УДК (681.310)

БЫСТРОЕ МОДЕЛИРОВАНИЕ ЦИФРОВЫХ СХЕМ С ИСПОЛЬЗОВАНИЕМ ИХ СТРОЧНОГО ОПИСАНИЯ НА ЯЗЫКЕ ALEX

Ш.Е. Бозоян
Государственный Инженерный Университет Армении
Армения, 375009, Ереван, ул. Теряна, 105
E-mail: bozoyan@rau.am

В.С. Егиазарян
Российско-Армянский Государственный Университет
Армения, 375051, Ереван, ул. Овсепа Эмина, 123
E-mail: pmi@rau.am

С.П. Погосян
Российско-Армянский Государственный Университет
Армения, 375051, Ереван, ул. Овсепа Эмина, 123
E-mail: psamvel@gmail.com

Ключевые слова: язык Alex, язык Verilog, моделирование интегральных схем, тестирование, верификация
Key words: language Alex, language Verilog, circuits simulation, testing, verification

Приведен метод быстрого моделирования цифровых схем, использующий строчное представление схем на языке Alex. Использование этого языка обусловлено тем, что в описании схемы на языке Verilog не отражается логика моделирования. Язык Verilog дает возможность описать схему, перечисляя все элементы схемы и их связи между ними в произвольном порядке. Это приводит к тому, что становится необходимым приведение описания схемы с языка Verilog в промежуточный вид более удобного для моделирования. В качестве промежуточного языка описания выбран язык Alex, который полностью отражает логику моделирования.

FAST SIMULATION OF DIGITAL CIRCUITS USING THEIR LINE DESCRIPTION IN LANGUAGE ALEX / Sh.E. Bozoyan (State Engineering University of Armenia, 105 Teryan, Yerevan 375009, Armenia, E-mail: bozoyan@rau.am), V.S. Egiazaryan (Russian-Armenian State University, 123 Hovsep Emin, Yerevan 375051, Armenia, E-mail: pmi@rau.am), S.P. Poghosyan (Russian-Armenian State University, 123 Hovsep Emin, Yerevan 375051, Armenia, E-mail: psamvel@gmail.com). A method for fast simulation of digital circuits using line description of circuit on language Alex is described. Use of this language is based on the fact, that in the description of circuit in Verilog language does not reflect the simulation logic. The Verilog language provides a possibility to describe a circuit by enumerating all elements of circuit and connections between them in an arbitrary order. This brings us to the fact, that it is necessary to translate the description from language Verilog to intermediate representation that is more convenient for simulation. As an intermediate representation for a description language, it was taken the Alex language, which fully reflects the simulation logic.

1. Введение

Благодаря достижениям в области технологии изготовления интегральных схем (ИС) неуклонно уменьшаются как число дефектов, так и минимальный размер элемента кристалла. Это дает возможность разрабатывать и выпускать все более сложные и насыщенные кристаллы. Однако с ростом плотности компоновки ИС быстро усложняются и проблемы верификации и тестирования для выявления потенциальных ошибок: кристалл сверхбольшой интегральной схемы (СБИС) может содержать миллионы внутренних схемных узлов, которые недоступны для непосредственного управления и наблюдения через контакты ввода-вывода этого кристалла. Задачи верификации и тестирования могут оказаться исключительно трудными, а затраты времени и усилий на решение проблем тестирования могут привести к значительному возрастанию себестоимости производства СБИС.

Основной составной частью верификации и тестирования является функциональное моделирование схемы. Данная работа посвящена разработке способа быстрого моделирования схем из логических и запоминающих элементов. Метод базируется на использование языка строчного описания схем Alex.

2. Представление схем в памяти

Для этой цели был создан проект ifs (infrastructure), где были определены все используемые структуры данных. Схемы в памяти задаются с помощью структуры ifs::design.

2.1. ifs::design

Структура ifs::design представляет собой совокупность модулей. Она дает возможность создавать новые модули, удалять модули, получать модуль по имени и копировать модули по предварительно заданному префиксу. При создании структуры ifs::design в нем создаются все примитивные модули, как and, nand, or, nor, xor, xnor, not, buf и dff. Кроме этого создаются модули из стандартных библиотек для языка Verilog (and2, and3, and4, … or2, or3, or4, …), где вместе с именем модуля четко прописывается количество входов.

2.2. ifs::module

Структура ifs::module представляет собой смысловое описание модулей языка Verilog в памяти. Модули различаются по типу – module, primitive и gate. Модуль также содержит множество контактов, связей и объектов.

Контакт описывает либо вход или выход модуля. Связь описывает внутренние связи модуля. Объект – это применение другого модуля в этом модуле.

2.3. ifs::object

Это базисный тип для всех остальных типов, описывающий все общие атрибуты объектов, такие как тип объектов - module, instance, net, port, instance_port и их имена.

2.4. ifs::connector

Структура ifs::connector это абстрактный тип контактов. Она предназначена для участия в цепях и играет роль контактов в цепях. Connector имеет атрибут Net, который ссылается на ту цепь, которая содержит эго. Имеет также виртуальные методы для определения типа контакта.

- внешний (is_external) – один из входных или выходных портов модуля
- внутренний (is_internal) – входной или выходной контакт размещенного элемента.

