ФГБОУ ВПО «Воронежский государственный
технический университет»

А.В. Строгонов

ПРОЕКТИРОВАНИЕ УСТРОЙСТВ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ ДЛЯ РЕАЛИЗАЦИИ В БАЗИСЕ ПРОГРАММИРУЕМЫХ ЛОГИЧЕСКИХ ИНТЕГРАЛЬНЫХ СХЕМ

Утверждено Редакционно-издательским советом
университета в качестве учебного пособия

Воронеж 2013

УДК 621.3.049.77

Строгонов А.В. Проектирование устройств цифровой обработки сигналов для реализации в базисе программируемых логических интегральных схем: учеб. пособие / А.В. Строгонов. Воронеж: ФГБОУ ВПО «Воронежский государственный технический университет», 2013. 323 с.

В учебном пособии рассматривается проектирование цифровых устройств для реализации в базисе ПЛИС. Даются практические примеры проектирования цифровых фильтров с использованием высокоуровневого языка описания аппаратурных средств VHDL и мегафункций в САПР ПЛИС Quartus II компании Altera. Уделено внимание вопросам проектирования цифровых автоматов и функциональных узлов микропроцессорных устройств для реализации в базисе ПЛИС с использованием системы визуально-имитационного моделирования Matlab/Simulink.

Издание соответствует требованиям Федерального государственного образовательного стандарта высшего профессионального образования по направлению подготовки бакалавров 210100 «Электроника и наноэлектроника», профилю «Микроэлектроника и твердотельная электроника», дисциплинам «Проектирование БИС», «Проектирование ПЛИС», «Проектирование цифровых устройств в базисе ПЛИС». Учебное пособие подготовлено в электронном виде в текстовом редакторе MS Word for Windows и содержится в файле Проектирование устройств ЦОС в базисе ПЛИС.doc.

Табл. 18. Ил. 141. Библиогр.: 21 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);
д-р техн. наук, проф. М.И. Горлов

ВВ ИЕ

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.

ПЛИС широко используются в качестве интерфейсных схем, в микропроцессорных системах для организации обмена и стыковки различных ИС между собой и устройствами ввода-вывода. В базисе ПЛИС могут быть спроектированы логические блоки и системы, преобразователи кодов, периферийные контроллеры, микропрограммные устройства управления, конечные автоматы, умножители, микропроцессорные ядра и др.

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 преобразователи двоичного кода в двоично-десятичный и двоично-десятичного в двоичный на ПЛИС с использованием высокоуровневого языка описания аппаратных средств VHDL.

Во второй главе рассматривается моделирование КИХ-фильтра в системе Matlab/Simulink (пакет Signal Processing, среда FDATool), проектирование КИХ-фильтров на последовательной и параллельной арифметиках с использованием операций умножения с накоплением (MAC), демонстрируются различные варианты реализации КИХ-фильтров с использованием перемножителей на мегафункциях САПР Quartus II компании Altera.

В третьей главе приводятся сведения по проектированию цифровых автоматов Мура, Мили по диаграммам переходов. Подробно рассматривается метод кодирования с одним активным состоянием, а также методы и приемы (стили кодирования цифровых автоматов на языке VHDL), позволяющие повысить эффективность использования ресурсов ПЛИС.

В четвертой главе рассмотрены различные подходы в проектировании микропроцессорных ядер для реализации в базисе ПЛИС с использованием системы визуально-имитационного моделирования Matlab/Simulink с приложениями StateFlow и Simulink HDL Coder. Микропроцессорные ядра, представленные в виде сложно-функциональных блоков в базисе ПЛИС, позволяют реализовать современную концепцию “система на кристалле”. Использование более высокой степени абстракции в проектировании БИС и сложно-функциональных блоков в виде готовых модулей позволяют создавать конкурентоспособные изделия в кратчайшие сроки.

1. ПРОЕКТИРОВАНИЕ КОМБИНАЦИОННЫХ И ПОСЛЕДОВАТЕЛЬНОСТНЫХ УСТРОЙСТВ 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 это не удобно 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)-разрядная. 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 $X_{10} \leq 0$, 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 5 разрядов. Рассмотрим отрицательное число -13_{10}, считая представление шестиразрядным, включая знак. В пятиразрядном представлении $\left|-13_{10}\right|=13_{10}=01101_2$ и $2^5-1_{10}=31_{10}=11111_2$ то

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

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 $X_{10}\geq 0$, 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 $2^{n-1}-\left|X_{10}\right|$

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 и 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:

$$\begin{array}{lll} -3 & 101 & (-2^2 + 2^0) \\ 3 & 011 & (2^1 + 2^0) \end{array},$$

в четырехразрядном представлении:

$$\begin{array}{ll} 1101 & (-2^3 + 2^2 + 2^0) \\ 0011 & (2^1 + 2^0) \end{array}$$

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 в дальнейшем воспользуемся при проектировании масштабирующего аккумулятора цифрового фильтра.

1.2. Преобразователи кодов на ПЛИС

Преобразователем кода называется логическая схема, которая изменяет данные, представленные в одном двоичном виде, в другой вид, также двоичный. Преобразование двоичного кода (ДК) в двоично-десятичный (ДДК) и ДДК в ДК может быть выполнено как на аппаратном уровне с использованием ИС средней степени интеграции, в т.ч. ИС ПЗУ, или на ПЛИС с применением мегафункций ИС 74xx серии и высокоуровневых языков описания аппаратных средств HDL, так и программным способом на языках программирования микроконтроллеров. Каждый из способов преобразований имеет свои преимущества

и недостатки. Цель статьи рассмотреть азы преобразований на основе различных схемных решений.

Для представления любой десятичной цифры 0, 1, …, 9 достаточно использовать два символа 0 и 1. На практике применяется 4-разрядный код 8-4-2-1 (двоично-десятичный код, ДДК). Числа 8, 4, 2 и 1 являются весами разрядов ДДК. Например, запись десятичной цифры в коде 8-4-2-1 совпадает с записью двоичных чисел от 0 до 9 (например, 0101 в двоичном коде (ДК) соответствует 5), а n- разрядное десятичное число (ДЧ) представляется с помощью тетрад, каждая из которых состоит из четырех двоичных разрядов (например, 975 - 100101110101). Одно 4-разрядное двоичное число позволяет представить десятичные числа от 0 до 15. Для записи двоично-десятичного числа требуется больше разрядов, чем для записи двоичного. Не предусмотренные ДДК цифры от 10 до 15 называются псевдотетрадами.

Преобразователи двоично-десятичного кода в двоичный код.

Преобразование ДДК в ДК можно сделать путем последовательного деления десятичного числа на 2. Если оно нечетное, то в остатке получится 1, т.е. в разряде 2^0 записывается 1. Затем частное от деления еще раз делится на 2, и, если остаток равен нулю, в разряде 2^1 записывается 0. Если остаток равен 1, то в этом разряде записывается 1. Аналогично получают и более старшие разряды двоичного числа.

Преобразование ДДК в ДК выполняется с помощью операции сдвига числа в сторону младших разрядов и коррекции числа, получаемого после сдвига. Сдвиг двоичного числа на один разряд в сторону младших разрядов (на один разряд вправо) эквивалентен делению числа на два без учета младшего разряда, который теряется или поступает в другой сдвигающий регистр. При сдвиге двоично-десятичного числа на один разряд вправо получаемое число не равно исходному, деленному на два. Чтобы в результате сдвига получалось такое

число, необходимо производить коррекцию результата сдвига (табл.1.2).

Таблица 1.2

Фрагмент таблицы переключений для преобразований ДДК в ДК (сдвиг вправо, деление на 2, в предположении, что в старший разряд сдвигается 0)

ДЧ	ДДК	Получается при сдвиге ДЧ вправо на один разряд в ДДК-сетке	Требуемый результат, для выполнения деления на 2	Коррекция
0	00000	00000 (0)	00000 (0)	нет
2	00010	00001 (1)	00001 (1)	нет
4	00100	00010 (2)	00010 (2)	нет
6	00110	00011 (3)	00011 (3)	нет
8	01000	00100 (4)	00100 (4)	нет
10	10000	01000 (8)	00101 (5)	-3
12	10010	01001 (9)	00110 (6)	-3
14	10100	01010 (10)	00111 (7)	-3
16	10110	01011 (11)	01000 (8)	-3
18	11000	01100 (12)	01001 (9)	-3

На рис.1.1 показан способ преобразования ДДК в ДК. В строке изображены все три преобразователя (К-коррекция). Границы тетрад сдвигаются справа на лево. Если старший разряд корректирующего элемента не используется, то коррекция не нужна (такие элементы изображены пунктирной линией). Пример 1 демонстрирует преобразование 3-разрядного ДДК в 10-разрядный ДК на языке VHDL.

Создается вспомогательный 21-разрядный вектор z, содержимого которого перед началом преобразования обнуляется. Разряды с 19 по 8 отводятся под ДДК-входы (вектор P) а разряды с 7 по 0 под ДК-выходы (вектор B). Но с учетом того, что получаемое двоичное число больше на два разряда, чем вектор B, то под выходы ДК-сетки отводится 10 разрядов вектора z (B <= z(9 downto 0)). Старший 20-разряд необходим для корректного сдвига вправо.

Разряды вектора z с 19 по 16 используются под входы элементарного преобразователя –“Сотни”, с 15 по 12 под “Десятки”, с 11 по 8 под “Единицы” ДДК-сетки. В процессе преобразования требуется 8 операций сдвига вправо вектора z, из них 7 в теле цикла оператора loop. В теле оператора цикла осуществляется коррекция разрядов ДДК-сетки, в случае если на входы элементарного преобразователя поступает число больше или равное 8.

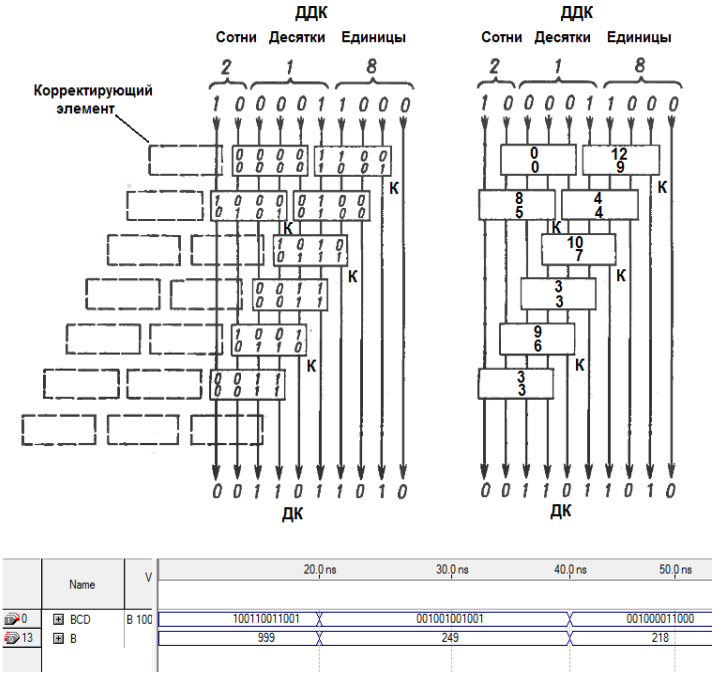


Рис.1.1. Способ преобразования ДДК в ДК

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity bcdtobin is
   port (
      P: in STD_LOGIC_VECTOR (11 downto 0);
      B: out STD_LOGIC_VECTOR (9 downto 0));
end bcdtobin;
```

```
architecture a of bcdtobin is
begin
  process(P)
  variable z: STD_LOGIC_VECTOR (20 downto 0);
   begin
     for i in 0 to 20 loop
   z(i) := '0';
     end loop;
  z(19 downto 8) := P;
  z(19 downto 7):= z(20 downto 8);
     for i in 0 to 6 loop
   if z(11 downto 8) >=8 then
      z(11 downto 8) := z(11 downto 8) - 3;
   end if;
    if z(15 downto 12) >=8 then
      z(15 downto 12) := z(15 downto 12) - 3;
   end if;
   z(19 downto 0):= z(20 downto 1);
   end loop;
B <= z(9 downto 0);
  end process;
end a;
```

Пример 1. Преобразование 3-разрядного ДДК в 10-разрядный ДК

Рассмотрим элементарный преобразователь кодов (КС – коррекция, сдвиг) на сумматорах. Пусть элементарный преобразователь имеет четыре входа и четыре выхода. Операция сдвига реализуется подачей на три входа трех старших разрядов j-й тетрады, а на четвертый вход – первого разряда j+1 –й тетрады.

На вход преобразователя поступают двоичные 4-разрядные числа $X = (x_4, x_3, x_2, x_1)$. Числа $X = 5,6,7,13,14,15$ не могут поступать на вход преобразователя. Минимальное и максимальное число j-й тетрады лежит в диапазоне от 0 до 9: $A_{\min} = (0,0,0,0)$ и $A_{\min} = (1,0,0,01)$. В случае непоступления единицы из младшего разряда j+1-й тетрады:

$x_4 = 0$ то $X_{min} = (0,0,0,0)$ и $X_{max} = (0,1,0,0) = 4(Д)$.

В случае поступления единицы из младшего разряда j+1-й тетрады: $x_4 = 1$ то $X_{min} = (1,0,0,0) = 8(Д)$ и $X_{max} = (1,1,0,0) = 12(Д)$. Таким образом преобразователь кода выполняет функцию:

$$Y = f(X) = \begin{cases} X, & 0 \le X \le 4, \\ X - 3, & 8 \le X \le 12 \end{cases}, \qquad (1)$$

где Y – двоичное число, получаемое на выходе преобразователя кода.

На рис.1.2, *а* показан элементарный преобразователь кодов. В левом и правом дополнительных полях указаны веса, с которыми воспринимаются и выдаются входные и выходные сигналы. Вес старшего входного разряда преобразователя кодов на три меньше, чем вес выходного старшего разряда.

На рис.1.2, *б* показана схема преобразователя кодов КС, выполненная на сумматоре. В случае поступления единицы из младшего разряда j+1-й тетрады $x_4 = 1$, то от числа $X = (x_4, x_3, x_2, x_1)$ следует отнять число 3, что эквивалентно сложению числа $X = (0, x_3, x_2, x_1)$ с числом 5, которое является дополнением числа 3 до 8, например, сложение десятичного числа 2 с десятичным числом 5 эквивалентно вычитанию из 10 числа 3.

Преобразователь 3-разрядного двоично-десятичного числа 975 в двоичный код показан на рис.1.3. Максимальное 3-разрядное десятичное число равно 999, поэтому максимальный вес старшего двоичного разряда будет равен $2^9 = 512$. Если на вход КС с весом 5 подается ноль, то КС не может изменять входных сигналов.

На входы подается десятичное число 975, которое представляется в ДДК как 100101110101, а с выхода снимается двоичное число 111001111 (D-десятичное число, К-коррекция).

Преобразователь имеет пирамидальную структуру. Так как самый младший разряд ДДК совпадает с младшим разрядом

ДК, то этот разряд не преобразуется, т.е. подается с входа на выход. Следующие по старшинству разряды ДДК подаются со сдвигом на входы двух КС (производится сдвиг на один разряд). Второй сдвиг на один разряд осуществляется с помощью следующих двух КС и т.д. Веса разрядов входных сигналов всех КС находятся в отношении 1:2:4:5, т.к. каждый КС преобразует только один двоично-десятичный разряд в двоичный разряд (вес 5 изменяется на вес 8). Пирамида строится из КС до тех пор, пока не будут получены выходные сигналы со всеми весами 2^p, где $p = 0,\ \ 1,\ \ 2\ldots$ при условии, что полученное двоичное число не меньше исходного двоично-десятичного.

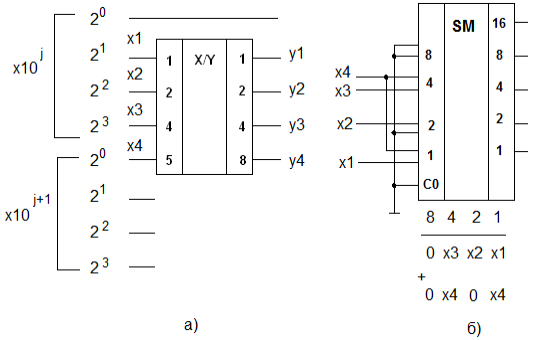


Рис.1.2. Элементарный преобразователь кодов ДДК в ДК (а) и преобразователь кодов на сумматоре (б)

Преобразователи ДДК в ДК могут быть построены с использованием сумматоров на ИС типа SN7483 (рис.1.4). Рассмотрим преобразователь двузначного числа представленного в ДДК в 7-разрядное двоичное число. Данный преобразователь просто и экономично выполняется двумя 4-разрядными сумматорами.

Необходимые соединения определяются выражением каждого из весомозначных двоично-десятичных разрядов через числа, являющиеся разными степенями 2:

$$80 = 64 + 16 = 2^6 + 2^4$$
$$40 = 32 + 8 = 2^5 + 2^3$$
$$10 = 8 + 2 = 2^3 + 2^1$$

Располагая двоично-десятичные и двоичные числа в упорядочные ряды (табл.1.3) видно, какие из двоично-десятичных входов должны быть, просуммированы для получения различных двоичных выходов. Например, выход 2^0 соответствует младшему значащему разряду двоично-десятичного знака единиц, для получения выхода 2^1 необходимо просуммировать входы с весами 2 и 10. Сумма 2^3 имеет более двух входов (веса 8, 10 и 40) поэтому не может быть реализована одиночным каскадом сумматора. Для выхода 2^3 сумма частично образуется в первом сумматоре и завершается во втором (рис.1.4).

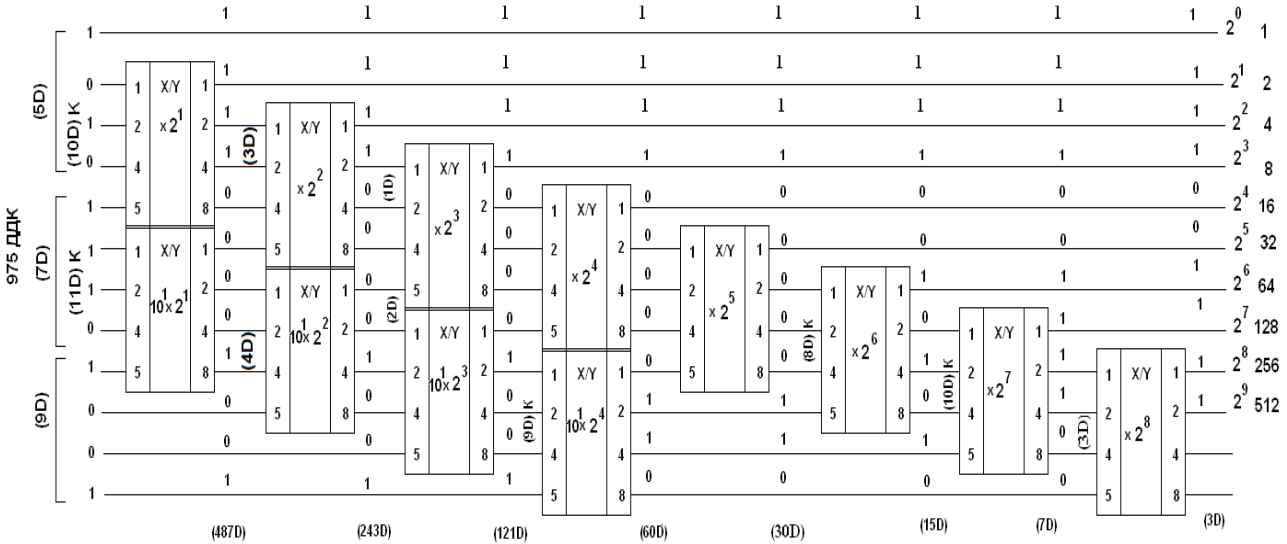


Рис.1.3. Преобразователь 3-разрядного двоично-десятичного числа в двоичный код

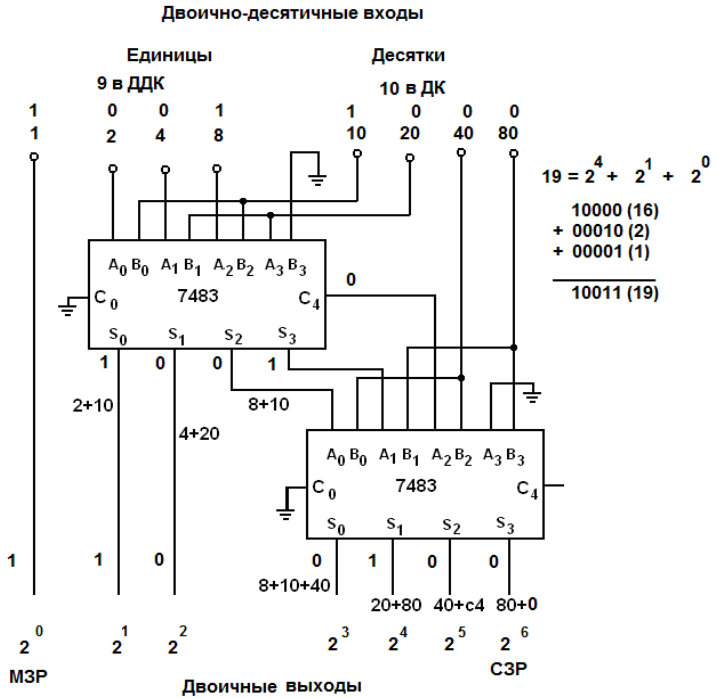


Рис.1.4. Преобразование десятичного числа 19 представленного ДДК в ДК на сумматорах ИС типа SN7483

Таблица 1.3

Двоично-десятичное преобразование в двоичное на сумматорах

Входы (десятичный вес)	Двоичные выходы						
	2^0	2^1	2^2	2^3	2^4	2^5	2^6
	1	2	4	8	16	32	64
(1)	1						
(2)		1					
(4)			1				
(8)				1			
(10)		1		1			
(20)			1		1		
(40)				1		1	
(80)					1		1

Схемотехническое решение с использованием сумматоров на ИС средней степени интеграции может представлять сложную структуру. Более эффективным является использование ПЗУ или ПЛИС.

ИС типа К155ПР6 (зарубежный функциональный аналог ИС типа SN74181) и К155ПР7 – одинаковые кристаллы ПЗУ с программами взаимного преобразования ДДК и ДК выполненные по ТТЛ-технологии. Организация кристалла ПЗУ 32 х 8 бит, дешифратор адресов – 5 –разрядный (входы A, B, C, D, E). ИС ПР6 по адресам A, B, C, D, E принимает ДДК с весом разрядов 1-2-4-5-10 и генерирует ДК. При G=L преобразование разрешено, при G=H запрещено, на выходах Y1…Y5 – H. Шестиразрядный преобразователь ПР6 принимает ДДК и имеет вес МЗДР (младшие значащие десятичные разряды): 1,2,4,5 и старших СЗДР (старшие значащие десятичные разряды): 10 и 20. Входной байт у ПР6 6-разрядный. Младший разряд $2^0 = 1$ можно давать на прямую. Код с большим числом разрядов получается каскадированием ПР6. Вес ДДК старших ИС надо увеличить на декаду.

Преобразователь ПР6 выполняет следующую функцию:

$$Y = f(X) = \begin{cases} X, & 0 \le X \le 4, \\ X - 3, & 8 \le X \le 12, \\ X - 6, & 16 \le X \le 20, \\ X - 9, & 24 \le X \le 28, \end{cases} \qquad (2)$$

где $X = (x_5, x_4, x_3, x_2, x_1)$, $Y = (y_5, y_4, y_3, y_2, y_1)$. Значения $X = 5, 6, 7, 13, 14, 15, 21, 22, 23, 29, 30, 31$ не могут появляться на входах преобразователя кодов.

Преобразователи двоичного кода в двоично-десятичный код. Для преобразования десятичного числа представленного в двоичном коде (ДК) в двоично-десятичный код (ДДК), двоичное число, начиная со старшего разряда, вдвигается справа налево в двоично-десятичную разрядную

сетку (рис.1.5). Когда какая-либо единица пересекает границу между двоично-десятичными разрядами, возникает ошибка. Например, при сдвиге двоичного числа 1000 (8) влево, разрядное значение 1 увеличивается с 8 до 16 (10000), тогда как для двоично-десятичного числа оно возрастает с 8 до 10 (вес 8 увеличивается до веса 10). Поэтому двоично-десятичное число уменьшается как бы на 6. Следовательно, для коррекции необходимо прибавлять 6 к числу во всех случаях, когда единица пересекает границу между двоично-десятичными разрядами.

К числу десятков надо прибавить 6, если единица перейдет в разряд сотен, и т.д. Составленное таким образом двоично-десятичное число имеет правильное значение, однако оно может содержать псевдотетрады. Чтобы этого не было, возникающие псевдотетрады корректируют непосредственно после каждого шага сдвига, прибавляя 6 к соответствующей декаде с переносом 1 в следующую.

На практике перед сдвигом прибавляют 3 а не 6. Необходимость корреции определяют перед сдвигом. Если значение тетрады меньше или равно 4 (0100), то при последующем сдвиге не произойдет перехода единицы через границу между декадами и не возникнут псевдотетрады. Если значение тетрады перед сдвигом 5 (0101), 6 (0110) или 7 (0111), то также не произойдет перехода 1 через границу, поскольку старший разряд равен 0.

Однако при этом возникнут псевдотетрады: 10, 12, 14 или 11, 13, 15 в зависимости от того, будет ли в младший разряд сдвинут 0 или 1. Следовательно, в этих случаях необходима коррекция псевдотетрад путем прибавления 3 перед сдвигом (табл.1.4). Для значений тетрад 8 и 9 так же необходима коррекция.

Таблица 1.4

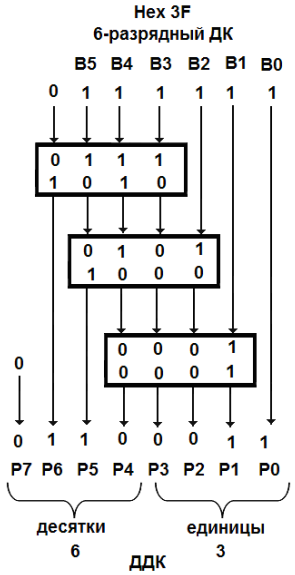
Таблица преобразований ДК в ДДК методом сдвига числа влево (умножение на 2, в предположении, что в младший разряд сдвигается 0)

Вход (ДЧ)	ДДК-сетка	После сдвига влево (псевдотетрады)	Требуемое значение, с учетом коррекции и сдвига влево	Коррекция исходного значения перед сдвигом
0	00000 (0 ДК)	00000 (0 ДК)	00000 (0 ДК)	нет
1	00001 (1 ДК)	00001 (1 ДК)	00001 (1 ДК)	нет
2	00010 (2 ДК)	00100 (4 ДК)	00100 (4 ДК)	нет
3	00011 (3 ДК)	00110 (6 ДК)	00110 (6 ДК)	нет
4	00100 (4 ДК)	01000 (8 ДК)	01000 (8 ДК)	нет
5	00101 (5 ДК)	01010 (10 ДК)	1000 (10 ДДК)	+3
6	0110 (6 ДК)	01100 (12 ДК)	10010 (12 ДДК)	+3
7	0111 (7 ДК)	01110 (14 ДК)	10100 (14 ДДК)	+3
8	01000 (8 ДК)	10000 (10 ДДК)	10110 (16 ДДК)	+3
9	01001 (9 ДК)	10010 (12 ДДК)	11000 (18 ДДК)	+3

Рассмотрим преобразователь ДК в ДДК. Такой преобразователь должен выполнять функцию, обратную функции (1), т.е. при $X \geq 5$, надо производить сложение числа $X = (x_4, x_3, x_2, x_1)$ с числом 3. Таким образом, преобразователь выполняет функцию (3):

$$Y = f(X) = \begin{cases} X, & 0 \le X \le 4, \\ X+3, & 5 \le X \le 9 \end{cases}. \qquad (3)$$

На рис.1.5 показан способ преобразования ДК в ДДК с помощью комбинационной схемы. Вместо сдвига числа справа налево здесь слева направо сдвигаются границы двоично-десятичных разрядов (по принципу каскадирования К155ПР7), а каждая полученная тетрада корректируется в соответствии с функцией 3. На вход преобразователей нельзя подавать двоичные числа 10…15, т.к. они превышают сумму весов выходных сигналов 5+4+2+1=12.



Операция	ДДК-сетка		ДК
	Десятки	Единицы	
Двоичные входы, B			5 4 3 2 1 0
HEX			3 F
Старт			1 1 1 1 1 1
Сдвиг 1		1	1 1 1 1 1
Сдвиг 2		1 1	1 1 1 1
Сдвиг 3		1 1 1	1 1 1
Коррекция, + 3		1 0 1 0	1 1 1
Сдвиг 4	1	0 1 0 1	1 1
Коррекция, + 3	1	1 0 0 0	1 1
Сдвиг 5	1 1	0 0 0 1	1
Сдвиг 6	1 1 0	0 0 1 1	
BCD	6	3	
ДДК-выходы	7 4	3 0	
z	13 10	9 6	5 0

Рис.1.5. Преобразование ДК в ДДК методом сдвига границ двоично-десятичных разрядов с лева направо с последующей коррекцией тетрад, на примере десятичного числа 63 и таблица преобразований для написания VHDL-кода

Ниже приведен код языка VHDL для 6-разрядного преобразователя ДК в ДДК (пример 2). Создается

вспомогательный 13-разрядный вектор z, содержимого которого перед началом преобразования обнуляется. Семь старших разрядов которого с 12 по 6 отводятся под ДДК-выходы (вектор P) а разряды с 5 по 0 под ДК-входы (вектор B). Далее осуществляется сдвиг вектора B на три разряда влево z(8 downto 3) := B. Разряды вектора z с 9 по 6 используются под входы элементарного преобразователя. Согласно рис.5 требуется три сдвига границ двоично-десятичных разрядов (Сдвиг 4, 5 и 6), поэтому в теле оператора цикла loop осуществляется проверка выполнения условия z(9 downto 6) > 4 и если это так, то содержимое должно быть увеличено на 3. Затем, в цикле, осуществляется сдвиг вектора z на 1 разряд в лево z(12 downto 1) := z(11 downto 0).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity binbcd6 is
    port (
        B: in STD_LOGIC_VECTOR (5 downto 0);
        P: out STD_LOGIC_VECTOR (6 downto 0));
end binbcd6;
architecture binbcd6_arch of binbcd6 is
begin
  bcd1: process(B)
  variable z: STD_LOGIC_VECTOR (12 downto 0);
   begin
     for i in 0 to 12 loop
   z(i) := '0';
     end loop;
  z(8 downto 3) := B;
     for i in 0 to 2 loop
   if z(9 downto 6) > 4 then
      z(9 downto 6) := z(9 downto 6) + 3;
   end if;
   z(12 downto 1) := z(11 downto 0);
   end loop;
P <= z(12 downto 6);
```

```
  end process bcd1;
end binbcd6_arch;
```

Пример.2 Код языка VHDL для 6-разрядного преобразователя ДК в 2-разрядный ДДК

Регистры на ИС типа SN74195 могут быть использованы для разработки схемы поразрядного преобразователя двоичного кода в двоично-десятичный (рис.1.6). Для построения преобразователя для каждой конечной двоично-десятичной цифры так же потребуются один 4-разрядный сумматор с последовательным переносом ИС типа SN7483 и один инвертор. На рис.1.6 обозначено: МЗДР – младший значащий десятичный разряд; СЗДР – старший значащий десятичный разряд.

Двоичное слово, начиная со старшего значащего разряда, вводится в сдвиговый регистр, состоящий из нескольких соединенных последовательно ИС 74195. Каждый сдвиг удваивает содержимое регистров, выраженное в двоично-десятичном коде. Поэтому требуется коррекция всякий раз, когда любой из 4-разрядных регистров содержит число больше чем четыре, которое при сдвиге вырабатывает неправильный код. Эта коррекция выполняется добавлением трех к содержимому регистра и введением суммы в параллельные входы данных, сдвигая на один раз вниз.

Если содержимое регистра больше 4-х, то вместо 3-х прибавляем десятичное число 11 (1011) к содержимому регистра с помощью 4-разрядного сумматора. Если перенос в старший разряд С4 на выходе сумматора равен 1, то осуществляется коррекция содержимого регистра путем параллельной загрузки частичных сумм S0, S1, S2 4-разрядного сумматора на входы P1, P2, P3 регистра и следующего сдвигаемого бита с входов J и K на вход P0. Старший значащий разряд S3 игнорируется.

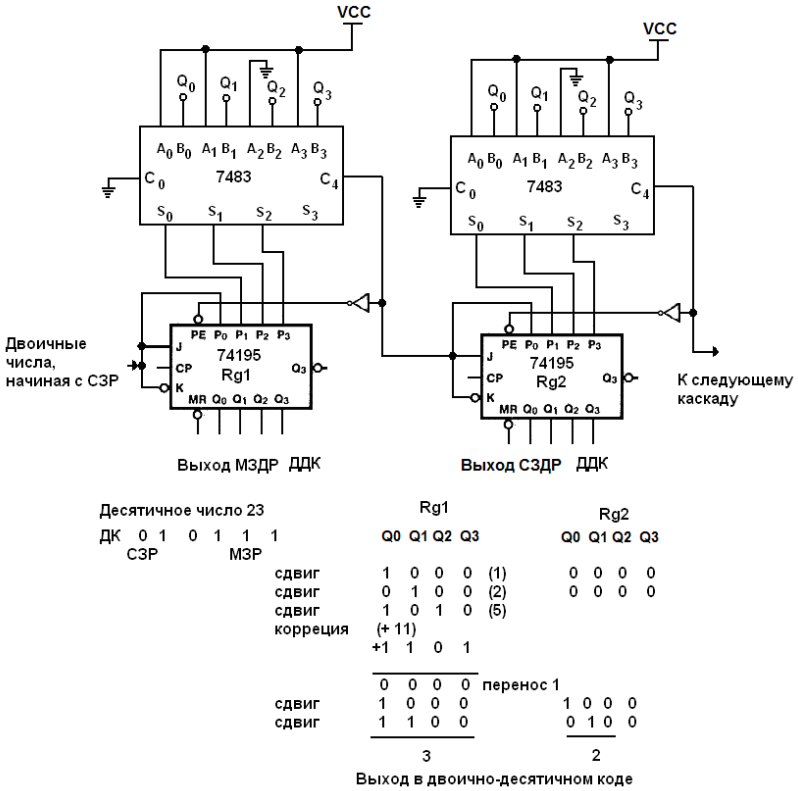


Рис.1.6. Поразрядный преобразователь двоичного кода в двоично-десятичный на ИС типа SN74195

Двоичное число полностью преобразуется, когда введен его МЗР. Сдвиговый регистр должен быть достаточно длинным, чтобы вместить двоично-десятичный результат, который всегда длиннее, чем двоичное число. Данная схема может быть использована для любого числа разрядов и цифр.

На рис.1.7 показан поразрядный преобразователь на макрофункциях 74195 (функциональный аналог ИС типа SN74195) в САПР ПЛИС Quartus II описанный выше. В качестве сумматоров используются макрофункции 7483 (ИС типа SN7483). В дальнейшем макрофункции САПР ПЛИС

Quartus II Altera 74 серии будем отождествлять с ИС 74 серии. Рис.1.8 демонстрирует процесс преобразования ДЧ 23 представленного ДК в ДДК (старшими разрядами ДК вперед).

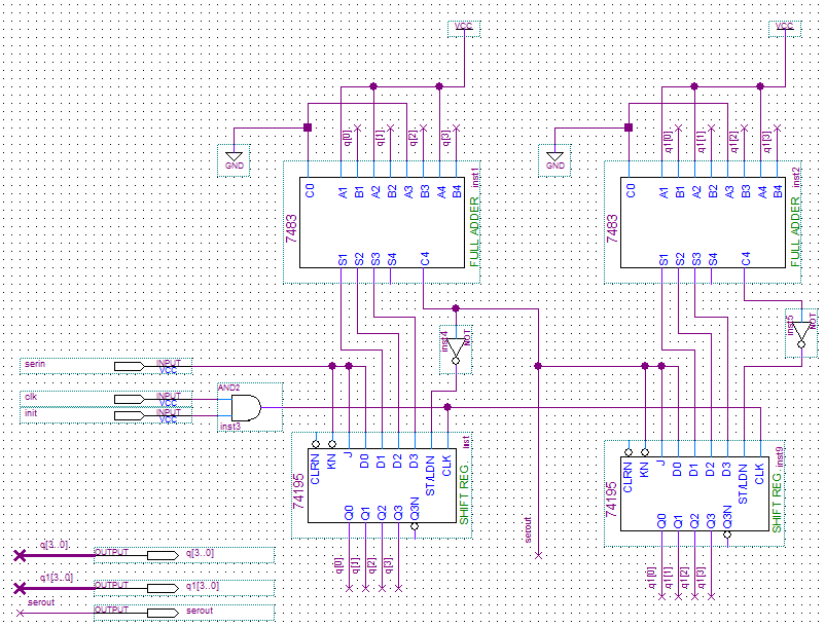


Рис.1.7. Поразрядный преобразователь ДК в ДДК на ИС типа SN74195 в САПР ПЛИС Quartus II

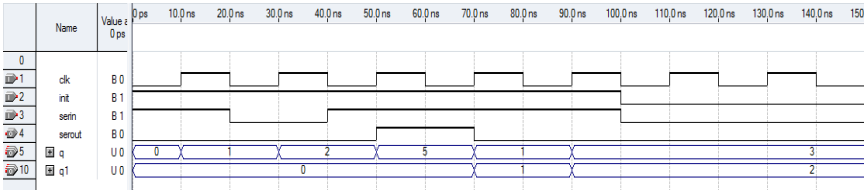


Рис.1.8. Процесс преобразования ДЧ 23 представленного ДК в ДДК (старшими разрядами ДК вперед)

Существует и другой способ поразрядного преобразования ДК в ДДК удобный для реализации на ПЛИС, коррекция в котором осуществляется после сдвига. В табл.1.4

показан последовательный сдвиг на один разряд влево десятичных чисел от 0 до 9, подаваемых на вход преобразователя по двоично-десятичной сетке. Каждый сдвиг влево удваивает содержимое четырех разрядного регистра. Следовательно, при сдвиге десятичного числа 5 в двоично-десятичной сетке должно получиться число 10 в ДДК (10000 или 16 в ДК) а вместо этого получается число 10 в ДК, поэтому необходима коррекция всякий раз, когда содержимое регистра более пяти. 5 заменяется на 0, 6 на 2, 7 на 4, 8 на 6 и 9 на 8, т.е. тройка перед сдвигом не прибавляется (табл.1.5).

Таблица 1.5

Таблица переключений преобразователя ДК в ДДК (сдвиг влево, умножение на 2)

Вход (ДЧ)	ДДК		Результат в ДДК	Неправильное значение в ДДК-сетке (псевдотетрады)
	Дес.	Един.		
0	0	0	0	
1	0	2 (0010)	2	
2	0	4 (0100)	4	
3		6 (0110)	6	
4		8 (1000)	8	
5	1	0	10 (10000, 16 ДК)	01010 (10 ДК)
6	1	2	12 (10010, 18 ДК)	01100 (12 ДК)
7	1	4	14 (10100, 20 ДК)	01110 (14 ДК)
8	1	6	16 (10110, 22 ДК)	10000 (16 ДК)
9	1	8	18 (11000, 24 ДК)	10010 (18 ДК)

На рис.1.9 показан разряд преобразователя ДК в ДДК методом последовательного сдвига старшего разряда ДК в лево с последующей коррекцией. На рис.1.10 показан двухразрядный преобразователь. Для реализации такого

преобразователя на ПЛИС потребуется 3 конфигурируемых логических блока ПЛИС серии XC3000 на каждый разряд ДДК. На рис.1.11 показан разряд преобразователя ДК в ДДК методом последовательного сдвига старшего разряда ДК влево с последующей коррекцией на ИС средней степени интеграции 74 серии, в основе которого лежит схема, показанная на рис.1.9. Используется компаратор ИС типа 7485 и сдвиговый регистр ИС типа 74195. На рис.1.12 представлен двухразрядный преобразователь ДК в ДДК, а на рис.1.13 процесс преобразования десятичного числа 23 представленного ДК в ДДК.

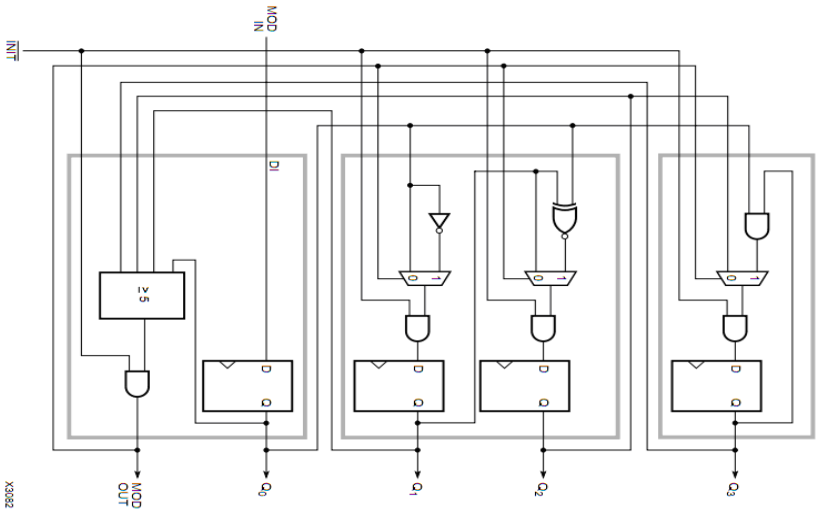


Рис.1.9. Разряд преобразователя ДК в ДДК методом последовательного сдвига старшего разряда ДК в лево с последующей коррекцией

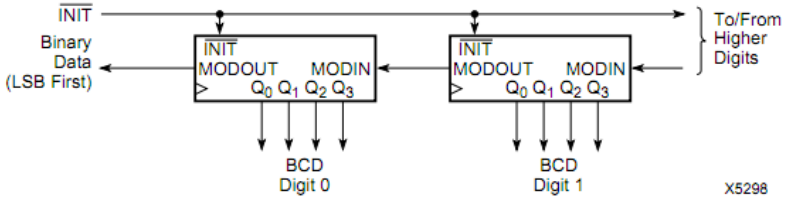


Рис.1.10. Двухразрядный преобразователь ДК в ДДК методом последовательного сдвига

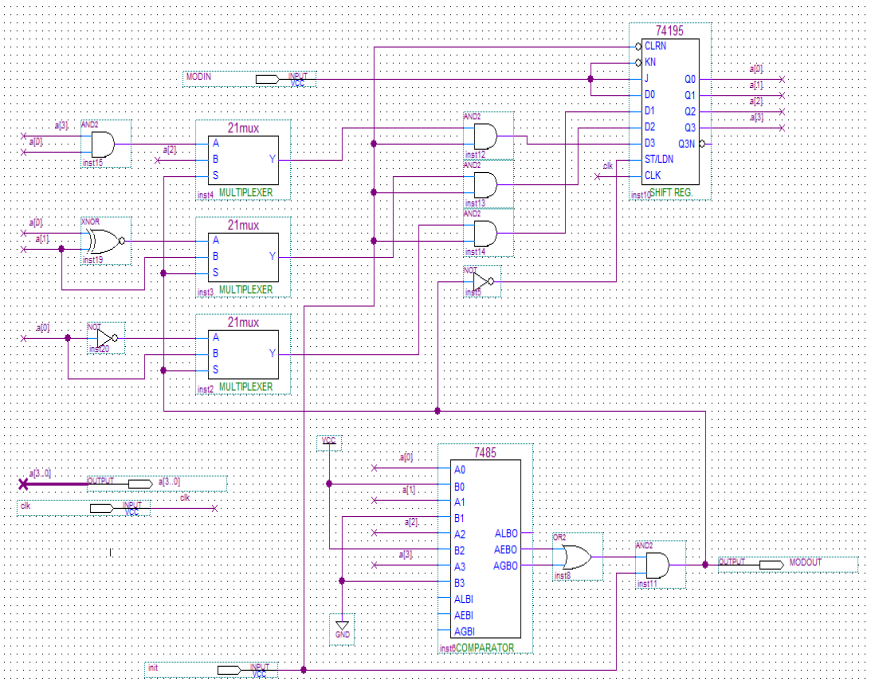


Рис.1.11. Разряд преобразователя ДК в ДДК методом последовательного сдвига старшего разряда ДК влево с последующей коррекцией на ИС средней степени интеграции 74 серии в САПР ПЛИС Quartus II

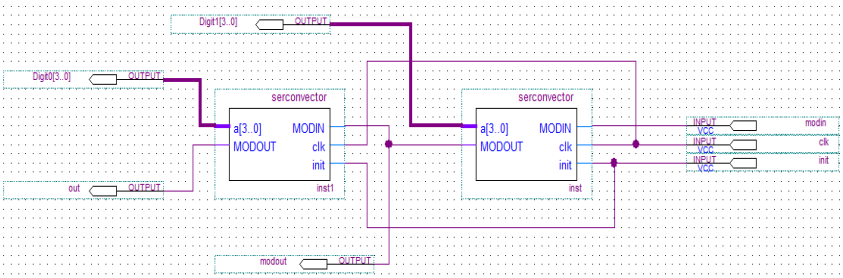


Рис.1.12. Двухразрядный преобразователь ДК в ДДК методом последовательного сдвига

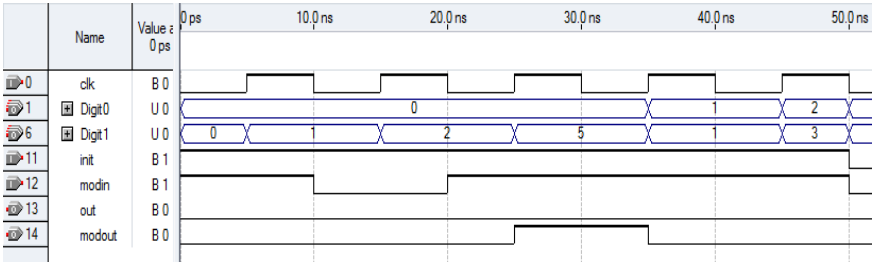


Рис.1.13. Преобразование десятичного числа 23 представленного ДК в ДДК

В данной главе рассмотрено представление чисел в прямом, обратном и дополнительном кодах и арифметические действия над ними.

Показано, что многоразрядные преобразователи кодов могут быть выполнены на ИС средней степени интеграции или в базисе ПЛИС с учетом или без их архитектурных особенностей с использованием высокоуровневого языка описания аппаратных средств VHDL.

2. ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ ФИЛЬТРОВ В БАЗИСЕ ПЛИС

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

Рассмотрим особенности расчета КИХ-фильтра в системе Matlab/Simulink (пакет Signal Processing, среда FDATool) и с использованием мегафункции Mega Core FIR Compiler САПР ПЛИС Quartus. Цель примера спроектировать два одинаковых по характеристикам КИХ-фильтра в среде FDATool и с использованием мегафункции Mega Core FIR Compiler. Главным достоинством среды FDATool от других программ расчета параметров КИХ-фильтров является возможность генерации кода языка VHDL с использованием Simulink HDL Coder. Однако полученный код языка VHDL в САПР ПЛИС Quartus непосредственно использовать нельзя из-за особенностей компилятора-синтезатора. Сгенерированный в автоматическом режиме код языка VHDL может быть использован в симуляторе 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 которой 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:

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

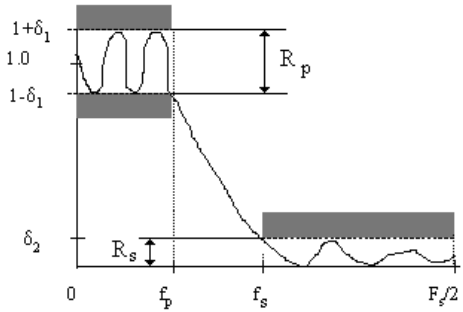


Рис.2.1. Амплитудно-частотная характеристика фильтра нижних частот

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 осуществляется 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).

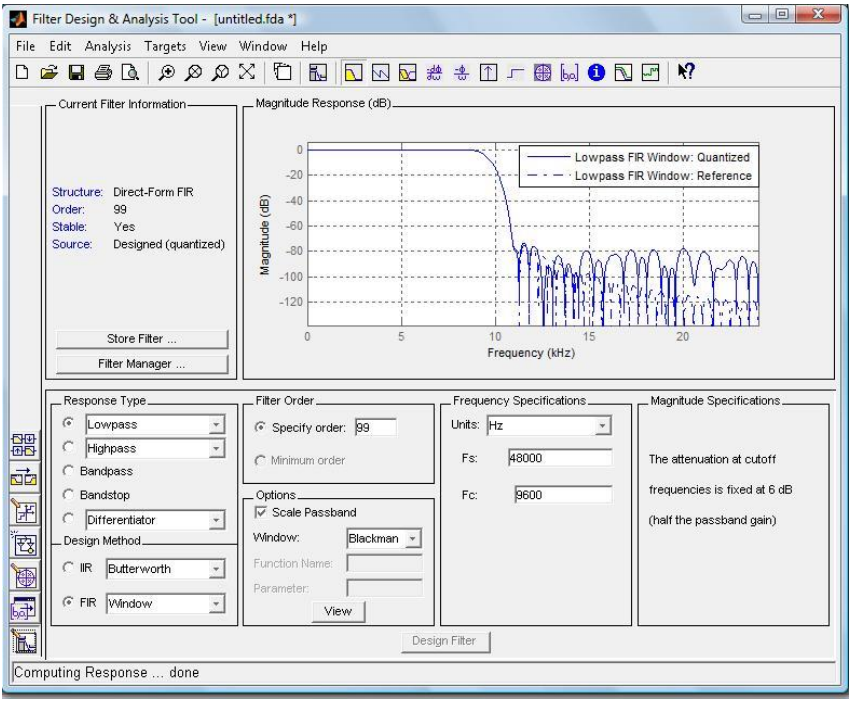


Рис.2.2. Интерфейс среды FDATool. Пример расчета АЧХ КИХ-фильтра

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.

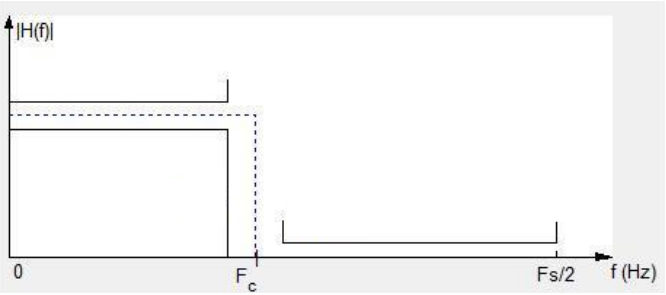


Рис.2.3. Характеристики синтезируемой АЧХ (окно Blackman) КИХ-фильтра в среде FDATool

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 позволяет сгенерировать код языка VHDL фильтра (рис.2.5). Выберем параллельную архитектуру КИХ-фильтра, обладающей высокой производительностью.

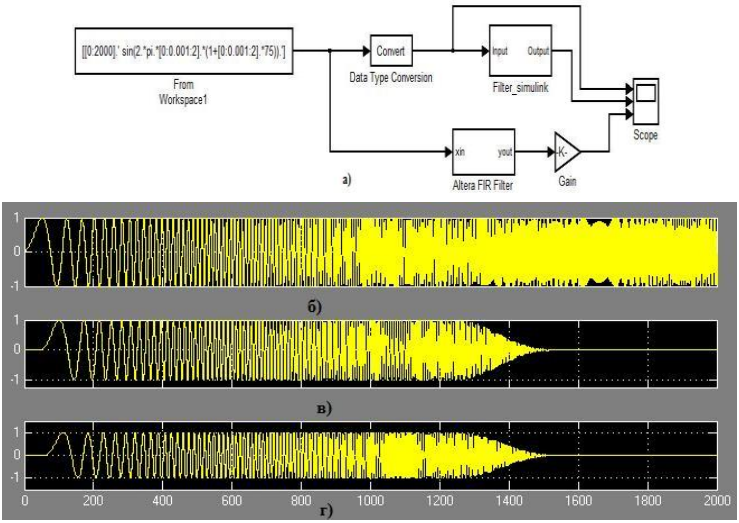


Рис.2.4. Модель КИХ-фильтра в системе Matlab/Simulink (а) и сигнал до (б) и после фильтрации КИХ-фильтром нижних частот, с использованием среды FDATool (в) и с использованием мегафункции Core FIR Compiler САПР ПЛИС Quartus

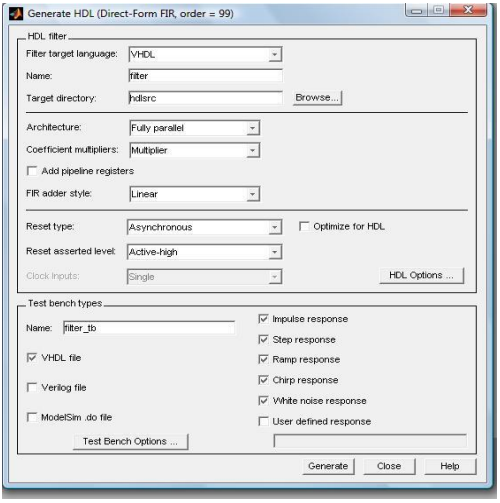


Рис.2.5. Окно Simulink HDL Coder

2.2. 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 вейфлет-преобразования и др., для реализации мультимедиа систем в базисе ПЛИС. Поэтому представляет определенный интерес рассмотреть основы такой арифметики на примере проектирования КИХ-фильтра на 4 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 все положительные, либо положительными или отрицательными числами. При проектировании фильтров используются такие понятия как дополнение до единицы и дополнение до двух, т.е. обратный и дополнительный код, а так же операция “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:

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

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

На рис.2.6 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 4 отвода (такой функциональный узел называют многоразрядный

сдвиговый регистр), перемножителей, для умножения сигналов на константы и одного сумматора с внутренней организацией “дерево сумматоров”. На рис.2.7 показана общепринятая методика умножения с накоплением (MAC), характерная для реализации в базисе сигнальных процессоров, используемая для построения КИХ-фильтра на 4 отвода.

На рис.2.8 показан 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.2.9 показывает умножение десятичного числа 11 на 5 методом правого сдвига с накоплением по рекуррентной формуле, показанной на рис.2.8.

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 отсутствия умножителей. Структура КИХ-фильтра на 4 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.2.10).

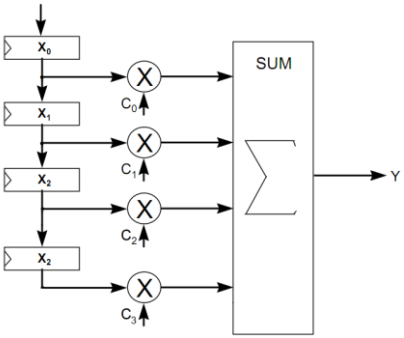


Рис.2.6. Прямая реализация КИХ-фильтра на 4 отвода

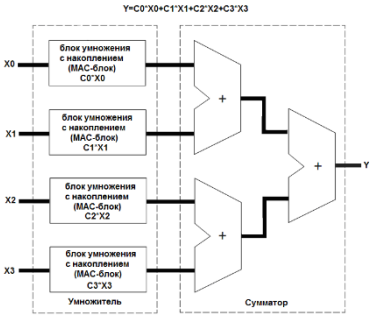


Рис.2.7. Параллельный алгоритм реализации уравнения КИХ-фильтра на 4 отвода с использованием 4 блоков умножения с накоплением

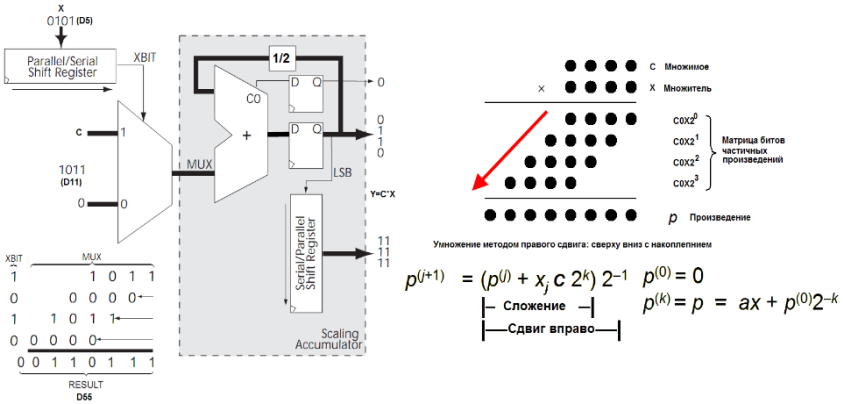


Рис.2.8. Блок умножения с накоплением и алгоритм реализации умножения методом сдвига и сложения с накоплением

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

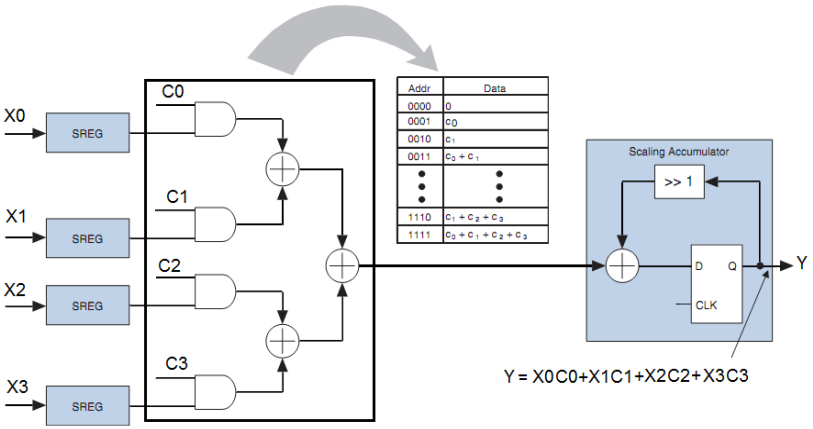


Рис.2.10. Идея использования распределенной арифметики на примере КИХ-фильтра на 4 отвода

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

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

где B – разрядность кода. Подставим выражение 2 в 1, получим:

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

$$= -\sum_{k=0}^{K-1} x_{k(B-1)} 2^{B-1} C_k + \sum_{k=0}^{K-1} \sum_{b=0}^{B-2} x_{kb} 2^b C_k \quad . \qquad (3)$$

Раскроем все суммы в выражении (3) и сгруппируем числа по степеням B:

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

(4)

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

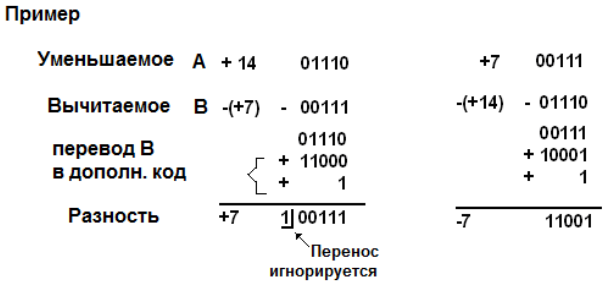
$$y = P_0 + P_1 \cdot 2^1 + P_2 \cdot 2^2 - P_3 \cdot 2^3 \quad .$$

$$P_0 = X_{00} C_0 + X_{10} C_1 + X_{20} C_2 + X_{30} C_3$$
$$P_1 = X_{01} C_0 + X_{11} C_1 + X_{21} C_2 + X_{31} C_3$$
$$P_2 = X_{02} C_0 + X_{12} C_1 + X_{22} C_2 + X_{32} C_3$$
$$P_3 = X_{03} C_0 + X_{13} C_1 + X_{23} C_2 + X_{33} C_3 \quad ,$$

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 в лево) 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.2.11). Одна 4-х 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.2.11).



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

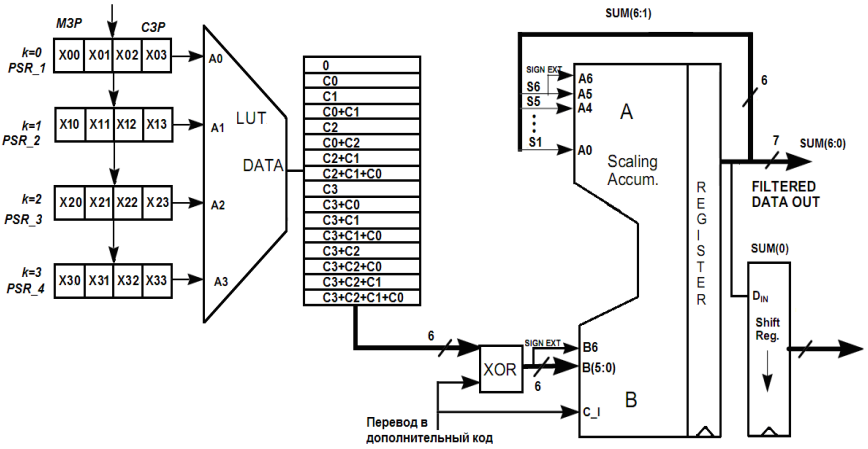


Рис.2.11. Упрощенная схема КИХ-фильтра на распределенной арифметики

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.2.12). 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.2.11.

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.2.13, информационный поток показан красным цветом) 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 (информационный поток показан зеленым цветом) позволяет накопить в регистре Shif Reg 2 число 010 (рис.2.14). Четвертый цикл обработки (информационный поток показан синим цветом) заканчивается вычитанием всех битов знакового разряда всех k входных переменных LUT-таблицы (рис.2.15).

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 4 отвода с использованием последовательной распределенной арифметике в САПР ПЛИС Quartus II компании Altera показана на рис.2.16.

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 $f = x_0 C_0 + x_1 C_1 + x_2 C_2 + x_3 C_3$.

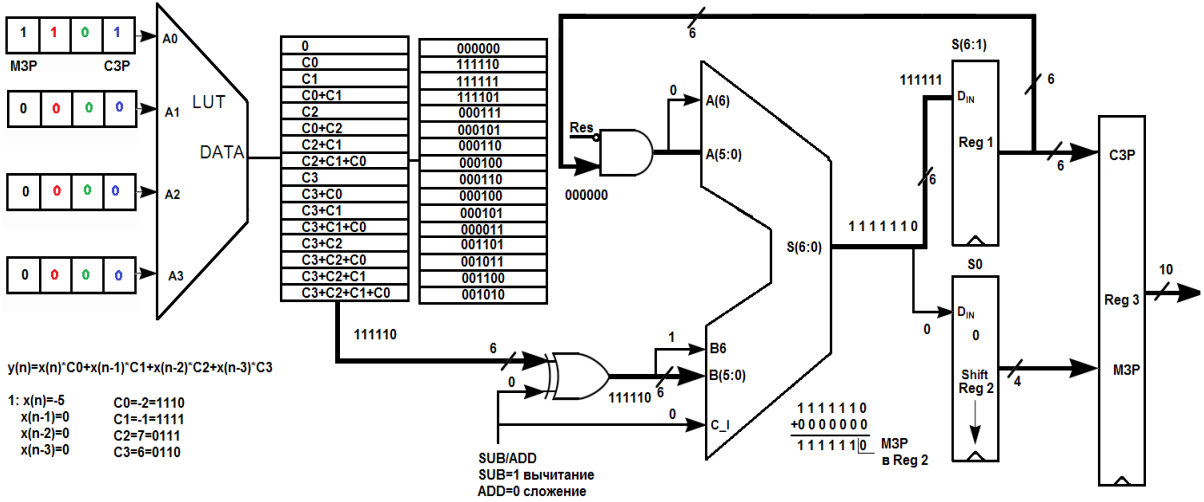


Рис.2.12. На вход КИХ-фильтра подается число десятичное число равное -5 представленное в дополнительном четырехзначном двоичном коде как 1011, первый цикл обработки (адресация 0001). Суммирование частичного произведения P_0 с весом 2^0 с 0

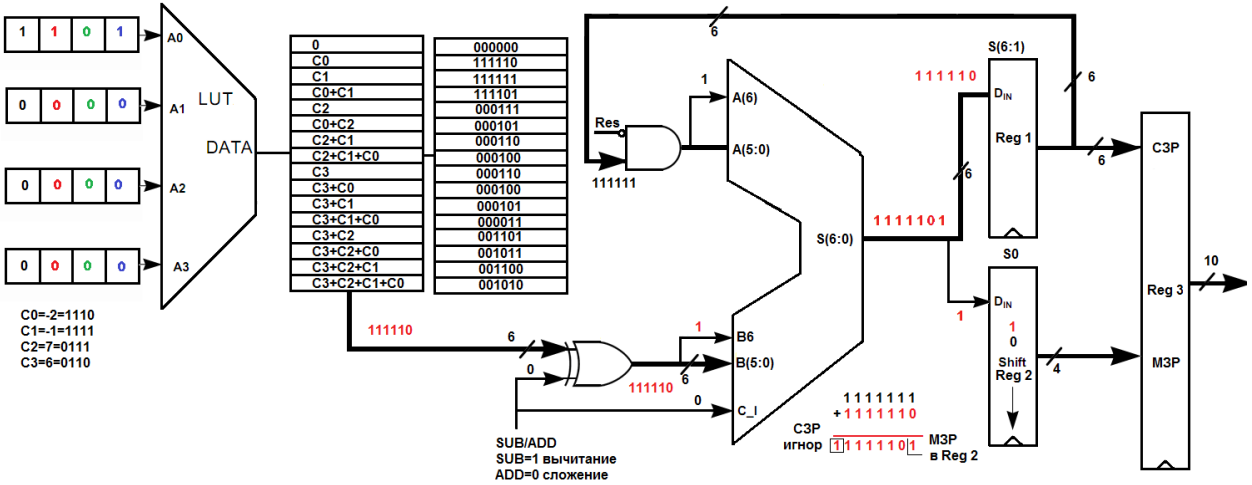


Рис.2.13. Второй цикл обработки (адресация 0001, информационный поток показан красным цветом). Суммирование частичного произведения P_1 с весом 2^1 с частичным произведением P_0 с весом 2^0

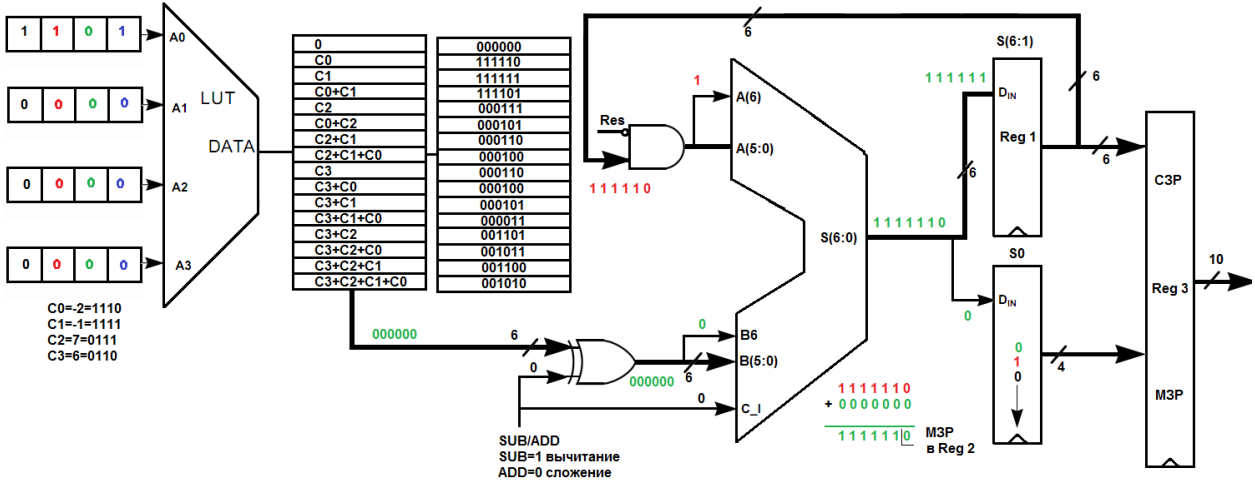


Рис.2.14. Третий цикл обработки (адресация 0000, информационный поток показан зеленым цветом). Суммирование частичного произведения P_2 с весом 2^2 с суммой частичных произведений P_0 с весом 2^0 и P_1 с весом 2^1

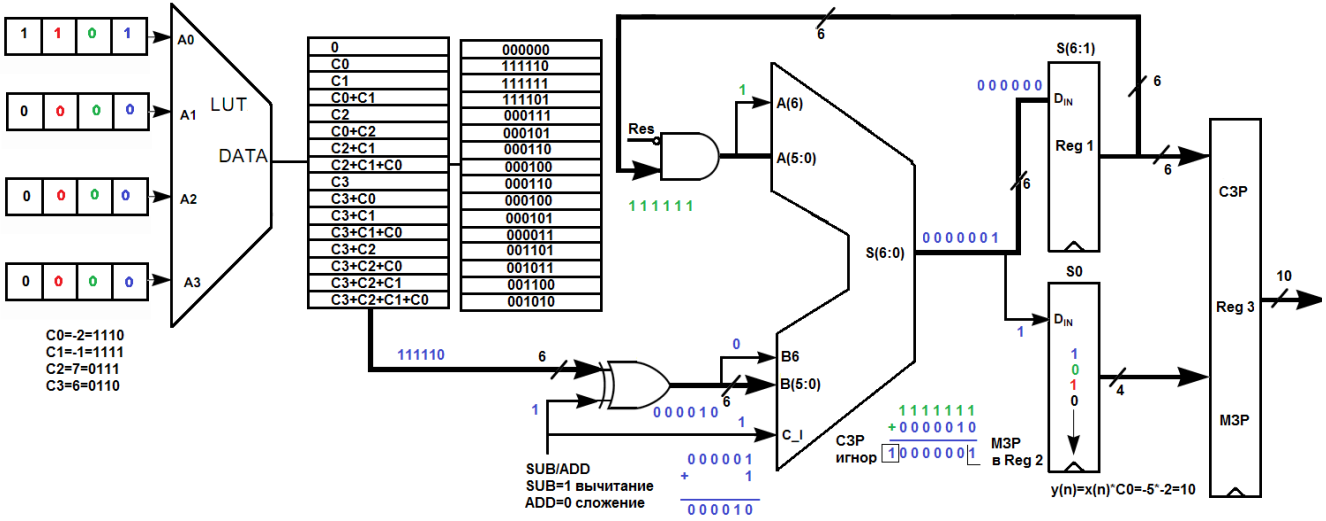


Рис.2.15. Четвертый цикл обработки (адресация 0001, информационный поток показан синим цветом). Суммирование (вычитание) частичного произведения P_3 с весом -2^3 представленного в дополнительном коде с суммой частичных произведений P_2 с весом 2^2, P_1 с весом 2^1 и P_0 с весом 2^0

На адресные входы мультиплексора поступают биты младшего значащего разряда всех k входных переменных LUT-таблицы. Перед началом работы, регистры линии задержки (рис.2.17) 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 4 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) поступающих 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.2.18 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 4 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 делает данное программное обеспечение незаменимым при проектировании КИХ-фильтров в базисе ПЛИС фирмы Altera.

