RISC-V based Multi-Core Virtual Prototype: An Extensible and Configurable Platform for Modeling and Verification

Vladimir Herdt[1]
Daniel Große[1,2]
Rolf Drechsler[1,2]

[1]University of Bremen, Germany
[2]DFKI Bremen, Germany
vherdt@informatik.uni-bremen.de

16ES0565

edaclusterforschung

graduate school, funded by
German Excellence Initiative

RISC-V (1)

- Completely open ISA that is freely available
- No license costs involved
- Efficient and versatile design
- High-performance to small embedded devices
- Widely adopted

RISC-V (2)

- Mandatory Integer Instruction Set "I" (~47 instrs.)
  - 32/64/128 Bit

- + Optional Extensions
  - "M", "A", "F", "D", etc.

RV32IMAFD = RV32G

- Control and Status Registers (CSRs), Environment Interaction and Privilege Levels (M=Machine, S=Supervisor, U=User)

VP Overview (1)

- RV32IMAFC+SUN, RV64GC+SUN
- Implemented in SystemC/C++
  - TLM-2.0 compliant
  - approx. 12k LOC (w/o comments, blanks)
- **Open Source via GitHub**
  - http://www.systemc-verification.org/riscv-vp
  - MIT license
  - FDL 2018 overview paper:
    "Extensible and Configurable RISC-V based Virtual Prototype"
    http://www.informatik.uni-bremen.de/agra/doc/konf/2018FDL_RISCV_VP.pdf

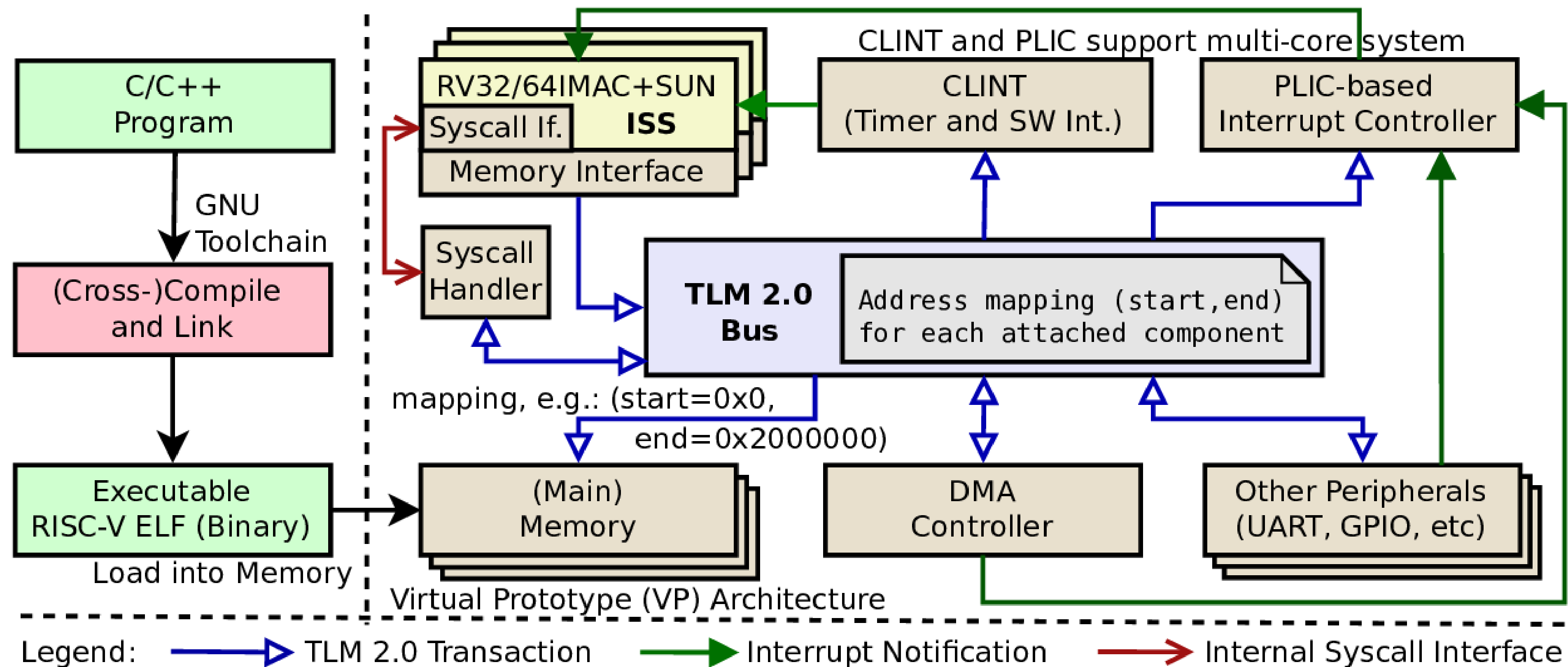
VP Overview (2)

