- 4 -
Функциональные модели устройств на языке Verilog на основе потоков данных и поведенческих конструкций

4.1. Синтез Verilog-моделей цифровых устройств на уровне потоков данных

Язык Verilog позволяет строить функциональные модели устройств, как уже отмечалось в главе 1, на четырех уровнях абстракции. Уровень потоков данных является для разработчика компромиссом между сложностью описания устройства и сложностью логического синтеза такого устройства программными средствами. Читателям, не знакомым с теорией графов потоков данных, рекомендуется обратиться к работам [3; 16].

В данном разделе основное внимание уделяется средствам языка Verilog, позволяющим формировать эффективные модели на уровне потоков данных. К таким средствам, в первую очередь, относится непрерывный оператор присваивания.

4.1.1. Непрерывный оператор присваивания

Синтаксис непрерывного оператора присваивания можно представить следующим образом:

assign *Приоритет #Задержка Имя_Сигнала = Выражение*

где **assign** – ключевое слово, определяющее тип оператора; *Приоритет* – необязательный параметр, позволяющий разрешать кон-

фликты при наложении сигналов; его возможные значения описаны в разделе 2.6; *Задержка* – заданная в условных единицах (у.е.) величина модельного времени, определяющая время выполнения оператора присваивания; *Имя_Сигнала* – идентификатор изменяемого сигнала; *Выражение* – новое значение изменяемого сигнала, представляющее собой арифметическое либо логическое выражение и состоящее из сигналов, констант, операций и обращений к функциям.

Непрерывный оператор присваивания получил такое название, поскольку его задача состоит в постоянном отслеживании изменений в сигналах, входящих в *Выражение,* с последующим обновлением значения изменяемого сигнала. Непрерывный оператор присваивания обладает следующими свойствами:

- левая часть оператора присваивания может быть одиночным сигналом, вектором, элементом массива либо результатом конкатенации любых из перечисленных вариантов;
- непрерывное присваивание всегда активно, его нельзя временно отключать, в случае возникновения событий на любом входящем в *Выражение* сигнале изменяемый сигнал также поменяет свое значение.

Рассмотрим пример использования оператора **assign** для построения функциональной модели устройства, осуществляющего вычисление следующей полиномиальной функции 3-х переменных:

$$f(a, b, c) = 2a^2 + 3ab - bc^2 = 4c.$$

Разработку устройства будем осуществлять исходя из предположения, что входные сигналы a, b, c представляют собой восьмиразрядные беззнаковые целые числа. Текст программы приведен ниже:

```
`timescale 10 ns / 1ps

module Assign_Demo (a ,b ,c ,f);

input [7:0] a ;
wire [7:0] a ;
input [7:0] b ;
wire [7:0] b ;
input [7:0] c ;
```

```
wire [7:0] c ;

output [15:0] f ;
wire [15:0] f ;

wire [15:0] s1,s2,s3,s4;

assign s1 = 2*a*a;
assign s2 = 3*a*b;
assign s3 = b*c*c;
assign s4 = 4*c;
assign f = s1 + s2 – s3 + s4;

endmodule
```

Промоделируем работу описанного устройства для начальных значений $a = 11$, $b = 28$, $c = 4$, если в момент модельного времени 40 *ns* сигнал *c* изменяет свое значение на 5. Временная диаграмма представлена на рис. 4.1.

