Automatic Compilation for Domain Specific Accelerators

Ross Daly

Caleb Donovick

Jackson Melchert

Golden Age of Computer Architecture!

Golden Age of Computer Architecture!

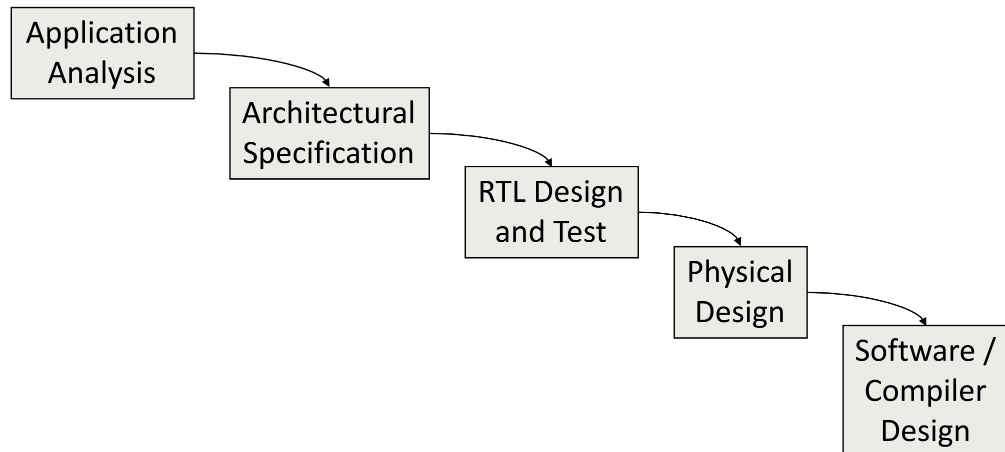
- Architecture Specifications change frequently

Golden Age of Computer Architecture!

- Architecture Specifications change frequently
- Compiler is the (often overlooked) key component!

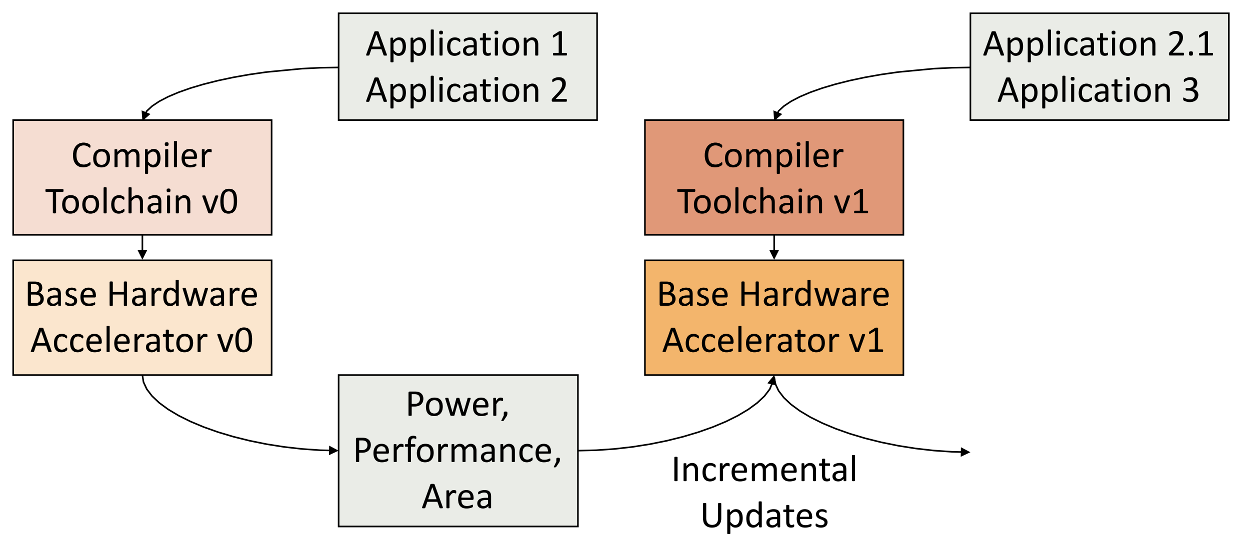
Golden Age of Computer Architecture!

- Architecture Specifications change frequently
- Compiler is the (often overlooked) key component!
- Waterfall methodology:



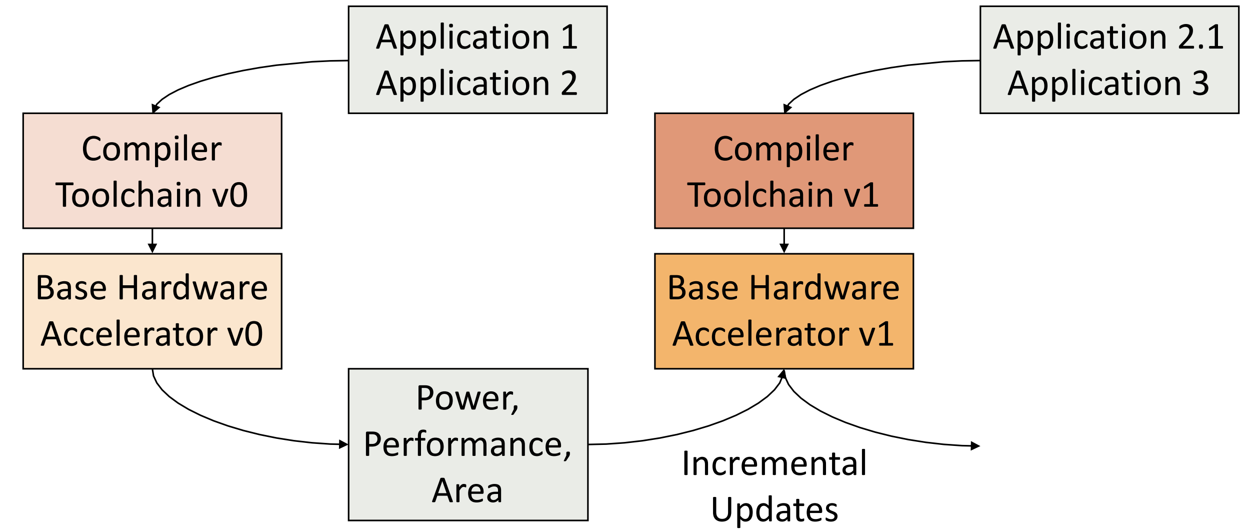
Golden Age of Computer Architecture!