- **SW debug** capabilities (GDB RSP interface) with **Eclipse IDE**
- **SW coverage** measurement (GCOV)
- **FreeRTOS** and **Zephyr OS** support
- **CLINT** and **PLIC**-based interrupt controller + e.g. display, flash controller, preliminary Ethernet
- 64-bit VP boots **Linux**

VP Overview (3)

- Instruction-based **timing model** + annotated TLM 2.0 transaction delays
- Example configuration for the **SiFive HiFive1** board
- Support for simulation of **multi-core** platforms
- Supervisor- and user-mode **privilege levels**

VP Setup and Architecture



Performance Optimization

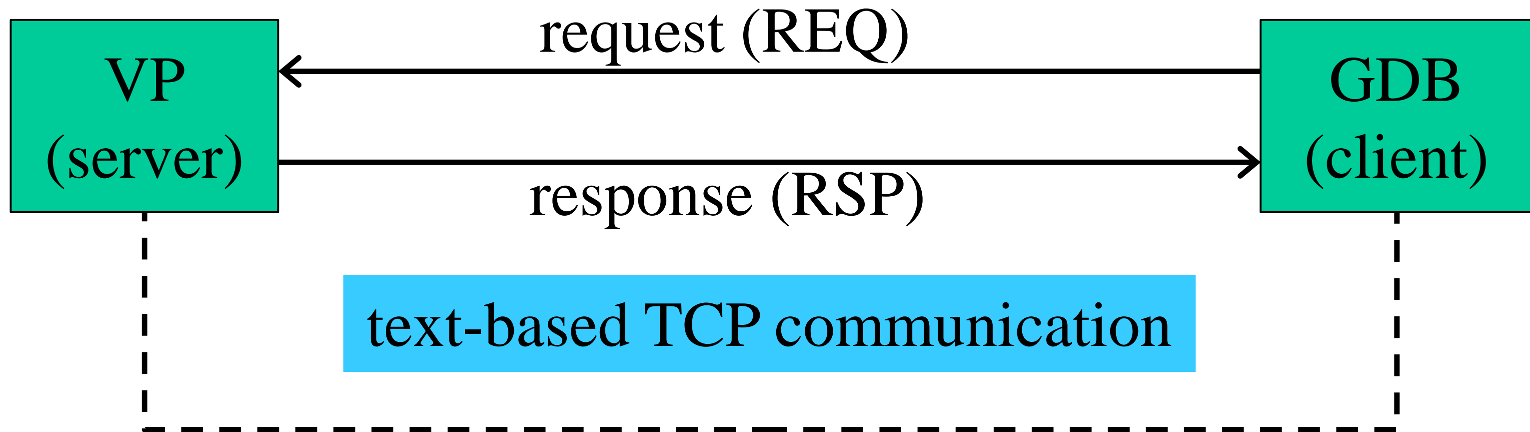
- DMI for main memory access
  (core concept: char* access to memory)
  - DMI for instruction fetching
  - DMI for data access
- Temporal decoupling in CPU core
  - Evaluate different local time quantums
- Experiments:
  - VP ~40-50 million instructions per second (Intel Xeon Gold 5122, 3.6GHz)
  - ~1000x faster than RTL simulation

Timing Model

- Simple Instruction-based
  - Fixed execution times for each instruction
  - Easy to configure
- TLM blocking transactions
  - b_transport(*tlm::generic_payload* &payload, *sc_core::sc_time* &delay)
  - Peripherals increment the delay parameter
- More precise models can be integrated

GDB Integration

RSP Interface



main.c

```
1: int main() {
2:     int a = 4;
3:     a++;
4:     return a;
5: }
```

REQ: $m111c4,4#f7
RSP: +$05000000#85

```
> file main.elf
> target remote :5005
> b main.c:4
> c
> print(a)
```

