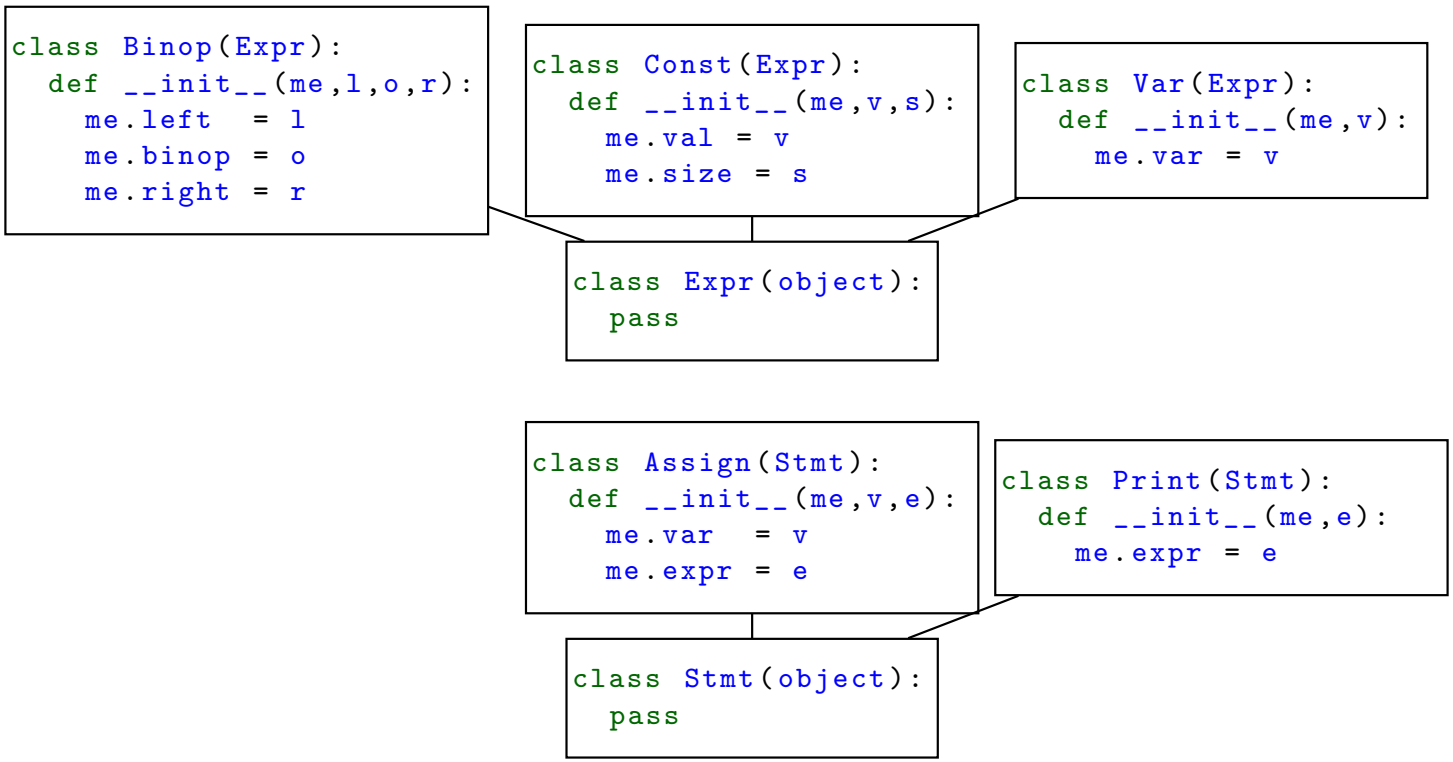
Basics of Analyzing Programs

EvalPrint

- We illustrate basic concepts in the analysis of computer programs using a toy language called EvalPrint.
- The language supports:
  - 16- and 32-bit integer constants and variables.
  - The two binary operators + and ∗.
  - Two statement types:
    1. Assign a value to a variable
    2. Print the value of an expression
- This program prints 3:

```
y.32 = 1.32; // Assign 1 to y
x.32 = 1.32 + (2.32 * y.32);
print x.32; // Print the value of x
```

Class Hierarchy



expr	:=	Binop	expr	binop expr
	|	Const	int	size
	|	Var	var	
stmt	:=	Assign	var	expr
	|	Print	expr	

- The class hierarchy mirrors the grammar exactly.

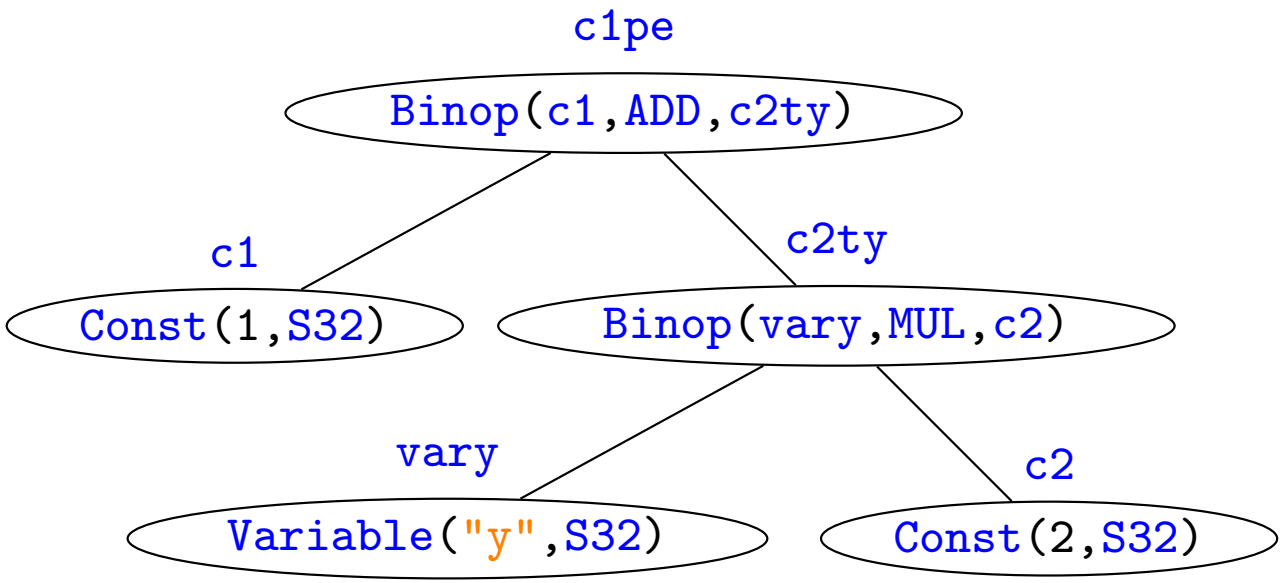
Abstract Syntax Tree

```
1.32 + (y.32 * 2.32);
```

Parser

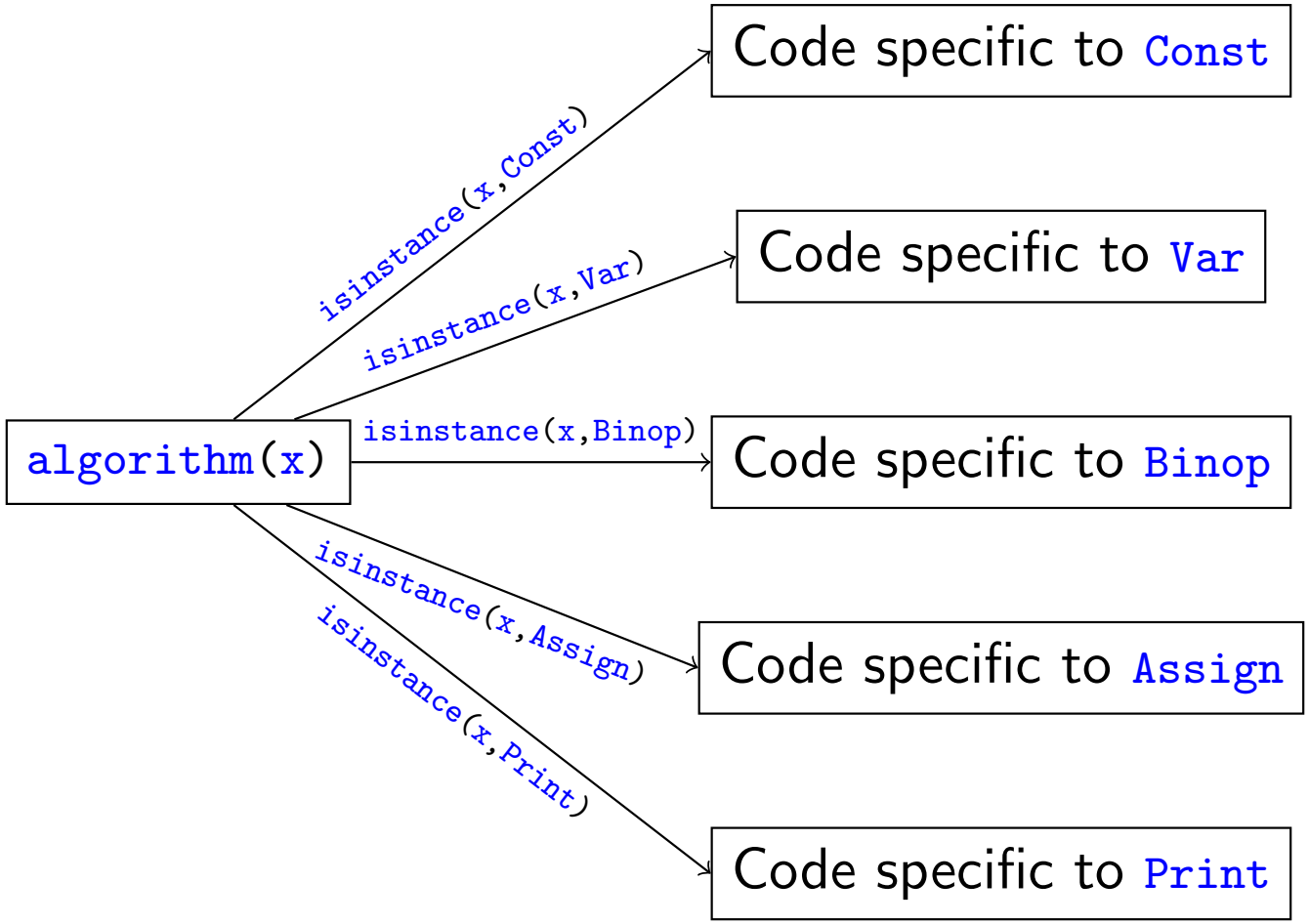
```
c2    = Const(2,S32)
vary  = Variable("y",S32)
c2ty  = Binop(vary,MUL,c2)

c1    = Const(1,S32)
c1pe  = Binop(c1,ADD,c2ty)
```



- The data structure returned by a parser is commonly called an **abstract syntax tree** representation of the program.

Program Analysis in Practice



- Program analysis algorithms usually have separate logic for each grammatical element. We show several useful examples.
  - Interpreter
  - Type-checker

Propositional Logic

Correspondence Between C and Propositional Logic

Most of the concepts from propositional logic have direct analogues in common programming languages such as C.

C concept	Propositional logic concept
boolean variables `a`,`b`,`c`,...	propositional variables $a, b, c, \ldots$
constants `false`, `true`	truth values $\bot, \top$
type, e.g. `bool`	sort, e.g. Boolean
logical negation operator `!`/`~`	negation connective $\neg$
AND operator `&&`	conjunction connective $\wedge$
OR operator `||`	disjunction connective $\vee$
`a ? b : true`	implication connective $a \Rightarrow b$
equals operator `==`	if-and-only-if connective $\Leftrightarrow$
ternary operator `b ? c : d`	if-then-else operator $\mathrm{ITE}(b, c, d)$

Solution Space for a Propositional Formula

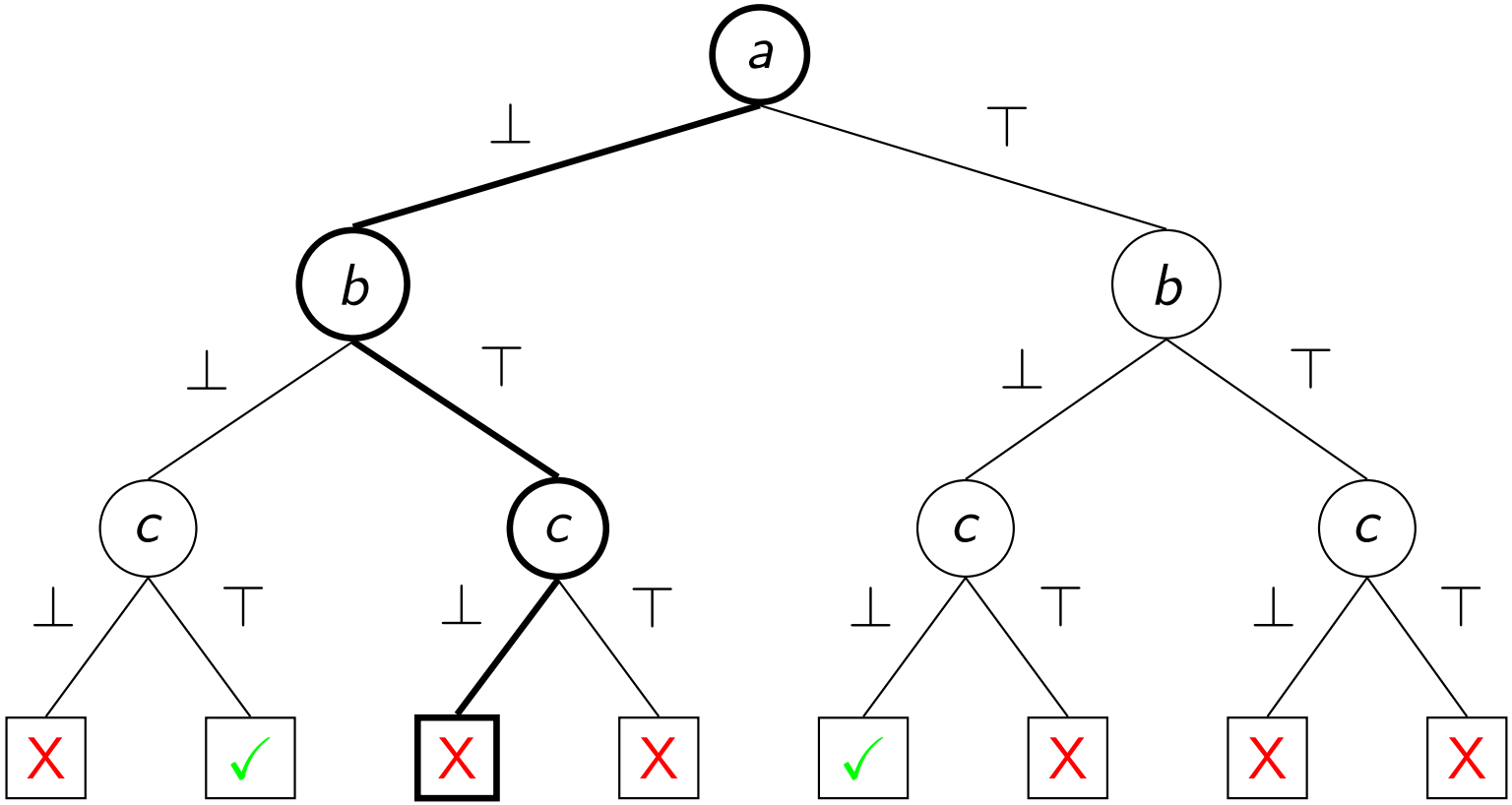


Figure: Solutions to $(a \Rightarrow b) \Leftrightarrow (\neg b \wedge c)$

- The **thick path** corresponds to the assignment
  - $[a \mapsto \bot, b \mapsto \top, c \mapsto \bot]$

Satisfiability (SAT) Solvers

- **SAT solver**, (n). An algorithm that finds **satisfying assignments** to **propositional formulae**, and reports **unsatisfiability** if no such **assignment** is possible.
  - Also known as a **decision procedure** for propositional logic.

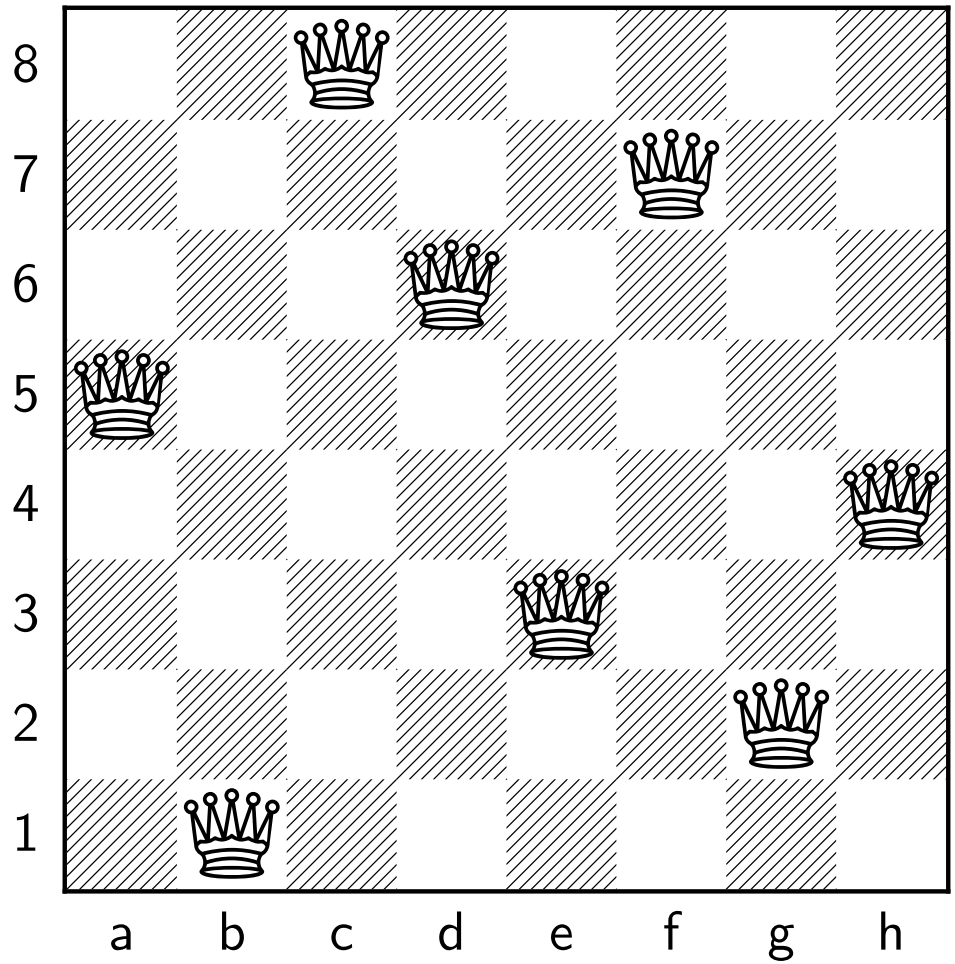
Solving SAT with Truth Tables

- Enumerate every variable assignment and stop when one causes the formula to evaluate to true.
- Corresponds to traversing each path in the solution tree.

```
void sat(Formula *phi)
{
  Assignment cur = /*all false*/;
  while(!is_last_assignment(cur))
  {
    // You coded evaluate already
    if(evaluate(phi,cur))
    {
      printf("Satisfiable\n");
      print_assignment(cur);
      return;
    }
    cur = next_assignment(cur);
  }
  printf("Unsatisfiable\n");
}
```

Applications of SAT Solvers

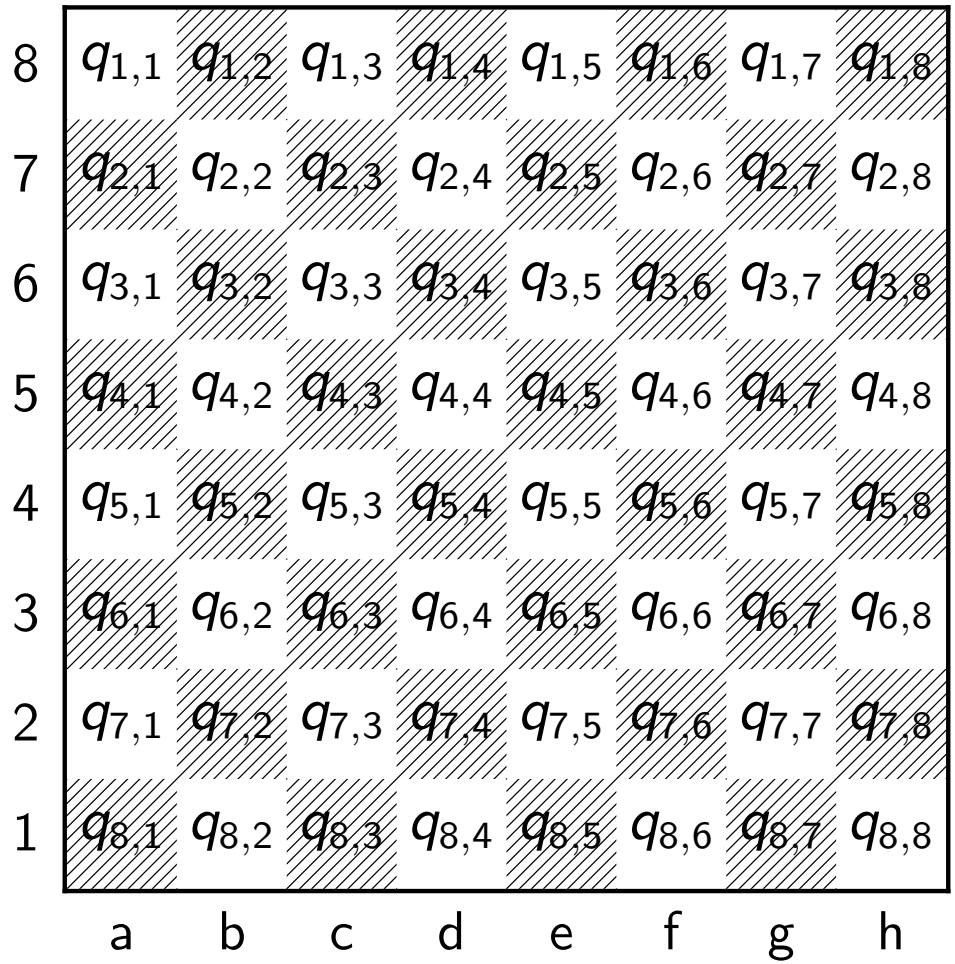
N-Queens Problem



- Can we place N queens on an N $\times$ N chessboard such that none of them attack one another?

