Synthesizing Instruction Selection Rewrite Rules from RTL using SMT

Ross Daly, Caleb Donovick,
Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze,
Priyanka Raina, Clark Barrett, Pat Hanrahan

A New Golden Age for Computer Architecture

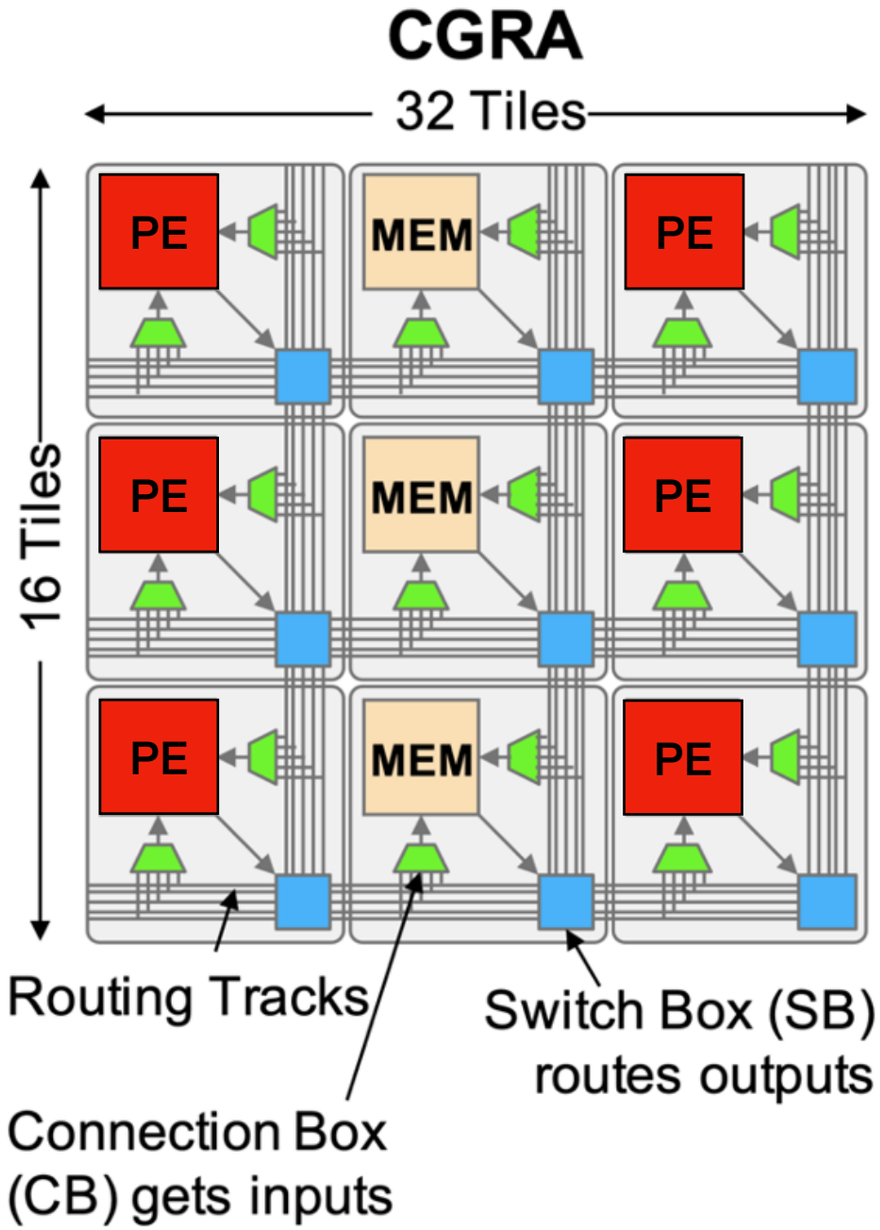
–

Patterson & Hennessey

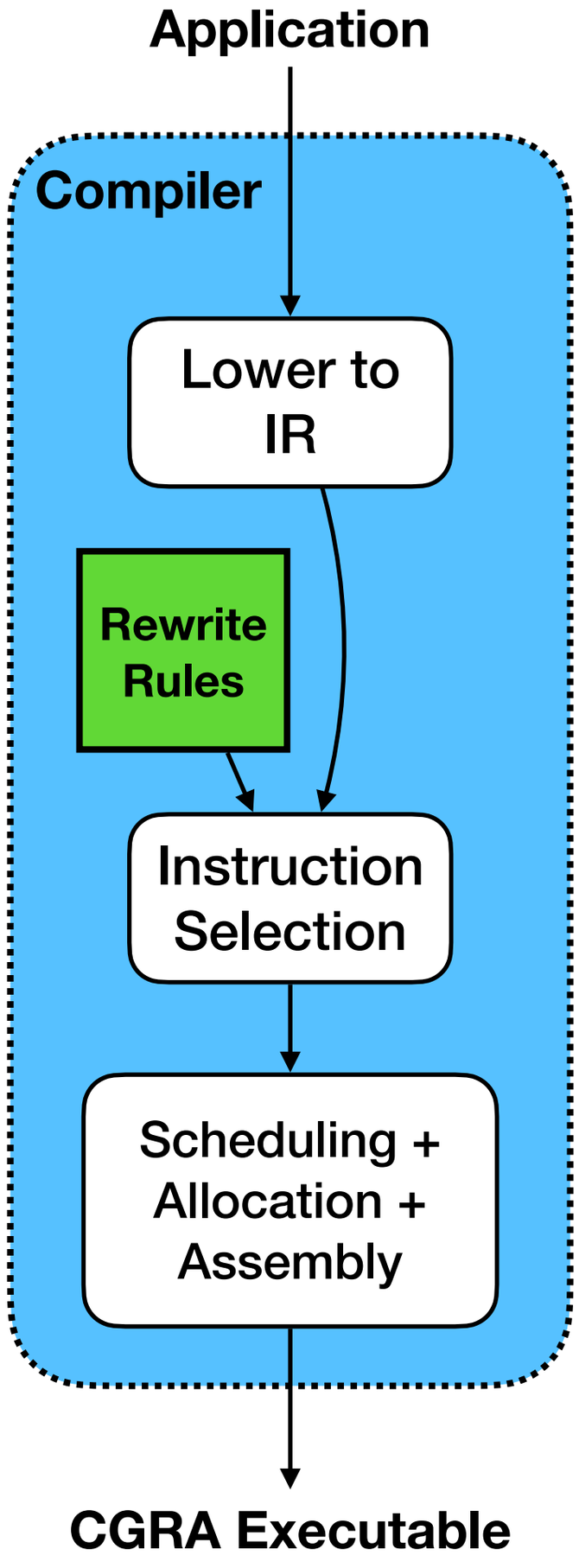
Need: *A New Golden Age for Compilers*

Coarse Grain Reconfigurable Array (CGRA)

- Analogous to an FPGA
- Array of Processing Elements (PE) and local memory (MEM)
- Configurable Spatial Routing Fabric

