Using x86isa for Microcode Verification

Shilpi Goel & Rob Sumners
{shilpi,rsumners}@centtech.com

SpISA 2019

Overview

- ***Broad Verification Objective:*** prove that Centaur's processors correctly implement the x86 ISA.
  - Verification of microoperations (*uops*), algorithms, prototypes, (parts of the) memory system.
  - Also: signal mapping, linting.

Overview

- ***Broad Verification Objective:*** prove that Centaur's processors correctly implement the x86 ISA.
  - Verification of microoperations (*uops*), algorithms, prototypes, (parts of the) memory system.
  - Also: signal mapping, linting.
- ***New-ish Focus:*** **prove that Centaur's processors execute a single x86 instruction correctly**. This involves reasoning about:
  - Instruction decoding.
  - For legal instructions, translation to corresponding uops.
  - Relating execution of these uops to the execution of the x86 instruction.

Challenges: Part 1

Complexity of an x86 Instruction Itself:

- *Picking a candidate instruction:*
  - Modifiers like prefixes, modes of operation.

Challenges: Part 1

Complexity of an x86 Instruction Itself:

- *Picking a candidate instruction:*
  - Modifiers like prefixes, modes of operation.
- *Decoding:*
  - Variable-length instructions.
  - Exceptions.

Challenges: Part 1

Complexity of an x86 Instruction Itself:

- *Picking a candidate instruction:*
  - Modifiers like prefixes, modes of operation.
- *Decoding:*
  - Variable-length instructions.
  - Exceptions.
- *Functional Behavior:*
  - Long specifications; lots of little details.
  - Several instructions affect a lot of machine state.
  - Deviations from“usual” behavior.

Challenges: Part 1

Complexity of an x86 Instruction Itself:

- *Picking a candidate instruction:*
  - Modifiers like prefixes, modes of operation.
- *Decoding:*
  - Variable-length instructions.
  - Exceptions.
- *Functional Behavior:*
  - Long specifications; lots of little details.
  - Several instructions affect a lot of machine state.
  - Deviations from“usual” behavior.

Context (configuration bits, CPU features) affects almost everything.

Challenges: Part 2

Complexity of Microarchitecture:

- *Translation:*
  - Queues, feedback loops, instruction caches.
  - ISA-level instructions often translate to several uops.
  - Additionally, there can be a trap to the microcode ROM.

Challenges: Part 2

Complexity of Microarchitecture:

- *Translation:*
  - Queues, feedback loops, instruction caches.
  - ISA-level instructions often translate to several uops.
  - Additionally, there can be a trap to the microcode ROM.
- *Execution:*
  - Uops can be difficult to specify.
    - Specifications obtained by talking to logic designers.
  - Microcode ROM can have arbitrary-length programs.
    - Loops and jumps are common.

Challenges: Part 2

Complexity of Microarchitecture:

- *Translation:*
  - Queues, feedback loops, instruction caches.
  - ISA-level instructions often translate to several uops.
  - Additionally, there can be a trap to the microcode ROM.
- *Execution:*
  - Uops can be difficult to specify.
    - Specifications obtained by talking to logic designers.
  - Microcode ROM can have arbitrary-length programs.
    - Loops and jumps are common.

Microarchitecture changes are frequent.

Goal: Single-Instruction Correctness

- ***Scope of this project:***
  - **Front-end** (i.e., translation of an instruction byte sequence to uops) and **execution units** (i.e., where uops are executed).
  - Do not account for register mapping, uop reordering, instruction caches, etc.

Goal: Single-Instruction Correctness

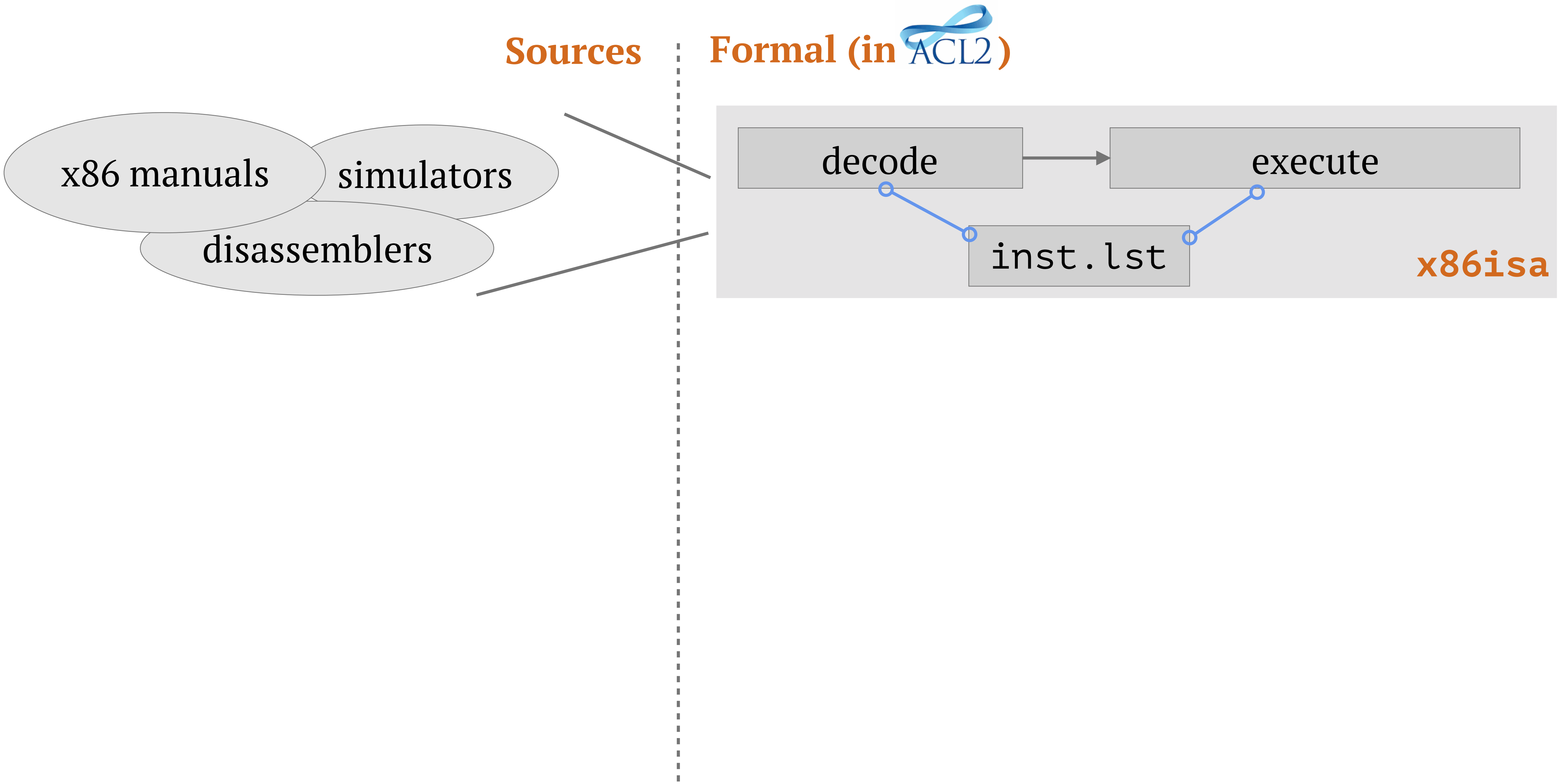
- *Scope of this project:*
  - **Front-end** (i.e., translation of an instruction byte sequence to uops) and **execution units** (i.e., where uops are executed).
  - Do not account for register mapping, uop reordering, instruction caches, etc.
- *Focus:*

  (1) **Specification:**
  `x86isa` — formal, executable model of x86 ISA in ACL2.

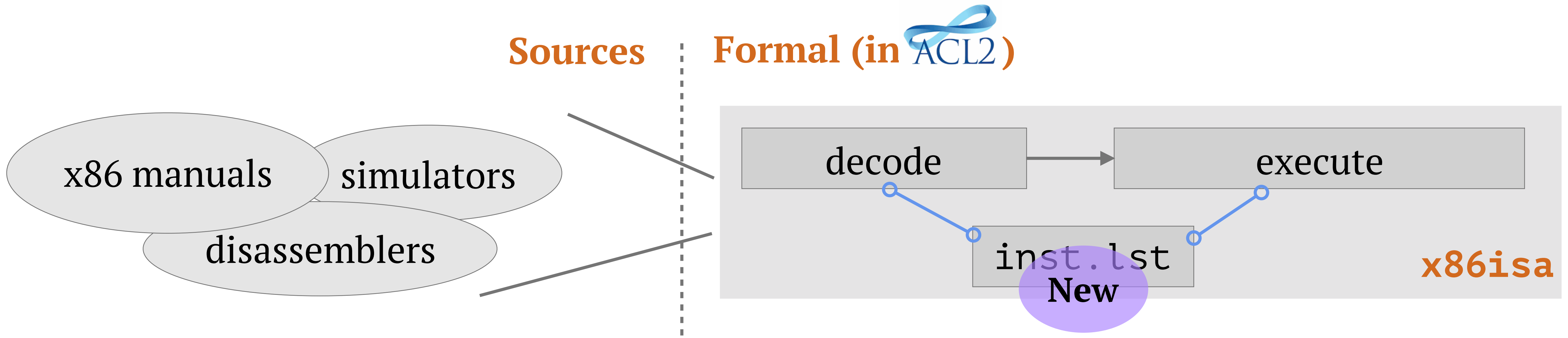
  (2) **Implementation:**
  Centaur's microarchitecture RTL-level design definition.

  (3) **Proof Methodology:**
  A scalable way to relate (1) and (2).