FreeRTOS + Eclipse GDB

Debug

- fdl-18-sensor Default [C/C++ Remote Application]
  - main
    - Thread #1 <main> (Suspended : Step)
      - dump_sensor_data() at main.c:22 0x100d4
      - main() at main.c:33 0x1016c
  - riscv32-unknown-elf-gdb (8.0.50.20170808)

Variables | Breakpoints | Registers | Modules

Name	Type	Value
i	int	6

main.c | irq.c | bootstrap.S

```
 9 _Bool has_sensor_data = 0;
10
11 void sensor_irq_handler() {
12     has_sensor_data = 1;
13 }
14
15 void dump_sensor_data() {
16     while (!has_sensor_data) {
17         asm volatile ("wfi");
18     }
19     has_sensor_data = 0;
20
21     for (int i=0; i<64; ++i) {
22         *TERMINAL_ADDR = *(SENSOR_INPUT_ADDR + i);
23     }
24 }
25
26 int main() {
27     register_interrupt_handler(2, sensor_irq_handler);
28
```

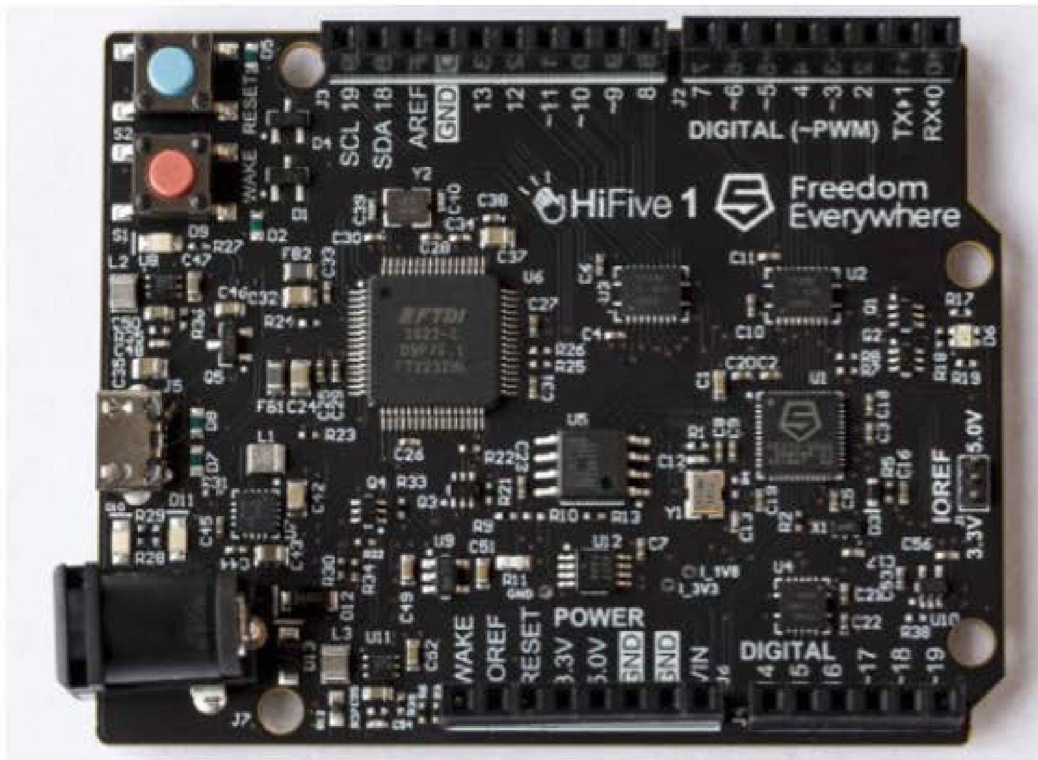
Outline | Disassembly

Enter location here

```
21            for (int i=0; i<64; ++i) {
000100cc:   sw      zero,-20(s0)
000100d0:   j       0x100fc <dump_sensor_data+96>
22              *TERMINAL_ADDR = *(SENSOR_INPUT_ADDR + i);
000100d4:   lui     a4,0x50000
000100d8:   lw      a5,-20(s0)
000100dc:   add     a4,a4,a5
000100e0:   lui     a5,0x20000
000100e4:   lbu     a4,0(a4) # 0x50000000
000100e8:   andi    a4,a4,255
000100ec:   sb      a4,0(a5) # 0x20000000
21            for (int i=0; i<64; ++i) {
000100f0:   lw      a5,-20(s0)
000100f4:   addi    a5,a5,1
000100f8:   sw      a5,-20(s0)
000100fc:   lw      a4,-20(s0)
00010100:   li      a5,63
```

