Automated Synthesis of Symbolic Instruction Encodings from I/O Samples

Patrice Godefroid
Microsoft Research

Ankur Taly
Stanford University

Need for Symbolic Instruction Encodings

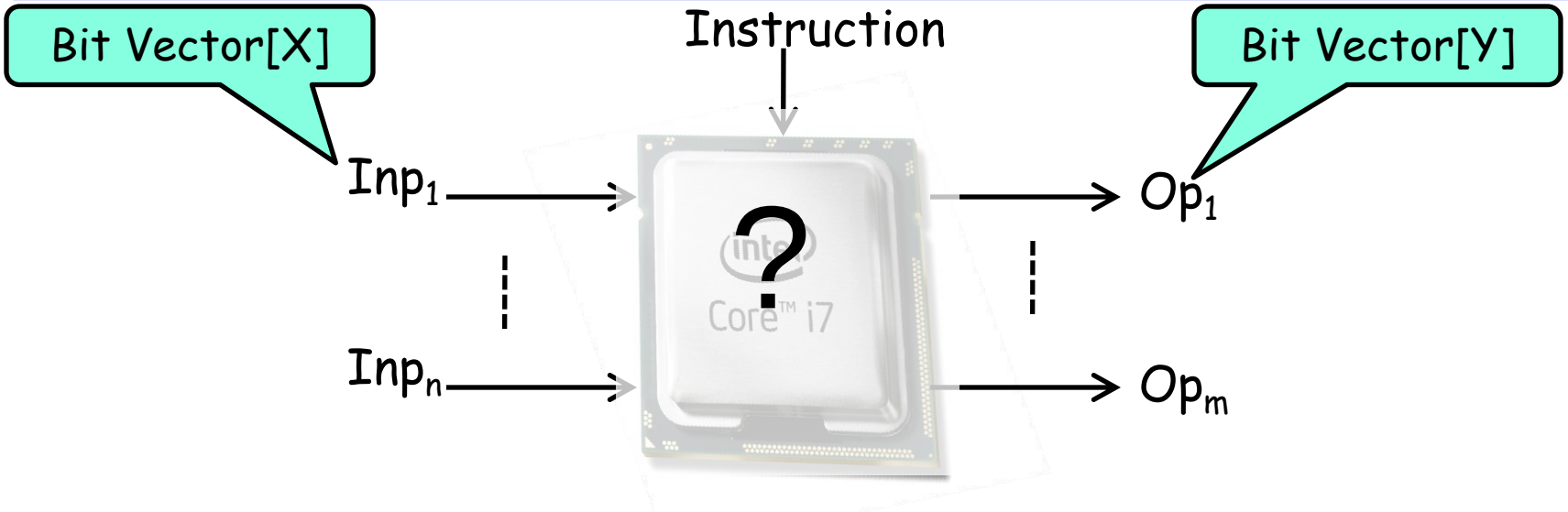
```
l1:mov eax, inp1
   mov cl,  inp2
   shl eax, cl
   jnz l2
   jmp          l3
l2: div         ebx, eax
// Is this safe ?
// Is eax != 0 ?
l3: …
```

Symbolic Execution is a key component of precise binary program analysis tools

- SAGE, BitBlaze, BAP, etc.

- Static analysis tools

Problem: Symbolic Instruction Encoding



Problem: Given a processor and an instruction name, symbolically describe the input-output function for the instruction

- Express the encoding as bit-vector constraints
  (ex: SMT-Lib format)

So far, only manual solutions…

- From the instruction architecture manual (X86, ARM, …) implemented by the processor

SHLD—Double Precision Shift Left (Continued)