Modelling N-Queens with SAT

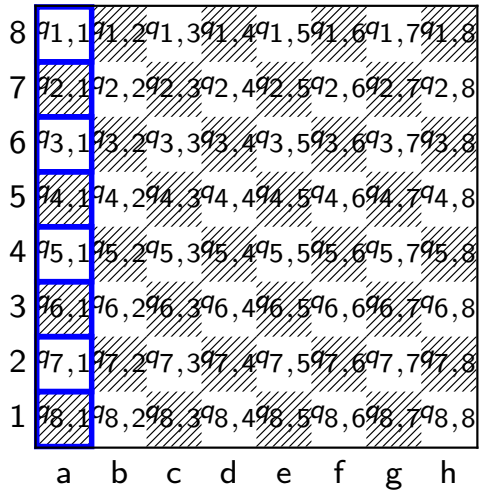
Modelling The Board



- We create one variable per board square.

Modelling N-Queens with SAT

Modelling Column Inhabitance

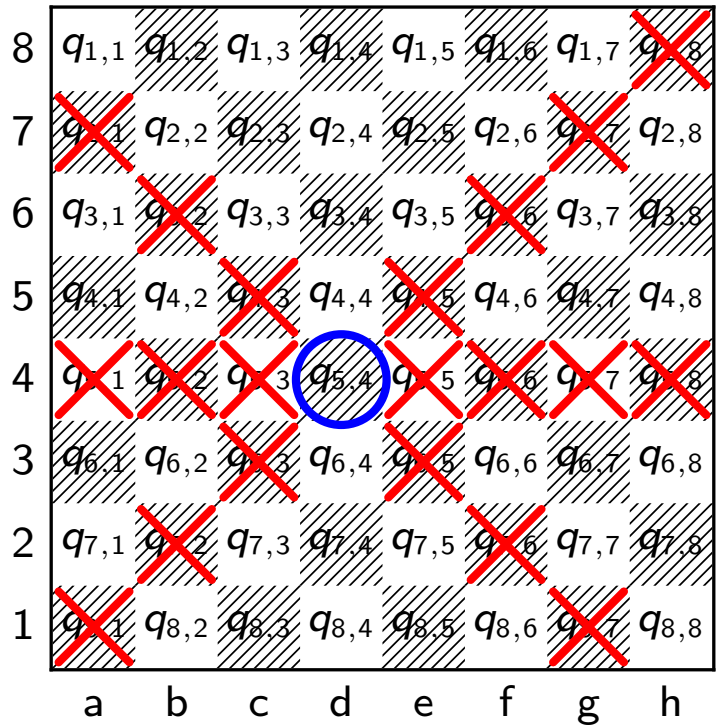


Every column must have at least one queen.

$$
\begin{aligned}
&(q_{1,1} \vee q_{2,1} \vee q_{3,1} \vee q_{4,1} \vee q_{5,1} \vee q_{6,1} \vee q_{7,1} \vee q_{8,1}) \wedge \\
&(q_{1,2} \vee q_{2,2} \vee q_{3,2} \vee q_{4,2} \vee q_{5,2} \vee q_{6,2} \vee q_{7,2} \vee q_{8,2}) \wedge \\
&(q_{1,3} \vee q_{2,3} \vee q_{3,3} \vee q_{4,3} \vee q_{5,3} \vee q_{6,3} \vee q_{7,3} \vee q_{8,3}) \wedge \\
&(q_{1,4} \vee q_{2,4} \vee q_{3,4} \vee q_{4,4} \vee q_{5,4} \vee q_{6,4} \vee q_{7,4} \vee q_{8,4}) \wedge \\
&(q_{1,5} \vee q_{2,5} \vee q_{3,5} \vee q_{4,5} \vee q_{5,5} \vee q_{6,5} \vee q_{7,5} \vee q_{8,5}) \wedge \\
&(q_{1,6} \vee q_{2,6} \vee q_{3,6} \vee q_{4,6} \vee q_{5,6} \vee q_{6,6} \vee q_{7,6} \vee q_{8,6}) \wedge \\
&(q_{1,7} \vee q_{2,7} \vee q_{3,7} \vee q_{4,7} \vee q_{5,7} \vee q_{6,7} \vee q_{7,7} \vee q_{8,7}) \wedge \\
&(q_{1,8} \vee q_{2,8} \vee q_{3,8} \vee q_{4,8} \vee q_{5,8} \vee q_{6,8} \vee q_{7,8} \vee q_{8,8})
\end{aligned}
$$

Modelling N-Queens with SAT

Modelling Attack Constraints

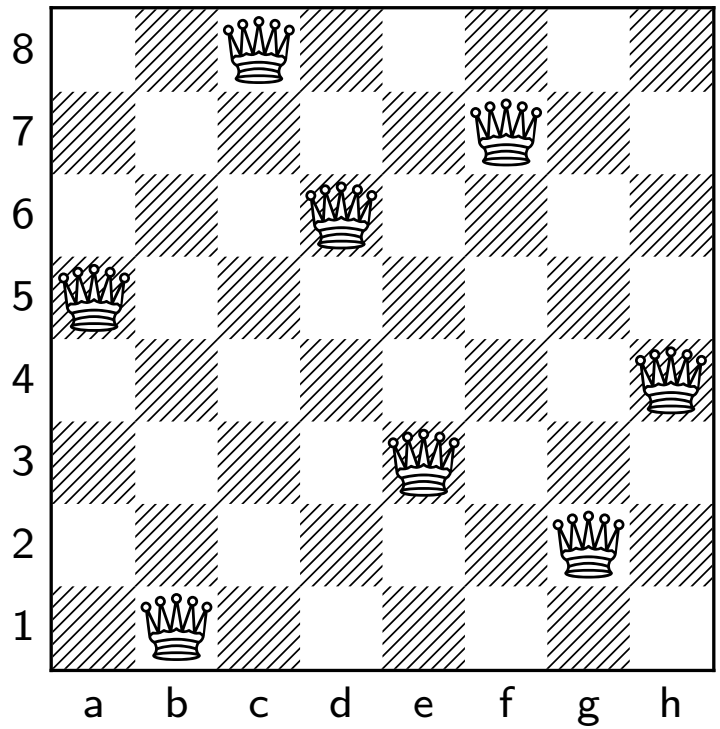


Every square rules out attackable squares.

$$q_{5,4} \Rightarrow \neg(q_{5,1} \vee q_{5,2} \vee q_{5,3} \vee q_{5,5} \vee q_{5,6} \vee q_{5,7} \vee q_{5,8} \vee q_{4,3} \vee q_{3,2} \vee q_{2,1} \vee q_{6,3} \vee q_{7,2} \vee q_{8,1} \vee q_{4,5} \vee q_{3,6} \vee q_{2,7} \vee q_{1,8} \vee q_{6,5} \vee q_{7,6} \vee q_{8,7})$$

Modelling N-Queens with SAT

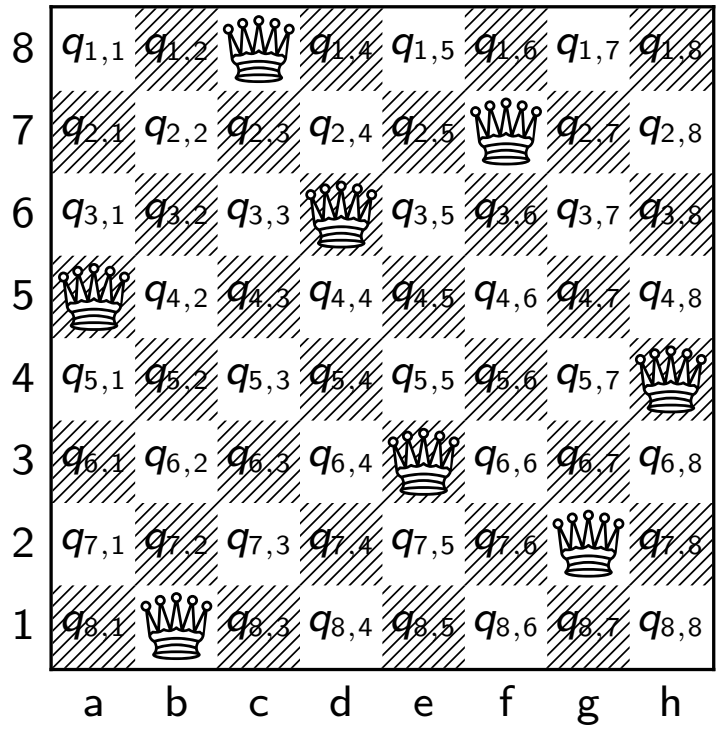
Recap



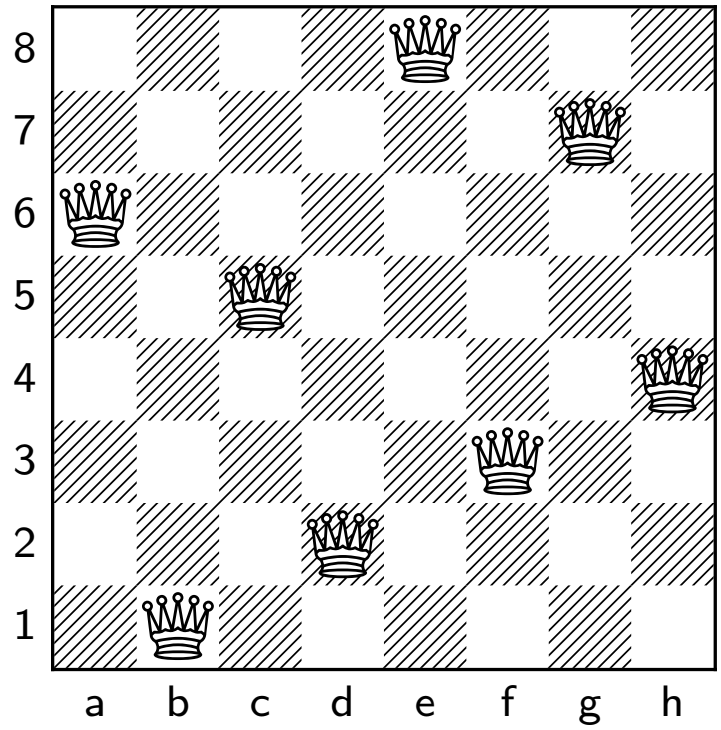
1. Declare variables for each chess square.
2. Assert that every row is inhabited by a queen.
3. Assert that, if a queen is present in a given square, then there can be no queens in any square attackable therefrom.
4. Solve!

Modelling N-Queens with SAT

Obtaining Different Solutions



Old Solution



New Solution

Append to the existing formula a clause asserting that at least one of the pieces is different from the solution obtained previously.

$$\neg(q_{4,1} \wedge q_{8,2} \wedge q_{1,3} \wedge q_{3,4} \wedge q_{6,5} \wedge q_{2,6} \wedge q_{7,7} \wedge q_{5,8})$$

SMT Solvers Overview

- SMT logical theories model computer programming constructs.
- We can convert code directly into SMT formulas.

```
bool b = ((x + y) & z) == 0x5 && array[2] == 0x0
```

```
(= b (and (= (bvand z (bvadd x y)) #x5) (= (select array #x2) #x0))))
```

SMT Solvers Overview

- SMT logical theories model computer programming constructs.
- We can convert code directly into SMT formulas.

```
bool b = ((x + y) & z) == 0x5 && array[2] == 0x0
```

```
(= b (and (= (bvand z (bvadd x y)) #x5) (= (select array #x2) #x0))))
```

- Each construct is modeled by a different **logical theory**.
  - `bool`s and their operations: **propositional logic**

SMT Solvers Overview

- SMT logical theories model computer programming constructs.
- We can convert code directly into SMT formulas.

```
bool b = ((x + y) & z) == 0x5 && array[2] == 0x0
```

```
(= b (and (= (bvand z (bvadd x y)) #x5) (= (select array #x2) #x0))))
```

- Each construct is modeled by a different **logical theory**.
  - bools and their operations: **propositional logic**
  - integers and their operations: **bit-vector logic**

SMT Solvers Overview

- SMT logical theories model computer programming constructs.
- We can convert code directly into SMT formulas.

```
bool b = ((x + y) & z) == 0x5 && array[2] == 0x0
```

```
(= b (and (= (bvand z (bvadd x y)) #x5) (= (select array #x2) #x0))))
```

- Each construct is modeled by a different **logical theory**.
  - bools and their operations: **propositional logic**
  - integers and their operations: **bit-vector logic**
  - equality between variables (==): **equality logic**

SMT Solvers Overview

- SMT logical theories model computer programming constructs.
- We can convert code directly into SMT formulas.

```
bool b = ((x + y) & z) == 0x5 && array[2] == 0x0
```

```
(= b (and (= (bvand z (bvadd x y)) #x5) (= (select array #x2) #x0))))
```

- Each construct is modeled by a different **logical theory**.
  - bools and their operations: **propositional logic**
  - integers and their operations: **bit-vector logic**
  - equality between variables (==): **equality logic**
  - arrays and reading/updating: **array logic**
- SMT solvers allow the user to create and solve formulas with terms from multiple supported theories.

Example SMT Logic: Bit-Vectors

Translation from C

- We can translate integer parts of C programs directly into bit-vector logic.

```
d = a & (b | c);
```

C Statement

```
(assert (= d (bvand a (bvor b c))))
```

Bitvector Logical Statement

Bit-Vector Logic

Translating Bit-Vector Formulas to Propositional Logic

- Since processors operate upon bits internally, it is relatively straightforward to translate machine integer operations into bit-level (i.e., *propositional*) formulas that describe them.

```
(assert (= d (bvand a (bvor b c))))
```

Bitvector Logical Statement

$$
\begin{aligned}
\mathtt{d}_3 &\Leftrightarrow (\mathtt{a}_3 \wedge (\mathtt{b}_3 \vee \mathtt{c}_3)) \wedge \\
\mathtt{d}_2 &\Leftrightarrow (\mathtt{a}_2 \wedge (\mathtt{b}_2 \vee \mathtt{c}_2)) \wedge \\
\mathtt{d}_1 &\Leftrightarrow (\mathtt{a}_1 \wedge (\mathtt{b}_1 \vee \mathtt{c}_1)) \wedge \\
\mathtt{d}_0 &\Leftrightarrow (\mathtt{a}_0 \wedge (\mathtt{b}_0 \vee \mathtt{c}_0))
\end{aligned}
$$

Encoding in Propositional Logic

Bit-Vector Logic

Posing Queries Regarding Code

- We can ask the SMT solver questions about programs.
- Example: are there values of `a`, `b`, and `c` such that `d` is `5`?
  - This question is pronounced: `(assert (= d #x5))`.

$$
\begin{aligned}
\mathtt{d}_3 &\Leftrightarrow (\mathtt{a}_3 \wedge (\mathtt{b}_3 \vee \mathtt{c}_3)) \quad \wedge \\
\mathtt{d}_2 &\Leftrightarrow (\mathtt{a}_2 \wedge (\mathtt{b}_2 \vee \mathtt{c}_2)) \quad \wedge \\
\mathtt{d}_1 &\Leftrightarrow (\mathtt{a}_1 \wedge (\mathtt{b}_1 \vee \mathtt{c}_1)) \quad \wedge \\
\mathtt{d}_0 &\Leftrightarrow (\mathtt{a}_0 \wedge (\mathtt{b}_0 \vee \mathtt{c}_0)) \quad \wedge \\
\neg \mathtt{d}_3 \wedge \mathtt{d}_2 \wedge \neg \mathtt{d}_1 \wedge \mathtt{d}_0 &
\end{aligned}
$$

Previous Propositional Encoding, Plus Query `(assert (= d #x5))`

Bit-Vector Logic

Solving Queries Regarding Code

$$\text{SAT}\left(\begin{array}{ll} d_3 \Leftrightarrow (a_3 \wedge (b_3 \vee c_3)) & \wedge \\ d_2 \Leftrightarrow (a_2 \wedge (b_2 \vee c_2)) & \wedge \\ d_1 \Leftrightarrow (a_1 \wedge (b_1 \vee c_1)) & \wedge \\ d_0 \Leftrightarrow (a_0 \wedge (b_0 \vee c_0)) & \wedge \\ \neg d_3 \wedge d_2 \wedge \neg d_1 \wedge d_0 & \end{array}\right) =$$

- The formula is satisfiable, for instance with:

d_3	$\bot$	a_3	$\bot$	b_3	$\top$	c_3	$\bot$
d_2	$\top$	a_2	$\top$	b_2	$\top$	c_2	$\bot$
d_1	$\bot$	a_1	$\top$	b_1	$\bot$	c_1	$\bot$
d_0	$\top$	a_0	$\top$	b_0	$\bot$	c_0	$\top$
d	5	a	7	b	12	c	1

Propositional Encoding

Addition

Consider decimal addition $z = x + y$, where $x = 954$, and $y = 55$. Let *cin* and *cout* respectively denote "carry-in" and "carry-out".

cout	0	*0*	1	1	0
x	0	0	9	5	4
y	0	0	0	5	5
cin	*0*	1	1	0	**0**
z	0	1	0	0	9

Set $cin_0 = \mathbf{0}$. At each position i, we have that

- $z_i = x_i + y_i + cin_i \mod 10$
- $cout_i = x_i + y_i + cin_i \geq 10 \,?\, 1 : 0$
- $cin_i = cout_{i-1}$

Propositional Encoding

Addition

Consider binary addition $z = x + y$, where $x = 01011010$, and $y = 01101100$. *cin* and *cout* are as before.

cout	0	1	1	1	*1*	0	0	0
x	0	1	0	1	1	0	1	0
y	0	1	1	0	1	1	0	0
cin	1	1	1	*1*	0	0	0	**0**
z	1	1	0	0	0	1	1	0

Set $cin_0 = \mathbf{0}$. At each position i, we have that

- $z_i = x_i + y_i + cin_i \mod 2$
- $cout_i = x_i + y_i + cin_i \geq 2 \,?\, 1 : 0$
- $cin_i = cout_{i-1}$

Propositional Encoding

Addition

Table: The differences between decimal and binary addition

	Base-**10** arithmetic	Base-**2** arithmetic
z_i	$x_i + y_i + cin_i \mod \mathbf{10}$	$x_i + y_i + cin_i \mod \mathbf{2}$
$cout_i$	$x_i + y_i + cin_i \geq \mathbf{10} \,?\, 1 : 0$	$x_i + y_i + cin_i \geq \mathbf{2} \,?\, 1 : 0$
cin_i	$cout_{i-1}$	$cout_{i-1}$
cin_0	0	0

Propositional Encoding

Addition, z_i Term

To encode binary addition in propositional logic, we need propositional encodings for the z_i and $cout_i$ terms.

Table: Truth table for $z_i = x_i + y_i + cin_i \mod 2$

x_i	y_i	cin_i	z_i
0	0	0	0
0	**0**	**1**	**1**
0	**1**	**0**	**1**
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	1	0	0
1	**1**	**1**	**1**

Pattern: the output is 1 if either only one variable is 1, or all three variables are 1. This is an exercise.

Propositional Encoding

Addition, $cout_i$ Term

Table: Truth table for $cout_i = x_i + y_i + cin_i \geq 2 \,?\, 1 : 0$

x_i	y_i	cin_i	$cout_i$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Exercise: fill in the truth table, determine the pattern for when the output is 1, and write a propositional expression.

