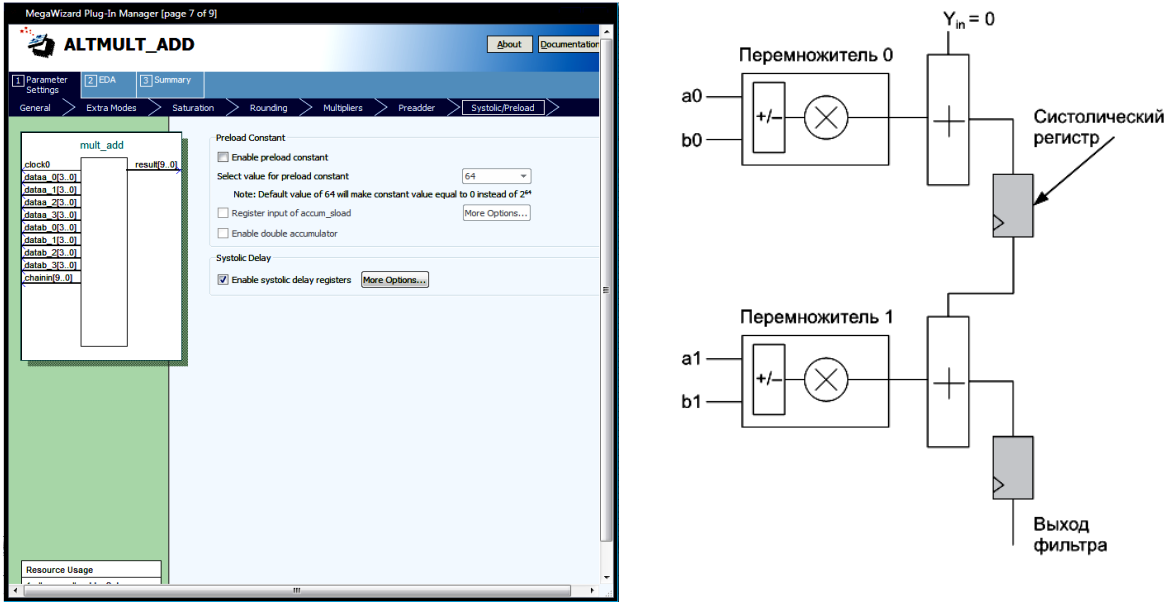


Рис. 4.15. Включение систолического регистра в последовательную цепь сумматоров в мегафункции ALTMULT_ADD для ПЛИС серии Cyclon V

Таблица 4.1

Реализация параллельных КИХ-фильтров на четыре отвода в базис ПЛИС серии Stratix III

Ресурсы ПЛИС	Систолический КИХ-фильтр с однотипными процессорными элементами без использования встроенных ЦОС-блоков	Четыре мегафункции умножения с накоплением ALTMULT_ACCUM (внешняя линия задержки и дерево сумматоров)	Мегафункция умножения и сложения ALTMULT_ADD (внешняя линия задержки)
Максимальная частота/ частота в наихудшем случае, МГц	432/400	429/400	514/400
Число LUT для реализации комбинационныъх функций	59	18	0
Адаптивных логических модулей (ALM)	44	18	8
Число выделенных регистров для реализации последовательностной логики	65	12	12
Встроенные ЦОС-блоки, 18x18 бит	-	16	4

Реализации параллельных КИХ-фильтров на четыре отвода в базис ПЛИС серии Stratix III представлены в табл. 4.1. Все рассмотренные варианты фильтров реализованы на мегафункциях. Первый вариант реализован без использования мегафункциями ЦОС-блоков (рис. 4.7), второй вариант задействует лишь по одному умножителю в блоке из четырех возможных мегафункции ALTMULT_ACCUM (рис. 2.13), а третий – четыре умножителя из четырех возможных в блоке мегафункции ALTMULT_ADD (рис.2.14). Приведенная таблица показывает оптимальное использование ресурсов ЦОС-блоков мегафункцией ALTMULT_ADD за счет использования четырех перемножителей и встроенного дерева сумматоров в блоке, реализующих первые и вторые уровни суммирования. Во всех случаях частота работы фильтров ограничивается величиной 400 МГц.

4.2. Проектирование систолических КИХ-фильтров в базисе ПЛИС с использованием системы цифрового моделирования ModelSim-Altera

Кратко рассмотрим особенности проектирования цифровых фильтров на примере систолического КИХ-фильтра в САПР ПЛИС Quartus II версии 11.1 Web Edition. Начиная с версии 10.0 из САПР Quartus II исключен векторный редактор, а моделирование предлагается вести с использованием различных симуляторов высокоуровневых языков описания аппаратурных средств, например Active-HDL, Riviera-Pro, ModelSim и др. В качестве свободно распространяемого симулятора с ограниченными возможностями пользователю предлагается использовать систему моделирования ModelSim-Altera Free.

Рассмотрим уравнение КИХ-фильтра (нерекурсивного цифрового фильтра с конечно-импульсной характеристикой)

которое представляется как арифметическая сумма произведений:

$$y = \sum_{k=0}^{K-1} c_k \cdot x_k \,, \qquad (4.1)$$

где $y-$ отклик цепи; $x_k -$ $k-$я входная переменная; c_k – весовой коэффициент $k-$й входной переменной, который является постоянным для всех n; K - число отводов фильтра.

Разработаем модель систолического фильтра на четыре отвода $y = C_0x_0 + C_1x_1 + C_2x_2 + C_3x_3$ в САПР ПЛИС Quartus II версии 11.1 в базисе ПЛИС серии Cyclone II с однотипными процессорными элементами (рис. 4.16 и рис. 4.17). ПЛИС серии Cyclone II выбраны из соображений, что в САПР ПЛИС Quartus II Web Edition не поддерживаются ПЛИС серий Cyclon V и Stratix V.

Дадим произвольное имя файлу верхнего уровня проектной иерархии - poly_syst_main.bdf. Предположим что коэффициенты фильтра целочисленные со знаком известны и равны $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$. На вход КИХ-фильтра поступают входные отсчеты -5, 3, 1 и 0. Правильные значения на выходе фильтра: 10, -1, -40, -10, 26, 6 и т.д., т.е. согласно формуле (4.1).

На рис. 4.18 показаны временные диаграммы работы систолического КИХ-фильтра реализованного в базисе ПЛИС Stratix III на четыре отвода с однотипными процессорными элементами в САПР ПЛИС Quartus II версии 9.1.

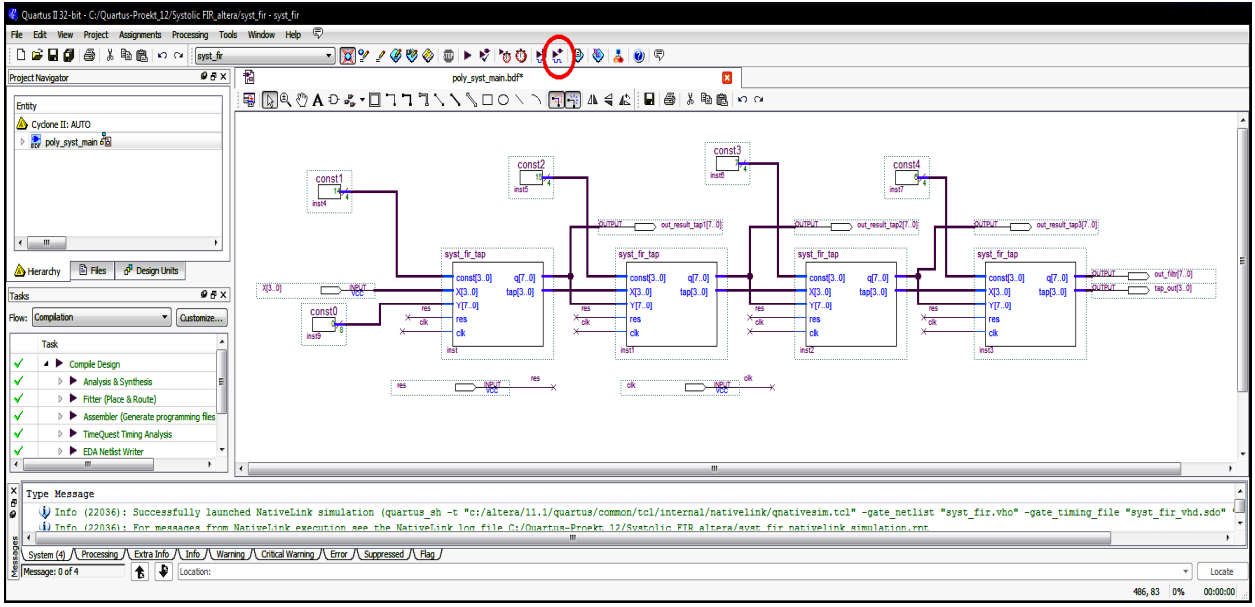


Рис. 4.16. Систолический КИХ-фильтр на четыре отвода в САПР ПЛИС Quartus II версии 11.1 с однотипными процессорными элементами

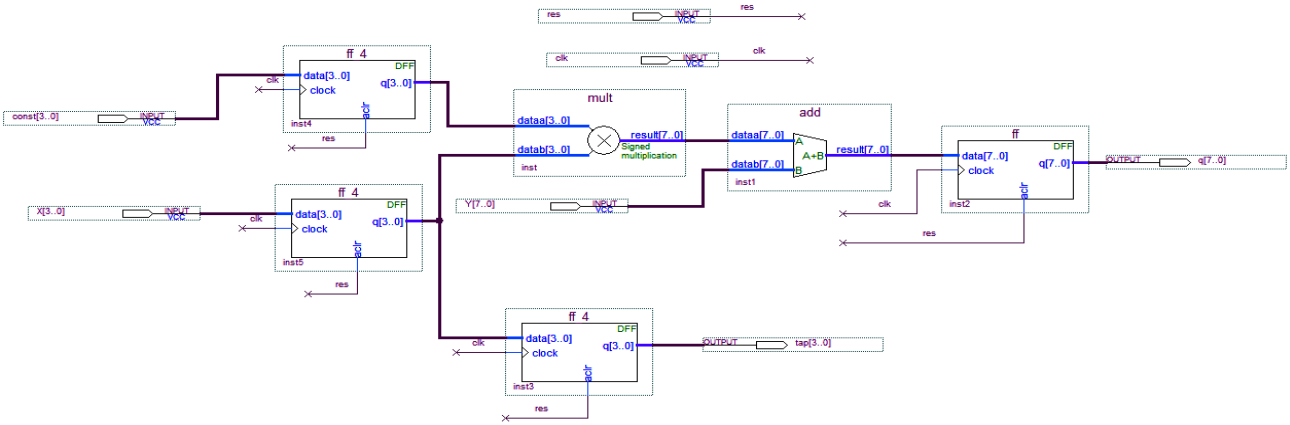


Рис. 4.17. Однотипный процессорный элемент

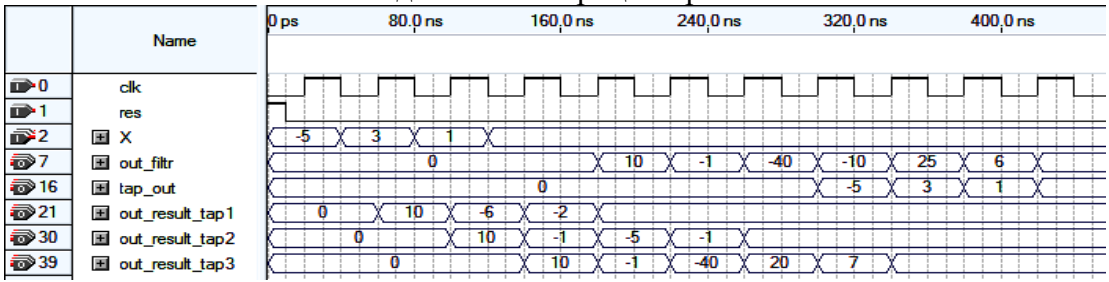


Рис. 4.18. Временные диаграммы работы систолического КИХ-фильтра на четыре отвода с однотипными процессорными элементами в САПР ПЛИС Quartus II версии 9.1

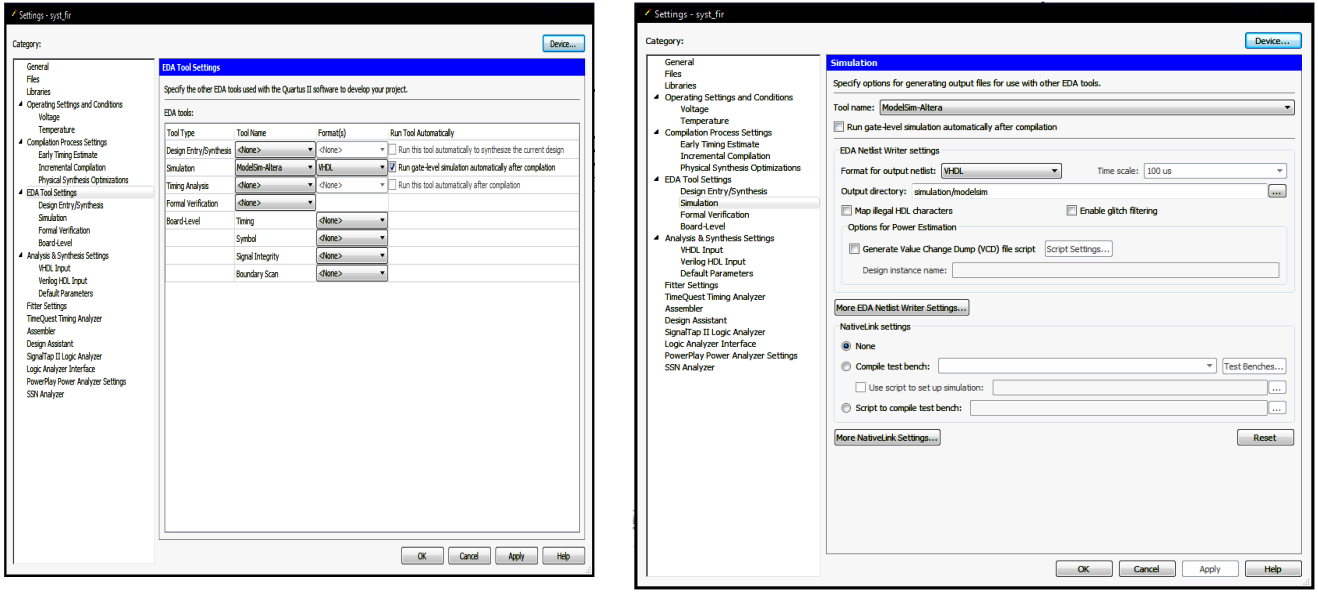


Рис. 4.19. Настройки закладки EDA Tool Settings САПР ПЛИС Quartus II версии 11.1

Рассмотрим моделирование КИХ-фильтра с использованием симулятора ModelSim-Altera STARTER EDITION. Предварительно его необходимо подключить. Это осуществляется с помощью меню Assigments/settings/EDA Tool settings. В поле Tool name САПР ПЛИС Quartus II версии 11.1 необходимо выбрать симулятор ModelSim-Altera а в поле Format Netlist Writer settings указать выходной формат “нетлиста” – язык VHDL (рис. 4.19).

Предварительно необходимо создать текстовый сценарий функционирования систолического КИХ-фильтра с использованием структурного стиля языка VHDL (“тестбенч”). В качестве промежуточного шаблона, который необходимо отредактировать, можно взять файл poly_syst_main.vht. Для этого необходимо его сформировать следующими действиями: меню Processing/Start/Start Test Bench Template Writer (пример 1). Отредактируем объект poly_syst_main_vhd_tst, который основан на компоненте poly_syst_main, и сохраним его под именем test_poly_syst_main.vhd (пример 2).

```
-- Vhdl Test Bench template for design  :  poly_syst_main
-- Simulation tool : ModelSim-Altera (VHDL)

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY poly_syst_main_vhd_tst IS
END poly_syst_main_vhd_tst;
ARCHITECTURE poly_syst_main_arch OF poly_syst_main_vhd_tst IS
-- constants
-- signals
SIGNAL clk : STD_LOGIC;
SIGNAL out_filtr : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL out_result_tap1 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL out_result_tap2 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL out_result_tap3 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL res : STD_LOGIC;
```

```
SIGNAL tap_out : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL X : STD_LOGIC_VECTOR(3 DOWNTO 0);
COMPONENT poly_syst_main
        PORT (
        clk : IN STD_LOGIC;
        out_filtr : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        out_result_tap1 : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        out_result_tap2 : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        out_result_tap3 : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        res : IN STD_LOGIC;
        tap_out : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        X : IN STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
END COMPONENT;
BEGIN
        i1 : poly_syst_main
        PORT MAP (
-- list connections between master ports and signals
        clk => clk,
        out_filtr => out_filtr,
        out_result_tap1 => out_result_tap1,
        out_result_tap2 => out_result_tap2,
        out_result_tap3 => out_result_tap3,
        res => res,
        tap_out => tap_out,
        X => X
        );
init : PROCESS
-- variable declarations
BEGIN
    -- code that executes only once
WAIT;
END PROCESS init;
always : PROCESS
-- optional sensitivity list
-- (       )
-- variable declarations
BEGIN
    -- code executes for every event on sensitivity list
```

```
WAIT;
END PROCESS always;
END poly_syst_main_arch;
```

Пример 1. Шаблон теста систолического КИХ-фильтра на языке VHDL (poly_syst_main.vht)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;
ENTITY test_poly_syst_main IS
END test_poly_syst_main;

ARCHITECTURE behavior OF test_poly_syst_main IS
COMPONENT poly_syst_main
PORT(
res :  IN  STD_LOGIC;
clk :  IN  STD_LOGIC;
X :  IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
out_filtr :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
out_result_tap1 :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
out_result_tap2 :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
out_result_tap3 :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
tap_out :  OUT  STD_LOGIC_VECTOR(3 DOWNTO 0));
END COMPONENT;
--Inputs
SIGNAL clk :  std_logic := '0';
SIGNAL res :  std_logic := '1';
SIGNAL X_in:  STD_LOGIC_VECTOR(3 DOWNTO 0) := "1011";
--Outputs
SIGNAL out_systol_filtr : std_logic_VECTOR(7 DOWNTO 0);
SIGNAL out_result_tap1 :  std_logic_vector(7 downto 0);
SIGNAL out_result_tap2 :  std_logic_vector(7 downto 0);
SIGNAL out_result_tap3 :  std_logic_vector(7 downto 0);
SIGNAL tap_out :  std_logic_vector(3 downto 0);
BEGIN
        uut: poly_syst_main PORT MAP(
```