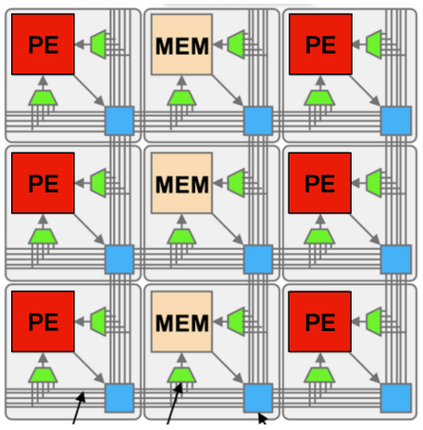


Compiling Applications

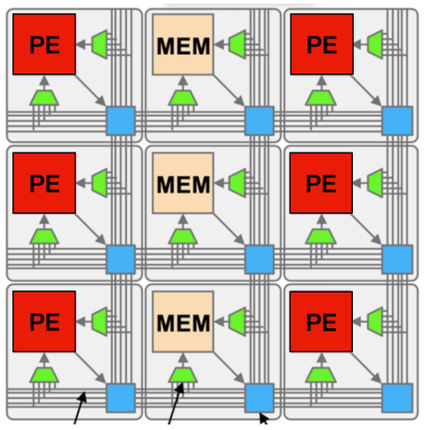


Create Domain Specific Accelerators by ***Specializing*** PEs

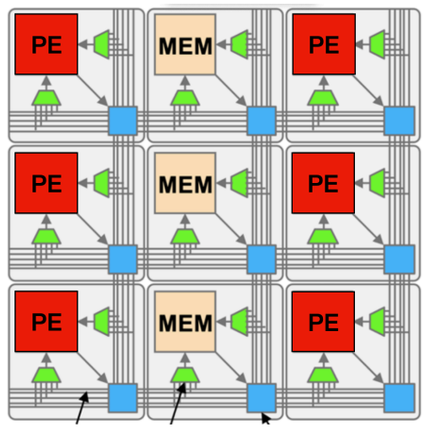
ML CGRA



Cryptographic CGRA



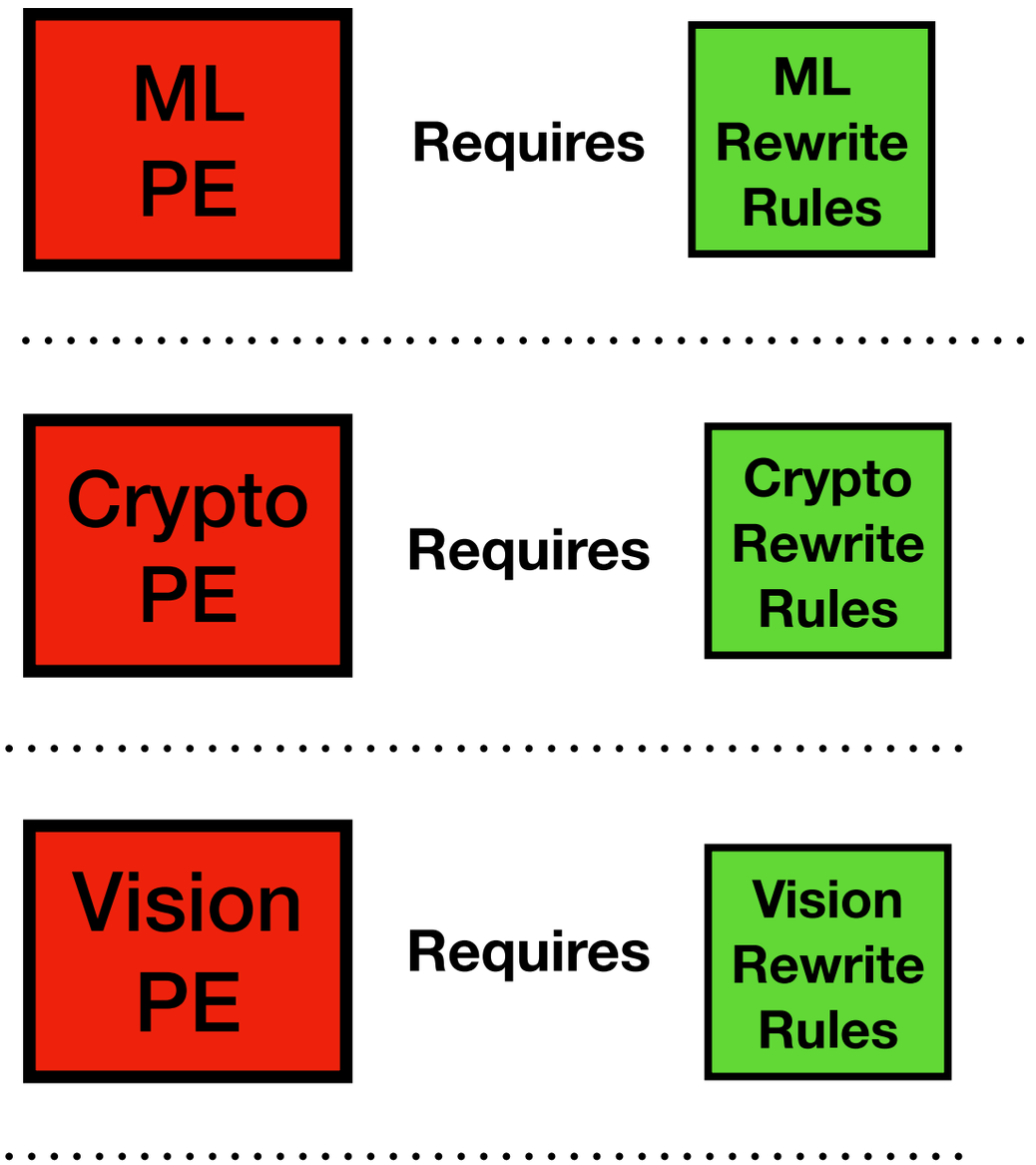
Computer Vision CGRA



...

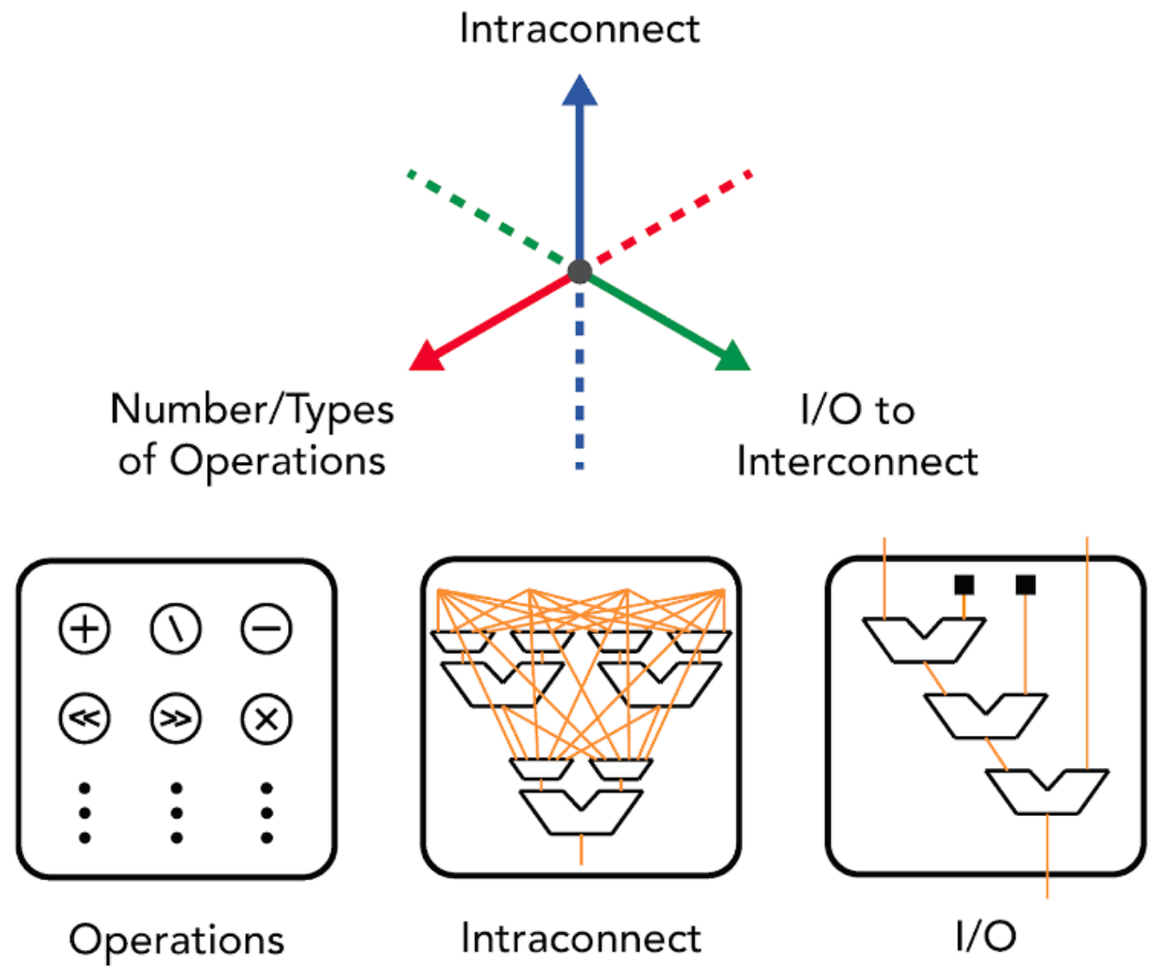
Different PEs Require Different Rewrite Rules

- Each PE version requires own set of rewrite rules.
- Writing rewrite rules is labor intensive
- Discrepancies between RTL and compiler are a common source of bugs



PE Design Space Exploration

- **For a domain**: Large space of possible PE designs
- Generate 100s of PEs
- Benchmark and compare each PE
- **Requires a working compiler per PE!**



RTL of
Target Architecture

PE

IR
Instructions

Rewrite Rule
Generation

Application

Compiler

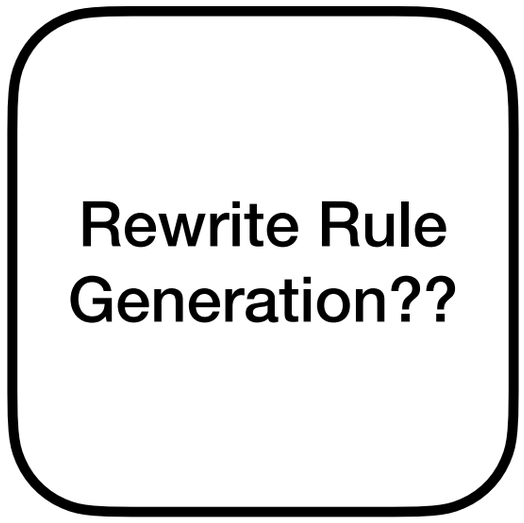
Lower to
IR

Rewrite
Rules

Instruction
Selection

Scheduling +
Allocation +
Assembly

CGRA Executable



Leverage the power of SMT solvers to synthesize rewrite rules!

Instruction Selection

```
Kernel(x: BV[8], y: BV[8])
  T0 = IRMul(x, y)
  T1 = IRAdd(T0, x)
  T2 = IRSub(T1, y)
  …
  return T16
```

Instruction Selection

Rewrite Rule Table

IR Pattern	*PE Instruction*	*Cost*
IRAdd(a,b)	**PEAdd(a,b)**	**5**
IRSub(a,b)	**PESub(a,b)**	**5**
IRAdd(a, IRMul(b,c))	**PEFMA(c,b,a)**	**2**
...	...	...

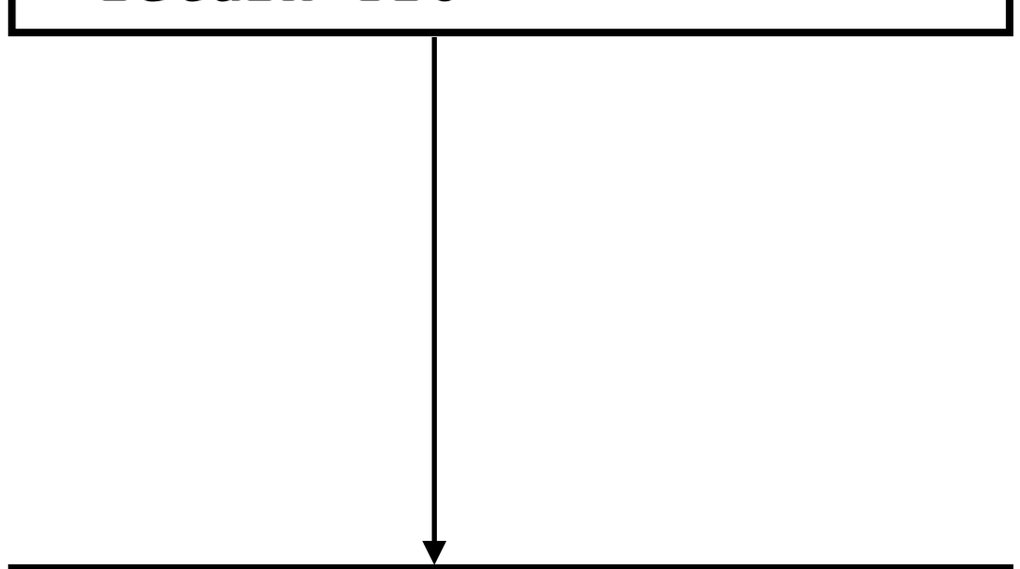
```
Kernel(x: BV[8], y: BV[8])
  T0 = IRMul(x, y)
  T1 = IRAdd(T0, x)
  T2 = IRSub(T1, y)
  …
  return T16
```

Instruction Selection

Rewrite Rule Table

IR Pattern	*PE Instruction*	*Cost*
`IRAdd(a,b)`	`PEAdd(a,b)`	5
`IRSub(a,b)`	`PESub(a,b)`	5
`IRAdd(a, IRMul(b,c))`	`PEFMA(c,b,a)`	2
...	...	...

```
Kernel(x: BV[8], y: BV[8])
  T0 = IRMul(x, y)
  T1 = IRAdd(T0, x)
  T2 = IRSub(T1, y)
  …
  return T16
```