На рис.2.19 показаны настройки мегафункции 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. Выбирается структура КИХ-фильтра на последовательной распределенной арифметике, 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. Задается так же входная и выходная спецификации фильтра (разрядность представления входных и выходных данных). На рис.2.20 показана тестовая схема КИХ-фильтра с использованием мегафункции FIR Compiler, а на рис.1.21 временные диаграммы работы. Входные отсчеты -5, 3, 1, 0. Профильтрованные значения на выходе: 10, -1, -40, 25.

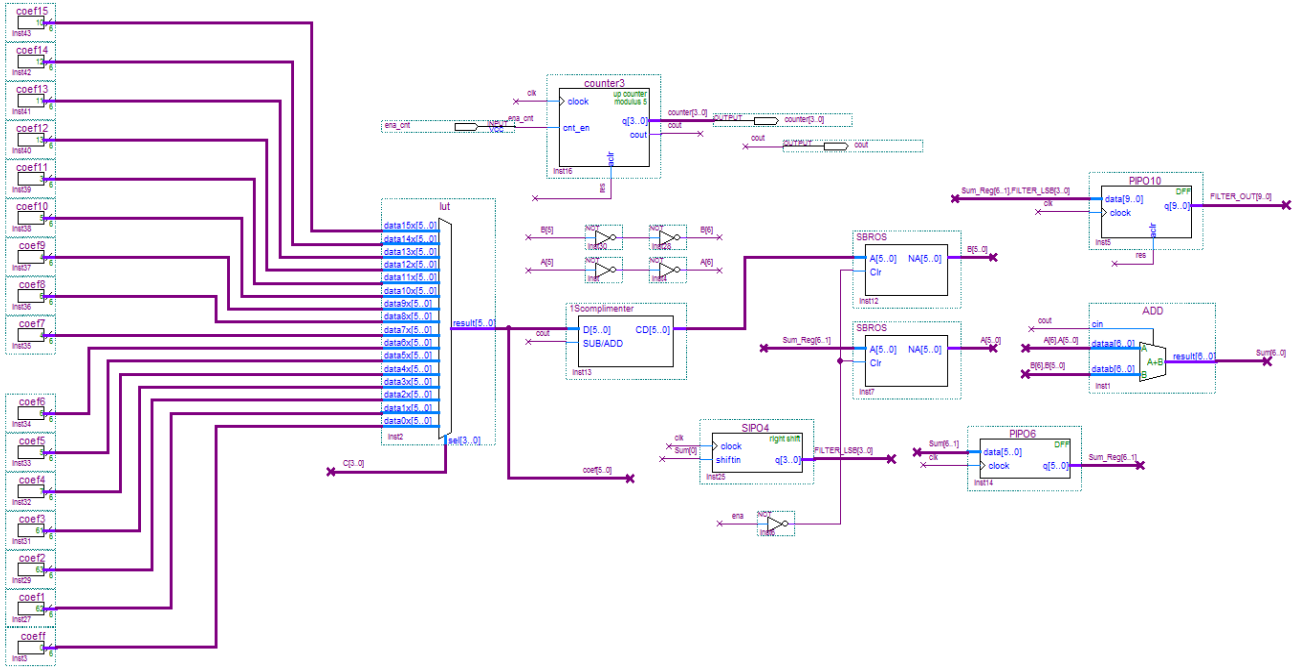


Рис.2.16. Фрагмент схемы КИХ-фильтра на 4 отвода. Показаны мультиплексор 16 в 1 для хранения комбинации сумм коэффициентов, блок вычисления обратного кода, два блока очистки данных на входах сумматора, счетчик с модулем счета 5 и вспомогательные регистры

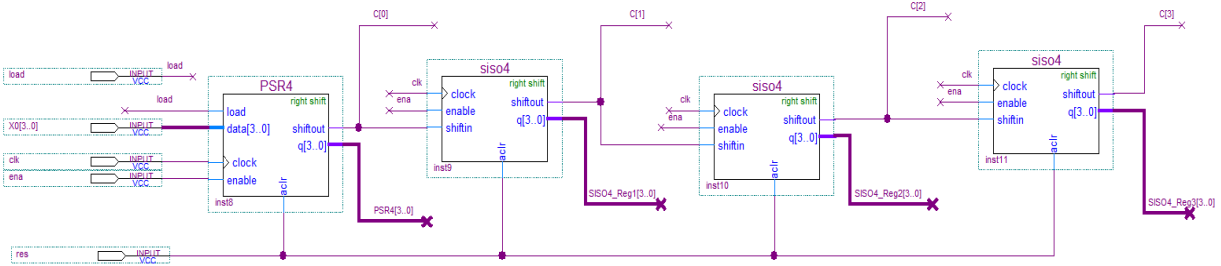


Рис.2.17. Фрагмент схемы КИХ-фильтра. Линия задержки

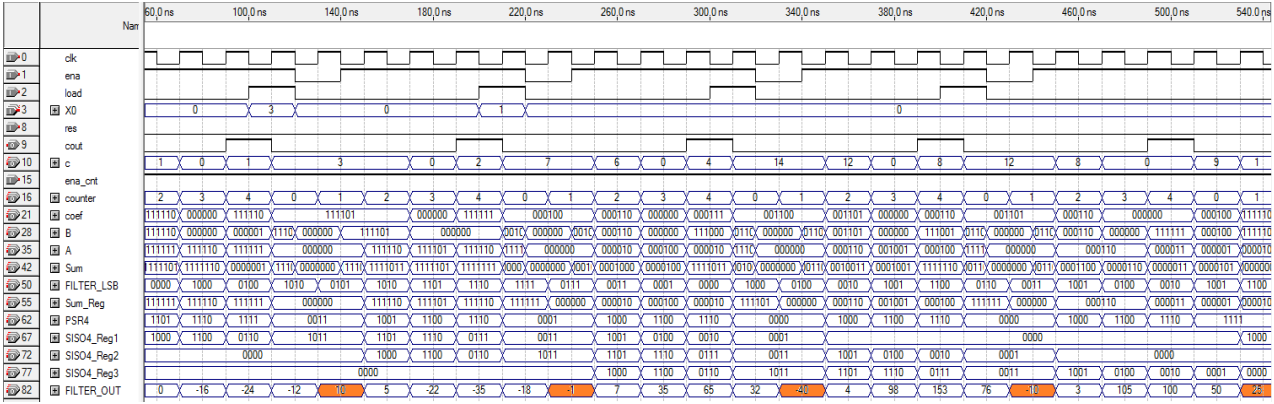


Рис.2.18. Временные диаграммы работы КИХ-фильтра на распределенной арифметике

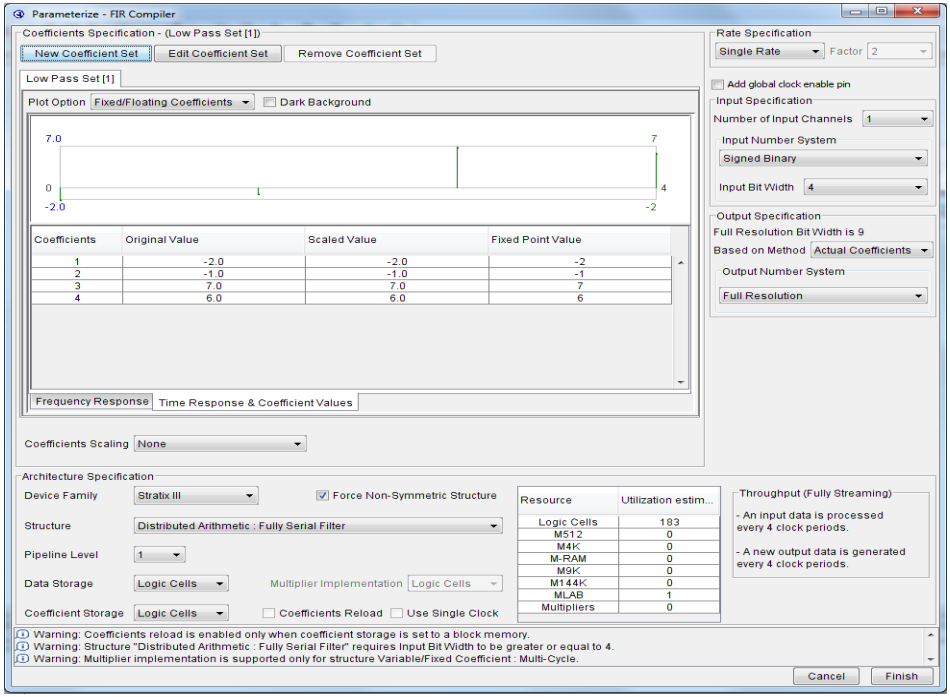


Рис.2.19. Настройки мегафункции FIR Compiler САПР ПЛИС Quartus II

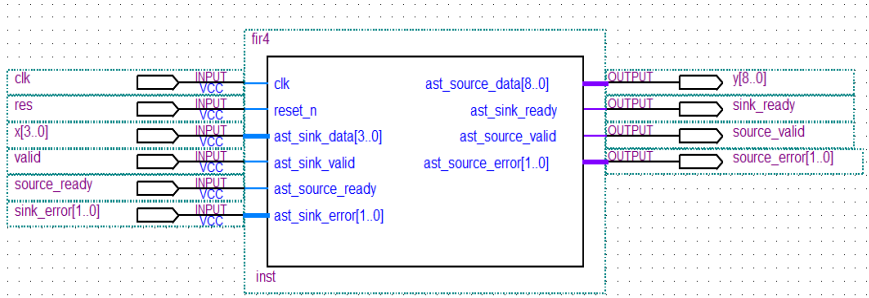


Рис.2.20.Тестовая схема КИХ-фильтра с использованием мегафукнкции FIR Compiler

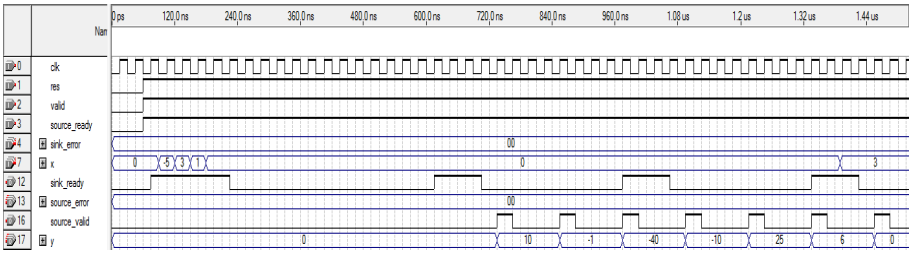


Рис.2.21. Временные диаграммы работы КИХ-фильтра на мегафункции FIR Compiler

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.2.22). Сравнивая временные диаграммы (рис.2.18 и рис.2.22 с рис.2.21) видим, что профильтрованные значения на выходе у трех фильтров совпадают, однако у фильтров на распределенной арифметике “нужные” выходные значения обновляются 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;
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 4 отвода

```
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 : in std_logic;
          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);
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 4 отвода (вариант)

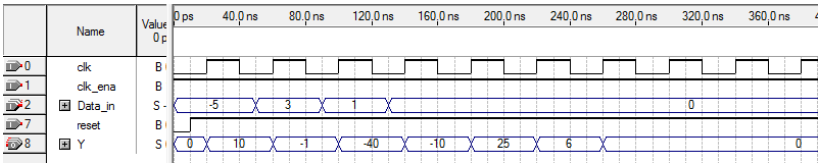


Рис.2.22. Временные диаграммы работы КИХ-фильтра на 4 отвода по коду языка VHDL (пример 2)

В данном разделе на примере проектирования простейшего КИХ-фильтра на 4 отвода показаны основы распределенной арифметики широко используемой для разработки высокопроизводительных цифровых устройств цифровой обработки сигналов.

2.3. Проектирование параллельных КИХ-фильтров в базисе ПЛИС

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:

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

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 4 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 4 отвода с использованием 4 блоков умножения с накоплением. В проекте используются 4 мегафункции ALTMULT_ACCUM (рис.2.23). Каждый блок использует 1 перемножитель и 1 сумматор-аккумулятор. Для параллельной реализации фильтра на 4 отвода требуется 4 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.2.23 также показана внешняя линия задержки на 4 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 4 отвода с использованием 4 перемножителей в блоке на мегафункции ALTMULT_ADD (функция умножения и сложения) в САПР ПЛИС Quartus II показана на рис.2.24. В мегафункции ALTMULT_ACCUM используется три встроенных сумматора. Профильтрованные значения показаны на рис.2.25. 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). Фактические это одна мегафункция (1 блок с 4 перемножителями), в которой линия задержки организована на внутренних регистрах 4 перемножителей. В мегафункции используются встроенные два сумматора и один сумматор-аккумулятор (рис.2.26). Временные диаграммы работы фильтров, показанные на рис.2.27 не отличаются от диаграмм на рис.2.25.

Рассмотрим умножение десятичного числа 11 на 10 на примере мегафункции ALTMEMMULT (рис.2.28). Мегафункция ALTMEMMULT 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 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.2.29).

Третий вариант. Применяя мегафункцию ALTMEMMULT, разработаем параллельную реализацию КИХ-фильтра на 4 отвода с использованием 4 перемножителей (4 блока по 1 перемножителю в каждом) в САПР ПЛИС Quartus II и дерева сумматоров (рис.2.30). Внешняя линия задержки состоит не из 3 регистров как в первых двух вариантах, а из 4 регистров. Дополнительно требуется, как и в первом варианте три однотипных многоразрядных сумматора. Латентность каждого умножителя 2. В каждый умножитель по умолчанию загружено число 0. Временные диаграммы работы фильтра на 4 отвода с использованием мегафункции ALTMEMMULT показаны на рис.2.31. 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.

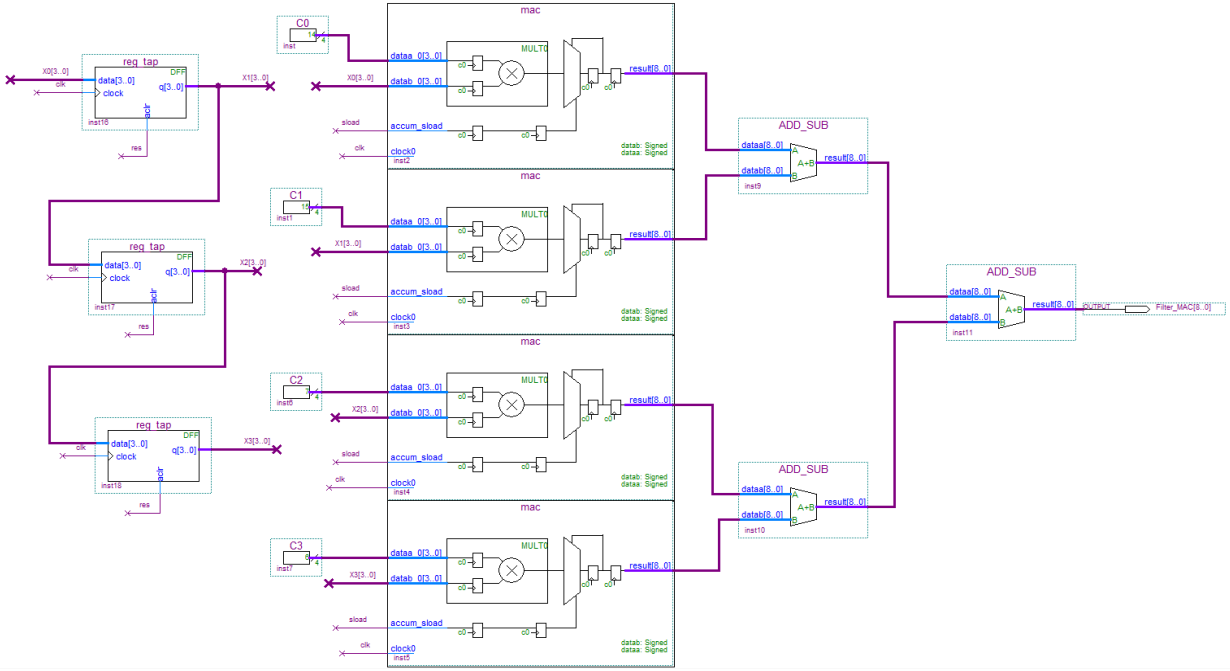


Рис.2.23. Параллельная реализация КИХ-фильтра на 4 отвода с использованием 4 блоков в САПР ПЛИС Quartus II (мегафункция ALTMULT_ACCUM)

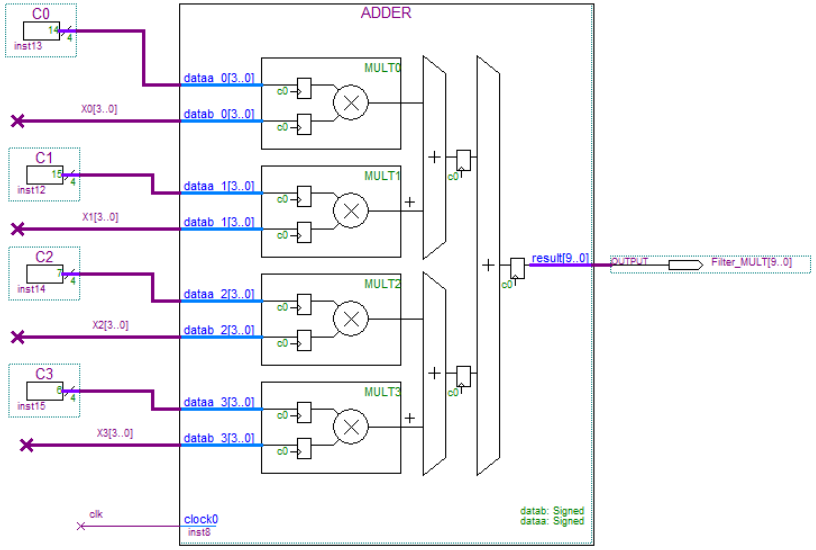


Рис.2.24. Параллельная реализация КИХ-фильтра на 4 отвода с использованием 4 перемножителей в блоке (мегафункция ALTMULT_ADD, линия задержки такая же, как и на рис.2.23)

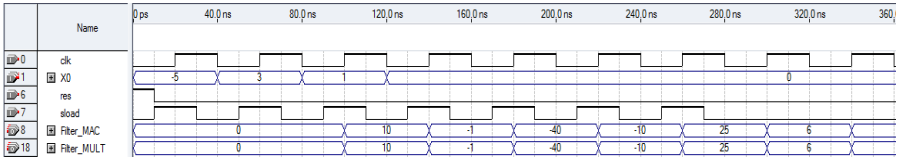


Рис.2.25. Временные диаграммы работы параллельных фильтров на 4 отвода с использованием мегафункции ALTMULT_ACCUM и ALTMULT_ADD

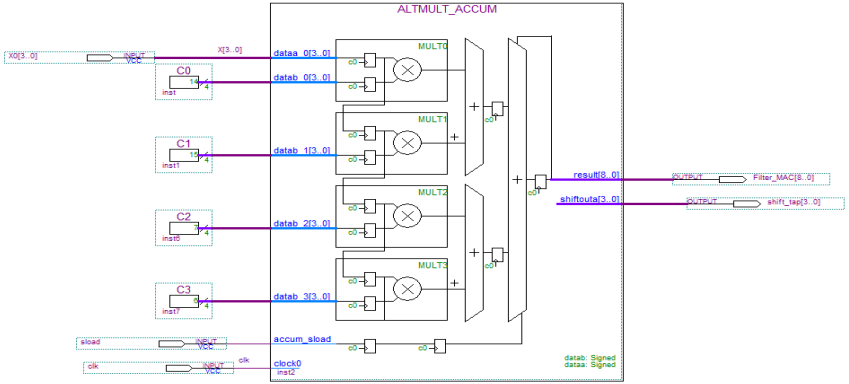


Рис.2.26. Параллельная реализация КИХ-фильтра на 4 отвода с использованием 1 блока с 4 перемножителями в САПР ПЛИС Quartus II (мегафункция ALTMULT_ACCUM, линия задержки построена на внутренних регистрах перемножителей MULT0-MULT3)

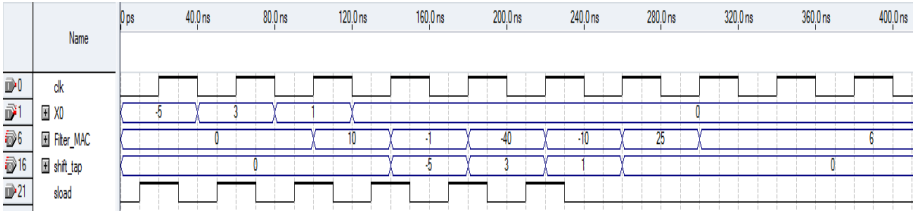


Рис.2.27. Временные диаграммы работы фильтра на 4 отвода с использованием мегафункции ALTMULT_ACCUM

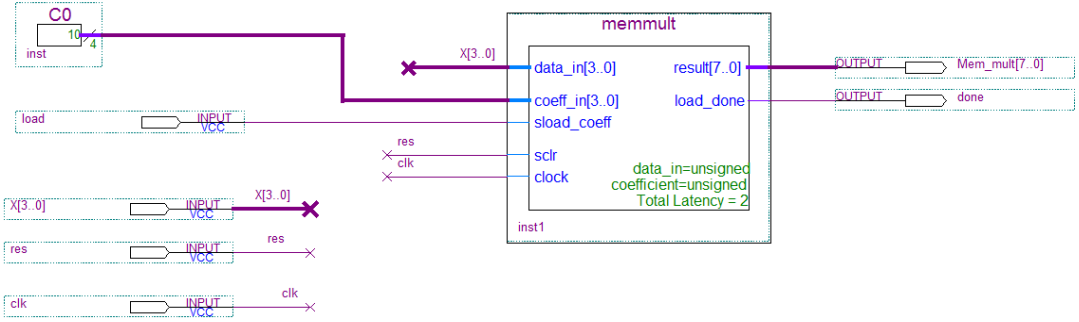


Рис.2.28. Умножение 11 на 10 с помощью мегафункции ALTMEMMULT

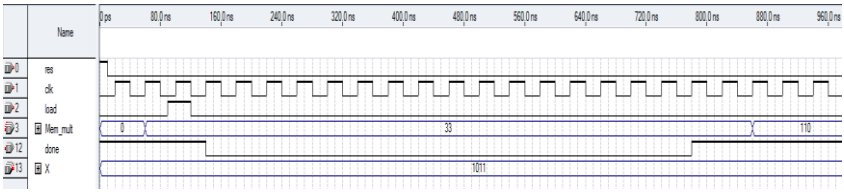


Рис.2.29. Временные диаграммы умножения 11 на 10. По умолчанию в мегафункцию загружена константа 3. Результат 110

Рассмотрим вариант, когда коэффициенты фильтра загружаются из блочной памяти в ПЛИС. В мегафункции ALTMEMMULT коэффициенты представляются как целочисленные значения со знаком (рис.2.32). Временные диаграммы работы фильтра на 4 отвода с использованием мегафункции ALTMEMMULT показаны на рис.2.33. Сравнивая рис.2.31 и рис.2.33 видим, что быстродействие фильтра в этом случае значительно увеличивается за счет хранения коэффициентов в блочной памяти.

Масштабирующий аккумулятор входит не только в состав последовательных и параллельных КИХ-фильтров на распределенной арифметике но и может применяться для других целей. Рассмотрим проектирование масштабирующего аккумулятора с использованием метода правого сдвига с накоплением, на примере умножения числа 11 на константу 10 (целочисленные без знаковые числа). На рис.2.34 показана идея схемы метода умножения методом правого сдвига с накоплением.

Для практической реализации в базисе ПЛИС потребуется преобразователь параллельного кода в последовательный со сдвигом в право (мегафункция LPM_SHIFTREG), умножитель одноразрядного числа на четырехразрядное (мегафункция LPM_MULT, старший значащий разряд игнорируется) и масштабирующий аккумулятор (рис.2.35). На рис.2.36 показан масштабирующий аккумулятор, который состоит из 4-х разрядного сумматора

(мегафункция LPM_ADD_SUB), сдвиговых регистров lpm_shiftreg0 и shift_LSB, вспомогательного регистра ff5 и регистра результата ff8. Регистр ff5 объединяет результат суммирования 2p[3..0] и разряд переноса Cout сумматора. Младший значащий разряд накапливается в регистре shift_LSB. Полученный 5 разрядный сигнал сдвигается вправо на один разряд с помощью сдвигового регистра lpm_shiftreg0, при этом старший значащий разряд отбрасывается, а оставшиеся более младшие разряды вновь поступают на один из входов сумматора. В регистре ff8 происходит сохранение результата преобразования, который представляет собой 8 разрядный сигнал, полученный объединением сигналов p[3..0] и LSB[3..0]. На рис.2.37 показаны временные диаграммы умножения числа 11 на 10. Результат 110. Сравнивая результат умножения числа 11 на 10 с использованием мегафункции ALTMEMMULT показанный на рис.2.31 и на рис.2.33 с использованием метода правого сдвига с накоплением видим, что для умножения во втором случае потребовалось 18 синхроимпульсов. Если в разработанном варианте в качестве перемножителя использовать мультиплексор многоразрядных сигналов, то можно получить “без перемножительную” схему умножения и построить КИХ-фильтр без операции явного умножения.

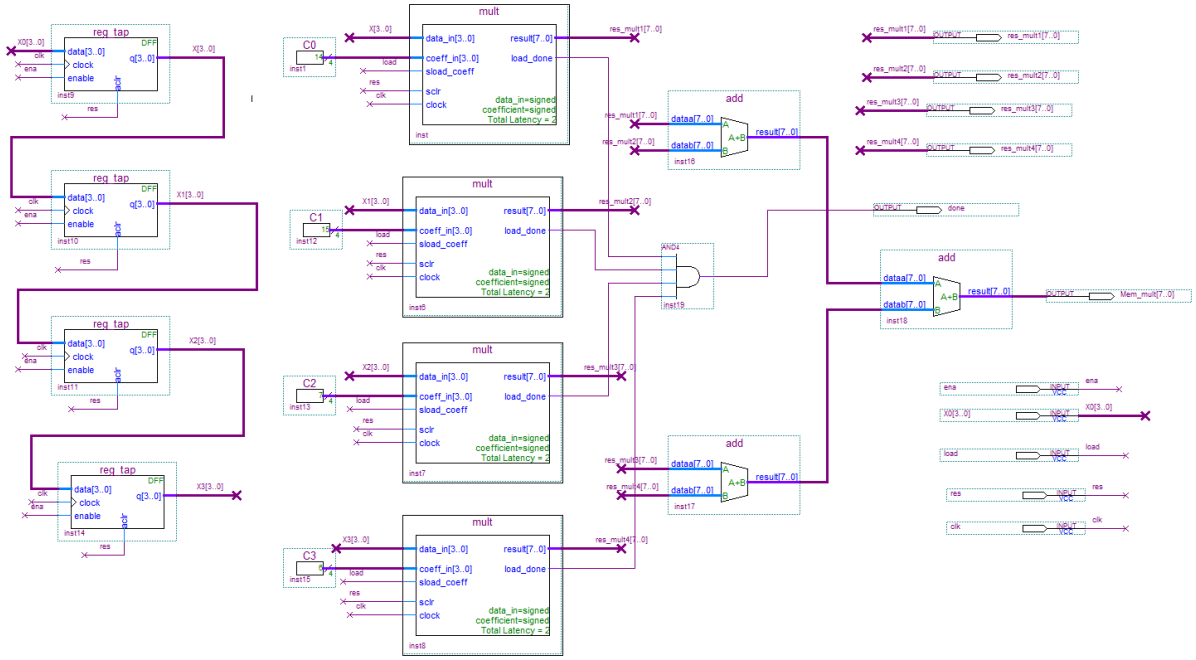


Рис.2.30. Параллельная реализация КИХ-фильтра на 4 отвода с использованием 4 умножителей в САПР ПЛИС Quartus II (мегафункция ALTMEMMULT, линия задержки построена на 4 регистрах, коэффициенты фильтра загружаются из внешнего порта)

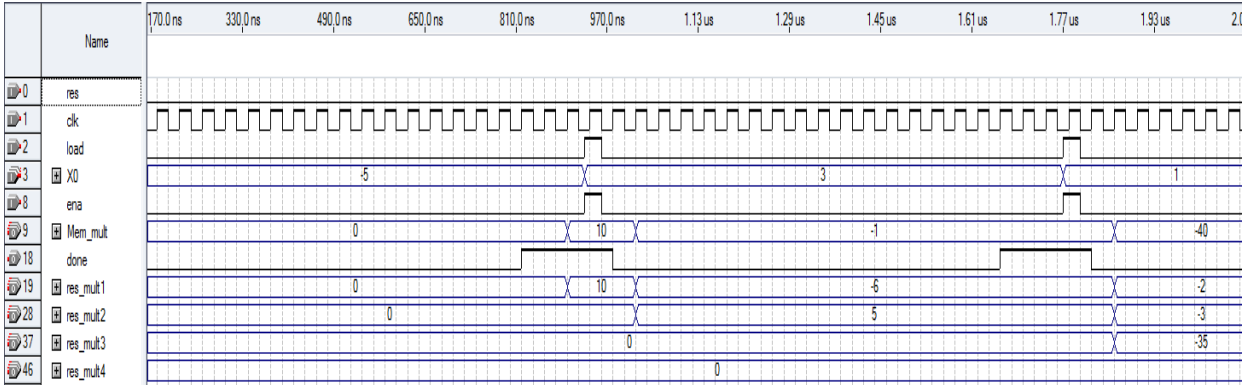


Рис.2.31. Временные диаграммы работы фильтра на 4 отвода с использованием мегафункции ALTMEMMULT (коэффициенты фильтра загружаются из внешнего порта)

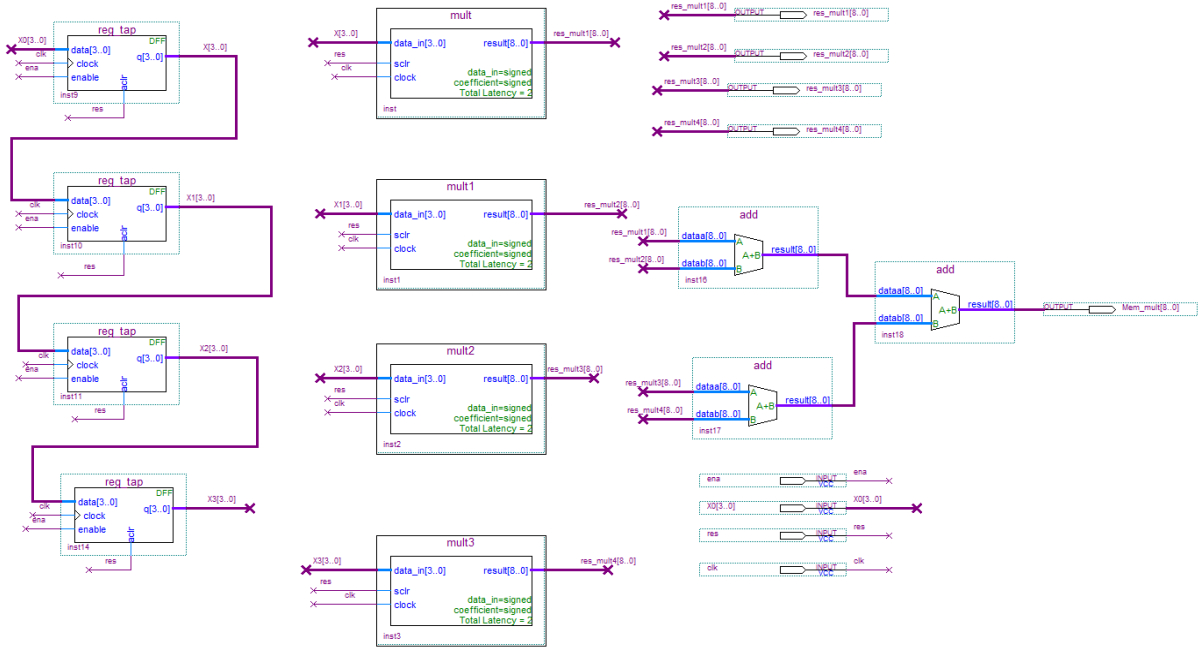


Рис.2.32. Параллельная реализация КИХ-фильтра на 4 отвода с использованием 4 перемножителей в САПР ПЛИС Quartus II (мегафункция ALTMEMMULT, линия задержки построена на 4 регистрах, коэффициенты фильтра загружаются из блочной памяти)

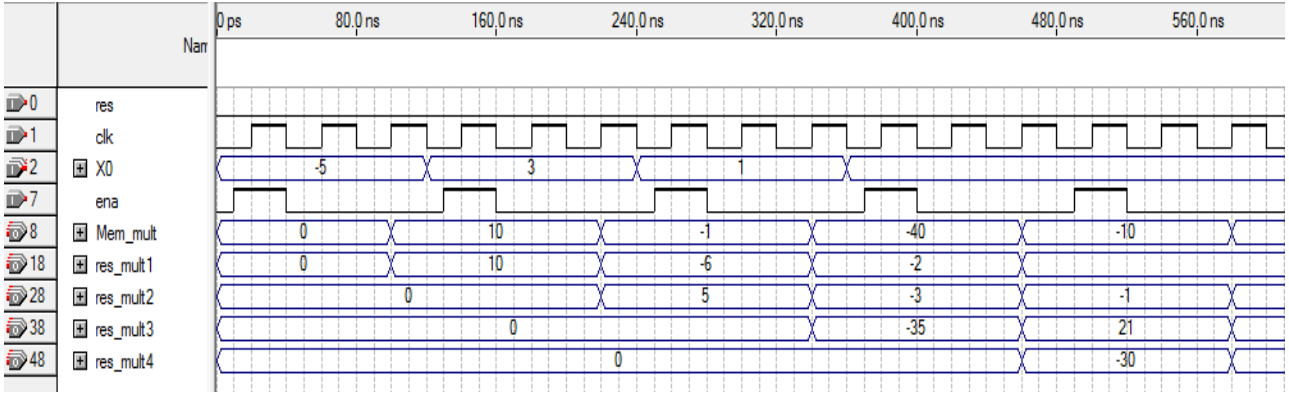
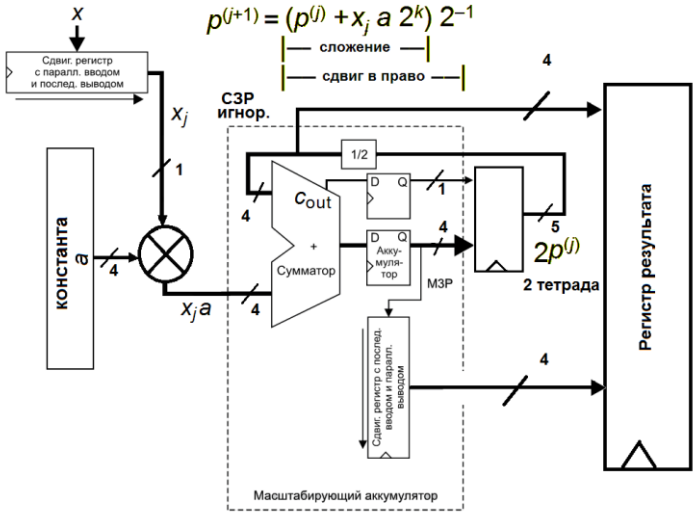


Рис.2.33. Временные диаграммы работы фильтра на 4 отвода с использованием мегафункции ALTMEMMULT (коэффициенты фильтра загружаются из блочной памяти ПЛИС)

	2 тетрада	1 тетрада	
a		1 0 1 0	(10 Dec)
x		1 0 1 1	(11 Dec)
$p^{(0)}$	0 0 0 0	0 0 0 0	(0 Dec)
$+x_0a$	1 0 1 0		
$2p^{(1)}$	0 1 0 1 0		
$p^{(1)}$	0 1 0 1	0 0 0 0	(80 Dec)
$+x_1a$	1 0 1 0		
$2p^{(2)}$	0 1 1 1 1	0	
$p^{(2)}$	0 1 1 1	1 0 0 0	(120 Dec)
$+x_2a$	0 0 0 0		
$2p^{(3)}$	0 0 1 1 1	1 0	
	0 0 1 1	1 1 0 0	(60 Dec)
17	1 0 1 0		
$2p^{(4)}$	0 1 1 0 1	1 1 0	
$p^{(4)}$	0 1 1 0	1 1 1 0	(110 Dec)

а)



б)

Рис.2.34. Идея схемы метода умножения методом правого сдвига с накоплением: а) - алгоритм умножение десятичного числа 11 на 10; б) - структурная схема метода

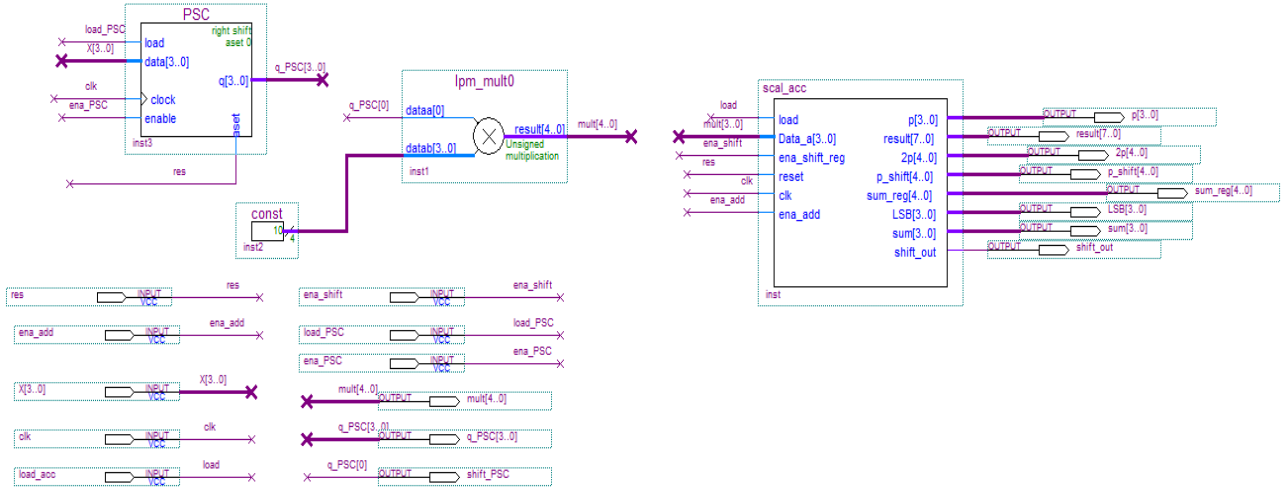


Рис.2.35. Схема умножения методом правого сдвига с накоплением

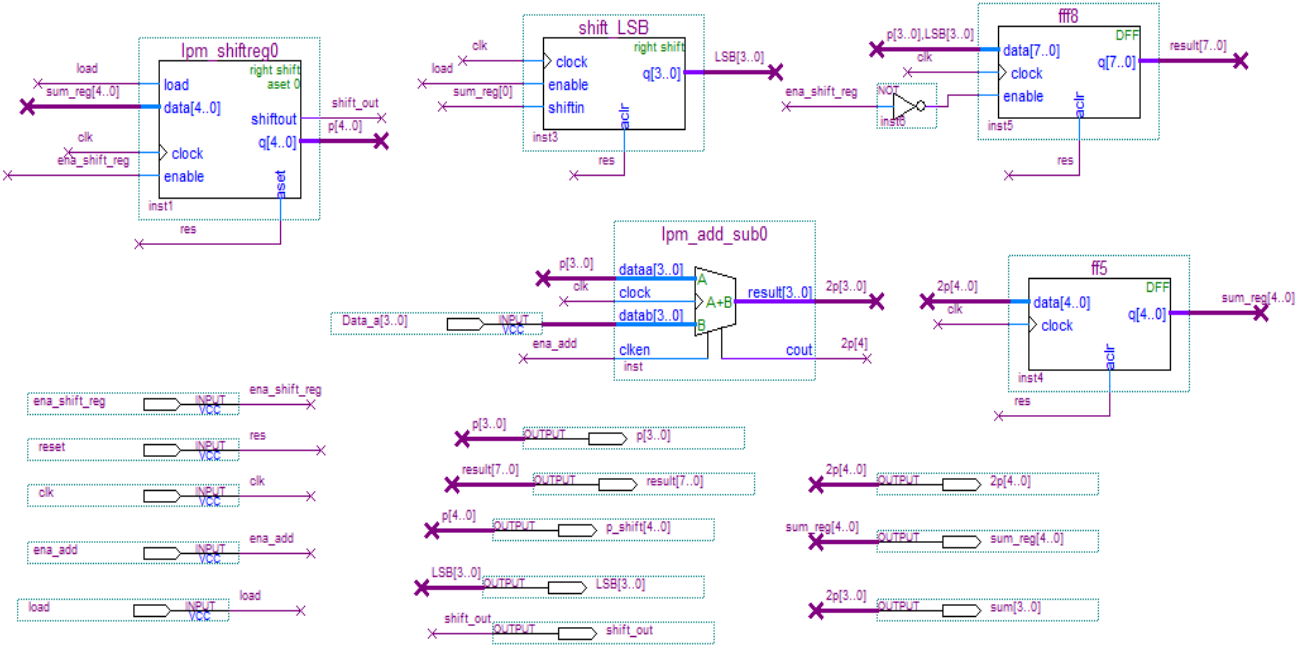


Рис.2.36. Масштабирующий аккумулятор

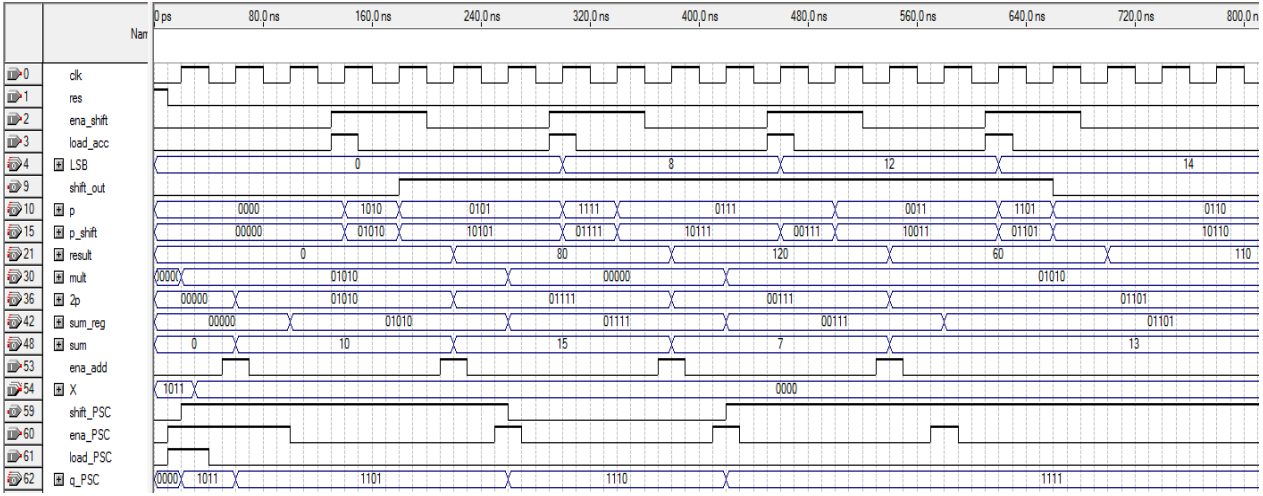


Рис.2.37. Временные диаграммы умножения числа 11 на 10 методом правого сдвига с накоплением. Результат 110

Таблица 2.2

Анализ задействованных ресурсов ПЛИС серии Stratix при реализации параллельных КИХ-фильтров на 4 отвода с использованием мегафункций

Ресурсы ПЛИС серии Stratix	Мегафункция ALTMULT_ACCUM. 4 блока по 1 перемножителю в каждом, внешняя линия задержки из 3 регистров / 1 блок (4 перемножителя в блоке, встроенная линия задержки)		Мегафункция ALT-MULT_ADD. 4 перемножителя в блоке, внешняя линия задержки из 3 регистров	Мегафункция ALTMEM-MULT. 4 блока по 1 перемножителю в каждом (коэффициенты фильтра загружаются из внешнего порта)	Мегафункция ALTMEMMULT. 4 блока по 1 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

В данном разделе рассмотрены различные варианты проектирования параллельных КИХ-фильтров с использованием мегафункций САПР ПЛИС Quartus II компании Altera а также умножение методом правого сдвига с накоплением.

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, которая позволяет построить параллельный КИХ-фильтр на 4 отвода с использованием всего лишь 1 блока с 4 перемножителями, встроенными двумя сумматорами, сумматором-аккумулятором и линией задержки.

Мегафункции ALTMULT_ADD так же позволяет построить параллельный КИХ-фильтр с использованием всего лишь 1 блока с 4 перемножителями, встроенными тремя сумматорами. Использование внешней линии задержки из 3 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 4 аппаратных перемножителей 18x18.

Фильтр на мегафункции ALTMEMMULT с загрузкой коэффициентов из внешнего порта обладает пониженным быстродействием. Использование же блочной памяти для хранения коэффициентов фильтра внутри ПЛИС значительно упрощает процесс разработки и не приводит к существенному увеличению ресурсов за счет использования внешней линии задержки на 4 регистрах, дополнительных трех однотипных многоразрядных сумматоров и не снижает быстродействие проекта.

2.4. 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:

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

где $y-$ отклик цепи; $x_k -$ $k-$ая входная переменная; $C_k -$ весовой коэффициент $k-$ой 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:

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

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

$$\begin{aligned} y &= \sum_{k=1}^{K} C_k \cdot \left[- x_{k(\mathrm{B}-1)} 2^{B-1} + \sum_{b=0}^{B-2} x_{kb} \cdot 2^b \right] = \\ &= -\sum_{k=1}^{K} x_{k(B-1)} 2^{B-1} C_k + \sum_{k=1}^{K} \sum_{b=0}^{B-2} x_{kb} 2^b C_k \end{aligned} \qquad (3)$$

Раскроем все суммы в выражении (3) и сгруппируем числа по степеням B:

$$
\begin{aligned}
y = &\left[x_{10}\cdot C_1 + x_{20}\cdot C_2 + x_{30}\cdot C_3 + ... + x_{K0}\cdot C_K\right]\cdot 2^0 \\
&+\left[x_{11}\cdot C_1 + x_{21}\cdot C_2 + x_{31}\cdot C_3 + ... + x_{K1}\cdot C_K\right]\cdot 2^1 \\
&+\left[x_{12}\cdot C_1 + x_{22}\cdot C_2 + x_{32}\cdot C_3 + ... + x_{K2}\cdot C_K\right]\cdot 2^2 \\
&.... \\
&.... \\
&+\left[x_{1(B-2)}\cdot C_1 + x_{2(B-2)}\cdot C_2 + x_{3(B-2)}\cdot C_3 + ... + x_{K(B-2)}\cdot C_K\right]\cdot 2^{B-2} \\
&-\left[x_{1(B-1)}\cdot C_1 + x_{2(B-1)}\cdot C_2 + x_{3(B-1)}\cdot C_3 + ... + x_{K(B-1)}\cdot C_K\right]\cdot 2^{B-1}
\end{aligned}
\qquad (4)
$$

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 8 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 8 отводов 8 бит. Перепишем уравнение (4) в следующем виде:

$$
\begin{aligned}
y = &[sum0] + [sum1]2^1 + [sum2]\cdot 2^2 + ... + \\
&+ [sum6]\cdot 2^6 - [sum7]\cdot 2^7
\end{aligned}
\qquad (5)
$$

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 5 в массив подобных сумм на основе двухвходовых сумматоров:

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

или

$$y = \left\{\left\{[sum0] + [sum1]2^1\right\} + \left\{[sum2] + [sum3]2^1\right\}2^2\right\} + \left\{\left\{[sum4] + [sum5]2^1\right\} + \left\{[sum6] - [sum7]2^1\right\}2^2\right\}2^4 \quad (7)$$

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 В 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.

Вышеприведенные уравнения (6) и (7) 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 8 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:

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

$$y(n) = \sum_{k=1}^{K} C_k \cdot \left[-x_{k0} + \sum_{b=1}^{B-1} x_{kb} \cdot 2^{-b} \right] = \\ = -\sum_{k=1}^{K} x_{k0} \cdot C_k + \sum_{k=1}^{K} \sum_{b=1}^{B-1} x_{kb} \cdot C_k \cdot 2^{-b} \quad . \qquad (8)$$

Аналогично выражению (4) уравнение КИХ-фильтра для формата с фиксированной запятой будет иметь вид:

$$\begin{aligned} y(n) = &-\left[x_{10} \cdot C_1 + x_{20} \cdot C_2 + x_{30} \cdot C_3 + ... + x_{K0} \cdot C_K\right] \cdot 2^0 + \\ &+ \left[x_{11} \cdot C_1 + x_{21} \cdot C_2 + x_{31} \cdot C_3 + ... + x_{K1} \cdot C_K\right] \cdot 2^{-1} + \\ &+ \left[x_{12} \cdot C_1 + x_{22} \cdot C_2 + x_{32} \cdot C_3 + ... + x_{K2} \cdot C_K\right] \cdot 2^{-2} + \\ &.... \\ &.... \\ &+ \left[x_{1(B-2)} \cdot C_1 + x_{2(B-2)} \cdot C_2 + x_{3(B-2)} \cdot C_3 + ... + x_{K(B-2)} \cdot C_K\right] \cdot 2^{-(B-2)} + \\ &+ \left[x_{1(B-1)} \cdot C_1 + x_{2(B-1)} \cdot C_2 + x_{3(B-1)} \cdot C_3 + ... + x_{K(B-1)} \cdot C_K\right] \cdot 2^{-(B-1)} \end{aligned} \quad . \qquad (9)$$

Переформулируем выражение (9) и представим его в следующем виде и сравним с уравнением (8):

$$y(n) = -\sum_{k=1}^{K} C_k x_{k0} + \sum_{b=1}^{B-1} \left[\sum_{k=1}^{K} C_k x_{kb} \right] 2^{-b} = \\ = -P_0 + \left[\sum_{b=1}^{B-1} 2^{-b} P_b \right] \quad , \qquad (10)$$

где P – частичные произведения (значения в квадратных скобках выражения (9), 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.2.38). По формуле (8) получается фильтр с использованием операций умножения с накоплением (4 MAC-фильтр) а по формуле (10) 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 8 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	$y(n) = \left[\sum_{b=0}^{B-2} 2^b P_b \right] - 2^{B-1} P_{B-1}$
Дополнительный код, дробные значения	$y(n) = -P_0 + \left[\sum_{b=1}^{B-1} 2^{-b} P_b \right]$

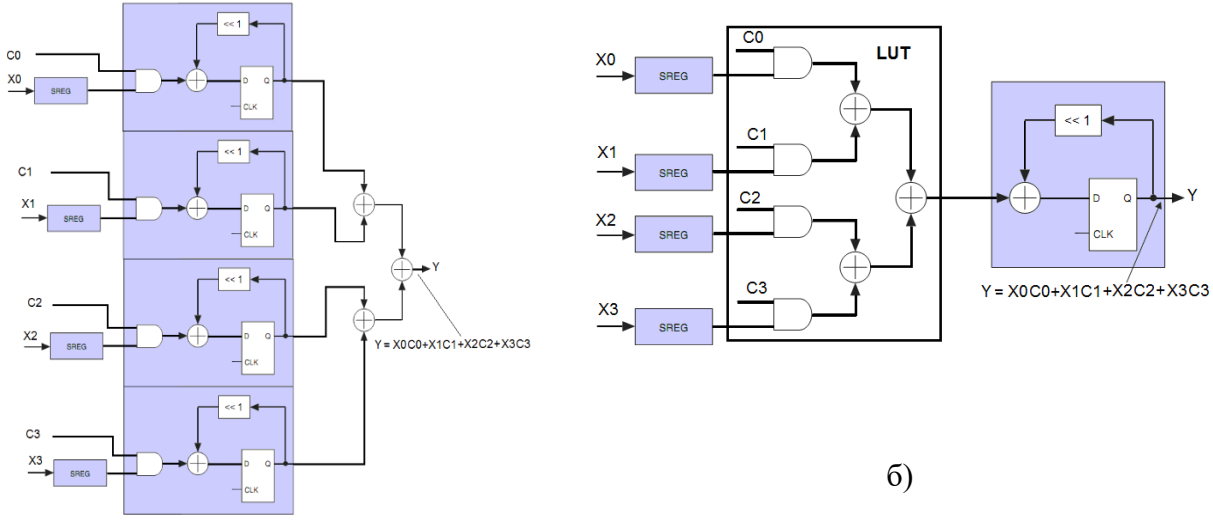


Рис.2.38. КИХ-фильтр на 4 отвода: а) аппаратная реализация фильтра по формуле (8); б) по формуле (10)

Рассмотрим КИХ-фильтр (рис.2.39) 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), являющуюся неотъемлемой частью логического блока ПЛИС. Рассмотрим простейший параллельный векторный перемножитель 4-х двухразрядных сигналов на 4 двухразрядные константы (четырехразрядный векторный перемножитель), в предположении, что все величины целочисленные и положительные, представлены в прямом коде (рис.2.40). Умножение и сложение происходят параллельно с использованием LUT (табл.2.3).

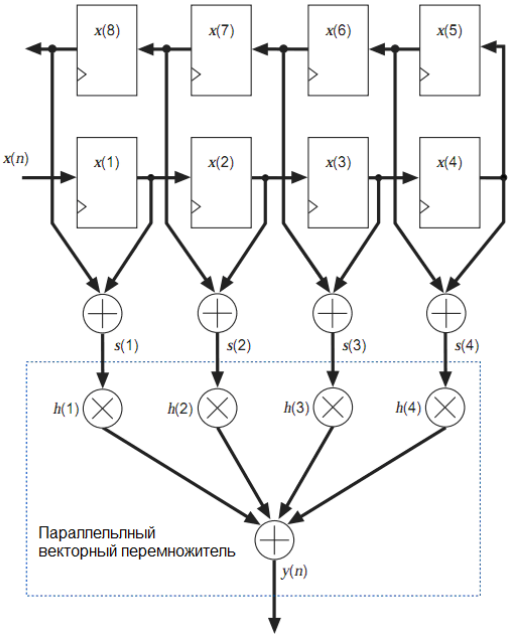


Рис.2.39. Структура КИХ-фильтра на 8 отводов с симметричными коэффициентами, в основе которой лежит параллельный векторный перемножитель

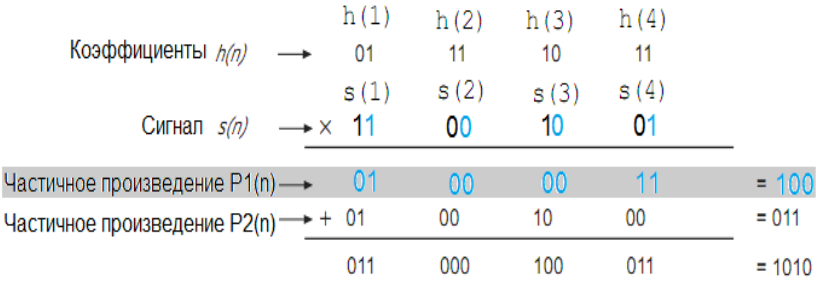


Рис.2.40. Принцип параллельного векторного перемножения

Принцип формирования частичного произведения P1(n) на рис.2.41, показан, синим цветом. 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.1.41 показан параллельный векторный перемножитель четырех двухразрядных сигналов. Таким образом, требуются две идентичные таблицы, двоичный сдвиг влево и операция суммирования.

На рис.2.42 показана структура КИХ-фильтра 8 отводов 8 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 2.3

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

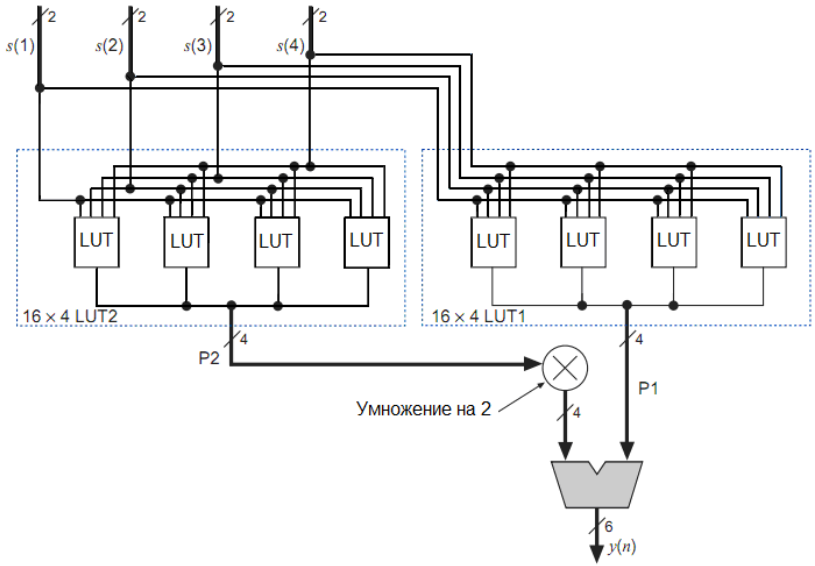


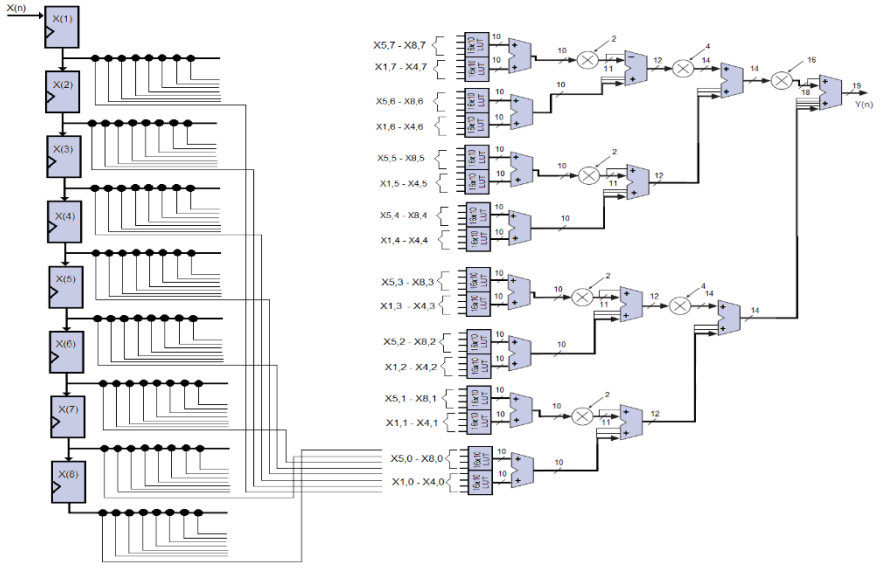
Рис.2.41. Параллельный векторный перемножитель четырех 2-х разрядных сигналов на четыре 2-х разрядные константы

Входные данные на линии задержки представлены с 8 битной точность параллельным кодом. Для фильтра с симметричными коэффициентами требуется на выходах линии

задержки 4 параллельных сумматора, которые своими выходами непосредственно адресуют 4-х входовые LUT. Для того что бы переполнение гарантировано не произошло необходимы 9-ти 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, 8 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).

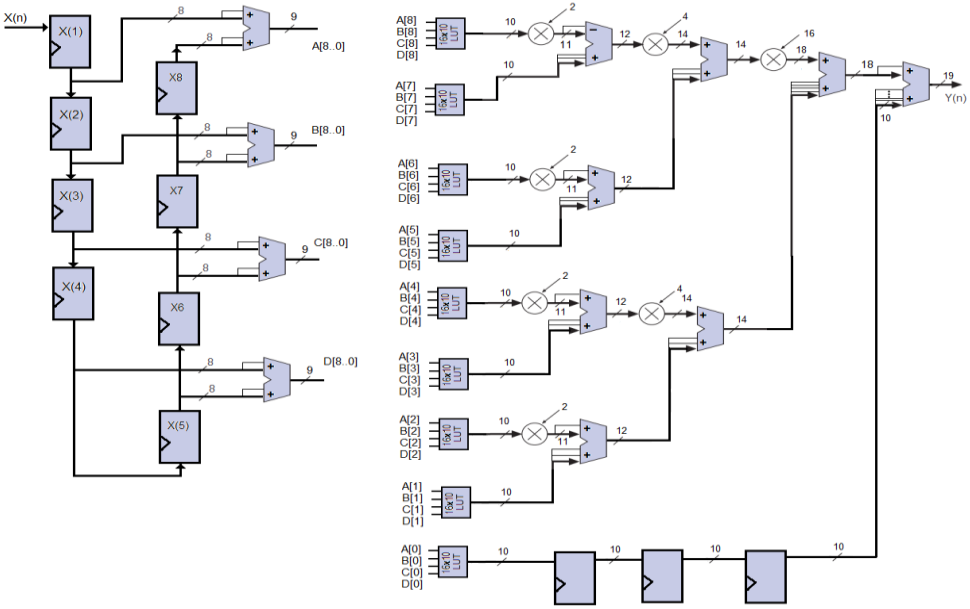
Частичные произведения, представляющие комбинацию сумм 8-ми разрядных коэффициентов, хранящиеся в LUT представлены с 10 битной точность с запасом в 2 разряда. Поэтому, в случае фильтра с несимметричными коэффициентами для суммирований значений с выходов LUT используется восемь 10 разрядных сумматоров. На входах последующих 12, 14 и 19-ти разрядных сумматоров требуется коррекция разрядности. Для фильтра с симметричными коэффициентами необходимы 12, 14, 18 и 19-ти разрядные сумматоры, с соответствующей коррекцией на входах, а для получения 19-ти битной точности дополнительно требуется конвейер из трех регистров для суммирования значений выхода самой младшей LUT.

Для ускорения процесса разработки целесообразно воспользоваться мегафункциями. На рис. 2.43 приведена тестовая схема КИХ-фильтра с использованием мегафукнкции FIR Compiler САПР Quartus II компании Altera на последовательной и параллельной распределенной арифметике.



а)

Рис.2.42. Структура КИХ-фильтра 8 отводов 8 бит на распределенной параллельной арифметике: а) с несимметричными коэффициентами; б) с симметричными



б)

Рис.2.42. Структура КИХ-фильтра 8 отводов 8 бит на распределенной параллельной арифметике: а) с несимметричными коэффициентами; б) с симметричными (продолжение)

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. 2.44 и рис.2.45 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 4 отвода с использованием мегафункции FIR Compiler показан в табл.2.4.

Анализ табл.2.4 показывает, что при числе отводов равным 4 существенной разницы между последовательной и параллельной арифметике 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 арифметики используется параллельный векторный перемножитель.

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

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

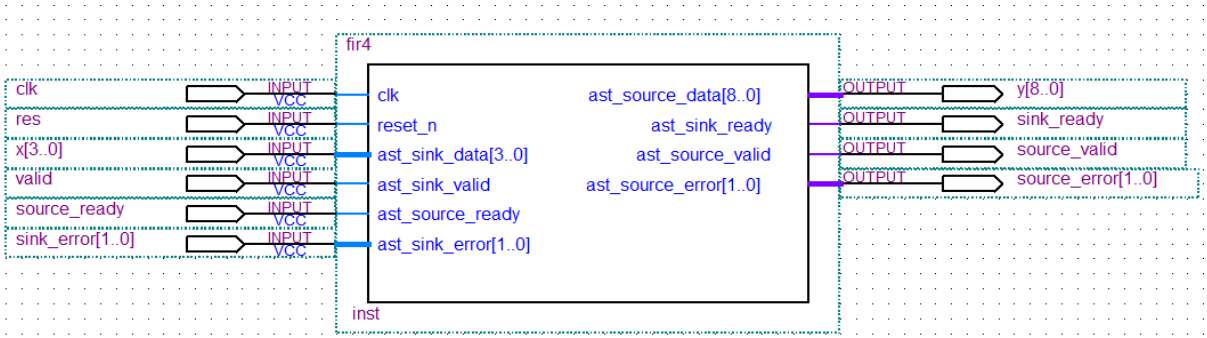


Рис.2.43.Тестовая схема КИХ-фильтра с использованием мегафукнкции FIR Compiler на последовательной и параллельной распределенной арифметике

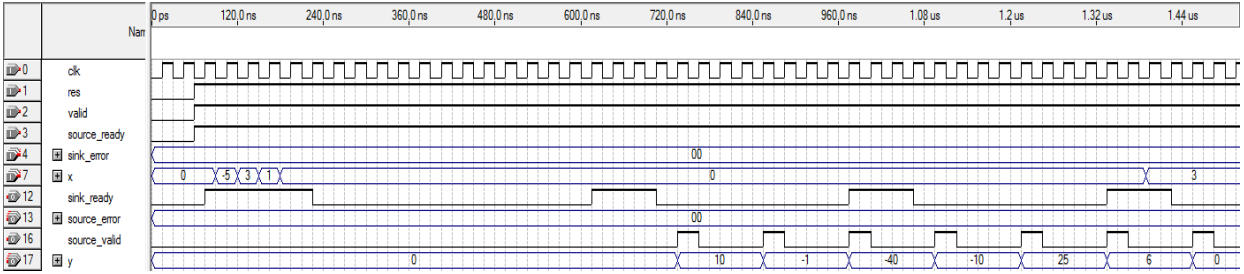


Рис.2.44. Временные диаграммы работы КИХ-фильтра с использованием последовательной распределенной арифметики на мегафункции FIR Compiler

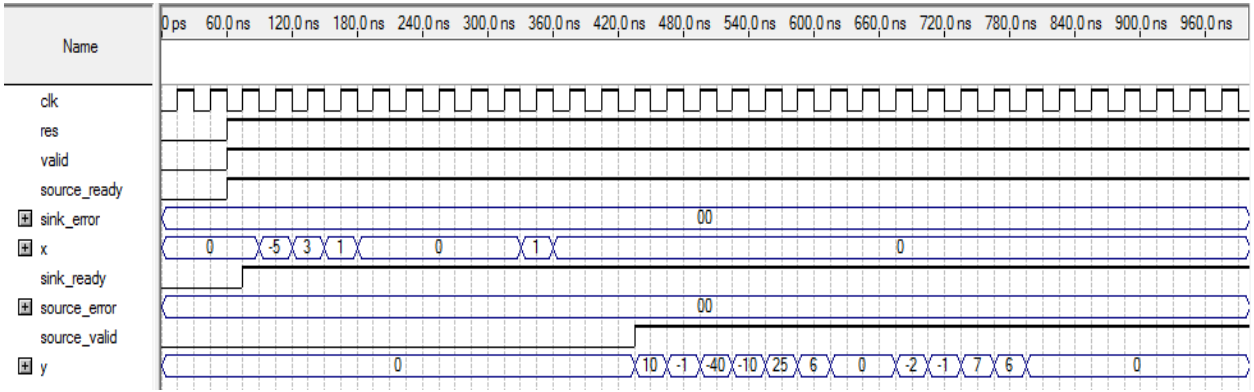


Рис.2.45. Временные диаграммы работы КИХ-фильтра с использованием параллельной распре деленной арифметики на мегафункции FIR Compiler

Таблица 2.4

Анализ задействованных ресурсов ПЛИС серии Stratix III при реализации КИХ-фильтров на 4 отвода с использованием мегафункции 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. ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ АВТОМАТОВ НА ЯЗЫКЕ VHDL ДЛЯ РЕАЛИЗАЦИИ В БАЗИСЕ ПЛИС

3.1. Проектирование цифровых автоматов Мура, Мили по диаграммам переходов

В ряде случаев автоматная модель (описание) устройства позволяет получить быструю и эффективную реализацию последовательностного устройства. Обычно рассматривают два типа автоматов – автомат Мили (Mealy) и Мура (Moore). Конечные автоматы широко используются в различных цифровых системах и устройствах, особенно в контроллерах. Выход автомата Мура является функцией только текущего состояния, выход автомата Мили – функция как текущего состояния, так и начального внешнего воздействия. Обычно конечный автомат состоит из трех основных частей:

Регистр текущего состояния. Этот регистр представляет собой набор тактируемых D-триггеров, синхронизируемых одним синхросигналом, используемый для хранения кода текущего состояния автомата. Для автомата с n состояниями требуется $Log_2(n)$ триггеров;

Логика переходов. Конечный автомат может находиться в каждый конкретный момент времени только в одном состоянии. Каждый тактовый импульс вызывает переход автомата из одного состояния в другое. Правила перехода определяются комбинационной схемой, называемой логикой переходов. Следующее состояние определяется как функция текущего состояния и входного воздействия;

Логика формирования выхода. Выход цифрового автомата обычно определяется как функция текущего состояния и исходной установки (в случае автомата Мили). Формирование выходного сигнала автомата определяется с помощью логики формирования выхода.

В качестве примера рассмотрим проектирование простейшего синхронного автомата, который формирует два

неперекрывающихся импульса Out1 и Out2 в ответ на появление сигнала Run на входе автомата (рис.3.1, а). Полностью синхронный конечный автомат использует регистры для фиксации всех выходных сигналов управления и состояний, а также для асинхронных входных сигналов. Следует заметить, что синхронные конечные автоматы по быстродействию уступают асинхронным конечным автоматам.

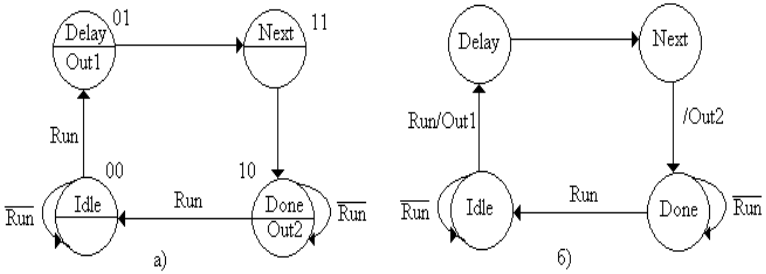


Рис.3.1. Граф-автомат проектируемого устройства

Метка, расположенная в каждом круге выше линии, - это имя состояния, а метки ниже линий - это выходные сигналы, которые выдаются, когда данное состояние активно. “Дуги”, которые возвращаются в то же самое состояние, - это переходы, которые работают по умолчанию. Эти дуги будут иметь истинные значения только в случае, когда не будет истинных значений других условий переходов. Каждое условие перехода из состояния в состояние, имеет соответствующее логическое условие, которое должно выполняться, чтобы конечный автомат мог перейти в следующее состояние.

Автомат принимает четыре состояния: Idle, Delay, Next, Done (рис.3.1, а). Воспользуемся методом двоичного кодирования состояний, который обеспечивает высокую степень кодирования последовательности состояний. Для кодирования состояний потребуется два триггера. Запишем булевы логические уравнения:

$$Idle = \overline{S0} * \overline{S1}; \quad Delay = \overline{S1} * S0; \; Next = S1 * S0;$$

$$Done = S1 * \overline{S0}; \quad S0 := (Idle, Run) + Delay;$$

$$S1 := Delay + Next + (Done, \overline{Run});$$

$$Out1 := Idle, Run; \quad Out2 := Next.$$

Символ = обозначает комбинационную схему, ответственную за переход по состояниям, а символ := обозначает триггерный выход, необходимый для хранения кода текущего состояния автомата и выходных сигналов. Схема, построенная по булевым уравнениям в САПР ПЛИС Quartus II фирмы Altera, показана на рис.3.2.

На рис.3.3 показаны временные диаграммы работы автомата. Уравнение для выходного сигнала Out1 представляет собой функцию, как состояния, так и входного сигнала Run. Конечный автомат с таким видом стробирования выходов называется автоматом Мили. Уравнение для выходного сигнала Out2 записывается как функция только состояния автомата, что соответствует структуре автомата Мура. Для автоматов Мили удобно представлять диаграммы в другом виде (рис.3.1, б).

Рассмотрим проектирование автомата Мили (пример 1) на языке VHDL с высокой степенью кодирования последовательностей состояний (двоичное кодирование). На рис.3.4 показана диаграмма переходов автомата (граф - автомат). На рис.3.5 представлена тестовая схема (“черный ящик”, функционирующий согласно описанию на языке VHDL) проектируемого автомата.

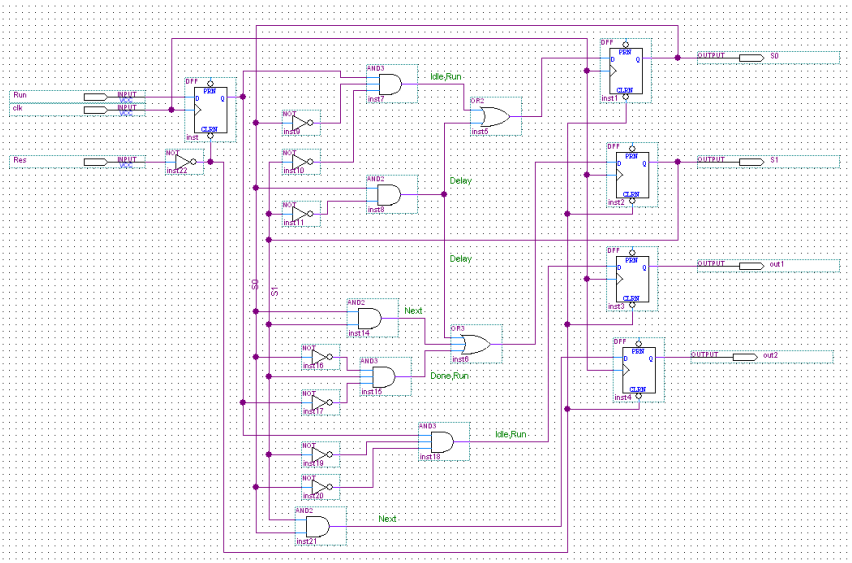


Рис.3.2. Схема конечного автомата, построенного по логическим уравнениям

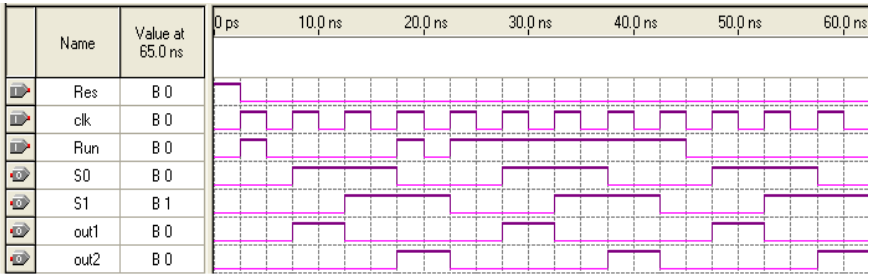


Рис.3.3. Временные диаграммы работы конечного автомата

Каждое состояние конечного автомата кодируется с использованием трех триггеров, представляющих двоичные значения в диапазоне от 000 до 0100. Временные диаграммы работы проектируемого автомата Мили, переход по состояниям ST0, ST1, ST2, ST1, показаны на рис.3.6. Используется трехпроцессорный шаблон. На рис.3.7 показан переход по

состояниям ST0, ST2, ST3, ST4, а на рис.3.8 - переход по состояниям ST0, ST4, ST0, ST1.

