December 3-6, 2018
Santa Clara Convention Center,
Santa Clara, CA

RISC-V Summit

Machine-Readable Specifications *of* RISC-V ISA

M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, A. Tatarnikov

ISP RAS Ivannikov Institute for System Programming
Russian Academy of Sciences

RISC-V Formalization Motivation

- **High-level synthesis (HLS)**
  - High-level HDL (Chisel, BSV, SystemC) $\to$ RTL (Verilog, VHDL)
- **Cross development tools and virtual platforms**
  - ADL (CodAL, nML) $\to$ simulator + compiler (Codasip, Imperas)
- **Functional and security verification**
  - ADL + scenarios $\to$ assembly (MicroTESK, Genesys-Pro, RAVEN)
  - HLS (Clash) + equivalence checking (Yosys, ABC, JasperGold)
  - Theorem provers and proof assistants (Isabelle/HOL, Coq, PVS)
  - ADL + SW code + properties (MicroTESK + Why3 + SMT)

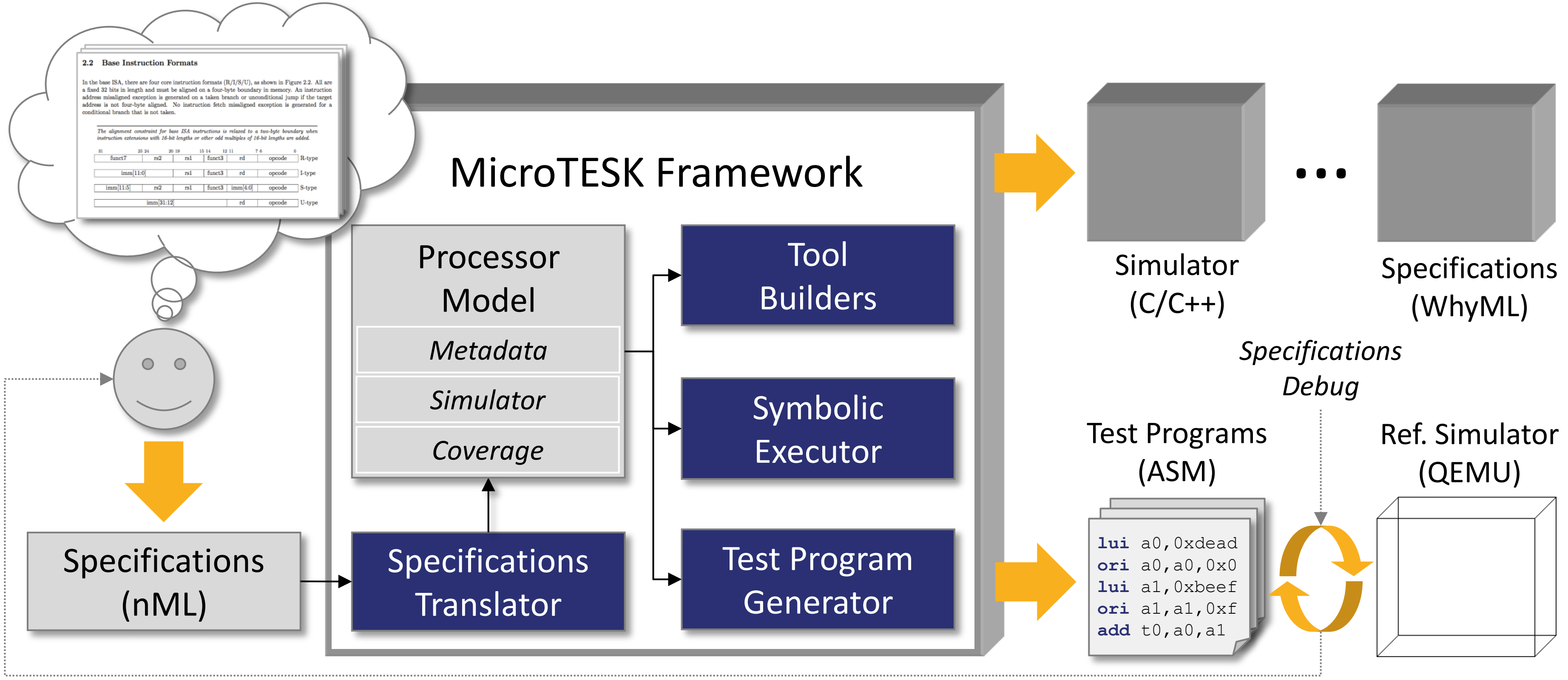
RISC-V Formalization Activities

- **Technical activities**
  - SW models: C/C++ (Spike, QEMU, RV8, gem5, Imperas, etc.)
  - ISA-centric ADL: nML (ISP RAS), CodAL (Codasip)
  - High-level HDL: Chisel, Bluespec SystemVerilog
- **Research activities**
  - Functional languages: Haskell (Bluespec, Galois, MIT)
  - ISA-centric ADL: L3 DSL (SRI), nML (ISP RAS)
  - Logic frameworks: Coq (MIT), HOL4 (SRI), Why3 (ISP RAS)
