RISC-V Compliance & Verification Techniques
Processor Cores and Custom Extensions

Simon Davidmann & Lee Moore - Imperas Software Ltd.
Richard Ho - Google, Inc.
Doug Letcher - Metrics Technology, Inc.
Kevin McDermott - Imperas Software Ltd.

© Accellera Systems Initiative
Introduction

• DV solution for RISC-V RTL cores

• Collaboration between Google, Imperas and Metrics

• Industry adoption has started

• The basis for RISC-V core verification in:
Agenda

• RISC-V Intro and Status
• Industry requirements for ISA Compliance and Hardware Design Verification
• RISC-V ISA Compliance
• Processor Hardware Design Verification
• RISC-V Instruction Stream Generation
• Hardware Design Verification Flow
• Demonstration walk-through
• Scaling verification using Cloud resources
Agenda

• RISC-V Intro and Status
  • Industry requirements for ISA Compliance and Hardware Design Verification
  • RISC-V ISA Compliance
  • Processor Hardware Design Verification
  • RISC-V Instruction Stream Generation
  • Hardware Design Verification Flow
  • Demonstration walk-through
  • Scaling verification using Cloud resources
What is RISC-V

- An open-source hardware Instruction Set Architecture (ISA)
- Started in 2010 at UC Berkeley
- Frozen base user spec released in 2014
- Fifth major RISC ISA design effort out of Berkeley
- Ratification of RISC-V Base ISA and Privileged Specifications, June 2019
RISC-V ushers in new era of silicon design
Ecosystem Growth

Foundries

Compilers / SW Dev Tools

EDA Tools / Design Enablement

IP Block Portfolios

OS Support

Simulators
Technical priorities in 20 focus areas

 Opcode Space Mgmt Standing Committee
 Software Standing Committee
 Base ISA Ratification Task Group
 Privileged ISA Spec Task Group
 UNIX-Class Platform Spec Task Group
 Formal Specification Task Group
 Trusted Execution Env Spec Task Group
 B Extension (Bit Manipulation) Task Group
 J Extension (Dynam. Translated Lang) Task Group
 P Extension (Packed-SIMD Inst) Task Group
 V Extension (Vector Ops) Task Group
 Cryptographic Extension Task Group
 Debug Specification Task Group
 Fast Interrupts Spec Task Group
 Memory Model Spec Task Group
 Processor Trace Spec Task Group
 Sv128 Specification Task Group
 Compliance Task Group

+ Security Committee and proposed Safety Task Group
RISC-V Bit Manipulation Instructions Working Group

https://github.com/riscv/riscv-bitmanip

• New, optional, not yet standardized, instruction extension
• The bitmanip instructions extend the RISC-V instruction set to enable efficient bit manipulation
• This includes operations like:
  – counting bits, leading zeros, etc.
  – bit extraction
  – rotations, shifting and reversing
• Current status: specification under review – 0.9.0

• Imperas has modeled / maintains and includes in its executable reference model
• Available now: https://github.com/riscv/riscv-bitmanip
  – Includes Imperas free riscvOVPsim simulator as reference
RISC-V Vector Instructions Working Group

https://github.com/riscv/riscv-v-spec

• Optional, not yet standardized, instruction extension
• Been developed by RISC-V founders for years
• Specifications available 0.7.1

• GitHub free download reference simulator
  https://github.com/riscv/riscv-ovpsim
• Imperas has implemented latest draft Vector to riscvOVPsim free simulator
• Imperas is a reference implementation delivered to early users of Vector RTL
More Information:

- [https://riscv.org](https://riscv.org)
- [www.imperas.com](http://www.imperas.com), [www.OVPworld.org](http://www.OVPworld.org)
- [https://github.com/riscv/riscv-compliance](https://github.com/riscv/riscv-compliance) (RISC-V compliance suite & reference simulator)
- [https://github.com/riscv/riscv-bitmanip](https://github.com/riscv/riscv-bitmanip) (bitmanip spec & reference simulator)
- [https://github.com/riscv/riscv-v-spec](https://github.com/riscv/riscv-v-spec) (vector spec)
- [https://github.com/riscv/riscv-ovpsim](https://github.com/riscv/riscv-ovpsim) (free riscvOVPsim reference simulator)
RISC-V Cores - Commercial and Open Source

• Commercial IP Providers
  – SiFive, Andes, Codasip, Syntacore, ....

• OpenHW Group
  – PULP RI5CY, Ariane /ETH Zurich => Core-V

• CHIPS Alliance
  – SweRV / WD

• lowRISC
  – PULP Zero-riscy /ETH Zurich => Ibex
RISC-V: Flexibility within the ISA framework

Ecosystem support - Commercial and Open Source

• Commercial
  – IAR, Lauterbach, Segger, ExpressLogic, Imperas, ...