Также дает возможность определить является ли этот контакт входным (is_input) или выходным (is_output).

2.5. ifs::port

Описывает 0х входы и выходы модуля. Имеет атрибут, определяющий тип, который может быть одним из следующих – input, output или inout.

2.6. ifs::net

Описывает 0х цепи модуля. Содержит совокупность всех контактов (connectors), которые участвуют в цепи. Дает возможность добавить новые или удалить старые контакты.

2.7. ifs::instance_port

Описывает контакты размещенных модулей, которые тоже могут участвовать в цепях. Имеет ссылки на объект размещенного модуля, т.е. на ifs::instance и на действительный вход или выход (port) этого модуля.

2.8. ifs::instance

Описывает размещенный объект модуля. Представляет собой ссылки на размещенный модуль и на тот модуль в котором он был размещен. Содержит также совокупность внутренних контактов размещенного модуля, которые находятся на этом объекте.

3. Формат Bench

Формат используется для описания эталонных схем ISCAS85 / ISCAS89 / ITC99 [4-6]. Это наиболее легкий способ описания по сравнению с языком описания схем Verilog. Он предназначен для описания схем только на одном (flat) уровне, означающем что схема состоит только из примитивных логических элементов.

В описании допускается использование комментариев. Эти строки начинаются с ‘#’ символа. В эталонных схемах это используется для указания количества входов и выходов. Также указывается количество использованных логических элементов.

В начале описываются входы схемы с помощью ключевого слово INPUT («имя входа»)**.** Потом перечисляются все выходы с помощью ключевого слово OUTPUT(«имя выхода»)**.**

Описание запоминающих элементов имеет следующий вид:

«выход» = DFF («вход»)

Здесь «вход» является входом для запоминающего элемента, а «выход» – выходом.

Потом описываются остальные элементы следующим образом –

«выход» = «логический элемент» («вход1»[, «вход2», «вход3», …«входN»])

Здесь «входN» является N-ым входом для логического элемента, а «выход» – выходом.

3.1. Пример схемы заданный на формате Bench (эталонная схема S27 из ISCAS89)

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)

G15 = OR(G12, G8)
G16 = OR(G3, G8)

G9 = NAND(G16, G15)

G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

4. Моделирование комбинационных схем, заданных с помощью формата Bench

Этот предварительный метод моделирования был создан для проверки результатов основной программы. Он имеет очень простой способ работы, поэтому можно положится на его результаты и считать их правильными.

Так как в комбинационных схемах не встречаются запоминающие элементы, то если создать все зависимости выходов от питающих их входов, то граф схемы будет ациклическим. Поэтому, множество выходов можно частично упорядочить.

Например из записи

```
G9 = NAND(G16, G15)
```

получаются следующие отношения – (G9, G16) и (G9, G15). Полученный таким образом класс можно частично упорядочить известным алгоритмом топологической сортировки (в программе была использована реализация topological_sort из библиотеки boost). Алгоритм в результате выдает список, где элемент A из отношения (A, B) находится в списке левее B. Это означает, что вычисляя справа налево можем, не нарушая зависимости, вычислить все выходные элементы, а также внешние выходы (которые находятся левее всех остальных элементов).

Допустим, что у нас уже есть отсортированное таким образом множество записей. Для ускорения вычислений эти записи мы будем транслировать в эквивалентные выражения в языке программирования C++. Например из вышеприведенной схемы (не считая запоминающие элементы) мы получим следующий отрезок программы:

```
G14 = !G0;
G8 = (G14 && G6);
G16 = (G3 || G8);
G12 = !(G1 || G7);
G15 = (G12 || G8);
G9 = !(G16 && G15);
G11 = !(G5 || G9);
G17 = !G11;
G10 = !(G14 || G11);
G13 = !(G2 || G12);
```

Таким образом полученная программа и будет играть роль программы симулятора, которая будет получать входные данные из файла и полученные результаты выходных данных запишет в другой файл.

Эту программу легко можно преобразовать таким образом, чтобы она поддерживала также и схемы с запоминающими элементами.

5. Язык Verilog HDL (Hardware Description Language)

Этот язык является на сегодняшний день одним из самых распространенных языков описания схем. Он имеет разнообразные способы представления, начиная с простого описания при помощи логических элементов (netlist), и заканчивая так называемым behavioral-описанием, которое очень похоже на язык программирования C++, где могут встречаться элементы условного перехода, циклы и т.д.

Язык Verilog имеет возможность описания как цифровых так и аналоговых схем. Сейчас распространяется Verilog AMS (Analog Mixed Signal) - новый вариант языка, который используется для описания смешанных, аналого-цифровых схем.

В дальнейшем будем рассматривать цифровые схемы в netlist форме языка Verilog.

5.1. Распознавание схемы на языке Verilog

Целью этой задачи является распознавание схем, заданных на языке Verilog, то есть конвертирование Verilog-схемы в структуры памяти. Все остальные алгоритмы уже напрямую будут работать с этими структурами.

Для распознавания формата был использован транслятор VL2MV, который переводит из Verilog-формата в BLIF-MV. При этом транслятор VL2MV был модифицирован для перевода Verilog-формата в структуры для представления схемы в памяти. Во время трансляции делается еще одно преобразование, которое облегчает дальнейшее использование: все шины встречающиеся в Verilog описании раскрываются, то есть, вместо описания A[15:0] создаются 16 отдельных сигналов – с именами A[15], A[14], … , A[1], A[0].