- *… and possibly more (not familiar with all of them)*

RISC-V Open Specifications

- https://github.com/mit-plv/riscv-semantics
  by MIT in **Haskell**
- https://github.com/cliffordwolf/riscv-formal
  by Clifford Wolf (Symbiotic EDA) in **SystemVerilog**
- https://github.com/samuelgruetter/riscv-coq
  by Samuel Gruetter (MIT) in **Coq**
- https://github.com/rsnikhil/RISCV_ISA_Formal_Spec_in_BSV
  by Rishiyur Nikhil (Bluespec) in **Bluespec SystemVerilog**
- https://github.com/rsnikhil/RISCV-ISA-Spec
  by Rishiyur Nikhil (Bluespec) in **Haskell**
- https://github.com/SRI-CSL/l3riscv
  by Prashant Mundkur (SRI International) in **L3**
- https://forge.ispras.ru/projects/microtesk-riscv
  by ISP RAS in **nML**

Specification Approach and Framework



2.2 Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

```
lui  a0,0xdead
ori  a0,a0,0x0
lui  a1,0xbeef
ori  a1,a1,0xf
add  t0,a0,a1
```

https://forge.ispras.ru/projects/microtesk

nML Language Overview

```
#ifdef RV64I
 let XLEN = 64
 ...
#else
 ...
#endif
```

```
type WORD    = card(32)
type XWORD   = card(XLEN)
type FLOAT32 = float(23, 8)
```

```
reg XREG [32, XWORD]
reg PC   [XWORD]
```

```
let MEMORY_SIZE_IN_WORDS = 2 ** (XLEN - 2)
shared mem MEM[MEMORY_SIZE_IN_WORDS, WORD]
```