Propositional Encoding

Addition, All Together

Let $z(x, y, c)$ denote the propositional expression that you devised in a previous exercise for the z_i bit in terms of the x_i, y_i, and cin_i bits, and similarly for $cout(x, y, c)$. The propositional expression for the expression `(= z (bvadd x y))` is then:

$$
\begin{array}{rcrl}
(z_0 & \Leftrightarrow & z(x_0, y_0, \bot)) & \wedge \\
(cout_0 & \Leftrightarrow & cout(x_0, y_0, \bot)) & \wedge \\
(z_1 & \Leftrightarrow & z(x_1, y_1, cout_0)) & \wedge \\
(cout_1 & \Leftrightarrow & cout(x_1, y_1, cout_0)) & \wedge \\
(z_2 & \Leftrightarrow & z(x_2, y_2, cout_1)) & \wedge \\
(cout_2 & \Leftrightarrow & cout(x_2, y_2, cout_1)) & \wedge \\
 & \ldots & & \\
(z_7 & \Leftrightarrow & z(x_7, y_7, cout_6)) & \wedge \\
(cout_7 & \Leftrightarrow & cout(x_7, y_7, cout_6)) &
\end{array}
$$

Applying SMT Solvers to Code

Formal Verification

```
uchar swapbits(uchar b,
    char n, char h, char l)
{
     x = (b>>l)^(b>>h);
     m = (1<<n)-1;
     y = x & m;
     z = (y<<l)|(y<<h);
     return b ^ z;
}
```

- We need two more concepts for posing queries about code sequences.
  - **Preconditions**
  - **Postconditions**
- We shall illustrate them while proving the correctness of the code above, a process called **formal verification**.

Formal Verification

Preconditions

- API documentation for a function often specifies:
  - **Preconditions** that must be true for it to work properly.
  - **Postconditions** that should be true after execution.

```
uchar swapbits(uchar b, char n, char h, char l)
```

- Postcondition: swaps the `n` bits within `b` beginning at position `l` with those at position `h`.
- Preconditions:
  - `n` is positive: can't swap a negative number of bits.
  - `l` is non-negative.
  - `h` is non-negative.
  - `n+h<=8`: the high bits are contained within `b`.
  - `n+l<=h`: the low bits don't overlap the high ones.

Formal Verification

Preconditions

A conscientious developer will check function preconditions.

```
uchar swapbits(uchar b,
    char n, char h, char l)
{
    assert(n > 0);
    assert(l => 0);
    assert(h => 0);
    assert(n+h <= 8);
    assert(n+l <= h);
    // .. rest of code
}
```

To be very precise:

- **Preconditions** are assertions made about the values of variables before some piece of code executes.

Formal Verification

Preconditions

```
uchar swapbits(uchar b,
    char n, char h, char l)
{
    assert(n > 0);
    assert(l => 0);
    assert(h => 0);
    assert(n+h <= 8);
    assert(n+l <= h);
    // .. rest of code
}
```

```
(assert
  (and
    (bvsgt n #x00)
    (bvsge l #x00)
    (bvsge h #x00)
    (bvule (bvadd n h) 8)
    (bvule (bvadd n l) h)
  )
)
```

We can model such preconditions easily in SMT.

Formal Verification

Our Target: `swapbits(b,n,h,l)`

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;
  y = x & m;
  z = (y<<l)|(y<<h);
  return b ^ z;
}
```

`swapbits(b,n,h,l)` swaps within `b` the `n` bits at positions `h` and `l`.

$$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$$

`swapbits(b,2,5,1)`

$$b_7\ b_2\ b_1\ b_4\ b_3\ b_6\ b_5\ b_0$$

We investigate and verify this property.

Formal Verification

`swapbits(b,2,5,1)`, Computation of `x`

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);◀
  m = (1<<n)-1;
  y = x & m;
  z = (y<<l)|(y<<h);
  return b ^ z;
}
```

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
			`...(b>>l)...`				
0	b_7	b_6	b_5	b_4	b_3	b_2	b_1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
			`...(b>>h)...`				
0	0	0	0	0	b_7	b_6	b_5
			`x = (b>>l)^(b>>h);`				
0	b_7	b_6	b_5	b_4	x_2	$\mathbf{x_1}$	$\mathbf{x_0}$

The bottom two bits of `x` are $\mathbf{x_0} = b_1 \otimes b_5$, $\mathbf{x_1} = b_2 \otimes b_6$.

Formal Verification

`swapbits(b,2,5,1)`, Computation of `m`

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;◀
  y = x & m;
  z = (y<<l)|(y<<h);
  return b ^ z;
}
```

0	0	0	0	0	0	0	1
			... (1<<n) ...				
0	0	0	0	0	1	0	0
			m = (1<<n)-1;				
0	0	0	0	0	0	1	1

`m` is a mask for the bottom two bits.

Formal Verification

`swapbits(b,2,5,1)`, Computation of `y`

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;
  y = x & m;◀
  z = (y<<l)|(y<<h);
  return b ^ z;
}
```

				`x`			
0	b_7	b_6	b_5	b_4	x_2	x_1	x_0
				`m`			
0	0	0	0	0	0	1	1
				`y = x & m;`			
0	0	0	0	0	0	x_1	x_0

`y` contains the XORed bits $x_0 = b_1 \otimes b_5$, $x_1 = b_2 \otimes b_6$.

Formal Verification

`swapbits(b,2,5,1)`, Computation of `z`

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;
  y = x & m;
  z = (y<<l)|(y<<h);◀
  return b ^ z;
}
```

			`y`				
0	0	0	0	0	0	x_1	x_0
			`···` `(y<<l)` `···`				
0	0	0	0	0	x_1	x_0	0
			`···` `(y<<h)` `···`				
0	x_1	x_0	0	0	0	0	0
			`z = (y<<l)|(y<<h);`				
0	x_1	x_0	0	0	x_1	x_0	0

`z` contains two copies of x_1 and x_0 at positions j and i.

Formal Verification

`swapbits(b,2,5,1)`, Computation of Return Value

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;
  y = x & m;
  z = (y<<l)|(y<<h);
  return b ^ z;◀
}
```

			`b`				
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
			`z`				
0	x_1	x_0	0	0	x_1	x_0	0
			`return b ^ z;`				
b_7	b_2	b_1	b_4	b_3	b_6	b_5	b_0

Since $x_0 = b_1 \otimes b_5$ and $x_1 = b_2 \otimes b_6$:

- $b_6 \otimes (b_2 \otimes b_6) = b_2$
- $b_5 \otimes (b_1 \otimes b_5) = b_1$
- $b_2 \otimes (b_2 \otimes b_6) = b_6$
- $b_1 \otimes (b_1 \otimes b_5) = b_5$

Formal Verification

Translation to SMT-LIB

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;
  y = x & m;
  z = (y<<l)|(y<<h);
  return b ^ z;
}
```

```
(declare-fun x () (_ BitVec 8))
; Etc. for b,n,h,l,m,y,z,r

(assert
  (and
    (= x (bvxor
      (bvlshr b l)
      (bvlshr b h)))
    (= m (bvsub
      (bvshl #x01 n)
      #x01))
    (= y (bvand x m))
    (= z (bvor
      (bvshl y l)
      (bvshl y h)))
    (= r (bvxor b z))
  )
)
```

Conversion to SMT-LIB is very straightforward.

Formal Verification

Postconditions

Recall our API documentation for `swapbits`.

```
uchar swapbits(uchar b, char n, char h, char l)
```

- Postcondition: swaps the `n` bits within `b` beginning at position `l` with those at position `h`.
- Preconditions: ...

The **postcondition** describes what the code should do. Precisely:

- **Postconditions** are assertions made about the values of variables after some piece of code executes.

We formulate and model postconditions for `swapbits`.

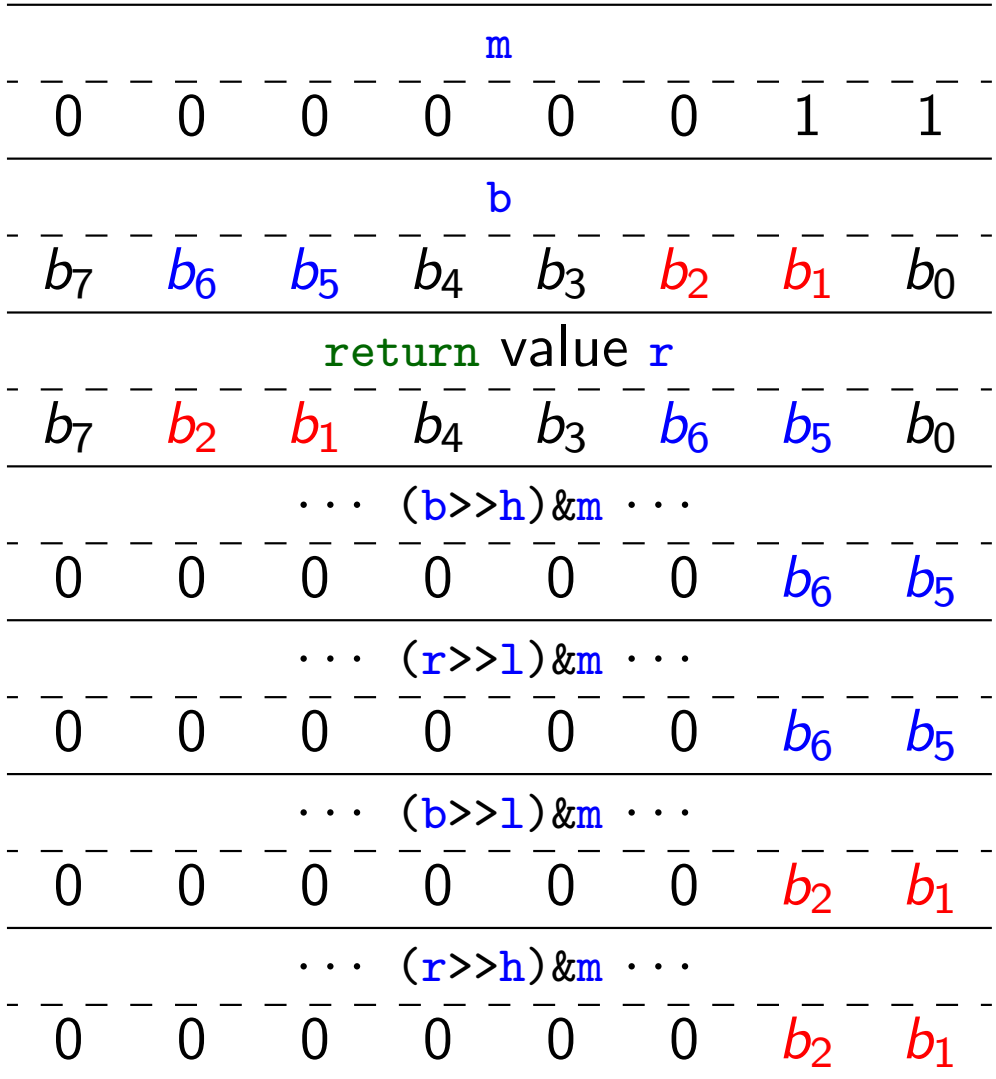
Formal Verification

Postconditions: The Bits Were Swapped

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;
  y = x & m;
  z = (y<<l)|(y<<h);
  return b ^ z;
}
```

The bits were successfully swapped if the following two postconditions are true:

- `(b>>h)&m == (r>>l)&m`
- `(b>>l)&m == (r>>h)&m`

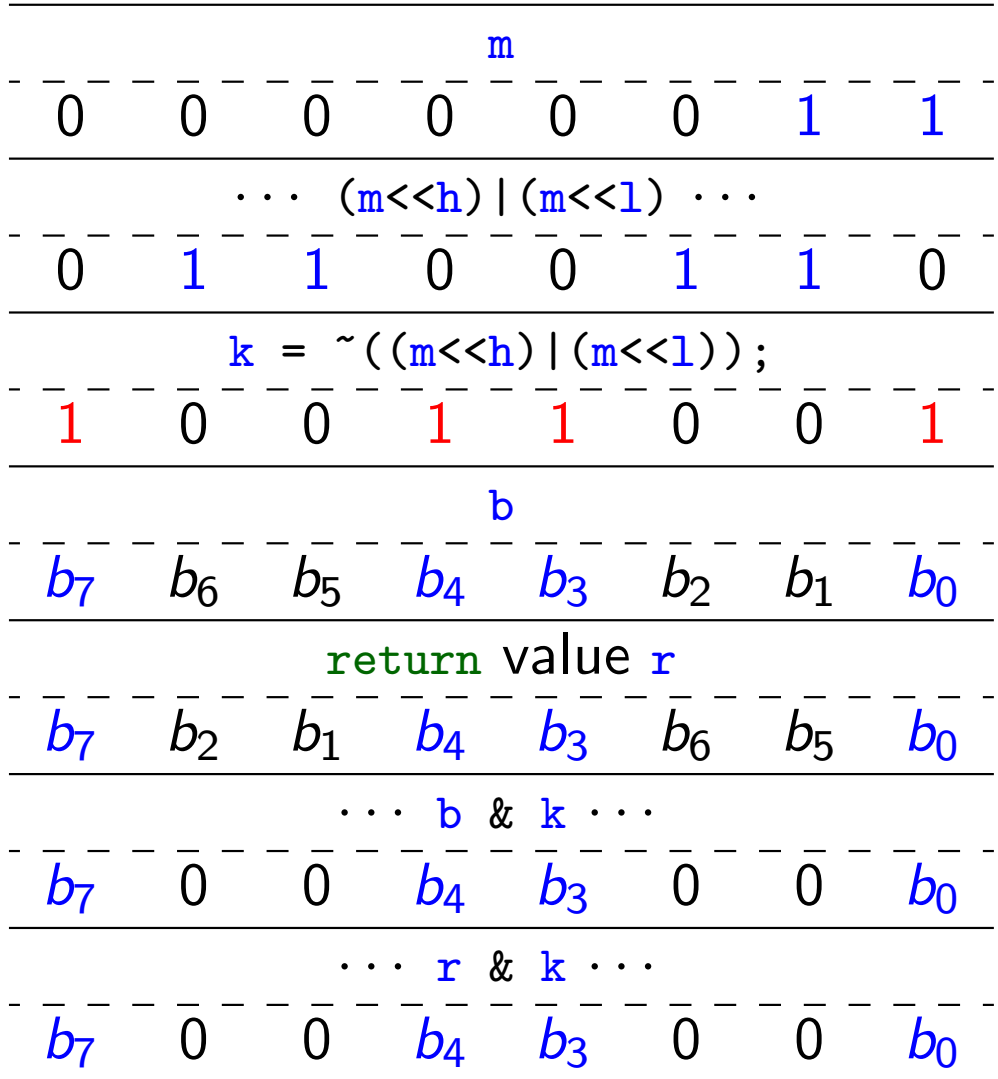


Formal Verification

Postconditions: Other Bits Unmolested

```
char swapbits(b,n,h,l)
{
  x = (b>>l)^(b>>h);
  m = (1<<n)-1;
  y = x & m;
  z = (y<<l)|(y<<h);
  return b ^ z;
}
```

- The mask `k` selects the un-swapped bits.
- They should match in `b` and `r`: `b&k==r&k`.



Formal Verification

Postconditions

The postconditions for `swapbits` are rendered below.

```
(and
  (= (bvand (bvlshr b h) m) (bvand (bvlshr r l) m))
  (= (bvand (bvlshr b l) m) (bvand (bvlshr r h) m))
  (= k (bvnot (bvor (bvshl m l) (bvshl m h))))
  (= (bvand b k) (bvand r k))
)
```

- We would like to prove that all inputs allowed by the preconditions cause these postconditions to be true.

Formal Verification

Postconditions

The postconditions for `swapbits` are rendered below.

```
(not ◀ (and
  (= (bvand (bvlshr b h) m) (bvand (bvlshr r l) m))
  (= (bvand (bvlshr b l) m) (bvand (bvlshr r h) m))
  (= k (bvnot (bvor (bvshl m l) (bvshl m h))))
  (= (bvand b k) (bvand r k))
))
```

- We would like to prove that all inputs allowed by the preconditions cause these postconditions to be true.
- This is a **universal query**, so we negate the formula.
- If the result is UNSAT, correctness is proven.

Formal Verification

Total Formula

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 8))
; Etc. for b,n,h,l,m,y,z,r

(assert
  (and
    (preconditions)
    (code)
    (not (postconditions))
  )
)
(check-sat)
```

Solver returns UNSAT, thus proving correctness.

Formal Verification

Removal of Preconditions

Suppose we removed the precondition mandating `n` positive.

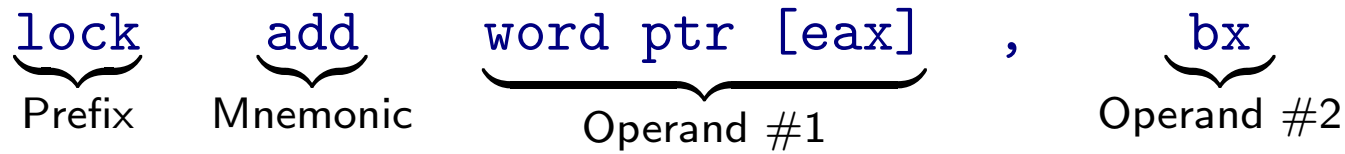
SAT

`b`	`0x20` ◀	`m`	`0xff`
`n`	`0xfe` ◀	`y`	`0x8`
`h`	`0x06`	`z`	`0x20`
`l`	`0x02`	`r`	`0x0` ◀
		`x`	`0x8`

- The solver finds an input where the postconditions are false.
- `n` ◀ has a negative value.
- Output `r` ◀ is incorrect with respect to input `b` ◀.

X86 Assembly Language

Overview



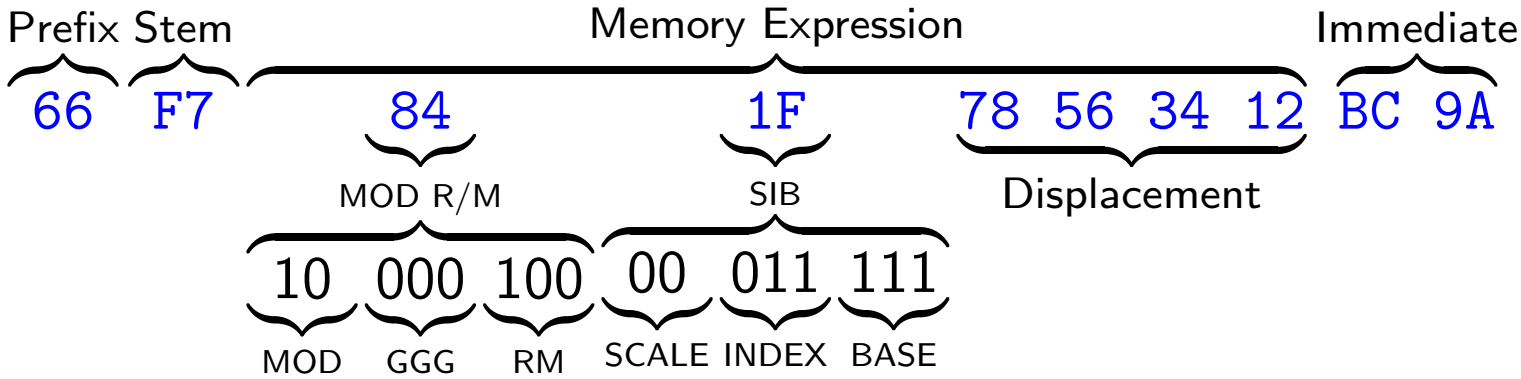
An X86 instruction consists of:

1. Zero or more **prefixes** (dictating sizes, repetition (REP), atomicity behavior (LOCK), etc.).
2. A **mnemonic**, which gives a human-friendly name to the operation being performed by the processor.
3. Zero to three **operands**, the "arguments".

X86 Machine Code

Overview