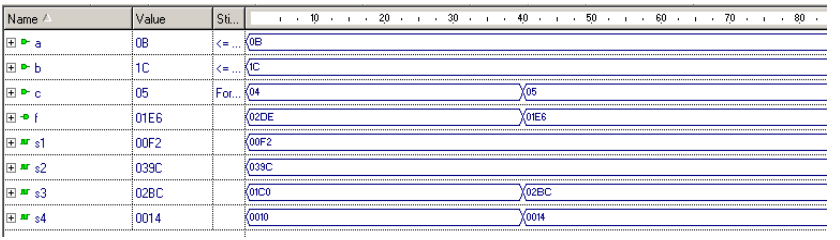

Рис. 4.1. Моделирование устройства, описанного на уровне потоков данных для функциональной модели

Анализ результатов моделирования (рис. 4.1) показывает, что изменение значений выхода устройства и его внутренних сигналов происходит мгновенно после изменения значений входных сигналов, поскольку данная модель не содержит временных задержек. Заметно, как изменение сигнала *c* в момент модельного времени 40 *ns* привело к изменению зависящих от него внутренних сигналов *s*3, *s*4 и выходного сигнала устройства *f*.

4.1.2. Введение временных задержек в непрерывные операторы присваивания

В приведенной выше модели (раздел 4.1.1) все вычисления для реализации функциональной зависимости происходят мгновенно. В то же время данная модель является не адекватной реальным процессам. Если предположить, что операция сложения/вычитания выполняется микросхемой за 13.3 *ns*, операция умножения выполняется за 40 *ns*, а время передачи данных между ячейками намного меньше и им можно пренебречь, то преобразованный текст Verilog-программы будет отражать также и временные аспекты работы описываемого устройства:

```
`timescale 10 ns / 1ps

module Assign_Demo ( a ,b ,c ,f );

input [7:0] a ;
wire [7:0] a ;
input [7:0] b ;
wire [7:0] b ;
input [7:0] c ;
wire [7:0] c ;

output [15:0] f ;
wire [15:0] f ;

wire [15:0] s1,s2,s3,s4;

assign #8 s1 = 2*a*a;
                        //2 операции умножения длительностью
                        // по 4 модельных шага (10 ns)
assign #8 s2 = 3*a*b;
                        //2 операции умножения по 4 модельных шага
assign #8 s3 = b*c*c;
                        //2 операции умножения по 4 модельных шага
assign #4 s4 = 4*c;
                        //1 операция умножения по 4 модельных шага

//3 операции сложения/вычитания по 1.33 модельных шага
assign #4 f = s1 + s2 – s3 + s4;
endmodule
```

Результаты моделирования преобразованной Verilog-программы при тех же задающих воздействиях, что и на рис. 4.1 (за исключением времени изменения сигнала *c*, равном 200 *ns*), показаны на рис. 4.2.

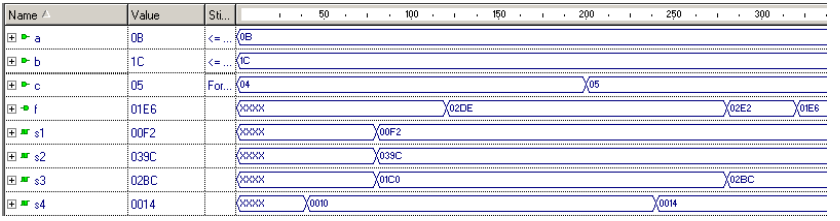

Рис. 4.2. Модель потоков данных с учетом длительности вычислений

Сравнивая временные диаграммы на рис. 4.1 и 4.2, можно увидеть, что промежуток времени между изменением значений входных сигналов и установкой правильного выходного значения 02DE составляет 0 *ns* (рис. 4.1) и 120 *ns* (рис. 4.2). То же самое время вычислений 120 *ns* получено на основе рассчета времени выполнения операций по исходному графу потоков данных (с учетом параллельной работы операторов присваивания), а значит, модель является адекватной реальным процессам.

Другой способ задания задержек при формировании сигнала состоит в их описании в операторе объявления сигнала **`wire`**:

wire #10 *f*;
assign *f* = *s1* + *s2* – *s3* + *s4*;

Представленная форма может быть более предпочтительной в случаях, когда конкретное значение присваивается сигналу не одним, а многими операторами **assign** с одинаковой задержкой. Если в таком случае задержка задана в операторах присваивания, то для ее изменения нужно внести столько исправлений, сколько присваиваний существует в программе. Если же задержка задана при объявлении сигнала, то достаточно внести только одно изменение.