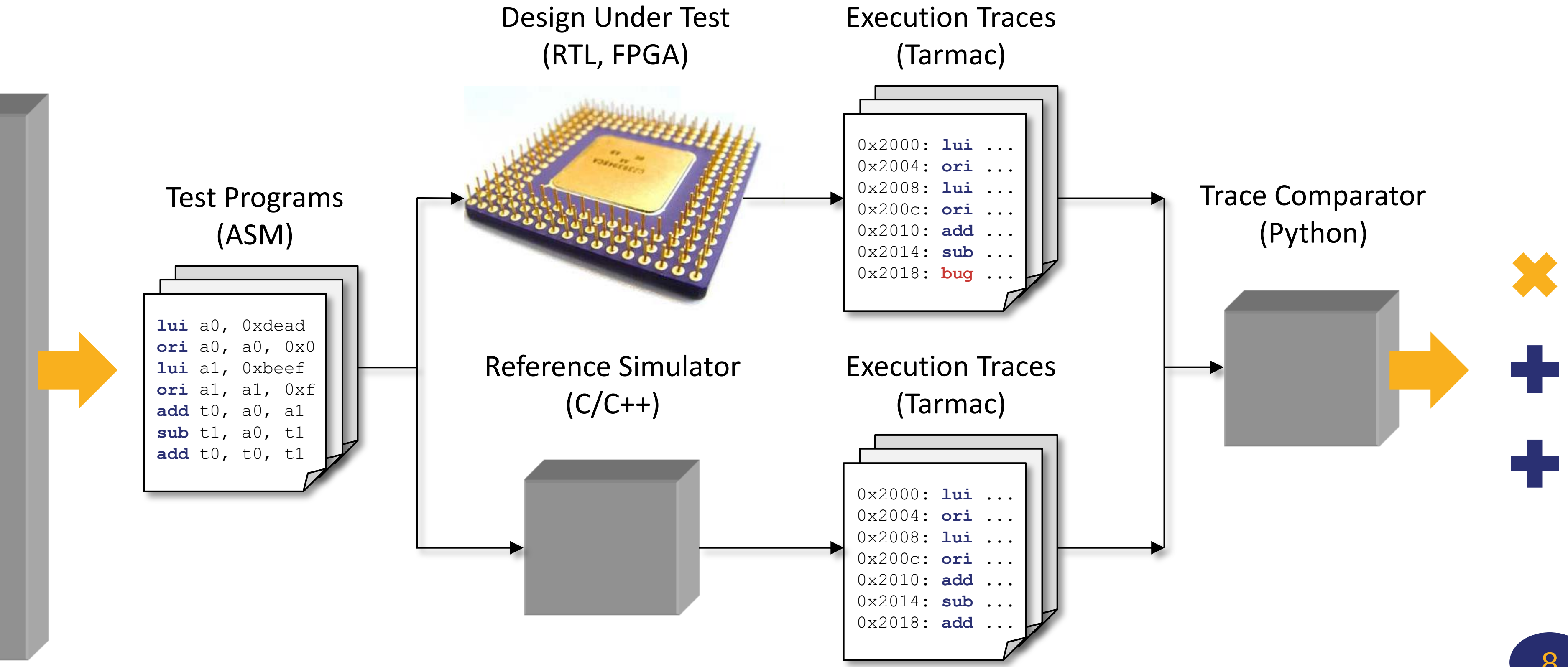
```
op add (rd: X, rs1: X, rs2: X)
 syntax = format("add %s, %s, %s",
  rd.syntax, rs1.syntax, rs2.syntax)
 image  = format("0000000%s%s000%s0110011",
  rs2.image, rs1.image, rd.image)
 action = {
  rd = rs1 + rs2;
 }
```

```
mode X (i: card(5)) = XREG[i]
 syntax = format("%s",
  if   i==0 then ZERO().syntax
  elif i==1 then RA().syntax
  elif i==2 then SP().syntax
  elif i==3 then GP().syntax
  elif i==4 then TP().syntax
  elif i>=5 && i<=7 then
    Temp(coerce(card(3), i-5)).syntax
  elif i>=8 && i<=9 then
    Saved(coerce(card(4), i-8)).syntax
  elif i>=10 && i<=17 then
    Func(coerce(card(3), i-10)).syntax
  elif i>=18 && i<=27 then
    Saved(coerce(card(4), i-16)).syntax
  else
    Temp(coerce(card(3), i-25)).syntax
 image = format("%5s", i)
```

RISC-V Specifications

Specifications Characteristic					Value		
RISC-V Instruction Set Manual Vol. I: User-Level ISA (v. 2.2)					**145** pages (**94** pages in chapters 2-18)		
Base	Extension				Base	Extension	In Total
RV32I	M	F	C	P	24 pages	44 pages	68 pages
RV32E	A	D	B	V			
RV64I		Q	J	N	2 pages	24 pages	26 pages
RV128I		L	T				
# Specified Instructions					**200+** instructions		
ISA Specifications Size (nML)					**4 500** LOC (w/o comments)		
MMU Specification Size (MMUSL)					**0** LOC (unspecified)		

