Глава 3

Литеральные значения и встроенные типы данных в SystemVerilog

SystemVerilog расширяет встроенные типы Verilog, и показывает как могут быть определены литеральные значения. Эта глава объясняет эти улучшения и предлагает рекомендации по использованию. Эти улучшения иллюстрируются небольшим количеством примеров. Последующие главы содержат другие примеры, которые используют расширенные типы и литеральные значения. Следующая глава освещает другие важные расширения типов переменных и типов определяемых пользователем.

Расширения представленные в этой главе:

- Расширенные литеральные значения
- Тексто-замещение `define
- Временные значения
- Новые типы переменных
- Знаковые и без-знаковые типы
- Инициализация переменной
- Статические и автоматические переменные
- Приведение типов
- Константы

3.1 Расширенные литеральные значения

Заполнение вектора литеральным значением

В языке Verilog, вектор может быть легко заполнен нулями, неопределенными состояниями X, высоко-импедансными состояниями Z.

```
parameter SIZE = 64;
reg [SIZE-1:0] data;

data = 0;   // fills all bits of data with zero
data = 'bz; // fills all bits of data with Z
data = 'bx; // fills all bits of data with X
```

Каждое присвоение в примере выше масштабируемо. Если параметр SIZE переопределить на 128, то заполнение значениями будет распространяться на новый размер данных. Однако Verilog не предоставляет удобный механизм заполнения вектора. Чтобы заполнить все биты одним значением, размер должен быть фиксированным. На пример:

```
data=64'hFFFFFFFFFFFFFFFF;
```

Этот пример не масштабируемый. Если значение константы SIZE задать 128, литеральное значение должно быть изменено вручную, чтобы отражать новый размер данных. Для того чтобы сделать присвоение масштабируемым, необходимо использовать определённые трюки для заполнения вектора, вместо литеральных значений. Следующие два примера это иллюстрируют:

```
data = ~0;  // one's complement operation
data = -1;  // two's complement operation
```

Специальное литеральное значение для заполнения вектора

SystemVerilog расширяет возможности присвоения литерального значения двумя способами. Первое, добавлен простой синтаксис, который позволяет определять значение без символа основания системы счисления. Второе, значение может быть логическая 1. Значению, которым будет заполняться каждый бит, предшествует апостроф ('). Таким образом имеем:

- '0 fills all bits on the left-hand side with 0
- '1 fills all bits on the left-hand side with 1
- 'z or 'Z fills all bits on the left-hand side with z
- 'x or 'X fills all bits on the left-hand side with x

Обратите внимание, символ апострофа должен быть ('), а не (`).

Используя SystemVerilog, вектор любого размера может быть заполнен как

```
data = '1; // fills all bits of data with 1
```

Литеральные значения масштабируются с размером вектора по левостороннему принципу

Это улучшение упрощает написание модулей, которые работают с векторами больших размеров. Это улучшение также делает возможным разработку модулей, которые автоматически масштабируются к новым размерам вектора без модификации логики модели. Это автоматическое масштабирование особенно полезно, когда используется инициализация переменных, которые параметризуют размер вектора.

3.2 Расширения `define

SystemVerilog расширяет возможности замещения текста в макросе `define, позволяя включать специальные символы.

3.2.1 Макро аргумент замещения внутри строк

Verilog позволяет использовать кавычки (") в **`define,** но текст в кавычках становится литеральной строкой. Это значит, что в Verilog невозможно создать строку, используя макрос тексто-замещения где строка содержит встроенные макро аргументы.

В Verilog следующий пример работать не будет:

```
`define print(v) \
   $display("variable v = %h", v)

`print(data);
```

В этом примере макрос `print(data) развернётся в

```
$display("variable v = %h", data);
```

Макро аргумент v будет замещаться фактическим значением аргумента data. Однако, v в кавычках не будет замещено значением аргумента data.

`" *позволяет замещение внутри строки макроса*

SystemVerilog позволяет делать замещение внутри текстовой строки макроса, используя кавычки.

Замещаемый текст будет содержать строку, содержимое которой включает имя и логическое значение аргумента. Формат %h внутри строки будет корректно интерпретирован, как формат аргумента.

```
`define print(v) \
   $display(`"variable v = %h`", v)

`print(data);
```

В этом примере макрос `print() развернётся в:

```
$display("variable data = %h", data);
```

В Verilog кавычки внутри строки должны использоваться с `\"`, таким образом не оказывается влияние на кавычки внешней строки. Следующий Verilog пример встраивает кавычки в сообщение.

```
$display("variable \"data\" = %h", data);
```

`\`" *позволяет избежать кавычек в текстовой строке макроса, содержащей замещаемый аргумент*

Когда текстовая строка макроса содержит замещаемую переменную, то вместо `\"` нужно использовать `` `\`" ``. На пример:

```
`define print(v) \
   $display(`"variable `\`"v`\`" = %h`", v)

`print(data);
```

В этом примере макрос `print() будет разворачиваться в:

```
$display("variable \"data\" = %h", data);
```

3.2.2 Конструирование идентификатора имён макросом

Используя Verilog **`define**, невозможно создать идентификатор имени объединением двух или более текстовых макросов. Проблема заключается в символе пробела между частями идентификатора имени.

`` служит как разделитель внутри макроса

SystemVerilog предоставляет способ разделить идентификатор имени без использования пробела, используя символы апострофа ``. Это позволяет объединять два, и более имён.

Применение `` упрощает исходный код, там где необходимо объявлять похожие имена несколько раз, и нельзя использовать массивы для объявления. В следующем примере переменные **bit** и **wand** нужно объявить с похожими именами и значение переменной присвоить сети. Тип **bit** более детально разбирается дальше в этой главе. Если коротко, то тип **bit** похож на тип **reg**, но переменная **bit** имеет 2 состояния, а переменная **reg** 4 состояния значения.

Исходный код без текстового замещения:

```
bit d00_bit;  wand d00_net = d00_bit;
bit d01_bit;  wand d01_net = d01_bit;
... // repeat 60 more times, for each bit
bit d62_bit;  wand d62_net = d62_bit;
bit d63_bit;  wand d63_net = d63_bit;
```

Используя SystemVerilog расширения для **`define**, эти объявления можно упростить:

```
`define TWO_STATE_NET(name) bit name``_bit; \
  wand name``_net = name``_bit;

`TWO_STATE_NET(d00)
`TWO_STATE_NET(d01)
...
`TWO_STATE_NET(d62)
`TWO_STATE_NET(d63)
```

3.3 Переменные SystemVerilog

3.3.1 Объектные типы и типы данных

Типы данных Verilog

Аппаратные типы Verilog

Язык Verilog имеет аппаратно-ориентированные типы данных и сетевые типы (net). Эти типы имеют специальную семантику симуляции и синтеза чтобы показать поведение соединений в чипе или системе.

- Verilog переменные **reg**, **interger** и **time** имеют 4 логических значения: 0, 1, Z и X.
- Verilog переменные **wire**, **wor**, **wand** и другие сетевые типы (4 логических значения и уровни значения сигнала в сети).

Типы данных SystemVerilog

Объявления данных

Verilog не делает различий между типами сигналов, значение сигналов могут сохраняться или передаваться. В Verilog все сети и переменные используют 4 состояния, поэтому нет необходимости в чётком разделении сигналов. Для большей гибкости в типах переменных и сетей и значений, которые эти типы могут сохранять и передавать, стандарт SystemVerilog определяет два вида типов ***type*** и ***data type***.

“ type” определяет либо это данные net либо переменная

Type показывает является сигнал сетью или переменной. SystemVerilog использует все типы переменных Verilog, такие как **reg** и **interger**, плюс добавляет несколько типов переменных, таких как **byte** и **int.** SystemVerilog не добавляет расширения в сетевые типы (net).