```
LIBRARY ieee; USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY mealy IS
        PORT(clock, reset : IN STD_LOGIC;
                data_in : IN STD_LOGIC_VECTOR(1 downto 0);
                data_out : OUT STD_LOGIC);
END mealy;
ARCHITECTURE a OF mealy IS
TYPE state_values IS (st0, st1, st2, st3, st4);
signal state, next_state: state_values;
BEGIN
-- регистерный блок
statereg: process(clock,reset)
begin
        if (reset = '0') then state<=st0;
                elsif (clock'event and clock='1') then
                                state<=next_state;
        end if;
end process statereg;
-- комбинаторный блок (логика переходов)
process(state, data_in)
begin
case state is
when st0=>
case data_in is
when "00"=>next_state<=st0;
when "01"=>next_state<=st4;
when "10"=>next_state<=st1;
when "11"=>next_state<=st2;
when others => next_state<=st0;
end case;
when st1=>
case data_in is
when "00"=>next_state<=st0;
when "10"=>next_state<=st2;
```

```
when others => next_state<= st1;
end case;
when st2=>
case data_in is
when "00"=>next_state<=st1;
when "01"=>next_state<=st1;
when "10"=>next_state<=st3;
when "11"=>next_state<=st3;
when others => next_state<= st2;
end case;
when st3=>
case data_in is
when "01"=>next_state<=st4;
when "11"=>next_state<=st4;
when others => next_state<=st3;
end case;
when st4=>
case data_in is
when "11"=>next_state<=st4;
when others => next_state<=st0;
end case;
when others => next_state<=st0;
end case;
end process;
-- логика формирования выхода
process (state, data_in)
begin
case state is
when st0=>
case data_in is
when "00"=>data_out<='0';
when others => data_out<='1';
end case;
when st1=>data_out<='0';
when st2=>
case data_in is
when "00"=>data_out<='0';
when "01"=>data_out<='0';
```

```
when others => data_out<='1';
end case;
when st3=>data_out<='1';
when st4 =>
case data_in is
when "10"=>data_out<='1';
when "11"=>data_out<='1';
when others => data_out<='0';
end case;
when others => data_out<='0';
end case;
end process;
end a;
```

Пример 1. Код языка VHDL автомата Мили

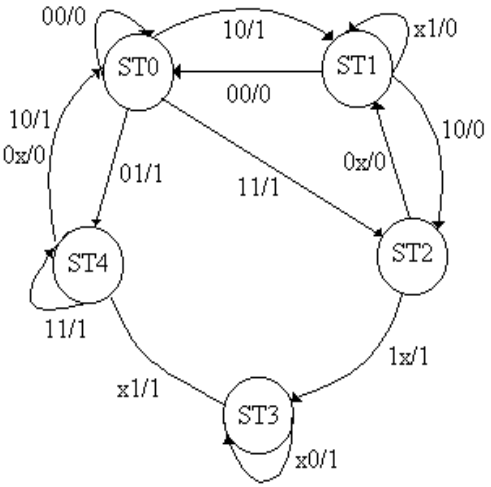


Рис.3.4. Граф переходов проектируемого автомата Мили

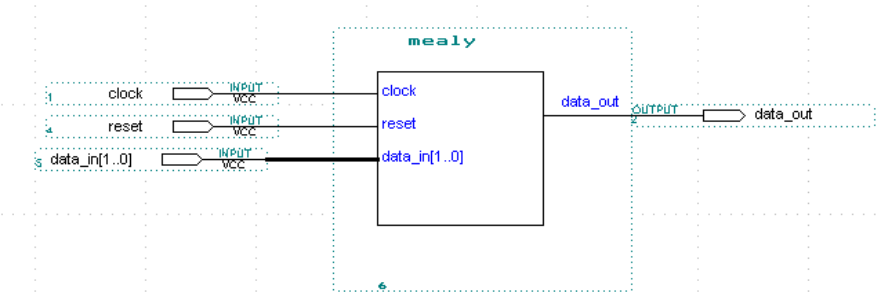


Рис.3.5. Тестовая схема проектируемого автомата Мили

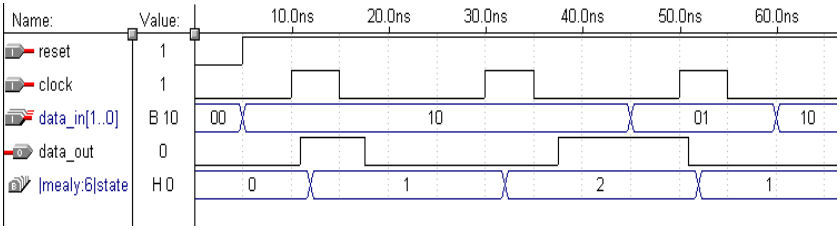


Рис.3.6. Временная диаграмма работы проектируемого автомата Мили (переход по состояниям ST0, ST1, ST2, ST1)

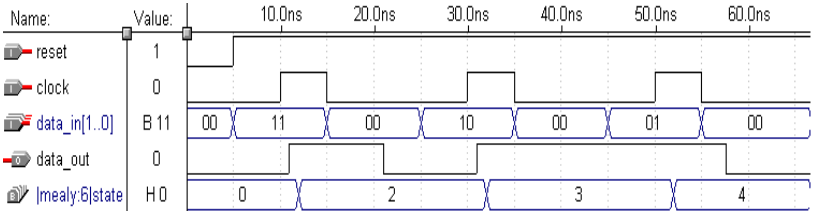


Рис.3.7. Временная диаграмма работы проектируемого автомата Мили (переход по состояниям ST0, ST2, ST3, ST4)

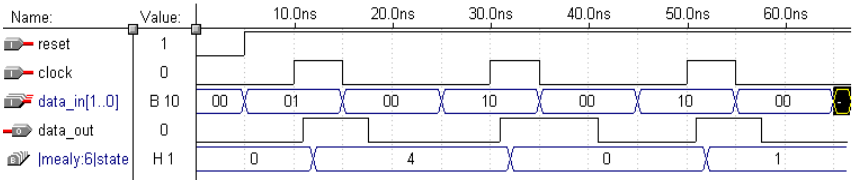


Рис.3.8. Временная диаграмма работы проектируемого автомата Мили (переход по состояниям ST0, ST4, ST0, ST1)

Рассмотрим проектирование автомата Мура (пример 2) на языке VHDL. На рис.3.9 показана временная диаграмма работы проектируемого автомата.

```
LIBRARY ieee; USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY moore IS
PORT(clock,reset        : IN STD_LOGIC;
        data_in         : IN STD_LOGIC_VECTOR(1 downto 0);
        data_out        : OUT STD_LOGIC);
END moore;
ARCHITECTURE a OF moore IS
TYPE state_values IS (st0, st1, st2, st3, st4);
signal state, next_state: state_values;
BEGIN
-- блок регистров
statereg: process(clock,reset)
begin
if (reset = '0') then state<=st0; elsif (clock'event and clock='1') then
state<=next_state; end if;
end process statereg;
-- комбинаторный блок (логика переходов)
process(state, data_in)
begin
case state is
when st0=>
case data_in is
when "00"=>next_state<=st0;
when "01"=>next_state<=st4;
when "10"=>next_state<=st1;
when "11"=>next_state<=st2;
when others => next_state<=st0;
end case;
when st1=>
case data_in is
when "00"=>next_state<=st0;
when "10"=>next_state<=st2;
when others => next_state<= st1;
```

```
end case;
when st2=>
case data_in is
when "00"=>next_state<=st1;
when "01"=>next_state<=st1;
when "10"=>next_state<=st3;
when "11"=>next_state<=st3;
when others => next_state<= st2;
end case;
when st3=>
case data_in is
when "01"=>next_state<=st4;
when "11"=>next_state<=st4;
when others => next_state<=st3;
end case;
when st4=>
case data_in is
when "11"=>next_state<=st4;
when others => next_state<=st0;
end case;
when others => next_state<=st0;
end case; end process;
-- логика формирования выхода
process (state)
begin
case state is
when st0=> data_out<='1';
when st1=> data_out<='0';
when st2=> data_out<='1';
when st3=> data_out<='1';
when st4=> data_out<='1';
when others =>data_out<='0';
end case; end process;
END a;.
```

Пример 2. Код языка VHDL автомата Мура

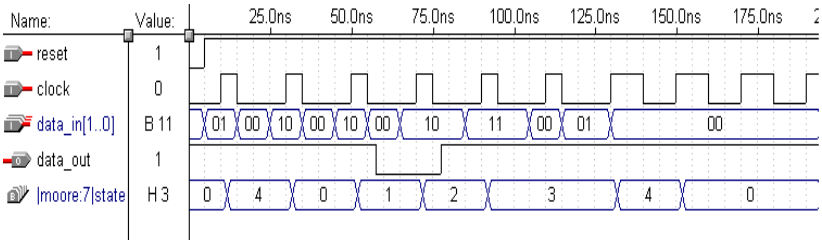


Рис.3.9. Временная диаграмма работы проектируемого автомата Мура (переход по состояниям ST0, ST4, ST0, ST1)

3.2. Кодирование с одним активным состоянием

3.2.1. Использование “ручного” способа кодирования состояний цифрового автомата

Метод one hot encoding, (OHE - кодирование с одним активным, или горячим состоянием или унитарное кодирование) получил такое название потому, что в каждый конкретный момент времени активным (hot) может быть только один триггер состояния. Применение метода OHE для ПЛИС по архитектуре ППВМ (программируемые пользователем вентильные матрицы, в зарубежной аббревиатуре FPGA) было предложено компанией High-Gate Design.

Построение конечного автомата с использованием метода OHE осуществляется по следующей методике: вначале для отображения каждого состояния автомата выделяется индивидуальный триггер, а затем организуется схема, позволяющая в каждый конкретный момент времени только одному состоянию быть активным.

Рассмотрим конечный автомат Мура, предусматривающий, семь различных состояний. Построим граф-автомат проектируемого устройства (рис.3.10). Автомат переходит из состояния в состояние по переднему фронту синхроимпульса, который отмечен “крестиком”. Иерархическая блок-схема автомата, состоящая из 7 блоков S1-

S7 и логики формирования выхода, в САПР ПЛИС Quartus II компании Altera показана на рис.3.11.

В примере имеется семь состояний (Stage1-7), каждый блок ответственен за формирования своего состояния, например блок S1 отвечает за формирование состояния 1. Все логические входы помечаются как переменные от A до E. Выходы конечного автомата носят названия Multi, Contig и Single. В данном примере состояние 1, в котором должен находиться конечный автомат при включении питания, имеет структуру триггера с двумя инверторами (схема S1, рис.3.12).

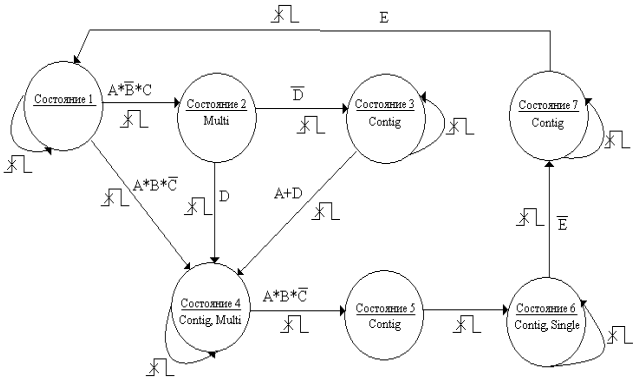


Рис.3.10. Граф-автомат проектируемого устройства

Для того чтобы конечный автомат при включении питания всегда принимал известное начальное состояние, выход триггера состояния 1 инвертируется, а чтобы обеспечить логическую непротиворечивость, входной информационный сигнал этого триггера также инвертируется. Таким образом, состояние 1 в начальный момент времени принимает значение логической единице. Для всех других состояний 2-7 используется D-триггер с асинхронным сбросом, тактируемый фронтом синхросигнала.

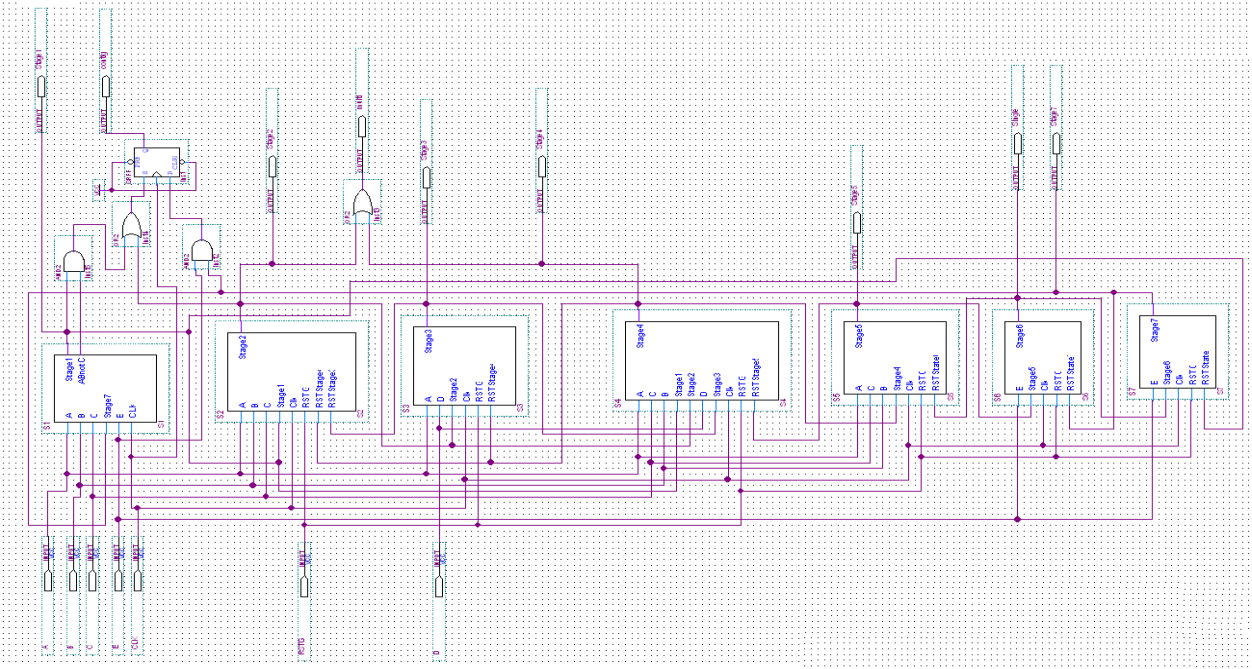


Рис.3.11. Иерархическая блок-схема автомата с кодированием по методу ОНЕ в САПР ПЛИС Quartus II

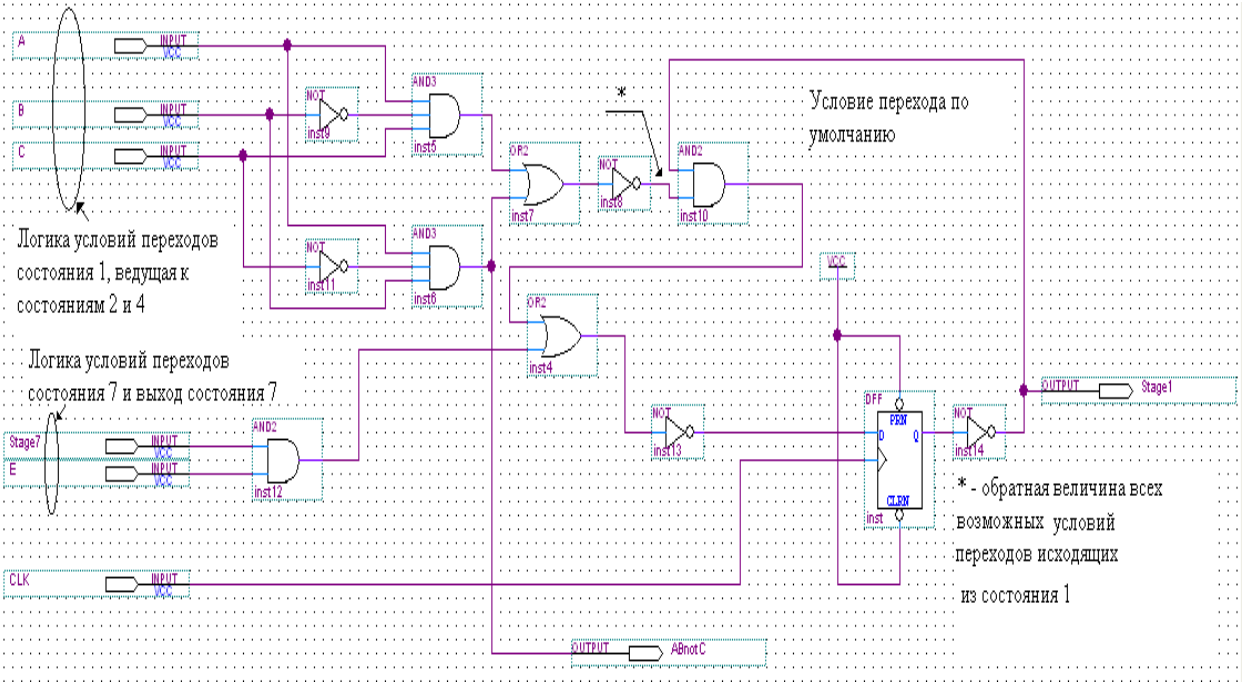


Рис.3.12. Схема для состояния 1

Автомат спроектирован так, что активный низкий уровень сигнал RSTG (глобальный сброс состояний всего автомата, кроме состояния 1) в начальный момент сбрасывает состояния 2-7 (S2-S7) в ноль, а состояние 1 будет находиться в единице. Далее сигнал RSTG должен всегда оставаться логической 1. В случае если конечный автомат все же окажется в недопустимом состоянии, например, в состояние 3, то с приходом следующего переднего фронта синхроимпульса будет установлено состояние 4. Состояние 4 сбросит состояние 3. Состояние 4 может сбросить и состояние 2. Состояние 5 сбрасывает состояние 4, состояние 6 сбрасывает состояние 5, состояние 7 сбрасывает состояние 6, а состояние 1 сбросит состояние 7. Таким образом, для правильной работы конечного автомата достаточно его один раз сбросить с помощью сигнала RSTG, а далее автомат, шагая по состояниям способен сам их сбрасывать.

После того как установлены начальные состояния, необходимо построить логику перехода в следующее состояние. Вначале подсчитывается число условий переходов, ведущих к данному состоянию, и добавляется еще один путь, если условие по умолчанию должно оставлять конечный автомат в том же самом состоянии. Далее строится логический вентиль ИЛИ с числом входов, равным числу условий переходов, определенных ранее.

Далее, для каждого входа вентиля ИЛИ строится логический вентиль И, входами которого служат предыдущие состояния и его логика условия. Если по умолчанию конечный автомат должен оставаться в том же самом состоянии, строится логический вентиль И, входами которого служат данное состояние и обратная величина всех возможных условий переходов, исходящих из данного состояния.

Чтобы определить число условий переходов для состояния 1, рассмотрим граф-автомат. Из рис.3.12 видно, что состояние 1 имеет один переход от состояния 7, когда переменная E истинна. Другой переход - это условие по умолчанию, ведущее

в состояние 1. Таким образом, состояние 1 имеет два условия переходов. После этого можно построить двухвходовой логический вентиль 2ИЛИ - с одним входом для условия перехода от состояния 7, а другим для перехода по умолчанию, чтобы оставаться в состоянии 1.

Следующий шаг - это построение логики переходов для данного вентиля 2ИЛИ. Каждый вход вентиля 2ИЛИ есть логическая функция И, предыдущего состояния и логики переходов состояния 1. Например, состояние 7 поступает на вход состояния 1, когда E имеет истинное значение. Это обеспечивается при помощи логического вентиля 2И (рис.3.12). Второй вход вентиля ИЛИ - переход по умолчанию, когда конечный автомат должен оставаться в состоянии 1. Если текущее состояние есть состояние 1, и нет условий переходов, выходящих из состояния 1, которые истинны, то конечный автомат должен оставаться в состоянии 1. Состояние 1 на диаграмме состояний имеет два исходящих условия переходов (рис.3.12).

Первый переход является действительным, когда истинно условие ($A\overline{B}C$), и ведет в состояние 2. Второй переход, ведущий в состояние 4, является действительным при истинном значении условия ($AB\overline{C}$). Логика по умолчанию - это функция И для состояния 1 обратной величины всех условий переходов, исходящих из состояния 1. Эта логическая функция реализуется с использованием вентиля 2И с инвертором на одном из входов и логических элементов, формирующих сигнал для инвертирующего входа вентиля 2И (рис.3.12). Комбинационная логика обеспечивает декодирование с учетом входных сигналов и сигнала обратной связи.

Состояние 4 не является начальным состоянием, поэтому для его представления используется D-триггер без инверторов, с входом асинхронного сброса RSTG. Триггер может быть сброшен и выходом состояния 5 (сигнал RSTState5). Имеется три входящих условия перехода и условие по умолчанию, чтобы конечный

автомат мог оставаться в состоянии 4. Поэтому на входе триггера используется вентиль 4ИЛИ (схема S4, рис.3.13).

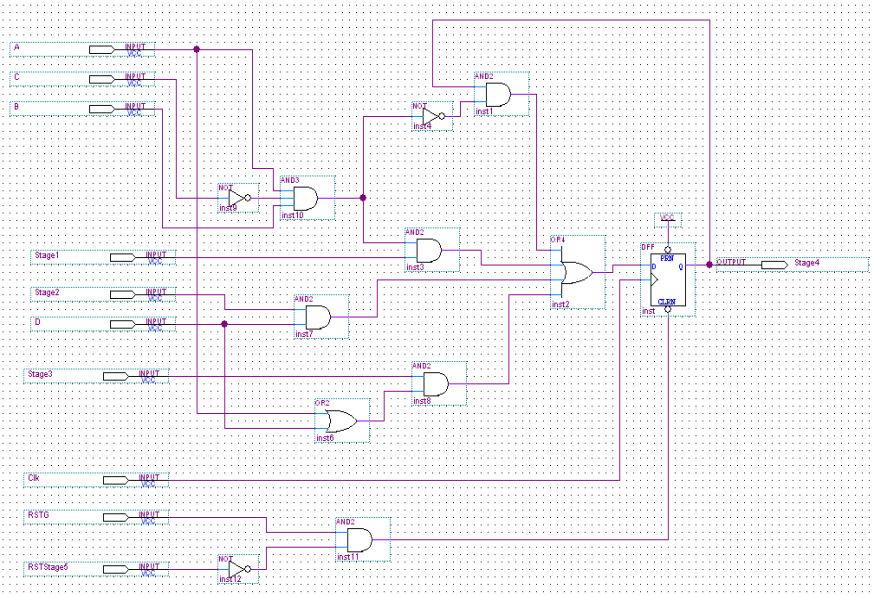


Рис.3.13. Схема S4 для состояния 4

Первое условие перехода исходит из состояния 3. В соответствии с изложенными выше правилами необходимо построить функцию И для состояния 3 и логику условия, которая имеет вид $A + D$ (рис.3.13).

Следующее условие перехода исходит из состояния 2, оно требует логической функции И для состояния 2 и переменной D. Последнее условие перехода для состояния 4 - от состояния 1. Выход состояния 1 должен пройти через схему 2И с логикой его условия перехода - логическим произведением $AB\overline{C}$ (рис.3.13).

Далее нужно построить логику, обеспечивающую сохранение состояния 4, когда ни одно из условий переходов, исходящих из состояния 4, не имеет истинного значения. Переход, исходящий из состояния 4, является действительным,

когда логическое произведение $AB\overline{C}$ истинно. Следовательно, необходимо пропустить состояние 4 через вентиль И с обратной величиной произведения $AB\overline{C}$. Это необходимо для поддерживания триггера в высоком уровне, пока не произойдет действительный переход в следующее состояние. В логике перехода по умолчанию используется вентиль 2И и выход вентиля 3И с инвертором на входе C.

Состояние 2 имеет только одно условие перехода, которое приходит от состояния 1, когда произведение $A\overline{B}C$ истинно. Конечный автомат будет немедленно переходить по одному из двух переходов из состояния 2 в зависимости от значения сигнала D. Состояние 3, подобно состояниям 1 и 4, имеет переход по умолчанию, и для управления входом D-триггера используется комбинация сигналов A, D, состояния 2 и состояния 3. Состояние 5 управляет состоянием 6 без всяких условий. Конечный автомат ждет в состоянии 6, пока переменная E не переключится в низкий уровень, прежде чем перейти в состояние 7. В состоянии 7 конечный автомат ждет переключения переменной E в истинное значение, после чего переходит в состояние 1.

После описания всей логики переходов по состояниям, следующим этапом является описание выходной логики. В примере используются три выходных сигнала - Multi, Contig и Single, - каждый из которых относится к одной из трех основных категорий выходных сигналов:

1. Выходные сигналы, формируемые в одном состоянии. Примером может служить выходной сигнал Single, формируемый только в состоянии 6, т.е. выходным сигналом является выход триггера.

2. Выходные сигналы, формируемые во многих смежных состояниях. Например, выходной сигнал Contig, который формируется в состояниях 3-7, хотя имеется ветвь для состояния 2.

3. Выходные сигналы, формируемые по многим несмежным состояниям. Здесь обычно оптимальное решение - это простое декодирование активных состояний. Например, сигнал Multi, который формируется для состояний 2 и 4.

Для формирования логики выходного сигнала Multi используется декодирование состояний 2 и 4 при помощи вентиля 2ИЛИ. Каждый раз, когда конечный автомат окажется в одном из этих состояний, будет сформирован активный сигнал Multi. Для декодирования выходных сигналов для смежных состояний используется синхронный RS-триггер. RS-триггер устанавливается при входе в смежное состояние и сбрасывается при выходе (рис.3.11). Временная диаграмма проектируемого автомата представлена на рис.3.14.

Для размещения автомата выберем ПЛИС по архитектуре ППВМ APEX20K (EP20K30ETC144-1). Архитектура ПЛИС семейства APEX20K сочетает в себе достоинства ППВМ ПЛИС с их таблицами перекодировок (LUT). После компиляции проекта оказалось задействовано 20 логических элементов, 8 триггеров. Моделирование проводилось без учета реальных задержек распространения сигналов в ПЛИС. С учетом реальных задержек период тактового сигнала CLK для стабильной работы автомата должен быть не менее 15 нс. Уменьшить число триггеров на один позволяет декодирование состояний 3-7 при помощи 5-входового вентиля ИЛИ. Каждый раз, когда конечный автомат окажется в одном из этих состояний, будет сформирован сигнал Conting. В этом случае и сокращается число логических элементов. Максимальная тактовая частота в обоих схемных решениях составляет $f_{MAX} = 290.02$ МГц.

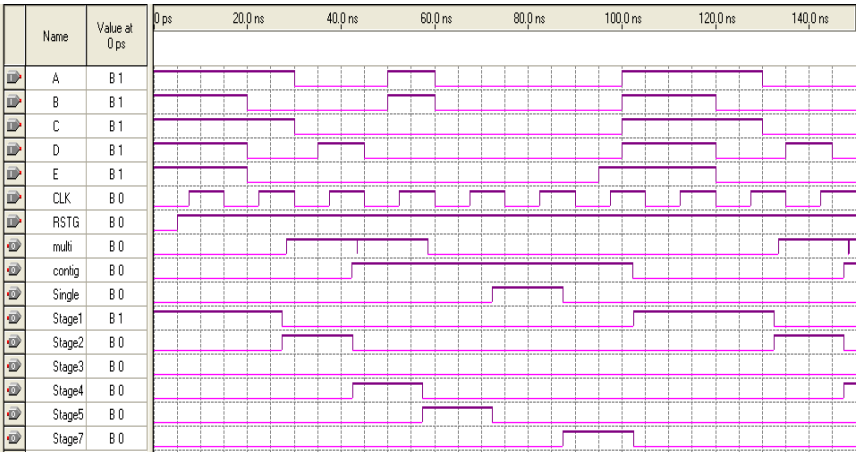


Рис.3.14. Результаты моделирования работы конечного автомата с принудительным сбросом состояний. Показаны переходы по состояниям 1-7

3.2.2. Использование различных стилей кодирования состояний цифровых автоматов на языке VHDL

Опишем функционирование данного автомата на языке описания аппаратурных средств VHDL. Xilinx рекомендует использовать кодирование конечных автоматов с использованием перечисляемого типа, т.к. в этом случае имеется возможность предоставить САПР использовать модуль логического синтеза и в зависимости от архитектуры ПЛИС самостоятельно выбирать метод кодирования.

Проектирование конечного автомата осуществим с использованием перечисляемого типа данных (Enumerated type) на языке VHDL. Перечисляемый тип – это такой тип данных, при котором количество всех возможных состояний конечно. Такой тип наиболее часто используется для обозначений состояний конечных автоматов. Любой перечисляемый тип

имеет внутреннюю нумерацию: первый элемент всегда имеет номер 0, второй - 1 и т.д.

Перечисляемый тип определяется списком (перечислением) всех возможных значений этого типа. Перечисляемые типы имеют следующий синтаксис определения:

type type_name is (enumeration_literal
{, enumeration_literal});

где type_name - идентификатор типа, а каждый enumeration_literal - либо идентификатор (enum_6), ибо литерал символа (’A’). Идентификатор является последовательностью букв, символов подчеркивания и цифр. Идентификатор должен начинаться с буквы и не может являться зарезервированным словом VHDL, таким как TYPE. Литерал символа является любым значением типа CHARACTER в одиночных кавычках. Например, FPGA Express Synopsys автоматически кодирует перечисляемые значения в битовые вектора, которые основаны на каждой позиции значения. Длина кодирующего битового вектора равна минимальному количеству бит, необходимых для кодирования номера перечисляемых значений. Например, перечисляемый тип с пятью значениями имеет трехбитовый кодирующий вектор.

В САПР Quartus II в меню Analysis&Synthesis Settings, закладка More Analysis&Synthesis Settings, выберем установку кодирования конечных автоматов State Machine Processing – one-Hot (рис.3.15). При этом установка Safe State Machine – Off (разработка автомата с учетом восстановления из неправильных состояний в случае нарушения временных требований требует дополнительной логики, поэтому эта опция отключена). Комбинационную логику автомата реализуем на таблицах перекодировок – установка LUT (для ПЛИС серии APEX2).

Пользователь может выбрать установку Auto, которая позволяет средствам синтеза автоматически выбрать для каждого конечного автомата наилучший алгоритм кодирования. В случае выбора установки User-Encoded,

средствами синтеза будет использоваться алгоритм кодирования, представленный в файле исходного описания.

В случае выбора пользователем установки Auto для ПЛИС FPGA используется метод кодирования с одним активным состоянием (модифицированный, известен в литературе как почти прямое кодирование) а для ПЛИС типа CPLD, где большое количество термов произведений, используется кодирование с минимальным количеством триггеров (minimal-bits encoding). В САПР Quarus II используется не чистый метод кодирования с одним активным состоянием, а модифицированный. Первое (начальное) состояние кодируется нулями, т.е. все триггеры сдвигового регистра устанавливаются в ноль, а последующие состояния кодируются как в обычном методе НОТ, при этом выход первого состояния всегда активен, кроме начального. Это объясняется тем, что при запуске автомата его легко установить в состояние 0000000, а не в состояние 0000001, как в классическом методе НОТ.

Модуль синтеза (компилятор) САПР Quartus II “использует патентованные перспективные эвристические алгоритмы”, позволяющие сделать такие автоматические назначения состояний, которые минимизируют логические ресурсы, нужные для реализации конечного автомата. Затем компилятор выполняет автоматически следующие функции: назначает биты, выбирая для каждого бита либо Т-триггер, либо D-триггер; присваивает значения состояний; применяет сложные методы логического синтеза для получения уравнений возбуждения.

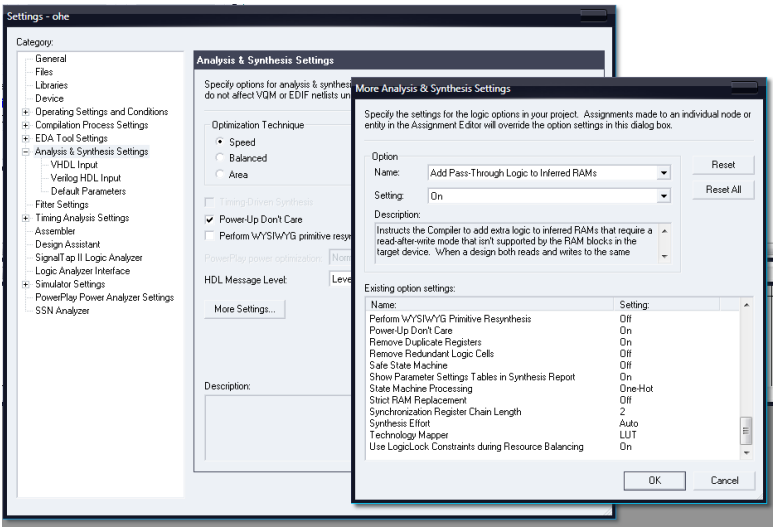


Рис.3.15. Настройки компилятора САПР Quartus II для синтеза конечного автомата

Рассмотрим двухпроцессный шаблон (рис.3.16) описания конечного автомата с использованием перечисляемого типа и настроек компилятора (State Machine Processing – one-Hot). Особенностью является использование одного сигнала state перечисляемого типа state_values для реализации логики переходов совместно с регистром текущего состояния. Второй оператор процесса используется для описания логики формирования выхода (пример 1). Для обеспечения стабильной и безотказной работы используется сброс автомата в начальное состояние (активный высокий уровень сигнала TRST). Таким образом, всегда обеспечивается инициализация автомата в начальное состояние.

Рассмотрим трехпроцессный шаблон описания работы конечного автомата (пример 2 и рис.3.17). В данном случае используются два сигнала state и next_state перечисляемого типа и три оператора Process. Стиль кодирования конечного автомата на языке VHDL, рассмотренный в примерах 1 и 2, будем называть неявным, т.к. из кода не ясно, какой метод

кодирования используется, при этом предполагается, что метод кодирования задан в настройках компилятора САПР.

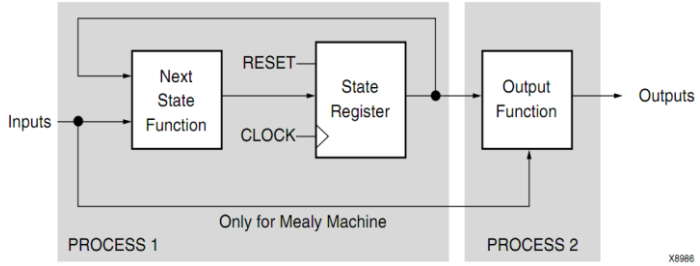


Рис.3.16. Двухпроцессный шаблон описания работы конечного автомата на языке VHDL

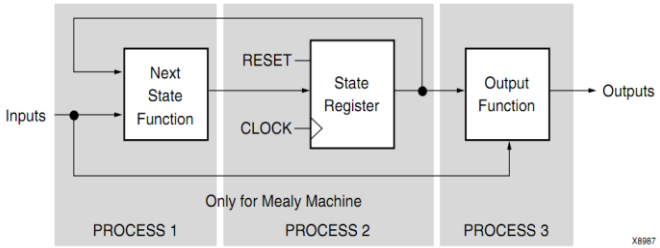


Рис.3.17. Трехпроцессный шаблон описания работы конечного автомата на языке VHDL

На рис.3.18 представлены временные диаграммы работы. Показаны переходы по состояниям 1, 4, 5, 6 и 7. По первому такту синхроимпульса и при выполнении условия $AB\overline{C}$, автомат переходит в состояние 4. В Состоянии 4 формируются выходные сигналы Contig и Multi. По второму такту синхроимпульса и по условию $AB\overline{C}$, автомат переходит в состояние 5, с формированием на выходе сигнала Multi. По третьему такту синхроимпульса автомат без всяких условий переходит в состояние 6 с формированием выходных сигналов Contig и Single.

ARCHITECTURE a OF avtOHE2 IS

ARCHITECTURE a OF avtOHE IS

```
TYPE state_values IS (Stage1, Stage2, Stage3, Stage4, Stage5, Stage6, Stage7);
signal state: state_values;
BEGIN
process(TCK,TRST)
begin
        if (TRST = '1') then state<=Stage1;
        elsif (TCK'event and TCK='1')
then
        case state is
when Stage1=>
IF (A='1' and B='0' and C='1') THEN state<=Stage2;
ELSIF (A='1' and B='1' and C='0') THEN state<=Stage4;
ELSE state<=Stage1;
END IF;
…
process (state)
begin
case state is
when Stage1=>
Multi  <= '0';
Contig <= '0';
Single  <= '0';
…
end case;
end process;
END a;
```

Пример 1. Фрагмент кода двухпроцессного шаблона описания конечного автомата на языке VHDL

```
TYPE state_values IS (Stage1, Stage2, Stage3, Stage4, Stage5, Stage6, Stage7);
signal state, next_state: state_values;
BEGIN
process(TCK,TRST)
begin
        if (TRST = '1') then state<=Stage1;
                elsif (TCK'event and TCK='1') then
        state<=next_state;
        end if;
end process;
process(state, A,B,C,D,E)
begin
case state is
when Stage1=>
IF (A='1' and B='0' and C='1') THEN next_state<=Stage2;
ELSIF (A='1' and B='1' and C='0') THEN next_state<=Stage4;
ELSE next_state<=Stage1;
END IF;
…
process (state)
begin
case state is
when Stage1=>
Multi  <= '0'; Contig <= '0'; Single <= '0';
…
end case;
end process;
END a;
```

Пример.2 Фрагмент кода трехпроцессного шаблона описания конечного автомата на языке VHDL

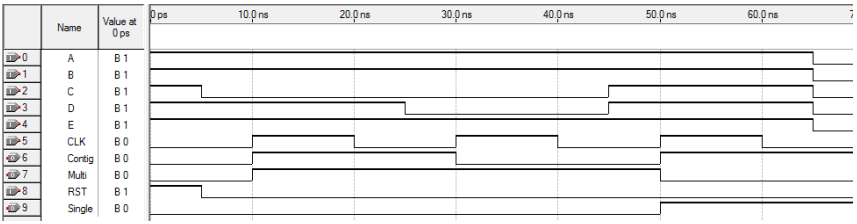
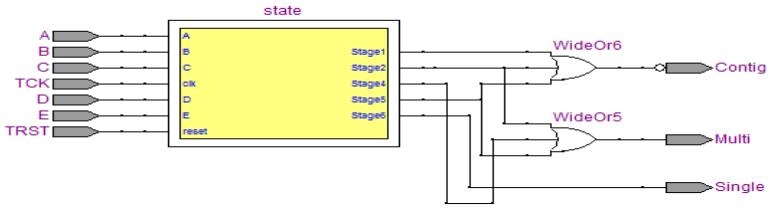
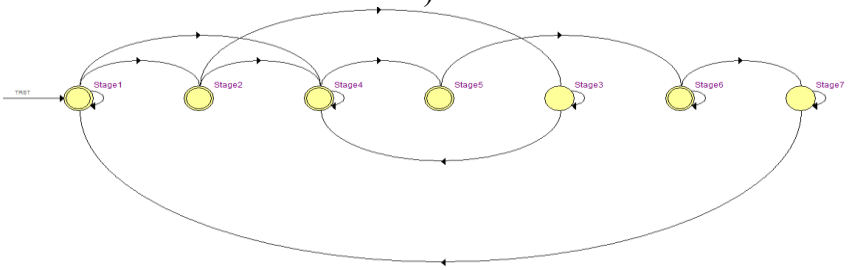


Рис.3.18. Временная диаграмма, конечного автомата, описанного на VHDL. Показаны переходы по состояниям 1, 4, 5, 6 и 7

Использование перечисляемого типа с настройками one-Hot позволяет модулю синтеза оптимизировать проектируемый автомат. В этом случае доступен просмотровщик состояний конечного автомата (State Machine Viewer). На рис.3.19 *а* показан синтезированный цифровой автомат по VHDL-коду в САПР Quartus II с логикой формирования выходов с использованием трехпроцессного шаблона языка VHDL. На рис.3.19 *б* граф-автомат восстановленный из VHDL-кода. Булевы уравнения для логики переходов по состояниям и для условий по умолчанию показаны на рис.3.19 *в*. Рис.3.19 *г* демонстрирует таблицу переходов, характерную для метода кодирования с одним активным состоянием. На рис.3.20 показана схема автомата, готовая для размещения в базисе ПЛИС серии Sratix III на которой видна структура 7-разрядного сдвигового регистра, блоки комбинационной логики и то, что выходы триггеров могут использоваться как непосредственные выходы автомата (выход single) и как следствие, применение дополнительной комбинационной логики на выходе не требуется.



а)



б)

	Source State	Destination State	Condition
1	Stage1	Stage1	(!A) + (A).(!B).(!C) + (A).(B).(C)
2	Stage1	Stage2	(C).(A).(!B)
3	Stage1	Stage4	(A).(B).(!C)
4	Stage2	Stage3	(!D)
5	Stage2	Stage4	(D)
6	Stage3	Stage3	(!A).(!D)
7	Stage3	Stage4	(!A).(D) + (A)
8	Stage4	Stage4	(!A) + (A).(!B) + (A).(B).(C)
9	Stage4	Stage5	(A).(B).(!C)
10	Stage5	Stage6	
11	Stage6	Stage6	(E)
12	Stage6	Stage7	(!E)
13	Stage7	Stage1	(E)
14	Stage7	Stage7	(!E)

в)

	Name	Stage7	Stage6	Stage5	Stage4	Stage3	Stage2	Stage1
1	Stage1	0	0	0	0	0	0	0
2	Stage2	0	0	0	0	0	1	1
3	Stage3	0	0	0	0	1	0	1
4	Stage4	0	0	0	1	0	0	1
5	Stage5	0	0	1	0	0	0	1
6	Stage6	0	1	0	0	0	0	1
7	Stage7	1	0	0	0	0	0	1

г)

Рис.3.19. Синтез конечного автомата по VHDL-коду в САПР Quartus II (трехпроцессный шаблон): а) управляющий автомат с логикой формирования выходов; б) граф-автомат восстановленный из VHDL-кода; в) условия переходов по состояниям; г) таблица переходов, демонстрирующая использование метода ОНЕ

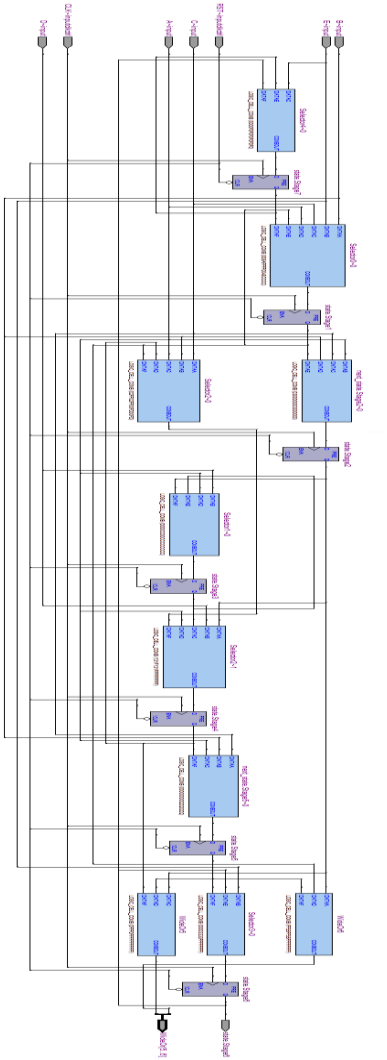


Рис.3.20. Схема конечного автомата с использованием метода ONE, готовая для размещения в базисе ПЛИС серии Stratix III (трехпроцессный шаблон)

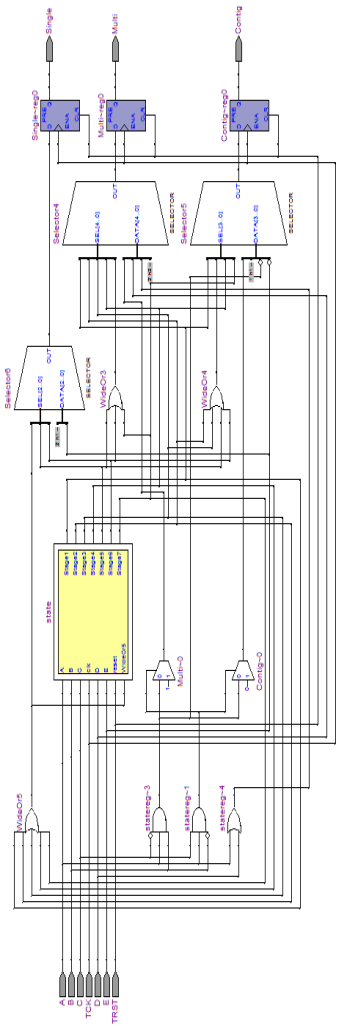


Рис.3.21. Схема конечного автомата с использованием метода ONE (однопроцессный шаблон)

Рассмотрим использование однопроцессного шаблона (пример 3) и перечисляемого типа. В этом случае число триггеров увеличивается с 7 до 10, т.к. включение логики формирования выхода в оператор case располагающегося внутри блока регистров конечного автомата, приводит к тому, что выходы Multi, Contig и Single становятся регистерными (рис.3.21).

```
BEGIN
process(TCK,TRST)
begin
if (TRST = '1') then state<=Stage1;
                     Multi  <= '0';
                     Contig <= '0';
                     Single <= '0';
elsif (TCK'event and TCK='1')
then
case state is
when Stage1=>
IF (A='1' and B='0' and C='1') THEN
state<=Stage2;
Multi  <= '1'; Contig <= '0'; Single <= '0';
ELSIF (A='1' and B='1' and C='0') THEN
state<=Stage4;
Multi  <= '1'; Contig <= '1'; Single <= '0';
ELSE   state<=Stage1;
Multi  <= '0'; Contig <= '0'; Single <= '0';
END IF;
```

Пример.3. Фрагмент кода однопроцессного шаблона описания конечного автомата на языке VHDL

Рассмотрим варианты – двоичное кодирование и кодирование по методу ONE определено явно в коде языка VHDL с использованием двухпроцессного шаблона и перечисляемого типа данных. В этих случаях пользователь сам кодирует состояния Stage1 - Stage7: либо двоичное кодирование 001, 010, 011 и до 111 (пример 3) или кодирование с одним

активным состоянием 0000001, 0000010, 0000100 и до 10000000 (пример 4).

Отменим автоматическое кодирование перечисления и определим свои собственные коды с помощью атрибута enum_encoding. Атрибут enum_encoding должен быть строкой (STRING), содержащей набор векторов, по одному для каждого перечисляемого литерала в соответствующем типе. Кодирующий вектор определяется символами ’0’,’1’,’D’,’U’ и ’Z’, разделенными пробелами. Первый вектор в строке атрибута определяет код для первого перечисляемого литерала, второй вектор - для второго литерала и т.д. Атрибут enum_encoding должен следовать сразу же за объявлением типа.

При двоичном кодировании синтезируется конечный автомат с использованием 3 триггеров для реализации 7 состояний - 1 триггер и 1 двухразрядный регистр (рис.3.22). При кодировании с одним активным состоянием - 6 триггеров (рис.3.23). При кодировании состояний пользователем редактор состояний конечного автомата САПР Quartus II (State Machine Viewer) недоступен.

Рассмотрим проектирование конечного автомата с использованием редактора состояний САПР Quartus II (рис.3.24 и рис.3.25). Условия переходов по умолчанию специально не заданы, для того что бы посмотреть как модуль синтеза справится с этой задачей. Рис.3.25, *а* показывает, что модуль синтеза самостоятельно доопределил условия переходов по умолчанию, такие переходы как и в исходном задании отсутствуют у состояний 2 и 5. Код языка VHDL извлеченный из граф-автомата в автоматическом режиме, демонстрирует пример 5.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY avtOHE2 IS
PORT(
A,B,C,D,E,TRST,TCK :IN
STD_LOGIC;
Multi,Contig,Single:
OUTSTD_LOGIC);
END avtOHE2;
ARCHITECTURE a OF avtOHE2 IS
TYPE state_values IS (Stage1, Stage2,
Stage3, Stage4, Stage5, Stage6,
Stage7);
attribute enum_encoding: string;
attribute enum_encoding of
state_values: type is "001 010 011 100
101 110 111";
signal state: state_values;
BEGIN
statereg: process(TCK,TRST)
begin
if (TRST = '1') then state<=Stage1;
elsif (TCK'event and TCK='1')
then
case state is
when Stage1=>
IF (A='1' and B='0' and C='1') THEN
state<=Stage2;
ELSIF (A='1' and B='1' and C='0')
THEN state<=Stage4;
ELSE state<=Stage1;END IF;
…
process (state)
begin
case state is
when Stage1=>
Multi <= '0';
Contig <= '0';
Single <= '0';
….
end case;
end process;
END a;
```

Пример 3. Фрагмент двухпроцессного шаблона описания конечного автомата на языке VHDL (явное использование стиля двоичного кодирования, атрибут enum_encoding)

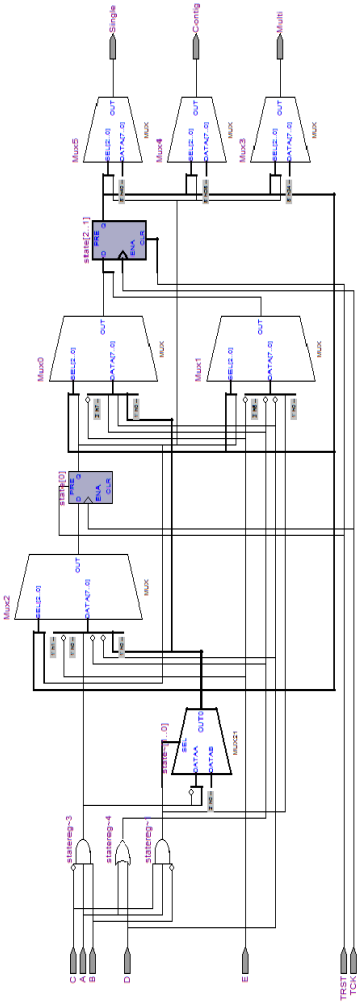


Рис.3.22. Синтезированный конечный автомат с использованием двоичного кодирования (явное использование стиля кодирования, атрибут enum_encoding)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY OHE2 IS
PORT(
A,B,C,D,E,TRST,TCK:
INSTD_LOGIC;
Multi,Contig,Single:          OUT
          STD_LOGIC);
END OHE2;
ARCHITECTURE a OF OHE2 IS
TYPE state_values IS (Stage1,
Stage2, Stage3, Stage4, Stage5,
Stage6, Stage7);
attribute enum_encoding: string;
attribute enum_encoding of
state_values: type is "0000001
0000010 0000100 0001000
0010000 0100000 1000000";
signal state: state_values;

BEGIN
statereg: process(TCK,TRST)
begin
if (TRST = '1') then state<=Stage1;
elsif (TCK'event and TCK='1')
then
case state is
when Stage1=>
IF (A='1' and B='0' and C='1')
THEN state<=Stage2;
ELSIF (A='1' and B='1' and C='0')
THEN state<=Stage4;
ELSE state<=Stage1;
END IF;
….
```

Пример. 4. Фрагмент кода двухпроцессорного шаблона описания конечного автомата на языке VHDL (явное использование стиля кодирования с одним активным состоянием, атрибут enum_encoding)

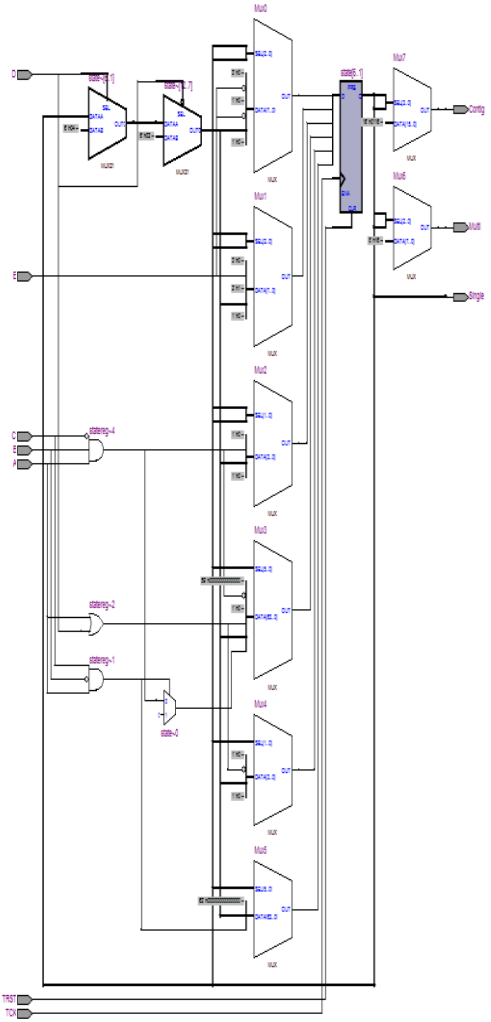


Рис.3.23. Синтезированный конечный автомат с использованием кодирования с одним активным состоянием (явное использование стиля кодирования, атрибут enum_encoding)

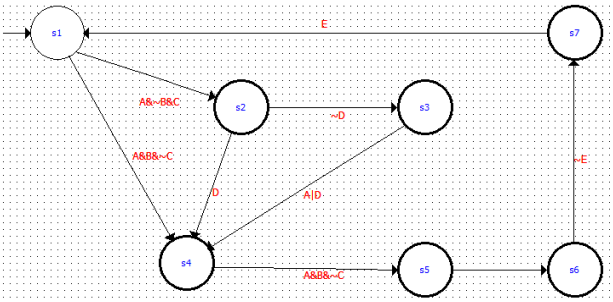
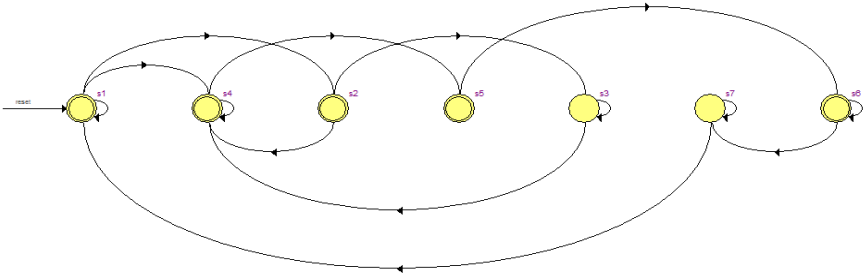


Рис.3.24. Граф-автомат, разработанный с помощью редактора состояний в САПР Quartus II



а)

	Source State	Destination State	Condition
1	s1	s1	(!B).(!A) + (!B).(A).(!C) + (B).(!A) + (B).(A).(C)
2	s1	s4	(A).(!C).(B)
3	s1	s2	(!B).(A).(C)
4	s4	s4	(!A) + (A).(!C).(!B) + (A).(C)
5	s4	s5	(!C).(B).(A)
6	s5	s6	
7	s6	s6	(E)
8	s6	s7	(!E)
9	s7	s1	(E)
10	s7	s7	(!E)
11	s2	s4	(D)
12	s2	s3	(!D)
13	s3	s4	(!A).(D) + (A)
14	s3	s3	(!A).(!D)

б)

	Name	s3	s2	s7	s6	s5	s4	s1
1	s1	0	0	0	0	0	0	0
2	s4	0	0	0	0	0	1	1
3	s5	0	0	0	0	1	0	1
4	s6	0	0	0	1	0	0	1
5	s7	0	0	1	0	0	0	1
6	s2	0	1	0	0	0	0	1
7	s3	1	0	0	0	0	0	1

в)

Рис.3.25. Синтез конечного автомата по VHDL-коду извлеченного из граф-автомата созданного с помощью редактора состояний в САПР Quartus II: а) граф-автомат, восстановленный из VHDL-кода; б) условия переходов по состояниям; в) таблица переходов, демонстрирующая использование метода ОНЕ

```
ARCHITECTURE BEHAVIOR OF SM2 IS
TYPE type_fstate IS (s1,s4,s5,s6,s7,s2,s3);
SIGNAL fstate : type_fstate;
SIGNAL reg_fstate : type_fstate;
BEGIN
   PROCESS (clock,reset,reg_fstate)
   BEGIN
      IF (reset='1') THEN
         fstate <= s1;
      ELSIF (clock='1' AND clock'event) THEN
         fstate <= reg_fstate;
      END IF;
   END PROCESS;
   PROCESS (fstate,A,B,C,E,D)
   BEGIN
      Multi <= '0'; Contig <= '0'; Single <= '0';
      CASE fstate IS
         WHEN s1 =>
      IF ((((A = '1') AND (B = '1')) AND NOT((C = '1')))) THEN
               reg_fstate <= s4;
     ELSIF ((((A = '1') AND NOT((B = '1'))) AND (C = '1'))) THEN
               reg_fstate <= s2;
            ELSE
               reg_fstate <= s1;
            END IF;
          …
         WHEN OTHERS =>
          Multi <= 'X'; Contig <= 'X'; Single <= 'X';
      END CASE;
   END PROCESS;
END BEHAVIOR;
```

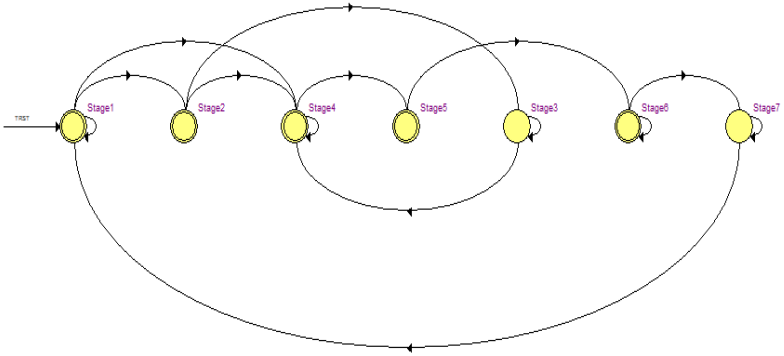
Пример.5. Фрагмент автоматически извлеченного кода языка VHDL из граф-автомата

Рассмотрим, чем отличается использование атрибута syn_encoding от enum_encoding на примере двухпроцессного шаблона с использованием простейшего двоичного кодирования (пример 3). В случае применения атрибута enum_encoding при кодировании состояний пользователем, компилятор не использует специальные методы логического

синтеза, применяемые к конечным автоматам, например, для минимизации состояний и получения уравнений возбуждения, а использует методы синтеза булевой логики. Атрибут syn_encoding (рекомендуется использовать в САПР Quartus II) переопределяет модуль синтеза на использование собственной кодировки состояний автомата. В процессе синтеза, компилятор добавляет дополнительную логику при минимизации для исключения попадания автомата в запрещенные состояния. В этом случае доступен редактор состояний (рис,3.26 *а*). Логика переходов и таблица переходов показана на рис.3.26, *б* и рис.3.26, *в*. На рис.3.26, *в,* видим не двоичную последовательность кодирования состояний "001 010 011 100 101 110 111" полученную путем увеличения содержимого регистра состояний на единицу (простейшее кодирование) а измененную "000 011 010 101 100 111 110" (кодирование с разбиением). Что говорит о том, что если существуют неиспользуемые состояния (если $s < 2^n$, $n = \lceil Log_2 s \rceil$, где s - число состояний автомата, а n – число триггеров), то компилятор выбирает “лучшие” из имеющихся n-разрядных целых чисел. Для атрибута syn_encoding предусмотрены следующие ключевые слова "default", "sequential", "gray", "johnson", "one-hot", "compact" а для атрибута enum_encoding только первые пять (пример 6).