Application 1: Test Program Generation (1/3)

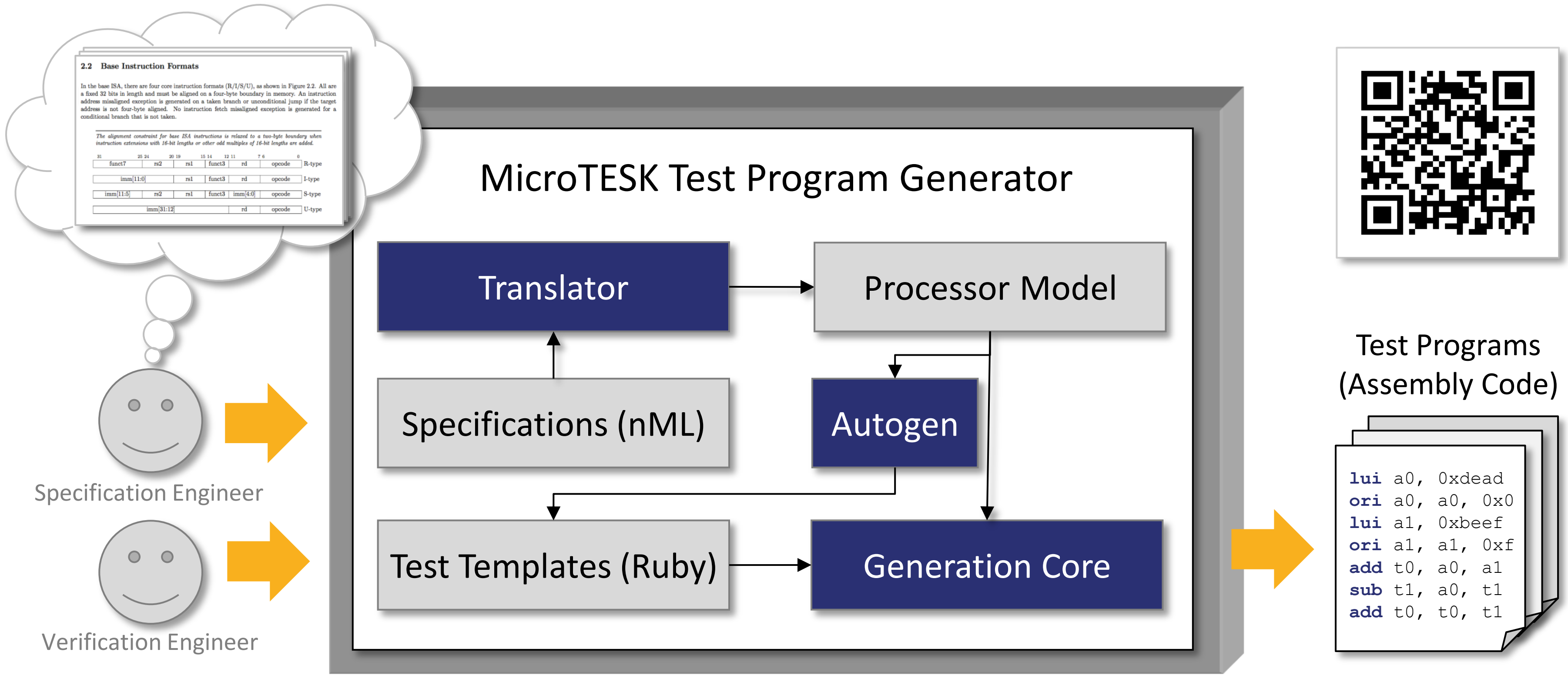


```
lui a0, 0xdead
ori a0, a0, 0x0
lui a1, 0xbeef
ori a1, a1, 0xf
add t0, a0, a1
sub t1, a0, t1
add t0, t0, t1
```

```
0x2000: lui ...
0x2004: ori ...
0x2008: lui ...
0x200c: ori ...
0x2010: add ...
0x2014: sub ...
0x2018: bug ...
```

```
0x2000: lui ...
0x2004: ori ...
0x2008: lui ...
0x200c: ori ...
0x2010: add ...
0x2014: sub ...
0x2018: add ...
```