Our Process

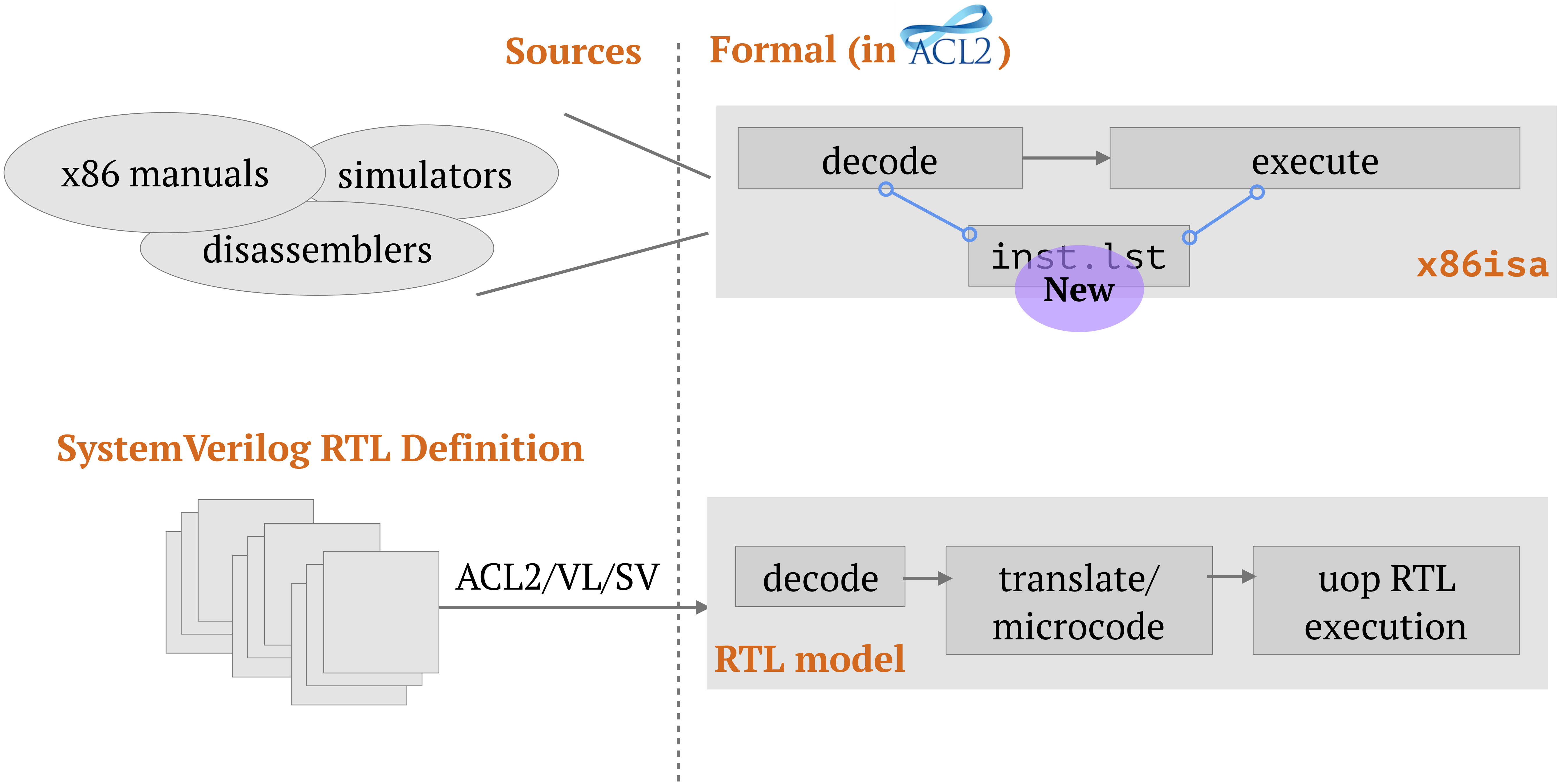

Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
x86isa

Our Process

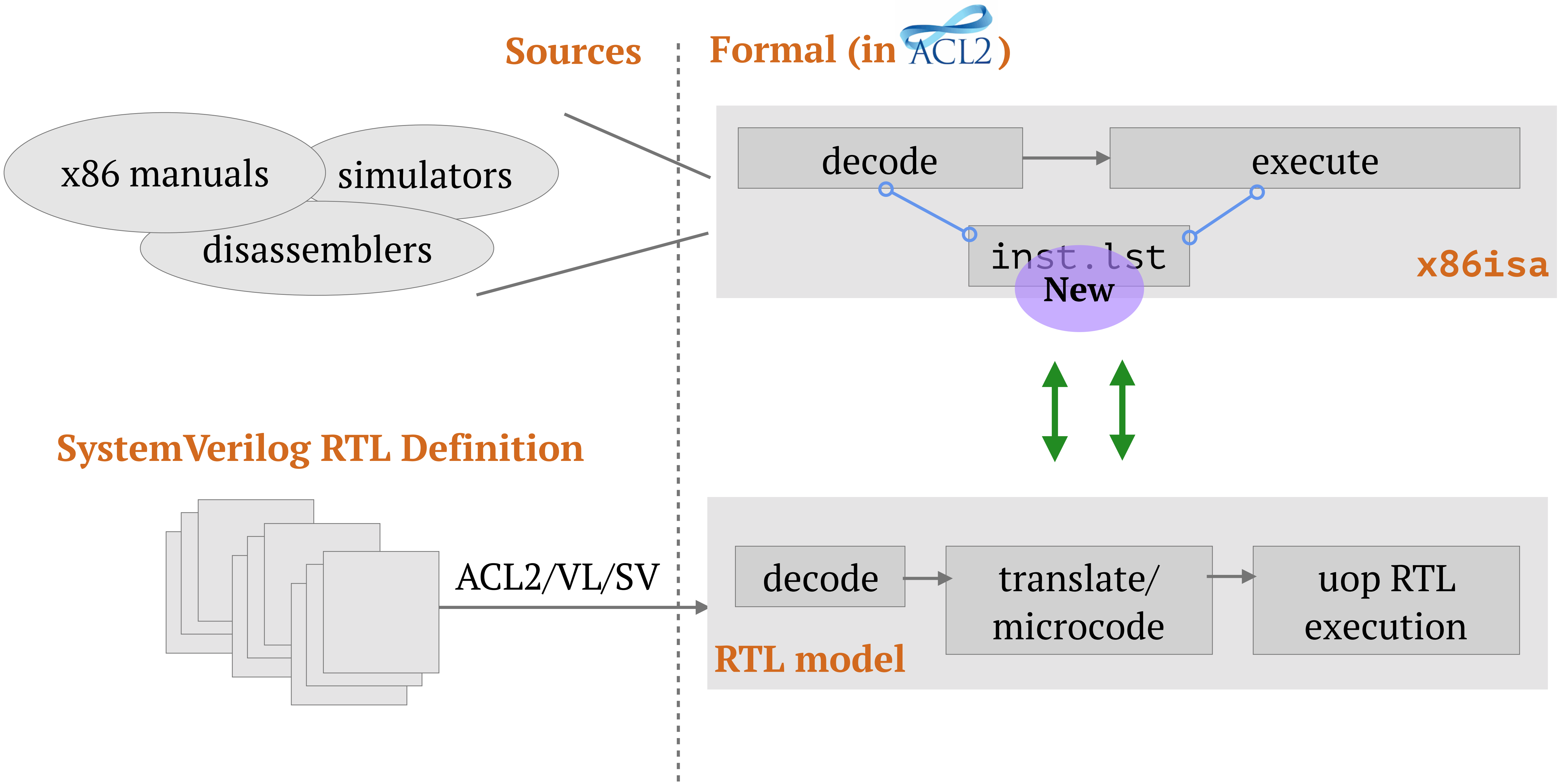


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa

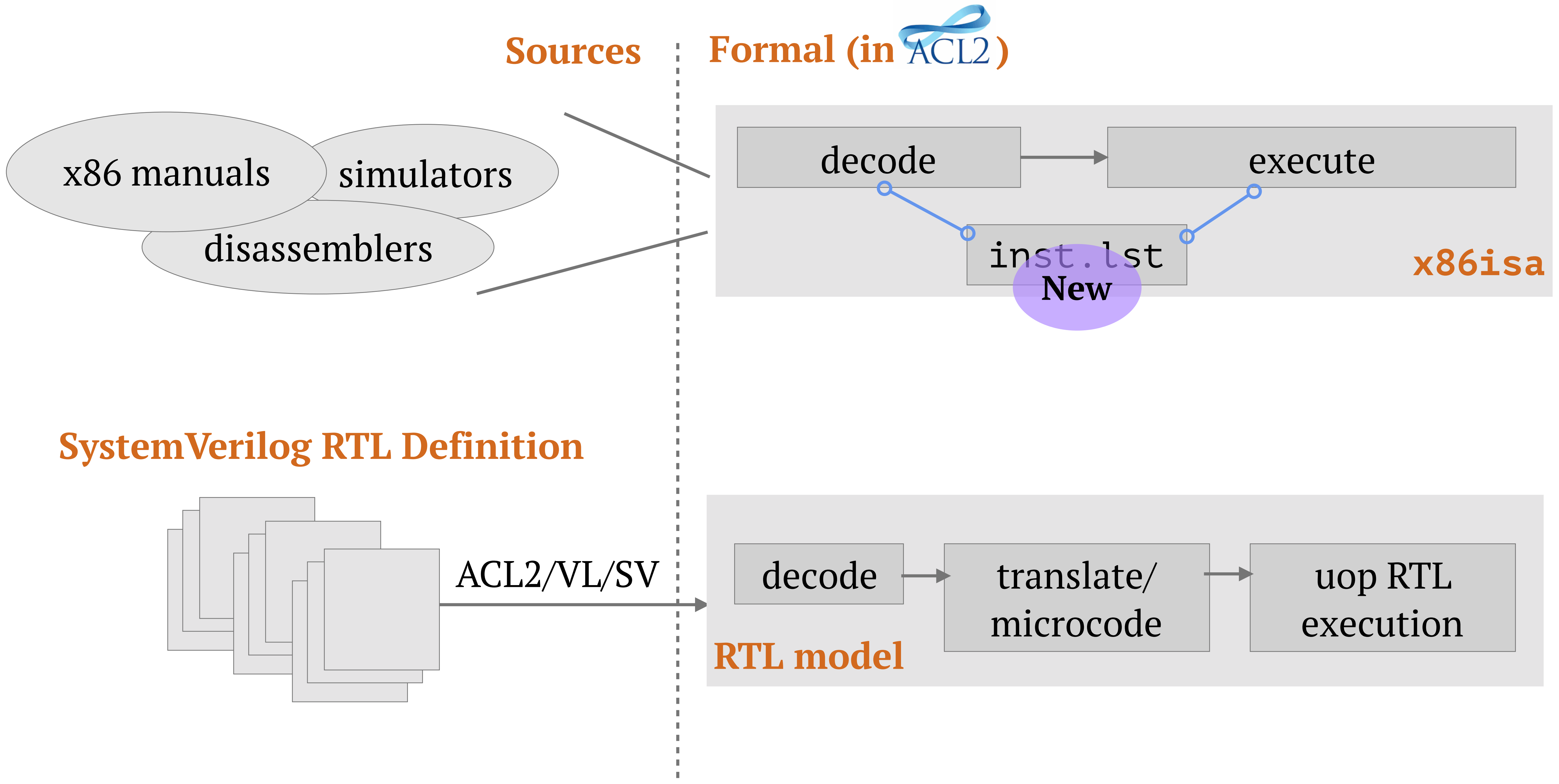
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