```
                clk => clk,
                res => res,
                X => X_in,
                out_filtr => out_systol_filtr,
                out_result_tap1 => out_result_tap1,
                out_result_tap2 => out_result_tap2,
                out_result_tap3 => out_result_tap3,
                tap_out => tap_out
        );
  process
   begin
                clk <= '0';
                wait for 50 ns;
                clk <= '1';
                wait for 50 ns;
   end process;
process
   begin
                wait for 125 ns;
                res <= '0';
   end process;
tb : process
        begin
                wait for 100 ns;
                X_in <= "1011";
                wait for 100 ns;
                X_in <= "0011";
                wait for 100 ns;
                X_in <= "0001";
                wait for 100 ns;
                X_in <= "0000";
        wait;
        END process;
END;
```

Пример 2. Тест систолического КИХ-фильтра на языке VHDL (test_poly_syst_main.vhd)

После компиляции в САПР Quartus II при нажатии кнопки EDA Gate Level Simulation (моделирование на уровне вентилей, временная модель “Slow Model”) автоматически должен запуститься симулятор ModelSim-Altera (на рис. 4.16 пиктограмма кнопки отмечена кружком). Возможны два варианта. Рассмотрим первый вариант с созданием “тестбенча”. Для этого необходимо откомпилировать файл test_poly_syst_main.vhd с помощью меню Compile симулятора ModelSim. После компиляции в рабочей библиотеке work должны появиться два объекта poly_syst_main и test_poly_syst_main (рис. 4.20). Двойным щелчком мыши по объекту test_poly_syst_main автоматически запускаются различные вспомогательные окна (рис. 4.21).

Ставим курсор в окно Objects, нажимаем на правую кнопку мыши меню Add/To Wave/Signals in Region и в окне Wave появляется список сигналов проекта. Далее целесообразно настроить окно Wave, в котором отображаются временные диаграммы работы фильтра. Выбираем Simulate\Runtime Options. В поле, система счисления (Default Radix) нажимаем радиокнопку Decimal, что позволяет перейти от двоичной системы счисления, представленной в тестбенче, к десятичной со знаком. В поле Default Run задаем шаг моделирования 100 ns. Последовательно нажимая на пиктограмму кнопки Run с шагом 100 ns, получим временные диаграммы работы КИХ-фильтра (рис. 4.21). Сравниваем полученные результаты с временными диаграммами на рис. 4.18, убеждаемся в правильности работы систолического КИХ-фильтра на четыре отвода в базисе ПЛИС Cyclone II.

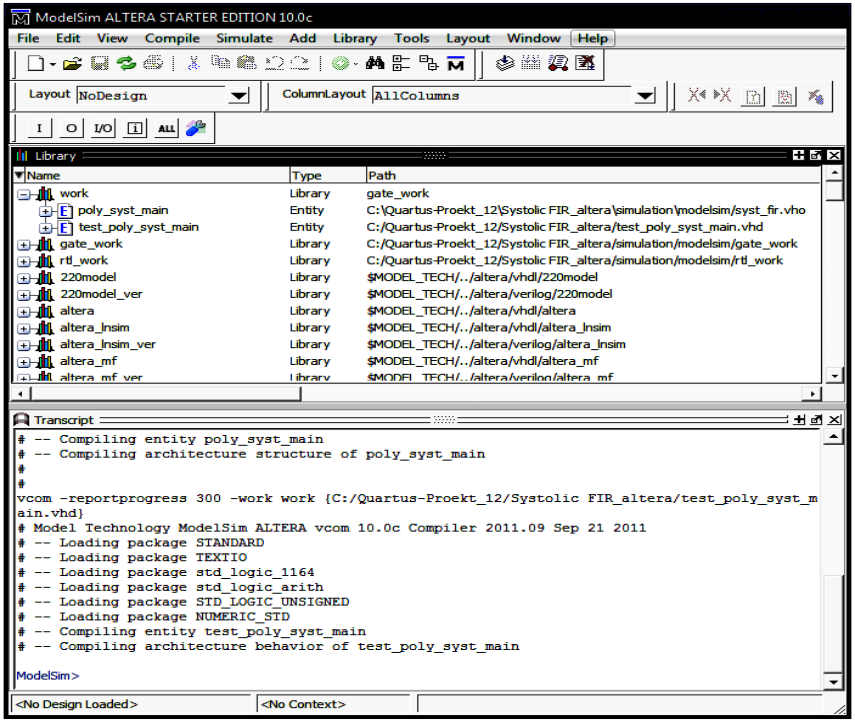


Рис. 4.20. Создается рабочая библиотека work, в которую помещаются два объекта poly_syst_main и test_poly_syst_main

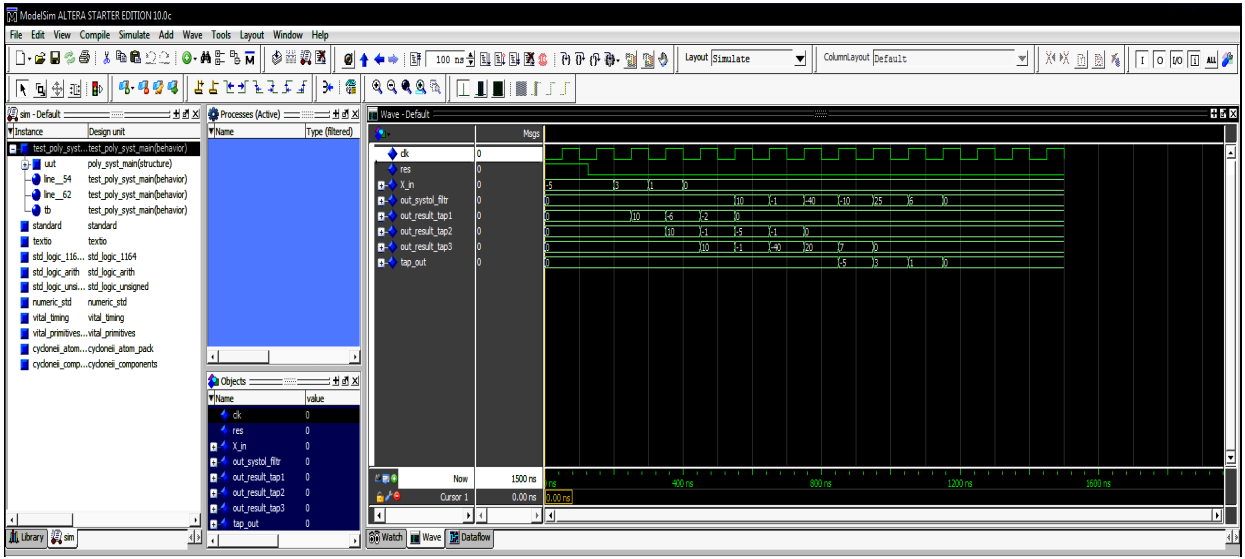


Рис. 4.21. Временные диаграммы работы систолического КИХ-фильтра на четыре отвода с однотипными процессорными элементами в симуляторе ModelSim-Altera версии 10.c

Рассмотрим второй вариант без использования тестбенча. Для этого будем использовать файл poly_syst_main.vhd, а задание на моделирование сформируем непосредственно в векторном редакторе (окно Wave) с помощью специальных инструментов Clock и Force. Двойным щелчком мыши по объекту poly_syst_main автоматически запускаются различные окна. Ставим курсор в окно Objects, как и в первом варианте, нажимаем на правую кнопку мыши меню Add/To Wave/Selected Signals и выбираем только интересующие нас сигналы.

В окне Wave выбираем синхросигнал clk и нажимаем на правую кнопку мыши, выбираем меню Clock. В окне Define Clock в полях задается период синхросигнала - 100 ns, коэффициент заполнения - 50, уровни логической единицы и нуля, радиокнопкой выбираем активным передний фронт синхросигнала (рис. 4.22).

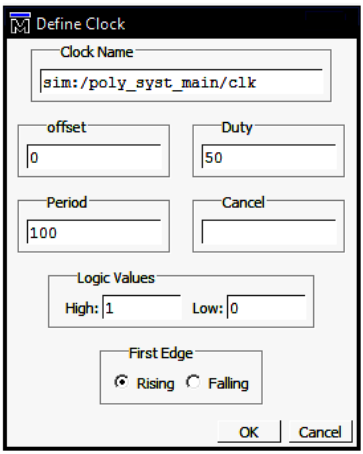


Рис. 4.22. Окно настройки тактового синхросигнала clk

Сигнал асинхронного сброса res (активный – высокий уровень) внутренних регистров процессорных элементов зададим равным нулю с помощью меню Force (действие над сигналом). Радиокнопкой отмечаем вид действия над

сигналом Freeze (“замороженный”), в поле Value зададим логический ноль, т.е. на всем временном промежутке моделирования сигнал сброса res будет неактивным (рис. 4.23).

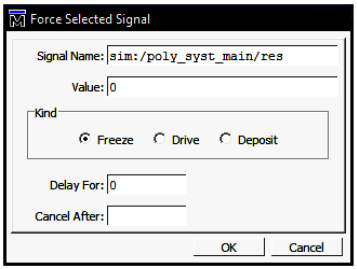


Рис. 4.23. Настройка сигнала сброса res

Далее выбираем сигнал X и с помощью меню Force задаем десятичное число -5 (рис. 4.24). Затем нажимаем на пиктограмму кнопки Run для осуществления моделирования с шагом 100 ns. Далее будем изменять только значения, поступающие на вход X с помощью меню Force, и последовательно нажимать на кнопку Run до получения нужных откликов (пример 3 и рис. 4.25). Задание для моделирования (пример 3) можно использовать, повторно сохранив в текстовый файл.

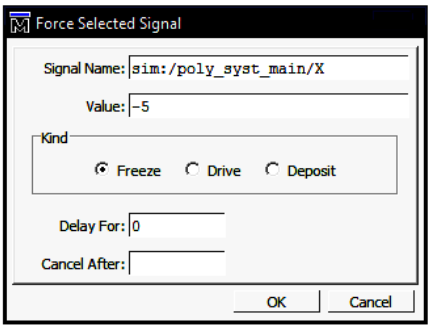


Рис. 4.24. Настройка сигнала X

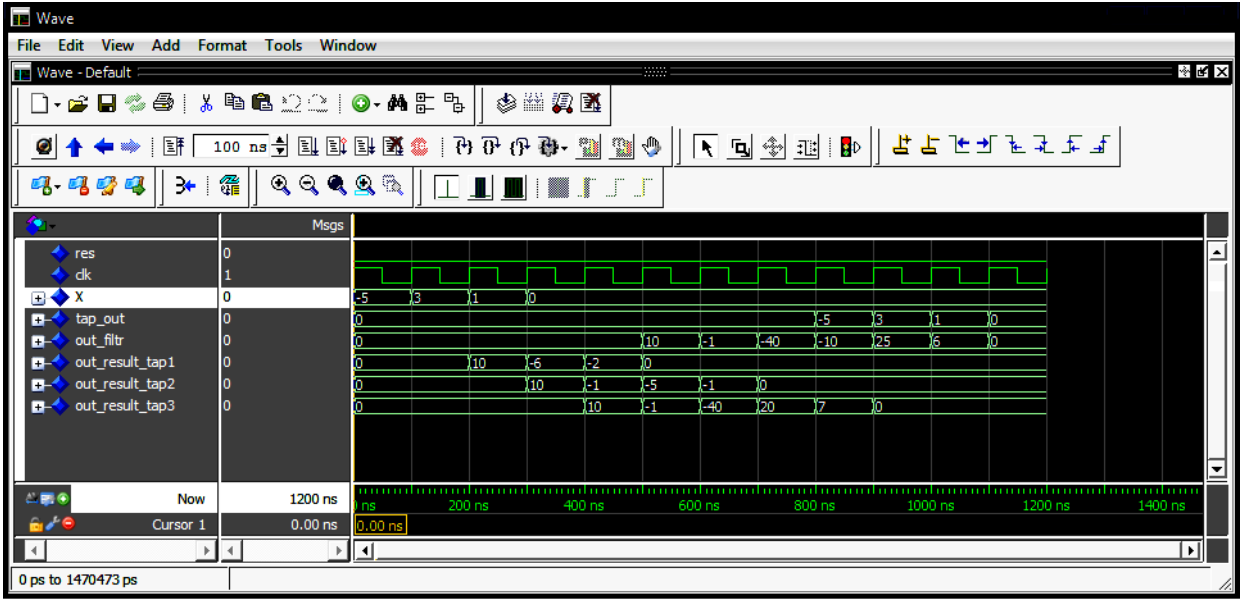


Рис. 4.25. Окно Wave с результатами моделирования КИХ-фильтра

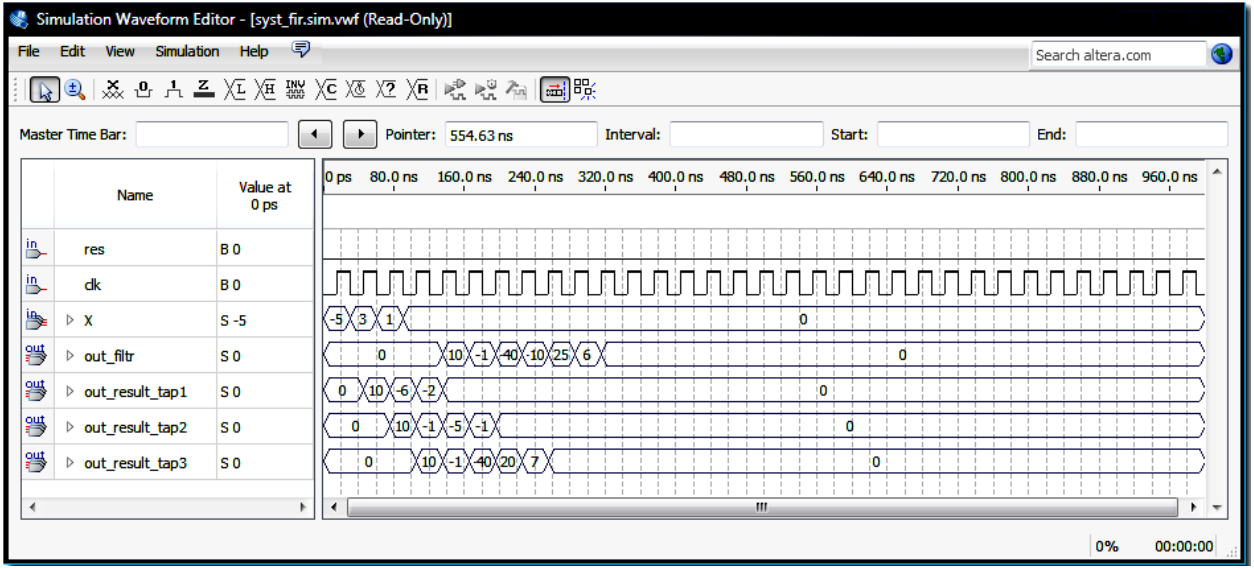


Рис. 4.26. Временные диаграммы работы систолического КИХ-фильтра на четыре отвода с однотипными процессорными элементами в САПР ПЛИС Quartus II версии 13.0

```
force -freeze sim:/poly_syst_main/res 0 0
force -freeze sim:/poly_syst_main/clk 1 0, 0 {50000 ps} -r {100 ns}
force -freeze sim:/poly_syst_main/X -5 0
run
force -freeze sim:/poly_syst_main/X 3 0
run
force -freeze sim:/poly_syst_main/X 1 0
run
force -freeze sim:/poly_syst_main/X 0 0
```

Пример 3. Задание для моделирования из окна Transcript

В версии 13.0 САПР ПЛИС Quartus вновь появился встроенный векторный редактор с собственной системой моделирования. Создать векторный файл можно с помощью меню File/New/Verification/Debugging Files/University Program VWF. Запуск моделирования осуществляется непосредственно из окна Simulation Waveform Editor с помощью меню Simulation/Run Functional Simulation. На рис. 4.26 показаны временные диаграммы работы систолического КИХ-фильтра на четыре отвода в САПР ПЛИС Quartus II версии 13.0. Проект размещен в ПЛИС серии MAX II.

Использование текстового сценария на языке VHDL совместно с симулятором ModelSim-Altera Free позволяет пользователю отлаживать сложные проекты в кратчайшие сроки.

4.3. Проектирование КИХ-фильтров в САПР ПЛИС Xilinx ISE 14.2

Рассмотрим простейшие примеры проектирования КИХ-фильтров в базисе ПЛИС фирмы Xilinx. По VHDL-коду КИХ-фильтра на четыре отвода (файл filter4_4.vhd, пример 3, раздел 3.1, глава 3) в САПР ПЛИС ISE разработаем проект для реализации в базисе ПЛИС фирмы Xilinx. Для симуляции проекта разработаем тестбенч (файл теста на языке VHDL для заданий значений входных сигналов или Test Bench-файл) (fir4_test_bench) (пример 1). На вход КИХ-фильтра будут поступать входные отсчеты -5, 3, 1 и 0 (метка tb в файле fir4_test_bench). На рис. 4.27 показана закладка реализация проекта в САПР ПЛИС Xilinx ISE 14.2. По VHDL-коду (файл filter4_4.vhd) создаем символ (рис. 4.28) и делаем верхним уровнем иерархии проекта схемный файл (sch-файл). На рис. 4.29 показаны временные диаграммы работы КИХ-фильтра на четыре отвода.

```
-- VHDL Test Bench Created by ISE for module: FIR4
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY fir4_test_bench IS
END fir4_test_bench;
ARCHITECTURE behavior OF fir4_test_bench IS
   -- Component Declaration for the Unit Under Test (UUT)
   COMPONENT filter_4
   PORT(
        din : IN  std_logic_vector(3 downto 0);
        Sout : OUT  std_logic_vector(7 downto 0);
        reset : IN  std_logic;
        clk : IN  std_logic
       );
   END COMPONENT;
  --Inputs
  signal din : std_logic_vector(3 downto 0) := "1011";
```

```
  signal reset : std_logic := '1';
  signal clk : std_logic := '0';
        --Outputs
  signal Sout : std_logic_vector(7 downto 0);
BEGIN
        -- Instantiate the Unit Under Test (UUT)
  uut: filter_4    PORT MAP (
       din => din,
       Sout => Sout,
       reset => reset,
       clk => clk
      );
  -- Clock process definitions
  clk_process :process
  begin
                clk <= '0';
                wait for 50 ns;
                clk <= '1';
                wait for 50 ns;
  end process;
  -- Stimulus process
 process
  begin
                wait for 80 ns;
                reset <= '0';
  end process;

        tb : process
                begin
                        wait for 100 ns;
                        din <= "1011";
                        wait for 100 ns;
                        din <= "0011";
                        wait for 100 ns;
                        din <= "0001";
                        wait for 100 ns;
                        din <= "0000";
```

```
        wait;
        END process;
END;
```

Пример 1. Тестбенч КИХ-фильтра на четыре отвода

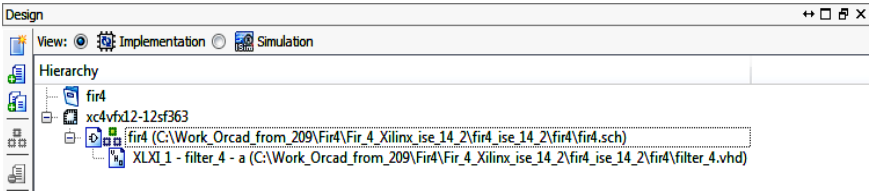


Рис. 4.27. Закладка реализация проекта в САПР ПЛИС Xilinx ISE 14.2. Верхний уровень иерархии sch-файл

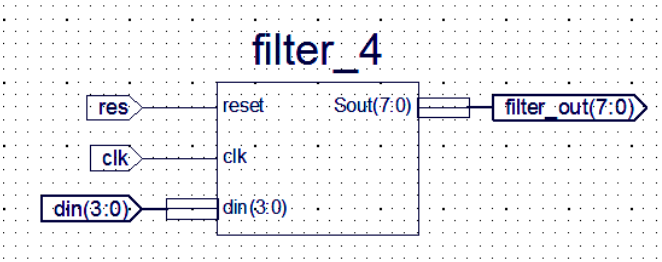


Рис. 4.28. Проект КИХ-фильтра на четыре отвода

Рассмотрим пример использования edif-формата для проектирования КИХ-фильтров. Созданные проектировщиком файлы описания проекта на языке VHDL могут быть преобразованы средствами синтеза логики в технологически специфицируемый (Technology-specific netlist) файл соединений или список связей в формате EDIF (Electronic design interchange format, формат для обмена электронными проектами между различными САПР) (**.edf**), который затем используется в САПР ПЛИС Xilinx ISE 14.2 на этапах размещения и трассировки.

Иерархическая структура edif-формата включает в себя библиотечные модули, модули ячеек, программу просмотра,

интерфейсный модуль, информационный модуль и схемные представления.

Программа синтеза логики Synplicity (рис. 4.30) транслирует и оптимизирует VHDL-проекты на уровне RTL-представления в список соединений в формате EDIF, эквивалентный уровню примитивных логических элементов (уровень вентилей), т.е. осуществляет переход с RTL-уровня на уровень вентилей. Комбинационные функции при этом отображаются в таблицы соответствий (LUT) ПЛИС. Этот формат затем компилируется в различный технологический базис ПЛИС. Доступны технологические базисы различных производителей ПЛИС: Xilinx, Altera, Actel, Lattice, QuickLogic и Silicon Blue.

Предположим, что проект КИХ-фильтра состоит из двух VHDL-файлов. Верхнего vir4.vhd (пример 2) и нижнего filter_4.vhd уровня (рис. 4.30).

На рис. 4.31 показан выбор технологического базиса для реализации проекта. Выбирается ПЛИС серии Viterx 2 XC2V40.

В настройках синтезатора Synplicity заказываем выходной edif-файл (пример 3, рис. 4.32) и выбираем для реализации проекта ПЛИС типа ППВМ серии Viterx 2 XC2V40.

После генерации edif-файла разработаем проект в САПР ПЛИС Xilinx ISE 14.2. Назначим верхним уровнем иерархии проекта файл filter_4.edf сгенерированный в Synplicity и для реализации проекта будем использовать современную ПЛИС XC4VFX12 серии Virtex IV.

Разработаем тестбенч (fir4_test_bench) для симуляции проекта и подключим его к проекту (рис. 4.34). Имя компоненты необходимо изменить на fir4 (пример 4) в отличие от тестбенча по коду VHDL (пример 1). Тестбенч описывает моделирование импульсной характеристики фильтра (рис. 4.35).

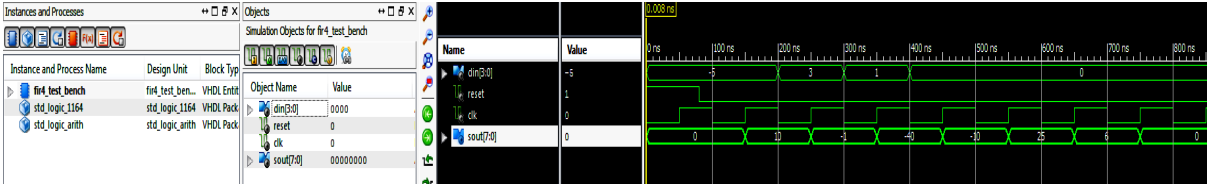


Рис. 4.29. Симулятор ISim САПР ПЛИС Xilinx ISE 14.2. Временные диаграммы работы КИХ-фильтра на четыре отвода. На вход КИХ-фильтра поступают входные отсчеты -5, 3, 1 и 0. Правильные значения на выходе фильтра:
10, -1, -40, -10, 26, 6

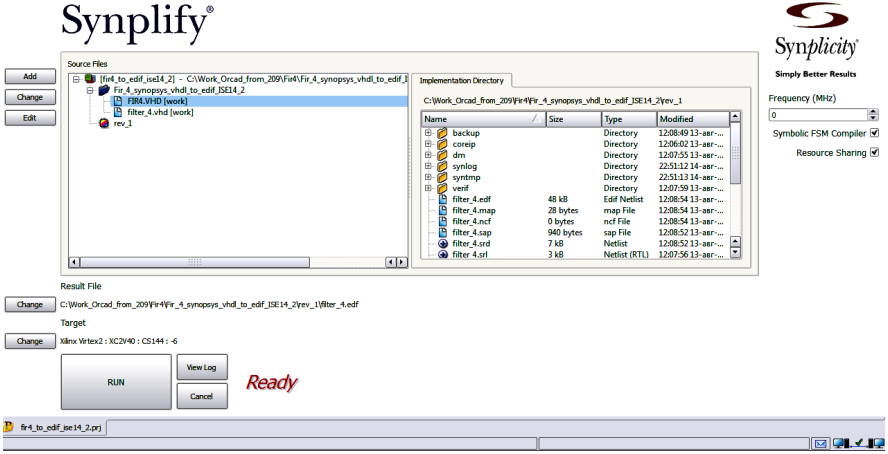


Рис. 4.30. Программа синтеза логики Synplicity

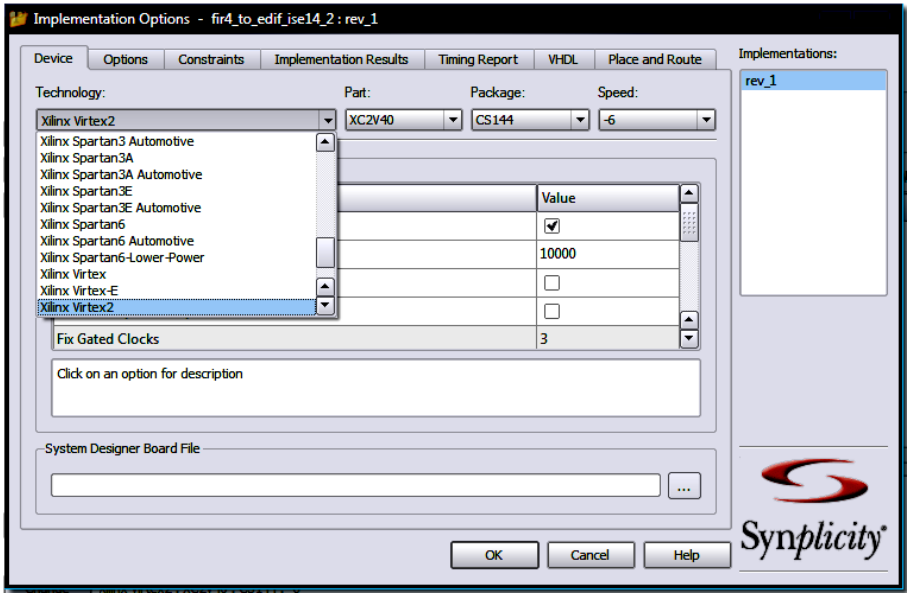


Рис. 4.31. Закладка “приборы”. Выбор технологического базиса ПЛИС Viterx 2 XC2V40 для реализации проекта

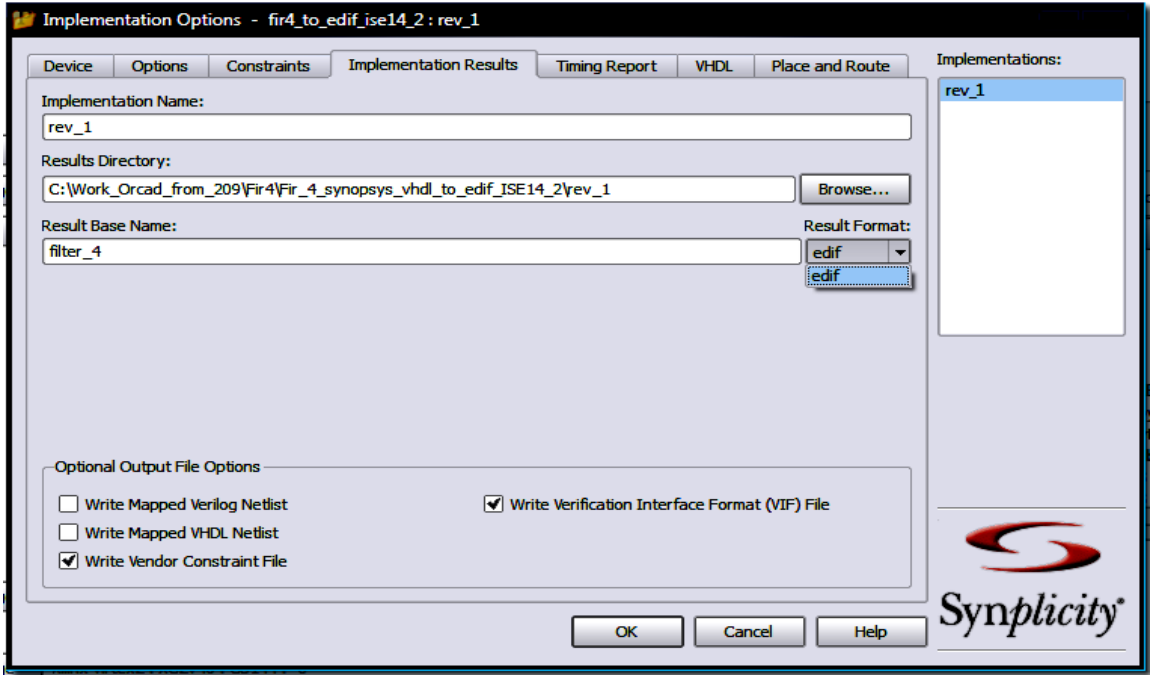


Рис. 4.32. Закладка “реализация результатов синтеза”. Заказываем выходной edif-файл

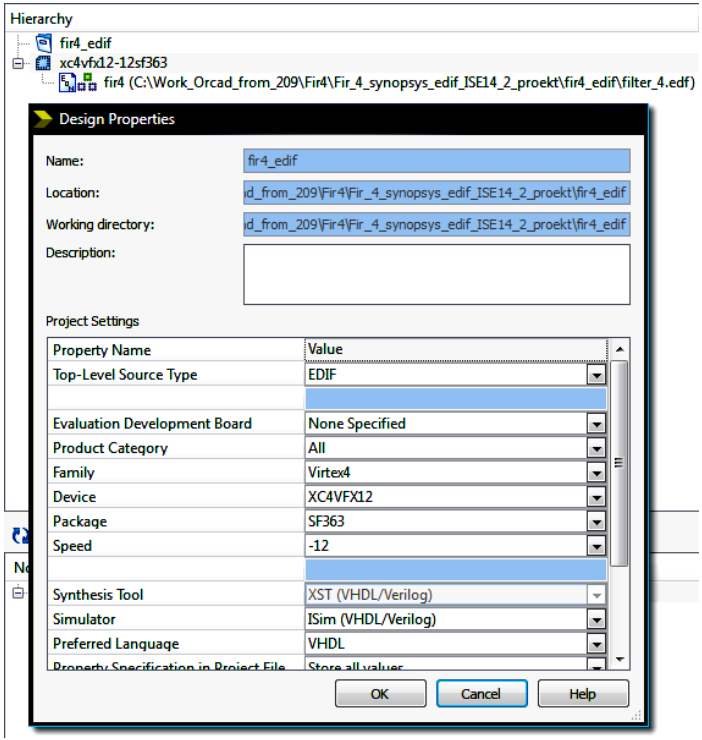


Рис. 4.33. Закладка “реализация проекта” в САПР ПЛИС Xilinx ISE 14.2. Верхний уровень иерархии edf-файл (filter_4.edf)

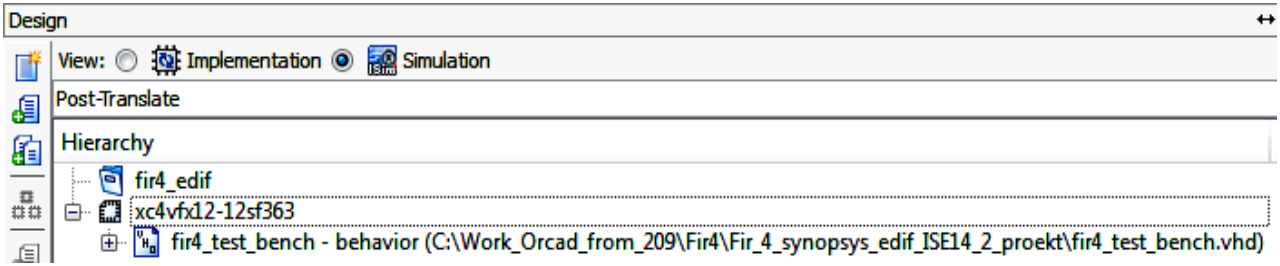


Рис. 4.34. Закладка “симуляция проекта”. Подключение тест бенча fir4_test_bench к проекту

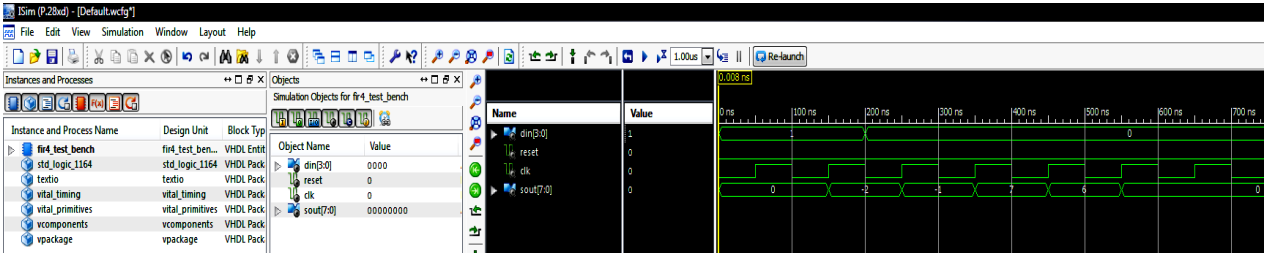


Рис. 4.35. Симулятор ISim САПР ПЛИС Xilinx ISE 14.2. Импульсная характеристика КИХ-фильтра

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
ENTITY fir4 IS PORT (
din : IN std_logic_vector(3 DOWNTO 0);
Sout : OUT std_logic_vector(7 DOWNTO 0);
reset : IN std_logic;
clk : IN std_logic
);
END fir4;
ARCHITECTURE STRUCTURE OF fir4 IS
-- COMPONENTS
COMPONENT filter_4
PORT (
din : IN std_logic_vector(3 DOWNTO 0);
reset : IN std_logic;
clk : IN std_logic;
Sout : OUT std_logic_vector(7 DOWNTO 0)
); END COMPONENT;
BEGIN
fir : filter_4       PORT MAP(
din(3) => DIN(3),
din(2) => DIN(2),
din(1) => DIN(1),
din(0) => DIN(0),
reset => RESET,
clk => CLK,
Sout(7) => SOUT(7),
Sout(6) => SOUT(6),
Sout(5) => SOUT(5),
Sout(4) => SOUT(4),
Sout(3) => SOUT(3),
Sout(2) => SOUT(2),
Sout(1) => SOUT(1),
Sout(0) => SOUT(0)
);
END STRUCTURE;
```

