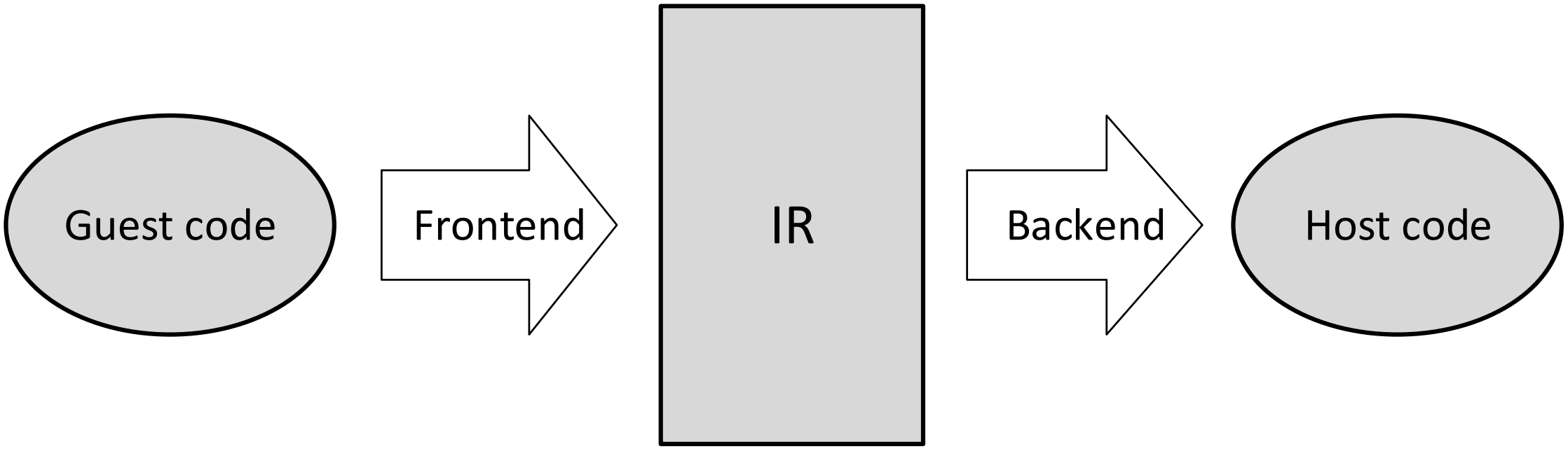
Automatic Binary Translator Generation from Instruction Set Description

