The Life of a SystemVerilog Variable

Dave Rich

Mentor, a Siemens Business

46871 Bayside Pkwy, Fremont, CA 94538

***Abstract*- The life of a SystemVerilog variable may seem like a mundane topic, but there are nuances that get overlooked leading to issues that are difficult to debug. Some of the most common issues are how and when variables get initialized, how concurrent threads interact with the same variable, and how certain variable lifetimes interact with other SystemVerilog features in terms of performance considerations. This paper presents a background on the different categories of variable lifetimes, what their intended use models are, and how improper usage can be corrected.**

I. Introduction

A. What exactly is a Variable?

Believe it or not, this is the number one most difficult concept for people when learning Verilog and now SystemVerilog[1]. When Verilog[2] was created as a Hardware Description Language (HDL), it introduced the keyword concepts of **`wire`**s and **`reg`**s. A **`wire`** is a named network of connections between drivers and receivers with no storage implied. A **`reg`** does imply some kind of storage, hence **`reg`** is short for 'register'. The key difference between them is how they are assigned values. The driver of a wire is a process continuously applying a value with rules for resolution when there are multiple drivers. A reg gets its value from a procedural statement and holds that value until the next procedural assignment. The last assignment determines the current value of the register.

As synthesis tools evolved, a **`reg`** became to be used more than just for hardware storage, and more like the software concept of a variable—simply a place to hold a value for later use. Verilog-2001 began to remove the hardware register terminology and replaced it with a *variable*, and SystemVerilog replaced the **`reg`** keyword with **`logic`**. But there is still no concept of distinctive lifetimes for an HDL and the objects they describe. Hardware exists for the life of the device and cannot be created or destroyed while operational. For example, field programmable gate arrays (FPGAs) for the most part are not programable while in operation. They must be set up or "burned in" before becoming operational.

B. What is a lifetime?

In software programming, *lifetime* is defined by when and from where a variable is available for access. This is particularly important when there are multiple process threads with lifetimes of their own trying to access the same variable. A variable becomes a symbolic name for a particular range of memory locations allocated to a specific data type. Some languages like C/C++ let you see and manipulate the actual memory address associated with a variable in the form of pointers, but SystemVerilog does not. It has the concept of *safe pointers* which prevents you from accessing memory that does not belong to you or has been reclaimed for some other use. This memory safety feature is like what exists in other programming languages like Java[3] or Go[4].

The most basic form of a variable's lifetime is the kind from hardware design where everything always exists. This is what's called a *static* lifetime. Static variables exist from the beginning of time when they get initialized, to the end of time. I'll discuss what *the beginning of time* means in the next section.

All other forms of variable lifetimes involve the execution of procedural code which involves a deeper understanding of scopes, like those introduced by tasks and functions and other named blocks. As SystemVerilog is intended for both design and verification, it must blend hardware and software concepts together within the same language. This becomes especially important as verification testbenches typically cross probe into variables within the design. Understanding when variables get initialized and when they are accessible from other scopes becomes critical.

II. Variable Declarations and Name Spaces

A. *Static Variables and Scopes*

A name space or scope is a region of code where all user defined identifiers must be unique. There are a few different name spaces in SystemVerilog, but we only need to look at regions where variables can be declared. The most basic scope is the *module* name space which is really all the design elements including **`module`**, **`interface`**, **`package`**, **`program`**, and **`checker`** constructs. This name space is the scope inside each design element where we can declare variables, data types, procedures, and other structural concepts. For the purposes of discussing variable lifetimes, we can stick to the **`module`** construct as there is no conceptual difference with the other design elements.

Another basic scope is the *block* name space which nests inside other design elements. Blocks are typically introduced by procedural code like tasks, functions, and user defined types like structures and classes. There is also a scope outside the declaration of design elements called the *compilation-unit scope* name space where a variable may be declared. This is only useful for small descriptions because as you increase the size of your code, name collisions are more likely to occur.

