тРасширения языка System verilog , обеспечивающие улучшение представления аппаратуры на RTL уровне

the typical features of an HVL that distinguish it from a Hardware Description Language such as Verilog or VHDL are

- Constrained-random stimulus generation
- Functional coverage
- Higher-level structures, especially Object Oriented Programming
- Multi-threading and interprocess communication
- Support for HDL types such as Verilog's 4-state values
- Tight integration with event-simulator for control of the design

There are many other useful features, but these allow you to create test-benches at a higher level of abstraction than you are able to achieve with an HDL or a programming language such as C.

3.1Расширения типов данных

shortint	2state SystemVerilog data type, 16-bit signed intege1	2-стабильное кодирование может обеспечить экономию времени моделирования. Но главная цель- обеспечение совместимости с С-кодом при написании тестов
int	2state SystemVerilog data type, 32-bit signed intege1	
longint	2state SystemVerilog data type, 64bit signed intege1	
byte	2state SystemVerilog data type, 18-bit signed intege1 or ASCII character	
bit	2state SystemVerilog data type, user defined vector size	
logic	4state SystemVerilog data type, size	The keywords **logic** and **reg** are equivalent types
reg	4-state Verilog data type, user defined vector	The keywords **logic** and **reg** are equivalent types
integer	4-state Verilog data type, 32-bit signed integer	The difference between **int** and **integer** is that **int** is a 2-state type and **integer** is a 4-state type.
time	4-state Verilog data type, 64-bit signed integer	

Real and shortreal data types **real**7 data type is the same as a C **double**. The **shortreal** is the same as a C **float**.

Void data type represents nonexistent data. This type can be specified as the return type of functions to indicate no return value.

Chandle data type

The **chandle** data type represents storage for pointers passed using the DPI. The size of a value of this data type is platform dependent, but shall be at least large enough to hold a pointer on the machine in which the tool is running.

The syntax to declare a handle is as follows:

chandle `variable_name ;`

String data type -
Verilog supports string literals, but only at the lexical level. In Verilog, string literals behave like packed arrays of a width that is a multiple of 8 bits. SystemVerilog also supports
the **string** data type to which a string literal can be assigned. When using the **string** data type
instead of an integral variable, strings can be of arbitrary length and no truncation occurs
Более подробное изучение cfмостоятельно- стандарт SYSTEM-VERILOG,(4.7)

Типы,определяемые пользователем

type_declaration ::=
typedef data_type type_identifier { variable_dimension } **;**
| **typedef** interface_instance_identifier **.** type_identifier type_identifier **;**
| **typedef** [**enum** | **struct** | **union** | **class**] type_identifier **;**

Пример.
typedef int `intP;`
Тогда объект декларируется:
`intP a, b;`

typedef enum `type_identifier;`
typedef struct `type_identifier;`
typedef union `type_identifier;`
typedef class `type_identifier;`
typedef `type_identifier;`

Перечислимые типы

data_type ::=
| **enum** [enum_base_type] **{** enum_name_declaration { **,** enum_name_declaration } **}**
enum_base_type ::=
integer_atom_type [signing]
| integer_vector_type [signing] [packed_dimension]
| type_identifier [packed_dimension]
enum_name_declaration ::=
enum_identifier [**[** integral_number [**:** integral_number] **]**] [**=** constant_expression]

Примеры
а) некодированное перечисление
enum `{red, yellow, green} light1, light2; // anonymous int type`
б) частично-кодированное
enum integer `{IDLE=0, XX='x, S1='b01, S2='b10} state, next;`
Значения могут быть отнесены к integer types и инкрементироваться от начального значения.
enum `{bronze=3, silver, gold} medal; // silver=4, gold=5`

Можно ввести перечислимый тип через typedef и вводить переменные как экземпляры типа (особенно удобно в пакете)
typedef enum `{NO, YES}` **`boolean`**`;`
`boolean myvar1,myvar2; // named type`

СТРУКТУРЫ и объединения

Structured_data_type ::=
...
struct_union [**packed** [signing]] **{** struct_union_member { struct_union_member } **}**
{ packed_dimension }
struct_union_member ::=
{ attribute_instance } [random_qualifier] data_type_or_void list_of_variable_decl_assignments **;**
data_type_or_void ::= data_type | **void**
struct_union ::= **struct** | **union** [**tagged**]

Пример

```
Typegef struct { bit [7:0] opcode; bit [23:0] addr; }IR1; // anonymous structure
// defines variable IR
Ir1 ir;
IR.opcode = 1; // set field in IR.
```

Объединения

В аппаратуре интерпретация совокупности битов может зависеть от других битов Например команда процессора может иметь несколько полей. Например при прямой непосредственной часть команды представляет код операции, а другая – операнд, причем это поле может трактоваться по разному в целочисленных командах и командах плавающей точки

В следующем примере одно и то же поле может трактоваться и как целое и как данное плавающей точки

```
typedef union { int i; real f; enum p} num_u;
num_u un;
un.f = 0.0; // set un in floating point format
un.i==1; //set un in integer format
g=un.p;
```

объединения полезны если часто требуется читать и писать в регистр в разных форматах
Однако не рекомендуется перегружаться только для экономии памяти Можно сэкономить несколько байт ценой переусложнения структуры данных. Лучше в подобных случаях сосдать «плоский» класс с дополнительным параметром-лискриминантом, который будет указыватькакой вид передачи задается и соответственно какие поля читать, писать и рандомизировать

.