5.2. Пример схемы, описанный на языке Verilog

В следующем рисунке (рис. 1) приведено описание схемы на языке Verilog вычисляющее сумму двух двубитовых чисел. Схема описывается модулем первого уровня - two_bits_adder, который получает два числа – A2A1 и B2B1 и в результате вычисляет их сумму в CS2S1. Этот модуль в свою очередь использует четыре однобитных сумматора one_b1, one_b2, one_b22 и one_cb. Однобитный сумматор на входе получает два бита – A и B, и вычисляет их сумму в CS. Для этой цели были использованы 6 логических элементов – 2 *отрицания (not)*, 3 *и (and)* и 1 *или (or)*.

```
module two_bits_adder (C, A1, A2, B1, B2, S1, S2);
        input A1, A2, B1, B2;
        output C, S1, S2;

        wire S2x, Cx, C1, C2, C3;

        one_bit_adder one_b1  (C1, S1 , A1 , B1);
        one_bit_adder one_b2  (C2, S2x, A2 , B2);
        one_bit_adder one_b22 (C3, S2 , S2x, C1);
        one_bit_adder one_cb  (Cx, C  , C2 , C3);

endmodule /* two_bits_adder */

module one_bit_adder (C, S, A, B);

  input    A, B;
  output   C, S;

  wire e, f, r, t;

  not i1 (e, B);
  not i2 (f, A);
  and a1 (r, A, e);
  and a2 (t, f, B);
  and a3 (C, A, B);
  or  o1 (S, r, t);

endmodule /* one_bit_adder */
```

Рис. 1. Пример схемы, заданный на языке Verilog.

6. Приведение иерархического модуля к одному уровню

Так как мы рассматриваем упрощенную форму Verilog (схемы состоят только из логических элементов), то, очевидно, мы ничего не потеряем, если

приведем иерархический модуль к одному уровню. Это намного облегчит задачу симулятора, потому что уже все элементы, используемые в схеме, будут примитивными, и не будет вложенных модулей. Кроме того, это позволит нам оптимизировать полученный модуль, где могут встречаться много неиспользованных выходных сигналов.

На рис. 2 приведена схема two_bits_adder до преобразования, а на рис. 3 – после преобразования.

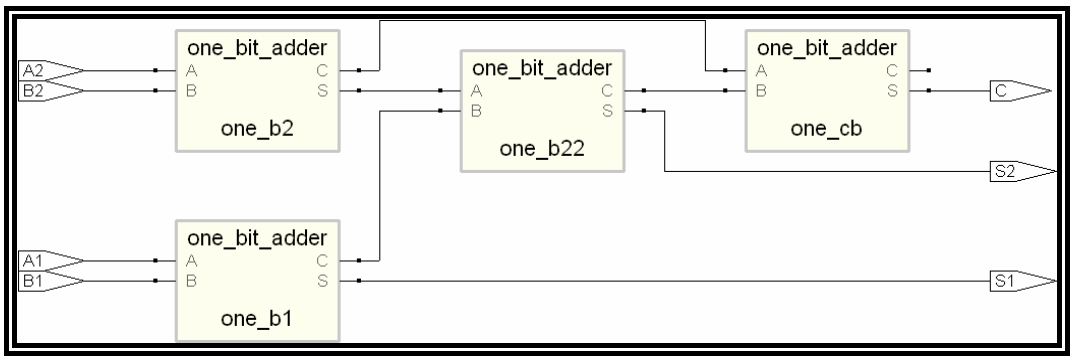

Рис. 2. Схема до преобразования.

Отчетливо видно, что значение сигнала one_cb.C (на рис. 3 находится в правом нижнем углу) не используется, потому и можно его игнорировать и вообще не вычислять.