```
66 F7 84 1F 78 56 34 12 BC 9A
test word ptr [edi+ebx+12345678h], 9ABCh
```

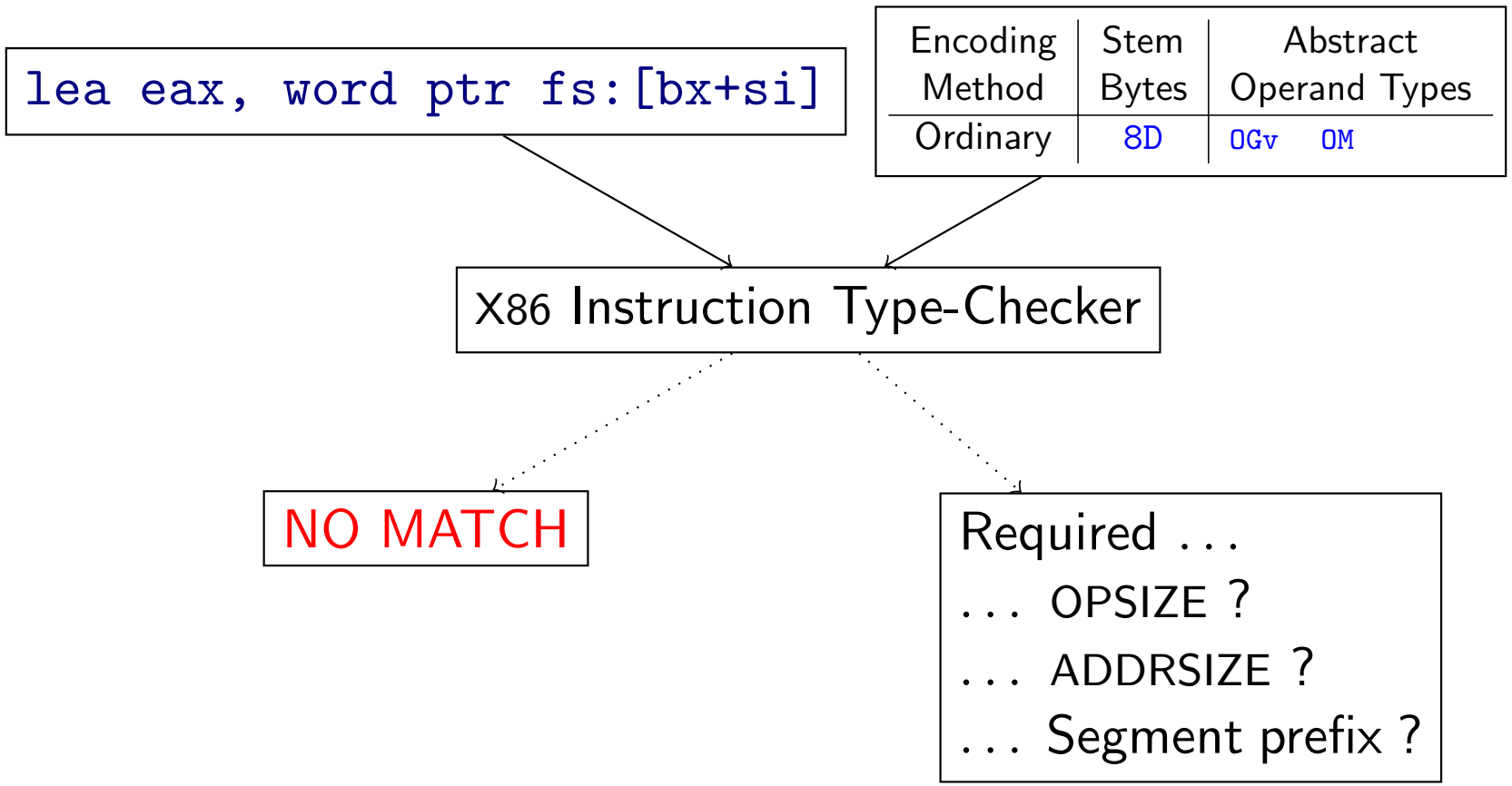


X86 machine code instructions consist of four parts:

1. Optional **prefixes**
2. An **instruction stem**
3. An optional **memory expression**, consisting of:
   - 3.1 A **MOD R/M** byte
   - 3.2 An optional **SIB** byte
   - 3.3 An optional **displacement** (1, 2, or 4 bytes)
4. Zero, one, or two **immediate** (constant) values

X86 Type Checking

Overview



- The **instruction type-checker** decides whether a purported X86 instruction matches a given encoding.
- Additionally, it reports any prefixes required to encode it.

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
      ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv ▲	OM

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd)) ◀
OM	AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))

- Check the first operand against OGv.

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
      ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv ▲	OM

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw)◀,GPart(Gd)) ◀
OM	AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))

- Check the first operand against GPart(Gw).
  - No match.

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
      ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv ▲	OM

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd)◀) ◀
OM	AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))

- Check the first operand against GPart(Gd).
  - Matches. OPSIZE possible, not required: SizePFX(False).

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
              ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM ▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OM	AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md)) ◀

- Check the first operand against GPart(Gd).
  - Matches. OPSIZE possible, not required: SizePFX(False).
- Check the second operand against OM.

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
              ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM ▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OM	AddrPrefix(RegOrMem(None,Mw)◀,RegOrMem(None,Md)) ◀

- Check the first operand against GPart(Gd).
  - Matches. OPSIZE possible, not required: SizePFX(False).
- Check the second operand against RegOrMem(None,Mw).

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
              ▲
```

Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM ▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OM	AddrPrefix(RegOrMem(None◀,Mw)◀,RegOrMem(None,Md)) ◀

- Check the first operand against GPart(Gd).
  - Matches. OPSIZE possible, not required: SizePFX(False).
- Check the second operand against None.
  - No match.

X86 Type Checking

Example

```
lea   eax   , word ptr fs:[bx+si]
              ▲
```

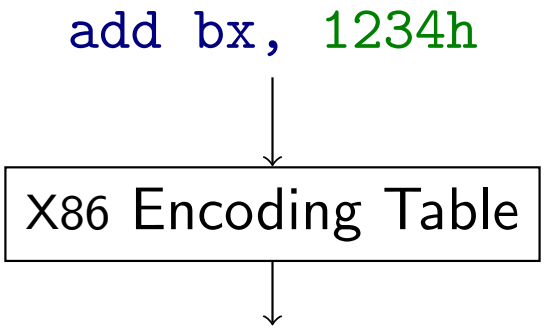
Encoding Method	Stem Bytes	Abstract Operand Types	
Ordinary	8D	OGv	OM ▲

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd))
OM	AddrPrefix(RegOrMem(None,Mw◀)◀,RegOrMem(None,Md)) ◀

- Check the first operand against GPart(Gd).
  - Matches. OPSIZE possible, not required: SizePFX(False).
- Check the second operand against Mw.
  - Matches. ADDRSIZE possible, required: AddrPFX(True).
  - fs prefix required: SegPFX(FS).

Encoding X86 Instructions

Retrieve Encodings



Encoding Method	Stem Bytes	Abstract Operand Types
Ordinary	00	OEb OGb
Ordinary	01	OEv OGv
MOD R/M Group #0	80	OEb OIb
MOD R/M Group #0	81	OEv OIz

Encoding Table for `add` (Partial)

- First, look up the mnemonic's encodings.

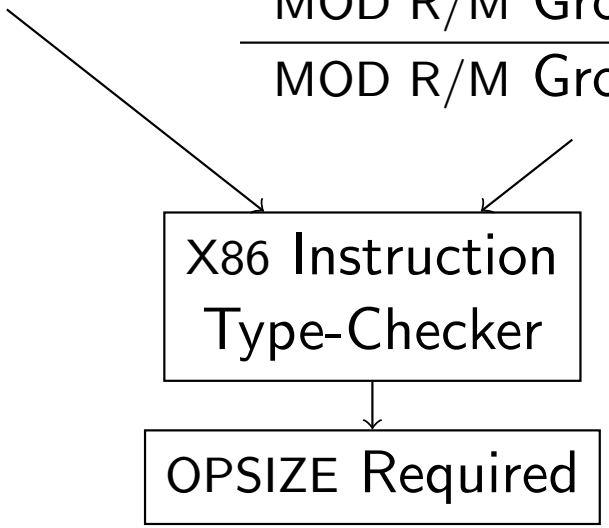
Encoding X86 Instructions

Finding a Suitable Encoding

```
add bx, 1234h
```

Encoding Table for `add` (Partial)

Ordinary	00	OEb	OGb
Ordinary	01	OEv	OGv
MOD R/M Group #0	80	OEb	OIb
MOD R/M Group #0	81	OEv	OIz ◀



- Type-check the instruction against every encoding.
- Find the first one that matches ◀.
- Return it and the prefix information.

Encoding X86 Instructions

The Encoding Context

`add bx, 1234h` MOD R/M Group #0 | 81 | OEv OIz

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81		

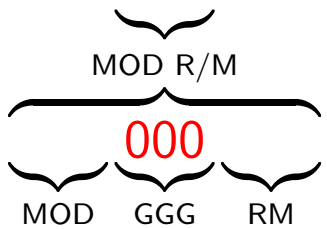
Encoding Context

- Allocate an **encoding context** for subsequent use.
- Type-checking indicated OPSIZE was required.
- The stem is 81.

Encoding X86 Instructions

Attend to Encoding Method

`add bx, 1234h` MOD R/M Group #0 | 81 | OEv OIz

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81		

Encoding Method	Action
Ordinary	Do nothing.
Size Prefix	Set OPSIZE.
MOD R/M Group #n	Set MOD R/M.GGG to n.

- Perform the action required by the encoding method.

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | `81` | `OEv` `OIz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	MOD R/M: 000 (MOD GGG RM)	

Operand Type	AOTDL
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))` ◀
`OIz`	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))`

- Encode the first operand using `OEv`.
  - OPSIZE present, so look at prefixed part.

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | 81 | OEv OIz

▲ ▲

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	MOD R/M: 11 000 (MOD GGG RM)	

Operand Type	AOTDL
OEv	SizePrefix(RegOrMem(Gw,Mw)◀,RegOrMem(Gd,Md)) ◀
OIz	SizePrefix(ImmEnc(Iw),ImmEnc(Id))

- Encode the first operand using RegOrMem(Gw,Mw).
  - Operand is a register, so set MOD R/M.MOD to 11.

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | `81` | `OEv` `OIz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	C3 = MOD R/M: 11 (MOD) 000 (GGG) 011 (RM)	

Operand Type	AOTDL
OEv	`SizePrefix(RegOrMem(Gw◀,Mw),RegOrMem(Gd,Md))` ◀
OIz	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))`

- Encode the first operand using `Gw`.
  - Set MOD R/M.RM to 011 (`bx`'s register number).

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | `81` | `OEv` `OIz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	C3 (MOD R/M: 11 000 011 — MOD GGG RM)	

Operand Type	AOTDL
OEv	SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))
OIz	SizePrefix(ImmEnc(Iw),ImmEnc(Id))

- Encode the first operand using `Gw`.
  - Set MOD R/M.RM to 011 (`bx`'s register number).
- Encode the second operand using `OIz`.
  - OPSIZE present, so look at prefixed part.

Encoding X86 Instructions

Emit Operands

`add bx , 1234h` MOD R/M Group #0 | `81` | `OEv` `OIz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE	81	C3 (MOD R/M: 11 000 011 — MOD GGG RM)	34 12◀

Operand Type	AOTDL
OEv	SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))
OIz	SizePrefix(ImmEnc(Iw)◀,ImmEnc(Id)) ◀

- Encode the first operand using `Gw`.
  - Set MOD R/M.RM to 011 (`bx`'s register number).
- Encode the second operand using `ImmEnc(Iw)`.
  - Encode as an immediate ◀.

Encoding X86 Instructions

Producing X86 Machine Code

`add bx, 1234h` MOD R/M Group #0 | `81` | `OEv` `OIz`

Required Prefixes	Stem	MOD R/M	Immediates
OPSIZE (66)	81	C3 (MOD R/M: 11 000 011 — MOD GGG RM)	34 12

66 81 C3 34 12

- Concatenate all fields together to produce X86 machine code.

Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

```
66 6B 84 1F 78 56 34 12 9A
```

Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

1. Consume prefixes.
   - 66 is OPSIZE.

66 6B 84 1F 78 56 34 12 9A

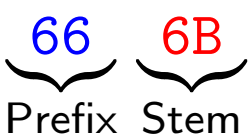
Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

1. Consume prefixes.
   - `66` is OPSIZE.
2. Consume stem, retrieve corresponding decoder entry.
   - Entry is `Direct(Imul,[OGv,OEv,OIb])`.

```
66 6B 84 1F 78 56 34 12 9A
```



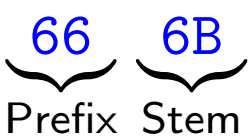
Decoding X86 Instructions

Overview: Decode Prefixes and Stem

Given a stream of bytes:

1. Consume prefixes.
   - `66` is OPSIZE.
2. Consume stem, retrieve corresponding decoder entry.
   - Entry is `Direct(Imul,[OGv,OEv,OIb])`.
3. Process decoder entry.
   - Decode operands (shown on next slide).

```
66 6B 84 1F 78 56 34 12 9A
```



Decoding X86 Instructions

Overview: Decode Operands

imul | OGv OEv OIb

Operand Type	AOTDL
OGv	SizePrefix(GPart(Gw),GPart(Gd)) ◀
OEv	SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))
OIb	ImmEnc(Ib)

- Decode the first operand using OGv.
  - OPSIZE present, so look at prefixed part.

66 6B 84 1F 78 56 34 12 9A

66 6B

Prefix Stem

Decoding X86 Instructions

Overview: Decode Operands

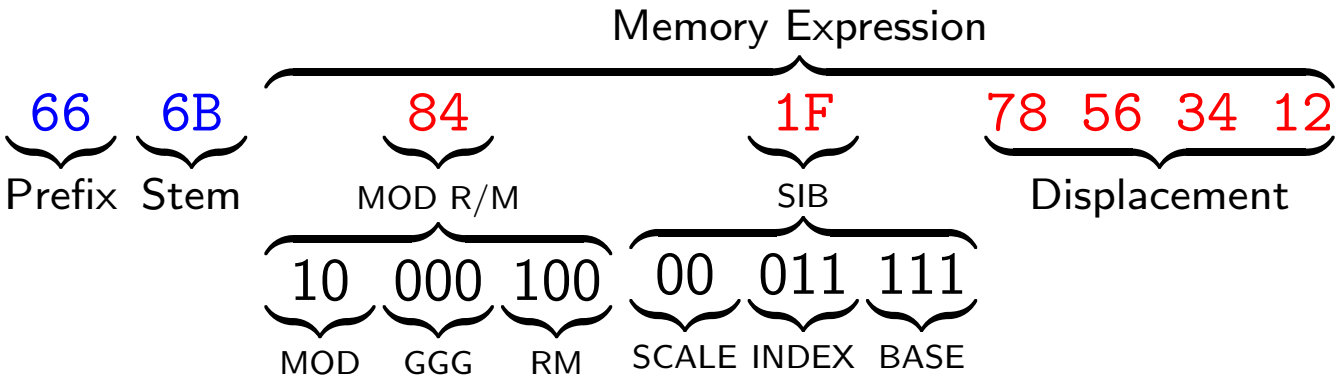
`imul` | `OGv` `OEv` `OIb`

▲

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw)◀,GPart(Gd))` ◀
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIb`	`ImmEnc(Ib)`

- Decode the first operand using `GPart(Gw)`
  - Decode MOD R/M, return `OGw` register numbered GGG.

`66 6B 84 1F 78 56 34 12 9A`



Decoding X86 Instructions

Overview: Decode Operands

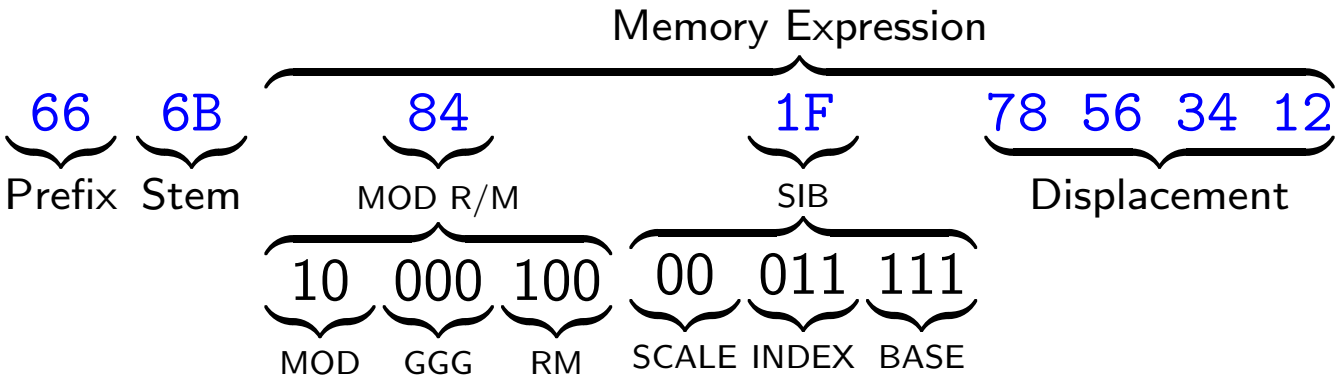
`imul` | `OGv` `OEv` `OIb`

▲

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))` ◀
`OIb`	`ImmEnc(Ib)`

- Decode the second operand using `OEv`.
  - First operand is `ax`.
  - OPSIZE present, so look at prefixed part.

`66 6B 84 1F 78 56 34 12 9A`



Decoding X86 Instructions

Overview: Decode Operands

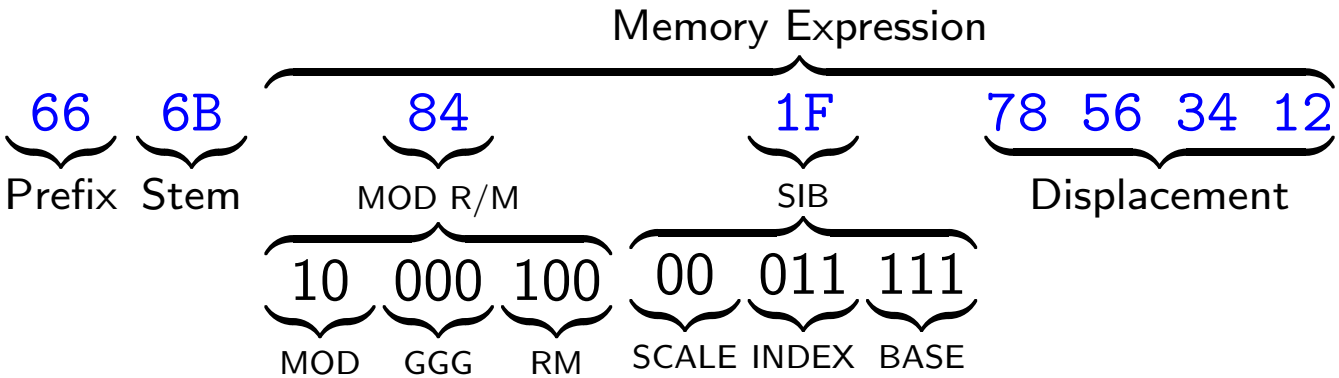
`imul` | `OGv` `OEv` `OIb`

▲

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw)◀,RegOrMem(Gd,Md))` ◀
`OIb`	`ImmEnc(Ib)`

- Decode the second operand using `RegOrMem(Gw,Mw)`.
  - First operand is `ax`.
  - MOD R/M specifies a memory location.

`66 6B 84 1F 78 56 34 12 9A`



Decoding X86 Instructions

Overview: Decode Operands

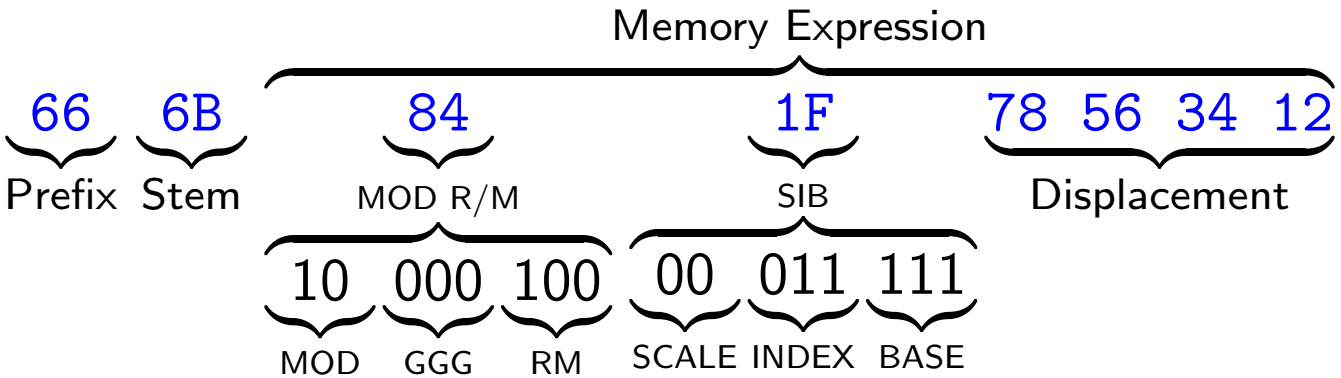
`imul` | `OGv` `OEv` `OIb`

▲

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw◀),RegOrMem(Gd,Md))` ◀
`OIb`	`ImmEnc(Ib)`

- Decode the second operand using `Mw`.
  - First operand is `ax`.
  - Decode a `word`-sized memory expression.

`66 6B 84 1F 78 56 34 12 9A`



Decoding X86 Instructions

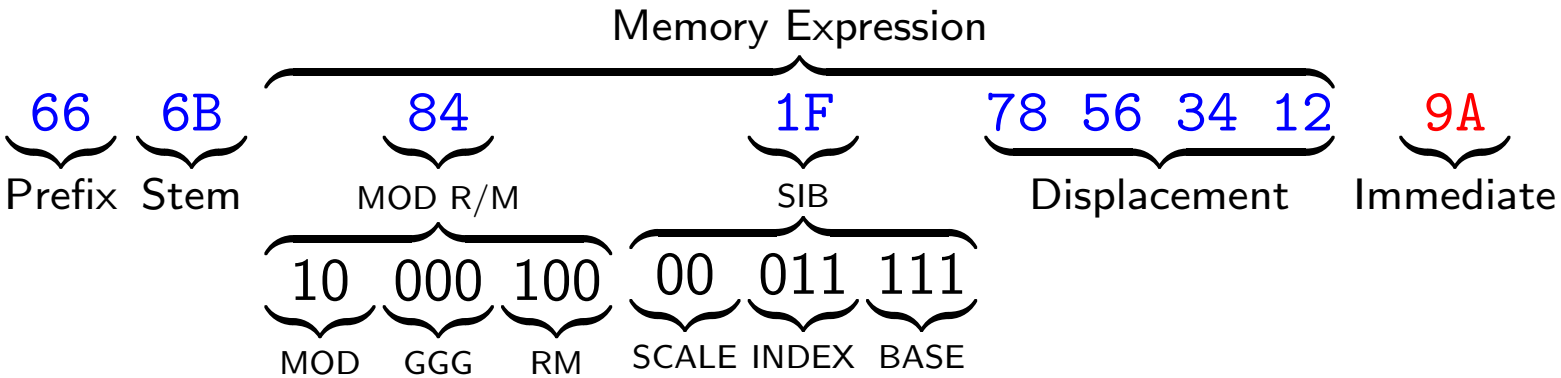
Overview: Decode Operands

`imul` | `OGv` `OEv` `OIb`

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIb`	`ImmEnc(Ib)` ◀

- Decode the third operand using `ImmEnc(Ib)`.
  - First operand is `ax`.
  - Second operand is `word ptr [edi+ebx+12345678h]`.
  - Retrieve a byte from the instruction stream.