Operation

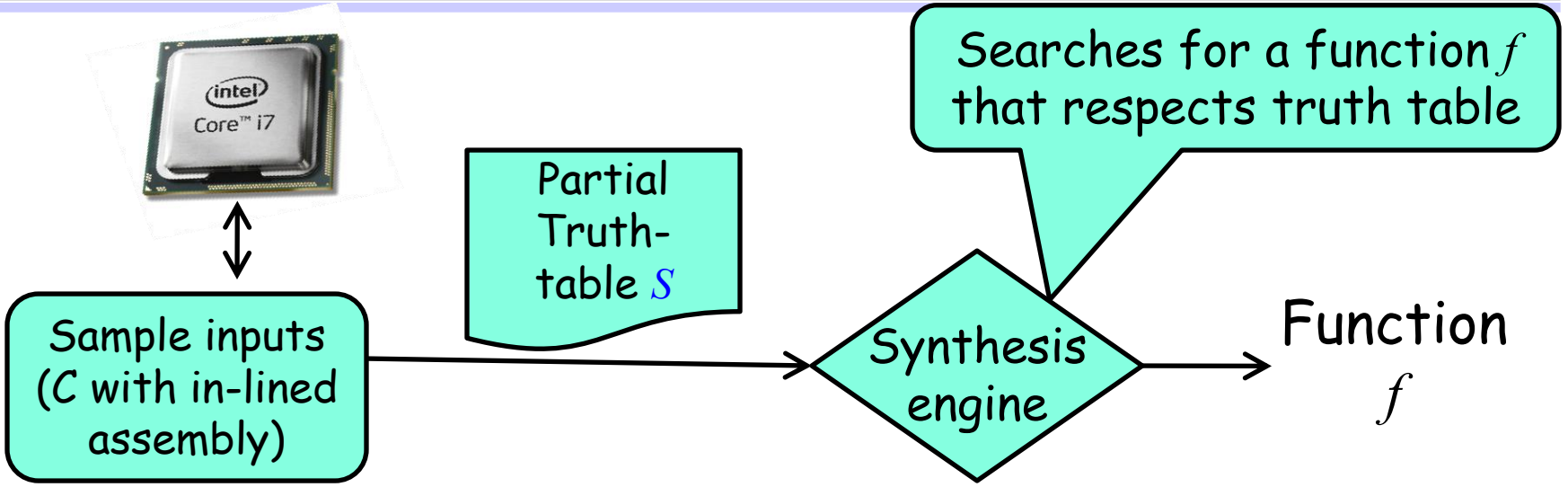
```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
    THEN
        no operation
    ELSE
        IF COUNT ≥ SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, SIZE – COUNT];
                (* Last bit shifted out on exit *)
                FOR i ← SIZE – 1 DOWNTO COUNT
                DO
                    Bit(DEST, i) ← Bit(DEST, i – COUNT);
                OD;
                FOR i ← COUNT – 1 DOWNTO 0
                DO
                    BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
                OD;
        FI;
FI;
```

X86 spec for SHLD

Limitations:

- Tedious, expensive
  - X86 has more than 300 unique instructions, each with ~10 OPCodes, 2000 pages
- Error-prone
  - Written in English, many corner cases
- Imprecise
  - Spec is often under-specified
- Partial
  - Not all instructions are covered
- Can we trust the spec ?

Here: Automated Synthesis Approach



Goals:

- As automated as possible so that we can boot-strap a symbolic execution engine on an arbitrary instruction set
  - But search spaces are enormous (ex: 2^{2048} 8-bit to 8-bit functions!)
- As precise as possible: f must capture behavior for inputs outside the partial truth table S as well
  - But exhaustive sampling is impossible (32x32bits = 2^64 inputs!)

Challenge: Enormous Search Space

$$\exists f: \Lambda_{i,o \in S}\ o = f(i) \quad \textit{(Higher-order quantifier)}$$

How can we reduce the search space ?

