Open-Source Virtual Platforms for Industry and Research

Nils Bosbach, RWTH Aachen University
Lukas Jünger, MachineWare GmbH
Rainer Leupers, RWTH Aachen University
About Us

**Nils Bosbach**
- PhD student at *Chair for Software for Systems on Silicon (SSS), RWTH Aachen University*

**Lukas Jünger**
- Co-founder of *MachineWare GmbH*
- Chair of the *SystemC Configuration, Control and Inspection (CCI) Working Group*
What to Expect?

Open-Source Virtual Platforms for Industry and Research
What to Expect?

“A virtual platform is a functional representation of a digital system written entirely in software.”
- Cadence
What to Expect?

Open-Source Virtual Platforms for Industry and Research

- Prototyping
- Develop & test models
- Embedded SW development
- Prototyping of new architectures
- Architecture exploration
- Improve simulation performance
What to Expect?

Many commercial solutions available

Solution we will talk about today:

Available on GitHub

- Clone
- Play around
- Extend
Free and/or Open-Source Software

• Disclaimer: I am not a lawyer!
• "Free of charge" or "free as in freedom"?
• Licenses

<table>
<thead>
<tr>
<th></th>
<th>Apache 2.0</th>
<th>GPLv2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closed-source</td>
<td>Allowed</td>
<td>Not-allowed</td>
</tr>
<tr>
<td>Commercialization</td>
<td>Easy (no submarine patents)</td>
<td>Difficult</td>
</tr>
<tr>
<td>IP Protection</td>
<td>Easy (derived works under another license)</td>
<td>Difficult</td>
</tr>
</tbody>
</table>

• Are Apache 2.0 and GPLv2 compatible? Maybe
Virtual Platforms

A brief introduction
Why Do We Need Virtual Platforms?

1. Which device should be designed?
2. Which components are needed?

- **HW Engineers**
  3. Develop chip
  4. Tape out chip
  5. Test & verify embedded SW on the HW chip

- **SW Engineers**
  3. Develop Virtual Platform
  4. Develop, test & verify embedded SW
Why Should You Use Virtual Platforms?

- Requirements
- Component selection
- HW development
- SW development

Development time
Why Should You Use Virtual Platforms?

- **Development time**
  - Requirements
  - Component selection
  - HW development
  - SW development
  - "shift left"

**Scalability**
- Easy distribution
- Integrates with CI

**Introspection**
- Stop at any time
- Access all register values

**Tracing**
- Record tracing data
- Analyze SW behavior
Abstraction Level

- Many different abstraction levels
- Tradeoff between performance and accuracy

Choose the level that fits your needs
Virtual Platform Design

Let’s get started!
How to Start?

**Specification**
- List of components
- Interface description
- Functional behavior

**Virtual Platform**
- Simulates the behavior
- Can execute target SW

**VCML**
- VCML-based models to mimic the behavior of the SoC
- Virtual Components Modeling Library – adds modeling primitives
- Standardized “design and verification language”
  (IEEE Std. 1666™-2011)
Why Do We Need a Modeling Library?

- Concept of time
- Simulated parallelism
- Standardized interfaces (ports, TLM sockets)
- Hierarchy (modules)
- Models (e.g., buses)
- Frequently-needed parts (e.g., register model)
- Often-used communication-protocol implementations (SPI, CAN, I²C, …)
- TLM logging/tracing, (parametrization, configuration)
Virtual Components Modeling Library (VCML)

• Loosely-timed simulation framework based on SystemC TLM-2.0
• Apache-2.0 license
• Windows, Linux, MacOS
  • x86, arm64 CI builds
• Provides
  • Commonly used features (registers, peripherals, etc.)
  • Abstract protocols based on TLM-2.0 (interrupt, SPI, I²C, etc.)
  • Models (memory, memory-mapped buses, UARTs, etc.)
• Extensive unit test suite

https://github.com/machineware-gmbh/vcml
Why Should You Use VCML?