• Key Open Source activities include
  – GNU tools, gcc, gdb, ...
  – LLVM, LLDB, ...
  – FreeRTOS, Zephyr, Linux, ...
Verification Tools

- Simulation: Imperas (OVPsim), spike, qemu, ...
- Verification: Google (open source test generator), Valtrix (commercial test generator), ...
Agenda

- RISC-V Intro and Status
- Industry requirements for ISA Compliance and Hardware Design Verification
- RISC-V ISA Compliance
- Processor Hardware Design Verification
- RISC-V Instruction Stream Generation
- Hardware Design Verification Flow
- Demonstration walk-through
- Scaling verification using Cloud resources
Compliance for RISC-V is Important

Q: What is meant by “compliance”?
A: The device works within the envelope of the agreed specifications

Q: Is there an easy process or path to follow to develop methodologies/tools for compliance?
A: NO – all established ISAs are single company controlled and those companies work extremely hard on proprietary solutions to ensure that all designs that go out their door work correctly – so RISC-V has to pioneer compliance collectively

Q: What happens if the RISC-V industry builds devices that are not complying with specifications?
A: Users cannot assume that tools like C compilers, operating systems, and application software will be transferable across devices and work correctly
Imperas key contributor to the RISC-V Compliance Suite

- Compliance suite is ‘work in progress’
- Two components
  - Test suites
    - Each suite focuses on a feature set of the RISC-V envelope
    - Initial focus is instructions, user mode spec, e.g. rv32i, rv32im, rv32imc, rv64i, ...
    - Awaiting RISC-V platform specifications to subset privilege spec, before starting privilege suites
  - Framework
    - Make, bash, and scripts
      - Encapsulate compiler tools, linkers, simulators, and targets (Devices Under Test)
    - Includes simulator: as example target, and to generate reference signatures
    - Run: Select suite and target
      - Runs each test, target produces signatures, compares to saved golden reference signature
- Available: www.github.com/riscv/riscv-compliance
Agenda

• RISC-V Intro and Status
• Industry requirements for ISA Compliance and Hardware Design Verification
• RISC-V ISA Compliance
• Processor Hardware Design Verification
• RISC-V Instruction Stream Generation
• Hardware Design Verification Flow
• Demonstration walk-through
• Scaling verification using Cloud resources
ISA Compliance (agenda)

• An Intro to testing RISC-V ISA standard Compliance
• Scope and approaches to ISA compliance
• Components needed
  – Test Framework
  – Tests
  – Assembler/compiler
  – Reference simulator to generate known good results
  – Device Under Test with encapsulation to run it
• Overview of riscvOVPsim – compliance reference simulator
• RISC-V Foundation Compliance Suite walk-through and demo
  – Github
  – Flow, tests, select target, run, compare
  – Adding a new target
• Status and roadmap
RISC-V Getting your chip Verified

• Compliance Tests
• DV Tests
• Formal tools
• Simulators
• Emulators
• FPGA’s
RISC-V Compliance

• What is Compliance
• What is NOT Compliance
  – Where does Compliance end and DV begin
• Possible Methods
What is Compliance (in RISC-V)

• Participating in the Compliance WG, I have asked to get a definition - this has proven difficult.
• There are deep discussions about how and what to test in the compliance working group.
• Discussions frequently drift into – ‘oh-no that is DV, not compliance’, or ‘that is too deep/detailed for compliance testing’
• So what can we definitively say is compliance?
What is Compliance (in RISC-V)

• The device works within the envelope of the agreed specifications

• Compliance is adhering to the definition of the Instruction Set Architecture specification and its intended semantic behavior.

• Proving the ISA behavior should be independent of any understanding of the underlying micro-architectural implementation.
What is NOT Compliance (in RISC-V)

• We should not be concerned about compliance testing a given implementation in order to prove that it may
  – perform out of order execution
  – perform parallel execution (multiple dispatch)
  – execute multi/single cycle instruction(s)
  – have a shallow or deep pipeline
  – implement complex branch predictions
  – perform speculative execution based upon branch prediction
  – *enable contentious mode*
    – implement an instruction through trapped emulation
      • Extension-M : providing Mult but no Div
  – *disable contentious mode*
Compliance Testing Definition

• Compliance is attempting to test all of the possible instruction alternatives, without attempting to test all of the possible values either provided to, or produced by an execution unit, or the data paths being exercised.