Solution: Templates (Program Sketching, Oracle-guided component synthesis)

- A template is a parametric function $T(c_1,\ldots,c_n,i,o)$ with certain unknown parameters/coeffs $c_1,\ldots,c_n$
- A concretization of the template is obtained by substituting specific values for the coefficients
- Restrict the search space to all possible concretizations of $T(c_1,\ldots,c_n,i,o)$

$$\exists c_1,\ldots,c_n: \Lambda_{i,o \in S}\ T(c_1,\ldots,c_n,i,o) \quad \textit{(First-order quantifier)}$$

Warning: this fails if template cannot express the actual function

Designing Templates

Design Principles

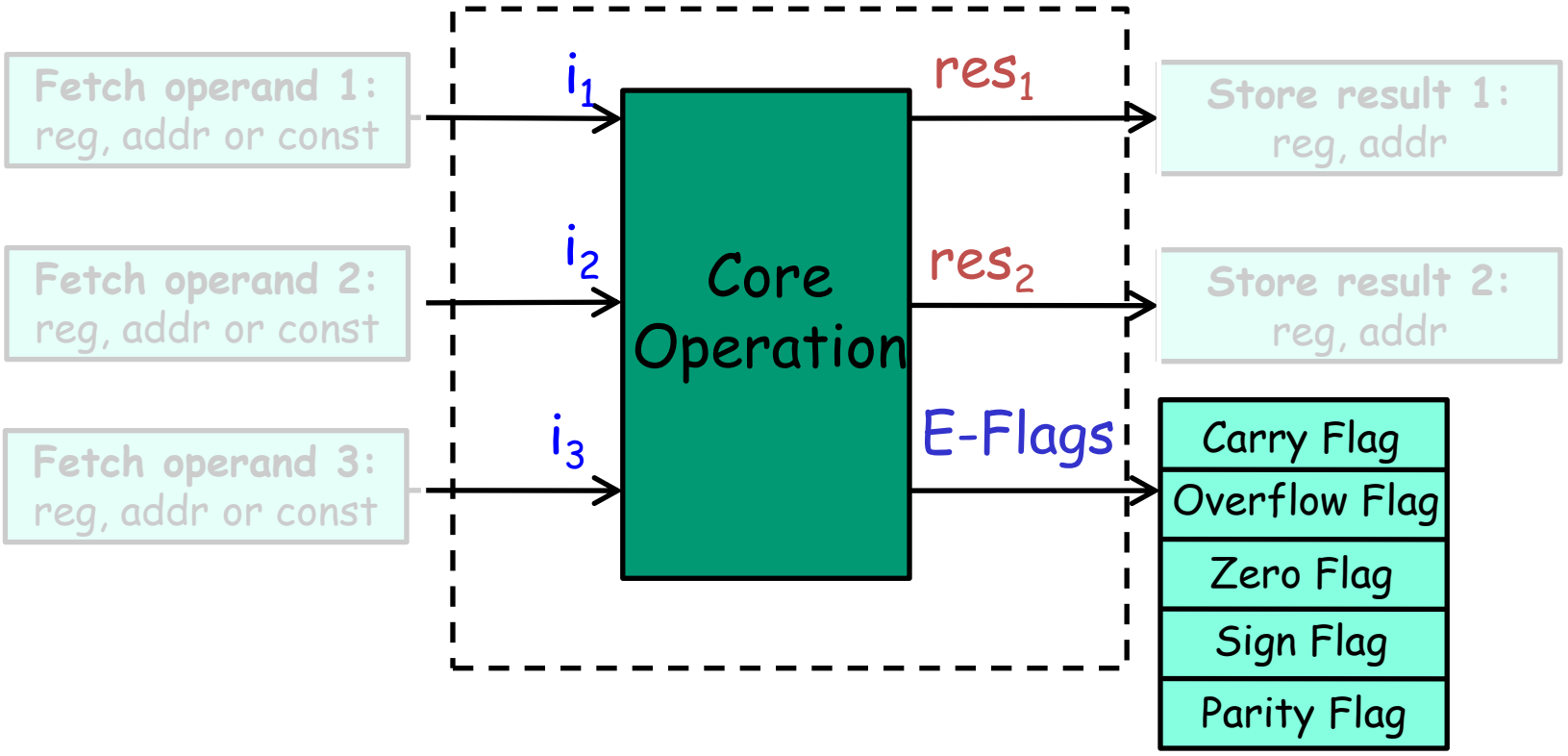
- Template $T(c_1,\ldots,c_n,i)$ must be expressible using bit-vector constraints (for compactness requirements)
- Must capture the common structure of a set of instructions
  - A template abstracts a set of concrete instructions
- Must not have too much freedom => enormous search space
- Must not have too little freedom => cover too few instructions