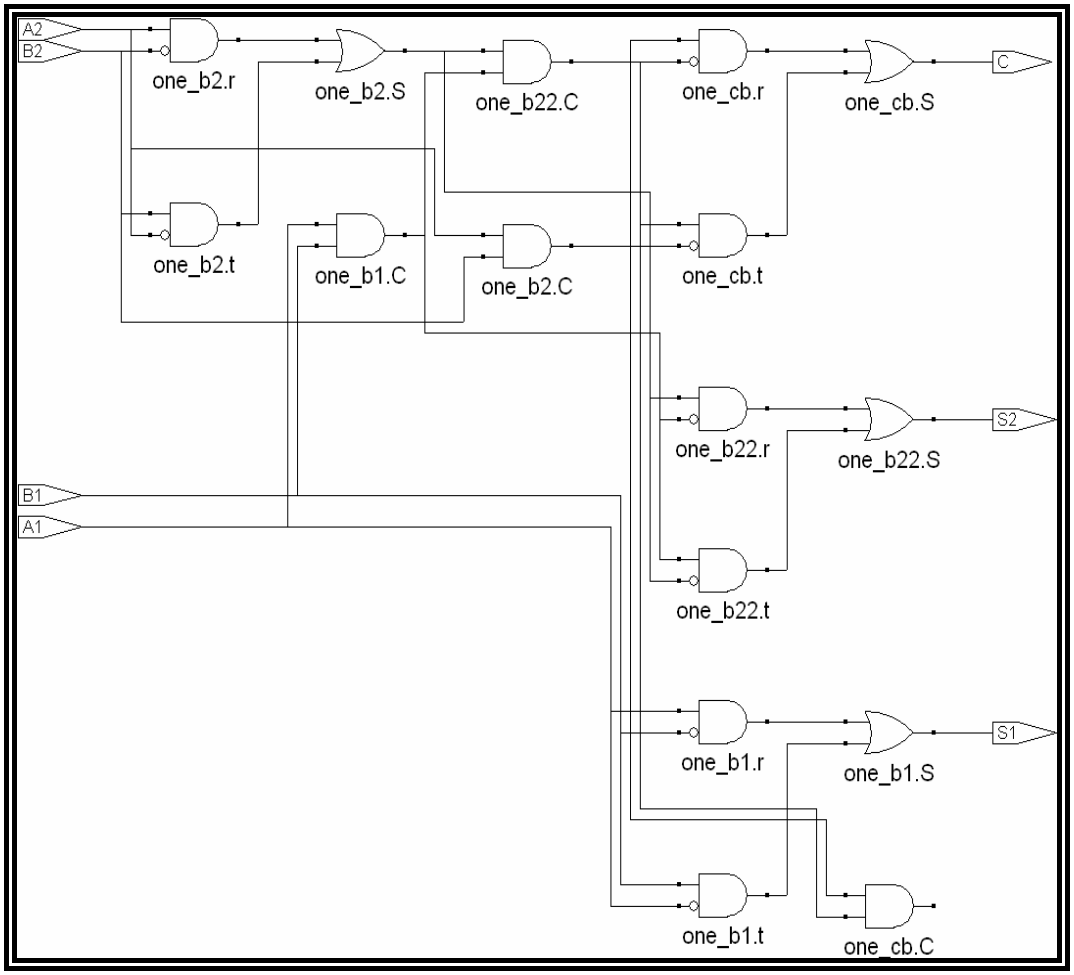

Рис. 3. Схема после преобразования.

7. Язык Alex

7.1. Почему Alex?

Одним из недостатков языка Verilog [3] является то, что в описании схемы на этом языке не отражается логика моделирования. Он дает возможность описать схему перечисляя все использованные элементы и их связи в произвольном порядке. Это приводит к тому что становится необходимым приведение описания из языка Verilog в промежуточный вид, более удобного для моделирования. В качестве промежуточного языка описания выбран язык строчного описания цифровых схем Alex [1, 2]. Для облегчения создания симулятора внесены некоторые модификации в язык Alex.

- описание ветвящихся элементов было выделено из записи в отдельную запись, так как мы в любом случае должны их отдельно вычислить. Также требуется, чтобы описания задавались по уровням, т.е. описание любой метки ветвления должно встречаться до описания элемента, где она будет использован;
- описания выходных сигналов производится отдельно;
- в описание добавлены скобки, чтобы сделать возможным распознавание формата Alex стандартными средствами создании компиляторов lex/yacc.

7.2. Описание сумматора двух двубитных чисел на языке Alex

Ниже приведен пример использования языка Alex для описания устройства вычисляющий сумму двух двубитных чисел:

```
*B_1 = B_1
*A_1 = A_1
*B_2 = B_2
*A_2 = A_2
*C_1 = and(2)(*A_1(0)(),*B_1(0)())
*C_2 = and(2)(*A_2(0)(),*B_2(0)())
*S2x = or(2)(and(2)(*A_2(0)(), not(1)(*B_2(0)())),
             and(2)(not(1) (*A_2(0)()), *B_2(0)()))
*C_3 = and(2)(*S2x(0)(),*C_1(0)())
C_C = or(2)(and(2)(*C_2(0)(), not(1)(*C_3(0)())),
             and(2)(not(1) (*C_2(0)()), *C_3(0)()))
S_1 = or(2)(and(2)(*A_1(0)(), not(1)(*B_1(0)())),
             and(2)(not(1) (*A_1(0)()), *B_1(0)()))
S_2 = or(2)(and(2)(*S2x(0)(), not(1)(*C_1(0)())),
             and(2)(not(1) (*S2x(0)()), *C_1(0)()))
```

7.3. Синтаксис языка Alex (используется формат yacc)

```
BODY    : DESCRIPTION
        | DESCRIPTION BODY
        ;

DESCRIPTION     : BRANCH_DEFINITION
                | DFF_DEFINITION
                | OUTPUT
                ;
```

```
    OUTPUT  : YYVARIABLE YYEQ MODEL
            ;

    BRANCH_DEFINITION : YYSTAR YYVARIABLE YYEQ MODEL
            ;

    DFF_DEFINITION : YYSHARP YYVARIABLE YYEQ MODEL
            ;

    MODEL   : GATE_NAME YYLEFT YYNUMBER YYRIGHT
              YYLEFT ARGUMENTS_LIST YYRIGHT
            | INPUT
            | BRANCH
            ;

    INPUT   : VARIABLE_NAME
            ;

    BRANCH     :   YYSTAR   VARIABLE_NAME   YYLEFT   YYNUMBER   YYRIGHT
BRANCH_INPUT
            ;

    BRANCH_INPUT    : YYLEFT YYRIGHT
                    | YYLEFT MODEL YYRIGHT
                    ;

    GATE_NAME       : YYAND
                    | YYNAND
                    | YYOR
                    | YYNOR
                    | YYXOR
                    | YYNOT
                    | YYBUF
                    | YYDFF
                    ;

    VARIABLE_NAME   : YYVARIABLE
                    ;

    ARGUMENTS_LIST  : ARGUMENT
                    | ARGUMENT YYCOMMA ARGUMENTS_LIST
                    ;

    ARGUMENT        : MODEL
                    | BRANCH
                    | INPUT
                    ;
```