• This causes many grey areas of overlaps between compliance and conventional DV, or a simple question:

Where does Compliance end, and DV begin?
Running the RISC-V compliance suite Framework

- RV32I
- RV32M
- ... (suite)
- eg, Ibex (design)
- eg, riscvOVPsim (golden)

Flow:
1. Suite
2. Compile
3. Execute
4. Compare
5. Report

© Accellera Systems Initiative
Compliance – Possible Methods

- For example, is it sufficient to exercise an `add` instruction by the following code snippet

```assembly
la x31, RESULT_ADDR
li x1, 2
li x2, 3
add x3, x2, x1  // x3 = 5
sw x3, x31(0)   // Save to SIGNATURE

RESULT_ADDR:
.word 0x0
```
Compliance – Possible Methods

- Or should specific bit patterns be exercised, for example a walking 1, or walking 0 in register values

```assembly
la x29, RESULT_ADDR
li x30, 0
for i1 in 0 to 31; do
    li x1, (0x1 << $i1)
    for i2 in 0 to 31; do
        li x2, (0x1 << $i2)
        add x3, x2, x1
        add x31, x30, x29 // Calculate a new (sw) ptr
        sw x3, x31(0)
        addi x30, 4
    done
done
RESULT_ADDR:
.word 0x0
```
Compliance – Possible Methods

• Given that we are adding a walking '1' to a 0 value in the first inner loop pass, then we know we are also creating a walking '1' in the result register (for an add instruction).

• This is pretty simple given an 'add' instruction, in the case of a more complex instruction, then to ensure we have a walking '1' in the result, we probably need some coverage metrics to ensure we meet our goals - more on that later.
Compliance – Possible Methods

• Question: is a walking '1' sufficient? This will not test a result overflow, underflow etc. so do we need to extend this simple test with some corner case values?

• for example define a list of special values
  – list = {(-ve largest), (-ve mid), -1, 0, 1, (+ve mid), (+ve largest)}
  – list = {0x80000000, 0xC0000000, 0xFFFFFFFF, 0x00000000, 0x00000001, 0x3FFFFFFF, 0x7FFFFFFF}

• Now unroll the lists as the data values
WARNING

• Serious Risk of Encroaching upon DV Territory
Compliance – Possible Methods

- Question: why not just go for the full range of values?

```
la x29, RESULT_ADDR
li x30, 0
for i1 in 0 to 0xFFFFFFFF; do
    li x1, $i1
    for i2 in 0 to 0xFFFFFFFF; do
        li x2, $i2
        add x3, x2, x1
        add x31, x30, x29
        sw x3, x31(0)
        addi x30, 4
    done
done
RESULT_ADDR:
.word 0x0
```
DANGER

• Deep inside DV Territory
Compliance – Possible Methods

• Question, do we also need to exercise each possible register combination
  – rDlist = x31, x30, x29, ...
  – rS1list = x31, x30, x29, ...
  – rS2list = x31, x30, x29, ...

• In which case, why not simply ...
Compliance – Possible Methods

// Allocate 3 registers from pool for BASE, OFFSET, PTR
la x(ADDBASE), RESULT_ADDR
li x(ADDOFF), 0
li x(ADDPTR), 0
foreach rD in rDlist; do
    foreach rS1 in rS1list; do
        foreach rS2 in rS2list; do
            for i1 in 0 to 0xFFFFFFFF; do
                li x(rS1), $i1
                for i2 in 0 to 0xFFFFFFFF; do
                    li x(rS2), $i2
                    add x(rD), x(rS2), x(rS1)
                    add x(ADDPTR), x(ADDOFF), x(ADDBASE)
                    sw x(rD), x(ADDPTR)(0)
                    addi x(ADDOFF), 4
                done
            done
        done
    done
done
Game Over

• You just generated too many permutations, and exceeded way out past DV territory
• RD   RS1   RS2   RS1v   RS2v
• $2^{25} \times 2^{25} \times 2^{25} \times 2^{32} \times 2^{32}$
• In this case it is a 32bit Architecture – change to 64bit, and the permutations increase accordingly
• This is simply not feasible
• remember we will also need to adhere to the framework constraints (what are these ?)
Test framework Constraints

• As yet not clearly defined!
• How much available memory for Code/Data?
• Effect:
  – How to handle large displacements for Branch, Jump
  – How to handle large immediate offsets in Load/Store
• Result:
  – Tradeoff, Tradeoff, Tradeoff
Compliance goals, For each Instruction

- Attempt to use every possible register as both an input and output (where an output is produced)
- For all immediate values (offset, displacement, shift etc) Attempt to show that all bits in the value have been exercised as all 1's, all 0's and both a 1 and 0, preferably in isolation
  - eg for a 4 bit immediate value, the following possible values
    - 0000, 0001, 0010, 0100, 1000,
    - 1110, 1101, 1011, 0111, 1111
- ie, avoiding simple 2-state toggle: 0000, 1111
Compliance goals, For each Instruction

• Attempt to provide input register values whereby corner cases are exercised
  – -MAX -MID, -MIN, -1, 0, 1, +MIN, +MID, +MAX (FP: +INF, -INF +0, -0)
  – Walking '1', Walking '0' values

