Глава 4

Типы определяемые пользователем и перечисления в SystemVerilog

В SystemVerilog пользователи могут определять новые сети (net) и типы переменных, что является важным расширением для языка Verilog. Типы, определяемые пользователем, позволяют моделировать сложный дизайн на более абстрактном уровне, который является точным и синтезируемым. Использование типов определяемых пользователем, функциональность проекта может быть смоделирована в меньшем количестве кода, с добавлением таких преимуществ как само-документация и лёгкость чтения кода.

Расширения, представленные в этой главе включают:

- Использование **typedef** для создания типов определяемых пользователем
- Использование **enum** для создания перечесляемых типов
- Работа со значениями перечесляемых типов

4.1 Типы определяемые пользователем

Язык Verilog не предоставляет для пользователя механизм расширения типов net и переменных. Существующие типы в Verilog полезны для RTL и моделирования на вентильном уровне, они не предоставляют C – подобные типы переменных которые могут быть использованы на более высоких уровнях абстракции. SystemVerilog добавляет новые типы для моделирования на системном, и архитектурном уровне. Так же SystemVerilog

добавляет возможность для пользователя определять новые net (сети) и типы переменных.

Typedef определяет пользовательский тип

В SystemVerilog типы определяемые пользователем создаются с использованием ключевого слова **typedef**. Новые типы создаются от существующих типов. Когда новый тип определён, переменные нового типа могут быть объявлены. На пример:

```
typedef int unsigned uint;
...
uint a, b;  // two variables of type uint
```

4.1.1 Локальные определения typedef

Локальное использование typedef

Типы, определяемые пользователем, могут быть определены локально, в пакете, или внешне, в области видимости единицы компиляции. Когда тип определяемый пользователем используется внутри определённой части проекта, **typedef** используется внутри модуля или интерфейса представляющего часть проекта. Интерфейсы представлены в главе 10. В следующем фрагменте кода, объявляется тип nibble определяемый пользователем, который используется для объявления переменных внутри модуля alu. Поскольку тип nibble определён локально, только alu модуль может видеть его. Другие модули или интерфейсы проекта могут использовать тот же идентификатор nibble для своих целей без каких-либо конфликтов с локальным определением **typedef** в модуле alu.

```
module alu (...);

  typedef logic [3:0] nibble;

  nibble opA, opB;  // variables of the
                    // nibble type

  nibble [7:0] data; // a 32-bit vector made
                     // from 8 nibble types
  ...
endmodule
```

4.1.2 Разделяемые (shared) определения typedef

Typedef определения в пакетах

Когда тип определяемый пользователем используется во многих различных моделях, определение **typedef** может быть объявлено в пакете. К этим определениям можно обращаться напрямую, или импортировать в каждый модуль, интерфейс или программный блок который использует типы определяемые пользователем. Использование пакетов обсуждалось в главе 2, секция 2.1.

Typedef определения в $unit

Определение **typedef** может быть объявлено внешне, в области видимости еденицы компиляции. Внешние объявления создаются размещением **typedef** вне модуля, интерфейса или программного блока, как обсуждалось в главе 2, секция 2.2 на странице 14.

Пример 4-1 показывает использование пакетного определения **typedef** для создания типа определяемого пользователем dtype_t, который будет использоваться в проекте. Определение **typedef** находится под директивой

условной компиляции **`ifdef** , которая определяет dtype_t как тип **bit** с двумя состояниями или как тип **logic** с четырьмя состояниями.

Example 4-1: Directly referencing typedef definitions from a package

```
package chip_types;
  `ifdef TWO_STATE
    typedef bit dtype_t;
  `else
    typedef logic dtype_t;
  `endif
endpackage

module counter
(output chip_types::dtype_t [15:0] count,
 input  chip_types::dtype_t clock, resetN);

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

Импортирование определений пакета в $unit

Также возможно импортировать пакетные определения в пространство единицы компиляции $unit. Это может быть полезным, когда много портов модуля являются пользовательскими типами, и становится утомительным обращаться напрямую через имя пакета к каждому порту. Пример 4-2 показывает импортирование пакетного определения в пространство $unit.

Example 4-2: Importing package typedef definitions into $unit

```
package chip_types;
  `ifdef TWO_STATE
    typedef bit dtype_t;
  `else
    typedef logic dtype_t;
  `endif
endpackage

import chip_types::dtype_t;  // import definition into $unit

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

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

Если пакет содержит много typedefs, вместо импортирования конкретных пунктов пакета в пространство $unit, пакет может быть импортирован в $unit через символ *

```
import chip_types::*;  // wildcard import
```