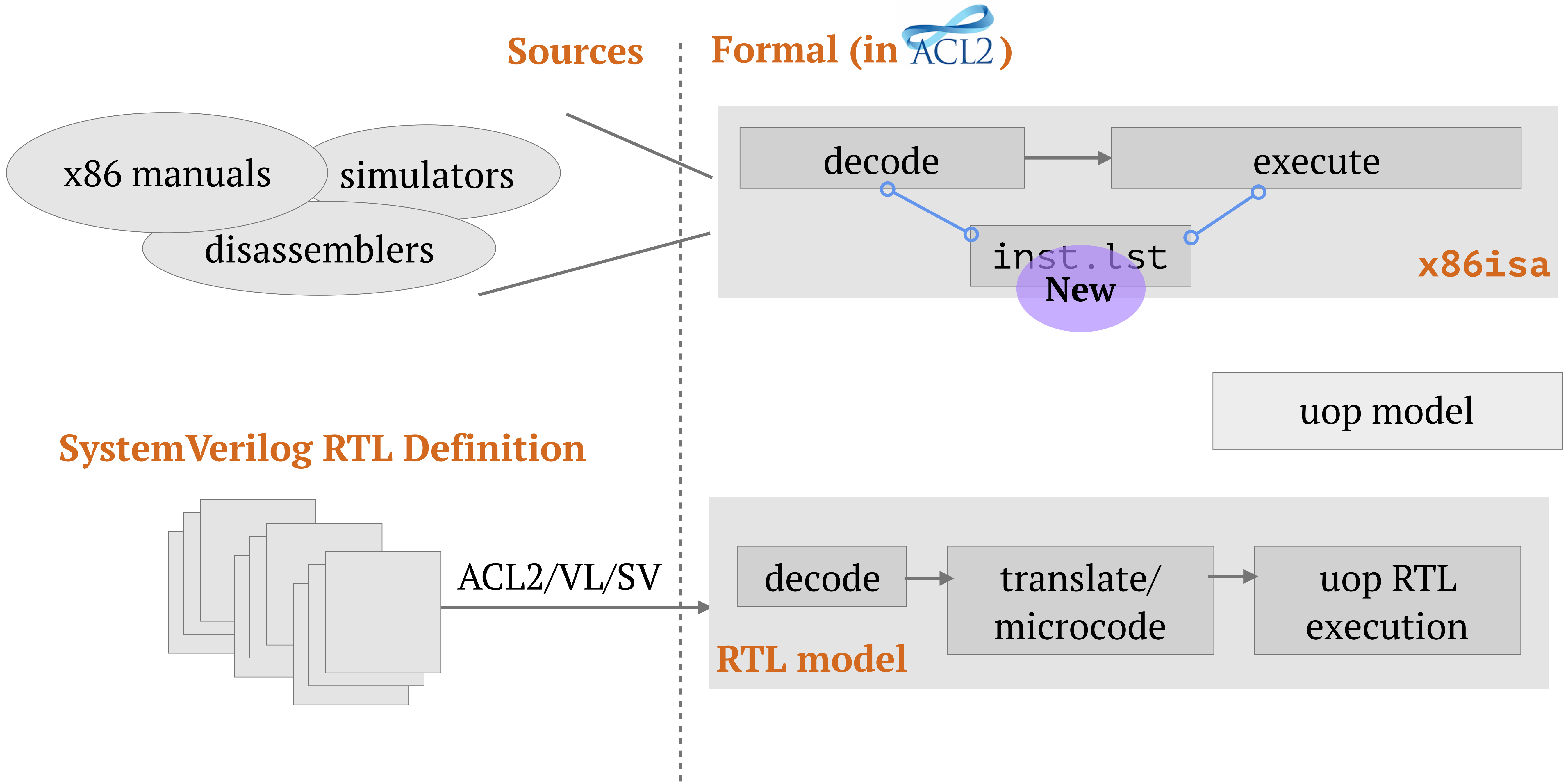
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

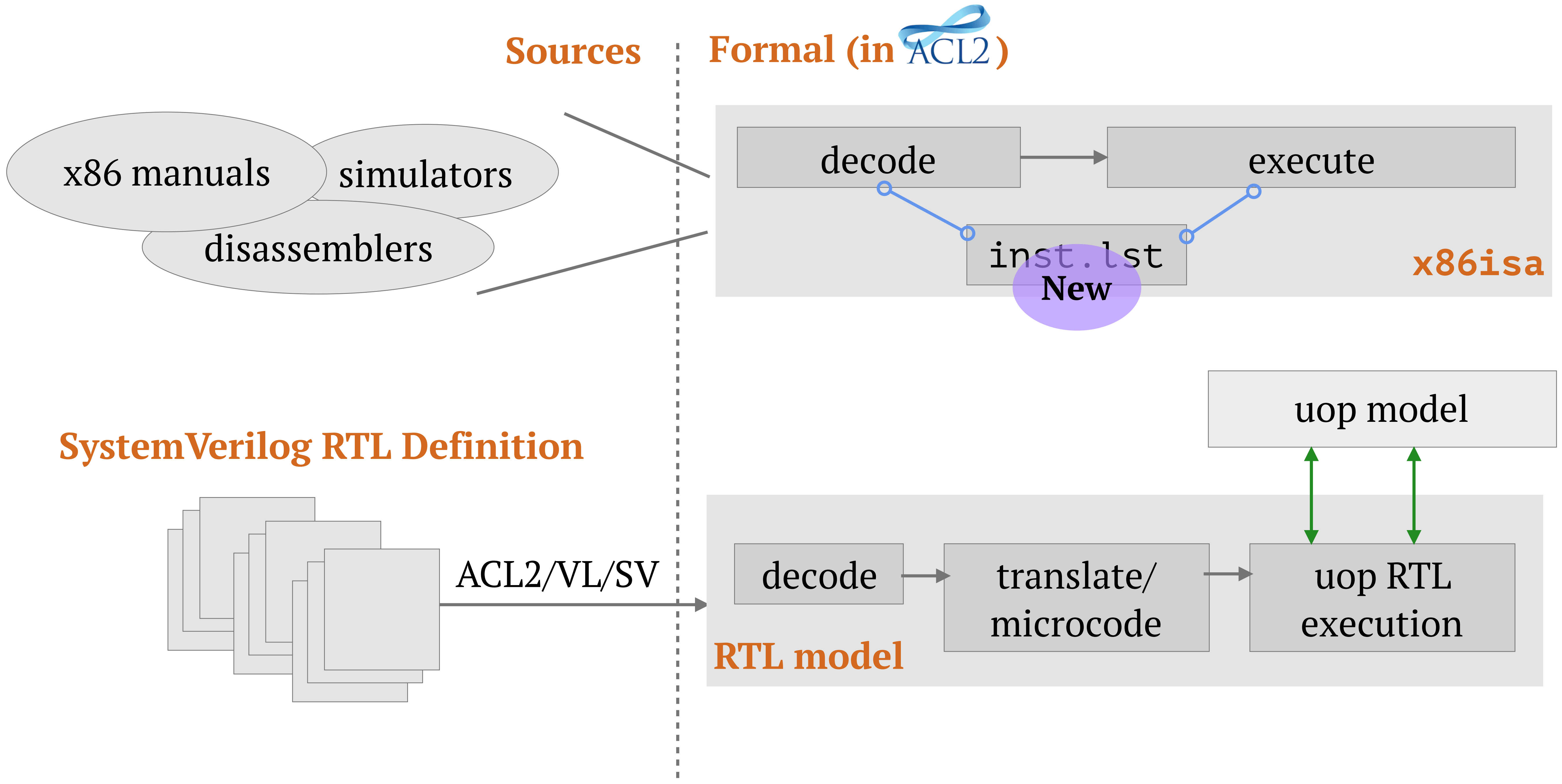
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