Пример.2. Структурное описание проекта на языкеVHDL (vir4.vhd)

```
(edif (rename filter_4 "fir4")
```

```
 (edifVersion 2 0 0)
 (edifLevel 0)
 (keywordMap (keywordLevel 0))
 (status
  (written
   (timeStamp 2014 8 13 12 8 52)
   (author "Synopsys, Inc.")
   (program "Synplify" (version "E-2010.09-1, mapper maprc, Build
142R"))
  )
 )
 (library VIRTEX
  (edifLevel 0)
  (technology (numberDefinition ))
  (cell IBUFG (cellType GENERIC)
   (view PRIM (viewType NETLIST)
    (interface
     (port O (direction OUTPUT))
     (port I (direction INPUT))
    )
   )
  )
```

Пример 3. Фрагмент edf-файла, полученный с помощью синтезатора Synplicity

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY fir4_test_bench IS
END fir4_test_bench;
ARCHITECTURE behavior OF fir4_test_bench IS
   -- Component Declaration for the Unit Under Test (UUT)
   COMPONENT fir4
   PORT(
        din : IN  std_logic_vector(3 downto 0);
        Sout : OUT  std_logic_vector(7 downto 0);
        reset : IN  std_logic;
        clk : IN  std_logic
       );
   END COMPONENT;
  --Inputs
  signal din : std_logic_vector(3 downto 0) := "0001";
```

```
  signal reset : std_logic := '0';
  signal clk : std_logic := '0';
--Outputs
  signal Sout : std_logic_vector(7 downto 0);
BEGIN
        -- Instantiate the Unit Under Test (UUT)
  uut: fir4       PORT MAP (
       din => din,
       Sout => Sout,
       reset => reset,
       clk => clk
      );
  -- Clock process definitions
  clk_process :process
  begin
                clk <= '0';
                wait for 50 ns;
                clk <= '1';
                wait for 50 ns;
  end process;
  -- Stimulus process
 process
  begin
                wait for 30 ns;
                reset <= '0';
  end process;

        tb : process
                begin
                        wait for 100 ns;
                        din <= "0001";
                        wait for 100 ns;
                        din <= "0000";
                        wait for 100 ns;
                        din <= "0000";
                        wait for 100 ns;
                        din <= "0000";
        wait;
        END process;
END;
```

Пример 4. Тестбенч импульсной характеристики КИХ-фильтра на четыре отвода

4.4. Пример проектирования КИХ-фильтров в базисе ПЛИС с применением генератора параметризированных ядер XLogiCORE IP и функции FIR Compiler v6.3

Рассмотрим пример проектирования КИХ-фильтра в базисе ПЛИС фирмы Xilinx с использованием САПР ISE Design Suite версии 14.2. Для ускорения процесса разработки проекта КИХ-фильтра воспользуемся генератором параметризированных ядер XLogiCORE IP и функцией FIR Compiler v6.3. Выберем бюджетную ПЛИС Spartan-6 XC6SLX4 с поддержкой протокола AXI содержащую 8 ЦОС-блоков DSP48A1 располагающихся в двух секциях по четыре в каждой.

На рис. 4.36 показан проект КИХ-фильтра в САПР ПЛИС Xilinx ISE 14.2 с использованием генератора параметризированных ядер XLogiCORE IP FIR Compiler v6.3.

Настройка функции FIR Compiler v6.3 осуществляется в несколько шагов. Задаются варианты считывания значений коэффициентов КИХ-фильтра: из текстового файла (coe-файл) или представление в виде вектора значений, а также параметры спецификации фильтра (рис .4.37).

По известным коэффициентам происходит построение АЧХ КИХ-фильтра в автоматическом режиме. Задаются границы полосы пропускания и задерживания (подавления), неравномерность АЧХ в полосе пропускания и минимальное затухание в полосе задерживания.

Выбирается одноканальная структура фильтра типа Single-Rate FIR (входная частота дискретизации равна выходной частоте дискретизации). Частота дискретизации определяется как fclk/N для несимметричного и как fclk/N+1 для симметричного фильтра, где fclk-частота тактирования ядра фильтра; N-разрядность входной шины данных (точность представления входных значений подлежащих фильтрации).

Частота тактирования ядра фильтра установлена в 250 МГц а входная частота дискретизации – 50 МГц.

Проектирование КИХ-фильтра осуществляется в формате с фиксированной запятой. На рис. 4.38 показан учет эффектов квантования. Коэффициенты фильтра не симметричные, целочисленные со знаком $C_0 = -2$, $C_1 = -1$, $C_2 = 7$ и $C_3 = 6$, разрядность представления коэффициентов – четыре бита. Предполагаем, что на вход фильтра поступают только целые значения, как со знаком, так и без, например -5, 3, 1, 0. Разрядность представления значений входного сигнала подлежащего фильтрации – четыре бита, профильтрованного сигнала – восемь бит. Под дробную часть числа в обоих случаях отводим 0 бит (Input Data Fractional Bits и Output Fractional Bits).

На рис. 4.39 показан выбор структуры фильтра. Используем прямую форму систолического фильтра, в котором операции умножения и сложения выполняются параллельно с конвейеризацией (рис.4.40). В каждой секции доступно 4 ЦОС-блока располагающиеся в столбец, а для реализации КИХ-фильтра требуется 1 ЦОС-блок.

Применение систолических КИХ-фильтров в проектах пользователя позволяет существенно уменьшить число используемых ресурсов и повысить быстродействие. Поддержка систолических структур также обеспечивается мегафункцией (ALTMULT_ADD) САПР Altera Quartus II для работы с ЦОС-блоками ПЛИС серий Cyclon V, Arria V и Stratix V.

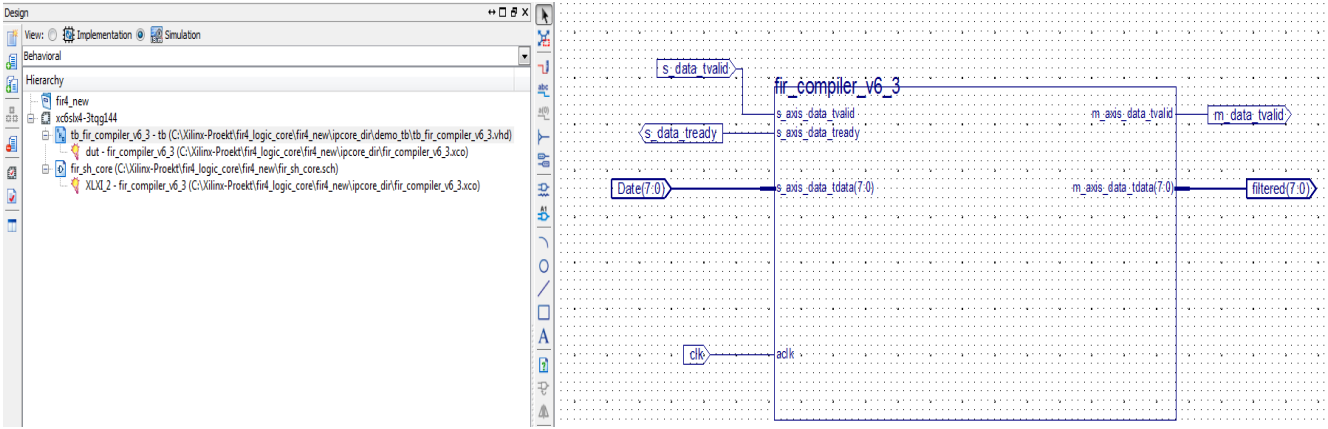


Рис. 4.36. Проект КИХ-фильтра в САПР ПЛИС Xilinx ISE 14.2 с использованием генератора параметризированных ядер XLogiCORE IP FIR Compiler v6.3. Верхний уровень иерархии - схемный файл (sch-файл)

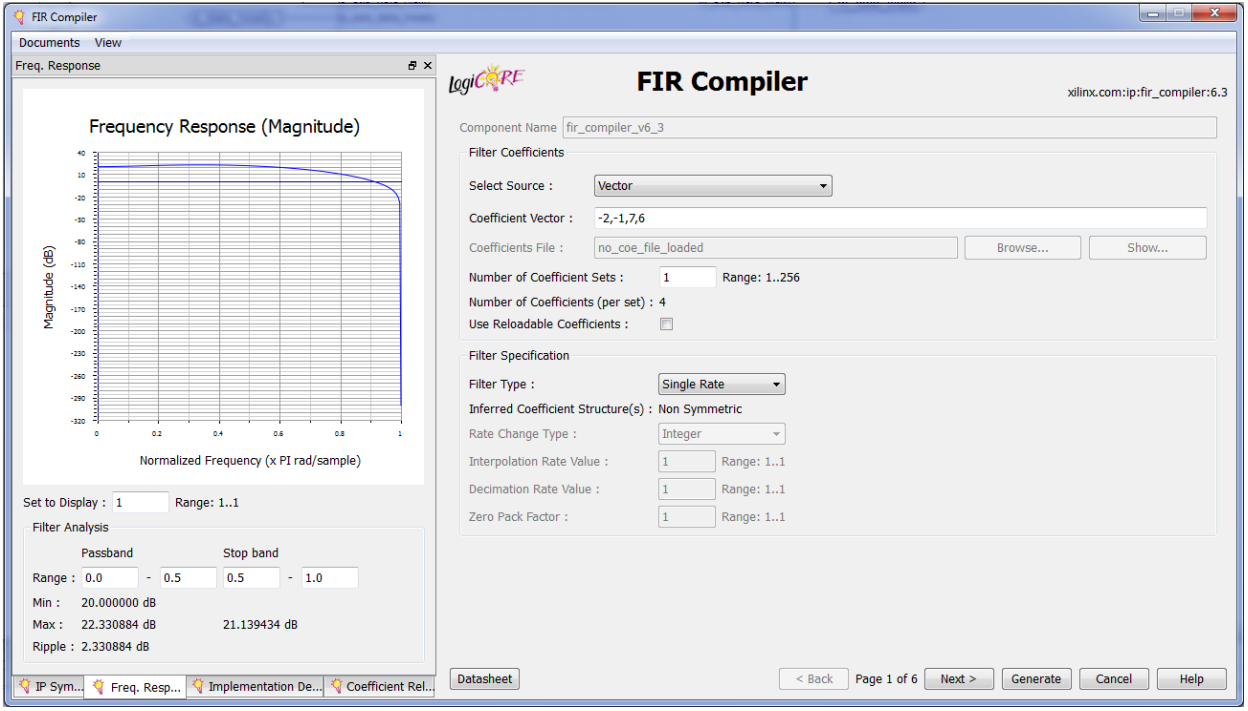


Рис. 4.37. Настройки функции FIR Compiler

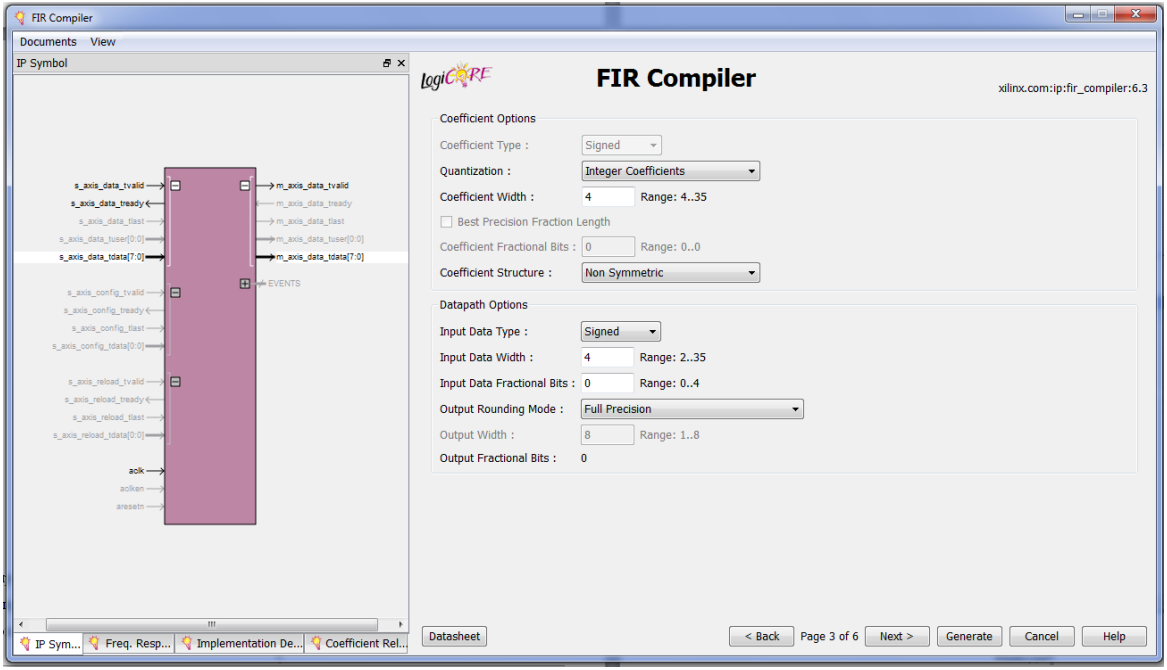
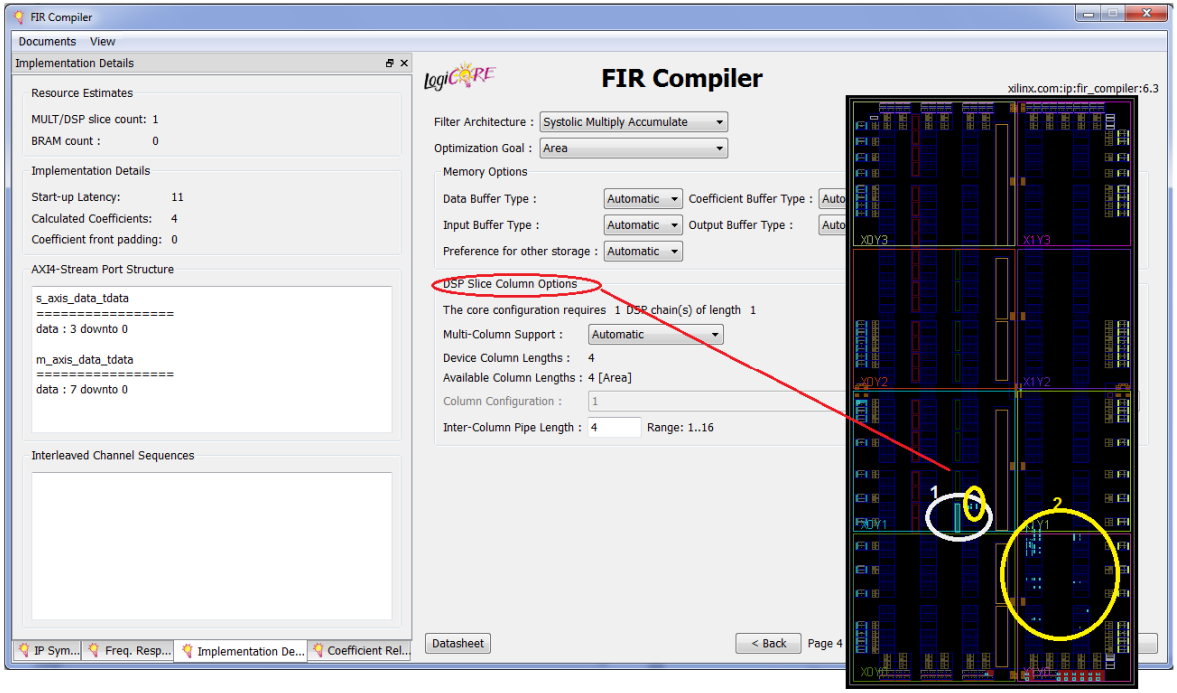


Рис. 4.38. Учет эффектов квантования коэффициентов КИХ-фильтра, точности представления входных и выходных значений (сигналов). Опции позволяющие настроить форматы представления коэффициентов и входных/выходных значений фильтра



1 ЦОС-блок DSP48A1;
2 Логические ресурсы

Рис. 4.39. Выбор архитектуры фильтра и настройка секций ЦОС-блоков

Для симуляции проекта в автоматическом режиме сгенерируем тестбенч (файл теста на языке VHDL для заданий значений входных сигналов или Test Bench-файл) (пример 1). На рис. 4.41 показан симулятор ISim САПР ПЛИС Xilinx ISE 14.2. Демонстрируется прохождение единичного импульса по структуре фильтра (импульсная характеристика КИХ-фильтра на четыре отвода полученная с использованием тестбенча сгенерированного в автоматическом режиме). Латентность фильтра 11 тактов синхрочастоты. Для размещения проекта в базис ПЛИС XC6SLX4 требуется 48 триггеров тактируемых фронтом синхросигнала из общих логических ресурсов ПЛИС и один ЦОС-блок DSP48A1.

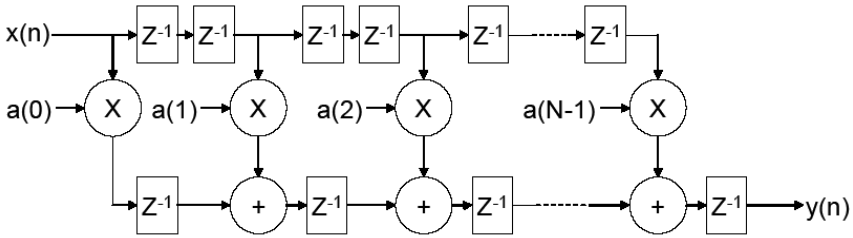


Рис. 4.40. Прямая форма систолического КИХ-фильтра с конвериезацией

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity tb_fir_compiler_v6_3 is
end tb_fir_compiler_v6_3;
architecture tb of tb_fir_compiler_v6_3 is
  ---------------------------------------------------------------------
  -- Timing constants
  ---------------------------------------------------------------------
  constant CLOCK_PERIOD : time := 100 ns;
  constant T_HOLD       : time := 10 ns;
  constant T_STROBE     : time := CLOCK_PERIOD - (1 ns);
```

```
  ------------------------------------------------------------------------
  -- DUT signals
  ------------------------------------------------------------------------
General signals
signal aclk  : std_logic := '0';  -- the master clock

-- Data slave channel signals
signal s_axis_data_tvalid : std_logic := '0';  -- payload is valid
signal s_axis_data_tready : std_logic := '1';  -- slave is ready
signal s_axis_data_tdata  : std_logic_vector(7 downto 0) := (others
=> '0');  -- data payload
-- Data master channel signals
signal m_axis_data_tvalid: std_logic := '0';  -- payload is valid
signal m_axis_data_tdata  : std_logic_vector(7 downto 0) := (others
=> '0');  -- data payload
---------------------------------------------------------------------
-- Aliases for AXI channel TDATA and TUSER fields
-- These are a convenience for viewing data in a simulator
waveform viewer.
-- If using ModelSim or Questa, add "-voptargs=+acc=n" to the
vsim command
-- to prevent the simulator optimizing away these signals.
---------------------------------------------------------------------
-- Data slave channel alias signals
signal s_axis_data_tdata_data: std_logic_vector(3 downto 0) :=
(others => '0');
-- Data master channel alias signals
signal m_axis_data_tdata_data: std_logic_vector(7 downto 0) :=
(others => '0');
begin
---------------------------------------------------------------------
-- Instantiate the DUT
---------------------------------------------------------------------
dut : entity work.fir_compiler_v6_3
```

```
 port map (
     aclk                    => aclk,
     s_axis_data_tvalid      => s_axis_data_tvalid,
     s_axis_data_tready      => s_axis_data_tready,
     s_axis_data_tdata       => s_axis_data_tdata,
     m_axis_data_tvalid      => m_axis_data_tvalid,
     m_axis_data_tdata       => m_axis_data_tdata
     );
-------------------------------------------------------------------------
-- Generate clock
-------------------------------------------------------------------------
  clock_gen : process
  begin
    aclk <= '0';
    wait for CLOCK_PERIOD;
    loop
      aclk <= '0';
      wait for CLOCK_PERIOD/2;
      aclk <= '1';
      wait for CLOCK_PERIOD/2;
    end loop;
  end process clock_gen;
-------------------------------------------------------------------------
-- Generate inputs
-------------------------------------------------------------------------
stimuli : process
   -- Procedure to drive a number of input samples with specific
data
   -- data is the data value to drive on the tdata signal
   -- samples is the number of zero-data input samples to drive
procedure drive_data ( data    : std_logic_vector(7 downto 0);
                       samples : natural := 1 ) is
     variable ip_count : integer := 0;
   begin
```

```
      ip_count := 0;
      loop
        s_axis_data_tvalid <= '1';
        s_axis_data_tdata  <= data;
        loop
          wait until rising_edge(aclk);
          exit when s_axis_data_tready = '1';
        end loop;
        ip_count := ip_count + 1;
        wait for T_HOLD;
      -- Input rate is 1 input each 5 clock cycles: drive valid inputs at
this rate
        s_axis_data_tvalid <= '0';
        wait for CLOCK_PERIOD * 4;
        exit when ip_count >= samples;
      end loop;
    end procedure drive_data;
    -- Procedure to drive a number of zero-data input samples
    -- samples is the number of zero-data input samples to drive
    procedure drive_zeros ( samples : natural := 1 ) is
    begin
      drive_data((others => '0'), samples);
    end procedure drive_zeros;
-- Procedure to drive an impulse and let the impulse response
emerge on the data master channel
-- samples is the number of input samples to drive; default is enough
for impulse response output to emerge
    procedure drive_impulse ( samples : natural := 11 ) is
      variable impulse : std_logic_vector(7 downto 0);
    begin
      impulse := (others => '0');  -- initialize unused bits to zero
      impulse(3 downto 0) := "0001";
      drive_data(impulse);
      if samples > 1 then
```

```
      drive_zeros(samples-1);
    end if;
  end procedure drive_impulse;
 begin
  -- Drive inputs T_HOLD time after rising edge of clock
  wait until rising_edge(aclk);
  wait for T_HOLD;
  -- Drive a single impulse and let the impulse response emerge
  drive_impulse;
  -- Drive another impulse, during which demonstrate use and
effect of AXI handshaking signals
  drive_impulse(2);  -- start of impulse; data is now zero
  s_axis_data_tvalid <= '0';
  wait for CLOCK_PERIOD * 25;  -- provide no data for 5 input
samples worth
  drive_zeros(2);  -- 2 normal input samples
  s_axis_data_tvalid <= '1';
  wait for CLOCK_PERIOD * 25;  -- provide data as fast as the
core can accept it for 5 input samples worth
  drive_zeros(2);  -- back to normal operation
  -- End of test
  report "Not a real failure. Simulation finished successfully."
severity failure;
  wait;
 end process stimuli;
 ------------------------------------------------------------------
 -- Check outputs
 ------------------------------------------------------------------
 check_outputs : process
  variable check_ok : boolean := true;
 begin
  -- Check outputs T_STROBE time after rising edge of clock
  wait until rising_edge(aclk);
  wait for T_STROBE;
```

```
    -- Do not check the output payload values, as this requires the behavioral model
    -- which would make this demonstration testbench unwieldy.
    -- Instead, check the protocol of the master DATA channel:
    -- check that the payload is valid (not X) when TVALID is high
    if m_axis_data_tvalid = '1' then
      if is_x(m_axis_data_tdata) then
        report "ERROR: m_axis_data_tdata is invalid when m_axis_data_tvalid is high" severity error;
        check_ok := false;
      end if;
    end if;
    assert check_ok
      report "ERROR: terminating test with failures." severity failure;
  end process check_outputs;
  ------------------------------------------------------------------------
  -- Assign TDATA / TUSER fields to aliases, for easy simulator waveform viewing
  ------------------------------------------------------------------------
  -- Data slave channel alias signals
  s_axis_data_tdata_data        <= s_axis_data_tdata(3 downto 0);
  -- Data master channel alias signals: update these only when they are valid
  m_axis_data_tdata_data        <= m_axis_data_tdata(7 downto 0) when m_axis_data_tvalid = '1';
end tb;
```

