Vlang

A System Level Verification Perspective

Puneet Goel <puneet@coverify.com>

accellera
SYSTEMS INITIATIVE™

2015
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
INDIA

In this section …

Runtime Efficiency

Multi UVM Root

Coverification

Top Down Verification

Modeling

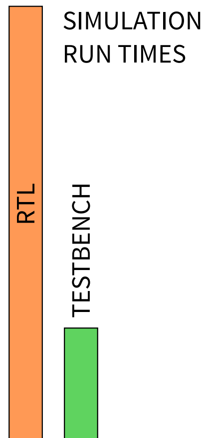
Open Source

Testbenches for System Level

- State-of-the-art HVLs like SystemVerilog were drafted with RTL simulation in mind
- SV performance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes

ESL Testbenches need to be …

- Faster by at least an order of magnitude
- ABI Compatible with C/C++



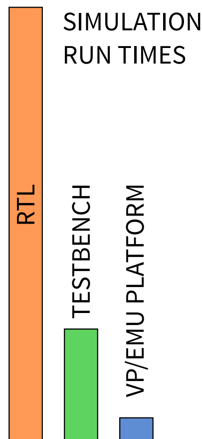
© Accellera Systems Initiative

Testbenches for System Level

- State-of-the-art HVLs like SystemVerilog were drafted with RTL simulation in mind
- SV performance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes

ESL Testbenches need to be …

- Faster by at least an order of magnitude
- ABI Compatible with C/C++



Testbenches for System Level

- State-of-the-art HVLs like SystemVerilog were drafted with RTL simulation in mind
- SV performance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes

ESL Testbenches need to be …

- Faster by at least an order of magnitude
- ABI Compatible with C/C++

Testbenches for System Level

- State-of-the-art HVLs like SystemVerilog were drafted with RTL simulation in mind
- SV performance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes

ESL Testbenches need to be …

- Faster by at least an order of magnitude
- ABI Compatible with C/C++

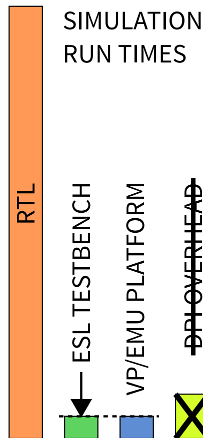
© Accellera Systems Initiative

Testbenches for System Level

- State-of-the-art HVLs like SystemVerilog were drafted with RTL simulation in mind
- SV performance becomes a bottleneck when testbenching Emulation/ESL Platforms
- SV DPI overhead adds to testbench performance woes

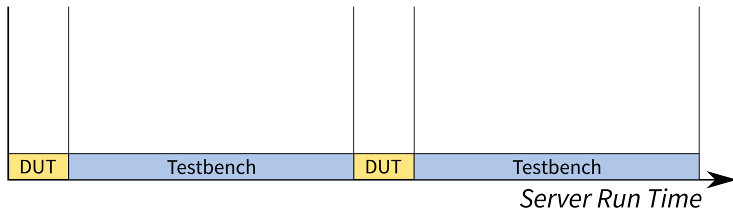
ESL Testbenches need to be …

- Faster by at least an order of magnitude
- ABI Compatible with C/C++



HVLs Are Essentially Single Threaded

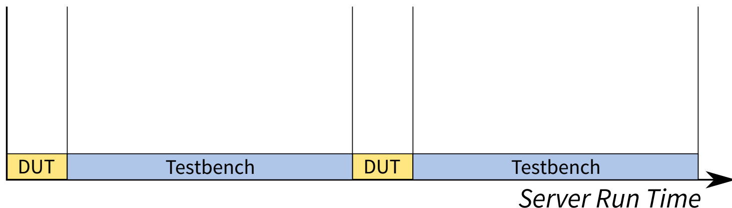
- Both SystemVerilog and SystemC (as of now) run on a single OS thread
  - SV/SystemC use Cooperative Threading for Fork/Spawn
- Testbench can be made efficient by invoking concurrent threads
- And even more efficient by running the testbench in parallel



© Accellera Systems Initiative

HVLs Are Essentially Single Threaded

- Both SystemVerilog and SystemC (as of now) run on a single OS thread
  - SV/SystemC use Cooperative Threading for Fork/Spawn
- Testbench can be made efficient by invoking concurrent threads
- And even more efficient by running the testbench in parallel

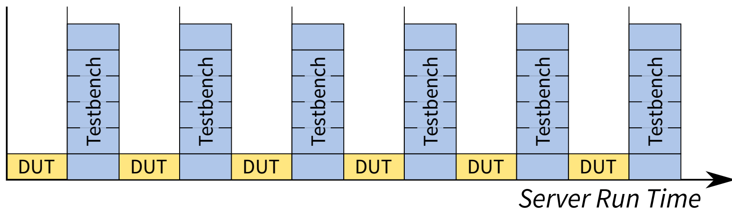


© Accellera Systems Initiative

accellera SYSTEMS INITIATIVE
2015 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