4.1.3. Сокращенная форма записи непрерывного оператора присваивания

Для сокращения длины исходных текстов программ допускается совмещение оператора непрерывного присваивания с оператором объявления сигнала (например, **wire**). При этом ключевое слово **assign** опускается, например:

wire[Индекс**]***Приоритет #Задержка Имя_Сигнала=Выражение*;

где Индекс – необязательный параметр, определяющий диапазон изменения индексов элементов в векторе (см. раздел 2.6.1).

С использованием сокращенной записи предыдущий фрагмент (первый пример из раздела 4.1.2) можно представить следующим образом:

```
`timescale 10 ns / 1ps

module Assign_Demo ( a ,b ,c ,f );

input [7:0] a ;
wire [7:0] a ;
input [7:0] b ;
wire [7:0] b ;
input [7:0] c ;
wire [7:0] c ;

output [15:0] f ;

wire [15:0] #8 s1 = 2*a*a;
wire [15:0] #8 s2 = 3*a*b;
wire [15:0] #8 s3 = b*c*c;
wire [15:0] #4 s4 = 4*c;
wire [15:0] #4 f = s1 + s2 – s3 + s4;

endmodule
```

Последние два фрагмента программ являются абсолютно адекватными и при любых условиях дадут одни и те же результаты моделирования.

4.1.4. Выражения и операнды

Моделирование устройств на уровне потоков данных предполагает широкое применение арифметических и логических операций для формирования выражений, реализующих соответствующий разрабатываемому устройству вычислительный алгоритм.

Выражением называется комбинация операций и операндов, формирующая значение определенного типа. Примеры выражений приведены ниже:

```
a * b
{carry, X1} + X2
Reg01[7:0] – Reg02[10:3]
```

В качестве операндов в выражениях могут участвовать сигналы, константы и результаты обращений к функциям:

```
2 * DATA + $random    //    Выражение
                      //    2 – константа, DATA – сигнал,
                      //    $random – вызов функции
```

Сигналы и результаты обращений к функциям должны принадлежать к одному из типов сигналов, перечисленных в разделе 2.6. Кроме того, можно использовать фрагменты и результаты конкатенации элементов указанных типов, а также массивы, состоящие из таких элементов.

4.1.5. Операции

Язык Verilog содержит 9 типов операций: арифметические, логические, отношения, равенства, побитовые, свертки, сдвига, конкатенации и условные. По числу операндов операции делятся на унарные, бинарные, тернарные и многооперандные. В табл. 4.1 приведен полный список операций языка Verilog с указанием числа их операндов.

Арифметические операции производятся в соответствии со своими литералами.

Результат операции вычисления остатка от деления «%» принимает знак, совпадающий со знаком первого операнда:

Таблица 4.1

Операции в языке Verilog

№	Типы операций	Литерал	Описание	Число операндов
1	Арифметические	-	Унарный минус	1
2		*	Умножение	2
3		/	Деление	2
4		+	Сложение	2
5		-	Вычитание	2
6		%	Остаток от деления	2
7	Логические	!	Логическое отрицание	1
8		&&	Логическое И	2
9		||	Логическое ИЛИ	2
10	Отношения	>	Больше	2
11		<	Меньше	2
12		>=	Больше или равно	2
13		<=	Меньше или равно	2
14	Равенства	==	Равно	2
15		!=	Не равно	2
16		===	Побитовое равно	2
17		!==	Побитовое не равно	2
18	Побитовые	~	Побитовое НЕ (not)	1
19		&	Побитовое И (and)	2
20		|	Побитовое ИЛИ (or)	2
21		^	Побитовое исключающее ИЛИ (xor)	2
22		^~ или ~^	Побитовое исключающее ИЛИ-НЕ (xnor)	2
23	Свертки	&	Свертывающее И (and)	1
24		~&	Свертывающее И-НЕ (nand)	1
25		|	Свертывающее ИЛИ (or)	1
26		~|	Свертывающее ИЛИ-НЕ (nor)	1
27		^	Свертывающее исключающее ИЛИ (xor)	1
28		^~ или ~^	Свертывающее искл. ИЛИ-НЕ (xnor)	1
29	Сдвига	>>	Сдвиг вправо	2
30		<<	Сдвиг влево	2
31	Конкатенации	{ }	Конкатенация	Произвольное
32		{ { } }	Повторение	
33	Условная	?:	Условная	3

```
5 % -3 == 2
-5 % 3 == -2
```

