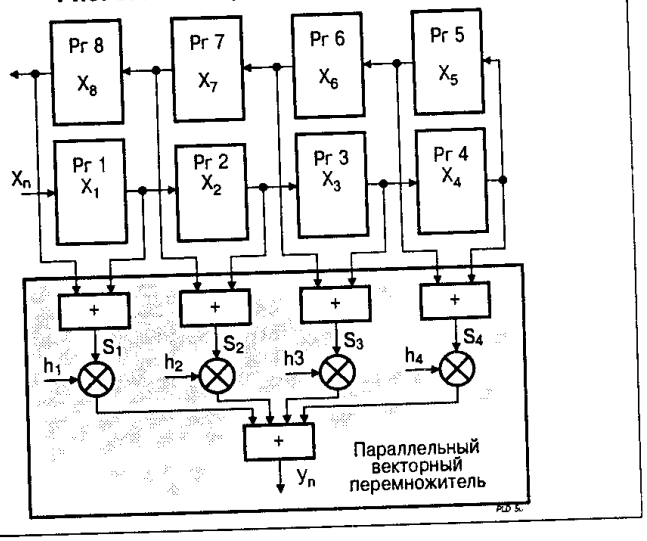
5.1. РЕАЛИЗАЦИЯ ЦИФРОВЫХ ФИЛЬТРОВ НА ПЛИС СЕМЕЙСТВА FLEX ФИРМЫ ALTERA

Многие ЦФ достаточно просто реализовать в виде КИХ-фильтра С появлением БИС семейства FLEX8000 и FLEX10K фирмы ALTERA появилась возможность создания высокопроизводительных и гибких КИХ-фильтров высокого порядка При этом ПЛИС благодаря особенностям своей архитектуры позволяют достигнуть наилучших показателей производительности по сравнению с другими способами реализации ЦФ Так, при реализации ЦФ на базе ЦПОС среднего класса можно достигнуть производительности обработки данных порядка 5 милионнов отсчетов в секунду (MSPS, Million samples per second) Использование готовых специализированных БИС позволяет обеспечить производительность 30 35 MSPS При использовании ПЛИС семейств FLEX8000 и FLEX10K достигаются величины более 100 MSPS

Рис. 5.1. КИХ-фильтр с 8 отводами



Рассмотрим особенности реализации КИХ-фильтров на базе ПЛИС семейств FLEX8000 и FLEX10K с учетом специфики их архитектуры на примере КИХ-фильтра с 8 отводами (**Рис. 5.1**) На **Рис. 5.1** введены следующие обозначения x_n — сигнал на выходе n-го регистра, h_n — коэффициент ЦФ, y_n — выходной сигнал Сигнал на выходе фильтра будет иметь вид

$$y_n = \sum_{n=1}^{8} x_n h_n$$

Для обеспечения линейности фазовой характеристики КИХ-фильтра коэффициенты h_n ЦФ-фильтра выбираются симметричными относительно центральной величины Такое построение позволяет сократить число перемножителей Поскольку компоненты вектора h постоянны для любого фильтра с фиксированными характеристиками, то в качестве параллельного векторного перемножителя (ПВП) удобно использовать таблицу перекодировок (ТП, LUT, Look-up table), входящую в состав логического элемента (ЛЭ) ПЛИС Работа параллельного векторного перемножителя описывается следующим уравнением

$$y = s_1 h_1 + s_2 h_2 + s_3 h_3 + s_4 h_4$$

При использовании ТП операция перемножения выполняется параллельно. В качестве примера рассмотрим реализацию двухразрядного векторного перемножителя Пусть вектор коэффициентов h имеет вид

h_1	h_2	h_3	h_4
01	11	10	11

Вектор сигналов s имеет вид

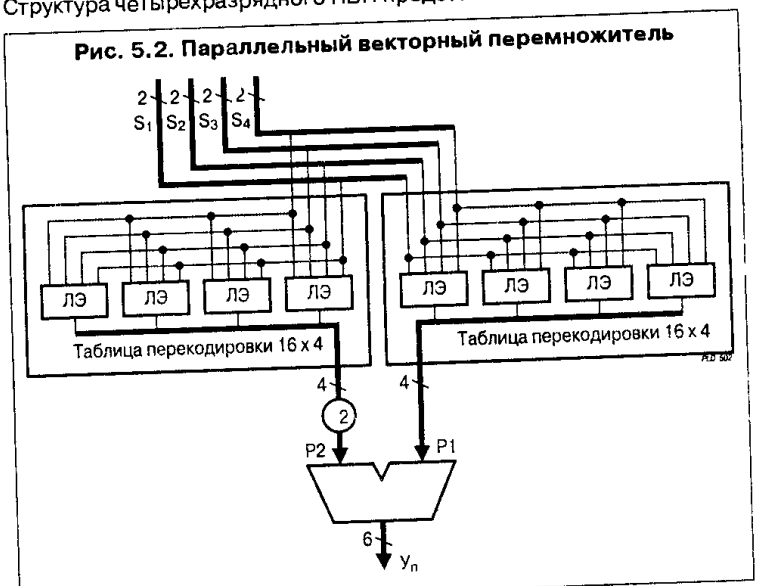
s_1	s_2	s_3	s_4
11	00	10	01

Произведение на выходе ПВП принимает вид

H_n	01	11	10	11	
S_n	11	00	10	01	
Частичное произведение *P1*	01	00	00	11	= 100
Частичное произведение *P2*	01	00	10	00	= 011
Выходной сигнал *y*	011	000	100	011	=1010

Аналогично формируется результат P1в случае четырехразрядных сигналов s_n Частичное произведение P2 вычисляется аналогично, только результат необходимо сдвинуть на 1 разряд влево Структура четырехразрядного ПВП представлена на **Рис. 5.2**

Рис. 5.2. Параллельный векторный перемножитель



Очевидно, что с ростом разрядности представления данных возрастает размерность ТП, а следовательно, требуется большее число ЛЭ для реализации алгоритма фильтрации Так, для реализации восьмиразрядного ПЛП требуется 8 ТП размерностью 16 x 8 Операция умножения на 2 легко обеспечивается сдвиговыми регистрами

Включение сдвиговых регистров на выходах промежуточных сумматоров позволяет обеспечить конвейеризацию обработки данных и, следовательно, повысить производительность При этом регистры входят в состав соответствующих ЛЭ, поэтому сохраняется компактность структуры, не требуется задействовать дополнительные ЛЭ

Основным достоинством параллельной архитектуры построения фильтров является высокая производительность Однако если число задействованных ЛЭ является критичным, предпочтительнее строить фильтр по последовательной или комбинированной архитектуре В **Таблице 5.1** приведены сравнительные характеристики КИХ-фильтров одинаковой разрядности и порядка но выполненных по различной архитектуре

Таблица 5.1. Реализация ЦФ на ПЛИС

Архитектура	Разрядность данных	Порядок	Размер (число ЛЭ)	Тактовая частота, МГц	Число тактов до получения результата	Быстродействие, MSPS
Параллельная	8	16	468	101	1	101
Последовательная	8	16	272	63	9	7

С целью уменьшения числа используемых ЛЭ применяется последовательная архитектура построения фильтра Так же, как и при параллельном построении фильтра, для вычисления частичных P1, P2, , PN, N = W + 1 произведений применяется ТП, W – разрядность данных Такой ЦФ обрабатывает только один разряд входного сигнала в течение такта Последовательно вычисляемые частичные произведения накапливаются в масштабирующем аккумуляторе (МА). МА обеспечивает сдвиг содержимого вправо на один разряд каждый такт. После W + 1 тактов на выходе появляется результат Блок управления обеспечивает формирование управляющих сигналов, обеспечивающих корректное выполнение операций

Комбинированная архитектура является компромиссом между экономичностью и производительностью В этом случае применяются как последовательные, так и параллельные регистры В результате распараллеливания вычислений обеспечивается несколько большая производительность, чем у последовательного фильтра

Одним из эффективных способов повышения производительности фильтра является конвейеризация В случае если длина вектора S не пропорциональна 2^h, используются дополнительные регистры для обеспечения синхронизации С целью построения фильтров более высокого порядка используют последовательное соединение фильтров (каскадирование) меньшего порядка

Увеличение разрядности данных требует обеспечения большей точности вычислений и разрядности коэффициентов Увеличение разрядности данных на один бит требует использования дополнительной ТП при параллельной архитектуре и увеличения на один такт времени фильтрации при последовательной архитектуре При этом для обеспечения достаточной точности представления данных требуется для фильтра 32-го порядка 19 разрядов длины выходного слова при восьмиразрядных входных данных

Построение фильтра нечетного порядка достигается удалением одного из регистров сдвига Если применять не сумматоры, а вычитатели то легко реализовать фильтр с антисимметричной характеристикой

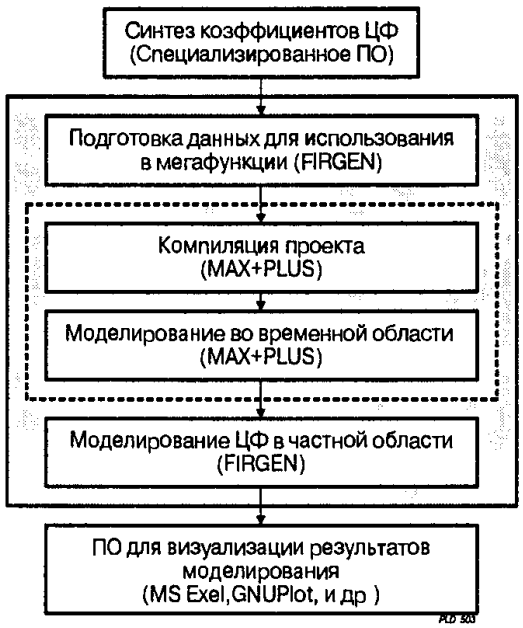
Довольно легко реализуются фильтры с прореживанием или, наоборот, с интерполяцией отсчетов При проектировании прореживающего фильтра обеспечивается покаскадное уменьшение тактовой частоты, что позволяет существенно снизить энергопотребление Интерполирующий фильтр выполняет противоположную операцию — увеличение частоты выборки в определенное число раз Одним из способов реализации такого алгоритма является дополнение отсчетов нулями

Возможности архитектуры ПЛИС семейств FLEX позволяют реализовать двумерные фильтры для обработки изображений, а также решетчатые структуры

Для проектирования фильтров необходим набор специализированных программных средств, позволяющих синтезировать требуемую системную функцию фильтра, провести моделирование его работы, а также библиотеки параметризируемых мегафункций, содержащих реализации типовых структур ЦФ

Фирма ALTERA предоставляет в составе Altera DSP Design KIT програмный продукт Firgen В функции программы Firgen входит моделирование КИХ-фильтра с заданными коэффициентами, а также генерация файлов, предназначенных для реализации фильтра штатными средствами пакета MAX+Plus II. Для построения отклика фильтра, полученного в результате моделирования средствами Firgen можно использовать либо пакет Microsoft Exel, либо входящий в состав Altera DSP Design KIT продукт GNUplot. Схематически процесс разработки фильтра представлен на **Рис 5.3.**

Рис. 5.3. Процесс разрабстки фильтра в пакете MAX+PLUS II



В **Таблице 5.2** сведены данные о мегафункциях фильтров, входящих в состав Altera DSP Design KIT

Таблица 5.2. Фильтры из пакета DSP Design KIT

Модель	Разрядность входных данных	Число отводов	Разрядность представления			Размер (число ЛЭ)	Производительность (MSPS)	
			Коэффициентов	Внутренняя	Выходных данных		С конвейером	Без конвейера
Fir_08tp	8	8	8	17	17	296	101	28
Fir_16tp	8	16	8	10	10	468	101	20
Fir_24tp	8	24	8	10	10	653	100	18
Fir_32tp	8	32	8	10	10	862	101	18
Fir_16ts	8	16	8	18	18	272	7	3 4
Fir_64ts	8	64	8	24	24	920	6 5	2 4
Fir_3x3	8	9	8	18	18	327	102	24

5.2. РЕАЛИЗАЦИЯ ЦИФРОВЫХ ПОЛИНОМИАЛЬНЫХ ФИЛЬТРОВ

В общем случае цифровой полиномиальный фильтр размерности *r* и порядка *M*, определяется конечным дискретным рядом Вольтерра (функциональным полиномом) вида·

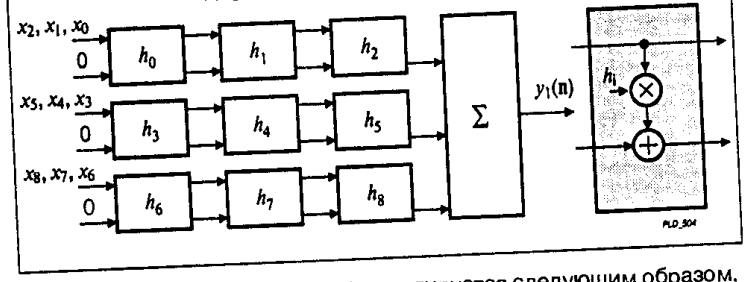
$$y(n) = h_0 + \sum_{m=1}^{M} y_m(n) = h_0 + \sum_{m=1}^{M} \sum_{n_1} \cdots \sum_{n_m} h_m(n_1,\ldots,n_m) \prod_{i=1}^{m} x(n-n_i) ,$$

где $h_m(\mathbf{n}_1, \ldots, \mathbf{n}_m)$ — многомерные импульсные характеристики (ядра) фильтра, зависящие от векторных аргументов $\mathbf{n}_i = [n_{i1} \ldots n_{ik}]$. Фильтры данного вида часто называются также фильтрами Вольтерра.