HVLs Are Essentially Single Threaded

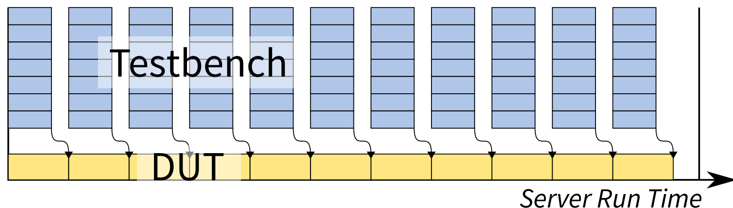
- Both SystemVerilog and SystemC (as of now) run on a single OS thread
  - SV/SystemC use Cooperative Threading for Fork/Spawn
- Testbench can be made efficient by invoking concurrent threads
- And even more efficient by running the testbench in parallel



© Accellera Systems Initiative

HVLs Are Essentially Single Threaded

- Both SystemVerilog and SystemC (as of now) run on a single OS thread
  - SV/SystemC use Cooperative Threading for Fork/Spawn
- Testbench can be made efficient by invoking concurrent threads
- And even more efficient by running the testbench in parallel



© Accellera Systems Initiative

Inspired by 61 Cores!

By end of 2015, Intel's Knights Landing (KNL) processor would become commercially available

- It will have minimum 60 and a maximum 72 cores
- Each core will run 4 threads in parallel
  - With up to 288 threads running in parallel, concurrency in application programs becomes an essential aspect of coding
- KNL will also have a minimum 16GB on-chip DRAM
  - To maximize potential, Go Parallel
  - Running multiple simulations on a Multicore Server is the quickest way to hit *Memory Wall*
- Multicore is here to Stay! Are you Ready!!

Inspired by 61 Cores!

By end of 2015, Intel's Knights Landing (KNL) processor would become commercially available

- It will have minimum 60 and a maximum 72 cores
- Each core will run 4 threads in parallel
  - With up to 288 threads running in parallel, concurrency in application programs becomes an essential aspect of coding
- KNL will also have a minimum 16GB on-chip DRAM
  - To maximize potential, Go Parallel
  - Running multiple simulations on a Multicore Server is the quickest way to hit *Memory Wall*
- Multicore is here to Stay! Are you Ready!!

© Accellera Systems Initiative

Inspired by 61 Cores!

By end of 2015, Intel's Knights Landing (KNL) processor would become commercially available

- It will have minimum 60 and a maximum 72 cores
- Each core will run 4 threads in parallel
  - With up to 288 threads running in parallel, concurrency in application programs becomes an essential aspect of coding
- KNL will also have a minimum 16GB on-chip DRAM
  - To maximize potential, Go Parallel
  - Running multiple simulations on a Multicore Server is the quickest way to hit *Memory Wall*
- Multicore is here to Stay! Are you Ready!!

© Accellera Systems Initiative

Inspired by 61 Cores!

By end of 2015, Intel's Knights Landing (KNL) processor would become commercially available

- It will have minimum 60 and a maximum 72 cores
- Each core will run 4 threads in parallel
  - With up to 288 threads running in parallel, concurrency in application programs becomes an essential aspect of coding
- KNL will also have a minimum 16GB on-chip DRAM
  - To maximize potential, Go Parallel
  - Running multiple simulations on a Multicore Server is the quickest way to hit *Memory Wall*
- Multicore is here to Stay! Are you Ready!!

© Accellera Systems Initiative

Inspired by 61 Cores!

By end of 2015, Intel's Knights Landing (KNL) processor would become commercially available

- It will have minimum 60 and a maximum 72 cores
- Each core will run 4 threads in parallel
  - With up to 288 threads running in parallel, concurrency in application programs becomes an essential aspect of coding
- KNL will also have a minimum 16GB on-chip DRAM
  - To maximize potential, Go Parallel
  - Running multiple simulations on a Multicore Server is the quickest way to hit *Memory Wall*
- Multicore is here to Stay! Are you Ready!!

Inspired by 61 Cores!

By end of 2015, Intel's Knights Landing (KNL) processor would become commercially available

- It will have minimum 60 and a maximum 72 cores
- Each core will run 4 threads in parallel
  - With up to 288 threads running in parallel, concurrency in application programs becomes an essential aspect of coding
- KNL will also have a minimum 16GB on-chip DRAM
  - To maximize potential, Go Parallel
  - Running multiple simulations on a Multicore Server is the quickest way to hit *Memory Wall*
- Multicore is here to Stay! Are you Ready!!