```
TYPE state_values IS (Stage1, Stage2, Stage3, Stage4, Stage5,
Stage6, Stage7);
attribute syn_encoding: string;
attribute syn_encoding of state_values: type is "one-hot";
```

Пример 6. Кодирование состояний с использованием метода ОНЕ и атрибута syn_encoding



а)

	Source State	Destination State	Condition
1	Stage1	Stage1	(!A) + (A).(!B).(!C) + (A).(B).(C)
2	Stage1	Stage2	(C).(A).(!B)
3	Stage1	Stage4	(A).(B).(!C)
4	Stage2	Stage3	(!D)
5	Stage2	Stage4	(D)
6	Stage3	Stage3	(!A).(!D)
7	Stage3	Stage4	(!A).(D) + (A)
8	Stage4	Stage4	(!A) + (A).(!B) + (A).(B).(C)
9	Stage4	Stage5	(A).(B).(!C)
10	Stage5	Stage6	
11	Stage6	Stage6	(E)
12	Stage6	Stage7	(!E)
13	Stage7	Stage1	(E)
14	Stage7	Stage7	(!E)

б)

	Name	state~16	state~15	state~14
1	Stage1	0	0	0
2	Stage2	0	1	1
3	Stage3	0	1	0
4	Stage4	1	0	1
5	Stage5	1	0	0
6	Stage6	1	1	1
7	Stage7	1	1	0

в)

Рис.3.26. Синтез конечного автомата по VHDL-коду в случае использования атрибута syn_encoding (двоичное кодирование): а) граф-автомат, восстановленный из VHDL-кода; б) условия переходов по состояниям; в) таблица переходов

Сравнительные результаты методов ONE и двоичного кодирования для ПЛИС серий APEX и STRATIX с использованием явных и не явных стилей кодирования представлены в таблице.

Сравнительные результаты методов ONE и двоичного кодирования

Стиль кодирования		Число триггеров	Максимальная тактовая частота, f_{MAX}, МГц	
			APEX20KE	STRATIX III
Ручной способ кодирования по методу ONE с RS-триггером на выходе		8	290	-
Двоичное кодирование (двухпроцессный шаблон, явный, атрибут enum_encoding)		3	262	400
HOT (двухпроцессный шаблон, явный, атрибут enum_encoding)		6	290	400
Настройки модуля синтеза - HOT	Однопроцессный шаблон (не явный)	10	290	400
	Двухпроцессный шаблон (не явный)	7	290	400
	Трехпроцессный шаблон (не явный)	7	290	400

При разработке конечных автоматов в базисе ПЛИС FPGA на языке VHDL наиболее эффективным решением является использование не явного стиля кодирования или явного с применением атрибута syn_encoding, поручая

компилятору-синтезатору САПР Quartus II минимизацию логических ресурсов.

Метод ONE применительно к ПЛИС FPGA, дает возможность строить конечные автоматы, которые в общем случае требуют меньших ресурсов и отличаются более высокими скоростными показателями, чем аналогичные конечные автоматы с двоичным кодированием состояний. Однако, если число состояний не более 8, то двоичное кодирование может быть более эффективным.

Повышенное быстродействие по методу ONE обеспечивается меньшим числом уровней логики между рабочими фронтами синхросигналов, чем в случае двоичного кодирования. Логические схемы при этом упрощаются, поскольку метод ONE практически не требует логики декодирования состояний. Получающийся в результате построения конечного автомата набор триггеров похож на структуру типа сдвигового регистра.

Быстродействие конечного автомата типа ONE остается постоянным с увеличением числа состояний. И напротив, быстродействие конечного автомата с высокой степенью кодирования состояний снижается с увеличением количества состояний, поскольку в этом случае для декодирования требуется большее число уровней логики с большим числом линий.

При проектировании цифровых автоматов на языках описания аппаратурных средств возможно появление недопустимых состояний. В этом случае необходимо доопределять состояния автомата, которые в свою очередь могут снижать его быстродействие.

3.3. Использование цифровых автоматов в технологии периферийного сканирования БИС

Термином JTAG обозначают совокупность средств и операций, позволяющих проводить тестирование БИС без физического доступа к каждому ее выводу. Аббревиатура JTAG возникла по наименованию разработчика – объединенной группы по тестам Joint Test Action Group. Термином “периферийное сканирование” или по-английски Boundary Scan Test (BST) называют тестирование по JTAG стандарту (IEEE Std 1149.1). Такое тестирование возможно только для ИС, внутри которых имеется набор специальных элементов – ячеек периферийного сканирования (Boundary Scan Cells) и схем управления их работой. Механизм граничного сканирования (Boundary Scan) является промышленным стандартом, который был разработан группой специалистов по проблемам тестирования электронных компонент. Все семейства ПЛИС фирмы ALTERA (FLEX 10K, MAX9000,APEX20K и др.) в той или иной степени поддерживают этот стандарт. В разных семействах по-разному реализованы схемы поддержки стандарта, они имеют разные временные параметры и состав сигналов интерфейса. Преимуществом архитектуры JTAG является отсутствие необходимости явного задания адресов устройств, поскольку все JTAG-устройства объединяются в последовательную цепочку и неявно адресуются своим положением в ней. На рис.3.27 показаны блок схема метода по стандарту IEEE Std 1149.1 (а) и структурная схема JTAG – контроллера используемого в ПЛИС фирмы Altera (б).

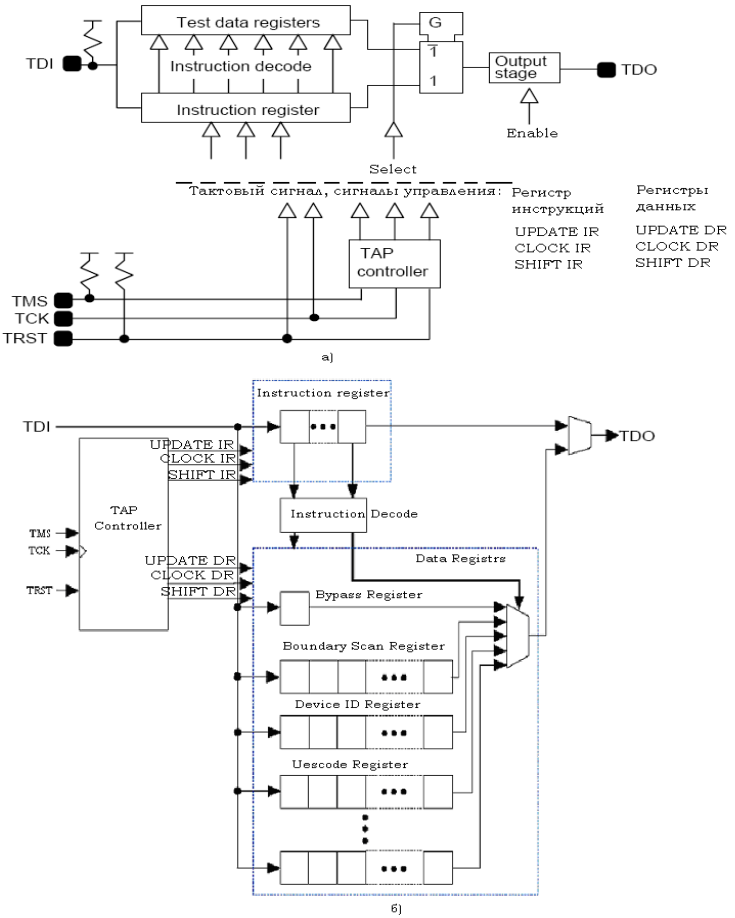


Рис.3.27. Блок схема метода по стандарту IEEE Std 1149.1 (а) и структурная схема JTAG – контроллера используемого в ПЛИС фирмы Altera (б)

Как видно из рис.3.27 *а*, имеется возможность выбирать путь прохождения данных от TDI к TDO: либо через регистр команд (Instruction Register), либо через регистр данных (Data Register). Регистром данных в каждый момент времени может быть один из следующих регистров: последовательный

сдвиговый регистр граничного сканирования (Boundary-Scan register); внутренний регистр, предусмотренный разработчиком устройства (Internal register); регистр обхода (ByPass register); идентификационный регистр (Identification register).

Регистр инструкций состоит из собственно сдвигового регистра, некоторой декодирующей логики (в зависимости от количества и типов реализуемых команд) и секции хранения декодированной команды. Длина IR-регистра должна быть больше двух.

На рис.3.28, *а* приведена структура размещения аппаратных средств интерфейса JTAG proASIC БИС фирмы Actel. Также показано использование JTAG для тестирования внутренней логики ИС (рис.3.28, *б*) и для тестирования межсоединений (рис.3.28, *в*).

На рис.3.29 показан граф состояний и переходов управляющего автомата TAP-контроллера (Test Acces Port Controller), обеспечивающего выполнение функций JTAG – схематики.

На регистр инструкций (Instruction Register) по сигналу TDI приходит управляющая команда, она дешифрируется в дешифраторе (Instruction Decode) и в зависимости от полученного кода к выходу TDO подключается один из внутрисхемных регистров. Boundary-Scan Register – служит для ввода или установки сигналов на выводы испытуемой ИС, Device ID Register – в этом регистре хранится идентификационный код ИС, ByPass Register – однобитовый регистр, замыкает сигналы TDI и TDO (уменьшает длину JTAG-пути и обеспечивает возможность прозрачной ретрансляции данных через JTAG-контроллер), UESCODE – в этом регистре пользователь может хранить свою информацию об ИС (например, ее порядок в JTAG - цепочке).

Внешние относительно TAP – контроллера сигналы: TDI, TMS, TCK, TRST, TDO. Сигналы UPDATEIR, CLOCKIR, SHIFTIR, UPDATEDR, CLOCKDR, SHIFTDR формирует TAP – контроллер. TAP – контроллер представляет собой синхронный

конечный автомат с шестнадцатью состояниями, изменяющий состояние по фронту сигнала TCK (синхроимпульса) и по включению питания. Сменой состояний управляет сигнал TMS, воспринимаемый по переднему фронту сигнала TCK.

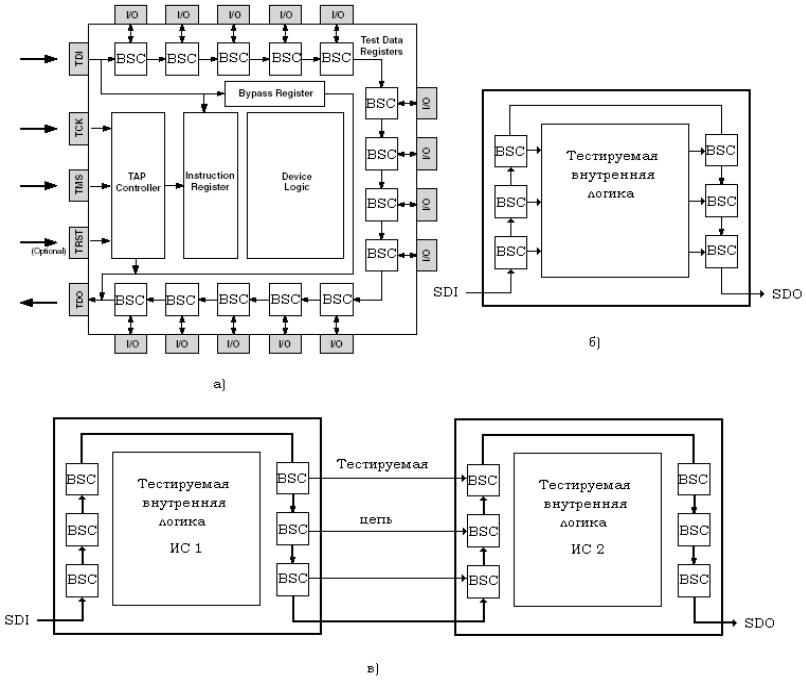


Рис.3.28. Структура аппаратных средств интерфейса JTAG proASIC БИС фирмы Actel (а); б) - использование JTAG для тестирования внутренней логики; в) - для тестирования межсоединений на печатной плате

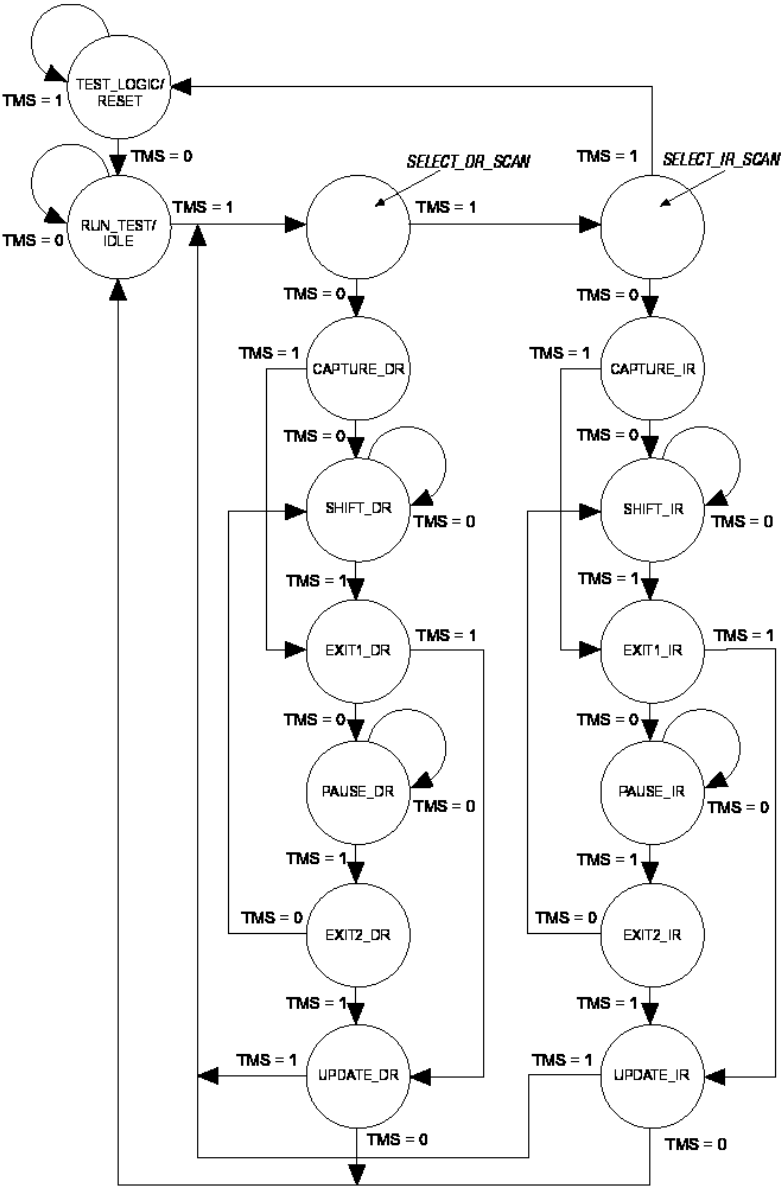


Рис.3.29. Граф состояний и переходов управляющего автомата TAP-контроллера

Когда TAP – контроллер находится в состоянии TEST_LOGIC/RESET, регистры данных не активированы, устройство находится в своем нормальном состоянии и инициализируется регистр инструкций.

При включении питания или при подаче импульса на вывод TRST (сброс), автомат переходит в состояние TEST_LOGIC/RESET. Вывод TDO находится в Z–состоянии при всех состояниях TAP – котроллера кроме состояний SHIFT_IR (регистр команд работает в режиме сдвига) и SHIFT_DR (регистр данных работает в режиме сдвига). Вывод TDO переходит в активное состояние по первому нарастающему фронту сигнала TCK после перехода TAP – контроллера в состояние SHIFT_IR или в состояние SHIFT_DR. Вывод TDO возвращается в Z–состояние по фронту спада сигнала TCK после выхода TAP – контроллера из состояния SHIFT_IR или состояния SHIFT_DR.

Правая часть графа рис.3.29 относится к записи в IR-регистр управляющей JTAG команды. Чтобы загрузить управляющую команду в IR-регистр, нужно:

1. Перейти из состояния TEST_LOGIC/RESET в состояние SHIFT_IR. Для этого на вход TMS подать последовательность 01100, синхронную с сигналами TCK. В режиме SHIFT_IR TAP – контроллер подключает сдвиговый IR-регистр к выводуам TDI и TDO. Теперь на сигнал TDI в сдвиговый IR-регистр можно подать любую JTAG команду.

2. Для записи введенной команды в IR-регистр нужно перейти в состояние UPDATE_IR (последовательность на вход 11 TMS) или пройти через состояния EXIT1_IR - > PAUSE_IR - > EXIT2_IR (последовательность 1010 на входе TMS).

3. После записи JTAG команды осуществляется переход в состояние SHIFT_DR. В состоянии SHIFT_DR возможна запись или чтение данных с регистров данных TAP – контроллера (зависит от введенной команды в состоянии SHIFT_IR).

Левая часть графа (рис.3.29) идентична правой части графа, но вместо работы с командами осуществляется работа с данными. Формирование на входе TMS последовательности 0100 синхронно с нарастающим фронтом синхроимпульса TCK приведет к загрузке регистра данных (какой именно регистр будет загружаться определяется инструкцией в IR-регистре) с входа TDI. Если у выбранного регистра данных параллельный выход содержит защелку, то защелкивание происходит в состоянии UPDATE_DR. Состояния EXIT1_DR, PAUSE_DR и EXIT2_DR используются только для навигации по граф-автомату.

Стандарт IEEE 1149.1 предписывает только 3 обязательные команды (BYPASS, EXTEST, SAMPLE/PRELOAD), необходимые для функционирования аппарата Boundary-Scan, а все остальные являются необязательными (дополнительными).

Обязательные и дополнительные команды в стандарте только описываются функционально, а их реализация на аппаратном уровне оставлена полностью на усмотрение разработчика. Команда BYPASS, позволяет эффективно организовывать длинные последовательно объединённые цепочки из тестируемых ИС. Команда EXTEST обеспечивает возможность снимать или устанавливать логические значения на контактах ИС. Команда SAMPLE/PRELOAD позволяет тестировать ядро ИС в статическом режиме, выставляя или снимая значения логических уровней на границе его выходных буферов.

В стандарте также приводятся электрические схемы ячейки сдвигового регистра IR-регистра (рис.3.30, *а*) и ячейка BSR-регистра (рис.3.30, *б*). Ячейка BSR-регистра состоит из двух D-триггеров, работающих по фронту нарастания синхроимпульса, двух мультиплексоров. Ранее такие ячейки использовались для реализации элементов памяти в методе сканирования пути (Scan Path). Регистр инструкций может быть трех битным как у ПЛИС фирмы Actel (рис.3.30, *в*), так и 10-ти

битным, как у ПЛИС фирмы Altera. Ячейки BSR-регистра могут работать в разных режимах. В рабочем режиме они пропускают сигналы через себя слева направо и не изменяют функционирование схемы. Для выходов обычного логического типа нужна одна ячейка, для выходов с третьим состоянием – две, для двунаправленных три (рис.3.30, *г*).

Разработчик может самостоятельно описать функционирование TAP-контроллера с использованием высокоуровневых языков описания аппаратуры. На врезке показано описание TAP-контроллера на языке AHDL САПР MAX+Plus II, а на рис.3.31 временные диаграммы работы TAP-контроллера.

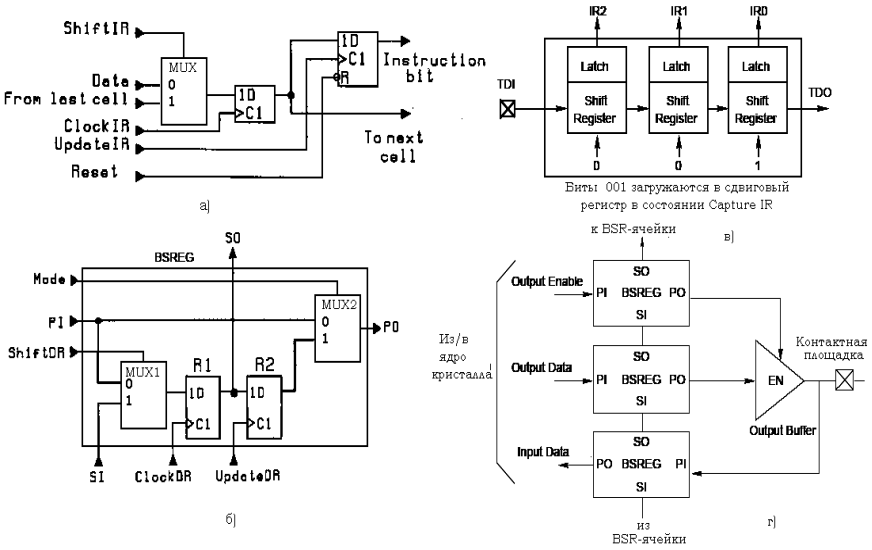


Рис.3.30. Ячейка сдвигового IR-регистра (а) и ячейка BSR-регистра (б), трехбитный регистр инструкций ПЛИС Actel (в), использование ячеек BSR-регистра для организации двунаправленного выхода (г)

Язык описания аппаратуры AHDL разработан фирмой Altera и предназначен для описания комбинационных и

последовательностных логических устройств, групповых операций, цифровых автоматов и таблиц истинности с учётом архитектурных особенностей ПЛИС фирмы Altera. Он полностью интегрируется с САПР ПЛИС MAX+PlusII и Quartus II. Файлы описания аппаратуры, написанные на языке AHDL, имеют расширение “*.TDF” (*Text design file*). Для создания TDF-файла можно использовать как текстовый редактор системы MAX+PlusII, так и любой другой. Проект, выполненный в виде TDF-файла, компилируется, отлаживается и используется для формирования файла программирования или загрузки ПЛИС фирмы Altera.

Операторы и элементы языка AHDL являются достаточно мощным и универсальным средством описания алгоритмов функционирования цифровых устройств, удобным в использовании. Язык описания аппаратуры AHDL даёт возможность создавать иерархические проекты в рамках одного этого языка или использовать TDF-файлы, разработанные на языке AHDL, наряду с другими типами файлов. Для создания проектов на AHDL можно, конечно, пользоваться любым текстовым редактором, но текстовый редактор системы MAX+PlusII предоставляет ряд дополнительных возможностей для ввода, компиляции и отладки проектов.

Проекты (пример 1), созданные на языке AHDL, легко внедряются в иерархическую структуру. Система MAX+PlusII позволяет автоматически создать символ компонента, алгоритм функционирования которого описывается TDF-файлом, и затем вставить его в файл схемного описания (GDF-файл). Подобным же образом можно вводить в любой TDF-файл собственные функции разработчика и около 300 макрофункций, разработанных фирмой Altera.

```
SUBDESIGN avtomat
(TCK,TRST,TMS: INPUT;
UPDATEIR,CLOCKIR,SHIFTIR, UPDATEDR,CLOCKDR,SHIFTDR:
OUTPUT;)
```

```
VARIABLE
        TAP_controller: MACHINE
        OF BITS (q3,q2,q1,q0)
        WITH STATES (TEST_LOGIC_RESET, RUN_TEST_IDLE,
SELECT_DR, CAPTURE_DR, SHIFT_DR, EXIT1_DR, PAUSE_DR,
EXIT2_DR, UPDATE_DR, SELECT_IR, CAPTURE_IR, SHIFT_IR,
EXIT1_IR, PAUSE_IR,EXIT2_IR,UPDATE_IR);
BEGIN
        TAP_controller.clk = TCK;
        TAP_controller.reset = TRST;
TABLE
% Present                 Next            %
% State    Inputs         State    Outputs  %
TAP_controller,TMS => TAP_controller, UPDATEIR, CLOCKIR,
SHIFTIR, UPDATEDR, CLOCKDR, SHIFTDR;
TEST_LOGIC_RESET, 1 => TEST_LOGIC_RESET, 0, 0, 0, 0, 0, 0;
TEST_LOGIC_RESET, 0 => RUN_TEST_IDLE, 0, 0, 0, 0, 0, 0;
RUN_TEST_IDLE, 0 => RUN_TEST_IDLE, 0, 0, 0, 0, 0, 0;
RUN_TEST_IDLE, 1 => SELECT_DR, 0, 0, 0, 0, 0, 0;
SELECT_DR, 1 => SELECT_IR, 0, 0, 0, 0, 0, 0;
SELECT_DR, 0 => CAPTURE_DR, 0, 0, 0, 0, 0, 0;
CAPTURE_DR, 0 => SHIFT_DR, 0, 0, 0, 0, 1, 1;
CAPTURE_DR, 1 => EXIT1_DR, 0, 0, 0, 0, 1, 1;
SHIFT_DR, 0 => SHIFT_DR, 0, 0, 0, 0, 0, 1;
SHIFT_DR, 1 => EXIT1_DR, 0, 0, 0, 0, 0, 1;
EXIT1_DR, 1 => UPDATE_DR, 0, 0, 0, 0, 0, 0;
EXIT1_DR, 0 => PAUSE_DR, 0, 0, 0, 0, 0, 0;
PAUSE_DR, 0 => PAUSE_DR, 0, 0, 0, 0, 0, 0;
PAUSE_DR, 1 => EXIT2_DR, 0, 0, 0, 0, 0, 0;
EXIT2_DR, 0 => SHIFT_DR, 0, 0, 0, 0, 0, 0;
EXIT2_DR, 1 => UPDATE_DR, 0, 0, 0, 0, 0, 0;
UPDATE_DR, 1 => SELECT_DR, 0, 0, 0, 1, 0, 0;
UPDATE_DR, 0 => RUN_TEST_IDLE, 0, 0, 0, 1, 0, 0;
SELECT_IR, 1 => TEST_LOGIC_RESET, 0, 0, 0, 0, 0, 0;
SELECT_IR, 0 => CAPTURE_IR, 0, 0, 0, 0, 0, 0;
CAPTURE_IR, 0 => SHIFT_IR, 0, 1, 1, 0, 0, 0;
CAPTURE_IR, 1 => EXIT1_IR, 0, 1, 1, 0, 0, 0;
SHIFT_IR, 0 => SHIFT_IR, 0, 0, 1, 0, 0, 0;
```

SHIFT_IR, 1 => EXIT1_IR, 0, 0, 1, 0, 0, 0;
EXIT1_IR, 1 => UPDATE_IR, 0, 0, 0, 0, 0, 0;
EXIT1_IR, 0 => PAUSE_IR, 0, 0, 0, 0, 0, 0;
PAUSE_IR, 0 => PAUSE_IR, 0, 0, 0, 0, 0, 0;
PAUSE_IR, 1 => EXIT2_IR, 0, 0, 0, 0, 0, 0;
EXIT2_IR, 0 => SHIFT_IR, 0, 0, 0, 0, 0, 0;
EXIT2_IR, 1 => UPDATE_IR, 0, 0, 0, 0, 0, 0;
UPDATE_IR, 1 => SELECT_DR, 1, 0, 0, 0, 0, 0;
UPDATE_IR, 0 => RUN_TEST_IDLE, 1, 0, 0, 0, 0, 0;
END TABLE;
END;

Пример 1. Описание TAP-контроллера на языке AHDL

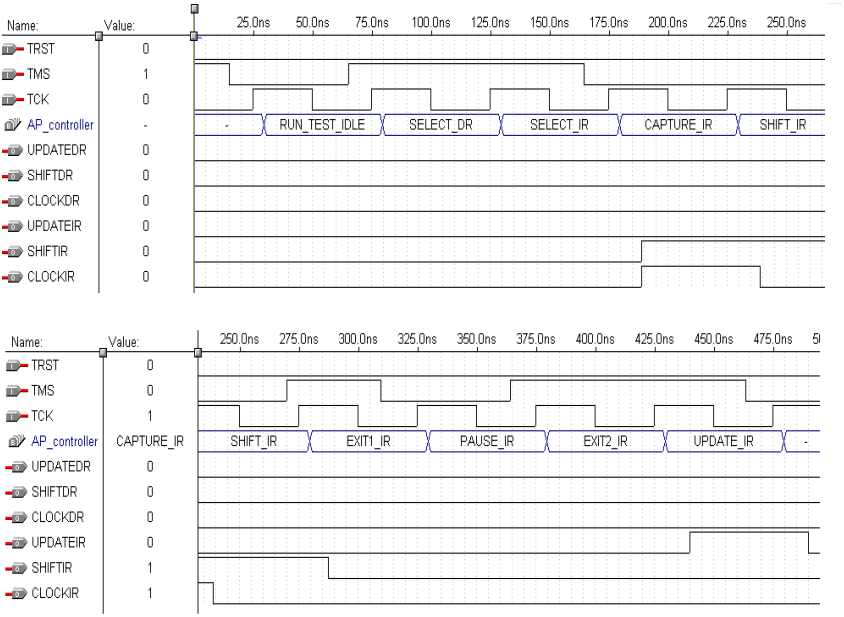


Рис.3.31. Временные диаграммы работы TAP-контроллера

Пример 2 демонстрирует описание TAP-контроллера на языке VHDL. На рис.3.32 приведено тестирование TAP-контроллера. Тестируются всевозможные условия переходов граф-автомата.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY JTAG IS
PORT(TCK,TRST,TMS: IN STD_LOGIC;
UPDATEIR, CAPTUREIR, SHIFTIR, UPDATEDR, CAPTUREDR,
SHIFTDR, ENA_TDO, SEL_IR : OUT  STD_LOGIC;
STATE_JTAG: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END JTAG;

ARCHITECTURE a OF JTAG IS
TYPE state_values IS (TEST_LOGIC_RESET, RUN_TEST_IDLE,
SELECT_DR, CAPTURE_DR, SHIFT_DR,
EXIT1_DR, PAUSE_DR, EXIT2_DR, UPDATE_DR, SELECT_IR,
CAPTURE_IR, SHIFT_IR, EXIT1_IR, PAUSE_IR, EXIT2_IR,
UPDATE_IR);
signal state,next_state: state_values;
BEGIN
-- регистерный блок
statereg: process(TCK,TRST)
begin
        if (TRST = '1') then state<=TEST_LOGIC_RESET;
                elsif (TCK'event and TCK='1') then
                                state<=next_state;
        end if;
end process statereg;
-- комбинаторный блок (логика переходов)
process(state, TMS)
begin
case state is
when TEST_LOGIC_RESET=>
STATE_JTAG <= "0000";
IF (TMS='0')THEN next_state<=RUN_TEST_IDLE;
ELSE next_state<=TEST_LOGIC_RESET;
END IF;
when RUN_TEST_IDLE=>
STATE_JTAG <= "0001";
```

```
IF (TMS='1')THEN next_state<=SELECT_DR;
ELSE next_state<=RUN_TEST_IDLE;
END IF;
when SELECT_DR=>
STATE_JTAG <= "0010";
IF (TMS='1')THEN next_state<=SELECT_IR;
ELSE next_state<=CAPTURE_DR;
END IF;
when CAPTURE_DR=>
STATE_JTAG <= "0011";
IF (TMS='1')THEN next_state<=EXIT1_DR;
ELSE next_state<=SHIFT_DR;
END IF;
when SHIFT_DR=>
STATE_JTAG <= "0100";
IF (TMS='1')THEN next_state<=EXIT1_DR;
ELSE next_state<=SHIFT_DR;
END IF;
when EXIT1_DR=>
STATE_JTAG <= "0101";
IF (TMS='1')THEN next_state<=UPDATE_DR;
ELSE next_state<=PAUSE_DR;
END IF;
when PAUSE_DR=>
STATE_JTAG <= "0110";
IF (TMS='1')THEN next_state<=EXIT2_DR;
ELSE next_state<=PAUSE_DR;
END IF;
when EXIT2_DR=>
STATE_JTAG <= "0111";
IF (TMS='1')THEN next_state<=UPDATE_DR;
ELSE next_state<=SHIFT_DR;
END IF;
when UPDATE_DR=>
STATE_JTAG <= "1000";
IF (TMS='1')THEN next_state<=SELECT_DR;
ELSE next_state<=RUN_TEST_IDLE;
END IF;
```

```
when SELECT_IR=>
STATE_JTAG <= "1001";
IF (TMS='1')THEN next_state<=TEST_LOGIC_RESET;
ELSE next_state<=CAPTURE_IR;
END IF;
when CAPTURE_IR=>
STATE_JTAG <= "1010";
IF (TMS='1')THEN next_state<=EXIT1_IR;
ELSE next_state<=SHIFT_IR;END IF;
when SHIFT_IR=>
STATE_JTAG <= "1011";
IF (TMS='1')THEN next_state<=EXIT1_IR;
ELSE next_state<=SHIFT_IR;END IF;
when EXIT1_IR=>
STATE_JTAG <= "1100";
IF (TMS='1')THEN next_state<=UPDATE_IR;
ELSE next_state<=PAUSE_IR;END IF;
when PAUSE_IR=>
STATE_JTAG <= "1101";
IF (TMS='1')THEN next_state<=EXIT2_IR;
ELSE next_state<=PAUSE_IR;END IF;
when EXIT2_IR=>
STATE_JTAG <= "1110";
IF (TMS='1')THEN next_state<=UPDATE_IR;
ELSE next_state<=SHIFT_IR;END IF;
when UPDATE_IR=>
STATE_JTAG <= "1111";
IF (TMS='1')THEN next_state<=SELECT_DR;
ELSE next_state<=RUN_TEST_IDLE;END IF;
end case;
end process;
-- логика формирования выхода
process (state)
begin
case state is
when CAPTURE_DR => UPDATEIR <= '0'; CAPTUREIR <= '0';
SHIFTIR <= '0'; UPDATEDR <= '0'; CAPTUREDR <= '1';
SHIFTDR <= '1'; ENA_TDO <= '0'; SEL_IR <= '0';
```

```
when SHIFT_DR=> UPDATEIR <= '0'; CAPTUREIR <= '0';
SHIFTIR <= '0'; UPDATEDR <= '0'; CAPTUREDR <= '0';
SHIFTDR <= '1'; ENA_TDO <= '1'; SEL_IR <= '0';
when UPDATE_DR=> UPDATEIR <= '0'; CAPTUREIR <= '0';
SHIFTIR <= '0'; UPDATEDR <= '1'; CAPTUREDR <= '0';
SHIFTDR <= '0'; ENA_TDO <= '0'; SEL_IR <= '0';
when CAPTURE_IR=> UPDATEIR  <= '0'; CAPTUREIR <= '1';
SHIFTIR <= '1'; UPDATEDR <= '0'; CAPTUREDR <= '0';
SHIFTDR <= '0'; ENA_TDO <= '0'; SEL_IR <= '0';
when SHIFT_IR=> UPDATEIR <= '0'; CAPTUREIR <= '0';
SHIFTIR <= '1'; UPDATEDR <= '0'; CAPTUREDR <= '0';
SHIFTDR <= '0'; ENA_TDO <= '1'; SEL_IR <= '1';
when UPDATE_IR=> UPDATEIR  <= '1'; CAPTUREIR <= '0';
SHIFTIR <= '0'; UPDATEDR  <= '0'; CAPTUREDR <= '0';
SHIFTDR <= '0'; ENA_TDO <= '0'; SEL_IR <= '0';
when others =>  UPDATEIR  <= '0'; CAPTUREIR <= '0';
SHIFTIR <= '0'; UPDATEDR <= '0'; CAPTUREDR <= '0';
SHIFTDR <= '0'; ENA_TDO <= '0'; SEL_IR <= '0';
end case;
end process;
END a;
```

Пример 2. Описание TAP-контроллера на языке VHDL

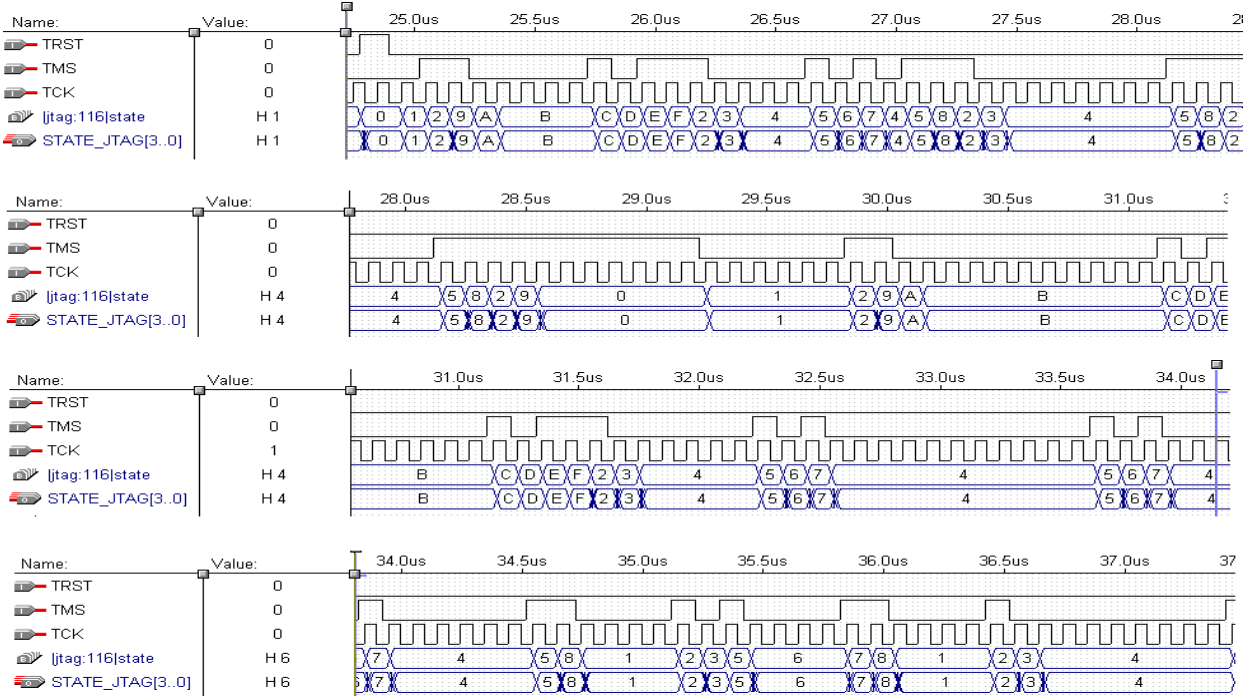


Рис.3.32. Примеры выполнение BST – команд

3.4. Проектирование цифровых автоматов с использованием системы MATLAB/SIMULINK и САПР ПЛИС Quartus II

Целью данного раздела является демонстрация возможностей системы Matlab/Simulink (пакет расширения Stateflow) по проектированию цифровых автоматов, представленных графом переходов, с последующей их реализацией в базисе ПЛИС. Simulink – графическая среда имитационного моделирования аналоговых и дискретных систем. Предоставляет пользователю графический интерфейс для конструирования моделей из стандартных блоков, без единой строчки кода. Simulink работает с линейными, нелинейными, непрерывными, дискретными и многомерными системами. Система Matlab/Simulink содержит встроенный генератор кода языка описания аппаратных средств HDL (Simulink HDL Coder) и ориентирована на поддержку симулятора VHDL ModelSim (Mentor Graphics HDL simulator). Simulink HDL Coder – программный продукт для генерации VHDL-кода без привязки к конкретной архитектуре ПЛИС и платформе по Simulink-моделям и граф-автоматам (Stateflow-диаграммы). Система Matlab/Simulink эффективна также для разработки цифровых фильтров для реализации в базисе ПЛИС и ЦОС процессоров, т.к. содержит Filter Design HDL Coder.

Многие САПР БИС, например, Mentor Graphics (HDL Designer) и САПР ПЛИС, такие как Foundation фирмы Xilinx (система синтеза FPGA Express Synthesis, разработанная компанией Synopsys), StateCAD фирмы Visual Software Solutions, Quartus II (начиная с версии 7.2) фирмы Altera, также содержат встроенные средства проектирования цифровых автоматов, позволяют задавать цифровой автомат графом переходов и получать автоматически код языка VHDL или Verilog.

ModelSim - наиболее распространенный в мире VHDL и VHDL/Verilog-симулятор. Популярность ModelSim отражает

стремление фирмы Mentor Graphics предоставить пользователям самую передовую технологию моделирования, высокую производительность и полную техническую поддержку. Семейство ModelSim имеет уникальную архитектуру, основанную на принципе "оптимизированной прямой компиляции" и "едином ядре моделирования".

Архитектура, базирующаяся на принципе оптимизированной прямой компиляции, является технологией нового поколения в области HDL моделирования. Она составляет основу всех продуктов семейства ModelSim. В соответствии с этим принципом исходный VHDL или Verilog код компилируется в машинно-независимый объектный код, исполняемый на любой поддерживаемой платформе (САПР БИС или ПЛИС). Непосредственно скомпилированные, HDL-объекты, автоматически оптимизируются для любой поддерживаемой платформы в момент запуска программы ModelSim.

Рассмотрим проектирование автомата Мили (Mealy) с использованием системы Matlab/Simulink и САПР ПЛИС Quartus. На рис.3.33, *а* показан испытательный стенд (модель) автомата Мили в системе Matlab/Simulink. Пример автомата Мили позаимствуем из справочной системы Simulink. Торговый автомат (рис.3.33, *б*), предназначен для выдачи бутылки сладкой шипучей жидкости (сигнал soda), когда опущено 15 центов или более. Торговый автомат не совершенен и сдачи не дает, т.е. оставляет “себе’ монету в 5 центов, которая будет добавлена к общему вкладу. Пример более совершенного торгового автомата можно найти в книге известных американских специалистов Хоровиц, Хилл “Искусство схемотеники”.

Существует некоторый вид монетного интерфейса, который ‘заглатывает’, распознает монету и посылает на входы автомата сигнал Coin (монета). Монетный интерфейс реализуется с использованием сигнала Coin (рис.3.33, *в*). Аналоговый входной сигнал Coin на диаграмме переходов

кодируется следующим образом: [Coin==1] - брошена монета в 5 центов (nickel); [Coin==2] - брошена монета в 10 центов (dime)), где 1, 2 – переменные вещественного типа. Поэтому сигнал Coin должен принимать значения 1 или 2 (рис.3.33, *в*). Выходной сигнал Soda кодируется следующим образом:

{Soda=0} – нет бутылки;

{Soda=1} – бутылка.

Квадратные скобки [] обозначают условие, фигурные {} – действие по условию. Запись [Coin==1]{Soda=0} говорит о том, что выход автомата Мили является функцией как текущего состояния, так и начального внешнего воздействия, т.е. от сигнала Coin.

Автомат может принимать три состояния (рис.3.33, *б*): got_0, got_nickel, got_dime. Переходы по состояниям помечены цифрами. Когда состояние got_0 активно, возможны следующие переходы: брошена монета в 5 центов (Coin==1) выход торгового автомата принимает значение Soda=0, а следующим активным состоянием будет got_nickel (переход 1). Если брошена монета в 10 центов (Coin==2), то выход торгового автомата принимает значение Soda=0, а следующим активным состоянием будет got_dime (переход 2). Если не брошена ни одна из монет, то автомат остается в состоянии got_0. Остальные переходы по состояним видны на рис.3.33, *б*.

После того как, будет создана модель цифрового автомата, необходимо выбрать численный метод решения системы дифференциальных уравнений. С помощью проводника модели (Model Explorer) выбираем дискретный метод решения (discrete) в настройках Solver и настраиваем генератор кода языка VHDL в меню HDL coder (рис.3.34). Результат моделирования показан на рис.3.33, *в*.

Для получения кода на языке VHDL необходимо в проводнике модели нажать на кнопку Generate. При компиляции проекта цифрового автомата, генератор кода языка VHDL, согласно ранее проведенным настройкам, автоматически добавляет сигнал тактирования clk, сигнал

разрешения тактирования clk_enable, асинхронный сигнал сброса reset. Пример 1 демонстрирует код автомата Мили на языке VHDL, полученный с использованием Simulink HDL Coder системы Matlab/Simulink. Тип сигналов Coin и Soda – вещественный (real).

Анализируя стиль кодирования цифрового автомата, видим, что метод кодирования не определен в коде языка VHDL. Используется двухпроцессорный шаблон, оператор выбора CASE и перечисляемый тип данных (Enumerated type). Перечисляемый тип – это такой тип данных, при котором количество всех возможных состояний конечно. Такой тип наиболее часто используется для обозначений состояний конечных автоматов. В этом случае имеется возможность предоставить САПР ПЛИС использовать модуль логического синтеза и в зависимости от архитектуры ПЛИС самостоятельно выбирать метод кодирования. Сигнал разрешения тактирования clk_enable, генерируется как синхронный (стоит после атрибута срабатывания по переднему фронту clk'EVENT AND clk= '1').

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY avt_mealy IS
   PORT (
      clk, clk_enable: IN std_logic;
      reset : IN std_logic;
      coin : IN real;
      soda : OUT real);
END avt_mealy;
ARCHITECTURE fsm_SFHDL OF avt_mealy IS
   TYPE T_state_type_is_avt_mealy is (IN_NO_ACTIVE_CHILD,
IN_got_0, IN_got_dime, IN_got_nickel);
   SIGNAL is_avt_mealy : T_state_type_is_avt_mealy;
   SIGNAL is_avt_mealy_next : T_state_type_is_avt_mealy;
BEGIN
   PROCESS (reset, clk)
      -- local variables
```

```
BEGIN
   IF reset = '1' THEN
      is_avt_mealy <= IN_got_0;
   ELSIF clk'EVENT AND clk= '1' THEN
      IF clk_enable= '1' THEN
         is_avt_mealy <= is_avt_mealy_next;
      END IF;
   END IF;
END PROCESS;
avt_mealy : PROCESS (is_avt_mealy, coin)
   -- local variables
   VARIABLE is_avt_mealy_temp : T_state_type_is_avt_mealy;
BEGIN
   is_avt_mealy_temp := is_avt_mealy;
   soda <= 0.0;
   CASE is_avt_mealy_temp IS
      WHEN IN_got_0 =>
         IF coin = 1.0 THEN
            soda <= 0.0;
            is_avt_mealy_temp := IN_got_nickel;
         ELSE
            IF coin = 2.0 THEN
               soda <= 0.0;
               is_avt_mealy_temp := IN_got_dime;
            END IF;
         END IF;
      WHEN IN_got_dime =>
         IF coin = 2.0 THEN
            soda <= 1.0;
            is_avt_mealy_temp := IN_got_nickel;
         ELSE
```

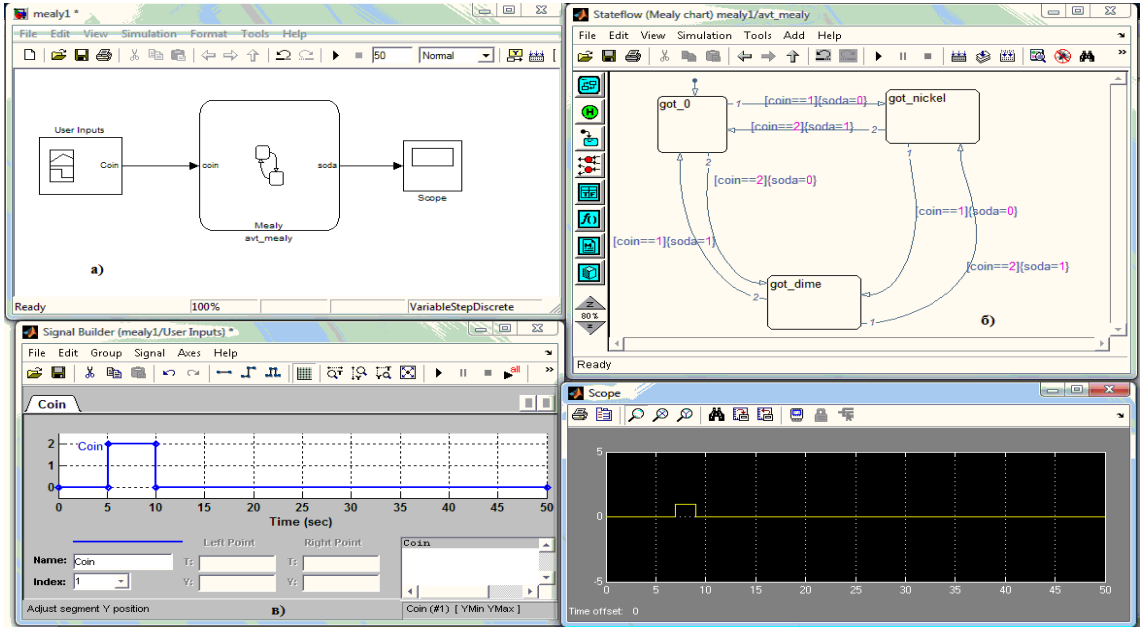


Рис.3.33. Автомат Мили построенный с использованием системы Matlab/Simulink: а – испытательный стенд; б – граф-автомат; г – входной сигнал Coin; д – выходной сигнал Soda

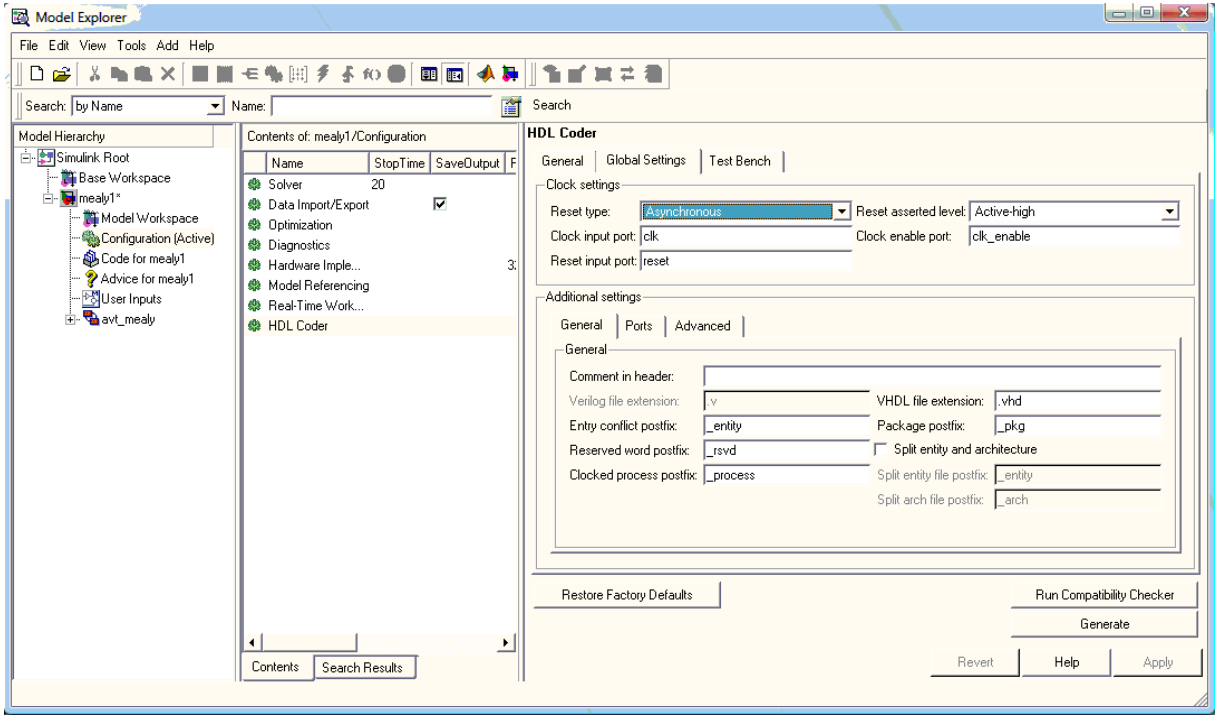


Рис.3.34. Окно проводника модели. Настройка генератора кода языка VHDL

```
            IF coin = 1.0 THEN
               soda <= 1.0;
               is_avt_mealy_temp := IN_got_0;
            END IF;
         END IF;
      WHEN IN_got_nickel =>
         IF coin = 1.0 THEN
            soda <= 0.0;
            is_avt_mealy_temp := IN_got_dime;
         ELSE
            IF coin = 2.0 THEN
               soda <= 1.0;
               is_avt_mealy_temp := IN_got_0;
            END IF;
         END IF;
      WHEN OTHERS =>
         is_avt_mealy_temp := IN_got_0;
   END CASE;
   is_avt_mealy_next <= is_avt_mealy_temp;
 END PROCESS;
END fsm_SFHDL;
```

Пример 1. Код автомата Мили на языке VHDL полученный с использованием Simulink HDL Coder системы Matlab/Simulink

Для реализации проекта в базисе ПЛИС фирмы Altera, аналоговый сигнал Coin закодируем 2-х битным цифровым сигналом Coin[1..0], действительным для одного такта сигнала Clk, показывающего монету, которую опустили:

Coin[00] – B00 - нет монеты;

Coin[01] – B 01 - брошена монета в 5 центов (nickel);

Coin[10] – B 10 - брошена монета в 10 центов (dime)).

Сигнал Soda, закодируем двухбитным сигналом Soda[1..0]:

Soda[00] – нет бутылки;

Soda[01] – бутылка.

Пример 2 демонстрирует “подправленный” код автомата Мили на языке VHDL в САПР ПЛИС Quartus II. Simulink HDL

Coder кодировал переходы по состояним в следующей последовательности (пример 1): вначале рассматривались все возможные переходы из состояния IN_got_0, затем из состояния IN_got_dime и IN_got_nickel. Поэтому, состояния проектируемого автомата Мили в САПР ПЛИС Quartus кодируются в этой же последовательности: В 01, В 10, В 11, т.е., 1, 2, 3. В. На рис.3.35 показана тестовая схема автомата Мили, а на рис.3.36 временная диаграмма. В процессе работы автомат “пробегает” по состояниям (сигнал (или узел) is_avt_mealy представляет из себя регистр состояния построенный на двух разрядной шине; на временной диаграмме узел отображается значком контактной ножки с буквой R) с номерами В 01, В 11, В 10 и В 01, т.е. 1, 3, 2, 1. Таким образом, тестируется переход по состояниям got_0, got_nickel, got_dime, got_0. Анализируя временную диаграмму, можно сделать вывод, что торговый автомат работает корректно. В случае, если на вход будет подан сигнал Coin[11] – В 11, то автомат по состояниям переходить не будет.

На рис.3.37, *а* показано проектирование автомата Мили в ModelSim SE Plus. Также показаны состояния автомата реализованные на сигналах is_avt_mealy (регистр текущего состояния) и is_avt_mealy_next (регистр следующего состояния). Из рис.3.27, *б* видно, как кодируются состояния автомата. Состояние IN_NO_ACTIVE_CHILD является состоянием по умолчанию, введенным генератором кода Simulink HDL Coder. Компилятор САПР Quartus его сокращает на этапе компиляции проекта (минимизирует), а симулятор ModelSim SE Plus, начинает работу именно с этого состояния. Сравнивая рис.3.36 и рис.3.37, видим, что автоматы спроектированные в различных платформах, работают корректно, несмотря на разные способы представления результатов моделирования. Однако в САПР ПЛИС Quartus сигнал soda В 01 появляется асинхронно, а в ModelSim с приходом тактового импульса, т.е. синхронно.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE ieee.std_logic_unsigned.all;
ENTITY avt_mealy IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      coin : IN std_logic_vector(1 downto 0);
      soda : OUT std_logic_vector (1 downto 0));
END avt_mealy;
ARCHITECTURE fsm_SFHDL OF avt_mealy IS
   TYPE T_state_type_is_avt_mealy is (IN_NO_ACTIVE_CHILD,
IN_got_0, IN_got_dime, IN_got_nickel);
   SIGNAL is_avt_mealy : T_state_type_is_avt_mealy;
   SIGNAL is_avt_mealy_next : T_state_type_is_avt_mealy;
BEGIN
   PROCESS (reset, clk)
      -- local variables
   BEGIN
      IF reset = '1' THEN
         is_avt_mealy <= IN_got_0;
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            is_avt_mealy <= is_avt_mealy_next;
         END IF;
      END IF;
   END PROCESS;
   avt_mealy : PROCESS (is_avt_mealy, coin)
      -- local variables
      VARIABLE is_avt_mealy_temp : T_state_type_is_avt_mealy;
   BEGIN
      is_avt_mealy_temp := is_avt_mealy;
      soda <= "00";
      CASE is_avt_mealy_temp IS
         WHEN IN_got_0 =>
            IF coin = "01" THEN
               soda <= "00";
               is_avt_mealy_temp := IN_got_nickel;
```

```
            ELSE
               IF coin = "10" THEN
                  soda <= "00";
                  is_avt_mealy_temp := IN_got_dime;
               END IF;
            END IF;
         WHEN IN_got_dime =>
            IF coin = "10" THEN
               soda <= "01";
               is_avt_mealy_temp := IN_got_nickel;
            ELSE
               IF coin = "01" THEN
                  soda <= "01";
                  is_avt_mealy_temp := IN_got_0;
               END IF;
            END IF;
         WHEN IN_got_nickel =>
            IF coin = "01" THEN
               soda <= "00";
               is_avt_mealy_temp := IN_got_dime;
            ELSE
               IF coin = "10" THEN
                  soda <= "01";
                  is_avt_mealy_temp := IN_got_0;
               END IF;
            END IF;
         WHEN OTHERS =>
            is_avt_mealy_temp := IN_got_0;
      END CASE;
      is_avt_mealy_next <= is_avt_mealy_temp;
   END PROCESS;
END fsm_SFHDL;
```

Пример 2. Код автомата Мили на языке VHDL в САПР ПЛИС Quartus

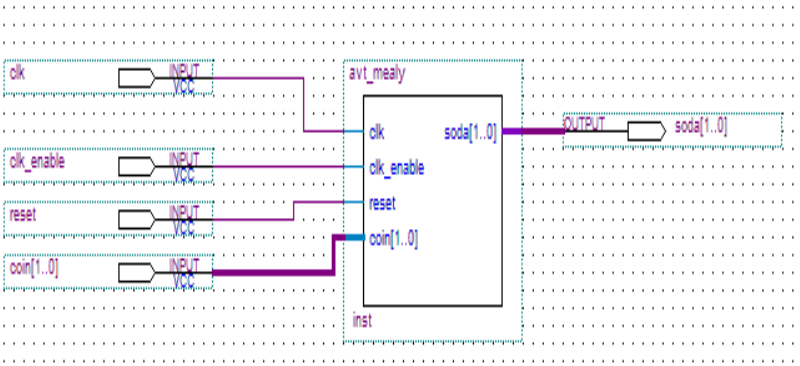
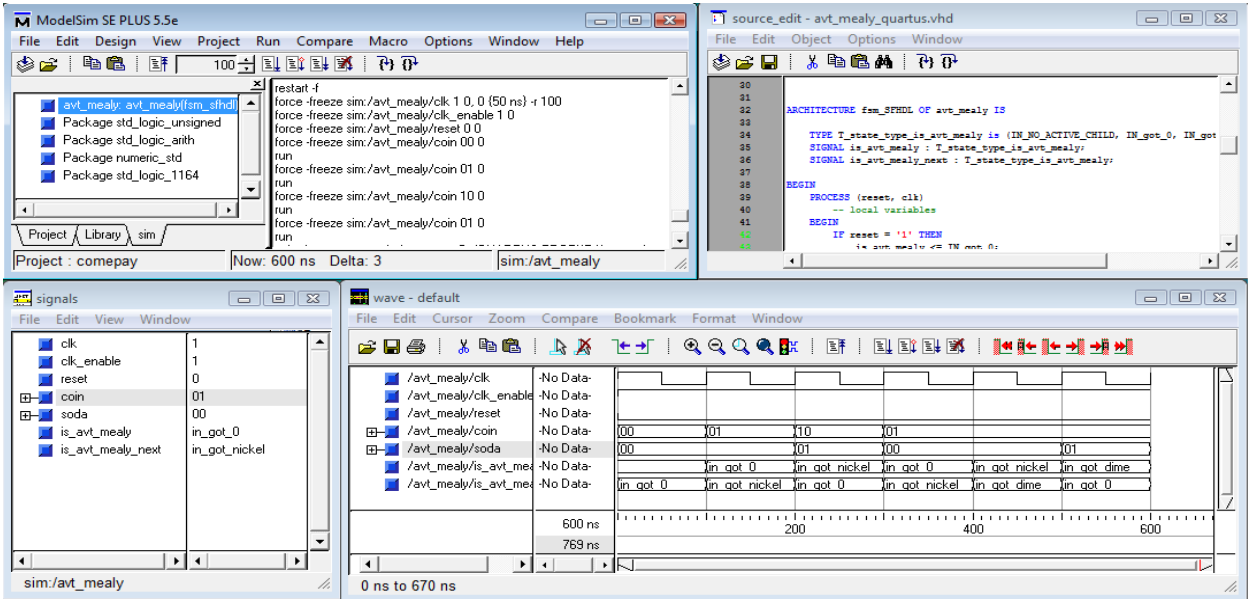


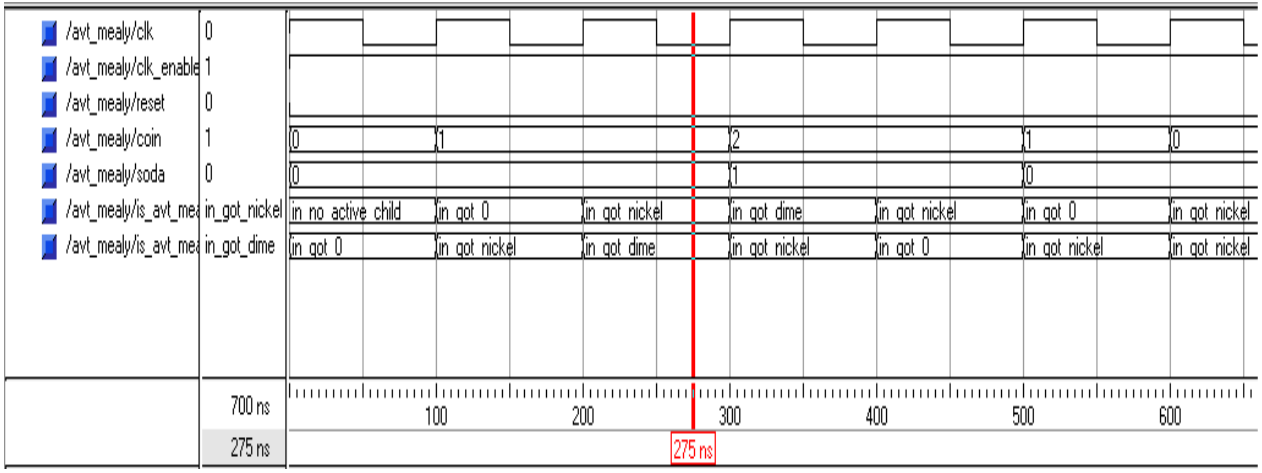
Рис.3.35. Тестовая схема автомата Мили в САПР Quartus

Рис.3.36. Временная диаграмма работы автомата Мили в САПР Quartus. Тестируется переход по состояниям got_0, got_nickel, got_dime, got_0



а)

Рис.3.37. Проектирование автомата Мили в симуляторе ModelSim SE Plus: а) – тестируется переход по состояниям got_0, got_nickel, got_dime, got_0; б) – особенности отображения состояний автомата в векторном редакторе



б)

Рис.3.37. Проектирование автомата Мили в симуляторе ModelSim SE Plus: а) – тестируется переход по состояниям got_0, got_nickel, got_dime, got_0; б) – особенности отображения состояний автомата в векторном редакторе (продолжение)

Рассмотрим проектирование простейшего автомата Мура. Выход автомата Мура является функцией только текущего состояния. Граф-автомат Мура показан на рис.3.38. Пример 3 демонстрирует код языка VHDL, полученный с использованием генератора кода Simulink VHDL Coder. Анализируем код, видим, что используется также двухпроцессорный шаблон и перечисляемый тип данных, но два оператора CASE. Первый оператор CASE используется для формирования логики выхода, второй для описания логики переходов.