`66 6B 84 1F 78 56 34 12 9A`



Decoding X86 Instructions

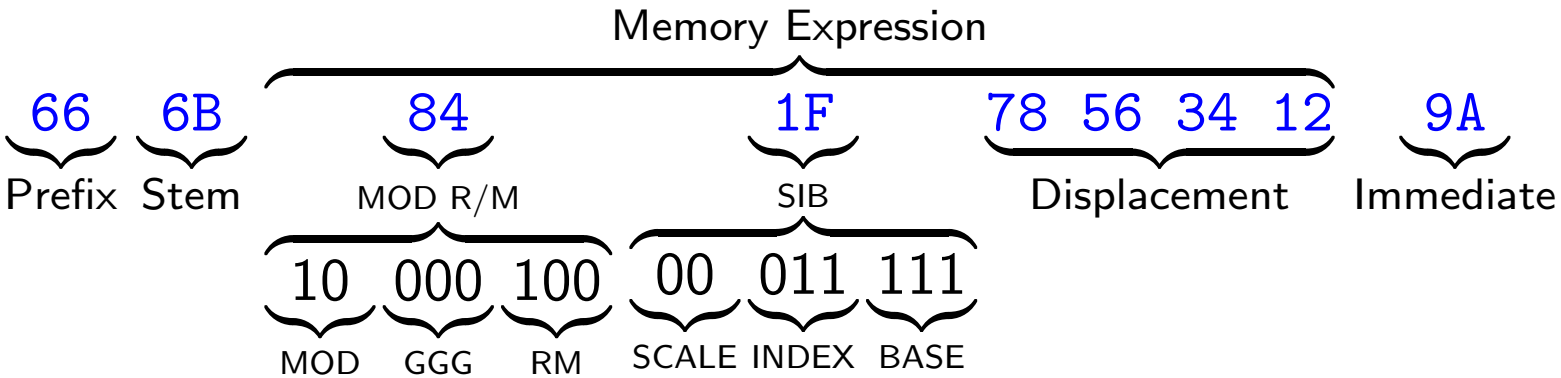
Overview: Decode Operands

`imul` | `OGv` `OEv` `OIb`

Operand Type	AOTDL
`OGv`	`SizePrefix(GPart(Gw),GPart(Gd))`
`OEv`	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
`OIb`	`ImmEnc(Ib)`

- Decode the third operand using `ImmEnc(Ib)`.
  - First operand is `ax`.
  - Second operand is `word ptr [edi+ebx+12345678h]`.
  - Third operand is `9Ah`.

`66 6B 84 1F 78 56 34 12 9A`



Instruction is `imul ax, word ptr [edi+ebx+12345678h], 9Ah`

Intermediate Representations

In Compiler Theory

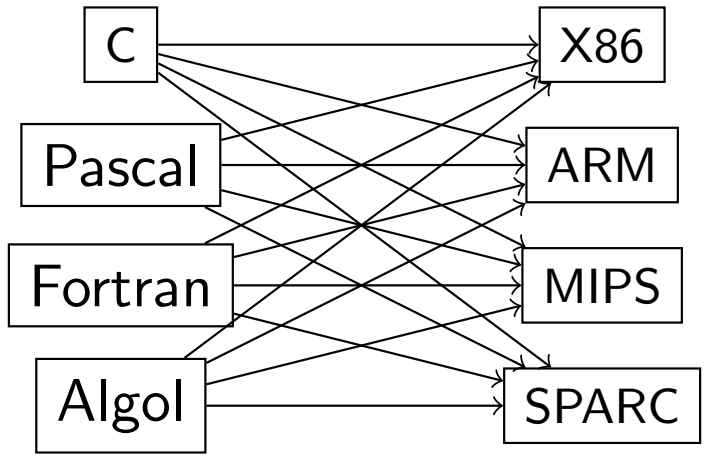


Figure: 16 Compilers

- Classic problem in compiler design.
- We want to compile four languages into four assembly languages apiece without writing 16 compilers.
  - Wasted effort: the input and output languages are similar.
- How much of the compiler can be reused across all projects?

Intermediate Representations

In Compiler Theory

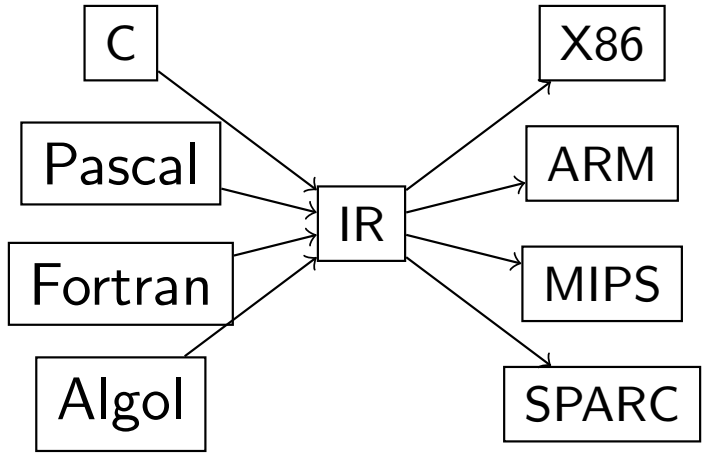
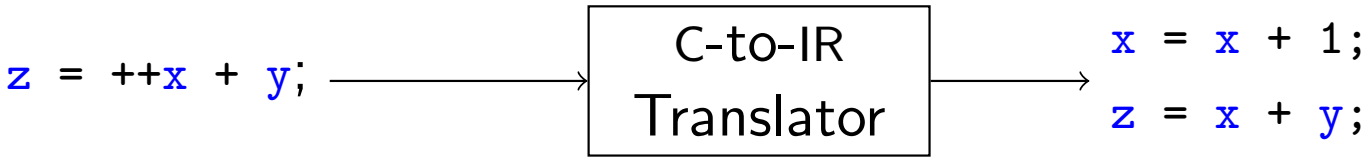


Figure: One Compiler, Four IR Translators, Four Code Generators

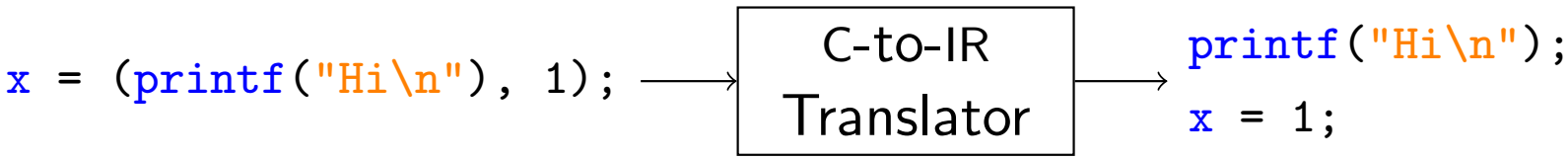
- We design an **intermediate language** (IR) that is capable of expressing every program in the set of input languages. Then, we write translators from those languages into the IR.
- The compiler is written to take the IR as input. It invokes a machine-specific **code generator** to produce the output.

Intermediate Representations

In Compiler Theory



C Preincrement Translation



C Sequence Operator Translation

- Beyond merely allowing code reuse, IRs also simplify the compiler's analysis and transformation tasks.
- Language-specific features are rewritten uniformly in the IR.
  - E.g., the increment and sequence operators are simplified.

Intermediate Representations

In Compiler Theory

```
x = x + 1;
z = x + y;
```

High-Level IR

```
add x, x, 1
add z, x, y
```

Low-Level IR

Two Compiler IRs

- Compilers use several IRs during compilation, as different representations make certain analyses easier.
- The IRs begin looking somewhat like C, and look more like assembly language in time for code generation.

Intermediate Representations

In Binary Program Analysis

- We adopt an IR for binary analysis.
  - Can analyze multiple architectures with the same code.
  - Language is vastly simpler to analyze than X86.
- Ours will be a higher-level IR.
- Next, we explore how to think about the effects of instructions (their **semantics**).

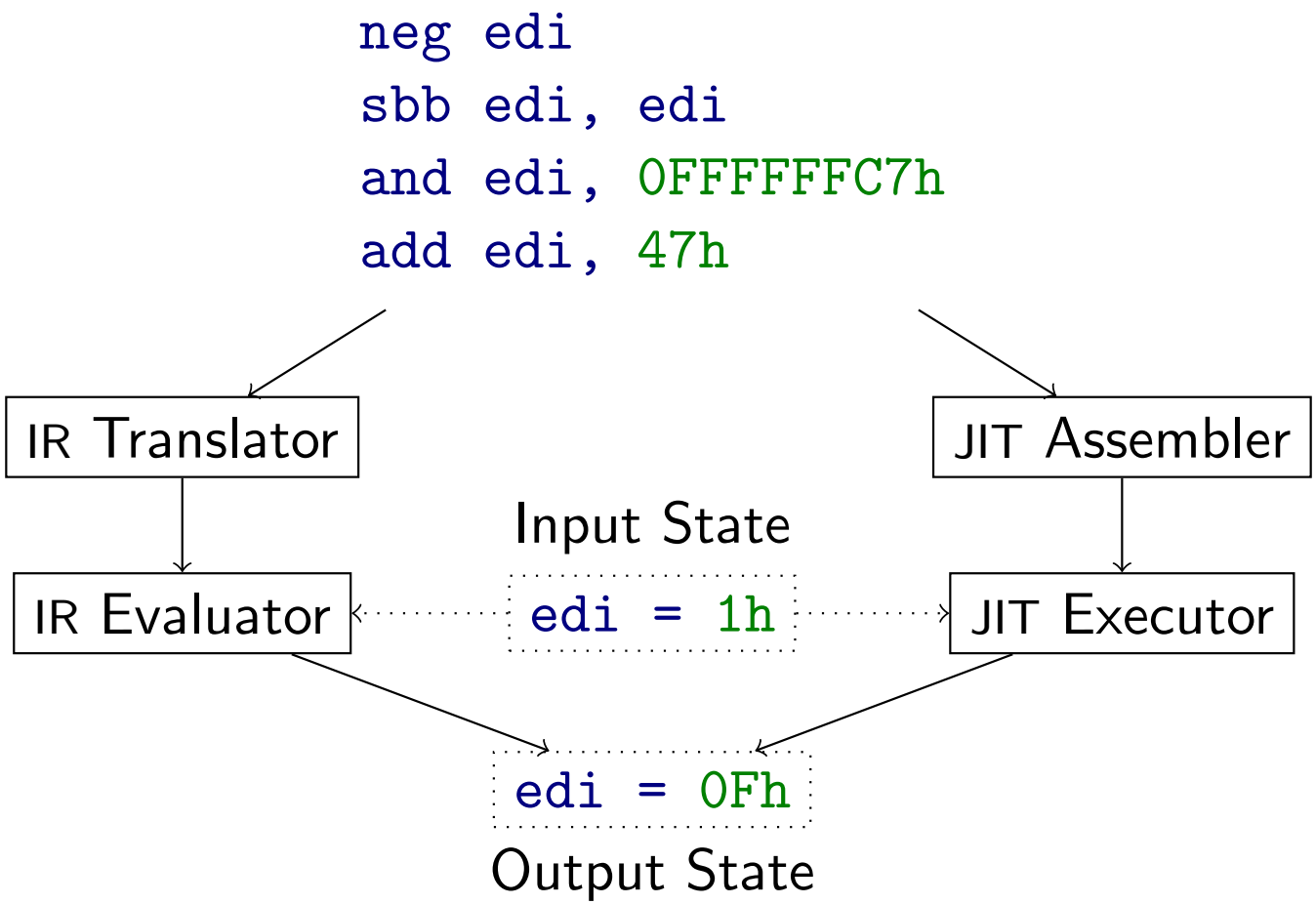
IR Framework Components

Motivating Example: Assistance with Human Comprehension

```
neg edi
sbb edi, edi
and edi, 0FFFFFFC7h
add edi, 47h
```

- As reverse engineers, we may encounter this code and wonder what it does.
- We can simply:
  - Translate the X86 into IR.
  - Apply the IR evaluator with varying values of `edi`.

IR Evaluation as CPU Emulation



- Evaluating IR should yield the same results as CPU execution.

X86-to-IR Translator

```
add edi, 47h
```

```
// add edi, 47h
T1012d = vEdi + 0x47.32;
vZF = T1012d == 0x0.32;
T1013d = T1012d ^ (T1012d >> 0x4.8);
T1014d = T1013d ^ (T1013d >> 0x2.8);
T1015d = T1014d ^ (T1014d >> 0x1.8);
vPF = \~Cast(Low,S1,T1015d);
vSF = T1012d <s 0x0.32;
vCF = T1012d <u 0x47.32;
vAF = Cast(Low,S1,((vEdi ^ 0x47.32) ^ T1012d) >> 0x4.8);
vOF = ((T1012d <s 0x0.32) ^ (vEdi <s 0x0.32)) \&
      ((T1012d <s 0x0.32) ^ (0x47.32 <s 0x0.32));
vEdi = T1012d;
vDi = Cast(Low,S16,vEdi);
```

IR Evaluator

IR Translation

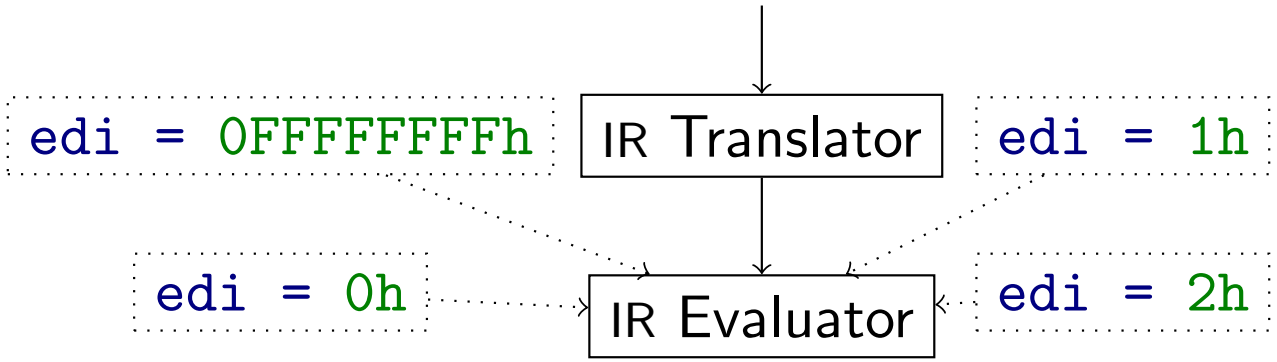
```
// add edi, 47h
T1012d = vEdi + 0x47.32;
vZF = T1012d == 0x0.32;
T1013d = T1012d ^ (T1012d >> 0x4.8);
T1014d = T1013d ^ (T1013d >> 0x2.8);
T1015d = T1014d ^ (T1014d >> 0x1.8);
vPF = ~Cast(Low,S1,T1015d);
vSF = T1012d <s 0x0.32;
vCF = T1012d <u 0x47.32;
vAF = Cast(Low,S1,((vEdi ^ 0x47.32) ^ T1012d) >> 0x4.8);
vOF = ((T1012d <s 0x0.32) ^ (vEdi <s 0x0.32)) &
      ((T1012d <s 0x0.32) ^ (0x47.32 <s 0x0.32));
vEdi = T1012d;
vDi = Cast(Low,S16,vEdi);
```

IR Evaluator Context

```
T1012d = 0xf.32;
vZF = 0x0.32;
T1012d = 0xc.32;
T1013d = 0xa.32;
T1014d = 0x0.32;
vPF = 0x1.8;
vSF = 0x0.8;
vCF = 0x1.8;
vAF = 0x1.8;
vOF = 0x0.8;
vEdi = 0xf.32;
vDi = 0xf.16;
```

Examining Behavior via IR Evaluation

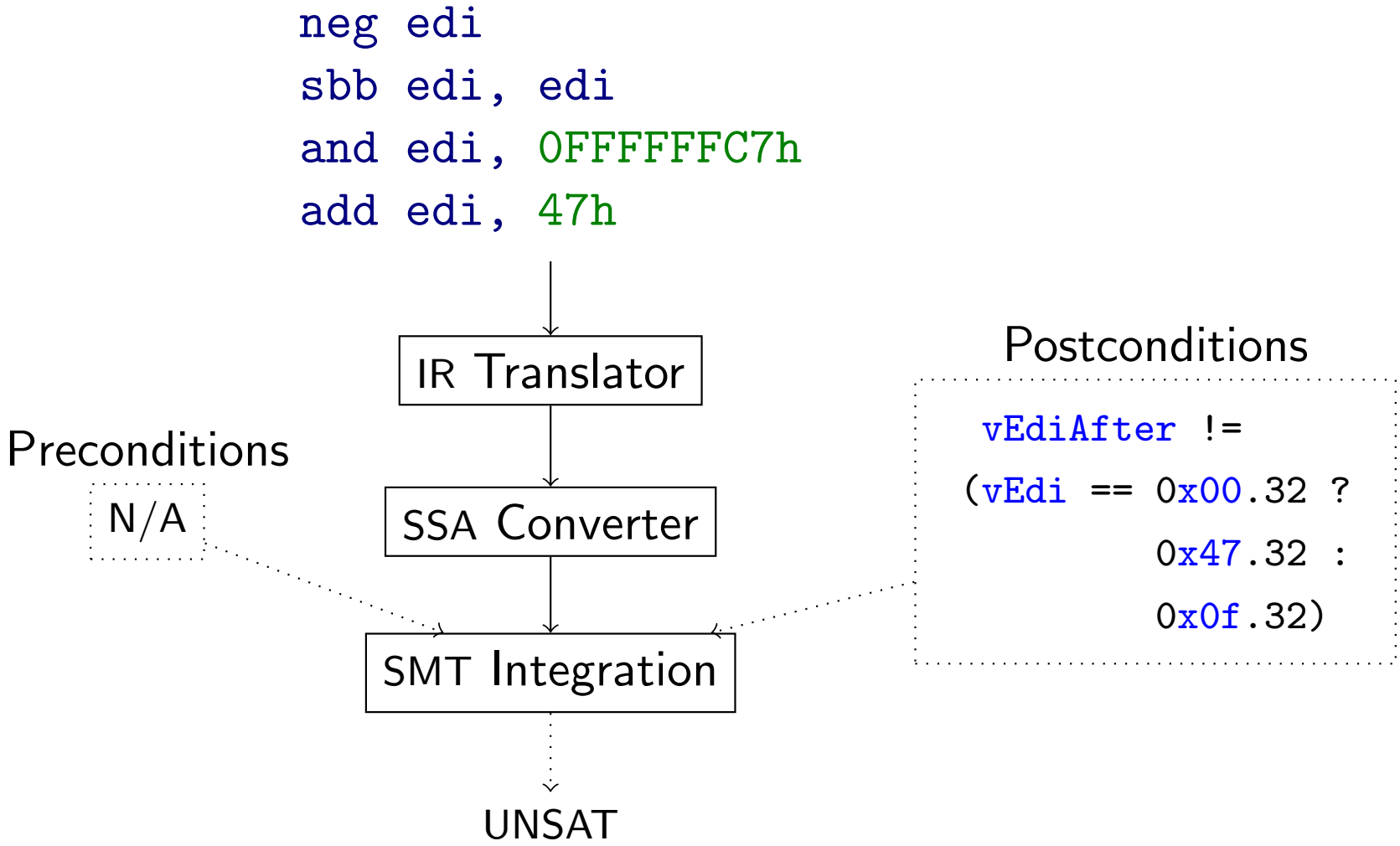
```
neg edi
sbb edi, edi
and edi, 0FFFFFFC7h
add edi, 47h
```



Input	Output	Input	Output
0FFFFFFFFh	0Fh	1h	0Fh
0h	47h	2h	0Fh

- Evaluating in several input states yields a pattern.
  - Is the behavior `vEdiAfter == vEdi ? 0xF.32 : 0x47.32`?

Solving Queries Regarding IR via SMT



- We can solve queries about IR sequences with an SMT solver.

IR Translator Synthesis

Overview

- IR translators require accurate descriptions of the operation of the X86 processor.
- Sadly, the Intel manuals are at best vague, at worst wrong.

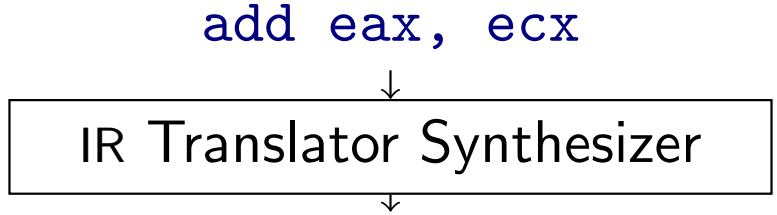
```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
WRONG:          AL ← AL + 6;        AX ← AX + 0x106
WRONG:          AH ← AH + 1;
                AF ← 1;
                CF ← 1;
                AL ← AL AND 0FH;
            ELSE
                AF ← 0;
                CF ← 0;
                AL ← AL AND 0FH;
        FI;
FI;
```

- That's from the first instruction in the manual.