2.11.3 Packed structures

SystemVerilog allows you more control in how data is laid out in memory by using packed structures. A packed structure is stored as a contiguous set of bits with no unused space. The `struct` for a pixel, shown above, used three data values, so it is stored in three longwords, even though it only needs three bytes. You can specify that it should be packed into the smallest possible space.

Example 2-30 Packed structure

```
typedef struct packed {bit [7:0] r, g, b;} pixel_p_s;
pixel_p_s my_pixel;
```

Packed structures are used when the underlying bits represent a numerical value, or when you are trying to reduce memory usage. For example, you could pack together several bit-fields to make a single register. Or you might pack together the opcode and operand fields to make a value that contains an entire processor instruction.

<имя родительского типа> <спецификация подтипа> <список объектов>
typedef <имя родительского типа> <спецификация подтипа> <имя наследующего типа>
<<имя родительского типа> <список объектов>

enum {<список значений>}x,y,z
typedef {<список значений>} my_type;
my_type x,y,z;
Struct{<список спецификаций полей>}x;
typedef {<список значений>} struct_type;
struct_type a;

<ТИП элемента> “[< диапазон>]” <имя массива> “[< диапазон>]
Typedef <ТИП элемента> [< диапазон>] array_type
Array_type my_array

массивы фиксированного размера

SystemVerilog предлагает несколько видов массивов вместо одномерных, массивов фиксированного размера Verilog-1995. Многие усовершенствования были внесены в и в эти классические массивы.

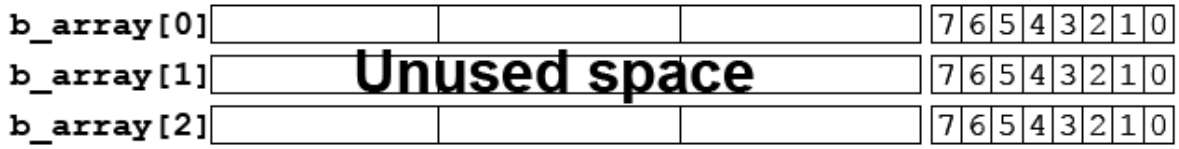
<ТИП элемента> “[< диапазон>]” <имя массива> “[< диапазон>]
Typedef <ТИП элемента> [< диапазон>] array_type
Array_type my_array

Объявление и инициализация массива фиксированного размера
Verilog требует чтобы высшие и низшие пределов массива были приведены в декларации. Почти все массивы используют нижний индекс 0, поэтому SystemVerilog позволяет просто дать размер массива, похожий на C:
Пример Объявление массивов фиксированного размера
Int lo_hi [0:15] / / 16 целых чисел [0] .. [15]
Int c_style [16]; / / 16 целых чисел [0] .. [15]
Logic [15:0] lo_hi [0:15]

Вы можете создавать многомерные массивы фиксированного размера, указав Размеры после имени переменной. Это касается распакованного массива, упакованные массивы обсуждаются позже. В Следующем примере создается несколько двумерных массивов целых чисел, 8 x4, и устанавливает последнюю элемент 1. Многомерные массивы были введены в Verilog-2001, но компактные формы декларации являются новыми.

Int array2 [0:7] [0:3] / / обычная декларации
Int array3 [8] [4] / / Компактная декларация
array2 [7] [3] = 1; / / Установить последний элемент массива

SystemVerilog сохраняет каждый элемент в длинном слове (32 бит) . Таким образом, байт, SHORTINT и Int все хранятся в одном длинном слове, в то время LONGINT хранится в двух longwords. (Симуляторы часто хранят четырехзначные данные такие, как logic и integer e в двух или более longwords).
Пример декларация Распакованного массива
bit [7:0] b [3] // int b [3]// int [8] b [3]; b[4][2]=
/ / Распакованный
B[5][1]<=



Методы для массивов
FOR и Foreach

initial begin
bit [31:0] src[5], dst[5];
for (int i=0; i<$size(src); i++)
src[i] = i;
foreach (dst[j])
dst[j] = src[j] * 2; // dst doubles src values
end
Отметим, что синтаксис FOREACH для многомерных массивов может отличаться от Ваших ожиданий. Спски индексов пишутся не в разных скобках[i][j] – они запbсываются в одной скобке через запятые
Пример
// Initialize and step through a multidimensional array
initial begin
$display("Initial value:");
foreach (md[i,j]) // Yes, this is the right syntax!!!
$display("md[%0d][%0d] = %0d", i, j, md[i][j]);
$display("New value:");
md = '{{9, 8, 7}, 3{5}}; // Replicate last 3 values
foreach (md[i,j]) // Yes, this is the right syntax
$display("md[%0d][%0d] = %0d", i, j, md[i][j]);
end
на дисплей выводится:
Initial value:
md[0][0] = 0
md[0][1] = 1

md[0][2] = 2
md[1][0] = 3
md[1][1] = 4
md[1][2] = 5
New value:
md[0][0] = 9
md[0][1] = 8
md[0][2] = 7
md[1][0] = 5
md[1][1] = 5
md[1][2] = 5

COPY и Compare

```
initial begin
bit [31:0] src[5] = '{0,1,2,3,4},
dst[5] = '{5,4,3,2,1};
// Aggregate compare the two arrays
if (src==dst)
$display("src == dst");
else
$display("src != dst");
// Aggregate copy all src values to dst
dst = src;
// Change just one element
src[0] = 5;
// Are all values equal (no!)
$display("src %s dst", (src == dst) ? "==" : "!=");
// Are last elements 1-4 equal (yes!)
$display("src[1:4] %s dst[1:4]",
(src[1:4] == dst[1:4]) ? "==" : "!=");
End
```