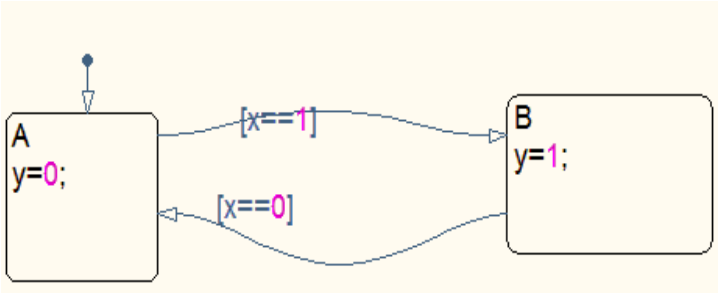


Рис.3.38. Автомат Мура в системе Matlab/Simulink

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY avt_moore_test IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      x : IN real;
      y : OUT real);
END avt_moore_test;
ARCHITECTURE fsm_SFHDL OF avt_moore_test IS
   TYPE T_state_type_is_avt_moore_test is
(IN_NO_ACTIVE_CHILD, IN_A, IN_B);
   SIGNAL is_avt_moore_test : T_state_type_is_avt_moore_test;
```

```
   SIGNAL is_avt_moore_test_next :
T_state_type_is_avt_moore_test;

BEGIN
   PROCESS (reset, clk)
      -- local variables
   BEGIN
      IF reset = '1' THEN
         is_avt_moore_test <= IN_A;
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            is_avt_moore_test <= is_avt_moore_test_next;
         END IF;
      END IF;
   END PROCESS;

   avt_moore_test : PROCESS (is_avt_moore_test, x)
      -- local variables
      VARIABLE is_avt_moore_test_temp :
T_state_type_is_avt_moore_test;
   BEGIN
      is_avt_moore_test_temp := is_avt_moore_test;
      y <= 0.0;
      CASE is_avt_moore_test_temp IS
         WHEN IN_A =>
            y <= 0.0;
         WHEN IN_B =>
            y <= 1.0;
         WHEN OTHERS =>
            is_avt_moore_test_temp := IN_NO_ACTIVE_CHILD;
      END CASE;
      CASE is_avt_moore_test_temp IS
         WHEN IN_A =>
            IF x = 1.0 THEN
               is_avt_moore_test_temp := IN_B;
            END IF;
         WHEN IN_B =>
```

```
            IF x = 0.0 THEN
               is_avt_moore_test_temp := IN_A;
            END IF;
         WHEN OTHERS =>
            is_avt_moore_test_temp := IN_A;
      END CASE;
      is_avt_moore_test_next <= is_avt_moore_test_temp;
   END PROCESS;

END fsm_SFHDL;
```

Пример 3. Код автомата Мура на языке VHDL, полученный с использованием Simulink HDL Coder системы Matlab/Simulink

Таким образом, автоматически сгенерированный и оптимизированный код языка VHDL по графу переходов цифрового автомата, с использованием Simulink HDL Coder системы Matlab/Simulink, позволяет значительно ускорить процесс разработки цифровых автоматов, для реализации их в базисе БИС и ПЛИС.

В данной главе рассмотрены два метода проектирования синхронных последовательностных схем, которые имеют конечное число логических состояний. С использованием ручного метода кодирования с применением высокоуровневого языка описания аппаратных схем VHDL и с использованием графических средств ввода диаграмм состояний, например, приложение StateFlow системы Matlab/Simulink.

4. ПРОЕКТИРОВАНИЕ МИКРОПРОЦЕССОРНЫХ ЯДЕР ДЛЯ РЕАЛИЗАЦИИ В БАЗИСЕ ПЛИС

4.1. Проектирование учебного процессора для реализации в базисе ПЛИС с помощью конечного автомата

Микропроцессорные ядра представляют важный класс вычислительных заготовок, так как от их качеств, по большей части зависят технические и потребительские свойства систем на кристалле. Эти заготовки различаются по степени гибкости настройки под условия потребителя как программные (“мягкие” описанные на языке HDL), жесткие (логическая схема) и аппаратные (“твердые” маски под определенную технологию). Программные заготовки можно легко подстраивать к условиям нового проекта, обладают высоким быстродействием и они независимы от технологии. Их реализация в ПЛИС (например, 8-разрядное микропроцессорное ядро PicoBlaze для реализации в базисе ПЛИС семейств Spartan и Virtex) позволяет ускорить процесс разработки микропроцессорных систем. Наиболее важными потребительскими свойствами вычислительных заготовок процессоров являются: повторяемость, быстродействие, аппаратурные затраты.

Путем несложной перенастройки “мягкой” заготовки можно получить ряд модификаций микроконтроллера с различным сочетанием объема памяти, периферийных устройств, источников прерывания и т.п. Такой процессор можно реализовать в ПЛИС различных фирм. Описание модели на VHDL позволяет не только сделать ее перенастраиваемой и независимой от технологии, но и выполнять ее моделирование и синтез на симуляторах и средствах синтеза различных фирм. На рис.4.1 показано отображение процессора в ПЛИС.

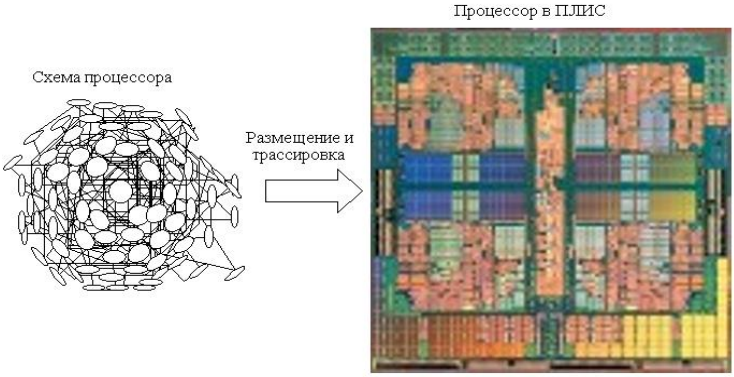


Рис.4.1. Отображение схемы процессора в базисе ПЛИС

Рассмотрим систему команд синхронного процессора, реализованного с помощью конечного автомата, с циклом работы в два такта. При разработке системы команд процессора используется слабое кодирование. В табл.4.1 представлена система команд процессора с синхронной архитектурой.

Процессор способен работать с синхронным ОЗУ. Это обеспечивается использованием оператора case, который используется в ветви оператора if при детектировании атрибута переднего фронта синхроимпульса clk и позволяет организовать цикл работы в два такта.

Реализуем процессор в ПЛИС фирмы Altera APEX20KE как с асинхронным ПЗУ (мегафункция LPM_ROM). На рис.4.2 показана тестовая схема управляющего автомата процессора в графическом редакторе САПР ПЛИС Quartus II версии 2.0. На рис.4.3 показано содержимое конфигурационного файла ПЗУ.

В описание процессора на языке VHDL добавлен асинхронный сброс регистров A, B и счетчика команд на регистре ip (врезка 1). Декодирование переменной-селектора cmd осуществляется с помощью оператора case. Оператор преобразования типов conv_integer(cmd) переводит вектор в десятичное число отдельных кодов.

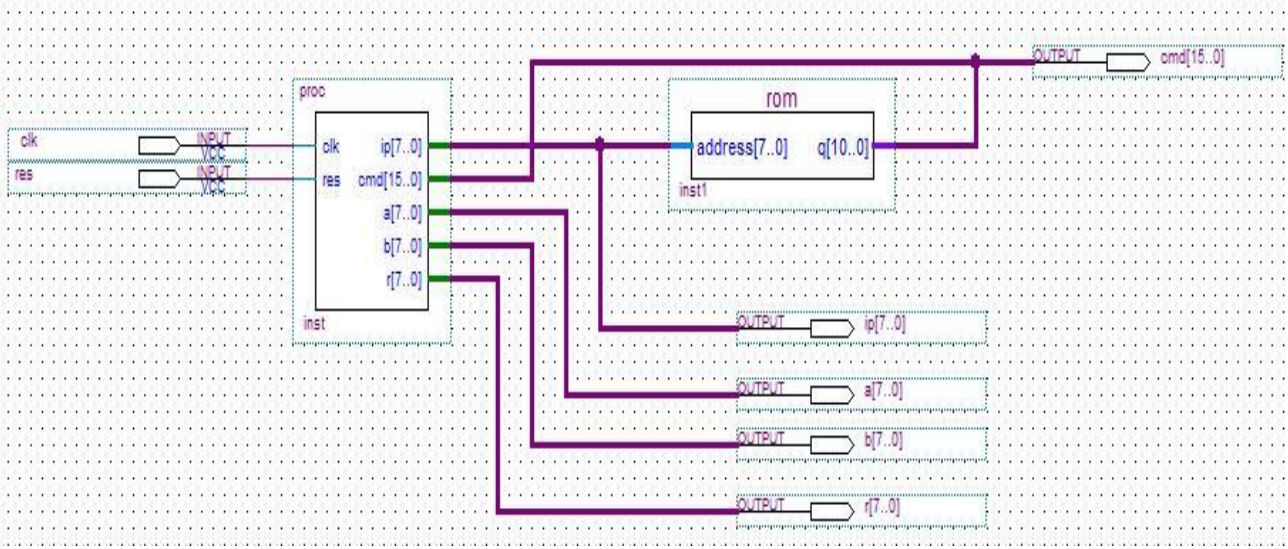


Рис.4.2. Тестовая схема микропроцессорного ядра в графическом редакторе САПР ПЛИС Quartus II версии 2.0 с использованием управляющего автомата с циклом работы в два такта на языке VHDL и асинхронного ПЗУ (мегафункция LPM_ROM)

Таблица 4.1

Система команд процессора с синхронной архитектурой

Код операции	Мнемоника	Описание
0	NOP	Нет операции
01xxH	JMP	Безусловный переход по адресу, заданному младшим байтом команды
02xxH	JMPZ	Переход по адресу, заданному младшим байтом команды, если содержимое регистра A равно нулю
03xxH	CALL	Вызов подпрограммы по адресу, заданному младшим байтом команды
04xxH	MOV A,xx	Непосредственная загрузка в регистр A значения, заданного младшим байтом команды
05xxH	MOV B,xx	Непосредственная загрузка в регистр B значения, заданного младшим байтом команды
0600H	RET	Возврат из подпрограммы
0601H	MOV A,B	Загрузка в регистр A значения, содержащегося в регистре B
0602H	MOV B,A	Загрузка в регистр B значения, содержащегося в регистре A
0603H	XCHG A,B	Обмен местами значений в регистрах A и B
0604H	ADD A,B	Сложение значений в регистрах A и B, результат помещается в A
0605H	SUB A,B	Вычитание значений в регистрах A и B, результат помещается в A
0606H	AND A,B	Побитное логическое И значений в регистрах A и B, результат помещается в A
0607H	OR A,B	Побитное логическое ИЛИ значений в регистрах A и B, результат помещается в A
0608H	XOR A,B	Побитное логическое ИСКЛЮЧАЮЩЕЕ ИЛИ значений в регистрах A и B, результат помещается в A

При каждом допустимом значении кода команды, происходят различные действия, которые состоят в назначении регистрам новых значений в соответствии с описанием команд. Счетчик команд (регистр) не обновляется автоматически, поэтому в каждом варианте кода команды, присваивание счетчику нового значения, указывается явно. Процессор ограничивается двумя регистрами общего назначения (A и B). Процессор имеет указатель инструкций ip и регистр r (стек), для хранения адреса, с которого произошел вызов подпрограммы, поддерживает минимальный набор команд: команда пересылки “регистр - регистр”; команды непосредственной загрузки; команда безусловного перехода к новому адресу; команды перехода по условию; набор арифметико-логических операций. Пример 1 показывает управляющий автомат на языке VHDL.

Наиболее сложными являются команды передачи управления JMP и JMPZ, и команда обращения к подпрограммам CALL и команда возврата из подпрограммы RET. Временные диаграммы на рис.4.4 демонстрируют принцип работы управляющего автомата с асинхронным ПЗУ при отработке команд CALL (0305H) и RET (0600H). При нормальной последовательности работы процессора отрабатываються регистровые команды. Последовательно загружаются регистры A и B. В регистр A загружается число 1D (1H), а в регистр B, число 17D (11H). По команде 0305H происходит запись содержимого счетчика команд ip в регистр r (2D) и загрузка в счетчик команд числа 5D. Таким образом, процессор начнет выполнять подпрограмму хранящуюся в ПЗУ с адреса 5H. По указанному адресу извлекается регистровая команда 0403H. Происходит загрузка в регистр A числа 3H, командой 0404P загрузка числа 4H. Следующей командой с кодом 0604H произойдет сложение содержимых регистров с сохранением результата в регистре A (число 21D). Далее будут отработаны команды 0406H и 0407H. По команде возврата из подпрограммы 600H произойдет изменение содержимого счетчика с 10D на 2D+1D, т.е. на 3D.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity proc is
port (ip: inout std_logic_vector(7 downto 0);
        cmd: inout std_logic_vector(15 downto 0);
        clk,res: in std_logic;
        a: inout std_logic_vector(7 downto 0);
        b,r: inout std_logic_vector(7 downto 0));
end proc;
architecture a of proc is
signal stage: std_logic;
begin
process(clk)
begin
if (res = '1') then
                a <="00000000";
                b <="00000000";
                ip <="00000000";
elsif clk'event and clk='1' then
case stage is
when '0'=> stage<='1';
when others=> stage<='0';
case conv_integer(cmd) is
when 0=> ip <= ip+1;
when 256 to 511 =>ip<=cmd(7 downto 0);
when 512 to 767 =>if conv_integer(a)=0
                                        then ip<=cmd(7 downto 0);
                                        else ip<=ip+1;
                                        end if;
when 768 to 1023 =>r<=ip; ip<=cmd(7 downto 0);
when 1024 to 1279 => a<=cmd(7 downto 0); ip<=ip+1;
when 1280 to 1535 => b<=cmd(7 downto 0); ip<=ip+1;
when 1536 =>ip<=r+1;
when 1537=>a<=b; ip<=ip+1;
when 1538=>b<=a; ip<=ip+1;
```

```
when 1539=>a<=b;b<=a; ip<=ip+1;
when 1540=>a<=a+b; ip<=ip+1;
when 1541=>a<=a-b; ip<=ip+1;
when 1542=>a<=a and b; ip<=ip+1;
when 1543=>a<=a or b; ip<=ip+1;
when 1544=>a<=a xor b; ip<=ip+1;
when 1545=>a<=a-1; ip<=ip+1;
when others=>ip<=ip+1;
end case;
end case;
end if;
end process;
end a;
```

Пример 1. Управляющий автомат на языке VHDL

Addr	+0	+1	+2	+3	+4	+5	+6	+7
00	0401	0511	0305	0512	0402	0403	0404	0604
08	0406	0407	0600	0000	0000	0000	0000	0000

Рис.4.3. Файл конфигурации ПЗУ для тестирования команды обращения к подпрограммам CALL и возврата RET

Рассмотрим вариант процессора без использования управляющего автомата. С этой целью регистровые команды 04xxH, 05xxH, 0601H-0609H предлагается реализовать на тактируемом дешифраторе (пример 2), выполняющим функцию арифметически-логического устройства (АЛУ), а команды передачи управления JMP, JMPZ и обращения к подпрограммам с кодами 01xxH-03xxH, 0600H на 8-ми разрядном суммирующем счетчике адресов памяти команд (пример 3), тактируемым фронтом синхросигнала (рис.4.5).

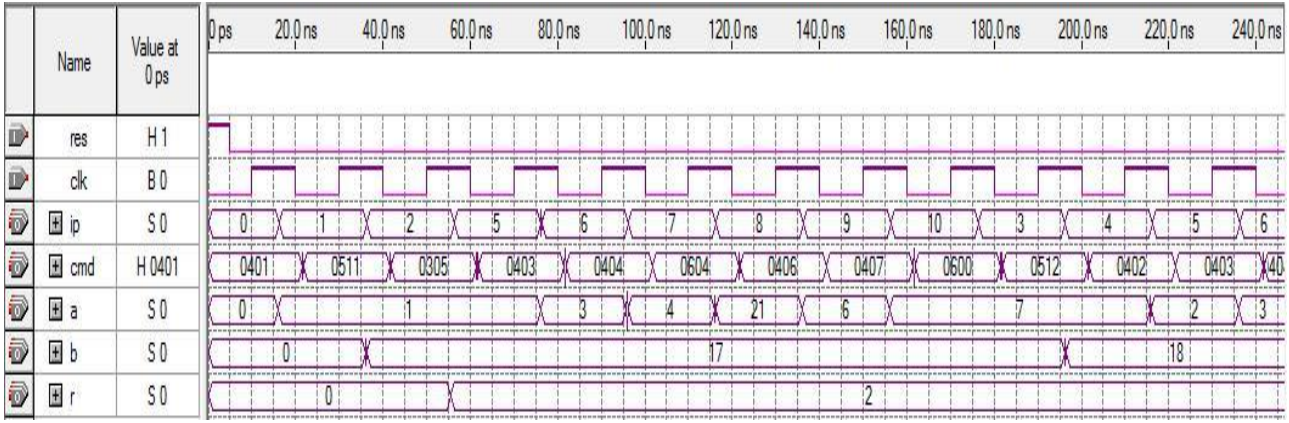


Рис.4.4. Временная диаграмма работы процессора, с использованием управляющего автомата с циклом работы в два такта и мегафункции асинхронного ПЗУ. Отрабатываются регистровые команды и команда вызова подпрограммы с кодом 0305H (CALL) и команда возврата из подпрограммы 0600H (RET)

Счетчик содержит асинхронный сброс Reset. Активным является сигнал высокого уровня. Во вложенных ветвях оператора if происходит проверка условий и синхронная загрузка счетчика команд. Счетчик команд ip и регистр r при инициализации системы по сигналу Reset устанавливаются в состояние 0, после чего производит счет адресов памяти программ хранимых в ПЗУ. Регистр r выполняет функцию стека, в который заносится прежнее состояние счетчика команд. Из шины cmd[10..0] для счетчика команд выделяется поле cop[10..8] и поле data[7..0]. Поле cop означает код операции, который используется для идентификации команд JMP, JMPZ и CALL. Для команды RET поле cop не формируется, а задается полный адрес на шине cmd "11000000000", это связано с тем, что 8 младших бит для команд JMP, JMPZ и CALL могут принимать любые значения, а для команды RET только указанный. Поле data содержит 8-разрядный операнд, который загружается в регистр команд.

ПЗУ реализовано с использованием мегафункции LPM_ROM. В табл.4.2 представлены сведения по общему числу задействованных ресурсов ПЛИС. В обоих случаях проект отображается в ПЛИС APEX20KE (EP20K30ETC144). На рис.4.6 показано тестирование процессора.

Таблица 4.2

Общие сведения по числу задействованных ресурсов ПЛИС APEX20KE

Номер проекта	Общее число логических элементов	Общее число используемых ESB-бит памяти	D-триггеров
С использованием управляющего автомата	198/1200 (16 %)	2816/24576 (11 %)	32
Вариант с асинхронным ПЗУ	164/1200 (13 %)	2816/24576 (11 %)	32

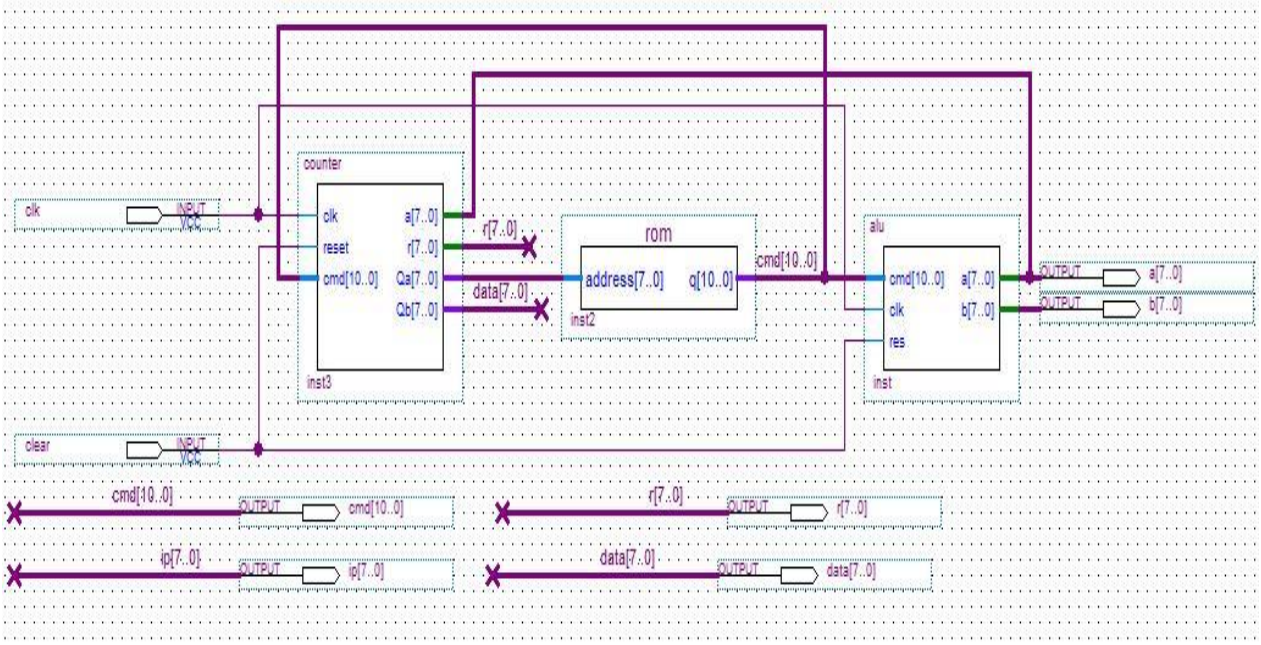


Рис.4.5. Тестовая схема микропроцессорного ядра без использования управляющего автомата с асинхронным ПЗУ (мегафункция LPM_ROM) в графическом редакторе САПР ПЛИС Quartus II версии 2.0

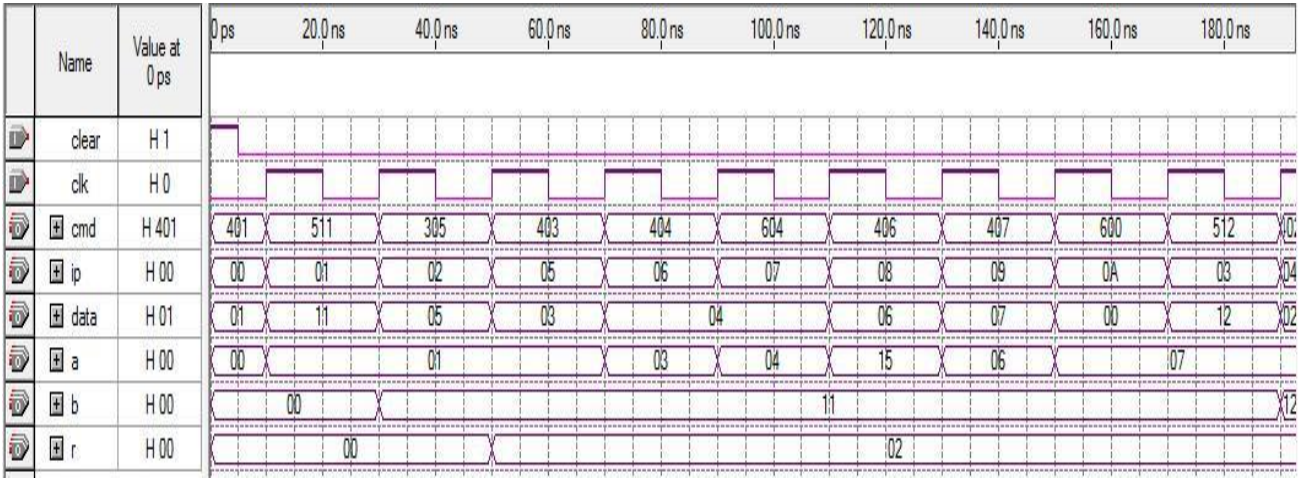


Рис.4.6. Временные диаграммы работы микропроцессорного ядра без использования управляющего автомата с мегафункцией асинхронного ПЗУ

```
LIBRARY ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY alu IS
PORT
(cmd                : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
clk,res          : IN STD_LOGIC;
a,b              : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END alu;
ARCHITECTURE a OF alu IS
signal regA,regB: std_logic_vector(7 downto 0);
BEGIN
PROCESS (clk,res)
BEGIN
        if (RES = '1') then
                        regA <="00000000";
                        regB <="00000000";
        elsif (clk'event and clk='1') then
        case conv_integer(cmd) is
        when 1024 to 1279 => regA<=cmd(7 downto 0);
        when 1280 to 1535 => regB<=cmd(7 downto 0);
        when 1537=>regA<=regB;
        when 1538=>regB<=regA;
        when 1539=>regA<=regB; regB<=regA;
        when 1540=>regA<=regA+regB;
        when 1541=>regA<=regA-regB;
        when 1542=>regA<=regA and regB;
        when 1543=>regA<=regA or regB;
        when 1544=>regA<=regA xor regB;
        when 1545=>regA<=regA-1;
        when others=> a<=regA; b<=regB;
        end case;
end if;
a<=regA;
b<=regB;
END PROCESS;
```

```
END a;
```
Пример 2. Арифметически-логическое устройство процессора

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

ENTITY counter IS
PORT(
clk       : IN  STD_LOGIC;
reset     : IN       STD_LOGIC;
cmd       : IN  STD_LOGIC_VECTOR(10 downto 0);
a          : INOUT  STD_LOGIC_VECTOR(7 downto 0);
r          : INOUT STD_LOGIC_VECTOR(7 downto 0);
Qa,Qb   : OUT STD_LOGIC_VECTOR(7 downto 0));
END counter;
ARCHITECTURE a OF counter IS
SIGNAL pci,pc,pcplus,data,regA: STD_LOGIC_VECTOR(7 downto 0);
SIGNAL cop: STD_LOGIC_VECTOR(2 downto 0);
BEGIN
regA<=a;
cop<=cmd(10 downto 8);
data<=cmd(7 downto 0);
process(clk,reset,cop,data,a)
begin
          if         (reset = '1') then
                                        pci <=(others=>'0');
                                        r <=(others=>'0');
          elsif (clk'event and clk='1') then
          if
          cmd="001"  then pci<=data; --JMP H1
          else if
(cop="010" and conv_integer(regA)=0) then pci<=data; --JMPZ H2
          else if
                    cop="011"  then r<=pci; pci<=data;--CALL H3
          else if
                    cmd="11000000000" then pci<=r+1; --RET H6
```

```
else pci<=pci+1;
end if; end if; end if; end if; end if;
END PROCESS;
Qa <= pci;
Qb <=data;
a<=regA;
END a;
```

Пример 3. Счетчик адресов памяти команд процессора с асинхронным сбросом

Рассмотрим вариант реализации проектируемого процессора с использованием асинхронного ОЗУ (рис.4.7). Для этого воспользуемся мегафункцией LPM_RAM_IO. Для того, чтобы ОЗУ выполняло функцию ПЗУ, необходимо сигнал разрешения записи we “посадить” на землю, т.к. активным является сигнал высокого уровня, а сигнал разрешения вывода outenab подключить к питанию. На рис.4.8 показан файл конфигурации ОЗУ для тестирования команды JMPZ с кодом 0205H. По команде 0205H осуществляется переход по адресу, заданному младшим байтом команды (на адрес в ОЗУ под номером 5, где хранится команда 0403H), если содержимое регистра A равно нулю. Чтобы содержимое регистра A оказалось равным нулю, необходимо воспользоваться регистровыми командами 0405H и 0505H, для загрузки в регистры A и B числа 5, а затем с помощью команды 0605H (SUB A,B) осуществить операцию A-B (рис.4.9).

Данный вариант процессора, реализованный в базисе ПЛИС, можно отнести к классу RISC-процессоров, у которых все команды выполняются за один такт синхрочастоты.

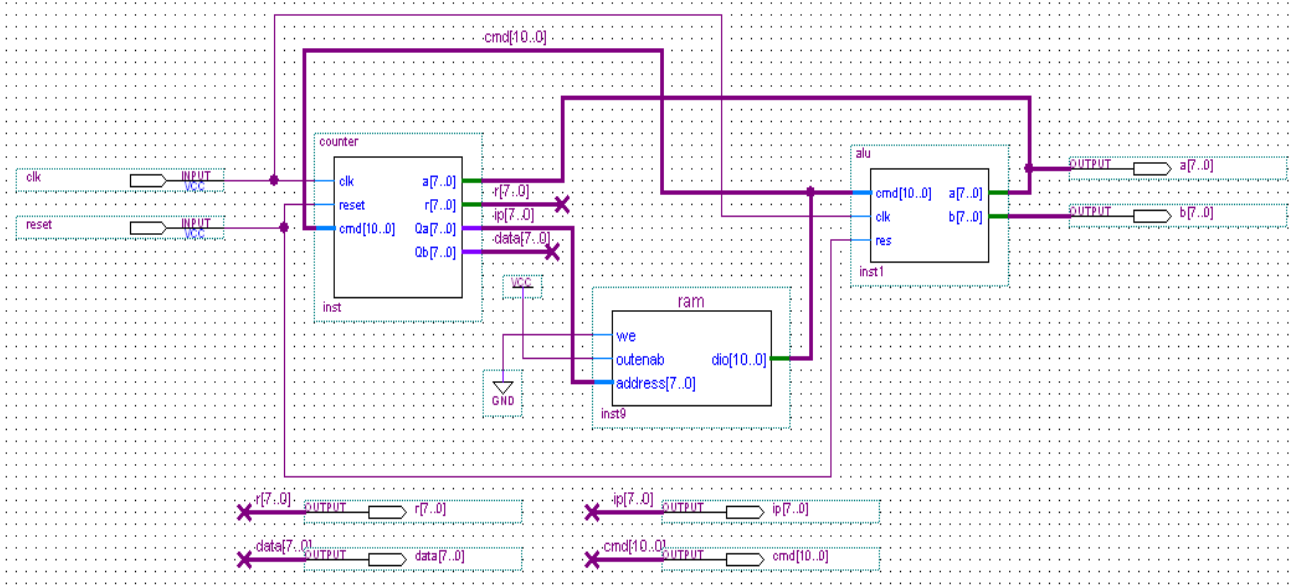


Рис.4.7. Тестовая схема процессора без использования управляющего автомата с асинхронным ОЗУ (мегафункция LPM_RAM_IO) в графическом редакторе САПР ПЛИС Quartus II версии 2.0

Addr	+0	+1	+2	+3	+4	+5	+6	+7
00	0405	0505	0605	0205	0402	0403	0000	0000

Рис.4.8. Файл конфигурации ОЗУ для тестирования команды JMPZ

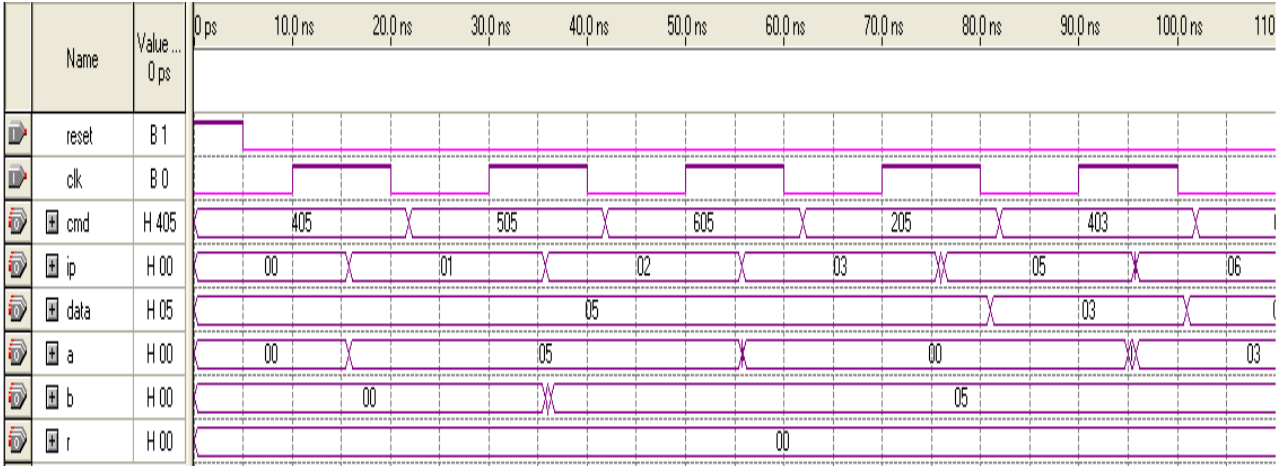


Рис.4.9. Временные диаграммы работы микропроцессорного ядра с мегафункцией асинхронного ОЗУ. Тестирование команды условного перехода JMPZ

Процессор может быть модифицирован путем наращивания блока регистров, добавлением блока управления прерываний, добавления блока управления ввода/вывода и других функциональных блоков. Вариант с асинхронным ПЗУ реализованный в ПЛИС APEX20KE EP20K30ETC144-1 занимает всего лишь 13 % от общего числа логических элементов и способен работать на частоте 60 МГЦ.

4.2. Использование различных типов памяти при проектировании учебного микропроцессорного ядра для реализации в базисе ПЛИС

Предлагается повторно переработать проект микропроцессорного ядра из раздела 4.1 в базисе ПЛИС APEX20KE и Stratix III компании Altera с использованием САПР ПЛИС Quartus II версии 8.1, с целью изучения особенностей использования различных видов памяти. В качестве микропроцессорного ядра используется автомат с циклом работы в два такта.

Особенности современной цифровой микроэлектроники обуславливают преимущественное использование синхронных интерфейсов устройств разного типа, в том числе и памяти. Попытки реализовывать асинхронный интерфейс могут в ряде случаев получать схемы с различными задержками распространения сигналов («гонки фронтов», «перекосы»), которые не выявляются средствами моделирования, но оказывают существенное негативное влияние на характеристики проекта, загружаемого в ПЛИС, вплоть до неработоспособности схемы или перемежающихся неисправностей.

Семейство ПЛИС Stratix III позволяет реализовать на одном кристалле булевы логические функции с помощью адаптивных логических блоков, блоки памяти, при этом память может быть реализована без затрат основной логики и блоки

цифровой обработки сигналов. ПЛИС APEX20KE содержит лишь встроенные блоки памяти (встроенное ОЗУ/ПЗУ).

ПЛИС Stratix имеют структуру памяти TriMatrix, составленную из встроенных блоков памяти RAM трех видов: блоки MLAB (Memory LAB, емкость блока 320 бит, блок может быть представлен как простое двух портовое ОЗУ); блоки M9K (емкость блока 9216 бит); блоки M144K (емкость 147456 бит). Также в ПЛИС Stratix III встроена дополнительная собственная память (встроенное реконфигурируемое ОЗУ). Блоки MLAB используются для реализации сдвиговых регистров, буферов FIFO, линий задержек для цифровых фильтров и др. Блоки M9K используются как блоки памяти общего назначения. Блоки M144K используются для хранения исполняемого кода синтезируемых процессорных ядер, для реализации буферов большого объема в задах видеообработки сигналов. Все блоки памяти TriMatrix поддерживают синхронный режим работы. Блоки M9K и M144K в ПЛИС Stratix не поддерживают асинхронную память, а блок MLAB поддерживает только асинхронный режим чтения данных. Каждый из блоков памяти с помощью мегафункции altsyncram может быть сконфигурирован как: RAM: 1 PORT, RAM: 2 PORT, RAM: 3 PORT, ROM: 1 PORT, ROM: 2 PORT, shift register (RAM-based), FIFO.

Рассмотрим два варианта построения микропроцессорного ядра: с асинхронным ПЗУ (рис.4.10, а) с использованием устаревших серий ПЛИС, например, ПЛИС APEX20KE и синхронным ПЗУ (рис.4.10, б) с использованием ПЛИС Stratix III. Вариант микропроцессорного ядра с мегафункцией асинхронного ПЗУ (LPM_ROM) рассмотрен в разделе 4.1 с использованием САПР ПЛИС Quartus II версии 2.0. В обоих случаях емкость ПЗУ 256 слов на 16 бит, т.е. входная адресная шина 8-ми (address[7..0]), а выходная шина данных (q[15..0]) 16-ти разрядная. Файл прошивки ПЗУ приведен ниже (mif – файл, пример 1).

```
-- Quartus II generated Memory Initialization File (.mif)
WIDTH=16;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;
CONTENT BEGIN
        000 :  0401;
        001 :  0511;
        002 :  0305;
        003 :  0512;
        004 :  0402;
        005 :  0403;
        006 :  0404;
        007 :  0604;
        008 :  0406;
        009 :  0407;
        00A :  0600;
        [00B..0FF] :  0000;
END;
```

Пример 1. Файл прошивки ПЗУ

Мегафункция LPM_ROM для ПЛИС Stratix III работает в режиме совместимости. Поэтому в случае повторной переработки проектов выполненных на устаревших сериях ПЛИС, например на APEX20KE, при смене серии ПЛИС на Stratix III (рис.4.11, а, галочка Match project/default снята) возникает предупреждение, что мегафункция LPM_ROM будет сконфигурирована на синхронный режим работы. Возникнет предупреждение о рекомендации к использованию мегафункции altsyncram (ROM: 1 PORT). Фактически произойдет замещение мегафункции LPM_ROM на мегафункцию ROM: 1 PORT.

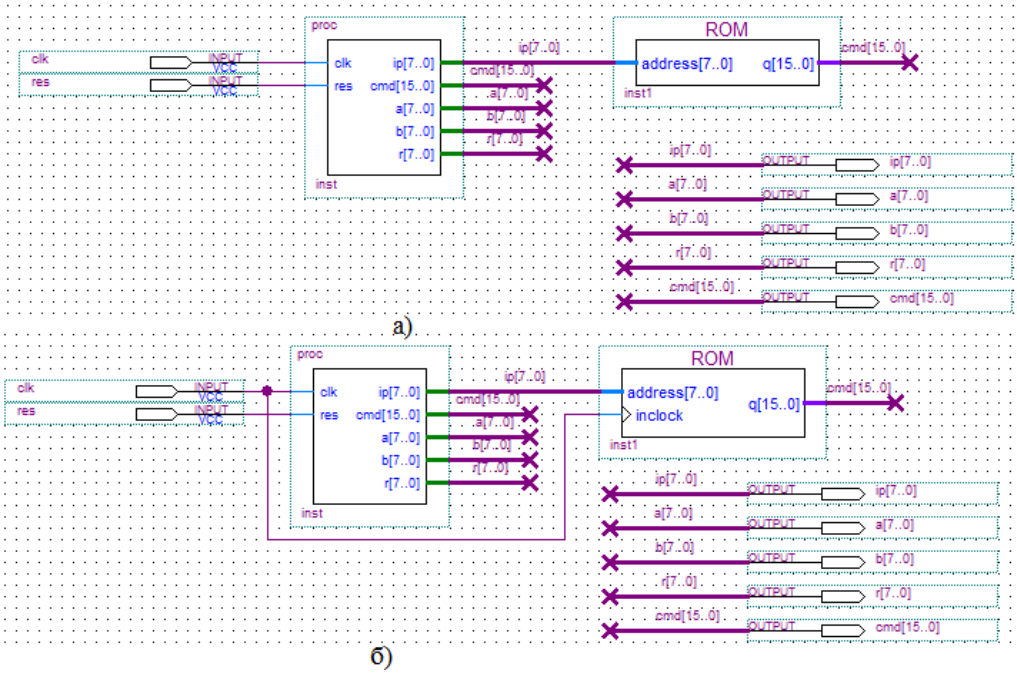
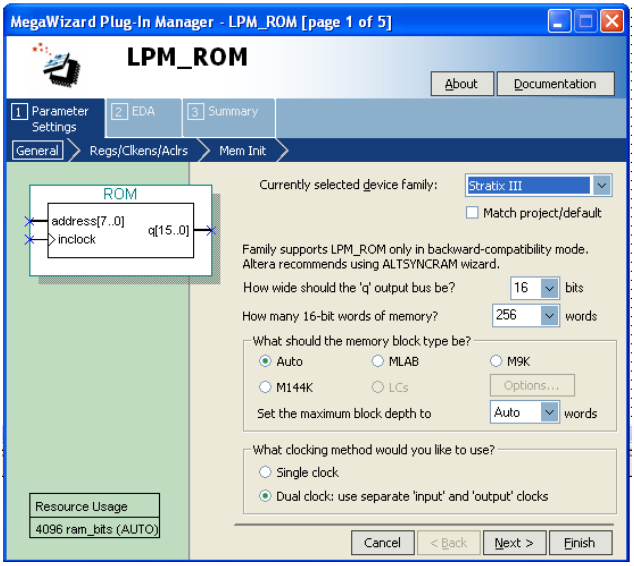
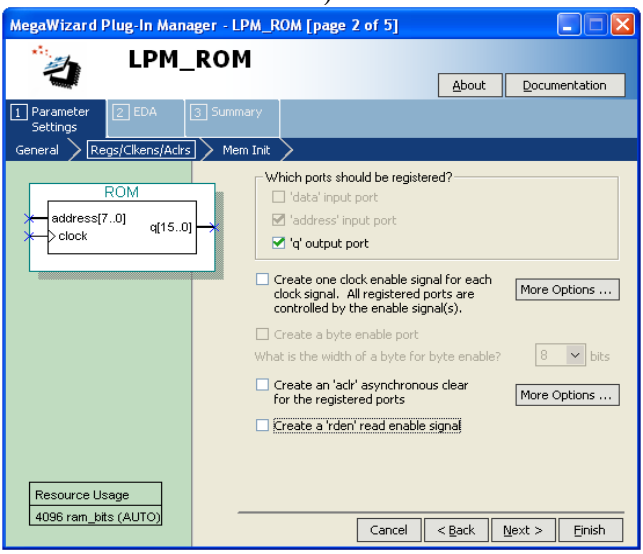


Рис.4.10. Микропроцессорное ядро с асинхронным ПЗУ в базисе ПЛИС APEX20KE (а) и синхронным (б) ПЗУ в базисе ПЛИС Stratix III (САПР ПЛИС Quartus II версии 8.1)



а)



б)

Рис.4.11. Менеджер по работе с мегафункцией ПЗУ LPM_ROM ПЛИС Quartus II версии 8.1 для ПЛИС Stratix III в режиме совместимости: а) шаг 1; б) шаг 2

При настройке мегафункции для ПЛИС на Stratix III пользователь должен самостоятельно выбрать один из типов блоков памяти или выбрать тип Auto. Для ПЛИС APEX20KE доступен только тип Auto (мегафункция LPM_ROM), что объясняется архитектурными особенностями данной ПЛИС, а именно встроенными блоками памяти. Если выбрать тип Auto для ПЛИС Stratix III, то по умолчанию доступны 4096 бит памяти (зависит от разрядности адресной и выходной шины данных).

Галочку с ‘address’ input port снять нельзя, т.е. адресный порт по умолчанию настраивается как регистерный (на условном обозначении появляется указатель динамического входа – треугольник, т.е. адресация к словам ПЗУ осуществляется по переднему фронту синхроимпульса inclock, а выходной порт q – асинхронный) (рис.4.11, б, мегафункции LPM_ROM для ПЛИС Stratix III, шаг 2). Это говорит о том что ПЗУ будет сконфигурирована на синхронный режим работы. По желанию работу выходного порта q можно также синхронизовать сигналом outclock, таким образом можно реализовать режим двойного тактирования.

Использование внутренних триггеров адаптивных логических модулей в качестве памяти ПЗУ в мегафункции для ПЛИС на Stratix III недопустимо, поэтому опция LC (логические элементы, задействуется основная логика: таблицы перекодировок и внутренние триггеры логических элементов) не доступна. Опция LC доступна при разработке ОЗУ без предварительной конфигурации. В случае выбора опции MLAB и режима Auto доступны в качестве памяти 20 таблиц перекодировок (LUT-таблица, таблица перекодировок или логическая таблица, служит для реализации комбинационных функций и может быть упрощенно представлена как столбец ОЗУ с мультиплексором) + 7 MLAB + 24 триггера. При использовании одного из трех типов блоков памяти MLAB, M9K и M144K, дополнительный режим Auto назначает максимально доступный к использованию объем памяти. Число

используемых слов для каждого блока памяти может быть ограничено пользователем.

Чтобы избежать данного предупреждения и повторно переработать проект с использованием мегафункции однопортового ПЗУ (ROM: 1 PORT) для ПЛИС Stratix III, необходимо удалить из проекта блок памяти ПЗУ созданный с помощью мегафункции LPM_ROM и заново с помощью мастера MegaWizard Plugin сгенерировать блок памяти (рис.4.12). Сменив предварительно в Assignments/Device серию ПЛИС APEX20KE на Stratix III.

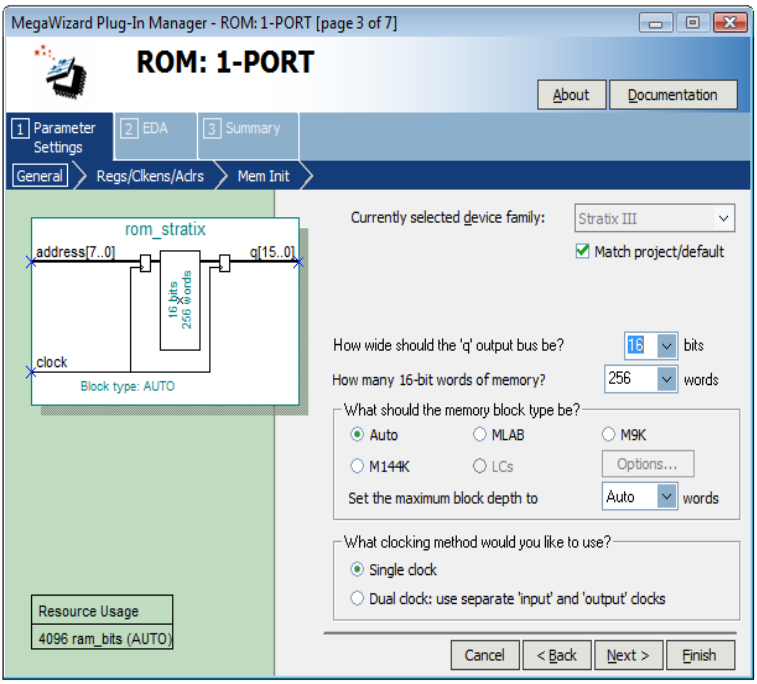


Рис.4.12. Менеджер по работе с мегафункцией однопортовой памяти ПЗУ LPM_ROM ПЛИС Quartus II версии 8.1 для ПЛИС Stratix III (ROM: 1 PORT)

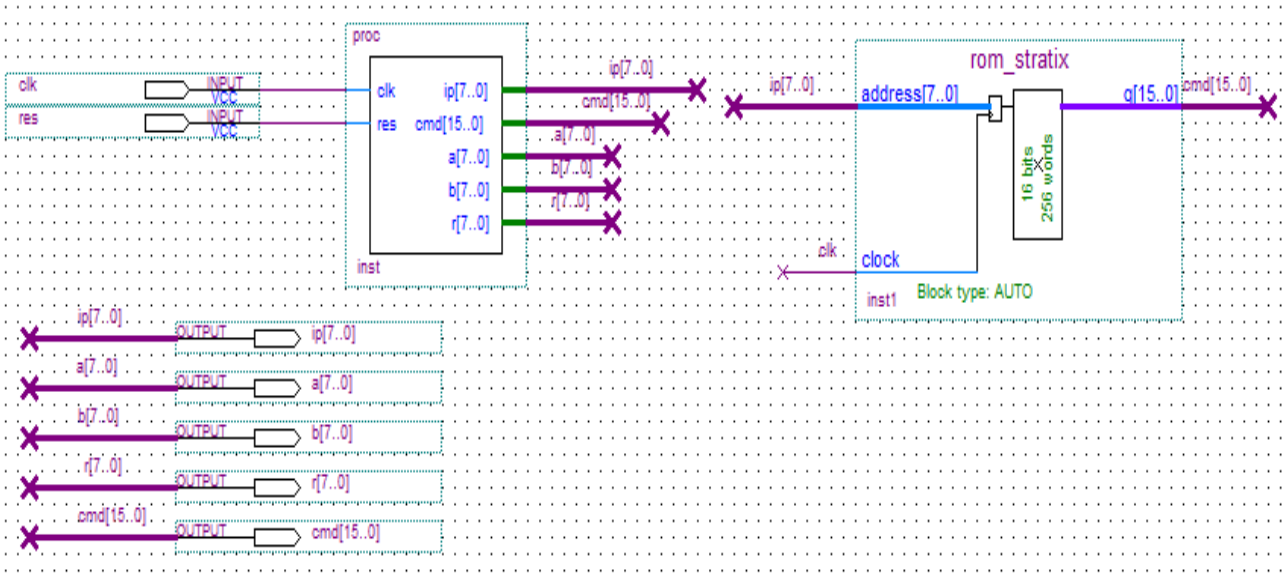


Рис.4.13. Микропроцессорное ядро с синхронным ПЗУ (синхронный режим адресации и асинхронный режим чтения) в базисе ПЛИС Stratix III созданное с помощью мегафункции ROM: 1 PORT САПР ПЛИС Quartus II версии 8.1

На рис.4.13 показано микропроцессорное ядро с синхронным ПЗУ в базисе ПЛИС Stratix III созданное с помощью мегафункции ROM: 1 PORT. Сравнивая рис.4.11 и рис.4.12, видим, что условное обозначение мегафункции ROM: 1 PORT несколько отличается от обозначения мегафункции LPM_ROM в режиме совместимости. Это означает, что все входы в память и выходы из нее защелкиваются в регистрах.

На рис.4.14 и рис.4.15 показано функциональное и временное моделирование работы микропроцессорного ядра с асинхронным ПЗУ в базисе ПЛИС APEX20KE (а) и синхронным (б) ПЗУ в базисе ПЛИС Stratix III с использованием мегафункции ПЗУ LPM_ROM САПР Quartus II версии 8.1.

Для того чтобы обеспечить переносимость проекта с одной серии ПЛИС на другую, без применения мегафункций, рассмотрим использование языка VHDL для проектирования ПЗУ. Пример 2 демонстрирует проектирование асинхронного ПЗУ на языке VHDL с использованием элементов поведенческого описания (используются абстрактные логические структуры, такие как циклы и процессы). На рис.4.16 показаны результаты функционального моделирования в базисе ПЛИС APEX20KE.

Пример 3 демонстрирует проектирование синхронного ПЗУ на языке VHDL, а на рис.4.17 показаны результаты функционального моделирования в базисе ПЛИС Stratix III. Проект микропроцессора с синхронным ПЗУ на языке VHDL, демонстрирует работоспособность, как на старых, так и на новых сериях ПЛИС фирмы Altera. Пример 4 показывает описание асинхронного ПЗУ с использованием элементов потокового описания (представление на уровне регистровых передач). Оператор выбора case обеспечивает параллельную обработку и используется для выбора одного варианта из нескольких в зависимости от условий.

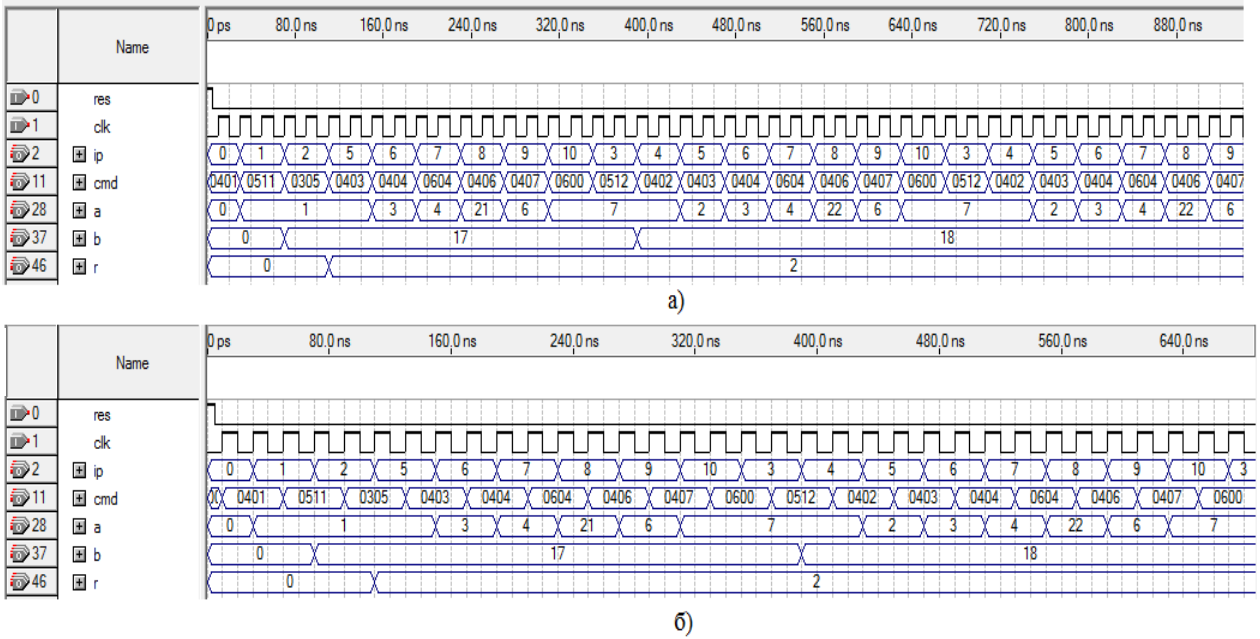
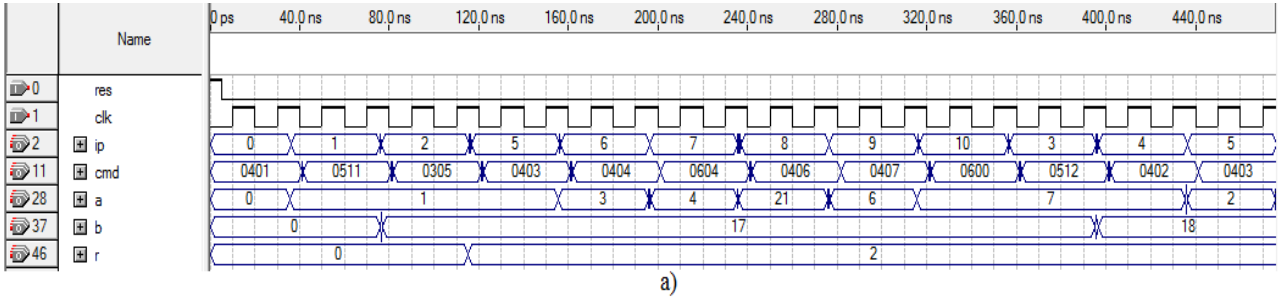
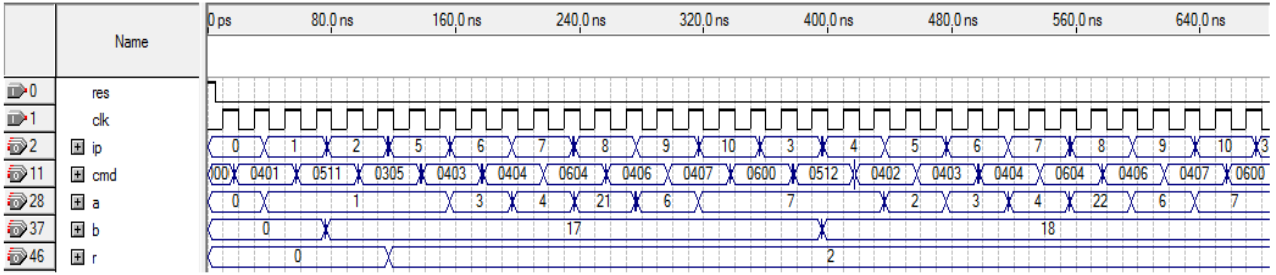


Рис.4.14. Функциональное моделирование работы микропроцессорного ядра с асинхронным ПЗУ в базисе ПЛИС APEX20KE (а) и синхронным (б) ПЗУ в базисе ПЛИС Stratix III с использованием мегафункции ПЗУ LPM_ROM



а)



б)

Рис.4.15. Временное моделирование работы микропроцессорного ядра с асинхронным ПЗУ в базисе ПЛИС APEX20KE (а) и синхронным (б) ПЗУ в базисе ПЛИС Stratix III с использованием мегафункции ПЗУ LPM_ROM

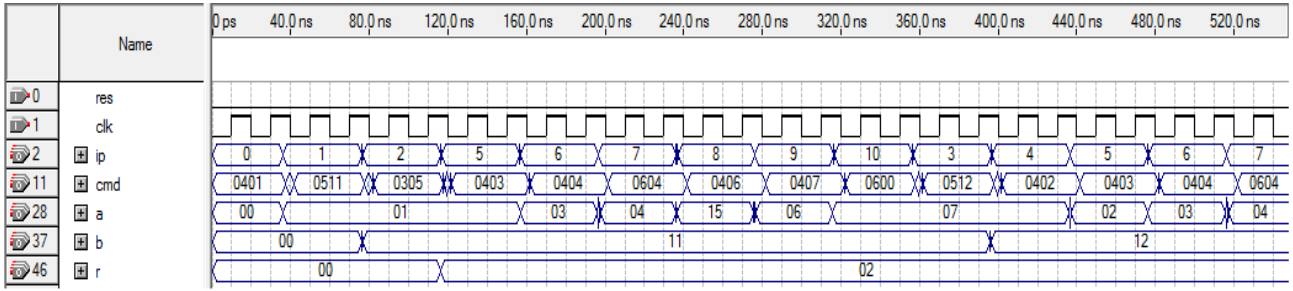


Рис.4.16. Функциональное моделирование работы микропроцессорного ядра с асинхронным ПЗУ на языке VHDL (ПЛИС APEX 20KE)

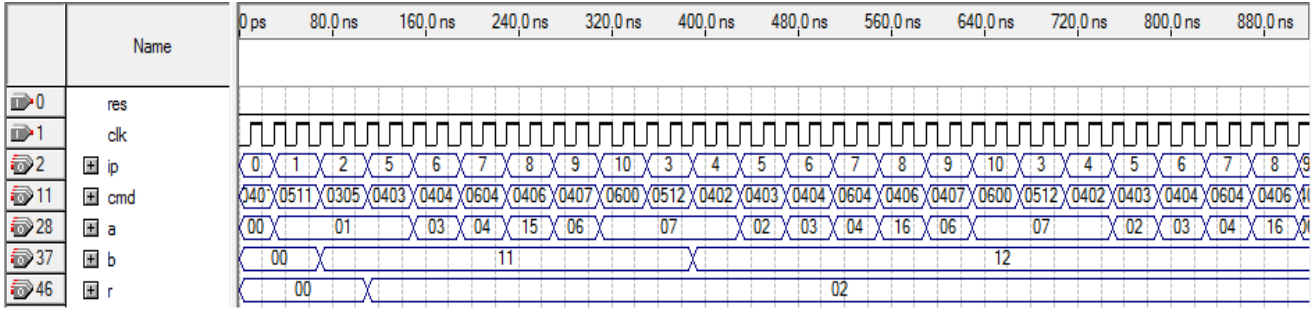


Рис.4.17. Функциональное моделирование работы микропроцессорного ядра с синхронным ПЗУ на языке VHDL (ПЛИС Stratix III)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY rom_syn IS
   PORT (
      clk : IN std_logic;
      addr : IN std_logic_vector(7 DOWNTO 0);
      rom_out : OUT std_logic_vector(15 DOWNTO 0));
END rom_syn;
ARCHITECTURE a OF rom_syn IS
   TYPE T_UFIX_16_256 IS ARRAY (255 DOWNTO 0) of
unsigned(15 DOWNTO 0);
BEGIN
  PROCESS (addr)
      -- local variables
      VARIABLE data_temp : T_UFIX_16_256;
   BEGIN
      FOR b IN 0 TO 255 LOOP
         data_temp(b) := to_unsigned(0, 16);
      END LOOP;
      data_temp(0) := to_unsigned(1025, 16);
      data_temp(1) := to_unsigned(1297, 16);
      data_temp(2) := to_unsigned(773, 16);
      data_temp(3) := to_unsigned(1298, 16);
      data_temp(4) := to_unsigned(1026, 16);
      data_temp(5) := to_unsigned(1027, 16);
      data_temp(6) := to_unsigned(1028, 16);
      data_temp(7) := to_unsigned(1540, 16);
      data_temp(8) := to_unsigned(1030, 16);
      data_temp(9) := to_unsigned(1031, 16);
      data_temp(10) := to_unsigned(1536, 16);
   rom_out <= std_logic_vector(data_temp(to_integer(unsigned(addr))));
   END PROCESS;
END a;
```

Пример 2. Описание асинхронного ПЗУ на языке VHDL

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY ozu_sin IS
   PORT (
      clk : IN std_logic;
      reset : IN std_logic;
      addr : IN std_logic_vector(7 DOWNTO 0);
      rom_out : OUT std_logic_vector(15 DOWNTO 0));
END ozu_sin;
ARCHITECTURE a OF ozu_sin IS
TYPE T_UFIX_16_256 IS ARRAY (255 DOWNTO 0) of unsigned(15
DOWNTO 0);
SIGNAL data, data_next: T_UFIX_16_256;
BEGIN
  PROCESS (reset, clk)
     VARIABLE b : INTEGER;
   BEGIN
      IF reset = '1' THEN
         FOR b IN 0 TO 255 LOOP
            data(b) <= to_unsigned(0, 16);
         END LOOP;
      ELSIF clk'EVENT AND clk= '1' THEN
            FOR b IN 0 TO 255 LOOP
               data(b) <= data_next(b);
            END LOOP;
      END IF;
   END PROCESS;
   PROCESS (addr)
      -- local variables
          VARIABLE data_temp : T_UFIX_16_256;
   BEGIN
      FOR b IN 0 TO 255 LOOP
         data_temp(b) := to_unsigned(0, 16);
      END LOOP;
      data_temp(0) := to_unsigned(1025, 16);
      data_temp(1) := to_unsigned(1297, 16);
      data_temp(2) := to_unsigned(773, 16);
      data_temp(3) := to_unsigned(1298, 16);
```

```
      data_temp(4) := to_unsigned(1026, 16);
      data_temp(5) := to_unsigned(1027, 16);
      data_temp(6) := to_unsigned(1028, 16);
      data_temp(7) := to_unsigned(1540, 16);
      data_temp(8) := to_unsigned(1030, 16);
      data_temp(9) := to_unsigned(1031, 16);
      data_temp(10) := to_unsigned(1536, 16);
      rom_out <=
std_logic_vector(data_temp(to_integer(unsigned(addr))));
      FOR c IN 0 TO 255 LOOP
         data_next(c) <= data_temp(c);
      END LOOP;
   END PROCESS;
END a;
```

Пример 3. Описание синхронного ПЗУ на языке VHDL (поведенческий уровень)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY ROM IS
PORT( clk                    : IN    std_logic;
      reset                  : IN    std_logic;
      addr                   : IN    std_logic_vector(7 DOWNTO 0);
      cmd                    : OUT   std_logic_vector(15 DOWNTO 0)
      );
END ROM;
ARCHITECTURE rtl OF ROM IS
  -- Signals
  SIGNAL s                      : unsigned(7 DOWNTO 0);
  SIGNAL Lookup_Table_out1      : unsigned(15 DOWNTO 0);
BEGIN
  s <= unsigned(addr);
  PROCESS(s)
  BEGIN
    CASE s IS
WHEN "00000000" => Lookup_Table_out1 <= "0000010000000001";
```

```
WHEN "00000001" => Lookup_Table_out1 <= "0000010100010001";
WHEN "00000010" => Lookup_Table_out1 <= "0000001100000101";
WHEN "00000011" => Lookup_Table_out1 <= "0000010100010010";
WHEN "00000100" => Lookup_Table_out1 <= "0000010000000010";
WHEN "00000101" => Lookup_Table_out1 <= "0000010000000011";
WHEN "00000110" => Lookup_Table_out1 <= "0000010000000100";
WHEN "00000111" => Lookup_Table_out1 <= "0000011000000100";
WHEN "00001000" => Lookup_Table_out1 <= "0000010000000110";
WHEN "00001001" => Lookup_Table_out1 <= "0000010000000111";
WHEN "00001010" => Lookup_Table_out1 <= "0000011000000000";
WHEN OTHERS => Lookup_Table_out1 <= "0000000000000000";
END CASE;
END PROCESS;
cmd <= std_logic_vector(Lookup_Table_out1);
END rtl;
```

Пример 4. Описание асинхронного ПЗУ на языке VHDL (уровень регистровых передач)

Отредактируем исходный код языка VHDL управляющего автомата микропроцессорного ядра, приведенного в разделе 4.1, с использованием перечислимых типов. Применение перечислимых типов преследует две цели: улучшение смысловой читаемости программы; более четкий и простой визуальный контроль значений. Наиболее часто перечислимый тип используется для обозначения состояний конечных автоматов.