Application 1: Test Program Generation (2/3)



2.2 Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

```
lui a0, 0xdead
ori a0, a0, 0x0
lui a1, 0xbeef
ori a1, a1, 0xf
add t0, a0, a1
sub t1, a0, t1
add t0, t0, t1
```

ISPRAS

https://forge.ispras.ru/projects/microtesk-riscv

Application 1: Test Program Generation (3/3)

```
class MyTemplate < RiscVBaseTemplate
 def run
  block(:combinator => 'product',
        :compositor => 'random') {
   iterate {
    xor x(_), x(_), x(_) do situation(...) end
    lui x(_), _
   }
   iterate {
    and x(_), x(_), x(_)
    or  x(_), x(_), x(_)
   }
  }.run
 end
end
```

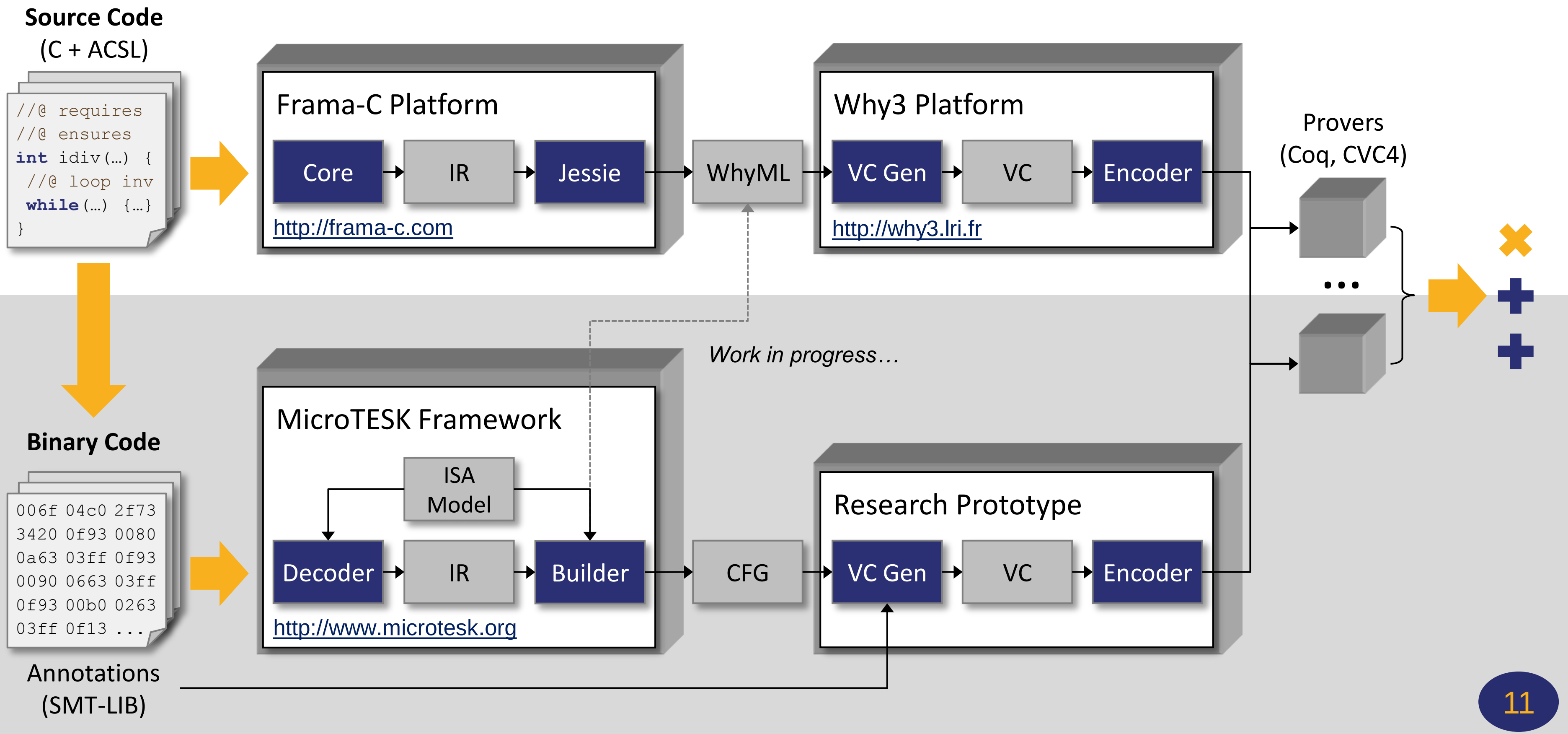
- *Combinatorial Brute Force*
- *Randomization*
- *Constraint Solving*
- *Self Checking*

```
# Single Test Case
# Initialization
ori     a7, a7, 0x2d7
slli    a7, a7, 0xb
ori     a7, a7, 0x1
slli    a7, a7, 0xb
ori     a7, a7, 0x3d2
ori     t3, t3, 0x164
slli    t3, t3, 0xb
ori     t3, t3, 0x52b
slli    t3, t3, 0xb
ori     t3, t3, 0x24e
# Stimulus
and     s4, a7, a7
xor     s8, s4, t3
```