7.4. Перевод схемы в промежуточный язык Alex

Для того, чтобы в результате получилось описание схемы по уровням, мы используем волновой алгоритм, где началом будут входы а концом выходы. Алгоритм работает следующим образом:

Все входы получают свои представления на языке Alex и добавляются к фронту волны.

Все входы элементов типа DFF помечаются как имеющие представление (хотя они еще реально не имеют представления). Находятся все элементы, которые питаются с фронта волны, т.е. все входы уже имеют представления на языке Alex, следовательно мы уже можем получить представление единственного вы-

хода этого элемента. После этой операции данный выход также добавляется к фронту волны.

Так продолжается, пока все выходы схемы не получат представления на языке Alex. В итоге, создается окончательное их представления, так как все выходы элементов DFF уже будут иметь представления.

7.5. Логический синтез схемы, описанной на языке Alex

Теперь допустим, что нам дано представление какого-нибудь выхода на языке Alex. Очевидно, что для любого такого отрезка, если разность количества всех встречающихся элементов и количества всех их входов равна 1, то это будет подсхемой нашей первоначальной схемы, другими словами, является входом для некоторого элемента.

Этот факт позволяет нам очень легко делать логический синтез: заменить формулы на эквивалентные, но более короткие.

Для перечисления таких тождеств используются следующие обозначения:

Вместо имен элементов используются следующие символы:

```
and  <-> '&'
nand <-> '$'
or   <-> '|'
nor  <-> '#'
xor  <-> '^'
not  <-> '!'
```

А любой другой символ, если он встречается в формуле несколько раз, то это означает, что в этих местах должны стоять одинаковые выражения.

```
##ab#ca = |a&bc ⇔
nor(2)(nor(2)(a, b), nor(2)(c, a)) = or(2)(a, and(2)(b, c))

##ab#ac = |a&bc ⇔
nor(2)(nor(2)(a, b), nor(2)(a, c)) = or(2)(a, and(2)(b, c))

##ba#ca = |a&bc ⇔
nor(2)(nor(2)(b, a), nor(2)(c, a)) = or(2)(a, and(2)(b, c))

##ba#ac = |a&bc ⇔
nor(2)(nor(2)(b, a), nor(2)(a, c)) = or(2)(a, and(2)(b, c))

|&a!b&!ab = ^ab ⇔
or(2)(and(2)(a, not(1)(b)), and(2)(not(1)(a), b))) = xor(2)(a, b)

|&a!b&b!a = ^ab ⇔
or(2)(and(2)(a, not(1)(b)), and(2)(b, not(1)(a)))) = xor(2)(a, b)

|&!ab&a!b = ^ab ⇔
or(2)(and(2)(not(1)(a), b), and(2)(a, not(1)(b)))) = xor(2)(a, b)

|&!ab&!ba = ^ab ⇔
or(2)(and(2)(not(1)(a), b), and(2)(not(1)(b), a))) = xor(2)(a, b)

$$a!b$b!a = ^ab ⇔
nand(2)(nand(2)(a, not(1)(b)), nand(2)(b, not(1)(a))) =
xor(2)(a, b)
```

```
$$!ba$b!a = ^ab ⇔
nand(2)(nand(2)(not(1)(b), a), nand(2)(b, not(1)(a))) =
xor(2)(a, b)

$$!ba$!ab = ^ab ⇔
nand(2)(nand(2)(not(1)(b), a), nand(2)(not(1)(a), b)) =
xor(2)(a, b)

$$a!b$!ab = ^ab ⇔
nand(2)(nand(2)(a, not(1)(b)), nand(2)(not(1)(a), b)) =
xor(2)(a, b)

#!a#ba = a ⇔
nor(2)(not(1)(a, nor(2)(b, a))) = a

#!a#ab = a ⇔
nor(2)(not(1)(a, nor(2)(a, b))) = a

##ba!a = a ⇔
nor(2)(nor(2)(b, a), not(1)(a))) = a

##ab!a = a ⇔
nor(2)(nor(2)(a, b), not(1)(a))) = a

!!a = a ⇔
not(1)(not(1)(a)) = a
```

7.6. Моделирование схемы, описанной на языке Alex

Рассматривается алгоритм для моделирования цифровых схем, используя преимущества языка Alex. Результат моделирования также может быть использован для сверки с результатами сгенерированного кода, рассматриваемого позже.

По ходу создаются три вектора, где соответственно сохраняются текущие значения входных сигналов, ветвящихся сигналов и сигналов, являющихся выходами элементов DFF.

Каждый раз, когда входам задаются новые значения, все эти данные стираются и начинается новый цикл вычислений.