Следует также отметить, что для записи отрицательных целых чисел (типа **integer**) не следует пользоваться формой с явным указанием числа разрядов (например, –32'd18), так как компилятором такое число будет преобразовано в дополнительный код, что приведет к некорректному результату вычисления. К правильному результату приведет запись –18.

Операндами **логических операций** могут служить как одноразрядные, так и многоразрядные величины. При этом соблюдаются следующие правила:

- результат логической операции всегда имеет длину в 1 бит, независимо от длины операндов, и может принимать значения 0, 1 или x;
- если операнд не равен 0, то он воспринимается как логическая единица (истина), равный нулю операнд воспринимается как логический ноль (ложь), а если какой-либо операнд имеет хотя бы один разряд, равный z или x, то результат будет представлен значением x (неопределенный);
- операндами логических операций могут служить выражения и сигналы.

Операции сравнения выполняются по схожему принципу. Они могут в качестве результата возвращать одно из трех значений: 0, 1 или x, причем результат равен x (неизвестному), если любой операнд имеет хотя бы один разряд, равный z или x. Так же работают и простые **операции равенства** (== – равно и != – не равно). Побитовые операции равенства === и !== рассматривают z и x как обычные символы и обрабатывают их стандартным образом, т.е.

```
4'b101x == 4'b101x // Возвращает x
4'b101x == 4'b101z // Возвращает x
4'b101x === 4'b101x // Возвращает 1
4'b101x === 4'b101z // Возвращает 0
```

Как видно из примера, побитовые операции равенства никогда не возвращают значение x.

Результат выполнения **побитовых операций** имеет такое же чис-

ло разрядов, как у наиболее длинного из операндов. Соответствующая операция применяется к каждому биту операндов поочередно, более короткий операнд дополняется слева нулями. Таблицы истинности для побитовых операций приводятся в табл. 4.2 – 4.6.

Во всех побитовых операциях сигнал `z` интерпретируется аналогично с сигналом `x`.

Операции свертки относятся к унарным операциям и, соответственно, имеют один многоразрядный операнд (одноразрядные операнды в результате возвращаются операциями свертки без изменений). Операцию свертки можно интерпретировать как соответствующую ей побитовую операцию, применяемую ко всем битам операнда:

// Выражения, содержащие операцию свертки:

Таблица 4.2
Таблица истинности для операции «Побитовое И»

Побитовое И (and)	0	1	x
0	0	0	x
1	0	1	x
x	x	x	x

Таблица 4.3
Таблица истинности для операции «Побитовое ИЛИ»

Побитовое ИЛИ (or)	0	1	x
0	0	1	x
1	1	1	x
x	x	x	x

Таблица 4.4
Таблица истинности для операции «Побитовое исключающее ИЛИ»

Побитовое исключающее ИЛИ (xor)	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

Таблица 4.5
Таблица истинности для операции «Побитовое исключающее ИЛИ-НЕ»

Побитовое исключающее ИЛИ-НЕ (xnor)	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Таблица 4.6
Таблица истинности для операции «Побитовое отрицание»

Побитовое отрицание (not)	Результат
0	1
1	0
x	x

```
^8'b10101101
|8'b10101101
&8'b10101101

// Адекватные выражения, реализованные побитовыми
// операциями:
1^0^1^0^1^1^0^1
1|0|1|0|1|1|0|1
1&0&1&0&1&1&0&1
```

Заметим, что операция свертки вычисляется в направлении справа налево или, другими словами, от менее значащих разрядов к более значащим.