Нельзя выполнять арифметические операции целиком над массивами – следует организовывать циклы. Для логических операций типа XOR возможны циклы, а также использование упакованных массивов

Упакованные массивы

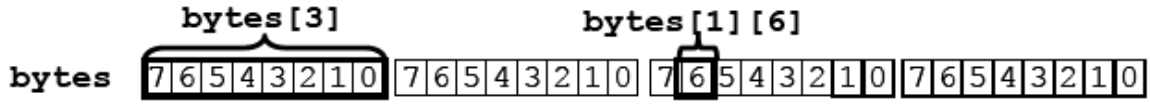
Для некоторых типов данных, вы можете как получить доступ к полной записи, а также разделить его на более мелкие элементы. Например, у вас может быть 32-разрядный регистр который иногда нужно рассматривать как четыре 8-битных значения и в другое время как единое целое без знака. В SystemVerilog упакованный массив рассматривается как и как массив и как одно значение. Он хранится в виде непрерывного набора битов, без неиспользованного пространства, в отличие от распакованного массива.

примеры Упакованных массивов

для Упакованных битов и слов размерности указаны как часть типа перед именем переменной. Эти размерности должны быть указаны в [lo: hi] формате. Переменная bites это упакованный массив из четырех байтов, которые хранятся в одном длинном слове

```
bit [3:0] [7:0] bytes;  // 4 bytes packed into 32-bits
bytes = 32'hdead_beef;
$displayh(bytes,,        // Show all 32-bits
          bytes[3],      // most significant byte "de"
          bytes[3][7]);  // most significant bit  "1"
```

Figure 2-2 Packed array layout

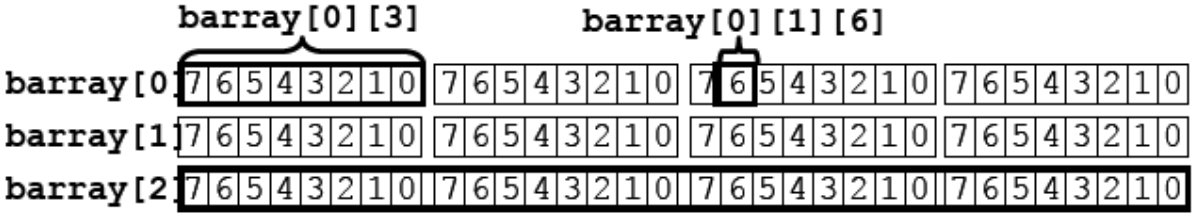


Example 2-14 Declaration for mixed packed/unpacked array

```
bit [3:0] [7:0] barray [3];     // Packed: 3x32-bit
barray[0] = 32'h0123_4567;
barray[0][3] = 8'h01;
barray[0][1][6] = 1'b1;
```

The variable **bytes** is a packed array of four bytes, which are stored in a single longword. **barray** is an array of three of these elements.

Figure 2-3 Packed arrays



Выбор между упакованным и распакованным массивом

Какой вы должны выбрать - или упакованный или распакованный массив? Упакованный массив удобен, если вам нужно конвертировать в скаляры и обратно. Например, вам может быть нужны обращения к содержимому памяти и как к байту и как длинному слово. Массив barray, приведенный выше поможет справиться с этим требованием. Только массивы фиксированного размера могут быть упакованы, а динамические массивы, ассоциативные массивы, или очередей (как показано ниже) не могут .
Если вам нужно ждать изменений в массиве, вы должны использовать упакованные
массиве. Возможно, ваш testbench, будет возбуждаться при изменении значений в памяти, и вы хотите использовать оператор @. Но это только допустимо только скаляров и упакованных массивы. Так в предыдущих примерах, можно допустить в списках чувствительности
Переменную LW и barray[0], но не весь массив barray , если ‘nj dct ;t ye;yj cktletn
расширить список чувствительности : @ (barray [0] or barray [1] or barray [2]).

Динамические массивы

Основной тип массива Verilog это массив фиксированного размера: его размер устанавливается во время компиляции. Но что, если вы не знаете размер массива, до его использования ? Вы можете выбрать количество передачь случайно
от 1000 до 100000, но вы не хотите использовать фиксированный размер массива, он
будет наполовину пуст. SystemVerilog обеспечивает динамический массив, которому может быть модифицирован размера во время моделирования.
Динамический массив объявляется с пустыми индексами слово []. Это означает,
что вы не хотите, чтобы размер массива определялся во время компиляции, а вместо этого вы указываете что это делается во время выполнения. Массив изначально пуст, так что вы должны вызвать оператор new[] и выделить место, записав число записей в квадратные скобки. Если Вы передаете имя массива в оператор new [ ], значения копируются
в новые элементоы.

```
int dyn[], d2[]; // Empty dynamic arrays
initial begin
dyn = new[5]; // Allocate 5 elements
foreach (dyn[j])
dyn[j] = j; // Initialize the elements
d2 = dyn; // Copy a dynamic array
d2[0] = 5; // Modify the copy
$display(dyn[0],d2[0]); // See both values (0 & 5)
dyn = new[20](dyn); // Expand and copy
dyn = new[100]; // Allocate 100 new integers
// Old values are lost
dyn.delete; // Delete all elements
end
```