“data type” определяет данные с 2-мя или с 4-мя состояниями

Data type показывает системное значение сети или переменной, 0 или 1 для типов данных с 2 состояниями, и 0, 1, Z или X для типов данных с 4 состояниями. Ключевое слово **bit** определяет, что объект есть тип данных с 2 состояниями. Ключевое слово **logic** определяет, что объект есть тип данных с 4 состояниями. В стандарте SystemVerilog-2005, типы переменных могут быть либо с 2-мя состояниями, либо 4-мя состояниями, в то время как сетевые типы могут быть только с 4 состояниями.

3.3.2 SystemVerilog переменные с 4-мя состояниями

Тип logic 4-мя состояния

Тип reg Verilog

Язык Verilog использует тип **reg** как переменную общего назначения для моделирования поведения аппаратуры в процедурных блоках **initial** и **always**. Ключевое слово **reg** часто вводит в заблуждение новичков. Термин *“reg”*, казалось бы, подразумевает аппаратный регистр *“register”*, построенный на последовательной логике. В действительности, нет какой-либо взаимосвязи между переменной **reg** и аппаратурой. Переменная **reg** определяет, какой логикой представлена аппаратура, комбинационной или последовательностной.

SystemVerilog использует ключевое слово **logic** для представления типа данных общего назначения, аппаратно-ориентированного типа. Пример использования типа **logic:**

```
logic resetN; // a 1-bit wide 4-state variable

logic [63:0] data; // a 64-bit wide variable

logic [0:7] array [0:255]; // an array of 8-bit
                           variables
```

Ключевое слово logic

Ключевое слово **logic** это не тип переменной, это тип данных, показывающий что сигнал может иметь 4 состояния. Однако, когда используется просто **logic** подразумевается переменная. Переменная с 4-мя состояниями может быть объявлена явно, используя пару ключевых слов **var logic**. На пример:

```
var logic [63:0] addr; // a 64-bit wide variable
```

Тип сеть в Verilog имеет 4 логических состояния. Тип сеть может быть объявлена явно, используя ключевое слово **logic.** Например:

```
wire logic [63:0] data; // a 64-bit wide net
```

Явно объявляемые типы данных сети и переменные обсуждаются более подробно в секции 3.3.4.

Семантически, переменные типа **logic** идентичны типу **reg** в Verilog. Эти два слова синонимы, и являются взаимозаменяемыми (исключение: ключевое слово **reg** не может использоваться вместе типом net). Также как переменная типа reg в Verilog, переменная типа **logic** может хранить 4 состояния значения(0, 1, Z и X), и может быть определена как вектор с любым размером.

Так как ключевое слово logic не несёт в себе ложного представления о типе представляемого аппаратного обеспечения, оно является более интуитивным для описания аппаратного обеспечения с 4 состояниями. В последующих примерах в этой книге, тип **logic** используется вместо типа **reg** (исключая когда иллюстрируется код Verilog).

3.3.3 Переменные с 2-мя состояниями SystemVerilog

Типы с 2-мя состояниями SystemVerilog

SystemVerilog добавляет несколько типов с 2-мя состояниями, подходящими для моделирования на более абстрактных уровнях чем RTL, таких как системный уровень и уровень операций. Эти типы включают:

- `bit` — a 1-bit 2-state integer
- `byte` — an 8-bit 2-state integer, similar to a C `char`
- `shortint` — a 16-bit 2-state integer, similar to a C `short`
- `int` — a 32-bit 2-state integer, similar to a C `int`
- `longint` — a 64-bit 2-state integer, similar to a C `longlong`

Использование типа bit с 2-мя состояниями

Уровни абстрактного моделирования не нуждаются в значениях с 4-мя состояниями

Переменные типа reg и logic используются для моделирования поведения аппаратного обеспечения в процедурных блоках. Эти типы хранят 4 логических состояния 0, 1, Z и X. Такие типы предпочтительны для синтезируемых RTL моделей. Z значение показывает отсоединение или третье состояние. X значение помогает выявлять ошибки. На таких высоких уровнях моделирования как системный и уровень операций, логические значения Z и X редко востребованы.

Переменная bit с 2-мя состояниями может быть использована вместо reg или logic

В SystemVerilog можно объявлять переменные как тип **bit**. Синтаксически переменная **bit** может использоваться вместо **reg** и **logic**. Однако, семантически тип **bit** отличается, он имеет только 2 состояния 0 и 1. Тип **bit** может быть использован для моделирования аппаратуры на верхних уровнях абстракций.

Переменные типа **bit** объявляются также как типы **reg** и **logic**. Объявления могут быть вектором любого размера, от 1 бита до максимального размера поддерживаемого программным инструментарием(стандарт Verilog IEEE

1364 определяет, что все программные инструменты поддерживают размер вектора 2^16 бит).

```
bit resetN; // a 1-bit wide 2-state variable

bit [63:0] data; // a 64-bit 2-state variable

bit [0:7] array [0:255]; // an array of 8-bit
                            2-state variables
```

Ключевое слово bit

Ключевое слово **bit** это не тип переменной, это тип данных показывающий что переменная может иметь 2 состояния. Однако, когда используется одно ключевое слово **bit**, подразумевается переменная. Переменная с 2-мя состояниями может быть явно объявлена, используя пару ключевых слов **var bit**. Например:

```
var bit [63:0] addr; // a 64-bit wide variable
```

Явно объявленные переменные более подробно обсуждаются в секции 3.3.4.

Использование С-подобных типов.

Типы с 2-мя состояниями могут быть применены к моделям C и C++

Для более абстрактного моделирования, используются такие С-подобные типы с 2-мя состояниями как **int** и **byte**. На этом уровне нет необходимости в таких представлениях аппаратуры как шины с третьим состоянием и X значениями. Другой ключевой момент использования С-подобных типов это взаимодействие между моделями Verilog и моделями C или C++, используя Direct Programming Interface (DPI) от SystemVerilog. Использование типов, которые имеют общие представление в обоих языках, делает простым и эффективным обмен данными между этими языками.

Тип int может быть использован как переменная управлением циклом for

Другое использование типа **int** как переменная управления циклом **for**. В синтезируемых RTL моделях, переменная управления циклом является временной переменной, которая исчезает на вентильном уровне синтеза. Переменная управления циклом не нуждается в 4-х состояниях. Тип **int**

хорошо работает как переменная управления циклом, как для абстрактных моделей так и для синтезируемых RTL моделей.

Семантика симуляции 2-х состояний

Типы с 4-мя состояниями начинают симуляцию с логического значения X

До начала симуляции, значение всех бит, таких переменных с 4-мя состояниями как **reg**, **logic** и **interger** будет X. Эти переменные рассматриваются как неинициализированные, до тех пор пока не присвоено первое значение. Переменные с 4-мя состояниями могут быть определены до начала симуляции каким-либо значением, используя in-line инициализацию. In-line инициализация обсуждается в секции 3.8

Типы с 2-мя состояниями начинают симуляцию с логического значения 0

Типы данных с 2-мя состояниями, до начала симуляции, будут иметь значение 0. Поскольку у таких типов данных нет состояния X, то это является одной из причин использования типов с 4-мя состояниями в синтезируемых RTL моделях.

Значения X и Z транслируются в 0 в типах с 2-мя состояниями

Вполне легально присваивать значения с 4-мя состояниями переменным с 2-мя состояниями. Например, значение входа с 4-мя состояниями может быть присвоено типу **bit** с 2-мя состояния внутри модуля. Биты, которые имеют значение X или Z в типе с 4-мя состояниями будут транслироваться в логический 0 в соответствующие битовые позиции переменной с 2-мя состояниями.

Другие абстрактные типы

Тип void ничего не хранит

SystemVerilog добавляет тип **void** который ничего не хранит. Тип void может быть использован в tagged союзах (см. глава 5) и в определении функции, которая ничего не возвращает (см. глава 6).

shortreal эквивалентен float в языке C

SystemVerilog также добавляет тип переменной **shortreal**, который соответствует типу **real** в Verilog. **shortreal** хранит 32 битное значение одинарной точности с плавающей точкой такое же как **float** в C, а **real** хранит переменную двойной точности, как **double** в C. Типы real и shortreal не синтезируемые, но могут быть полезны в абстрактных аппаратных моделях и в тестбенчах.