• Completely standard SystemC TLM-2.0 compatible
• Saves your time when construction a new VP
  • Reuse VCML features and models
• Supports all major Systems: Windows, Linux, MacOS, x86 and arm64
• Easy commercialization through Apache 2.0 license
• Example VPs available open-source (ARM and OpenRISC)
• Commercial support available through MachineWare
VCML-Highlights

- Modeling primitives
- Tracing
- Debugging support
- Backends
- Models
- Configurability
- Session Protocol
- Scripting
Implementation of the CPU Model

- Basic building block of SystemC
- Basic building block of VCML
- Adds support for VCML features (sockets, properties, commands, session)
- Basic block for hardware models
- Provides reset and clock sockets
- Starting point for processor implementation
- Implements debugging & temporal decoupling

Interrupt target socket to receive interrupts

TLM initiator socket to access memory
Target Software Debugging

- Full debugging support
  - Including OS
- GDB/Lauterbach’s Trace32
Lauterbach’s Trace32

- Multicore debugging
- OS awareness
- User-space debugging

Watch our Trace32 demo:

https://youtu.be/B6zys6_M-k4
SIM-V and Lauterbach Trace32
Like real hardware, just better
How to Implement Peripherals?

- Basic building block of SystemC
- Basic building block of VCML
- Adds support for VCML features (sockets, properties, commands, ...)
- Basic block for hardware models
- Provides reset and clock sockets
- Starting point for custom I/O peripheral models
- Adds support for registers
Peripheral Interfaces – TLM Protocols

Interfaces
- Memory accesses
- Interrupt
- Communication/data transfer (e.g., SPI, I²C, ...)

Based on TLM blocking transports

Payload
- Operation (R/W)
- Data pointer
- Attributes
Peripheral Interfaces – TLM Protocols

**Interfaces**

- Memory accesses
- Interrupt
- Communication/data transfer (e.g., SPI, I²C, ...)

**Based on TLM blocking transports**

- Process payload
  - Execute operation (R/W)

---

**Diagram:**

- Peripheral
- Initiator
- Target
- Process payload
  - Execute operation (R/W)
Peripheral Interfaces – TLM Protocols

**Interfaces**
- Memory accesses
- Interrupt
- Communication/data transfer (e.g., SPI, I²C, ...)

**Based on TLM blocking transports**
- Send payload back
  - Analyze returned payload

---

Peripheral Interfaces

- Initiator
- Target
Peripheral Interfaces – TLM Protocols

**Interfaces**

- Memory accesses
- Interrupt
- Communication/data transfer (e.g., SPI, I²C, ...)

**Based on TLM blocking transports**

**Implemented protocols**

- CAN
- Clock
- Ethernet
- GPIO
- I²C
- PCI
- SD
- Serial
- SPI
- VirtIO
Predefined Peripherals in VCML

- Memory
- Interrupt controllers (ARM, RISC-V)
- Ethernet, SPI, $I^2C$ controller
- SD controller
- UARTs (e.g., PL011)
- Sensors (SPI/$I^2C$-based)
- VirtIO (controller, console, RNG, block, ...)
- CAN (bridge & bus)
Configurability

- VCML provides properties to configure modules
  - CCI-compatible
- LUA scripting
- Session protocol
Session

Virtual Platform Explorer (ViPER)

- Control the simulation
- Access/inspect models
- Open, TCP-based protocol
Session

- Python-based implementation of a Session client
- Use Python to script the simulation
Session

Virtual Platform Explorer (ViPER)

- Infinite number of use cases
- Very powerful interface
- Simple usage

PyVP

VP

VCML Session Protocol

?
Free Implementations

• OpenRISC 1000 Multicore Virtual Platform (OR1KMVP)
  • Embeds the open-source OR1KISS into vcml::processor
    • Simple interpreter-based instruction-set simulator (ISS)
  • https://github.com/janweinstock/or1kmvp

• An ARMv8 Virtual Platform (AVP64)
  • Embeds the ARMv8 unicorn implementation (QEMU-based) into vcml::processor
    • Dynamic-binary-translation (DBT)-based ISS
  • https://github.com/aut0/avp64
Commercial Implementations

• SIM-V
  • Ultra-fast RISC-V simulator
  • Supports state-of-the-art extensions
  • Custom-instruction API

• SIM-A
  • Ultra-fast ARM Cortex-M simulator

• QBox
  • QEMU in SystemC TLM-2.0 through VCML
Full-System Integration

- Instantiate models
- Bind sockets
- Start simulation
SW Development

Demo - Find the tutorial on the AVP64 GitHub page

https://github.com/aut0/avp64/tree/master/vscode
Summary

From spec to chip
Summary

VP Benefits
- Development Time
- Introspection
- Scalability
- Tracing

SW Stack
- (Custom) Models
- VCML

Slides: https://mwa.re/vcml-tutorial