Функция $size Размер возвращает размер как массива фиксированного размера так и динамический массив.
Для Динамических массивов определены несколько специализированных процедур, таких как delete и size. Последняя функция возвращает размер, но не работает с массивами фиксированного размера .
Если вы хотите объявить константный массив, но не хотите беспокоиться
о подсчете количества элементов, используйте динамический массив
Пример есть 9 масок для 8 бит, но вы должны позволить SystemVerilog
считать их, вместо того, чтобы задать фиксированный размер массива и случайно выбрали неправильный размер 8.
Пример 2-16 Использование динамического массива для списка неизвестной длины

```
bit [7:0] mask [] = ‘{8'b0000_0000, 8'b0000_0001,
8'b0000_0011, 8'b0000_0111,
8'b0000_1111, 8'b0001_1111,
8'b0011_1111, 8'b0111_1111,
8'b1111_1111};
```

можете передавать значения между массивами фиксированного размера и динамические массивы если они имеют одинаковый тип , например int. Вы можно копировать динамический массив
в массив с фиксированной если они имеют одинаковое число элементов.
При копировании массива фиксированного размера в динамический массив SystemVerilog

вызывает конструктор new [] xnj выделить место, а затем копирует значения.

Очереди

SystemVerilog вводит новый тип данных, очереди, который обеспечивает легкий
поиск и сортировку в структурах и универсальны, как связанный список для определения очереди используется символ $ в квадратных скобках.
Как и любой динамический массив, очередь может увеличиваться и уменьшаться, но в очереди возможно легко добавлять и удалять элементы в любом месте.

Пример операции с очередями: добавляет иудаляет значения из очереди.

```
int j = 1,
b[$] = {3,4},
q[$] = {0,2,5}; // {0,2,5} Initial queue
initial begin
q.insert(1, j); // {0,1,2,5} Insert 1 before 2
q.insert(3, b); // {0,1,2,3,4,5} Insert whole q.
q.delete(1); // {0,2,3,4,5} Delete elem. #1
// The rest of these are fast
q.push_front(6); // {6,0,2,3,4,5} Insert at front
j = q.pop_back; // {6,0,2,3,4} j = 5
q.push_back(8); // {6,0,2,3,4,8} Insert at back
j = q.pop_front; // {0,2,3,4,8} j = 6
foreach (q[i])
$display(q[i]);
end
```

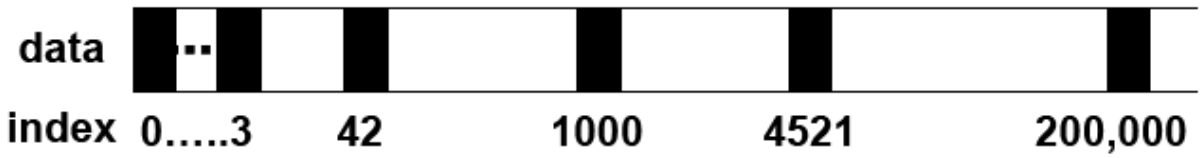
При создании очереди, SystemVerilog на самом деле выделяет дополнительное пространство так что можно быстро добавить дополнительные элементы. Обратите внимание, что не нужно вызывать оператор new[] для очереди. Если вы добавляете столько элементов, что очереди не хватает места, SystemVerilog автоматически выделяет дополнительное пространство.
В результате, вы можете увеличивать или сокращать очереди без потери производительности что характерно для динамического массива.
очень эффективно добавлять (push) и удалять (pop) элементы в начало и конец
очереди, что занимает одинаковый промежуток времени независимо от длины очереди. Добавление и удаление элементов в середине медленнее, особенно для больших очередей, как как SystemVerilog должен сдвигать до половины элементов.
Вы можете скопировать содержимое очереди в фиксированный или динамический массив

.

Ассоциативные массивы

Динамические массивы хороши, если вы хотите, чтобы время от времени создавать большие массивы
Но если вы хотите создать что-то действительно огромное ? Возможно, вы моделируете процессор, который имеет диапазон адресов в несколько гигабайт. В типичном тесте
Процессор может обращаться только несколько сотен или тысяч ячеек памяти содержащие исполняемый кода и данные, поэтому выделение и инициализация гигабайт
хранения расточительно.
SystemVerilog предлагает ассоциативные массивы для хранения записей в разреженной матрице. Это означает, что при моделировапние очень большого адресного пространства SystemVerilog, выделяет память для элемента только когда вы обращаетесь к нему к нему. На Следующем рисунке , ассоциативный массив содержит значения 0:3, 42, 1000,
4521 и 200000. для хранения этих данных используется гораздо меньшая Память, чем было бы, необходимых для хранения фиксированных или динамических массивов с 200000 путями доступа (entryes)

Figure 2-4 Associative array



Пример показывает объявление, инициализация и пошаговое продвижение в ассоциативном массиве. Ассоциативные массивы объявляются с в соответствии с синтаксисом <имя >[*]. Однако надо помнить , что массив может быть проиндексирована
любы целым числом.

Example 2-18 Declaring, initializing, and using associative arrays

```
initial begin
  logic [63:0] assoc[*], idx = 1;

  // Initialize widely scattered values
  repeat (64) begin
    assoc[idx] = idx;
    idx = idx << 1;
  end

  // Step through all index values with foreach
  foreach (assoc[i])
    $display("assoc[%h] = %h", i, assoc[i]);

  // Step through all index values with functions
  if (assoc.first(idx))
    begin                    // Get first index
    do
      $display("assoc[%h]=%h", idx, assoc[idx]);
    while (assoc.next(idx));  // Get next index
  end

  // Find and delete the first element
  assoc.first(idx);
  assoc.delete(idx);
end
```

имеется ассоциативный массив, **assoc** с очень разбросанными элементами:
1, 2, 4, 8, 16 и т.д. простой FOR-цикл не не может «пройти» по ним,
необходимо использовать FOREACH цикл по каждому элементу, или, если вы хотите более точного управления, вы можете использовать функции first
и next в цикле Do ... while. Эти функции изменяют Индекс аргумента, и возвращает 0 или 1 в зависимости от того, остаются ли элементы в массиве.
Ассоциативные массивы также может быть адресованы при использовании индекса строки, похожие на хэш массивв.