https://github.com/agra-uni-bremen/riscv-freertos

SiFive HiFive 1



- Open-Source RTL
- Arduino-Compatible
- Freedom E SDK
- Arduino IDE Environment

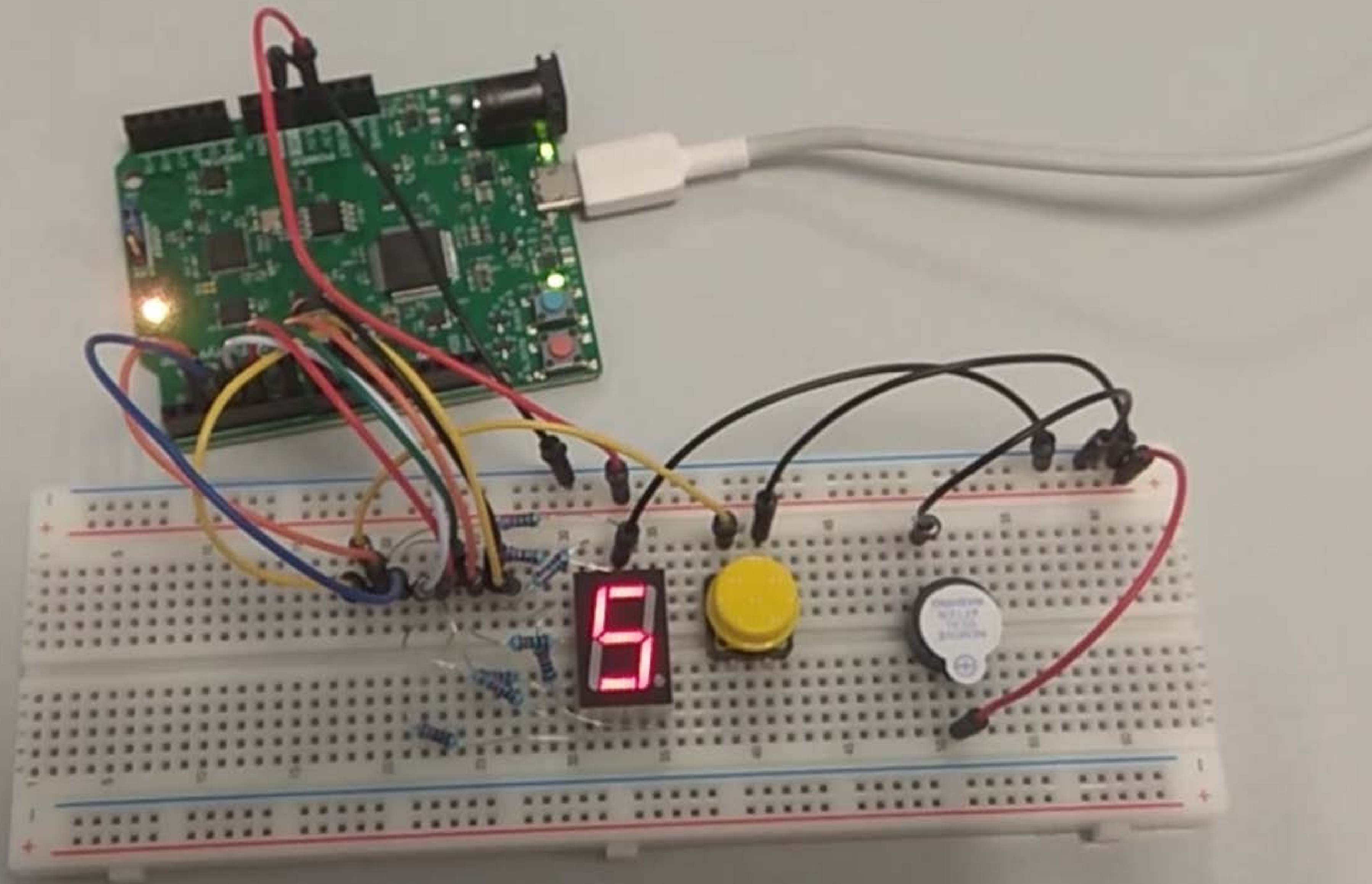
https://www.crowdsupply.com/sifive/hifive1

QT-based virtual env.