Непосредственная реализация цифровых полиномиальных фильтров связана с вычислением произведения векторов. Вычислительные затраты могут быть сокращены за счет использования свойства симметрии изотропных фильтров.

Таким образом, путем последовательного применения процедуры декомпозиции полиномиальный фильтр произвольного порядка может быть предстаален в виде параллельной структуры, состоящей из линейных фильтров Операция линейной фильтрации связана с вычислением двухмерной свертки и допускает высокоэффективную реализацию в виде структур систолического типа, матричных и волновых процессоров. Процесс двухмерной свертки изображения с маской $N \times N$, в свою очередь, можно свести к вычислению N одномерных сверток, что в конечном итоге позволяет выполнять полиномиальную фильтрацию изображений путем параллельного вычисления обычных одномерных сверток На **Рис. 5.4** представлен один из наиболее простых вариантов реализации двухмерной свертки изображения с маской 3 x 3 в виде систолической структуры, состоящей из 9 идентичных процессорных элементов.

Рис. 5.4. Систолическая реализация двумернсй линейнсй свертки



Задача двумерной свертки формулируется следующим образом. даны веса $w_{i,j}$ для $i, j = 1, 2\ldots, k$, так что $k \times k$ — размер ядра, и входное изображение $x_{i,j}$ для $i, j = 1, 2\ldots, n$. Требуется вычислить элементы изображения $y_{i,j}$ для $i, j = 1, 2\ldots, n$, определяемые в виде:

$$y_{i,j} = \sum_{l=1}^{k} \sum_{h=1}^{k} w_{hl} x_{i+h-1, j+l-1}.$$

При k = 3 двумерная свертка выполняется в виде трех последовательных одномерных сверток (использующих в качестве весов одну и ту же последовательность ($w_{11}, w_{21}, \ldots, w_{33}$).

1 Вычисление (y_{11}, y_{12}, y_{13}, y_{14}, .) при использовании (x_{11}, x_{21}, x_{31}, x_{12}, x_{22}, x_{32}, x_{13}, x_{23}, x_{33},) в качестве входной последовательности.

2 Вычисление (y_{21}, y_{22}, y_{23}, y_{24},) при использовании (x_{21}, x_{31}, x_{41}, x_{22}, x_{32}, x_{42}, x_{23}, x_{33}, x_{43}, ..) в качестве входной последовательности.

3. Вычисление (y_{31}, y_{32}, y_{33}, y_{34},. .) при использовании (x_{31}, x_{41}, x_{51}, x_{32}, x_{22}, x_{32}, x_{33}, x_{43}, x_{53},...) в качестве входной последовательности

Каждую из этих сверток можно выполнить на одномерном систолическом массиве из девяти ячеек Отметим, что в любом из этих одномерных массивов ячейка занята вычислениями y_{ij} только 1/3 времени.

Для восстановления ячейка с двумя потоками должна быть восьмиразрядной для входных данных и иметь 12 бит для представления весов Для сетки 32 x 32 пикселей в каждом блоке памяти требуется хранить 1024 различных весовых коэффициента и иметь 10-разрядную адресную шину для указания положения каждой точки 12-разрядный умножитель и 24-разрядный сумматор для получения промежуточных результатов вполне обеспечивают сохранение требуемой точности в процессе вычислений

Для выполнения многомерной свертки требуется единственная модификация базовой ячейки с двумя потоками — добавление требуемого числа систолических входных потоков и соответствующее увеличение размера мультиплексора для выбора данных.

При решении задач фильтрации изображения с целью удаления шумов, восстановления изображения или улучшения его качества приходится иметь дело с данными, имеющими широкий динамический диапазон Поэтому в качестве формата данных необходимо использовать числа с плавающей запятой

Для реализации систолических структур полиноминальных фильтров наиболее пригодны ПЛИС семейства FLEX10K, содержащие встроенные блоки памяти (EAB — embedded array blocks), предназначенные для эффективной реализации функций памяти и сложных арифметических и логических устройств (умножителей, конечных автоматов, цифровых фильтров и т. д.). Один такой блок имеет емкость 2 килобита и позволяет сформировать память с организацией 2048 x 1, 1024 x 2, 512 x 4 или 256 x 8, работающую с циклом 12.. 14 нс. Использование ВБП значительно повышает эффективность и быстродействие создания сложных логических устройств, например умножителей Так, каждый ВБП может выполнять функции умножителя 4 x 4, 5 x 3 или 6 x 2

На **Рис. 5.5** приведена реализация структуры на ПЛИС, а на **Рис. 5.6** результаты моделирования систолической структуры в среде MAX PLUS II

Включение сдвиговых регистров на выходах промежуточных сумматоров позволяет обеспечить конвейеризацию обработки данных и, следовательно, повысить производительность. При этом регистры входят в состав соответствующих ЛЭ, поэтому сохраняется компактность структуры, не требуется задействовать дополнительные ЛЭ

При реализации операции умножения на константу и возведения в степень целесообразно использовать конструкцию TABLE языка AHDL Как показывает сравнение, такое построение позволяет обеспечить более компактную и быстродействующую реализацию Ниже приводится пример такой конструкции

```
TABLE
   net[11..1]  =>     out[15..1],
B"0000000XXXX" => B"000000000000000";
B"00000010000" => B"000000101011101";
B"00000100000" => B"000001010111010";
.
.B"11111110000" => B"010011100001101";
B"00000010001" => B"010011100001110";
 End TABLE;
```

При реализации перемножителей, как правило, используются матричные структуры, построенные каскадированием битовых сумматоров.

Рис 5.5. Реализация систолической структуры на ПЛИС

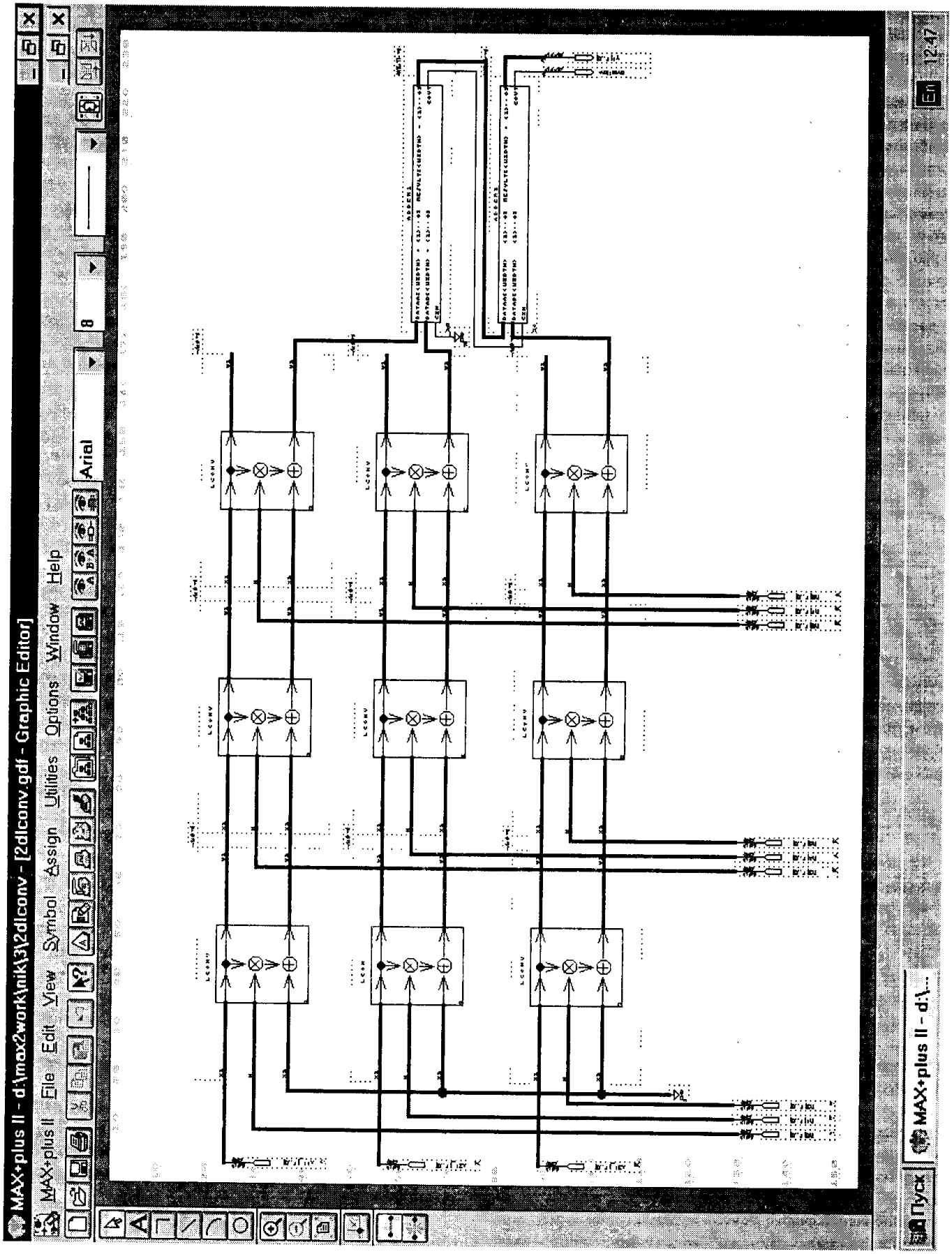
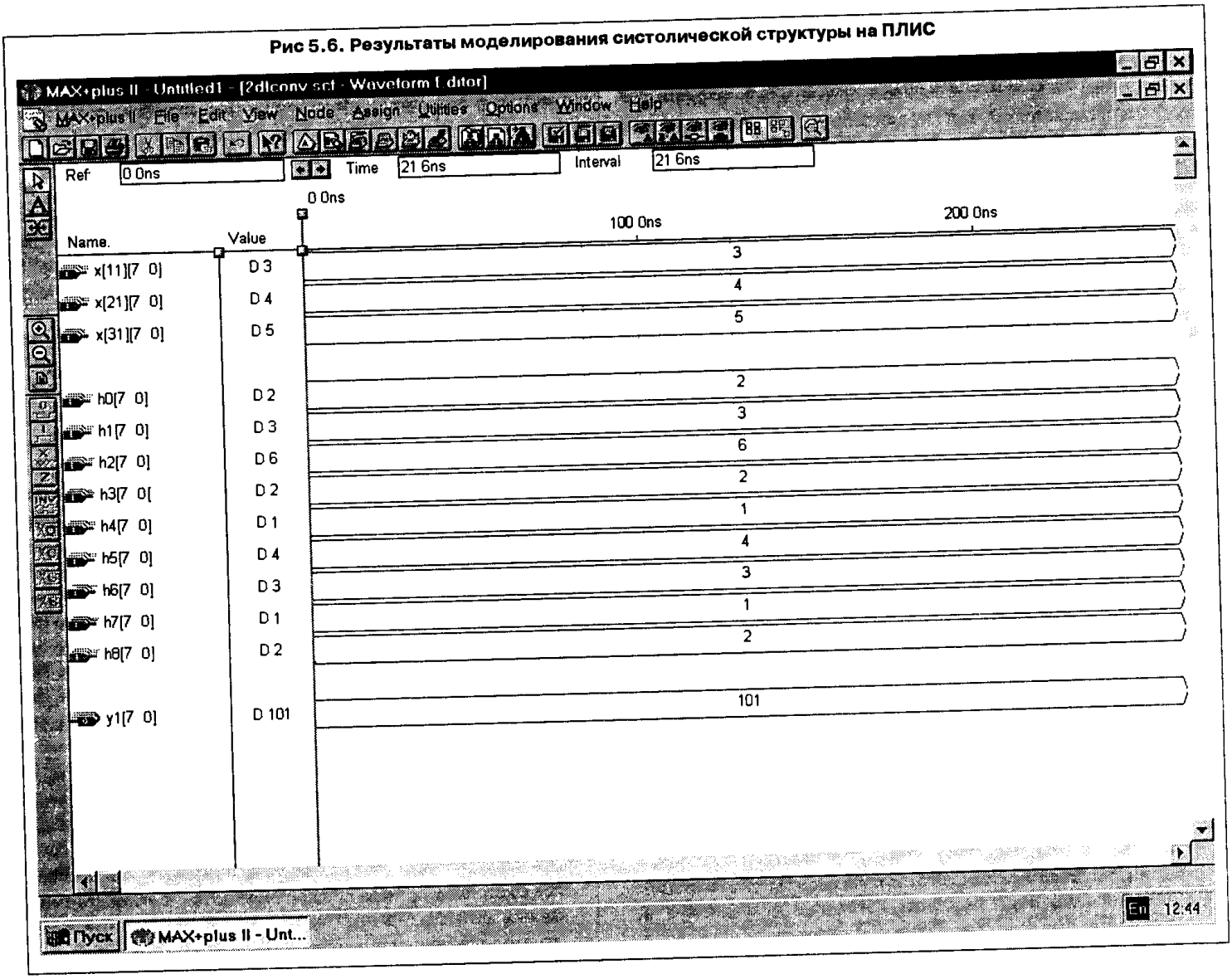