Перечислимый тип объявляется путем перечисления названий элементов-значений. Объекты, тип которых объявлен как перечислимый, могут содержать только те значения, которые указаны при перечислении, следовательно, количество всех возможных значений конечно. Объявление перечислимого типа имеет вид:

TYPE *имя_типа* **IS (** *название_элемента* [**,** *название_элемента*] **);**

```
Type State_type IS (stateA, stateB, stateC);
VARIABLE State : State_type;
```

.
State := stateB;

В данном примере объявляется переменная State, допустимыми значениями которой являются stateA, stateB, stateC. Пример 4 демонструет использование перечислимого типа для проектирования управляющего автомата.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity proc is
port (ip: inout std_logic_vector(7 downto 0);
        cmd: inout std_logic_vector(15 downto 0);
        clk,res: in std_logic;
        a: inout std_logic_vector(7 downto 0);
        b,r: inout std_logic_vector(7 downto 0));
end proc;
architecture a of proc is

TYPE state_values IS (st0, st1);
signal state, next_state: state_values;

begin
process(clk)
begin
if (res = '1') then
                a <="00000000";
                b <="00000000";
                ip <="00000000";
                r <="00000000";
elsif clk'event and clk='1' then
case state is
when st0=> next_state <=st1;
case conv_integer(cmd) is
when 0=> ip <= ip+1;
when 256 to 511 =>ip<=cmd(7 downto 0);
```

```
when 512 to 767 =>if conv_integer(a)=0
                                    then ip<=cmd(7 downto 0);
                                    else ip<=ip+1;
                                    end if;
when 768 to 1023 =>r<=ip; ip<=cmd(7 downto 0);
when 1024 to 1279 => a<=cmd(7 downto 0); ip<=ip+1;
when 1280 to 1535 => b<=cmd(7 downto 0); ip<=ip+1;
when 1536 =>ip<=r+1;
when 1537=>a<=b; ip<=ip+1;
when 1538=>b<=a; ip<=ip+1;
when 1539=>a<=b;b<=a; ip<=ip+1;
when 1540=>a<=a+b; ip<=ip+1;
when 1541=>a<=a-b; ip<=ip+1;
when 1542=>a<=a and b; ip<=ip+1;
when 1543=>a<=a or b; ip<=ip+1;
when 1544=>a<=a xor b; ip<=ip+1;
when 1545=>a<=a-1; ip<=ip+1;
when others=>ip<=ip+1;
end case;
when st1=> next_state<=st0;
end case;
end if;
end process;
end a;
```

Пример 5. Описание управляющего автомата микропроцессорного ядра с использованием перечислимого типа на языке VHDL с циклом работы в два такта

Изучая рис.4.14-4.17, видим, что функциональное и временное моделирование подтверждают правильность работы микропроцессорного ядра, реализованного в базисах ПЛИС APEX20KE и Stratix III САПР ПЛИС Quartus II версии 8.1. Независимо от типа используемой памяти (синхронный или асинхронный режим работы) процессор работает в два такта. По первому такту синхроимпульса происходит выборка и дешифрирование команды, а по второму такту происходит непосредственная отработка команды, например, выдача

результатов в регистры общего назначения или загрузка новых команд, как в случае с “прыжковыми” командами, например, команды CALL или RET.

Таблица 4.3

Используемые ресурсы ПЛИС Stratix III EP3SL50F484C2

Проект	Logic utilization		Total block memory bits
	Combinati-onal ALUTs	Dedicated logic registers	
Синхронное ПЗУ, ПЛИС Stratix III, мегафункция LPM_ROM, тип Auto	109	33	4096
*Синхронное ПЗУ, ПЛИС APEX20KE EP20K30ETC144, мегафункция LPM_ROM, тип Auto	188 LE		4096
Синхронное ПЗУ, ПЛИС Stratix III, мегафункция ROM: 1 PORT, тип Auto	109	33	4096
Синхронное ПЗУ, ПЛИС Stratix III, мегафункция ROM: 1 PORT, тип Auto (управляющий автомат, пример 5)	94	32	4096
Асинхронное ПЗУ на языке VHDL, пример 4	111	33	-
Синхронное ПЗУ на языке VHDL, управляющий автомат, пример 5	113	33	-

* приводится для сравнения

Для разработки микропроцессорных ядер в базисе ПЛИС Stratix III в качестве синхронного ПЗУ необходимо использовать универсальную мегафункцию altsyncram, которая может быть сконфигурирована как ROM: 1 PORT (мегафункция altsyncram может также использоваться и для конфигурирования ОЗУ), что позволяет снизить нагрузку по

числу используемых логических элементов (табл.4.3) и обеспечить синхронный режим работы ПЗУ. Если управляющий автомат и ПЗУ будут спроектированы с использованием языка VHDL, то в этом случае будет задействована основная логика ПЛИС: таблицы перекодировок (ALUT) и внутренние триггеры логических элементов (logic registers). Из анализа представленной таблицы можно сделать вывод, что в простейшем случае наиболее оптимальным оказывается использование языка VHDL, а блоки памяти используются не эффективно.

4.3. Проектирование учебного процессора для реализации в базисе ПЛИС с использованием системы Matlab/Simulink

Целью данного раздела является демонстрация возможностей системы визуально-имитационного моделирования Matlab/Simulink по проектированию микропроцессорных ядер для реализации в базисе ПЛИС фирмы Altera.

При количестве логических вентилей в проекте свыше 5000 визуализировать выполняемые функции устройства, крайне сложно. Гарантировать, что весь проект можно осознать, только взглянув на его схему, практически невозможно.

Разработчики, использующие в своих проектах методологию системного уровня проектирования (Electronic System Level (ESL) Design - проектирование на системном уровне или проектирование “сверху вниз”), получат очевидные преимущества. Во-первых, любой отдельно взятый разработчик может обеспечить решение задачи повышенной (и постоянно увеличивающейся) сложности, обращаясь к более высоким уровням абстракции и передавая реализацию мелких деталей проекта автоматизированному процессу. Проще и быстрее разрабатывать модели сложно-функциональных устройств на

более высоких уровнях абстракции, чем с использованием уровня регистровых передач (RTL-уровень, register transfer level. Самый простой способ понять концепцию RTL-описания – представить сложно-функциональное устройство в виде совокупности регистров, связанных между собой элементами комбинационной логики и управляемых с помощью общего синхросигнала). Симуляция в этом случае выполняется на порядки быстрее, поскольку не симулируются несущественные для данного уровня абстракции детали.

Во-вторых, разработчики могут значительно сократить цикл производства и улучшить качество изделий благодаря проверке функциональных возможностей ещё на этапе проектирования, когда внесение изменений в системы легко и относительно дешево.

Первым этапом в потоке ESL-проектирования является определение требований к проекту. Требования устанавливаются конкретным заказчиком или определяются в результате соответствующих маркетинговых исследований.

Далее проводится системное проектирование и верификация алгоритмов, как правило, с помощью языка программирования $C / C^{++} / System\ C$ или с помощью специальных средств визуально-иммитационного моделирования типа MathLab/Simulink компании The MathWorks или SPW2000 (signal processing worksystem - рабочая среда обработки сигналов) компании Cadence. На этом этапе могут быть использованы и ESL-продукты компании Summit Design, такие как Visual Elite, FastC, System Architect и Virtual CPU. С помощью текстовых или графических средств создаются исполняемые функциональные спецификации, которые описывают поведение проекта в рамках заданных ограничений. Функциональная верификация технологически независима. Такое описание лишено деталей реализации.

Simulink – графическая среда визуально-имитационного моделирования аналоговых и дискретных систем. Предоставляет пользователю графический интерфейс для

конструирования моделей из стандартных функциональных блоков. Simulink работает с линейными, нелинейными, непрерывными, дискретными и многомерными системами. Система Matlab/Simulink содержит встроенный генератор кода языка описания аппаратных средств HDL (Simulink HDL Coder) и ориентирована на поддержку симулятора VHDL ModelSim. Simulink HDL Coder – программный продукт для генерации VHDL-кода без привязки к конкретной архитектуре ПЛИС и платформе по Simulink-моделям. Справедливости ради, следует заметить, что стиль кодирования может повлиять на производительность проектируемого устройства, особенно на ПЛИС. Логические эквивалентные, но разные RTL-операторы могут выдавать различные результаты.

Покажем возможности системы Matlab/Simulink на этапе ESL-проектирования по созданию микропроцессорных ядер. Воспользуемся системой команд синхронного процессора, реализованного с помощью управляющего автомата, с циклом работы в два такта.

Для учебных целей разработаем два проекта с использованием асинхронного и синхронного ПЗУ на языке VHDL. Для ускорения процесса проектирования предлагается использовать возможности Simulink HDL Coder.

Запуск процессора в Simulink следует производить с использованием отладчика (Simulink Debugger). Перед отладкой необходимо зайти в меню Simulation/Configuration Parameters и выбрать диалог Solver (“решатели”, методы численного решения дифференциальных и дифференциально-алгебраических уравнений). В Solver options выбрать **Type**: Fixed-step; **Solver**: discrete (no continuous state); **Fixed step size (fundamental sample time)** – 1.0.

На рис.4.18 показана отладка процессора. Процессор состоит из следующих блоков: ROM - ПЗУ процессора; COP – блок выделения полей команды; ALU – АЛУ процессора; RON – регистры общего назначения, регистры A и B; RSN – регистры специального назначения, регистр R для обеспечения команд обращения к подпрограммам (CALL) и возврата (RET) и

регистр-счетчик Ip. На рис.4.19 показан файл прошивки ПЗУ. Для построения ПЗУ используется функциональный блок Lookup Table, который формирует таблицу, в строках которой находятся порядковые номера (адреса) команд. Например, строке 1 соответствует команда 1036D (мнемонический код MOV A,12). Адреса команд (входной порт addr), вещественные числа (Real World Values) 0,1,2.. 23, представляются в формате uint8 (Unsigned integer fixed-point data type, целые числа без знака в формате с фиксированной запятой, с 8-ми битной точностью), а команды (выходной порт) представляются в формате uint16, с 16-ти битной точностью. Все сигналы в процессоре представлены в формате uint8 кроме сигналов cmd и cmdData, они представлены в формате uint16.

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

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

где b_i – двоичное число, $b_i = 1,0$; ws – размер слова, ответственно за точность представления десятичных чисел с запятой, максимально 128 бит; S – масштаб, $S = 2^E$, E – число битов левее младшего значащего бита (LSB) в слове, обозначающие месторасположение позиционной точки (разделительной точки или разделительной запятой).

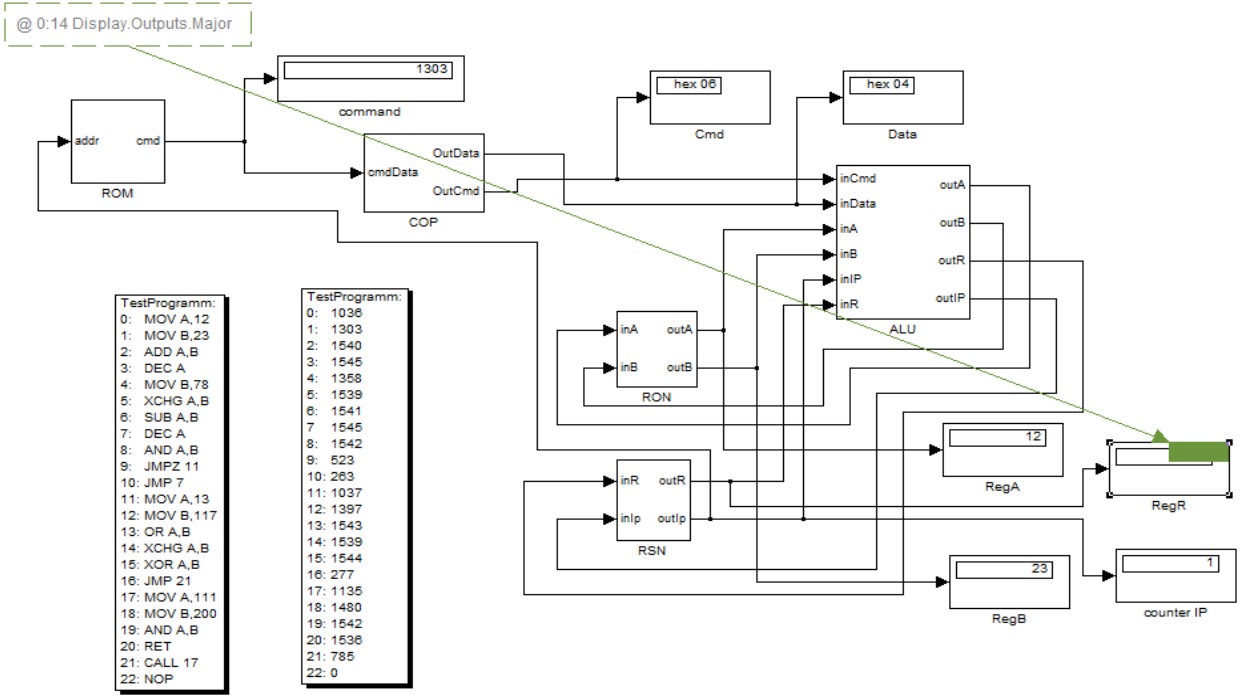


Рис.4.18. Отладка модели микропроцессорного ядра в системе Matlab/Simulink и тестовая программа (файл прошивки ПЗУ, содержимое блока ROM)

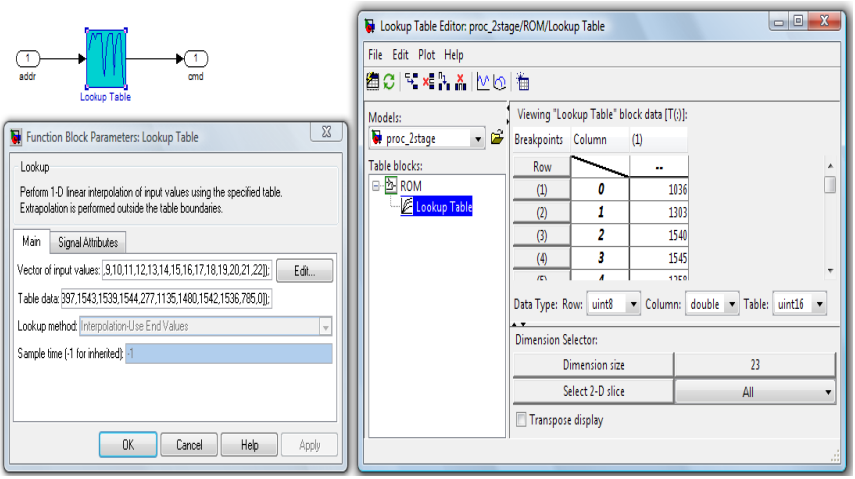


Рис.4.19. Файл прошивки ПЗУ (содержимое блока ROM) в системе Matlab/Simulink с использованием функционального блока Lookup Table

Для целых чисел $S=1$. Например, число 0011.0101 при длине слова $ws=8$:

$$3.3125=2^{-4}\left(\begin{array}{l}0*2^7+0*2^6+1*2^5+\\+1*2^4+0*2^3+1*2^2+0*2^1+1*2^0\end{array}\right).$$

Выделение полей команды осуществляется в блоке COP (рис.4.20). Сигнал OutCmd (uint8) – 8 старших бит 16-ти разрядной команды CmdData (uint16) извлекаемой из ПЗУ. Сигнал OutData (uint8) – 8 младших бит, содержащие данные. Функциональные блоки Shift Arithmetic (арифметический сдвиг) и Data Type Conversion (преобразование типа данных) используются для формирования сигнала OutCmd. Они осуществляют логический сдвиг вправо сигнала CmdData, младшими битами вперед. Для формирования сигнала OutData осуществляется логический сдвиг влево сигнала CmdData, старшими битами вперед, а затем сдвиг вправо, на 8 бит вперед.

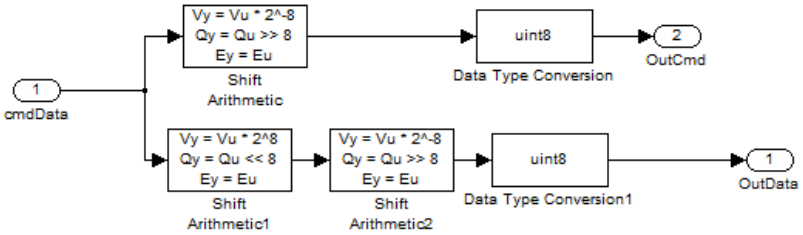


Рис.4.20. Выделение полей команды с помощью операций арифметического сдвига

В блоках Display (command, Cmd, Data, RegA, RegB, RegR, counter IP) отображается содержимое внутренних узлов процессора. Например, по команде MOV A,12 в регистр A загружается число 12, по команде MOV B,23 в регистр B загружается число 23. А по команде 1540 должно произойти сложение содержимых регистров, с сохранением результата в регистре A.

На рис.4.21 показан блок АЛУ. Пример 1 показывает M-файл функции АЛУ в системе Matlab/Simulink. Текст функции начиается с заголовка function, имеющего следующий вид:

function [y1, y2, ...]=fname(x1,x2, ...),

где fname – имя фунции; x1, x2 и т.д. – входные параметры; y1, y2 и т.д. – выходные параметры. В условном операторе switch выбирается одна из возможных альтернатив. Запись оператора выбора производится с помощью ключевых слов switch, case, otherwise. При выполнении оператора switch значение inCmd поочередно сравнивается с 1,2,3,… При обнаружении первого совпадения выполняются операторы соответствующей ветви, после чего производится выход из оператора выбора.

На рис.4.22 показано построение регистров общего назначения с использованием функциональных блоков Memory. Аналогично формируются и блоки регистров специального назначения.

Далее с помощью Simulink HDL Coder сгенерируем код языка VHDL функциональных блоков проектируемого процессора. При этом необходимо помнить, что САПР ПЛИС Quartus II относится к компиляторам-синтезаторам и имеет привязку к конкретной архитектуре, а сгенерированный код языка VHDL с помощью Simulink HDL Coder в первую очередь предназначен для ModelSim SE (Mentor Graphics HDL simulator). Последняя версия САПР ПЛИС Quartus II (версия 8.1) содержит адаптированный симулятор ModelSim-Altera.

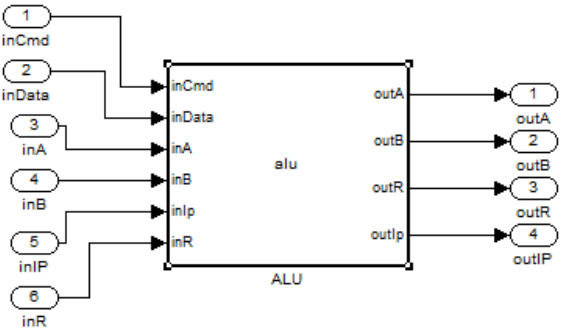


Рис.4.21. Блок АЛУ процессора

```
function [outA, outB, outR, outIp] = alu(inCmd,inData,inA,inB,inIp,inR)
outA = inA;
outB = inB;
outR = inR;
outIp = inIp+1;
switch inCmd
   case 0 %NOP
   case 1 %JMP
      outIp = inData;
   case 2 %JMPZ
      if inA == 0
         outIp = inData;
      end
   case 3 % CALL
      outR = inIp+1;
```

```
        outIp = inData;
    case 4 %MOV A,xx
        outA = inData;
    case 5 %MOV B,xx
        outB = inData;
    case 6
            switch inData
                case 0 %RET
                    outIp = inR;
                case 1 %MOV A,B
                     outA = inB;
                case 2 %MOV B,A
                     outB = inA;
                case 3 %XCHG A,B
                    X = inB;
                    outB = inA;
                    outA = X;
                case 4 %ADD A,B
                    outA = inA+inB;
                case 5 %SUB A,B
                    outA = inA-inB;
                case 6 %AND A,B
                    outA = bitand(inA,inB);
                case 7 %OR A,B
                    outA = bitor(inA,inB);
                case 8 %XOR A,B
                    outA = bitxor(inA,inB);
                case 9 %DEC A
                    outA = inA-1;
            end
end
```

Пример 1. М-файл функции АЛУ в системе Matlab/Simulink

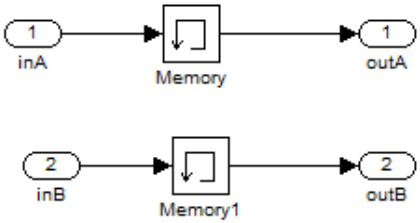


Рис.4.22. Регистры общего назначения процессора

Пример 2 показывает сгенерированное описание прошивки ПЗУ на языке VHDL, пример 3 выделение полей команды, пример 4 поведенческое описание АЛУ, пример 5 – регистры общего назначения. С целью сокращения избыточности кода, все блоки процессора подверглись ручному редактированию. При генерации кода языка VHDL функциональных блоков процессора, Simulink HDL Coder использует только два пакета std_logic_1164 и numeric_std. Так как заранее предполагается, что процессор работает только с целыми положительными двоичными числами (тип unsigned). Добавим в каждый блок следующие пакеты std_logic_arith и std_logic_unsigned. Компилятор-синтезатор наиболее эффективно отображает в ресурсы ПЛИС функции (например, арифметические), которые используются из пакетов std_logic_1164, numeric_std, std_logic_arith и std_logic_unsigned.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
ENTITY ROM IS
PORT(
        addr:   IN    std_logic_vector(7 DOWNTO 0);
        cmd:    OUT   std_logic_vector(15 DOWNTO 0));
END ROM;
```

```
ARCHITECTURE rtl OF ROM IS
SIGNAL Lookup_Table_out1: std_logic_vector(15 DOWNTO 0);

BEGIN
PROCESS(addr)
BEGIN
CASE addr IS
WHEN "00000000" => Lookup_Table_out1 <= "0000010000001100";
WHEN "00000001" => Lookup_Table_out1 <= "0000010100010111";
WHEN "00000010" => Lookup_Table_out1 <= "0000011000000100";
WHEN "00000011" => Lookup_Table_out1 <= "0000011000001001";
WHEN "00000100" => Lookup_Table_out1 <= "0000010101001110";
WHEN "00000101" => Lookup_Table_out1 <= "0000011000000011";
WHEN "00000110" => Lookup_Table_out1 <= "0000011000000101";
WHEN "00000111" => Lookup_Table_out1 <= "0000011000001001";
WHEN "00001000" => Lookup_Table_out1 <= "0000011000000110";
WHEN "00001001" => Lookup_Table_out1 <= "0000001000001011";
WHEN "00001010" => Lookup_Table_out1 <= "0000000100000111";
WHEN "00001011" => Lookup_Table_out1 <= "0000010000001101";
WHEN "00001100" => Lookup_Table_out1 <= "0000010101110101";
WHEN "00001101" => Lookup_Table_out1 <= "0000011000000111";
WHEN "00001110" => Lookup_Table_out1 <= "0000011000000011";
WHEN "00001111" => Lookup_Table_out1 <= "0000011000001000";
WHEN "00010000" => Lookup_Table_out1 <= "0000000100010101";
WHEN "00010001" => Lookup_Table_out1 <= "0000010001101111";
WHEN "00010010" => Lookup_Table_out1 <= "0000010111001000";
WHEN "00010011" => Lookup_Table_out1 <= "0000011000000110";
WHEN "00010100" => Lookup_Table_out1 <= "0000011000000000";
WHEN "00010101" => Lookup_Table_out1 <= "0000001100010001";
WHEN "00010110" => Lookup_Table_out1 <= "0000000000000000";
WHEN OTHERS => Lookup_Table_out1 <= "0000000000000000";
END CASE;
END PROCESS;
cmd <=Lookup_Table_out1;
END rtl;
```

Пример 2. Файл прошивки асинхронного ПЗУ на языке VHDL

```
library ieee;
USE IEEE.std_logic_1164.ALL;
```

```
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
ENTITY COP IS
PORT(
cmdData :   IN    std_logic_vector(15 DOWNTO 0);
OutData :   OUT   std_logic_vector(7 DOWNTO 0);
OutCmd  :   OUT   std_logic_vector(7 DOWNTO 0));
END COP;
ARCHITECTURE rtl OF COP IS
BEGIN
OutCmd<=cmdData(15 downto 8);
OutData<=cmdData(7 downto 0);
END rtl;
```

Пример 3. Выделение полей команды на языке VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
ENTITY ALU_entity IS
PORT (
      inCmd : IN std_logic_vector(7 DOWNTO 0);
      inData : IN std_logic_vector(7 DOWNTO 0);
      inA : IN std_logic_vector(7 DOWNTO 0);
      inB : IN std_logic_vector(7 DOWNTO 0);
      inIp : IN std_logic_vector(7 DOWNTO 0);
      inR : IN std_logic_vector(7 DOWNTO 0);
      outA : OUT std_logic_vector(7 DOWNTO 0);
      outB : OUT std_logic_vector(7 DOWNTO 0);
      outR : OUT std_logic_vector(7 DOWNTO 0);
      outIp : OUT std_logic_vector(7 DOWNTO 0));
END ALU_entity;
ARCHITECTURE fsm_SFHDL OF ALU_entity IS
BEGIN
PROCESS (inCmd, inData, inA, inB, inIp, inR)
   BEGIN
      outA <= inA;
      outB <= inB;
```

```
outR <= inR;
outIp <= inIp+1;
CASE inCmd IS
   WHEN "00000000" =>
      --NOP
      NULL;
   WHEN "00000001" =>
      --JMP
      outIp <= inData;
   WHEN "00000010" =>
      IF inA = 0 THEN
         --JMPZ
         outIp <= inData;
      END IF;
   WHEN "00000011" =>
      -- CALL
     outR <= inIp+1;
      outIp <= inData;
   WHEN "00000100" =>
      --MOV A,xx
      outA <= inData;
   WHEN "00000101" =>
      --MOV B,xx
      outB <= inData;
   WHEN "00000110" =>
      CASE inData IS
         WHEN "00000000" =>
            --RET
            outIp <= inR;
         WHEN "00000001" =>
            --MOV A,B
            outA <= inB;
         WHEN "00000010" =>
            --MOV B,A
            outB <= inA;
         WHEN "00000011" =>
            --XCHG A,B
            outB <= inA;
            outA <= inB;
         WHEN "00000100" =>
```

```
                --ADD A,B
                outA <= inA + inB;
              WHEN "00000101" =>
                --SUB A,B
                outA <= inA - inB;
              WHEN "00000110" =>
                --AND A,B
                outA <= inA AND inB;
              WHEN "00000111" =>
                --OR A,B
                outA <= inA OR inB;
              WHEN "00001000" =>
                --XOR A,B
                outA <= inA XOR inB;
              WHEN "00001001" =>
                --DEC A
                outA <= inA - 1;
              WHEN OTHERS =>
                NULL;
            END CASE;
          WHEN OTHERS =>
            NULL;
        END CASE;
    END PROCESS;
END fsm_SFHDL;
```

Пример 4. Асинхронное АЛУ на языке VHDL

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
ENTITY RON IS
PORT( clk :   IN    std_logic;
      reset :   IN    std_logic;
      enb :   IN    std_logic;
      inA  :   IN    std_logic_vector(7 DOWNTO 0);
      inB :   IN    std_logic_vector(7 DOWNTO 0);
      outA :   OUT   std_logic_vector(7 DOWNTO 0);
      outB :   OUT   std_logic_vector(7 DOWNTO 0));
```

```
END RON;
ARCHITECTURE rtl OF RON IS
BEGIN
  PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      outA <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
      IF enb = '1' THEN
       outA <= inA;
      END IF;
    END IF;
  END PROCESS;
  PROCESS (clk, reset)
  BEGIN
    IF reset = '1' THEN
      outB <= (OTHERS => '0');
    ELSIF clk'event AND clk = '1' THEN
      IF enb = '1' THEN
        outB <= inB;
      END IF;
    END IF;  END PROCESS; END rtl;
```

Пример 5. Регистры общего назначения (РОН)

На рис.4.23 показана тестовая схема микропроцессорного ядра в графическом редакторе САПР ПЛИС Quartus II, построенная по модели, разработанной в системе Matlab/Simulink а временные диаграммы работы на рис.4.24. Анализируя полученные результаты, приходим к выводу, что ПЗУ и АЛУ являются асинхронными блоками. Регистры общего и специального назначения представляют собой 8-ми разрядные регистры, тактируемые фронтом синхросигнала, с асинхронным сбросом reset и синхронным сигналом ena. Недостатком сгенерированного кода языка VHDL является нетактируемое АЛУ и ПЗУ.

Сгенерируем файл прошивки ПЗУ, используя М-файл и fi-объекты системы Matlab. Fi-объекты позволяют представлять числа в формате с фиксированной запятой. Например, по

команде a=fi(1536) целое положительное десятичное число 1536 будет представлено в формате M.N, где M - общее число двоичных разрядов, N - число разрядов дробной части. Наиболее распространенный формат 16.15. Пятнадцать разрядов после запятой обеспечивают дискретность представления равную $2^{-15} \approx 3*10^{-5}$. В нашем случае используем следующий формат: a = fi(v,s,w,f), где v - объект со значением, 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. Например, по команде a=fi(1536, 0, 16, 0) целое положительное десятичное число 1536 будет представлено в формате 16.0. По команде disp(bin(a)) можно посмотреть десятичное число в двоичной форме: 0000011000000000. Пример 6 показывает M-файл прошивки ПЗУ в системе Matlab/Simulink. А пример 7 отредактированный вариант сгенерированного файла программой Simulink HDL Coder в САПР Quartus II.

Анализируя код (пример 7), можно сделать вывод, что сгенерировано синхронное ПЗУ с асинхронным сбросом, синхронным сигналом разрешения тактирования clk_enable. Для организации массива памяти используется последовательный оператор **for ... loop**, выполняющий повторяющиеся операторы. Оператор **for ... loop** имеет целую схему итерации, при которой количество повторов определяется целым диапазоном. Цикл повторяется один раз для каждого значения диапазона. После того, как будет достигнуто последнее значение диапазона итерации, цикл пропускается, и выполнение программы продолжается, начиная с оператора, стоящего вслед за циклом.

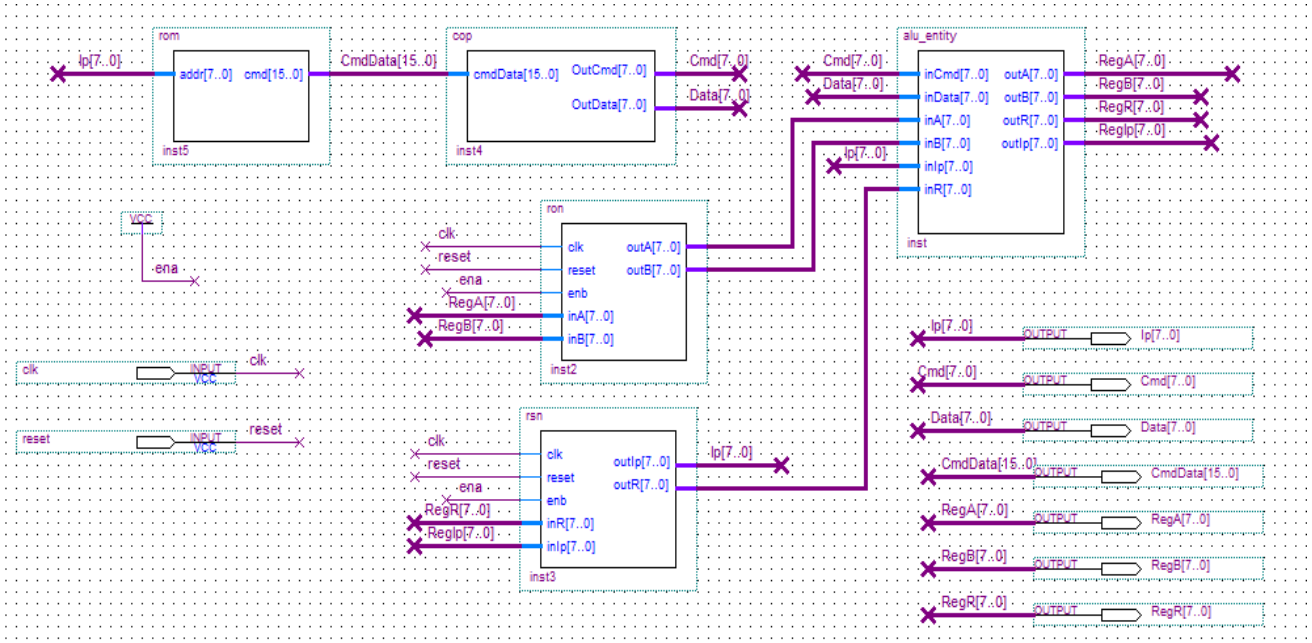


Рис.4.23. Тестовая схема микропроцессорного ядра в графическом редакторе САПР ПЛИС Quartus II версии 8.1, построенная по модели. Код языка VHDL функциональных блоков модели сгенерирован с помощью приложения Simulink HDL Coder системы Matlab/Simulink

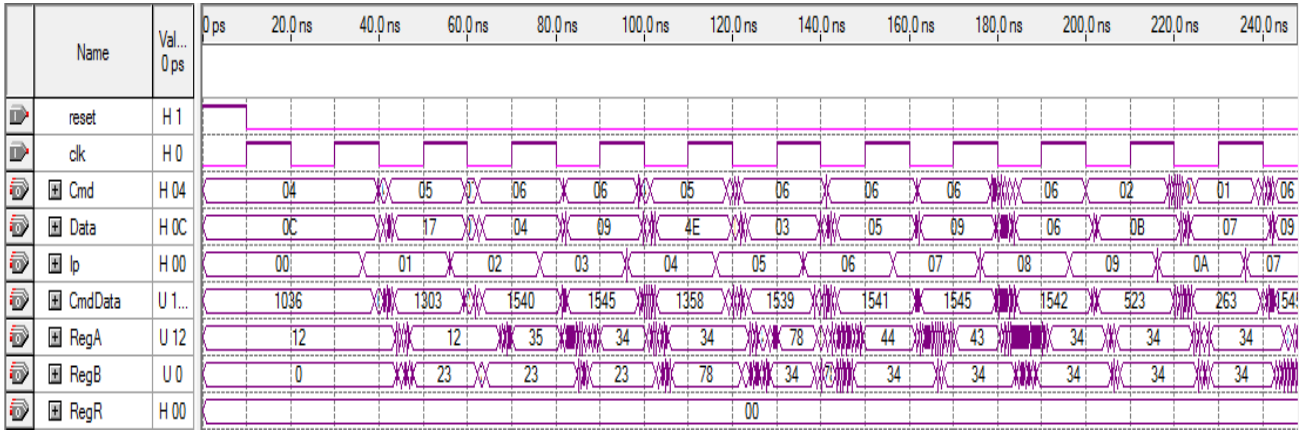


Рис.4.24. Временная диаграмма работы микропроцессорного ядра, код языка VHDL функциональных блоков которого сгенерирован с помощью приложения Simulink HDL Coder системы Matlab/Simulink

```
function rom_out = Memory(addr)
persistent data;
data = fi(zeros(1, 256), 0, 16, 0);
data(1) = fi(1036, 0, 16, 0);
data(2) = fi(1303, 0, 16, 0);
data(3) = fi(1540, 0, 16, 0);
data(4) = fi(1545, 0, 16, 0);
data(5) = fi(1358, 0, 16, 0);
data(6) = fi(1539, 0, 16, 0);
data(7) = fi(1541, 0, 16, 0);
data(8) = fi(1545, 0, 16, 0);
data(9) = fi(1542, 0, 16, 0);
data(10) = fi(523, 0, 16, 0);
data(11) = fi(263, 0, 16, 0);
data(12) = fi(1037, 0, 16, 0);
data(13) = fi(1397, 0, 16, 0);
data(14) = fi(1543, 0, 16, 0);
data(15) = fi(1539, 0, 16, 0);
data(16) = fi(1544, 0, 16, 0);
data(17) = fi(277, 0, 16, 0);
data(18) = fi(1135, 0, 16, 0);
data(19) = fi(1480, 0, 16, 0);
data(20) = fi(1542, 0, 16, 0);
data(21) = fi(1536, 0, 16, 0);
data(22) = fi(785, 0, 16, 0);
data(23) = fi(0, 0, 16, 0);
rom_out = data(addr+1);
```

Пример 6. М-файл прошивки ПЗУ в системе Matlab/Simulink

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY rom_syn IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      addr : IN std_logic_vector(7 DOWNTO 0);
```

```
      rom_out : OUT std_logic_vector(15 DOWNTO 0));
END rom_syn;
ARCHITECTURE a OF rom_syn IS
   TYPE T_UFIX_16_256 IS ARRAY (255 DOWNTO 0) of unsigned(15
DOWNTO 0);
   SIGNAL data : T_UFIX_16_256;
   SIGNAL data_next : T_UFIX_16_256;
BEGIN
   PROCESS (reset, clk)
      -- local variables
      VARIABLE b : INTEGER;
   BEGIN
      IF reset = '1' THEN
         NULL;
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            FOR b IN 0 TO 255 LOOP
               data(b) <= data_next(b);
            END LOOP;
         END IF;
      END IF;
   END PROCESS;
  PROCESS (addr)
      -- local variables
      VARIABLE data_temp : T_UFIX_16_256;
   BEGIN
      FOR b IN 0 TO 255 LOOP
         data_temp(b) := to_unsigned(0, 16);
      END LOOP;
      data_temp(0) := to_unsigned(1036, 16);
      data_temp(1) := to_unsigned(1303, 16);
      data_temp(2) := to_unsigned(1540, 16);
      data_temp(3) := to_unsigned(1545, 16);
      data_temp(4) := to_unsigned(1358, 16);
      data_temp(5) := to_unsigned(1539, 16);
      data_temp(6) := to_unsigned(1541, 16);
      data_temp(7) := to_unsigned(1545, 16);
      data_temp(8) := to_unsigned(1542, 16);
      data_temp(9) := to_unsigned(523, 16);
      data_temp(10) := to_unsigned(263, 16);
```

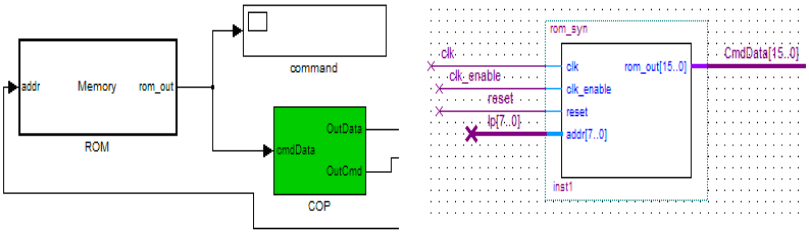
```
    data_temp(11) := to_unsigned(1037, 16);
    data_temp(12) := to_unsigned(1397, 16);
    data_temp(13) := to_unsigned(1543, 16);
    data_temp(14) := to_unsigned(1539, 16);
    data_temp(15) := to_unsigned(1544, 16);
    data_temp(16) := to_unsigned(277, 16);
    data_temp(17) := to_unsigned(1135, 16);
    data_temp(18) := to_unsigned(1480, 16);
    data_temp(19) := to_unsigned(1542, 16);
    data_temp(20) := to_unsigned(1536, 16);
    data_temp(21) := to_unsigned(785, 16);
    data_temp(22) := to_unsigned(0, 16);
  rom_out <= std_logic_vector(data_temp(to_integer(unsigned(addr))));
  END PROCESS;
END a;
```

Пример 7. Файл прошивки синхронного ПЗУ на языке VHDL

Следует упомянуть про функции преобразования типов to_integer и to_unsigned из пакета Numeric_std, которые используются при проектировании синхронного ПЗУ. Функция to_integer преобразует тип unsigned в подтип natural (встроенный подтип natural используют для объектов, которые, не должны принимать отрицательные значения), а функция to_unsigned преобразует подтип natural в тип unsigned, при этом необходимо указывать размер желаемого слова.

На рис.4.25 показаны изменения, которые необходимо внести в проект для ПЗУ с использованием М-функции в системе Matlab/Simulink (рис.4.25, *а*) и для синхронного ПЗУ в САПР ПЛИС Quartus II (рис.4.25, *б*). На рис.4.26 представлена временная диаграмма работы процессора с синхронным ПЗУ.

Проект микропроцессора с синхронным ПЗУ, код языка VHDL которого был получен с использованием Simulink HDL Coder, продемонстрировал работоспособность как на старых, так и на новых сериях ПЛИС фирмы Altera, что не удавалось осуществить с использованием встроенных мегафункций ОЗУ и ПЗУ.



а) б)

Рис.4.25. Фрагмент модели и схемы процессора: а – ПЗУ с использованием М-функции в системе Matlab/Simulink; б - символ синхронного ПЗУ в САПР Quartus II

На рис.4.27 показана тестовая схема процессора в графическом редакторе САПР ПЛИС Quartus II (версия 8) с управляющим автоматом с циклом работы в два такта (пример 1, раздел 4.1) и синхронным ПЗУ на языке VHDL. Сравнивая рис.4.26 и рис.4.28, видим, что два процессора логически работают одинаково.

Система Matlab/simulink с Simulink HDL Coder может быть эффективно использована для ускорения процесса разработки моделей микропроцессорных ядер. Проект микропроцессора с асинхронным ПЗУ на языке VHDL может быть успешно размещен в ПЛИС APEX20KE EP20K30ETC144-1, при этом общее число задействованных ресурсов составляет 22 %, с рабочей частотой до 60 МГц.

Недостатком процессора реализованного в системе Matlab/simulink и адаптированного в САПР Quartus II, является отсутствие управляющего автомата.

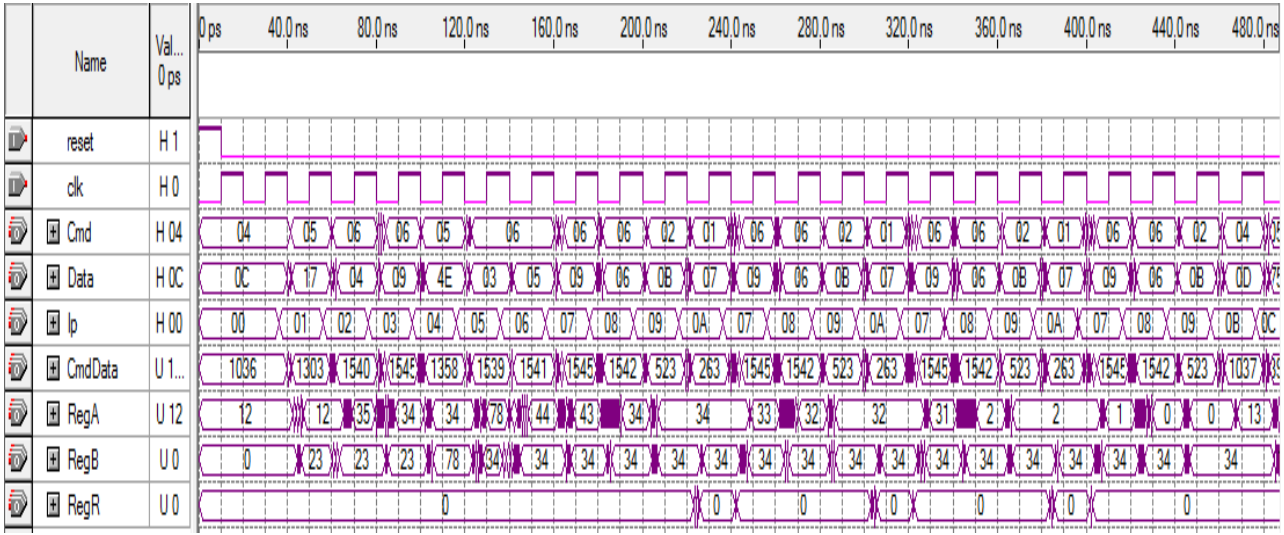


Рис.4.26. Временная диаграмма работы микропроцессорного ядра с синхронным ПЗУ на языке VHDL (код языка VHDL функциональных блоков сгенерирован с помощью приложения Simulink HDL Coder системы Matlab/Simulink)

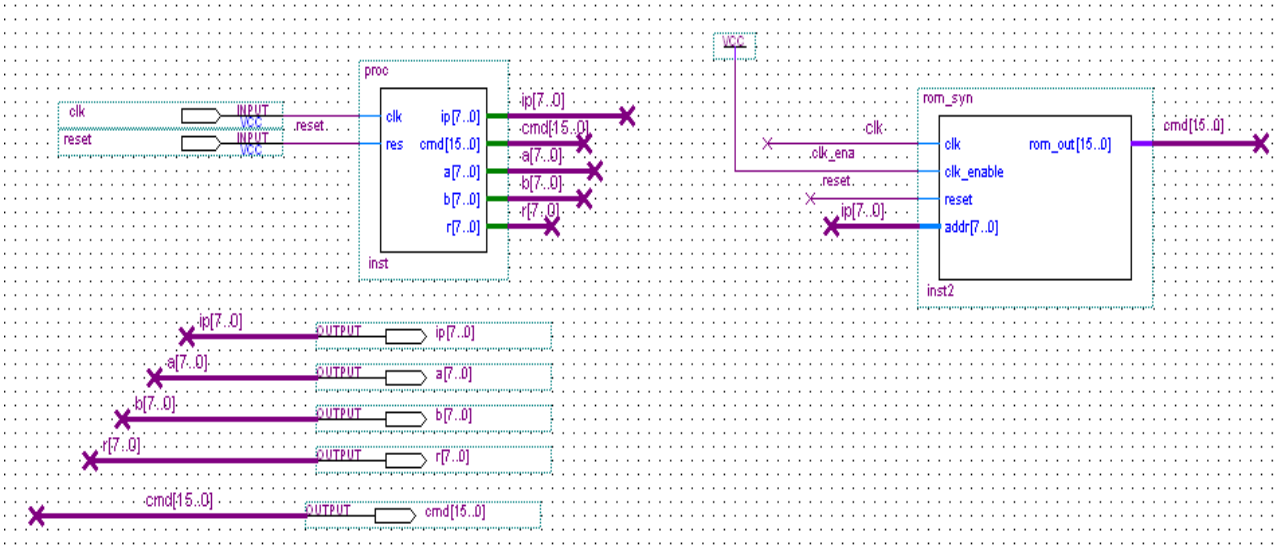


Рис.4.27. Тестовая схема микропроцессорного ядра в графическом редакторе САПР ПЛИС Quartus II (версия 8) с управляющим автоматом с циклом работы в два такта (пример 1, раздел 4.1) и синхронным ПЗУ на языке VHDL, код языка которого сгенерирован с помощью приложения Simulink HDL Coder системы Matlab/Simulink

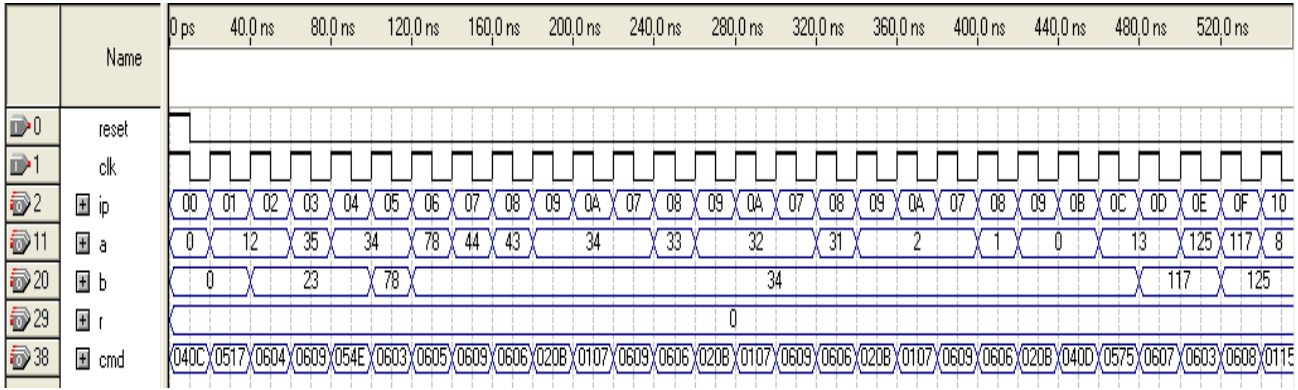


Рис.4.28. Временная диаграмма работы микропроцессорного ядра с управляющим автоматом с циклом работы в два такта (пример 1, раздел 4.1) и синхронным ПЗУ на языке VHDL

4.4. Проектирование учебного процессора с фиксированной запятой в системе Matlab/Simulink

В вышеприведенных примерах показаны примеры проектирования микропроцессорных ядер для реализации в базисе ПЛИС фирмы Altera, как с использованием мегафункций асинхронного ОЗУ/ПЗУ САПР Quartus II, так и с использованием функциональных блоков на языке VHDL, сгенерированных с помощью Simulink HDL Coder системы Matlab/Simulink. Общим недостатком является отсутствие управляющего автомата. Предлагается спроектировать в системе Matlab/Simulink процессор с управляющим автоматом и позволяющим проводить вычисления с фиксированной зяпятой. Выполнение арифметических операций над операндами, представленными в формате с фиксированной запятой, позволяет получать высокую скорость вычислений, но возможно переполнение разрядной сетки либо значительной погрешности из-за округления.

На рис.4.29 показан процессор с управляющим автоматом на шесть состояний и его отладка в системе Matlab/Simulink с использованием отладчика (Simulink Debugger). Перед отладкой необходимо в меню Simulation/Configuration Parameters выбрать диалог Solver (“решатели”, методы численного решения дифференциальных и дифференциально-алгебраических уравнений). В Solver options выбрать **Type**: Fixed-step; **Solver**: discrete (no continuous state); **Fixed step size (fundamental sample time)** – 1.0. Осуществляется тестирование команд MOV A,12; MOV B,23; ADD A,B.

Проектируемый процессор состоит из следующих блоков: управляющий автомат (блок CPU_Controller, пример 1); память программ - ПЗУ процессора (блок Memory, пример 4); АЛУ процессора (блок alu, пример 7); двух регистров общего назначения (РОН, блоки RegisterA, пример 6 и RegisterB); регистра специального назначения (РСН, блок PC_Inc, пример

2), необходимого для обеспечения “прыжковых” команд, таких как JMP, JMPZ, CALL и RET; счетчика команд (блок PC, пример 3); регистра инструкций (блок Instruction_Reg, пример 5).

Процессор реализован в формате с фиксированной запятой, с использованием fi-объектов системы Matlab. Будем используем 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. Все используемые десятичные числа в процессоре беззнаковые (положительные) и целые. В системе Matlab пользователь имеет возможность определить беззнаковые (например, uint8, uint16) и знаковые целые числа (sint), с помощью внутренних форматов.

При проектировании процессоров с фиксированной запятой необходимо учитывать следующие факторы: диапазон для результатов вычислений; требуемую погрешность результата; ошибки, связанные с квантованием; алгоритм реализации вычислений и др.

Это связано с тем, что десятичное число v представляется с использованием формулы: $V \cong 2^{-m} \times Q$, где $m -$ длина дробной части числа; для беззнаковых чисел $Q = \sum_{i=0}^{n-1} W_i \times 2^i$, $W_i -$ весовые коэффициенты, $2^i -$ веса двоичных разрядов машинного слова, $n -$ длина двоичного слова в битах. Диапазон целого беззнакового числа определяется выражением: $0 \le V \le 2^n - 1$.

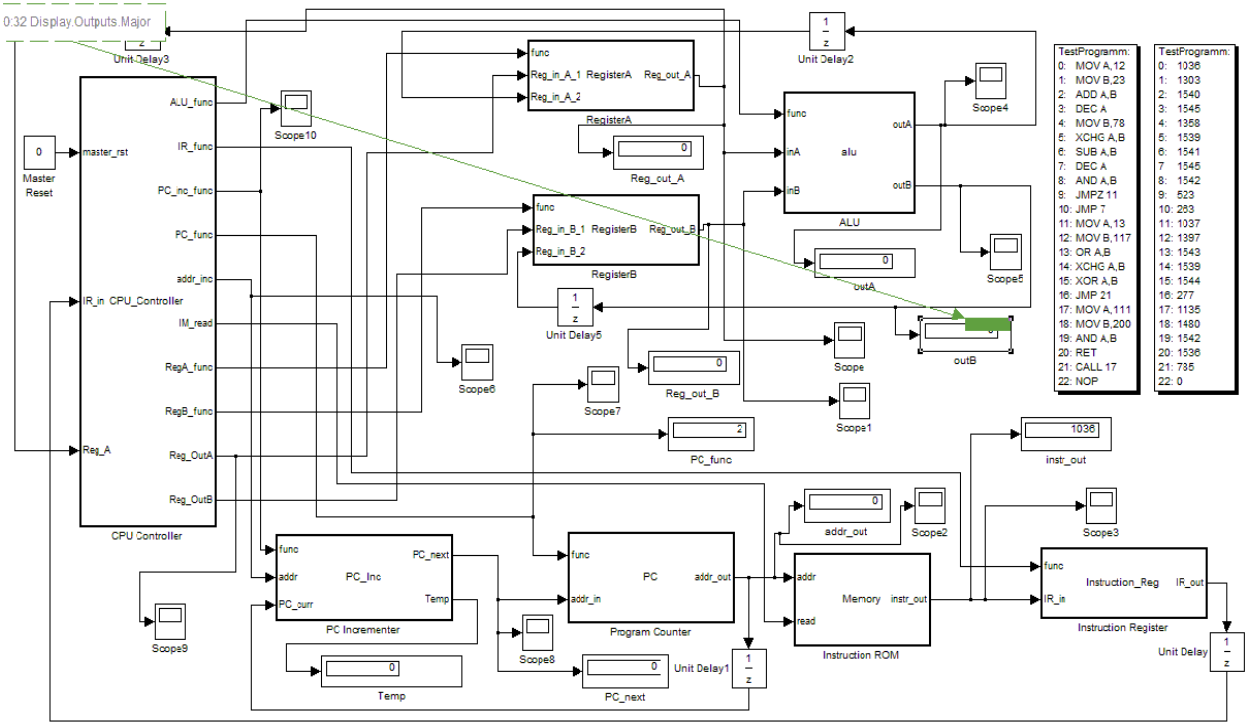


Рис.4.29. Модель процессора с управляющим автоматом в системе Matlab/Simulink. Тестирование команд MOV A,12; MOV B,23; ADD A,B

Это можно осуществить с использованием 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);
```

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.

Для блоков РОН в качестве примера, используем формат a = fi(v, s, w, f, fimath). Можно также добавить учет выше приведенных факторов и в другие М-файлы функций блоков процессора. Это позволит “управлять” встроенным генератором кода языка HDL (Simulink HDL Coder). Если этого не сделать, то необходимо с использованием проводника модели осуществить настройки блоков процессора для вычислений в формате с фиксированной запятой (рис.4.30).

Процессор имеет распределенное управление. В блоках alu, RegisterA, RegisterB, PC_Inc и PC имеется свой локальный управляющий сигнал func, дешифрация которого внутри блоков будет приводить к выполнению некоторых операций, например, к изменению внутреннего содержимого блока или, наоборот, к его сохранению.

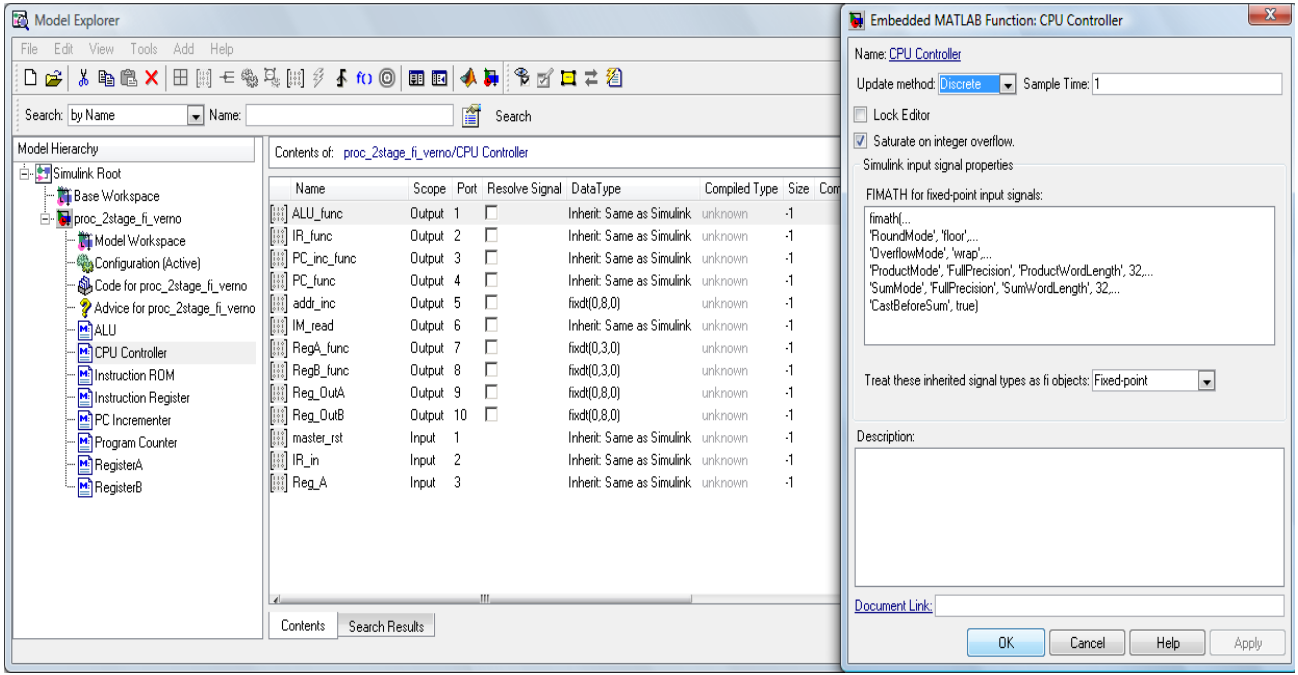


Рис.4.30. Настройка блоков модели процессора с помощью проводника модели для вычислений в формате с фиксированной запятой

Например, в блоке АЛУ локальный сигнал func 4 – х разрядный, десятичные числа с 0 по 8 кодируют логико-арифметические операции процессора, такие как ADD A,B; SUB A,B; AND A,B; OR A,B; XOR A,B и DEC и команды пересылки, такие как MOV A,B; MOV B,A; XCHG A,B. В блоках PC_Inc, PC и Instruction_Reg сигнал func 2 – х разрядный, а в блоках RegisterA и RegisterB 3 – х разрядный.

Пример 1 показывает М-файл функции управляющего автомата микропроцессора в системе Matlab/Simulink (блок CPU_Controller). Управляющий автомат может принимать 6 состояний. Состояния кодируются сигналом CPU_state в формате uint8 (целое десятичное число без знака с размером слова 8 бит). По сигналу master_rst (логическая 1), происходит установка автомата в нулевое состояние CPU_state = uint8(0). Далее происходит настройка блоков процессора с помощью локальных управляющих сигналов func.

```
PC_inc_func = fi(0, 0, 2, 0);
IR_func = fi(3, 0, 2, 0);
PC_func = fi(3, 0, 2, 0);
IM_read = fi(0, 0, 1, 0);
addr_inc = fi(0, 0, 8, 0);
Reg_OutA = fi(0, 0, 8, 0);
Reg_OutB = fi(0, 0, 8, 0);
RegA_func = fi(4, 0, 3, 0);
RegB_func = fi(4, 0, 3, 0);
ALU_func = fi(9, 0, 4, 0);
```

Управляющий автомат формирует на выходе PC_inc_func десятичный ноль, по которому внутреннее содержимое блока РСН будет сброшено (распознается блоком как сигнал сброса), на выходах PC_func и IR_func формируется десятичное число 3, по которому текущее содержимое счетчика команд и регистра инструкций остается неизменным.

На выходах RegA_func и RegB_func формируется десятичное число 4, по которому текущее содержимое регистров общего назначения РОН А и В также остается неизменным. На выходе ALU_func формируется десятичное

число 9, по которому в блоке АЛУ произойдет обход логико-арифметических операций и команд пересылки, а значение сигналов на входах inA и inB будут переданы на выход outA и outB без изменений.

```
outA = fi(inA, 0, 8, 0);
outB = fi(inB, 0, 8, 0);
```

На выходах IM_read, addr_inc, Reg_OutA, Reg_OutB автомат формирует десятичные нули. Ноль на выходе IM_read запрещает чтение из ПЗУ программ. А десятичные нули на выходах addr_inc, Reg_OutA и Reg_OutB означают обнуление этих выходов.

В нулевом состоянии (case 0) осуществляется загрузка в РОН (блоки RegisterA, RegisterB), в РСН (блок PC_Inc) и в счетчик команд (блок PC) нуля (десятичный нуль преобразуется в формат с фиксированной запятой с размером слова 8 бит), а в регистр инструкций (Instruction_Reg) также загружается десятичный ноль, но он преобразуется в формат с фиксированной запятой с размером слова 16 бит. Эти операции осуществляются с помощью локальных сигналов управления PC_inc_func, PC_func, IR_func, RegA_func, RegB_func:

```
case 0,
    PC_inc_func = fi(0, 0, 2, 0);
    PC_func = fi(0, 0, 2, 0);
    IR_func = fi(0, 0, 2, 0);
    RegA_func = fi(0, 0, 3, 0);
    RegB_func = fi(0, 0, 3, 0);
    CPU_state = uint8(1);
```

Следующим состоянием автомата будет CPU_state = uint8(1). В этом состоянии и в двух последующих состояниях uint8(2) и uint8(3) происходит выделение полей команды. В состоянии 1 управляющий автомат формирует сигал разрешения чтения команды из памяти IM_read = fi(1, 0, 1, 0). Поскольку порядковые номера строк в памяти программ начинаются с 1, например, data(1) = fi(1036, 0, 16, 0), то счетчик

команд предварительно должен быть обнулен, т.е. нулевое значение счетчика указывает на строку в ПЗУ с порядковым номером 1.

Для того чтобы счетчик команд содержал адрес следующей команды, управляющий автомат должен сформировать локальный сигнал управления счетчиком PC_func = fi(2, 0, 2, 0), т.е. на выходе PC_func должно присутствовать десятичное число 2, по которому текущее значение счетчика увеличится на 1. Поэтому эта строка стоит второй в операторе case 1. Извлеченную команду (в первоначальный момент и в последующие, в регистре инструкций сохраняются текущие команды, а не следующие, загруженные в счетчик по команде PC_func = fi(2, 0, 2, 0)) из памяти программ в этом состоянии необходимо сохранить в регистре инструкций (16-битный регистр). Поэтому автомат сформирует локальный сигнал управления IR_func = fi(1, 0, 2, 0), разрешающий запись команды в регистр. Следующим состоянием, которое примет автомат, будет состояние CPU_state = uint8(2).

```
case 1,
    % Read from IM (ROM)
    IM_read = fi(1, 0, 1, 0);
    % PC increment PC+1
    PC_func = fi(2, 0, 2, 0);
    % store into IR
    IR_func = fi(1, 0, 2, 0);
    CPU_state = uint8(2);
```

Рассмотрим состояние 3 (case 3) управляющего автомата проектируемого процессора. Для того, чтобы понять, как работает формат с фиксированной запятой, необходимо последовательно копировать ниже приведенные строки фрагмента М-файла и вставлять их в командную строку системы Matlab.

Например, рассмотрим, как обрабатывается команда 1536 (RET). Из регистра инструкций целое беззнаковое

десятичное число 1536 (размер слова 16 бит) поступает на вход IR_in управляющего автомата CPU Controller и присваивается переменной main_opcode, которая представляет 16 – ти битную инструкцию. Из этой инструкции выделяется переменная major_opcode путем сдвига 16 – ти битного вектора вправо на 8 позиций, с размером слова в 4 бита, таким образом мы выделяем, биты с 9 по 12 из 16 – ти разрядной инструкции. В рассматриваемой системе команд разряды с 13 по 16 нулевые, поэтому выделение переменной minor_opcode путем побитного И переменной major_opcode (4 разряда) и маски (переменная mask4, 4 разряда) в принципе не обязательно, но не обходимо в случае последующей модификации системы команд процессора. Для выделения операнда (переменная address_data) из инструкции потребуется маска в 16 разрядов. Побитное И с переменной IR_in и с маской mask8 (0000000011111111) позволяет выделить переменную address_data с размером слова 8 бит. Для команды 1536 переменная address_data это 8 нулей. Следующим состоянием которое примет автомат будет состояние CPU_state = uint8(4).

```
IR_in=fi(1536,0,16,0);
main_opcode = fi(IR_in, 0, 16, 0);
disp(bin(main_opcode))
%0000011000000000
% Сдвиг вектора в право на 8 позиций
major_opcode= fi(bitsrl(main_opcode, 8), 0, 4, 0);
disp(bin(major_opcode))
%0110
mask4 = fi(15, 0, 4, 0);
disp(bin(mask4))
%1111
% Выделение команды, ширина поля 4 бита
minor_opcode = fi(bitand(major_opcode, mask4), 0, 4, 0);
disp(bin(minor_opcode))
%0110
% Выделение из команды операнда
mask8 = fi(255, 0, 16, 0);
```

```
disp(bin(mask8))
% 0000000011111111
address_data = fi(bitand(main_opcode, mask8), 0, 8, 0);
```

В состоянии 4 (case 4) происходит декодирование и выполнение инструкции (case 4). Декодирование происходит по сигналу minor_opcode (фактически 9, 10 и 11 биты сигнала IR_in, 12 бит не используется, т.к. он нулевой). Далее, декодируются 6 команд: NOP, JMP, JMPZ, CAL, MOV A,XX, MOV B,XX. Рассмотрим команду JMP. Выделенный операнд address_data из инструкции содержит адрес команды в ПЗУ на который необходимо перейти. Операнд присваивается переменной addr_inc. Автомат формирует локальные сигналы управления PCH - PC_inc_func (десятичное число 1) и счетчика команд - PC_func (десятичное число 1). Далее выделенный операнд (содержит адрес команды на который необходимо перейти) будет загружен в PCH и в счетчик команд. При загрузки операнда в PCH, содержимое счетчика команд при этом сохраняется во внутренней переменной PC_Temp данного регистра (пример 2).

```
   case 4,
            switch(uint8(minor_opcode))
               case 0,
                  % NOP
                 CPU_state = uint8(1);
               case 1,
                 % JMP
                  temp_addr_data = fi(address_data, 0, 8, 0);
                  addr_inc = fi(temp_addr_data, 0, 8, 0);
                  PC_inc_func = fi(1, 0, 2, 0);
                  PC_func = fi(1, 0, 2, 0);
                  CPU_state = uint8(1);
……….……….……
……….……
               case 6,
                    switch(uint8(address_data))
                       case 0,
```

```
%RET
 PC_inc_func = fi(2, 0, 2, 0);
 PC_func = fi(2, 0, 2, 0);
 CPU_state = uint8(5);
```

Если ни одна их этих команд не выполняется, то далее дешифрируются и отрабатываются команда RET, логико-арифметические команды (ADD A,B, OR A,B, XOR A,B, DEC A) и команды пересылки (MOV A,B, MOV B,A, XCHG A,B,). Последним состоянием является состояние case 5. В этом состоянии обновляются регистры РОН A и B, затем будет осущетвлен переход в состояние 1. И весь описанный выше процесс обработки команды повторится вновь и то тех пор, пока не будет отработана последняя команда в программе.

```
function [ALU_func, IR_func, PC_inc_func, PC_func, ...
   addr_inc, IM_read, RegA_func, RegB_func, ...
   Reg_OutA, Reg_OutB] = CPU_Controller(master_rst, IR_in, Reg_A)

% CPU Controller
% 16-bit Instruction Encoding:
% ------------minor_opcode---------
%   NOP:            00000 000 <00000000>
%   JMP:            00000 001 <8-bit>
%   JMPZ:           00000 010 <8-bit>
%   CALL:           00000 011 <8-bit>
%   MOV A,xx:   00000 100 <8-bit>
%   MOV B,xx:   00000 101 <8-bit>
% ---------------------------------
%   RET:            00000 110 <00000000>
%   MOV A,B:    00000 110 <00000001>
%   MOV B,A:    00000 110 <00000010>
%   XCHG A,B:  00000 110 <00000011>
%   ADD A,B:    00000 110 <00000100>
%   SUB A,B:    00000 110 <00000101>
%   AND A,B:    00000 110 <00000110>
%   OR A,B:      00000 110 <00000111>
%   XOR A,B:    00000 110 <00001000>
%   DEC A:       00000 110 <00001001>
```

```
persistent CPU_state;
if(isempty(CPU_state))
   CPU_state = uint8(0);
end

if(master_rst)
   CPU_state = uint8(0);
end

PC_inc_func = fi(0, 0, 2, 0);
IR_func = fi(3, 0, 2, 0); % NOP
PC_func = fi(3, 0, 2, 0); % NOP
IM_read = fi(0, 0, 1, 0);
addr_inc = fi(0, 0, 8, 0);
Reg_OutA = fi(0, 0, 8, 0);
Reg_OutB = fi(0, 0, 8, 0);
RegA_func = fi(4, 0, 3, 0); % NOP
RegB_func = fi(4, 0, 3, 0); % NOP
ALU_func = fi(9, 0, 4, 0); % NOP

% main_code: <16..1>
% major_opcode: <16..9>
% minor_opcode: <12..9>
% address_data: <8..1>

persistent main_opcode;
persistent major_opcode;
persistent minor_opcode;
persistent address_data;

if(isempty(major_opcode))
   main_opcode = fi(0, 0, 16, 0);
   major_opcode = fi(0, 0, 4, 0);
   minor_opcode = fi(0, 0, 4, 0);
   address_data = fi(0, 0, 8, 0);
end
switch(CPU_state)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   %  RESETTING OUTPUTS
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   case 0,
      PC_inc_func = fi(0, 0, 2, 0);
      PC_func = fi(0, 0, 2, 0);
      IR_func = fi(0, 0, 2, 0);
      RegA_func = fi(0, 0, 3, 0);
      RegB_func = fi(0, 0, 3, 0);
      CPU_state = uint8(1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   %  FETCH
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   case 1,
       % Read from IM (ROM)
      IM_read = fi(1, 0, 1, 0);
       % PC increment PC+1
      PC_func = fi(2, 0, 2, 0);
      % store into IR
      IR_func = fi(1, 0, 2, 0);
      CPU_state = uint8(2);
   case 2,
      % Read from IR
      IR_func = fi(2, 0, 2, 0);
      % Accommodating for the 'unit delay' from IR_out to IR_in
      CPU_state = uint8(3);
   case 3,
      % IR_in <16..1>
      main_opcode = fi(IR_in, 0, 16, 0);
      % IR_in <16..9>
      major_opcode = fi(bitsrl(main_opcode, 8), 0, 4, 0);
      % for instructions NOP,JMP,JMPZ,CALL,MOV A,xx MOV
B,xx,RET
      % IR_in <12..9>
      mask4 = fi(15, 0, 4, 0);
      minor_opcode = fi(bitand(major_opcode, mask4), 0, 4, 0);
      % IR_in <8..1>
      mask8 = fi(255, 0, 16, 0);
      address_data = fi(bitand(main_opcode, mask8), 0, 8, 0);
      % Go to the decode stage
      CPU_state = uint8(4);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
   % DECODE AND EXECUTE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   case 4,
         switch(uint8(minor_opcode))
            case 0,
               % NOP
             CPU_state = uint8(1);
            case 1,
              % JMP
               temp_addr_data = fi(address_data, 0, 8, 0);
               addr_inc = fi(temp_addr_data, 0, 8, 0);
               PC_inc_func = fi(1, 0, 2, 0);
               PC_func = fi(1, 0, 2, 0);
               CPU_state = uint8(1);
            case 2,
               %JMPZ
               temp_addr_data = fi(address_data, 0, 8, 0);
               if fi(Reg_A,0,8,0) == fi(0,0,8,0)
               addr_inc = fi(temp_addr_data, 0, 8, 0);
               PC_inc_func = fi(1, 0, 2, 0);
               PC_func = fi(1, 0, 2, 0);
               end
               CPU_state = uint8(1);
            case 3,
             % CALL
               temp_addr_data = fi(address_data, 0, 8, 0);
               addr_inc = fi(temp_addr_data, 0, 8, 0);
               PC_inc_func = fi(1, 0, 2, 0);
               PC_func = fi(1, 0, 2, 0);
               CPU_state = uint8(1);
            case 4,
             %MOV A,xx
               temp_addr_data = fi(address_data, 0, 8, 0);
               Reg_OutA = fi(temp_addr_data , 0, 8, 0);
               RegA_func = fi(1, 0, 3, 0);
               CPU_state = uint8(1);
            case 5,
              %MOV B,xx
              temp_addr_data = fi(address_data, 0, 8, 0);
               Reg_OutB = fi(temp_addr_data , 0, 8, 0);
```

```
    RegB_func = fi(1, 0, 3, 0);
    CPU_state = uint8(1);
case 6,
     switch(uint8(address_data))
        case 0,
            %RET
             PC_inc_func = fi(2, 0, 2, 0);
             PC_func = fi(2, 0, 2, 0);
             CPU_state = uint8(5);
        case 1,
            %MOV A,B
            ALU_func = fi(0, 0, 4, 0);
            RegA_func = fi(2, 0, 3, 0);
            RegB_func = fi(2, 0, 3, 0);
            CPU_state = uint8(5);
        case 2,
             %MOV B,A
            ALU_func = fi(1, 0, 4, 0);
            RegA_func = fi(2, 0, 3, 0);
            RegB_func = fi(2, 0, 3, 0);
            CPU_state = uint8(5);
        case 3,
             %XCHG A,B
            ALU_func = fi(2, 0, 4, 0);
            RegA_func = fi(2, 0, 3, 0);
            RegB_func = fi(2, 0, 3, 0);
            CPU_state = uint8(5);
        case 4,
             %ADD A,B
            ALU_func = fi(3, 0, 4, 0);
            RegA_func = fi(2, 0, 3, 0);
            RegB_func = fi(2, 0, 3, 0);
            CPU_state = uint8(5);
        case 5,
           %SUB A,B
            ALU_func = fi(4, 0, 4, 0);
            RegA_func = fi(2, 0, 3, 0);
            RegB_func = fi(2, 0, 3, 0);
            CPU_state = uint8(5);
        case 6,
```

```
                    %AND A,B
                      ALU_func = fi(5, 0, 4, 0);
                      RegA_func = fi(2, 0, 3, 0);
                      RegB_func = fi(2, 0, 3, 0);
                      CPU_state = uint8(5);
                  case 7,
                    %OR A,B
                      ALU_func = fi(6, 0, 4, 0);
                      RegA_func = fi(2, 0, 3, 0);
                      RegB_func = fi(2, 0, 3, 0);
                      CPU_state = uint8(5);
                  case 8,
                    %XOR A,B
                      ALU_func = fi(7, 0, 4, 0);
                      RegA_func = fi(2, 0, 3, 0);
                      RegB_func = fi(2, 0, 3, 0);
                      CPU_state = uint8(5);
                  case 9,
                    %DEC A
                      ALU_func = fi(8, 0, 4, 0);
                      RegA_func = fi(2, 0, 3, 0);
                      CPU_state = uint8(5);
                end
            end
        case 5,
            RegA_func = fi(2, 0, 3, 0);
            RegB_func = fi(2, 0, 3, 0);
            CPU_state = uint8(1);
end
```

Пример 1. М-файл функции управляющего автомата микропроцессора (CPU_Controller) в системе Matlab/Simulink

```
function [PC_next,Temp] = PC_Inc(func, addr, PC_curr)

% func = 0 => reset PC_Inc
% func = 1 => store into PC_Inc when JMP, JMPZ, CALL
% func = 2 => load from PC_Inc when RET

persistent PC_Temp;
if(isempty(PC_Temp))
```

```
  PC_Temp = fi(0, 0, 8, 0);
end
PC_next = fi(PC_curr, 0, 8, 0);
Temp = fi(0, 0, 8, 0);
switch(uint8(func))
   case 0,
      % reset PC_Inc
      PC_next = fi(0, 0, 8, 0);
   case 1,
      % store into PC_Inc when JMP, JMPZ, CALL
        PC_next = fi(addr, 0, 8, 0);
        PC_Temp = fi(PC_curr, 0, 8, 0);
        Temp = fi(PC_Temp, 0, 8, 0);
   case 2,
      % load from PC_Inc when RET
      PC_next = fi(PC_Temp, 0, 8, 0);
end
```

Пример 2. М-файл функции блока специального назначения (PC_Inc) в системе Matlab/Simulink

```
function addr_out = PC(func, addr_in)

% Program Counter
% func = 0 => reset PC
% func = 1 => load PC
% func = 2 => increment PC

persistent PC_value;
if(isempty(PC_value))
   PC_value = fi(0, 0, 8, 0);
end
addr_out = fi(PC_value, 0, 8, 0);
switch(uint8(func))
   case 0,
      % reset
      PC_value = fi(0, 0, 8, 0);
   case 1,
      % store into PC
      PC_value = fi(addr_in, 0, 8, 0);
```

```
   case 2,
      % increment PC
      PC_value = fi(PC_value + 1, 0, 8, 0);
end
```

Пример 3. М-файл функции блока счетчика команд (PC) в системе Matlab/Simulink

```
function instr_out = Memory(addr,read)
persistent data;
if(isempty(data))
data = fi(zeros(1, 256), 0, 16, 0);
end
data(1) = fi(1036, 0, 16, 0);
data(2) = fi(1303, 0, 16, 0);
data(3) = fi(1540, 0, 16, 0);
data(4) = fi(1545, 0, 16, 0);
data(5) = fi(1358, 0, 16, 0);
data(6) = fi(1539, 0, 16, 0);
data(7) = fi(1541, 0, 16, 0);
data(8) = fi(1545, 0, 16, 0);
data(9) = fi(1542, 0, 16, 0);
data(10) = fi(523, 0, 16, 0);
data(11) = fi(263, 0, 16, 0);
data(12) = fi(1037, 0, 16, 0);
data(13) = fi(1397, 0, 16, 0);
data(14) = fi(1543, 0, 16, 0);
data(15) = fi(1539, 0, 16, 0);
data(16) = fi(1544, 0, 16, 0);
data(17) = fi(277, 0, 16, 0);
data(18) = fi(1135, 0, 16, 0);
data(19) = fi(1480, 0, 16, 0);
data(20) = fi(1542, 0, 16, 0);
data(21) = fi(1536, 0, 16, 0);
data(22) = fi(785, 0, 16, 0);
data(23) = fi(0, 0, 16, 0);
if(read == 1)
   instr_out = data(addr+1);
else
   instr_out = fi(0, 0, 16, 0);
end
```

Пример 4. М-файл функции блока памяти программ (Memory) в системе Matlab/Simulink

```
function IR_out = Instruction_Reg(func, IR_in)
% A 16-bit Instruction Register with the following func:
% func == 0 => reset
% func == 1 => store into IR
% func == 2 => read from IR
% otherwise, preserve old value and return 0
persistent IR_value;
if(isempty(IR_value))
   IR_value = fi(0, 0, 16, 0);
end
IR_out = fi(0, 0, 16, 0);
switch(uint8(func))
   case 0,
      % reset
      IR_value = fi(0, 0, 16, 0);
   case 1,
      % store into IR
      IR_value = fi(IR_in, 0, 16, 0);
   case 2,
      % read IR
      IR_out = fi(IR_value, 0, 16, 0);
end
```

Пример 5. М-файл функции блока регистра инструкций (Instruction_Reg) в системе Matlab/Simulink

```
function Reg_out_A = RegisterA(func, Reg_in_A_1, Reg_in_A_2)
% func == 0 => reset;
% func == 1 => store into RegisterA from port 1;
% func == 2 => store into RegisterA from port 2;
% func == 3 => read from RegisterA;
% HDL specific fimath
hdl_fm = fimath(...
'RoundMode', 'floor',...
'OverflowMode', 'wrap',...
'ProductMode', 'FullPrecision', 'ProductWordLength', 32,...
'SumMode', 'FullPrecision', 'SumWordLength', 32,...
'CastBeforeSum', true);
```

```
persistent Reg_value;
if(isempty(Reg_value))
   Reg_value = fi(0, 0, 8, 0, hdl_fm);
end
Reg_out_A = fi(Reg_value, 0, 8, 0, hdl_fm);
switch(uint8(func))
   case 0,
      % reset
      Reg_value = fi(0, 0, 8, 0, hdl_fm );
   case 1,
      % store into Reg_A from port 1
      Reg_value = Reg_in_A_1;
   case 2,
      % store into Reg_A from port 2
      Reg_value = Reg_in_A_2;
   case 3,
      % read Reg_A
      Reg_out_A = Reg_value;
end
```

Пример 6. М-файл функции блока регистра общего назначения A (RegisterA) в системе Matlab/Simulink

```
function [outA, outB] = alu(func,inA,inB)
% This 8-bit ALU supports the following operations:
% MOV, XCHG, ADD, SUB, AND, OR, XOR, DEC
% func = 0 => MOV A,B
% func = 1 => MOV B,A
% func = 2 => XCHG A,B
% func = 3 => ADD A,B
% func = 4 => SUB A,B
% func = 5 => AND A,B
% func = 6 => OR A,B
% func = 7 => XOR A,B
% func = 8 => DEC A
% Simply pass the inA, when there is no designated func
outA = fi(inA, 0, 8, 0);
% Simply pass the inB, when there is no designated func
outB = fi(inB, 0, 8, 0);
switch (uint8(func))
            case 0, %MOV A,B
                outA = fi(inB, 0, 8, 0);
```

```
        case 1, %MOV B,A
            outB = fi(inA, 0, 8, 0);
        case 2, %XCHG A,B
            X_temp = fi(inB, 0, 8, 0);
            outB = fi(inA, 0, 8, 0);
            outA = fi(X_temp, 0, 8, 0);
        case 3, %ADD A,B
            outA = fi(inA + inB, 0, 8, 0);
        case 4, %SUB A,B
            outA = fi(inA - inB, 0, 8, 0);
        case 5, %AND A,B
            outA = fi(bitand(inA,inB), 0, 8, 0);
        case 6, %OR A,B
            outA = fi(bitor(inA,inB), 0, 8, 0);
        case 7, %XOR A,B
            outA = fi(bitxor(inA,inB), 0, 8, 0);
        case 8, %DEC A
            outA = fi(inA - 1, 0, 8, 0);
end
```

Пример 7. М-файл функции блока АЛУ в системе Matlab/Simulink

На рис.4.31 показаны временные диаграммы работы процессора с управляющим автоматом в системе Matlab/Simulink. По оси у откладываются целые беззнаковые десятичные числа (которые преобразуются в процессе вычислений в формат с фиксированной запятой), а по оси x время моделирования.