vp-breadboard

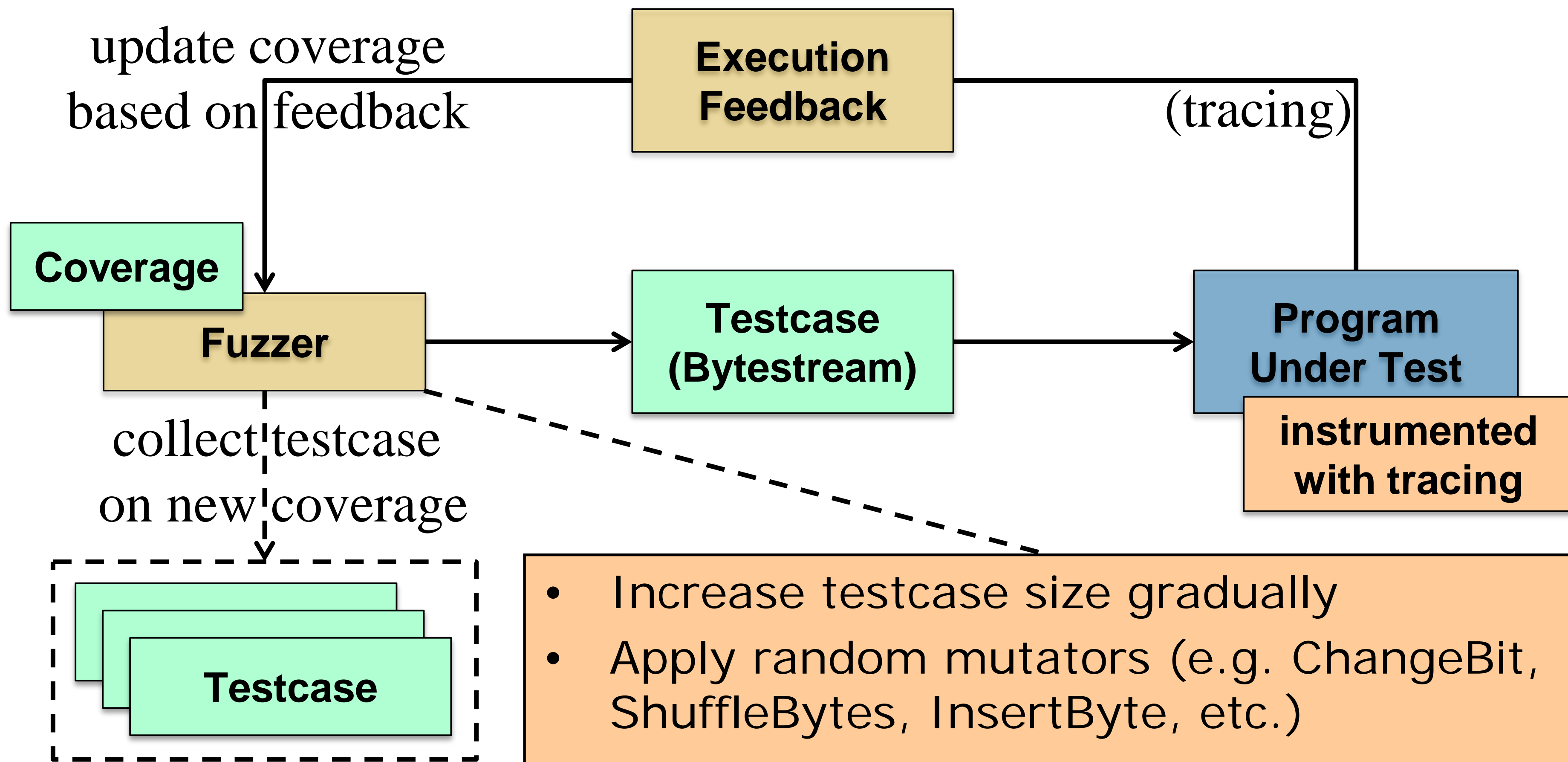
```
196     //clear irq - interrupt pending register is write 1 to clear
197     GPIO_REG(GPIO_FALL_IP) |= (1 << mapPinToReg(10));
198      puts("button press handled\r\n");
199 }
200
201 /*configures Button0 as a global gpio irq*/
202 void b0_irq_init()  {
203
204     //disable hw io function
205     GPIO_REG(GPIO_IOF_EN )    &=  ~(1 << mapPinToReg(10));
206
207     //set to input
208     GPIO_REG(GPIO_INPUT_EN)   |= (1 << mapPinToReg(10));
209     GPIO_REG(GPIO_PULLUP_EN)  |= (1 << mapPinToReg(10));
210
211     //set to interrupt on falling edge
212     GPIO_REG(GPIO_FALL_IE)    |= (1 << mapPinToReg(10));
213
214     PLIC_init(&g_plic,
```

```
   5555                                     5555
  5555          yeeeeeeeeeeee                5555
 5555                                         5555
55555           9999999999999                55555
 55555            555555555                 55555
   55555            55555                 55555
     55555            5                 55555
       55555                          55555
         55555                      55555
           55555                  55555
             55555              55555
               55555   55555
                 555555555
                   55555
                     5

               'sevensegment' Demo

Config String:
/cs-v1/;/{model = "SiFive,FE310G-0000-Z0";compatible = "sif

5555555555555555555555555555555555555555555555
555555 Are able to press Buttons? [y/n]  555555
5555555555555555555555555555555555555555555555
Inited button
Number: 0
server: got connection from 127.0.0.1
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Number: 0
```