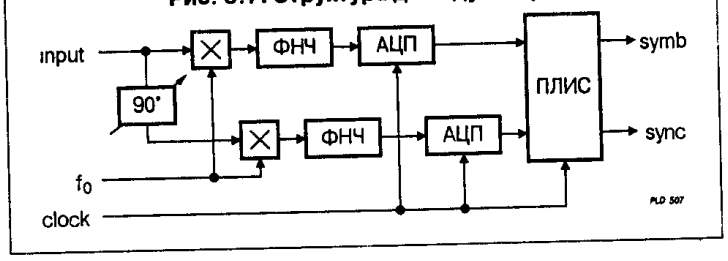
Рис 5.6. Результаты моделирования систолической структуры на ПЛИС



5.3. АЛГОРИТМЫ ФУНКЦИОНИРОВАНИЯ И СТРУКТУРНЫЕ СХЕМЫ ДЕМОДУЛЯТОРОВ

Обобщенная структурная схема, по которой реализован демодулятор сигналов с частотной манипуляцией, приведена на **Рис. 5.7**

Рис. 5.7. Структура демодулятора



Как видим, используется аналоговая квадратурная обработка (перемножители, фазовращатель, фильтры нижних частот) и цифровое восстановление символов и частоты, реализованное на ПЛИС. В качестве формирователя квадратур использована ИС RF2711 фирмы RF Microdevices Данная микросхема содержит в своем составе два перемножителя и фазовращатель и позволяет работать в диапазоне частот от 0 1 до 200 МГц при ширине спектра до 25 МГц. Опорная частота f_0 формируется с помощью синтезатора прямого синтеза частот AD9830 [1] фирмы Analog Devices Сигнал с выхода синтезатора фильтруется с помощью активного ФНЧ, реализованного на ОУ AD8052 по схеме Рауха. В настоящее время наблюдается тенденция к "оцифровыванию" обрабатываемого сигнала на промежуточных частотах (ПЧ) порядка десятков МГц, формируя квадратуры "в цифре". Для этих целей возможно использовать ИС AD6620 Однако это не всегда оправдано в основном из-за сложностей с управлением такой микросхемой в системах, где отсутствует собственный контроллер.

В качестве АЦП удобно использовать специализированный квадратурный АЦП AD9201. Пожалуй, единственным его недостатком является необходимость демультиплексирования отсчетов синфазной и квадратурной составляющих

Ниже приводится описание конструкций каждого из блоков в составе ПЛИС детектора ЧМ-сигнала и синхронизатора Определены преимущества каждой из предлагаемых схем

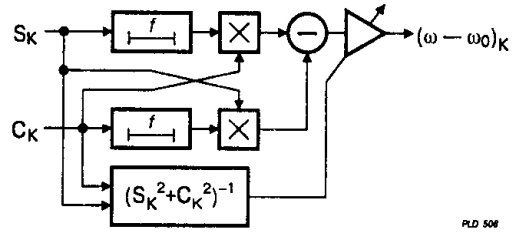
Детектор ЧМ-сигнала предназначен для преобразования исходного модулированного радиосигнала в последовательность прямо-

угольных импульсов, появляющихся с частотой следования символов и обладающих той же полярностью. Частота исходного радиосигнала равна $f = f_0 - \Delta f/2$, если передается символ "0", и $f = f_0 + \Delta f/2$, если передается символ "1", при этом, по техническому заданию, $\Delta f T_C = 1$, где T_C — длительность символа (индекс модуляции единица, сигнал без разрыва фазы) Таким образом, измерение разности $f - f_0$ — это и есть та операция, которую должен осуществлять детектор В Приложении 1 показано, что при наличии отсчетов квадратур исходного радиосигнала S_K и C_K и , k = 0, 1, 2 и т д , величина может быть вычислена следующим образом

$$(f - f_0)_k \approx \frac{1}{2\pi}\left[\frac{S_k C_{k-1} - C_k S_{k-1}}{S_k^2 + C_k^2}\right]$$

Отсюда вытекает структурная схема детектора, которая приведена на **Рис. 5.8**

Рис. 5.8. Цифровой частотный детектор



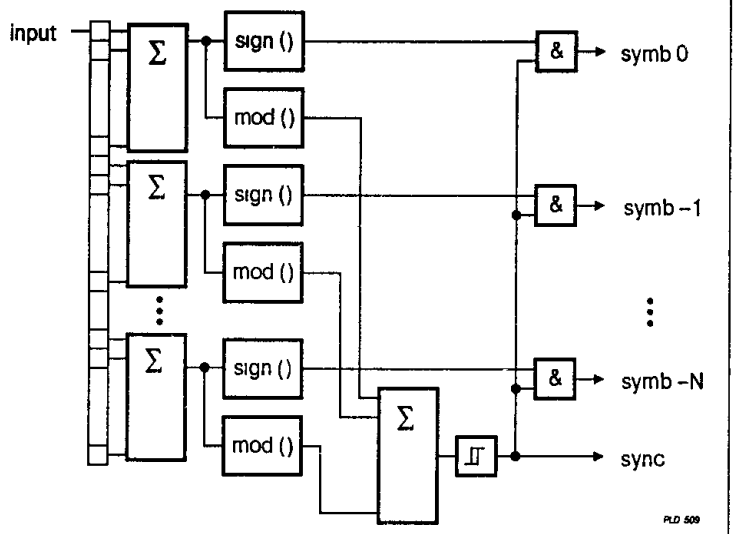
Следует отметить преимущества предлагаемого алгоритма демодуляции ЧМ-сигнала

- детектор не требует точной настройки квадратурного генератора (рис 1) на частоту f_0, что позволяет ему устойчиво функционировать при значительных (до 30%) уходах частоты входного сигнала вследствие эффекта Доплера,
- операция деления на двучлен $S_K^2 + C_K^2$ не является обязательной, если динамика входного сигнала невелика, либо стабилизация амплитуды осуществляется при помощи АРУ в ВЧ-тракте,
- инвариантность алгоритма к фазе опорного и входного сигналов, а также к амплитуде входного сигнала (при наличии нормирующего множителя) увеличивает помехоустойчивость.

При достаточно больших отношениях сигнал/шум (ОСШ) на входе демодулятора (20 30 дБ) восстановленную последовательность символов можно снимать непосредственно с выхода детектора Однако при снижении ОСШ (до 10...15 дБ) форма сигнала на выходе детектора начинает искажаться (появляются ложные перепады, смещение фронтов по времени и т п) Поэтому на выход детектора (внутри ПЛИС) подключается еще один блок – синхронизатор, предназначение которого – восстановить истинную форму демодулированного радиосигнала за счет его накопления и анализа в течение N подряд идущих символов (в описанной далее версии демодуляторов N = 10) Синхронизатор реализует оптимальный (по критерию максимума правдоподобия) алгоритм оценки сигнала прямоугольной формы на фоне белого гауссовского шума Восстановлению подлежат истинные моменты смены символов в исходном радиосигнале (тактовая синхронизация), а также истинная полярность символов

Главными элементами синхронизатора (**Рис. 5.9**) являются линия задержки на N x M отсчетов (M— число отсчетов на символ) и (N + 1) сумматоров, реализующих операцию накопления Синхро-

Рис. 5.9. Цифровой синхронизатор



низатор функционирует следующим образом каждый отсчет входного сигнала порождает сдвиг в линии задержки, после чего вычисляются суммы каждых M подряд идущих отсчетов, определяются их модули и производится усреднение результатов по N суммам (символам). Если в какой-то момент времени каждое суммирование (по M отсчетам) будет производиться внутри одного символа, значение усредненного сигнала будет максимальным, на выходе порогового устройства появится синхроимпульс, и в этот же момент будут считаны знаки накопленных сумм, с высокой вероятностью совпадающие с полярностями символов

К преимуществам предлагаемого алгоритма следует отнести

- высокую эффективность (устойчивость к помехам, к уходу частоты следования символов от номинальной, к снижению частоты дискретизации и др),
- способность точно восстанавливать моменты смены символов во входном сигнале при длинных (до N – 1 включительно) сериях "нулей" и "единиц", причем в конце серии отсутствует переходный процесс (направленный на устранение накопленной ошибки), что характерно для аналоговых устройств,
- наличие на выходе демодулятора одновременно N подряд идущих символов, что может быть важно при корреляционной обработке потока данных (например, при поиске синхропосылок),
- простоту операций (суммирование, сдвиг) и хорошую адаптацию к реализации на базе ПЛИС, что не характерно для большинства традиционных алгоритмов, содержащих петли обратной связи (синхронизатор с запаздывающим и опережающим стробированием и др.)

Реализация цифровой части алгоритма демодуляции и выделения синхроимпульса была выполнена на кристалле Altera FLEX10K50. Реализованное устройство состоит из входных цифровых КИХ-фильтров с восемью отводами, непосредственно блока демодуляции сигнала и блока синхронизатора.

Для реализации входных КИХ-фильтров был использован пакет Altera DSP Design Kit. Данный пакет был выбран ввиду того, что он позволяет по рассчитанным коэффициентам цифрового фильтра быстро получить AHDL описание устройства с данными характеристиками и максимально доступной точностью при заданных значениях точности входных данных и внутреннего представления коэффициентов фильтрации Для этого исходные коэффициенты

фильтра масштабируются, что приводит к тому, что выходной сигнал фильтра также является промасштабированным на ту же величину, которую в большинстве случаев необходимо учитывать при дальнейших вычислениях Однако для данной задачи это не является существенным, и масштабирующий множитель не учитывался в последующих операциях. Кроме этого, DSP Design Kit позволяет сгенерировать векторный файл для моделирования работы фильтра, а также позволяет преобразовать выходные данные отклика фильтра к масштабу входных данных и построить график отклика фильтра на входное воздействие.

Реализованный восьмиотводный фильтр имеет симметричную характеристику

В отличие от классической реализации КИХ-фильтров в виде набора умножителей для взвешивания задержанных отсчетов входного сигнала и выходного сумматора, данная реализация вообще не содержит умножителей. Все операции умножения заменены операциями распределенной арифметики, что возможно благодаря постоянству коэффициентов фильтрации и наличию в логических элементах FLEX10K таблиц перекодировки (look-up-tables, LUTs)

Сигналы с выходов фильтров (восемь бит в дополнительном коде) подаются на квадратурные входы блока частотной демодуляции

Для реализации демодулятора ЧМ-сигналов понадобились два регистра для хранения значений квадратур в предыдущий (k-й) момент времени — S_K и C_K, два умножителя и один сумматор Все вычисления в схеме производятся в дополнительном коде за исключением умножителей, операнды и выходные данные которых представлены в прямом коде со знаком, что требует предварительно преобразовывать сигналы в дополнительный код до умножения и конвертировать в дополнительный код результат умножения Выходной сигнал демодулятора имеет разрядность равную пятнадцати битам, однако для выделения символов нужно рассматривать только старший (знаковый) разряд результата

Входным сигналом синхронизатора является выход блока демодулятора.