When we declare variables at the top level of any module name space, we are declaring variables with static lifetimes. Static implies allocation as part of the elaboration step before simulation begins until the end of simulation time. Elaboration is the process of bringing together all the design elements that make up a simulation environment, replicating and making connections as needed. After design elaboration, simulation is ready to start at time 0.

```
package P;
 int A=1;
endpackage
module X;
  import P::A;
  int B=A++;
endmodule
module Y;
  import P::A;
  bit [10:0] B=A++;
  X m1();
  X m2();
endmodule
```

Figure 1

In Figure 1, modules **`X`** and **`Y`** import the variable **`A`** from package **`P`**. Module **`Y`** instantiates the module **`X`** twice; creating two declarations of a static variable **`B`** with the hierarchical path names **`Y.m1.B`** and **`Y.m2.B`**. Module **`Y`** contains another variable with the same name **`B`**, which is allowed because it is inside a different name space. Note that although the variable **`A`** in package **`P`** gets imported multiple times, there is still only one allocation of the variable that remains inside the package. Imports do not create multiple instances of a package—they only admit references to the name. So, in this example there is one variable named **`A`** and three variables named **`B`**.

These variables all have initializations associated with their declarations to show a problem called the "static initialization order fiasco[5]". This occurs because all static variables get initialized at time 0, but there is no way to predict their order of execution (there are ${}_4P_4=24$ possibilities). One way avoiding this problem is not initializing static variables with dependencies on other static variables and instead assigning them within procedural code. Another option that is useful when dealing with static class variable initialization is called "construct on first use". Instead of accessing a static variable directly, we call a static function which checks to see if the variable is initialized and that function initializes it if not already. That way it always returns an initialized variable value.

B. *Automatic Variables and Procedural Blocks*

Automatic lifetimes are always associated with the execution of procedural blocks of code. A procedural scope gets introduced by the **`begin/end`** or **`fork/join`** constructs, or by a **`task`** or **`function`**. Allocation and initialization commence upon procedural entry to the scope and the variable's life ends upon exit of the scope. Because initial versions of Verilog only had the concept of static lifetimes, the **`automatic`** keyword is needed to explicitly declare variables with that lifetime. In Figure 2, there are 5 variables named **`A`** with different lifetimes declared implicitly or explicitly. Note that static variables can always be referenced from outside their scope using a named hierarchical path, but automatic variables can only be referenced from within their procedural scope.

```
module top;
  int A; // static-outside a procedural
block
  initial begin : a_block // named block
    bit A; // implicitly static
    automatic int B;
    $display(A,,B);
  end
  initial begin // unnamed block
    static bit A; // explicitly static
    automatic int B;
    $display(A,,B);
  end
  initial $display(  // references
          A,,        // the int A
          a_block.A, // the bit A
          f1.A, f2.A );
  function void f1;
     int A; // implicitly static
     automatic int B;
  endfunction
  function automatic void f2;
     static int A; // explicitly static
     int B;        // implicitly automatic
  endfunction
endmodule
```

Figure 2

Variables declared with automatic lifetimes can also include initializations, but a special rule exists to avoid problems with SystemVerilog's default static lifetime, which is opposite of most other programming languages. This shows up

```
module top;
  function int countones(bit [7:0] in);
     int count = 0; // illegal - implicitly static
     for(int i=0;i<8;i++) count += in[i];
     return count;
  endfunction
  initial begin
       $display(countones(8'b10101010)); // prints 4
       $display(countones(8'b01010101)); // prints 8
  end
endmodule
// solution 1 - break up the initialization
function int countones(bit [7:0] in);
     int count; // implicitly static with no initialization
     count = 0;
     for(int i=0;i<8;i++) count += in[i];
     return count;
endfunction
// solution 2 - // add explicit automatic to the
                // variable declaration
function int countones(bit [7:0] in);
     automatic int count=0;
     count = 0;
     for(int i=0;i<8;i++) count += in[i];
     return count;
endfunction
// solution 3 - // add explicit automatic to the
                // function declaration to change the default
function automatic int countones(bit [7:0] in);
     int count=0; // implicitly automatic
     count = 0;
     for(int i=0;i<8;i++) count += in[i];
     return count;
endfunction
// solution 4 - // add explicit static to the variable
                // declaration if that is the intent
function automatic int countones(bit [7:0] in);
     static int count=0; // explicit static
     count = 0;
     for(int i=0;i<8;i++) count += in[i];
     return count;
endfunction
```