Верификационные расширения в SystemVerilog добавляют классы и другие динамические типы для использования в высокоуровневых тестбенчах. Эти типы не рассматриваются в этой книге.

3.3.4 Явная и неявная переменная и типы данных сеть (net)

net и типы переменных, типы данных с 2-мя и 4-мя состояниями

В терминологии SystemVerilog переменные и сети это типы, которые могут иметь 2 или 4 состояния (в стандарте 2005 SystemVerilog, сети имеют только 4 состояния). Типы данных с 4-мя состояниями представлены ключевым словом **logic**. Типы данных с 2-мя состояниями представлены ключевым словом **bit**. Когда явно не определено, являются ли данные переменной или сетью, то подразумевается переменная.

```
logic [7:0] busA;  // infers a variable that is
                   // a 4-state data type

bit [31:0] busB;  // infers a variable that is
                  // a 2-state data type
```

В Verilog ключевые слова **interger** и **time** это типы данных с 4-мя состояниями с предопределённым размером вектора. В SystemVerilog ключевые слова **int**, **byte**, **shortint** и **longint** это типы данных с 2-мя состояниями с предопределённым размером вектора.

В SystemVerilog перед типами данных можно использовать ключевое слово **var**. Например:

```
var logic [7:0] a;    // 4-state 8-bit variable
var bit [31:0] b;     // 2-state 32-bit variable
var int i;            // 2-state 32-bit variable
```

“var” это сокращённое “variable”

Ключевое слово **var** показывает, что объект является переменной. Ключевое слово **var** не оказывает влияния на поведение переменной при симуляции или синтезе. Использование **var** делает код самодокументирующимся. Такая явная документация кода помогает делать его более читабельным и сопровождаемым, когда переменные созданы от типа определяемого пользователем. Например:

```
typedef enum bit {FALSE, TRUE} bool_t;
var bool_t c;  // variable of user-defined type
```

Переменная может быть объявлена с использованием **var** без явного типа данных. В этом случае переменная рассматривается как тип **logic**.

```
var [7:0] d;          // 4-state 8-bit variable
```

Все типы net в Verilog (**wire, uwire, wand, wor, tri, triand, trior, tri0, tri1, trireg, supply0, supply1**) имеют логический тип данных с 4-мя состояниями.

```
wire [31:0] busB; // declares a net type
                  // that is implicitly a
                  // a 4-state logic data type
```

Дополнительно, тип net может быть объявлена с использованием типа сеть (net) и типа данных logic.

```
wire logic [31:0] busC;
```

Чтобы предотвратить путаницу в комбинации ключевых слов, SystemVerilog не позволяет ключевому слову **reg** быть в паре с ключевыми словами сетевого типа.

```
wire reg [31:0] busD; // ILLEGAL keyword pair
```

3.3.5 Рекомендации по синтезу

Типы с 2-мя состояниями синтезируются также как типы с 4-мя состояниями

Тип **logic** с 4-мя состояниями и типы **bit**, **byte**, **shortint**, **int**, **longint** с 2-мя состояниями – синтезируемы. Компилятор одинаково рассматривает типы с 2-мя и 4-мя состояниями.

Синтез игнорирует значение по умолчанию типов с 2-мя состояниями

Типы с 2-мя состояниями, перед началом симуляции имеют значение по умолчанию логический 0. Синтез игнорирует начальное значение по умолчанию. Пост-синтезная модель не гарантирует нулевые значения.

Секция 8.2 представляет дополнительное рассмотрение начального значения по умолчанию типов с 2-мя состояниями.

3.4 Использование типов с 2-мя состояниями в RTL моделях

Симуляция типов с 2-мя состояниями отличается от типов с 4-мя состояниями. Начальное значение типов с 2-мя состояниями на момент симуляции – 0, отличается от типов с 4-мя состояниями, и состояние неопределённости или логическая ошибка также отличается. В этом разделе рассматривается использование типов с 2-мя состояниями в RTL моделях.

3.4.1 Характеристики типов с 2-мя состояниями

SystemVerilog добавляет типы с 2-мя состояниями
SystemVerilog добавляет несколько типов с 2-мя состояниями к языку Verilog: **bit**(1-бит), **byte**(8-бит), **shortint**(16-бит), **int**(32-бита) и **longint**(64-бита). Эти типы с 2-мя состояниями позволяют моделирование на абстрактном уровне, где значения с третьим состоянием редко применяются, и где схемотехнические условия, которые могут привести к непредсказуемым значениям, представленными логическим X, произойти не могут.

Отображение значений с 4-мя состояниями на значения с 2-мя состояниями

SystemVerilog позволяет смешивать типы с 2-мя и 4-мя состояниями внутри модуля. Verilog слабо-типизированный язык, и эта характеристика также справедлива для типов с 2-мя состояниями в SystemVerilog. Таки образом, можно присваивать значение с 4-мя состояниями типу с 2-мя состояниями. Когда это происходит, значение с 4-мя состояниями отображается на значение с 2-мя состояниями как показано в следующей таблице:

Table 3-1: Conversion of 4-state values to 2-state values

4-state Value	Converts To
0	0
1	1
Z	0
X	0

3.4.2 Типы с 2-мя состояниями против симулятора в режиме 2-х состояний

Режимы с 2-мя состояниями программного инструментария

Некоторые программные инструменты, особенно симуляторы, предлагают режим 2-х состояний, когда в проекте не требуется использовать значения Z или X. Такой режим позволяет симуляторам оптимизировать симуляцию структур данных и алгоритмов и ускорять процесс симуляции. Типы с 2-мя состояниями позволяют программным инструментам оптимизировать такие типы. Однако типы с 2-мя состояниями в SystemVerilog имеют важные преимущества над режимом 2-х состояний симулятора.

SystemVerilog стандартизует смешивание типов с 2-мя и с 4-мя состояниями

Программные инструменты, которые предоставляют режим 2-х состояний, обычно устанавливают алгоритмы такого режима через опцию. Зачастую вызов такой опции применяется ко всем файлам в списке вызывающей команды. Это затрудняет смешивать типы с 2-мя и 4-мя логическими состояниями. Некоторые программные инструменты предоставляют более гибкое управление, компилируя одни модули в режиме 2-х состояний, а другие в нормальном режиме 4-х состояний. Эти инструменты могут использовать специфичные прагмы или собственные механизмы для определения переменных внутри модуля для режима 2-х или 4-х состояний.
Все эти механизмы специфичны, и отличаются от одного программного инструмента к другому. Типы с 2-мя состояниями SystemVerilog дают разработчику стандартный способ определять, в какой части проекта следует использовать логику с 2-мя состояниями, а в какой с 4-мя.

В SystemVerilog, отображение 2-х состояний на 4 состояния стандартизовано

Как отображать логическое значение Z или X в значение с 2-мя состояниями, при симуляции в режиме 2-х состояний, есть специфика программного инструментария, которая не стандартизирована. Разные симуляторы отображают значения по разному. На пример, некоторые коммерческие симуляторы будут отображать логический X в 0, когда другие – логический X в 1. Использование разных алгоритмов разными программными инструментами означает, что результаты симуляции, одного и того же проекта, будут отличаться. Типы с 2-мя состояниями в SystemVerilog имеют стандартный алгоритм отображения, предоставляющий результаты всех программных инструментов.