Architecture-specification is useful

- help in grouping instructions based on similar behaviors
- help in capturing the common structure

Intel X86 Instruction Set

- Complex Instruction-set Architecture (CISC)
  - 300+ unique instructions, each with ~10 OPCodes

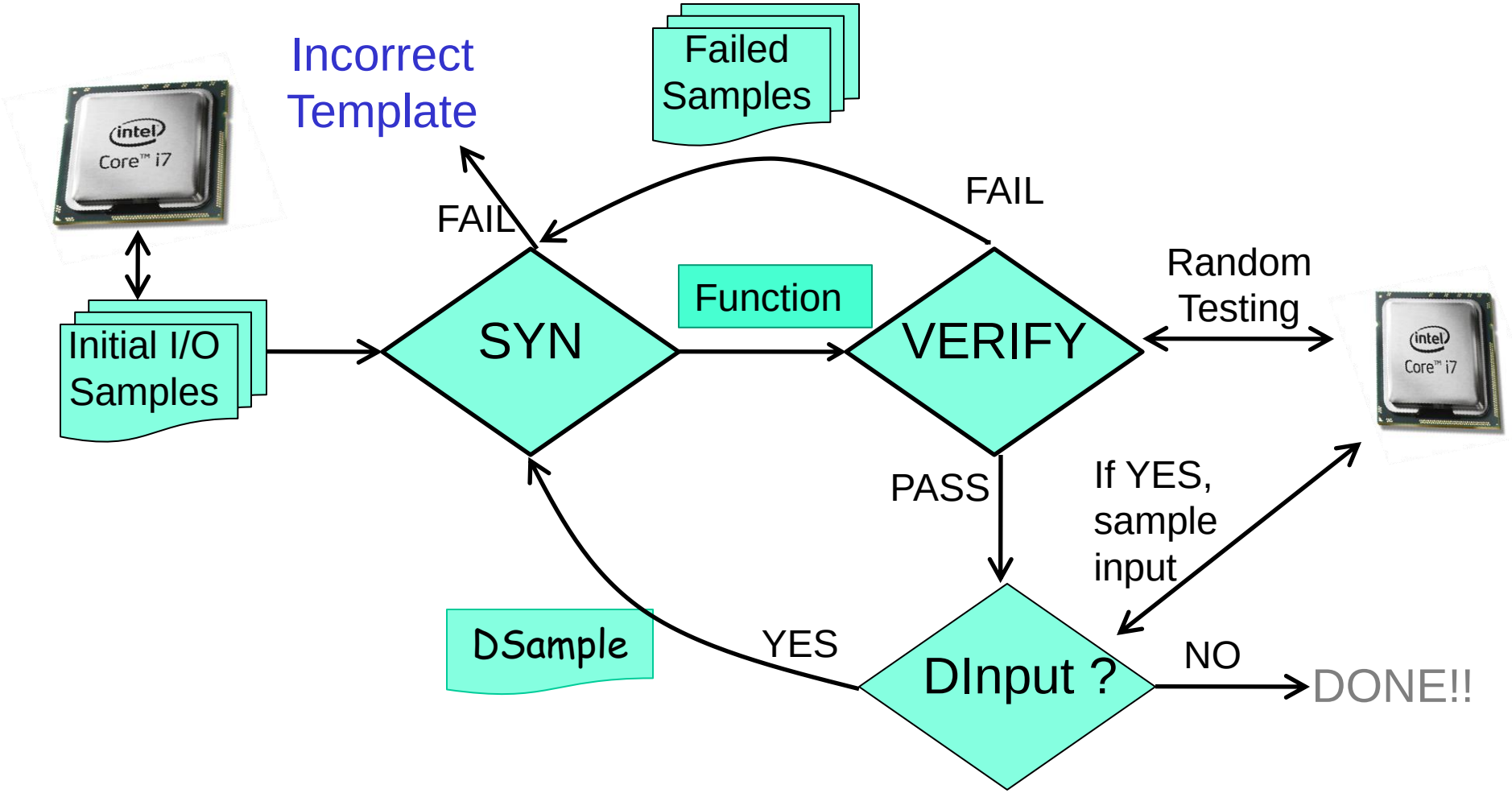


- Assumption: behavior is independent of where the operands come from
- We want a symbolic function from i_1, i_2, i_3 to res_1, res_2 and the E-FLAGS

This Work: ALU Instructions from X86

- Why ALU? Current bit-vector solvers provide the necessary building blocks
- 46 relevant unique instructions (irrelevant instr: `MOV, LOAD, …`)
  - Each has approx. 6 to 21 instances (8/16/32 bits, 2Result + 5 EFLAGS)
- Based on the spec, we divide ALU instructions into 3 groups:
  - Bit-shift instructions (BS): `SHL, SHR, ROL, …`
  - Bit-wise instructions (BW): `AND, OR, NOT, …`
  - Arithmetic instructions (ARI): `ADD, MUL, IMUL …`
- We define 2 templates (Result + EFlags) for each group
  - templates are parametric on the register size (8/16/32)
  - In total 3*2 = 6 templates to cover 534 ALU instruction instances !