Заметим, что для реализации суммирования вида

$$sum_i = y_k + y_{k-1} + y_{k-2} + \quad + y_{k-n+1}$$

нецелесообразно использовать каскад из n двухвходовых сумматоров, так как на каждом такте результат этого суммирования может быть получен из значения суммы на предыдущем такте путем вычитания y_{k-n-1} и прибавления y_{k+1} А именно $sum_{i+1} = sum_i + y_{k+1} + y_{k-2} - y_{k-n-1}$. Таким образом, для реализации этой части алгоритма синхронизации понадобится один регистр для хранения значения суммы на предыдущем такте и три сумматора, один из которых используется для изменения знака у значения y_{k-n-1} (так как все числа представлены в дополнительном коде) Кроме того, необходимы регистры для хранения значений y_k, y_{k-1}, y_{k-n+1} Если же не учитывать эти n регистров, то количество элементов для выполнения такой операции суммирования не будет зависеть от количества операндов и можно сэкономить ячейки ПЛИС при количестве слагаемых в сумме $n > 4$ В данном же случае количество слагаемых в каждой сумме равно количеству отсчетов сигнала, приходящихся на символ, то есть $n = 8$ и эффект в увеличении скорости, а главное, в уменьшении занимаемого места, ощутим

Блок сумматора был реализован в виде мегафункций, выходными сигналами которого являлись как значение суммы, так и значение y_{k-n-1}, для удобства последовательного соединения таких блоков по входам, что необходимо виду того, что выходящее из одной суммы слагаемое становится слагаемым следующей суммы

$$X_k = \ell n\,ch\,(sum_i) + \ell n\,ch\,(sum_{i+1} + \quad + \ell n\,ch\,(sum_{i+m-1})$$

Так как синхронизатор работает по принципу максимума правдоподобия, то схема должна выставлять синхроимпульс в момент достижения выходным сигналом X_k максимума Для определения момента наступления локальных максимумов этот сигнал дифференцируется, и определяются моменты смены знака продифференцированного сигнала

Результаты моделирования в системе Max+Plus II приведены на **Рис. 5.10**

В заключение отметим, что все узлы системы были реализованы в виде параметризованных мегафункций с использованием языка описания аппаратуры AHDL, что позволяет с легкостью их использовать для приложений, требующих другой точности вычислений

Рис. 5.10. Результаты моделирования

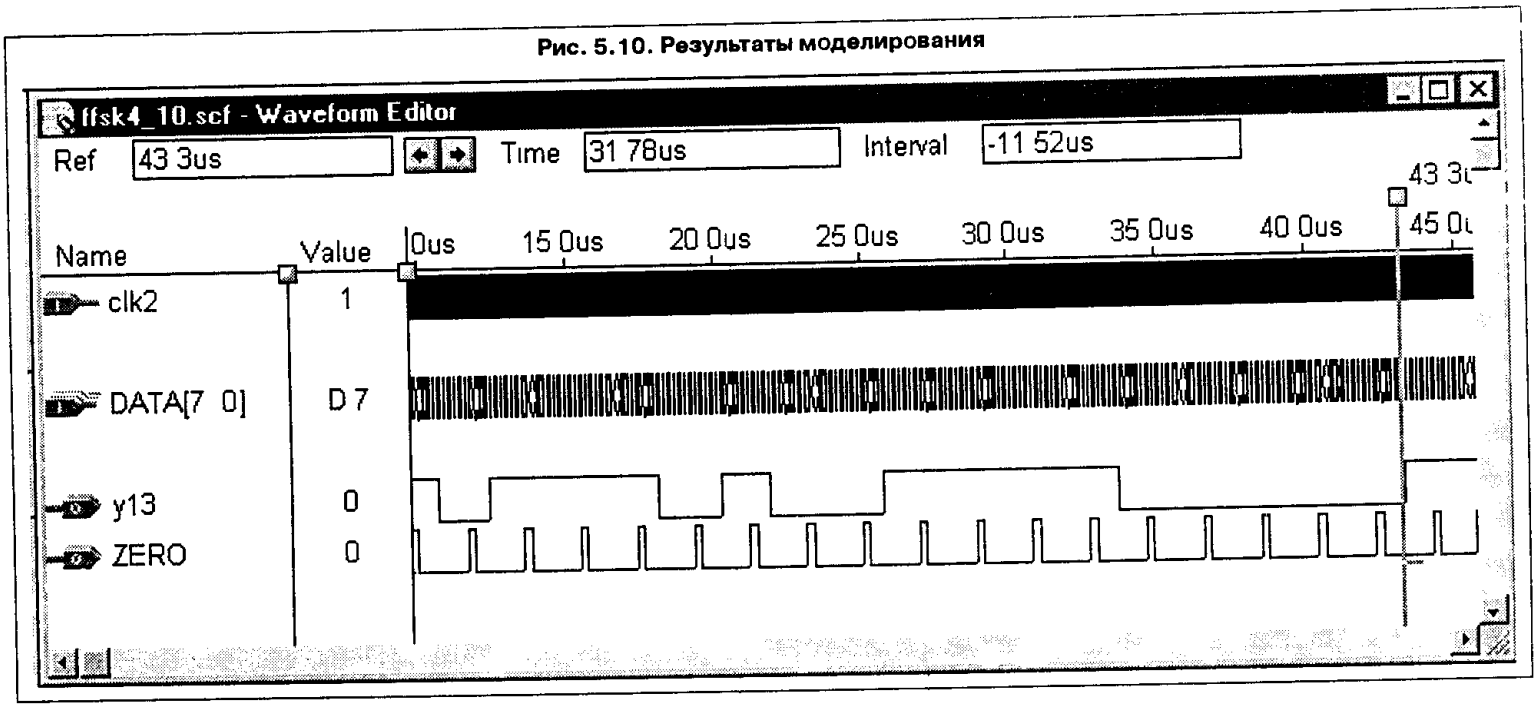
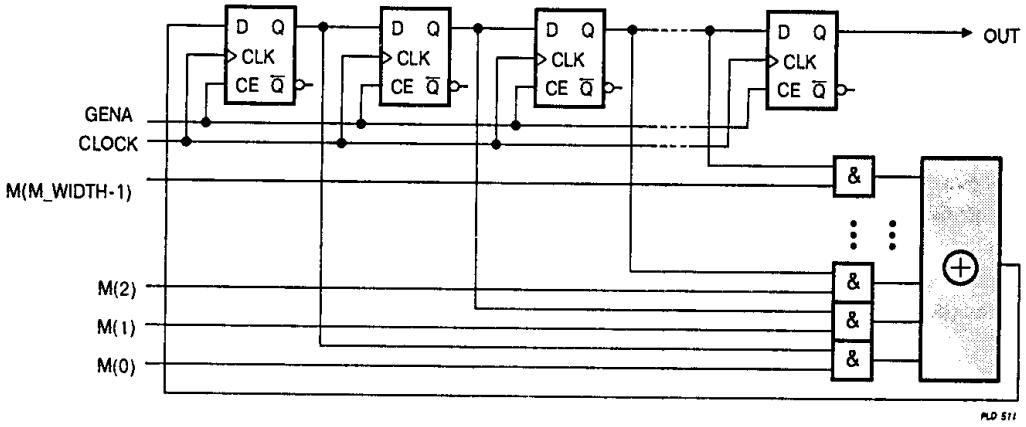


Рис. 5.11. Генератор ПСП



5.4. РЕАЛИЗАЦИЯ ГЕНЕРАТОРА ПСП НА ПЛИС

Генератор формирования М-последовательностей был написан в виде параметризованной макрофункции, описывающей устройство, упрощенная структура которого показана на **Рис. 5.11**. Параметрами макрофункции являются длина характеристического многочлена и число, описывающее начальные состояния триггеров

Ниже приводится листинг описания этой функции на AHDL

Функция M_GENERATE

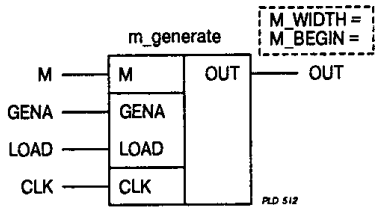
```
%=============================================%
%   Функция M_GENERATE                        %
%   Генерирует М-последовательность, заданную
%   характеристическим многочленом М          %
%                                             %
%   Входы                                     %
%     M                 - вход многочлена,    %
%     GENA      - разрешение генерации (default=GND); %
%     CLK       - тактовый вход,              %
%     LOAD      - вход предустановки (default=GND), %
%   Выходы                                    %
%     OUT       - выход М-последовательности  %
%                                             %
%   Параметры                                 %
%     M_WIDTH   - длина полинома;             %
%     M_BEGIN   - начальное состояние         %
%                       (default="100000 .00"),
%                                             %
%   Версия 1 0                                %
%=============================================%

INCLUDE "lpm_xor.inc",
INCLUDE "lpm_constant.inc",

PARAMETERS
(
 M_WIDTH,
 M_BEGIN = 0
),

SUBDESIGN m_generate
(
    M[M_WIDTH-1..0]                 : INPUT = GND;
    CLK                                     : INPUT;
    GENA                            : INPUT = GND;
    LOAD                            : INPUT = GND;
    OUT                                     : OUTPUT;
   )
   VARIABLE
    dffs[M_WIDTH-2..0]              : DFFE; -  триггеры
регистра сдвига
    shift_node[M_WIDTH-2..0]        : NODE;
    xor_node[M_WIDTH-2..0]          : NODE;
    shiftin, shiftout                       : NODE; -
вход и выход регистра сдвига

    IF (USED(M_BEGIN)) GENERATE
       ac . lpm_constant
                  WITH (LPM_WIDTH = M_WIDTH, LPM_CVALUE
= M_BEGIN),
    END GENERATE,

   BEGIN

    ASSERT (M_WIDTH>0)
       REPORT "Значение параметра M_WIDTH должно быть
больше нуля"
       SEVERITY ERROR;

   % ------------ общие выводы триггеров ------------%
    dffs[].ena = GENA;
    dffs[].clk = CLK;

   % ------------ асинхронные операции ------- %
    IF (USED(M_BEGIN)) GENERATE
       dffs[] clrn = !load # ac.result[M_WIDTH-2..0];
% установка начального состояния %
       dffs[].prn = !load # !ac.result[M_WIDTH-2..0];
% триггеров, заданное M_BEGIN        %
    ELSE GENERATE
       dffs[M_WIDTH-3..0].clrn = !load;             %
установка начального состояния %
```

```
      dffs[M_WIDTH-2].prn = !load;
% триггеров "10000..00                                    %
   END GENERATE;

   % ———— обратные связи ———————————————————————— %
   xor_node[] = dffs[] & M[M_WIDTH-2..0];
   shiftin = lpm_xor(xor_node[])
                        WITH (LPM_SIZE = M_WIDTH-1,
LPM_WIDTH=1);

   % ——— операции сдвига ——————————————————————— %
   shift_node[] = (shiftin, dffs[M_WIDTH-2..1]);
   shiftout = dffs[0];

   % ——— синхронные операции ——————————————————— %
   dffs[].d = !load & shift_node[];

   % ———— подключим выход ———————————————————— %
   OUT = shiftout;
   END;
```

Описание функции.

Входы:

Рис. 5.12. Символ генераторв ПСП



M — шина (группа выводов) с размером M_WIDTH На этот вход подаются коэффициенты характеристического многочлена,

CLK — вход тактового сигнала,

GENA — вход разрешения генерации При GENA = "0" генерация М-последовательности запрещена,

LOAD — вход предустановки При LOAD = "1" триггеры регистра сдвига устанавливаются в состояние, определяемое параметром M_BEGIN,

Выходы

OUT — выход М-последовательности

Для включения данной функции в другие схемы были также созданы включаемый файл, содержащий описание, и графический символ-элемент (см **Рис. 5.12**)

Рис. 5.14. Результаты моделировввния генервтора ПСП

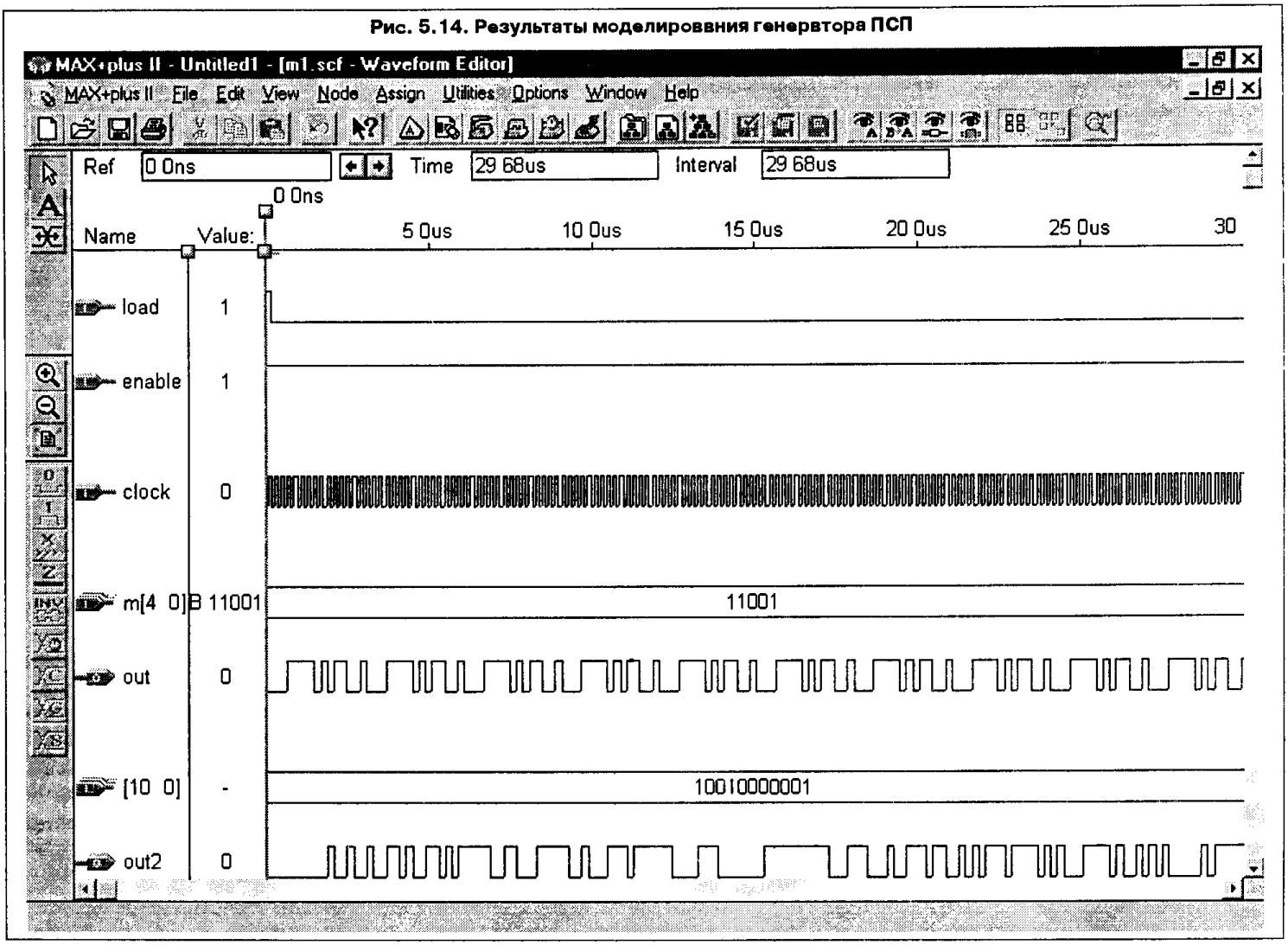
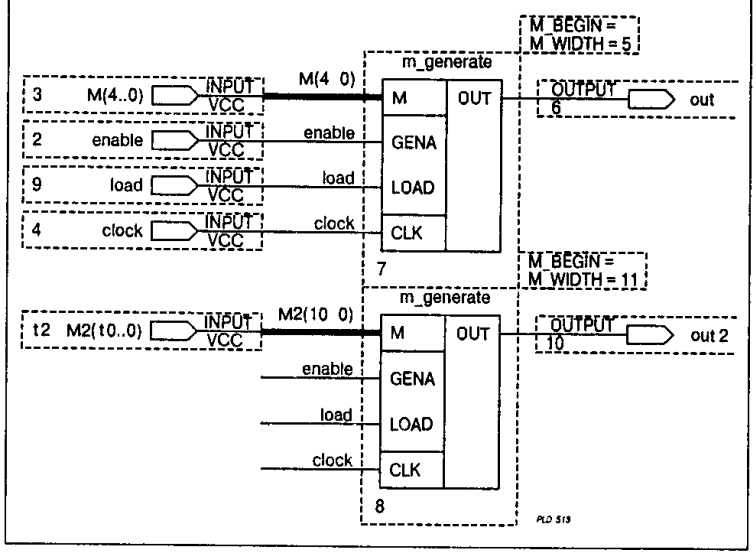