- Architecture Specifications change frequently
- Compiler is the (often overlooked) key component!
- Agile methodology:



Golden Age of Computer Architecture!

- Architecture Specifications change frequently
- Compiler is the (often overlooked) key component!
- Agile methodology:
  - Automatically generate compiler for every spec change



CGRA/FPGA

- Compile to IR (CoreIR)
- Common Optimizations
- Mapping
- Packing
- Placement
- Routing
- Bitfile generation

CPU

- Compile to IR (LLVM)
- Common Optimizations
- Instruction Selection
- Peephole Optimization
- Instruction Scheduling
- Register Allocation
- Assembly

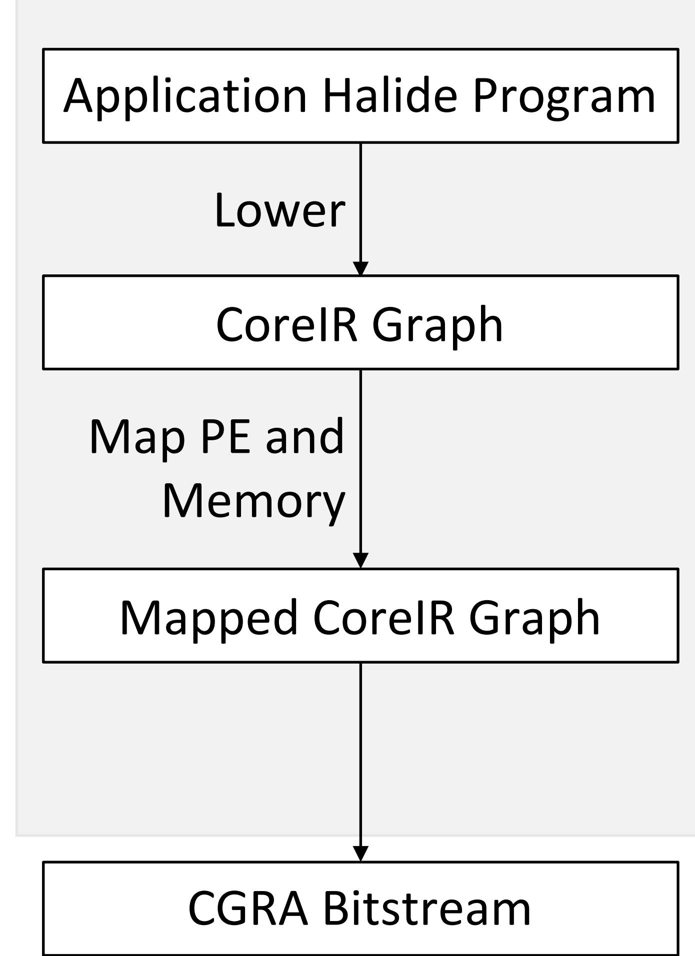
CGRA/FPGA

- Compile to IR (CoreIR)
- Common Optimizations
- **Mapping**
- Packing
- Placement
- Routing
- Bitfile generation