Aleksandr Bezzubikov, Nikita Belov, Kirill Batuzov

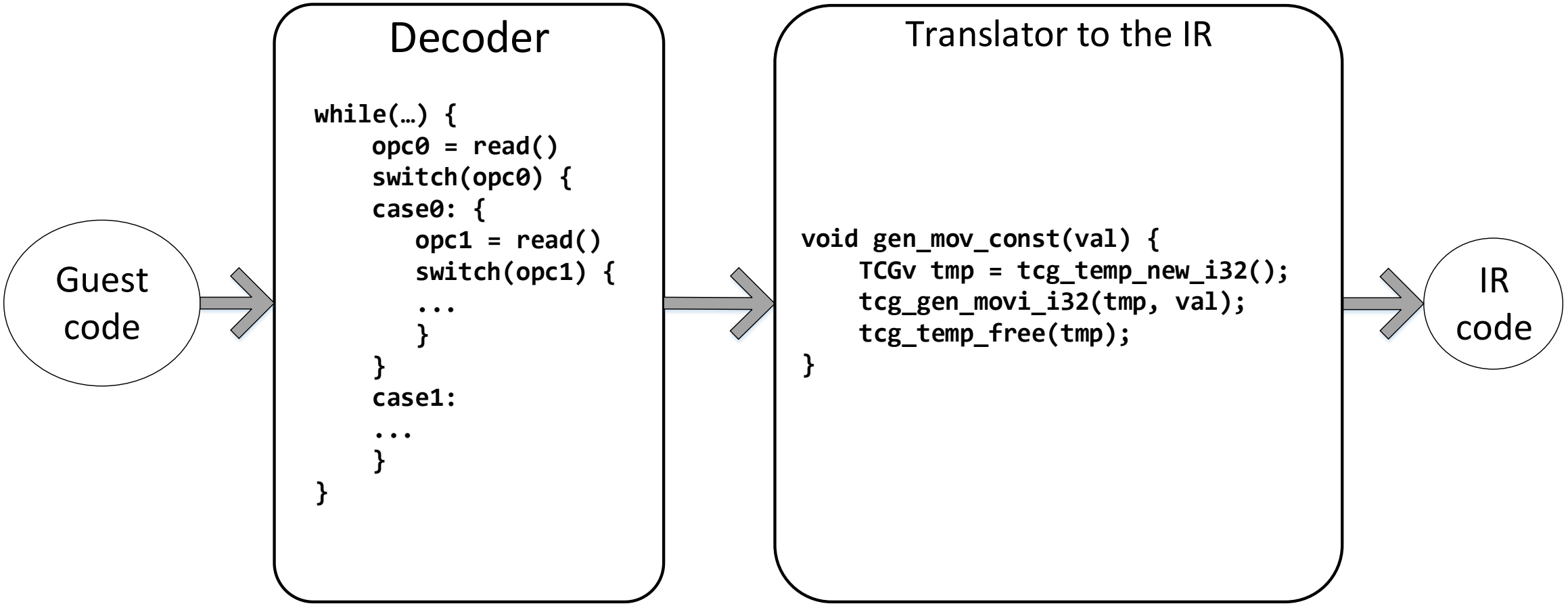
ISP RAS

1 December 2017

Typical emulator scheme



Emulator Frontend



Motivation

- Writing a decoder
  - Time-consuming as performed manually
  - Lots of boilerplate code (nested switches)