Рис. 5.13. Схема тестирсвания генератсра ПСП



Файл "**M_GENERATE.INC**"

```
FUNCTION m_generate (m[(m_width) - (1) 0], clk,
gena, load)
WITH (M_WIDTH, M_BEGIN)
        RETURNS (out),
```

Для моделирования работы генератора в графическом редакторе пакета MAX+PLUS II была создана тестовая схема, показанная на **Рис. 5.13** Результаты моделирования показаны на **Рис. 5.14** (на вход одного генератора подан характеристический многочлен M_1 = 11001, на вход другого – M_2 = 10010000001)

5.5. ПРИМЕРЫ ОПИСАНИЯ ЦИФРОВЫХ СХЕМ НА VHDL

Рассмотрим некоторые примеры описания цифровых схем на VHDL

Примером описания цифрового автомата является преобразователь параллельного кода в последовательный Преобразователь кода представляет собой устройство, на вход которого подается n-битное число в параллельном коде 'd' сигнал загрузки 'load' и синхроимпульсы 'clk' По сигналу загрузки происходит запись входного слова во внутренний регистр и последовательная выдача в течении n тактов этого входного слова в последовательном коде на выходе 'o' синхроимпульсами 'oclk' После окончания преобразования на выходе 'e' появляется высокий уровень сигнала в течение одного такта Такого рода преобразователи кода часто используются для управления синтезаторами частот 1104ПЛ1 и им подобными

Описание этого устройства на языке VHDL приведено ниже

```
library ieee,
use ieee.std_logic_1164 all,

entity Serial is
    port (
          clk   . in STD_LOGIC,
          load  . in STD_LOGIC;
          reset: in STD_LOGIC;
          d     · in STD_LOGIC_vector (3 downto 0);
          oclk : out STD_LOGIC;
          o     : out STD_LOGIC;
           e     : out STD_LOGIC
          );
end;

architecture behavioral of Serial is
    type t1 is range 0 to 4,
 signal s : STD_LOGIC_vector (2 downto 0);
    signal i . t1;

begin

process (clk)
 begin
 if reset = '1' then
    i <= 0;
 else
    if (clk'event and clk='1') then
       if (i = 0 and load = '1') then
                s(2 downto 0) <= d(3 downto 1),
                o <= d(0);
                i <= 4,
       end if,
       if (i > 1) then
                o <= s(0),
                s(1 downto 0) <= s(2 downto 1),
                i <= i - 1;
       end if;
       if (i = 1) then
                e <= '1';
                i <= 0;
       else
                e <= '0',
       end if,
    end if;
 end if,
 if i>0 then
    oclk <= not clk;
 else
    oclk <= '0';
 end if;

end process;

end behavioral,
```

По переднему фронту синхроимпульса 'clk' при высоком уровне на входе загрузки происходит загрузка трех старших битов входного слова d[3 .1] во временный регистр s[2 0] Младший бит входного слова d[0] подается на выход 'o' На выходе 'oclk' появляются синхроимпульсы Сигнал 'i' выполняет функцию счетчика, выдающего сигнал окончания преобразования 'e' При поступлении последующих синхроимпульсов происходит выдача на выход 'o' остальных бит входного слова, хранящихся в регистре s[2 0]

Моделирование этого устройства было проведено в системе проектирования OrCAD 9 0

Для проверки схемы использовался тест

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity test_serial is end test_serial;
architecture testbench of test_serial is
component serial
  port (
     clk : in std_logic;
     load : in std_logic;
     reset : in std_logic;
     d : in std_logic_vector(3 downto 0);
     oclk : out std_logic;
     o : out std_logic;
     e : out std_logic
      );
end component;

signal  clk : std_logic;
signal   load : std_logic := '0',
signal   reset : std_logic;
signal   d : std_logic_vector(3 downto 0);
signal   oclk : std_logic;
signal   o : std_logic;
signal   e . std_logic;
begin
 process begin
    for i in 0 to 50 loop
       clk <= '0'; wait for 5 ns;
       clk <= '1'; wait for 5 ns;
    end loop;
 end process;
 process begin
    reset <= '1'; wait for 10 ns,
    reset <= '0' ;
    load <= '1';
    d <= "1010", wait for 10 ns,
    load <= '0',
    d <= "0000"; wait for 500 ns;
 end process;
dut .  serial  port map (
      clk => clk,
      load => load,
      reset => reset,
      d => d,
      oclk => oclk,
      o => o,
      e => e
       );
end testbench;
```

Результаты моделирования представлены на **Рис. 5.15**

В качестве примера описания устройства ЦОС рассмотрим цифровой КИХ-фильтр Работа цифрового КИХ-фильтра описывается разностным уравнением

$y_n=A_0x_n + A_1x_{n-1} + A_2x_{n-2} + . .,$

где y_n – реакция системы в момент времени n,

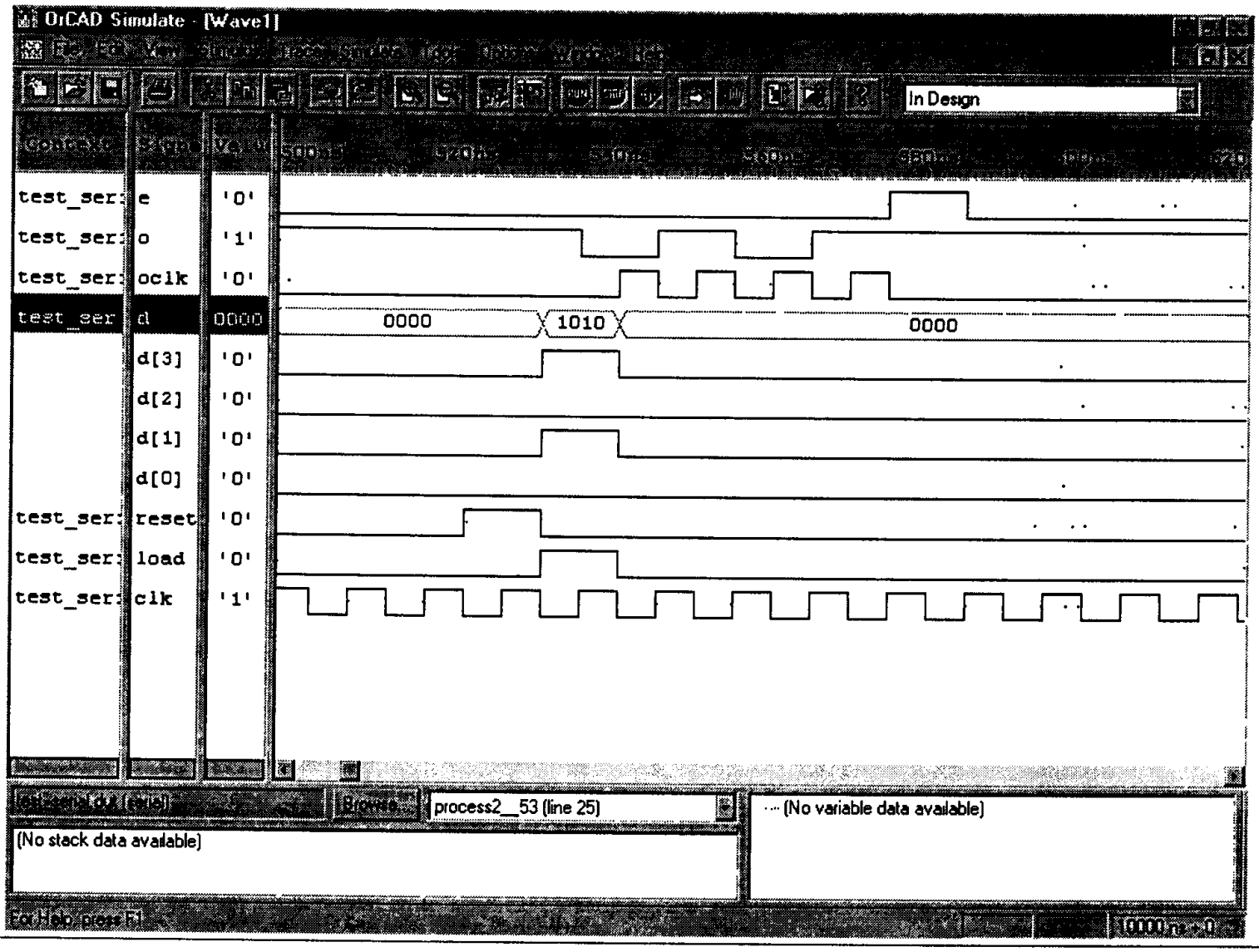
x_n – входное воздействие,

A_i – весовой коэффициент i-й входной переменной

На VHDL описание фильтра имеет вид:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity f is
port (
 din: in std_logic_vector(7 downto 0);
    sout: out std_logic_vector(15 downto 0);
 r: in std_logic;
 c: in std_logic
 );
end f;

architecture behavior of f is

  constant  h00  :  std_logic_vector(7  downto  0)  :=
"00000000";
  constant  h01  :  std_logic_vector(7  downto  0)  :=
"00000001";
  constant  h02  :  std_logic_vector(7  downto  0)  :=
"00000100";
  constant  h03  :  std_logic_vector(7  downto  0)  :=
"00001111";
  constant  h04  :  std_logic_vector(7  downto  0)  :=
"00100100";
  constant  h05  :  std_logic_vector(7  downto  0)  :=
"01000010",
  constant  h06  :  std_logic_vector(7  downto  0)  :=
"01100100";
  constant  h07  :  std_logic_vector(7  downto  0)  :=
"01111100";
  constant  h08  :  std_logic_vector(7  downto  0)  :=
"01111111";
  constant  h09  :  std_logic_vector(7  downto  0)  :=
"01101010";
  constant  h10  :  stc_logic_vector(7  downto  0)  .=
"01000010",
  constant  h11  :  std_logic_vector(7  downto  0)   =
"00010011";
  constant  h12  :  std_logic_vector(7  downto  0)   =
"11101100";
  constant  h13  .  std_logic_vector(7  downto  0)  :=
"11010110";
  constant  h14  :  std_logic_vector(7  downto  0)  :=
"11010101";
  constant  h15  :  std_logic_vector(7  downto  0)  :=
"11100011";
  constant  h16  :  std_logic_vector(7  downto  0)  :=
"11110111";
  constant  h17  :  std_logic_vector(7  downto  0)  .=
"00001010";
  constant  h18  :  std_logic_vector(7  downto  0)  ·=
"00010100";
  constant  h19  :  std_logic_vector(7  downto  0)  .=
"00010011";
  constant  h20  .  std_logic_vector(7  downto  0)   =
"00001100";
```

