LVM BIT BASHING

About LVM Bit Bashing

WHAT IS BIT BASHING FROM LVM

- Enhanced version of UVM Bit Bashing with added a lot of features
- Coded to be robust, and loaded with easy debug-ability and user-friendly features
- Already built in for AXI / AHB / APB VIP!

OBJECTIVES OF LVM Bit Bashing

- To help user run bit bashing verification that can cater with special design on register.

Control Total Iteration

UVM

- Bashing every bit by writing 1 and 0

LVM

- Controllable number of iteration

```
// Total iteration for the bit bashing per register
`LVM__ITERATION     (m_apb_env.ral_bit_bash, 12)
```

Skipping Mechanism

UVM

- Can do reg / field skipping using resource DB and also set_compare API

LVM

- Other than these 2, can skip based on keyword / exact path

```
//
// Skipping API
//
    // Example of skipping reg/field per keyword / exact path
    //      `LVM__SKIP_FULLPATH(<seq instance name>,"<exact fullpath>"           ,0)
    `LVM__SKIP_FULLPATH(m_apb_env.ral_bit_bash,"urm.control_0_0C.B1",0)

    //      `LVM__SKIP_FULLPATH(<seq instance name>,"<regular expression pattern>",1)
    `LVM__SKIP_FULLPATH(m_apb_env.ral_bit_bash,"control_8.b",1)
```

Bit Bashing Reorder API

UVM

- Only can do in order

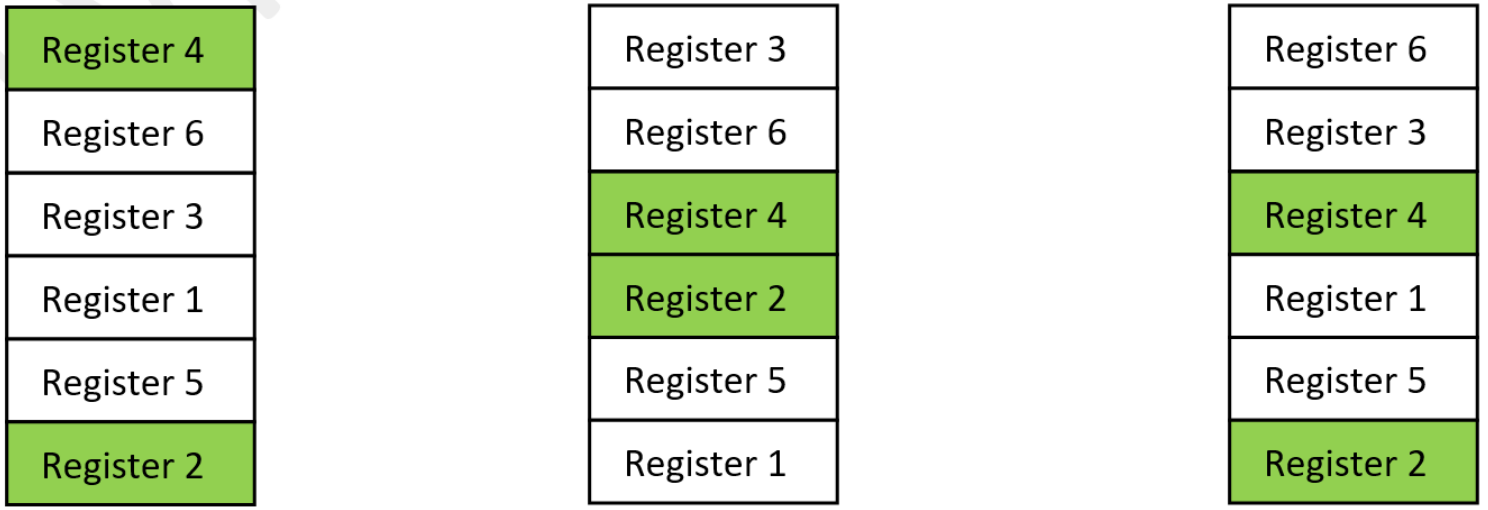
LVM

- Various API to control the sequence of execution of bit bashing

```
// Fashion 1: To shuffle the queue
// This allows the bit bashing done in random order
`LVM__SHUFFLE_MODE_ON(m_apb_env.ral_bit_bash)
```

Bit Bashing Reorder API

```
// Fashion 2: To constrain 2 registers order among the queue
// This controls the bit bashing done order for 2 registers
// This example says that control_4 will be done first, then only go to control_2
// Pls take note that this macro turn on the SHUFFLE mode above
`LVM__A_FIRST_B_LATER(m_apb_env.ral_bit_bash,   urm.control_4,     urm.control_2)
```



Bit Bashing Reorder API

User can define as many as the "constraint" involving 2 registers, for example

Register 4 > Register 2 &&

Register 6 > Register 1

The outcomes can be in any form with the sequences constraint satisfied.