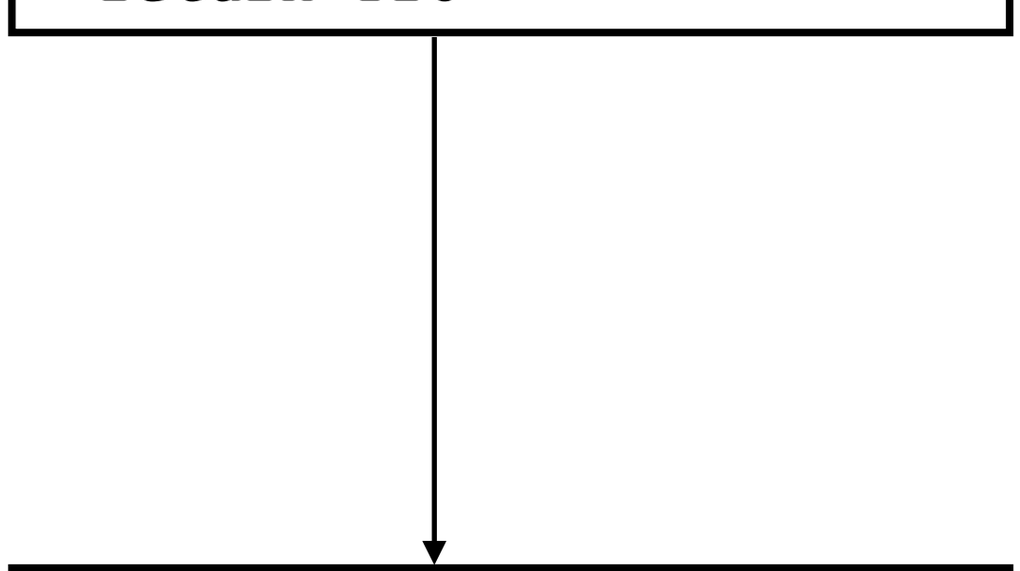
```
Kernel(x: BV[8], y: BV[8])
  T0 = PEFMA(x, y, x)
  T1 = PESub(T0, y)
  …
  return T12
```

Instruction Selection

Rewrite Rule Table

IR Pattern	*PE Instruction*	*Cost*
IRAdd(a,b)	**PEAdd(a,b)**	**5**
IRSub(a,b)	**PESub(a,b)**	**5**
IRAdd(a, IRMul(b,c))	**PEFMA(c,b,a)**	**2**
...	...	...

```
Kernel(x: BV[8], y: BV[8])
  T0 = IRMul(x, y)
  T1 = IRAdd(T0, x)
  T2 = IRSub(T1, y)
  …
  return T16
```



```
Kernel(x: BV[8], y: BV[8])
  T0 = PEFMA(x, y, x)
  T1 = PESub(T0, y)
  …
  return T12
```

Instruction Selection

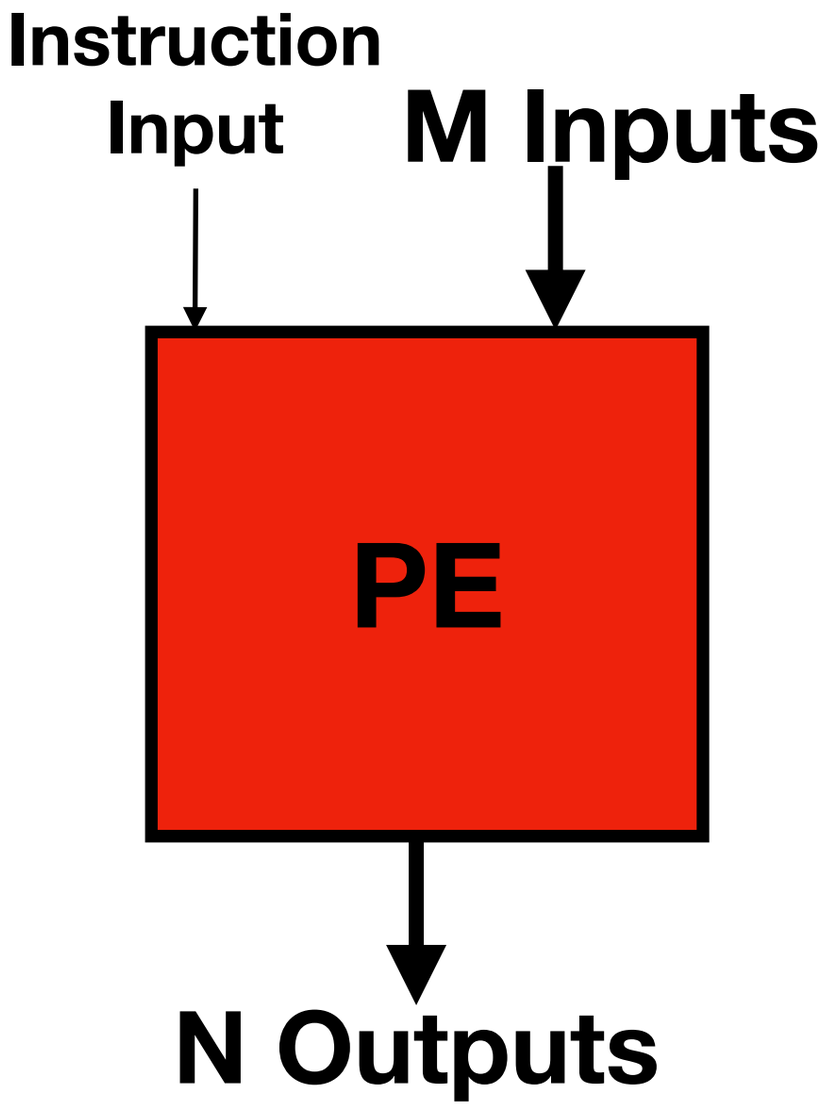
Rewrite Rule Table

IR Pattern	*PE Instruction*	*Cost*
IRAdd(a,b)	PEAdd(a,b)	5
IRSub(a,b)	PESub(a,b)	5
IRAdd(a, IRMul(b,c))	PEFMA(c,b,a)	2
...	...	...

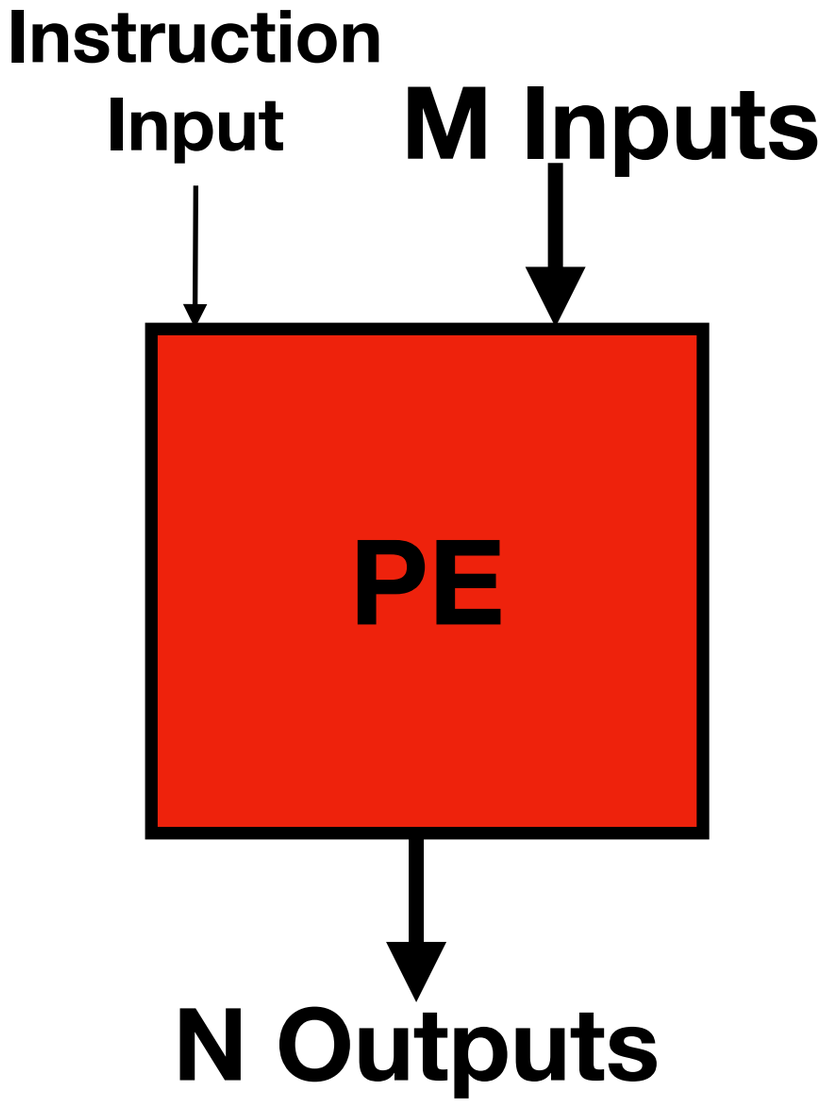
```
Kernel(x: BV[8], y: BV[8])
  T0 = IRMul(x, y)
  T1 = IRAdd(T0, x)
  T2 = IRSub(T1, y)
  …
  return T16
```

```
Kernel(x: BV[8], y: BV[8])
  T0 = PEFMA(x, y, x)
  T1 = PESub(T0, y)
  …
  return T12
```