$2 \times 2 = 4$ test cases

- Test templates similar to **RISC-V Foundation's Tests**
  https://github.com/riscv/riscv-tests
- Test templates similar to **RISC-V Torture Test Generator**
  https://github.com/ucb-bar/riscv-torture

Application 2: Binary Code Verification (1/2)



```
//@ requires
//@ ensures
int idiv(…) {
 //@ loop inv
 while(…) {…}
}
```

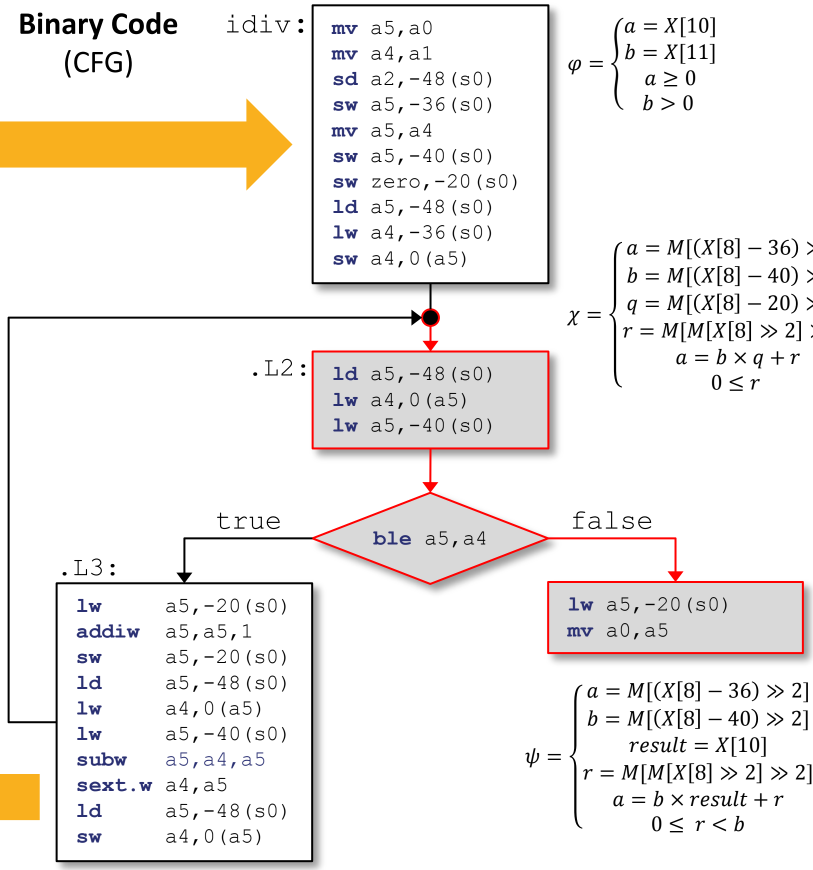
```
006f 04c0 2f73
3420 0f93 0080
0a63 03ff 0f93
0090 0663 03ff
0f93 00b0 0263
03ff 0f13 ...
```