В SystemVerilog инициализация 2-х состояний стандартизована

Другое отличие между режимами с 2-мя состояниями и типами с 2-мя состояниями это инициализация переменной значением с 2-мя состояниями. IEEE 1364 Verilog стандарт определяет, что симуляция переменных с 4-мя состояниями начинается с логического значения X, показывая, что переменная не инициализирована. Первый раз переменная с 4-мя состояниями будет инициализирована нулём или единицей в результате события во время симуляции. Большинство патентованных алгоритмов режима с 2-мя состояниями изменяют начальное значение переменной с 4-мя состояниями на 0 вместо X, но когда эта инициализация произойдёт, стандартом не определяется. Некоторые симуляторы в режиме с 2-мя состояниями будут устанавливать начальное значение переменной без события во время симуляции. Другие симуляторы будут устанавливать начальное значение переменной из X в 0 по событию на нулевой момент времени, которое может распространяться на другие конструкции. Такие различия в патентованных алгоритмах режима с 2-мя состояниями могут привести к различным результатам симуляции. При старте симуляции переменные с 2-мя состояниями в SystemVerilog имеют значение логический 0, без события симуляции. Это стандартное правило гарантирует чёткое поведение во всех программных инструментах (software tools).

2- состояния стандартизовано в SystemVerilog

Verilog выражения **casez** и **casex** могут быть под влиянием режимов симуляции с 2-мя состояниями. Выражение **casez** рассматривает логическое состояние Z как незначительное вместо высокоимпедансного. Выражение **casex** рассматривает логические состояния X и Z как незначительные. Когда используются патентованные алгоритмы режима с 2-мя состояниями, то нет

стандарта, определяющего как это повлияет на выражения **casez** и **casex.** Более того, когда режимы симуляции с 4-мя состояниями меняют поведение внутри одного инструмента (tool), другой инструмент, который может не иметь режима с 2-мя состояниями, может интерпретировать поведение одной и той же модели по другому. В стандарте SystemVerilog определена семантика типов с 2-мя состояниями, которая предоставляет определённое поведение со всеми программными инструментами.

3.4.3 Использование типов с 2-мя состояниями в выражениях case

На абстрактном уровне RTL моделирования, логическое состояние X часто используется как флаг внутри модели, чтобы показать неопределённое состояние. На пример, стиль моделирования с выражением **case** (Verilog) создаёт ветку default, в которой выходам назначается логическое значение X, как показано в следующем фрагменте кода:

```
case (State)
  RESET:   Next = WAITE;
  WAITE:   Next = LOAD;
  LOAD:    Next = DONE;
  DONE:    Next = WAITE;
  default: Next = 4'bx; // unknown state
endcase
```

Присваивание по умолчанию логической X служит двум целям. Синтез рассматривает логическое X, назначенное по умолчанию, как специальный флаг, показывающий, что для любого другого условия выходное значение не важно. Синтез будет оптимизировать логику декодирования пунктов case, не принимая во внимание значения, которые попадают в ветку default, тем самым предоставляя лучшую оптимизацию для явно определённых случаев case.

В симуляции, присваивание по умолчанию логической X служит как run-time ошибка, неожидаемое значение выражения case. Это может помочь в выявлении ошибок в RTL моделях. Однако, это преимущество теряется после синтеза, поскольку у пост-синтезной модели на выходе не будет значений X для неожидаемых значений выражения case.

Присваивание логической X переменной с 2-мя состояниями корректно. Однако, результатом присвоения логической X переменной будет 0 вместо X. Если State или Next переменные с 2-мя состояниями, и значение 0 можно использовать для State и Next, тогда возможность использовать X для

выявления ошибок на RTL уровне теряется. Присваивание логической X по умолчанию будет позволять синтезу оптимизировать логику декодирования для выражения case. Пост-синтезное поведение проекта, не будет тем же самым, потому что оптимизированное декодирование не будет выдавать нулевой результат для неожидаемых значений выражения case.

3.5 Ослабление правил типизации

Verilog ограничивает использование переменных и сетей (net)

В Verilog, существуют семантические ограничения по использованию таких типов переменных как **reg,** и типов net как **wire**. Когда использовать **reg** и когда использовать **wire** базируется на контексте использования сигнала в модели. Переменная должна использоваться в процедурных блоках **initial** и **always**, net используется в непрерывных присваиваниях, объектах модуля или примитива.

Эти ограничения на использование типа зачастую разочаровывают инженеров, кто впервые изучает язык Verilog. Ограничения также создают трудность рассмотрения модели от абстрактного системного уровня к RTL и к вентильному уровню, как контекст изменений модели.

SystemVerilog ослабляет ограничения на использование переменных

SystemVerilog упрощает использование типа, ослаблением правил где переменная может быть использована. Переменная может получать значение не более чем одним из следующих способов:

- Присваивание значения в процедурных блоках **initial** или **always**
- Присваивание значения в процедурных блоках **always_comb**, **always_ff** или **always_latch**. Эти процедурные блоки обсуждаются в главе 6.
- Присваивание значения в выражении непрерывного назначения.
- Получение значения от портов output или inout модуля или примитива.

Большая часть сигналов может быть объявлена как logic или bit

Эти правила по использованию переменных позволяют большенству сигналов быть объявленными как переменная. Нет необходимости менять тип сигнала по мере эволюции модели от системного уровня к RTL и к вентильному уровню.

Следующий простой пример показывает использование переменных в соответствии с правилами ослабления типа.

Example 3-1: Relaxed usage of variables

```
module compare (output logic        lt, eq, gt,
                input  logic [63:0] a, b );

  always @(a, b)
    if (a < b) lt = 1'b1;      // procedural assignments
  else         lt = 1'b0;

  assign gt = (a > b);         // continuous assignments

  comparator u1 (eq, a, b);    // module instance

endmodule

module comparator (output logic  eq,
                   input  [63:0] a, b);
  always @(a, b)
    eq = (a==b);
endmodule
```

Ограничения на переменные могут предотвращать ошибки в проекте

переменные не могут управляться несколькими источниками

Ограничения SystemVerilog на использование переменных

Важно отметить, что хотя SystemVerilog разрешает использовать переменные в местах, где Verilog не разрешает, SystemVerilog всё же имеет некоторые ограничения на использование переменных.

В SystemVerilog считается ошибкой запись в одну переменную от нескольких портов output или нескольких выражений непрерывного назначения, или комбинационных выражений с выражениями непрерывного назначения.

Причиной этих ограничений является то, что переменные не имеют встроенной функциональности определять конечное значение, когда два или более устройств управляет одним и тем же выходом. Только типы net в Verilog, такие как **wire**, **wand** и **wor** имеют встроенные функции логического мульти-управления (multi-driver). (стандарт Verilog-2005 также имеет тип net **uwire,** который управляется одним драйвером).

Example 3-2: Illegal use of variables

```
module add_and_increment (output logic [63:0] sum,
                          output logic        carry,
                          input  logic [63:0] a, b );
  always @(a, b)
    sum = a + b;          // procedural assignment to sum

  assign sum = sum + 1;   // ERROR! sum is already being
                          // assigned a value

  look_ahead i1 (carry, a, b);  // module instance drives carry

  overflow_check i2 (carry, a, b); // ERROR! 2nd driver of carry
endmodule

module look_ahead (output wire         carry,
                   input  logic [63:0] a, b);
  ...
endmodule

module overflow_check (output wire         carry,
                       input  logic [63:0] a, b);
  ...
endmodule
```

 Используйте переменные для логики с единственным управлением, и nets для логики с мульти-управлением.

SystemVerilog ограничение, которое гласит, что переменные не могут принимать значения от нескольких источников, может помочь в предотвращении ошибок в проекте. Сигналу в проекте следует иметь один источник. Единственным источником может быть процедурный блок, единственное выражение непрерывного назначения, или единственный порт output/inout модуля или примитива. Второй источник, ненамеренно подсоединённый к тому же сигналу, будет определяться как ошибка, потому как каждая переменная может иметь только один источник.