• Attempt to produce output register values whereby corner cases are exercised
  – -MAX -MID, -MIN, -1, 0, 1, +MIN, +MID, +MAX (FP: +INF, -INF +0, -0)
  – Walking '1', Walking '0' values
Compliance OUT-OF-SCOPE, do not attempt

- Exercise Register/Value Cross coverage
  - Register indices, eg rs1 rs2 rd - cross coverage => 2**5 x 2**5 x 2**5
  - Register values, eg rs1=X rs2=Y - cross coverage => 2*32 x 2*32
- Exercise all possible immediate values
  - for a given 2**n immediate, do not exercise 2**n values
  - (Unless) small set, eg shift/rotate amount 2**5
- Branch/Jump Targets
  - TBD, related to the framework constraints (relative to PC is difficult to fully exploit)
- Load/Store offsets
  - Where an address is R + I, the the R can be inversely adjusted in order to offset far load/store
  - eg For a given address X, Rx=(X-4), Imm=4,

** note :

Register+Immediate targets of Load/Store/Branch/Jump, the target address can be brought back into range by rebasing the base pointer register so that an immediate is provided resulting in a valid target address, given a potential out of range intermediate immediate/displacement.
Compliance Test in Memory Structure

- TEXT section
  - Exception Handling Code (**contentious**)  
  - Startup Code
  - Test Code (Body of the test)
  - Shutdown Code

- DATA section
  - Debug/logging strings
  - Signature memory (Pre-initialized, Post-extracted)
Compliance Test Semantics

• Loader program to get memory in initial state
  – RTL Simulator $writemem()$
  – ISS readelf()
  – HW debug load
• Execute
  – Method of finish/exit detection
• Extract Signature (results)
• Compare Signature to golden reference
Compliance test Porting to target

• Compilation infrastructure
• Macro body definitions
• Linker script to describe the target memory structure
Compliance Code snippet RV32I - ADD

# Addresses for test data and results
la x1, test_A1_data
la x2, test_A1_res

# Load test data
lw x3, 0(x1)

# Register initialization
li x4, 0
li x5, 1

# Test: add <dst-reg>, <src1-reg>, <src2-reg>
add x4, x3, x4
add x5, x3, x5

# Store results to signature memory for extraction to signature file
sw x3, 0(x2)
sw x4, 4(x2)
sw x5, 8(x2)
Compliance Linker snippet (ibex)

```assembly
OUTPUT_ARCH( "riscv" )
ENTRY(_start)

SECTIONS {
   . = 0x00000000;
   .text.trap : { *(.text.trap) }
   . = 0x00000080;
   .text.init : { *(.text.init) }
   . = ALIGN(0x1000);
   .text : { *(.text) }
   . = ALIGN(0x1000);
   .data : { *(.data) }
   .data.string : { *(.data.string) }
   .bss : { *(.bss) }
   _end = .;
}
```
Running the RISC-V compliance suite

Suite

Compile

Execute

Compare

Report

RV32I
RV32M
...

eg, Ibex

eg, riscvOVPsim

Design

Golden

© Accellera Systems Initiative
Running the RISC-V compliance suite

1. Start with ssh to remote shell in Metrics Cloud Platform
2. Load tests etc. from Git, e.g. compliance suite
3. Run as if in local shell
4. In this example RV32I compliance suite
[moore@lnx33 ~]$ make hel~C
[moore@lnx33 ~]$
[moore@lnx33 ~]$
Compliance Test Quality Measurement

• Observing and measuring the Coverage Points
  – Instruction Type
  – Target RD Register usage
  – Source RS1, RS2 Register usage
  – Immediate Value usage (buckets, data points)
  – Register Value usage (buckets, data points)
Compliance Test Quality Measurement

• Coverage Results for ADD

  – Contact Lee Moore at moore@imperas.com
  – For latest results and updates presented at DVCon Europe 2019 in Munich
Compliance Test Quality Measurement

• Proving the Coverage measurements, by propagation of an intended result

Coverage observes the following

addi x2, x1=0x1, 0x1 // -> x2=0x2

This implies the following points are met for Instruction addi

Target Register [x2] => Covered
Target Register Value [1]=1 => Covered
Source Register [x1] => Covered
Source Register Value [0]=1 => Covered
Immediate Value [0]=1 => Covered
Compliance Test Quality Measurement

• If our code snippet looks like this, we are gradually improving our coverage - right?

```assembly
la x31, ADDR_SIGNATURE
addi x2, x1=0x1, 0x0 // -> x2=0x1
addi x2, x1=0x1, 0x1 // -> x2=0x2
addi x2, x1=0x1, 0x2 // -> x2=0x3
addi x2, x1=0x1, 0x4 // -> x2=0x5
addi x2, x1=0x1, 0x8 // -> x2=0x9
addi x2, x1=0x1, 0xF // -> x2=0x10
sw x2, x31(0) // Store Signature
```
Compliance Test Quality Measurement