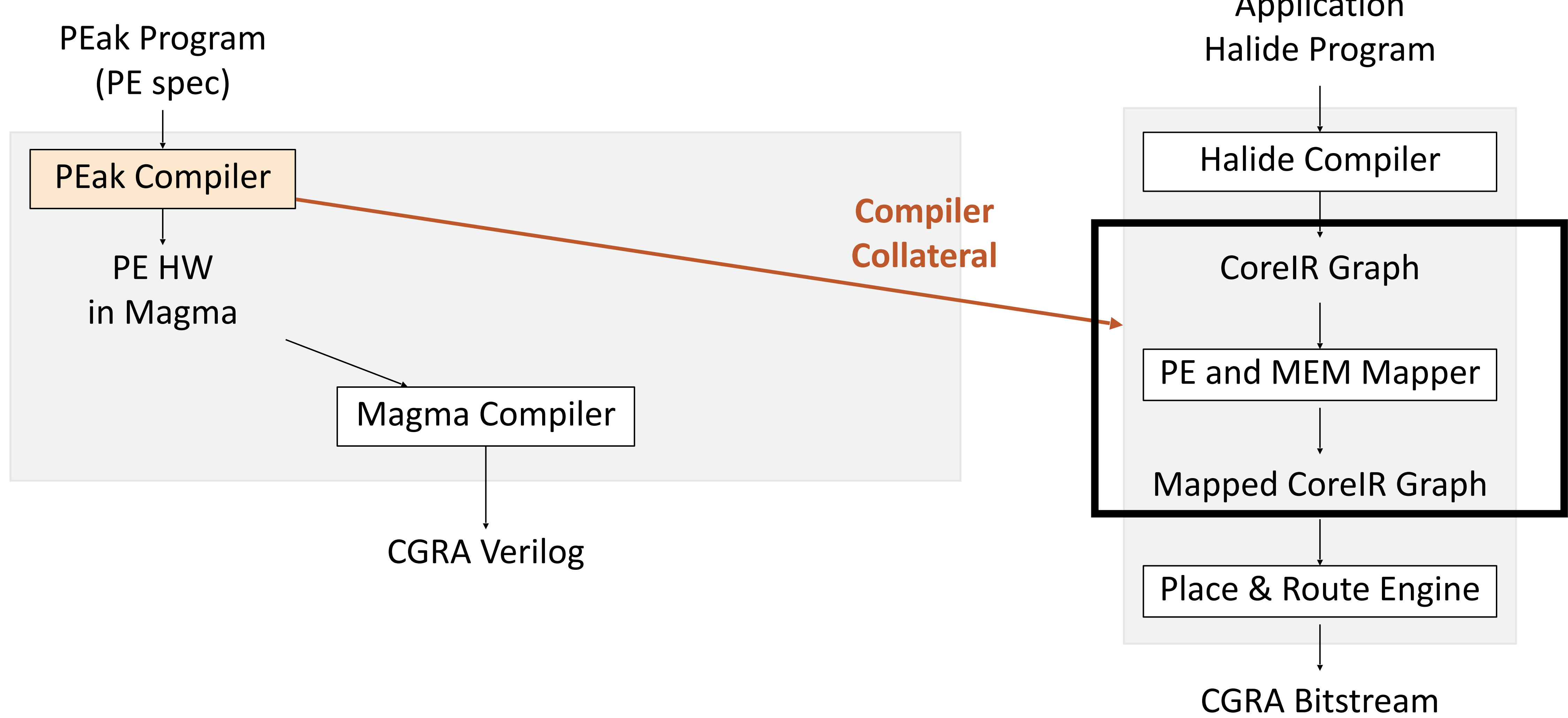
CPU

- Compile to IR (LLVM)
- Common Optimizations
- **Instruction Selection**
- Peephole Optimization
- Instruction Scheduling
- Register Allocation
- Assembly