PE RTL

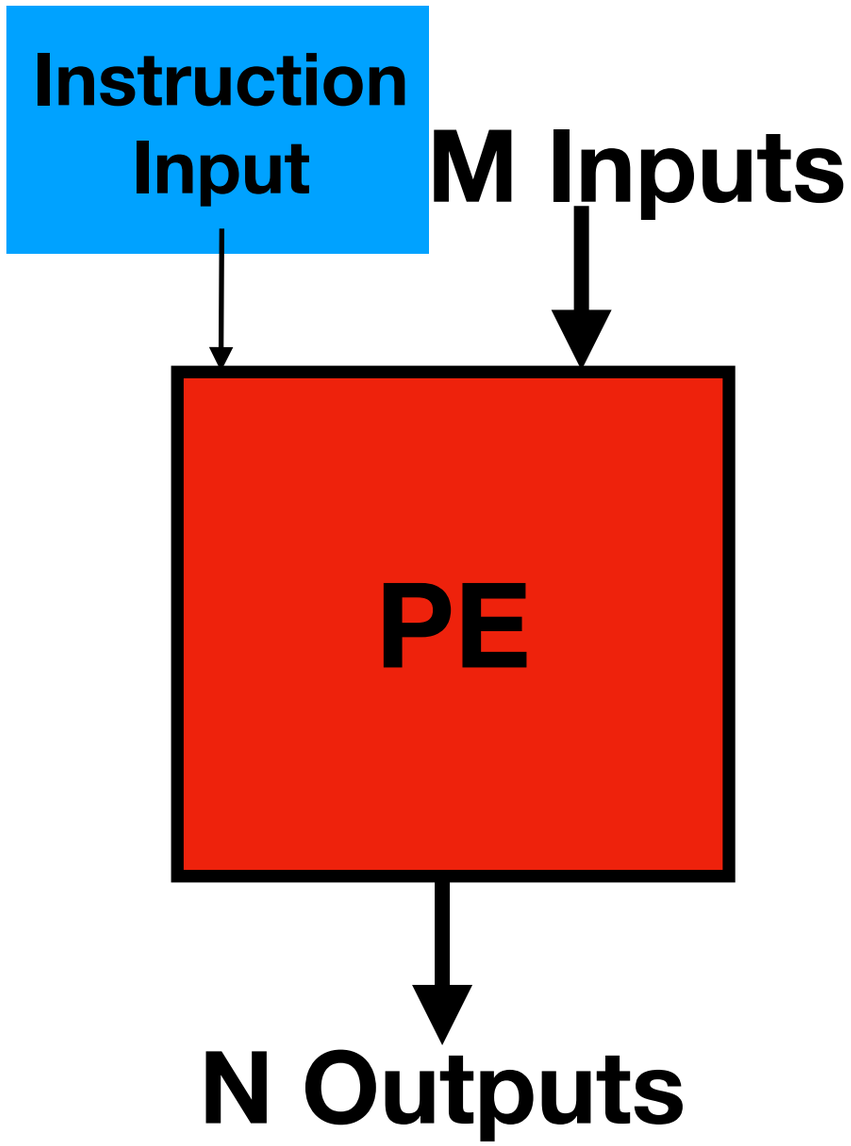


PE Example



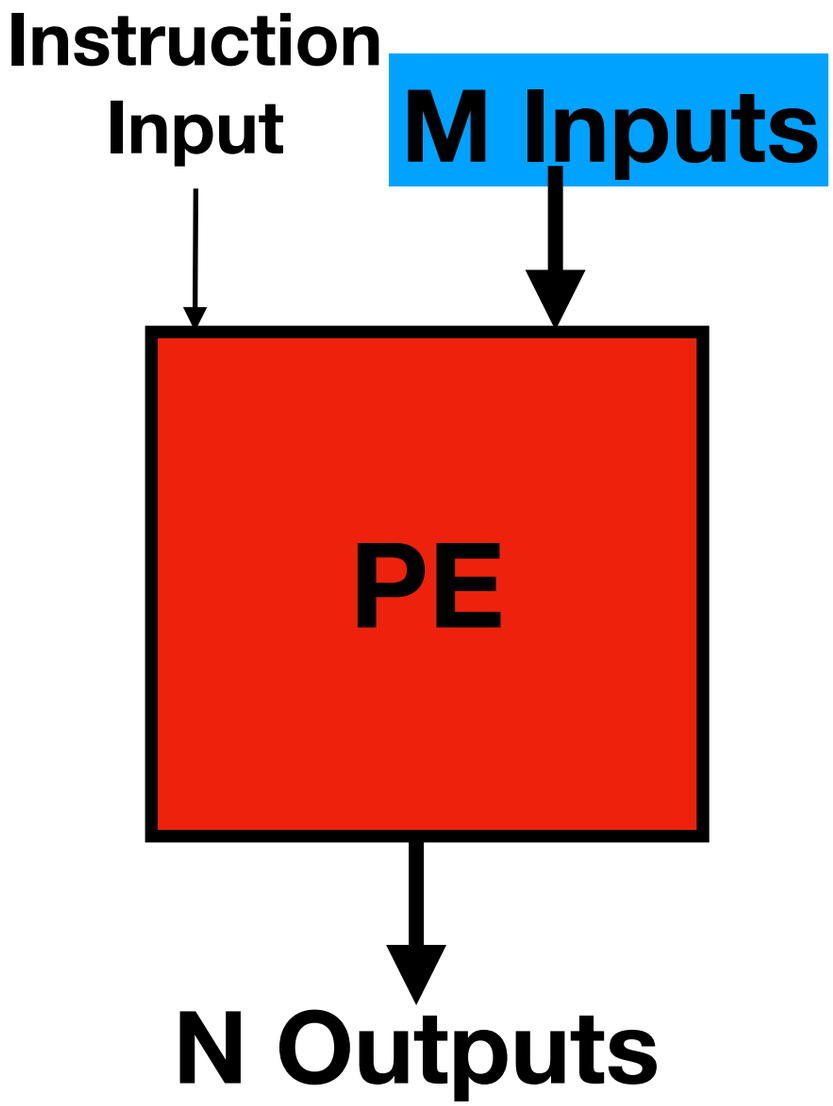

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    if I == 0:
        return a + b
    elif I == 1:
        return b - a
    elif I == 2:
        return -a
    elif I == 4:
        return FloatMul(a, b)
```

PE Example



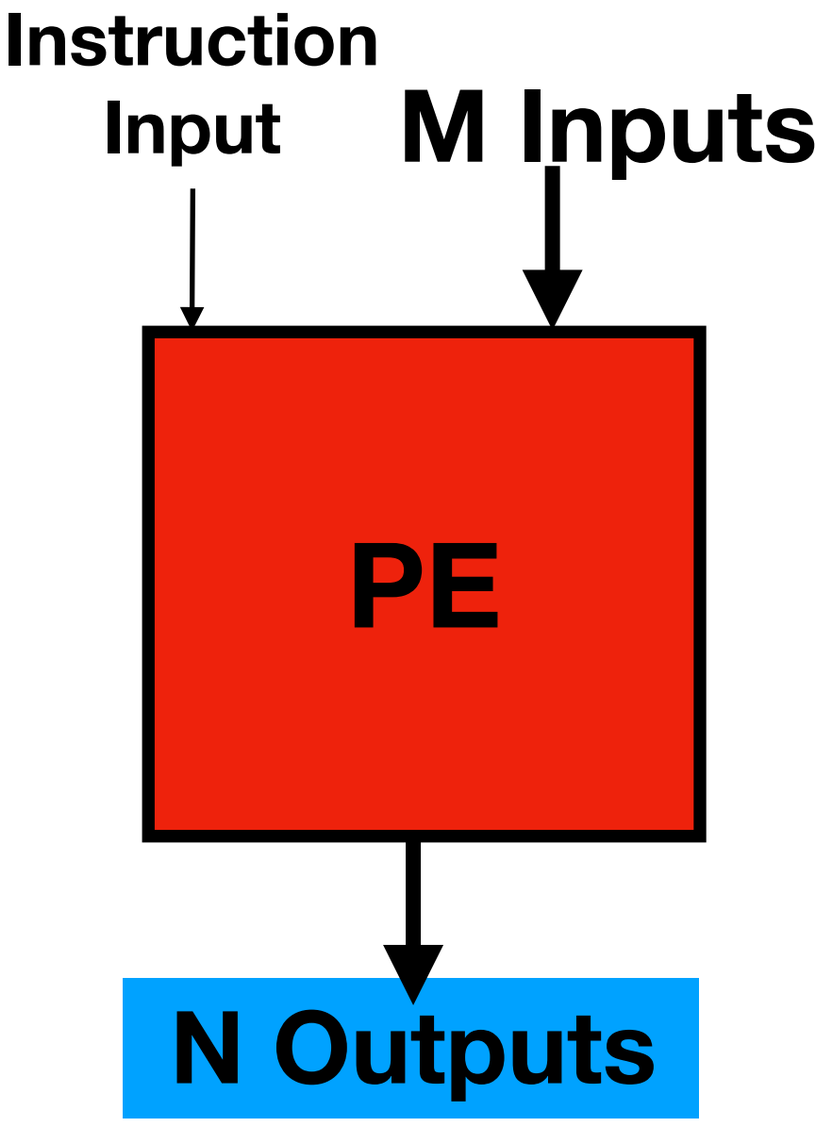

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    if I == 0:
        return a + b
    elif I == 1:
        return b - a
    elif I == 2:
        return -a
    elif I == 4:
        return FloatMul(a, b)
```

PE Example



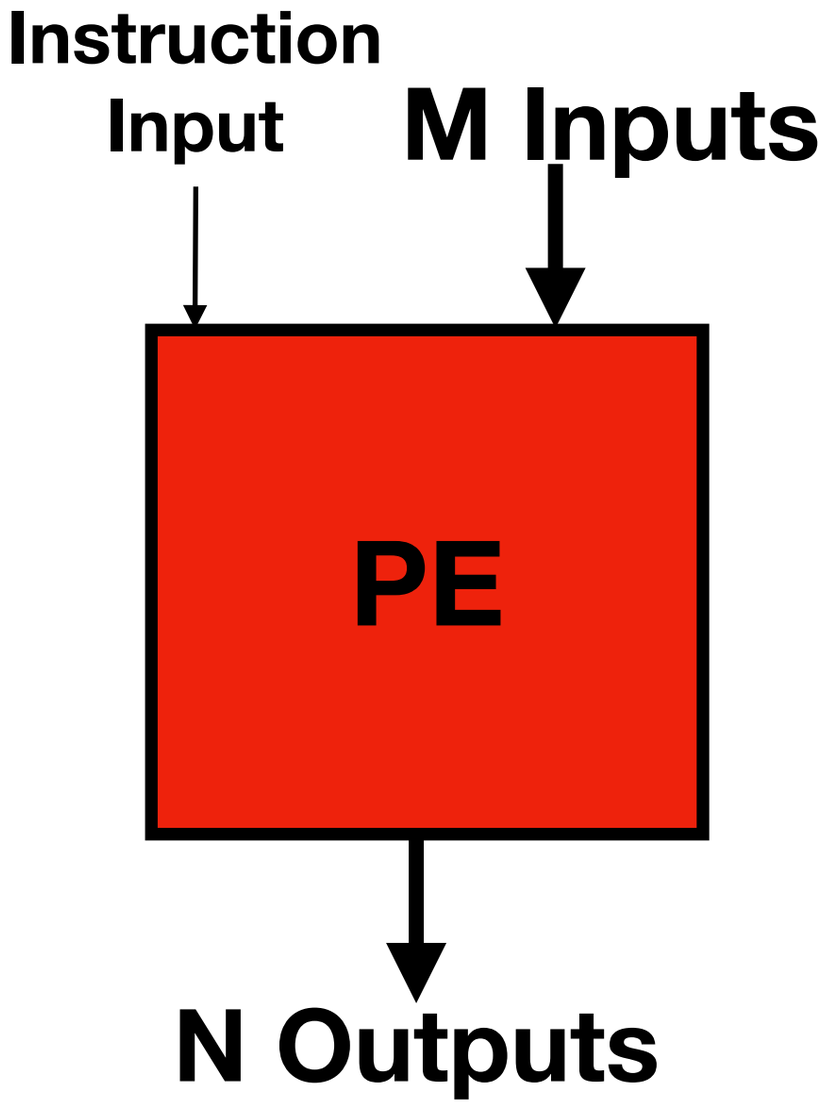

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    if I == 0:
        return a + b
    elif I == 1:
        return b - a
    elif I == 2:
        return -a
    elif I == 4:
        return FloatMul(a, b)
```

PE Example