Пример 2-19 считывает пары имя / значение из файла в
ассоциативный массив. Если попытаться прочитать из элементов, которые не были размещены ,
SystemVerilog возвращает 0 для компонентов двухзначного типа типа или X для 4—х значного типа.
Вы можете использовать функции **exis**t, чтобы проверить существует ли элемент, как показано ниже.
Пример Использование ассоциативного массива со строчным индексом

```
/*
Input file looks like:
  42   min_address
  1492 max_address
*/

int switch[string], min_address, max_address;
initial begin
  int i, r, file;
  string s;
  file = $fopen("switch.txt", "r");
  while (! $feof(file)) begin
    r = $fscanf(file, "%d %s", i, s);
    switch[s] = i;
  end
  $fclose(file);

  // Get the min address, default is 0
  mid_address = switch["min_address"];

  // Get the max address, default = 1000
  if (switch.exists("max_address"))
    max_address = switch["max_address"];
  else
    max_address = 1000;
end
```

Ассоциативный массив может быть сохранен симулятором в виде дерева. Эти дополнительные расходы приемлемы, когда нужно хранить массивы с широко
разнесенными значениями индекса, такие как пакеты индексированные с 32-битным адресом, или имеющие 64 - бита поле данных.

Связанные списки

SystemVerilog обеспечивает связанный список структур данных, аналогичную
Списку контейнеров STL (стандартная библиотека шаблонов). Контейнер определяется как параметризованный классов, а это означает, что он может быть настроен для хранения данных любого типа.

```
`include <List.vh>
...
List#(T) dl;      // dl is a List of 'T' elements
```

```
class List#(parameter type T);
   extern function new();
   extern function int size();
   extern function int empty();
   extern function void push_front( T value );
   extern function void push_back( T value );
   extern function T front();
   extern function T back();
   extern function void pop_front();
   extern function void pop_back();
   extern function List_Iterator#(T) start();
   extern function List_Iterator#(T) finish();
   extern function void insert( List_Iterator#(T) position, T value );
   extern function void insert_range( List_Iterator#(T) position,
                                      first, last );
   extern function void erase( List_Iterator#(T) position );
   extern function void erase_range( List_Iterator#(T) first, last );
   extern function void set( List_Iterator#(T) first, last );
   extern function void swap( List#(T) lst );
   extern function void clear();
   extern function void purge();
endclass
```

Теперь вы знаете, что связанный список в SystemVerilog определен,но , не используйте его.
C + + программисты, могут быть знакомы с версией STL, но в SystemVerilog
Очереди являются более эффективными и простыми в использовании.

Почтовые ящики

декларация объекта

Mailbox <name>

В сущности это очередь, но без права доступа к средним элементам. Но для объектов этого типа определен ряд полезных методов: put, get, которые кроме значений данных и сдвига возвращают признаки наличия данных.(рассмотрим позже)

методы массива

Методы для массивов

FOR и Foreach

```
initial begin
bit [31:0] src[5], dst[5];
for (int i=0; i<$size(src); i++)
src[i] = i;
foreach (dst[j])
dst[j] = src[j] * 2; // dst doubles src values
end
```

Отметим, что синтаксис FOREACH для многомерных массивов может отличаться от Ваших ожиданий. Спски индексов пишутся не в разных скобках[i][j] – они запbсываются в одной скобке через запятые
Пример

```
// Initialize and step through a multidimensional array
initial begin
$display("Initial value:");
foreach (md[i,j]) // Yes, this is the right syntax!!!
$display("md[%0d][%0d] = %0d", i, j, md[i][j]);
```

```
$display("New value:");
md = '{{9, 8, 7}, 3{5}}; // Replicate last 3 values
foreach (md[i,j]) // Yes, this is the right syntax
$display("md[%0d][%0d] = %0d", i, j, md[i][j]);
end
```

на дисплей выводится:

```
Initial value:
md[0][0] = 0
md[0][1] = 1
md[0][2] = 2
md[1][0] = 3
md[1][1] = 4
md[1][2] = 5
New value:
md[0][0] = 9
md[0][1] = 8
md[0][2] = 7
md[1][0] = 5
md[1][1] = 5
md[1][2] = 5
```

COPY и Compare

```
initial begin
bit [31:0] src[5] = '{0,1,2,3,4},
dst[5] = '{5,4,3,2,1};
// Aggregate compare the two arrays
if (src==dst)
$display("src == dst");
else
$display("src != dst");
// Aggregate copy all src values to dst
dst = src;
// Change just one element
src[0] = 5;
// Are all values equal (no!)
$display("src %s dst", (src == dst) ? "==" : "!=");
// Are last elements 1-4 equal (yes!)
$display("src[1:4] %s dst[1:4]",
(src[1:4] == dst[1:4]) ? "==" : "!=");
End
```

Нельзя выполнять арифметические операции целиком над массивами – следует организовывать циклы. Для логических операций типа XOR возможны циклы, а также использование упакованных массивов

Есть много методов массива, который можно использовать на распакованных массивах любых типов: стационарные, динамические, очередь, и ассоциативно. Эти процедуры могут быть и достаточно простыми простыми, например

предоставление текущего размера массива и сложными вплоть до сортировки элементов.