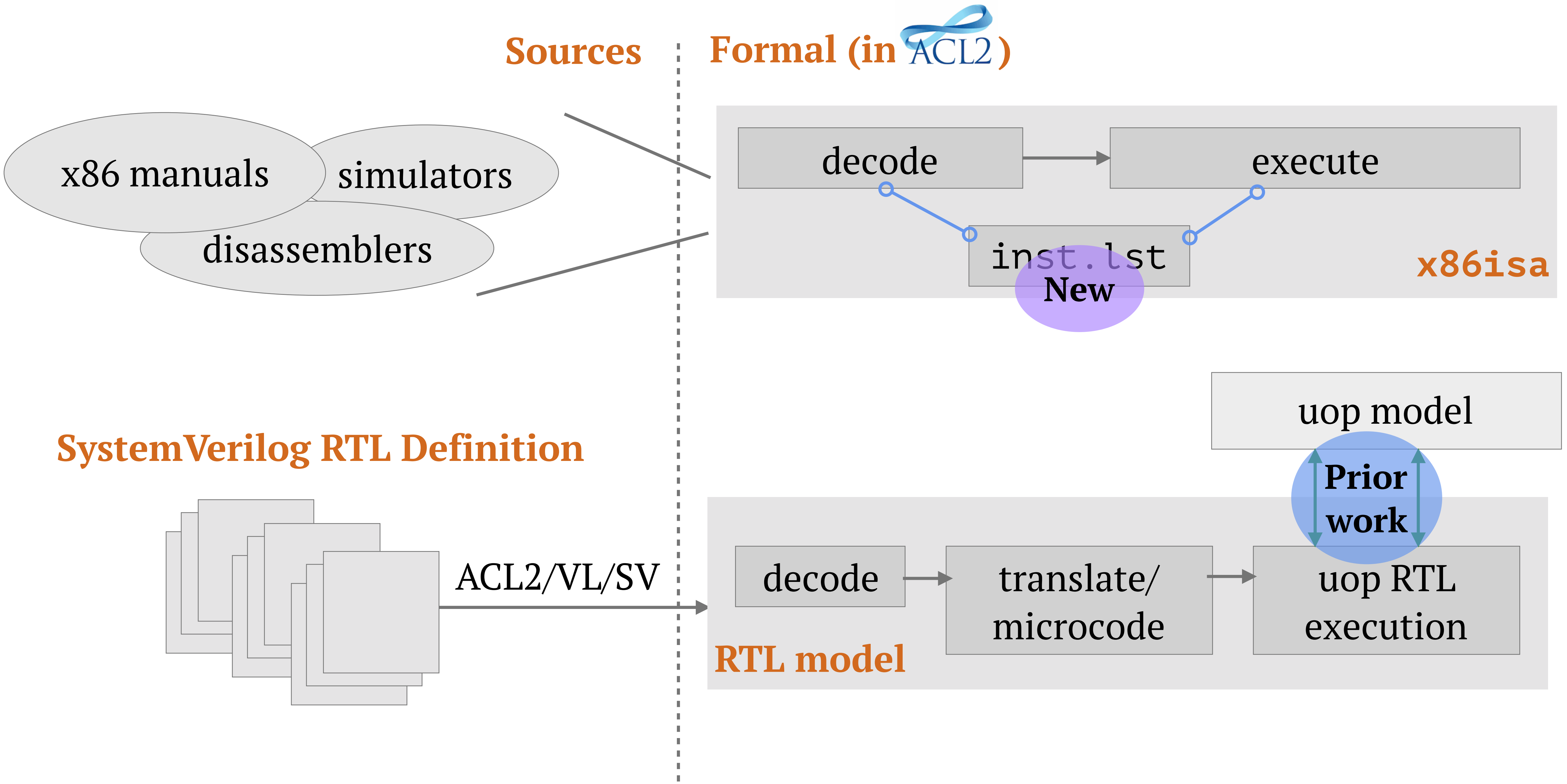
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
uop model
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/
microcode
uop RTL
execution
RTL model

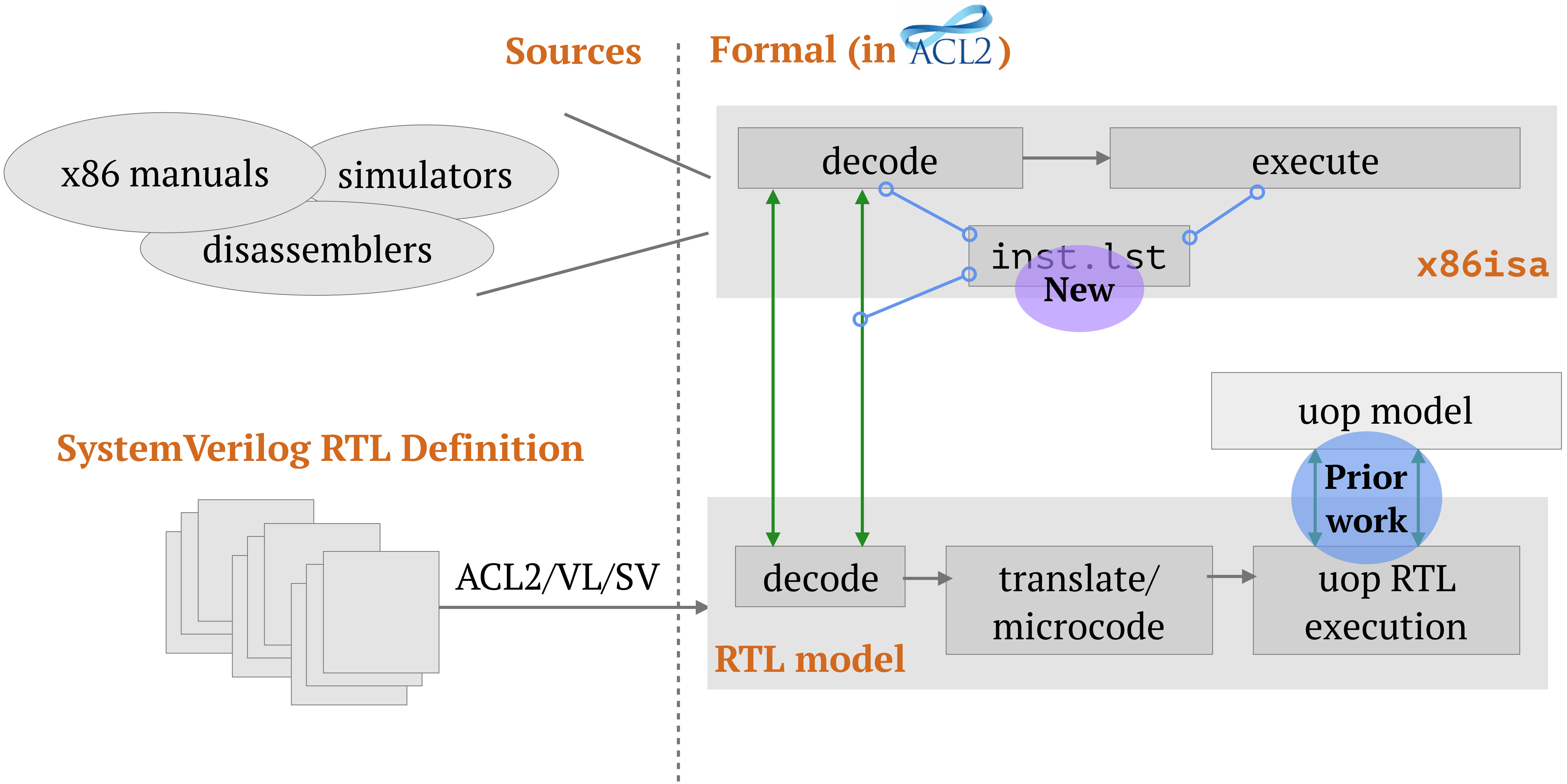
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
SystemVerilog RTL Definition
ACL2/VL/SV
uop model
decode
translate/
microcode
uop RTL
execution
RTL model