Application 2: Binary Code Verification (2/2)

```c
//@ requires a >= 0;
//@ requires b > 0;
//@ ensures a == b*\result + *r;
//@ ensures 0 <= *r < b;
int idiv (int a, int b, int *r) {
 int q = 0; *r = a;
 //@ loop invariant (a == b*q + *r);
 //@ loop invariant (0 <= *r)
 while (*r >= b) {
  q++; *r -= b;
 }
 //@ assert (a == q*b + *r);
 //@ assert (0 <= *r < b);
 return q;
}
```

Source Code
(C + ACSL)

Binary Code
(CFG)



```
idiv:
mv  a5,a0
mv  a4,a1
sd  a2,-48(s0)
sw  a5,-36(s0)
mv  a5,a4
sw  a5,-40(s0)
sw  zero,-20(s0)
ld  a5,-48(s0)
lw  a4,-36(s0)
sw  a4,0(a5)

.L2:
ld  a5,-48(s0)
lw  a4,0(a5)
lw  a5,-40(s0)

ble a5,a4
true → .L3, false →

.L3:
lw      a5,-20(s0)
addiw   a5,a5,1
sw      a5,-20(s0)
ld      a5,-48(s0)
lw      a4,0(a5)
lw      a5,-40(s0)
subw    a5,a4,a5
sext.w  a4,a5
ld      a5,-48(s0)
sw      a4,0(a5)

lw  a5,-20(s0)
mv  a0,a5
```

$$\varphi = \begin{cases} a = X[10] \\ b = X[11] \\ a \geq 0 \\ b > 0 \end{cases}$$

$$\chi = \begin{cases} a = M[(X[8]-36) \gg 2] \\ b = M[(X[8]-40) \gg 2] \\ q = M[(X[8]-20) \gg 2] \\ r = M[M[X[8] \gg 2] \gg 2] \\ a = b \times q + r \\ 0 \leq r \end{cases}$$

$$\psi = \begin{cases} a = M[(X[8]-36) \gg 2] \\ b = M[(X[8]-40) \gg 2] \\ result = X[10] \\ r = M[M[X[8] \gg 2] \gg 2] \\ a = b \times result + r \\ 0 \leq r < b \end{cases}$$

Verification Conditions
(SMT-LIB)

```
(assert (= op0.rd.i!1  #b01111))
(assert (= op0.rs1.i!1 #b01000))
(assert (= op0.imm!1   #b111111010000))
(assert (and                                 ; ld a5, -48(s0)
 (= op0.rs1!1 (select X!1 op0.rs1.i!1))
 (= addr!1 (bvadd op0.rs1!1 ((_ sign_extend 52) op0.imm!1)))
 (= index!1 (bvlshr addr!1 #b0...010))
 ...
 (= dword!9 (select M!1 index!1))
 (= X!2 (store X!1 op2.rd.i!1 dword!9))
...
(assert (and                                 ; mv a0, a5
 (= X!10 (store X!9 op5.rd.i!1 op5.rd!1))
 (= op5.rs1!1 (select X!9 op5.rs1.i!1))
 (= op5.rd!1 (bvadd op5.rs1!1 ((_ sign_extend 52) op5.imm!1)))
```

