2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Efficient Constrained Random Generation of Address Blocks

A comparative study of simulation speeds

Meenakshy Ramachandran, R&D Engineer, Synopsys India Pvt. Ltd., Hyderabad, India
(meenakshy.ramachandran@synopsys.com)

***Abstract*—With increasing computational requirements, high-level memory controllers are being modeled for faster memory access. A memory controller controls the flow of data between the host memory and a dynamic memory coupled to it. The host memory is divided into pages which are blocks of contiguous addresses. The main challenge in the verification of a memory controller lies in writing efficient constraints that cover a wide range of address stimuli in a faster and accurate manner. The paper illustrates various methods for the constrained random generation of non-overlapping memory blocks. The key learning in the form of dos and don'ts from each method can be extended to any other application.**

Keywords— System Verilog; Constrained Randomization; Memory Controller; non-overlapping memory blocks

I. INTRODUCTION

In order to improve the system performance by bringing about incredible speed and lower latencies, high speed memory controllers with a wide range of advanced capabilities are being introduced in the market. The memory controller handles the data flow between the host memory and the dynamic memory, usually a DDR memory.

The system consists of a host comprising a host memory, a memory controller and a non-volatile dynamic memory. The controller stores the data which is read by the host or written by the host in the non-volatile memory.

The host posts the commands into the host memory. The command is fetched by the memory controller. It decodes the command and initiates data movement between the host memory and the dynamic memory depending on whether data transfer is involved. Most memory controllers write a completion entry back into the host memory to notify the successful or erroneous completion of the command execution. A simple illustration of a memory controller is shown in Figure 1.

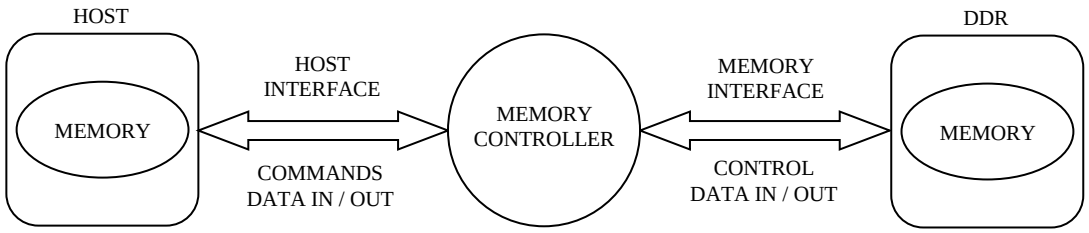

Figure 1. A simple representation of a memory controller.

Depending on the architecture of the memory controller, the physical memory which belongs to the host uses a variety of memory structures. These memory structures are usually used for storing commands or data to be written into the dynamic memory or data which is read from the dynamic memory by the host. The memory structures can be of different kinds. For e.g., circular queues are used in Non-Volatile Memory Express (NVMe) to pass information in the form of commands and command completion entries while Physical Region Pages (PRPs) are used to store the data to be written to the device or read from the device.

The host memory is divided into pages. A page is a block of physically contiguous addresses where data can be stored. The number of data bytes it holds is called the page size. In the conventional systems, the page size is usually set to be 4KB or 8KB. But based on the system architecture, very huge page sizes are also allowed.

The addresses can be either aligned or unaligned. An address is said to be memory page aligned when the address denotes the start of a page. In other words, the address should be a multiple of the page size. Consider the page size to be **n** bytes. A page aligned address would be n-byte aligned. When expressed in binary, an n-byte aligned address would have $\log_2(\mathbf{n})$ least-significant zeros. If the address is unaligned, the $\log_2(\mathbf{n})$ least significant bits of the address form the offset. The offset is the difference between the start address of the page and the unaligned address, in terms of the number of bytes. Thus the offset is the resultant remainder when the unaligned address is divided by the page size. Figure 2 illustrates an unaligned memory block with start address 0x04 and an offset of 0x03.