- Coverage is monitoring the Execution unit, but ignoring the Load/Store Unit, so in fact

```
la x31, ADDR_SIGNATURE
addi x2, x1=0x1, 0x0 // -> x2=0x1 - DISCARDED
addi x2, x1=0x1, 0x1 // -> x2=0x2 - DISCARDED
addi x2, x1=0x1, 0x2 // -> x2=0x3 - DISCARDED
addi x2, x1=0x1, 0x4 // -> x2=0x5 - DISCARDED
addi x2, x1=0x1, 0x8 // -> x2=0x9 - DISCARDED
addi x2, x1=0x1, 0xF // -> x2=0x10 - Saved
sw x2, x31(0) // - by this Store
```
Compliance Test Quality Measurement

• Coverage is monitoring the execution, but is unaware of the propagation of the effect, and the observability of the program flow to the persistent storage in the SIGNATURE section

• How can we prove that a reportedly covered item, is observed within the SIGNATURE section?
Mutation Testing by Fault Injection

• Fault Injection to provide Mutation Testing
• For a given reported coverage point, mutate the instruction to have provided the contribution and test the propagation of that mutation to an observable entry in the SIGNATURE section

• Ref: https://www.geeksforgeeks.org/software-testing-mutation-testing/
Mutation Testing by Fault Injection

- For the given code snippet previously, mutate the operation to a legal alternate

```assembly
la x31, ADDR_SIGNATURE
// MUTATE ON:
<original-instruction>  addi x2, x1=0x1, 0x0
<mutated-instruction>   add x0, x0, x0
// MUTATE OFF
addi x2, x1=0x1, 0x1  // -> x2=0x2  - DISCARDED
addi x2, x1=0x1, 0x2  // -> x2=0x3  - DISCARDED
addi x2, x1=0x1, 0x4  // -> x2=0x5  - DISCARDED
addi x2, x1=0x1, 0x8  // -> x2=0x9  - DISCARDED
addi x2, x1=0x1, 0xF  // -> x2=0x10 - Saved
sw x2, x31(0)          // - by this Store
```
Mutation Testing by Fault Injection

• The mutated instruction could simply be
  – replace by a NOP : add, x0, x0, x0

• Or, override parts of the decode, for example the decode for an addi is as follows
  – ADDI : 12'b(Imm), 5'b(rs1), 3'b(000), 5'b(rd), 7'b(0010011)
  – The fixed parts of the decode are field [3](000) and field [5](0010011)
  – All other bits affect the immediate values and register indices for source and destination
  – In effect the decode appears thus
  – .... .... .... .... .... .... 000 .... .... 001 0011
  – all bits indicated as '.' could be mutated, in order to observe whether this incurs a propagated effect to our SIGNATURE section.
Mutation Testing by Fault Injection

- We need to be careful to ensure that introducing the mutated instructions does not cause a 'fault model explosion', in other words for the ADDI case above there are 22 bits which can be considered for mutation, but it would not be prudent to generate $2^{22}$ fault models.

- A pragmatic approach is to create the following fault models:
  - NOP Replacement
  - Bit inversion

- This would produce a total of 23 fault models for this one instruction - bearing in mind that the same approach is taken for every possible ADDI instruction.

- the encoding for
  - `addi x2, x1=0x1, 0x1`

- would thus be

<table>
<thead>
<tr>
<th></th>
<th>imm12</th>
<th>rs1</th>
<th>dec</th>
<th>rd</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>........</td>
<td>....0</td>
<td>....0</td>
<td>0010011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000000001</td>
<td>00001</td>
<td>0000010</td>
<td>0010011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mutation Testing by Fault Injection

• Potential Fault Models would be

<table>
<thead>
<tr>
<th></th>
<th>imm12</th>
<th>rs1</th>
<th>dec</th>
<th>rd</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000000001 00001</td>
<td>000</td>
<td>00010</td>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000000000001 00001</td>
<td>000</td>
<td>00010</td>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000000000001 00001</td>
<td>000</td>
<td>00010</td>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000000000001 00001</td>
<td>000</td>
<td>00010</td>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>000000000001 00001</td>
<td>000</td>
<td>00010</td>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>000000000001 00001</td>
<td>000</td>
<td>00010</td>
<td>0010011</td>
<td></td>
</tr>
<tr>
<td>etc</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mutation Testing by Fault Injection