Inspired by 61 Cores!

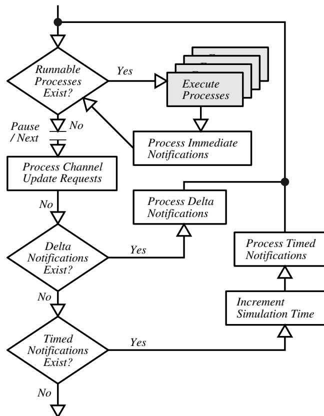
By end of 2015, Intel's Knights Landing (KNL) processor would become commercially available

- It will have minimum 60 and a maximum 72 cores
- Each core will run 4 threads in parallel
  - With up to 288 threads running in parallel, concurrency in application programs becomes an essential aspect of coding
- KNL will also have a minimum 16GB on-chip DRAM
  - To maximize potential, Go Parallel
  - Running multiple simulations on a Multicore Server is the quickest way to hit *Memory Wall*
- Multicore is here to Stay! Are you Ready!!

© Accellera Systems Initiative

Vlang is Multi-Core Enabled

- Vlang Simulator comes fitted with a Multicore Task Scheduler
- Customizing Multicore Parallelism in Vlang is easy



© Accellera Systems Initiative

Vlang is Multi-Core Enabled

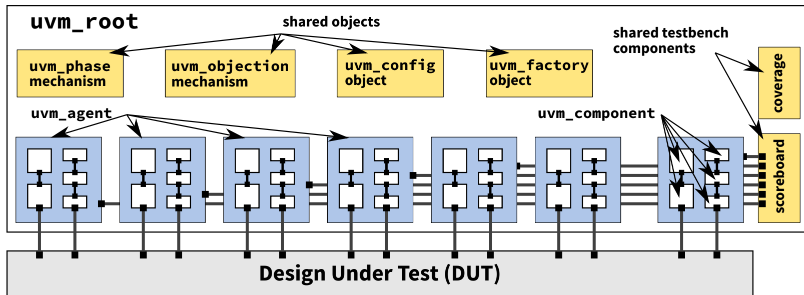
- Vlang Simulator comes fitted with a Multicore Task Scheduler
- Customizing Multicore Parallelism in Vlang is easy

```
class TestBench: RootEntity {
  uvm_root_entity!(apb_root) tb;
  this(string name) {
    super(name);
  }
}
int main() {
  TestBench test =
    new TestBench("test");
  test.multiCore(4, 0);
  test.elaborate();
  test.simulate();
  return 0;
}
```


© Accellera Systems Initiative

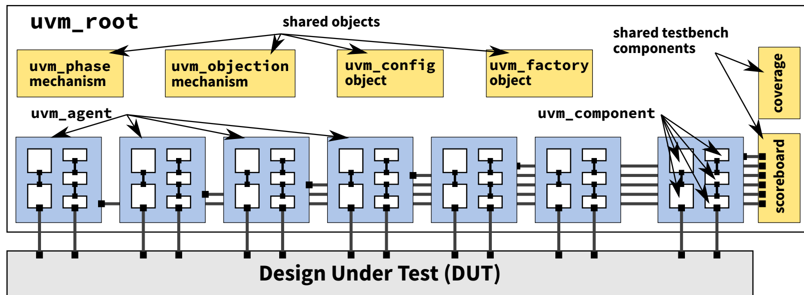
Vlang UVM Innovation – Multi Core UVM

- Most System Level Designs have multiple (TLM) Interfaces
- Each (TLM) Interface requires a VIP (or a `uvm_agent`)
  - Most VPIs have no interaction with other VIPs
  - This provides the right opportunity for parallelism
  - Vlang UVM implementation runs `uvm_agent` threads parallelly



Vlang UVM Innovation – Multi Core UVM

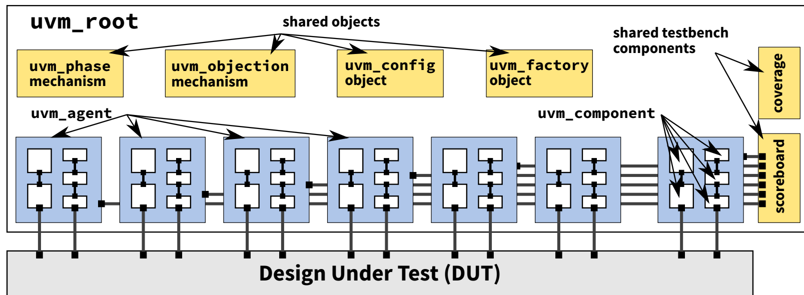
- Most System Level Designs have multiple (TLM) Interfaces
- Each (TLM) Interface requires a VIP (or a uvm_agent)
  - Most VPIs have no interaction with other VIPs
  - This provides the right opportunity for parallelism
  - Vlang UVM implementation runs uvm_agent threads parallelly