4.1.3 Соглашение о именовании для пользовательских типов

Пользовательский тип может быть любым именем в языке Verilog. В больших проектах, и когда используются внешние объявления области видимости единицы компиляции, исходный код где определён новый пользовательский тип и исходный код где пользовательский тип используется, могут быть разделены большим количеством строк кода, или быть в разных файлах. Такое разделение и использование новых типов может создавать трудности в чтении и поддержке кода для больших проектов. Когда имя используется в исходном коде, может быть не очевидным, что это имя есть пользовательский тип.

Чтобы код был легко читаемым и поддерживаемым, пользовательский тип должен заканчиваться символами “_t”. Это соглашение об именовании использовалось в примере 4-1 выше, а также во многих примерах этой книги.

4.2 Перечисляемые типы

Перечисляемые типы предоставляют возможность объявлять абстрактную переменную, которая имеет список действительных значений. Каждое значение идентифицировано именем определённым пользователем, или меткой (label). В следующем примере, переменная RGB может иметь значения red, green, и blue:

```
enum {red,green,blue} RGB;
```

Verilog стиль для значений меток (labels)

Verilog использует константы вместо перечисляемых типов

В языке Verilog нет перечисляемых типов. Чтобы создать псевдо метки для данных, необходимо определить константу как **parameter**, и присвоить значение этой константе. Альтернативный вариант, использовать **`define** для создания макро имени с определённым значением.

Следующий пример показывает конечный автомат, использующий константы как **parameter** и **`define** макро имена: Параметры используются, чтобы определить состояния автомата, и макро имена используются чтобы определить команды которые декодируются конечным автоматом.

Example 4-3: State machine modeled with Verilog ‘define and parameter constants

```
`define FETCH 3'h0
`define WRITE 3'h1
`define ADD   3'h2
`define SUB   3'h3
`define MULT  3'h4
`define DIV   3'h5
`define SHIFT 3'h6
`define NOP   3'h7

module controller (output reg         read, write,
                   input  wire [2:0] instruction,
                   input  wire        clock, resetN);

  parameter WAITE = 0,
            LOAD  = 1,
            STORE = 2;

  reg [1:0] State, NextState;

  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always @(State) begin
    case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end

  always @(State, instruction) begin
    read = 0; write = 0;
    if (State == LOAD && instruction == `FETCH)
      read = 1;
    else if (State == STORE && instruction == `WRITE)
      write = 1;
  end
endmodule
```

Переменные, которые используют константные значения – State и NextState в предыдущем примере – должны быть объявлены как стандартные типы Verilog. Это значит, программный инструментарий не может ограничить действительные значения этих сигналов до значений констант. Нет ничего, что бы ограничивало State и NextState в примере выше, имеющим значение 3, или значение с одним или более бит установленным в X или Z. Сама модель должна добавлять проверку ограничений для значений. Как пример, применение прагмы “full case”, в операторе case, говорит о том, что возможно совпадение только с одним из перечисленных вариантов. Использование прагм синтеза не влияет на процесс симуляции, в результате получаем несоответствие между симуляцией и структурным дизайном созданным синтезем.

SystemVerilog стиль для значений меток (labels)

SystemVerilog добавляет перечисляемый тип в язык Verilog, используя ключевое слово **enum**, как в C. Форма объявления перечисляемого типа

```
enum {WAITE, LOAD, STORE} State, NextState;
```

Значения перечисления идентифицированы с метками

Перечисляемые типы делают код более читабельным, самодокументирующимся и более легким для отладки. Используя перечисляемые метки, получаем доступ к перечисляемым типам.

Пример 4-4 такой же как пример 4-3, но модифицирован с использованием перечисляемых типов SystemVerilog.

Example 4-4: State machine modeled with enumerated types

```
package chip_types;
  typedef enum {FETCH, WRITE, ADD, SUB,
                MULT, DIV, SHIFT, NOP  } instr_t;
endpackage

import chip_types::*;  // import package definitions into $unit

module controller (output logic   read, write,
                   input  instr_t instruction,
                   input  wire    clock, resetN);

  enum {WAITE, LOAD, STORE} State, NextState;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always_comb begin
    case (State)
      WAITE: NextState = LOAD;
      LOAD:  NextState = STORE;
      STORE: NextState = WAITE;
    endcase
  end

  always_comb begin
    read = 0; write = 0;
    if (State == LOAD && instruction == FETCH)
      read = 1;
    else if (State == STORE && instruction == WRITE)
      write = 1;
  end
endmodule
```

Перечислительные типы ограничивают значения

В этом примере , переменные State и NextState могут иметь только такие значения как WAITE, LOAD, STORE. Программный инструментарий (tools) будет интерпретировать ограничения значения для этих перечисляемых типов одинаково, включая симуляцию, синтез и верификацию.