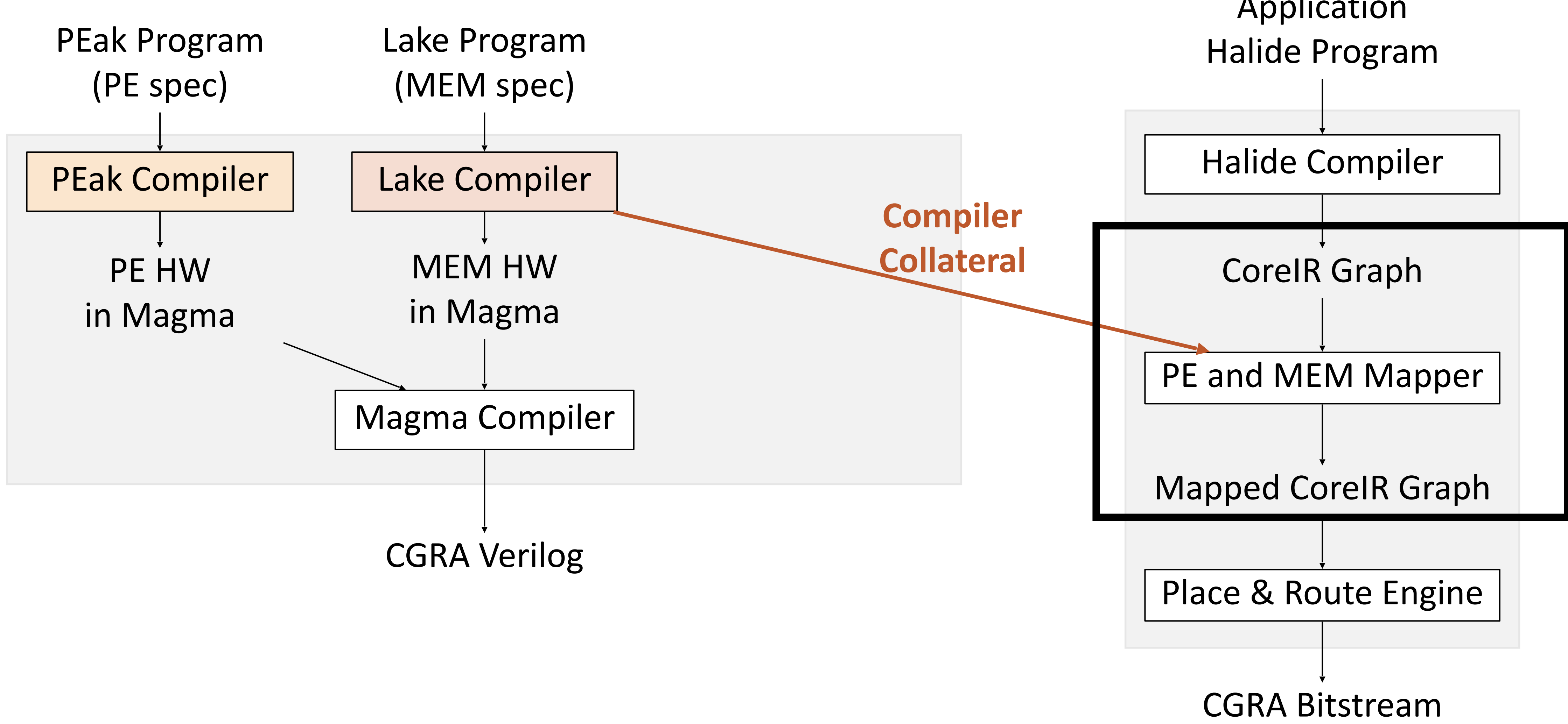
CGRA Mapping



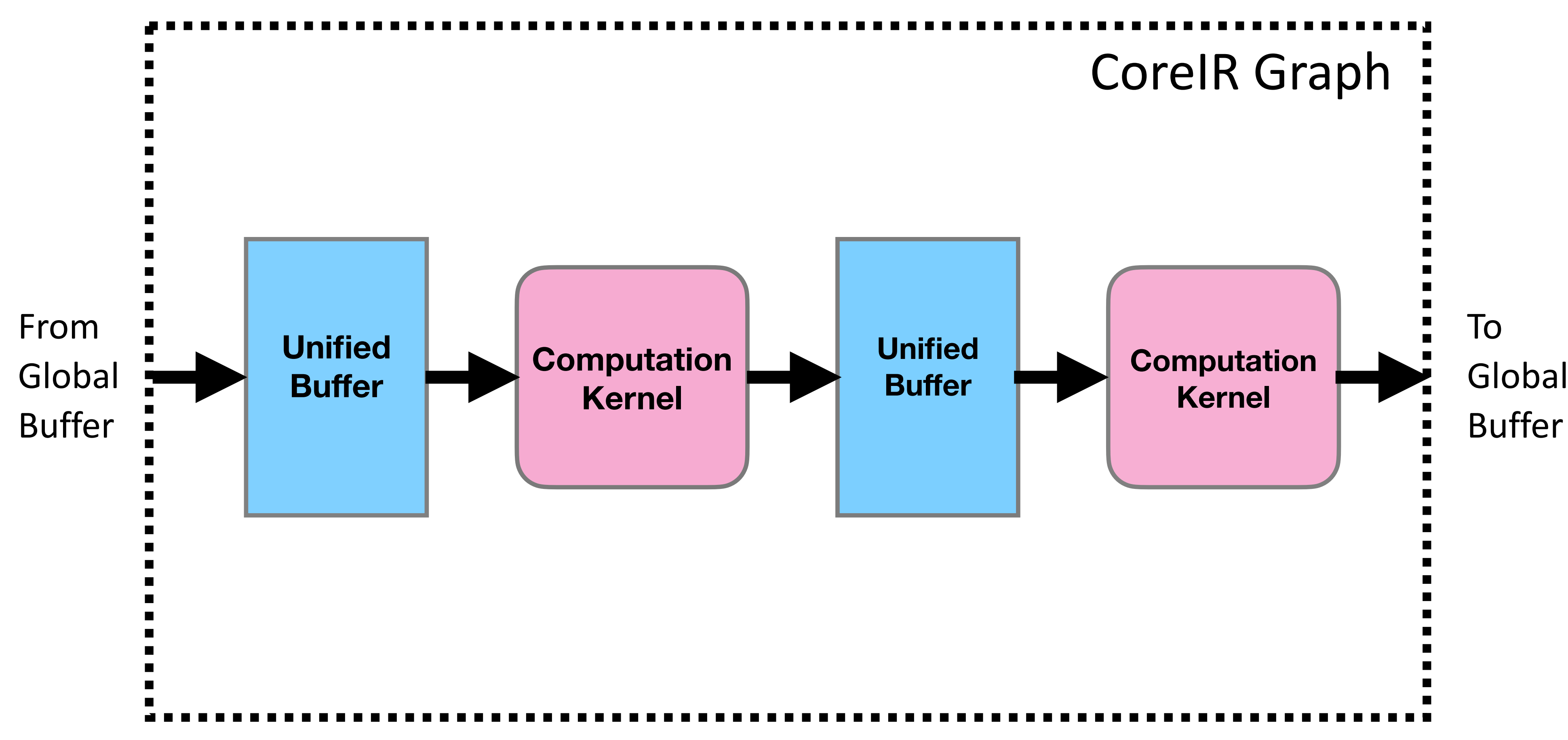
Our DSL-based Hardware Generation and Software Compilation Flow



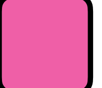
Our DSL-based Hardware Generation and Software Compilation Flow