В SystemVerilog, в одну переменную могут записывать несколько процедурных блоков **always,** которые рассматриваются как несколько источников. Это условие обеспечивает обратную совместимость с языком Verilog. Глава 6 представляет три новых типа процедурных блока: **always_comb**, **always_latch** и **always_ff**. Эти новые процедурные блоки имеют ограничение на использование переменной только одним блоком. Возникает необходимость проверки, что объявленный сигнал как переменная имеет единственный источник. Только сети (nets) могут иметь несколько источников, таких как несколько выражений непрерывного назначения и/или соединения к нескольким портам модуля или объектам примитива. Такие сигнала проекта как шина данных или шина адреса, которые могут управляться несколькими устройствами, следует объявлять как тип net, такой как **wire**. Двунаправленные порты модуля, которые используются как input и output, также должны быть объявлены как тип net.

Также корректна запись в автоматическую переменную из выражения непрерывного назначения или выходного порта модуля. Только статические переменные могут быть непрерывно назначены или подсоединяться к выходному порту. Статические переменные необходимы симулятору для непрерывной записи. Автоматическим переменным нет необходимости существовать в течение всего времени симуляции.

3.6 Знаковые и беззнаковые модификаторы

Знаковые типы Verilog-1995

Первый стандарт IEEE Verilog, Verilog-1995, имел один знаковый тип, объявляемый с ключевым словом **integer**. Этот тип имел фиксированный размер 32 бита. Из-за этого, и некоторых ограничений литеральных значений, Verilog 1995 был ограничен в знаковых операциях над 32 битными векторами. Знаковые операции могли бы производиться над векторами ручным тестированием и манипуляцией знаковым битом (способ, применяемый на аппаратном уровне), но это требует большого объёма кода, и появления ошибок кодирования, которые трудно обнаружить.

Знаковые типы Verilog

В стандарт IEEE Verilog-2001 добавлено несколько важных расширений для знаковых арифметических операций над типом и размером вектора. Расширение позволяет объявлять тип как **signed.** Этот модификатор отменяет определение по умолчанию беззнаковых типов в Verilog. На пример:

```
reg [63:0] u; // unsigned 64-bit variable
reg signed [63:0] s; // signed 64-bit variable
```

Знаковые и беззнаковые типы SystemVerilog

В SystemVerilog добавлены новые типы, которые знаковые по умолчанию. Это типы: **byte, shortint, int, и longint**. SystemVerilog предоставляет механизм явно отменять знаковое поведение этих типов, используя ключевое слово **unsigned.**

```
int s_int; // signed 32-bit variable
int unsigned u_int; // unsigned 32-bit variable
```

знаковые объявления в SystemVerilog это не тоже самое, как в C.

В языке C разрешено использовать ключевые слова **signed** или **unsigned** до или после типа.

```
unsigned int u1;  /* legal C declaration */
int unsigned u2;  /* legal C declaration */
```

Verilog размещает ключевое слово **signed** (в Verilog нет ключевого слова **unsigned**) после объявления типа:

```
reg signed [31:0] s;  // Verilog declaration
```

В SystemVerilog, ключевые слова **signed** или **unsigned** размещаются после объявления типа. Это совместимо с Verilog, но отличается от C.

```
int unsigned u;  // SystemVerilog declaration
```

3.7 Статические и автоматические переменные

Переменные Verilog-1995 статические

В стандарте Verilog-1995, все переменные статические, в связи с тем, что используются для моделирования аппаратного обеспечения, природа которого также статична.
Verilog-2001 использует автоматические переменные в задачах и функциях

Стандарт Verilog-2001 добавляет возможность определять переменные в задачах и функциях как **automatic,** что означает, область хранения переменной выделяется динамически по необходимости программным инструментарием, и освобождается, если больше не нужна. К автоматическим переменным, также обращаются как к динамическим, предназначенным для представления программ верификации в тестбенче, или на абстрактном системном уровне, уровне транзакций или моделях с шинной функциональностью. Ещё одно использование автоматических переменных это реентерабельная задача, задача может быть вызвана, пока предыдущий вызов задачи всё ещё работает.

Автоматические переменные, также позволяют кодировать рекурсивные функции, когда функция вызывает сама себя. Каждый раз при вызове задачи или функции создаётся новая область для переменной. При выходе из задачи или функции область разрушается. Следующий пример показывает сложение элементов массива. Младший и старший адреса элементов массива передаются как аргументы. Чтобы сложить элементы массива, функция вызывается рекурсивно. В этом примере аргументы lo и hi автоматические, также как внутренняя переменная mid. Каждый рекурсивный вызов выделяет новые переменные.

```
function automatic int b_add (int lo, hi);
  int mid = (lo + hi + 1) >> 1;
  if (lo + 1 != hi)
    return(b_add(lo,(mid-1)) + b_add(mid,hi));
  else
    return(array[lo] + array[hi]);
endfunction
```

В Verilog, автоматические переменные объявляются объявлением задачи или функции как автоматической. Все переменные в автоматической задаче или функции динамические.

SystemVerilog добавляет объявления статической и автоматической переменной

В SystemVerilog появляется возможность объявлять статическую и автоматическую переменные. Добавляется ключевое слово **static**, и любая переменная может быть явно объявлена как **static** или **automatic**. Такое объявление может находиться внутри задач, функций, блоков **begin…end**, или блоков **fork…join.** Обратите внимание, переменные не могут быть явно

объявлены как **static** или **automatic** на уровне модуля. На уровне модуля все переменные статические.
Следующий фрагмент кода показывает явные автоматические объявления в статической функции:

```
function int count_ones (input [31:0] data);
  automatic logic [31:0] count = 0;
  automatic logic [31:0] temp = data;

  for (int i=0; i<=32; i++) begin
    if (temp[0]) count++;
    temp >>= 1;
  end
  return count;
endfunction
```

Следующий пример показана статическая переменная в автоматической задаче. Автоматические задачи часто используются в верификации, чтобы тестовый код вызвал задачу, пока предыдущий вызов всё ещё исполняется. Этот пример проверяет значение на ошибки, и инкрементирует счётчик ошибок когда обнаруживает ошибку. Если бы переменная error_count была бы автоматической, как другие переменные этой задачи, она бы создавалась каждый раз при вызове задачи, и хранила бы счётчик ошибок только для этой задачи. Статическая переменная error_count сохраняет своё значения от одного вызова задачи к другому, и таким образом хранит общее число ошибок.

```
typedef struct packed {...} packet_t;
task automatic check_results
  (input  packet_t sent, received);
   output int      total_errors);
  static int error_count;
  ...
  if (sent !== received) error_count++;
  total_errors = error_count;
endtask
```

Обратная совместимость

По умолчанию, для области хранения, в SystemVerilog существует обратная совместимость с Verilog. В модулях, блоках **begin…end**, блоках **fork…join**, и неавтоматических задачах и функциях, все области хранения по умолчанию

статические, если явно не объявлены как автоматические. Это также как статические области хранения в Verilog модулях, **begin…end** или **fork…join** блоках и неавтоматических задачах и функциях. Если задача или функция объявлена как **automatic,** область хранения для переменных будет автоматической, если явно не объявлена как статическая. Также как в Verilog, где вся область хранения в автоматической задаче или функции является автоматической.

3.7.1 Инициализация статической и автоматической переменной

in-line инициализация переменной Verilog

В Verilog можно инициализировать переменные в строке, если они объявлены на уровне модуля. Переменные, объявленные в задачах, функциях и блоках begin..end или fork… join, не могут быть инициализированы в строке (иметь начальное значение как часть объявления переменной).

in-line инициализация переменной SystemVerilog

Инициализация автоматических переменных

В SystemVerilog можно объявлять переменные внутри задач и функций с in-line инициализацией. Переменная, объявленная в неавтоматической задаче или функции, будет по умолчанию статической. Начальное значение в строке присваивается один раз перед симуляцией. Вызовы задачи или функции не инициализируют переменную заново.

инициализация статических переменных в задаче или функции не рассматривается как синтезируемая, и может не поддерживаться в некоторых программных инструментах (software tools).