Same RISC-V ELF-binary on real HiFive1 board

Coverage-guided Fuzzing (CGF)



LLVM libFuzzer: llvm/projects/compiler-rt/lib/fuzzer/

CGF for ISS Verification

```
void __sanitizer_cov_trace_cmp8(uint64_t Arg1, uint64_t Arg2) {
  // ... update coverage accordingly ...
}
```

branch coverage instrumentation added by clang to ISS

based on RISC-V Torture

ELF Template (Exec. Frame)

libFuzzer (CGF)

+ custom mutators

ELF Testcase

ISS Under Test

+ tracing of func. cov.

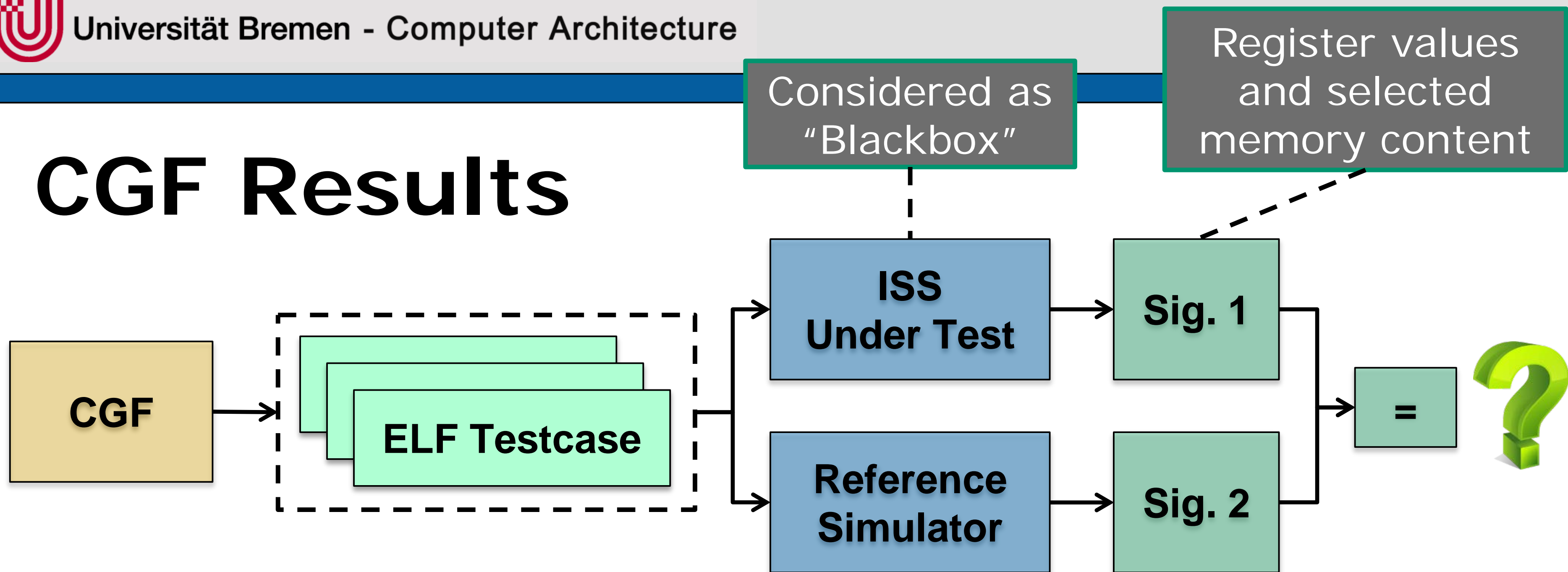
Testcase (Bytestream)

interpreted as instructions

```
extern "C" int LLVMFuzzerTestOneInput(
    const uint8_t *Data, size_t Size) {
  // ... process the input ...
  return 0;
}
```