Figure 3

most often with loop initializations where you expect the initialization to occur every time you enter the block code but instead it only happens once at times zero as shown in Figure 3. You are not allowed to add an initialization to a

variable inside a procedural block whose lifetime is implicitly static. There are several solutions to this problem that can achieve your intent, shown in the same figure.

```
module top;

  initial for(int vi=0;vi<2;vi++) begin
     automatic int vj=vi;
     fork
       $display(vi,,vj);
     join_none
   end
endmodule
// Displays
// 2 1
// 2 0
// or
// 2 1
// 2 0
```

Figure 4

Two places where the default variable lifetime is automatic are in the declaration of a procedural **for** loop, and variables declared inside class methods. Changing the default lifetime for these constructs is possible since there were no backward compatibility issues with Verilog.

A nuance to the end of life of automatic variables comes into play when dealing with concurrent processes created using the **fork** construct. A scope's life gets extended to include the life of any nested concurrent processes. In Figure 4, The loop variable **vi** is implicitly declared with an automatic lifetime. The unnamed block declares the variable **vj** with an automatic lifetime and it gets initialized with the current value of **vi**. Since there are two iterations of the for loop, two allocations and initializations of vj occur with the values 0 and 1 respectively. Since processes spawned by a **fork/join_none** block do not start execution until the parent process blocks or terminates, the two processes containing the **$display** statement do not start executing until after the loop variable **vi** reaches 2 and the **initial** block terminates. However, the automatic variables **vi** and the two copies of **vj** remain until the two display statements finish.

III. Data Types with Dynamic Elements

By now we have covered the basic operation of static and automatic variables. Certain data types introduce another dimension to lifetime; these are the dynamically sized arrays and classes. Class objects will be covered in the next section. These dynamic data types have in common the ability to allocate or deallocate storage based on a procedural assignment or method call. SystemVerilog provides three different kinds of dynamically sized unpacked arrays. Unpacked arrays are aggregates of identically typed elements. As with any variable, all elements associated with dynamically sized pipes are removed when the life of the variable ends.

A. Dynamic Arrays

Dynamic arrays are simply a continuous range of array elements where the range can be resized by an assignment. Resizing a dynamic array might involve copying all existing elements to a new space regardless of whether the new array size is increased or reduced. Typically, you size a dynamic array once at the beginning of its lifetime instead of having to use a fixed size array whose size must be determined at elaboration. Its elements are indexed starting with integer 0. This is the most efficient way of accessing a block of memory, especially when you need to access the entire array. Note the distinction in the use of square brackets in Figure 5 with **new[]** to allocate a dynamic array versus the parentheses of a class constructor **new()**.

```
module top;
  int da[]; // static variable
  initial begin
     // dynamically allocate 100 elements
     da = new[100];
     // 100 elements replaced with 3
     da = {10,20,30};
  end
endmodule
```

Figure 5

B. Queues

```
initial begin
  automatic int qa[$];
  // allocate 1 element
  qa = {10};
  // append 1 element
  qa.push_back(20);
  // concatenate 2 elements
  qa = {qa,30,40};
  // Array's life ends after #10
  #10 $display("tail = ",qa[$]);
end
```

Figure 6

When using a queue, you typically access only the head or tail element, and are repeatedly adding and deleting one element at a time from the head or tail using the push and pop methods. A queue is implemented as a linked list, which makes head and tail access very simple. But this comes at the expense of taking up more memory for each element and requires a traversal of elements to get to an element in between the head and tail.