Статические переменные инициализируются только один раз

Следующий пример работать корректно не будет. count_ones является статической функцией, и всё пространство хранения, внутри функции, также статическое, если явно не объявлено как **automatic**. В этом примере, переменная count будет иметь начальное значение 0 при первом вызове функции. Переменная не будет инициализирована заново при следующем вызове функции. Статическая переменная будет хранить значение от

предыдущего вызова в счётчике count. Статическая переменная temp, в отличии от значения data, будет иметь значение 0 при первом вызове функции. Потому что, инициализация в строке происходит в момент времени ноль, а не в момент вызова функции.

```
function int count_ones (input [31:0] data);
  logic [31:0] count = 0;   // initialized once
  logic [31:0] temp = data; // initialized once

  for (int i=0; i<=32; i++) begin
    if (temp[0]) count++;
    temp >>= 1;
  end
  return(count);
endfunction
```

Автоматические переменные инициализируются при каждом вызове

Переменная, явно объявленная как **automatic,** в неавтоматической задаче или функции будет динамически создаваться каждый раз при входе в задачу или функцию, и существовать до тех пор, пока задача или функция существуют. in-line инициализация будет происходить при каждом вызове задачи или функции. Следующая версия функции count_ones будет работать корректно, потому что автоматические переменные count и temp инициализируются каждый раз при вызове функции.

```
function int count_ones (input [31:0] data);
  automatic logic [31:0] count = 0;
  automatic logic [31:0] temp = data;

  for (int i=0; i<=32; i++) begin
    if (temp[0]) count++;
    temp >>= 1;
  end
  return(count);
endfunction
```

Переменная, объявленная в автоматической задаче или функции, будет автоматической по умолчанию. Пространство хранения для переменной будет динамически создаваться при каждом входе в задачу или функцию, и разрушаться при выходе из задачи или функции. in-line инициализация будет

происходить при каждом входе в задачу или функцию, и будет создаваться новое пространство хранения.

3.7.2 Рекомендации по синтезу для автоматических переменных

Динамическое пространство хранения автоматических переменных может быть использовано в тестбенчах и аппаратных моделях. Чтобы быть синтезируемыми, в аппаратной модели, автоматические переменные следует использовать только во временных пространствах хранения, которые не выходят за рамки задачи, функции или процедурного блока.

Инициализация статической переменной не синтезируема, инициализация автоматической переменной синтезируема.

Инициализация статической переменной не синтезируема, и зарезервирована для использования в тестбенче и абстрактных моделях с шинной функциональностью.

in-line инициализация автоматической переменной синтезируема. Пример с функцией count_ones, показанный ранее в этой главе, в секции 3.7, соответствует этим критериям синтеза. Автоматические переменные count и temp используются только внутри функции, и значения переменных используются только в текущем вызове функции.

in-line инициализация переменных объявленных с квалификатором const также синтезируема. Секция 3.10 освещает объявления const.

3.7.3 Рекомендации по использованию статических и автоматических переменных

Следующие рекомендации будут вспомогательными при принятии решения, когда использовать статические и автоматические переменные.

- В блоках **always** или **initial**, использовать статические переменные без in-line инициализации, и автоматические переменные, с in-line инициализацией. Использование автоматических переменных с in-line инициализацией будет давать более интуитивное поведение, потому что переменная будет инициализироваться каждый раз при выполнении блока.

- Если задача или функция реентерабельна, то она должна быть автоматической. Переменные также должны быть автоматическими, если нет необходимости сохранять значение от вызова к вызову. Как простой пример, когда нужно хранить число вызовов автоматической задачи или функции, переменная хранения должна быть статической.

- Если задача или функция представляет поведение единственной части аппаратного обеспечения, и нет необходимости в реентерабельности, то их следует объявлять статическими, и все переменные внутри задачи или функции тоже статическими.

3.8 Инициализация детерминированной переменной

3.8.1 Детерминированность инициализации

Инициализация переменной Verilog-1995

В языке Verilog, который стандартизирован в 1995, переменные не могут быть инициализированы на момент объявления, как это можно сделать в языке C. В процедурном блоке **initial,** необходимо присвоить начальное значение переменным. На пример:

```
integer i;    // declare a variable named i
integer j;    // declare a variable named j

initial
  i = 5;      // initialize i to 5
initial
  j = i;      // initialize j to the value of i
```

В Verilog-1995 инициализация может быть недетерминированной

Стандарт Verilog явно говорит, что порядок, в котором программный инструментарий выполняет несколько процедурных блоков **initial,** не детерминирован. Таким образом, в предыдущем примере, невозможно определить будет ли значение переменной i присвоено переменной j до инициализации i значением 5 или после. Если, в предыдущем примере, сначала переменной i присваивается значение 5, и затем переменной j присваивается значение переменной i, то единственный детерминированный способ инициализации это группирование обоих присвоений, в одном процедурном блоке **initial**, заключённых в операторные скобки **begin…end**. Выражения внутри **begin…end** выполняются в последовательности, заданной пользователем.

```
integer i;    // declare a variable named i
integer j;    // declare a variable named j

initial begin
  i = 5;    // initialize i to 5
  j = i;    // initialize j to the value of i
end
```

Инициализация переменной Verilog-2001

Стандарт Verilog-2001 добавляет удобный способ для инициализации переменных, следуя синтаксису языка C, определяя начальное значение переменной как часть объявления переменной. Предыдущий пример можно записать короче:

```
integer i = 5;    // declare and initialize i
integer j = i;    // declare and initialize j
```

Verilog инициализация не детерминирована

Verilog определяет семантику для in-line инициализации переменной так, как будто начальное значение присвоено в процедурном блоке **initial.** Это означает, что in-line инициализация будет происходить в неопределённом порядке, одновременно с событиями в другом процедурном блоке **initial** и процедурных блоках **always,** которые выполняются в нулевой момент времени симуляции. Это неопределённое поведение может привести к неожиданным результатам симуляции, как в следующем примере:

```
integer i = 5;    // declare and initialize i
integer j;        // declare a variable named j

initial
  j = i;      // initialize j to the value of i
```

В этом примере, было бы интуитивно ожидать, что первой инициализируется переменная i, и поэтому переменная j инициализировалась бы значением 5. Неопределённый порядок события, в стандарте Verilog, не гарантирует этого. По спецификации стандарта Verilog, переменной j будет присвоено значение переменной i до того как она была инициализирована, это означает, что j получит значение X вместо 5.

Порядок инициализации в SystemVerilog

in-line инициализация в SystemVerilog до нулевого момента времени

SystemVerilog расширяет семантику in-line инициализации переменной. SystemVerilog определяет, что все in-line инициализации начальных значений будут происходить в порядке выполнения событий при старте симуляции (нулевой момент времени). Это гарантирует, что процедурные блоки **initial** или **always** корректно прочитают значения переменных, с in-line инициализацией. Такое детерминированное поведение убирает неопределённость, которая может быть в стандарте Verilog.

in-line инициализация переменной, в SystemVerilog, не вызывает события симуляции.

in-line инициализация Verilog может вызвать событие

Существует важное различие между семантикой Verilog и семантикой SystemVerilog для in-line инициализации переменной. По семантическим правилам Verilog, in-line инициализация переменной будет выполнена в течение нулевого времени симуляции. Это значит, событие симуляции произойдёт, если начальное значение, присвоенное переменной, отличается от её текущего значения. Однако, текущее значение переменной не может быть известно с какой-либо определённостью, потому что in-line инициализация происходит в недетерминированном порядке с другими инициализациями – in-line или процедурными – которые выполняются в нулевой момент времени. Таким образом, исходя из семантики Verilog, in-line инициализация переменной может или не может являться пичиной событий симуляции на нулевой момент времени.

SystemVerilog инициализация не вызывает события

Семантика SystemVerilog изменяет поведение in-line инициализации переменной. in-line инициализация переменной происходит до нулевого момента симуляции. Поэтому, инициализация никогда не будет вызывать событие в течение симуляции.

