An Introduction to the Next Generation Verification Language – Vlang

Puneet Goel <puneet@coverify.com>
Sumit Adhikari <sumit.adhikari@nxp.com>

In this section …

Verification Trends and Challenges

Vlang Language Features and Implementation

Widening the Horizon

Testbenches for System Level Verification

The Trend

- More and more designs are being verified using Emulation Platforms
- Shorter product cycles are driving design teams to use Virtual Platforms

The Challenge

- Contemporary Verification Languages were drafted with RTL simulation in mind
- SV perfomance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes

Testbenches for System Level Verification

The Trend

- More and more designs are being verified using Emulation Platforms
- **Shorter product cycles are driving design teams to use Virtual Platforms**

The Challenge

- Contemporary Verification Languages were drafted with RTL simulation in mind
- SV perfomance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes

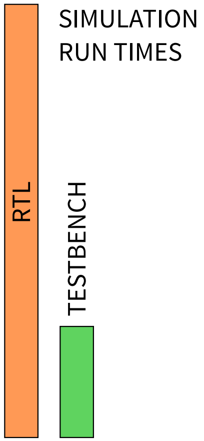
Testbenches for System Level Verification

The Trend

- More and more designs are being verified using Emulation Platforms
- Shorter product cycles are driving design teams to use Virtual Platforms

The Challenge

- **Contemporary Verification Languages were drafted with RTL simulation in mind**
- SV perfomance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes



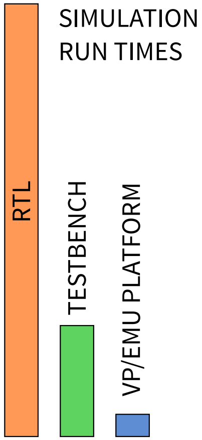
Testbenches for System Level Verification

The Trend

- More and more designs are being verified using Emulation Platforms
- Shorter product cycles are driving design teams to use Virtual Platforms

The Challenge

- Contemporary Verification Languages were drafted with RTL simulation in mind
- **SV perfomance becomes a bottleneck when testbenching Emulation/ESL Platforms**
- SV DPI overhead adds to testbench performance woes



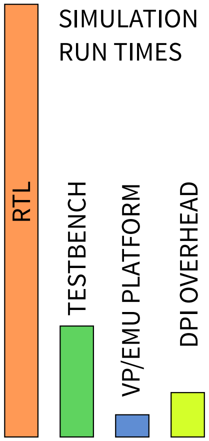
Testbenches for System Level Verification

The Trend

- More and more designs are being verified using Emulation Platforms
- Shorter product cycles are driving design teams to use Virtual Platforms

The Challenge

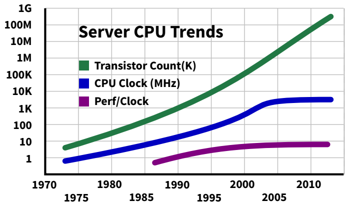
- Contemporary Verification Languages were drafted with RTL simulation in mind
- SV perfomance becomes a bottleneck when testbenching Emulation/ESL Platforms
- **SV DPI overhead adds to testbench performance woes**



The Multicore Challenge

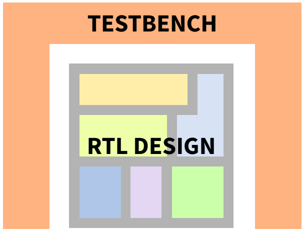
The Trend

- ***The Free Lunch is over***
- The numbers of cores in server processors are projected to grow to hundreds in next 5 years



The Challenge

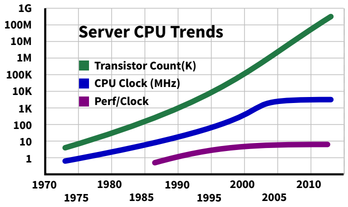
- Possible to partition RTL structurally because all variables are statically allocated
- No such automatic partioning possible for the dynamically allocated testbench
- Contemporary Verification Languages do not support parallel programming for behavioral code



The Multicore Challenge

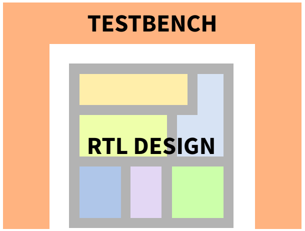
The Trend

- *The Free Lunch is over*
- **The numbers of cores in server processors are projected to grow to hundreds in next 5 years**



The Challenge

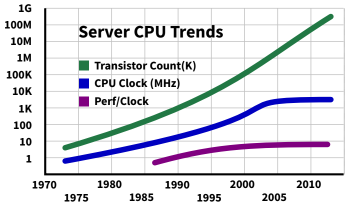
- Possible to partition RTL structurally because all variables are statically allocated
- No such automatic partioning possible for the dynamically allocated testbench
- Contemporary Verification Languages do not support parallel programming for behavioral code



The Multicore Challenge

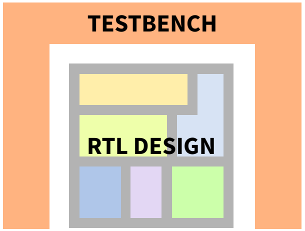
The Trend

- *The Free Lunch is over*
- The numbers of cores in server processors are projected to grow to hundreds in next 5 years



The Challenge

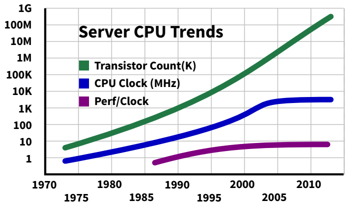
- **Possible to partition RTL structurally because all variables are statically allocated**
- No such automatic partioning possible for the dynamically allocated testbench
- Contemporary Verification Languages do not support parallel programming for behavioral code



The Multicore Challenge

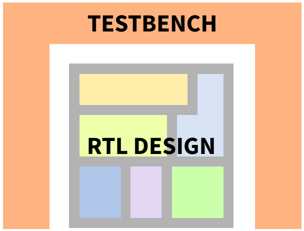
The Trend

- *The Free Lunch is over*
- The numbers of cores in server processors are projected to grow to hundreds in next 5 years