Пример 1. Тестбенч КИХ-фильтра на четыре отвода сгенерированный в автоматическом режиме для моделирования импульсной характеристики

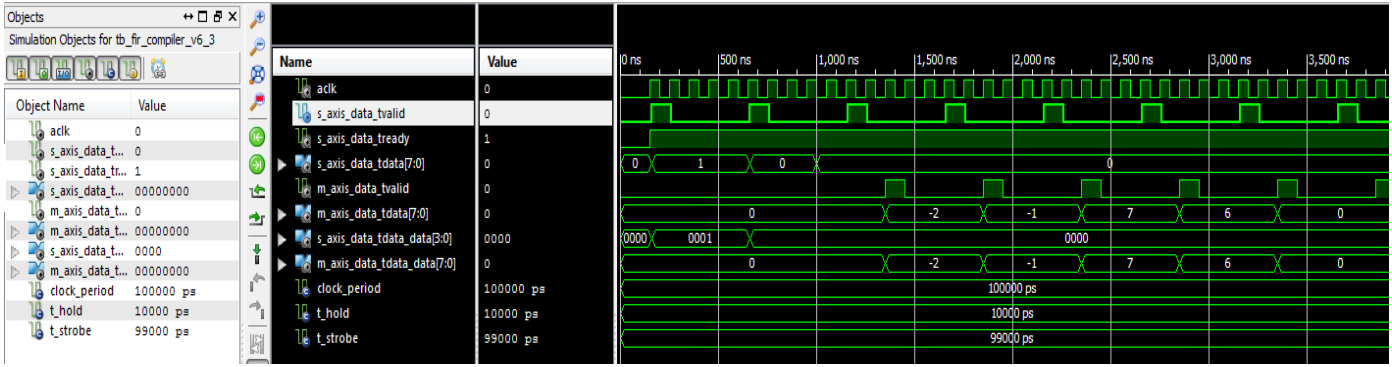


Рис. 4.41. Симулятор ISim САПР ПЛИС Xilinx ISE 14.2. Импульсная характеристика КИХ-фильтра на четыре отвода полученная с использованием тестбенча сгенерированного в автоматическом режиме

4.5. Пример проектирования КИХ-фильтров в базисе ПЛИС с применением генератора параметризированных ядер XLogiCORE IP и функции FIR Compiler v5.0

В данном разделе предлагается рассмотреть вопрос проектирования КИХ-фильтров на распределенной арифметике с использованием генератора параметризированных ядер XLogiCORE IP FIR Compiler Compiler v5.0. Выигрыш от использования распределенной арифметики заключается в том, что с ростом числа отводов производительность КИХ-фильтра остается постоянной за счет применения “безумножительных” схем умножения, при этом обеспечивается повышенное быстродействие, экономия по использованию встроенных ЦОС-блоков, а недостатком – повышенный расход логических ресурсов ПЛИС.

Генератор параметризированных ядер XLogiCORE IP FIR Compiler v5.0 предлагает на выбор три структуры фильтра: прямую форму систолического фильтра, в котором операции умножения и сложения выполняются параллельно с конвейеризацией; обратную и на распределенной арифметике. Функция FIR Compiler v5.0 не поддерживает современный протокол AXI4-Stream.

На рис. 4.42, *а* показана структурная схема КИХ-фильтра на четыре отвода с использованием последовательной распределенной арифметики (см. раздел 3.1). Применение которой например, для последовательного КИХ-фильтра позволяет отказаться от использования четырех аппаратных умножителей на константу и сократить дерево сумматоров (рис. 4.42, *б*) заменив их единственной таблицей перекодировок. На практике, для реализации адресуемых массивов комбинаций весовых коэффициентов фильтра используют таблицы перекодировок (LUT) ПЛИС.

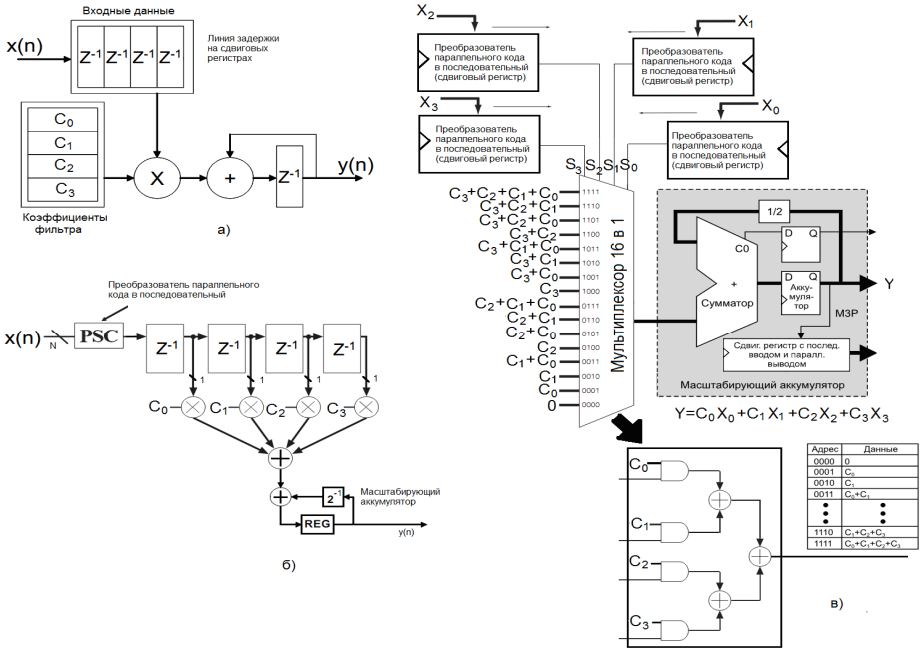


Рис. 4.42. Структуры КИХ-фильтров: а) – один MAC-фильтр; б) – последовательный КИХ-фильтр на четыре отвода; в) – упрощенное представление структуры КИХ-фильтра на четыре отвода с использованием последовательной распределенной арифметики

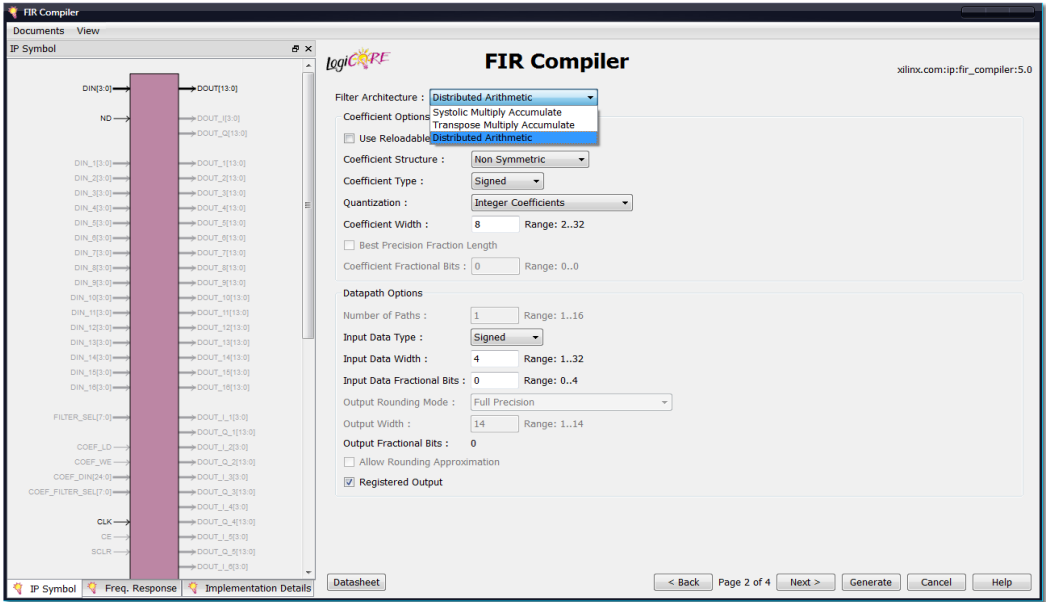


Рис. 4.43. Опции генератора параметризированных ядер XLogiCORE IP FIR Compiler Compiler v5.0. Выбирается структура фильтра на распределенной арифметике. Коэффициенты фильтра несимметричные, со знаком, целые, представляются с 8-битной точностью. Входные значения – целые со знаком, представляются с 4-битной точностью. Выходные значения представляются с 14-битной точностью

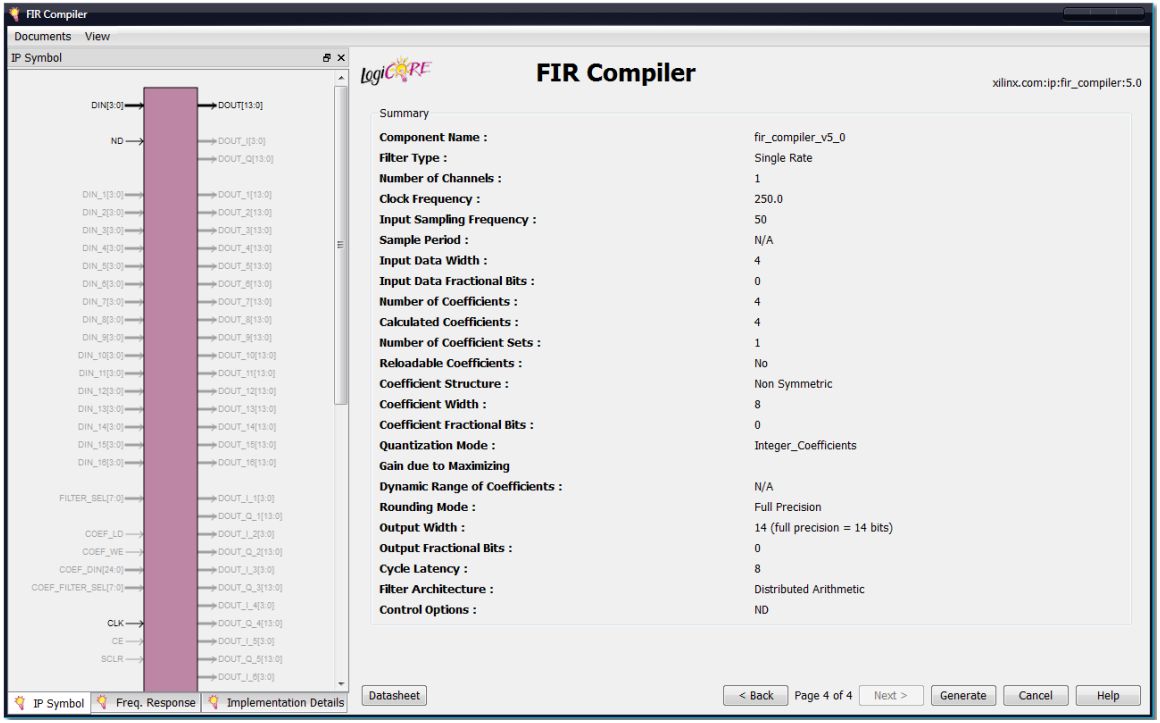


Рис. 4.44. Отчет о характеристиках КИХ-фильтра на четыре отвода

Выберем тип фильтра Single-Rate FIR и формат представления чисел с фиксированной запятой. В случае реализации КИХ-фильтра на распределенной арифметике генератор автоматически определяет тип арифметики: параллельная или последовательная. Степень параллелизма зависит от соотношения частоты взятия входных отчетов (fs) и частоты тактирования системы (fclk). Чем выше fs по отношению к fclk, тем меньше латентность фильтра L (задержка появления профильтрованного сигнала, измеряется в тактах синхроимпульса).

Частота взятия входных отчетов fs=50 МГц, частота тактирования системы fclk=250 МГц. Коэффициенты фильтра такие же, разрядность представления коэффициентов – восемь бит. Предполагаем, что на вход фильтра поступают только целые значения, как со знаком, так и без, например -5, 3, 1, 0. Разрядность представления значений входного сигнала подлежащего фильтрации (B) – четыре бита, профильтрованного сигнала – четырнадцать бит (рис. 4.43). Под дробную часть числа в обоих случаях отводим 0 бит (Input Data Fractional Bits и Output Fractional Bits).

Для размещения проекта в базис ПЛИС XC6SLX4 требуется 57 триггеров тактируемых фронтом синхросигнала из общих логических ресурсов ПЛИС LUT – 41, из них 30 используются как логические ресурсы, 8 LUT для реализации сдвигового регистра при этом максимальная частота составила 439 МГц.

На рис. 4.44 показан отчет о характеристиках спроектированного КИХ-фильтра на 4 отвода. Частота тактирования ядра фильтра установлена в 250 МГц, а входная частота дискретизации – 50 МГц. Разрядность входной шины данных – четыре бита. Структура коэффициентов фильтра – не симметричная, разрядность представления коэффициентов – восемь бит. Задана полная точность вычисления выходных

значений профильтрованного сигнала. Латентность фильтра восемь тактов синхрочастоты.

На рис. 4.45 представлен проект КИХ-фильтра на четыре отвода в САПР ПЛИС Xilinx ISE 14.2 с использованием генератора параметризированных ядер XLogiCORE IP FIR Compiler Compiler v5.0. Испытательный стенд для моделирования прохождения сигнала по структуре КИХ-фильтра на четыре отвода показывает пример 2. Моделирование прохождения сигнала по структуре КИХ-фильтра демонстрирует рис. 4.46. На вход фильтра подаются значения -5, 3, 1.