```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    if I == 0:
        return a + b
    elif I == 1:
        return b - a
    elif I == 2:
        return -a
    elif I == 4:
        return FloatMul(a, b)
```

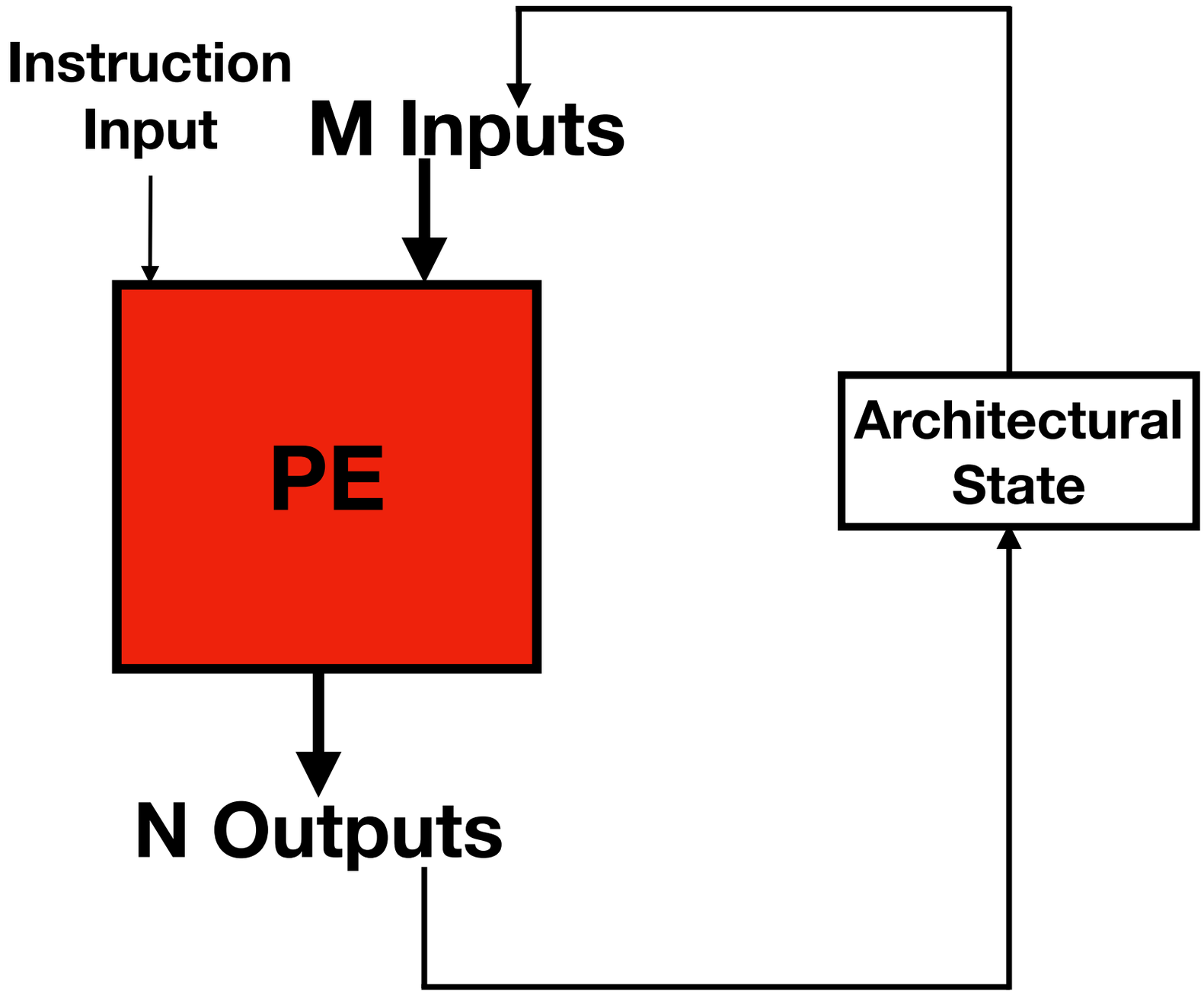
PE Example



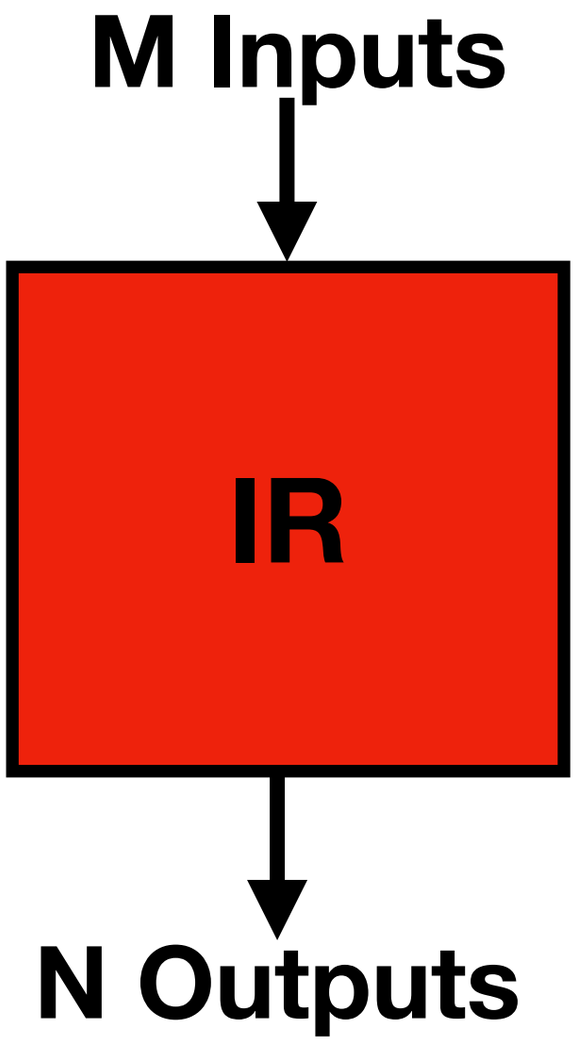

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    if I == 0:
        return a + b
    elif I == 1:
        return b - a
    elif I == 2:
        return -a
    elif I == 4:
        return FloatMul(a, b)
```

PE(I=0, a, b) = PEAdd(a, b)

PE Architectural State



IR Semantics




```
def IRAdd(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x + y
```

Rewrite Rule Formalization

Example: Addition

```
def IRAdd(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x + y
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    if I == 0:
        return a + b
    elif I == 1:
        ...
    ...
```

Rewrite Rule:

$$(IRAdd,\ PE,\ I = 0)$$

Example: Addition

```
def IRAdd(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x + y
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    if I == 0:
        return a + b
    elif I == 1:
        ...
    ...
```

Rewrite Rule:

$$(IRAdd,\ PE,\ I = 0)$$

$$\forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$IRAdd(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{0},\ \mathbf{x},\ \mathbf{y})$$

Rewrite Rule Formalization: Basic

Rewrite Rule:

$(IR,\ PE,\ I)$

$$\forall \vec{\mathbf{X}} .$$

$$IR(\vec{\mathbf{X}}) = PE(I, \vec{\mathbf{X}})$$

Example: Subtraction

```
def IRSub(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x - y
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 1:
        return b - a
    ...
```

Example: Subtraction

```
def IRSub(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x - y
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 1:
        return b - a
    ...
```

Example: Subtraction

```
def IRSub(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x - y
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 1:
        return b - a
    ...
```

Rewrite Rule:

$$(IRSub,\ PE,\ I = 1,\ \textbf{Swapped}??)$$

Example: Subtraction

```
def IRSub(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x + y
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 1:
        return b - a
    ...
```

```
def Bind(
    x: BV[8],
    y: BV[8]
) -> (BV[8], BV[8])
    return y, x
```

Rewrite Rule:

$$(IRSub,\ PE,\ I = 1,\ Bind)$$

Example: Subtraction

```
def IRSub(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x + y
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 1:
        return b - a
    ...
```

```
def Bind(
    x: BV[8],
    y: BV[8]
) -> (BV[8], BV[8)
    return y, x
```

Rewrite Rule:

$$(IRSub,\ PE,\ I = 1,\ Bind)$$

$$\forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{1}, Bind(\mathbf{x},\ \mathbf{y}))$$

Rewrite Rule Formalization: Input and Output Permutations

Rewrite Rule:

$$(IR,\ PE,\ I,\ b^{in},\ b^{out})$$

$$\forall\, \vec{\mathbf{X}}\ .$$

$$IR(\vec{\mathbf{X}}) = b^{out}\Big(PE(I,\ b^{in}(\vec{\mathbf{X}}))\Big)$$

Example: Negation

```
def IRNeg(
    x: BV[8]
) -> BV[8]
    return -x
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
   ...
   elif I == 2:
       return -a
   ...
```

```
def Bind(
    x: BV[8],
    b: BV[8]
) -> (BV[8], BV[8)
    return x, b
```

Rewrite Rule:

$$(IRNeg,\ PE,\ I = 2,\ Bind)$$

Example: Negation

```
def IRNeg(
    x: BV[8]
) -> BV[8]
    return -x
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
   ...
   elif I == 2:
       return -a
   ...
```

```
def Bind(
    x: BV[8],
    b: BV[8]
) -> (BV[8], BV[8)
    return x, b
```

Rewrite Rule:

$$(IRNeg,\ PE,\ I = 2,\ Bind)$$

$$\forall\, \mathbf{x}, \mathbf{b}.$$

$$IRNeg(\mathbf{x}) = PE(\mathbf{2},\ Bind(\mathbf{x},\ \mathbf{b}))$$

Rewrite Rule Formalization: Mismatch in Arity

Rewrite Rule:

$$(IR,\ PE,\ I,\ b^{in},\ b^{out})$$

$$\forall\, \vec{\mathbf{X}},\ \vec{\mathbf{Y}}\ .$$

$$IR(\vec{\mathbf{X}}) = b^{out}\Big(PE(I,\ b^{in}(\vec{\mathbf{X}},\ \vec{\mathbf{Y}}))\Big)$$

How to Synthesize the Rewrite Rule?

Rewrite Rule:

$$(IR,\ PE,\ I,\ b^{in},\ b^{out})$$

$$\forall\, \vec{\mathbf{X}},\ \vec{\mathbf{Y}}\ .$$

$$IR(\vec{\mathbf{X}}) = b^{out}\Big(PE(I,\ b^{in}(\vec{\mathbf{X}},\ \vec{\mathbf{Y}}))\Big)$$

Synthesis Query

Rewrite Rule:

$$(IR,\ PE,\ I,\ b^{in},\ b^{out})$$

$$\exists\ \mathbf{I},\ \mathbf{b}^{\mathbf{in}},\ \mathbf{b}^{\mathbf{out}}\ .\ \forall\ \vec{\mathbf{X}},\ \vec{\mathbf{Y}}\ .$$

$$IR(\vec{\mathbf{X}}) = \mathbf{b}^{\mathbf{out}}\left(PE(\mathbf{I},\ \mathbf{b}^{\mathbf{in}}(\vec{\mathbf{X}},\ \vec{\mathbf{Y}}))\right)$$

Parametric IR instructions

```
def IRAddImm(
    imm: BV[8],
    x: BV[8]
) -> BV[8]
    return x + imm
```

```
def PE(
    I: (BV[4], BV[8]),
    a: BV[8],
    b: BV[8]
) -> BV[8]
    op, imm = decode(I)
    ...
    if op == 3:
        return a + imm
    ...
```

Parametric IR instructions

```
def IRAddImm(
    imm: BV[8],
    x: BV[8]
) -> BV[8]
    return x + imm
```

```
def PE(
    I: (BV[4], BV[8]),
    a: BV[8],
    b: BV[8]
) -> BV[8]
    op, imm = decode(I)
    ...
    if op == 3:
        return a + imm
    ...
```

Parametric PEs

```
def IRAddImm(
    imm: BV[8],
    x: BV[8]
) -> BV[8]
    return x + imm
```

```
def PE(
    I: (BV[4], BV[8]),
    a: BV[8],
    b: BV[8]
) -> BV[8]
    op, imm = decode(I)
    ...
    if op == 3:
        return a + imm
    ...
```