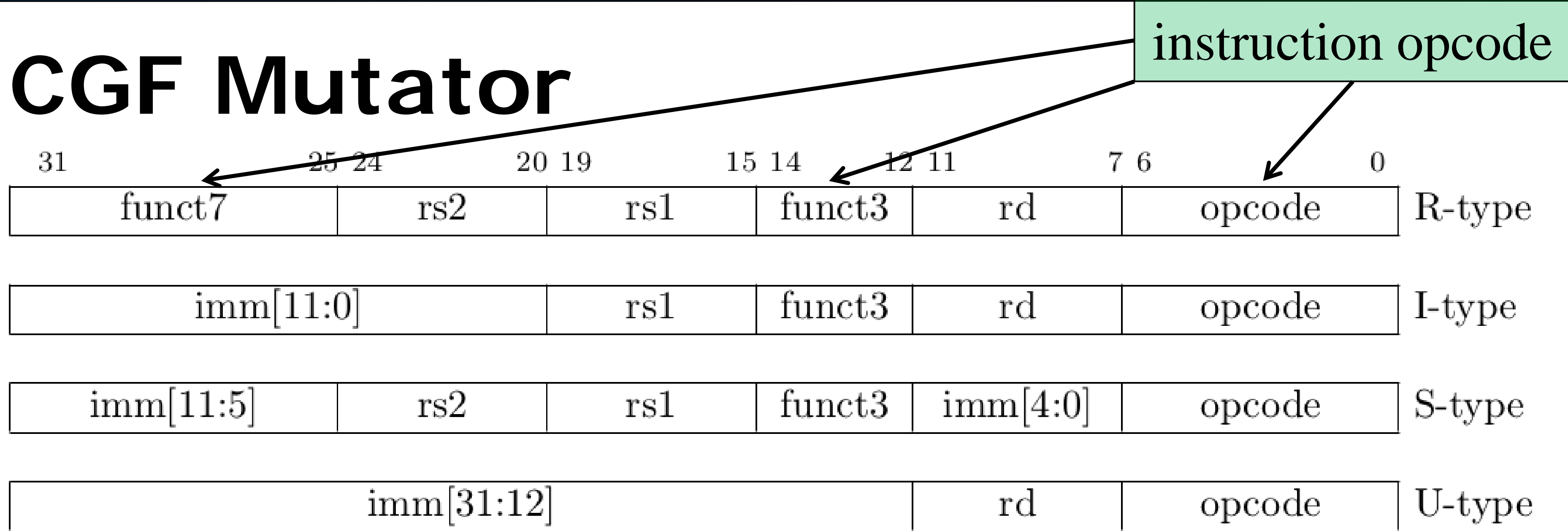
"in-process" fuzzing

CGF Results



- *CGF strength: corner- and error-cases*
- *"Spurious"* mismatch possible:
  - Different memory sizes
  - Devices mapped into the address space
  - CSRs and privilege levels
- Debug script to find the first mismatch
  - Start with 1 instruction and increase step-by-step

CGF Mutator



RISC-V instruction format, source: official RISC-V ISA spec.

- Fuzzing completely random, without domain specific knowledge
- Good idea/intelligent?
  - -> Explain the fuzzer what is the instruction format
  - -> Generate longer sequences of valid instructions

Selected with same probability as existing mutators

CGF Custom Mutator Extension

1. Inject random instruction opcode
   - Register and immediate values stay random
2. Inject random instruction sequence
   - e.g. load large immediate: two subsequent instructions with same RD but random value

---

fuzzer generated bytestream (interpreted as instruction list):

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...

↑inject position

1. *inject ADDI opcode*

xxxxxxxxxxxxxxxxxxx000xxxxx0010011xxxxxxxxxxxxxxxxxxxxx...

imm[11:0] RS1 RD ↑inject position

CGF Coverage Metric

- Code (Branch) Coverage
  - covers switch-case based on opcode in ISS
  - tracing instrumentation performed by clang
- Enough for ISS verification?