© Accellera Systems Initiative

Vlang UVM Innovation – Multi Core UVM

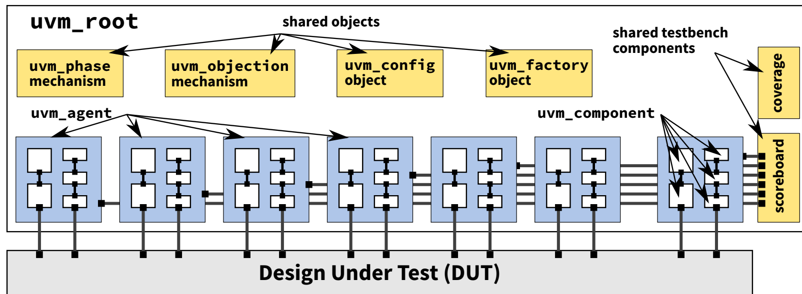
- Most System Level Designs have multiple (TLM) Interfaces
- Each (TLM) Interface requires a VIP (or a `uvm_agent`)
  - Most VPIs have no interaction with other VIPs
  - This provides the right opportunity for parallelism
  - Vlang UVM implementation runs `uvm_agent` threads parallelly



© Accellera Systems Initiative

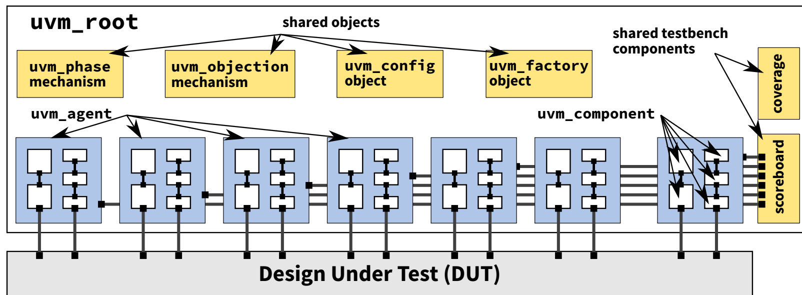
Vlang UVM Innovation – Multi Core UVM

- Most System Level Designs have multiple (TLM) Interfaces
- Each (TLM) Interface requires a VIP (or a `uvm_agent`)
  - Most VPIs have no interaction with other VIPs
  - This provides the right opportunity for parallelism
  - Vlang UVM implementation runs `uvm_agent` threads parallelly



Vlang UVM Innovation – Multi Core UVM

- Most System Level Designs have multiple (TLM) Interfaces
- Each (TLM) Interface requires a VIP (or a `uvm_agent`)
  - Most VPIs have no interaction with other VIPs
  - This provides the right opportunity for parallelism
  - Vlang UVM implementation runs `uvm_agent` threads parallelly



In this section …

Runtime Efficiency

Multi UVM Root

Coverification

Top Down Verification

Modeling

Open Source

accellera SYSTEMS INITIATIVE™
2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Vlang UVM Innovation – Multi UVM Root

- To conserve power many modules are switched off when inactive
- When the module is activated, it gets reset and driver is loaded
- UVM implementation provides singleton phases, not good enough for System Level Verification
- Hot plugin is another use case where singleton phasing becomes a bottleneck
- Vlang allows multiple UVM Root instances to overcome this limitation

```
class TestBench: RootEntity {
  uvm_root_entity!(sys1) tb1;
  uvm_root_entity!(sys2) tb2;
  uvm_root_entity!(sys3) tb3;
  uvm_root_entity!(sys4) tb4;
  this(string name) {
    super(name);
  }
}
int main() {
  TestBench test =
    new TestBench(”test”);
  test.multiCore(4, 0);
  test.elaborate();
  test.simulate();
  return 0;
```

Vlang UVM Innovation – Multi UVM Root

- To conserve power many modules are switched off when inactive
- When the module is activated, it gets reset and driver is loaded
- UVM implementation provides singleton phases, not good enough for System Level Verification
- Hot plugin is another use case where singleton phasing becomes a bottleneck
- Vlang allows multiple UVM Root instances to overcome this limitation

```
class TestBench: RootEntity {
  uvm_root_entity!(sys1) tb1;
  uvm_root_entity!(sys2) tb2;
  uvm_root_entity!(sys3) tb3;
  uvm_root_entity!(sys4) tb4;
  this(string name) {
    super(name);
  }
}
int main() {
  TestBench test =
    new TestBench("test");
  test.multiCore(4, 0);
  test.elaborate();
  test.simulate();
  return 0;
```