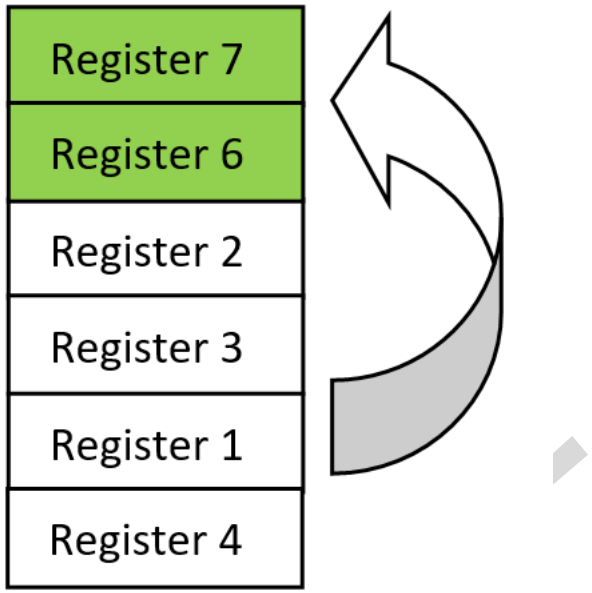
Bit Bashing Reorder API

```
// Fashion 3: To fix more than 2 registers sequence order
// Control more than 2 registers in sequencing of bit bash
// Lower number comes first
`LVM__N_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_3,1)
`LVM__N_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_1,2)
`LVM__N_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_4,3)
`LVM__N_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_8,4)
```

Suitable if there are larger order to be constrained, for example the sequence must be register 3 → 1 → 4→ 8. User can define unlimited times of this order relationship.

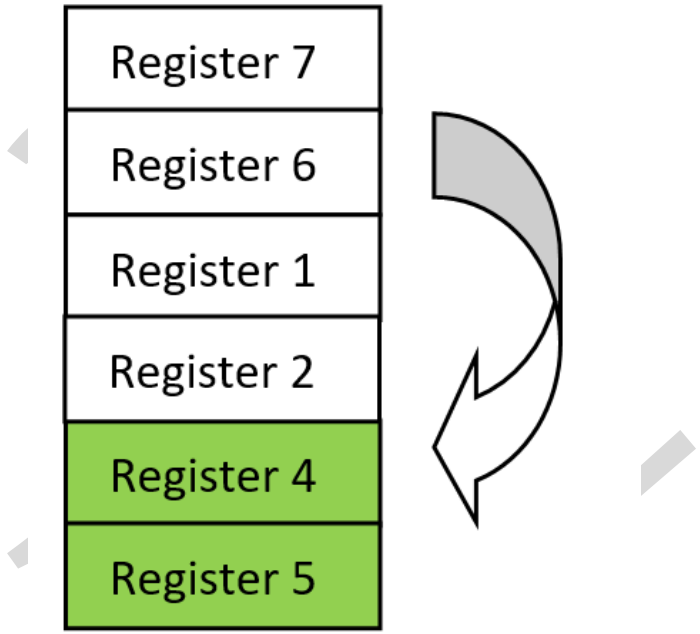
Bit Bashing Reorder API

```
// User to specify among the registers, which shall be done first before the rest.
// User can define to any number of preference. Can be used together with 1 out of the 3 fashions above.
// urm.control_7 will be executed first, directly followed by urm.control_6
`LVM__TOP_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_7)
`LVM__TOP_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_6)
```