В предыдущем примере SystemVerilog использует процедурные блоки **always_ff** и **always_comb**, детальное обсуждение которых будет в главе 6.

Импортирование перечисляемых типов из пакетов

Импортирование определения перечисляемого типа автоматически не импортирует значение меток

Когда определение перечисляемого типа импортируется из пакета, импортируется только имя типа. Значение меток в перечисляемом списке не импортируется и не делает его видимым в пространстве имён, в которое имя перечисляемого типа импортировано. Следующий фрагмент кода не работает

```
package chip_types;

  typedef enum {WAITE, LOAD, READY} states_t;

endpackage

module chip (...);

  import chip_types::states_t; // imports the
                               // typedef name,
                               // only

states_t  state, next_state;

always_ff @(posedge clk, negedge resetN)
  if (!resetN)
    state <= WAITE;    // ERROR: "WAITE" has not
                       // been imported!
  else
    state <= next_state;

  ...

endmodule
```

т.

Для того чтобы сделать метки перечисляемого типа видимыми, каждая метка должна быть явно импортирована, или пакет должен быть импортирован с символом * . Такой импорт делает имя перечисляемого типа и значение меток видимыми. На пример:

```
import chip_types::*; // wildcard import
```

4.2.1 Последовательности меток перечисляемого типа

SystemVerilog предоставляет два обозначения чтобы определить границу меток в списке перечисляемого типа.

Table 4-1: Specifying a sequence of enumerated list labels

state	creates a single label called **state**
state[N]	creates a sequence of labels, beginning with **state0**, **state1**, ... **stateN-1**
state[N:M]	creates a sequence of labels, beginning with **stateN**, and ending with **stateM**. If **N** is less than **M**, the sequence will increment from **N** to **M**. If **N** is greater than **M**, the sequence will decrement from **N** to **M**.

Следующий пример создаёт перечисляемый список с метками RESET, S0 до S4, и W6 до W9:

```
enum {RESET, S[5], W[6:9]} state;
```

4.2.2 Область видимости метки перечисляемого типа

Метки перечисления должны быть уникальны

Метки в списке перечисляемого типа должны быть уникальными внутри области видимости: еденицы компиляции, модулей, интерфейсов, программ, блоков **begin…end**, **fork…join**, tasks и functions.

Следующий фрагмент кода с ошибкой, так как метка GO используется дважды в одной и той же области видимости.

```
module FSM (...);
  enum {GO, STOP} fsm1_state;
  ...
  enum {WAITE, GO, DONE} fsm2_state;  // ERROR
  ...
```

Ошибка в предыдущем примере может быть исправлена размещением одного перечислителя в блоке **begin…end**, который имеет собственную область видимости.

```
module FSM (...);
  ...
  always @(posedge clock)
    begin: fsm1
      enum {STOP, GO} fsm1_state;
      ...
    end
  always @(posedge clock)
    begin: fsm2
      enum {WAITE, GO, DONE} fsm2_state;
      ...
    end
  ...
```

4.2.3 Значения перечисляемых типов

Метки перечисляемого типа имеют значение по умолчанию

По умолчанию, значение меток в списке перечисляемых типов есть тип interger **int** type**.** Значение первой метки списка есть 0, второй 1, третьей 2 и так далее.

Пользователи могут определять значения меток

SystemVerilog позволяет явно присваивать значение каждой метке в списке. Это позволяет перечисляемому типу быть более понятным при детальном описании каких-либо характеристик. На пример, конечный автомат может

быть явно смоделирован со значениями one-hot, one-cold, Johnson-count, Gray-code, или с другими значениями.