Разрешение на прием фильтром новых значений определяется высоким уровнем сигнала nd. Готовность фильтром прочитать новую порцию информации определяется выходными стробирующими импульсами сигнала rfd. Результат фильтрации определяется выходными стробирующими импульсами сигнала rdy. Смена профильтрованных значений на выходе фильтра осуществляется через четыре такта синхрочастоты (рис. 4.46). Систолический КИХ-фильтр на 4 отвода имеет латентность (время появления отклика фильтра) 11, а фильтр на последовательной распределенной арифметике без использования встроенных умножителей в ПЛИС - 8 тактов синхрочастоты.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;
ENTITY fir4_test_5 IS
END fir4_test_5;
```

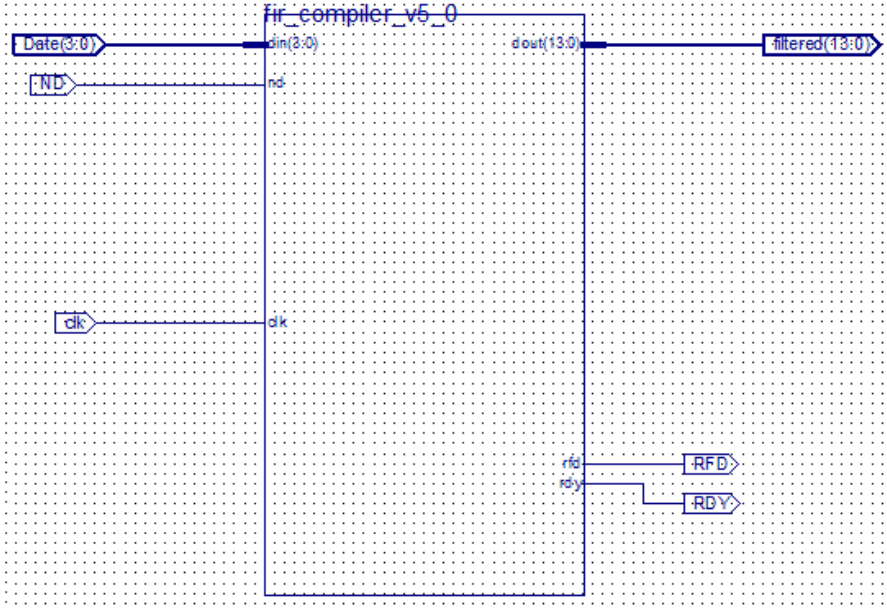


Рис. 4.45. Проект КИХ-фильтра на четыре отвода в САПР ПЛИС Xilinx ISE 14.2 с использованием генератора параметризированных ядер XLogiCORE IP FIR Compiler Compiler v5.0

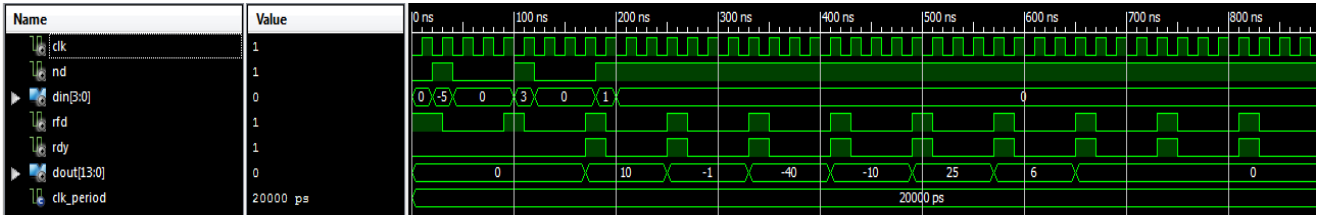


Рис. 4.46. Моделирование прохождения сигнала по структуре КИХ-фильтра на последовательной распределенной арифметике. На вход фильтра подаются значения -5, 3, 1. Правильные значения на выходе 10, -1, -40, -10, 25, 6. Латентность фильтра 8 тактов синхрочастоты

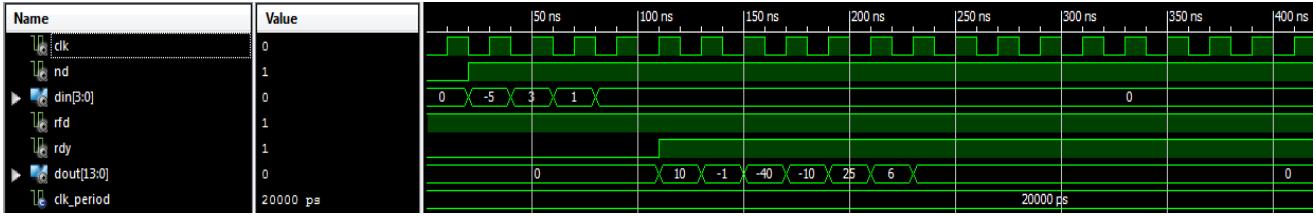


Рис. 4.47. Моделирование прохождения сигнала по структуре КИХ-фильтра на параллельной распределенной арифметике. На вход фильтра подаются значения -5, 3, 1. Правильные значения на выходе 10, -1, -40, -10, 25, 6. Латентность фильтра 5 тактов синхрочастоты

```
ARCHITECTURE behavior OF fir4_test_5 IS
   -- Component Declaration for the Unit Under Test (UUT)
   COMPONENT fir_compiler_v5_0
   PORT(
        clk : IN  std_logic;
        nd : IN  std_logic;
        rfd : OUT  std_logic;
        rdy : OUT  std_logic;
        din : IN  std_logic_vector(3 downto 0);
        dout : OUT  std_logic_vector(13 downto 0)
       );
   END COMPONENT;
  --Inputs
  signal clk : std_logic := '0';
  signal nd : std_logic := '0';
  signal din : std_logic_vector(3 downto 0) := (others => '0');
         --Outputs
  signal rfd : std_logic;
  signal rdy : std_logic;
  signal dout : std_logic_vector(13 downto 0);
  -- Clock period definitions
  constant clk_period : time := 20 ns;
BEGIN
         -- Instantiate the Unit Under Test (UUT)
  uut: fir_compiler_v5_0 PORT MAP (
        clk => clk,
        nd => nd,
        rfd => rfd,
        rdy => rdy,
        din => din,
        dout => dout
       );
  -- Clock process definitions
```

```
   clk_process :process
   begin
                clk <= '0';
                wait for clk_period/2;
                clk <= '1';
                wait for clk_period/2;
   end process;
   -- Stimulus process
   stim_proc: process
   begin
     -- hold reset state for 100 ns.
     wait for 100 ns;
     wait for clk_period*10;
     -- insert stimulus here
     wait;
   end process;
tb : process
                begin
                        wait for 20 ns;
                        nd<= '1';
                        din <= "1011";
                        wait for 20 ns;
                        nd<= '0';
                        din <= "0000";
                        wait for 20 ns;
                        din <= "0000";
                        wait for 20 ns;
                        din <= "0000";
                        wait for 20 ns;
                        nd<= '1';
                        din <= "0011";
                        wait for 20 ns;
                        nd<= '0';
                        din <= "0000";
```