Output of Halide Compiler



Desired Output of Mapper

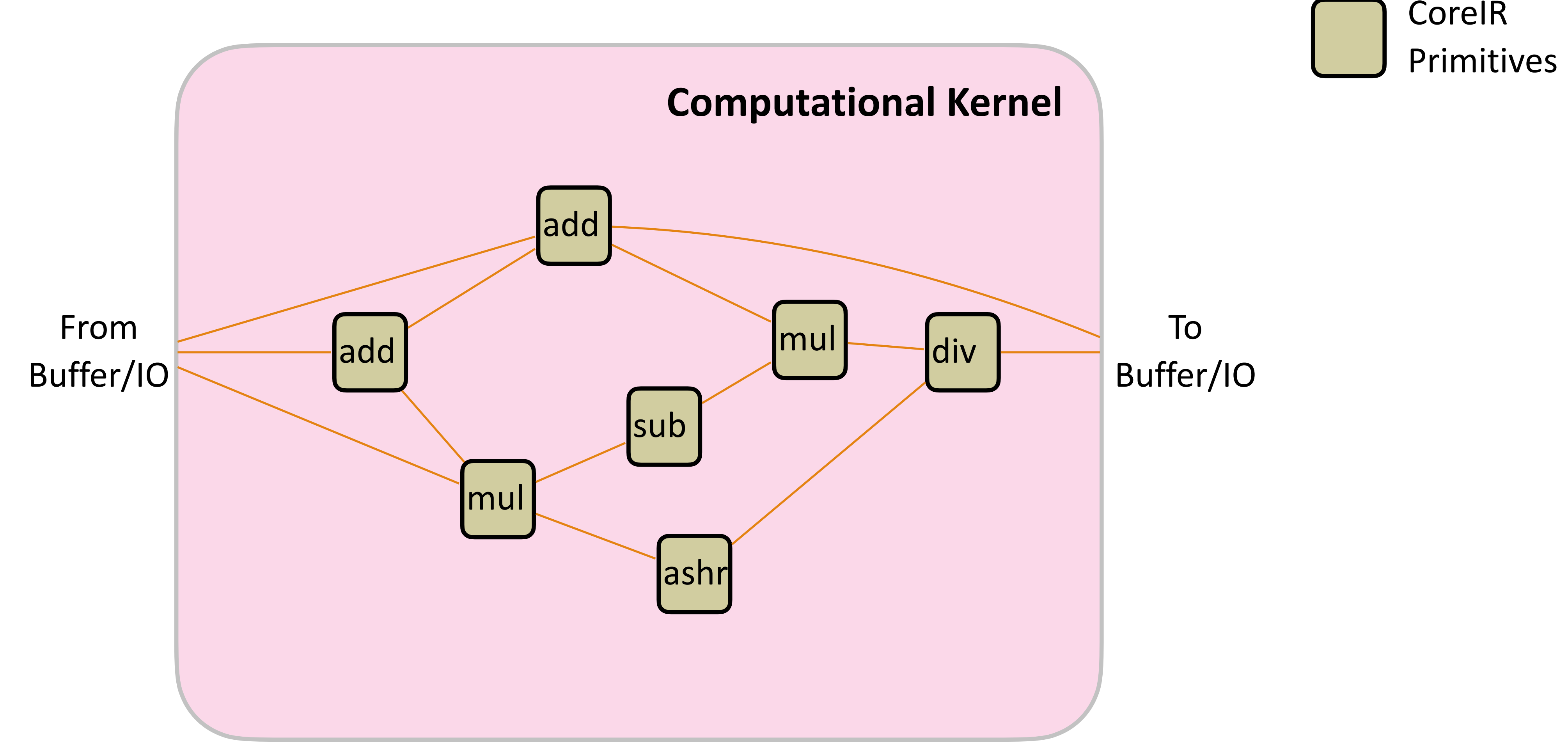


Mapped CoreIR Graph

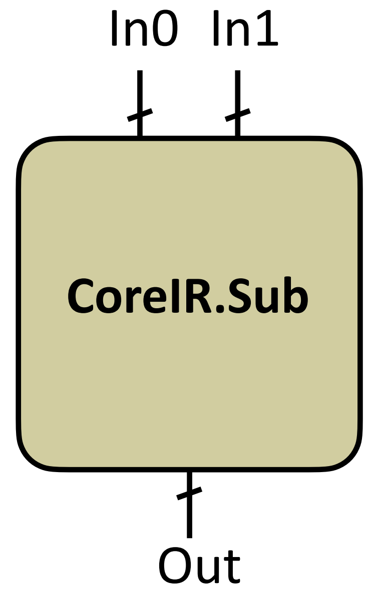
From Global Buffer

To Global Buffer

Kernels are composed of CoreIR Primitives

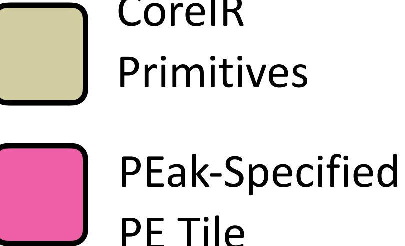


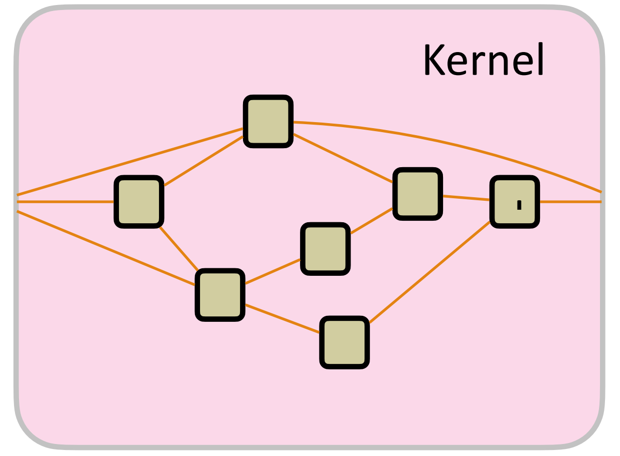
CoreIR has SMT QF BitVector Semantics

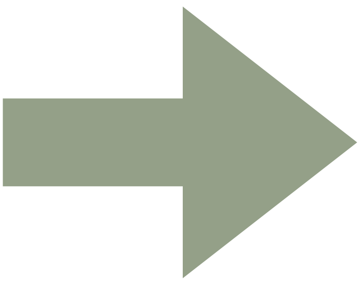


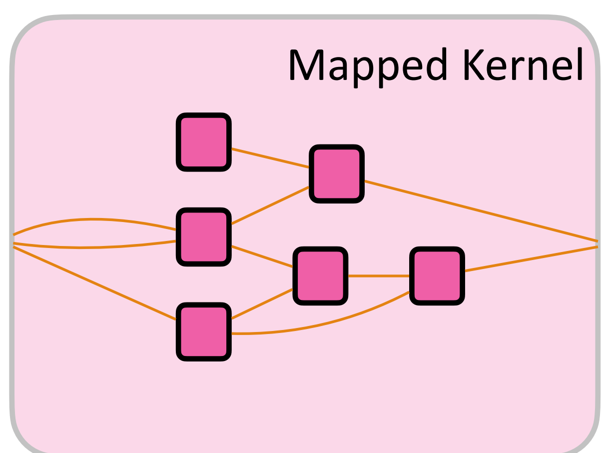
Out = In0 - In1

Mapping

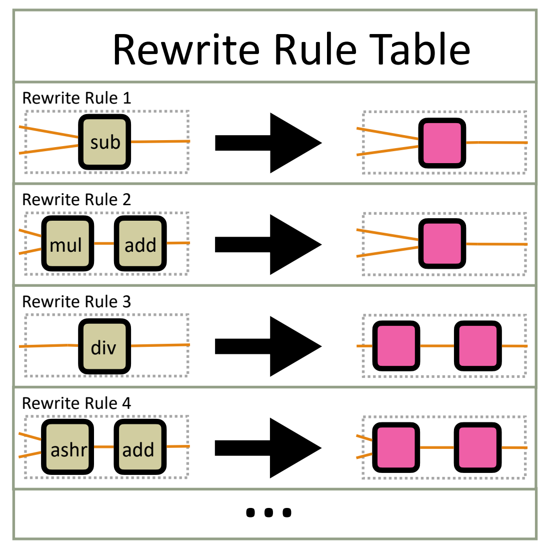


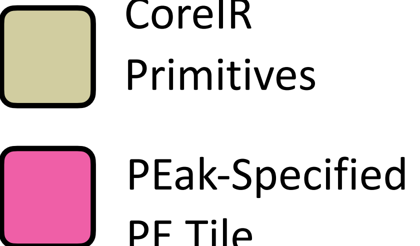


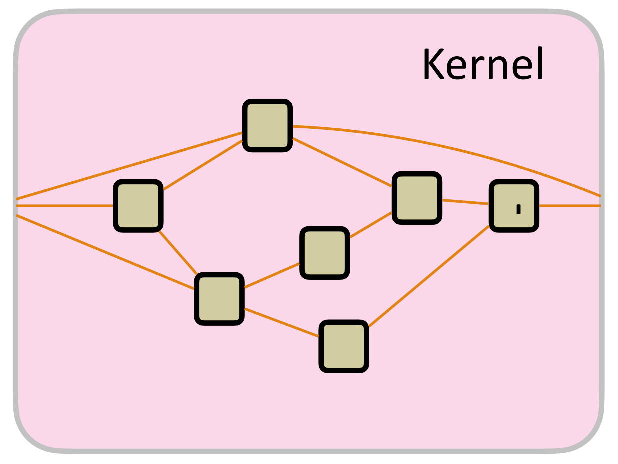


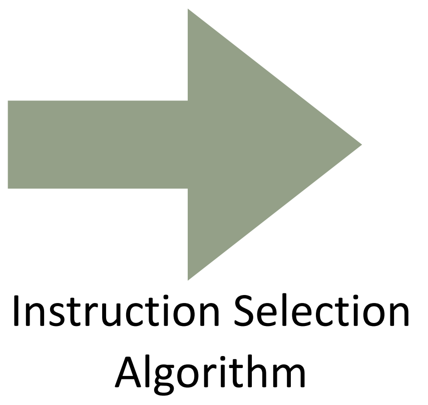


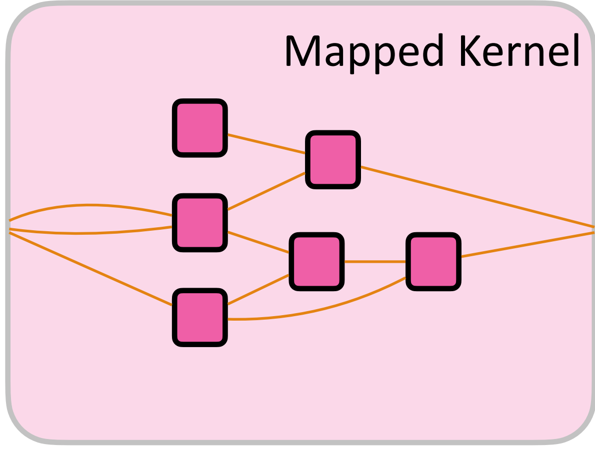
Instruction Selection



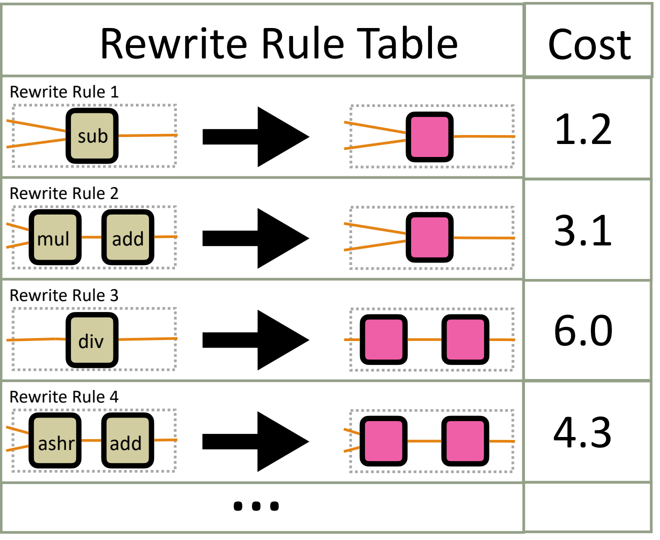


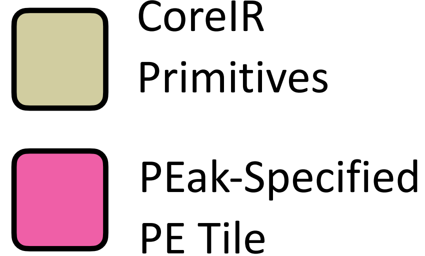