Моделирование производится по уровням (в порядке, описанном в файле).

Каждый раз, получая новые описания на языке Alex, сначала необходимо их синтезировать, потом уже моделировать.

Допустим уже смоделировано некоторое количество уровней и сейчас рассматривается следующая строка – описание выхода ветвящегося или запоминающего элемента.

Моделирование производится справа налево. Для этой строки создается вектор уже вычисленных значений. Рассматривается очередной элемент. Если он является входным или ветвящимся элементом, то в начало вектора добавляется значение этого элемента. А если это функция, то проверяется наличие достаточного количества входов для этого элемента в векторе и вычисляется значение функции. После вычисления функции удаляется соответствующее количество входов из вектора, после чего в начало вектора добавляется значение функции.

Так продолжается до завершения строки. Если все было сделано правильно, то проверяем, чтобы в конце в векторе осталось только одно значение, которое и является результатом моделирования этой строки.

```
> Taking inputs
A_1 = 1
A_2 = 0
B_1 = 1
B_2 = 1

> Simulation ...
B_1 = B_1(0)
B_1 = B_1(0)
B_1 =  1
A_1 = A_1(0)
A_1 = A_1(0)
A_1 =  1
B_2 = B_2(0)
B_2 = B_2(0)
B_2 =  1
A_2 = A_2(0)
A_2 = A_2(0)
A_2 =  0
C_1 = and(2)*A_1(0)*B_1(0)
C_1 = and(2)*A_1(0)*B_1(0)
C_1 = and(2)*A_1(0) 1
C_1 = and(2) 11
C_1 =  1
C_2 = and(2)*A_2(0)*B_2(0)
C_2 = and(2)*A_2(0)*B_2(0)
C_2 = and(2)*A_2(0) 1
C_2 = and(2) 01
C_2 =  0
S2x = or(2)and(2)*A_2(0)not(1)*B_2(0)and(2)not(1)*A_2(0)*B_2(0)
S2x = xor(2)*A_2(0)*B_2(0)
S2x = xor(2)*A_2(0) 1
S2x = xor(2) 01
S2x =  1
C_3 = and(2)*S2x(0)*C_1(0)
C_3 = and(2)*S2x(0)*C_1(0)
C_3 = and(2)*S2x(0) 1
C_3 = and(2) 11
C_3 =  1
C_C = xor(2)*C_2(0)*C_3(0)
C_C = xor(2)*C_2(0) 1
C_C = xor(2) 01
C_C =  1
S_1 = xor(2)*A_1(0)*B_1(0)
S_1 = xor(2)*A_1(0) 1
S_1 = xor(2) 11
S_1 =  0
S_2 = xor(2)*S2x(0)*C_1(0)
S_2 = xor(2)*S2x(0) 1
S_2 = xor(2) 11
S_2 =  0

> Outputs are
C_C = 1
S_1 = 0
S_2 = 0
```

Здесь можно заметить, что сигнал S2x был синтезирован и получил более короткое описание.

8. Перевод описания схемы с языка Alex на язык Verilog

Эта задача преследует одну цель – проверить, насколько транслирование из языка Verilog в Alex проведено безошибочно. Это проверяется с помощью одного из существующих симуляторов. Сначала симулятор работает с оригинальной версией Verilog-схемы, потом с транслированной версией. Если результаты совпадают на довольно большом количестве входных данных, то с большой уверенностью можем сказать, что наш перевод из языка Verilog на Alex проведен безошибочно.

Дедуктивное построение языка Alex и доказательство теоремы о записи [3] дают возможность осуществить синтаксический контроль правильности описания схемы на языке Alex.

Для трансляции требуется одна переменная – для хранения глубины в описании и два стека – имен выходов и пар (функция, кол. Входов).

Рассмотрим главные алгоритмические шаги в процессе трансляции.

1) Если во время чтения из файла встречаются строки «**переменная** =», «***переменная** =» или «**#переменная** =», то добавляется **переменная** к стеку имен и переходим к чтению ее описания.
2) Если во время чтения из файла встречается строка «**элемент (число)**», то
   - увеличивается глубина на 1;
   - добавляется ко второму стеку пару **(элемент, число)**;
   - если глубина больше 1, то добавляется к первому стеку имен выходов сгенерированное имя с префиксом _U;
   - рекурсивно распознается список входов «(**вход1, вход2, …)**»;
   - проверяется, пусты ли стеки;
   - берется верхняя функция из второго стека – **F**;
   - проверяется, чтобы количество имеющихся входов было больше или равно количеству входов функции. (обозначается через **C**);
   - берется верхнее имя из первого стека – **O**;
   - пишется – «**F** сгенерированное имя с префиксом __I (**O**, последние **C – 1** входы)»;
   - удаляется из вектора последние **C – 1** входов;
   - уменьшается глубина на 1;
   - добавляется **O** к списку аргументов;
3) Если встречаются входы или ветвящиеся элементы, то они добавляются к списку аргументов.

Пример: Рассмотрим следующую строку Alex и попытаемся шаг за шагом выполнить шаги алгоритма.

```
   C_C  =  or(2)(and(2)(*C_2(0)(),  not(1)(*C_3(0)())),  and(2)(not(1)
(*C_2(0)()), *C_3(0)()))
```

Таблица 1. В таблице показывается процесс перевода с языка Alex на язык Verilog.