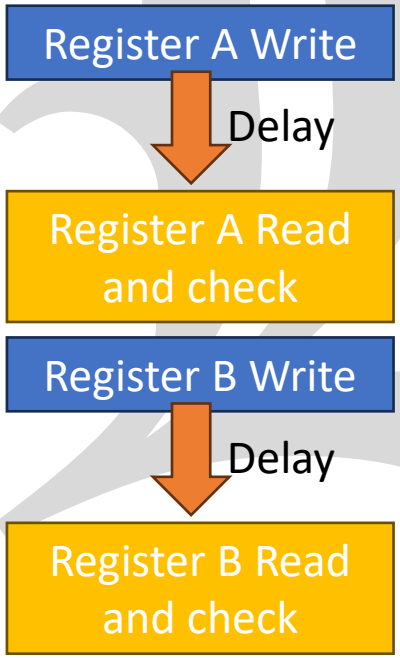
Bit Bashing Reorder API

```
// User to specify among the registers, which shall be done last after the rest.
// User can define to any number of preference. Can be used together with 1 out of the 3 fashions above.
// After the rest of the registers, urm.control_2 will be executed, directly followed by urm.control_1 before test ends
`LVM__BOTTOM_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_4)
`LVM__BOTTOM_SEQUENCING(m_apb_env.ral_bit_bash,urm.control_5)
```



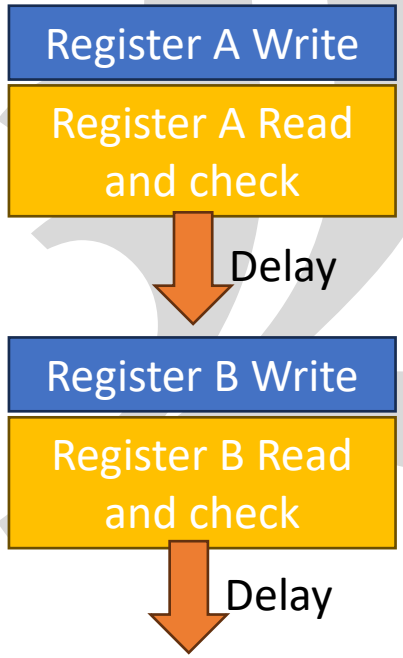
Bit Bashing Delay Injection API

```
//
// Write -> delay -> read check -> Write -> delay
//
// It will wait for the interval before executing the register read to check the value,
// as some design will take some times for the write to take effect.
// Note: This impacts all registers. To control the waiting time for 1 or more registers specifically, please refer to `LVM__REG_WAIT
// `LVM__WAITING_TIME(<seq instance name>,<integer>,"<unit>")
//  e.g
`LVM__WAITING_TIME(m_apb_env.ral_bit_bash,100,"ns")
```



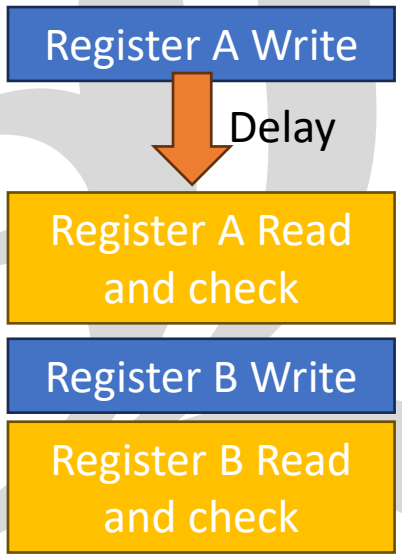
Bit Bashing Delay Injection API

```
//
// Write -> read check -> delay ->  Write -> read check -> delay - ...
//
// After the register operation (for example, a write followed by a read in bit bashing),
// it will wait for the interval before executing the operation for next register or next iteration for the same register.
// This is suitable for case where the bus needs to slow down.
//`LVM__TIME_INTERVAL(<seq instance name>,<integer>,"<unit>")
//  e.g
`LVM__TIME_INTERVAL(m_apb_env.ral_bit_bash,100,"ns")
```



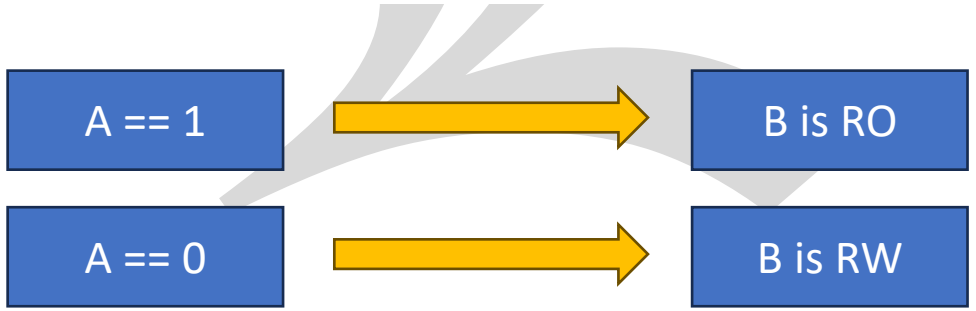
Bit Bashing Delay Injection API

```
//
// Write -> delay -> read check (for specific register only)
//
// Similar to waiting time, but this is per register: After the register write, it will wait for the interval before
// executing the register read to check the value, as some design will take some times for write value to take effect.
// This value will be used and overwrite the value from `LVM__WAITING_TIME if the register in `LVM__REG_WAIT is seen.
// `LVM__REG_WAIT(<register full path>,<integer>,"<unit>")
// for this register control_4, after writing, wait 1us, then only read
`LVM__REG_WAIT(urm.control_4, 1,"us")
```