© Accellera Systems Initiative

Vlang UVM Innovation – Multi UVM Root

- To conserve power many modules are switched off when inactive
- When the module is activated, it gets reset and driver is loaded
- UVM implementation provides singleton phases, not good enough for System Level Verification
- Hot plugin is another use case where singleton phasing becomes a bottleneck
- Vlang allows multiple UVM Root instances to overcome this limitation

```
class TestBench: RootEntity {
  uvm_root_entity!(sys1) tb1;
  uvm_root_entity!(sys2) tb2;
  uvm_root_entity!(sys3) tb3;
  uvm_root_entity!(sys4) tb4;
  this(string name) {
    super(name);
  }
}
int main() {
  TestBench test =
    new TestBench(”test”);
  test.multiCore(4, 0);
  test.elaborate();
  test.simulate();
  return 0;
```

Vlang UVM Innovation – Multi UVM Root

- To conserve power many modules are switched off when inactive
- When the module is activated, it gets reset and driver is loaded
- UVM implementation provides singleton phases, not good enough for System Level Verification
- Hot plugin is another use case where singleton phasing becomes a bottleneck
- Vlang allows multiple UVM Root instances to overcome this limitation

```
class TestBench: RootEntity {
  uvm_root_entity!(sys1) tb1;
  uvm_root_entity!(sys2) tb2;
  uvm_root_entity!(sys3) tb3;
  uvm_root_entity!(sys4) tb4;
  this(string name) {
    super(name);
  }
}
int main() {
  TestBench test =
    new TestBench(”test”);
  test.multiCore(4, 0);
  test.elaborate();
  test.simulate();
  return 0;
```


© Accellera Systems Initiative

Vlang UVM Innovation – Multi UVM Root

- To conserve power many modules are switched off when inactive
- When the module is activated, it gets reset and driver is loaded
- UVM implementation provides singleton phases, not good enough for System Level Verification
- Hot plugin is another use case where singleton phasing becomes a bottleneck
- Vlang allows multiple UVM Root instances to overcome this limitation

```
class TestBench: RootEntity {
  uvm_root_entity!(sys1) tb1;
  uvm_root_entity!(sys2) tb2;
  uvm_root_entity!(sys3) tb3;
  uvm_root_entity!(sys4) tb4;
  this(string name) {
    super(name);
  }
}
int main() {
  TestBench test =
    new TestBench(”test”);
  test.multiCore(4, 0);
  test.elaborate();
  test.simulate();
  return 0;
```

accellera SYSTEMS INITIATIVE

© Accellera Systems Initiative

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

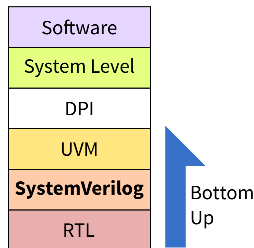
In this section …

- Runtime Efficiency
- Multi UVM Root
- **Coverification**
- Top Down Verification
- Modeling
- Open Source

Hardware Software Coverification

- First Level user of an SoC is a Software Programmer
- HVLs are built on top of RTL – Software interaction is Week
- Vlang is built on top of D Programming Language, A Systems Programming Laguage

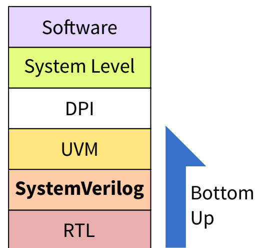
Hardware Software Coverification

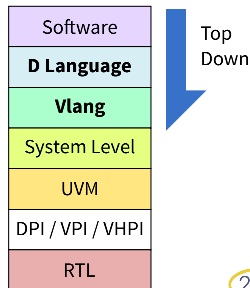


- First Level user of an SoC is a Software Programmer
- HVLs are built on top of RTL – Software interaction is Week
- Vlang is built on top of D Programming Language, A Systems Programming Laguage

Hardware Software Coverification

- First Level user of an SoC is a Software Programmer
- HVLs are built on top of RTL – Software interaction is Week
- Vlang is built on top of D Programming Language, A Systems Programming Laguage