SystemVerilog инициализация обратно совместима

Результаты симуляции, использующие семантику SystemVerilog, в норме, но не детерминированы результатам семантики инициализации Verilog.

```
logic resetN = 0; // declare & initialize reset

always @(posedge clock, negedge resetN)
  if (!resetN) count <= 0;  // active low reset
  else count <= count + 1;
```

in-line инициализация Verilog не детерминирована

Используя *недетерминированую* семантику Verilog, для in-line инициализации переменной, можно получить два разных результата симуляции.

- Симулятор может активировать процедурный блок **always** первым, до инициализации переменной resetN. Процедурный блок **always** отслеживает положительный переход сигнала clock или отрицательный переход сигнала resetN. Затем, находясь в нулевом моменте симуляции, сигнал resetN инициализируется нулём (переход из X в 0), процедурный блок **always** реагирует на событие, и сбрасывает счётчик на нулевом моменте симуляции.

- В соответствие с семантикой Verilog, симулятор может выполнять инициализацию resetN до активации процедурного блока. Затем, находясь в нулевом моменте симуляции, когда процедурный блок **always** активирован, он отслеживает положительный переход сигнала clock или отрицательный переход сигнала resetN. Когда инициализация resetN уже произошла, count не сработает в нулевой момент времени, вместо этого будет ожидание положительного фронта сигнала clock или отрицательного фронта сигнала resetN.

in-line инициализация SystemVerilog детерминирована

По правилам in-line инициализации стандарта Verilog возможны два варианта, описанные выше. SystemVerilog убирает эту неопределённость. SystemVerilog гарантирует, что in-line инициализация будет происходить первой, это значит, что только второй сценарий может иметь место для примера, показанного выше. Это поведение полностью обратно-совместимо со стандартом Verilog, но является детерминированным.

3.8.2 Инициализация последовательностной логики с асинхронными входами

Недетерминированный порядок инициализации переменной, в Verilog, может являться причиной неопределённого поведения симулятора для асинхронного сброса или предварительных установок в последовательностных моделях. Эта неопределённость может влиять на сбросы или предварительные установки, которые применяются в начале симуляции.

Example 3-3: Applying reset at simulation time zero with 2-state types

```
module counter (input  wire         clock, resetN,
                output logic [15:0] count);

  always @(posedge clock, negedge resetN)
    if (!resetN) count <= 0;  // active low reset
    else count <= count + 1;
endmodule

module test;
  wire [15:0] count;
  bit         clock;
  bit         resetN = 1;  // initialize reset to inactive value

  counter dut (clock, resetN, count);

  always #10 clock = ~clock;

  initial begin
    resetN = 0;     // assert active-low reset at time 0
    #2 resetN = 1;  // de-assert reset before posedge of clock
    $display("\n count=%0d (expect 0)\n", count);
    #1 $finish;
  end
endmodule
```

В примере выше, счётчик имеет асинхронный вход сброса. Активный уровень сброса низкий, это значит. Что счётчик следует сбросить в момент, когда resetN перейдёт в 0. Для того, чтобы сбросить счётчик в нулевой момент симуляции, вход resetN должен перейти в логический 0. Если resetN объявлена как тип с 2-мя состояниями, такой как **bit**, начальное значение этой переменной, по умолчанию, будет 0. Первый тест в тестбенче это проверка сброса установкой resetN в 0. Однако, когда resetN есть тип данных с 2-мя состояниями, начальное значение по умолчанию будет 0. Первый тест не вызовет события в симуляторе по resetN, поэтому список

чувствительности не среагирует на resetN и не сбросит счётчик в процедурном блоке.

Чтобы гарантировать, срабатывание по resetN, когда resetN переходит в 0, нужно объявить resetN с in-line инициализацией в логическую 1, т.е не активное состояние сброса.

```
bit  resetN = 1;   // initialize reset
```

Исходя из правил семантики Verilog, in-line инициализация выполняется в течение нулевого времени симуляции, в недетерминированном порядке с другими присвоениями в нулевой момент времени. В предыдущем примере возможны два варианта:

- in-line инициализация может выполняться первой, resetN устанавливается в 1, с последующей установкой resetN в 0 в процедурном блоке.

- Процедурное присвоение может выполниться первым, resetN устанавливается в 0, с последующей in-line инициализация resetN в 1. Переход в 0 не произойдёт, resetN будет 1.

SystemVerilog убирает эту неопределённость. В SystemVerilog, in-line инициализация происходит до нулевого времени симуляции. В примере, показанным выше, resetN всегда будет инициализироваться первым единицей, и затем выполняется присвоение в процедурном блоке с установкой resetN в 0.

 в тестбенчах следует инициализировать переменные неактивным состоянием.

Гарантия событий в нулевой момент времени

Детерминированное поведение in-line инициализации переменной гарантирует генерацию событий в нулевой момент симуляции. Если переменная инициализирована, с использованием in-line инициализации, в неактивное состояние, и затем устанавливается в активное состояние, используя процедурные блоки **initial** или **always**, семантика SystemVerilog гарантирует, что in-line инициализация будет происходить первой, с последующим присвоением начального значения в процедурном блоке.

В предыдущем примере, объявление и инициализация resetN выглядит как часть тестбенча, и процедурный блок **always**, представляющий счётчик, как часть RTL модели. В одном и том же модуле или в раздельных модулях, детерминированное поведение, для in-line инициализации переменной, гарантирует, что событие симуляции будет происходить в нулевой момент времени, если переменная инициализирована неактивным состоянием, используя in-line инициализацию, и затем изменение на активный уровень, в нулевой момент времени, используя присвоение в процедурном блоке. Не детерминированный порядок in-line инициализации, в Verilog, не гарантирует, что события будут происходить в нулевой момент времени симуляции.

3.9 Приведение типов

Verilog слабо типизирован

Verilog слабо типизированный язык, который позволяет значение одного типа присвоить переменной или сети другого типа. Когда присвоение сделано, значение преобразуется в новый тип, в соответствии с правилами определёнными в стандарте Verilog.

Приведение отличается от слабой типизации

SystemVerilog добавляет возможность приведения значения к другому типу. Приведение типа отличается от преобразования значения в процессе присваивания. С приведением типа значение может быть преобразовано к новому типу внутри выражения, без какого либо присваивания.

Verilog не имеет приведение типа

Стандарт Verilog 1995 не предоставлял способа приведения значения к другому типу. Verilog-2001 добавил ограниченную возможность приведения, которая может преобразовывать знаковые значения к беззнаковым, и беззнаковые к знаковым. Это преобразование использует системные функции **$signed** и **$unsigned**.

3.9.1 Статическое приведение (стадия компиляции)

SystemVerilog добавляет оператор приведения

SystemVerilog добавляет оператор приведения в язык Verilog. Этот оператор может быть использован чтобы привести значение одного типа к другому, как в языке C. Оператор приведения в SystemVerilog выходит за рамки языка C, однако, вектор может быть приведён к другому размеру, и знаковые значения приведены к беззнаковым или наоборот.

Для совместимости с существующим языком Verilog, синтаксис оператора приведения SystemVerilog отличается от C.

type casting **<type>'(<expression>)** приводит значение к любому типу, включая типы определяемые пользователем. На пример:

```
7+ int'(2.0 * 3.0);   // cast result of
                      // (2.0 * 3.0) to int,
                      //   then add to 7
```

size casting **<size>'(<expression>)** приводит значение к любому размеру вектора. На пример:

```
logic [15:0] a, b, y;

y = a + b**16'(2);   // cast literal value 2
                     //   to be 16 bits wide
```

sign casting **<sign>'(<expression>)** приводит значение к знаковому или беззнаковому. На пример:

```
shortint a, b;
int      y;

y = y - signed'({a,b}); // cast concatenation
                        //   result to a signed
                        //   value
```

Статическое приведение и проверка на ошибки.