As with any unpacked array, you can copy the entire queue from another array, or from an array literal as shown in Figure 6. The

queue gets resized to accommodate the new array size. Then the **initial** block ends at time 10, the life of **qa** ends along with all its elements.

C. *Associative Arrays*

Each element of an associative array gets allocated as you access them. There is a lot more overhead for the creation of an associative array versus the same size dynamic array. And since elements of an associative array are not always in a contiguous block of memory, there is overhead in accessing each element (i.e., must check if the element is allocated, and then where is it located).

```
// array with 8-bit address
bit [15:0] aa[bit [7:0]] = '{default:'1};
initial begin
  $displayh(aa[2]); // FFFF -no allocation
  // allocate 1 element
  aa[2] = 0;
  // allocate 1 element
  aa[4] = 1;
  // 2 elements replaced with 3 elements
  aa ='{0:10, 1:20, 2: 30};
end
```

Figure 7

A benefit of associative arrays is since each element gets allocated individually, you don't need to allocate a contiguous set of array elements. This is useful when you only plan to access a relatively few (sparse) elements of a memory address space. However, it you plan to write to most of the elements, a dynamic array would be more efficient. Another benefit is that you don't have to use an integer as an index to each element. You can associate any type of key, like a string or class handle to access each element.

In the example in Figure 7, **aa** is an associative array with a maximum of 256 elements. An assignment pattern with a default value is used with a declaration initialization to indicate the value returned for any uninitialized element. The first reference to **aa[4]** yields FFFF without allocating element [4] until the following assignment statement. Associative arrays cannot be copied from other arrays unless that are associative with the same index type.

D. *Arrays of Arrays*

One thing that may be unfamiliar to some users is that SystemVerilog does not really have multi-dimensional arrays—it has arrays of arrays. You can declare a declare a 1-dimentional array of any type, and that type can be a different kind of array of another type. In Figure 8, **DQ** is static variable whose data type is a dynamic array of queues of bytes. The **new[5]** operator allocates 5 empty queues, and the nested foreach/repeat loops allocate 1 queue element to **DQ[0]**, 2 queue elements to **DQ[1]**, etc. When the loops finish, each element of the first dimension will have a different number of elements for its second dimension. The total number of bytes in the array is 1+2+3+4+5=15. The last statement copies the entire 4-element queue in **DQ[4]** to **DQ[0]**, erasing the 1-element queue that was in **DQ[0]**. The total number of bytes goes up to 18.

```
module Y;
 byte DQ [][$];
 initial begin
  DQ = new [5]; // 5 empty queues
  foreach(DQ[i])
   repeat(i+1) DQ[i].push_back(i);
  DQ[0] = DQ[4];
  end
endmodule
```

Figure 8

E. *Class objects with dynamic memory management*

Class objects are a special case of dynamic lifetimes with some important distinctions. Comparable to dynamically sized arrays, class object creation is accomplished through procedural calls to the **new()** constructor method, however access to that object is always through an indirect reference called a handle. That handle can be stored in multiple variables that all reference the same object. The consequence is that there is no specific procedural statement that destroys a class object. SystemVerilog automatically keeps track of all variables referencing an object and deletes that object after no references to it remain. How soon after is not observable to the user since there are no remaining references to it. A class variable can have any kind of lifetime: static, automatic, dynamic as an element of an array, or dynamic as a member of another class object. The variable's lifetime indirectly influences the lifetime of the object by determining when the reference to the object gets removed.

```
module automatic top;
// default lifetime for all blocks
  class C;
    bit [7:0] packet [];
    function new(int psize);
      packet = new[psize];
      void'(randomize(packet));
    endfunction
    function void copy(C rhs);
      this.packet = rhs.packet;
    endfunction
  endclass

  C h = new(2);
  initial begin
    repeat (5) run1(h);
    repeat (5) run2(h);
  end

function void run1(C in);
    C temp = new (3);
    temp.copy(in);
  endfunction
function void run2(C in);
    C temp = new (1);
    static C list[$];
    temp.copy(in);
    list.push_back(temp);
endfunction

endmodule
```