Стек выходов	Стек <функция, кол. входов>	Глубина	Входы	Полученный результат
C_C = or(2)(and(2)(*C_2(0)(), not(1)(*C_3(0)())), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
C_C		0		
or(2)(and(2)(*C_2(0)(), not(1)(*C_3(0)())), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
C_C	or, 2	1		
(and(2)(*C_2(0)(), not(1)(*C_3(0)())), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
__U1 C_C	and, 2 or, 2	2		
(*C_2(0)(), not(1)(*C_3(0)())), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
__U1 C_C	and, 2 or, 2	2	C_2	
not(1)(*C_3(0)())), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
__U2 __U1 C_C	not, 1 and, 2 or, 2	3	C_2	
(*C_3(0)())), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
__U2 __U1 C_C	not, 1 and, 2 or, 2	3	C_3 C_2	
)), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
~~__U2~~ __U1 C_C	~~not, 1~~ and, 2 or, 2	2	__U2 ~~C_3~~ C_2	not __I1 (__U2, C_3)
), and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
~~__U1~~ C_C	~~and, 2~~ or, 2	1	__U1 ~~__U2~~ ~~C_2~~	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2)
and(2)(not(1) (*C_2(0)()), *C_3(0)()))				
__U3 C_C	and, 1 or, 2	2	__U1	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2)
(not(1) (*C_2(0)()), *C_3(0)()))				
__U4 __U3 C_C	not, 1 and, 2 or, 2	3	__U1	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2)
(*C_2(0)()), *C_3(0)()))				
__U4 __U3 C_C	not, 1 and, 2 or, 2	3	C_2 __U1	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2)
), *C_3(0)()))				
~~__U4~~ __U3 C_C	~~not, 1~~ and, 2 or, 2	2	__U4 ~~C_2~~ __U1	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2) not __I3 (__U4, C_2)
*C_3(0)()))				
__U3 C_C	and, 2 or, 2	2	C_3 __U4 __U1	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2) not __I3 (__U4, C_2)
))				
~~__U3~~ C_C	~~and, 2~~ or, 2	1	__U3 ~~C_3~~ ~~__U4~~ __U1	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2) not __I3 (__U4, C_2) and __I4 (__U3, __U4, C_3)
)				
~~C_C~~	~~or, 2~~	0	~~__U3~~ ~~__U1~~	not __I1 (__U2, C_3) and __I2 (__U1, C_2, __U2) not __I3 (__U4, C_2) and __I4 (__U3, __U4, C_3) or __I5 (C_C, C_2, __U1, __U3)

9. Трансляция с языка Alex в C++ и моделирование схемы

Как уже было сказано, одним из преимуществ языка Alex является то, что в его записи содержится логика моделирования. При описании схемы по уровням в любой записи встречаются или входные элементы, или ветвящиеся элементы, чьи описания уже встречались до этого. Т.е. значение функции можно вычислить одним проходом справа - налево.

Очевидно, что если имеется любое выражение, например a+b, то намного быстрее будет найдено его значение если мы составим программу, вычисляющую это выражение, нежели мы сами будем производить распознавание этого выражения в программе. Таким образом мы можем избежать лишних операций и сразу составлять программы, смысл работы которых и будет моделирование схемы.

Например:

Схема S27 моделируется с помощью следующего сгенерированного кода C++:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include "../inputs_reader.hxx"
#include "../signal.hxx"
using namespace alex;

inputs_reader ir;
bool __clock__ = true;
        signal br[22];
        signal brn[22];
        signal in[21];
std::ofstream out("output.dat",std::ios::out);
void model_inputs ()
{
        ir.read ("G0", in[11]);
        ir.read ("G1", in[13]);
        ir.read ("G2", in[20]);
        ir.read ("G3", in[19]);
}
void simple0()
{
        br[10] = !(in[11]);
        br[12] = !(in[13] || br[14]);
        br[15] = (br[10] && br[16]);
        br[17]  =  !(br[18]  ||  !((in[19]  ||  br[15])  &&  (br[12]  ||
br[15])));
        br[21] = !(br[17]);
        out << br[21];
}
void simple ()
{
    simple0 ();
}
void dff0()
{
```

```
        brn[18] = (!(br[10] || br[17]));
        brn[16] = (!(br[18] || !((in[19] || br[15]) && (br[12] ||
br[15]))));
        brn[14] = (!(in[20] || br[12]));
}
void dff ()
{
    dff0 ();
}