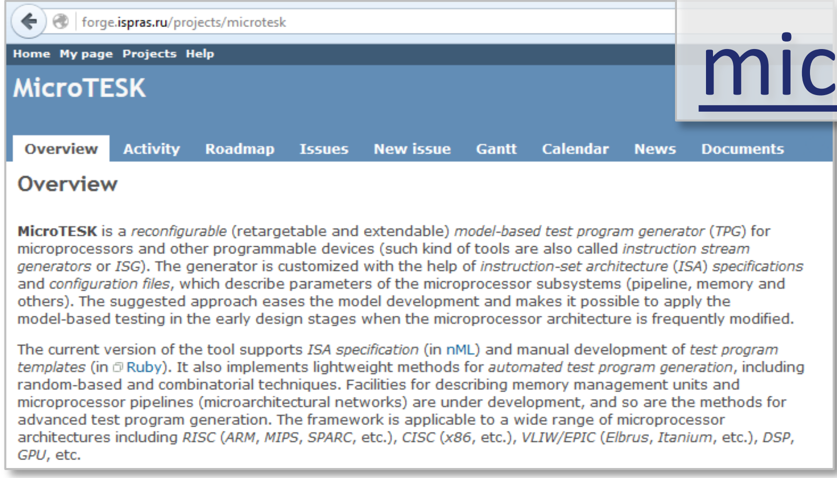
Future Work Directions

- **RISC-V specifications**
  - Specification of subsets (RV128I, Q, L, B, T, J, P, V, and N)
  - Specification of MMU (address translation mechanisms)
- **MicroTESK framework**
  - Online test program generation (post-silicon verification)
  - RTL generation (golden model for equivalence checking)
- **Specifications validation**
  - Testing (self checking, co-simulation, coverage analysis)
  - Formal equivalence checking (HLS to Verilog + BMC)

```
_start:    .org   0x50000
           nop
           adr    x0, page_table
           msr    ttbr0_el1, x0
           movz   x0, #0x8082, LSL #16
           movk   x0, #0x3219, LSL #0
           msr    tcr_el1, x0
           isb    #0
           movz   x0, #0x30c5, LSL #16
           movk   x0, #0x0831, LSL #0
           msr    sctlr_el1, x0
           isb    #0
           movz   x0, #0x0020, LSL #16
           movk   x0, #0x4000, LSL #0
           ldar   x1, [x0, #0]
           hlt    #0xf00d
```

http://www.microtesk.org

microtesk-support@ispras.ru


forge.ispras.ru/projects/microtesk
Home My page Projects Help
MicroTESK
Overview Activity Roadmap Issues New issue Gantt Calendar News Documents

Overview

MicroTESK is a *reconfigurable* (retargetable and extendable) *model-based test program generator* (*TPG*) for microprocessors and other programmable devices (such kind of tools are also called *instruction stream generators* or *ISG*). The generator is customized with the help of *instruction-set architecture* (*ISA*) *specifications* and *configuration files*, which describe parameters of the microprocessor subsystems (pipeline, memory and others). The suggested approach eases the model development and makes it possible to apply the model-based testing in the early design stages when the microprocessor architecture is frequently modified.

The current version of the tool supports *ISA specification* (in nML) and manual development of *test program templates* (in Ruby). It also implements lightweight methods for *automated test program generation*, including random-based and combinatorial techniques. Facilities for describing memory management units and microprocessor pipelines (microarchitectural networks) are under development, and so are the methods for advanced test program generation. The framework is applicable to a wide range of microprocessor architectures including *RISC* (*ARM*, *MIPS*, *SPARC*, etc.), *CISC* (*x86*, etc.), *VLIW/EPIC* (*Elbrus*, *Itanium*, etc.), *DSP*, *GPU*, etc.

Thank You!