	ADDRESS	MEMORY PAGE
Offset	0x0000_0000_0000_0000	Byte 0
	0x0000_0000_0000_0001	Byte 1
	0x0000_0000_0000_0002	Byte 2
	0x0000_0000_0000_0003	Byte 3
Memory Block	0x0000_0000_0000_0004	Byte 4
	.	.
	.	.
	.	.
	0x0000_0000_0000_0FFA	Byte 4090
	0x0000_00000_0000_0FFB	Byte 4091
	0x0000_0000_0000_0FFC	Byte 4092
	0x0000_0000_0000_0FFD	Byte 4093
	0x0000_0000_0000_0FFE	Byte 4094
	0x0000_0000_0000_0FFF	Byte 4095

Figure 2. Illustration of a memory page

II. NEED FOR AN ADDRESS GENERATOR

A thorough verification of any design requires effective test vectors. The host system has its own physical memory in which the command and data are stored. The memory controller consists of an address translation unit. The address translation unit in the memory controller performs the translation of virtual memory addresses to the corresponding physical memory addresses in the host. Similarly, the verification testbench will have its own address translation unit to convert the physical addresses into the virtual addresses. In order to achieve a vast distribution, the random but constrained generation of the address stimuli is very important. The address generators should generate valid random addresses to effectively verify the functionality of the memory controller by covering many scenarios. At the same time, the randomness should be easily controlled by the user to generate specific corner case scenarios.

Even though it is required that the stimuli is accurate and random, it is equally important that the generation is quick and the simulation time is lesser for quick closure of the verification process. The constraint solver is known to solve very complex constraints, but the time taken to solve these constraints depends on a myriad number of factors. In addition to being simulator specific, the runtime also depends on the complexity of the constraint and the logic employed to constrain the values. Therefore, it is highly recommended to constrain the values of the object in such a way that the constraint solver consumes the least amount of CPU time.

The address generator should generate valid addresses in the specified memory range. The user should have a control over the page size of the memory, the memory space range, the alignment of the address and the offset value in the case of an unaligned address. It is very important that the address regions do not overlap. If an erroneous generation produces overlapping address ranges or out of range addresses, the data would get overwritten and incorrect stimuli would result in undesirable results.

III. DIFFERENT APPROACHES FOR ADDRESS GENERATION

Consider a class **address_generator**, given below, which encapsulates all the members that serve as the deciding factors for address generation. **PAGE_SIZE** is parameterized as 4096 and it can be redefined to any other value.

```
parameter PAGE_SIZE =  4096; //4KB
parameter OFFSET_WIDTH = $clog2(PAGE_SIZE);
class address_generator;
 //The 64 bit start address of the memory block
 rand bit[63:0] addr;
 //Determines whether addr is aligned or not. Aligned-> align = 1. Unaligned-> align = 0
 rand bit align = 1;
 //The offset
 rand bit[OFFSET_WIDTH - 1:0]offset;
 //The lower limit of the address range
 bit [63:0]start_address = 64'h0000_0000_0000_0000;
 //The upper limit of the address range
 bit [63:0]end_address   = 64'hFFFF_FFFF_FFFF_FFFF;
 //The memory block size
 int block_size;
 //The memory page size
 int page_size = PAGE_SIZE;
 //The offset width
 int offset_width = OFFSET_WIDTH;
 //Constraint for the address offset
 //When the address is aligned, offset should be zero
 //When the address is unaligned, offset should be greater than zero
 constraint offset_c { (align == 1) -> (offset == 0);
                       (align == 0) -> (offset > 0);
                     }
endclass
```

The constrained randomization of an object of the **address_generator** class will generate the non-overlapping memory blocks of varying sizes. For the sake of uniformity across the methods, all the simulations are run using the Synopsys VCS simulator, but the results are similar for other simulators too.