The Challenge

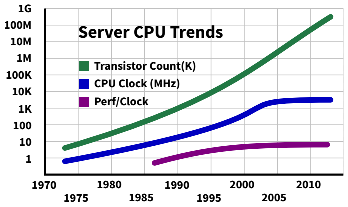
- Possible to partition RTL structurally because all variables are statically allocated
- **No such automatic partioning possible for the dynamically allocated testbench**
- Contemporary Verification Languages do not support parallel programming for behavioral code



The Multicore Challenge

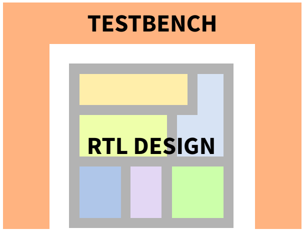
The Trend

- *The Free Lunch is over*
- The numbers of cores in server processors are projected to grow to hundreds in next 5 years



The Challenge

- Possible to partition RTL structurally because all variables are statically allocated
- No such automatic partioning possible for the dynamically allocated testbench
- **Contemporary Verification Languages do not support parallel programming for behavioral code**



In this section …

Verification Trends and Challenges

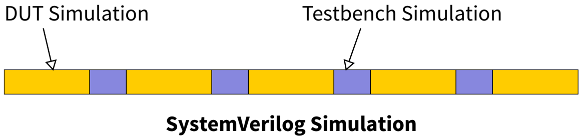
Vlang Language Features and Implementation

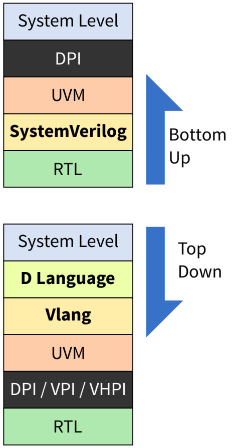
Widening the Horizon

Top Down Verification Stack

System Verilog integrates tightly with RTL;
Vlang with System Level

- Vlang is built on top of D Programming Language which provides ABI compatibility with C/C++
- Vlang interfaces with RTL using DPI
  - DPI overhead is compensated by parallel execution of testbench
- Vlang offers zero communication overhead when integrating with Emulation Platforms and with Virtual Platforms

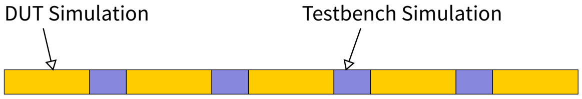




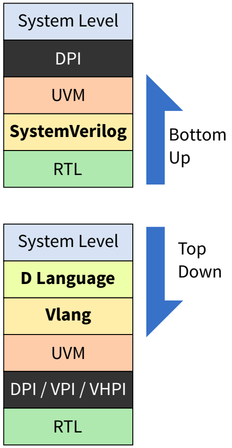
Top Down Verification Stack

System Verilog integrates tightly with RTL;
Vlang with System Level

- Vlang is built on top of D Programming Language which provides ABI compatibility with C/C++
- Vlang interfaces with RTL using DPI
    - DPI overhead is compensated by parallel execution of testbench
- Vlang offers zero communication overhead when integrating with Emulation Platforms and with Virtual Platforms



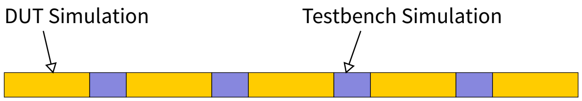
SystemVerilog Simulation



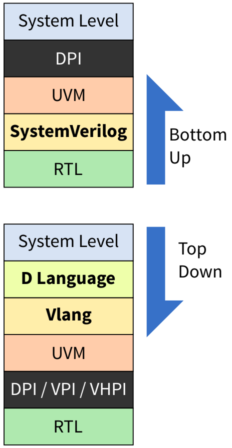
Top Down Verification Stack

System Verilog integrates tightly with RTL;
Vlang with System Level

- **Vlang is built on top of D Programming Language which provides ABI compatibility with C/C++**
- Vlang interfaces with RTL using DPI
  - DPI overhead is compensated by parallel execution of testbench
- Vlang offers zero communication overhead when integrating with Emulation Platforms and with Virtual Platforms



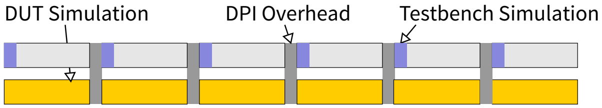
SystemVerilog Simulation



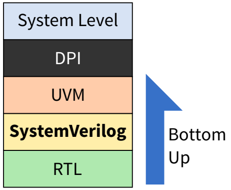
Top Down Verification Stack

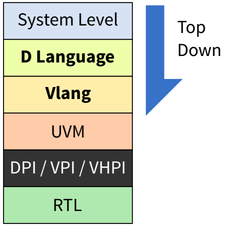
System Verilog integrates tightly with RTL;
Vlang with System Level

- Vlang is built on top of D Programming Language which provides ABI compatibility with C/C++
- **Vlang interfaces with RTL using DPI**
  - **DPI overhead is compensated by parallel execution of testbench**
- Vlang offers zero communication overhead when integrating with Emulation Platforms and with Virtual Platforms



Vlang RTL Cosimulation

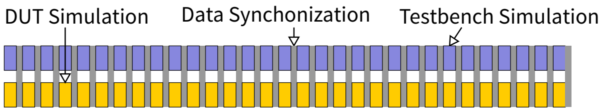




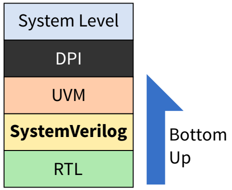
Top Down Verification Stack

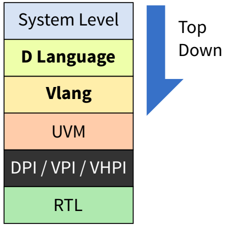
System Verilog integrates tightly with RTL;
Vlang with System Level

- Vlang is built on top of D Programming Language which provides ABI compatibility with C/C++
- Vlang interfaces with RTL using DPI
  - DPI overhead is compensated by parallel execution of testbench
- **Vlang offers zero communication overhead when integrating with Emulation Platforms and with Virtual Platforms**

