# Efficient Constrained Random Generation of Address Blocks

A comparative study of simulation speeds

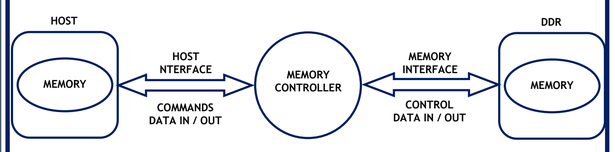

Meenakshy Ramachandran

Synopsys India Pvt. Ltd.


2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

## INTRODUCTION

### Memory Controller




- controls flow of data between host memory and dynamic memory
- host memory is divided into pages
- pages are blocks of contiguous addresses
- **Page aligned address:** Multiple of page size ($n$). Has $\log_2(n)$ least significant zeros
- **Unaligned address:** Offset from start address = $\log_2(n)$ significant bits

### Challenges in Verification

Generation of address stimuli must be:

- **Random** for vast distribution
- **Constrained** for accuracy
- **Controlled** for corner case scenarios

**Objective:** Faster simulation for quick closure of verification while achieving all of the above

| | 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 |
| | 0x0000_0000_0000_0004 | Byte 4 |
| Memory Block | . | . |
| | . | . |
| | . | . |
| | 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 |

## A SIMPLE ADDRESS GENERATOR

### Class Definition

```
parameter PAGE_SIZE =  4096; //4KB
parameter OFFSET_WIDTH = $clog2(PAGE_SIZE);

 class address_generator;
  rand bit[63:0] addr;                                    //start address of the memory block
  rand bit align = 1;                                     //Aligned: align = 1. Unaligned: align = 0
  rand bit[OFFSET_WIDTH - 1:0]offset; //The offset
  bit [63:0] start_address = 64'h0000_0000_0000_0000;     //The lower limit of the address range
  bit [63:0] end_address   = 64'hFFFF_FFFF_FFFF_FFFF;     //The upper limit of the address range
  int block_size;                                         //The memory block size
  int page_size = PAGE_SIZE;                              //The memory page size
  int offset_width = OFFSET_WIDTH;                        //The offset width

   /** Aligned: offset = 0. Unaligned: offset > 0 */
  constraint offset_c { (align == 1) -> (offset == 0);
                        (align == 0) -> (offset > 0);
                      }
endclass
```

Constrained randomization of an object of the *address_generator* class generates non-overlapping memory blocks of varying sizes.

**Simulations in all described approaches run on Synopsys VCS Simulator**

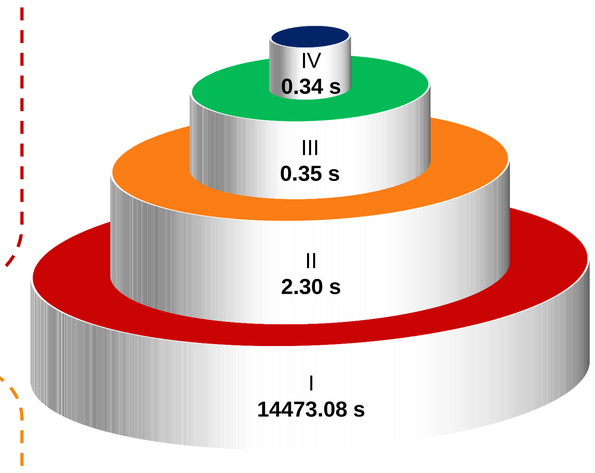
## PROPOSED APPROACHES FOR GENERATION OF ADDRESS BLOCKS

### Approach I - *Store all the addresses in a page*

```
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;
end
```

### Approach II - *Compress the address range*

```
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;
end
```




**CPU time in seconds**

### Approach III - *Move overlap check outside constraint*

```
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;
end
```

### Approach IV - *Use Verilog's $urandom_range function*

```
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;
end
```

## CONCLUSION – A COMPARATIVE ANALYSIS

### Approach I

- Memory blocks constrained within range {**start_address**, **end_address**}
- Alignment control exercised using the bit **align** set to 1 (**offset** = 0) or 0 (**offset** > 0)
- Constraint **offset == addr[OFFSET_WIDTH-1:0]** can be replaced by **offset == addr % page_size**
- Overlap of memory blocks avoided by maintaining a queue of used address blocks, **used_address_list**

**Verdict**
Using a for loop while populating the queue results in a huge amount of run time rendering the method highly inefficient. It is suitable only if the amount of data transfer involved is less.

### Approach II

- Memory block range compressed to {**start_address/page_size**, **end_address/page_size**}
- Actual address retrieved as [(Generated address * Page size) + Offset]

**Verdict**
Because the address range is now compressed, the queue **used_address_list** is populated with a single address instead of all addresses within a page. Constraint solver takes much lesser time since search list is very short.

### Approach III

- Address overlap checked outside constraint block
- Associative array **used_address_list** indexed by address within compressed range records used addresses

**Verdict**
The associative array's **exists()** function is used to check for duplicate addresses reducing the simulation time even further.

### Approach IV

- Addresses generated using Verilog function **$urandom_range**
- Lower and upper words generated separately because function generates only 32 bit integers

**Verdict**
Although slightly faster than the previous approach, randomization control is lost. It is a trade off between randomization control and simulation speed.

### Application of the most efficient address generator for 128 MB data transfer

- Comparative analysis highlights **Approach III** as the best suited for ***efficient constrained random generation of address blocks***
- Performance of same approach studied by applying it to generate address blocks capable of holding data up to 128 MB and more
- Around **43,529 address blocks**, aligned and unaligned, generated in random for the same
- A very nominal CPU time of **2.64 seconds** recorded for the entire simulation


*meenakshy.ramachandran@synopsys.com*


SYNOPSYS® Silicon to Software™