Bit Bashing Special Fields with ACCESS control

```
//
// Special Relationship between 1 field/register to another
//
    // User to specify a relationship between field A and B, where:
    // When field A is X, B is RW/RO/etc
    // During the bit bashing testing, programming testing, the relationship will be considered
    // so that the sequence is able to predict the best on the outcome.
    //
    // `LVM__WHEN_A_IS_X_B_BECOME_Y(<seq instance name>,<fullpath to fieldA>,<X>,<fullpath to fieldB>,<access>)
    //
    // Example below shows
    // When urm.control_2.sam0_lock is 0, then urm.control_0.sample_0 is RW
    // When urm.control_2.sam0_lock is 1, then urm.control_0.sample_0 is RO
    `LVM__WHEN_A_IS_X_B_BECOME_Y(urm.control_2.sam0_lock, 1'b0, urm.control_0.sample_0, RW)
    `LVM__WHEN_A_IS_X_B_BECOME_Y(urm.control_2.sam0_lock, 1'b1, urm.control_0.sample_0, RO)
```



Bit Bashing Special Fields with Set, Clear & Toggle effect

```
// If you have such design, where 1 register has 4 control registers, where
//     1. Normal RW register
//          Write value will take effect as it is
//     2. Set       register
//          Writing 1 to any position will set    the corresponding bit at Normal RW register above
//          For example, writing 1 to bit 31 will set    bit[31]
//     3. Clear     register
//          Writing 1 to any position will clear  the corresponding bit at Normal RW register above
//          For example, writing 1 to bit 31 will clear  bit[31]
//     4. Toggle    register
//          Writing 1 to any position will toggle the corresponding bit at Normal RW register above
//          For example, writing 1 to bit 31 will toggle bit[31]
//
// Example below shows:
// urm.control_3_set is the "set"      register for urm.control_3
// urm.control_3_clr is the "clear"    register for urm.control_3
// urm.control_3_tgl is the "toggle"   register for urm.control_3
`LVM__MAIN_SET_CLR_TGL(urm.control_3,urm.control_3_set,urm.control_3_clr,urm.control_3_tgl)
```

Master Register (RW)
Write: Value will be taken effect

Set Register
Write: Value of 1 will set the corresponding bit at master register

Clear Register
Write: Value of 1 will clear the corresponding bit at master register

Toggle Register
Write: Value of 1 will toggle the corresponding bit at master register

Bit Bashing Value Control

```
// During the bit bashing testing, the randomized value written to the field will not hit the value specified.
// `LVM__A_AVOID_X(<seq instance name>,<fullpath to fieldA>,<X>)
// During bit bashing, this value 2'b11 will be avoided while randomizing value to register urm.control_2.protect
`LVM__A_AVOID_X(m_apb_env.ral_bit_bash,urm.control_2.protect,2'b11)
```

```
// During the bit bashing testing, the value written to the field will be constrained to the specified value,
// while the other fields within the same register will get randomized
// `LVM__A_MUST_BE_X(<seq instance name>,<fullpath to fieldA>,<X>)
// During bit bashing, this value will be used on the field for all the iterations in the bit bash
`LVM__A_MUST_BE_X(m_apb_env.ral_bit_bash,urm.control_2.protect,2'b01)
```

About LVM VIP

1. At LVM, our mission is to enhance the performance and efficiency of design verification work in the industry by providing ultra-high-quality, low-cost VIPs. We offer a range of AMBA VIPs and serial VIPs, including:

   AXI4 / AXI4-LITE, AHB, APB, TCM, USART, JTAG, AXI Stream

2. To learn more about our services, please contact us at

   developer@lvmvip.com.

3. You can also visit our LinkedIn page at
   https://www.linkedin.com/in/lvm-vip-3444b21b5/