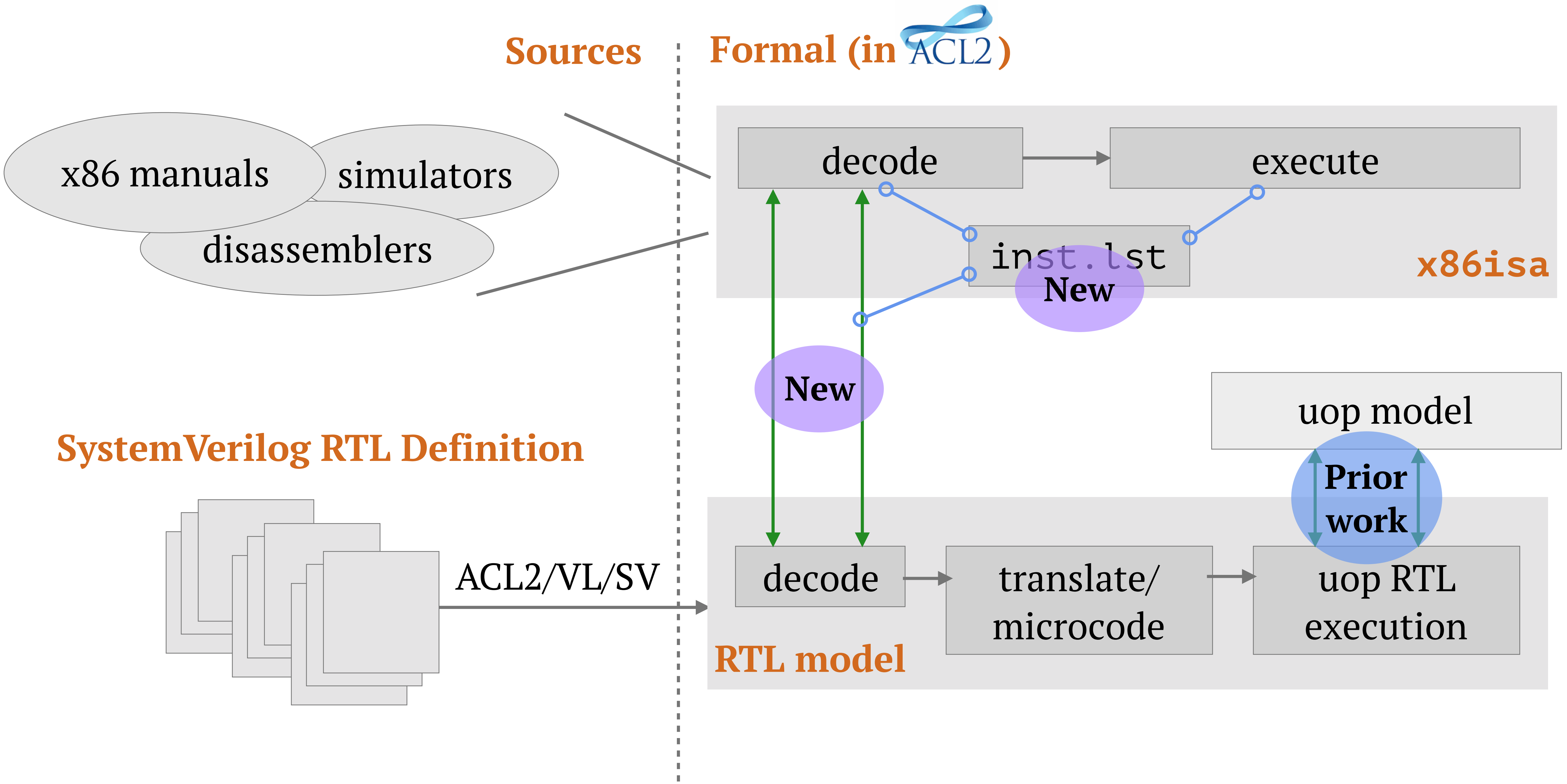
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
uop model
Prior work
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

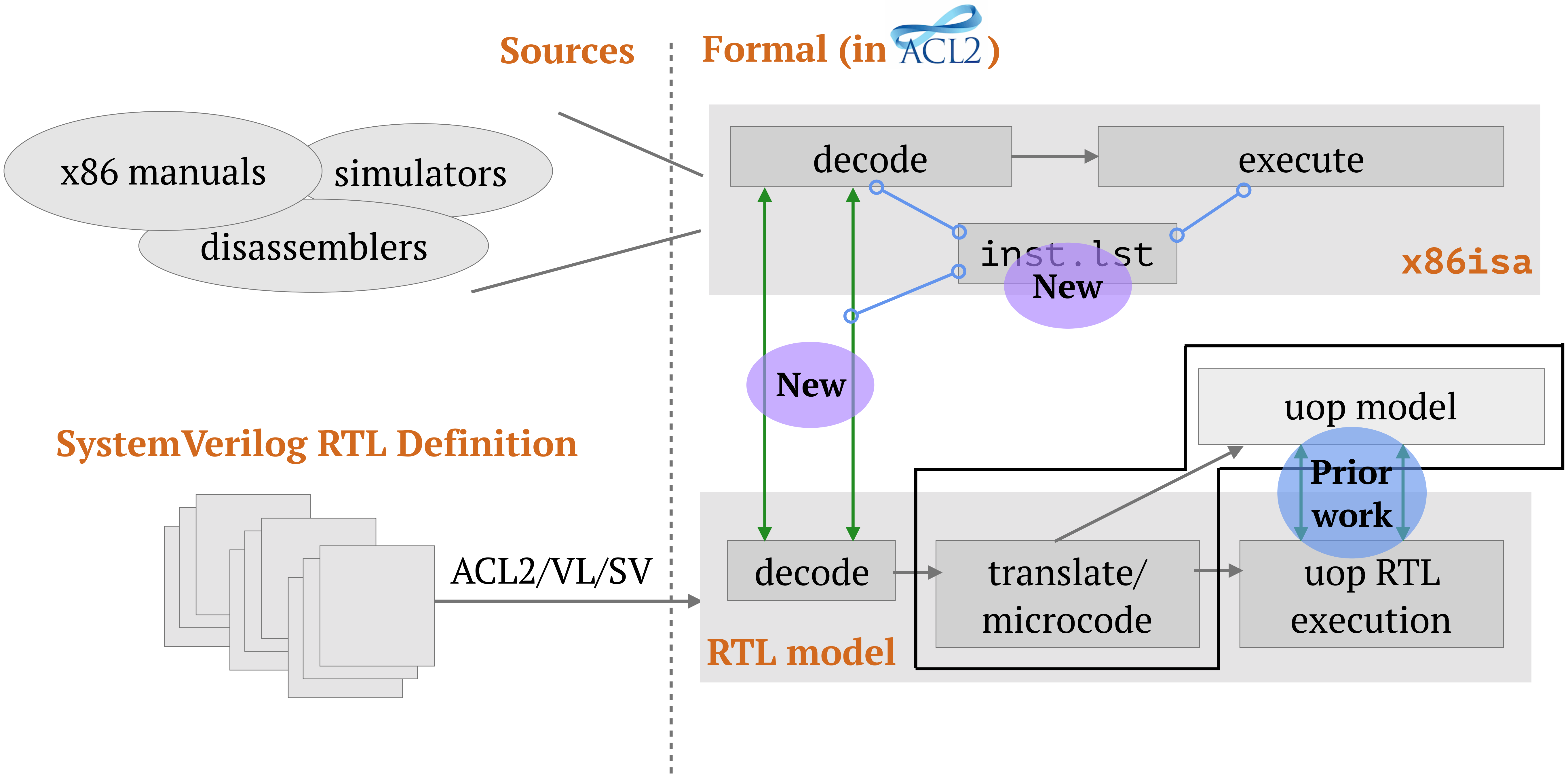
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
uop model
Prior work
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

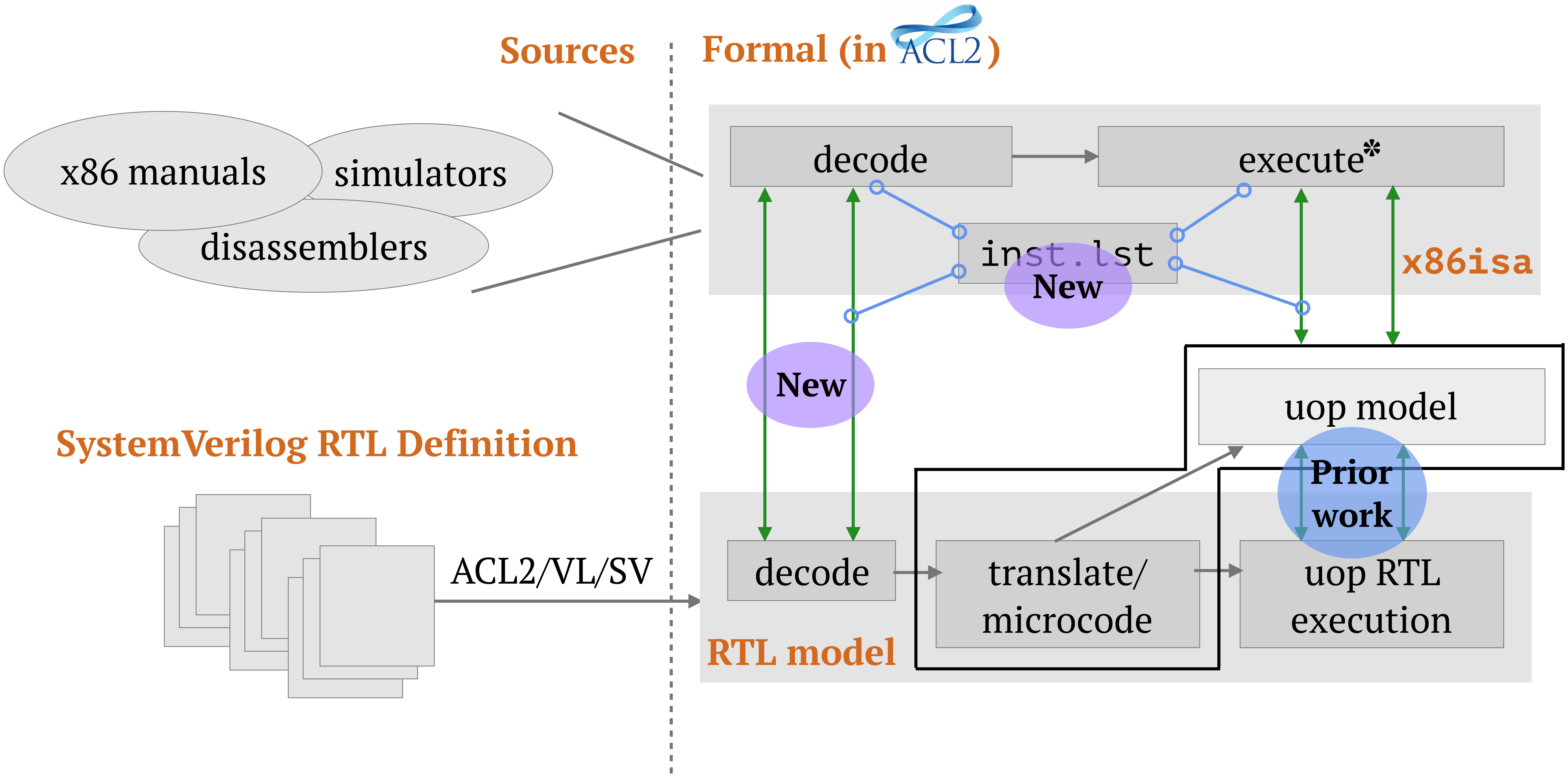
Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
New
uop model
Prior work
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