Статическое приведение не имеет проверки на ошибки при выполнении

Оператор статического приведения это приведение типа на стадии компиляции. Выражение, которое нужно привести к определённому типу,

будет преобразовываться в процессе выполнения, без проверок этого выражения на границы типа к которому приводится. В следующем примере статическое приведение используется, чтобы инкрементировать значение переменной перечислителя на 1. Оператор статического приведения не проверяет результат state + 1 на правильность значения для перечисляемого типа next_state. Присваивая значения, которые выходят за границы next_state, используя статическое приведение, мы не увидим сообщения об ошибках ни на стадии компиляции, ни при выполнении (run-time). Поэтому, нужно позаботиться, чтобы неправильное значение не было присвоено переменной next_state.

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

always_comb begin
  if (state != S3)
    next_state = states_t'(state + 1);
  else
    next_state = S1;
end
```

3.9.2 Динамическое приведение

Стадия компиляции против динамического приведения

Оператор статического приведения, описанный выше, это приведение типа на стадии компиляции. Приведение происходит без проверки на правильность результата. Когда необходима проверка, SystemVerilog предоставляет новую системную функцию **$cast**, которая осуществляет проверку значения на стадии выполнения (run-time).

Системная функция $cast

системная функция **$cast** принимает два аргумента, переменная назначения и переменная источник. Синтаксис:

```
$cast( dest_var, source_exp );
```

На пример:

```
int radius, area;

always @(posedge clock)
  $cast(area, 3.154 * radius ** 2);
    // result of cast operation is cast to
    // the type of area
```

Недействительные приведения

$cast пытается присвоить выражение источника переменной назначения. Если присвоение недействительно, то сообщается о run-time ошибке, и переменная назначения не изменяется. Некоторые примеры недействительного приведения:

- Приведение **real** к **int,** когда значение числа типа **real** на много больше **int (**как в примере выше**).**

- Приведение значения к перечисляемому типу, когда значения нет среди значений в списке перечисляемого типа, как в следующем примере.

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

always_latch begin
  $cast(next_state, state + 1);
end
```

$cast может вызываться как задача

$cast может вызываться как задача как в примере выше. Когда **$cast** вызывается как задача, и приведение терпит неудачу, то получаем сообщение об ошибке в run-time, и переменная назначения не изменяется. В примере выше, не меняющаяся переменная приведёт к блокировки фнкциональности.

$cast может возвращать флаг статуса

$cast может вызываться как системная функция. Функция возвращает флаг статуса, показывающий, успешным было приведение или нет. Если приведение было успешным, **$cast** возвращает 1, и 0 если потерпел неудачу.

Когда **$cast** вызывается как функция, сообщений о run-time ошибках не получаем.

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

int status;

always_comb begin
  status = $cast(next_state, state + 1);
  if (status == 0) // if cast did not succeed...
    next_state = S1;
end
```

Обратите внимание, что функция **$cast** не может быть использована с операторами, которые напрямую модифицируют выражение источника, такие как ++ или += .

```
$cast(next_state, ++state);   // ILLEGAL
```

Основное использование **$cast,** присвоение результата выражения переменным перечисляемого типа, которые сильно типизированы. Дополнительные примеры использования **$cast** представлены в секции 4.2.

3.9.3 Рекомендации по синтезу

 для синтеза используйте оператор приведения стадии компиляции (compile-time).

Статический оператор приведения стадии компиляции синтезируем. Динамическая системная функция **$cast** может не поддерживаться компиляторами синтеза. На тот момент, когда эта книга была написана, группа IEEE 1364.1 Verilog RTL synthesis standard ещё не определила рекомендации по синтезу для SystemVerilog. Основное правило: системные задачи и системные функции не рассматриваются как синтезируемые конструкции. Безопасный стиль кодирования для синтеза это использование оператора статического приведения.

3.10 Константы

Константы Verilog

Verilog предоставляет три типа констант: **parameter**, **specparam** и **localparam**. Вкратце:

- **parameter** это константа значение которой может быть переопределено в течение разработки, используя **defparam** или in-line параметр переопределения.

- **specparam** это константа, которая может быть переопределена во время разработки из файлов SDF.

- **localparam** константа времени разработки, которая не может быть переопределена напрямую, но для которой значение может базироваться на других константах.

Неправильно назначать константам иерархические ссылки

Все константы Verilog получают своё значение в время разработки.
В процессе разработки, программным инструментарием выстраивается иерархия проекта представленная объектами модуля. Некоторые программные инструменты имеют отдельную компиляцию и фазы разработки. Другие комбинируют компиляцию и разработку в один процесс. Может оказаться, что иерархия проекта ещё полностью не построена, и будет неправильным назначать константам **parameter**, **specparam** или **localparam** значения, которые получены из какого-либо места в иерархии проекта.

Константы нельзя использовать в автоматических задачах и функциях

Verilog имеет ограничения на объявление **parameter**, **specparam** и **localparam** констант в модулях, статических задачах и статических функциях. Неправильно объявлять эти константы в автоматической задаче и функции, или в блоках **begin…end** или **fork…join**.

C- подобное объявление константы

SystemVerilog добавляет возможность объявлять любую переменную как константу, используя ключевое слово **const. const** это форма константы без

присвоенного значения, до завершения разработки. Константа получает своё значение после разработки. Константа может:

- Быть объявленной в динамических контекстах таких как автоматические задачи и функции.
- Значение может быть присвоено от net или переменной вместо константного выражения.
- Значение может быть присвоено от объекта, который определён в иерархии проекта.

Объявление константы должно включать тип. Любые типы переменной Verilog или SystemVerilog могут быть определены как const, включая перечисляемые типы, и типы определяемые пользователем.

```
const logic [23:0] C1 = 7; // 24-bit constant
const int C2 = 15;         // 32-bit constant
const real C3 = 3.14;      // real constant
const C4 = 5;            // ERROR, no type
```

const может быть использован в автоматических задачах и функциях

Константа, по существу, есть переменная, которая может быть только инициализирована. Константа получает своё значение в процессе выполнения (run-time), константа может быть объявлена в автоматической задаче или функции, а также в модулях или статических задачах и функциях. Переменные, объявленные в блоках begin…end или fork…join, также могут быть объявлены как const.

```
task automatic C;
  const int N = 5;  // N is a constant
  ...
endtask
```

3.11 Резюме

В этой главе представлена область видимости единицы компиляции. Правильное использование объявлений области видимости единицы

компиляции облегчает моделирование функциональности в более краткой форме. Основное использование объявлений области видимости единицы компиляции, это определение новых типов с использованием **typedef**.

SystemVerilog расширяет возможность специфицировать логические значения, упрощая присваивание значений, которые легко масштабируются к любому размеру вектора. Расширения по текстозамещению, в `define, предоставляют новые возможности макросам в моделях и тестбенчах Verilog.

Также, SystemVerilog добавляет новые переменные с 2-мя состояниями: **bit, byte, shortint, int**, и **longint**. Эти типы переменных позволяют моделирование проекта на высоком уровне абстракции, используя значения с 2-мя состояниями. Семантические правила, для значений с 2-мя состояниями, определены так , что все программные инструменты будут интерпретировать и выполнять модели Verilog, используя логику с 2-мя состояниями одинаковым способом. Также добавлены новые типы **shortreal** и **logic**. Расширена инициализация переменных, что сокращает неопределённости, которые существуют в стандарте Verilog. Это также гарантирует, что все типы программных инструментов (software tools), будут интерпретировать модели SystemVerilog одинаковым способом. SystemVerilog расширил возможность объявления статических и автоматических переменных на различных уровнях иерархии проекта. Расширения включают возможность объявлять константы в блоках **begin…end** и в автоматических задачах и функциях.

Следующая глава продолжает тему типов SystemVerilog, освещая типы определяемые пользователем и перечисляемые типы.