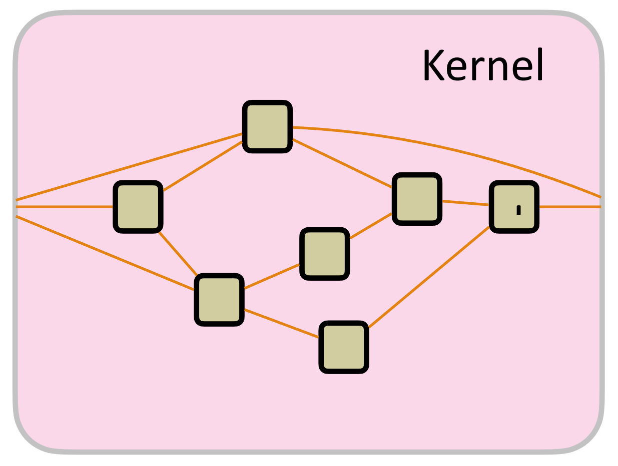


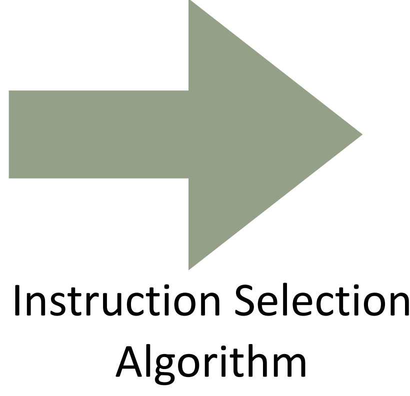


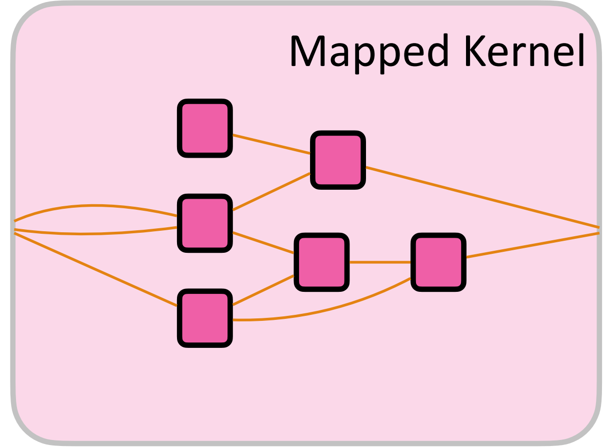
Instruction Selection



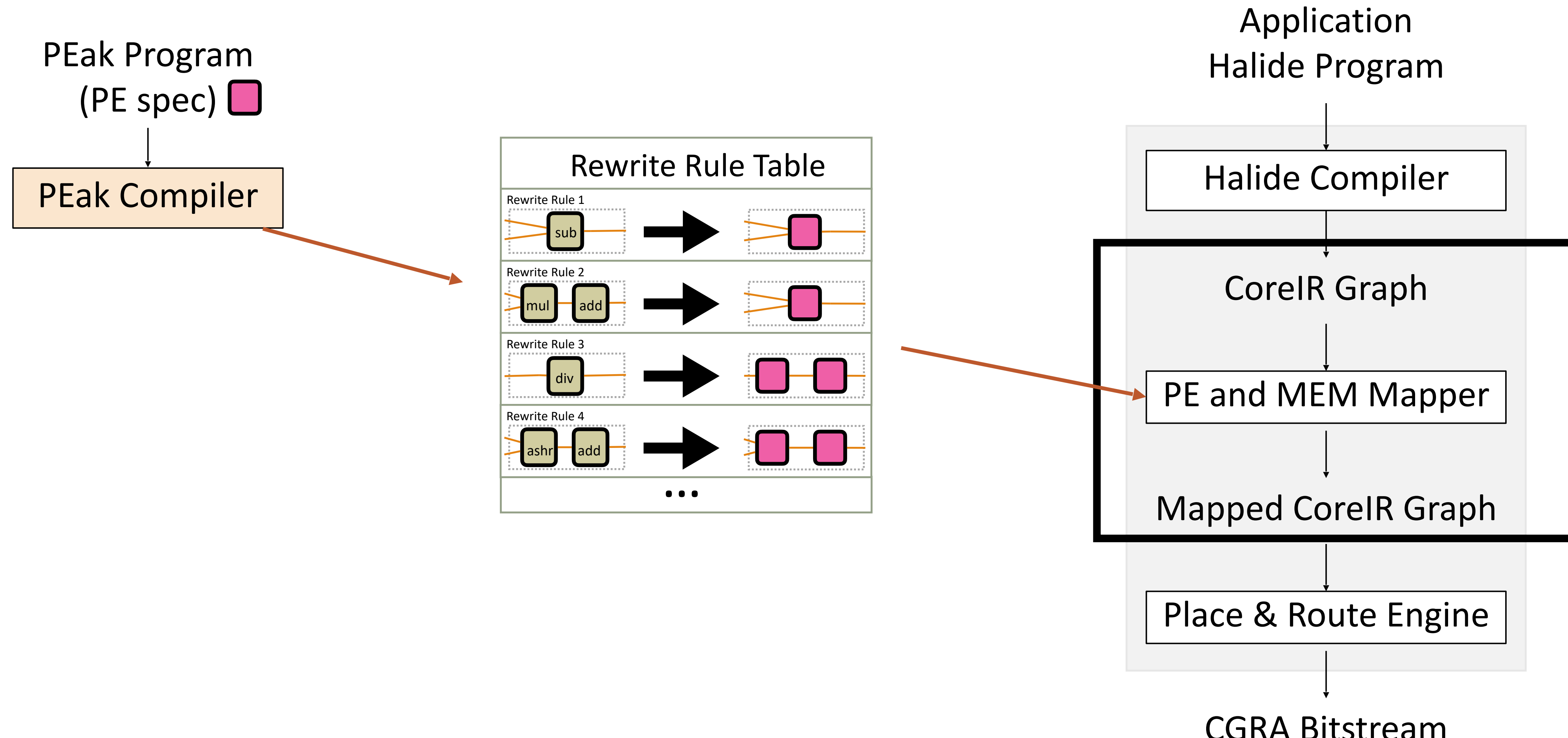








Peak Compiler generates a table of Rewrite Rules



PEak: PE DSL

PE ISA Specification

PE Functional Specification

PEak: PE DSL

PE ISA Specification

PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Const(Instruction), A: Word,
      B: Word, C: Word) -> {"res":Word, "flag":Bit}:
```

PEak: PE DSL

PE ISA Specification

PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Const(Instruction), A: Word,
      B: Word, C: Word) -> {"res":Word, "flag":Bit}:
```

Specific **types** (or composition of types) for operands and instructions

PEak: PE DSL

PE ISA Specification

```