Our Process

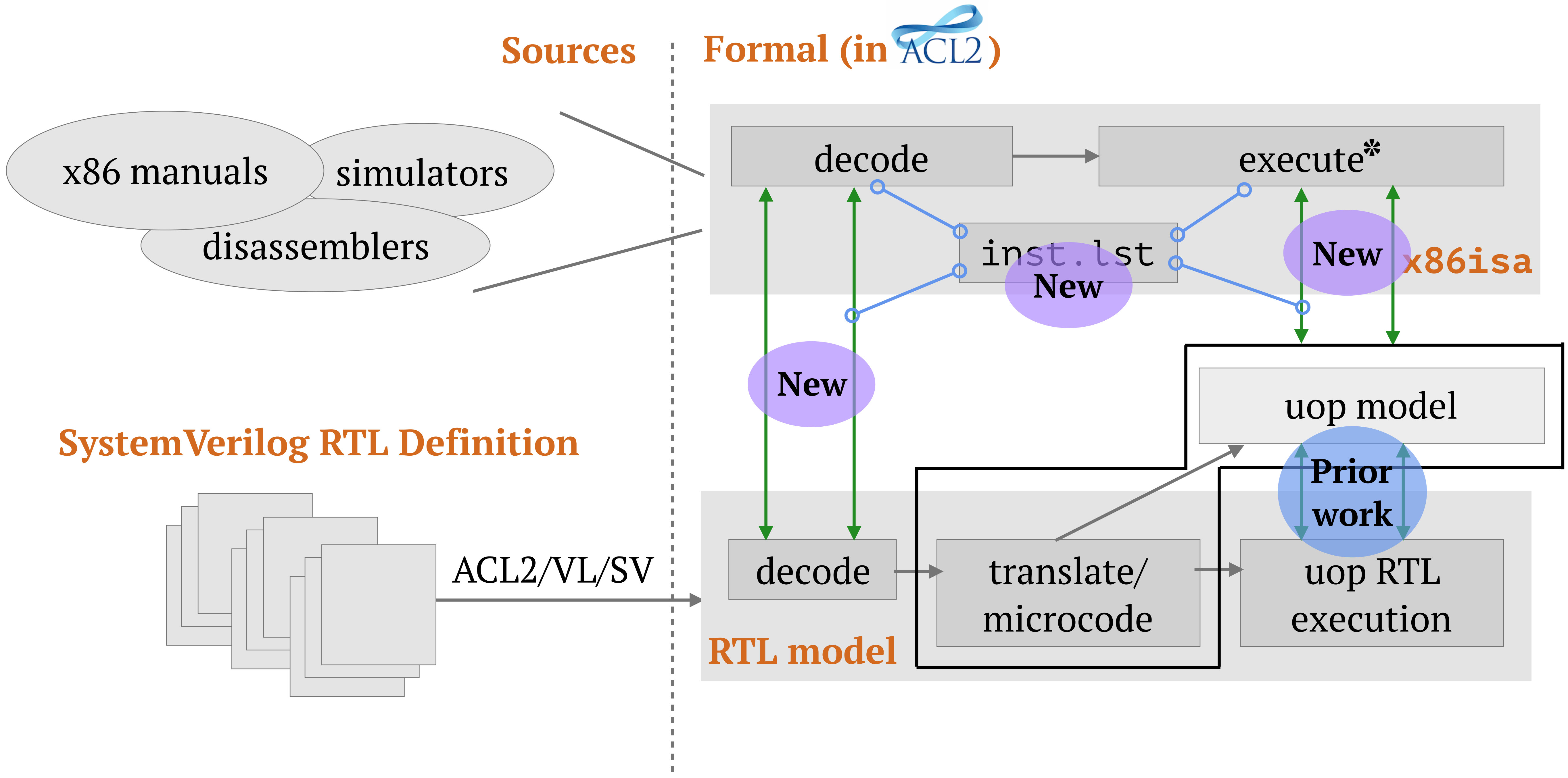

Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute
inst.lst
New
x86isa
New
uop model
Prior work
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

Our Process



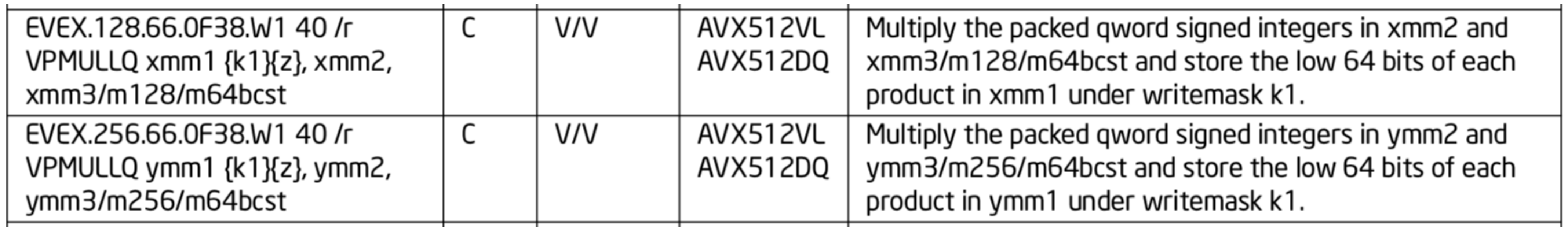
Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute*
inst.lst
New
x86isa
New
uop model
Prior work
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

Our Process


Sources
Formal (in ACL2)
x86 manuals
simulators
disassemblers
decode
execute*
inst.lst
New
New
New
x86isa
uop model
Prior work
SystemVerilog RTL Definition
ACL2/VL/SV
decode
translate/ microcode
uop RTL execution
RTL model

Specification: `inst.lst` in

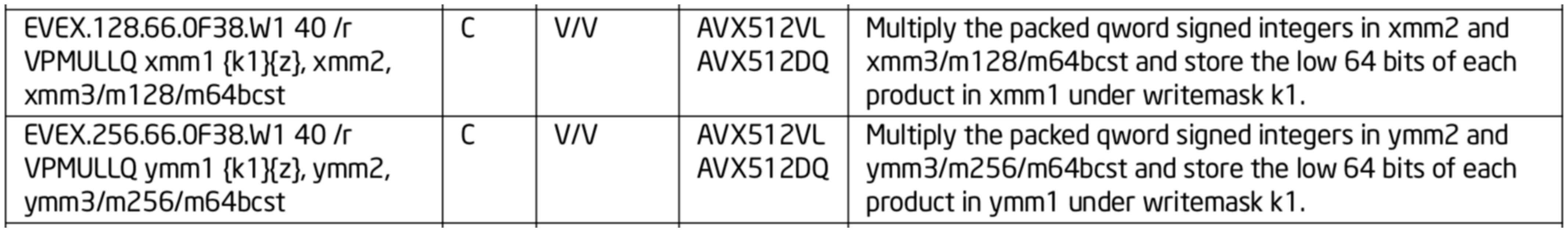
- `inst.lst` is a data structure that contains a list of all x86 instructions.
- Initial version obtained by parsing the instruction description pages.



EVEX.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.

Specification: `inst.lst` in

- `inst.lst` is a data structure that contains a list of all x86 instructions.
- Initial version obtained by parsing the instruction description pages.



EVEX.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.

```
(inst :mnemonic "VPMULLQ"
      :opcode (OP :OP #ux_0F_38_40
                  :EVEX '(:128 :66 :0F38 :W1)
                  :FEAT '(:AVX512VL :AVX512DQ))
      :operands (ARG :OP1 '(:ModR/M.reg :XMM)
                     :OP2 '(:EVEX.vvvv  :XMM)
                     :OP3 '(:ModR/M.r/m :XMM :MEM :M64BCST))
      :fn '(evex-vpmullq-spec)
      :excep '((:ex (chk-exc :TYPE-E4))))
```

Specification: `inst.lst` in

- We generate code automatically from `inst.lst`:
  - *Functions* to perform ***instruction decoding***.
  - *Functions* to ***dispatch control to semantic functions***.
  - *Functions* and *theorems* to ***verify Centaur's implementation***.
    - Pick a candidate family of instructions.

New

Specification: `inst.lst` in

- We generate code automatically from `inst.lst`:
  - *Functions* to perform ***instruction decoding***.
  - *Functions* to ***dispatch control to semantic functions***.
  - *Functions* and *theorems* to ***verify Centaur's implementation***.
    - Pick a candidate family of instructions.

- Easy to add support for new instructions:
  - **Decoding:** add an entry to `inst.lst`.
  - **Concrete/Symbolic Execution:** list the appropriate semantic function in that entry.