IR Translator Synthesis

Overview

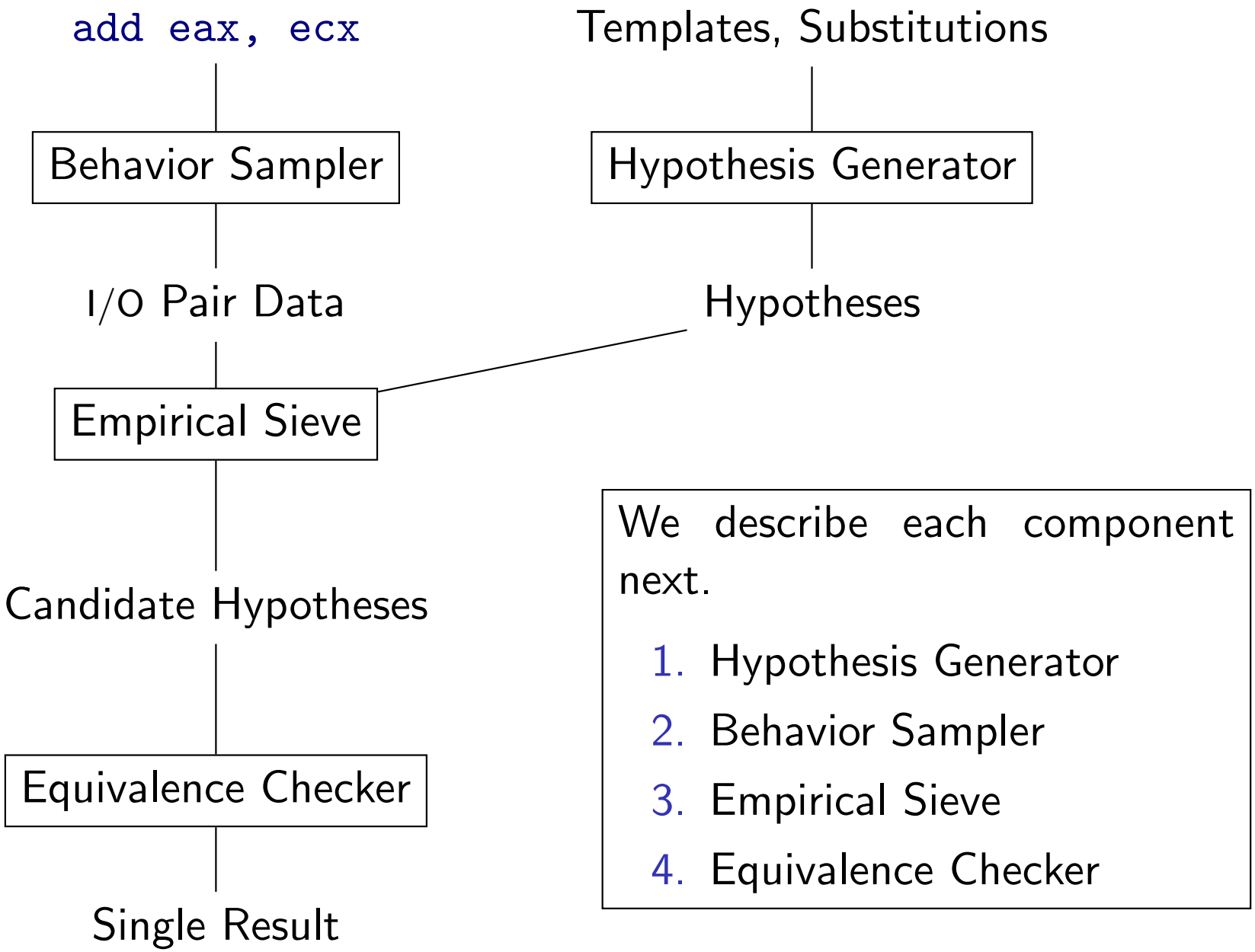
- Idea: use program synthesis to find instruction descriptions.




```
vRes = vEax + vEcx;
vZF  = vRes == 0;
vSF  = vRes <s 0;
vPF  = Parity(vRes);
vCF  = vRes <u vEax;
vOF  = (vEcx ^ vRes)
     & (vEax ^ vRes) <s 0;
vAF  = (vRes ^ vEax ^ vEcx)
     & 0x10 != 0;
vEax = vEax + vEcx;
```

IR Translator Synthesis

Overview



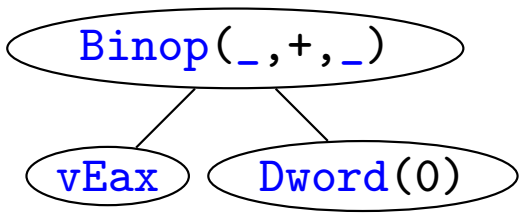
We describe each component next.

1. Hypothesis Generator
2. Behavior Sampler
3. Empirical Sieve
4. Equivalence Checker

IR Translator Synthesis

Hypothesis Generator

We generate a family of expressions from a **template expression** and a list of **substitutions**.



Tree representation of `vEax + 0`

Token	Replace With
+	+,-,\&,|,^
vEax	vEax,vEcx

Specified Substitutions

```
vEax + 0   vEax - 0   vEax & 0   vEax | 0   vEax ^ 0
vEcx + 0   vEcx - 0   vEcx & 0   vEcx | 0   vEcx ^ 0
```

The $5 * 2 = 10$ expressions generated from the above

IR Translator Synthesis

Hypothesis Generator

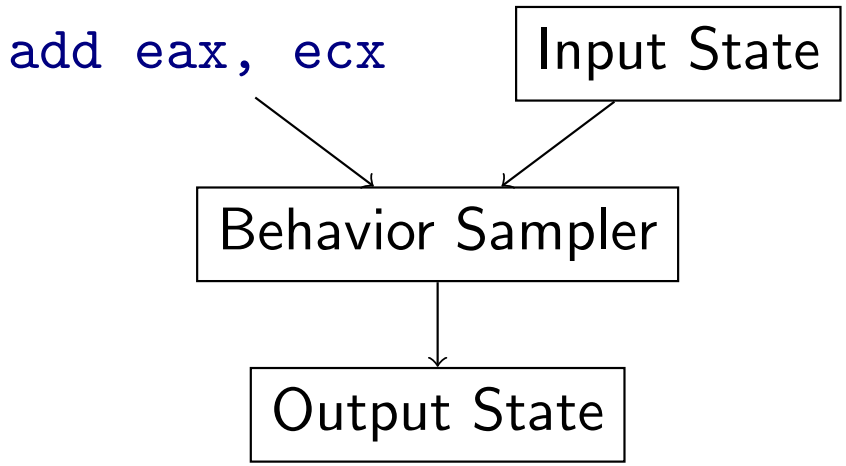
- A **result location** is a location modified by an instruction.
- A **hypothesis** is an IR expression of equality between a result location variable and a generated expression.
- For the instruction `add eax, ebx`, the result locations are $\texttt{vEax}_{out}$, $\texttt{vZF}_{out}$, $\texttt{vSF}_{out}$, $\texttt{vPF}_{out}$, $\texttt{vCF}_{out}$, $\texttt{vOF}_{out}$, and $\texttt{vAF}_{out}$.

$\texttt{vZF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vZF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vSF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vSF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vPF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vPF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vCF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vCF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vOF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vOF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vAF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vAF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0

Some hypotheses for `add eax, ebx`

IR Translator Synthesis

Behavior Sampler



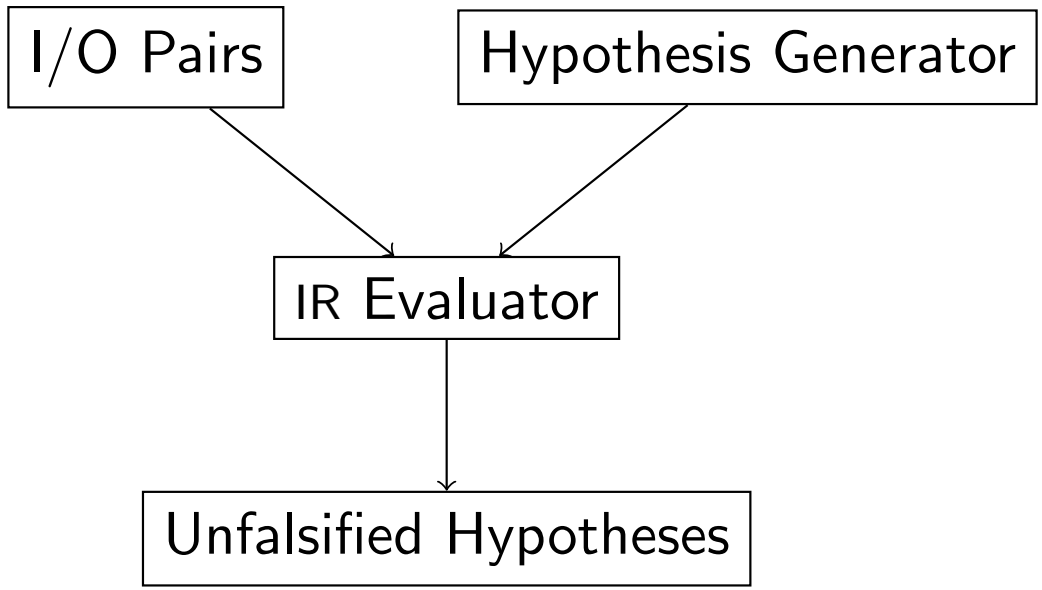
Given a set of register and flag values as input:

1. JIT assemble the instruction
2. Set the processor registers and flags to the input state
3. Execute the instruction
4. Collect the values of the registers and flags as output

These **I/O pairs** are samples of the instruction's behavior.

IR Translator Synthesis

Empirical Sieve



- We generate many hypotheses, test them against the I/O pairs, and discard any that evaluate to `false`.

IR Translator Synthesis

Equivalence Checker

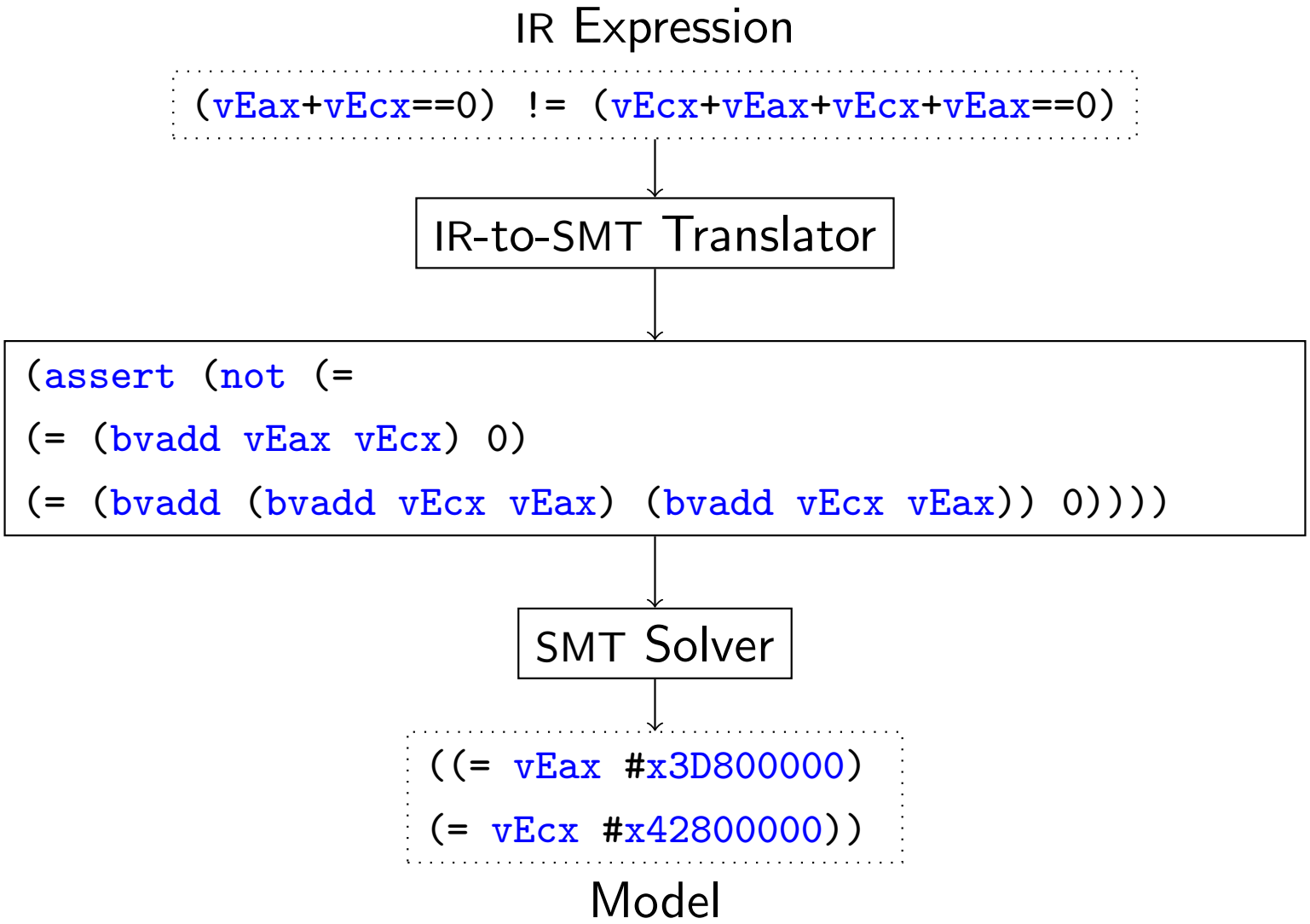
- If, after the empirical sieve, we have multiple unfalsified hypotheses, we may like to remove some of them.
- The **equivalence checker** uses an SMT solver to determine if two expressions evaluate equally for all inputs. If not, it finds an input that causes a different evaluation.

`(vEax+vEcx) == 0`	`(vEcx+vEax+vEcx+vEax) == 0`
Expression #1	Expression #2

Figure: Two hypotheses for $\texttt{ZF}_{out}$.

IR Translator Synthesis

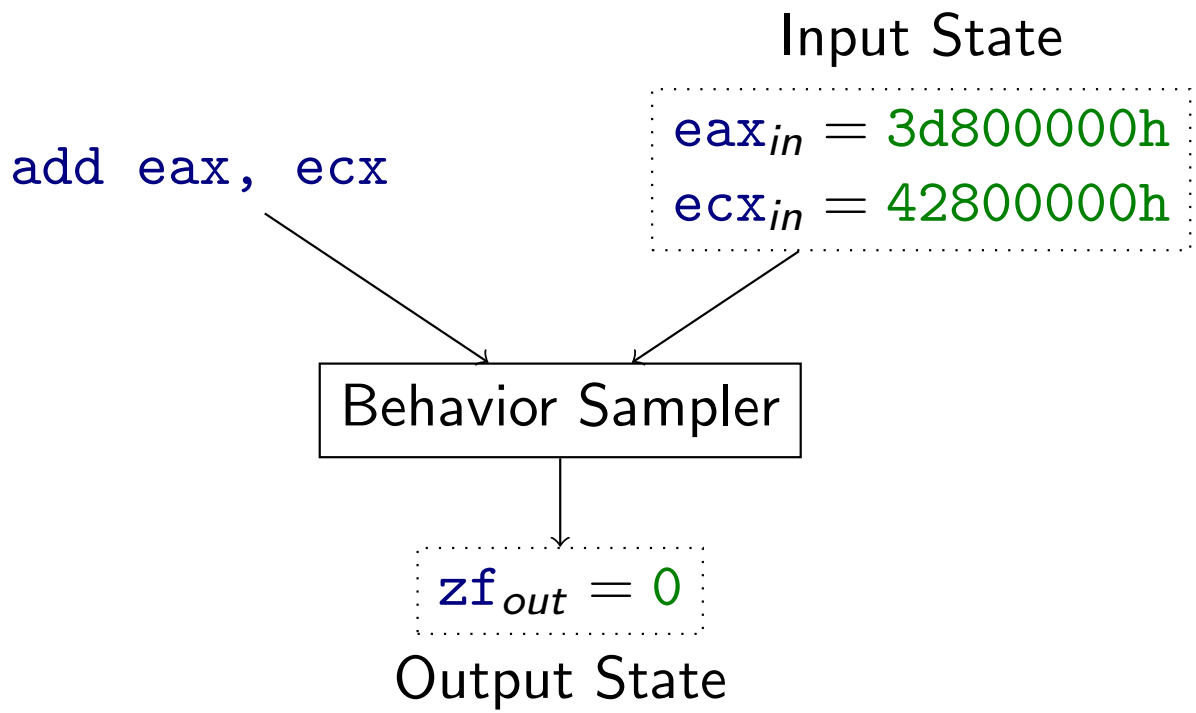
Equivalence Checker: SMT Integration



- The SMT solver gives a state where the hypotheses differ.

IR Translator Synthesis

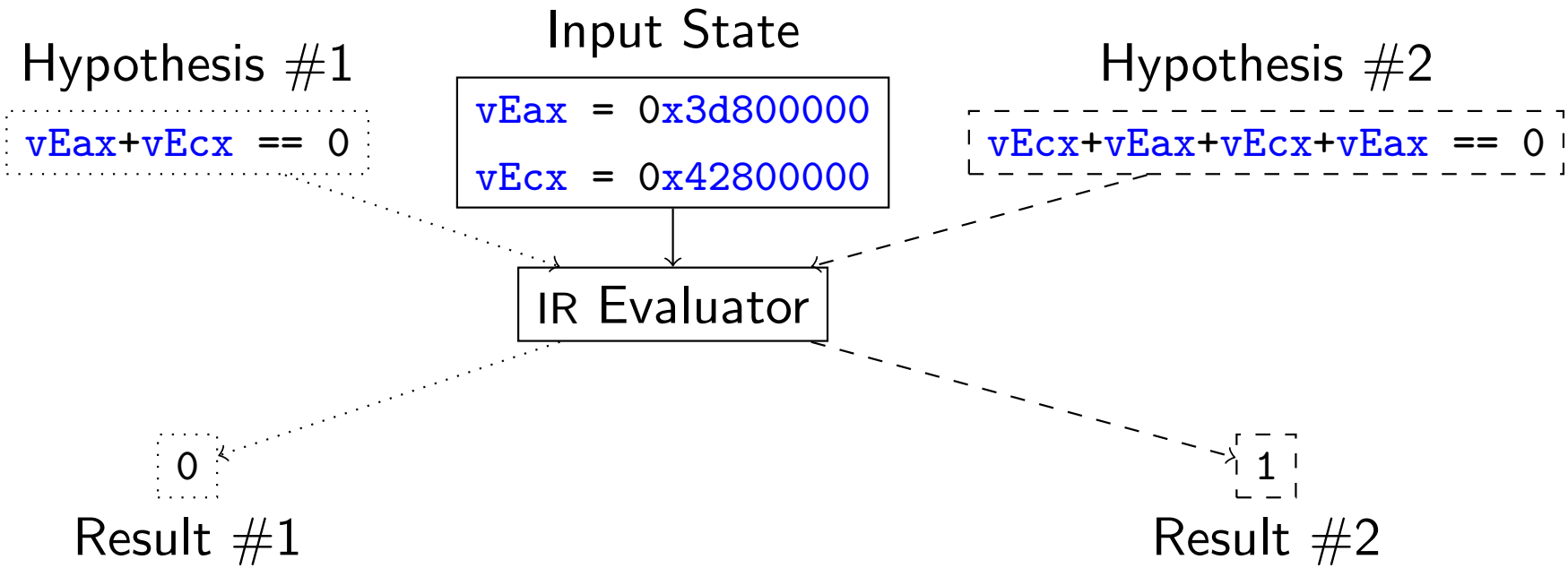
Equivalence Checker: Behavior Sampling



- We sample the instruction in the state provided by the SMT solver.

IR Translator Synthesis

Equivalence Checker: Distinguishment



- One hypothesis must differ from the behavior sample.
  - Hypothesis #2 evaluates to 1, while the sample is 0.
  - Thus, hypothesis #2 is proven false and removed.

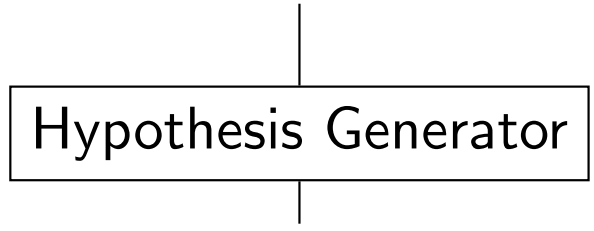
IR Translator Synthesis

Behavior Sampler

<out> == <in>,

<out> == <in> <+> <in>,

...