```
                wait for 20 ns;
                din <= "0000";
                wait for 20 ns;
                din <= "0000";
                wait for 20 ns;
                nd<= '1';
                din <= "0001";
                wait for 20 ns;
                din <= "0000";
                wait for 20 ns;
                din <= "0000";
                wait for 20 ns;
                din <= "0000";
                wait for 20 ns;
                nd<= '1';
                din <= "0000";
        wait;
END process;
END;
```

Пример 2. Испытательный стенд для моделирования прохождения сигнала по структуре КИХ-фильтра на четыре отвода

Рассмотрим случай с параллельной распределенной арифметикой. Частота взятия входных отчетов fs=250 МГц и частота тактирования системы fclk= 300 МГц. Функция FIR Compiler v5.0 исходя из соотношения частот автоматически определила латентность фильтра 5 тактов синхрочастоты. Для размещения проекта в базис ПЛИС XC6SLX4 требуется уже 111 триггеров тактируемых фронтом синхросигнала из общих логических ресурсов ПЛИС, LUT – 88, из них 74 используются как логические ресурсы, 1 LUT функционирует как блок памяти и 1 LUT как сдвиговый регистр, при этом максимальная частота составила 438 МГц. Рис. 4.47

показывает моделирование прохождения сигнала по структуре КИХ-фильтра. На вход фильтра подаются значения -5, 3, 1. Правильные значения на выходе 10, -1, -40, -10, 25, 6. Латентность фильтра составила 5 тактов синхрочастоты.

В табл. 4.2 приведен анализ задействованных ресурсов ПЛИС XC6SLX4 при реализации КИХ-фильтров на четыре отвода с использованием функции FIR Compiler v6.3 и FIR Compiler Compiler v5.0 САПР Xilinx ISE 14.2. Из анализа табл. 4.2 следует, что наиболее быстродействующими являются фильтры на распределенной арифметике.

Таблица 4.2

Анализ задействованных ресурсов ПЛИС XC6SLX4 при реализации КИХ-фильтров на 4 отвода с использованием функции FIR Compiler v6.3 и FIR Compiler v5.0 САПР Xilinx ISE 14.2

Ресурсы ПЛИС	Систоли-ческий	Распределенная арифметика	
		Последовательная	Параллельная
Триггеры логических блоков в секциях	48	57	111
Секций с LUT	33	41	88
LUT для выполнения комбинационных функций	22	30	74
LUT как блоки памяти	11	8	1
LUT как сдвиговые регистры	11	8	1
Кол-во ЦОС-блоков DSP48A1	1	-	-
Латентность фильтра	11	8	5
Рабочая частота, МГц	348	439	438

Рассмотрим размещение КИХ-фильтра в ресурсы ПЛИС Xilinx XC6SLX4 (рис. 4.48). Во всех случаях установлена целевая задача проектирования –

сбалансированная. Кристалл ПЛИС разбит на 8 тактовых регионов (доменов). КИХ-фильтр с использованием систолической структуры и ЦОС-блока занимает три клоковых региона, а фильтры на последовательной и параллельной арифметике по два. Последовательная, занимает меньшее число ресурсов ПЛИС, но они более широко разбросаны по кристаллу, что может увеличивать задержки в трассировочных ресурсах, а параллельная при более значительном потреблении ресурсов более локализована. Области локализации для систолического фильтра (рис. 4.48, *а*) и фильтров с использованием распределенной арифметики (рис. 4.48, *а* и рис. 4.48, *б*) показаны красными овалами.

В случае КИХ-фильтров на параллельной арифметике выходной сигнал (профильтрованные значения) формируется через каждый синхроимпульс, а для фильтра на последовательной арифметике с несимметричной - через B и через B+1 для фильтра с симметричной импульсной характеристикой, т.е. в нашем случае через 4 такта синхроимпульса.

Отказ от использования в функции FIR Compiler v6.3 структур фильтров на распределенной арифметике говорит о том, что в настоящее время идет ориентация на массовое использование ЦОС-блоков в ПЛИС, но в тоже время ведущие разработчики САПР Xilinx и Altera сохранили возможность использования распределенной арифметики из за ряда преимуществ. Например, структуры КИХ-фильтров на основе распределенной арифметике обладают рекордным быстродействием, которое не снижается с ростом числа отводов.

Такие решения особенно эффективны в низкобюджетных сериях ПЛИС, где существует недостаточное число встроенных аппаратных ЦОС-блоков. Фильтрам на распределенной арифметике присущи такие недостатки как меньшая точность представления коэффициентов и входных

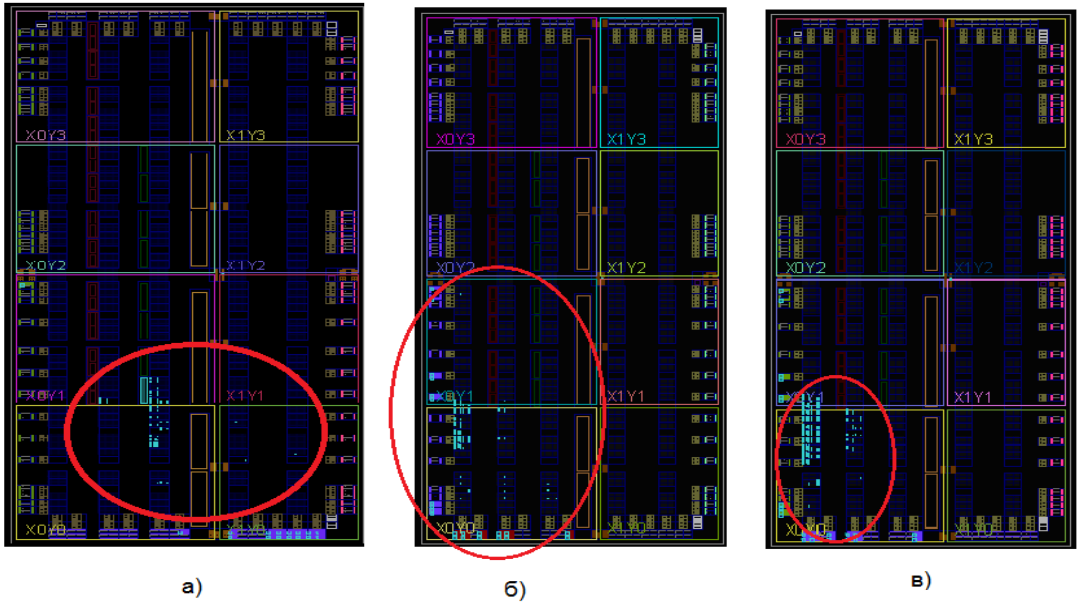


Рис. 4.48. Размещение КИХ-фильтра на четыре отвода в ресурсы ПЛИС XC6SLX4: а) систолическая структура с использованием ЦОС-блока; б) последовательная арифметика; в) параллельная арифметика

отсчетов, например, КИХ-фильтры на систолических структурах для ПЛИС серий Vitrex-5/6 позволяют иметь максимальную точность представления коэффициентов 49 бит против 32 бит и их нельзя перегрузить в режиме онлайн. Распределенная арифметика не позволяет так же реализовать полифазный банк фильтров и параллельную потоковую обработку информации.

4.6. Проектирование КИХ-фильтров в системе Xilinx System Generator САПР ISE Design Suite

System Generator IDS 14.4 — инструмент для разработки и отладки высокопроизводительных систем цифровой обработки сигналов в базисе ПЛИС фирмы Xilinx в системе визуально-имитационного моделирования Matlab/Simulink (версия 8.0.0.783 (R2012b)). Программный пакет обеспечивает высокоуровневое представление проекта, абстрагированное от конкретной аппаратной платформы, которое автоматически компилируется в ПЛИС Xilinx. System Generator является частью технологии XtremeDSP фирмы Xilinx.

System Generator сокращает время симуляции проектов за счет hardware-in-the-loop и HDL co-simulation. System Generator автоматически транслирует ЦОС-системы из Matlab/Simulink описаний в высокооптимизированные VHDL-описания для ПЛИС Xilinx и создает испытательные стенды. Методология "Hardware-in-the-loop" существенно ускоряет цикл проектирования, поскольку позволяет верифицировать проекты в ПЛИС непосредственно из системы Matlab/Simulink. "HDL co-simulation" позволяет пользователям импортировать HDL-код и симулировать всю систему в целом.

Рассмотрим разработку имитационной модели КИХ-фильтра на распределенной арифметике без использования встроенных ЦОС-блоков (тип фильтра - Single-Rate FIR) с применением функционального блока FIR Compiler v5.0 являющимся аналогом функции FIR Compiler v5.0 САПР Xilinx ISE получаемой с помощью генератора параметризированных ядер XLogiCORE IP (рис. 4.49).

Для верификации функционального блока FIR Compiler v5.0 используются фильтры DF2T (транспонированная реализация дискретного фильтра) и Digital Filter (прямая форма). Блок FIR Compiler v5.0 является параметризированным (рис. 4.50). На рис. 4.50 показаны настройки блока. Согласно настройкам блока реализуется КИХ-фильтр на параллельной распределенной арифметике. На рис.4.51 показано имитационное моделирование.

Для представления чисел со знаком в формате с фиксированной запятой Xilinx System Generator использует нотацию FIX, а для без знаковых - UFIX. Формат FIX можно рассматривать как пару чисел M.N, где M - общее число двоичных разрядов; N – число разрядов дробной части. Входной сигнал представляется в формате FIX_16_8, коэффициенты фильтра в формате FIX_8_4, профильтрованный сигнал FIX_26_12 (рис. 4.51). С помощью блока System Generator создадим в автоматическом режиме проект фильтра и испытательный стенд. Проект разместим в базис ПЛИС серии Spartan-6 xc6slx4-3tqg144.

На рис. 4.52 и рис. 4.53 показано функциональное моделирование с использованием моделирующей программы сгенерированной в автоматическом режиме для случая, когда используется функциональный блок FIR Compiler v5.0. Входной сигнал, подлежащий фильтрации умножается на масштабный множитель 256, а коэффициенты фильтра масштабируются на 16 согласно выбранному формату представления чисел.

Для получения правильного результата фильтрации необходимо предусмотреть деление на 4096. Уравнение фильтрации будет выглядеть следующим образом:

$$y/4096 = (C_0 * 16)(x_0 * 256) + (C_1 * 16)(x_1 * 256) + \\ + (C_2 * 16)(x_2 * 256) + (C_3 * 16)(x_3 * 256)$$

На рис. 4.54 показана имитационная модель КИХ-фильтра на 4 отвода с перегружаемыми коэффициентами с использованием блока FIR Compiler v5.0, а на рис. 4.55 результаты имитационного моделирования.

Рассмотрим разработку имитационной модели систолического КИХ-фильтра (тип фильтра - Single-Rate FIR) с использованием блока FIR Compiler v6.3 являющимся аналогом функции FIR Compiler v6.3 САПР Xilinx ISE получаемой с помощью генератора параметризированных ядер XLogiCORE IP (рис. 4.56). На рис. 4.57 показан формат представления входного сигнала и закладка “реализация” блока FIR Compiler 6.3. Входной сигнал представляется в формате FIX_16_8 а коэффициенты фильтра в формате FIX_4_0. Профильтрованный сигнал представляется с 20-битной точностью. На рис. 4.58 показано имитационное моделирование.

С помощью блока System Generator создадим в автоматическом режиме проект фильтра и испытательный стенд. На рис. 4.59 показано функциональное моделирование с использованием моделирующей программы сгенерированной в автоматическом режиме (период синхросигнала 100 нс).

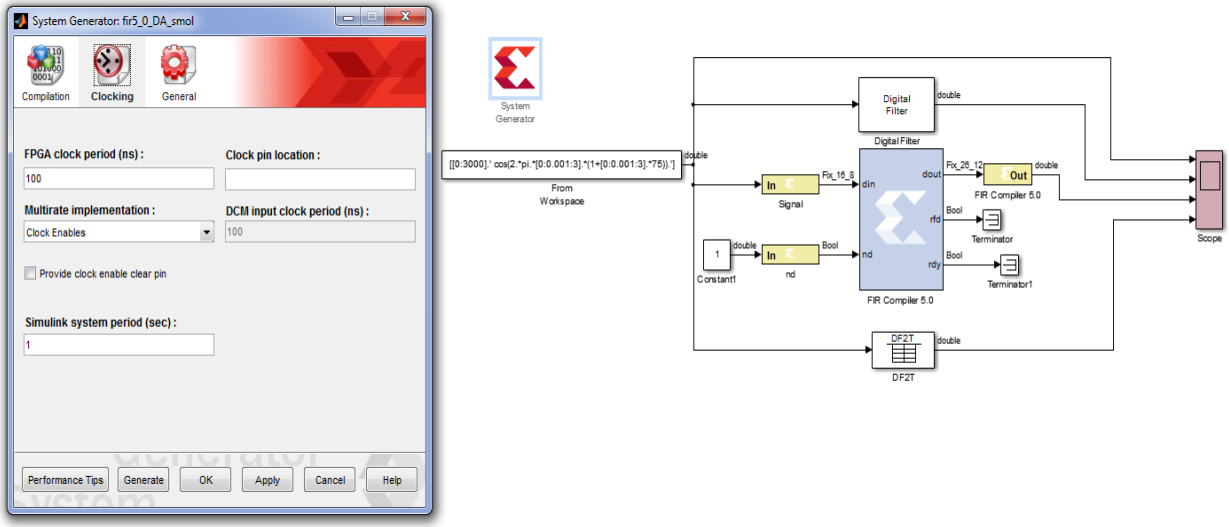
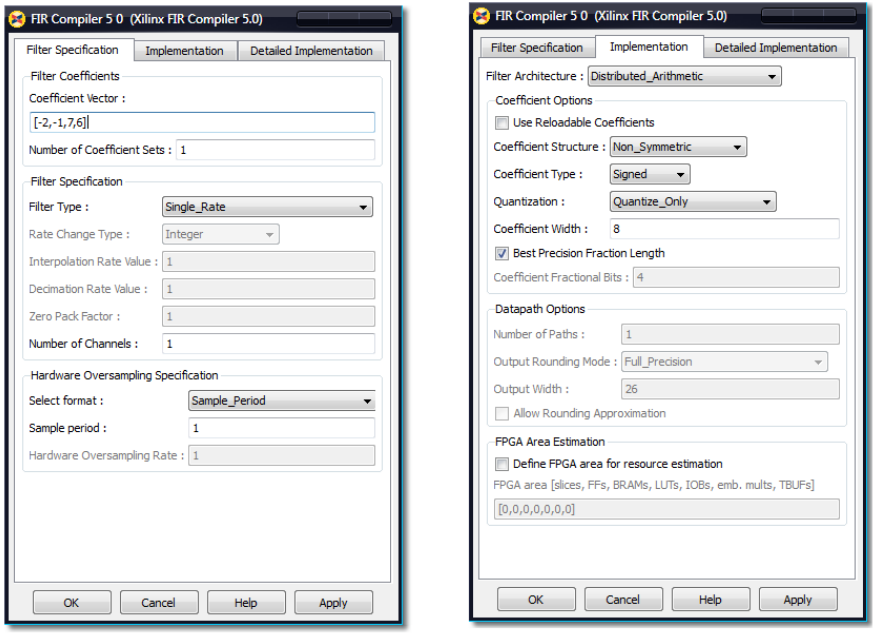


Рис. 4.49. (а) Задание частоты тактирования системы (фильтра) в САПР ISE (100 ns) и периода симуляции в Simulink (1 с); б) - модель КИХ-фильтра на четыре отвода с использованием блока FIR Compiler v5.0



а) б)

Рис. 4.50. Настройки блока FIR Compiler 5.0: а) – закладка спецификация фильтра; б) – закладка реализация фильтра. Коэффициенты фильтра несимметричные, со знаком, квантованные, представлены в формате FIX_8_4

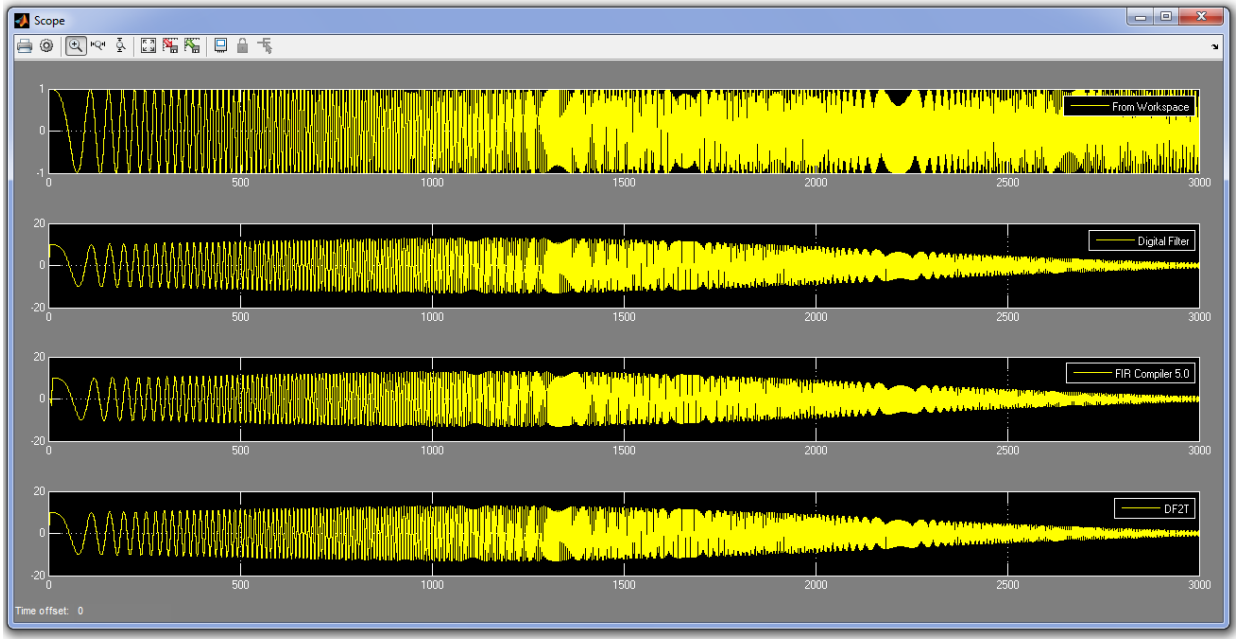


Рис. 4.51. Имитационное моделирование в системе Matlab/Simulink КИХ-фильтра на четыре отвода созданного с помощью блоков Digital Filter, FIR Compiler 5.0 и DF2T

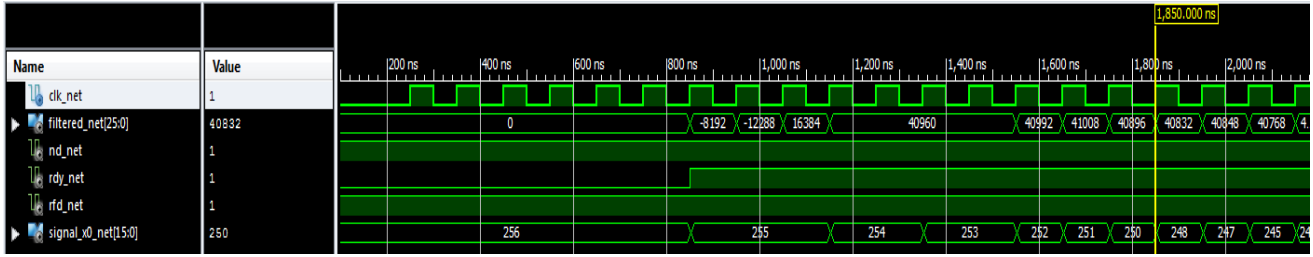


Рис. 4.52. Входной сигнал умножается на масштабный множитель 256, коэффициенты фильтра на 16. Период синхросигнала 100 нс

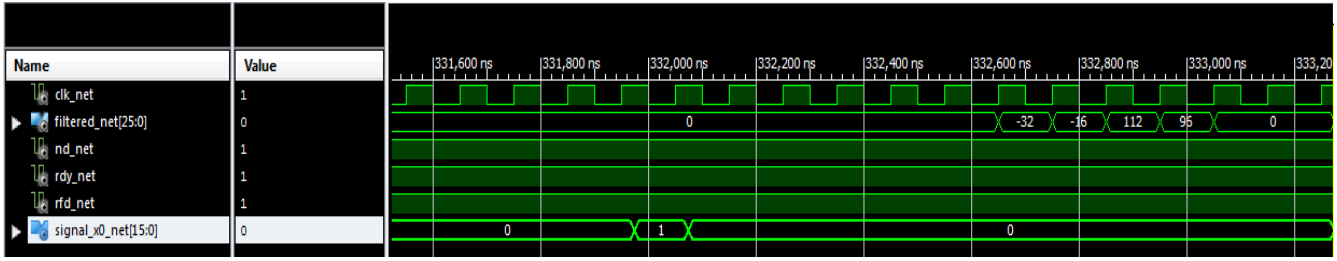


Рис. 4.53. Импульсная характеристика фильтра. Коэффициенты умножаются на 16

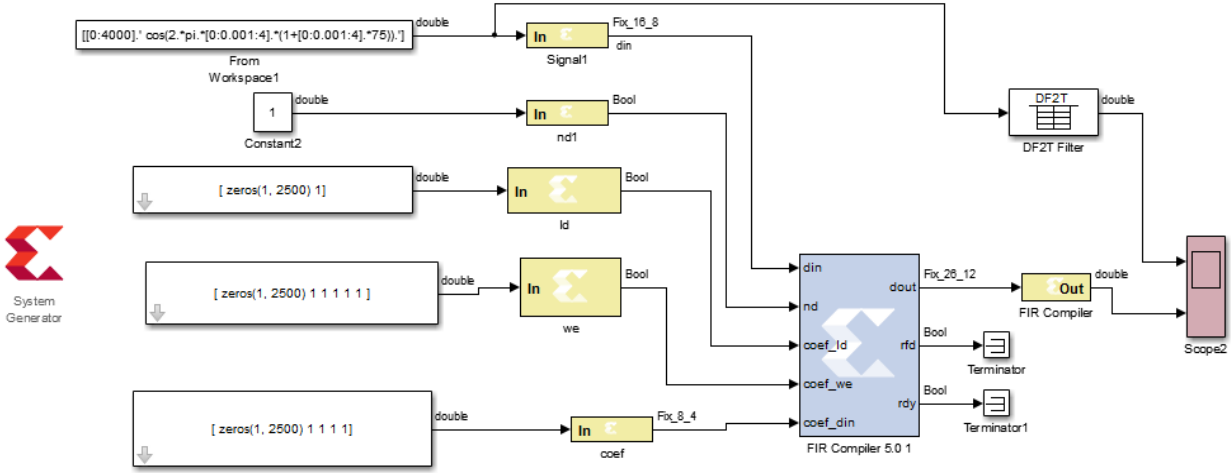


Рис. 4.54. Модель КИХ-фильтра на 4 отвода с перегружаемыми коэффициентами с использованием блока FIR Compiler v5.0

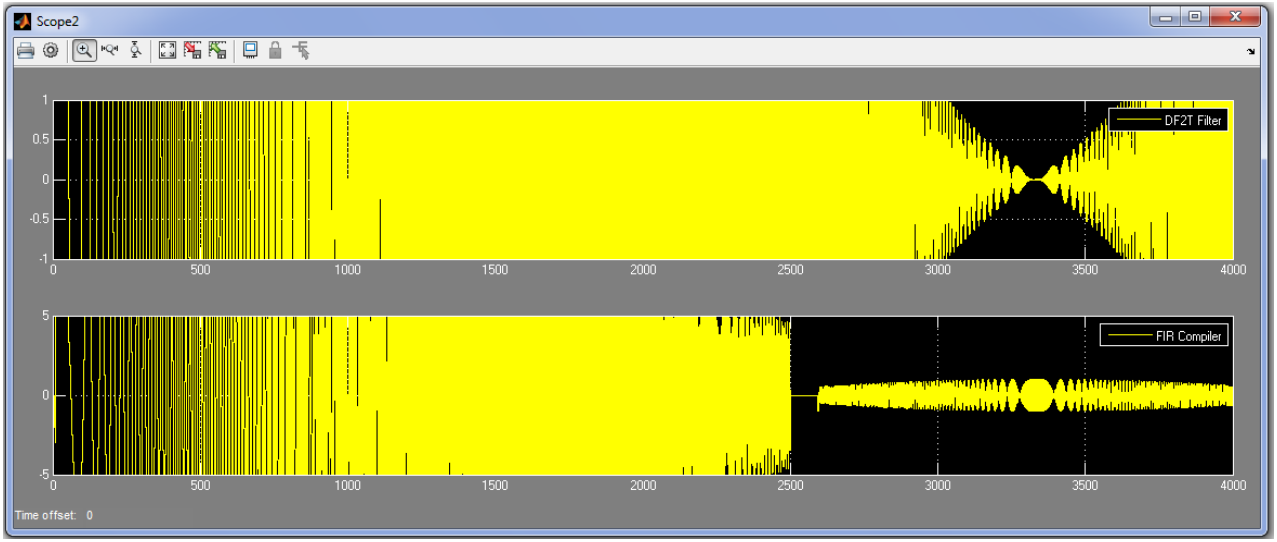


Рис. 4.55. При достижении 2500 отсчета происходит загрузка вектора значений коэффициентов [1 1 1 1] вместо [-2 -1 7 6]

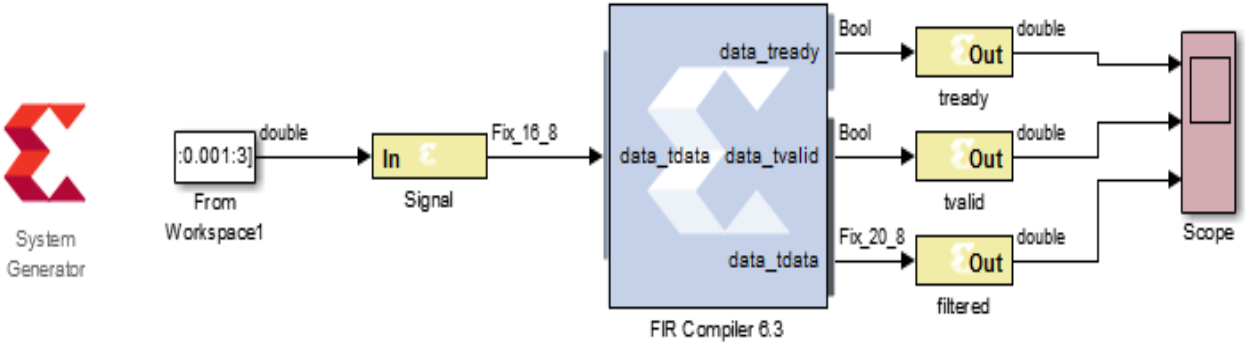
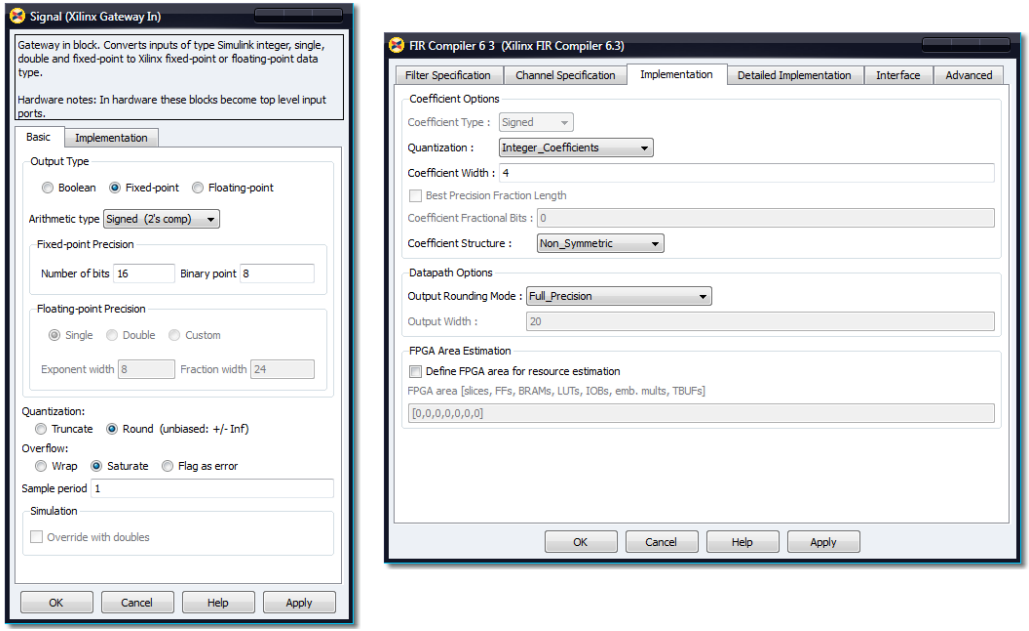


Рис. 4.56. Систолический КИХ-фильтр на четыре отвода с использованием блока FIR Compiler v6.3



а) б)

Рис. 4.57. Настройки блоков: а) – входной сигнал представляется в формате FIX_16_8; б) – закладка реализация, коэффициенты фильтра целые, со знаком, представлены в формате FIX_4_0

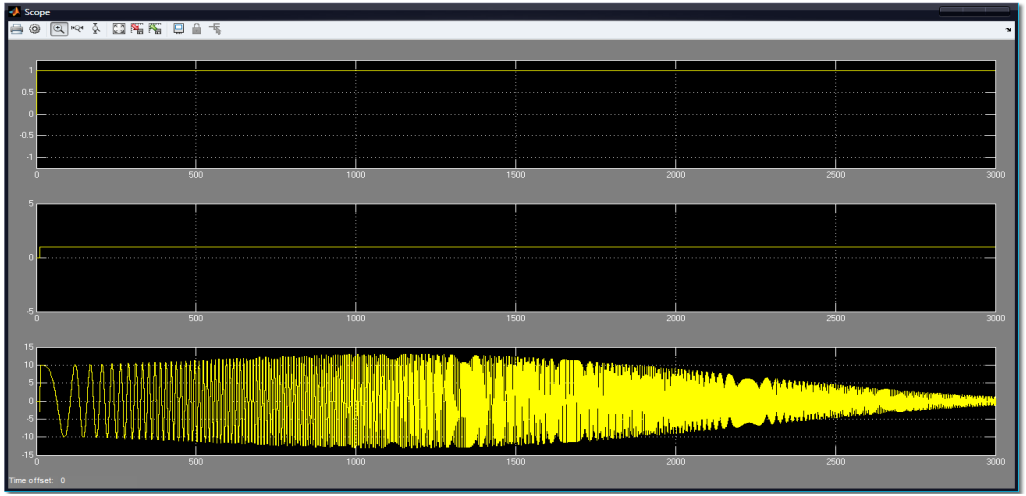


Рис. 4.58. Имитационное моделирование фильтра на четыре отвода созданного с помощью блока FIR Compiler 6.3

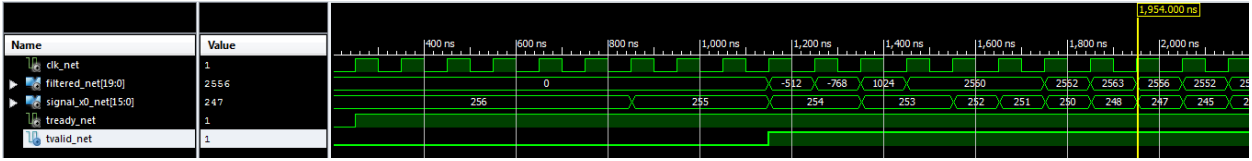


Рис. 4.59. Функциональное моделирование с использованием моделирующей программы на языке VHDL сгенерированной в автоматическом режиме. Входной сигнал умножается на 256, а коэффициенты фильтра не масштабируются