  - Lots of auxiliary code

  Can be generated automatically
- Translating to the IR
  - Existing QEMU IR is too assembly-like to be convenient
  - That’s why API is inconvenient too

Related work

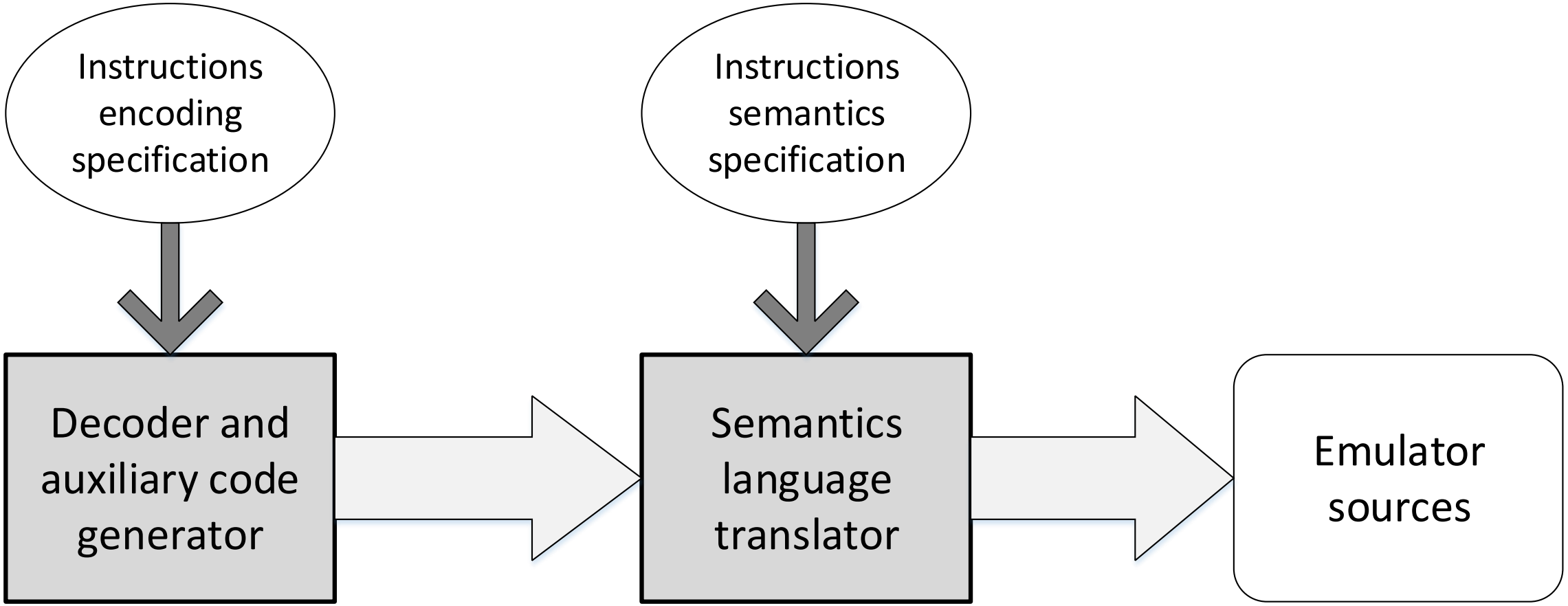
Common properties:

- High-level language for instruction specification
- Decoder and translator generation steps are splitted

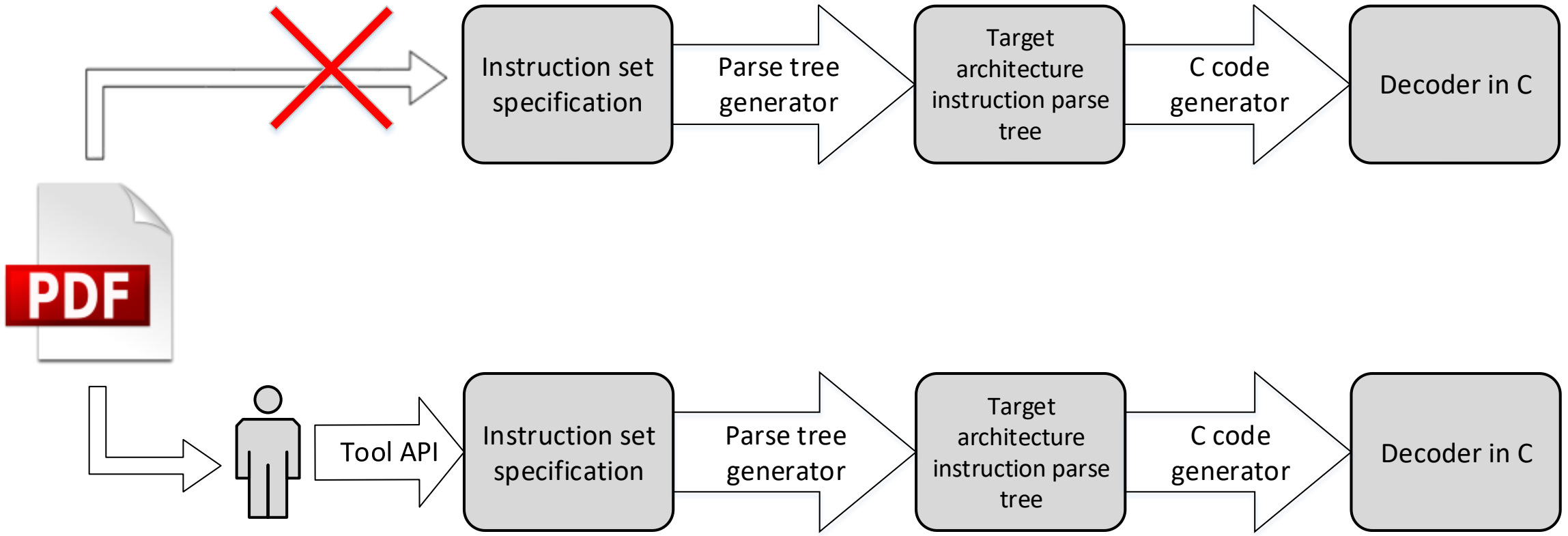
Existing languages:

- nML – doesn’t support loops
- ISDL – describes instruction set in grammars, no working translator generator exists
- LISA – too detailed, describes CPU execution process

System overview



Decoder generation



Target architecture specification

- In Python
- Instruction specification:
  - List of *Instruction* objects
  - Each *Instruction* is a list of fields:
    - `Opcode(length, value)`
    - `Operand(length, name, blockNum=0)`
    - `Reserved(length, value=0)`
  - Supports nested alternatives

    (different instructions with the same prefix)

```
instruction_list = [
    Instruction('AND',
        Opcode(8, 0b00100110),
        Operand(4, 'A'),
        Operand(4, 'B')),
    Instruction('ADD',
        Opcode(8, 0b00000101),
        Operand(4, 'A'),
        Operand(4, 'B')),
    Instruction('BRK',
        Opcode(8, 0b00110101),
        Reserved(8))
]
```

Target architecture specification

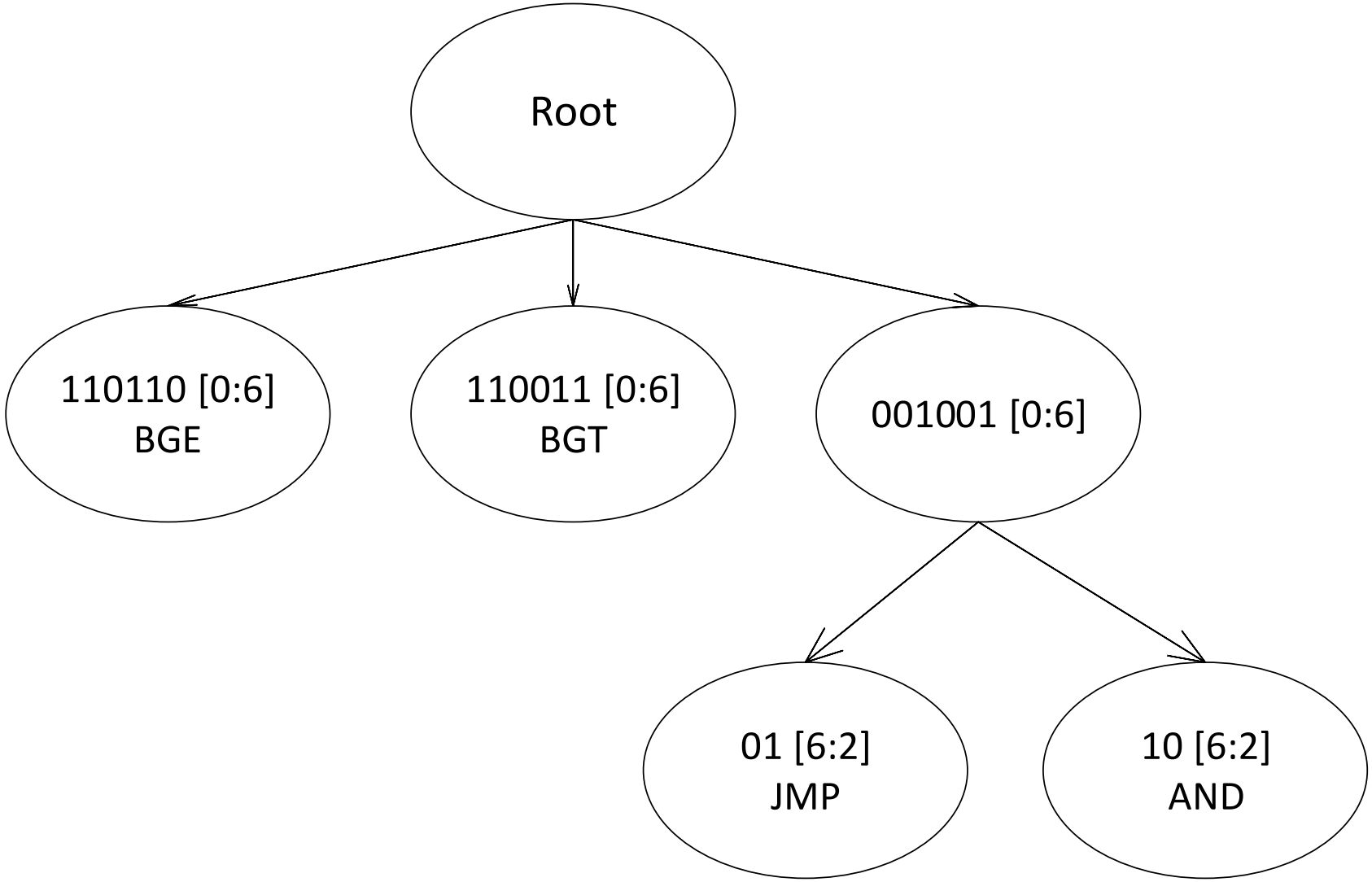
- In Python
- CPU specification:
  1) Attributes
  2) Registers
  3) CPU state fields

```
target_cpu = TargetCPU(
    ‘Moxie’,
    Attribute(‘TARGET_LONG_BITS’, 32),
    Attribute(‘TARGET_PAGE_BITS’, 12),
    Attribute(‘NB_MMU_MODES’, 1),
    StateField(‘uint32_t’, ‘flags’),
    StateField(‘uint32_t’, ‘pc’),
    RegisterGroup(‘gregs’, regs=
        RegisterRange(‘gr’, elem_size=32,
                      end=16).regs),
    Register(‘cc_a’, 32),
    Register(‘cc_b’, 32)
)
```