CF_{out} == (vEax == vEcx),

CF_{out} == (vEcx <s 0),

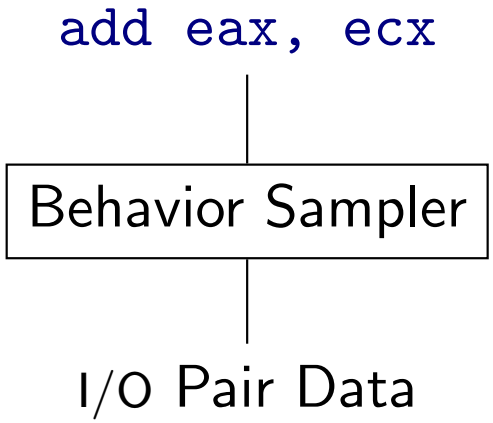
CF_{out} == (vEax + vEcx <u vEax),

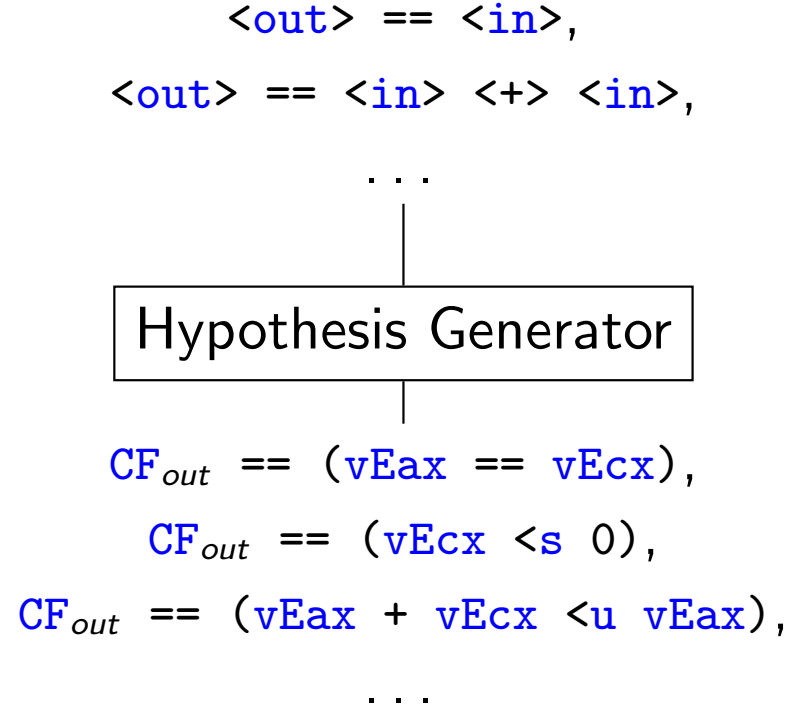
...

First, we generate hypotheses for the instruction's effects.

IR Translator Synthesis

Hypothesis Generator

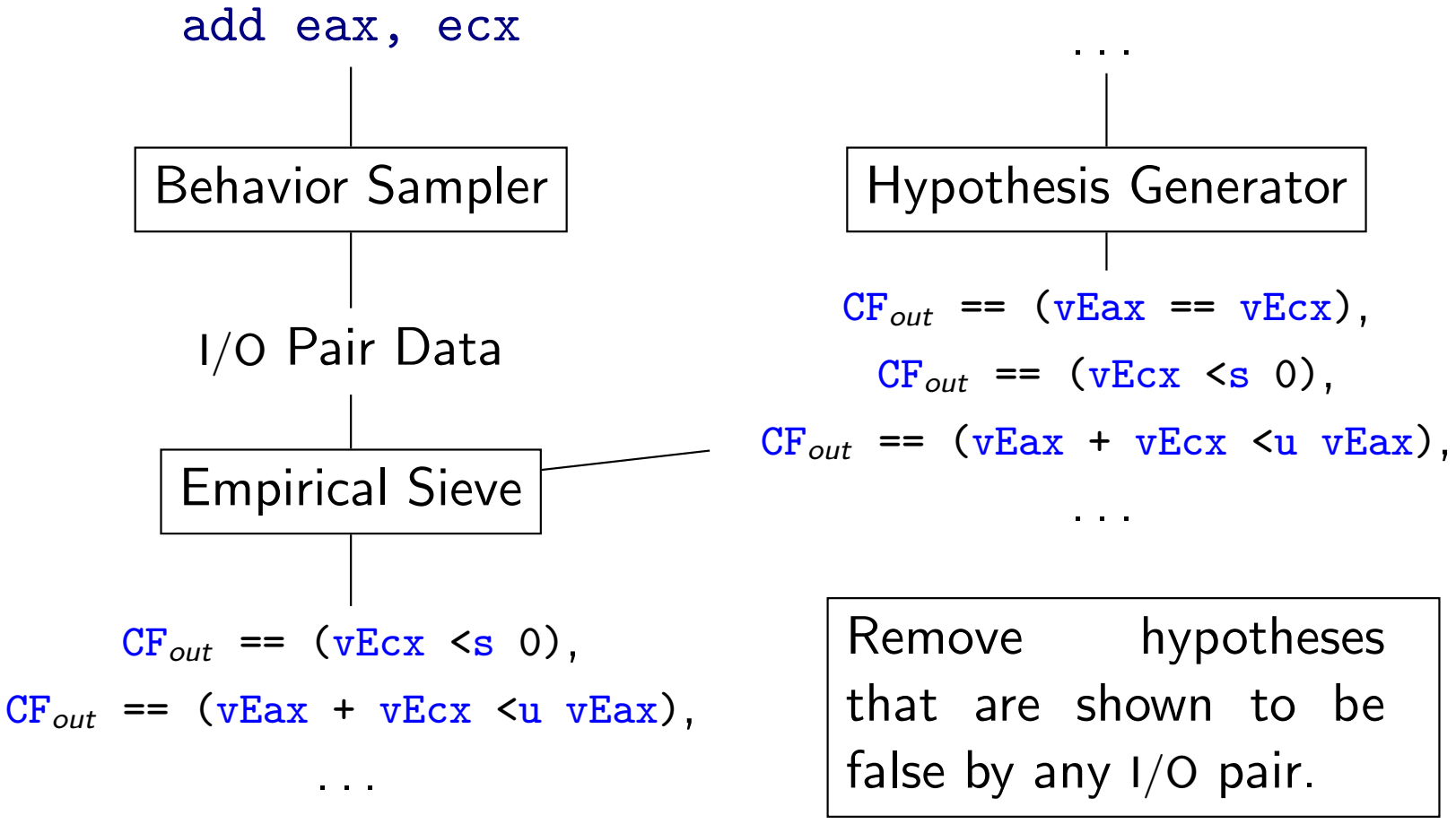




Next, we collect I/O pairs for an instruction.

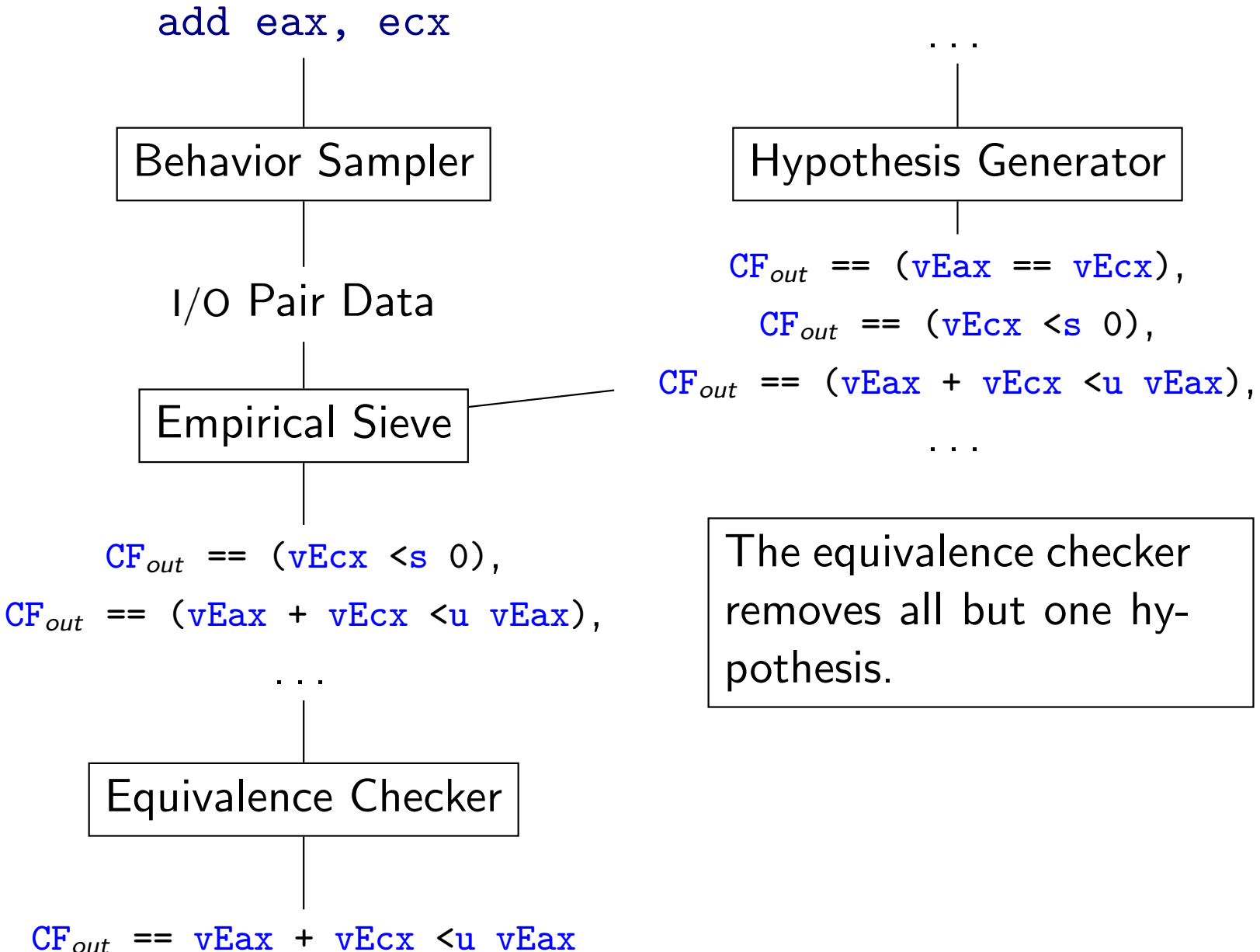
IR Translator Synthesis

Empirical Sieve



IR Translator Synthesis

Equivalence Checking



X86-Specific Considerations

Intrinsic Relationship of X86 Subregisters

- On X86, the general registers are **nested**.
  - `ax`, `ah`, `al` are contained within `eax`.
  - `ah`, `al` are contained within `ax`.
  - `di` is contained within `edi`.

X86-Specific Considerations

Non-Relationship of IR Variables

- In our IR, there is no intrinsic relationship between variables.
  - E.g., modifying `vAl` does not modify `vEax`.
- Our IR translator must explicitly update related variables.

```
// not al
vAl = ~vAl;
vAx = (vAx & 0xFF00.16) | Cast(Unsigned,16,vAl);
vEax = (vEax & 0xFFFFFF00.32) | Cast(Unsigned,32,vAl);
```

X86-Specific Considerations

Encapsulation and SSA

```
vAx = Cast(Low,16,vEax);
vAl = Cast(Low,8,vEax);
vAh = Cast(High,8,vAx);
vBx = Cast(Low,16,vEbx);
vBl = Cast(Low,8,vEbx);
vBh = Cast(High,8,vBx);
// etc. for all general registers
// IR translation of instructions
vZFAfter  = vZF;
vAlAfter  = vAl;
vMemAfter = vMem;
// etc. for all reserved variables
```

- Prepend subregister assignments.
- Append "after" variable assignments.
- Convert everything to SSA form.

Analysis of Branch-Free Code

Primary Application Archetypes: Input Generation

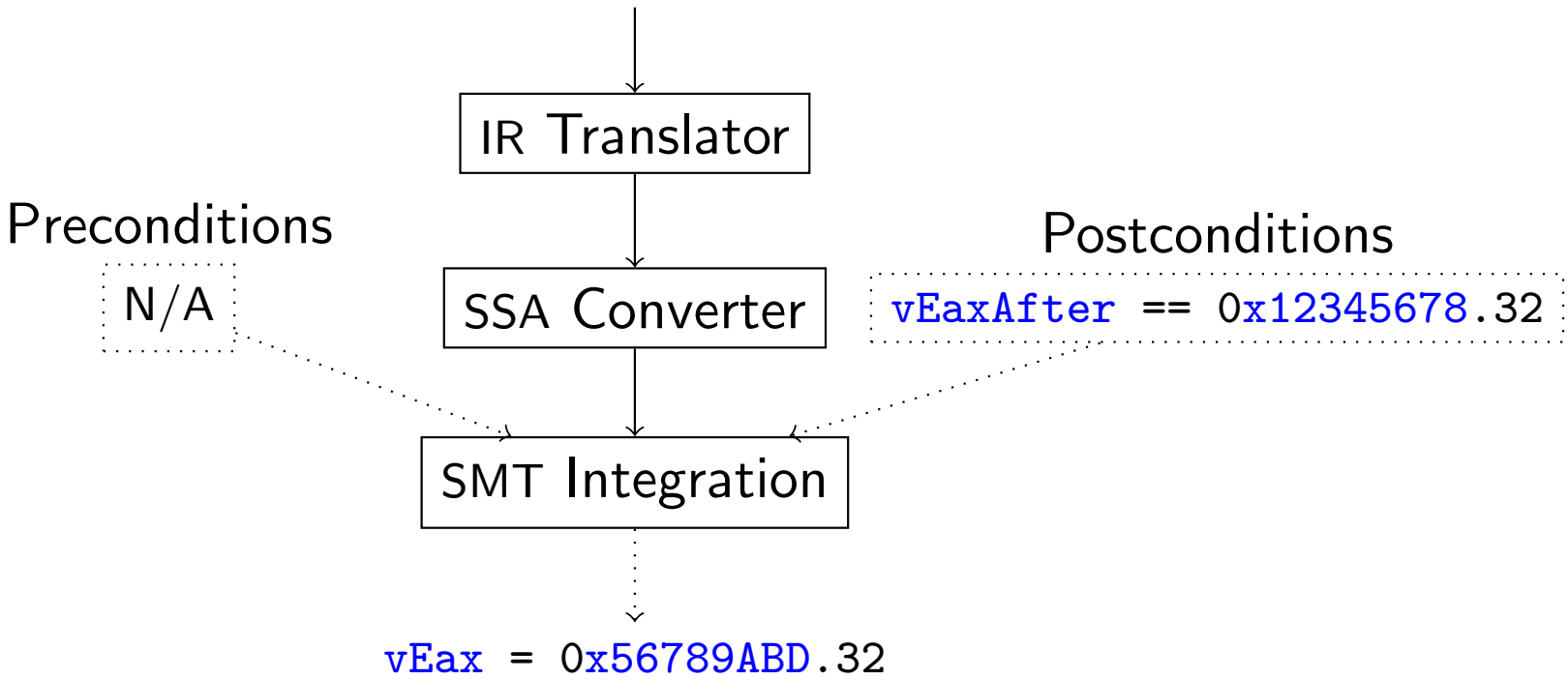
```
sub bl, bl
movzx ebx, bl
add ebx, 0BBBBBBBBh
add eax, ebx
```

- Suppose we want to know whether, after executing the code above, the final value of `eax` can ever be `12345678h`.
- This is an **input generation** problem.

Analysis of Branch-Free Code

Primary Application Archetypes: Input Generation

```
sub bl, bl
movzx ebx, bl
add ebx, 0BBBBBBBBh
add eax, ebx
```



- This is indeed possible, if initial `eax` is `56789ABDh`.

Analysis of Branch-Free Code

Primary Application Archetypes: Formal Verification

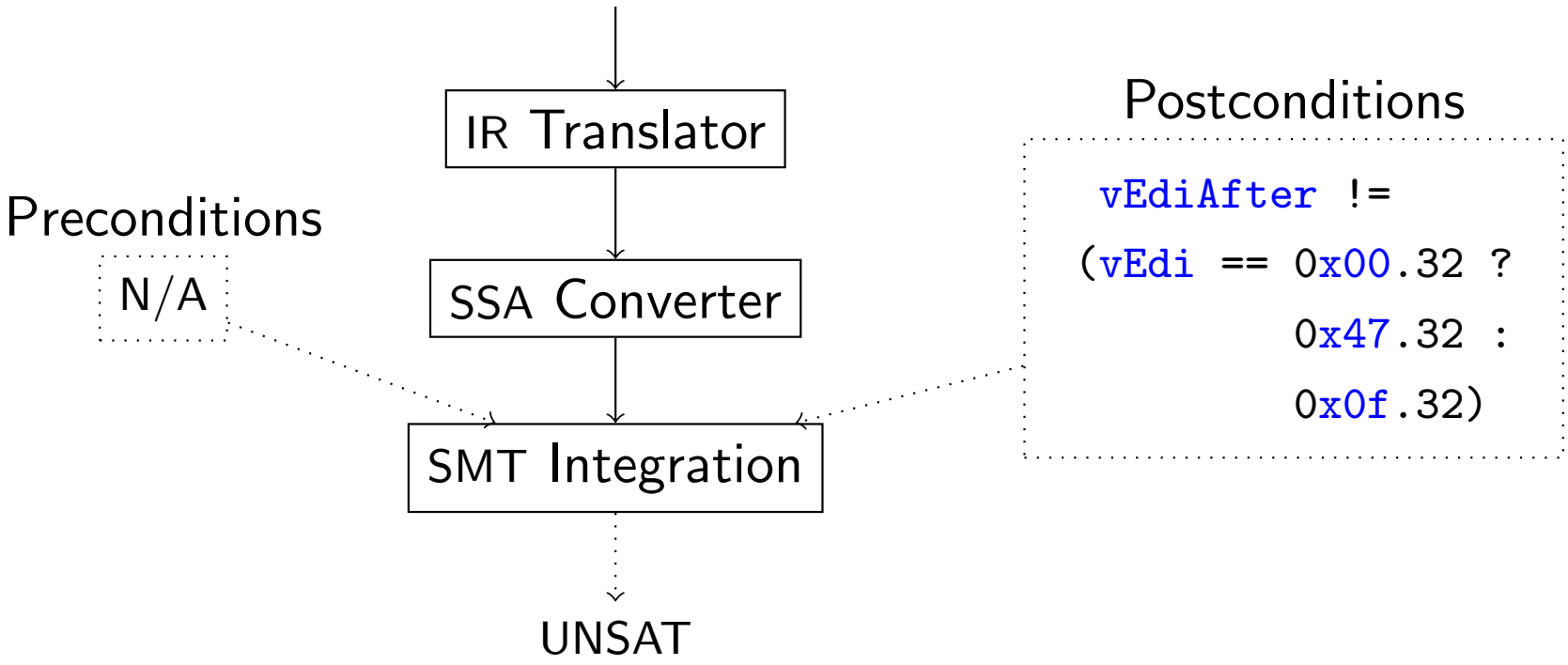
```
neg edi
sbb edi, edi
and edi, 0FFFFFFC7h
add edi, 47h
```

- Experimentation indicates that this code has the **property**:
  - If the initial value of `edi` is `0h`, then its final value is `47h`.
  - If the initial value of `edi` is not `0h`, then its final value is `0Fh`.
- Proving that the code always behaves this way is a **formal verification** problem.

Analysis of Branch-Free Code

Primary Application Archetypes: Formal Verification

```
neg edi
sbb edi, edi
and edi, 0FFFFFFC7h
add edi, 47h
```



- UNSAT proves that the code always has this property.