Рис. 5.15. Результаты моделирования преобразования кодов




```
  constant h21 : std_logic_vector(7 downto 0) :=
"00000010";
  constant h22 : std_logic_vector(7 downto 0) :=
"11111000";
  constant h23 : std_logic_vector(7 downto 0) :=
"11110101";

  signal x00, x01, x02, x03, x04, x05, x06, x07,
   x08, x09, x10, x11, x12, x13, x14, x15,
   x16, x17, x18, x19, x20, x21,
   x22, x23 : std_logic_vector(7 downto 0);
  signal m00, m01, m02, m03, m04, m05, m06, m07,
   m08, m09, m10, m11, m12, m13, m14, m15,
   m16, m17, m18, m19, m20, m21,
   m22, m23 : std_logic_vector(15 downto 0);

  begin
  m00 <= (signed(x00)*signed(h00));
  m01 <= (signed(x01)*signed(h01));
  m02 <= (signed(x02)*signed(h02));
  m03 <= (signed(x03)*signed(h03));
  m04 <= (signed(x04)*signed(h04));
  m05 <= (signed(x05)*signed(h05));
  m06 <= (signed(x06)*signed(h06));
  m07 <= (signed(x07)*signed(h07));
  m08 <= (signed(x08)*signed(h08));
  m09 <= (signed(x09)*signed(h09));
  m10 <= (signed(x10)*signed(h10));
  m11 <= (signed(x11)*signed(h11));
  m12 <= (signed(x12)*signed(h12));
  m13 <= (signed(x13)*signed(h13));
  m14 <= (signed(x14)*signed(h14));
  m15 <= (signed(x15)*signed(h15));
  m16 <= (signed(x16)*signed(h16));
  m17 <= (signed(x17)*signed(h17));
  m18 <= (signed(x18)*signed(h18));
  m19 <= (signed(x19)*signed(h19));
  m20 <= (signed(x20)*signed(h20));
  m21 <= (signed(x21)*signed(h21));
  m22 <= (signed(x22)*signed(h22));
  m23 <= (signed(x23)*signed(h23));

  sout <=
(signed(m00)+signed(m01)+signed(m02)+signed(m03)
```

```
    +signed(m04)+signed(m05)+signed(m06)+signed(m07)
    +signed(m08)+signed(m09)+signed(m10)+signed(m11)
    +signed(m12)+signed(m13)+signed(m14)+signed(m15)
    +signed(m16)+signed(m17)+signed(m18)+signed(m19)

+signed(m20)+signed(m21)+signed(m22)+signed(m23));

  process(c,r)
   begin
   if r='1' then
      x00 <= (others => '0');
      x01 <= (others => '0');
      x02 <= (others => '0');
      x03 <= (others => '0');
      x04 <= (others => '0');
      x05 <= (others => '0');
      x06 <= (others => '0');
      x07 <= (others => '0');
      x08 <= (others => '0');
      x09 <= (others => '0');
      x10 <= (others => '0');
      x11 <= (others => '0');
      x12 <= (others => '0');
      x13 <= (others => '0');
      x14 <= (others => '0');
      x15 <= (others => '0');
      x16 <= (others => '0');
      x17 <= (others => '0');
      x18 <= (others => '0');
      x19 <= (others => '0');
      x20 <= (others => '0');
      x21 <= (others => '0');
      x22 <= (others => '0');
      x23 <= (others => '0');
   elsif (c'event and c='1') then
      x00(7 downto 0) <= din(7 downto 0);
      x01(7 downto 0) <= x00(7 downto 0);
      x02(7 downto 0) <= x01(7 downto 0);
      x03(7 downto 0) <= x02(7 downto 0);
      x04(7 downto 0) <= x03(7 downto 0);
      x05(7 downto 0) <= x04(7 downto 0),
      x06(7 downto 0) <= x05(7 downto 0);
      x07(7 downto 0) <= x06(7 downto 0);
      x08(7 downto 0) <= x07(7 downto 0);
      x09(7 downto 0) <= x08(7 downto 0);
      x10(7 downto 0) <= x09(7 downto 0);
      x11(7 downto 0) <= x10(7 downto 0);
      x12(7 downto 0) <= x11(7 downto 0);
      x13(7 downto 0) <= x12(7 downto 0);
      x14(7 downto 0) <= x13(7 downto 0);
      x15(7 downto 0) <= x14(7 downto 0);
      x16(7 downto 0) <= x15(7 downto 0);
      x17(7 downto 0) <= x16(7 downto 0);
      x18(7 downto 0) <= x17(7 downto 0),
      x19(7 downto 0) <= x18(7 downto 0),
      x20(7 downto 0) <= x19(7 downto 0);
      x21(7 downto 0) <= x20(7 downto 0);
      x22(7 downto 0) <= x21(7 downto 0);
      x23(7 downto 0) <= x22(7 downto 0);
   end if;
   end process;
  end behavior;
```

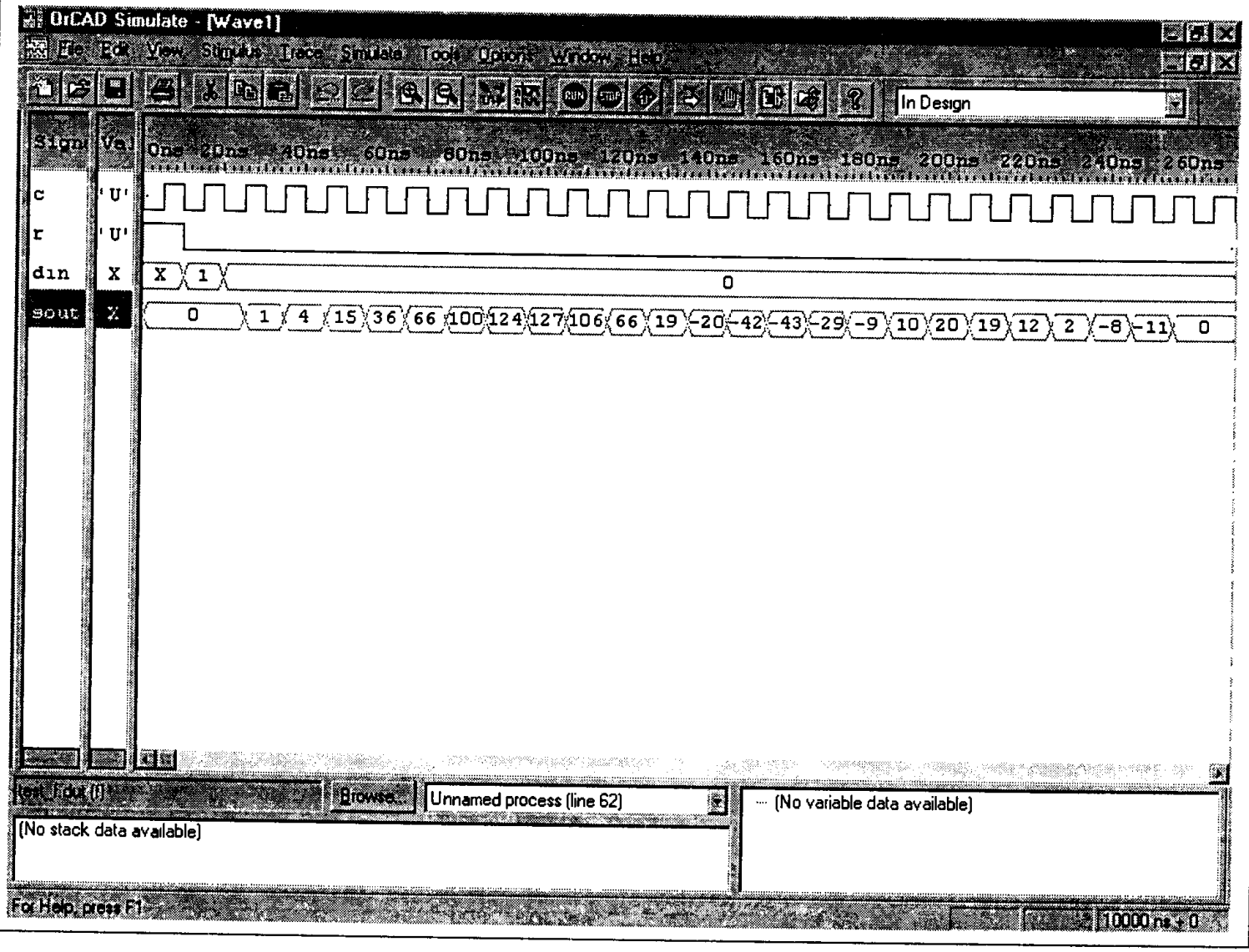
Входные данные считываются с входа din[7..0] в дополнительном коде по переднему фронту синхросигнала 'c'.

На сигналах x0- x23 построен сдвиговый регистр, обеспечивающий задержку данных на 24 такта. Сигналы с регистров умножаются на весовые коэффициенты h0-h23 и суммируются.

Для проверки схемы использован тест.

```
-- Test bench shell

library ieee;
use ieee.std_logic_1164 all;

entity test_f is end test_f;

architecture testbench of test_f is

component f
  port (
     din : in std_logic_vector(7 downto 0);
     sout : out std_logic_vector(15 downto 0);
     r : in std_logic;
     c : in std_logic
      );
end component;

signal   din : std_logic_vector(7 downto 0);
signal   sout : std_logic_vector(15 downto 0);
signal   r : std_logic,
signal   c · std_logic;

begin

 process begin
    for i in 0 to 50 loop
       c <= '0', wait for 5 ns;
       c <= '1'; wait for 5 ns;
    end loop,
 end process;

 process begin
    r <= '1'; wait for 10 ns;
    r <= '0' ;
    din <= "00000001"; wait for 10 ns;
    din <= "00000000"; wait for 500 ns;
 end process,

dut .  f  port map (
     din => din,
     sout => sout,
     r => r,
     c => c
      );
end testbench;
```

Рис. 5.16. Результаты мсделирования ЦФ

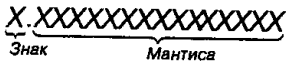


Тест моделирует подачу на цифровой фильтр аналога d-функции На выходе фильтра — его импульсная характеристика

Результаты моделирования представлены на **Рис. 5.16**

5.6. РЕАЛИЗАЦИЯ НЕЙРОНА НА AHDL

Рассмотрим реализацию на языке описания аппаратуры структуры трехвходового (количество входов легко варьируется) нейрона с логистической функцией активации (также может варьироваться) Весовые коэффициенты являются переменными, задаваемыми пользователем Входные величины (как входы, так и веса) представлены 16-разрядным кодом:



При обработке данных 16-разрядный код преобразуется к 12-разрядному коду вида

Эта операция снижает точность, но обеспечивает обработку данных, имеющих более широкий динамический диапазон

от 1.1111111 1111 + −4161536д до 0.1111111 1111 + 4161536д

Действия производятся только с целыми числами Значения весов и входов преобразуем к целому числу и округляем, например число −7 2341 может быть представлено числом −72341 В этом случае во всех остальных значениях входов и весов запятая должна быть передвинута на 4 знака вправо (для десятичного числа)

Модель нейрона описывается в виде основного тела программы, к которому подключается набор процедур, выполняющих основные действия

Рассмотрим назначение подключаемых процедур.

```
FUNCTION mult_nrn (xx[7..0], yy[7..0], a[6..0],
b[6..0], clk) RETURNS (p[14..0]);
FUNCTION sum8 (a[7..0], b[7..0], clk, cin) RETURNS
(sum[7..0], crr);
FUNCTION norm8 (a[16..1]) RETURNS (s[12..1]);
FUNCTION sum4 (a[3..0], b[3..0], clk, cin) RETURNS
(sum[3..0]);
FUNCTION norm_sum (a[12..1], b[12..1], clk) RETURNS
(as[12..1], bs[12..1]);
FUNCTION dop_kod (xx[7..0], clk) RETURNS (yy[7..0]),
```

```
FUNCTION perep (clk, xxa[11..6], xxb[4..1]) RETURNS
(yy[11..1]);
FUNCTION sigm (net[12..1]) RETURNS (out[16..1]); .
```

mult_nrn представляет собой 8-битный умножитель 7 бит данных, 1 бит — знаковый.

sum8 — 8-разрядный сумматор. **norm8** — выполняет нормализацию 16-разрядного числа к 12-разрядному При этом в 7 значащих разрядах старший бит обязательно значащий.

sum4 — 4-разрядный сумматор **norm_sum** — используется при необходимости сложения чисел с отличающейся показательной частью Данная функция приводит младшее по модулю из этих чисел к показательной части старшего числа.

dop_kod — преобразует 8-битное число из прямого кода в обратный и из обратного в прямой При этом, если входные данные находятся в дополнительном коде (что соответствует числу 0 в восьмом бите), то перевод осуществляется в прямой код и наоборот.

perep — функция, необходимая при переполнении разрядной сетки при выполнении действия функцией **sum8** При этом получаем единичный разряд в переменной crr Рассматриваемая функция осуществляет сдвиг на разряд вправо значащей части и прибавление единицы к показательной.

sigm — логистическая функция. Представляет функцию активации нейрона в виде сигмоиды out = 2 / (1+exp (–0 07net)) – 1. Сигмоида сжата в пределах net = (–142,142), out = int ([0,1) x 10000) На вход подается нормализованное число к виду 12 бит — знаковый (+1, –0), 11.. 5 бит — мантиса (если экспоненциальная часть не нулевая, то 11 бит = VCC), 4 .1 бит — экспоненциальная часть На выходе имеем число, вида 16 бит — знаковый (+1, –0), 15 1 бит — мантиса

Приведем функции, используемые в подпрограммах

```
FUNCTION mult16b (xx, yy, a, b) RETURNS (s, c);
FUNCTION dop_kod1 (xx[3..0], clk) RETURNS
(yy[3..0]); .
```

mult16b — задействует процедуру **MULT1B.** Является базовым одноразрядным умножителем, на котором происходит построение умножителей большей размерности

dop_kod1 — аналог функции **dop_kod**, но для 4-разрядных чисел (используется для показательной части)