Synthesis with Parametric IR

```
def IRAddImm(
    imm: BV[8],
    x: BV[8]
) -> BV[8]
    return x + imm
```

```
def PE(
    I: (BV[4], BV[8]),
    a: BV[8],
    b: BV[8]
) -> BV[8]
    op, imm = decode(I)
    ...
    if op == 3:
        return a + imm
    ...
```

$$\forall\ \mathbf{imm}\ .\ \exists\ \mathbf{I},\ \mathbf{Bind}\ .\ \forall\ \mathbf{x},\ \mathbf{y}\ .$$

$$\mathit{IRAddImm}(\mathbf{imm},\ \mathbf{x})$$

$$=$$

$$\mathit{PE}(\mathbf{I},\ \mathbf{Bind}(\mathbf{x},\ \mathbf{y}))$$

Synthesis with Parametric IR (Skolemized)

```
def IRAddImm(
    imm: BV[8],
    x: BV[8]
) -> BV[8]
    return x + imm
```

```
def PE(
    I: (BV[4], BV[8]),
    a: BV[8],
    b: BV[8]
) -> BV[8]
    op, imm = decode(I)
    ...
    if op == 3:
        return a + imm
    ...
```

$$\exists\ \mathbf{I},\ \mathbf{Bind}\ .\ \forall\ \mathbf{imm},\ \mathbf{x},\ \mathbf{y}\ .$$

$$\mathit{IRAddImm}(\mathbf{imm},\ \mathbf{x})$$

$$=$$

$$\mathit{PE}(\mathbf{I}(\mathbf{imm}),\ \mathbf{Bind}(\mathbf{x},\ \mathbf{y}))$$

Synthesis with Parametric IR (Skolemized)

```
def IRAddImm(
    imm: BV[8],
    x: BV[8]
) -> BV[8]
    return x + imm
```

```
def PE(
    I: (BV[4], BV[8]),
    a: BV[8],
    b: BV[8]
) -> BV[8]
    op, imm = decode(I)
    ...
    if op == 3:
        return a + imm
    ...
```

$$\exists\ \mathbf{I},\ \mathbf{Bind}\ .\ \forall\ \mathbf{imm},\ \mathbf{x},\ \mathbf{y}\ .$$

$$\mathit{IRAddImm}(\mathbf{imm},\ \mathbf{x})$$

$$=$$

$$\mathit{PE}(\mathbf{I(imm)},\ \mathbf{Bind}(\mathbf{x},\ \mathbf{y}))$$

IR+PE with Complex Ops

```
def IRFmul(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return FloatMul(x, y)
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 4:
        return FloatMul(a, b)
    ...
```

IR+PE with Complex Ops

```
def IRFmul(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return FloatMul(x, y)
```

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 4:
        return FloatMul(a, b)
    ...
```

Abstracting Complex Ops

```
def IRFmul(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return FloatMul(x, y)
```

Abstracting Complex Ops

```
def IRFmul(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return FloatMul(x, y)
```

```
def IRFmulAbs(
    F: (BV[8],BV[8])-> BV[8],
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return F(x, y)
```

Abstracting Complex Ops

```
def IRFmul(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return FloatMul(x, y)
```

```
def IRFmulAbs(
    F: (BV[8],BV[8])-> BV[8],
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return F(x, y)
```

Abstracting Complex Ops

```
def IRFmul(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return FloatMul(x, y)
```

```
def IRFmulAbs(
    F: (BV[8],BV[8])-> BV[8],
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return F(x, y)
```

$$\forall\, \mathbf{x},\ \mathbf{y}\ .$$
$$IRFmul(\mathbf{x}, \mathbf{y}) = IRFmulAbs(\mathbf{FloatMul},\ \mathbf{x},\ \mathbf{y})$$

Abstracting Complex Ops

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 4:
        return FloatMul(a, b)
    ...
```

```
def PEAbs(
    F: (BV[8],BV[8])-> BV[8],
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 4:
        return F(a, b)
    ...
```

$$\forall \mathbf{I}, \mathbf{a}, \mathbf{b} .$$

$$PE(\mathbf{I}, \mathbf{a}, \mathbf{b}) = PEAbs(\mathbf{FloatMul}, \mathbf{I}, \mathbf{a}, \mathbf{b})$$

Synthesis with Abstracted Functions

```
def IRFmulAbs(
    F: (BV[8],BV[8])-> BV[8],
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return F(x, y)
```

$$\exists\ \mathbf{I},\ \mathbf{Bind}\ .\ \forall\ \mathbf{F},\ \mathbf{x},\ \mathbf{y}\ .$$

$$\mathit{IRFmulAbs}(\mathbf{F},\ \mathbf{x},\ \mathbf{y})$$

$$=$$

$$\mathit{PEAbs}(\mathbf{F},\ \mathbf{I},\ \mathbf{Bind}(\mathbf{x},\ \mathbf{y}))$$

```
def PEAbs(
    F: (BV[8],BV[8])-> BV[8],
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 4:
        return F(a, b)
    ...
```

Final* Synthesis Query

Rewrite Rule:

$$(IR,\ PE,\ I,\ b^{in},\ b^{out})$$

$$\exists\ \mathbf{I},\ \mathbf{b^{in}},\ \mathbf{b^{out}}\ .\ \forall\ \vec{\mathbf{F}},\ \vec{\mathbf{C}},\ \vec{\mathbf{X}},\ \vec{\mathbf{Y}}\ .$$

$$IRAbs(\vec{\mathbf{F}}, \vec{\mathbf{C}}, \vec{\mathbf{X}})$$

$$=$$

$$\mathbf{b^{out}}\left(PEAbs(\vec{\mathbf{F}},\ \mathbf{I}(\vec{\mathbf{C}}),\ \mathbf{b^{in}}(\vec{\mathbf{X}},\ \vec{\mathbf{Y}}))\right)$$

2nd-Order Query!

Rewrite Rule:

$$(IR,\ PE,\ I,\ b^{in},\ b^{out})$$

$$\exists\ \mathbf{I},\ \mathbf{b^{in}},\ \mathbf{b^{out}}\ .\ \forall\ \vec{\mathbf{F}},\ \vec{\mathbf{C}},\ \vec{\mathbf{X}},\ \vec{\mathbf{Y}}\ .$$

$$IRAbs(\vec{\mathbf{F}}, \vec{\mathbf{C}}, \vec{\mathbf{X}})$$

$$=$$

$$\mathbf{b^{out}}\left(PEAbs(\vec{\mathbf{F}},\ \mathbf{I}(\vec{\mathbf{C}}),\ \mathbf{b^{in}}(\vec{\mathbf{X}},\ \vec{\mathbf{Y}}))\right)$$

Removing 2nd Order Existential Quantifiers (Bindings)

```
def IRSub(
    x: BV[8],
    y: BV[8]
) -> BV[8]
    return x + y
```

$$\exists\, \mathbf{I}, \mathbf{b^{in}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\,.$$

$$IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b^{in}}(\mathbf{x},\ \mathbf{y}))$$

```
def PE(
    I: BV[4],
    a: BV[8],
    b: BV[8]
) -> BV[8]
    ...
    elif I == 1:
        return b - a
    ...
```

Removing 2nd Order Existential Quantifiers (Bindings)

Enumerate all possible binding functions

```
def bin1(x, y):
    return x, y
```

```
def bin2(x, y):
    return y, x
```

$$\exists\, \mathbf{I}, \mathbf{b}^{\mathbf{in}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b}^{\mathbf{in}}(\mathbf{x},\ \mathbf{y}))$$

Removing 2nd Order Existential Quantifiers (Bindings)

Enumerate all possible binding functions

```
def bin1(x, y):
    return x, y
```

```
def bin2(x, y):
    return y, x
```

$$\exists\, \mathbf{I}, \mathbf{b}^{\mathbf{in}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b}^{\mathbf{in}}(\mathbf{x},\ \mathbf{y}))$$

$$\exists\, \mathbf{I}, \mathbf{i}_{\mathbf{bin}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\ .$$

Removing 2nd Order Existential Quantifiers (Bindings)

Enumerate all possible binding functions

```
def bin1(x, y):
    return x, y
```

```
def bin2(x, y):
    return y, x
```

$$\exists\, \mathbf{I}, \mathbf{b}^{\mathbf{in}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b}^{\mathbf{in}}(\mathbf{x},\ \mathbf{y}))$$

$$\exists\, \mathbf{I}, \mathbf{i}_{\mathbf{bin}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$(\mathbf{i}_{\mathbf{bin}} = \mathbf{1}) \wedge IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b}^{\mathbf{in1}}(\mathbf{x},\ \mathbf{y}))$$

Removing 2nd Order Existential Quantifiers (Bindings)

Enumerate all possible binding functions

```
def b^in1(x, y):
    return x, y
```

```
def b^in2(x, y):
    return y, x
```

$$\exists\, \mathbf{I}, \mathbf{b}^{\mathbf{in}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b}^{\mathbf{in}}(\mathbf{x},\ \mathbf{y}))$$

$$\exists\, \mathbf{I}, \mathbf{i}_{\mathbf{bin}} \,.\, \forall\, \mathbf{x},\ \mathbf{y}\ .$$

$$(\mathbf{i}_{\mathbf{bin}} = \mathbf{1}) \wedge IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b}^{\mathbf{in1}}(\mathbf{x},\ \mathbf{y})) \vee$$

$$(\mathbf{i}_{\mathbf{bin}} = \mathbf{2}) \wedge IRSub(\mathbf{x},\ \mathbf{y}) = PE(\mathbf{I}, \mathbf{b}^{\mathbf{in2}}(\mathbf{x},\ \mathbf{y}))$$

Removing 2nd Order Existential Quantifiers (Instruction)

- Instruction **I(imm)** is modeled as an Algebraic Data Type (ADT)

- ADTs annotated with which fields are immediates

- Enumerate only valid ADT value constructions allowing **imm** in immediate fields.