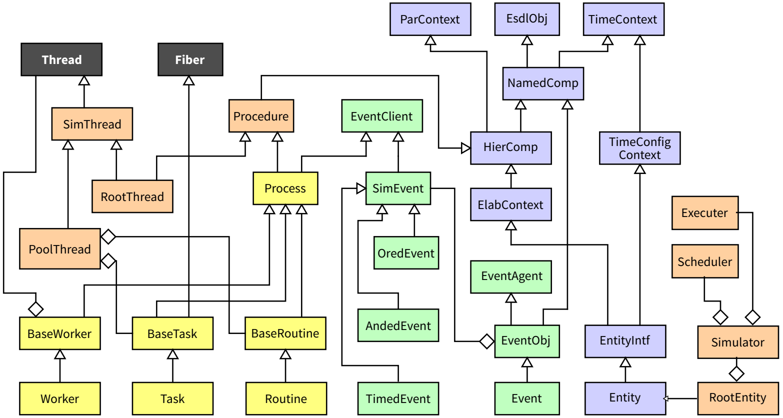


Vlang Cosimulation with Virtual/Emulation Platforms





Vlang Core Infrastructure



Bit Logic Queue Randomizable Constraint

vlang data package

Port ExPort Channel Signal

vlang comm package

Processes in Vlang

- **Processes and Forks (and for that matter everything else) in Vlang is an object**
  - **You can pass an Event, a Process, or a Fork as an argument to a function**
- Joining a fork can be done flexibly with the Fork object
- Forks and Processes can be suspended, disabled, aborted etc

```
void frop() {
  Fork zoo = fork
    ({ // fork1
       foo();
     },
     { // fork2
       bar();
     }).joinAny();
  // Some Code
  zoo.join();
  zoo.abortTree();
}
void foo() {
  auto proc = Process.self();
  proc.abortTree();
}
void bar() {
  // wait for fork branch
  Fork ff = Fork.self();
  ff.joinAny();
  // ....
}
```

Processes in Vlang

- Processes and Forks (and for that matter everything else) in Vlang is an object
  - You can pass an Event, a Process, or a Fork as an argument to a function
- **Joining a fork can be done flexibly with the Fork object**
- Forks and Processes can be suspended, disabled, aborted etc

```
void frop() {
  Fork zoo = fork
    ({ // fork1
       foo();
     },
     { // fork2
       bar();
     }).joinAny();
  // Some Code
  zoo.join();
  zoo.abortTree();
}
void foo() {
  auto proc = Process.self();
  proc.abortTree();
}
void bar() {
  // wait for fork branch
  Fork ff = Fork.self();
  ff.joinAny();
  // ....
}
```

Processes in Vlang

- Processes and Forks (and for that matter everything else) in Vlang is an object
  - You can pass an Event, a Process, or a Fork as an argument to a function
- Joining a fork can be done flexibly with the Fork object
- **Forks and Processes can be suspended, disabled, aborted etc**

```
void frop() {
  Fork zoo = fork
    ({ // fork1
       foo();
     },
     { // fork2
       bar();
     }).joinAny();
  // Some Code
  zoo.join();
  zoo.abortTree();
}
void foo() {
  auto proc = Process.self();
  proc.abortTree();
}
void bar() {
  // wait for fork branch
  Fork ff = Fork.self();
  ff.joinAny();
  // ....
}
```

Constrained Randomization

- ***System Verilog style Constrained Randomization***
- Class needs to be derived from `Randomizable`
- `mixin Randomization` magic makes `randomize` polymorphic
- Within UVM (and that is what matters) all the ugliness is gone
- Support for common arithmetic, logical and comparison operators
- Support for array and dynamic array randomization
- Support for `if-else`, `foreach`
  - Can be freely nested

```
class Foo: Randomizable {
  mixin Randomization;
  @rand!8 byte[] foo;
  @rand Logic!12 baz;
}
class Bar: Foo {
  mixin Randomization;
  @rand ubyte[8] bar;
  Constraint! q{
    foo.length > 2; // array
    baz[0..8] == 16;
  } cstFooLength;
  Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
      if(i > 4) /*condition*/
        f + i < 32 && f > 16;
    }
  } cstFoo;
}
void main() {
  Foo randObj = new Bar();
  for (size_t i=0; i!=10; ++i) {
    randObj.randomize();
  }
}
```

Constrained Randomization

- System Verilog style *Constrained Randomization*
- **Class needs to be derived from `Randomizable`**
- `mixin Randomization` magic makes `randomize` polymorphic
- Within UVM (and that is what matters) all the ugliness is gone
- Support for common arithmetic, logical and comparison operators
- Support for array and dynamic array randomization
- Support for `if-else`, `foreach`
  - Can be freely nested

```
class Foo: Randomizable {
  mixin Randomization;
  @rand!8 byte[] foo;
  @rand Logic!12 baz;
}
class Bar: Foo {
  mixin Randomization;
  @rand ubyte[8] bar;
  Constraint! q{
    foo.length > 2; // array
    baz[0..8] == 16;
  } cstFooLength;
  Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
      if(i > 4) /*condition*/
        f + i < 32 && f > 16;
    }
  } cstFoo;
}
void main() {
  Foo randObj = new Bar();
  for (size_t i=0; i!=10; ++i) {
    randObj.randomize();
  }
}
```

Constrained Randomization

- System Verilog style *Constrained Randomization*
- Class needs to be derived from `Randomizable`
- **`mixin Randomization` magic makes `randomize` polymorphic**
- Within UVM (and that is what matters) all the ugliness is gone
- Support for common arithmetic, logical and comparison operators
- Support for array and dynamic array randomization
- Support for `if-else`, `foreach`
  - Can be freely nested

```
class Foo: Randomizable {
  mixin Randomization;
  @rand!8 byte[] foo;
  @rand Logic!12 baz;
}
class Bar: Foo {
  mixin Randomization;
  @rand ubyte[8] bar;
  Constraint! q{
    foo.length > 2; // array
    baz[0..8] == 16;
  } cstFooLength;
  Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
      if(i > 4) /*condition*/
        f + i < 32 && f > 16;
    }
  } cstFoo;
}
void main() {
  Foo randObj = new Bar();
  for (size_t i=0; i!=10; ++i) {
    randObj.randomize();
  }
}
```