- These fault models are inserted automatically by the Imperas Fault Simulator at runtime – without changing the elf/binary.
- Having run these fault models on an early version of the compliance suite, yielded some very interesting results, whereby the test engineer was utterly convinced that he had covered a particular scenario, but actually injecting the fault and following the propagation of the program flow - proved the result had been in someway masked.
- We found numerous reasons for this, including calculated values not stored in the SIGNATURE region, values in the SIGNATURE overwritten, or default initialization SIGNATURE values, having the same value as calculated values.
- initialized at -1 (0xFFFFFFFF) and the value -1 (0xFFFFFFFF) stored - hence no observable difference.
Mutation Testing by Fault Injection

- Running the Imperas Mutating Fault Simulator on the existing Compliance suite to grade the coverage
- Automating the Fault injection to assert the coverage metrics
- Imperas Simulator
  - injected $x$ faults in $y$ simulations and executed in $n$ seconds (Schrodinger's Cat)
- <Show Running>
Agenda

- RISC-V Intro and Status
- Industry requirements for ISA Compliance and Hardware Design Verification
- RISC-V ISA Compliance
- Processor Hardware Design Verification
- RISC-V Instruction Stream Generation
- Hardware Design Verification Flow
- Demonstration walk-through
- Scaling verification using Cloud resources
Processor Hardware Design Verification (agenda)

• Objective of HW DV for processors
  – Prove beyond (reasonable) doubt design meets objectives

• Available technologies and tools (Dynamic)
  – Reference model
  – Compliance suites, tools, tests
  – Directed test suites
  – Industry benchmarks and Operating Systems
  – Random Instruction Generators

• Formal models, properties libraries, model checkers, equivalence checkers

• ISS, RTL simulators, FPGAs, Emulators, ...
Processor Hardware Design Verification
Dynamic Testing approaches

• Prove beyond (reasonable) doubt design meets objectives, using
  – Directed tests
    • Architectural tests
      – Instruction correctness
    • Micro-architectural
      – Pipeline correctness, hazards, speculative execution, branch prediction, multiple issue
  • Compliance
  • Stress
  • Industry Benchmark, eg Coremark, Dhrystones, Linpack
  – Pseudo Random – Instruction Stream Generators
    • Architectural tests
    • Micro-architectural
    • Constraint solving & Coverage metrics

© Accellera Systems Initiative
Processor Hardware Design Verification

Targets for execution

• Instruction Set Simulation (ISS)
• Cycle Accurate Simulation (CAS)
• Virtual Prototype/Platform, System Emulator
• RTL Simulation
• FPGA
• Emulator (FPGA Prototyping)
• Accelerator (Hardware RTL Simulator)
• Combination of Above, eg ISS + Accelerator
• Physical Device
Agenda

• RISC-V Intro and Status
• Industry requirements for ISA Compliance and Hardware Design Verification
• RISC-V ISA Compliance
• Processor Hardware Design Verification
• **RISC-V Instruction Stream Generation**
• Hardware Design Verification Flow
• Demonstration walk-through
• Scaling verification using Cloud resources
RISC-V Instruction Stream Generation (agenda)

- Motivation
- Existing open source solutions
- Missing pieces
- Google Open Source RISC-V ISG
- Key features
- Generator internal flow
RISC-V Instruction Stream Generation

- Design Verification is the Foundation of Reuse
- Quality is key to reuse
- Stress testing needed to ensure IP will operate correctly when integrated into new design
- DV accounts for >50% of project time for complex chips
Open source RISC-V processor verification solutions

Verification is one of the key challenges of modern processor development.

**riscv-tests**

Assembly unit test

A simple test framework focused on sanity testing the basic functionality of each RISC-V instruction. It's a very good starting point to find basic implementation issues.

**riscv-torture**

Scala-based RISC-V assembly generator

Provides a good mix of hand-written sequences. Supports most RISC-V ISA extensions which makes it very attractive. Simple program structure and fixed privileged mode setting.
Many missing pieces

- Complex branch structure
- MMU stress testing
- Exception scenarios
- Compressed instruction support
- Full privileged mode operation verification
- Coverage model
- ...

**Motivation**

Build a high quality open DV infrastructure that can be adopted and enhanced by DV engineers to improve the verification quality of RISC-V processors.
Google RISC-V Instruction Stream Generation

• High quality SystemVerilog UVM DV infrastructure
• Open source
• Drives a RISC-V core through corner cases and pushes it to the limit

https://github.com/google/riscv-dv
Key Features

01 Randomness
Randomize everything: instruction, ordering, program structure, privileged mode setting, exceptions..

02 Architecture Aware
The generated program should be able to hit the corner cases of the processor architectural features.

03 Performance
The instruction generator should be scalable to generate a large program in a short period of time.

04 Extendability
Easy to add new instruction sequences, custom instruction extension, custom CSR etc.
Randomness

**Instruction level randomization**
Cover all possible operands and immediate values of each instruction
Example: Arithmetic overflow, divide by zero, long branch, exceptions etc.