Removing 2nd Order Universal Quantifiers using Ackermannization

Replace every function application of F

$$\mathbf{F}(\mathbf{x}_1),\ \mathbf{F}(\mathbf{x}_2)$$

with universally quantified fresh variables

$$\forall \mathbf{f}_1,\ \mathbf{f}_2\ .$$

and embed functional consistency constraints

$$(\mathbf{x}_1 = \mathbf{x}_2) \implies (\mathbf{f}_1 = \mathbf{f}_2)$$

Solving Strategy

- Final formula is of the form:

$$\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$$

- Use Counter Example Guided Synthesis (CEGIS)
  - Alternates between Synthesis and Verification steps

GOAL: $\exists\ \overrightarrow{\mathbf{e}} . \forall\ \overrightarrow{\mathbf{a}} .\ \phi(\overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{a}})$

a_0 | *# Initialize*

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0 # *Initialize*

$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0)$ # *Synthesize*

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0 # *Initialize*

$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow UNSAT$ # *Synthesize*

GOAL: $\exists\ \overrightarrow{\mathbf{e}} . \forall\ \overrightarrow{\mathbf{a}} .\ \phi(\overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{a}})$

a_0 # *Initialize*

$\exists\ \overrightarrow{\mathbf{e}} . \phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow UNSAT$ # *Synthesize*

$\Longrightarrow$

$\exists\ \overrightarrow{\mathbf{e}} . \forall\ \overrightarrow{\mathbf{a}} .\ \phi(\overrightarrow{\mathbf{e}}, \overrightarrow{\mathbf{a}}) \rightarrow UNSAT$

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0 # *Initialize*

$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$ # *Synthesize*

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0	# *Initialize*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$	# *Synthesize*
$\forall\ \overrightarrow{\mathbf{a}}\ .\phi(e_{guess},\ \overrightarrow{\mathbf{a}})$	# *Verify*

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0	# *Initialize*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$	# *Synthesize*
$\exists\ \overrightarrow{\mathbf{a}}\ .\neg\phi(e_{guess},\ \overrightarrow{\mathbf{a}})$	# *Verify*

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0	# *Initialize*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$	# *Synthesize*
$\exists\ \overrightarrow{\mathbf{a}}\ .\neg\phi(e_{guess},\ \overrightarrow{\mathbf{a}}) \rightarrow UNSAT$	# *Verify*

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0 # *Initialize*

$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$ # *Synthesize*

$\exists\ \overrightarrow{\mathbf{a}}\ .\neg\phi(e_{guess},\ \overrightarrow{\mathbf{a}}) \rightarrow UNSAT$ # *Verify*

$\Longrightarrow$

$\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}}) \rightarrow (SAT, e_{guess})$

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0	# *Initialize*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$	# *Synthesize*
$\exists\ \overrightarrow{\mathbf{a}}\ .\neg\phi(e_{guess},\ \overrightarrow{\mathbf{a}}) \rightarrow (SAT,\ a_1)$	# *Verify*

GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0	# *Initialize*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$	# *Synthesize*
$\exists\ \overrightarrow{\mathbf{a}}\ .\neg\phi(e_{guess},\ \overrightarrow{\mathbf{a}}) \rightarrow (SAT,\ a_1)$	# *Verify*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \wedge \phi(\overrightarrow{\mathbf{e}}, a_1)$	# *Synthesize*

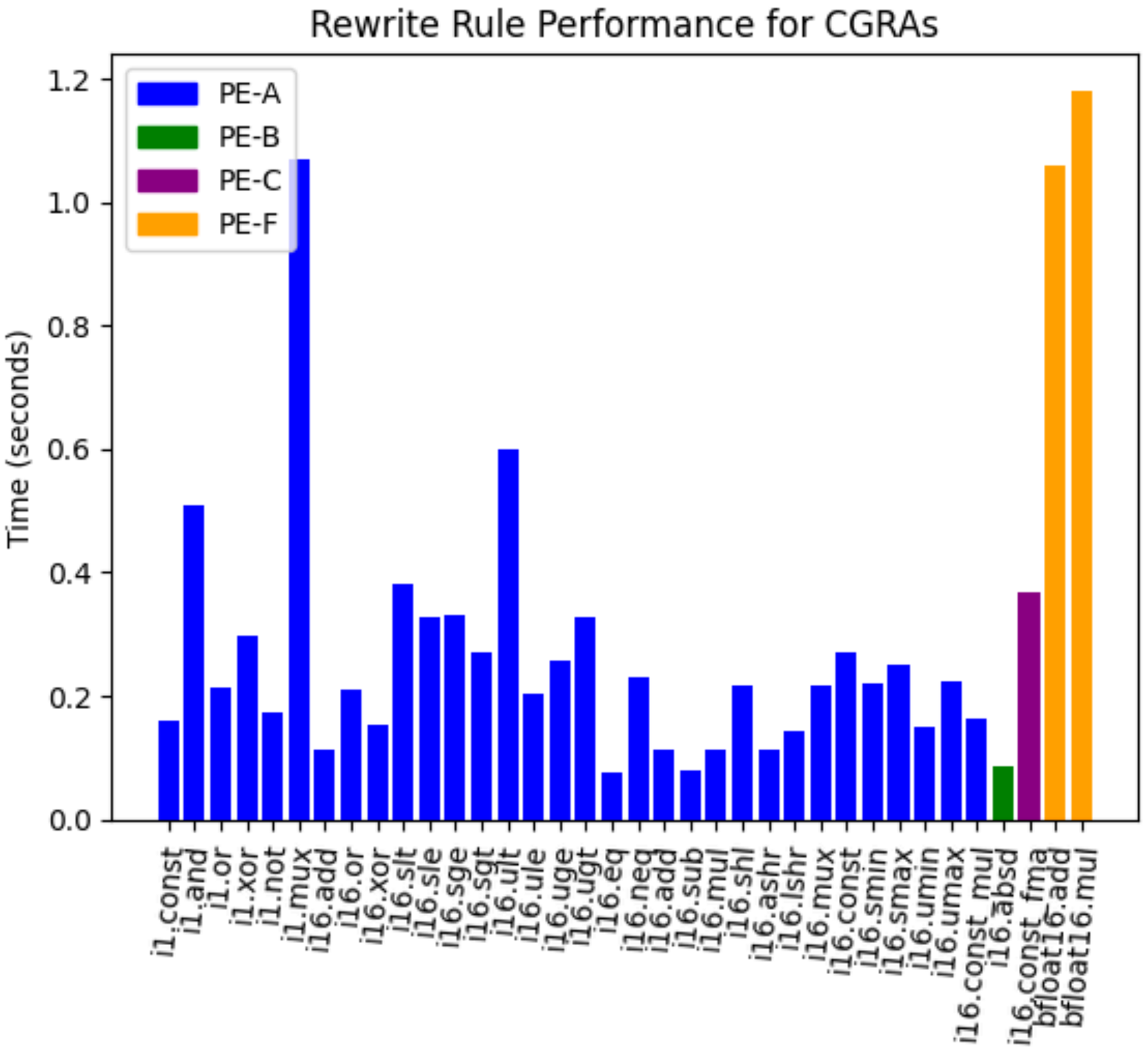
GOAL: $\exists\ \overrightarrow{\mathbf{e}}\ .\forall\ \overrightarrow{\mathbf{a}}\ .\ \phi(\overrightarrow{\mathbf{e}},\ \overrightarrow{\mathbf{a}})$

a_0	# *Initialize*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \rightarrow (SAT,\ e_{guess})$	# *Synthesize*
$\exists\ \overrightarrow{\mathbf{a}}\ .\neg\phi(e_{guess},\ \overrightarrow{\mathbf{a}}) \rightarrow (SAT,\ a_1)$	# *Verify*
$\exists\ \overrightarrow{\mathbf{e}}\ .\phi(\overrightarrow{\mathbf{e}}, a_0) \wedge \phi(\overrightarrow{\mathbf{e}}, a_1)$	# *Synthesize*
...	...

Evaluation

CGRA Case study

- CoreIR as IR
- 4 versions of the PE
  - PE-A: Bit-wise operations, comparisons shifts, addition, multiplication, LUT for boolean Ops
  - PE-B: PE-A + Absolute difference,
  - PE-C: PE-B + Fused Multiply Add with an immediate
  - PE-F: PE-A + Floating point Addition and multiplication

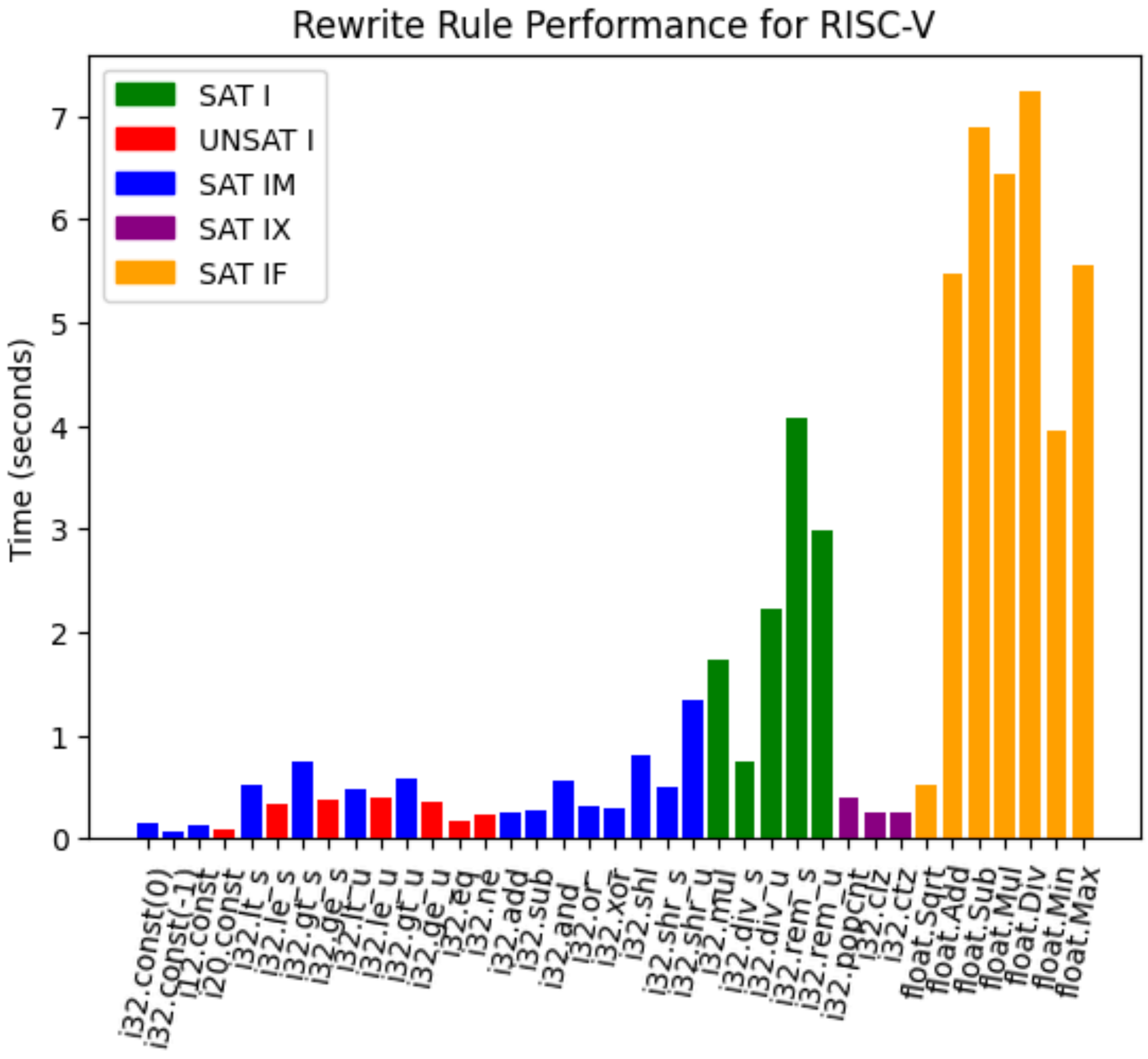

Rewrite Rule Performance for CGRAs
PE-A
PE-B
PE-C
PE-F
Time (seconds)
1.2
1.0
0.8
0.6
0.4
0.2
0.0
i1.const
i1.and
i1.or
i1.xor
i1.not
i1.mux
i16.add
i16.or
i16.xor
i16.slt
i16.sle
i16.sge
i16.sgt
i16.ult
i16.ule
i16.uge
i16.ugt
i16.eq
i16.neq
i16.add
i16.sub
i16.mul
i16.shl
i16.ashr
i16.lshr
i16.mux
i16.const
i16.smin
i16.smax
i16.umin
i16.umax
i16.const_mul
i16.absd
i16.const_fma
bfloat16.add
bfloat16.mul