Generating parse tree

- Instruction encoding: a string of ‘0’, ‘1’ and ‘x’
- Recursive algorithm over a set of instruction encodings:
  - step 0: set contains all the instructions
  - step k:
    - size(current_set) = 1
      1) save a reference to the instruction
      2) return
    - size(current_set) > 1:
      1) find fragment of length *L* starting from position *a* consisting only of ‘0’ and ‘1’
      2) split current set by different values of the substring found
      3) for each subset perform a recursive call

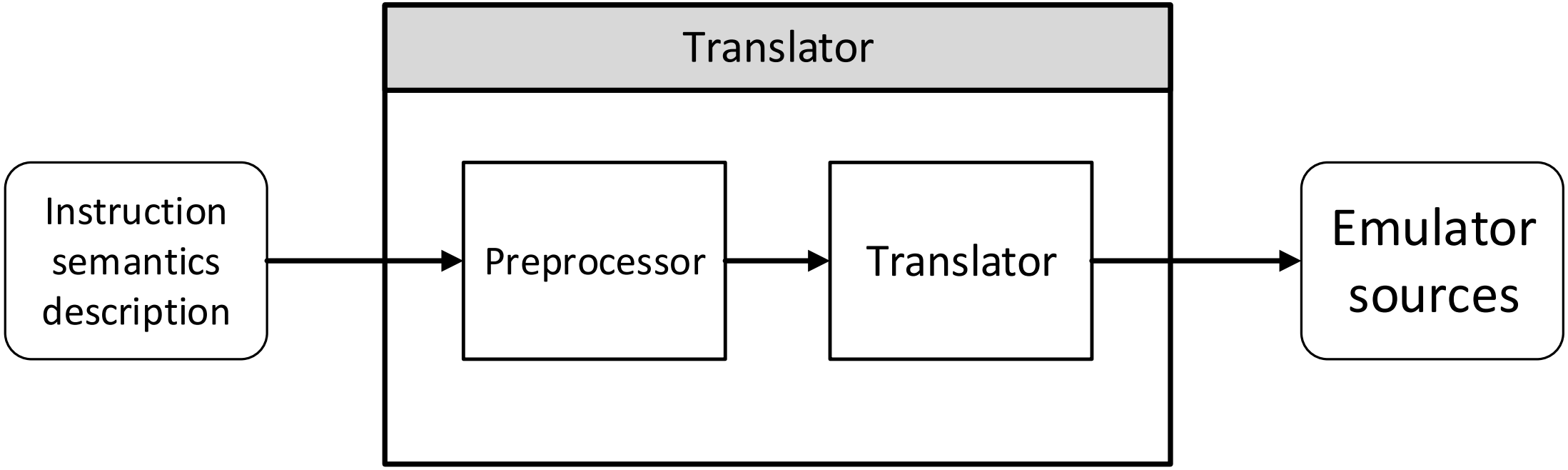
Parse tree example



Semantics description language

- Convenient syntax
  - Based on C11
- DBT-specific
  - Added some QEMU-specific operators: signed/unsigned extension, signed division/remainder, cyclic shift and signed shift
  - Runtime values distinction
    1) Compile-time constants
    2) Resulting translator runtime values
    3) Guest code runtime values – HLTTemp

Language translator overview



Semantics language translator: preprocessing

Transform into C11-compliant form:

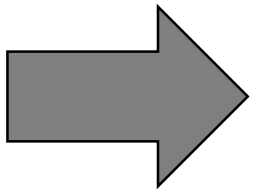
- Define HLTTemp type
- Replace non-standard operators with API function calls

Semantics language translator: translation

- Parse input into an AST using clang
- Traverse AST and transform it with clang::Rewriter
- Transformation:
  - Statements independent of HLTTemp are copied ‘as-is’
  - Statement which depend on HLTTemp are translated into SSA form that is then used as translator IR
  - Translator IR operations are mapped to QEMU API calls, translator simply outputs these calls
  - For each HLTTemp variable declaration its allocation and deallocation are generated

Semantics language translation example

```
extern HLTTemp gregs[16];
static void inc_0(DisasContext *ctx,
                  int r, int imm)
{
    if(imm == 1) {
        gregs[r]++;
    } else {
        gregs[r] += imm;
    }
}
```



```
extern TCGv cpu_gregs[16];
static void inc_0(DisasContext *ctx,
                  int r, int imm)
{
    if(imm == 1) {
        TCGv _tmp_1 = tcg_temp_new();
        tcg_gen_mov_tl(_tmp_1, cpu_gregs[r]);
        tcg_gen_addi_tl(cpu_gregs[r],
                        cpu_gregs[r], 1);
        tcg_temp_free(_tmp_1);
    } else {
        tcg_gen_addi_tl(cpu_gregs[r],
                        cpu_gregs[r], imm);
    }
}
```

Evaluation

- Moxie as a target architecture
- Decoder generator evaluation
  1) Generate binaries with random instruction sequences
  2) Pass this binaries to generated decoder
  3) Compare generated decoder output to originally compiled instructions sequence
- Semantics translator evaluation
  - Synthetic tests
- Complex Moxie architecture test – compare CPU state after each instruction execution in generated translator and existing one

Evaluation

Component	QEMU API	Proposed language
Auxiliary code	435 lines, 14656 characters	0 lines, 0 characters
CPU definition	37 lines, 1371 characters	16 lines, 639 characters
Instructions	53 lines, 1520 characters	21 lines, 430 characters

Instruction	TCG API	Proposed language
ADD	1 line, 40 characters	1 line, 10 characters
BEQ	8 lines, 568 characters	6 lines, 194 characters
CMP	2 lines, 93 characters	2 lines, 26 characters
MOV	1 line, 53 characters	1 line, 27 characters
PUSH	5 lines, 242 characters	4 lines, 125 characters
UDIV	2 lines, 116 characters	2 lines, 52 characters
XOR	1 line, 40 characters	1 line, 10 characters

Conclusion

- We proposed the new system for automatic dynamic binary translator generation
- The system allows developers to use high-level language for instruction encoding and semantics description
- The system generates correct dynamic binary translator for described target architecture
- The system was tested on Moxie architecture

Thanks!