Constrained Randomization

- System Verilog style *Constrained Randomization*
- Class needs to be derived from `Randomizable`
- `mixin Randomization` magic makes `randomize` polymorphic
- **Within UVM (and that is what matters) all the ugliness is gone**
- Support for common arithmetic, logical and comparison operators
- Support for array and dynamic array randomization
- Support for `if-else`, `foreach`
  - Can be freely nested

```
class Foo: uvm_object {
  mixin uvm_object_utils;
  @rand!8 byte[] foo;
  @rand Logic!12 baz;
}
class Bar: Foo {
  mixin uvm_object_utils;
  @rand ubyte[8] bar;
  Constraint! q{
    foo.length > 2; // array
    baz[0..8] == 16;
  } cstFooLength;
  Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
      if(i > 4) /*condition*/
        f + i < 32 && f > 16;
    }
  } cstFoo;
}
void main() {
  Foo randObj = new Bar();
  for (size_t i=0; i!=10; ++i) {
    randObj.randomize();
  }
}
```

Constrained Randomization

- System Verilog style *Constrained Randomization*
- Class needs to be derived from `Randomizable`
- `mixin Randomization` magic makes `randomize` polymorphic
- Within UVM (and that is what matters) all the ugliness is gone
- **Support for common arithmetic, logical and comparison operators**
- Support for array and dynamic array randomization
- Support for `if-else`, `foreach`
  - Can be freely nested

```
class Foo: uvm_object {
  mixin uvm_object_utils;
  @rand!8 byte[] foo;
  @rand Logic!12 baz;
}
class Bar: Foo {
  mixin uvm_object_utils;
  @rand ubyte[8] bar;
  Constraint! q{
    foo.length > 2; // array
    baz[0..8] == 16;
  } cstFooLength;
  Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
      if(i > 4) /*condition*/
        f + i < 32 && f > 16;
    }
  } cstFoo;
}
void main() {
  Foo randObj = new Bar();
  for (size_t i=0; i!=10; ++i) {
    randObj.randomize();
  }
}
```

Constrained Randomization

- System Verilog style *Constrained Randomization*
- Class needs to be derived from `Randomizable`
- `mixin Randomization` magic makes `randomize` polymorphic
- Within UVM (and that is what matters) all the ugliness is gone
- Support for common arithmetic, logical and comparison operators
- **Support for array and dynamic array randomization**
- Support for `if-else`, `foreach`
  - Can be freely nested

```
class Foo: uvm_object {
  mixin uvm_object_utils;
  @rand!8 byte[] foo;
  @rand Logic!12 baz;
}
class Bar: Foo {
  mixin uvm_object_utils;
  @rand ubyte[8] bar;
  Constraint! q{
    foo.length > 2; // array
    baz[0..8] == 16;
  } cstFooLength;
  Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
      if(i > 4) /*condition*/
        f + i < 32 && f > 16;
    }
  } cstFoo;
}
void main() {
  Foo randObj = new Bar();
  for (size_t i=0; i!=10; ++i) {
    randObj.randomize();
  }
}
```

Constrained Randomization

- System Verilog style *Constrained Randomization*
- Class needs to be derived from `Randomizable`
- `mixin Randomization` magic makes `randomize` polymorphic
- Within UVM (and that is what matters) all the ugliness is gone
- Support for common arithmetic, logical and comparison operators
- Support for array and dynamic array randomization
- **Support for `if-else`, `foreach`**
  - **Can be freely nested**

```
class Foo: uvm_object {
  mixin uvm_object_utils;
  @rand!8 byte[] foo;
  @rand Logic!12 baz;
}
class Bar: Foo {
  mixin uvm_object_utils;
  @rand ubyte[8] bar;
  Constraint! q{
    foo.length > 2; // array
    baz[0..8] == 16;
  } cstFooLength;
  Constraint! q{
    foreach(i, f; bar) f <= i;
    foreach(i, f; foo) {
      if(i > 4) /*condition*/
        f + i < 32 && f > 16;
    }
  } cstFoo;
}
void main() {
  Foo randObj = new Bar();
  for (size_t i=0; i!=10; ++i) {
    randObj.randomize();
  }
}
```

Constrained Randomization under the Hood

- **UDP `@rand` is used to identify the class elements that need randomization**
  - **UDP has no runtime performance or memory footprint overhead**
- A Constraint engine is initialized only when `randomize` is called on a *Randomizable Object* the first time
- Constraints are parsed and corresponding BDD equations are generated for each constraint block
  - To ease debug, these equations can be dumped using compile time flags
- *Randomization Stability* is incorporated by initializing each thread with its own random number generator and seed

```
override public CstBlock getCstExpr() {
  auto cstExpr = new CstBlock;
  cstExpr ~= (cstRandArrElem!q{bar}(_outer)).lte (cstRandArrIndex!q{bar}(_outer));
  /* IF Block: (    cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))*/
  /*condition*/
  cstExpr ~= // Conditions
    ((cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))).implies( // End of Conditi
      (((cstRandArrElem!q{foo}(_outer) + cstRandArrIndex!q{foo}(_outer)).lth
        (cstRand(32, _outer))) .logicAnd((cstRandArrElem!q{foo}(_outer)).gth
                                         (cstRand(16, _outer)))));
  // END OF BLOCK
    return cstExpr;
}
```

Constrained Randomization under the Hood

- UDP `@rand` is used to identify the class elements that need randomization
  - UDP has no runtime performance or memory footprint overhead
- **A Constraint engine is initialized only when `randomize` is called on a *Randomizable Object* the first time**
- Constraints are parsed and corresponding BDD equations are generated for each constraint block
  - To ease debug, these equations can be dumped using compile time flags
- *Randomization Stability* is incorporated by initializing each thread with its own random number generator and seed

```
override public CstBlock getCstExpr() {
  auto cstExpr = new CstBlock;
  cstExpr ~= (cstRandArrElem!q{bar}(_outer)).lte (cstRandArrIndex!q{bar}(_outer));
  /* IF Block: (    cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))*/
  /*condition*/
  cstExpr ~= // Conditions
    ((cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))).implies( // End of Conditi
      (((cstRandArrElem!q{foo}(_outer) + cstRandArrIndex!q{foo}(_outer)).lth
        (cstRand(32, _outer))) .logicAnd((cstRandArrElem!q{foo}(_outer)).gth
                                        (cstRand(16, _outer)))));
  // END OF BLOCK
    return cstExpr;
}
```