Ниже приводиться текст основной программы Neiron

```
FUNCTION mult_nrn (xx[7..0], yy[7 .0], a[6 .0],
b[6..0], clk) RETURNS (p[14..0]),
FUNCTION sum8 (a[7..0], b[7..0], clk, cin) RETURNS
(sum[7. 0], crr),
FUNCTION norm8 (a[16..1]) RETURNS (s[12..1]);
FUNCTION sum4 (a[3..0], b[3..0], clk, cin) RETURNS
(sum[3..0]);
FUNCTION norm_sum (a[12..1], b[12..1], clk) RETURNS
(as[12..1], bs[12..1]);
FUNCTION dop_kod (xx[7..0] clk) RETURNS (yy[7..0]);
FUNCTION perep (clk, xxa[11..6], xxb[4..1]) RETURNS
(yy[11..1]);
FUNCTION sigm (net[12..1]) RETURNS (out[16..1]);

SUBDESIGN neiron_
%Последние биты в процедурах умножения во входных
переменных и в ответе являются знаковыми%
%Реализуется трехвходовый однослойный нейрон %
(
clk:INPUT;

%n[3..0]:INPUT;%%Число входов нейрона 2-10%

w[16..1]:INPUT;%Веса%
ww[16..1]:INPUT;
www[16..1]:INPUT;

x_[16..1]:INPUT;%Входы%
xx[16..1]:INPUT;
xxx[16..1].INPUT;

z[12..1]:OUTPUT;%Выход%
zz[16..1]:OUTPUT;

%Тестовые выходы%
o[12..1]:OUTPUT;%Умножение%
oo[12..1]:OUTPUT;
ooo[12..1]:OUTPUT;

jas[12..1]:OUTPUT;%Выравнивание перед сложением%
jbs[12..1]:OUTPUT;
jjas[12..1]:OUTPUT,
jjbs[12..1]:OUTPUT;

f[12..1].OUTPUT;%Проверка суммы%
f_[11..5]:OUTPUT; %Проверка суммы (без переноса)%

ka[12..5]:OUTPUT;%Переменные для проверки работы с
доп. кодом и вычитания с ним%
kb[11..5]:OUTPUT;
kabs[12..5]:OUTPUT;

)
VARIABLE
rx[3..1]:norm8;%Переменные нормализации входов%
rw[3..1]:norm8;%Переменные нормализации весов%
rs[3..1]:norm8;%Переменные нормализации промежуточ-
ных результатов%

s:mult_nrn;%Переменные умножения%
ss.mult_nrn;
sss:mult_nrn;

e[3..1]:sum4;%Переменные сложения показателей при
умножении%
ee[3..1]:sum4;

q[2..1]:sum8;%Переменные сложения%
%qq.sum8;%

m[12..1]:NODE;%Вспомогательные переменные суммирова-
ния%
mm[12..1]:NODE;

t[2..1]:norm_sum;%Переменные приведения к одному по-
казателю перед сложением%
```

```
  d[4..1]:dop_kod;%Переменные переведения слагаемого в
дополнительный код%

  p[2..1]:perep;%Переменная, переносящая переполнение
разрядной сетки в показатель%

  sg:sigm;%Переменная сигмоиды%

  BEGIN
  %Нормализация входов%
     rx[1].a[16..1]=x_[16..1];
     rx[2].a[16..1]=xx[16..1];
     rx[3].a[16..1]=xxx[16..1];

  %Нормализация весов%
     rw[1].a[16..1]=w[16..1];
     rw[2].a[16..1]=ww[16..1];
     rw[3].a[16..1]=www[16..1];
  %----------%

     %Умножение входов на веса%
  %Примечание:  числа  были  приведены  к  виду
12.11.10.9.8.7.6.5.4.3.2.1    %
  %                                  знак числа|значе-
ние числа |показатель степени двойки - сдвиг%
  %Числа перемножаются, показатели складываются%
     s.xx[7..0]=rx[1].s[12..5];
     s.yy[7..0]=rw[1].s[12..5];
     s.a[6..0]=GND;
     s.b[6..0]=GND;
     s.clk=clk;
                  e[1].a[3..0]=rx[1].s[4..1];
                  e[1].b[3..0]=rw[1].s[4..1];
                  e[1].clk=clk;
                  e[1].cin=GND;
  %Проводим нормализацию промежуточных результатов%
     rs[1].a[16]=s.p[14]; %Знак%
     rs[1].a[15]=GND;
     rs[1].a[14..1]=s.p[13..0];
             %Добавление показаталя после нормализа-
ции%
                  ee[1].a[3..0]=e[1].sum[3..0];
                  ee[1].b[3..0]=rs[1].s[4..1];
                  ee[1].clk=clk;
                  ee[1].cin=GND;

  %----------%
     ss.xx[7..0]=rx[2].s[12..5];
     ss.yy[7..0]=rw[2].s[12..5];
     ss.a[6..0]=GND;
     ss.b[6..0]=GND;
     ss.clk=clk;
                e[2].a[3..0]=rx[2].s[4..1];
                e[2].b[3..0]=rw[2].s[4..1];
                e[2].clk=clk;
                e[2].cin=GND;
  %Проводим нормализацию промежуточных результатов%
     rs[2].a[16]=ss.p[14]; %Знак%
     rs[2].a[15]=GND;
     rs[2].a[14..1]=ss.p[13..0];
             %Добавление показаателя после нормалиаа-
ции%
                  ee[2].a[3..0]=e[2].sum[3..0];
                  ee[2].b[3..0]=rs[2].s[4..1];
                  ee[2].clk=clk;
                  ee[2].cin=GND;


  %----------%

     sss.xx[7..0]=rx[3].s[12..5];
     sss.yy[7..0]=rw[3].s[12..5];
     sss.a[6..0]=GND;
     sss.b[6..0]=GND;
     sss.clk=clk;
                  e[3].a[3..0]=rx[3].s[4..1];
                  e[3].b[3..0]=rw[3].s[4..1];
                  e[3].clk=clk;
                  e[3].cin=GND;
  %Проводим нормализацию промежуточных реаультатов%
     rs[3].a[16]=sss.p[14]; %Знак%
     rs[3].a[15]=GND;
     rs[3].a[14..1]=sss.p[13..0];
            %Добавление показателя после нормалиаации%
                  ee[3].a[3..0]=e[3].sum[3..0];
                  ee[3].b[3..0]=rs[3].s[4..1];
                  ee[3].clk=clk;
                  ee[3].cin=GND;


  %----------%
  %тест%
  %демонстрируют правильность выполнения умножения
входов на веса и нормализации%
  o[12..5]=rs[1].s[12..5]; o[4..1]=ee[1].sum[3..0];
  oo[12..5]=rs[2].s[12..5]; oo[4..1]=ee[2].sum[3..0];
  o o o [ 1 2 . . 5 ] = r s [ 3 ] . s [ 1 2 . . 5 ] ;
ooo[4..1]=ee[3].sum[3..0];

  %----------%


     %Общий сумматор%
     %во всех 3 случаях 14 бит - анаковый%
  %Для корректного сложения необходимо равенство пока-
зательных частей. Приводим показательные
  часть двух слагаемых к показательной части старшго
из них.%

     %Производим выравнивание по показательным частям%
                        t[1].a[12..5]=rs[1].s[12..5];
t[1].a[4..1]=ee[1].sum[3..0];
                        t[1].b[12..5]=rs[2].s[12..5];
t[1].b[4..1]=ee[2].sum[3..0];
     t[1].clk=clk;
```

```
  %------------%
  %тест%
  %демонстрируют правильность выполнения выравнивания
значений перед сложением%
  jas[12..1]=t[1].as[12..1];
  jbs[12..1]=t[1].bs[12..1];
  %------------%


     %Произведем проверку знака (0 соответствует (-)
и доп. коду) и запрос к дополнительному коду%
     %Перевод необходим лишь при различных знаках%
      If ((t[1].as[12]==GND)&(t[1].bs[12]==VCC))==VCC
then
       d[1].clk=clk;
       d[1].xx[7]=GND; d[1].xx[6..0]=t[1].as[11..5];

  %------------%
  %тест%
       ka[12..5]=t[1].bs[12..5];
       kb[11..5]=d[1].yy[6..0];
  %------------%


       %Суммируем%
       q[1].clk=clk;
       q[1].cin=GND;
       q[1].a[7..0]=t[1].bs[12..5];
       q[1].b[7]=GND; q[1].b[6..0]=d[1].yy[6..0];

  %------------%
  %тест%
       kabs[12..5]=q[1].sum[7..0];
  %------------%


       %Проверка на получение отрицательного результа-
та в доп. коде%
        %If (q[1].sum[7]==GND) then???% %То перевод из
доп. кода и постановка знака - (0)%
       If (q[1].sum[7]==VCC) then
         d[2].clk=clk;
         d[2].xx[7]=GND; d[2].xx[6 .0]=q[1].sum[6..0];

         m[12]=GND; m[11..5]=d[2].yy[6..0];
       Else
         m[12]=VCC;m[11..5]=q[1].sum[6..0];
       End If;
    m[4..1]=t[1].as[4..1];%Показательная часть при вы-
читании не изменяется. М.б. t[1].bs[4..1]%


     %Перевод необходим лишь при различных знаках%
                                                ElsIf
((t[1].as[12]==VCC)&(t[1].bs[12]==GND))==VCC then
       d[1].clk=clk;
       d[1].xx[7]=GND; d[1].xx[6..0]=t[1].bs[11..5];

  %------------%
  %тест%
       ka[12..5]=t[1].as[12..5];
       kb[11..5]=d[1].yy[6..0];
  %------------%


       %Суммируем%
       q[1].clk=clk;
       q[1].cin=GND;
       q[1].a[7..0]=t[1].as[12..5];
       q[1].b[7]=GND; q[1].b[6..0]=d[1].yy[6..0];

  %------------%
  %тест%
       kabs[12..5]=q[1].sum[7..0];
  %------------%


       %Проверка на получение отрицательного результа-
та в доп. коде%
        %If (q[1].sum[7]==GND) then???% %То перевод из
доп. кода и постановка знака - (0)%
       If (q[1].sum[7]==VCC) then
         d[2].clk=clk;
         d[2].xx[7]=GND; d[2].xx[6..0]=q[1].sum[6..0];

         m[12]=GND; m[11..5]=d[2].yy[6..0];
       Else
         m[12]=VCC;m[11..5]=q[1].sum[6..0];
       End If;
    m[4..1]=t[1].as[4..1];%Показательная часть при вы-
читании не изменяется. М.б. t[1].bs[4..1]%


     Else %В случае двух положительных или двух отри-
цательных слагаемых%
       %Суммируем%
       q[1].clk=clk;
       q[1].cin=GND;
       q[1].a[6 .0]=t[1] bs[11 .5];%Восьмой бит - зна-
ковый%
       q[1].b[6..0]=t[1].as[11..5];

  %------------%
  %тест%
  %демонстрируют правильность выполнения сложения (без
переносов)%
  f_[11..5]=q[1].sum[6..0];
  %------------%


       %Проверка на переполнение разрядной сетки уме-
стна лишь при сложении чисел одного знака%
      If (q[1].sum[7]==VCC) then
        p[1].clk=clk;
        p[1].xxa[11..6]=q[1].sum[6..1];
```

```
            p[1].xxb[4..1]=t[1].bs[4..1];%можно и
t[1].as[4..1]%

          m[12]=t[1].bs[12];%можно и t[1].as[12] - их
знак одинаков%
          m[11..1]=p[1].yy[11 .1];
        Else
          m[12]=t[1].bs[12];%можно и t[1].as[12] - их
знак одинаков%
          m[11..5]=q[1].sum[6..0];
          m[4..1]=t[1].bs[4..1],%можно и t[1].as[4..1]%

        End If;
      End If;


  %-----------%
  %тест%
  %демонстрируют правильность выполнения сложения%
  f[12..1]=m[12..1],
  %-----------%



  %-----------%

     %Производим выравнивание по показательным частям%
     t[2].a[12..5]=m[12..5]; t[2].a[4..1]=m[4..1];
                          t[2].b[12..5]=rs[3].s[12..5],
t[2].b[4..1]=ee[3].sum[3..0];
     t[2].clk=clk;

  %-----------%
  %тест%
  %демонстрируют правильность выполнения выравнивания
значений перед сложением%
  jjas[12..1]=t[2].as[12..1];
  jjbs[12..1]=t[2].bs[12..1];
  %-----------%



     %Произведем проверку знака (0 соответствует (-)
и доп. коду) и запрос к дополнительному коду%
     %Перевод необходим лишь при различных знаках%
      If ((t[2].as[12]==GND)&(t[2].bs[12]==VCC))==VCC
then
        d[3].clk=clk;
        d[3].xx[7]=GND, d[3].xx[6..0]=t[2] as[11..5];



        %Суммируем%
        q[2].clk=clk;
        q[2].cin=GND;
        q[2].a[7..0]=t[2].bs[12..5];
        q[2].b[7]=GND, q[2].b[6..0]=d[3].yy[6..0],

        %Проверка на получение отрицательного результа-
та в доп. коде%

        %If (q[2].sum[7]==GND) then???% %То перевод из
доп. кода и постановка знака - (0)%
        If (q[2].sum[7]==VCC) then
          d[4].clk=clk;
          d[4].xx[7]=GND; d[4].xx[6..0]=q[2].sum[6..0];

          mm[12]=GND; mm[11..5]=d[4].yy[6..0];
        Else
          mm[12]=VCC; mm[11..5]=q[2].sum[6..0];
        End If;
      mm[4..1]=t[2].as[4..1];%Показательная часть при
вычитании не изменяется. М.б. t[2].bs[4..1]%


     %Перевод необходим лишь при различных знаках%
                                                  ElsIf
((t[2].as[12]==VCC)&(t[2] bs[12]==GND))==VCC then
        d[3].clk=clk;
        d[3].xx[7]=GND; d[3] xx[6..0]=t[2].bs[11..5];

        %Суммируем%
        q[2].clk=clk;
        q[2].cin=GND;
        q[2] a[7..0]=t[2].as[12..5];
        q[2].b[7]=GND; q[2].b[6..0]=d[3].yy[6..0];

        %Проверка на получение отрицательного результа-
та в доп. коде%
        %If (q[2].sum[7]==GND) then???% %То перевод из
доп. кода и постановка знака - (0)%
        If (q[2].sum[7]==VCC) then
          d[4].clk=clk;
          d[4].xx[7]=GND; d[4].xx[6..0]=q[2].sum[6..0];

          mm[12]=GND; mm[11..5]=d[4].yy[6..0];
        Else
          mm[12]=VCC; mm[11..5]=q[2].sum[6..0];
        End If;
      mm[4..1]=t[2].as[4..1];%Показательная часть при
вычитании не изменяется. М.б. t[2].bs[4..1]%


     Else %В случае двух положительных или двух отри-
цательных слагаемых%
        %Суммируем%
        q[2].clk=clk;
        q[2].cin=GND;
        q[2].a[6..0]=t[2].bs[11..5],%Восьмой бит - зна-
ковый%
        q[2].b[6..0]=t[2].as[11..5];

        %Проверка на переполнение разрядной сетки уме-
стна лишь при сложении чисел одного знака%
        If (q[2].sum[7]==VCC) then
          p[2].clk=clk,
          p[2].xxa[11..6]=q[2].sum[6..1];
               p[2].xxb[4..1]=t[2].bs[4..1];%можно и
t[1].as[4..1]%
```