На рис.4.31, *а*, видно, что значения, накопленные счетчиком команд, непрерывно увеличиваются и только в случае команды JMP 7 (команда выполняется в программе 3 раза), счетчик изменяет свое содержимое на значение операнда, содержащееся в команде, т.е. на 7. На рис.4.31, *б* показано содержимое блока РСН, на рис.4.31, *в* – содержимое памяти программ а на рис.4.31, *г* – содержимое РОН А.

В системе Matlab/Simulink разработан учебный вариант 8-ми разрядного процессора, позволяющего проводить

вычисления в формате с фиксированной запятой, с управляющим автоматом на шесть состояний. Преимуществом такой архитектуры является ее адаптивность к последующим модификациям, например, в случае если потребуется добавить дополнительные команды. Недостатком является отсутствие памяти данных, поддержка незначительного числа команд, а также то, что процессор оперирует только с целыми положительными числами и отсутствие конвейера команд.

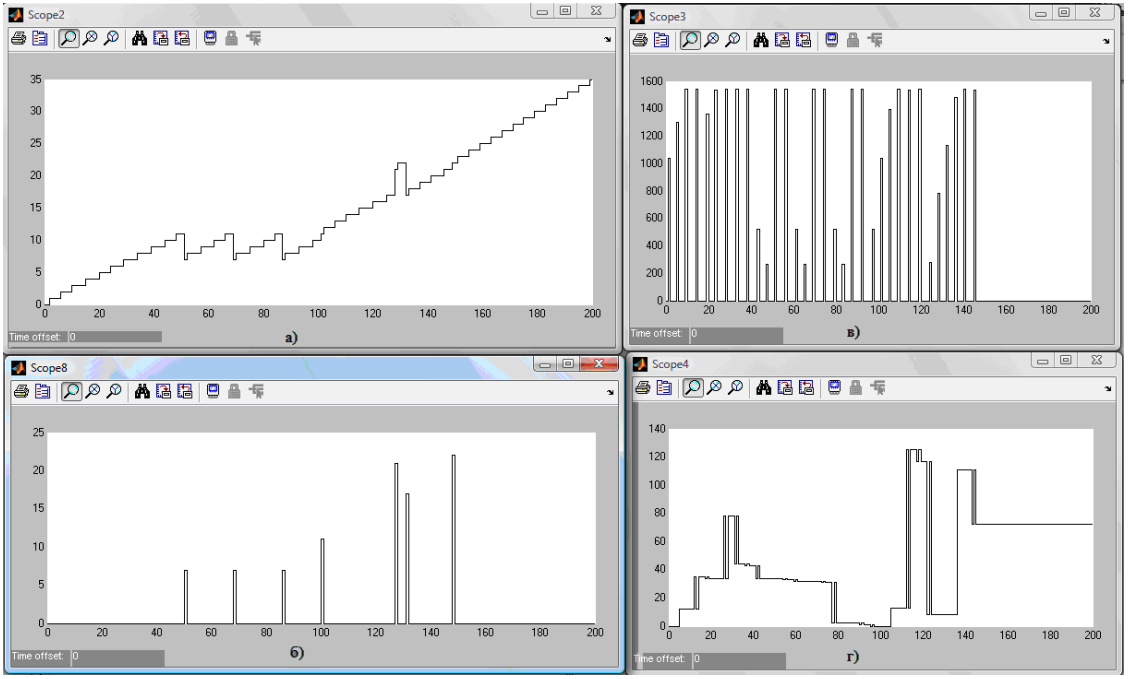


Рис.4.31. Временные диаграммы работы процессора с управляющим автоматом в системе Matlab/Simulink: а – счетчик команд; б – блок специального назначения; в – память программ; г – РОН А

4.5. Проектирование учебного процессора с фиксированной запятой в САПР ПЛИС Quartus II

В данном разделе предлагается на основе системы команд процессора с циклом работы в два такта и на основе модели процессора с управляющим автоматом на шесть состояний, позволяющим проводить вычисления с фиксированной зяпятой, реализованной в системе Matlab/Simulink (раздел 4.4), разработать процессор в базисе ПЛИС Stratix III компании Altera с использованием САПР Quartus II. Особенностью выше рассмотренной модели процессора является распределенная система управления функциональными блоками, т.е. каждый блок имеет свой локальный управляющий сигнал (шину), которым управляет цифровой автомат. Основные функциональные блоки проектируемого процессора описаны на языке VHDL, код языка которого был сгенерирован в автоматическом режиме с помощью Simulink HDL Coder системы Matlab/Simulink. На рис.4.32 представлена электрическая схема процессора в САПР ПЛИС Quartus II версии 8.1.

Проектируемый процессор состоит из следующих функциональных блоков (рис.4.32): управляющий автомат (блок CPU_Controller, пример 1); регистр специального назначения (РСН, блок PC_Incrementer, пример 2), необходим для обеспечения “прыжковых” команд, таких как JMP, JMPZ, CALL и RET; счетчик команд (блок Program_Counter, пример 3); память программ - ПЗУ процессора (блок Instruction_ROM, пример 4); регистр инструкций (блок Instruction_Register, пример 5); два регистра общего назначения (РОН, блок RegisterA, пример 6); АЛУ процессора (блок ALU, пример 7). Регистры на D-триггерах (четыре 8-ми (блок dff8) и один 16-ти разрядный (блок dff16)), тактируемые фронтом синхросигнала разработаны дополнительно (см. раздел 4.4) с использованием мегафункции LPM_FF.

На рис.4.33 и рис.4.34 показаны временные диаграммы работы процессора с управляющим автоматом в САПР Quartus II. Осуществляется тестирование команд MOV A,12; MOV B,23; ADD A,B и команд JMPZ11, JMP7.

Язык VHDL является языком со строгим контролем типов. Поэтому бывает необходимо преобразовать сигнал одного типа в сигнал другого типа. Даже при выполнении простых действий, если типы объектов не совпадают, может потребоваться обращение к функции преобразования типов. Различают два вида преобразования типа: переход типа и вызов функции преобразования типа. Переход типа применяется для преобразования тесно связанных типов или подтипов. Если типы не тесно связанные, то необходимо выполнить вызов функции преобразования типа.

Пакет numeric_std содержит стандартный набор арифметических, логических функций и функций сравнения для работы с типами signed, unsigned, integer, std_ulogic, std_logic, std_logic_vector. В пакете numeric_std существует функция преобразования векторного типа to_unsigned с операндами arg (тип integer), size (тип natural, битовая ширина) и типом результата unsigned. Тип unsigned интерпретируется как двоичное представление числа без знака, а тип signed обозначает двоичные числа со знаком в дополнительном коде. Например: CPU_state_temp := to_unsigned(3, 8);. Число 3 из одномерного массива целых чисел преобразуется в восьмиразрядное двоичное число без знака, которое присваивается переменной CPU_state_temp. А переменная CPU_state_temp объявлена как массив двоичных чисел без знака: VARIABLE CPU_state_temp: unsigned(7 DOWNTO 0);. Это есть явное преобразование тесно связанных между собой типов.

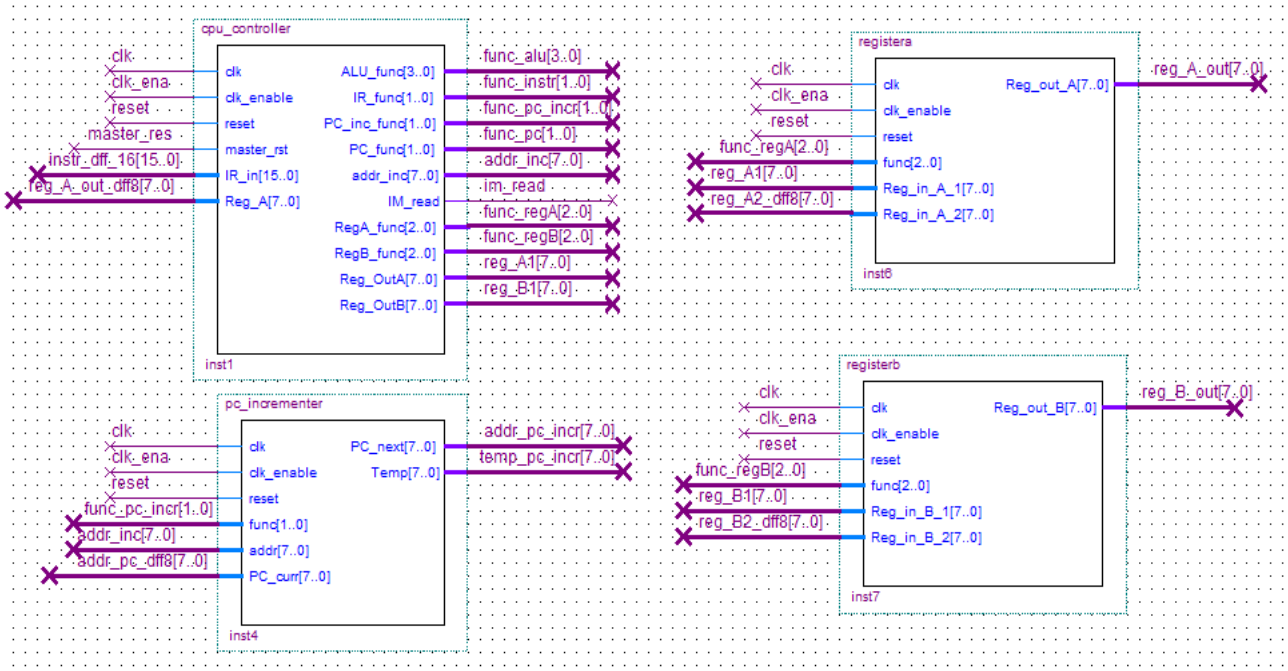


Рис.4.32. Схема процессора с управляющим автоматом для вычислений в формате с фиксированной запятой в САПР Quartus II, построенная с использованием модели, разработанной в системе Matlab/Simulink

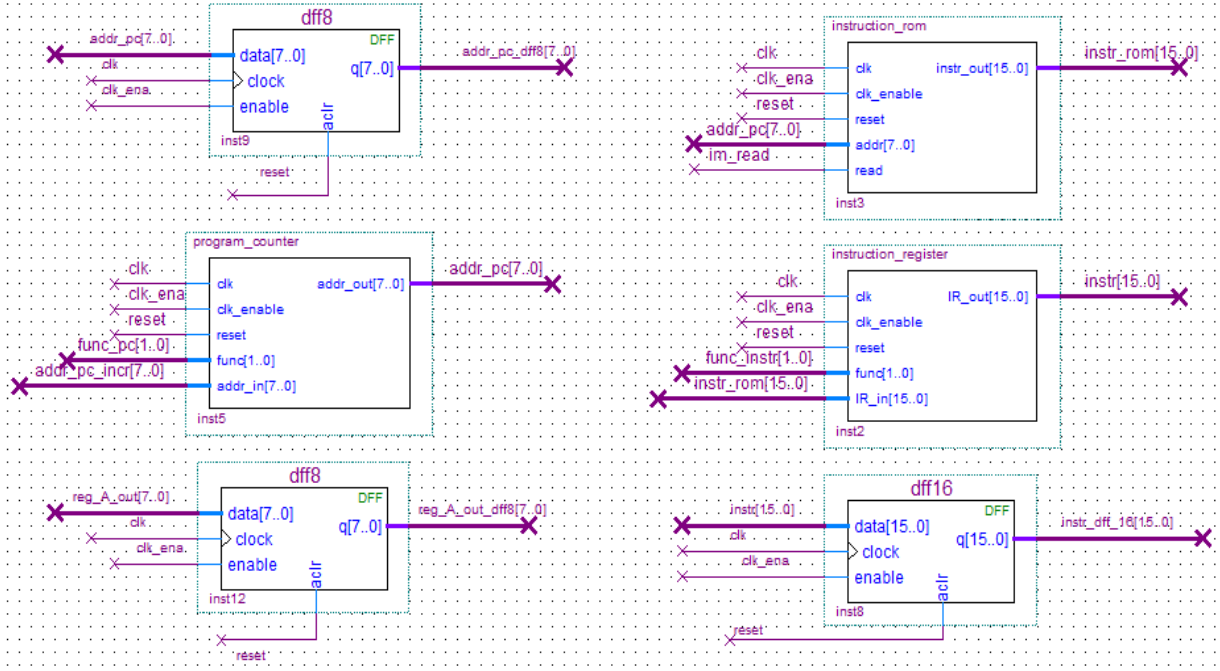


Рис.4.32. Схема процессора с управляющим автоматом для вычислений в формате с фиксированной запятой в САПР Quartus II, построенная с использованием модели, разработанной в системе Matlab/Simulink (продолжение)

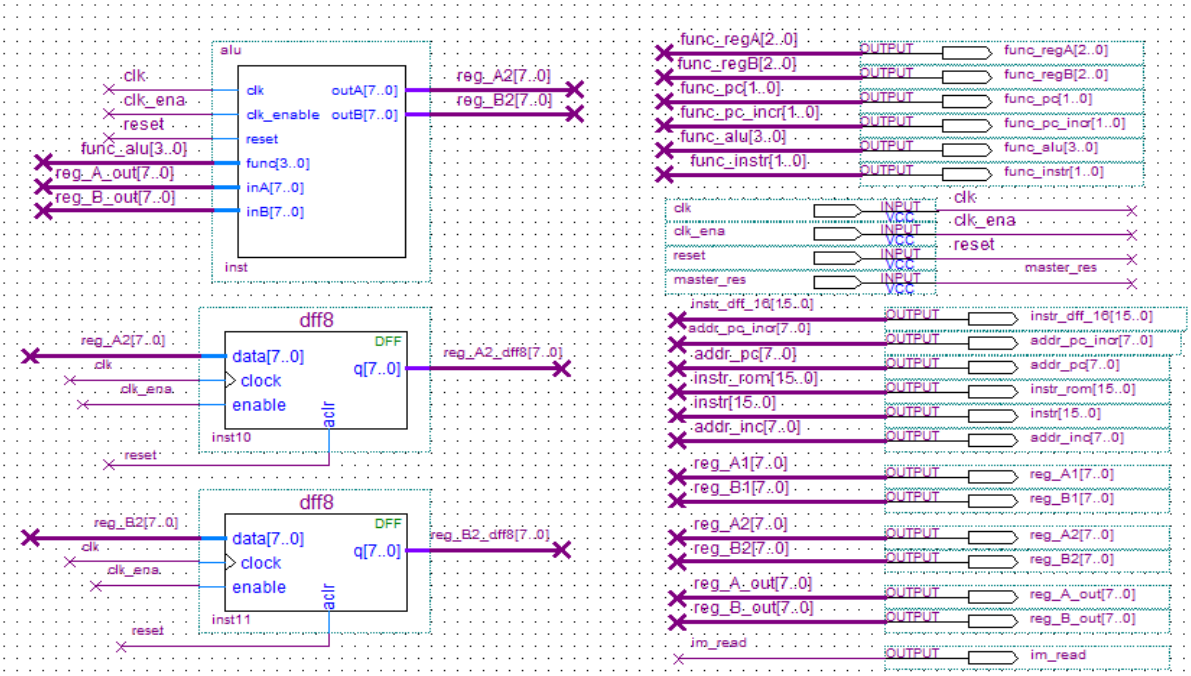


Рис.4.32. Схема процессора с управляющим автоматом для вычислений в формате с фиксированной запятой в САПР Quartus II, построенная с использованием модели, разработанной в системе Matlab/Simulink (окончание)

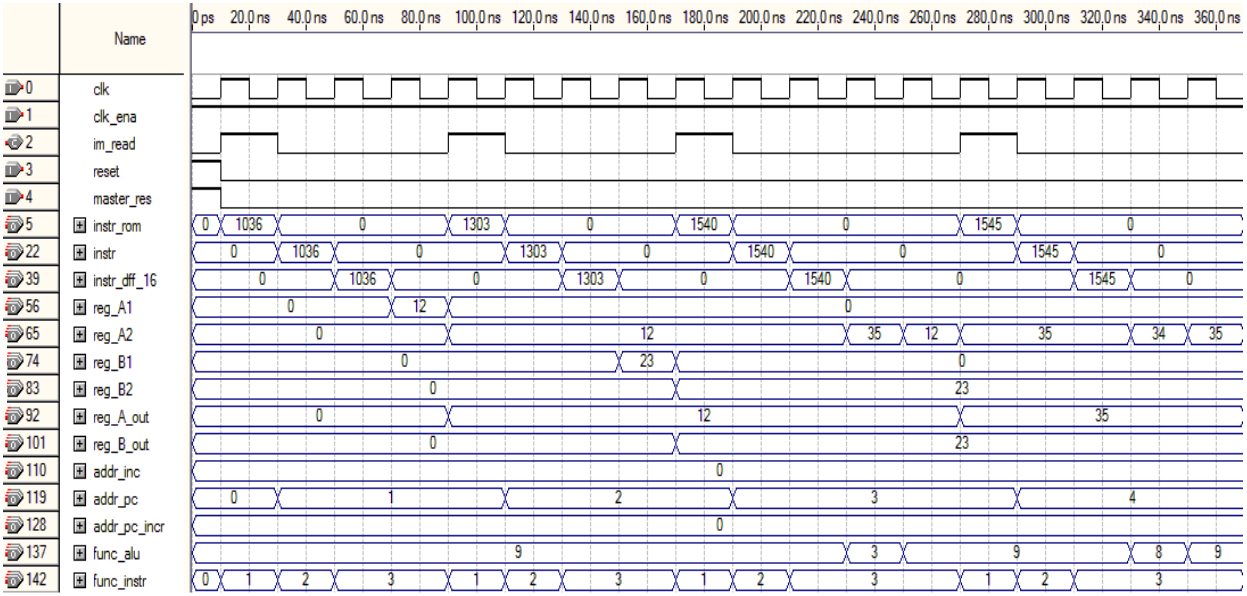


Рис.4.33. Временные диаграммы работы процессора с управляющим автоматом в САПР Quartus II. Тестирование команд MOV A,12; MOV B,23; ADD A,B

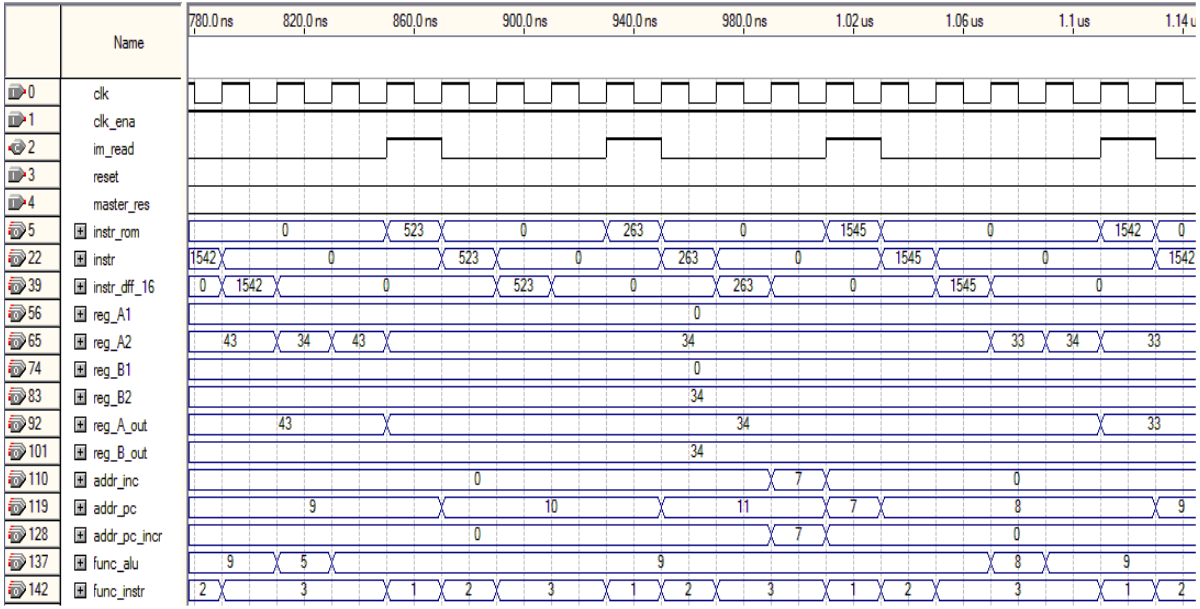


Рис.4.34. Временные диаграммы работы процессора с управляющим автоматом в САПР Quartus II. Тестирование команд JMPZ11, JMP7

Или IR_func <= std_logic_vector(to_unsigned(3, 2));. Десятичное число 3 преобразуется в двоичное число “11” типа unsigned, затем тип unsigned неявно преобразуется в тип std_logic_vector. Сигналу IR_func будет назначено двоичное число “11”.

При генерации кода языка VHDL блока АЛУ используется дополнительная функция tmw_to_signed, которая преобразует двоичное число типа unsigned в двоичное число типа signed (пример 7) с шириной битовой шины типа integer, что необходимо для обеспечения операции вычитания (вызов функции преобразования типа):

```
FUNCTION tmw_to_signed(arg: unsigned; width: integer) RETURN
signed IS
--SUB A,B
ina_1 := tmw_to_signed(unsigned(inA), 9) - tmw_to_signed(unsigned(inB), 9);
```

Оператор srl, введенный в стандарте VHDL93, осуществляет операцию логического сдвига одномерного массива (левый оператор) с элементами типа bit в право, на число указанное правым оператором типа integer. Например, одномерный массив main_opcode_temp типа unsigned(15 DOWNTO 0), представляющий из себя 16-ти битовое двоичное число, сдвигается вправо на 8 бит. Например:

```
main_opcode_temp := unsigned(IR_in); cr := main_opcode_temp srl 8;.
```

Логический оператор AND (тип левого операнда, правого и тип результата unsigned) используется для выделения 8-ми разрядного сигнала (операнда) address_data_next из 16-ти разрядной команды микропроцессора путем логического умножения сигнала major_opcode_temp с маской to_unsigned(255, 16). Например:

```
-- IR_in <8..1>
        c_uint := main_opcode_temp AND to_unsigned(255, 16);
```

```
    IF c_uint(15 DOWNTO 8) /= "00000000" THEN
       address_data_next <= "11111111";
    ELSE
       address_data_next <= c_uint(7 DOWNTO 0);
    END IF;
```

В примере 3 используется оператор конкатенации & который предопределен для всех одноразмерных массивов. Этот оператор выстраивает массивы путем комбинирования с их операндами. Оператор & используется для добавления одиночного элемента в конец массива PC_value типа unsigned(7 DOWNTO 0):

```
     -- increment PC
    ain := resize(PC_value & '0' & '0' & '0' & '0' & '0' & '0' & '0', 16);
    ain_0 := ain + 128;
IF (ain_0(15) /= '0') OR (ain_0(14 DOWNTO 7) = "11111111") THEN
       PC_value_next <= "11111111";
    ELSE
    PC_value_next <= ain_0(14 DOWNTO 7) + ("0" & (ain_0(6)));
    END IF;
```

Функция изменения размера resize (тип левого оператора unsigned, колличество позиций типа natural, тип результата unsigned) позволяет из восьми разрядного сигнала PC_value сконструировать локальную переменную ain типа unsigned(15 DOWNTO 0). Функция ‘+’ позволяет осуществить ариметическую операцию сложения, если тип левого оператора unsigned, а правого - integer с результатом unsigned.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY CPU_Controller IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
```

```
      master_rst : IN std_logic;
      IR_in : IN std_logic_vector(15 DOWNTO 0);
      Reg_A : IN std_logic_vector(7 DOWNTO 0);
      ALU_func : OUT std_logic_vector(3 DOWNTO 0);
      IR_func : OUT std_logic_vector(1 DOWNTO 0);
      PC_inc_func : OUT std_logic_vector(1 DOWNTO 0);
      PC_func : OUT std_logic_vector(1 DOWNTO 0);
      addr_inc : OUT std_logic_vector(7 DOWNTO 0);
      IM_read : OUT std_logic;
      RegA_func : OUT std_logic_vector(2 DOWNTO 0);
      RegB_func : OUT std_logic_vector(2 DOWNTO 0);
      Reg_OutA : OUT std_logic_vector(7 DOWNTO 0);
      Reg_OutB : OUT std_logic_vector(7 DOWNTO 0));
END CPU_Controller;

ARCHITECTURE fsm_SFHDL OF CPU_Controller IS

   SIGNAL CPU_state : unsigned(7 DOWNTO 0);
   SIGNAL major_opcode : unsigned(3 DOWNTO 0);
   SIGNAL main_opcode : unsigned(15 DOWNTO 0);
   SIGNAL minor_opcode : unsigned(3 DOWNTO 0);
   SIGNAL address_data : unsigned(7 DOWNTO 0);
   SIGNAL CPU_state_next : unsigned(7 DOWNTO 0);
   SIGNAL major_opcode_next : unsigned(3 DOWNTO 0);
   SIGNAL main_opcode_next : unsigned(15 DOWNTO 0);
   SIGNAL minor_opcode_next : unsigned(3 DOWNTO 0);
   SIGNAL address_data_next : unsigned(7 DOWNTO 0);
BEGIN
   initialize_CPU_Controller : PROCESS (reset, clk)
      -- local variables
   BEGIN
      IF reset = '1' THEN
         CPU_state <= to_unsigned(0, 8);
         main_opcode <= to_unsigned(0, 16);
         major_opcode <= to_unsigned(0, 4);
         minor_opcode <= to_unsigned(0, 4);
         address_data <= to_unsigned(0, 8);
      ELSIF clk'EVENT AND clk= '1' THEN
```

```
        IF clk_enable= '1' THEN
          CPU_state <= CPU_state_next;
          major_opcode <= major_opcode_next;
          main_opcode <= main_opcode_next;
          minor_opcode <= minor_opcode_next;
          address_data <= address_data_next;
        END IF;
      END IF;
    END PROCESS initialize_CPU_Controller;

    CPU_Controller : PROCESS (CPU_state, major_opcode,
main_opcode, minor_opcode,
address_data, master_rst, IR_in, Reg_A)
      -- local variables
      VARIABLE c_uint : unsigned(15 DOWNTO 0);
      VARIABLE b_c_uint : unsigned(3 DOWNTO 0);
      VARIABLE cr : unsigned(15 DOWNTO 0);
      VARIABLE CPU_state_temp : unsigned(7 DOWNTO 0);
      VARIABLE major_opcode_temp : unsigned(3 DOWNTO 0);
      VARIABLE main_opcode_temp : unsigned(15 DOWNTO 0);
      VARIABLE reg_a_0 : unsigned(7 DOWNTO 0);
    BEGIN
      minor_opcode_next <= minor_opcode;
      address_data_next <= address_data;
      CPU_state_temp := CPU_state;
      major_opcode_temp := major_opcode;
      main_opcode_temp := main_opcode;
      -- CPU Controller
      -- 16-bit Instruction Encoding:
      -- ------------minor_opcode---------
      --  NOP:          00000 000 <00000000>
      --  JMP:          00000 001 <8-bit>
      --  JMPZ:         00000 010 <8-bit>
      --  CALL:         00000 011 <8-bit>
      --  MOV A,xx:  00000 100 <8-bit>
      --  MOV B,xx:  00000 101 <8-bit>
      --  RET:          00000 110 <00000000>
      -- ---------------------------------------
```

```
--  MOV A,B:    00000 110 <00000001>
--  MOV B,A:    00000 110 <00000010>
--  XCHG A,B:  00000 110 <00000011>
--  ADD A,B:     00000 110 <00000100>
--  SUB A,B:      00000 110 <00000101>
--  AND A,B:     00000 110 <00000110>
--  OR A,B:        00000 110 <00000111>
--  XOR A,B:     00000 110 <00001000>
--  DEC A:         00000 110 <00001001>
IF master_rst /= '0' THEN
   CPU_state_temp := to_unsigned(0, 8);
END IF;
PC_inc_func <= std_logic_vector(to_unsigned(0, 2));
IR_func <= std_logic_vector(to_unsigned(3, 2));
PC_func <= std_logic_vector(to_unsigned(3, 2));
IM_read <= '0';
addr_inc <= std_logic_vector(to_unsigned(0, 8));
Reg_OutA <= std_logic_vector(to_unsigned(0, 8));
Reg_OutB <= std_logic_vector(to_unsigned(0, 8));
RegA_func <= std_logic_vector(to_unsigned(4, 3));
RegB_func <= std_logic_vector(to_unsigned(4, 3));
ALU_func <= std_logic_vector(to_unsigned(9, 4));
-- NOP
-- main_code: <16..1>
-- major_opcode: <16..9>
-- minor_opcode: <12..9>
-- address_data: <8..1>
CASE CPU_state_temp IS
   WHEN "00000000" =>
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      --  RESETTING OUTPUTS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      PC_inc_func <= std_logic_vector(to_unsigned(0, 2));
      PC_func <= std_logic_vector(to_unsigned(0, 2));
      IR_func <= std_logic_vector(to_unsigned(0, 2));
      RegA_func <= std_logic_vector(to_unsigned(0, 3));
      RegB_func <= std_logic_vector(to_unsigned(0, 3));
      CPU_state_temp := to_unsigned(1, 8);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      --  FETCH
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    WHEN "00000001" =>
      -- Read from IM (ROM)
      IM_read <= '1';
      -- PC increment PC+1
      PC_func <= std_logic_vector(to_unsigned(2, 2));
      -- store into IR
      IR_func <= std_logic_vector(to_unsigned(1, 2));
      CPU_state_temp := to_unsigned(2, 8);
    WHEN "00000010" =>
      -- Read from IR
      IR_func <= std_logic_vector(to_unsigned(2, 2));
      -- Accommodating for the 'unit delay' from IR_out to IR_in
      CPU_state_temp := to_unsigned(3, 8);
    WHEN "00000011" =>
      -- IR_in <16..1>
      main_opcode_temp := unsigned(IR_in);
      -- IR_in <16..9>
      cr := main_opcode_temp srl 8;
      IF cr(15 DOWNTO 4) /= "000000000000" THEN
        major_opcode_temp := "1111";
      ELSE
        major_opcode_temp := cr(3 DOWNTO 0);
      END IF;
 -- for instructions NOP,JMP,JMPZ,CALL,MOV A,xx MOV B,xx,RET
      -- IR_in <12..9>
      b_c_uint := major_opcode_temp AND to_unsigned(15, 4);
      minor_opcode_next <= b_c_uint;
      -- IR_in <8..1>
      c_uint := main_opcode_temp AND to_unsigned(255, 16);
      IF c_uint(15 DOWNTO 8) /= "00000000" THEN
        address_data_next <= "11111111";
      ELSE
        address_data_next <= c_uint(7 DOWNTO 0);
      END IF;
      -- Go to the decode stage
```

```
            CPU_state_temp := to_unsigned(4, 8);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            --  DECODE AND EXECUTE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
          WHEN "00000100" =>
            CASE minor_opcode IS
              WHEN "0000" =>
                -- NOP
                CPU_state_temp := to_unsigned(1, 8);
              WHEN "0001" =>
                -- JMP
                addr_inc <= std_logic_vector(address_data);
                PC_inc_func <= std_logic_vector(to_unsigned(1, 2));
                PC_func <= std_logic_vector(to_unsigned(1, 2));
                CPU_state_temp := to_unsigned(1, 8);
              WHEN "0010" =>
                --JMPZ
                reg_a_0 := unsigned(Reg_A);
                IF reg_a_0 = 0 THEN
                  addr_inc <= std_logic_vector(address_data);
                  PC_inc_func <= std_logic_vector(to_unsigned(1, 2));
                  PC_func <= std_logic_vector(to_unsigned(1, 2));
                END IF;
                CPU_state_temp := to_unsigned(1, 8);
              WHEN "0011" =>
                -- CALL
                addr_inc <= std_logic_vector(address_data);
                PC_inc_func <= std_logic_vector(to_unsigned(1, 2));
                PC_func <= std_logic_vector(to_unsigned(1, 2));
                CPU_state_temp := to_unsigned(1, 8);
              WHEN "0100" =>
                --MOV A,xx
                Reg_OutA <= std_logic_vector(address_data);
                RegA_func <= std_logic_vector(to_unsigned(1, 3));
                CPU_state_temp := to_unsigned(1, 8);
              WHEN "0101" =>
                --MOV B,xx
                Reg_OutB <= std_logic_vector(address_data);
```

```
    RegB_func <= std_logic_vector(to_unsigned(1, 3));
    CPU_state_temp := to_unsigned(1, 8);
  WHEN "0110" =>
    CASE address_data IS
      WHEN "00000000" =>
        --RET
       PC_inc_func <= std_logic_vector(to_unsigned(2, 2));
        PC_func <= std_logic_vector(to_unsigned(2, 2));
        CPU_state_temp := to_unsigned(5, 8);
      WHEN "00000001" =>
        --MOV A,B
        ALU_func <= std_logic_vector(to_unsigned(0, 4));
        RegA_func <= std_logic_vector(to_unsigned(2, 3));
        RegB_func <= std_logic_vector(to_unsigned(2, 3));
        CPU_state_temp := to_unsigned(5, 8);
      WHEN "00000010" =>
        --MOV B,A
        ALU_func <= std_logic_vector(to_unsigned(1, 4));
        RegA_func <= std_logic_vector(to_unsigned(2, 3));
        RegB_func <= std_logic_vector(to_unsigned(2, 3));
        CPU_state_temp := to_unsigned(5, 8);
      WHEN "00000011" =>
        --XCHG A,B
        ALU_func <= std_logic_vector(to_unsigned(2, 4));
        RegA_func <= std_logic_vector(to_unsigned(2, 3));
        RegB_func <= std_logic_vector(to_unsigned(2, 3));
        CPU_state_temp := to_unsigned(5, 8);
      WHEN "00000100" =>
        --ADD A,B
        ALU_func <= std_logic_vector(to_unsigned(3, 4));
        RegA_func <= std_logic_vector(to_unsigned(2, 3));
        RegB_func <= std_logic_vector(to_unsigned(2, 3));
        CPU_state_temp := to_unsigned(5, 8);
      WHEN "00000101" =>
        --SUB A,B
        ALU_func <= std_logic_vector(to_unsigned(4, 4));
        RegA_func <= std_logic_vector(to_unsigned(2, 3));
        RegB_func <= std_logic_vector(to_unsigned(2, 3));
```

```
                CPU_state_temp := to_unsigned(5, 8);
              WHEN "00000110" =>
                --AND A,B
                ALU_func <= std_logic_vector(to_unsigned(5, 4));
                RegA_func <= std_logic_vector(to_unsigned(2, 3));
                RegB_func <= std_logic_vector(to_unsigned(2, 3));
                CPU_state_temp := to_unsigned(5, 8);
              WHEN "00000111" =>
                --OR A,B
                ALU_func <= std_logic_vector(to_unsigned(6, 4));
                RegA_func <= std_logic_vector(to_unsigned(2, 3));
                RegB_func <= std_logic_vector(to_unsigned(2, 3));
                CPU_state_temp := to_unsigned(5, 8);
              WHEN "00001000" =>
                --XOR A,B
                ALU_func <= std_logic_vector(to_unsigned(7, 4));
                RegA_func <= std_logic_vector(to_unsigned(2, 3));
                RegB_func <= std_logic_vector(to_unsigned(2, 3));
                CPU_state_temp := to_unsigned(5, 8);
              WHEN "00001001" =>
                --DEC A
                ALU_func <= std_logic_vector(to_unsigned(8, 4));
                RegA_func <= std_logic_vector(to_unsigned(2, 3));
                CPU_state_temp := to_unsigned(5, 8);
              WHEN OTHERS =>
                NULL;
            END CASE;
          WHEN OTHERS =>
            NULL;
        END CASE;
      WHEN "00000101" =>
        RegA_func <= std_logic_vector(to_unsigned(2, 3));
        RegB_func <= std_logic_vector(to_unsigned(2, 3));
        CPU_state_temp := to_unsigned(1, 8);
      WHEN OTHERS =>
        NULL;
  END CASE;
  CPU_state_next <= CPU_state_temp;
```

```
      major_opcode_next <= major_opcode_temp;
      main_opcode_next <= main_opcode_temp;
   END PROCESS CPU_Controller;
END fsm_SFHDL;
```

Пример 1. Код языка VHDL управляющего автомата проектируемого процессора, сгенерированный в автоматическом режиме с помощью Simulink HDL Coder системы Matlab/Simulink

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY PC_Incrementer IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      func : IN std_logic_vector(1 DOWNTO 0);
      addr : IN std_logic_vector(7 DOWNTO 0);
      PC_curr : IN std_logic_vector(7 DOWNTO 0);
      PC_next : OUT std_logic_vector(7 DOWNTO 0);
      Temp : OUT std_logic_vector(7 DOWNTO 0));
END PC_Incrementer;

ARCHITECTURE fsm_SFHDL OF PC_Incrementer IS

   SIGNAL PC_Temp : unsigned(7 DOWNTO 0);
   SIGNAL PC_Temp_next : unsigned(7 DOWNTO 0);
BEGIN
   initialize_PC_Incrementer : PROCESS (reset, clk)
      -- local variables
   BEGIN
      IF reset = '1' THEN
         PC_Temp <= to_unsigned(0, 8);
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            PC_Temp <= PC_Temp_next;
         END IF;
```

```
    END IF;
  END PROCESS initialize_PC_Incrementer;

  PC_Incrementer : PROCESS (PC_Temp, func, addr, PC_curr)
    -- local variables
    VARIABLE PC_Temp_temp : unsigned(7 DOWNTO 0);
  BEGIN
    PC_Temp_temp := PC_Temp;
    -- func = 0 => reset PC_Inc
    -- func = 1 => store into PC_Inc when JMP, JMPZ, CALL
    -- func = 2 => load from PC_Inc when RET
    PC_next <= PC_curr;
    Temp <= std_logic_vector(to_unsigned(0, 8));
    CASE func IS
      WHEN "00" =>
        -- reset PC_Inc
        PC_next <= std_logic_vector(to_unsigned(0, 8));
      WHEN "01" =>
        -- store into PC_Inc when JMP, JMPZ, CALL
        PC_next <= addr;
        PC_Temp_temp := unsigned(PC_curr);
        Temp <= std_logic_vector(PC_Temp_temp);
      WHEN "10" =>
        -- load from PC_Inc when RET
        PC_next <= std_logic_vector(PC_Temp);
      WHEN OTHERS =>
        NULL;
    END CASE;
    PC_Temp_next <= PC_Temp_temp;
  END PROCESS PC_Incrementer;
END fsm_SFHDL;
```

Пример 2. Регистр специального назначения процессора на языке VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY Program_Counter IS
```

```
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      func : IN std_logic_vector(1 DOWNTO 0);
      addr_in : IN std_logic_vector(7 DOWNTO 0);
      addr_out : OUT std_logic_vector(7 DOWNTO 0));
END Program_Counter;
ARCHITECTURE fsm_SFHDL OF Program_Counter IS
   SIGNAL PC_value : unsigned(7 DOWNTO 0);
   SIGNAL PC_value_next : unsigned(7 DOWNTO 0);
BEGIN
   initialize_Program_Counter : PROCESS (reset, clk)
      -- local variables
   BEGIN
      IF reset = '1' THEN
         PC_value <= to_unsigned(0, 8);
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            PC_value <= PC_value_next;
         END IF;
      END IF;
   END PROCESS initialize_Program_Counter;
   Program_Counter : PROCESS (PC_value, func, addr_in)
      -- local variables
      VARIABLE ain : unsigned(15 DOWNTO 0);
      VARIABLE ain_0 : unsigned(15 DOWNTO 0);
   BEGIN
      PC_value_next <= PC_value;
      -- Program Counter
      -- func = 0 => reset PC
      -- func = 1 => load PC
      -- func = 2 => increment PC
      addr_out <= std_logic_vector(PC_value);
      CASE func IS
         WHEN "00" =>
            -- reset
            PC_value_next <= to_unsigned(0, 8);
```

```
          WHEN "01" =>
            -- store into PC
            PC_value_next <= unsigned(addr_in);
          WHEN "10" =>
            -- increment PC
  ain := resize(PC_value & '0' & '0' & '0' & '0' & '0' & '0' & '0' & '0', 16);
            ain_0 := ain + 128;
IF (ain_0(15) /= '0') OR (ain_0(14 DOWNTO 7) = "11111111") THEN
              PC_value_next <= "11111111";
            ELSE
PC_value_next <= ain_0(14 DOWNTO 7) + ("0" & (ain_0(6)));
            END IF;
          WHEN OTHERS =>
            NULL;
        END CASE;
    END PROCESS Program_Counter;
END fsm_SFHDL;
```

Пример 3. Счетчик команд микропроцессора на языке VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY Instruction_ROM IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      addr : IN std_logic_vector(7 DOWNTO 0);
      read : IN std_logic;
      instr_out : OUT std_logic_vector(15 DOWNTO 0));
END Instruction_ROM;
ARCHITECTURE fsm_SFHDL OF Instruction_ROM IS
   -- TMW_TO_SIGNED
   FUNCTION tmw_to_signed(arg: unsigned; width: integer) RETURN signed IS
   BEGIN
      IF arg(arg'right) = 'U' OR  arg(arg'right) = 'X' THEN
         RETURN to_signed(1, width);
      END IF;
      RETURN to_signed(to_integer(arg), width);
   END FUNCTION;
```

```
   TYPE T_UFIX_16_256 IS ARRAY (255 DOWNTO 0) of unsigned(15
DOWNTO 0);
   SIGNAL data : T_UFIX_16_256;
   SIGNAL data_next : T_UFIX_16_256;
BEGIN
   initialize_Instruction_ROM : PROCESS (reset, clk)
      -- local variables
      VARIABLE b_0 : INTEGER;
   BEGIN
      IF reset = '1' THEN
         FOR b IN 0 TO 255 LOOP
            data(b) <= to_unsigned(0, 16);
         END LOOP;
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            FOR b_0 IN 0 TO 255 LOOP
               data(b_0) <= data_next(b_0);
            END LOOP;
         END IF;
      END IF;
   END PROCESS initialize_Instruction_ROM;
   Instruction_ROM : PROCESS (data, addr, read)
      -- local variables
      VARIABLE data_temp : T_UFIX_16_256;
   BEGIN
      FOR b IN 0 TO 255 LOOP
         data_temp(b) := data(b);
      END LOOP;
      data_temp(0) := to_unsigned(1036, 16);
      data_temp(1) := to_unsigned(1303, 16);
      data_temp(2) := to_unsigned(1540, 16);
      data_temp(3) := to_unsigned(1545, 16);
      data_temp(4) := to_unsigned(1358, 16);
      data_temp(5) := to_unsigned(1539, 16);
      data_temp(6) := to_unsigned(1541, 16);
      data_temp(7) := to_unsigned(1545, 16);
      data_temp(8) := to_unsigned(1542, 16);
      data_temp(9) := to_unsigned(523, 16);
      data_temp(10) := to_unsigned(263, 16);
      data_temp(11) := to_unsigned(1037, 16);
      data_temp(12) := to_unsigned(1397, 16);
      data_temp(13) := to_unsigned(1543, 16);
```

```
      data_temp(14) := to_unsigned(1539, 16);
      data_temp(15) := to_unsigned(1544, 16);
      data_temp(16) := to_unsigned(277, 16);
      data_temp(17) := to_unsigned(1135, 16);
      data_temp(18) := to_unsigned(1480, 16);
      data_temp(19) := to_unsigned(1542, 16);
      data_temp(20) := to_unsigned(1536, 16);
      data_temp(21) := to_unsigned(785, 16);
      data_temp(22) := to_unsigned(0, 16);
      IF read = '1' THEN
instr_out <= std_logic_vector(data_temp(to_integer(tmw_to_signed(unsigned(addr) + 1,
32) - 1)));
      ELSE
         instr_out <= std_logic_vector(to_unsigned(0, 16));
      END IF;
      FOR c IN 0 TO 255 LOOP
         data_next(c) <= data_temp(c);
      END LOOP;
   END PROCESS Instruction_ROM;
END fsm_SFHDL;
```

Пример 4. Память программ процессора на языке VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY Instruction_Register IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      func : IN std_logic_vector(1 DOWNTO 0);
      IR_in : IN std_logic_vector(15 DOWNTO 0);
      IR_out : OUT std_logic_vector(15 DOWNTO 0));
END Instruction_Register;

ARCHITECTURE fsm_SFHDL OF Instruction_Register IS
   SIGNAL IR_value : unsigned(15 DOWNTO 0);
   SIGNAL IR_value_next : unsigned(15 DOWNTO 0);
BEGIN
   initialize_Instruction_Register : PROCESS (reset, clk)
```

```
    -- local variables
  BEGIN
    IF reset = '1' THEN
      IR_value <= to_unsigned(0, 16);
    ELSIF clk'EVENT AND clk= '1' THEN
      IF clk_enable= '1' THEN
        IR_value <= IR_value_next;
      END IF;
    END IF;
  END PROCESS initialize_Instruction_Register;
  Instruction_Register : PROCESS (IR_value, func, IR_in)
    -- local variables
  BEGIN
    IR_value_next <= IR_value;
    -- A 16-bit Instruction Register with the following func:
    -- func == 0 => reset
    -- func == 1 => store into IR
    -- func == 2 => read from IR;
    -- otherwise, preserve old value and return 0
    IR_out <= std_logic_vector(to_unsigned(0, 16));
    CASE func IS
      WHEN "00" =>
        -- reset
        IR_value_next <= to_unsigned(0, 16);
      WHEN "01" =>
        -- store into IR
        IR_value_next <= unsigned(IR_in);
      WHEN "10" =>
        -- read IR
        IR_out <= std_logic_vector(IR_value);
      WHEN OTHERS =>
        NULL;
    END CASE;
  END PROCESS Instruction_Register;
END fsm_SFHDL;
```

Пример 5. Блок регистра инструкций микропроцессора на языке VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY RegisterA IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      func : IN std_logic_vector(2 DOWNTO 0);
      Reg_in_A_1 : IN std_logic_vector(7 DOWNTO 0);
      Reg_in_A_2 : IN std_logic_vector(7 DOWNTO 0);
      Reg_out_A : OUT std_logic_vector(7 DOWNTO 0));
END RegisterA;

ARCHITECTURE fsm_SFHDL OF RegisterA IS

   SIGNAL Reg_value : unsigned(7 DOWNTO 0);
   SIGNAL Reg_value_next : unsigned(7 DOWNTO 0);

BEGIN
   initialize_RegisterA : PROCESS (reset, clk)
      -- local variables
   BEGIN
      IF reset = '1' THEN
         Reg_value <= to_unsigned(0, 8);
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            Reg_value <= Reg_value_next;
         END IF;
      END IF;
   END PROCESS initialize_RegisterA;

   RegisterA : PROCESS (Reg_value, func, Reg_in_A_1, Reg_in_A_2)
      -- local variables
   BEGIN
      Reg_value_next <= Reg_value;
      -- func == 0 => reset;
```

```
      -- func == 1 => store into RegisterA from port 1;
      -- func == 2 => store into RegisterA from port 2;
      -- func == 3 => read from RegisterA;
      -- HDL specific fimath
      Reg_out_A <= std_logic_vector(Reg_value);
      CASE func IS
         WHEN "000" =>
            -- reset
            Reg_value_next <= to_unsigned(0, 8);
         WHEN "001" =>
            -- store into Reg_A from port 1
            Reg_value_next <= unsigned(Reg_in_A_1);
         WHEN "010" =>
            -- store into Reg_A from port 2
            Reg_value_next <= unsigned(Reg_in_A_2);
         WHEN "011" =>
            -- read Reg_A
            Reg_out_A <= std_logic_vector(Reg_value);
         WHEN OTHERS =>
            NULL;
      END CASE;
   END PROCESS RegisterA;
END fsm_SFHDL;
```

Пример 6. Блок регистра общего назначения A микропроцессора на языке VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY ALU IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      func : IN std_logic_vector(3 DOWNTO 0);
      inA : IN std_logic_vector(7 DOWNTO 0);
      inB : IN std_logic_vector(7 DOWNTO 0);
      outA : OUT std_logic_vector(7 DOWNTO 0);
      outB : OUT std_logic_vector(7 DOWNTO 0));
```

```
END ALU;
ARCHITECTURE fsm_SFHDL OF ALU IS
   -- TMW_TO_SIGNED
   FUNCTION tmw_to_signed(arg: unsigned; width: integer) RETURN signed IS
   BEGIN
      IF arg(arg'right) = 'U' OR  arg(arg'right) = 'X' THEN
         RETURN to_signed(1, width);
      END IF;
      RETURN to_signed(to_integer(arg), width);
   END FUNCTION;
   -- TMW_TO_UNSIGNED
   FUNCTION tmw_to_unsigned(arg: signed; width: integer) RETURN unsigned
IS
      CONSTANT ARG_LEFT: INTEGER := ARG'LENGTH-1;
      ALIAS XARG: SIGNED(ARG_LEFT downto 0) is ARG;
      VARIABLE result : unsigned(width-1 DOWNTO 0);
      VARIABLE argSize : integer;
   BEGIN
      IF XARG(XARG'high-1) = 'U' OR  arg(arg'right) = 'X' THEN
         RETURN to_unsigned(1, width);
      END IF;
      IF (ARG_LEFT < width-1) THEN
         result := (OTHERS => XARG(ARG_LEFT));
         result(ARG_LEFT downto 0) := unsigned(XARG);
      ELSE
         result(width-1 downto 0) := unsigned(XARG(width-1 downto 0));
      END IF;
      RETURN result;
   END FUNCTION;
BEGIN
   ALU : PROCESS (func, inA, inB)
      -- local variables
      VARIABLE X_temp : unsigned(7 DOWNTO 0);
      VARIABLE ina_0 : unsigned(7 DOWNTO 0);
      VARIABLE ina_1 : signed(8 DOWNTO 0);
      VARIABLE ina_2 : signed(8 DOWNTO 0);
   BEGIN
      -- This 8-bit ALU supports the following operations:
      -- MOV, XCHG, ADD, SUB, AND, OR, XOR, DEC
      -- func = 0 => MOV A,B
      -- func = 1 => MOV B,A
      -- func = 2 => XCHG A,B
```

```
        -- func = 3 => ADD A,B
        -- func = 4 => SUB A,B
        -- func = 5 => AND A,B
        -- func = 6 => OR A,B
        -- func = 7 => XOR A,B
        -- func = 8 => DEC A
        -- Simply pass the inA, when there is no designated func
        outA <= inA;
        -- Simply pass the inB, when there is no designated func
        outB <= inB;
        CASE func IS
           WHEN "0000" =>
              --MOV A,B
              outA <= inB;
           WHEN "0001" =>
              --MOV B,A
              outB <= inA;
           WHEN "0010" =>
              --XCHG A,B
              X_temp := unsigned(inB);
              outB <= inA;
              outA <= std_logic_vector(X_temp);
           WHEN "0011" =>
              --ADD A,B
              ina_0 := unsigned(inA) + unsigned(inB);
              outA <= std_logic_vector(ina_0);
           WHEN "0100" =>
              --SUB A,B
ina_1 := tmw_to_signed(unsigned(inA), 9) - tmw_to_signed(unsigned(inB), 9);
              IF ina_1(8) = '1' THEN
                 outA <= "00000000";
              ELSE
outA <= std_logic_vector(resize(unsigned(ina_1(7 DOWNTO 0)), 8));
              END IF;
           WHEN "0101" =>
              --AND A,B
outA <= std_logic_vector(tmw_to_unsigned(tmw_to_signed(unsigned(inA), 32)
AND tmw_to_signed(unsigned(inB), 32), 8));
           WHEN "0110" =>
              --OR A,B
outA <= std_logic_vector(tmw_to_unsigned(tmw_to_signed(unsigned(inA), 32)
OR tmw_to_signed(unsigned(inB), 32), 8));
```

```
        WHEN "0111" =>
          --XOR A,B
outA <= std_logic_vector(tmw_to_unsigned(tmw_to_signed(unsigned(inA), 32)
XOR tmw_to_signed(unsigned(inB), 32), 8));
        WHEN "1000" =>
          --DEC A
          ina_2 := tmw_to_signed(unsigned(inA), 9) - 1;
          IF ina_2(8) = '1' THEN
            outA <= "00000000";
          ELSE
          outA <= std_logic_vector(resize(unsigned(ina_2(7 DOWNTO 0)), 8));
          END IF;
        WHEN OTHERS =>
          NULL;
      END CASE;
   END PROCESS ALU;
END fsm_SFHDL;
```

Пример 7. Блок АЛУ микропроцессора на языке VHDL

Процессор, позволяющий проводить вычисления в формате с фиксированной зяпятой, код языка которого был получен с использованием Simulink HDL Coder системы визуального имитационного моделирования Matlab/Simulink, показал свою работоспособность в САПР Quartus II компании Altera. Процессор может быть успешно размещен в ПЛИС Stratix III EP3SL50F484C2, и занимает менее 1 % ресурсов адаптивных таблиц перекодировок (ALUT, 209) для реализации комбинационной логики и менее 1 % ресурсов последовательностной логики (регистров, 105).

Автоматически сгенерированный код языка VHDL с использованием Simulink HDL Coder системы Matlab/Simulink, позволяет значительно ускорить процесс разработки пользовательских микропроцессорных ядер для реализации их в базисе ПЛИС.

К недостаткам следует отнести наличие достаточно большого числа явных преобразований тесно связанных между собой типов, что определяется форматом представления исходных данных системы Matlab/Simulink.

4.6. Проектирование микропроцессорных ядер с конвейерной архитектурой для реализации в базисе ПЛИС

Воспользуемся системой команд синхронного процессора с циклом работы в два такта и спроектируем микропроцессорное ядро с использованием конвейерной архитектуры. Типовой конвейер микропроцессора содержит пять стадий: выборка инструкции; декодирование инструкции; адресация и выборка операнда из ОЗУ; выполнение арифметических операций; сохранение результата операции. Каждый этап команды рассматривается как каскад конвейера. Таким образом, можно организовать наложение команд, при котором новая команда будет начинать выполняться в первый момент каждого такта. Благодаря использованию внутреннего параллелизма потока команд конвейерная обработка позволяет существенно снизить в среднем время выполнения одной команды. Пропускная способность машины с конвейерной обработкой определяется числом команд, пропущенных через конвейер за единицу времени.

Для реализации процессора необходима память, в которой будут храниться команды микропроцессора (память программ) и инструкции для управляющего автомата. Проектируемая память имеет асинхронный сигнал сброса reset, состоит из двух массивов памяти емкостью 4096 бит. Ниже приведен код языка VHDL асинхронной памяти (пример 1). ПЗУ разделено на 2 области и обладает двумя адресными шинами addr_cmd[15..0] и addr_avt[15..0]. По шине avt_out[15..0] передаются инструкции управляющего автомата, а по шине cmd_out[15..0] передаются команды микропроцессора.

Разработаем для микропроцессорного ядра управляющий автомат на девять состояний (рис.4.35). Использование управляющего автомата удовлетворяет современной концепции синхронного кодирования при реализации цифровых устройств в базисе ПЛИС.

Управляющий автомат имеет вход синхронизации clk и асинхронного сброса rst, который устанавливает автомат в начальное состояние INST. В начальном состоянии INST по шине instr[15..0] происходит загрузка из ПЗУ инструкции для управляющего автомата в регистр инструкций, в котором выделяются разряды [15..12] (шина instr[15..12]) для декодирования движений по веткам автомата и разряды [11..9] (шина instr[11..9]) для декодирования логико-арифметических операций.

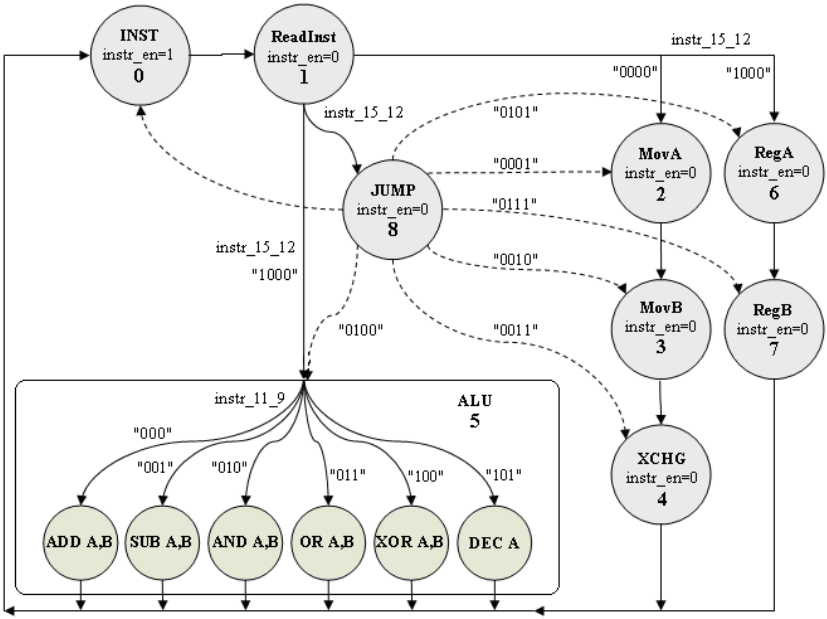


Рис.4.35. Блок-схема управляющего автомата микропроцессорного ядра на 9 состояний

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY rom_syn IS
PORT (addr_cmd : IN std_logic_vector(15 DOWNTO 0);
        addr_avt : IN std_logic_vector(15 DOWNTO 0);
```

```
        cmd_out : OUT std_logic_vector(15 DOWNTO 0);
        avt_out : OUT std_logic_vector(15 DOWNTO 0));
END rom_syn;
ARCHITECTURE a OF rom_syn IS
TYPE T_UFIX_16_256 IS ARRAY (255 DOWNTO 0) of unsigned(15
DOWNTO 0);
BEGIN
        PROCESS (addr_cmd)
        VARIABLE b : INTEGER;
        VARIABLE c : INTEGER;
        VARIABLE data_temp : T_UFIX_16_256;
BEGIN
        FOR b IN 128 TO 255 LOOP
        data_temp(b) := to_unsigned(0, 16);
        END LOOP;
        data_temp(128) := to_unsigned(0, 16);
        data_temp(129) := to_unsigned(1165, 16);
        data_temp(130) := to_unsigned(1358, 16);
        data_temp(131) := to_unsigned(1540, 16);
        data_temp(132) := to_unsigned(1541, 16);
        data_temp(133) := to_unsigned(1542, 16);
        data_temp(134) := to_unsigned(1543, 16);
        data_temp(135) := to_unsigned(1544, 16);
        data_temp(136) := to_unsigned(1545, 16);
        data_temp(137) := to_unsigned(1539, 16);
        data_temp(138) := to_unsigned(1546, 16);
        data_temp(139) := to_unsigned(1547, 16);
        cmd_out <=
std_logic_vector(data_temp(to_integer(unsigned(addr_cmd))));
END PROCESS;
PROCESS (addr_avt)
        VARIABLE data_temp : T_UFIX_16_256;
BEGIN
        FOR c IN 0 TO 255 LOOP
        data_temp(c) := to_unsigned(0, 16);
END LOOP;
        data_temp(0) := "0000000000000000";
        data_temp(1) := "1000000000000000";
```

```
        data_temp(2) := "1000001000000000";
        data_temp(3) := "1000010000000000";
        data_temp(4) := "1000011000000000";
        data_temp(5) := "1000100000000000";
        data_temp(6) := "1000101000000000";
        data_temp(7) := "0001000000000000";
        data_temp(8) := "0010000000000000";
        data_temp(9) := "0011000000000000";
        data_temp(10) := "0100000000000000";
        data_temp(11) := "1001000000000000";
        avt_out <=
std_logic_vector(data_temp(to_integer(unsigned(addr_avt))));
END PROCESS;
END a;
```

Пример 1. Код языка VHDL блока асинхронной памяти

На выходе автомата ip[15..0] с помощью битов контроля (внутренний сигнал control_signal) формируется адрес команды, хранящийся в ПЗУ. Высокий уровень сигнала instr_en разрешает получение новой инструкции из ПЗУ для управляющего автомата, и увеличивает содержимое счетчика на единицу. Сигнал num_state[3..0] показывает номер состояния, в котором находится управляющий автомат.

Следующее состояние, в которое переходит автомат по переднему фронту синхроимпульса clk – ReadInst. В состоянии ReadInst происходит чтение полученной инструкции и выбор следующего состояния автомата. В состояниях INST и ReadInst в АЛУ недолжно выполнять логико-арифметических операций. Автомат реализует команду NOP.

С приходом фронта синхроимпульса автомат перейдет в одну из возможных веток (всего возможны 4 ветки), например, в состояние MovA если на шине instr_15_12 присутствует код “0000” (ветка с состояниями MovA, MovB, XCHG для выполнения АЛУ трех регистровых операций пересылки), в состояние ALU если “1000” (в этом состоянии АЛУ выполняет шесть логико-арифметических операций), в состояние RegA

если “1001” (ветка с состояниями RegA, RegB, АЛУ выполняет загрузку РОН А и В с входного порта) или в состояние JUMP, если на шине instr_15_12 присутствует любое другое значение. В состоянии JUMP в зависимости от кода на шине instr_15_12 автомат может “перепрыгнуть” в другое состояние (возможные переходы показаны пунктирными линиями), при этом АЛУ не выполняет операций, а автомат реализует команду NOP.

В состоянии MovA происходит непосредственная загрузка в регистр А операнда, заданного младшим байтом команды. Следующим состоянием, в котором произойдет загрузка операнда в регистр В будет MovB. В состоянии XCHG произойдет обмен содержимого в регистрах А и В. После этого автомат возвращается в состояние INST и читает следующую инструкцию instr из памяти. В состоянии ALU код на шине instr_11_9 выбирает логико-арифметическую операцию, которая будет выполнена в АЛУ. В состояниях RegA и RegB происходит загрузка данных в регистры А и В с входного порта. VHDL описание проектируемого автомата с использованием двухпроцессорного шаблона показано ниже (пример 2):

```
ARCHITECTURE behave OF Control IS
-- Definition of the state names
TYPE state_type IS (Inst, ReadInst, MovA, MovB, XCHG, ALU, JUMP,
RegA, RegB);
SIGNAL state, next_state : state_type;
Signal control_signal: std_logic_vector(15 downto 0);
BEGIN
-- State process
PROCESS(clk, rst)
BEGIN
IF rst = '1' THEN
state <= Inst;
ELSIF clk'event and clk='1' THEN
state <= next_state;
END IF;
END PROCESS;
```

```
-- Logic Process
PROCESS(state)
BEGIN
CASE state IS
--Instruction
WHEN Inst =>
control_signal <= "0000000010000000"; -- 128(D) NOP
num_state <= "0000";
instr_en <= '1';
next_state <= ReadInst;
--Read Instruction
WHEN ReadInst =>
control_signal <= "0000000010000000"; -- 128(D) NOP
num_state <= "0001";
instr_en <= '0';
IF instr_15_12 = "0000" THEN next_state <=MovA;
ELSIF instr_15_12 = "1000" THEN next_state <=ALU;
ELSIF instr_15_12 = "1001" THEN next_state<=RegA;
ELSE next_state <= JUMP;
END IF;
--MovA
WHEN MovA =>
control_signal <= "0000000010000001"; -- 129(D) MOV A,xx
next_state <= MovB;
num_state <= "0010";
instr_en <= '0';
--MovB
WHEN MovB =>
control_signal <= "0000000010000010"; -- 130(D) MOV B,xx
next_state <= XCHG;
num_state <= "0011";
instr_en <= '0';
-- XCHG
WHEN XCHG =>
next_state <= Inst;
num_state <= "0100";
instr_en <= '0';
control_signal <= "0000000010001001"; --137(D) XCHG A,B
```

```
-- ALU
WHEN ALU =>
instr_en <= '0';
num_state <= "0101";
IF instr_11_9 = "000" THEN
control_signal <= "0000000010000011"; -- 131(D) ADD A,B
next_state <= INST;
ELSIF instr_11_9 = "001" THEN
control_signal <= "0000000010000100"; -- 132(D) SUB A,B
next_state <= INST;
ELSIF instr_11_9 = "010" THEN
control_signal <= "0000000010000101"; -- 133(D) AND A,B
next_state <= INST;
ELSIF instr_11_9 = "011" THEN
control_signal <= "0000000010000110"; -- 134(D) OR A,B
next_state <= INST;
ELSIF instr_11_9= "100" THEN
control_signal <= "0000000010000111"; -- 135(D) XOR A,B
next_state <= INST;
ELSIF instr_11_9 = "101" THEN
control_signal <= "0000000010001000"; -- 136(D) DEC A
next_state <= INST;
END IF;
--RegA
WHEN RegA =>
next_state <= RegB;
num_state <= "0110";
instr_en <= '0';
control_signal <= "0000000010001010"; --138(D) MOV A,indata
--RegB
WHEN RegB =>
next_state <= INST;
num_state <= "0111";
instr_en <= '0';
control_signal <= "0000000010001011"; --139(D) MOV B,indata
--JUMP
WHEN JUMP =>
control_signal <= "0000000010000000"; --128(D) NOP
```

```
instr_en <= '0';
num_state <= "1000";
IF instr_15_12 = "0001" THEN next_state <= MovA;
ELSIF instr_15_12 = "0010" THEN next_state <= MovB;
ELSIF instr_15_12 = "0011" THEN next_state <= ALU;
ELSIF instr_15_12 = "0100" THEN next_state <= XCHG;
ELSIF instr_15_12 = "0110" THEN next_state <= RegA;
ELSIF instr_15_12 = "0111" THEN next_state <= RegB;
ELSE next_state <= Inst;
END IF;
END case;
END process;
ip <= control_signal;
END behave;
```

Пример 2. Фрагмент кода языка VHDL управляющего автомата

Синхронное АЛУ выполняет различные логико-арифметические операции над операндами, значения которых сохраняются в регистрах-защелках A и B. В этом блоке реализованы следующие команды (команды JMPZ, CALL, RET не поддерживаются, добавлены две новые команды с кодом 1546(D) (MOV A,indata) и 1547(D) (MOV B,indata) для загрузки РОН A и B с входного порта): Mov A,xx; Mov B,xx; XCHG A,B; ADD A,B; SUB A,B; AND A,B; OR A,B; XOR A,B; DEC A; Reg A; Reg B (пример 3).

```
signal regA,regB,indata: std_logic_vector(7 downto 0);
BEGIN
PROCESS (clk,res)
BEGIN
regA<=a;
regB<=b;
indata<=input;
if (res = '1') then
                    regA <="00000000";
                    regB <="00000000";
elsif (clk'event and clk='1') then
```

```
case conv_integer(cmd) is
when 1024 to 1279 => regA<=cmd(7 downto 0); enaa<='1'; enab<='0';
when 1280 to 1535 => regB<=cmd(7 downto 0); enab<='1'; enaa<='0';
when 1537=>regA<=regB; enaa<='1'; enab<='0';
when 1538=>regB<=regA; enaa<='0'; enab<='1';
when 1539=>regA<=regB; regB<=regA;
enaa<='1'; enab<='1';
when 1540=>regA<=regA+regB; enaa<='1'; enab<='0';
when 1541=>regA<=regA-regB; enaa<='1'; enab<='0';
when 1542=>regA<=regA and regB;
enaa<='1'; enab<='0';
when 1543=>regA<=regA or regB;
enaa<='1'; enab<='0';
when 1544=>regA<=regA xor regB;
enaa<='1'; enab<='0';
when 1545=>regA<=regA-1; enaa<='1'; enab<='0';
when 1546=>regA<=indata; enaa<='1'; enab<='0';
when 1547=>regB<=indata; enaa<='0'; enab<='1';
when others=> dataa<=regA; datab<=regB;
enaa<='0'; enab<='0';
end case;
end if;
dataa<=regA; datab<=regB;
end process;
```

Пример 3. Фрагмент кода языка VHDL блока АЛУ

На рис.4.36 показана схема микропроцессорного ядра с конвейерной архитектурой. В первом состоянии управляющего автомата происходит чтение инструкции из ПЗУ (instr[15..0]) и выделение из нее полей – instr_15_12[3..0] и instr_11_9[2..0]. Адрес этой инструкции для автомата формирует счетчик (шина pc[15..0]), прибавляющий 1 к предыдущему адресу, когда автомат выполнит цикл команд и вернется в состояние INST. Автомат для каждого своего состояния вырабатывает адрес нужной команды, хранящейся в ПЗУ программ (шина ip[15..0]) с помощью битов контроля (сигнал control_signal). Эта команда по шине команд cmd_out[15..0] передается в АЛУ, где

выполняется требуемая операция, результаты помещаются в регистры. Схема имеет 2 регистра (восьмиразрядный регистр – защелку) общего назначения A и B, данные из которых попадают в АЛУ для выполнения следующей операции.