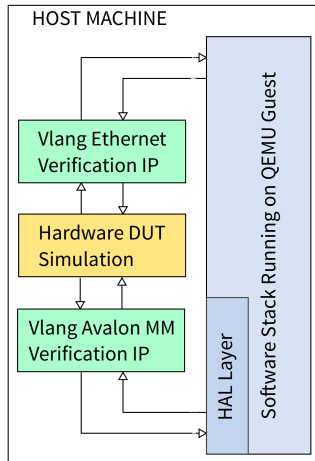




© Accellera Systems Initiative

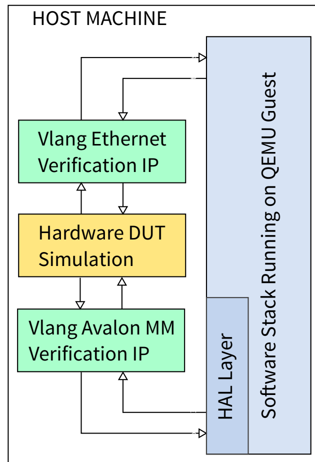
A Quick Coverification Use Case

- QEMU is fast becoming the platform of choice for embedded software development and test
- A convenient way to exchange data with QEMU is via shared file descriptors
- Vlang VIP can directly tap a file descriptor and feed the transaction to simulation
- Data coming out of DUT is reverse fed into QEMU



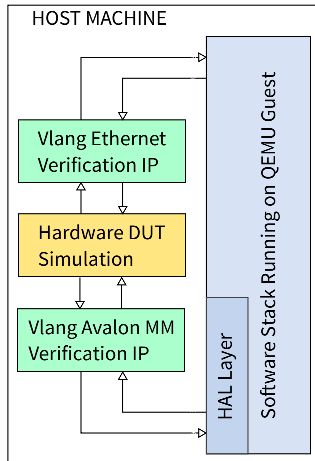
A Quick Coverification Use Case

- QEMU is fast becoming the platform of choice for embedded software development and test
- A convenient way to exchange data with QEMU is via shared file descriptors
- Vlang VIP can directly tap a file descriptor and feed the transaction to simulation
- Data coming out of DUT is reverse fed into QEMU



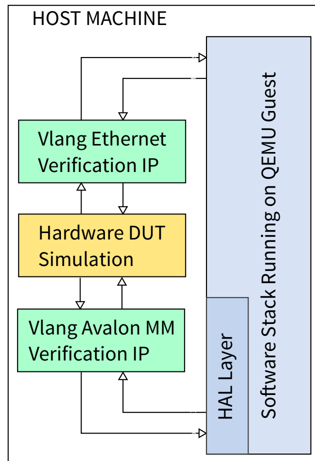
A Quick Coverification Use Case

- QEMU is fast becoming the platform of choice for embedded software development and test
- A convenient way to exchange data with QEMU is via shared file descriptors
- Vlang VIP can directly tap a file descriptor and feed the transaction to simulation
- Data coming out of DUT is reverse fed into QEMU



A Quick Coverification Use Case

- QEMU is fast becoming the platform of choice for embedded software development and test
- A convenient way to exchange data with QEMU is via shared file descriptors
- Vlang VIP can directly tap a file descriptor and feed the transaction to simulation
- Data coming out of DUT is reverse fed into QEMU



© Accellera Systems Initiative

In this section …

Runtime Efficiency

Multi UVM Root

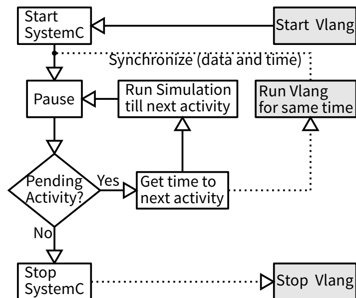
Coverification

Top Down Verification

Modeling

Open Source

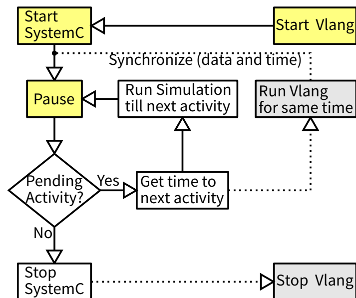
Interfacing with SystemC



- 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 ti
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time ste
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemC

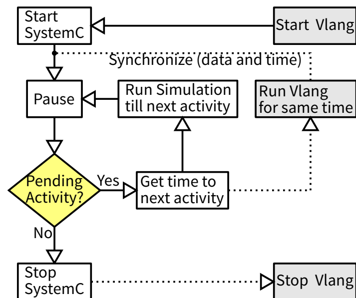



```
int main(int argc, char* argv[]) {
  initEsdl();          // initialize vlang
  int scresult =
    sc_core::sc_elab_and_sim(argc, argv);
  finalizeEsdl();      // stop vlang
  return 0;
}
int sc_main( int argc, char* argv[]) {
  sc_set_time_resolution(1, SC_PS);
  top = new SYSTEM("top");
  sc_start(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

- Vlang simulator can be fully synchronized with SystemC and SystemVerilog
- With Systemc, Vlang can lock at delta cycle level

accellera SYSTEMS INITIATIVE
2015 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

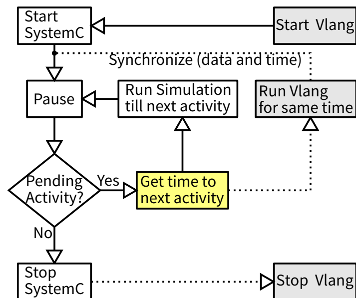
Interfacing with SystemC



- 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 ti
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time ste
    esdlWait();
  }
  return 0;
}
```