**Sequence level randomization**
Maximize the possibility of instruction orders and dependencies

**Program level randomization**
Random privileged mode setting, page table organization, program calls
Instruction randomization

**Easy part**

**Arithmetic:** ADD, SUB, LUI, MUL, DIV ...
**Shift:** SLLI, SRL, SRLI, SRAI ...
**Logical:** XOR, OR, AND, ANDI ...
**Compare:** SLTI, SLT, SLTU ...
**Others:** FENCE, SFENCE, EBREAK ...

Randomize each instruction individually with bias towards corner cases.
(overflow, underflow, compressed instruction)

**Tricky part**

- **Branch / jump instruction**
  Need a valid branch/jump target
  Avoid infinite loop

- **Load/store/jump instruction**
  Need an additional instruction to setup the base address
  The calculated address should be a valid location

- **CSR instruction**
  Avoid randomly changing the privileged state
  Result checking could be a challenge as the privileged CSR behavior could be implementation-specific.

Google Cloud
Architecture Aware

01 Branch prediction

02 MMU (TLB, Cache etc)

Instruction

BHT

Address
TAG
Prediction
bits

Jump

Jump

Trap

M
Code segment 0

U
Code segment n

U
Code segment k

S
Trap handler

Load

Store

Data page 0
Data page 1
Data page 2
Data page 3
Data page 4
Data page 5
Data page 6
Data page 7
Data page 8
Data page 9

Verification is one of the key challenges of modern processor development.

Open source RISC-V processor verification solutions

riscv-tests
Assembly unit test
A simple test framework focused on sanity testing the basic functionality of each RISC-V instruction. It’s a very good starting point to find basic implementation issues.

riscv-torture
Scala-based RISC-V assembly generator
Provides a good mix of hand-written sequences. Supports most RISC-V ISA extensions which makes it very attractive. Simple program structure and fixed privileged mode setting.
Architecture Aware

03 Issue, execute, commit

It’s not just a random stream of instructions, it should be designed to effectively verify the architectural features of the processor.
Open source RISC-V processor verification solutions

**riscv-tests**
Assembly unit test
A simple test framework focused on sanity testing the basic functionality of each RISC-V instruction. It’s a very good starting point to find basic implementation issues.

**riscv-torture**
Scala-based RISC-V assembly generator
Provides a good mix of hand-written sequences. Supports most RISC-V ISA extensions which makes it very attractive. Simple program structure and fixed privileged mode setting.

Verification is one of the key challenges of modern processor development.

---

**Generator flow**

1. Generate program header
2. Privileged mode setup
3. Page table randomization
4. Initialization routine
5. Generate main/sub programs
6. Branch target assignment
7. Generate data/stack section
8. Generate page tables
9. Generate intr/trap handler
10. Test completion section
11. Call stack randomization
12. Apply directed instructions

---

© Accellera Systems Initiative
Complete feature list

- **Supported ISA**
  RV32IMC, RV64IMC

- **Supported privileged mode**
  User mode, supervisor mode, machine mode

- **Supported spec version**
  User level spec 2.20, privileged mode spec 1.10

- **Supported RTL simulator**
  VCS, Incisive, Metrics

---

**Test suite**

- Basic arithmetic instruction test
- Random instruction test
- MMU stress test
- HW/SW interrupt test
- Page table exception test
- Branch/jump instruction stress test
- Interrupt/trap delegation test
- Privileged CSR test

---

Google Cloud

© Accellera Systems Initiative 2019
Agenda

• RISC-V Intro and Status
• Industry requirements for ISA Compliance and Hardware Design Verification
• RISC-V ISA Compliance
• Processor Hardware Design Verification
• RISC-V Instruction Stream Generation
• Hardware Design Verification Flow
• Demonstration walk-through
• Scaling verification using Cloud resources
Hardware Design Verification Flow (agenda)

• Flow
• Components
  – Google: open source riscv-dv instruction stream generator
  – Imperas: model and simulation golden reference of RISC-V CPU
  – Metrics: SystemVerilog design + UVM simulator for RTL
• Using
• Google: open source riscv-dv instruction stream generator
• Metrics: SystemVerilog design + UVM simulator for RTL
• Imperas: model and simulation golden reference of RISC-V CPU
Google SystemVerilog UVM
Instruction Stream Generator

Keys features:

– Randomness
  • Randomize everything: instruction, ordering, program structure, privileged mode setting, exceptions..