Figure 9

The module inside Figure 9 defines a class **C** containing one object packet which is a dynamic array. The constructor of the class takes one argument that allocates packet array to the desired size and randomizes the packet's values. The module defines a static class variable **h** that get initialized with a handle to the object returned by constructor. The constructor allocates the **packet** array with 2 elements. The **initial** block calls the **run1** function 5 times. Each call to **run1** allocates the automatic variable **temp** which gets initialized with a handle to an object with a 3-element packet. It then copies the **packet** whose handle was passed into the task into the newly created object. So, the 3-element **packet** gets replaced with a 2-element packet. When exiting **run1**, the life of the **temp** variable ends; the only reference to the object just created disappears.

Next, the **run2** function gets called 5 times. The only difference between **run1** and **run2** is that before **run2** exits, the handle that is stored in temp gets pushed into the static queue list. Even though the life of the variable **temp** ends, the reference to the newly created object remains as an element of the **list** queue. At the end of this simulation, there are 6 class objects references: 1 in the static variable **h**, and 5 different references in each element of the static queue **list**.

By keeping track of all references to objects until they are no longer needed, SystemVerilog ensures that every reference is safe reference to a valid object. The consequence of this is the user is still tasked with removing references when they are no longer needed, otherwise they can be penalized with memory leaks.

IV. Subroutine Lifetimes

The lifetime of a function or task is a concept peculiar to Verilog. The initial versions of Verilog started out having only static lifetimes of functions or tasks, meaning that there was no call stack for arguments or variables local to subroutines. This meant you could not have recursive or re-entrant routines, unlike most other modern programming languages. The original reasoning was synthesis could not create stacks of memory dynamically when called. Verilog-2001 added the 'automatic' lifetime qualifier to give routines the normal behavior of most programming languages. SystemVerilog added a lifetime qualifier for modules and interfaces so that all routines defined in that module would be considered automatic by default, so you didn't have to add the **automatic** keyword after each function or task declaration. Note that the lifetime of class methods is always automatic, you cannot declare them with a static lifetime. This is not to be confused with a **static** class qualifier, where the **static** keyword appears to the left of the function or task. This means it is a method of the class type, not of a class instance or object.

```
module top; // scopes are default static
  function int factorial(int N);
    if (N>=2)
      return factorial(N-1) * N;
    else
      return 1;
  endfunction
  initial $display(factorial(3));
endmodule
```

Figure 10

Figure 10 shows a function **factorial** with a static lifetime and a static argument **N**. When first called with N=3, it recursively calls **factorial(2)** and then **factorial(1)**. The one static variable **N** is set to 1, and the return value incorrectly becomes 1 * 1 * 1 = 1. The solution for this is declaring **module automatic** or **function automatic**. Then each call gets new instance of the automatic variable N, and the result becomes 1 * 2 * 3 = 6.

V. Summary

This paper has shown various interactions between named scopes and variable lifetimes. It provided guidance on how different data types allocate memory and consequences for choosing one type over another. It also showed several examples of coding errors because of misunderstanding variable lifetimes and gave tips to avoid them.

VI. References

[1] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) , 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.

[2] "IEEE Standard Verilog Hardware Description Language," in IEEE Std 1364-2001, 28 Sept. 2001, doi: 10.1109/IEEESTD.2001.93352.

[3] "Why is Java Knows as a Safe Language", Armin Zirak, 16 Nov. 2018, https://medium.com/@armin.zirak97/why-is-java-known-as-a-safe-language-be7a9cc42707

[4] "The Go Programming Language Specification", Google, Retrieved 11 Dec 2020, https://golang.org/ref/spec

[5] "What is the static initialization order fiasco?", Marshal Cline, 4 Jul 2012, http://www.cs.technion.ac.il/users/yechiel/c++-faq/static-init-order.html