State-of-the-art: Distinguishing Input Synthesis



[Jha-Gulwani-Seshia-Tiwari, ICSE'2010]

Problem: too slow ! (or OOM)

Intel XEON 3.07ghz processor, 8GB RAM

Instruction	n_{syn}	n_{ver}	S-Iters	D-Iters	Time(ms)
SHL32	10	100	31	4	24,168,853
	10	1000	31	3	20,107,259
	10	10000	31	1	11,754,805
	100	100	21	3	16,877,223
	100	1000	22	3	17,577,444
	100	10000	20	4	21,620,686
	1000	100	1	1	4,382,472
	1000	1000	1	1	4,456,942
	1000	10000	1	1	4,707,855
	10000	100	Z3 runs out of memory in the DInput phase		
	10000	1000			
	10000	10000			

New Approach: Smart Sampling

- The distinguishing-input check is expensive, can we eliminate it?
- Intuition:
  - 2 points are enough to uniquely determine coefficients of a linear template,
  - 3 points are enough for a circle template
  - …

Smart Inputs: A set of inputs I is said to be smart for a template T if for all samples obtained using the inputs, there exists a unique coefficient up to logical equivalence, for which the template respects the samples

Ex: there are 16 bitwise operations (functions from 1x1 bits to 1 bit)

What are the smart inputs?
Answer: Inputs must have 4 bit-wise pairs (0,0),(0,1),(1,0),(1,1)