– Architecture-aware
  • Generated program able to hit the corner cases of the processor architectural features

– Performance
  • Instruction generator is scalable to generate a large program in a short period of time

– Extendibility
  • Easy to add new instruction sequences, custom instruction extension, custom CSR etc.

https://github.com/google/riscv-dv
Imperas RISC-V Instruction Set Simulator and full RISC-V Specification Envelope Model

- Industrial quality model and simulator of RISC-V processors for use in compliance, verification and test development
- Complete, fully functional, configurable simulator
  - All 32bit and 64bit features of ratified User and Privilege RISC-V specs
    - Vector extension, version 0.7.1
    - Bit Manipulation extension, version 0.9.0
  - Model source included under Apache 2.0 open source license
- Used as golden reference in RISC-V Compliance Suite and Bit Manipulation group
- Extendibility: easy for user to extend with new instructions and functionality

http://www.imperas.com/riscv
https://github.com/riscv/riscv-ovpsim

© Accellera Systems Initiative
Metrics (1): SystemVerilog Simulator

- Complete SystemVerilog IEEE 1800-2012 compliant simulator including UVM
- Includes all the standard features of a modern SystemVerilog simulator including debug, APIs, language and testbench support
- Simulates the testbench, the RTL design, and the populates the coverage models

https://metrics.ca/

© Accellera Systems Initiative
Metrics (2): Full Cloud Platform

- True cloud platform for ASIC and complex FPGA design verification
- On-demand simulation resources
- Modern continuous integration workflow
- Easy access to simulation results from any web browser

=> So the full verification can be done in the Google Cloud using the Metrics Cloud Platform, the Metrics simulator, the Imperas RISC-V reference simulator and the Google Instruction Stream Generator
Agenda

• RISC-V Intro and Status
• Industry requirements for ISA Compliance and Hardware Design Verification
• RISC-V ISA Compliance
• Processor Hardware Design Verification
• RISC-V Instruction Stream Generation
• Hardware Design Verification Flow
• Demonstration walk-through
• Scaling verification using Cloud resources
Demonstration

• Uses RISC-V Core: lowRISC/ibex RTL (RV32IMC)
  – Originated from zero-RI5CY ETH Zurich PULP
DV Flow is controlled by Makefile and bash scripts and includes python scripts

- Compile up SystemVerilog UVM test generator and run it
  - can easily set how many tests to create each run
  - Creates .S files that are then converted to .o
- Run the Imperas ISS to generate reference results
- Compile the SystemVerilog RTL of ibex core and testbench
- Run RTL simulation & record RTL results
- Post-processor run logs and compare
Demo: Configuring flow

- Test generator settings
  - Device Under Test / Processor (ISA, xlen, extensions, modes, mmu, ...)
    - Show settings .sv file
  - Complexity & quantity of tests
    - Show Command line options, testlist.yaml

- Reference ISS riscvOVPsim config
  - Show yaml file for Ibex and config.ic file
Demo show test gen, iss, rtl, compare run

- Show running under Metrics cloud connection
  - make gen
  - make gcc_compile
  - make iss_sim
  - make iss_cmp
And results are simple pass, or detailed fail

• Example of detailed fail:
  • Shows mis-matching instructions
    • Configured here to show 5
  • Full traces etc are kept for review
  • Can dump full VCD for detailed waveform analysis
And... what about functional coverage you may ask...

• Sneak preview...
• New Open Source SystemVerilog UVM Coverage component added to flow
  – Adds coverage groups / bins in UVM testbench
  – Configure coverpoints based on DUT, eg RV32IMC, RV32IMAFDC, ...
• Post processes a test’s execution trace
• Collates coverage from many runs
• Uses SystemVerilog UVM simulator’s coverage analysis to report, review
  – Works with Cadence, Mentor, Synopsys, Metrics
Mentor Questa Coverage views
Agenda

- RISC-V Intro and Status
- Industry requirements for ISA Compliance and Hardware Design Verification
- RISC-V ISA Compliance
- Processor Hardware Design Verification
- RISC-V Instruction Stream Generation
- Hardware Design Verification Flow
- Demonstration walk-through
- Scaling verification using Cloud resources
Metrics Cloud Platform makes it all much simpler...

- Complete solution for DV
Metrics: integrated GitLab

- Git is the way to get data into and out of the Metrics Cloud Platform
Metrics: built in Continuous Integration to run jobs

• Push of a button (‘Run Pipeline’) to run jobs (that we ran from shells earlier)
Metrics: browse access to all results

- Showing test runs and summary coverage
Metrics: can drill down into details

- For example see the saved logs – so can see all detail of run
Metrics: can show functional coverage

- Uses SystemVerilog covergroups etc.
Metrics: can even see detailed contribution of each test including functional coverage
Metrics: includes top level overview dashboard

- Allows management overview of status of verification
Status

• The integrated DV infrastructure from Google, Metrics, Imperas is a work in progress
• Clearly shows direction and focus
Summary

• DV solution for RISC-V RTL cores

• Collaboration between Google, Imperas and Metrics

• Industry adoption has started

• The basis for RISC-V core verification in:
Questions