31	20	19	15	14	12	11	7	6	0
I_imm[11:0]		rs1		000		rd		op=0010011	

ADDI (*Add Immediate*): R[rd] = R[rs1] + I_imm

-> stronger coverage metric tailored for ISS verification is useful

CGF Coverage Metric Extension

Fuzzer extension required: map functional coverage to unique feature set

- Functional Coverage
  - **R1:** RD=0, RD≠0
  - **R2:** RD=RS1, RD≠RS1
  - **R3:** similar to R2 but for three registers
  - **vRx:** register *x* (RD, RS1, RS2) value within range {REG_MIN, -1, 0, 1, REG_MAX}
  - **vImm:** immediate value within range {IMM_MIN, -1, 0, 1, IMM_MAX}
- Trace instrumentation
  - Add trace function before/after instruction execution in ISS
  - Provide: register values and instruction, incl. opcode

VP (ISS) Testing: Experiments

1. Coverage-guided Fuzzing (CGF)

DATE 2019 paper:
"Verifying Instruction Set Simulators using Coverage-guided Fuzzing"
http://www.informatik.uni-bremen.de/agra/doc/konf/2019DATE_Verifying_Instruction_Set_Simulators_using_Coverage-guided_Fuzzing.pdf

2. (Official) RISC-V ISA Tests

3. RISC-V Torture test generator

4. Example Applications:

– bare-metal, C/C++ library, FreeRTOS and Zephyr OS, Linux boot

VP (ISS) Testing: RISC-V Tests

- RISC-V ISA tests from Berkeley: passed (57 tests, RV32IMA)

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

https://github.com/ucb-bar/riscv-torture

Register values and selected memory content

Random Test Gen. (Torture)

Testcase (RISC-V Program)

Our VP

Sig.

Reference (Spike + others)

Sig.

=

RV32IMA: 10000 tests (~14h runtime)

CGF Results

Coverage measured by our ISS

Test Generation	Time	Branch Cov.	Errors Found		
			Our ISS	Spike	Forvis
ISA Tests (RV32IMA)	2s	90%	V1,V2,V3	/	/
RISC-V Torture 1000	1.4**h**	74%	V1,V2	/	H2
RISC-V Torture 10000	14.4**h**	74%	V1,V2	/	H2
Coverage-guided Fuzz.	9**h**	100%	[V1..V7]	S1	H1,H2

Test Gen.	Functional Coverage							
	R1	R2	R3	vRS1	vRS2	vRD	vImm	vShm
ISA	58%	61%	50%	14%	2%	7%	8%	100%
T.1000	2%	66%	69%	58%	91%	52%	9%	100%
T.10000	2%	66%	69%	58%	91%	56%	63%	100%
CGF	100%	100%	100%	98%	100%	81%	100%	100%

Error Description

- Spike:
  - Decoder error interpreting illegal instruction as **fence** / **fence.i**
- Forvis:
  - **remu** fails because it performs a 64 bit operation in 32 bit mode
  - Illegal counter CSR access
- Our ISS:
  - Incomplete decoder checks
  - Illegal instruction has side effects
  - Wrong CSR update in case of (RD=RS1)

Conclusions

- Configurable, Extensible and Verified RISC-V based Virtual Prototype
- Future work:
  - (Formal) Verification
    - RISC-V VP Model:
      Symbolic simulation using SISSI [1]
      UVM / CRAVE [2]
    - SW running on RISC-V VP:
      Symbolic execution to check user assertions
      To appear at DAC 2019:
      "Early Concolic Testing of Embedded Binaries with Virtual Prototypes: A RISC-V Case Study"
  - Enhanced performance and power estimation

[1] Verifying SystemC using Intermediate Verification Language and Stateful Symbolic Simulation (TCAD 2018)
[2] CRAVE: An Advanced Constrained RAndom Verification Environment for SystemC (SoC 2012)

RISC-V based Multi-Core Virtual Prototype: An Extensible and Configurable Platform for Modeling and Verification

http://www.systemc-verification.org/riscv-vp

Vladimir Herdt[1]
Daniel Große[1,2]
Rolf Drechsler[1,2]

[1]University of Bremen & [2]DFKI Bremen, Germany
vherdt@informatik.uni-bremen.de

16ES0565

edaclusterforschung

graduate school, funded by
German Excellence Initiative