Constrained Randomization under the Hood

- UDP `@rand` is used to identify the class elements that need randomization
  - UDP has no runtime performance or memory footprint overhead
- A Constraint engine is initialized only when `randomize` is called on a *Randomizable Object* the first time
- **Constraints are parsed and corresponding BDD equations are generated for each constraint block**
  - **To ease debug, these equations can be dumped using compile time flags**
- *Randomization Stability* is incorporated by initializing each thread with its own random number generator and seed

```
override public CstBlock getCstExpr() {
  auto cstExpr = new CstBlock;
  cstExpr ~= (cstRandArrElem!q{bar}(_outer)).lte (cstRandArrIndex!q{bar}(_outer));
  /* IF Block: (    cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))*/
  /*condition*/
  cstExpr ~= // Conditions
    ((cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))).implies( // End of Conditi
      (((cstRandArrElem!q{foo}(_outer) + cstRandArrIndex!q{foo}(_outer)).lth
        (cstRand(32, _outer))) .logicAnd((cstRandArrElem!q{foo}(_outer)).gth
                                         (cstRand(16, _outer)))));
  // END OF BLOCK
    return cstExpr;
}
```

Constrained Randomization under the Hood

- UDP `@rand` is used to identify the class elements that need randomization
  - UDP has no runtime performance or memory footprint overhead
- A Constraint engine is initialized only when `randomize` is called on a *Randomizable Object* the first time
- Constraints are parsed and corresponding BDD equations are generated for each constraint block
  - To ease debug, these equations can be dumped using compile time flags
- ***Randomization Stability* is incorporated by initializing each thread with its own random number generator and seed**

```
override public CstBlock getCstExpr() {
  auto cstExpr = new CstBlock;
  cstExpr ~= (cstRandArrElem!q{bar}(_outer)).lte (cstRandArrIndex!q{bar}(_outer));
  /* IF Block: (    cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))*/
  /*condition*/
  cstExpr ~= // Conditions
    ((cstRandArrIndex!q{foo}(_outer)).gth ( cstRand(4, _outer))).implies( // End of Conditic
      (((cstRandArrElem!q{foo}(_outer) + cstRandArrIndex!q{foo}(_outer)).lth
        (cstRand(32, _outer))) .logicAnd((cstRandArrElem!q{foo}(_outer)).gth
                                         (cstRand(16, _outer)))));
  // END OF BLOCK
    return cstExpr;
}
```

Universal Verification Methodology

- **Vlang implements a *word by word* port of SV UVM**
- Plus lots of automation magic, like the `uvm_object_utils` *mixin*
- `uvm_component_utils` automatically builds the subcomponents not built in `build_phase`
  - `@UVM_ACTIVE` UDP is considered
- Parallelization policy can be specified at `uvm_component` level (more later…)
- Vlang makes *multiple* `uvm_root` instances possible

```
import uvm;
enum bus_op_t: ubyte
  {BUS_READ, BUS_WRITE};
@UVM_DEFAULT
class bus_trans:
         uvm_sequence_item {
  mixin uvm_object_utils;
  @rand Bit!12 addr;
  @rand Bit!8 data;
  @UVM_NOPRINT
    @rand!256 uint[] payload;
  @rand bus_op_t op;
  this(string name="") {
    super(name);
  }
  Constraint!q{
    payload.length >= 8;
  } arrayC;
}
```

Universal Verification Methodology

- Vlang implements a *word by word* port of SV UVM
- **Plus lots of automation magic, like the `uvm_object_utils` *mixin***
- `uvm_component_utils` automatically builds the subcomponents not built in `build_phase`
  - `@UVM_ACTIVE` UDP is considered
- Parallelization policy can be specified at `uvm_component` level (more later…)
- Vlang makes *multiple* `uvm_root` instances possible

```
import uvm;
enum bus_op_t: ubyte
  {BUS_READ, BUS_WRITE};
@UVM_DEFAULT
class bus_trans:
         uvm_sequence_item {
  mixin uvm_object_utils;
  @rand Bit!12 addr;
  @rand Bit!8 data;
  @UVM_NOPRINT
    @rand!256 uint[] payload;
  @rand bus_op_t op;
  this(string name="") {
    super(name);
  }
  Constraint!q{
    payload.length >= 8;
  } arrayC;
}
```

Universal Verification Methodology

- Vlang implements a *word by word* port of SV UVM
- Plus lots of automation magic, like the `uvm_object_utils` *mixin*
- **`uvm_component_utils` automatically builds the subcomponents not built in `build_phase`**
  - **`@UVM_ACTIVE` UDP is considered**
- Parallelization policy can be specified at `uvm_component` level (more later…)
- Vlang makes *multiple* `uvm_root` instances possible

```
class env: uvm_env {
  mixin uvm_component_utils;
  this(string name,
       uvm_component parent) {
    super(name, parent);
  }
  @parallelize(ParallelPolicy.SINGLE)
  agent foo_agent[16];
}

class agent: uvm_agent {
  mixin uvm_component_utils;
  @UVM_ACTIVE
  uvm_sequencer!(bus_req, bus_rsp)
              sequence_controller;
  @UVM_ACTIVE
  foo_driver  driver;
  foo_monitor monitor;
  this(string name,
```

Universal Verification Methodology

- Vlang implements a *word by word* port of SV UVM
- Plus lots of automation magic, like the `uvm_object_utils` *mixin*
- `uvm_component_utils` automatically builds the subcomponents not built in `build_phase`
  - `@UVM_ACTIVE` UDP is considered
- **Parallelization policy can be specified at `uvm_component` level (more later…)**
- Vlang makes *multiple* `uvm_root` instances possible