A. APPROACH 1

As shown in **address_generator_program_1** in this approach, the address blocks are constrained to lie in the memory range defined by **{[start_address:end_address]}**. If the bit **align** is set to 1, the memory block is aligned because the offset is set to zero by the constraint **(align == 1) -> (offset == 0)**. The results would not be accurate if the reverse constraint **(align == 0) -> (offset > 0)** is missed out because the offset should be non-zero for an unaligned address. The offset is constrained by the inline constraint **offset == addr[OFFSET_WIDTH-1:0]**.

This constraint can also be replaced by **offset == addr%page_size** which would serve the same purpose. In order to avoid overlap of memory blocks, all the addresses in the currently selected page are added to **used_address_list** inside a for loop. Because of the huge amount of run time, this method is highly inefficient. It is suitable only if the amount of data transfer involved is less.

```
program address_generator_program_1;
 initial begin
  //Handle of the address_generator class
  address_generator address_generator_h;
  //A list to store the used addresses
  //to avoid generation of overlapping blocks
  bit[63:0]used_address_list[$];
  //The aligned counterpart of the generated address
  bit[63:0]aligned_addr;
  //Generate 1000 address blocks of varying sizes
  address_generator_h = new();
  for(int i = 0; i < 1000; i = i + 1)begin
   void'(address_generator_h.randomize()with{ addr inside {[start_address:end_address]};
                                    !(addr inside {used_address_list});
                                     offset == addr[OFFSET_WIDTH-1:0];
                                   });
   aligned_addr = address_generator_h.addr - address_generator_h.offset;
   for(int i = 0; i < address_generator_h.page_size; i = i + 1)begin
    used_address_list.push_back(aligned_addr + i);
   end
   address_generator_h.block_size = address_generator_h.page_size - address_generator_h.offset;
   //Display the generated addresses
   $display("Value of align is %h, value of address is %h, value of offset is %h, value of memory block size is %h",
    address_generator_h.align, address_generator_h.addr, address_generator_h.offset, address_generator_h.block_size);
  end
 end
endprogram
```

RESULT SUMMARY

```
$finish at simulation time                0
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:  14473.080 seconds;       Data structure size:   0.0Mb
Wed Jul 27 05:22:12 2015
CPU time: .084 seconds to compile + .022 seconds to elab + .110 seconds to link + 14504.684 seconds in simulation
```

B. APPROACH II

As shown in **address_generator_program_2** in this approach, the address blocks are initially constrained to lie in a compressed range i.e. **{[(start_address/page_size): (end_address/page_size)]}**. Post randomization, the generated address is multiplied with the page size in order to get the actual address. Therefore, after the randomization, the offsets are also added to the aligned addresses. It is already taken care in the **address_generator** class that the offset is 0 for an aligned address and greater than 0 for an unaligned address.

Because the range is compressed, a single address is added to the **used_address_list** instead of adding all the addresses in the current page. In the subsequent cycles of randomization, the constraint solver looks for a duplicate address by just comparing it with the addresses already used. Since the list is not as exhaustive as earlier, the constraint solver takes lesser time.

```
program address_generator_program_2;
 initial begin
  //Handle of the address_generator class
  address_generator address_generator_h;
  //A list to store the used addresses
  //to avoid generation of overlapping blocks
  bit[63:0]used_address_list[$];
  //Generate 1000 address blocks of varying sizes
  address_generator_h = new();
  for(int i = 0; i < 1000; i = i + 1)begin
   void'(address_generator_h.randomize()with{ addr inside {[start_address/page_size:(end_address/page_size) -1 ]};
                                    !(addr inside {used_address_list});
                                   });
   used_address_list.push_back(address_generator_h.addr);
   address_generator_h.addr = address_generator_h.addr*address_generator_h.page_size + address_generator_h.offset;
   address_generator_h.block_size = address_generator_h.page_size - address_generator_h.offset;
   //Display the generated addresses
   $display("Value of align is %h, value of address is %h, value of offset is %h, value of memory block size is %h",
    address_generator_h.align, address_generator_h.addr, address_generator_h.offset, address_generator_h.block_size);
  end
 end
endprogram
```