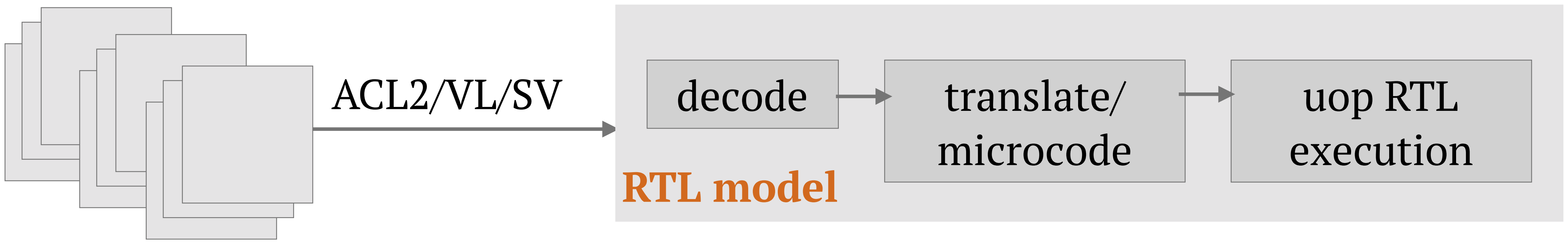
New

Implementation: CenTaur Technology Design

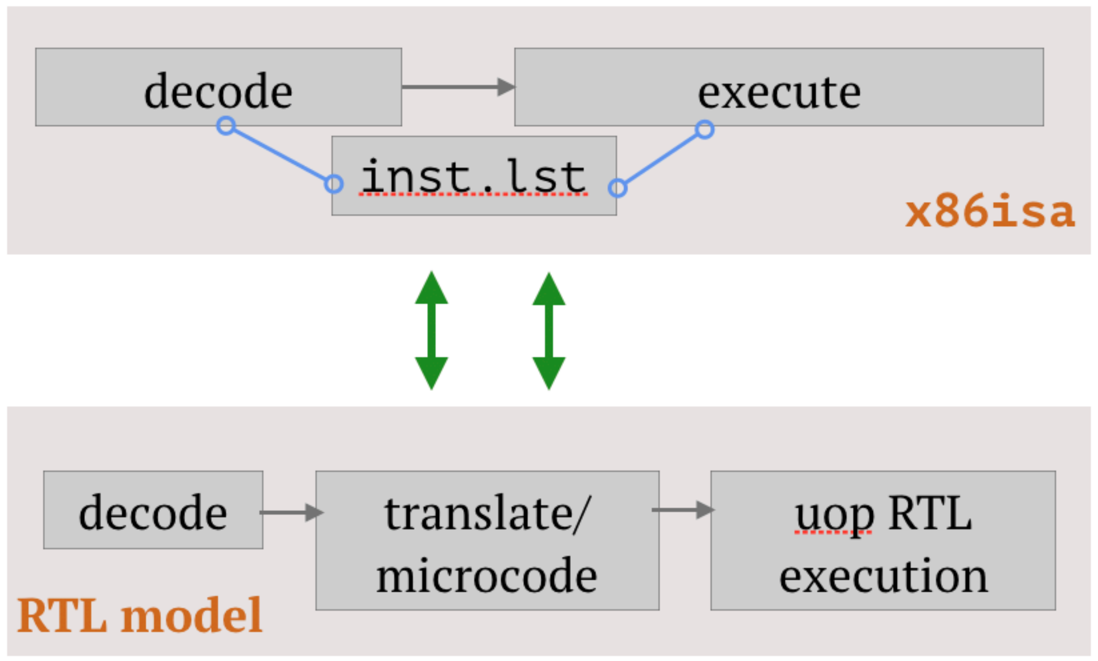
We obtain a model of the RTL (from SystemVerilog source) in ACL2 using ACL2's VL/SV libraries:

- **VL:** parse SystemVerilog code into an ACL2 representation.
- **SV:** assign semantics to the SystemVerilog code.

SystemVerilog RTL Definition



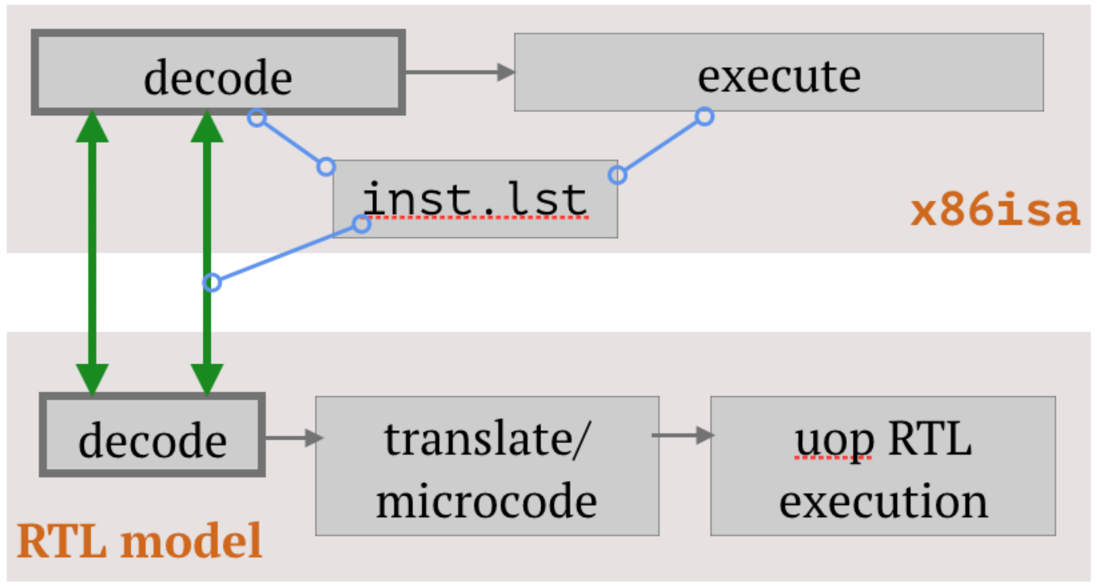
Proof Methodology



- **Goal:** prove that the RTL model is consistent with `x86isa`.
- **Strategy:** Reduce the problem into proving ***three main kinds of component lemmas***.

- Use GL library in ACL2 to ***translate each lemma into a propositional formula for bit-blasting*** using SAT, BDDs, and AIG rewriting.
- Each kind of lemma may require further decomposition.

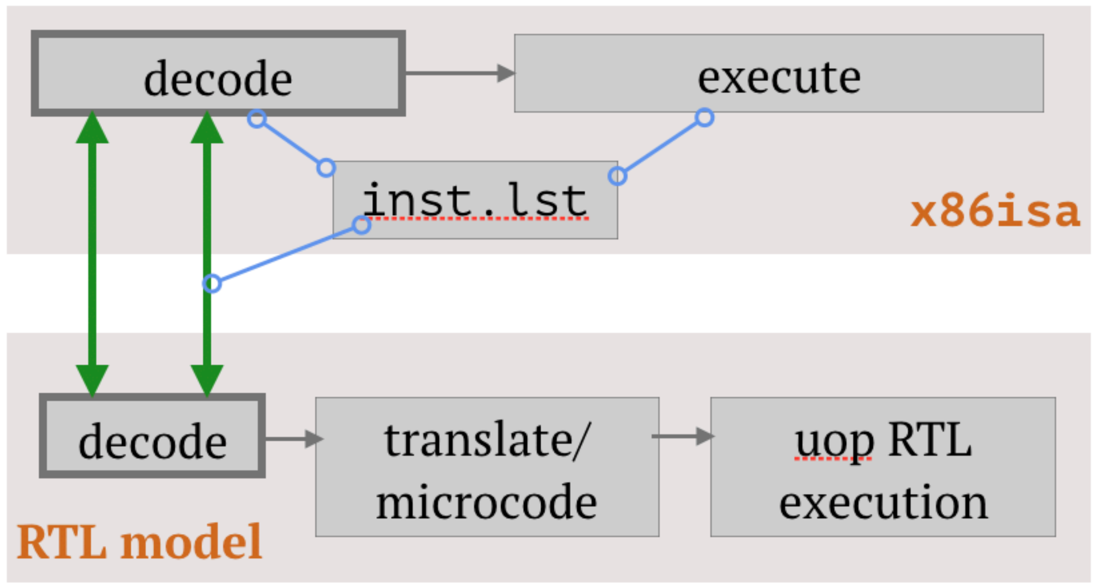
Lemma 1: Decode



- **Inputs:** a byte sequence and the current x86 configuration.
- **Outputs:** either a well-formed x86 instruction (correlated to `inst.lst`) or an exception.

New

Lemma 1: Decode

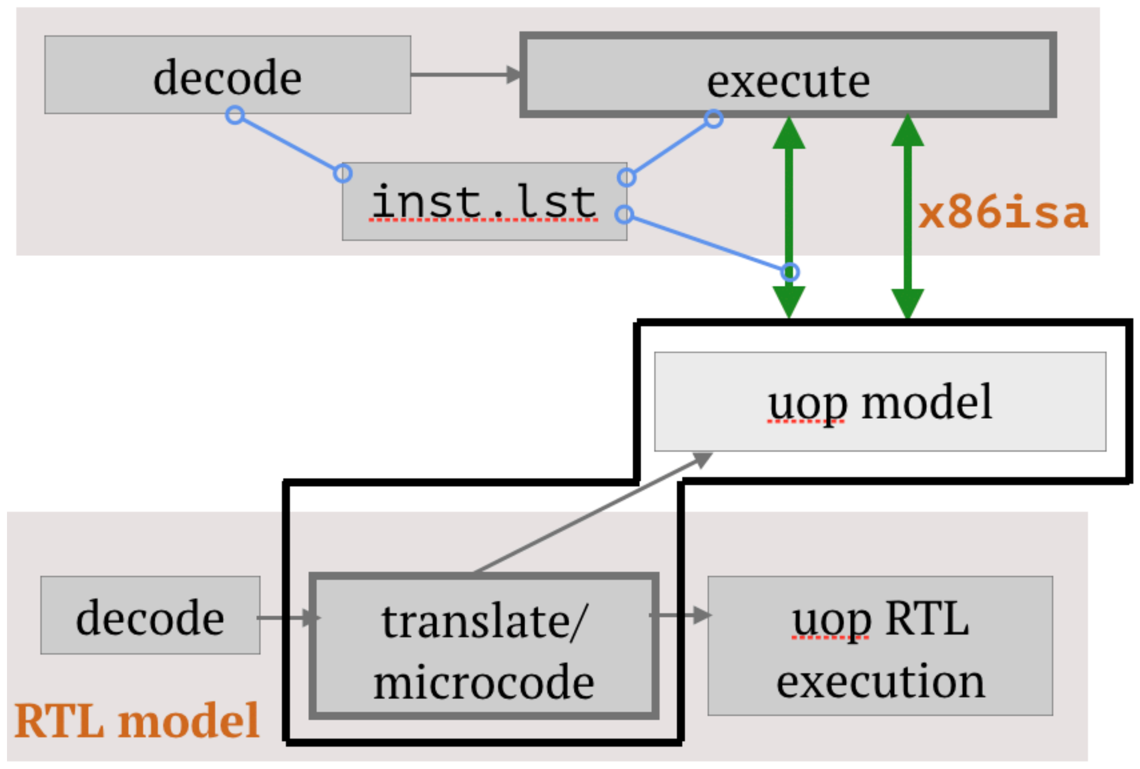


- **Inputs:** a byte sequence and the current x86 configuration.
- **Outputs:** either a well-formed x86 instruction (correlated to `inst.lst`) or an exception.