0	0	0	0	1	1	0	0	=12
0	0	0	0	1	0	1	0	=10

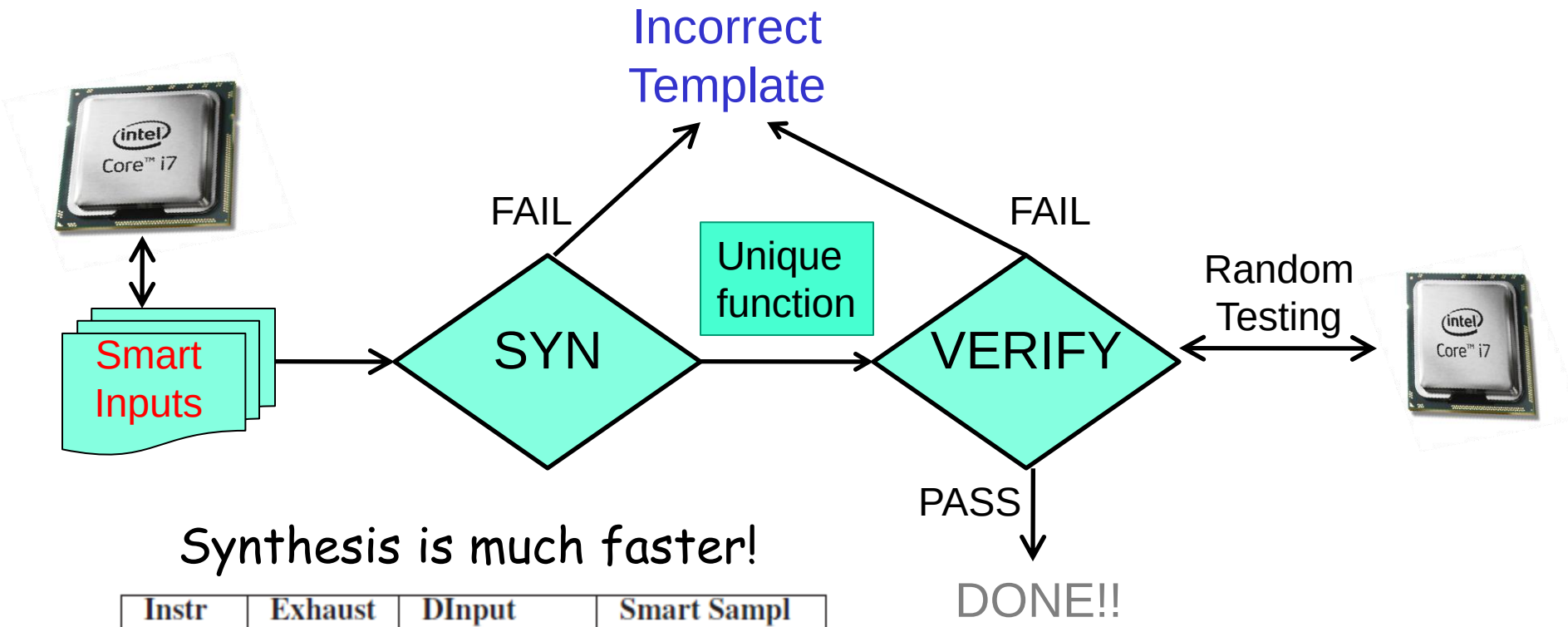
Smart Inputs for Bit-wise template is the singleton (12,10) !

Templates Summary

- n is the size of the input and output bit-vectors (8, 16, 32)

Template	Search space	Smart sample size	Circuit size [RESULT]	Circuit size [EFLAGS]
Bit-shift	$(2n+2)^{32n}$	$32 \quad (\log(n)+2)$	$O(n)$	$O(1)$
Bit-wise	16	1	$O(1)$	$O(1)$
Arithmetic	$21 \quad 2^{2n}$	3	$O(1)$	$O(1)$

Synthesis with Templates and Smart Sampling



Synthesis is much faster!

Instr	Exhaust	DInput	Smart Sampl
AND8	26,478	48 (÷551)	3 (÷16)
AND16	-	55	4 (÷14)
AND32	-	71	4 (÷18)
MUL8	32,462	189 (÷172)	17 (÷11)
MUL16	-	609	20 (÷30)
MUL32	-	1,997	29 (÷68)
SHL8	181,857	21,501 (÷9)	867 (÷25)
SHL16	-	250,105	8,064 (÷31)
SHL32	-	4,382,472	303,970 (÷14)

new "smart sampling" synthesis algorithm takes <2 hours with Z3 to synthesis functions for 534 x86 instruction instances

Lessons Learned

- Uncovered behaviors for "undefined" cases

  Ex: `ADD/SUB`: Overflow Flag (OF)

  - X86 Spec: OF is set "according to the result"
  - Intel XEON3.7: Is set only when XOR of MSB of the two inputs is negation of MSB of output!