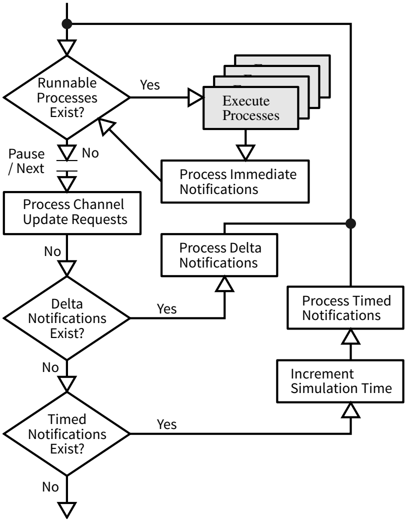
```
class env: uvm_env {
  mixin uvm_component_utils;
  this(string name,
       uvm_component parent) {
    super(name, parent);
  }
  @parallelize(ParallelPolicy.SINGLE)
  agent foo_agent[16];
}

class agent: uvm_agent {
  mixin uvm_component_utils;
  @UVM_ACTIVE
  uvm_sequencer!(bus_req, bus_rsp)
              sequence_controller;
  @UVM_ACTIVE
  foo_driver  driver;
  foo_monitor monitor;
  this(string name,
```

Universal Verification Methodology

- Vlang implements a *word by word* port of SV UVM
- Plus lots of automation magic, like the `uvm_object_utils` *mixin*
- `uvm_component_utils` automatically builds the subcomponents not built in `build_phase`
  - `@UVM_ACTIVE` UDP is considered
- Parallelization policy can be specified at `uvm_component` level (more later...)
- **Vlang makes *multiple* `uvm_root` instances possible**

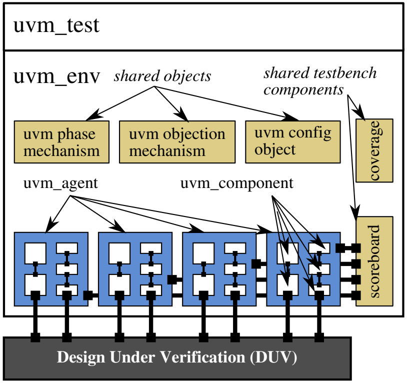
```
    fork({auto the_sequence =
      new sequenceA!(bus_req, bus_rsp)(
      the_sequence.start(sequence_contr
     });
    waitForks();
    phase.drop_objection(this);
  }
}
class my_root: uvm_root {
  mixin uvm_component_utils;
  env my_env;
  override void initial() {
    run_test();
  }
}
void main() {
  import std.random: uniform;
  auto root =
    uvm_fork!(my_root,"test")(42);
  root.wait_for_end_of_elaboration();
  // Vlang runs in background
  root.join();
}
```

Multicore UVM


Runnable Processes Exist?
Yes
Execute Processes
Process Immediate Notifications
Pause / Next
No
Process Channel Update Requests
No
Process Delta Notifications
Delta Notifications Exist?
Yes
Process Timed Notifications
No
Increment Simulation Time
Timed Notifications Exist?
Yes
No

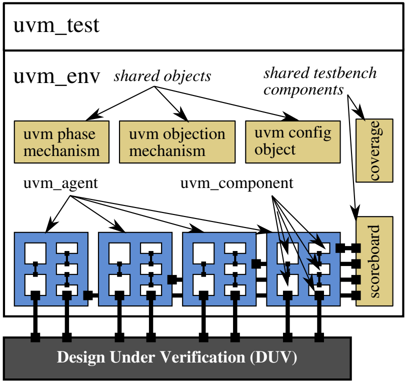
- **Vlang simulator is multicore capable**
- Vlang implementation of Multicore UVM takes advantage of the fact that there is minimal interaction between different `uvm_agents`
- Vlang provides an abstraction `ParContext` to manage parallelization
- The uvm constructs (like `uvm_objection`), that are shared between the components, are *synchronized* in the UVM base library implementation
  - In Vlang port of UVM, a `uvm_component` implements `ParContext`

Multicore UVM



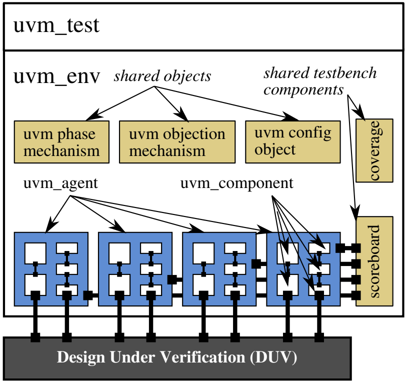
- Vlang simulator is multicore capable
- **Vlang implementation of Multicore UVM takes advantage of the fact that there is minimal interaction between different `uvm_agents`**
- Vlang provides an abstraction `ParContext` to manage parallelization
- The uvm constructs (like `uvm_objection`), that are shared between the components, are *synchronized* in the UVM base library implementation
    - In Vlang port of UVM, a `uvm_component` implements `ParContext`

Multicore UVM



- Vlang simulator is multicore capable
- Vlang implementation of Multicore UVM takes advantage of the fact that there is minimal interaction between different `uvm_agents`
- **Vlang provides an abstraction `ParContext` to manage parallelization**
- The uvm constructs (like `uvm_objection`), that are shared between the components, are *synchronized* in the UVM base library implementation
  - In Vlang port of UVM, a `uvm_component` implements `ParContext`

Multicore UVM



- Vlang simulator is multicore capable
- Vlang implementation of Multicore UVM takes advantage of the fact that there is minimal interaction between different `uvm_agents`
- Vlang provides an abstraction `ParContext` to manage parallelization
- **The uvm constructs (like `uvm_objection`), that are shared between the components, are *synchronized* in the UVM base library implementation**
  - **In Vlang port of UVM, a `uvm_component` implements `ParContext`**

Multicore UVM Implementation

- **Under the hood Vlang UVM implementation uses critical regions to make sure that there are no race conditions and deadlocks between the threads**
- UVM phase tasks are attached to `uvm_components` that they belong to – with each component provding a *Parallelization Context*