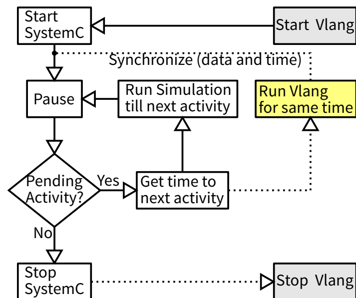
Interfacing with SystemC



- 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 ti
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time ste
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemC

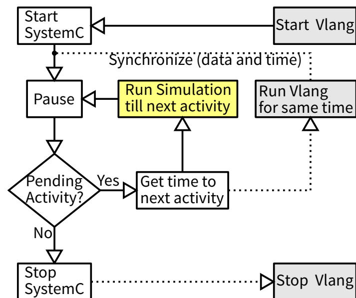


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog
- With Systemc, Vlang can lock at delta cycle level

```
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 ti
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time ste
    esdlWait();
  }
  return 0;
}
```


© Accellera Systems Initiative

Interfacing with SystemC

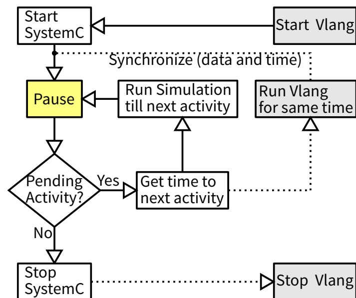


- Vlang simulator can be fully synchronized with SystemC and SystemVerilog
- With Systemc, Vlang can lock at delta cycle level

```
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 ti
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time ste
    esdlWait();
  }
  return 0;
}
```


© Accellera Systems Initiative

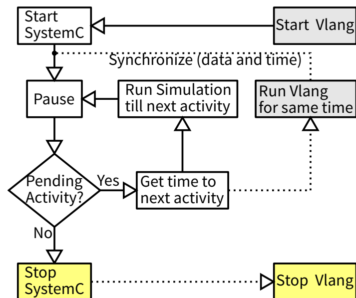
Interfacing with SystemC



- 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 ti
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time ste
    esdlWait();
  }
  return 0;
}
```

Interfacing with SystemC



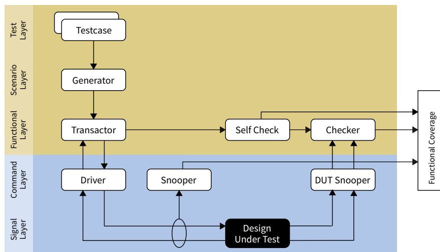
- Vlang simulator can be fully synchronized with SystemC and SystemVerilog
- With Systemc, Vlang can lock at delta cycle level

```
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 ti
    esdlStartSimFor(time_.value());
    sc_start(time_);
    // wait for vlang to complete time ste
    esdlWait();
  }
  return 0;
}
```


© Accellera Systems Initiative

Interfacing with SystemVerilog

- 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

- 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
    run_test();
  }
  override void connect_phase(uvm_phase phase
    my_env.drv.data_out.connect(fifo.put_expor
    data_in.connect(fifo.get_export);
  }
}
uvm_root_entity!my_root root;
extern(C) void dpi_pull_req(int* addr,int* da
    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_elaborati
  root.join();
}
```

accellera SYSTEMS INITIATIVE™

© Accellera Systems Initiative

Interfacing with SystemVerilog

- 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
    run_test();
  }
  override void connect_phase(uvm_phase phase
    my_env.drv.data_out.connect(fifo.put_expo
    data_in.connect(fifo.get_export);
  }
}
uvm_root_entity!my_root root;
extern(C) void dpi_pull_req(int* addr,int* da
    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_elaborati
  root.join();
}
```

accellera SYSTEMS INITIATIVE™
© Accellera Systems Initiative

Interfacing with SystemVerilog

- 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
    run_test();
  }
  override void connect_phase(uvm_phase phase
    my_env.drv.data_out.connect(fifo.put_expo
    data_in.connect(fifo.get_export);
  }
}
uvm_root_entity!my_root root;
extern(C) void dpi_pull_req(int* addr,int* da
    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_elaborati
  root.join();
}
```


© Accellera Systems Initiative

In this section …

Runtime Efficiency

Multi UVM Root

Coverification

Top Down Verification

Modeling

Open Source

accellera SYSTEMS INITIATIVE™

© Accellera Systems Initiative

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Modelling Hardware and Software