Операции сдвига влево и вправо являются бинарными, при этом в качестве 1-го операнда подставляется сдвигаемый вектор, а в качестве второго – количество сдвигаемых разрядов. Выталкиваемые разряды теряются, а недостающие – заполняются нулями, например:

```
8'b10101101 << 4 // Дает результат 8'b11010000
8'b10101101 >> 2 // Дает результат 8'b00101011
```

Операции сдвига очень удобны в программировании, так как сдвиг двоичного числа вправо на n разрядов соответствует делению этого числа на 2^n, а сдвиг влево – соответствует умножению на 2^n.

Операция конкатенации позволяет объединять произвольное количество отдельных векторов в один общий вектор, который в любых выражениях будет рассматриваться как единый вектор, длина которого равна сумме длин составляющих векторов. Операция конкатенации задается при помощи фигурных скобок:

```
{Вектор_1, Вектор_2, Вектор_3, … , Вектор_N }
```

Одно из очевидных применений операции конкатенации – это формирование бита переноса при сложении/вычитании сигналов:

```
. . .
```

```
// X1, X2 – Слагаемые
reg [7:0] X1;
reg [7:0] X2;

// Y – Результат сложения
reg [7:0] Y;

// carry – Бит переноса
wire carry;

assign {carry, Y} = X1 + X2;

. . .
```

Читателям, которые знакомы с языком Си или Паскаль, предлагается самостоятельно решить ту же задачу на любом из указанных языков и убедиться в значительно более высокой сложности полученной программы по сравнению с приведенным выше фрагментом Verilog-кода.

Операция повторения является расширенной версией операции конкатенации и позволяет сократить запись в случаях, когда один и тот же простой вектор включается в сложный несколько раз. Например:

```
wire [1:0] k,e;
wire [7:0] g;
wire [5:0] h;

assign k = 2'b11;
assign e = 2'b00;

// В операции конкатенации 4 раза повторяется сигнал k
assign g = { 4{k} }; // аналогичная запись {k, k, k, k}

// В операции конкатенации 3 раза повторяется сигнал e
assign h = { 3{e} }; // аналогичная запись {e, e, e}
```

В результате выполнения приведенного выше фрагмента сигналы

g и h получат значения 8'b11111111 и 6'b000000, соответственно.

Последний тип рассматриваемых операций – это **условная операция**. При помощи условной операции можно реализовывать простые разветвляющиеся алгоритмы, вычисляющие функции вида:

$$f(\bullet)=\begin{cases} u(\bullet), & при \quad "Условие", \\ v(\bullet), & при \quad "\overline{Условие}". \end{cases}$$

где $u(\bullet)$, $v(\bullet)$ – произвольные выражения, *Условие* – логическое выражение и $\overline{Условие}$ – логическое выражение, обратное *Условию.*

Синтаксис условной операции можно представить следующим образом:

```
Условие ? Выражение_u : Выражение_v
```

Например, необходимо записать Verilog-выражение для вычисления функции

$$a=\begin{cases} x_1+2x_2, & при \quad x_1 < x_2, \\ 2x_1-x_2, & при \quad x_1 \geq x_2. \end{cases}$$

Соответствующий фрагмент программы будет иметь вид

```
//. . .
reg [7:0] a, x1, x2;
assign a = (x1 < x2)? (x1 + 2*x2) : (2*x1 – x2);
//. . .
```

Операции в выражениях выполняются в порядке, соответствующем заранее установленным приоритетам. Ниже перечислены операции в порядке снижения приоритета:

- унарные операции;
- арифметические операции умножения, деления, вычисления остатка;
- арифметические операции сложения и вычитания, операции сдвига;

- операции сравнения;
- операции равенства;
- логические операции;
- условные операции.