RESULT SUMMARY

```
$finish at simulation time                0
      V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:     2.300 seconds;       Data structure size:   0.0Mb
Wed Jul 27 01:10:36 2015
CPU time: .090 seconds to compile + .019 seconds to elab + .118 seconds to link + 2.366 seconds in simulation
```

C. APPROACH III

As shown in **address_generator_program_3** in this approach, the address overlap check logic is brought outside the constraint which reduces the simulation time drastically. The address generation is performed inside a do-while loop. An associative array **used_address_list** that uses the address prior to multiplication by page size as index, is used for address overlap check. For every unique address generated, the element of the array corresponding to the address is set. The **exists()** function of the associative array checks if the address generated in the current iteration has been used previously. If yes, the do-while loop executes till a unique address is generated.

```
program address_generator_program_3;
 initial begin
  //Handle of the address_generator class
```

```
  address_generator address_generator_h;
  //A list to store the used addresses
  //to avoid generation of overlapping blocks
  bit[63:0]used_address_list[*];
  //Generate 1000 address blocks of varying sizes
  address_generator_h = new();
  for(int i = 0; i < 1000; i = i + 1)begin
   do begin
    void'(address_generator_h.randomize()with{ addr inside {[start_address/page_size:end_address/page_size - 1]};
                                    });
   end while(used_address_list.exists(address_generator_h.addr));
   used_address_list[address_generator_h.addr]=1;
   address_generator_h.addr = address_generator_h.addr*address_generator_h.page_size + address_generator_h.offset;
   address_generator_h.block_size = address_generator_h.page_size - address_generator_h.offset;
   //Display the generated addresses
   $display("Value of align is %h, value of address is %h, value of offset is %h, value of memory block size is %h",
    address_generator_h.align, address_generator_h.addr, address_generator_h.offset, address_generator_h.block_size);
  end
 end
endprogram
```

RESULT SUMMARY

```
$finish at simulation time                0
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:      0.350 seconds;       Data structure size:   0.0Mb
Wed Jul 27 01:09:27 2015
CPU time: .085 seconds to compile + .019 seconds to elab + .141 seconds to link + .410 seconds in simulation
```

D. APPROACH IV

As shown in **address_generator_program_4** in this approach, the addresses are generated using the Verilog function **$urandom_range**. The input to **$urandom_range** can be 32-bit integers only. Thus it cannot be used with ease for generation of addresses with width greater than 32. Since we need to generate 64-bit addresses, the lower and upper 32 bit words are generated separately.

Though this is slightly faster than the previous approach, the address loses its random control. It can be employed in cases where the System Verilog random control is not required. Thus it is a trade-off between randomization control and simulation speed because of which the decision to adopt the better approach should be based on the weightage of the two in a given scenario or application.

```
program address_generator_program_4;
 initial begin
  //Handle of the address_generator class
  address_generator address_generator_h;
  //A list to store the used addresses
  //to avoid generation of overlapping blocks
  bit[63:0]used_address_list[int];
```

```
  //The start address divided by the page size
  bit[63:0]start_addr_div;
  //The end address divided by the page size
  bit[63:0]end_addr_div;
  //Generate 1000 address blocks of varying sizes
  address_generator_h = new();
  start_addr_div = address_generator_h.start_address/address_generator_h.page_size;
  end_addr_div = address_generator_h.end_address/address_generator_h.page_size;
  for(int i = 0; i < 1000; i = i + 1)begin
   do begin
    address_generator_h.addr.rand_mode(0);
    address_generator_h.addr[31:0] = $urandom_range(start_addr_div[31:0], end_addr_div[31:0] -1);
    address_generator_h.addr[63:32] = $urandom_range(start_addr_div[63:32], end_addr_div[63:32]);
    void'(address_generator_h.randomize());
   end while(used_address_list.exists(address_generator_h.addr));
   used_address_list[address_generator_h.addr]=1;
   address_generator_h.addr = address_generator_h.addr*address_generator_h.page_size + address_generator_h.offset;
   address_generator_h.block_size = address_generator_h.page_size - address_generator_h.offset;
   //Display the generated addresses
   $display("Value of align is %h, value of address is %h, value of offset is %h, value of memory block size is %h",
    address_generator_h.align, address_generator_h.addr, address_generator_h.offset, address_generator_h.block_size);
  end
 end
endprogram
```