- How do you code an Array of Associative Arrays in C++?
- Here is how you do it in D

  …

```
#include <vector>
#include <map>
#include <string>

void foo () {
  std::vector
    <std::map
     <std::string, int> > myVect;
  std::map<std::string,int> entry1;
  std::map<std::string,int> entry2;
  entry1["ABC"] = 1;
  entry1["DEF"] = 2;
  myVect.push_back(entry1);
  entry2["ABC"] = 5;
  entry2["RKD"] = 9;
  myVect.push_back(entry2);
}
```

Modelling Hardware and Software

- How do you code an Array of Associative Arrays in C++?
- Here is how you do it in D …

```
void foo() {
  int[string][] myVect =
    [["ABC": 1, "DEF": 2],
     ["ABC": 5, "RKD": 9]];
}
```

Modelling Hardware and Software

- It is way too cumbersome to extend C++
  - The result is RAW Macro based code, SC_MODULE like
- CRAVE is a modern Constrained Randomization Library in C++
  - No User Defined Attributes, No Reflections in C++, and Virtually no CTFE
  - CRAVE uses wrapper templates to create Random Variables
  - A Random Integer in Crave takes > 50 bytes!!
- Vlang has support for multi-dimensional array randomization and much more

© Accellera Systems Initiative

Modelling Hardware and Software

- It is way too cumbersome to extend C++
  - The result is RAW Macro based code, SC_MODULE like
- CRAVE is a modern Constrained Randomization Library in C++
  - No User Defined Attributes, No Reflections in C++, and Virtually no CTFE
  - CRAVE uses wrapper templates to create Random Variables
  - A Random Integer in Crave takes > 50 bytes!!
- Vlang has support for multi-dimensional array randomization and much more

© Accellera Systems Initiative

Modelling Hardware and Software

- It is way too cumbersome to extend C++
  - The result is RAW Macro based code, SC_MODULE like
- CRAVE is a modern Constrained Randomization Library in C++
  - No User Defined Attributes, No Reflections in C++, and Virtually no CTFE
  - CRAVE uses wrapper templates to create Random Variables
  - A Random Integer in Crave takes > 50 bytes!!
- Vlang has support for multi-dimensional array randomization and much more

```
struct packet2 : public packet {
  rand<int> foo;
  rand_vec<unsigned int> bar;

  packet2() : foo(this), bar(this) {
   constraint( foo() > 4 &&
               foo() < 64 );
   constraint( bar().size() % 4 == 0
               && bar().size() < 100 );
   constraint.foreach(bar, i,
      IF_THEN( i == 0,
               10 <= bar()[i] &&
               bar()[i] <= 20));
   constraint.foreach( bar, i,
      IF_THEN( i != 0,
               40 <= bar()[i] &&
               bar()[i] <= 100));
  };
}
```

accellera SYSTEMS INITIATIVE
© Accellera Systems Initiative

Modelling Hardware and Software

- It is way too cumbersome to extend C++
  - The result is RAW Macro based code, SC_MODULE like
- CRAVE is a modern Constrained Randomization Library in C++
  - No User Defined Attributes, No Reflections in C++, and Virtually no CTFE
  - CRAVE uses wrapper templates to create Random Variables
  - A Random Integer in Crave takes > 50 bytes!!
- Vlang has support for multi-dimensional array randomization and much more

```
class packet2 :  packet {
  mixin Randomization;
  @rand int foo;
  @rand!(8, 8) ubyte[][] bar;
  Constraint !q{
    foo > 4 && foo < 64;
  } foo_cst;
  Constraint !q{
    bar.length % 4 == 0;
    foreach(f; bar) {
      f.length > 4;
      foreach(i, e; f) {
        if(i > 4) e <= i;
      }
    }
  } bar_cst;
}
```

On Choosing the right Verification Language

One way to achieve high confidence is for verification engineers to transform specifications into an implementation model in a language different from the design language. This language is called verification language…– Hardware Design Verification, William K. Lams

In this section …

Runtime Efficiency

Multi UVM Root

Coverification

Top Down Verification

Modeling

Open Source

Fork me on Github

Being Open Source is an essential element of system Level…

- Too many flows and methodologies require flexibility that only open source can provide

Home Page	`http://vlang.org`
Repository (Vlang)	`https://github.com/coverify/vlang`
Repository (Vlang UVM)	`https://github.com/coverify/vlang-uvm`
Compiler	DMD Version 2.068 (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`>

accellera SYSTEMS INITIATIVE™
© Accellera Systems Initiative

2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Questions?

accellera SYSTEMS INITIATIVE™
2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA