DVCon 2012

Design & Verification Conference & Exhibition

February 28 – March 1, 2012

Yikes! Why is my SystemVerilog Testbench So Slooooow?

Presented by

Justin Sprague

Senior Applications Engineer

Cadence Design Systems

cadence®

SV Performance and Productivity
Follow SW engineering guidelines

- IEEE 1800 SystemVerilog (SV) enables productivity
  - Classes, dynamic data types, randomization, etc. simplify creation of sophisticated verification environments
  - Accellera UVM simplifies creation of test and reuse of VIP speeding verification
- ... but can quickly yield large, accurate, and slooooow envs
  - SW engineering != HW engineering because the former involves dynamic code and data
  - SV != Verilog meaning new coding *concepts* needed for performance

Meh, It's Just Software

Attitude leads to accurate, slow SV TB

Hardware (Design)

- Static code structure
- Physical data – registers, memory space
- Physically limited loops
- Deterministic behavior
- Optimize for device performance

Software (Testbench)

- Dynamic code structure
- Abstract data – multi-dimensional arrays, dynamic types with no simple physical equivalent
- Unlimited loops
- Environment built around randomization
- Optimize for simulation performance

Loop Invariants

- Software looping can range into millions of cycles
- Loop limits are often dynamically set
- Loop invariants are executed on each cycle yielding identical results
- Solution: Move invariants outside of the loop

```
int i, a[256], b[256];
int length=4, count=6, l_end;

for (i=0; i < length*count; i++)
        a[i] = b[i];

l_end = length * count;
for (i=0; i < l_end; i++)
        a[i] = b[i]
```

Note: Red box is low performance coding style.

Short-Circuiting Branches

- Simulators optimize code to branch when minimum conditions are met
- Optimization follows order of operations so if the left-most operator is the minimum condition then code runs fast
- Solution: Where possible order the branch terms L2R

```
if (nearly_everytime() || sometimes_happens() || rarely_happens())
       code_to_execute
```

```
size = millions_of_pkts.size();
for (i = 0; i < size; i++) begin
    data = millions_of_pckts[i].randomize();
    live = millions_of_pktsp[i].live
    if (live == TRUE)
        inject(data);
end
```

Static Versus Dynamic Classes

- Static allocation is the object oriented extension to the general programming concept of macros
- When classes are created and destroyed repeatedly in large loops, memory can fragment and OS can become overhead
- Solution: Statically allocate classes in these situations or define a pool of classes deep enough to support the maximum working set

Know the Class Hierarchy

- Undocumented classes can be a performance nightmare
  - Methods and data in base classes exist in all derived classes
- Reimplementation in derived classes carries dead-code or unused/misused data
  - Memory grows faster than expected, performance slows
- Solution: Demand documentation! And supply it ☺. Also, only access information via standard interfaces

Track Class and Data Handles

- Garbage collection in SV works if properly managed
  - Number of references to the class handle falls to zero
  - Simulation engine tracing algorithm detects object graphs that are self-referencing but lack direct user references
- Poor handle control is the leading cause of memory leaks
  - Dynamic and associative arrays are most susceptible
  - Bugs can be insidiously hard to trace
- Solution: Add checks to code to detect overflows and be especially wary of global data as multiple threads operating on a single data structure can create unexpected side-effects

Thread Pool Vs. Create/Destroy

- Execution threads, like data and classes, can experience handle issues and memory fragmentation
- Solution: Follow similar techniques when implementing very high thread-count environments

```
forever begin
  @(posedge clk);
  fork
    for(int i = 0; i < 32; i++) begin
      automatic idx = i;
      wait(bus[idx] == 1);
    end
  join
end
```

Unforeseen Library Overhead

- Libraries like UVM enable fast creation of verification envs
- Be aware of library implementation when scaling-up env
  - Ex. Testbench channel may be clocked so sending many tiny pieces of data may be much more expensive than aggregating the data and sending a single, large structure
- Solution: Always implement functional code first. If performance issues arise, profile and consider alternate algorithms that better utilize standard library interfaces

Summary -

- Simulators are built to run legal SystemVerilog
- 2 algorithms can have equivalent functionality and vastly different performance
- Coding errors and code awareness can lead to unforeseen and hard to debug performance issues
- Start from a set of best practices
  - Think about performance from the beginning
  - Understand and apply SW engineering principles
  - Use profiling as an algorithm development tool

THANK YOU.

QUESTIONS?