В следующем примере переменная state имеет значения ONE, FIVE, TEN. Каждая метка в списке представлена целым значением.

```
enum {ONE  = 1,
      FIVE = 5,
      TEN  = 10 } state;
```

Нет необходимости определять значение каждой метки в списке. Если значения не определены, то значение метки будет инкремент значения предыдущей метки. В следующем примере, значение метки A явно задано 1, метка B автоматически получает значение 2, C получает значение 3. Значение X явно задано 24, значения Y и Z 25 и 26 соответственно.

```
enum {A=1, B, C, X=24, Y, Z} list1;
```

Значение метки должно быть уникальным

Каждая метка в списке должна иметь уникальное значение. Если две метки имеют одинаковое значение, это является ошибкой. Следующий пример будет генерировать ошибку, так как C и D имеют одинаковое значение 3:

```
enum {A=1, B, C, D=3} list2;  // ERROR
```

4.2.4 Базовый тип перечислителя

По умолчанию базовый тип перечислителя есть **int** 32-bit**.**

Базовый тип может быть явно определён

Для того чтобы представить аппаратное обеспечение на более детальном уровне, в SystemVerilog можно явно задавать базовый тип перечислителя. На пример:

```
// enumerated type with a 1-bit wide,
// 2-state base type
enum bit {TRUE, FALSE} Boolean;

// enumerated type with a 2-bit wide,
// 4-state base type
enum logic [1:0] {WAITE, LOAD, READY} state;
```

Размер значения enum

Если базовый тип назначен явно, то размер типа метки должен соответствовать базовому типу.

```
enum logic [2:0] {WAITE = 3'b001,
                  LOAD  = 3'b010,
                  READY = 3'b100} state;
```

Будет ошибкой, если метке присвоить значение, размер которого отличается от размера базового типа перечислителя. Следующий пример некорректен. **enum** имеет по умолчанию базовый тип **int**. Присвоение 3-bit значений является ошибкой.

```
enum {WAITE = 3'b001,          // ERROR!
      LOAD  = 3'b010,
      READY = 3'b100} state;
```

Также является ошибкой, если количество меток в списке больше чем размер базового типа может представить.

```
enum logic {A=1'b0, B, C} list5;
  // ERROR: too many labels for 1-bit size
```

Перечислительные типы с 4-мя состояниями

Если базовый тип перечислителя есть тип 4-х состояний (4-state type), то меткам можно присваивать значения X или Z.

```
enum logic {ON=1'b1, OFF=1'bz} out;
```

Если метке присвоено значение X или Z, следующая метка должна иметь значение явно назначенное. Автоматический инкремент значения следующей метки является ошибкой.

```
enum logic [1:0]
  {WAITE, ERR=2'bxx, LOAD, READY} state;
  // ERROR: cannot determine a value for LOAD
```

4.2.5 Типизированные и анонимные перечислители

Использование typedef при создании пользовательского типа с перечислителями

Перечисляемые типы могут быть объявлены как типы определяемые пользователем. Это предоставляет удобный способ объявлять несколько переменных или сетей (nets) с одинаковыми значениями. К перечисляемому типу, объявленному с использованием **typedef**, обращаются как к типизированному перечисляемому типу. Если **typedef** не используется, к перечисляемому типу обращаются как к анонимному перечисляемому типу.

```
typedef enum {WAITE, LOAD, READY} states_t;

states_t  state, next_state;
```

4.2.6 Операции с сильно типизированным перечисляемым типом

Большинство типов переменных слабо типизированы

Большинство типов переменных Verilog и SystemVerilog слабо типизированы, это означает, что любое значение любого типа может быть присвоено переменной. Значение будет автоматически преобразовано в тип переменной, в стандарте Verilog и SystemVerilog определены правила преобразования.

Перечисляемые типы сильно типизированы

Перечисляемые типы на половину сильно типизированы. Перечисляемый тип может быть назначен только:

- Метке из списка перечислителя
- Другому перечислителю такого же типа (объявленным с таким же определением typedef)
- Приведённому значению к типу typedef перечисляемого типа

Когда производится операция со значением перечисляемого типа, значение автоматически преобразуется в базовый тип и во внутреннее значение которое представляет метку в списке. Если базовый тип не объявлен явно, то базовый тип и метки будут **int**.

В следующем примере:

```
typedef enum {WAITE, LOAD, READY} states_t;
states_t state, next_state;
int foo;
```

WAITE будет представлено как **int** со значением 0, LOAD как **int** со значением 1, и READY как **int** со значением 2.

Следующая операция над перечисляемым типом является правильной:

```
state = next_state;  // legal operation
```

Обе переменные state и next_state являются переменными одного типа (states_t). Значение одной переменной перечисляемого типа может быть назначено другой переменной такого же типа.