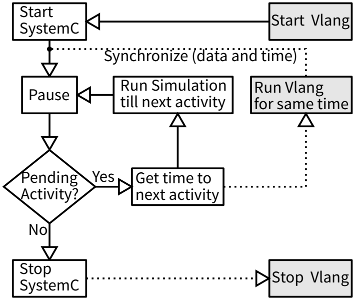
```
public void raise_objection
   (uvm_object obj = null,
    string description = ””,
    int count = 1) {
  if(obj is null) obj = m_top;
  synchronized(this) {
    _m_cleared = false;
    _m_top_all_dropped = false;
  }
  m_raise (obj,obj,description,count);
}
final public void m_raise
  (uvm_object obj,
   uvm_object source_obj,
   string description = ””,
   int count = 1) {
  synchronized(m_total_count) {
    if(obj in m_total_count) {
      m_total_count[obj] += count;
    }
    else {
      m_total_count[obj] = count;
    }
  }
  synchronized(m_source_count) {
    if(source_obj is obj) {
      if(obj in m_source_count) {
```

Multicore UVM Implementation

- Under the hood Vlang UVM implementation uses critical regions to make sure that there are no race conditions and deadlocks between the threads
- **UVM phase tasks are attached to `uvm_components` that they belong to – with each component provding a *Parallelization Context***

```
// class uvm_task_phase
void execute(uvm_component comp,
               uvm_phase phase) {
 fork({
  auto proc = Process.self;
  proc.srandom(uvm_create_random_seed
               (phase.get_type_name(),
                comp.get_full_name()));
  phase.inc_m_num_procs_not_yet_returned;
  uvm_sequencer_base seqr
    = cast(uvm_sequencer_base) comp;
  if (seqr !is null) {
    seqr.start_phase_sequence(phase);
  }
  exec_task(comp, phase);
  phase.dec_m_num_procs_not_yet_returned;
 }).setAffinity(comp);
}
```

Interfacing with SystemVerilog and SystemC

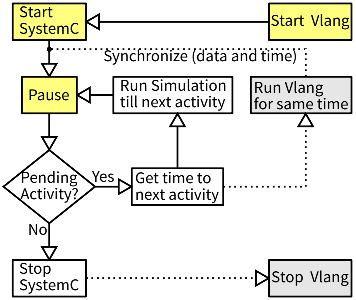


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM(”top”);
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC

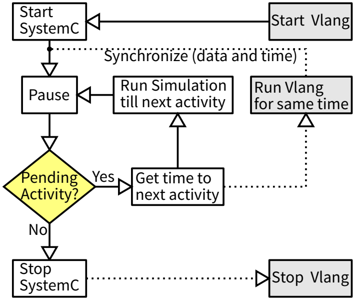


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM(”top”);
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC

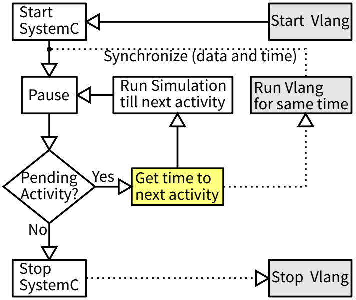


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM("top");
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC

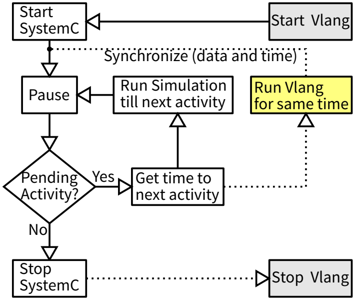


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM(”top”);
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC

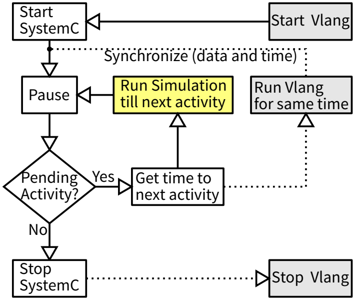


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM("top");
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC

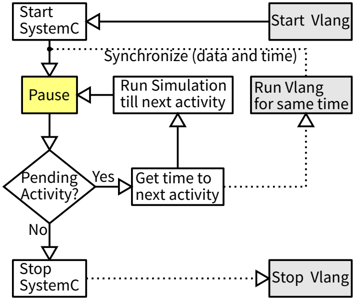


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();           // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();       // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM("top");
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC

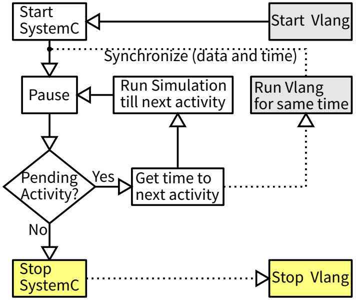


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM(”top”);
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC



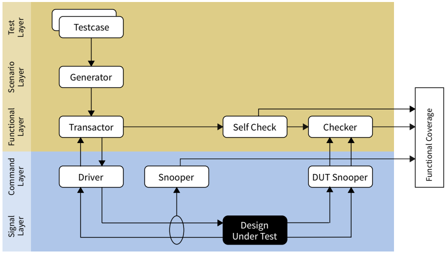
- Vlang simulator can be fully synchronized with SystemC and SystemVerilog

- With Systemc, Vlang can lock at delta cycle level

```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM(”top”);
  sc_start( SC_ZERO_TIME );
  while(sc_pending_activity()) {
    sc_core::sc_time time_ =
      sc_time_to_pending_activity();
    // start vlang simulation for given time
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time step
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemVerilog and SystemC

- **The idea is to implement the BFM along with the design and pass the transaction to the BFM from Vlang**
- Vlang implements special TLM channels for interfacing with external simulators
- Each Vlang UVM agent communicates independently with SV/SystemC blocking only when the transaction FIFO channel is full/empty
- SystemVerilog BFM pulls transactions from the channel using DPI-C interface



Interfacing with SystemVerilog and SystemC

- The idea is to implement the BFM along with the design and pass the transaction to the BFM from Vlang
- **Vlang implements special TLM channels for interfacing with external simulators**
- Each Vlang UVM agent communicates independently with SV/SystemC blocking only when the transaction FIFO channel is full/empty
- SystemVerilog BFM pulls transactions from the channel using DPI-C interface