Analysis of Branch-Free Code

Primary Application Archetypes: Equivalence Checking

Simple Code

`test al, 01h`	`test al, 10h`	
`setnz bl`	`setnz bl`	
`mov ah, bl`	`add ah, bl`	
`test al, 02h`	`test al, 20h`	
`setnz bl`	`setnz bl`	
`add ah, bl`	`add ah, bl`	
`test al, 04h`	`test al, 40h`	
`setnz bl`	`setnz bl`	
`add ah, bl`	`add ah, bl`	
`test al, 08h`	`test al, 80h`	
`setnz bl`	`setnz bl`	
`add ah, bl`	`add ah, bl`	

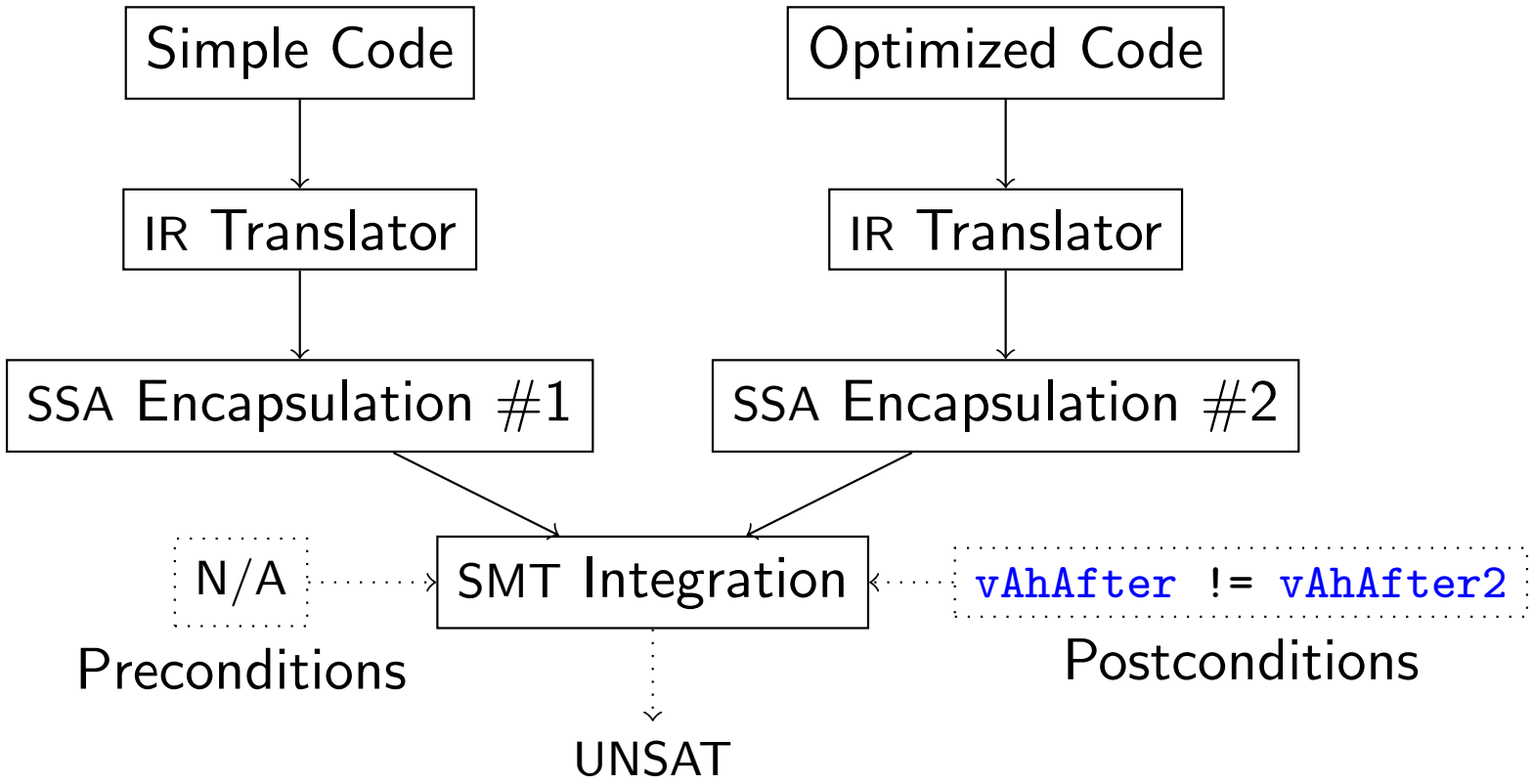
Optimized Code

`mov ah, al`	`mov ah, al`	`mov ah, al`
`and ah, 55h`	`and ah, 33h`	`and ah, 0Fh`
`shr al, 1`	`shr al, 2`	`shr al, 4`
`and al, 55h`	`and al, 33h`	`and al, 0Fh`
`add al, ah`	`add al, ah`	`add ah, al`

- We want to know whether these two computations of the population count for `al` always produce the same result in `ah`.
- This is an **equivalence checking** problem.

Analysis of Branch-Free Code

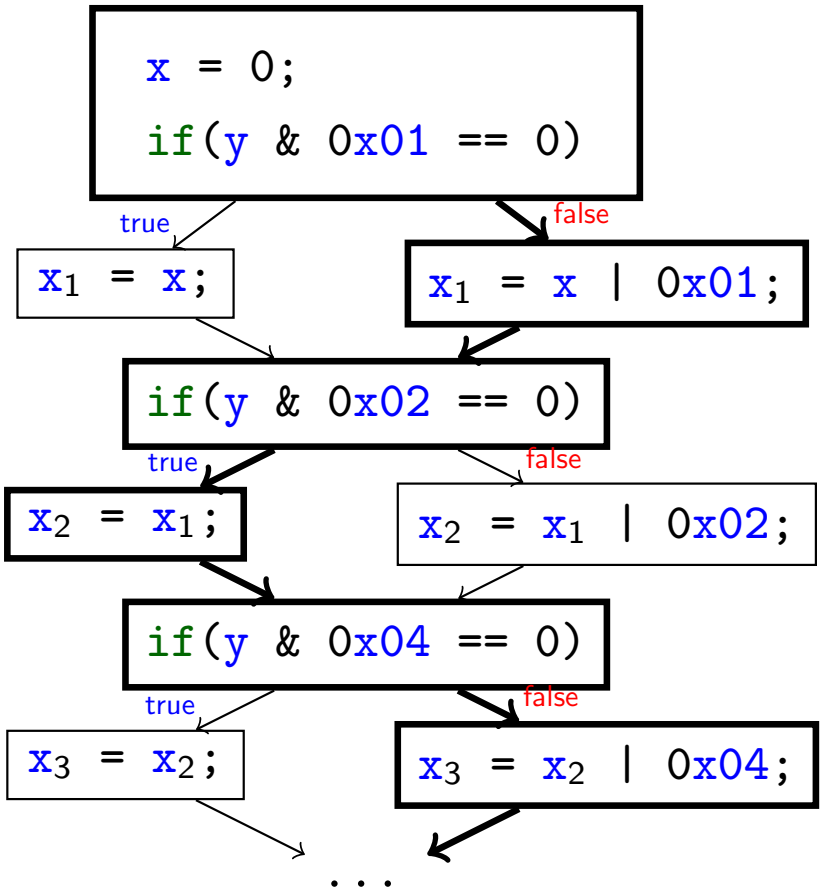
Primary Application Archetypes: Equivalence Checking



- The UNSAT result proves the equivalence of the sequences.

Concolic Execution

Recording Traces and Path Conditions



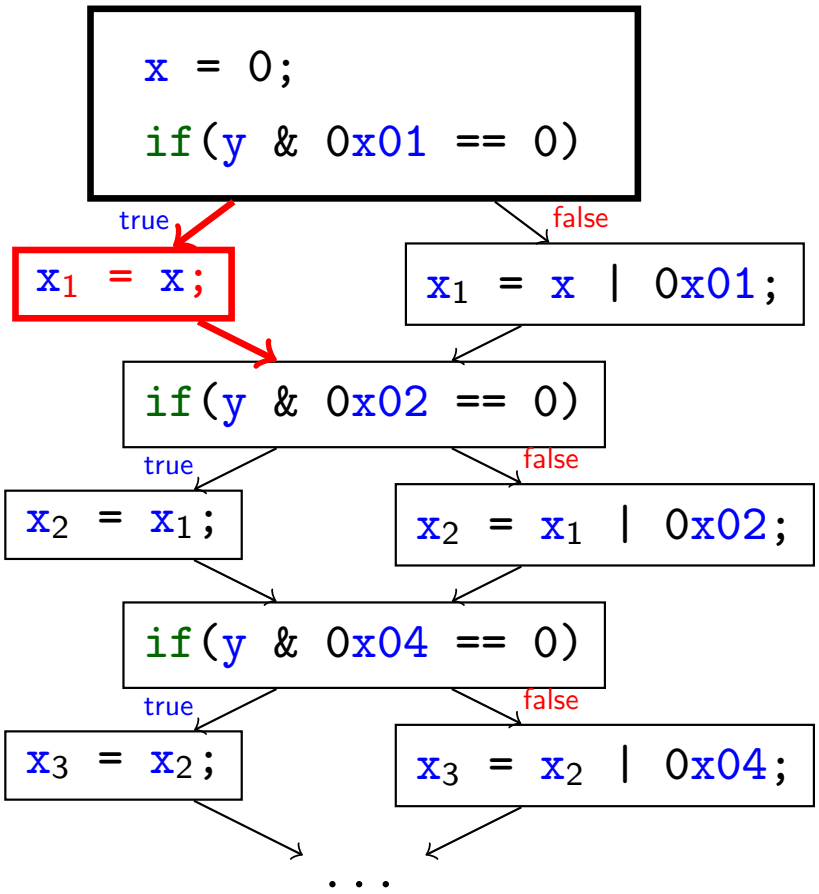
Execution Trace

```
x = 0;
Assert(y & 0x01 != 0);   ◀
x1 = x | 0x1;
Assert(y & 0x02 == 0);   ◀
x2 = x1;
Assert(y & 0x04 != 0);   ◀
x3 = x2 | 0x4;
```

- Suppose we recorded the execution shown above.
  - Path condition is indicated ◀.

Concolic Execution

Increase Code Coverage



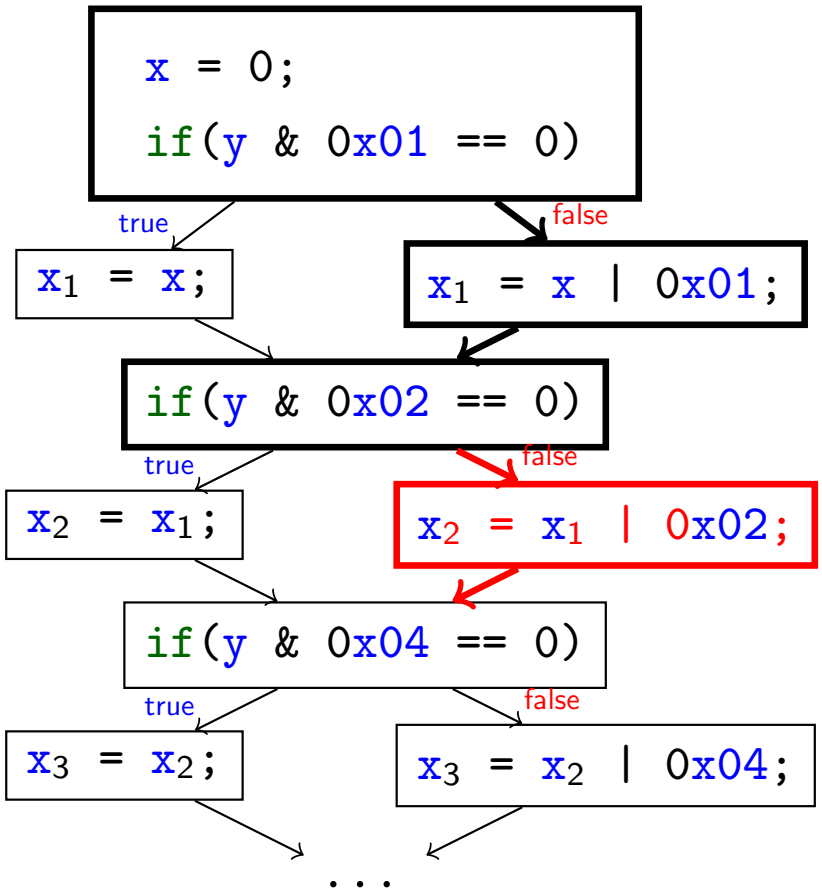
Path Condition

`Assert(y & 0x01 == 0);` ◀

- We can increase **code coverage** by trying to flip the branches along the path in sequence ◀.

Concolic Execution

Increase Code Coverage



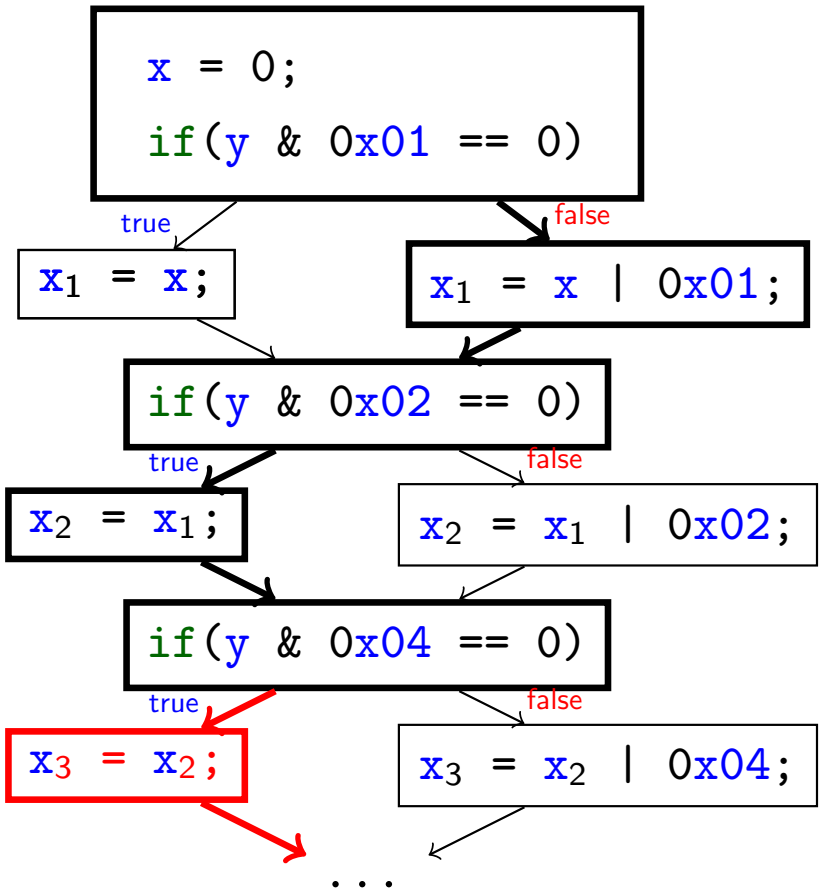
Path Condition

```
Assert(y & 0x01 != 0);
Assert(y & 0x02 != 0);   ◀
```

- We can increase **code coverage** by trying to flip the branches along the path in sequence ◀.

Concolic Execution

Increase Code Coverage



Path Condition

```
Assert(y & 0x01 != 0);
Assert(y & 0x02 == 0);
Assert(y & 0x04 == 0);   ◀
```

- We can increase **code coverage** by trying to flip the branches along the path in sequence ◀.

BitBlaze: Deviation Detection

Two programs implementing the same protocol **deviate** if:

BitBlaze: Deviation Detection

Two programs implementing the same protocol **deviate** if:

- In respose to the same network input,

Program #1

```
call recv
; many
; instructions
; execute
call send
```

$\phi_1 :=$ SMT(Trace #1)

Program #2

```
call recv
; different
; instructions
; execute
call send
```

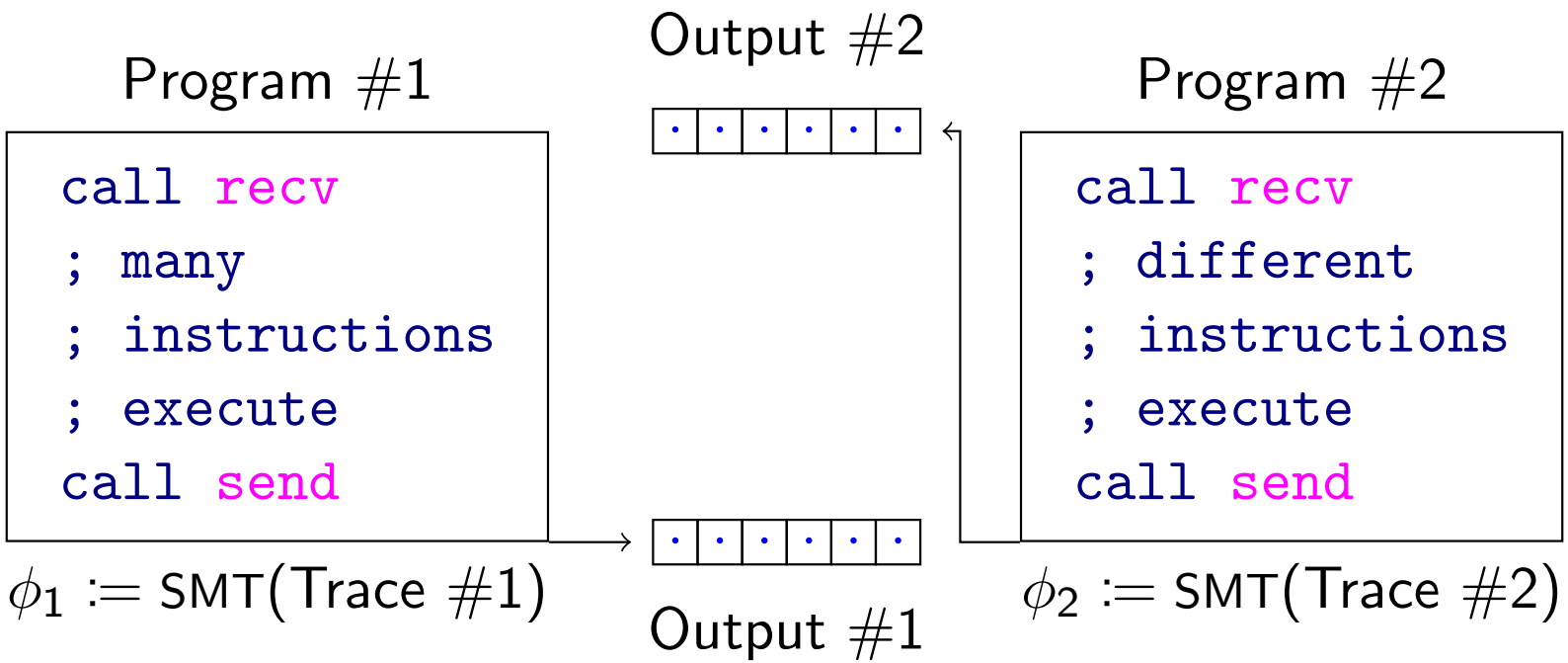
$\phi_2 :=$ SMT(Trace #2)

Step #1: Record traces of both programs between `recv` and `send` and convert them to SMT formulas ϕ_1, ϕ_2.

BitBlaze: Deviation Detection

Two programs implementing the same protocol **deviate** if:

- In respose to the same network input,
- They respond with network output,

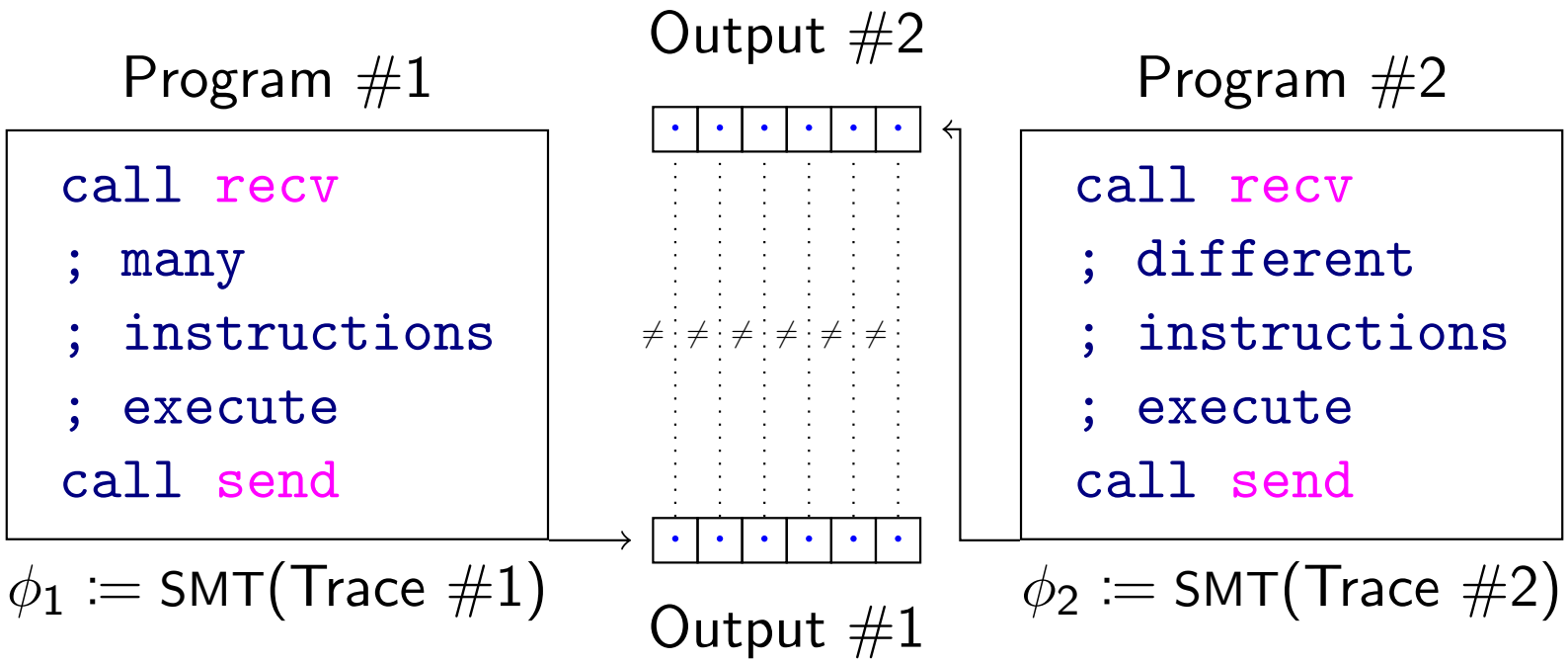


Step #2: The addresses of the output buffers will be in `[esp+X]` at the ends (before the call to `send`).

BitBlaze: Deviation Detection

Two programs implementing the same protocol **deviate** if:

- In respose to the same network input,
- They respond with network output,
- And the responses are different.



Step #3: Output difference is expressed by the formula

`(out1[0] != out2[0] || ... || out1[len] != out2[len]).`