- Discrepancies found compared to spec

  Ex: IMUL[8] 65, 254

  - X86 Spec: OF is set to 0
  - Intel XEON3.7: OF is set to 1

Discrepancies Found Across Machines

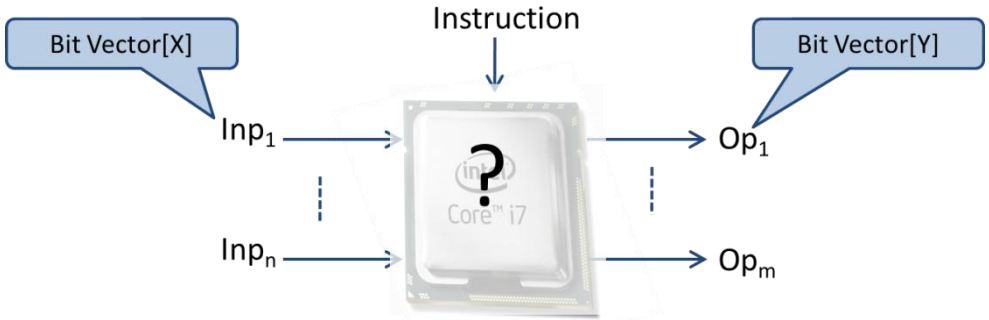
- X86 Spec and Intel XEON3.7 and Core2 (left laptop): instructions ROL, SHL, SHR do not set OF if count argument is not 1
- Intel I7-2620M 2.8ghz (right laptop): OF is set to 1 even for certain cases where count argument > 1

Current Limitations

- Instructions like `CMPXCHG` set EFLAGS according to an intermediate value that is throw away at the end
  - difficult to construct a template for such instructions
- Instructions like `DIV, IDIV` crash on certain inputs (example: when quotient is > register range)
  - these pre-conditions are currently hard-wired in the system
  - in future we would like to synthesize them automatically
- Instruction like `SHL, SHR` leave ZF, PF, SF "unchanged" when count operand = 0
  - therefore ZF, PF, SF must also be inputs to the functions
  - currently we sample all instructions after clearing all flags

Conclusion

- Automated Synthesis of Symbolic Instruction Encodings for X86-ALU instructions
  - 6 abstract instruction templates
  - for 534 x86 ALU instructions (8/16/32bits, outputs, EFLAGS)
  - new "smart sampling" synthesis algorithm takes <2 hours with Z3
  - building blocks are bit-vector constraints (SMT-lib format)
  - synthesis against specific x86 processor as I/O oracle :



- Future work: x64, AMD64, ARM, SIMD instructions, floating point instructions,…

Related Work

- Deriving Abstract Transfer Functions for Embedded CPUs
  - Ex: [HOIST, Regehr et al.]
  - Like us, but small CPUs (8-bits), large encodings (BDDs), abstraction (simplifications -> imprecise)
- Black-box analysis of processors/assemblers
  - **Ex:** [DERIVE, Hsieh-Engler-Back], [Giano, Forin et al.]
  - Emphasis on testing all aspects (addressing modes, clock cycles, privilege levels) of a processor (no symbolic/circuit generation)
- Connection with Machine Learning
  - Close connection between smart inputs for a template and VC dimension of a concept class, to be explored in the future
- Automatic Program Synthesis
  - From I/O examples [Gulwani et al., …], "Program Sketching" ("templates") [Bodik et al., Solar-Lezama et al.,…]
  - Here, new app. domain, smart sampling, verification oracle is a black box