Total time to Synthesize All Rewrite Rules

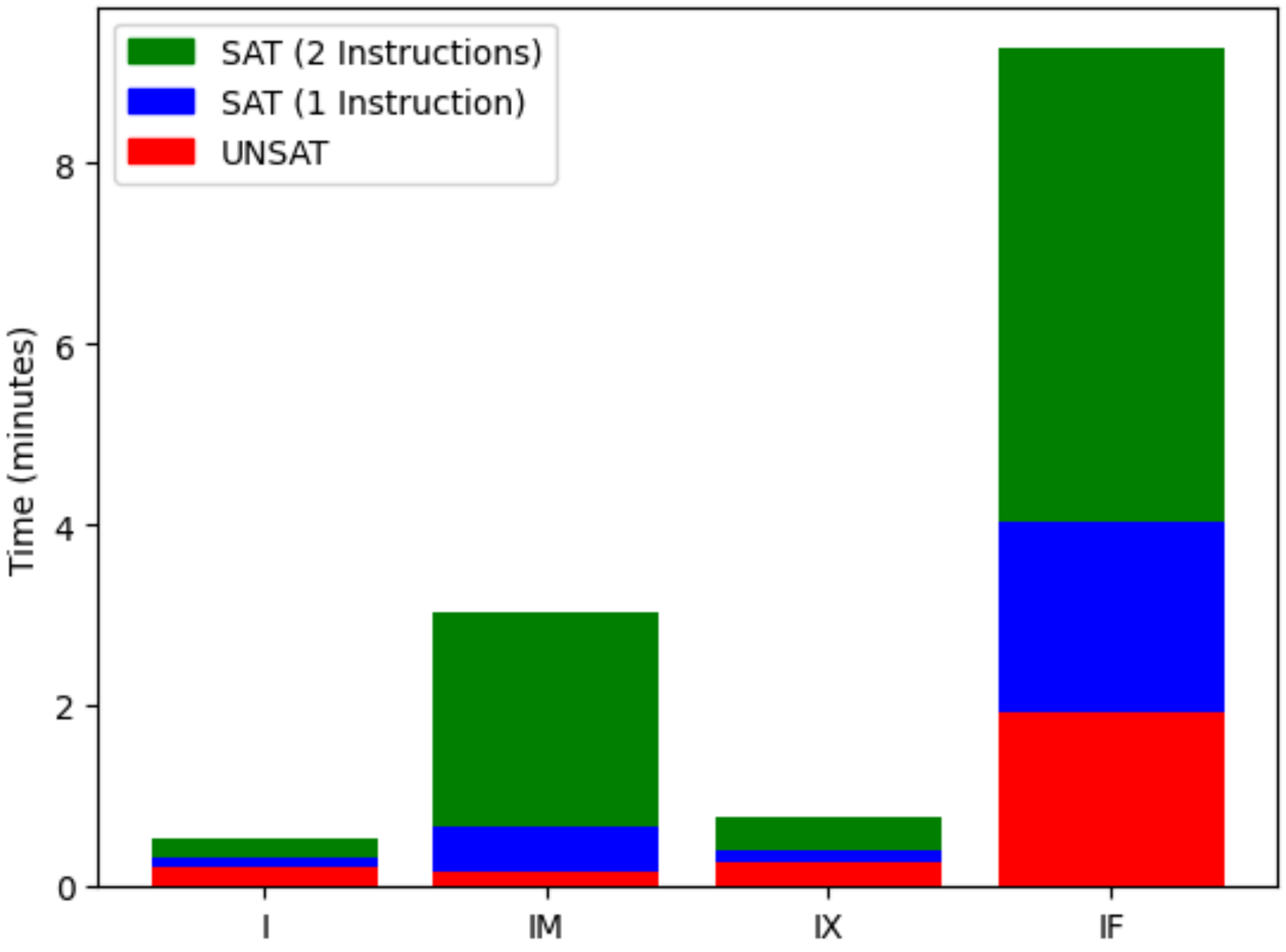
	PE-A	PE-B	PE-C	PE-F
UNSAT (s)	0.81	0.74	0.34	118.09
SAT (s)	8.63	10.15	11.06	91.49
Total (s)	**9.44**	**10.88**	**11.40**	**209.58**

RISC-V Case study

- WebAssembly as IR
- 4 versions of RISC-V
    - RV32I: Base instruction set
    - RV32IM: Base + Multiply/Divide
    - RV32IF: Base + Floating Point
    - RV32IX: Base + custom ops (popcnt, clz, ctz)
- PE uses post-fetch part of the processor RTL


Rewrite Rule Performance for RISC-V
SAT I
UNSAT I
SAT IM
SAT IX
SAT IF
Time (seconds)
0
1
2
3
4
5
6
7
i32.const(0)
i32.const(-1)
i12.const
i20.const
i32.lt_s
i32.le_s
i32.gt_s
i32.ge_s
i32.lt_u
i32.le_u
i32.gt_u
i32.ge_u
i32.eq
i32.ne
i32.add
i32.sub
i32.and
i32.or
i32.xor
i32.shl
i32.shr_s
i32.shr_u
i32.mul
i32.div_s
i32.div_u
i32.rem_s
i32.rem_u
i32.popcnt
i32.clz
i32.ctz
float.Sqrt
float.Add
float.Sub
float.Mul
float.Div
float.Min
float.Max

Total time to Synthesize All Rewrite Rules



Summary

- Presented a synthesis technique for a general class of rewrite rules between IRs and RTL-based architectures.

- Presented a technique for supporting *parametric* rewrite rules and rules for operations, the semantics of which are unknown or too complex

- Rewrite rule synthesis from an RTL description is not only possible, but *efficient*

- **Step towards fully automating compiler generation for hardware accelerators.**

Questions?

Backup

CGRA Compilation

	Hand-Coded	Synthesized			
Application	PE-F	PE-F	PE-A	PE-B	PE-C
Gaussian i16	20	20	20	20	12
Gaussian bfloat16	20	20	N/A	N/A	N/A
Harris	116	109	109	108	108
Camera	343	338	338	309	308

The number of PE instructions required for four Halide applications: Camera, Gaussian integer, Gaussian bfloat16, and Harris. The applications are compiled using both the hand-coded rewrite rules and the synthesized ones.

RiscV Compilation

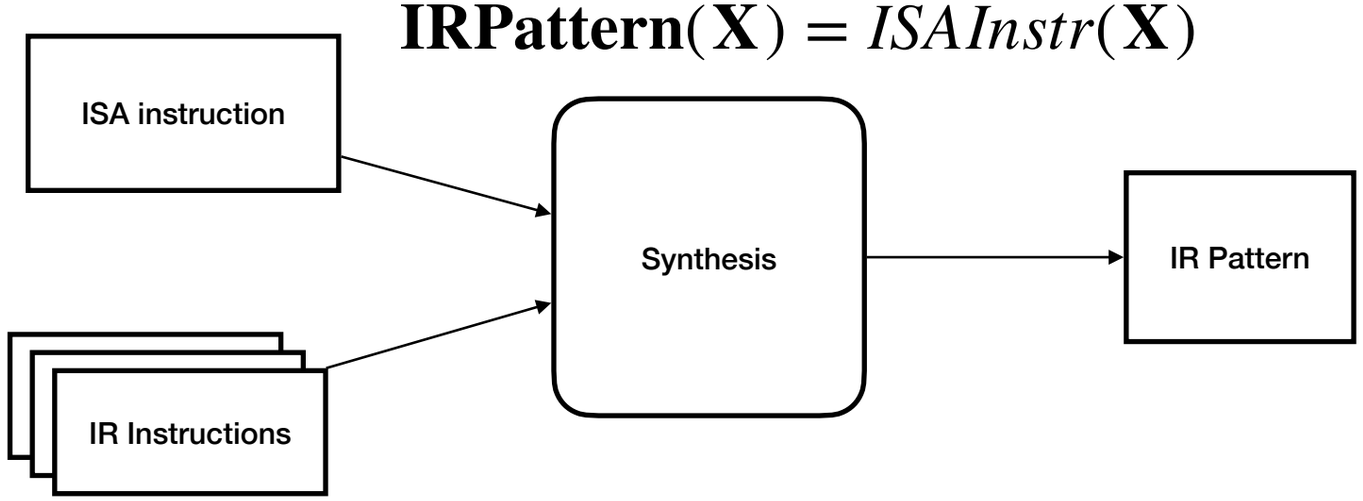
Number of RISC-V RV32IM instructions on 25 Hacker's Delight programs (P1-P25). We show our system versus gcc with two levels of optimization. *The compilation of P18 to WebAssembly generated a i32.popcnt and hence could only be compiled to RV32IX.

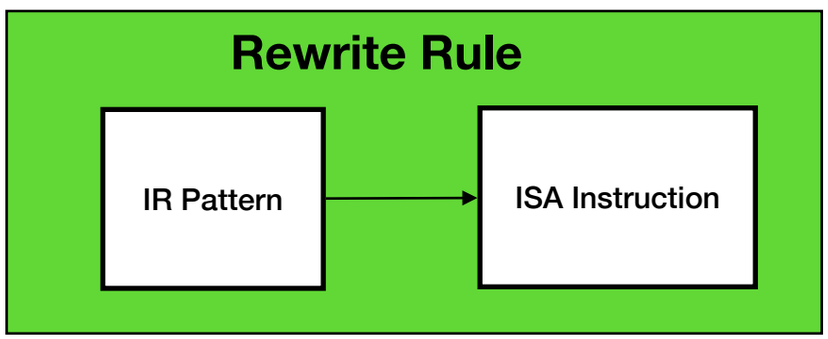
Benchmark	Synthesized	gcc -O0	gcc -O1
P1	3	16	3
P2	3	16	3
P3	3	16	3
P4	3	16	3
P5	3	16	3
P6	3	16	3
P7	5	19	4
P8	5	19	4
P9	4	20	4
P10	4	24	5
P11	4	22	4
P12	5	23	5
P13	5	22	4
P14	5	25	5
P15	5	25	5
P16	10	29	6
P17	6	23	5
P18	4*	36	7
P19	6	35	7
P20	9	35	8
P21	25	50	13
P22	26	39	11
P23	32	50	15
P24	18	50	12
P25	27	72	19

Buchwald/Gulwani

$$\exists\ \textbf{IRPattern}\ .\ \forall\ \vec{\mathbf{X}}\ .$$

$$\textbf{IRPattern}(\vec{\mathbf{X}}) = \mathit{ISAInstr}(\vec{\mathbf{X}})$$





Daly et al.
FMCAD 22

$$\exists\, \mathbf{I} . \forall\, \vec{\mathbf{X}} .$$
$$IRPattern(\vec{\mathbf{X}}) = RTL(\mathbf{I}, \vec{\mathbf{X}})$$