Входной сигнал, подлежащий фильтрации умножается на масштабный множитель 256, а коэффициенты фильтра не масштабируются согласно выбранному формату представления чисел. Для получения правильного результата фильтрации необходимо предусмотреть деление на 256. Уравнение фильтрации будет выглядеть следующим образом:

$$y/256 = C_0(x_0 * 256) + C_1(x_1 * 256) + C_2(x_2 * 256) + C_3(x_3 * 256).$$

4.7. Проектирование КИХ-фильтров со структурой MAC-блоков в системе Xilinx System Generator САПР ISE Design Suite

Рассмотрим проектирование КИХ-фильтра (рис. 4.59) с использованием MAC-блока с большим числом отводов для подавления высокочастотных шумов в аудиосистеме (CD-качество).Такие фильтры получили название MAC-фильтр. Данный пример основан на функциональном блоке n-tap Dual Port Memory MAC FIR Filter из библиотеки Xilinx Reference Blockset /DSP.

Для хранения коэффициентов фильтра и входных отсчетов будем использовать блочную память ПЛИС Xilinx (двух портовое ОЗУ). Параметры спецификации фильтра следующие: единицы измерения Гц; частота взятия отчетов $F_S = 44100$ Гц (44.1 кГц); порядок фильтра – автоматический выбор (Minimum Order); граница полосы пропускания $f_{Pass} = 6000$ Гц (6 кГц); граница полосы задерживания (подавления) $f_{Stop} = 7725$ Гц (7.725 кГц); неравномерность АЧХ в полосе пропускания A_{Pass} ($R_p = 1$ дБл); минимальное затухание в полосе задерживания A_{Stop} ($R_S = 48$ дБл). Осуществим синтез фильтров с равномерными пульсациями

АЧХ в полосах пропускания и задерживания методом Ремеза (Equiripple). По заданной технической спецификации синтезируется фильтр с порядком $n = 42$ (импульсная характеристика содержит $n+1$ ненулевых отсчетов) или 43 коэффициента.

На рис. 4.61 показаны настройки функционального блока источника случайного сигнала. В поле Sample Time необходимо указать период дискретизации сигнала $1/F_S = 1/44100$.

Библиотека DSP Xilinx blockset содержит встроенную графическую среду для синтеза и анализа фильтров FDATool представленную в виде одноименного блока помеченного маркером X (рис. 4.62, *а*). Использование специальной функции xlfda_numerator('FDATool') позволяет импортировать рассчитанные коэффициенты фильтра (числитель передаточной функции) непосредственно в блок MAC Based FIR. Коэффициенты можно так же посмотреть в системе Matlab с помощью команды xlfda_numerator('FDATool').

Блок MAC Based FIR является параметризованным (шесть переменных). Двойным кликом по блоку можно задать значения параметров в определенных полях (рис. 4.62, *б*). Входные данные представляются в формате FIX_10_8, а коэффициенты как FIX_12_12. Обозначения переменных блока можно посмотреть в маске (правая кнопка мыши, Mask, Edit Mask) рис. 4.63.

Максимальное значение коэффициента составляет 0.3022 (определяется командой: max(xlfda_numerator('FDATool')) а минимальное -0.067. Поэтому коэффициенты фильтра целесообразно представить в формате Fix_12_12. Это обеспечивает приведение коэффициентов к диапазону [-0.5, 0.4998]. Для сравнения можно было бы использовать формат Fix_12_11 (перенесение десятичной точки влево на один разряд при сохранении длины слова), что привело бы к диапазону [-1, 0.9995] (рис. 4.64).

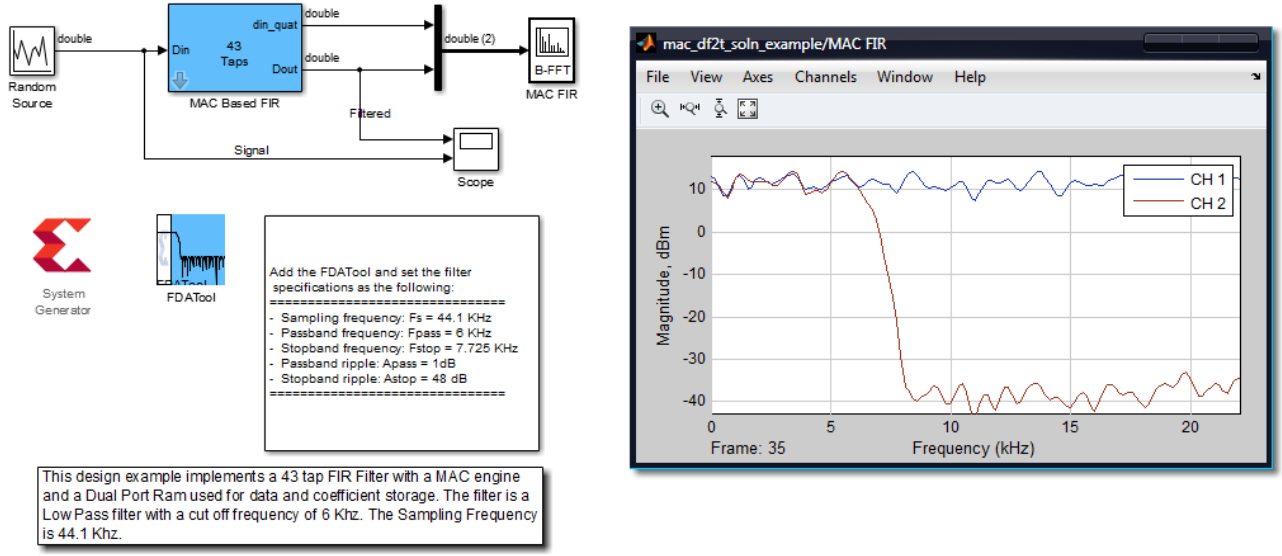


Рис. 4.60. Имитационная модель КИХ-фильтра на 43 отвода с возможностью вычисления коэффициентов по заданной спецификации с помощью графической среды FDATool и АЧХ. На вход фильтра подключен источник случайного сигнала

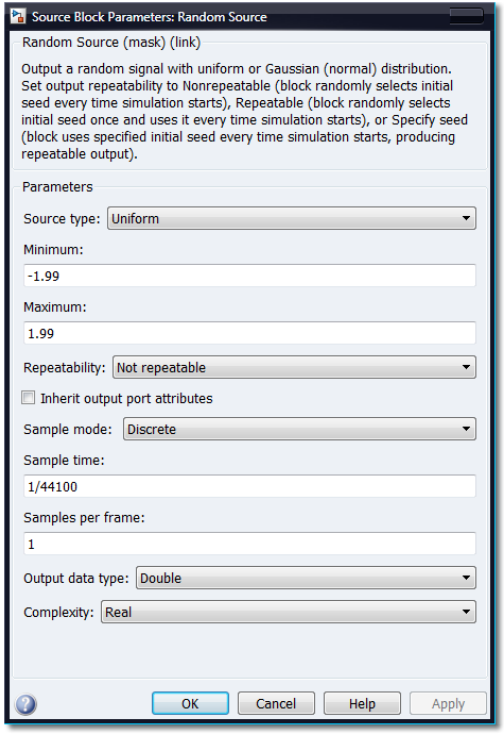
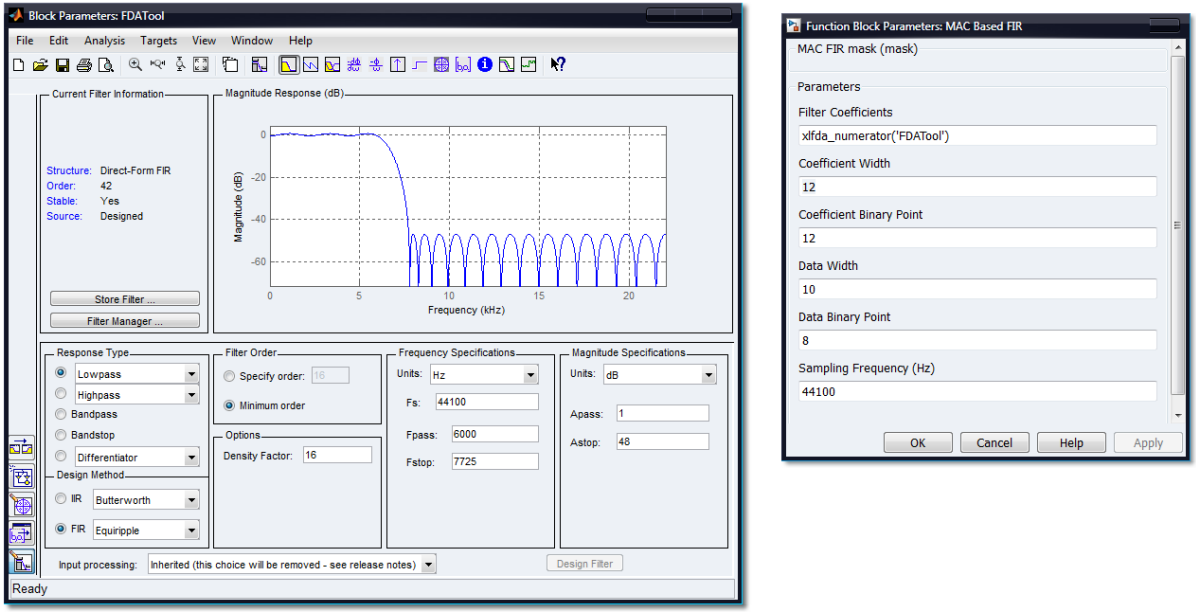


Рис. 4.61. Источник случайного сигнала с периодом дискретизации 1/44100



а) б)

Рис. 4.62. Блок FDATool (а) и маска с параметрами блока MAC Based FIR (б). Значения коэффициентов фильтра вычисляются с помощью команды xlfda_numerator('FDATool')

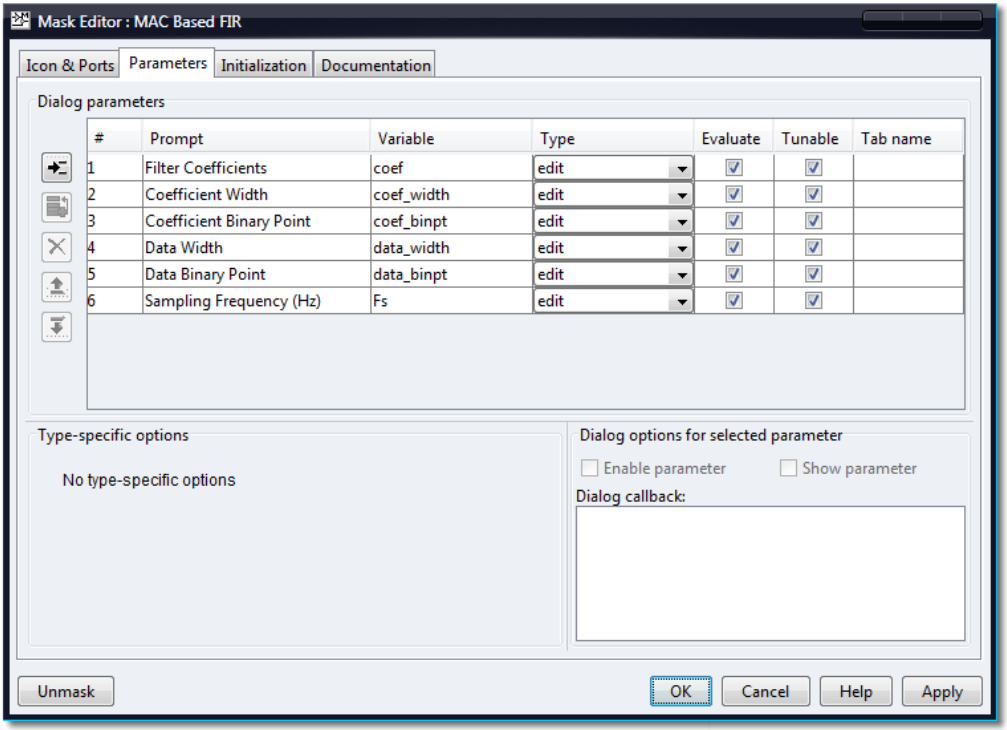


Рис. 4.63. Маска с параметрами функционального блока MAC Based FIR

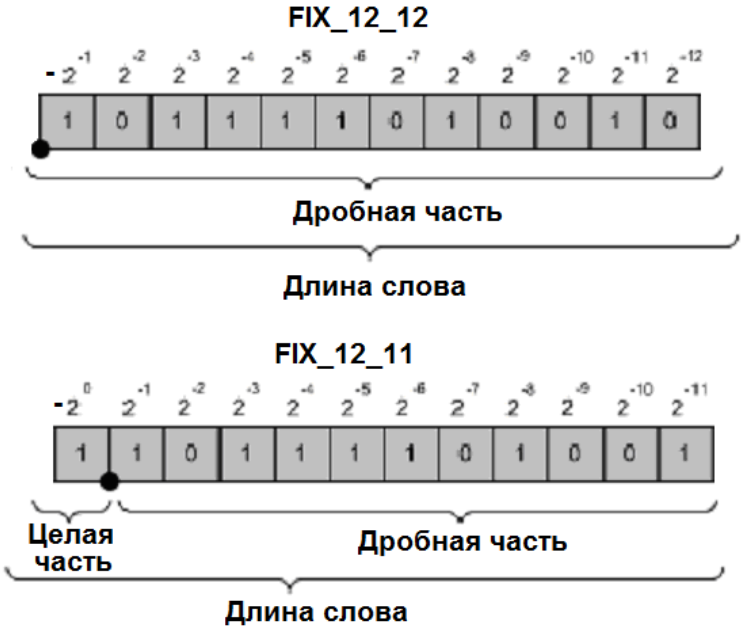


Рис. 4.64. Представление чисел в формате с фиксированной запятой в System Generator

На рис. 4.65, *а* показана структурная схема КИХ-фильтра на 43 отвода с использованием одного MAC-блока и блочной двух портовой памяти (ОЗУ). Входной сигнал перед записью в блочную память подвергается передискретизации на величину в 43 раза выше, чем частота взятия отсчетов F_S. Память разбита на два банка, которые используют различные режимы работы. Первый сконфигурирован как ОЗУ и работает в режиме циклического буффера, второй как ПЗУ.

Емкость первого банка памяти предназначенного для хранения отсчетов составляет 43 строки (адреса 0 - 42) на 10 столбцов (FIX_10_8) или 43 10-разрядных слов. Циклический буфер имитирует работу линии задержки фильтра (рис. 4.65, *а*). Второй банк предназначен для хранения коэффициентов фильтра емкостью 43 строки (адреса 43 - 85) на 12 столбцов (FIX_12_12) или 43 12-разрядных слов. На рис. 4.65, *б* показан пример реализации КИХ-фильтра на четыре отвода с использованием 1 MAC-блока.

На рис. 4.66 показана структурная схема MAC-фильтра с применением библиотек System Generator. Схема предназначена для размещения в логические ресурсы ПЛИС. Основные функциональные блоки: Memory (двух портовая память с управляющим автоматом), MAC engine (блок умножения с накоплением), Register (регистр захвата для операции конвейеризации), Down Sample (понижение частоты дискретизации в 43 раза), Convert 1 (представляет результат фильтрации с требуемой точностью или преобразует формат FIX_24_20 образующийся в процессе умножения и накопления в формат FIX_10_8). Блок din (Xilinx Gateway In, можно рассматривать как вход в ПЛИС) преобразует тип double в формат FIX_10_8. Блок MAC Out выполняет обратную функцию, преобразует формат FIX_10_8 в тип double. Его можно рассматривать как выход из ПЛИС.

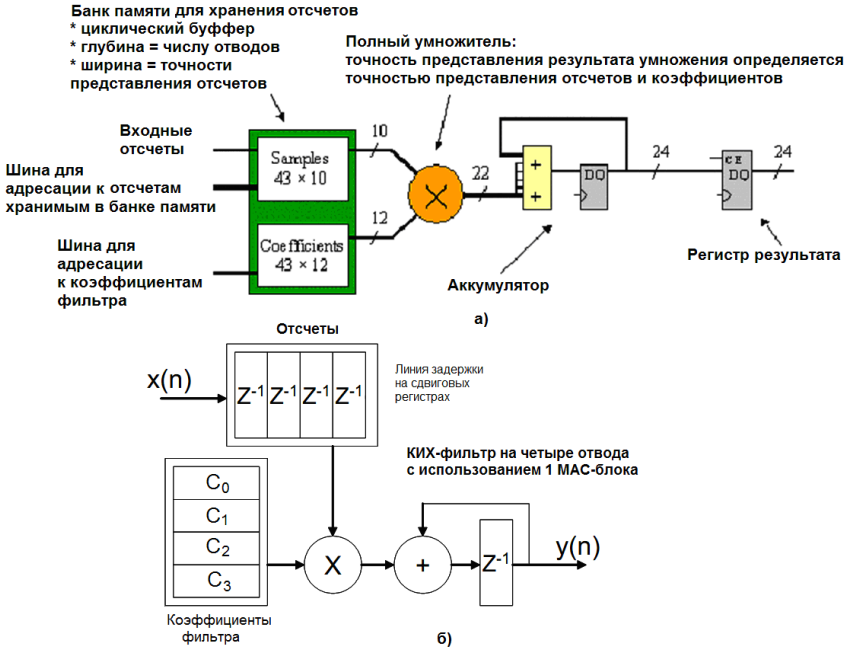


Рис. 4.65. Структурная схема КИХ-фильтра на 43 отвода с использованием одного MAC-блока и блочной памяти разбитой на два банка, один из которых используется в виде циклического буфера для хранения и считывания отсчетов, а другой – для хранения коэффициентов фильтра (а) и (б) КИХ-фильтр на четыре отвода с использованием одного MAC-блока

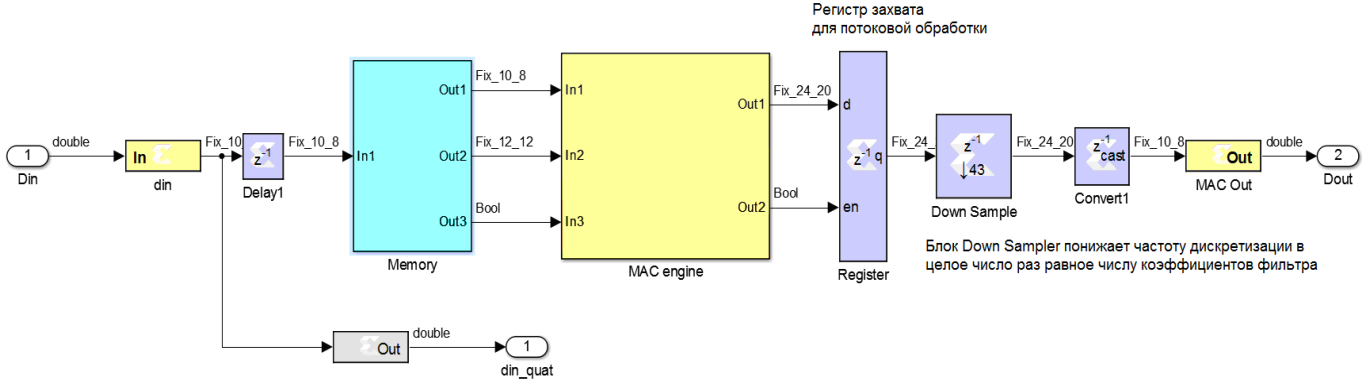


Рис. 4.66. Структурная схема MAC-фильтра с применением библиотек Xilinx System Generator

Блок Dual Port RAM представляет память ОЗУ с двумя портами A и B. Входные отсчеты записываются в и считываются из порта A (ОЗУ), коэффициенты считываются из порта B (рис. 4.67). Процессом записи и считывания управляет автомат (блок Address Control) (рис. 4.68).

Основные функциональные блоки автомата это два суммирующих счетчика coef_counter (предварительно загружается число 43) и Data_Counter (предварительно загружается 0) адресующихся к портам A и B ОЗУ и компаратор. На один из входов компаратора подключается константа 85 (команда 2*length(coef)-1). Компаратор управляет входом разрешения счета счетчика Data_Counter и формирует сигнал we (латентность 1) разрешающий запись в ОЗУ и сигнал сброса аккумулятора (латентность 5) с последующим формированием сигнала разрешения работы регистра захвата.

ОЗУ (рис. 4.69) инициализируется вектором значений с применением следующей команды [zeros(1, length(coef)) (coef)] (рис.4.69). Блок памяти имеет латентность - единица.

На рис. 4.69 показано, что сигнал, подлежащий фильтрации подвергается передискретизации в 43 раза перед тем как быть записанным в ОЗУ. Передискретизация и последующая операция понижения частоты дескритизации (децимация) обеспечивают по внешним входам фильтра организовывать структуру фильтра типа Single-Rate FIR.

Для выравнивания числа столбцов в банках памяти используется блок pad (основан на блоке Xilinx Bus Concatenator) выполняющего роль конкатенации (склеивания) двух шин. Склеиваются две шины в форматах UFix_10_0 (10-разрядная шина, младшие разряды lo) и UFix_2_0 (2-разрядная шина, старшие разряды hi). Результатом склеивания является шина в формате UFix_12_0 которая преобразуется в тип Fix_12_12.

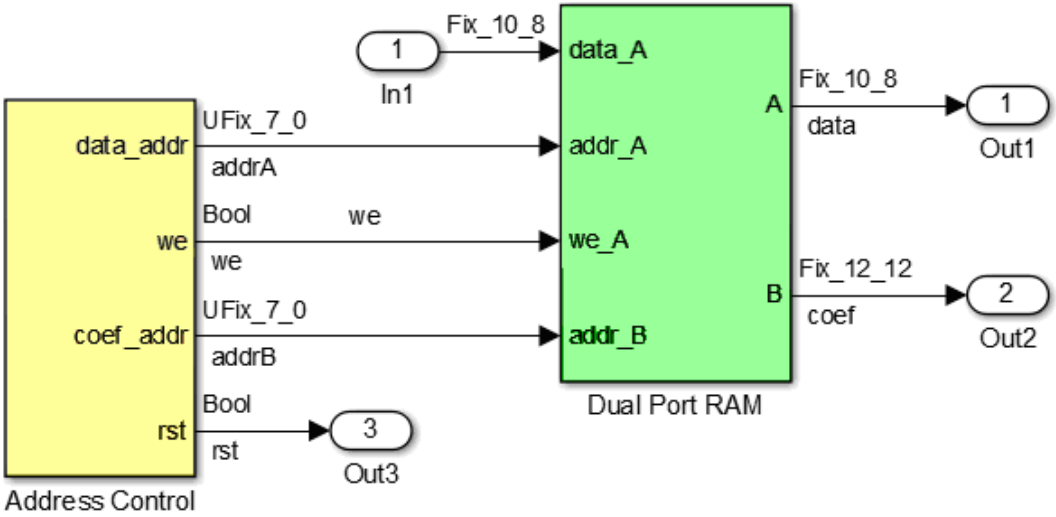


Рис. 4.67. Двух портовая память с управляющим автоматом

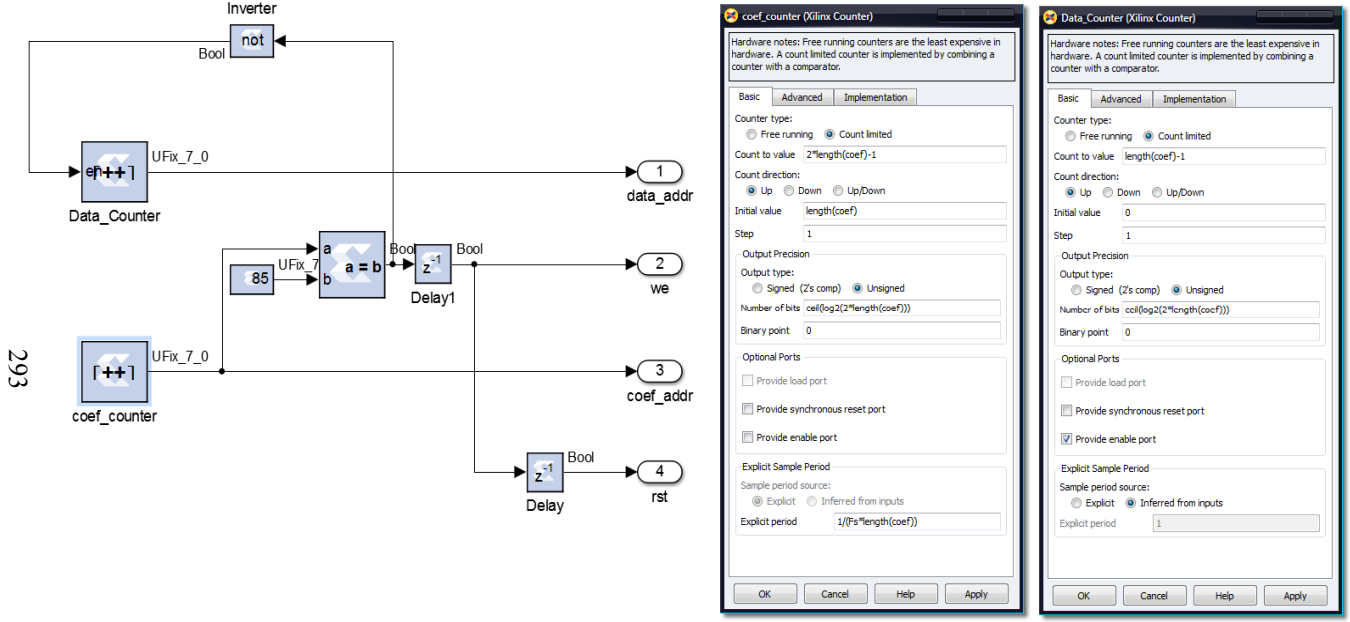


Рис. 4.68. Управляющий автомат двух портовой памяти и настройки блоков счетчиков coef_counter и Data_Counter

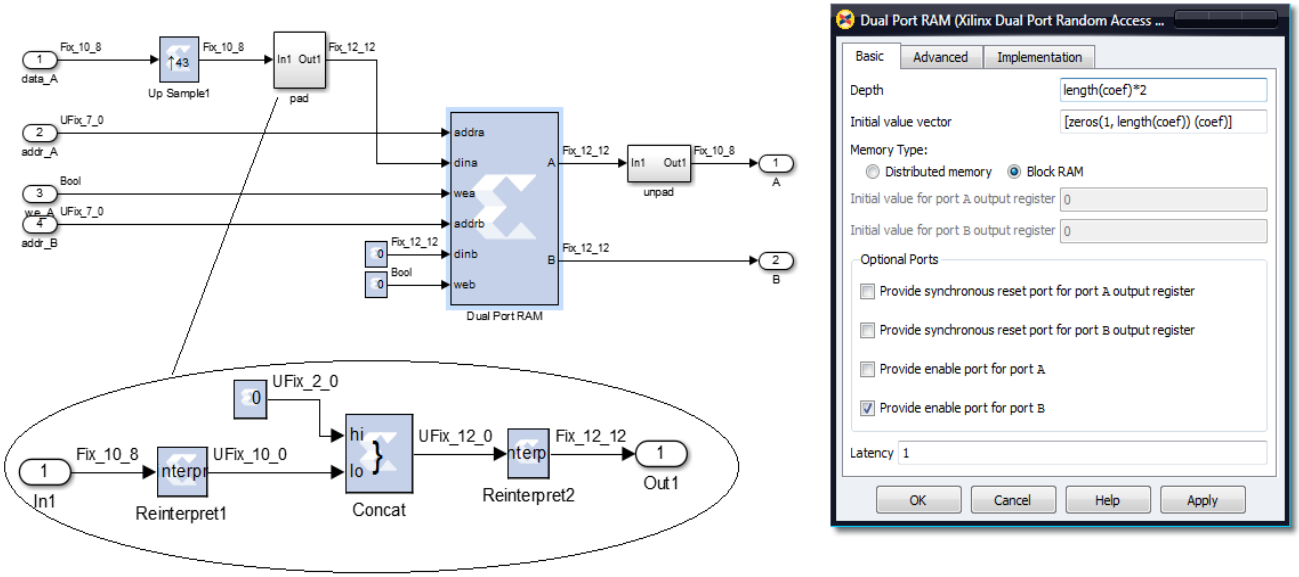


Рис. 4.69. Двух портовая память на основе блочной памяти ПЛИС и настройки блока Xilinx Dual Random Access

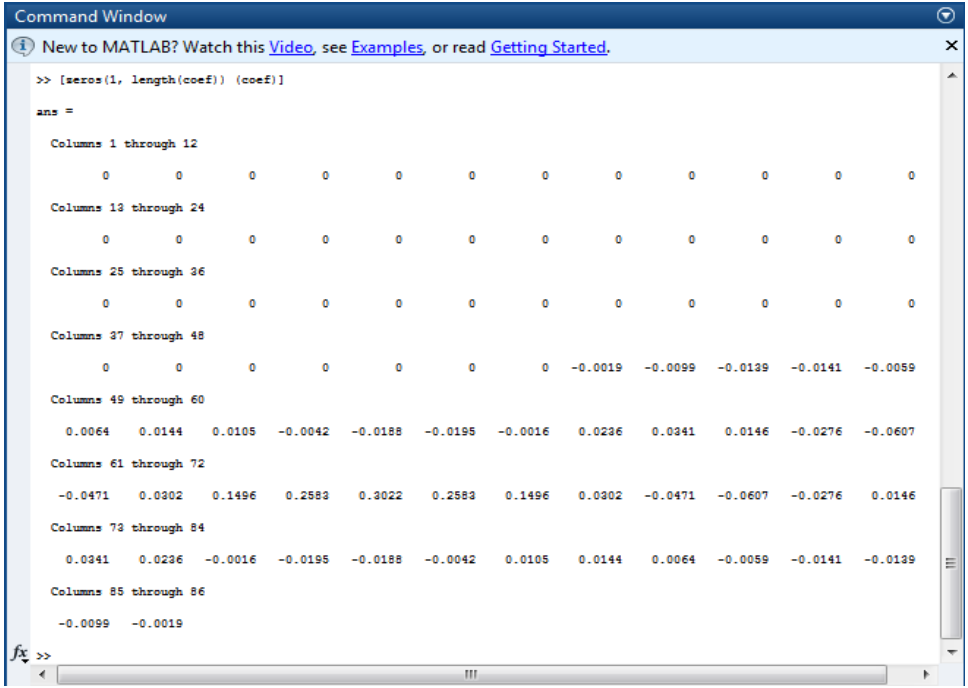


Рис. 4.70. Вектор инициализации блочной памяти ПЛИС (коэффициенты КИХ-фильтра симметричны)

На рис. 4.71 показан умножитель и аккумулятор. Результат умножения представляется в формате Fix_22_20, а результат сложения в формате Fix_24_20 с учетом переполнения и операции расширения знака числа. Разрядность выходной шины аккумулятора определяется следующей командой:

ceil(log2(max(1,sum(abs(coef*2^coef_binpt)))))+data_width+1.

Умножитель имеет латентность три для согласования с реальной латентностью равной трем характерной для встроенных умножителей в ПЛИС Xilinx.

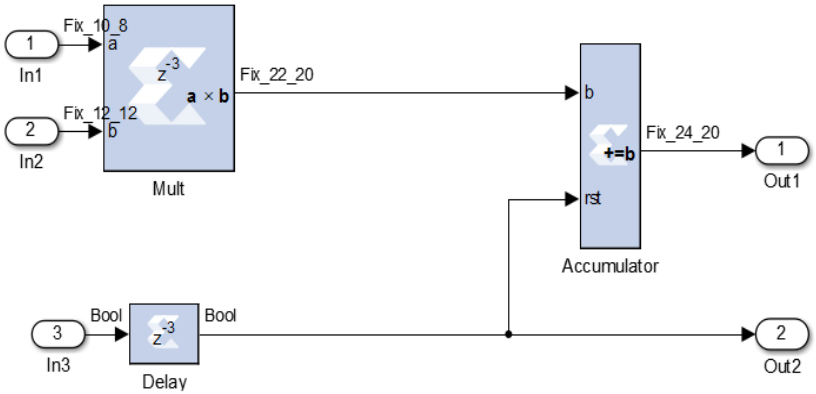


Рис. 4.71. Умножитель и аккумулятор

На рис. 4.72 показана настройка периода симуляции в Simulink. С учетом того что используется передискретизация период симуляции должен быть $1/(43F_S)$. Период синхросигнала задаем 10 нс. В меню Configuration Parameters задаем время симуляции 0.05 с. На рис. 4.73 показано удаление высокочастотных составляющих из случайного сигнала с помощью КИХ-фильтра на 43 отвода. Имитационная модель и моделирование отклика КИХ-

фильтра на единичный импульс в Simulink показано на рис. 4.74. Период синхросигнала увеличен до 100 нс. Функциональное моделирование показывает, что входной сигнал (единичный импульс по амплитуде с произвольной шириной, не путать с дельта-функцией позволяющей просмотреть коэффициенты фильтра) и выходной сигнал умножаются на 256 (рис. 4.75). Профильтрованные значения обновляются через 43 такта синхроимпульса. Фрагменты значений выходного сигнала и коэффициентов фильтра в форматах с плавающей double и фиксированной запятой FIX приведены в табл. 4.3.

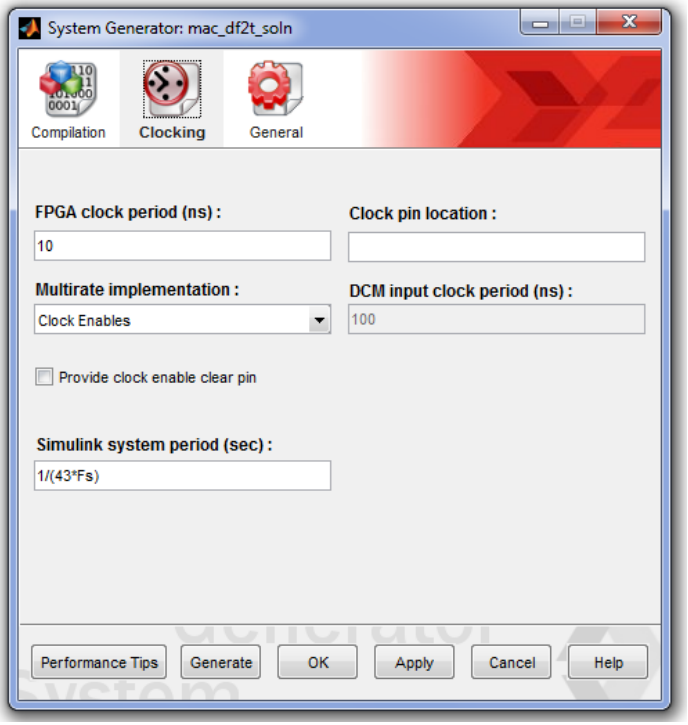


Рис. 4.72. Настройка периода симуляции в Simulink

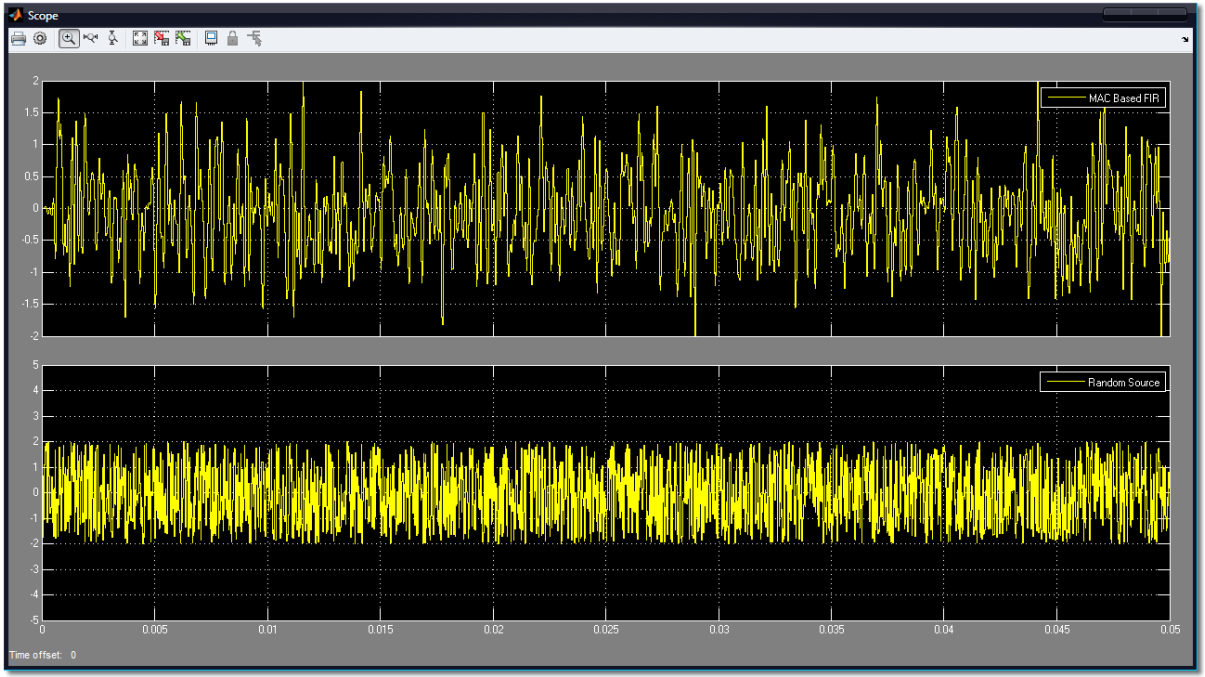
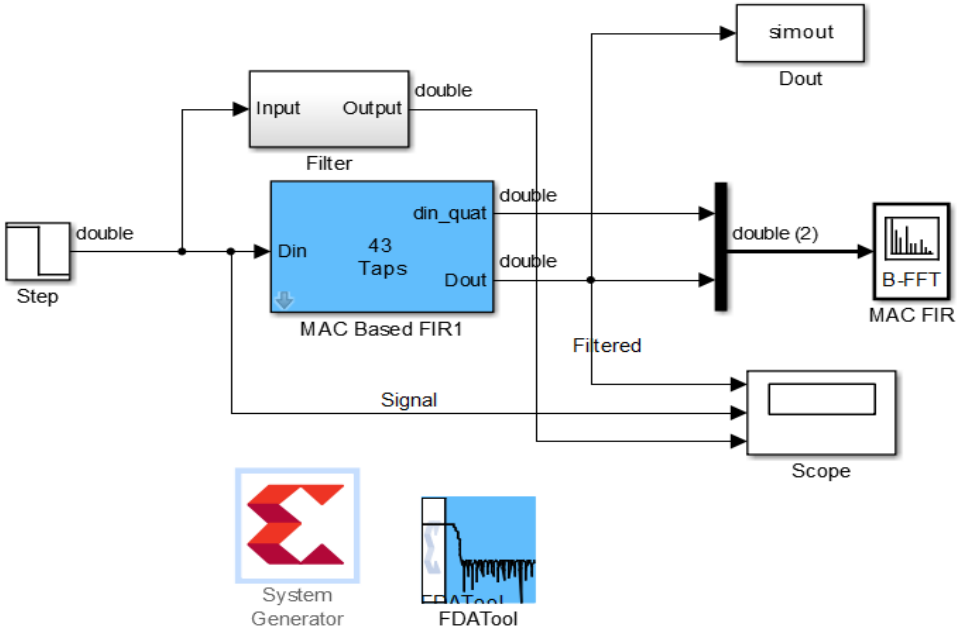
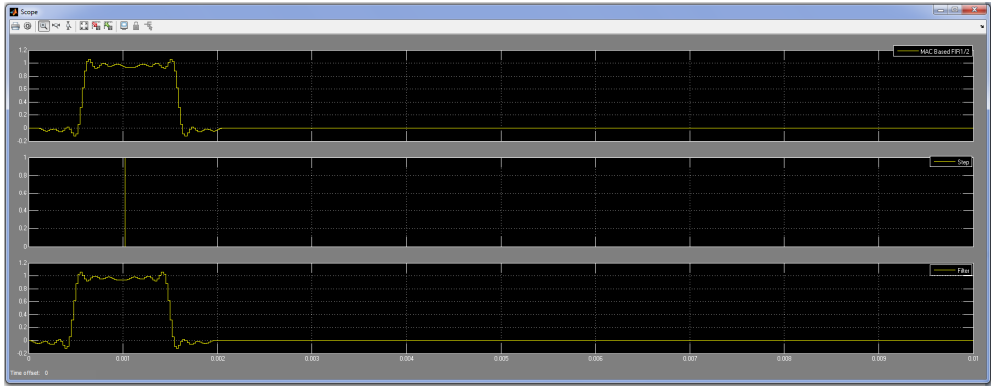


Рис. 4.73. Имитационное моделирование в системе Matlab/Simulink КИХ-фильтра на 34 отвода с использованием одного MAC-блока и блочной памяти



а)

Рис. 4.74. а) Имитационная модель КИХ-фильтра (на вход подается единичный импульс) и моделирование отклика КИХ-фильтра (б)



б)

Рис. 4.74. а) Имитационная модель КИХ-фильтра (на вход подается единичный импульс) и моделирование отклика КИХ-фильтра (б) (продолжение)

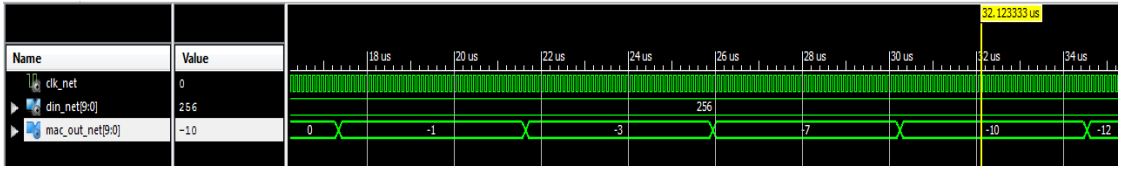


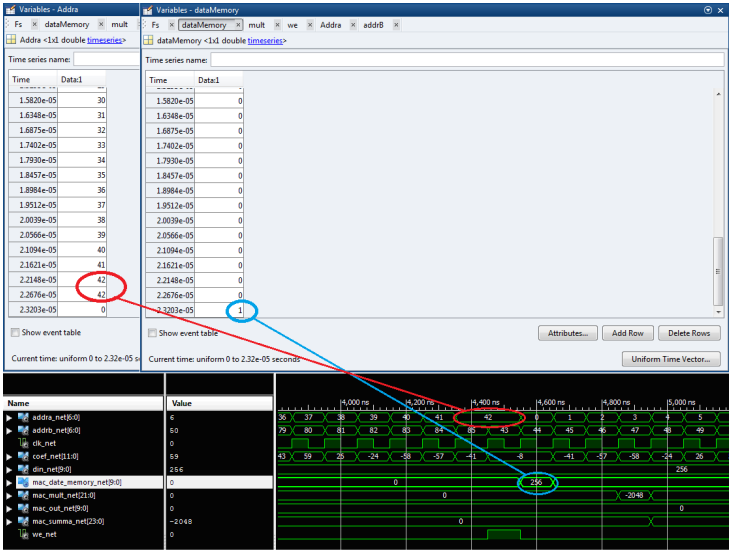
Рис. 4.75. Функциональное моделирование с использованием моделирующей программы на языке VHDL сгенерированной в автоматическом режиме. Входной сигнал (единичный импульс) и выходной сигнал умножаются на 256

Таблица 4.3

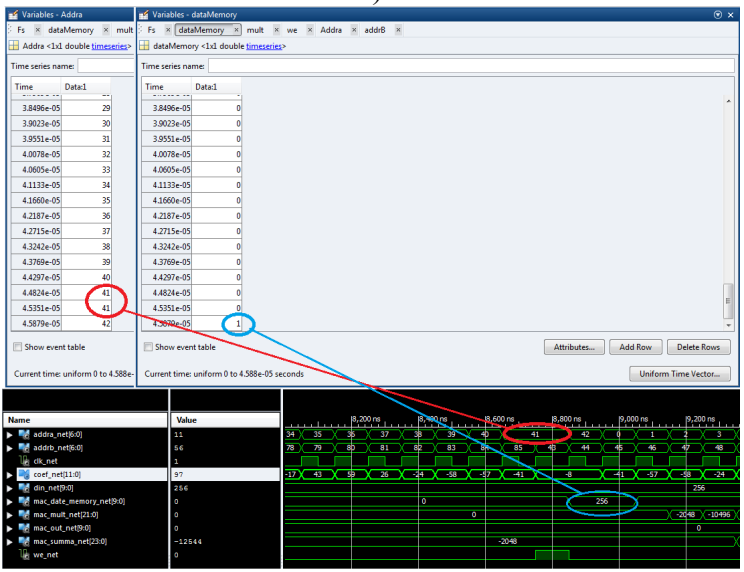