Методы редукции массивов

Основным методом редукции массива является преобразование его в скаляр.

Широко применяемым методом редукции является sum, которsq суммирует все значения в массиве

Пример Создание суммы массива

```
bit on[10];  // Array of single bits
int summ;

initial begin
  foreach (on[i])
    on[i] = i;   // on[i] gets 0 or 1

  // Print the single-bit sum
  $display("on.sum = %0d", on.sum); // on.sum = 1

  // Sum the values using 32-bits as summ is 32-bits
  summ = on.sum;
  $display("summ = %0d", summ);     // summ = 5

  // Compare the sum to a 32-bit value
  if (on.sum >=32'd5)               // True
    $display("sum has 5 or more 1's");
  end
```

Другие подобные методы редукции массива: and , or, и XOR.

Методы размещения массива

Каково наибольшее значение в массиве? Содержит ли массив определенную запись ? выполняется поиск данных в распакованном массива. Эти методы всегда возвращают . (очередь)

Пример 2-21 используется массив фиксированного размера, F [6], и очередь, Q [$]. Функции **min** и **max** находят наименьшее и наибольшее элементы в массиве. Обратите внимание, что они возвращаются в очередь, а не скаляр, как можно, ожидать. Эти методы работают для ассоциативных массивов. Метод **Unique** возвращает очереди содержащие уникальные значения из массива - повторяющиеся значения не включены.

Example 2-21 Array locator methods: `min, max, unique`

```
int f[6] = '{1,6,2,6,8,6};
int q[$] = '{1,3,5,7}, tq[$];

tq = q.min;                       // {1}
tq = q.max;                       // {7}
tq = f.unique;                    // {1,6,2,8}
```

Вы можете просматривать массив, используя цикл по каждому элементу, но SystemVerilog может сделать это за одну операцию с помощью метода размещения. Выражение **with**говорит SystemVerilog, как выполнить поиск.

методы с локатором:

FIND

```
int d[] = '{9,1,8,3,4,4}, tq[$];

// Find all elements greater than 3
tq = d.find with (item > 3);        // {9,8,4,4}
// Equivalent code
tq.delete;
foreach (d[i])
  if (d[i] > 3)
    tq.push_back(d[i]);

tq = d.find_index with (item > 3);  // {0,2,4}
tq = d.find_first with (item > 99); // {} - none found
tq = d.find_first_index with (item==8); // {2} d[2]=8
tq = d.find_last with (item==4);        // {4}
tq = d.find_last_index with (item==4);  // {6} d[6]=4
```

Выбор способа хранения

Дам некоторые рекомендации по выбору правильного типа хранения данных с учетом гибкости, объемом использования памяти, скорость и возможностей сортировк сортировки. Это всего лишь эмпирические правила, а результаты могут отличаться при симуляции.

Гибкость

Используйте массив фиксированного размера или динамический массив, если обращение выполняется с последовательными положительными числовыми индексами: 0, 1, 2, 3 ... Выберите фиксированный размер массива, если размер массива известен во время компиляции, а выбрать динамический массив используют , если размер не известен, во время исполнения. Например, переменный размер пакета легко хранить в динамическом массиве. Если вы пишете процедуры для управления массивами, предпочтительно использовать только динамические массивы, так как одна процедура будет работать при любом размер динамического Массива тех пор, пока подходит тип элемента (Int, String и т.д.) . Подобно можно передать в подпрограмму очереди любого размера , если тип элемента соответствует аргументу Очереди. Ассоциативные массивы также можно передавать независимо от размера.

В то же время , программа с фиксированным размером аргумента-массива принимает только массивы предопределенной длины.

ассоциативные массивы Выбирают при нестандартной индексации, таких как широко расставленных значениях индексов , в частности случайных значений данных или адреса. Ассоциативные массивы также могут быть использованы для моделирования CAM (Content-adressable memory).

Очереди, это хороший способ для хранения данных, гда число элементов растет
и сжимается много во время моделирования, такие как буфер обмена,
. И наконец, очереди удобны для поиска и сортировки.

Учет Использования памяти

Если вы хотите уменьшить использование памяти при моделировании, используйте элементов двузначных типов.

Желательно выбрать для данные размеры, кратные 32 бита, чтобы избежать
неиспользуемого пространства. Симуляторы обычно хранят все меньшие в 32-разрядных словах.
Например, Для массива из 1024 байт ¾ из памяти не используется, если симулятор сопоставиткаждому элемент 32-разрядное слово. Использование Упакованных массивов также может помочь сократить затраты на память.

Для массивов, которые содержат до тысячи элементов, выбираемый тип массива, не дает существенной разницы с точки зрения затрат памяти . Для массивов с тысяч до миллионов активных элементов массивы фиксированного размера и динамические являются наиболее эффективными для представления памяти. Но тестировщику может потребоваться пересмотреть свои алгоритмы, если нужны массивы с более чем миллионов активных элементов.

Очереди немного менее эффективны, чем массивы с фиксированным размером или динамический, потому что требуются дополнительные указатели. Однако, если ваш набор данных растет исокращается часто, то

при хранении в динамической памяти, вам придется «вручную» вызывать **new** [] для выделения памяти и копирования. Это дорогостоящая операция и будет снижать выгоды от использования динамической памяти.
Моделирование памяти большей, чем несколько мегабайт должно быть выполнено с использованием ассоциативного массива. Обратите внимание, что каждый элемент ассоциативного массива может занять в несколько раза больше памяти, чем элемент памяти в массиве фиксированного размер или динамическом из-за необходимости хранить указатель.