```
class my_root: uvm_root {
  mixin uvm_component_utils;
  env my_env;
  uvm_tlm_fifo_egress!bus_req fifo;
  uvm_get_port!bus_req data_in;
  override void initial() {
    fifo = new
      uvm_tlm_fifo_egress!bus_req("fifo",null,10);
    run_test();
  }
  override void connect_phase(uvm_phase phase) {
    my_env.drv.data_out.connect(fifo.put_export);
    data_in.connect(fifo.get_export);
  }
}
uvm_root_entity!my_root root;
extern(C) void dpi_pull_req(int* addr,int* data) {
    bus_req req;
    root.data_in.get(req);
    // get the addr and data from transaction
}
void main() {
  root = uvm_fork!(my_root, "test")(0);
  root.get_uvm_root.wait_for_end_of_elaboration();
  root.join();
}
```

Interfacing with SystemVerilog and SystemC

- The idea is to implement the BFM along with the design and pass the transaction to the BFM from Vlang
- Vlang implements special TLM channels for interfacing with external simulators
- **Each Vlang UVM agent communicates independently with SV/SystemC blocking only when the transaction FIFO channel is full/empty**
- SystemVerilog BFM pulls transactions from the channel using DPI-C interface

```
class my_root: uvm_root {
  mixin uvm_component_utils;
  env my_env;
  uvm_tlm_fifo_egress!bus_req fifo;
  uvm_get_port!bus_req data_in;
  override void initial() {
    fifo = new
      uvm_tlm_fifo_egress!bus_req("fifo",null,10);
    run_test();
  }
  override void connect_phase(uvm_phase phase) {
    my_env.drv.data_out.connect(fifo.put_export);
    data_in.connect(fifo.get_export);
  }
}
uvm_root_entity!my_root root;
extern(C) void dpi_pull_req(int* addr,int* data) {
    bus_req req;
    root.data_in.get(req);
    // get the addr and data from transaction
}
void main() {
  root = uvm_fork!(my_root, "test")(0);
  root.get_uvm_root.wait_for_end_of_elaboration();
  root.join();
}
```

Interfacing with SystemVerilog and SystemC

- The idea is to implement the BFM along with the design and pass the transaction to the BFM from Vlang
- Vlang implements special TLM channels for interfacing with external simulators
- Each Vlang UVM agent communicates independently with SV/SystemC blocking only when the transaction FIFO channel is full/empty
- **SystemVerilog BFM pulls transactions from the channel using DPI-C interface**

```
class my_root: uvm_root {
  mixin uvm_component_utils;
  env my_env;
  uvm_tlm_fifo_egress!bus_req fifo;
  uvm_get_port!bus_req data_in;
  override void initial() {
    fifo = new
      uvm_tlm_fifo_egress!bus_req(”fifo”,null,10);
    run_test();
  }
  override void connect_phase(uvm_phase phase) {
    my_env.drv.data_out.connect(fifo.put_export);
    data_in.connect(fifo.get_export);
  }
}
uvm_root_entity!my_root root;
extern(C) void dpi_pull_req(int* addr,int* data) {
    bus_req req;
    root.data_in.get(req);
    // get the addr and data from transaction
}
void main() {
  root = uvm_fork!(my_root, ”test”)(0);
  root.get_uvm_root.wait_for_end_of_elaboration();
  root.join();
}
```

In this section …

Software Freedom

A most important, but also most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language this influence is, whether we like it or not, an influence on our thinking habits.... A programming language is a tool that has profound influence on our thinking habits.

- Edsger Dijkstra

- **Vlang connects the verification technology to mainstream software world**
- In the process a host of software constructs and libraries become available

Software Freedom

A most important, but also most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language this influence is, whether we like it or not, an influence on our thinking habits.... A programming language is a tool that has profound influence on our thinking habits.

- Edsger Dijkstra

- Vlang connects the verification technology to mainstream software world
- **In the process a host of software constructs and libraries become available**

Self Checking Testbenches

- **The D Programming Language provides Unittest environment that can be included inside any user code scope**
- Unittest environments can be very useful in creating self checking testbenches in Vlang

```
class ethernet_pkt:
  uvm_sequence_item {
  @rand ubyte[6] dst_addr;
  @rand ubyte[6] src_addr;
  @rand ubyte[2] pkt_type;
  // other fields
  @rand!1500 ubyte[] payload;
  @rand ubyte[4] fcs;
  override void post_randomize() {
    fcs = calc_fcs(payload);
  }
  ubyte[4] calc_fcs(ubyte[] dt) {
    // implementation ....
  }
  unittest {
    auto data = [0, 42, 12, 54];
    assert(calc_fcs(data) ==
           [191, 198, 145, 119];
  }
}
```

Self Checking Testbenches

- The D Programming Language provides Unittest environment that can be included inside any user code scope
- **Unittest environments can be very useful in creating self checking testbenches in Vlang**

```
class ethernet_pkt:
  uvm_sequence_item {
  @rand ubyte[6] dst_addr;
  @rand ubyte[6] src_addr;
  @rand ubyte[2] pkt_type;
  // other fields
  @rand!1500 ubyte[] payload;
  @rand ubyte[4] fcs;
  override void post_randomize() {
    fcs = calc_fcs(payload);
  }
  ubyte[4] calc_fcs(ubyte[] dt) {
    // implementation ....
  }
  unittest {
    auto data = [0, 42, 12, 54];
    assert(calc_fcs(data) ==
           [191, 198, 145, 119];
  }
}
```

Fork me on Github

Home Page	`http://vlang.org`
Repository (Vlang)	`https://github.com/coverify/vlang`
Repository (Vlang UVM)	`https://github.com/coverify/vlang-uvm`
Compiler	DMD Version 2.066 (available at `http://dlang.org`)
License (Vlang)	Boost Software License, Version 1.0
Lincese (Vlang UVM)	Apache 2.0 License
Maintainer	Puneet Goel `<puneet@coverify.com>`

Questions?