Фрагменты значений выходного сигнала и коэффициентов фильтра в форматах double и FIX при прохождении по структуре единичного импульса

Значения отклика в формате double	Значения отклика в формате FIX_10_8 (значения отклика * на масштабный множитель 256)	Коэффициенты фильтра, в формате double	Коэффициенты фильтра в формате FIX_12_12 (коэф. фильтра * на масштабный множитель 4096)
-0,00195;	-1	-0.0019	-8
-0,01196;	-3	-0.099	-41
-0,02588;	-7	-0.0139	-57
-0,0400;	-10	-0.0141	-58

В рассматриваемом примере предложена оригинальная идея организации работы циклического буфера. Сигнал we приостанавливает на один такт синхроимпульса работу счетчика Data_Counter (рис. 4.76), тем самым происходит двойная адресация к строке 42 (два такта синхроимпульса). Это обеспечивает запись десятичного числа один в строку памяти с адресом 0. При работе в САПР ISE эта единица умножается на масштабный множитель 256. Далее это число будет умножено на коэффициент -8 (-0.0019 в формате double), что и даст результат -2048 (рис. 4.76, *а*). Через последующие 43 такта десятичная единица будет записана в строку с адресом 42 (рис. 4.76, *б*). Эта единица (256) выше описанным способом заполнит первый банк ОЗУ, т.е. “пробежится” по всем коэффициентам фильтра и процедура повторится снова в зависимости от ширины единичного импульса, которая задается параметром Step Time. В нашем случае это величина 0.001 (рис. 4.74, блок Step). Как только импульс упадет в ноль, им последовательно будет заполнен первый банк ОЗУ.



а)



б)

Рис. 4.76. Результаты расчетов в пошаговом режиме в системе Matlab и временные диаграммы в ISE Design Suite поясняющие принцип работы циклического буфера

Проверка результата фильтрации (рис. 4.75).

Фрагмент уравнения КИХ-фильтра:

$$y(FIX24\_20) = -8(FIX12\_12) * X_0 - 41(FIX12\_12) * X_1 - 57(FIX12\_12) - ...$$

Первый проход

$$X_0 = 1(FIX10\_8); y(FIX24\_20) = -8 * 256 = -2048;$$

$$y_{\text{double}} = -2048 / 1048576 \approx -0.00195$$

Переводим формат y_{double} в

$$y(FIX10\_8) = -0.00195 * 256 \approx -0.499 \approx -1.$$

Второй проход

$$X_1 = 1(FIX10\_8); \quad X_0 = 1(FIX10\_8);$$

$$y(FIX24\_20) = -8 * 256 - 41 * 256 = -2048 - 10496 = -12544;$$

$$y_{\text{double}} = -12544 / 1048576 \approx -0.01196.$$

Переводим формат y_{double} в

$$y(FIX10\_8) = -0.01196 * 256 \approx -3.06176 \approx -3.$$

Третий проход

$$X_2 = 1(FIX10\_8); \quad X_1 = 1(FIX10\_8); \quad X_0 = 1(FIX10\_8);$$

$$y(FIX24\_20) = -8 * 256 - 41 * 256 - 57 * 256 = -27136;$$

$$y_{\text{double}} = -27136 / 1048576 \approx -0.02588.$$

Переводим формат y_{double} в

$$y(FIX10\_8) = -0.02588 * 256 \approx -6.62528 \approx -7.$$

С использованием Xilinx System Generator рассмотрено проектирование КИХ-фильтров в формате с фиксированной запятой. Рассмотрение результатов функционального моделирования с использованием моделирующих программ на языке VHDL сгенерированных в автоматическом режиме при переходе от имитационных моделей КИХ-фильтров различной структуры созданных в системе Matlab/Simulink к функциональным в САПР ПЛИС Xilinx ISE Design Suite показало, что входной сигнал, подлежащий фильтрации и коэффициенты фильтра умножаются на масштабные

множители 2^N. Подытоживая можно отметить, что в функциональном блоке n-tap Dual Port Memory MAC FIR Filter используются такие понятия как интерполяция, децимация, двухпортовая память, циклический буфер, латентность.

ЗАКЛЮЧЕНИЕ

В учебном пособии на обширном иллюстративном материале показаны методы обработки цифровых сигналов базисе ПЛИС с учетом их архитектурных особенностей с применением высокоуровневого языка описания аппаратных средств.

Изложены основы проектирования умножителей цифровых сигналов. Подробно рассмотрен алгоритм реализации умножения целых чисел, представленных в дополнительном коде, методом правого сдвига и сложения с накоплением. Даются общие сведения по программным умножителям в базисе ПЛИС.

Показан пример расчета спецификации КИХ-фильтра, показаны эффекты квантования при работе в формате с фиксированной запятой, а также продемонстрировано имитационное моделирование модели КИХ-фильтра в системе Matlab/Simulink.

Демонстрируются различные варианты реализации параллельных КИХ-фильтров в базисе ПЛИС с использованием перемножителей на мегафункциях САПР Quartus II компании Altera.

Даются практические примеры проектирования КИХ-фильтров на последовательной и параллельной распределенной арифметике в САПР ПЛИС Altera Quartus II и Xilinx ISE Design Suite.

Показано, что систолический КИХ-фильтр является оптимальным решением для параллельных архитектур цифровых фильтров, позволяет существенно уменьшить число используемых ресурсов и повысить быстродействие системы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Computer Arithmetic: Algorithms and Hardware Designs (Oxford U. Press, 2nd ed., 2010, ISBN 978-0-19-532848-6).
2. Michael J.S. Smith. Application-Specific Integrated Circuits. VLSI Design Series. P.1040. ISBN 0-201-50022-1. LOC TK7874.6.S63. Addison Wesley Longman, http://www.awl.com
3. Уилкинсон, Б. Основы проектирования цифровых схем [Текст]: пер. с англ. / Б. Уилкинсон. - М.: Издательский дом Вильямс, 2004. - 320 с.
4. Армстронг, Дж. Р. Моделирование цифровых систем на языке VHDL [Текст]: пер. с англ. / Р. Дж. Армстронг. - М.: Мир, 1992. - 348 с.
5. Максфилд, К. Проектирование на ПЛИС: курс молодого бойца [Текст]: пер. с англ. / К. Максфилд. М.: Издательский дом Додэка XXI, 2007. 408 с.
6. Уэйкерли, Джон Ф. Проектирование цифровых устройств: пер. с англ. / Ф. Джон Уэйкерли. М.: Постмаркет, 2002. 533 с.
7. Рабаи, Ж.М. Цифровые интегральные схемы. Методология проектирования [Текст] / Ж.М. Рабаи, А. Чандракасан, Б. Николич. М.: Вильямс, 2007. - 911 с.
8. Угрюмов, Е.П. Цифровая схемотехника [Текст] / Е.П. Угрюмов. СПб.: БХВ, 2004. - 528 с.
9. Стешенко, В. ПЛИС фирмы ALTERA: проектирование устройств обработки сигналов [Текст] / В. Стешенко. М.: Додэка, 2000. - 457 с.
10. Ефремов, Н.В. Введение в систему автоматизированного проектирования Quartus II [Текст] / Н.В. Ефремов. М.: ГОУ ВПО МГУЛ, 2011. - 147 с.
11. Суворова, Е.А. Проектирование цифровых систем на VHDL [Текст] / Е.А. Суворова, Ю.Е. Шейнин. СПб.: БХВ-Петербург, 203. – 576 с.

12. Israel Koren. University of MASSACHUSETTS Dept. of Electrical & Computer Engineering. Digital Computer Arithmetic. ECE 666. Part 3. Sequential Algorithms for Multiplication and Division.

13. Строгонов, А.В. Проектирование умножителя методом правого сдвига и сложения с управляющим автоматом в базисе ПЛИС [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. - 2013. - N12. - С.6-10.

14. Строгонов А.В. Проектирование умножителя целых чисел со знаком методом правого сдвига и сложения в базисе ПЛИС [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. - 2014. - N1. - С.94-100.

15. Строгонов, А.В. Проектирование цифровых фильтров в системе Matlab/Simulink и САПР ПЛИС Quartus [Текст] / А.В. Строгонов // Компоненты и технологии. - 2008. - N6. - С.32-36.

16. Строгонов, А.В. Проектирование параллельных КИХ-фильтров в базисе ПЛИС [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. - 2013. - N6. - С.62-67.

17. Строгонов, А.В. КИХ-фильтр на распределенной арифметике: проектируем сами [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. - 2013. - N3. - С.131-138.

18. Строгонов, А.В. КИХ-фильтры на параллельной распределенной арифметике: проектируем сами [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. 2013. - N5. - С.44-48.

19. Строгонов, А.В. Систолические КИХ-фильтры в базисе ПЛИС [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. - 2013. - N8. - С.30-35.

20. Строгонов, А.В. Проектирование систолических КИХ-фильтров в базисе ПЛИС с помощью системымоделирования

ModelSim-Altera [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. - 2013. - N9. - С.24-28.

21. Строгонов, А.В. Эффективность разработки конечных автоматов в базисе ПЛИС FPGA [Текст] / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. - 2013. - N1. - С.66-72.

22. www.labfor.ru. Учебный лабораторный стенд LESO 2.1. Паспорт и Инструкция по эксплуатации. Новосибирск. 2009.

23. FIR Compiler. User Guide. Software Version: 11.0. May 2011. Altera Corporation.

24. Application Note 73 Implementing FIR Filters in FLEX Devices, February 1998, ver 1.01.

25. Строгонов, А.В. Проектирование КИХ-фильтров в САПР ПЛИС Xilinx ISE Design Suite [Текст] / А.В., Строгонов, С.А. Цыбин, П.С. Городков П // Компоненты и технологии, 2014, N11. С.96-102.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
1. ПРОЕКТИРОВАНИЕ УМНОЖИТЕЛЕЙ В БАЗИСЕ ПЛИС 5
1.1. Двоичная арифметика 5
1.2. Представление чисел со знаком 9
1.3. Матричные умножители 12
1.4. Проектирование умножителя методом правого сдвига и сложения с управляющим автоматом в базисе ПЛИС 18
1.5. Проектирование умножителя целых чисел со знаком методом правого сдвига и сложения в базисе ПЛИС 34
1.6. Общие сведения по программным умножителям в базисе ПЛИС 53
1.7. Разработка проекта умножителя размерностью 4x4 в базисе ПЛИС типа ППВМ серии Cyclone фирмы Altera с помощью учебного лабораторного стенда LESO2.1 63
2. ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ ФИЛЬТРОВ В БАЗИСЕ ПЛИС 79
2.1. Проектирование КИХ-фильтров с использованием системы визуально-имитационного моделирования Matlab/Simulink 79
2.2. Проектирование параллельных КИХ-фильтров в базисе ПЛИС 85
2.3. Проектирование КИХ-фильтра с использованием умножителя на методе правого сдвига и сложения 107
2.4. Проектирование квантованных КИХ-фильтров 116
3. ПРОЕКТИРОВАНИЕ КИХ-ФИЛЬТРОВ НА РАСПРЕДЕЛЕННОЙ АРИФМЕТИКЕ 144
3.1. КИХ-фильтры на последовательной распределенной арифметике 144
3.2. КИХ-фильтры на параллельной распределенной арифметике 165
3.3. Пример реализации КИХ-фильтра на параллельной распределенной арифметике 181

4. СИСТОЛИЧЕСКИЕ КИХ-ФИЛЬТРЫ В БАЗИСЕ ПЛИС 195
4.1. Проектирование систолических КИХ-фильтров в базисе ПЛИС с использованием САПР Quartus II 195
4.2. Проектирование систолических КИХ-фильтров в базисе ПЛИС с использованием системы цифрового моделирования ModelSim-Altera 213
4.3. Проектирование КИХ-фильтров в САПР ПЛИС Xilinx ISE 14.2 230
4.4. Пример проектирования КИХ-фильтров в базисе ПЛИС с применением генератора параметризированных ядер XLogiCORE IP и функции FIR Compiler v6.3 242
4.5. Пример проектирования КИХ-фильтров в базисе ПЛИС с применением генератора параметризированных ядер XLogiCORE IP и функции FIR Compiler v5.0 255
4.6. Проектирование КИХ-фильтров в системе Xilinx System Generator САПР ISE Design Suite 269
4.7. Проектирование КИХ-фильтров со структурой MAC-блоков в системе Xilinx System Generator САПР ISE Design Suite 281
ЗАКЛЮЧЕНИЕ 304
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 305

Учебное издание

Строгонов Андрей Владимирович

ЦИФРОВАЯ ОБРАБОТКА СИГНАЛОВ В БАЗИСЕ ПРОГРАММИРУЕМЫХ ЛОГИЧЕСКИХ ИНТЕГРАЛЬНЫХ СХЕМ

В авторской редакции

Компьютерная верстка А.В. Строгонова

Подписано к изданию 09.04.2015

Объем данных 34,3 Мб

ФГБОУ ВПО «Воронежский государственный технический университет»

394026 Воронеж, Московский просп., 14