Скорость

Выберите тип массива на основе того, сколько раз осуществляется доступ течении цикла. Для операций чтения и записи немного, вы можете использовать любой тип, так как накладные расходы является незначительным по сравнению с моделированием DUT. При более частом доступе , егоРазмер и тип имеют значение.
Массивы Фиксированного размера и динамические хранятся в непрерывной памяти, так что любой Элемент может быть найден за одинаковое времz, независимо от размера массива.
Очереди обеспечивают почти то же самое время доступа, Первый и последний элементы могут быть выдвинуты и внесены почти без накладных расходов. Для Вставки и удаление элементов в середине требует сдвинуть вверх или вниз много элементов, чтобы освободить место. Если вам нужно вставлять новые элементы в большой очереди, ваш testbench может замедлиться, так что следует рассмотреть вопрос об изменении способа хранения новых элементов.
При чтении и записи ассоциативных массивов, симулятор должен искать элемент в памяти удовлетворяющий признаку . Стандарт не уточняет, как это делается, но популярны способы это хэш-таблиц и деревья. Это требует больше вычислений, чемс другими типами массивов , и, следовательно, ассоциативные массивы являются самыми медленными.

Сортировка

В SystemVerilog можно сортировать одномерный массив (фиксированного размера, динамические и ассоциативные массивы, плюс очереди), и при выборе следует оценить как Часто данные добавляются в массив. Если данные получены сразу, следует выбрать
массив фиксированного размера или динамический, так что достаточно выделить массив один раз.
Если Данные поступают медленно выбраем очередь, как добавление новых элементов в голову или хвост очень эффективно.
Если у вас есть сущности , которые являются непоследовательными, например '{1, 10, 11, 50}, и к тому же уникальны, следует хранить их в ассоциативном массив, используя данные в качестве индекса.
Использование подпрограмм first, next , и previous, можно выполнить поиск в ассоциативном массиве и найти последовательные значения. Списки являются двусвязанными, так что Вы можете найти значения как большие, так и меньше, чем текущее значение. Список наравне с ассоциативным массивом позволяет быстро удалить значения. Тем не менее, доступе к любой элемент данного индекса в ассоциативном массиве гораздо быстрее,.
Когда вам нужно проверить было ли данное значение написано, используйте функцию exist. Когда “закончено”с элементом, используйте Delete, чтобы удалить его из ассоциативного массива.

Выбор лучшей структурой данных

Вот несколько советов по выбору структуры данных.

 Для сетевых пакетов. Свойства: фиксированный размер, последовательный доступ. Используйте массив фиксированного размера или динамический в зависимости фиксированной или переменной длины пакета.

 SCOREBOARD(временный архив) Свойства: переменнгоый размера, доступ по значению. Обячно , используют очереди, так как Постоянно в процессе моделирования выполняется добавление и удаление элементов. Если можно дать каждой передаче
фиксированный идентификатор, например, 1, 2, 3 ..., вы можете использовать его в качестве индекса в очереди.. Если SCOREBOARD может иметь сотни элементов, и вы часто делаете вставки иудаляя из середины, ассоциативный массив может быть быстрее.

 Сортировка структур. Используют очереди, если данные поступают в предсказуемом порядка или ассоциативный массив, если порядок не определен. Если в SCOREBOARD никогда не нужно искать, а просто хранят ожидаемые значения луше
почтовый ящик, (будет рассматриваться дальше).

 моделирование очень большой памяти (больше миллиона записей). Если
Не требуется каждое место, используйте ассоциативный массив в качестве разреженной памяти. Если вам действительно нужно каждое место, попробуйте другой подход, при котором не требуется так много реальных данных. Не подходит? Обязательно используйте 2-значеное представление данных упакованные в 32-бит

 имена команд и значения из файла. Особенности: поиск по строке. Следует Прочтите строки из файла, а затем искать команды вассоциативном массиве используя команду как строковый индекс.

Целесообразно создать массив указателей , которые указывают на объекты, как будет показано при рассмотрении основ основ ООП.

Лекция 6

МОДИфИЦИРОВАННЫЕ версии базовых операторов

17.1.1. Комбинационный процедурный блок.

В отличие от обычного блока always, блок always_comb не требует указания специального списка чувствительности. Он создается автоматически и в него включаются все переменные, значение которых читается в процедурном блоке, за исключением тех, что объявлены в нем. В следующем примере оператор always_comb будет выполняться при каждом изменении переменных a или b:

```
always_comb
 if (!mode)
    y = a + b;
else
    y = a - b;
```

Существует различие в моделировании между always_comb и always. Первый выполняется один раз в нулевой момент моделирования, после активации всех процедурных блоков.

Verilog-2001 предлагает использовать в списке чувствительности блока always групповой символ @* или @(*). Однако это только более короткая запись и не дает таких преимуществ, как always_comb (листинг 17.2).