RESULT SUMMARY

```
$finish at simulation time                0
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:      0.340 seconds;       Data structure size:   0.0Mb
Wed Jul 27 01:08:08 2015
CPU time: .124 seconds to compile + .025 seconds to elab + .179 seconds to link + .412 seconds in simulation
```

IV. APPLICATION OF THE ADDRESS GENERATOR FOR 128 MB DATA TRANSFER

Based on the study with the four different approaches for efficient constrained random generation of address blocks, the third approach is evidently the best suited. This paper culminates in the application of the same approach for generating addresses that can hold very large amounts of data to the scale of 128 MB and higher. The program **address_generator_program_128MB** given below highlights the same.

```
program address_generator_program_128MB;
 initial begin
  //Handle of the address_generator class
  address_generator address_generator_h;
  //A list to store the used addresses
  //to avoid generation of overlapping blocks
  bit[63:0]used_address_list[*];
```

```
//The total transfer size
int transfer_size;
//The temporary variable that holds the remaining transfer size
int transfer_size_remaining;
//Initialize transfer_size
transfer_size = 128*1048576;//128B in bytes
//Initialize transfer_size_remaining to transfer_size
transfer_size_remaining = transfer_size;
//Generate address blocks of varying sizes that holds a total of 128MB
address_generator_h = new();
while(transfer_size_remaining > 0)begin
 do begin
  void'(address_generator_h.randomize()with{ addr inside {[start_address/page_size:end_address/page_size - 1]};
                                  });
 end while(used_address_list.exists(address_generator_h.addr));
 used_address_list[address_generator_h.addr]=1;
 address_generator_h.addr = address_generator_h.addr*address_generator_h.page_size + address_generator_h.offset;
 address_generator_h.block_size = address_generator_h.page_size - address_generator_h.offset;
 //Display the generated addresses
 $display("Value of align is %h, value of address is %h, value of offset is %h, value of memory block size is %h",
  address_generator_h.align, address_generator_h.addr, address_generator_h.offset, address_generator_h.block_size);
  transfer_size_remaining = transfer_size_remaining - address_generator_h.block_size;
 end
end
endprogram
```

RESULT SUMMARY

```
$finish at simulation time                0
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:      2.640 seconds;       Data structure size:   0.0Mb
Mon Jul 27 02:55:59 2015
CPU time: .101 seconds to compile + .020 seconds to elab + .105 seconds to link + 2.846 seconds in simulation
```

Around **43,529 address blocks**, aligned and unaligned, were generated in a random manner in very less time, a snapshot of which is provide below.

```
Value of align is 1, value of address is 1079afad5c8db000, value of offset is 000, value of memory block size is 00001000
Value of align is 0, value of address is dfb0ebc30877fe1e, value of offset is e1e, value of memory block size is 000001e2
Value of align is 1, value of address is f4f265a975f80000, value of offset is 000, value of memory block size is 00001000
Value of align is 0, value of address is 79fbe25dcb261289, value of offset is 289, value of memory block size is 00000d77
Value of align is 1, value of address is dba9bd7fe015b000, value of offset is 000, value of memory block size is 00001000
Value of align is 1, value of address is 9c45f0f45a82f000, value of offset is 000, value of memory block size is 00001000
```

REFERENCES

[1] “SystemVerilog 3.1a Language Reference Manual Accellera’s Extensions to Verilog”, Accellera Organization, Napa, CA, 2004.

[2] Amber Huffman, “NVM Express 1.0e”, January 23, 2013.