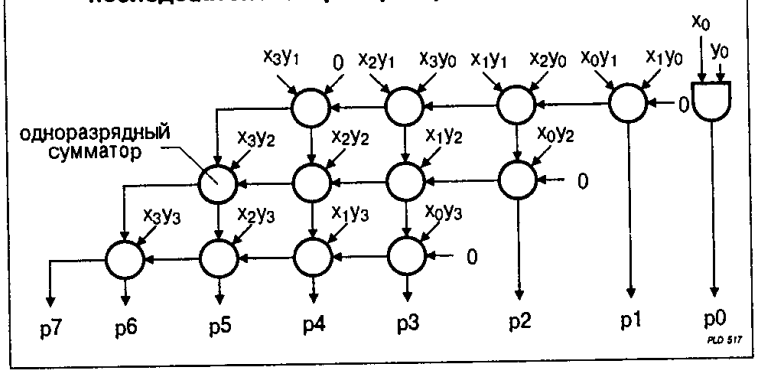
```
        mm[12]=t[2].bs[12];%можно и t[1].as[12] - их
знак одинаков%
        mm[11..1]=p[2].yy[11..1];
      Else
        mm[12]=t[2].bs[12];%можно и t[1].as[12] - их
знак одинаков%
        mm[11..5]=q[2].sum[6..0];
                  mm[4..1]=t[2].bs[4..1];%можно  и
t[1].as[4..1]%

      End If;
    End If;

    sg.net[12..1]=mm[12..1];
    zz[16..1]=sg.out[16..1];


    z[12..1]=mm[12..1];
  END
```

5.7. ПОСТРОЕНИЕ БЫСТРОДЕЙСТВУЮЩИХ ПЕРЕМНОЖИТЕЛЕЙ

Реализация операции умножения аппаратными методами всегда являлась сложной задачей при разработке высокопроизводительных вычислителей Аппаратная реализация алгоритма умножения в первую очередь предназначена для получения максимальной скорости выполнения этой операции в устройстве На ПЛИС можно разрабатывать умножители с быстродействием более 100 МГц и располагать более 100 модулей умножителей в одном кристалле

Для операндов небольшой разрядности (4 и менее) наиболее результативна структура простого матричного суммирования (**Рис. 5.17**) Она реализует параллельный умножитель как массив одноразрядных сумматоров, соединенных локальными межсоединениями, при этом общее число сумматоров напрямую определяется разрядностью множимого и множителя

Рис. 5.17. Структура матричного умножиталя с последоаательным распространением переноса



Так, полный параллельный умножитель 4 x 4 требует для своей реализации 12 сумматоров При дальнейшем наращивании разрядности матрица одноразрядных сумматоров значительно разрастается, одновременно увеличивается критический путь, и реализация умножителя становится нерациональной

Одним из способов уменьшения аппаратных затрат служит использование алгоритма Бута В соответствии с данным алгоритмом определенным образом анализируются парные биты множителя и в зависимости от их комбинации над множимым выполняются некоторые преобразования (пример для умножителя 12 x 12 бит приведен в **Таблице 5.3**)

Таблица 5.3. Преобразование множимого в зависимости от состояния множителя

Пары бит множителя 0-1, 2-3, 4-5, 6-7, 8-9	Выполняемое действие
00	Прибавить 0
01	Прибавить множимое
10	Прибавить (2 x множимое)
11	Прибавить (3 x множимое)
Пара бит множителя 10-11 (знак)	**Выполняемое действие**
00	Прибавить 0
01	Прибавить множимое
10	Прибавить – (2 x множимое)
11	Прибавить – множимое

Как видно из таблицы, выполняемые над множимым действия отличаются для двух старших бит, один из которых знаковый Подобная коррекция позволяет осуществлять операцию умножения над числами в дополнительном коде

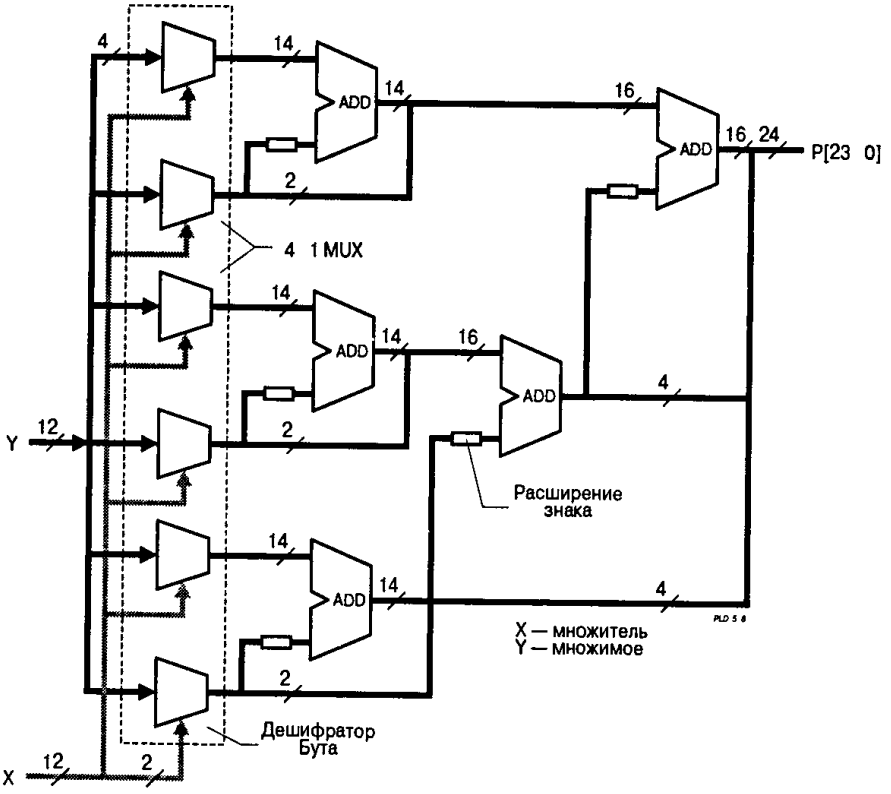
Сформированные частичные произведения, представляющие собой результат преобразования множимого, необходимо суммировать с соответствующим весовым фактором Результатом их суммирования по алгоритму простого матричного умножителя и будет конечный продукт произведения двух операндов Анализ различных методов свертки частичных произведений показал, что наиболее приемлемым для реализации на ПЛИС является алгоритм на основе иерархического дерева многоразрядных масштабирующих сумматоров При этом достигается сокращение аппаратных затрат по сравнению с параллельным матричным умножителем до двух раз

Пример реализации данного решения для случая 12-разрядных операндов приведен на **Рис. 5.18** В данной схеме выбор соответствующего преобразования множимого Y осуществляется мультиплексором на основании анализа парных бит множителя X в соответствии с **Таблицай 5.3** Входные значения мультиплексоров 0 и Y подаются непосредственно, значение 2Y формируется сдвигом множимого на один бит, а 3Y формируется путем сложения Y и 2Y

Сумматоры ADD осуществляют суммирование частичных произведений, причем их разрядность для разных ступеней иерархии неодинакова На каждом шаге суммирования, как видно, имеются несколько сквозных бит, непосредственно дополняющих выходные значения сумматоров. За счет них и за счет битов расширения знака осуществляется необходимое масштабирование результата Размерность произведения P в конечном итоге равна сумме размерностей операндов

За счет регулярности приведенной структуры быстродействие можно резко повысить путем введения конвейеризации каждой ступени При этом разделяются конвейерными регистрами как сумматоры, так и мультиплексоры Выигрыш в быстродействии, благодаря данному решению, оказывается особенно ощутимым для большой разрядности операндов

Рис. 5.18. Структурная схема 12-разрядного умнсжителя на ссновс алгсритма Бута



Для иллюстрации метода умножения на константу рассмотрим, как выполняется умножение двух десятичных чисел "в столбик" на примере 85 x 37:

```
   85                                  85
 x  7                                x 37
   35                                595
+ 560                             + 2550
  595  ──► 7 x 85          ──►     3145
           3 x 85 x 10
```

Как видно из примера, используется каждая цифра множителя для поразрядного перемножения со всеми цифрами множимого и совсем необязательно знать всю таблицу произведений операндов полной разрядности. Для перемножения двух переменных достаточно иметь таблицу произведений множимого, в данном случае константы, на весь ранг возможных цифр множителя, и осуществить корректное суммирование полученных частичных произведений. Для случая 85 x 476 это иллюстрируется следующим образом

```
0 x 85 =  000
1 x 85 =  085
2 x 85 =  170
3 x 85 =  255
4 x 85 =  340 ──►          85
5 x 85 =  425          x  476
6 x 85 =  510 ──►         510
7 x 85 =  595 ──►        5950
8 x 85 =  680          + 34000
9 x 85 =  765           40460
```

Таблица произведений множимого записывается во фрагмент памяти на ПЛИС и называется таблицей перекодировок (ТП) ТП адресуются четырьмя разрядами, следовательно, необходимо представить операнды в шестнадцатеричном коде, и массив ТП будет содержать, таким образом, массив произведений константы на ряд цифр 0, 1, .. , E, F

0 x 55 = 000	8 x 55 = 2A8
1 x 55 = 055	9 x 55 = 2FD
2 x 55 = 0AA	A x 55 = 352
3 x 55 = 0FF	B x 55 = 3A7
4 x 55 = 154	C x 55 = 3FC
5 x 55 = 1A9	D x 55 = 451
6 x 55 = 1FE	E x 55 = 4A6
7 x 55 = 253	F x 55 = 4FB

Как видно из приведенного примера, для того чтобы осуществить операцию умножения двух двенадцатиразрядных двоичных чисел, одно из которых — константа, необходимо иметь ТП на 16 значений с разрядностью выходного значения 16 (для положительных чисел) и два 16-разрядных сумматора — **Рис. 5.19.**

Четыре младших бита выходного произведения нижней на **Рис. 5.19** ТП непосредственно в соответствии с алгоритмом функционирования формируют четыре младших разряда окончательного произведения

При подаче на сумматор указанного произведения необходимо произвести расширение знака в соответствии со значением старшего бита. Например, для 12-разрядных операндов:

Младшее произведение (x 1) 1111 1110 0110 1101

↓ *Сквозное суммирование*

Старшее произведение (x 16) +0010 1101 0011

Результат 0010 1011 1001 1101

Реализация по данному принципу операции умножения отрицательных чисел, представленных в дополнительном коде, не требует каких-либо изменений в технической реализации и не задействует дополнительную специальную логику коррекции; изменится лишь прошивка просмотровой таблицы, адресуемой знаковым битом входного операнда.

Рис. 5.19. 12-разрядный умнсжиталь на константу