- **Proof Goal:** `x86isa` and RTL decode functions are consistent.
- **Decomposition:** reduction by cases due to:
  - Parsing of the byte sequence (e.g., instruction prefixes)
  - Configuration for exception generation drawn from `inst.lst`

New

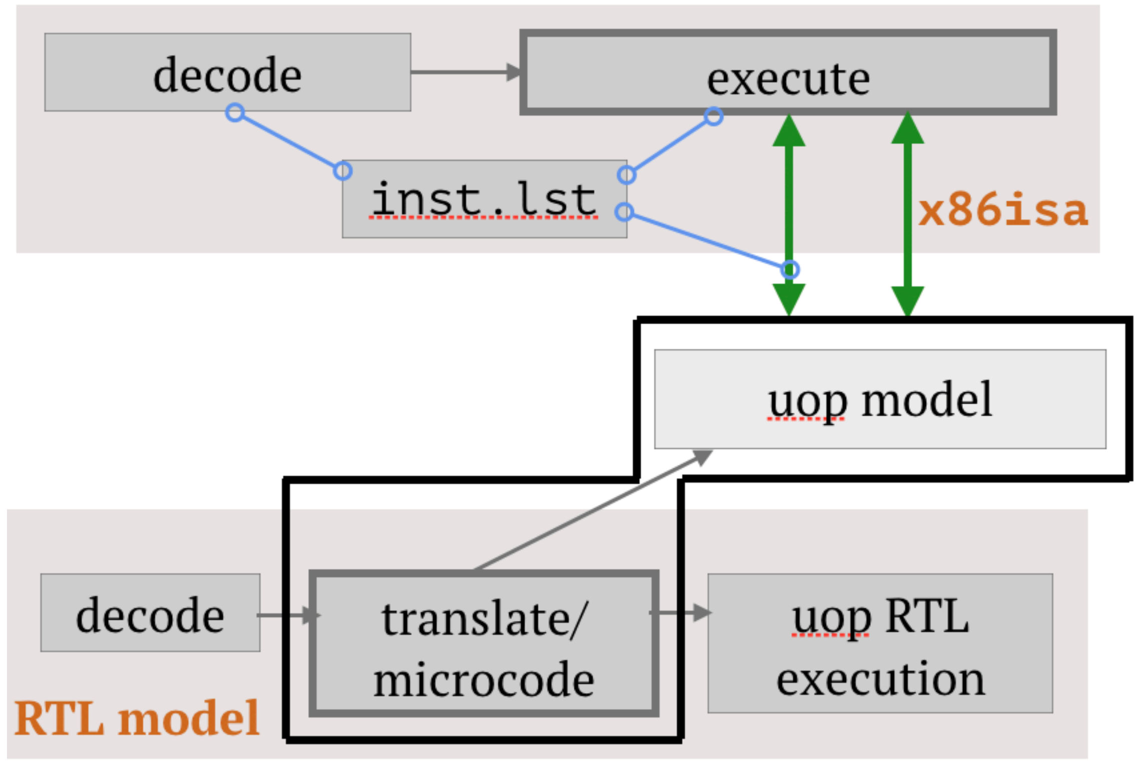
Lemma 2: Translation & Microcode



- **Inputs:** a well-formed x86 instruction (correlated to `inst.lst`) and the current x86 configuration.
- **Outputs:** a sequence of uops implementing the instruction.

New

Lemma 2: Translation & Microcode

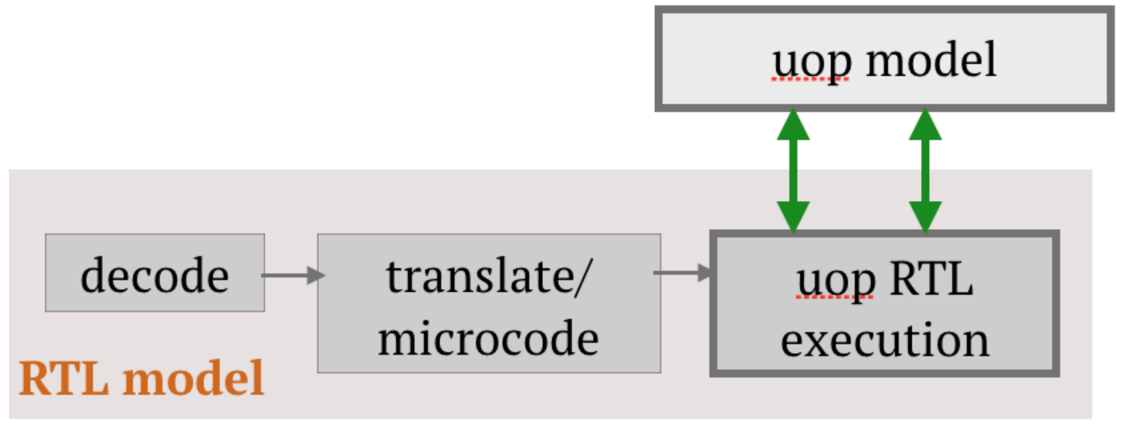


- **Inputs:** a well-formed x86 instruction (correlated to `inst.lst`) and the current x86 configuration.
- **Outputs:** a sequence of uops implementing the instruction.

- **Proof Goal:** execution of the generated uop sequence results in a state consistent with instruction execution.
- **Decomposition:** split proof for complex instructions that have:
  - Longer microcode programs
  - Special input/data cases

New

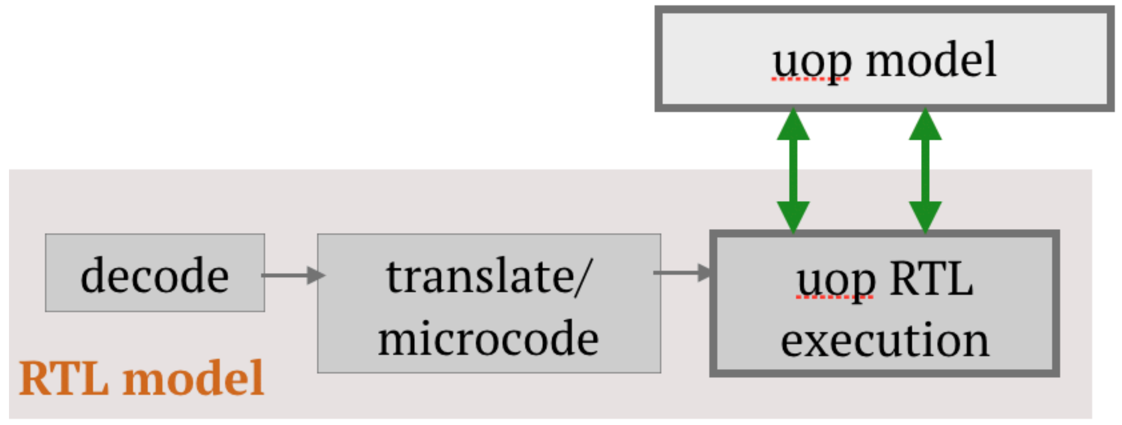
Lemma 3: Uop Execution



- **Inputs:** uop to execute and data.
- **Outputs:** result from uop execution.

Prior work

Lemma 3: Uop Execution



- **Inputs:** uop to execute and data.
- **Outputs:** result from uop execution.

- **Proof Goal:** the RTL execution units produce a computational result consistent with the specification used in the uop model.
- **Decomposition:** split proof for uops that have special input/data cases (e.g., near/far paths for FP adders).

Prior work

Conclusion

`x86isa` and the uop model:

- Provide ***specifications*** for the verification of the hardware design.
- Are leveraged for ***proof decomposition*** and ***lemma generation***.
- ***Reduce inputs required from the users***:
  - E.g., automatically derive constraints for legal instantiations of an instruction, given just the mnemonic and configuration.

Conclusion

`x86isa` and the uop model:

- Provide ***specifications*** for the verification of the hardware design.
- Are leveraged for ***proof decomposition*** and ***lemma generation***.
- ***Reduce inputs required from the users***:
    - E.g., automatically derive constraints for legal instantiations of an instruction, given just the mnemonic and configuration.

Along with formal verification of ***execution units***, it is entirely feasible to verify ***single-instruction execution of processor front-ends***.

Future Work/WIP

- **Automation:**
  - Automatically prove the ***correctness of simple instructions***.
    - Complex ones may require some manual intervention/guidance.
  - Automatically check that ***component lemmas cover all possible cases***.

Future Work/WIP

- **Automation:**
  - Automatically prove the ***correctness of simple instructions***.
    - Complex ones may require some manual intervention/guidance.
  - Automatically check that ***component lemmas cover all possible cases***.

- **Symbolizing Inputs:**
  - Allowing more ***symbolic fields*** in our inputs.
    - Coalesce many ***similar instruction invocations in one proof***.
  - E.g., the proofs for an AVX512 instruction that supports both masking modes can be combined.

Using x86isa for Microcode Verification

Shilpi Goel & Rob Sumners
{shilpi,rsumners}@centtech.com

Questions?

SpISA 2019