На рис.4.37 приведены временные диаграммы работы микропроцессорного ядра. В начальном состоянии управляющего автомата (INST) происходит запись инструкции в блок выделения полей (блок instreg) и из ПЗУ программ извлекается команда NOP с кодом 0, при которой нет операций. С приходом переднего фронта синхросигнала clk автомат переключается в состояние ReadInst, в котором читается полученная инструкция и выбирается следующее состояние.

Во втором состоянии выполняется команда Mov A,xx, которая загружает в регистр A значение, заданное младшим байтом команды. Из ПЗУ программ была получена команда 48D(H) (1165(D)) и в регистр A было загружено число 8D(H) или 141 в десятичной системе. Согласно схеме на рис.4.35 следующее состояние, которое принимает автомат – состояние номер 3 (MovB). В этом состоянии выполняется команда Mov B,xx. Команда загружает в регистр B значение, заданное младшим байтом команды. Тестирование команды пересылки Mov B,xx показано на рис.4.37. Из ПЗУ была получена команда 54E(H) и в регистр B было загружено число 4E(H) или 78 в десятичной системе. Согласно схеме на рис.4.35 следующее состояние автомата - XCHG. Из ПЗУ была извлечена команда 603(H) и регистры A и B обменялись значениями. При этом на шинах instr_15_12 = “0000” и instr_11_9 = “000”.

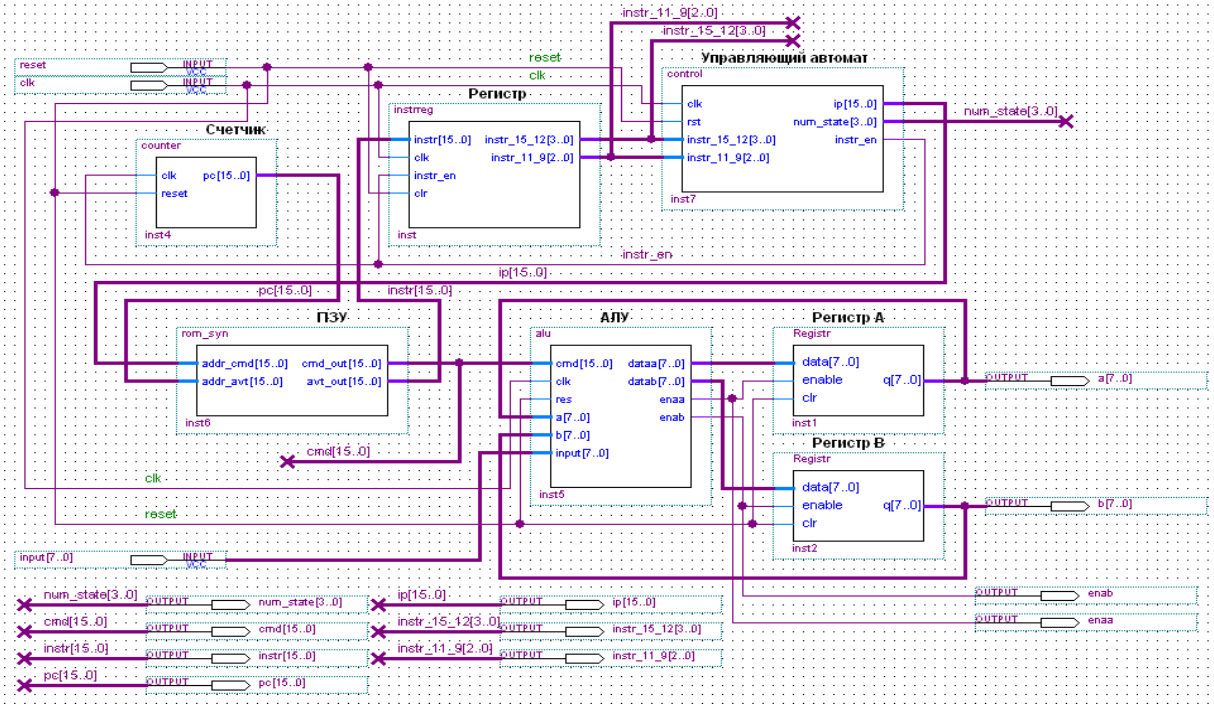


Рис.4.36. Схема микропроцессорного ядра с конвейерной архитектурой в графическом редакторе САПР ПЛИС Quartus II

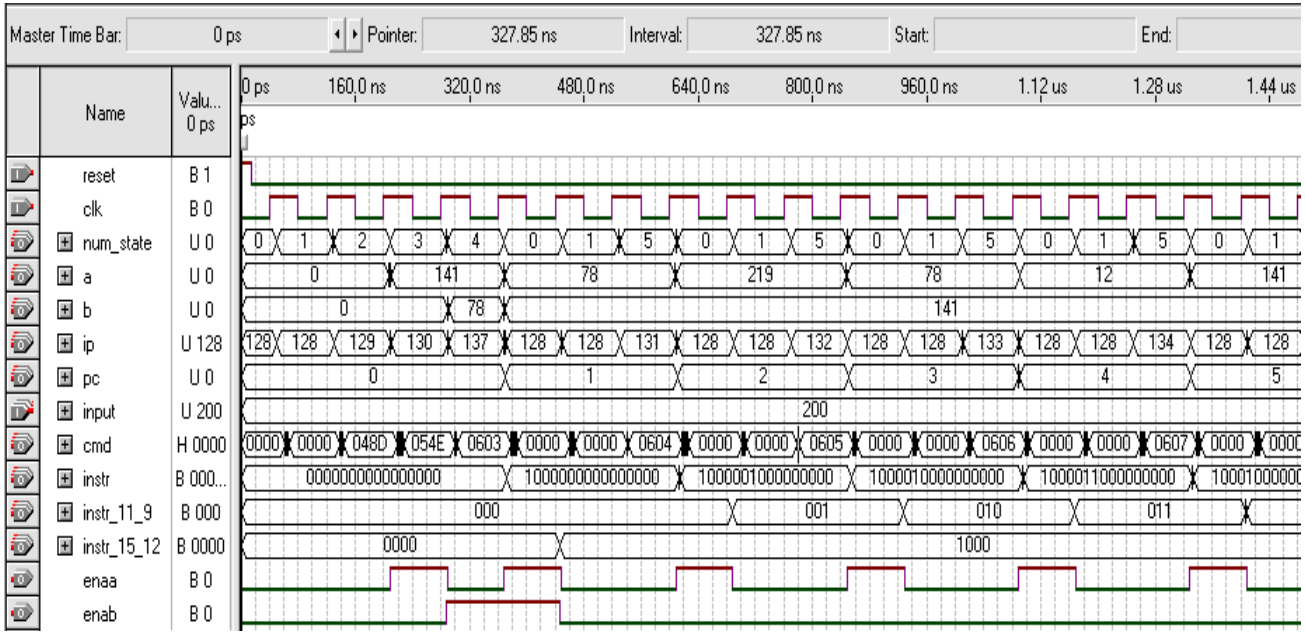


Рис.4.37. Временные диаграммы работы микропроцессорного ядра с конвейерной архитектурой в векторном редакторе САПР ПЛИС Quartus II

Код команды ADD – 604(H) или 1540(D). При этом на шинах instr_15_12 = “1000” а instr_11_9 = “000”. Значения регистров А и В были сложены, и результат помещен в регистр А. Команда SUB A,B выполнила вычитание значений в регистрах А и В, результат помещен в регистр А (код команды – 605(H)). Команда AND A,B, выполняющая операцию побитного логического И значений в регистрах А и В, также показана на рис.4.37. Команда логическое И с кодом 606(H) работает верно. Результат команды был помещен в регистр А. Команда логическое ИЛИ с кодом 607(H), выполняет операцию побитное логическое ИЛИ (команда OR A,B). Результат выполнения команды помещен в регистр А. Команда XOR A,B выполняет побитное логическое исключающее ИЛИ значений в регистрах А и В. Результат помещен в регистр А (код 608(H)).

Особенностью разработанного микропроцессорного ядра с конвейерной архитектурой является использование управляющего автомата и наличие двух блоков памяти: для хранения команд и для хранения инструкций управляющего автомата. При этом АЛУ выполняет только логико-арифметические операции, а прыжковые команды типа JMP реализует управляющий автомат. Проект микропроцессора на языке VHDL может быть успешно размещен в ПЛИС APEX20KE (EP20K160EB356-1), при этом общее число задействованных ресурсов составляет 70 %, с рабочей тактовой частотой до 33 МГц.

4.7. Использование ресурсов ПЛИС Stratix III фирмы Altera при проектировании микропроцессорных ядер

В данном разделе рассматривается использование ресурсов ПЛИС серии Stratix III при проектировании различных вариантов микропроцессорного ядра, система команд и управляющий автомат взяты из раздела 4.1. Рассматриваются варианты: управляющий автомат и синхронное ПЗУ разработанны на языке VHDL для реализации в базисе ПЛИС APEX20KE (EP20K30ETC144), вариант 1; микропроцессорное ядро для вычислений с фиксированной запятой в системе Matlab/Simulink, код языка VHDL получен с применением Simulink HDL Coder, вариант 2; управляющий автомат и синхронное ПЗУ на язке VHDL в базисе ПЛИС Stratix III EP3SL50F484C2, вариант 3; управляющий автомат на языке VHDL и мегафункция синхронного ПЗУ RAM: 1-PORT в базисе ПЛИС Stratix III, вариант 4; управляющий автомат созданный с применением приложения StateFlow Matlab/Simulink и асинхронное ПЗУ, код языка VHDL получен с применением Simulink HDL Coder в базисе ПЛИС Stratix III EP3SL50F484C2, вариант 5 (табл.4.4).

Используемые ресурсы ПЛИС Stratix III EP3SL50F484C2 и ПЛИС APEX20KE EP20K30ETC144 показаны в табл.4.4. Из анализа таблицы можно сделать выводы, что различные варианты микропроцессорного ядра на ПЛИС Stratix III занимают менее 1 % используемых ресурсов ПЛИС и обладают более чем в два раза большей тактовой частотой чем при реализации в базисе ПЛИС APEX20KE.

Варианты 1, 3 и 4 наиболее схожи между собой, т.к. базируются на одном варианте управляющего автомата. Варианты 2 и 5 базируются лишь на системе команд. Варианты 1, 3 и 4 задействуют одинаковое количество триггеров (33 триггера), а по числу упакованных элементов комбинационной логики ПЛИС Stratix III превосходят ПЛИС APEX20KE и обеспечивают выигрыш по быстродействию микропроцессорных ядер.

Наиболее удачный вариант по быстродействию – вариант 3, а по функциональной сложности, которая

обеспечивает ряд преимуществ, такие как поддержка формата с фиксированной запятой и распределенная система управления блоками процессора – вариант 2. Вариант 2 не использует блоки встроенной памяти. Справедливости ради следует отметить, что данные примеры не позволяют в полной мере оценить используемые ресурсы ПЛИС Stratix III, т.к. проекты слишком малы и позволяют максимально загрузить ПЛИС.

Вариант 2 наиболее близок к архитектуре микропроцессорного ядра PicoBlaze, для реализации в базисе ПЛИС Spartan II, Virtex (рис.4.38) и является его упрощенной версией.

Вариант 2 состоит из следующих блоков: управляющий автомат (блок CPU_Controller); память программ - ПЗУ процессора (блок Memory); АЛУ процессора (блок alu); двух регистров общего назначения (РОН, блоки RegisterA и RegisterB); регистра специального назначения (РСН), выполняющего роль стека (блок PC_Inc); счетчика команд (блок PC); регистра инструкций (блок Instruction_Reg).

Архитектура микропроцессорного ядра PicoBlaze основана на концепции раздельных шин данных и команд (гарвардская или двухшинная архитектура). Память для хранения данных и память для хранения программы располагаются в разных местах, допуская полное совмещение во времени операций вызова команды из памяти ее выполнения, что позволяет добиться высокой скорости выполнения операций. Варианты микропроцессорных ядер (вариант 1-5) условно характеризуются одношинной структурой, т.к. в рассматриваемых вариантах отсутствует память данных.

Таблица 4.4

Используемы ресурсы ПЛИС Stratix III EP3SL50F484C2 и APEX20KE EP20K30ETC144

Stratix III/ APEX20KE	Управляющий автомат и синхронное ПЗУ на языке VHDL APEX20KE EP20K30 ETC144	Микропро-цессорное ядро для вычислений с фиксированной запятой Matlab/ Simulink Stratix III EP3SL50 F484C2	Управляю-щий автомат и синхронное ПЗУ на язке VHDL Stratix III EP3SL50 F484C2	Управляю-щий автомат на языке VHDL, мегафун-кция синхронного ПЗУ RAM: 1-PORT Stratix III EP3SL50 F484C2	Управляющий автомат созданный с применением StateFlow Matlab /Simulink и асинхронное ПЗУ на языке VHDL ПЛИС Stratix III EP3SL50 F484C2
1	2	3	4	5	6
Вариант	1	2	3	4	5
Combinational ALUTs/LUT	116	201	111	106	103
7 входов/4 6/3 5/2 4/1 <=3/0	101 9 23 16 0	8 22 45 29 97	0 6 25 49 31	0 22 45 15 24	1 13 29 27 33

Продолжение табл.4.4

1	2	3	4	5	6
Режимы работы ALUTs: normal mode	-	158	94	89	85
extended LUT mode	-	8	0	0	1
arithmetic mode	-	26	17	17	17
shared arithmetic mode	-	9	0	0	0
Total registers/LC Registers	33	93	33	33	59
ALM/LC	149	119	61	58	64
Memory Bits (1880064)/ (24576)	1536 (6 %)	0	0	4096 (< 1 %)	0
Максимальная тактовая частота, МГц	73.84	318.57 (Slow 1100 mV 85C Model)	327.98 (Slow 1100 mV 85C Model)	245.64 (Slow 1100 mV 85 Model)	275.33 (Slow 1100 mV 85C Model)

Микропроцессорное ядро PicoBlaze содержит 16 восьмиразрядных регистров входящие в блок РОН (в варианте 2 их два), 8 разрядное АЛУ, регистр статуса и регистр фиксации флагов при выполнении обработки прерываний (в варианте 2 отсутствуют), программный счетчик, блок управления вводом/выводом (в варианте 2 отсутствует), стек (15 уровней, в варианте 2 стек организован на регистре R), схема управления прерываниями (в варианте 2 отсутствует), блок управления выбором адреса следующей команды (в варианте 2 отсутствует), дешифратор команд и ПЗУ на основе блочной памяти ПЛИС Block SelectRAM.

Вариант 2 поддерживает лишь несколько команд из 3 групп команд ядра PicoBlaze (всего 6 групп: 1 группа - команды, управляющие последовательностью выполнения операций в программе и команды обработки подпрограмм, например, JUMP, CALL, RETURN; 2 группа – логические команды, например, поразрядное умножение AND; 3 группа – арифметические команды, например, команда получения сумму двух операндов без учета переноса ADD; 4 группа – команды сдвига; 5 группа - команды ввода/вывода; 6 группа – команды для обслуживания прерываний). Ядро PicoBlaze поддерживает всего 49 команд, время выполнения команд - постоянное. В табл.4.5 и табл.4.6 для сравнения показан формат команд переходов JUMP ядра PicoBlaze и варианта 2 с системой команд из раздела 4.1. Ядро PicoBlaze поддерживает 1 безусловную и 3 условных команд переходов. В варианте 2 из за отсутствия развитого АЛУ (например, не предусмотрены арифметические команды с учетом переноса/заема и др), регистра статуса и блока управления выбора следующего адреса, поддерживается лишь одна команда перехода с условием JMPZ.

Трассировочная архитектура MultiTrack, используемая в ПЛИС Stratix III, обеспечивает связь между различными кластерами логических элементов и характеризуется определенным количеством шагов (hop), необходимых для того, чтобы соединить один LAB с другим. Чем меньше

количество шагов и предсказуемее модель трассировки, тем выше производительность и легче оптимизация архитектуры с помощью инструментов САПР.

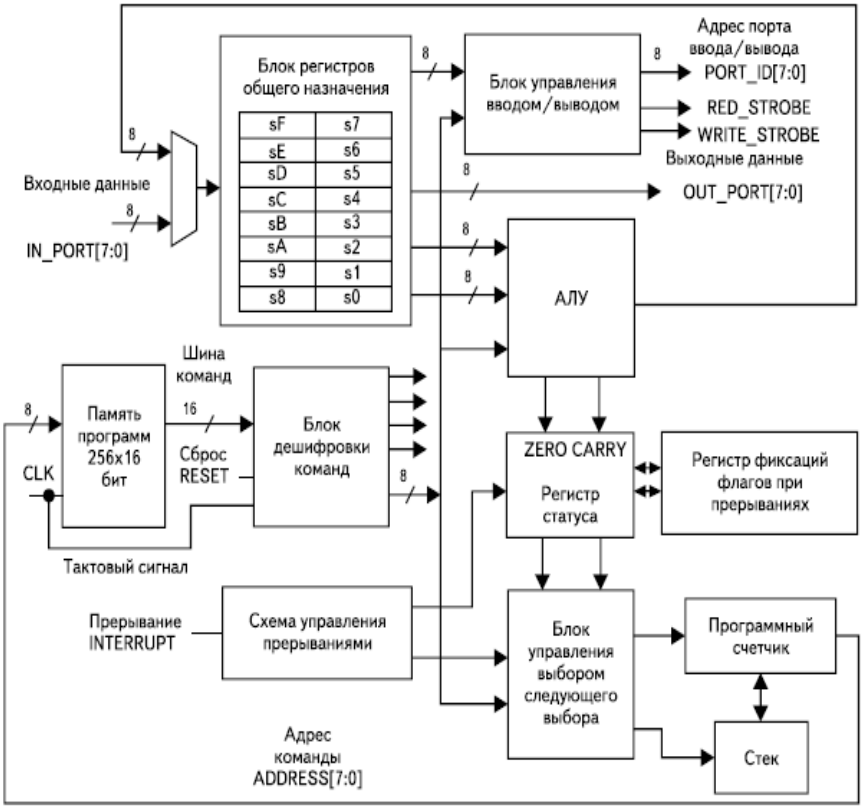


Рис.4.38. Архитектура микропроцессорного ядра PicoBlaze, для реализации в базисе ПЛИС Spartan II, Virtex

Таблица 4.5

Формат команды переходов JUMP микропроцессорного ядра PicoBlaze

Поле кода операции								Поле адреса переходов								Мнемоника	Выполняемая операция
1	0	0	0	x	x	0	1	A	A	A	A	A	A	A	A	JUMP aa	Безусловный переход
1	0	0	1	0	0	0	1	A	A	A	A	A	A	A	A	JUMP Z,aa	Переход при условии, что флаг ZERO Flag находится в установленном состоянии
1	0	0	1	0	1	0	1	A	A	A	A	A	A	A	A	JUMP NZ,aa	Переход при условии, что флаг ZERO Flag находится в сброшенном состоянии
1	0	0	1	1	0	0	1	A	A	A	A	A	A	A	A	JUMP C,aa	Переход при условии, что флаг CARRY Flag находится в установленном состоянии
1	0	0	1	1	1	0	1	A	A	A	A	A	A	A	A	JUMP NC,aa	Переход при условии, что флаг CARRY Flag находится в сброшенном состоянии

297

15	14	13	12	11	10	9	8	7	6		4	3	2	1	0	Номер разряда команды

Таблица 4.6

Формат команды переходов микропроцессорного ядра, вариант 2

Поле кода операции								Поле адреса переходов								Мне-мо-ника	Выполняемая операция
0	0	0	0	0	0	0	1	A	A	A	A	A	A	A	A	JMP	Безусловный переход по адресу, заданному младшим байтом команды
0	0	0	0	0	0	1	0	A	A	A	A	A	A	A	A	JMPZ	Переход по адресу, заданному младшим байтом команды, если содержимое регистра A равно нулю
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Номер разряда команды	

Архитектура трассировки межсоединений MultiTrack обеспечивает большую доступность ко всем окружающим LAB с помощью меньшего числа связей, что позволяет увеличить производительность, снизить энергопотребление и оптимизировать упаковку логики.

На рис.4.39 различными цветами (темно-синний и синий) показано число шагов, требующихся для соединения LAB (сноска), с окружающими LAB для реализации микропроцессорного ядра по варианту 2. Таким образом, трассировка до всех LAB выполняется за два шага.

Популярное микропроцессорное ядро oc_oc8051 (микропроцессорное ядро 8051 с сайта независимых разработчиков на интернет ресурсе OpenCores) может быть загружено в ПЛИС Stratix III EP3SL340 85 раз. При этом для реализации ядра oc_oc8051 требуется 4115 логических элементов. Общая емкость ПЛИС Stratix III EP3SL340 составляет 337.5 К логических элементов. ПЛИС Stratix III в среднем на 35 % быстрее ПЛИС Virtex-5, проекты компилируются в три раза быстрее, чем при компиляции в ПЛИС Virtex-5, а коэффициент заполнения кристалла в среднем равен 95 %. Микропроцессорное ядро PicoBlaze задействует 9 % логических ресурсов ПЛИС XC2S50E и 2.5 % ПЛИС XC2S300E.

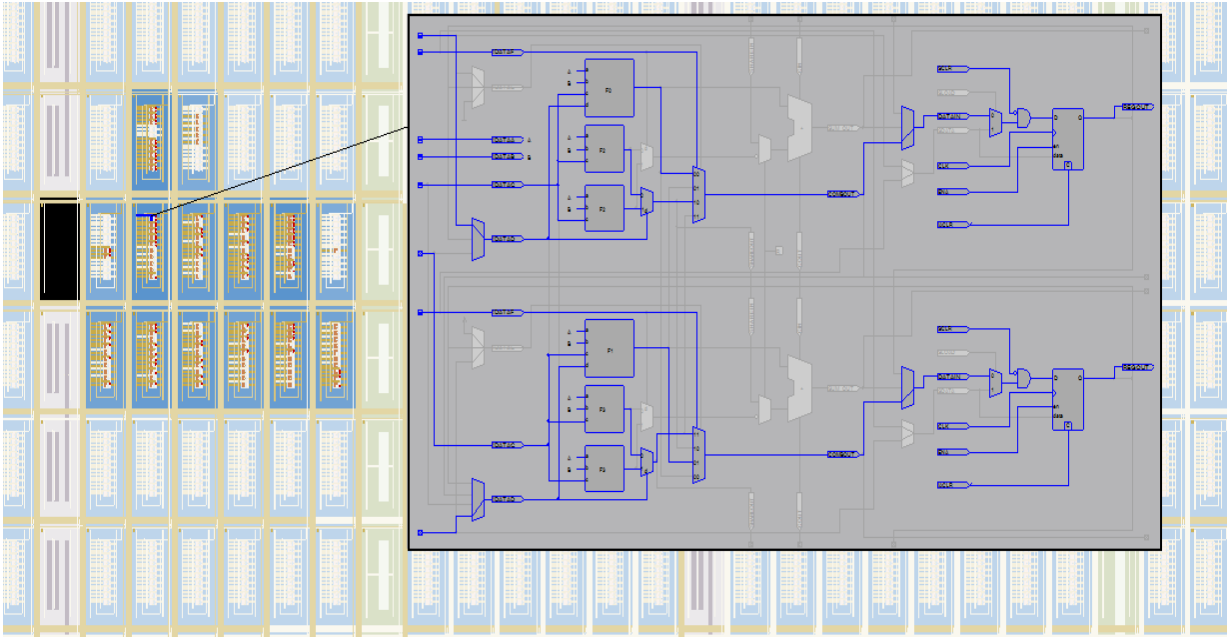


Рис.4.39. Трассировочные ресурсы ПЛИС Stratix III EP3SL50F484C2 задействованные при реализации микропроцессорного ядра по варианту 2

4.8. Проектирование микропроцессорных ядер с использованием приложения StateFlow системы Matlab/Simulink

Заменим описание управляющего автомата микропроцессорного ядра на языке М-файлов (табл.4.7) визуально-графическим автоматом (граф переходов) построенным с помощью приложения (пакет расширения) StateFlow системы Matlab/Simulink. Далее с помощью приложения Simulink HDL coder сгенерируем код языка VHDL и реализуем модель микропроцессора в базисе ПЛИС Stratix III EP3SL50F484C2 фирмы Altera. Отличительной особенностью данного типа ПЛИС семейства Stratix III является структура памяти TriMatrix с рабочими тактовыми частотами до 600 МГц, которая содержит 108 блоков памяти типа M9K (емкость блока 9216 бит), 6 блоков памяти типа M144K (емкость 147456 бит) и 950 блоков памяти MLAB (емкость 320 бит). Общий объем встроенной памяти ОЗУ 1.836 Кбит, 47.5 К логических элементов (LE), 19 К адаптивных логических модулей (ALM), которые способны работать в различных режимах и 38 К триггеров.

В табл.4.7 приведен М-файл и код управляющего автомата на языке VHDL. Из табл.4.7 видно, что по М-файлу сгенерирован асинхронный цифровой автомат с использованием двух операторов выбора CASE, представляющий из себя комбинационный дешифратор.

Процессор состоит из следующих блоков (рис.4.40): ROM - ПЗУ команд процессора (память программ); COP – блок выделения полей команды (дешифратор команд); ALU – 8-разрядное АЛУ процессора (управляющий автомат); RON – блок регистров общего назначения (8-разрядные регистры A и B); RSN – блок регистров специального назначения, 8-разрядный регистр R (стек команд) для обеспечения выполнения команд обращения к подпрограммам (CALL) и

возврата (RET) и 8-разрядный регистр Ip (для хранения значений счетчика команд). На рис.4.40 также показана тестовая программа (система команд и содержимое файла прошивки ПЗУ такое же, как и в разделе 4.2)

Для построения ПЗУ используется функциональный блок Lookup Table (таблица соответствия). Все сигналы в процессоре представлены в формате uint8 (Unsigned integer fixed-point data type, целые числа без знака в формате с фиксированной запятой, с 8-ми битной шиной) кроме команд (сигналы cmd и InCmd), они представлены в формате uint16.

Управляющий автомат микропроцессора разработан с помощью приложения StateFlow (рис.4.41). StateFlow является интерактивным инструментом разработки в области моделирования сложных, управляемых событиями систем. Он тесно интегрирован с Matlab и Simulink и основан на теории конечных автоматов. Диаграмма StateFlow - графическое представление конечного автомата, где состояния и переходы формируют базовые конструктивные блоки проектируемой системы.

StateFlow-диаграмма построена из отдельных объектов, таких как состояние, переход, переход по умолчанию и др. Состояние - условия, в которых моделируемая система пребывает некоторое время, в течение которого она ведет себя одинаковым образом. В диаграмме переходов состояния представлены прямоугольными полями со скругленными углами. Например, состояние COMM является родителем состояний MOVAB, RET, MOVBA, XCHG, ADD, SUB, END, OR, XOR, DEC. На рис.4.42 показаны временные диаграмы работы модели микропроцессорного ядра в системе Matlab/Simulink, A, B, R, IP – выходы соответствующих регистров РОН и РСН.

TestProgramm:

		DEC	HEX
0:	MOV A,1	1025	401
1:	MOV B,17	1297	511
2:	CALL 5	773	305
3:	MOV B,18	1298	512
4:	MOV A,2	1026	402
5:	MOV A,3	1027	403
6:	MOV A,4	1028	404
7:	ADD A,B	1540	604
8:	MOV A,6	1030	406
9:	MOV A,7	1031	407
10:	RET	1536	600

0	0	NOP
01xxH	256-511	JMP
02xxH	512-767	JMPZ
03xxH	768-1023	CALL
04xxH	1024-1279	MOV A,xx
05xxH	1280-1537	MOV B,xx
0600H	1536	RET
0601H	1537	MOV A,B
0602H	1538	MOV B,A
0603H	1539	XCHG A,B
0604H	1540	ADD A,B
0605H	1541	SUB A,B
0606H	1542	AND A,B
0607H	1543	OR A,B
0608H	1544	XOR A,B
0609H	1545	DEC A

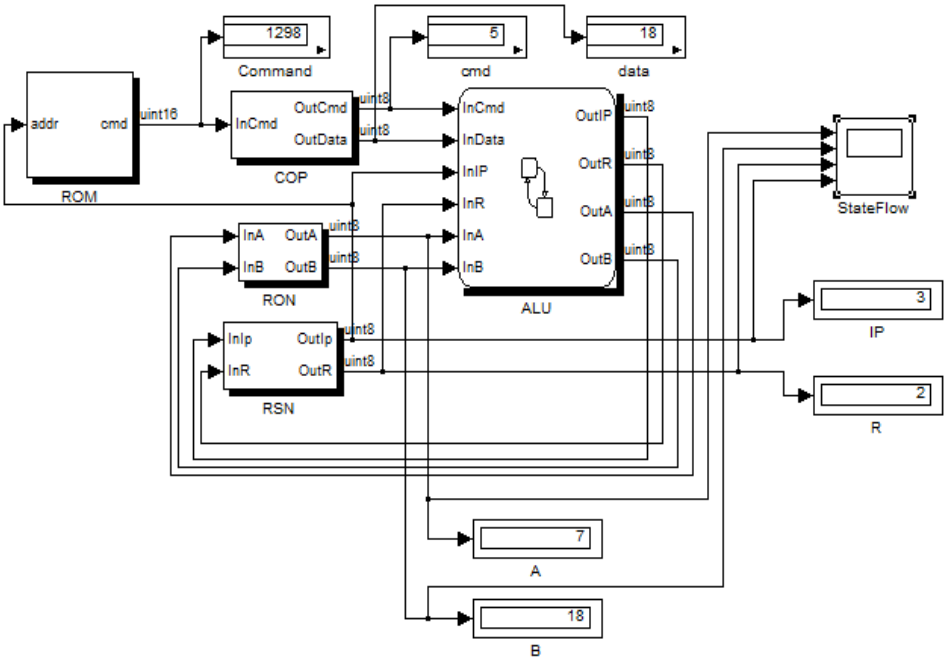


Рис.4.40. Модель микропроцессорного ядра с управляющим автоматом в системе Matlab/Simulink

Таблица 4.7

М-файл и код управляющего автомата на языке VHDL

М-файл управляющего автомата в системе Matlab/Simulink	Управляющий автомат на языке VHDL, код которого получен с помощью Simulink HDL coder
1	2

function [outA, outB, outR, outIp] = alu(inCmd,inData,inA,inB,inIp,inR)
 	
outA = inA;
outB = inB;
outR = inR;
outIp = inIp+1;
switch inCmd
%NOP
case 0
%JMP
case 1
outIp = inData;
%JMPZ
case 2
if inA == 0
outIp = inData;
end
% CALL
case 3
outR = inIp+1;
outIp = inData;
%MOV A,xx
case 4
outA = inData;
%MOV B,xx
case 5
outB = inData;
case 6
switch inData
%RET
case 0
outIp = inR;
%MOV A,B
case 1
outA = inB;
%MOV B,A
case 2
outB = inA;
%XCHG A,B
case 3
X = inB;
outB = inA;
outA = X;
	LIBRARY ieee;
	USE ieee.std_logic_1164.all;
	USE ieee.numeric_std.all;
	use ieee.std_logic_arith.all;
	use ieee.std_logic_unsigned.all;
	ENTITY ALU_entity IS
	PORT (
	inCmd : IN std_logic_vector(7 DOWNTO 0);
	inData : IN std_logic_vector(7 DOWNTO 0);
	inA : IN std_logic_vector(7 DOWNTO 0);
	inB : IN std_logic_vector(7 DOWNTO 0);
	inIp : IN std_logic_vector(7 DOWNTO 0);
	inR : IN std_logic_vector(7 DOWNTO 0);
	outA : OUT std_logic_vector(7 DOWNTO 0);
	outB : OUT std_logic_vector(7 DOWNTO 0);
	outR : OUT std_logic_vector(7 DOWNTO 0);
	outIp: OUT std_logic_vector(7 DOWNTO 0));
	END ALU_entity;
	ARCHITECTURE fsm_SFHDL OF ALU_entity IS
	BEGIN
	PROCESS (inCmd, inData, inA, inB, inIp, inR)
	BEGIN
	outA <= inA;
	outB <= inB;
	outR <= inR;
	outIp <= inIp+1;
	CASE inCmd IS
	WHEN "00000000" =>
	--NOP
	NULL;
	WHEN "00000001" =>
	--JMP
	outIp <= inData;
	WHEN "00000010" =>
	IF inA = 0 THEN
	--JMPZ
	outIp <= inData;
	END IF;
	WHEN "00000011" =>

Продолжение табл.4.7

1	2

%ADD A,B
case 4
outA = inA+inB;
%SUB A,B
case 5
outA = inA-inB;
%AND A,B
case 6
outA = bitand(inA,inB);
%OR A,B
case 7
outA = bitor(inA,inB);
%XOR A,B
case 8
outA = bitxor(inA,inB);
%DEC A
case 9
outA = inA-1;
end
end
	-- CALL
	outR <= inIp+1;
	outIp <= inData;
	WHEN "00000100" =>
	--MOV A,xx
	outA <= inData;
	WHEN "00000101" =>
	--MOV B,xx
	outB <= inData;
	WHEN "00000110" =>
	CASE inData IS
	WHEN "00000000" =>
	--RET
	outIp <= inR;
	WHEN "00000001" =>
	--MOV A,B
	outA <= inB;
	WHEN "00000010" =>
	--MOV B,A
	outB <= inA;
	WHEN "00000011" =>
	--XCHG A,B
	outB <= inA;
	outA <= inB;
	WHEN "00000100" =>
	--ADD A,B
	outA <= inA + inB;
	WHEN "00000101" =>
	--SUB A,B
	outA <= inA - inB;
	WHEN "00000110" =>
	--AND A,B
	outA <= inA AND inB;
	WHEN "00000111" =>
	--OR A,B
	outA <= inA OR inB;
	WHEN "00001000" =>
	--XOR A,B
	outA <= inA XOR inB;
	WHEN "00001001" =>
	--DEC A
	outA <= inA - 1;
	WHEN OTHERS => NULL;
	END CASE;
	WHEN OTHERS => NULL; END CASE; END PROCESS; END fsm_SFHDL;

Переход - это линия со стрелкой, соединяющая один графический объект с другим. В большинстве случаев переход представляет скачок системы из одного режима (состояния) в другой. Переход соединяет объект-источник с объектом-адресатом. Объект-источник - это место, где переход начинается, объект-адресат - это место, где переход заканчивается. Переходы по состояниям характеризуются метками. Метка может включать в себя имя события, условие, действие условия и/или действие перехода. Первоначально переходы помечаются символом (?).

Метки перехода имеют следующий основной формат: event[condition]{condition_action}/transition_action. Любая часть метки может отсутствовать. Условия - это булевы выражения, которые должны быть истинны для осуществления перехода. Условия заключаются в квадратные скобки ([]). Например, условие [InA==0] должно быть истинным для того, чтобы произошло действие условия и переход стал возможен.

Действия условий следуют за условиями и заключаются в фигурные скобки ({}). Они выполняются тогда, когда условие становится истинным, но перед тем, как переход осуществится. Если ни одно условие не определено, подразумеваемое условие принимается за истинное и действие выполняется. Если условие [InA==0] истинно, то действие {OutIP=InData} немедленно выполняется.

Линия со стрелкой и точкой на конце это безусловный переход. Безусловные переходы преимущественно используются для определения, какое последовательное состояние должно стать активным, когда есть неоднозначность между двумя или более ИЛИ-подсостояниями. Безусловные переходы имеют объект-адресат, но у них нет объекта-источника.

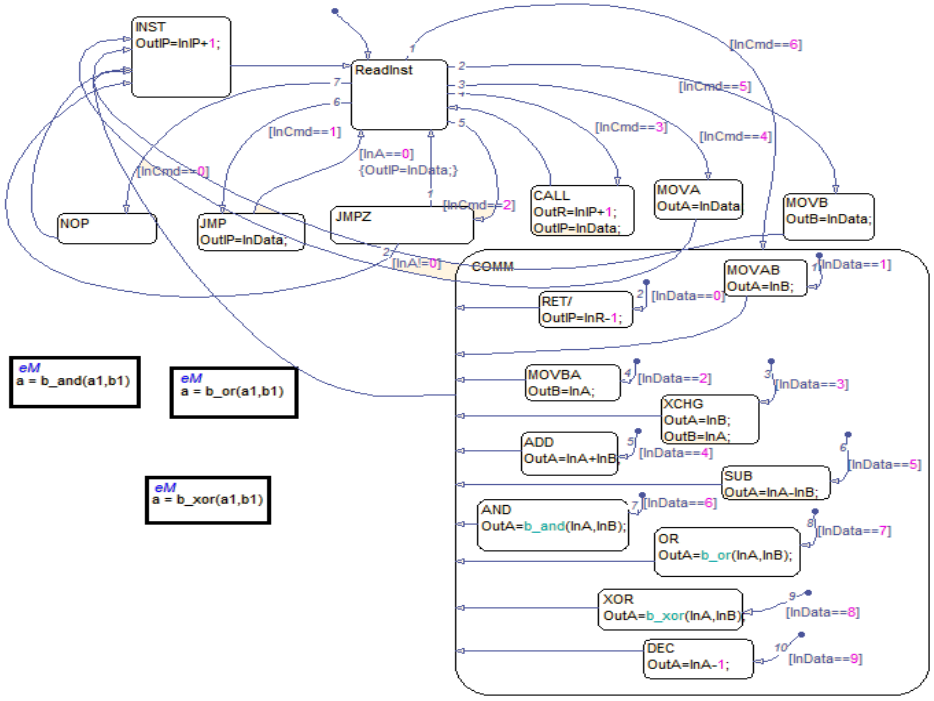


Рис.4.41. Управляющий автомат (блок АЛУ) созданный с помощью приложения StateFlow системы Matlab/Simulink в режиме отладки

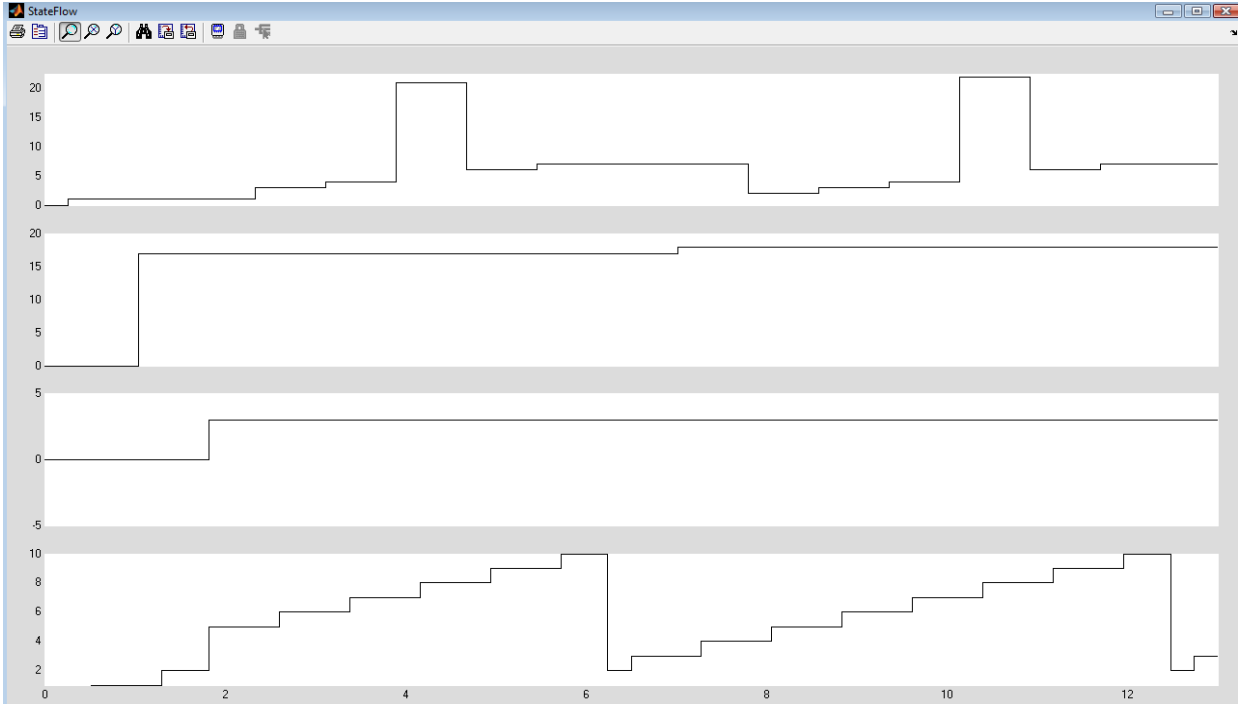


Рис.4.42. Временные диаграммы работы модели микропроцессорного ядра в системе Matlab/Simulink

На рис.4.43 показано микропроцессорное ядро с асинхронным ПЗУ в базисе ПЛИС Stratix III в САПР Quartus II, код языка которого получен с использованием Simulink HDL Coder системы Matlab/Simulink. ПЗУ, дешифратор команд – комбинационные устройства, регистры RON, RSN и ALU – синхронные последовательностные устройства. Для описания управляющего автомата микропроцессора (пример 1) используются перечислимые типы. Перечислимый тип часто используется для обозначения состояний конечных автоматов.

Анализируя код VHDL, можно сделать вывод, что сгенерирован синхронный автомат с асинхронным входом reset (активный – высокий уровень) и с синхронным сигналом разрешения тактирования clk_enable. Функциональное моделирование работы микропроцессорного ядра с асинхронным ПЗУ и синхронным управляющим автоматом в базисе ПЛИС Stratix III показано на рис.4.44.

Для сравнения, на рис.4.45 покзано функциональное моделирование работы микропроцессорного ядра с асинхронным ПЗУ и асинхронным управляющим автоматом в базисе ПЛИС Stratix III. Видно, что микропроцессорные ядра работают одинаково, но для микропроцессора с синхронным автоматом требуется большее число тактов на отработку команд и по разному выполняется команда с кодом 0600H (RET). На StateFlow- диаграмме можно по иному организовать выполнение команды вызова и возврата из подпрограммы. Для команды CALL: OutR=InIP и команды RET: OutIP=InR. Сведения по используемым ресурсам ПЛИС представлены в табл.4.8.

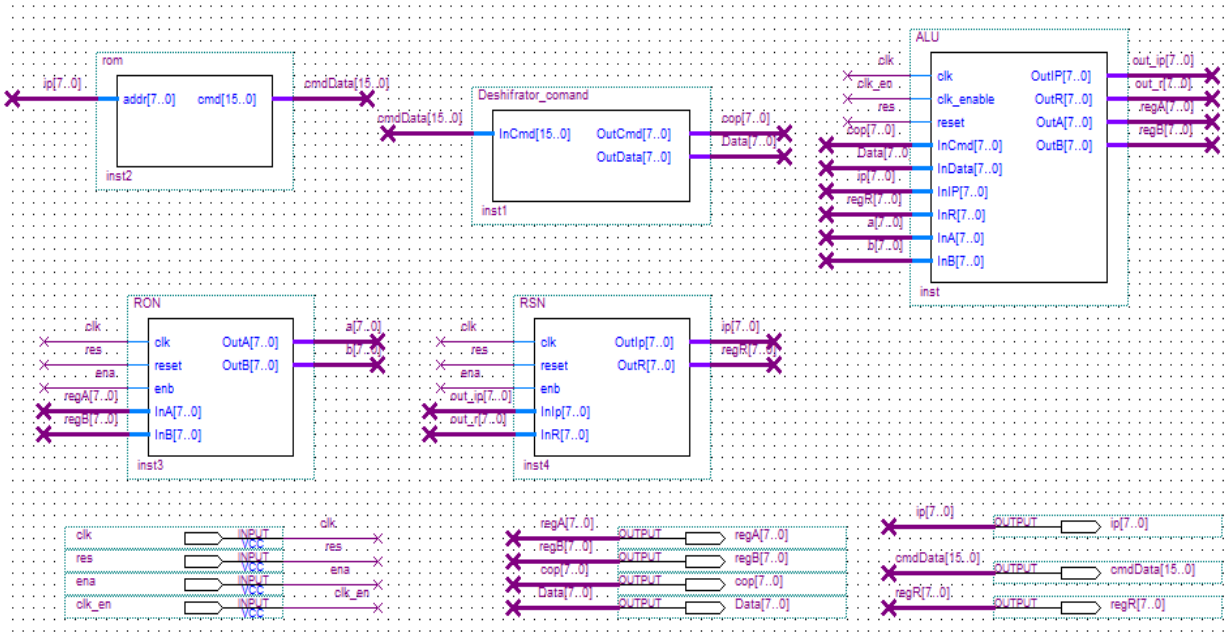


Рис.4.43. Микропроцессорное ядро с асинхронным ПЗУ и синхронным управляющим автоматом в базисе ПЛИС Stratix III в САПР ПЛИС Quartus II, код языка которого получен с использованием Simulink HDL Coder системы Matlab/Simulink

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY ALU IS
   PORT (
      clk : IN std_logic;
      clk_enable : IN std_logic;
      reset : IN std_logic;
      InCmd : IN std_logic_vector(7 DOWNTO 0);
      InData : IN std_logic_vector(7 DOWNTO 0);
      InIP : IN std_logic_vector(7 DOWNTO 0);
      InR : IN std_logic_vector(7 DOWNTO 0);
      InA : IN std_logic_vector(7 DOWNTO 0);
      InB : IN std_logic_vector(7 DOWNTO 0);
      OutIP : OUT std_logic_vector(7 DOWNTO 0);
      OutR : OUT std_logic_vector(7 DOWNTO 0);
      OutA : OUT std_logic_vector(7 DOWNTO 0);
      OutB : OUT std_logic_vector(7 DOWNTO 0));
END ALU;

ARCHITECTURE fsm_SFHDL OF ALU IS

   TYPE T_state_type_is_ALU is (IN_NO_ACTIVE_CHILD, IN_CALL,
IN_COMM, IN_INST, IN_JMP, IN_JMPZ, IN_MOVA, IN_MOVB,
IN_NOP, IN_ReadInst);
   TYPE T_state_type_is_COMM is (IN_NO_ACTIVE_CHILD,
IN_ADD, IN_AND, IN_DEC, IN_MOVAB, IN_MOVBA, IN_OR,
IN_RET, IN_SUB, IN_XCHG, IN_XOR);

   SIGNAL is_ALU : T_state_type_is_ALU;
   SIGNAL is_COMM : T_state_type_is_COMM;
   SIGNAL OutIP_reg : unsigned(7 DOWNTO 0);
   SIGNAL OutR_reg : unsigned(7 DOWNTO 0);
   SIGNAL OutA_reg : unsigned(7 DOWNTO 0);
   SIGNAL OutB_reg : unsigned(7 DOWNTO 0);
   SIGNAL is_ALU_next : T_state_type_is_ALU;
   SIGNAL is_COMM_next : T_state_type_is_COMM;
```

```
   SIGNAL OutIP_reg_next : unsigned(7 DOWNTO 0);
   SIGNAL OutR_reg_next : unsigned(7 DOWNTO 0);
   SIGNAL OutA_reg_next : unsigned(7 DOWNTO 0);
   SIGNAL OutB_reg_next : unsigned(7 DOWNTO 0);
BEGIN
   initialize_ALU : PROCESS (reset, clk)
      -- local variables
   BEGIN
      IF reset = '1' THEN
         is_COMM <= IN_NO_ACTIVE_CHILD;
         OutIP_reg <= to_unsigned(0, 8);
         OutR_reg <= to_unsigned(0, 8);
         OutA_reg <= to_unsigned(0, 8);
         OutB_reg <= to_unsigned(0, 8);
         is_ALU <= IN_ReadInst;
      ELSIF clk'EVENT AND clk= '1' THEN
         IF clk_enable= '1' THEN
            is_ALU <= is_ALU_next;
            is_COMM <= is_COMM_next;
            OutIP_reg <= OutIP_reg_next;
            OutR_reg <= OutR_reg_next;
            OutA_reg <= OutA_reg_next;
            OutB_reg <= OutB_reg_next;
         END IF;
      END IF;
   END PROCESS initialize_ALU;
   ALU : PROCESS (is_ALU, is_COMM, OutIP_reg, OutR_reg,
OutA_reg, OutB_reg, InCmd, InData, InIP, InR, InA, InB)
      -- local variables
   BEGIN
      is_ALU_next <= is_ALU;
      is_COMM_next <= is_COMM;
      OutIP_reg_next <= OutIP_reg;
      OutR_reg_next <= OutR_reg;
      OutA_reg_next <= OutA_reg;
      OutB_reg_next <= OutB_reg;
      CASE is_ALU IS
         WHEN IN_CALL =>
```

```
   is_ALU_next <= IN_ReadInst;
WHEN IN_COMM =>
   is_COMM_next <= IN_NO_ACTIVE_CHILD;
   is_ALU_next <= IN_INST;
   OutIP_reg_next <= unsigned(InIP) + 1;
WHEN IN_INST =>
   is_ALU_next <= IN_ReadInst;
WHEN IN_JMP =>
   is_ALU_next <= IN_ReadInst;
WHEN IN_JMPZ =>
   IF unsigned(InA) = 0 THEN
      OutIP_reg_next <= unsigned(InData);
      is_ALU_next <= IN_ReadInst;
   ELSIF unsigned(InA) /= 0 THEN
      is_ALU_next <= IN_INST;
      OutIP_reg_next <= unsigned(InIP) + 1;
   END IF;
WHEN IN_MOVA =>
   is_ALU_next <= IN_INST;
   OutIP_reg_next <= unsigned(InIP) + 1;
WHEN IN_MOVB =>
   is_ALU_next <= IN_INST;
   OutIP_reg_next <= unsigned(InIP) + 1;
WHEN IN_NOP =>
   is_ALU_next <= IN_INST;
   OutIP_reg_next <= unsigned(InIP) + 1;
WHEN IN_ReadInst =>
   IF unsigned(InCmd) = 6 THEN
      is_ALU_next <= IN_COMM;
      IF unsigned(InData) = 1 THEN
         is_COMM_next <= IN_MOVAB;
         OutA_reg_next <= unsigned(InB);
      ELSIF unsigned(InData) = 0 THEN
         is_COMM_next <= IN_RET;
         OutIP_reg_next <= unsigned(InR) - 1;
      ELSIF unsigned(InData) = 3 THEN
         is_COMM_next <= IN_XCHG;
         OutA_reg_next <= unsigned(InB);
```

```
        OutB_reg_next <= unsigned(InA);
      ELSIF unsigned(InData) = 2 THEN
        is_COMM_next <= IN_MOVBA;
        OutB_reg_next <= unsigned(InA);
      ELSIF unsigned(InData) = 4 THEN
        is_COMM_next <= IN_ADD;
        OutA_reg_next <= unsigned(InA) + unsigned(InB);
      ELSIF unsigned(InData) = 5 THEN
        is_COMM_next <= IN_SUB;
        OutA_reg_next <= unsigned(InA) - unsigned(InB);
      ELSIF unsigned(InData) = 6 THEN
        is_COMM_next <= IN_AND;
        OutA_reg_next <= unsigned(InA AND InB);
      ELSIF unsigned(InData) = 7 THEN
        is_COMM_next <= IN_OR;
        OutA_reg_next <= unsigned(InA OR InB);
      ELSIF unsigned(InData) = 8 THEN
        is_COMM_next <= IN_XOR;
        OutA_reg_next <= unsigned(InA XOR InB);
      ELSIF unsigned(InData) = 9 THEN
        is_COMM_next <= IN_DEC;
        OutA_reg_next <= unsigned(InA) - 1;
      END IF;
    ELSIF unsigned(InCmd) = 5 THEN
      is_ALU_next <= IN_MOVB;
      OutB_reg_next <= unsigned(InData);
    ELSIF unsigned(InCmd) = 4 THEN
      is_ALU_next <= IN_MOVA;
      OutA_reg_next <= unsigned(InData);
    ELSIF unsigned(InCmd) = 3 THEN
      is_ALU_next <= IN_CALL;
      OutR_reg_next <= unsigned(InIP) + 1;
      OutIP_reg_next <= unsigned(InData);
    ELSIF unsigned(InCmd) = 2 THEN
      is_ALU_next <= IN_JMPZ;
    ELSIF unsigned(InCmd) = 1 THEN
      is_ALU_next <= IN_JMP;
      OutIP_reg_next <= unsigned(InData);
```

```
            ELSIF unsigned(InCmd) = 0 THEN
               is_ALU_next <= IN_NOP;
            END IF;
         WHEN OTHERS =>
            is_ALU_next <= IN_ReadInst;
      END CASE;
   END PROCESS ALU;
   OutIP <= std_logic_vector(OutIP_reg_next);
   OutR <= std_logic_vector(OutR_reg_next);
   OutA <= std_logic_vector(OutA_reg_next);
   OutB <= std_logic_vector(OutB_reg_next);
END fsm_SFHDL;
```

Пример 1. Код языка VHDL управляющего автомата модели микропроцессорного ядра на StateFlow, полученный с использованием Simulink HDL Coder системы Matlab/Simulink

Приложение Simulink HDL coder для разработанного управляющего автомата микропроцессора, построенного с помощью StateFlow системы Matlab/Simulink, позволяет сгенерировать код языка VHDL синхронного автомата.

Система визуально-имитационного моделирования Matlab/Simulink с приложениями StateFlow и Simulink HDL Coder может быть эффективно использована для ускорения процесса разработки моделей микропроцессорных ядер.

Проект микропроцессора с асинхронным ПЗУ на языке VHDL может быть успешно размещен в ПЛИС Stratix III EP3SL50F484C2, при этом общее число задействованных логических ресурсов составляет менее 1 %.

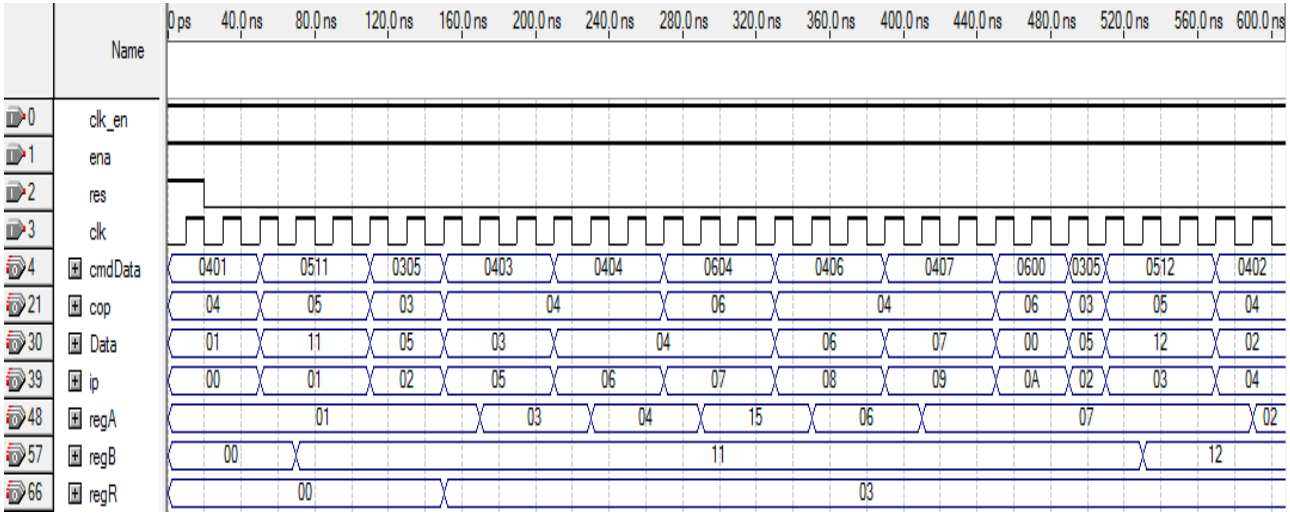


Рис.4.44. Функциональное моделирование работы микропроцессорного ядра с асинхронным ПЗУ и синхронным управляющим автоматом в базисе ПЛИС Stratix III

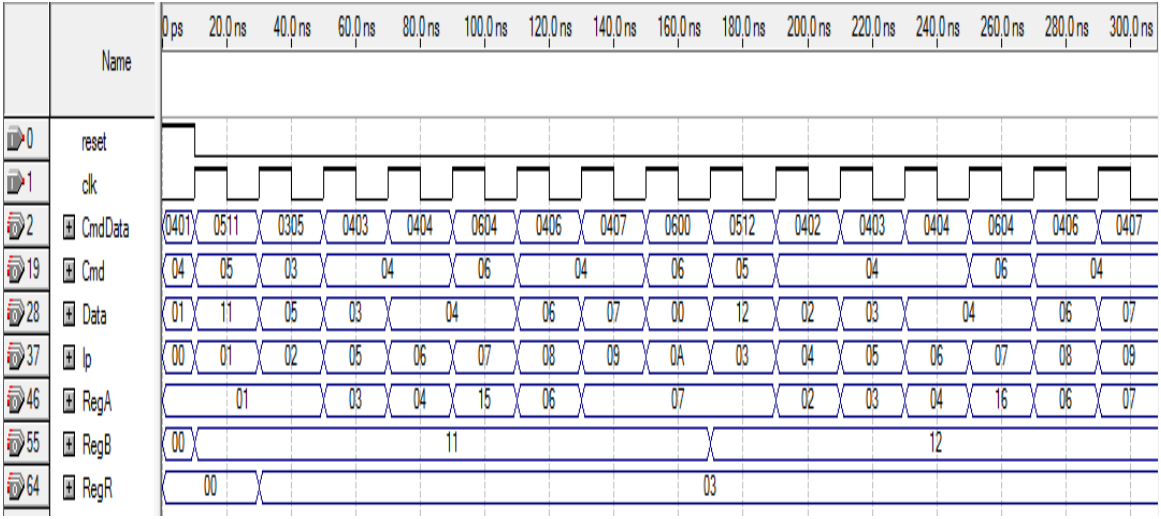


Рис.4.45. Функциональное моделирование работы микропроцессорного ядра с асинхронным ПЗУ и асинхронным управляющим автоматом в базисе ПЛИС Stratix III

Таблица 4.8

Используемы ресурсы ПЛИС Stratix III EP3SL50F484C2

Проект	Логические ресурсы		Максимальная тактовая частота синхро-сигнала, $f_{\max CLK}$, МГц
	Адаптивные таблицы перекодировок ALUTs, для реализации комбинационных функций	Выделенные триггеры логических элементов	
Асинхронное ПЗУ, асинхронный управляющий автомат (язык М-файлов)	82	26	360
Асинхронное ПЗУ, синхронный управляющий автомат (StateFlow)	103	59	275

В данной главе показаны основы проектирования микропроцессорных ядер для реализации в базисе ПЛИС с использованием системного уровня проектирования с привлечением системы визуально-имитационного моделирования аналоговых и дискретных систем Matlab/Simulink. Рассматриваются различные подходы в проектировании управляющего автомата микропроцессорного ядра: на языке VHDL, с использованием графического представления конечного автомата с помощью приложений StateFlow и Simulink HDL coder, с использованием языка М-файлов системы Matlab/Simulink.

ЗАКЛЮЧЕНИЕ

В учебном пособии на обширном иллюстративном материале показаны методы проектирования преобразователей кодов которые могут быть выполнены как на ИС средней степени интеграции, так и в базисе ПЛИС с учетом или без их архитектурных особенностей с использованием высокоуровневых языков описания аппаратных средств.

Изложены основы проектирования цифровых фильтров с конечно-импульсной характеристикой как с использованием метода умножения с накоплением характерным для процессоров цифровой обработки сигналов, так и с использованием теории распределенной арифметики для реализации в базисе ПЛИС.

Рассмотрен алгоритм реализации умножения методом правого сдвига и сложения с накоплением. Даются практические примеры проектирования КИХ-фильтров на последовательной распределенной арифметике в САПР ПЛИС Quartus II компании Altera.

При разработке конечных автоматов в базисе ПЛИС на языке VHDL наиболее эффективным решением является использование неявного стиля кодирования или явного с применением атрибута syn_encoding, поручая компилятору-синтезатору САПР Quartus II минимизацию логических ресурсов. Метод кодирования с одним активным состоянием применительно к ПЛИС дает возможность строить конечные автоматы, которые в общем случае требуют меньших ресурсов и отличаются более высокими скоростными показателями, чем аналогичные конечные автоматы с двоичным кодированием состояний. Уделено внимание использованию цифровых автоматов в технологии периферийного сканирования ИС.

Рассмотрены различные подходы в проектировании управляющего автомата микропроцессорного ядра: на языке VHDL, с использованием графического представления конечного автомата с помощью приложений StateFlow и Simulink HDL coder, с использованием M-файлов системы визуально-имитационного моделирования Matlab/Simulink.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Уилкинсон Б. Основы проектирования цифровых схем: пер. с англ. / Б. Уилкинсон. М.: Издательский дом Вильямс, 2004. 320 с.
2. Армстронг Дж. Р. Моделирование цифровых систем на языке VHDL: пер. с англ. / Р. Дж. Армстронг. М.: Мир, 1992. 348 с.
3. Максфилд К. Проектирование на ПЛИС: курс молодого бойца: пер. с англ. / К. Максфилд. М.: Издательский дом Додэка XXI, 2007. 408 с.
4. Джон Ф. Уэйкерли. Проектирование цифровых устройств: пер. с англ. / Уэйкерли Ф. Джон. М.: Постмаркет, 2002. 533 с.
5. Рабаи Ж.М. Цифровые интегральные схемы. Методология проектирования. / Ж.М. Рабаи, А. Чандракасан, Б. Николич. М.: Вильямс, 2007. - 911 с.
6. Угрюмов Е.П. Цифровая схемотехника / Е.П. Угрюмов. СПб.: БХВ, 2004. 528 с.
7. Стешенко В. ПЛИС фирмы ALTERA: проектирование устройств обработки сигналов / В. Стешенко. М.: Додэка, 2000. 457 с.
8. Строгонов А.В. Преобразователи кодов на ПЛИС / А.В. Строгонов, А.А. Винокуров // Компоненты и технологии. 2012. N12. С.16-22.
9. Строгонов А.В. Проектирование цифровых фильтров в системе Matlab/Simulink и САПР ПЛИС Quartus / А.В. Строгонов // Компоненты и технологии. 2008. N6. С.32-36.
10. Строгонов А.В. КИХ-фильтр на распределенной арифметике: проектируем сами / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. 2013. N3. С.131-138.
11. Строгонов А.В. КИХ-фильтры на параллельной распределенной арифметике: проектируем сами / А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. 2013. N3. С.44-48.
12. Строгонов А.В. Эффективность разработки конечных автоматов в базисе ПЛИС FPGA/ А.В. Строгонов, А.В. Быстрицкий // Компоненты и технологии. 2013. N1. С.66-72.

13. Строгонов А.В. Проектирование сложно-функциональных блоков в базисе ПЛИС: учеб. пособие / А.В. Строгонов, С.А. Цыбин. Воронеж: ГОУВПО “Воронежский государственный технический университет”, 2010. 333 с.

14. Строгонов А.В. Проектирование учебного процессора для реализации в базисе ПЛИС / А.В. Строгонов // Компоненты и технологии. 2009. N3. С.6-9.

15. Строгонов А.В. Проектирование учебного процессора для реализации в базисе ПЛИС с использованием системы Matlab/Simulink / А.В. Строгонов, А.И. Буслов // Компоненты и технологии. 2009. N5. С.10-14.

16. Строгонов А.В. Проектирование микропроцессорных ядер с конвейерной архитектурой для реализации в базисе ПЛИС фирмы Altera / А.В. Строгонов, С.И. Давыдов // Компоненты и технологии. 2009. N8. С.76-79.

17. Строгонов А.В. Проектирование учебного процессора с фиксированной запятой в системе Matlab/Simulink / А.В. Строгонов // Компоненты и технологии. 2009. N7. С.22-27.

18. Строгонов А.В. Проектирование учебного процессора с фиксированной запятой в САПР Quartus II компании Altera / А.В. Строгонов, А.И. Буслов, С.И. Давыдов // Компоненты и технологии. 2009. N11. С.20-25.

19. Строгонов А.В. Использование различных типов памяти при проектировании учебного микропроцессорного ядра для реализации в базисе ПЛИС / А.В. Строгонов, С.А. Цыбин // Компоненты и технологии. 2009. N12. С.92-96.

20. Строгонов А.В. Проектирование микропроцессорных ядер с использованием приложения StateFlow системы Matlab/Simulink / А.В. Строгонов, С.А. Цыбин, А.И. Буслов // Компоненты и технологии. 2010. N1. С.66-70.

21. Строгонов А.В. Использование ресурсов ПЛИС Stratix III фирмы Altera при проектировании микропроцессорных ядер / А.В. Строгонов, С.А. Цыбин // Компоненты и технологии. 2010. N2. С.70-73.

ОГЛАВЛЕНИЕ

Учебное издание

Строгонов Андрей Владимирович

ПРОЕКТИРОВАНИЕ УСТРОЙСТВ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ ДЛЯ РЕАЛИЗАЦИИ В БАЗИСЕ ПРОГРАММИРУЕМЫХ ЛОГИЧЕСКИХ ИНТЕГРАЛЬНЫХ СХЕМ

В авторской редакции

Компьютерная верстка А.В. Строгонова

Подписано к изданию 24.05.2013

Объём данных 45 МБ

ФГБОУ ВПО «Воронежский государственный технический университет»
394026 Воронеж, Московский просп., 14