Выражение ниже так же является правильным. Перечисляемый тип state представлен как базовый тип **int**, к которому прибавляется целое 1. Результат операции есть целое значение **int,** которое присваивается переменной типа **int.**

```
foo = state + 1;   // legal operation
```

Преобразование в предыдущем примере допустимо. Ошибка будет в том случае, если значение которое не является перечисляемым типом, присваивается переменной перечисляемого типа. На пример:

```
state = foo + 1;   // ERROR: illegal assignment
```

В этом примере результатом сложения foo + 1 является **int**, а state имеет тип state_t.

Следующие примеры так же являются неправильными:

```
state = state + 1;      // illegal operation
state++;                // illegal operation
next_state += state;    // illegal operation
```

Перечисляемый тип state представлен как базовый тип **int**, к которому прибавляется целое 1. Результат операции есть целое значение **int.** Прямое присваивание результата типа **int** переменной типа state_t есть является ошибкой.

4.2.7 Выражения приведения типов в перечислителях

Приведения к перечисляемому типу

Результат операции может быть приведён к перечисляемому типу, и затем присвоен переменной такого же типа. Для приведения может использоваться оператор приведения или системная динамическая функция **$cast** (см раздел 3.9 глава 3).

```
typedef enum {WAITE, LOAD, READY} states_t;
states_t state, next_state;

next_state = states_t'(state++);     // legal
$cast(next_state, state + 1);        // legal
```

Использование оператора приведения

Как обсуждалось ранее в разделе 3.9, существует важное различие между использованием оператора cast и системной динамической функцией. Оператор cast всегда выполняет приведение и присвоение. Присваемое значение будет лежать в легальных границах перечисляемого типа. Используем предыдущий пример для state и next_state, если state имело значение READY, то есть 2, инкрементируя его получаем целое значение 3. Присвоение этого значения к next_state выходит за границы значений внутри перечисляемого списка next_state.

Выход за границы значения является причиной неопределённого поведения. Различное программное обеспечение может по разному опрерировать с таким значением. Если такое значение присвоено, фактическое значение сохранённое в перечисляемом типе на стадии симуляции RTL модели может отличаться после стадии синтеза.

Чтобы избежать неоднозначного поведения, никогда не присваивайте значение, вышедшее за границы, переменной перечиляемого типа. Статичный оператор cast не может обнаруживать значения вышедшие за границы (out-of-range), потому что оператор cast не проверяет ошибки на стадии run-time.

Использование системной функции $cast

Динамическая системная функция **$cast** проверяет выражение на правильность результата, перед тем как изменить переменную. В предыдущем примере, если результат инкрементирования state выходит за границы для next_state, вызов системной функции $cast(next_state, state + 1) не изменит next_state, и run-time ошибка будет зафиксирована.

Два способа использования cast позволяют разработчику найти компромиссный вариант при разработке проекта. Динамический cast безопасный потому что проверяет ошибки на стадии run-time. Однако такой способ требует накладных расходов, которые влияют на производительность программного инструментария (software tools). Так же системная функция **$cast** может быть не синтезируема. Оператор cast не проверяет ошибки, но положительно влияет на производительность стадии run-time.

Пользователи могут выбирать, какой метод использовать, основываясь на природе модели. Если известно, что значения не будут выходить за границы, то используется статический cast (compile-time). Если же существует возможность выхода значений за границы, тогда безопаснее использовать системную функцию **$cast**. Обратите внимание, что выражение **assert** также можно использовать для выявления значений вышедших за границу, но **assert** не предотвращает эту ошибку. Assertions рассматриваются в книге *SystemVerilog for Verification*.

4.2.8 Специальные системные задачи и методы для перечисляемых типов

Итерация через список перечисляемого типа

SystemVerilog предоставляет несколько встроенных функций для перебора значений списка перечисляемого типа, обращение к которым осуществляется как к методам. Эти методы автоматически рассматривают природу типа перечислений как с полу-сильной типизацией (semi-strongly typed), что позволяет легко делать такие действия как инкремент следующего значения в списке, переход в начало списка или переход в конец списка. Используя эти методы, нет необходимости знать о метках или значениях в списке.

Методы перечисляемого типа используют синтаксис C++

Эти специальные методы для работы с перечисляемыми списками вызываются в манере методов класса C++. Имя метода ставится в конце имени перечисляемого типа с разделителем.