Листинг 17.2. Различие между always_comb и always

```
always @* begin                  // Обозначает @(data)
          a1 = data << 1;
          b1 = decode ();
...
end
always_comb begin                // Обозначает @(data, sel, c, d, e)
          a2 = data << 1;
          b2 = decode ();
...
end
function decode;                 // Функция не имеет входов
  begin
          case (sel)
            2'b01 : decode = d | e;
            2'b10 : decode = d & e;
            default : decode = c;
          endcase
  end
```

17.1.2. Последовательностный процедурный блок

Второй специализированный оператор always – always_latch. Это процедурный блок, моделирующий триггер-защелку.

```
always_latch
        if (enable) q <= d;
```

Пример

```
module register_reader (input clk, ready, resetN,
                        output logic [4:0] read_pointer);
                  logic enable; // internal enable signal for the counter
```

```
        logic overflow; // internal counter overflow flag
    always_latch begin // latch the ready input
        if (!resetN)
            enable <= 0;
        else if (ready)
            enable <= 1;
        else if (overflow)
            enable <= 0;
        end
    always @(posedge clk, negedge resetN) begin // 5-bit counter
        if (!resetN)
            {overflow,read_pointer} <= 0;
        else if (enable)
            {overflow,read_pointer} <= read_pointer + 1;
        end
endmodule
```

17.1.3. Последовательностный процедурный блок.

Блоки, описывающие последовательностную логику, могут моделироваться с помощью always_ff:

```
always_ff @(posedge clock, negedge resetN)
    if (!resetN) q <= 0;
    else q <= d;
```

Все сигналы в списке чувствительности должны быть записаны с указанием фронта posedge или negedge. Событийный контроль внутри блока не допускается.

Блоки always_comb, always_latch и always_ff являются синтезируемыми.

Новые операторы

Таблица 17.1. Операторы инкремента и декремента

Выражение	Операция	Описание
j = i++;	пост-инкремент	j получает значение i, после чего i увеличивается на 1
j = ++i;	пре-инкремент	i увеличивается на 1 и j получает обновленное значение i
j = i--;	пост-декремент	j получает значение i, после чего i уменьшается на 1
j = --i;	пре-декремент	i уменьшается на 1 и j получает обновленное значение i

Таблица 17.2. Новые операторы присвоения в SystemVerilog

Оператор	Описание
+=	Суммируется оператор с левой стороны выражения с оператором с правой стороны, затем выполняется присвоение
-=	Из оператора с левой стороны выражения вычитается значение с правой стороны, затем выполняется присвоение
*=	Умножение
/=	Деление оператора с левой стороны на значение выражения с правой
%=	Деление оператора с левой стороны на значение выражения с правой, присваивается остаток
&=	Побитовая операция И между левым и правым операндом
|=	Побитовая операция ИЛИ между левым и правым операндом
^=	Побитовая операция по модулю два между левым и правым операндом
<<=	Сдвиг оператора с левой стороны влево выражения на количество бит, задаваемых правым операндом
>>=	Сдвиг оператора с левой стороны вправо выражения на количество бит, задаваемых правым операндом
<<<=	Арифметический сдвиг оператора с левой стороны влево выражения на количество бит, задаваемых правым операндом
>>>=	Арифметический сдвиг оператора с левой стороны вправо выражения на количество бит, задаваемых правым операндом

17.3.3. Оператор inside.

SystemVerilog предлагает оператор inside, который предназначен для поиска значения переменной в заданном множестве:

```
logic [2:0] a;
        if (a inside {3'b001, 3'b010, 3'b100})
        ...
```

Такая форма упрощает запись множественных сравнений. Без оператора inside представленное выше выражение будет выглядеть следующим образом:

```
if ((a==3'b001) || (a==3'b010) || (a==3'b100))
        ...
```

В операторе inside множество значений для сравнения может быть представлено множеством сигналов

```
if ( data inside {bus1, bus2, bus3, bus4} )
        ...
```

или быть массивом

```
int d_array [0:1023];
if ( 13 inside {d_array} ).
```

В операторе inside значение Z можно заменять символом «?».

Циклы

FORVER

Декларация типа переменной цикла в декларации цикла

```
...
always_ff @(posedge clock) begin
  for (bit [4:0] i = 0; i <= 15; i++)
  ...
  end
always_ff @(posedge clock) begin
  for (int i = 1; i <= 1024; i += 1)
  ...
  end
```

Оператор do…..While

Операторы перехода

В Verilog для выхода из циклов и блоков использовался disable

```
// Поиск первого бита множества в заданном диапазоне битов
always @* begin
        begin: loop
          integer i;
          first_bit = 0;
          for (i=0; i<=63; i=i+1) begin: pass
                if (i < start_range)
                  disable pass;      // Продолжается работа блока loop
                if (i > end_range)
                  disable loop;      // Выход из блока loop
                if ( data[i] ) begin
                  first_bit = i;
                  disable loop;      // Выход из блока loop
                end
            end                      // Завершение блока pass
          end                        // Завершение блока loop
                // Остальные операторы, обрабатывающие данные
  ...
  end
```

В SystemVerilog добавляются операторы из языка Си: break, continue и return, что дает возможность создавать понятный и компактный код. Эти операторы могут быть применены и к текущему потоку.

Оператор continue выполняет переход к следующей итерации цикла (листинг 17.15). При этом нет необходимости использовать именные блоки begin...end, как в случае применения оператора disable. Оператор break осуществляет немедленное завершение работы цикла (листинг 17.16).

Листинг 17.15. Использование continue в цикле

```
logic [15:0] array [0:255];
always_comb begin
        for (int i = 0; i <= 255; i++) begin : loop
          if (array[i] == 0)
                continue;            // Пропуск пустых элементов
                transform_function(array[i]);
        end                          // Конец цикла
end
```

Декларация порта по умолчанию (.*)