int main()
{
        srand (0);
        ir.init();
        int start_time = time(0);
        int k = 0;
        for (; k < ir.get_number_of_tests(); ++k) {
            model_inputs ();
            simple ();
            if (__clock__) {
                    dff ();
            }
            out << std::endl;
            br[18] = brn[18];
            br[16] = brn[16];
            br[14] = brn[14];
            __clock__ = !__clock__;
        }
        out.close();
        int delta_time = time(0) - start_time;
        std::cout << "Took " << delta_time
                  << " sec" << std::endl;
        return 0;
}
```

10. Сравнение программы AlexSim с программой ModelSim v5.5

Для сравнений результатов времени моделирования предлагаемого симулятора AlexSim с симулятором ModelSim V5.5 рассматривались несколько эталонных схем [4, 5], создавались так называемые testbench-и для их моделирования. Они также пишутся на языке Verilog, но уже в behavioral форме и предназначены для определения входов схемы в каждом такте симуляции и записи выходов в файл. Входные данные тоже записываются в файл, чтобы в дальнейшем можно было ввести их в нашу программу и проверить идентичность результатов.

Таблица 2. В таблице приведены сравнения программы AlexSim с программой ModelSim 5.5. Все входные значения при моделировании брались из генератора случайных чисел. Количество наборов во всех случаях = 1.000.000. Эксперименты были выполнены под операционной системой RedHat Linux 9.0 на ПК с мощностью 1800Mhz.

Имя схемы	Кол. входов/ выходов	Кол. Элементов	Время симуляции ModelSim-a (с.)	Время симуляции программы AlexSim (компиляция/время симуляции) (с.)	Отношение времен симуляции
c1355	41 / 32	546	60	4 / 2.7	22,22
c1908	33 / 25	880	115	3 / 2.9	39,66
c2670	233 / 140	1193	154	28 / 16.1	9,57
c3540	50 / 22	1669	204	5 / 5.4	37,78
c499	41 / 32	202	28	3 / 3.6	7,78
c5315	178 / 123	2307	393	22 / 15.4	25,52
c6288	32 / 32	2416	417	9 / 8.5	49,06
c7552	207 / 108	3512	967	26 / 22.6	42,79
s1196a	14 / 14	547	57	2 / 1.9	30,00
s1196b	14 / 14	547	56	3 / 1.8	31,11
s1238a	14 / 14	526	57	2 / 1.9	30,00
s1238	14 / 14	526	56	3 / 1.9	29,47
s13207	62 / 152	8589	2106	15 / 16.0	131,63
s1423	17 / 5	731	105	2 / 3.1	33,87
s1488	8 / 19	659	30	2 / 1.5	20,00
s27	4 / 1	13	7	2 / 2.0	3,50
s298	3 / 6	133	20	2 / 0.4	50,00
s344	9 / 11	175	24	2 / 0.7	34,29
s349	9 / 11	176	25	2 / 0.7	35,71
s382	3 / 6	179	27	2 / 0.5	54,00
s386	7 / 7	165	17	1 / 0.7	24,29
s400	3 / 6	185	27	2 / 0.7	38,57
s420	18 / 1	234	30	2 / 1.4	21,43
s444	3 / 6	202	27	1 / 0.6	45,00
s510	19 / 7	217	13	2 / 1.4	9,29
s526a	3 / 6	215	26	1 / 0.6	43,33
s526	3 / 6	214	26	2 / 0.6	43,33
s5378	35 / 49	2958	288	8 / 7.5	38,40
s713	35 / 23	412	46	2 / 2.3	20,00
s820	18 / 19	294	22	2 / 1.5	14,67
s832	18 / 19	292	22	2 / 2.0	11,00
s838	34 / 1	478	58	2 / 2.6	22,31
s9234	36 / 39	5808	639	10 / 8.9	71,80
s953	16 / 23	424	53	2 / 1.7	31,18
B01_C	7/7	54	7	2/0.4	17.5
B02_C	5/5	32	5	1/0.4	12.5
B03_C	34/34	190	16	1/2.2	7.27
B04_C	77/74	803	73	3/5.4	13.51
B05_C	39/60	1032	127	4/4.4	28.86
B06_C	11/14	65	6	2/0.6	10
B07_C	50/57	490	42	3/3.4	12.35
B08_C	30/25	204	19	2/2.0	9.5
B10_C	28/23	223	20	2/2.0	10
B11_C	38/37	801	75	3/3.8	19.73
B12_C	128/125	1197	73	5/10.8	6.75
B13_C	63/63	415	33	2/4.2	7.85
B14_C	277/299	10343	5563	69/85	65.44
B15_C	485/519	9371	2290	81/55	41.63
B17_C	1452/1512	33741	11370	370/550	20.67
B20_C	522/512	20716	16460	150/145	113.51
B21_C	522/512	21061	16620	150/150	110.8
B22_C	767/757	30686	24880	239/415	59.95

Список литературы

1. Бозоян Ш.Е. Язык описания функциональных схем // Изв. АН СССР. Техническая кибернетика. 1978. № 4. С. 158-166.
2. Бозоян Ш.Е., Егиазарян В.С., Новый подход к модельно-ориентированному проектированию систем на чипах // Электронный журнал «ИССЛЕДОВАНО В РОССИИ». 2003. С. 1386-1395. http://zhurnal.ape.relarn.ru/articles/2003/115.pdf.
3. Pabmitkar S. Verilog HDL: Guide to Digital Design and Synthesis // SunSoftPress, Prentice Hall, 1996. P. 105-117.
4. Hansen M., Yalcin H., Hayes J.P. Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering // IEEE Design and Test. 1999. Vol. 16, No 3. P. 72-80.
5. Brgles F., Bryan D. Kozminski K. Combinational profiles of sequential benchmark circuits // International symposium of circuits and systems ISCAS-89. 1989. P. 1929-1934.
6. ITC' 99 Benchmarks (2nd release). http://www.cad.polito.it/ tools/itc99.htm.