<enum_variable_name>.first – возвращает значение первого члена списка

<enum_variable_name>.last - возвращает значение последнего члена списка

<enum_variable_name>.next(<N>) - возвращает значение следующего члена списка. Целое значение может быть передано как аргумент в **next.** В этом случае возвращается значение N члена в списке. Когда конец списка достигнут, то следующим значением будет значение первого члена списка.

Если текущее значение не является членом списка, то возвращается значение первого члена в списке.

<enum_variable_name>.prev(<N>) – возвращает значение предыдущего члена в списке. Как и в методе **next,** целое значение может быть передано как аргумент в **prev**. В этом случае возвращается N предыдущее значение в списке, начиная с позиции текущего значения. Когда начало списка достигнуто, то следующим значением будет значение последнего члена списка. Если текущее значение не является членом списка, то возвращается значение последнего члена в списке.

<enum_variable_name>.num – возвращает количество меток в списке.

<enum_variable_name>.name – возвращает символьное значение метки. Если значение не является членом перечисления, метод **name** возвращает пустую строку.

Пример 4-5 иллюстрирует автомат состояний, который используя методы перечисления, меняет эти состояния. Это простой пример счётчика от 0 до 15, где:

- Начальное значение выхода in_sync 0; устанавливается, когда счётчик достигает 8; in_sync очищается если счётчик достигает 0.
- Если флаги compare и synced оба ложь, счётчик остаётся в текущем состоянии.
- Если флаги compare и synced оба истина, счётчик инкрементируется на 1.
- Если флаг compare истина, а флаг synced ложь, счётчик декрементируется на 2.

Example 4-5: Using special methods to iterate through enumerated type lists

```
module confidence_counter(input  logic synced, compare,
                                       resetN, clock,
                          output logic in_sync);

  enum {cnt[0:15]} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= cnt0;
    else         State <= Next;

  always_comb begin
    Next = State;  // default NextState value
    case (State)
      cnt0 :   if (compare &&  synced) Next = State.next;
      cnt1 : begin
               if (compare &&  synced) Next = State.next;
               if (compare && !synced) Next = State.first;
             end
      cnt15:   if (compare && !synced) Next = State.prev(2);
      default begin
               if (compare &&  synced) Next = State.next;
               if (compare && !synced) Next = State.prev(2);
              end
    endcase
  end

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) in_sync <= 0;
    else begin
      if (State == cnt8) in_sync <= 1;
      if (State == cnt0) in_sync <= 0;
    end

endmodule
```

В предыдущем примере используются процедурные блоки **always_ff** и **always_comb**. Эти процедурные блоки более подробно обсуждаются в главе 6.

4.2.9 Вывод на печать перечисляемых типов

Вывод на печать значений и меток перечисляемого типа

Значения перечисляемого типа могут выводиться на консоль как внутреннее значение метки или как имя метки. Доступ к имени метки осуществляется через метод **name.** Этот метод возвращает строку содержащую имя. Эта строка может быть передана в **$display** для вывода на консоль.

Example 4-6: Printing enumerated types by value and by name

```
module FSM (input  logic       clock, resetN,
            output logic [3:0] control);

  enum logic [2:0] {WAITE=3'b001,
                    LOAD =3'b010,
                    READY=3'b010} State, Next;

  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= Next;

  always_comb begin
    $display("\nCurrent state is %s (%b)", State.name, State);
    case (State)
      WAITE: Next = LOAD;
      LOAD:  Next = READY;
      READY: Next = WAITE;
    endcase
    $display("Next state will be %s (%b)", Next.name, Next);
  end

  assign control = State;

endmodule
```

4.3 Резюме

С-подобный **typedef** позволяет пользователям определять новые типы от предопределённых типов или от других типов определённых пользователем. Типы определённые пользователем могут использоваться как порты модуля и как аргументы задач и функций.

Перечисляемые типы позволяют объявлять переменные с ограниченными значениями, и представлять эти значения абстрактными метками вместо логических значений аппаратуры. Перечисляемые типы позволяют моделировать более абстрактный уровень больших проектов с меньшим количеством кода. Детали аппаратной реализации могут быть добавлены в объявления перечисляемого типа.

SystemVerilog также добавляет тип **class**, разрешающий объектно-ориентированный стиль моделирования. Объекты класса и объектно-ориентированное программирование предназначены для верификации и не синтезируются. Детали и примеры классов SystemVerilog могут быть найдены в книге *SystemVerilog for Verification*.