class Opcode(Enum):
   Add = 0
   Mul = 1
   …
# Define Instruction
class Instruction(Product):
   op = Opcode
   invert_A = Bit
   c_in = Bit
```

PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Const(Instruction), A: Word,
      B: Word, C: Word) -> {"res":Word, "flag":Bit}:
```

Specific **types** (or composition of types) for operands and instructions

PEak: PE DSL

PE ISA Specification

```
class Opcode(Enum):
   Add = 0
   Mul = 1
   …
# Define Instruction
class Instruction(Product):
   op = Opcode
   invert_A = Bit
   c_in = Bit

# Define Word
Word = UnsignedBitVector[16]
```

Specific **types** (or composition of types) for operands and instructions

PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Const(Instruction), A: Word,
      B: Word, C: Word) -> {“res”:Word, “flag”:Bit}:
```

PEak: PE DSL

PE ISA Specification

```
class Opcode(Enum):
    Add = 0
    Mul = 1
    …
# Define Instruction
class Instruction(Product):
    op = Opcode
    invert_A = Bit
    c_in = Bit

# Define Word
Word = UnsignedBitVector[16]
```

Specific **types** (or composition of types) for operands and instructions

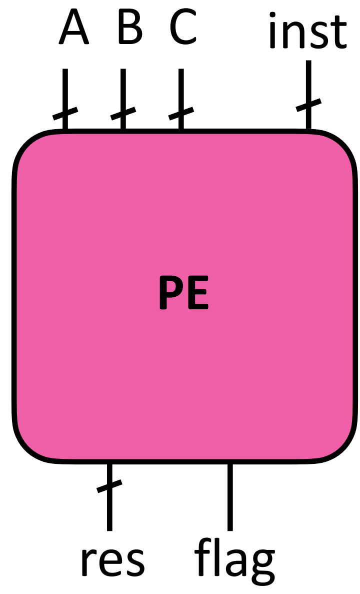
PE Functional Specification

```
class PE(Peak):
    def __call__(self, inst: Const(Instruction), A: Word,
        B: Word, C: Word) -> {"res":Word, "flag":Bit}:

        if inst.invert_A:
            A = ~A

        if inst.op == Opcode.Add:
            res, c_out = A.add(B, inst.c_in)
            flag = c_out
        elif inst.op == Opcode.Mul:
            res = A * B
            flag = (res == 0)
        elif ... :
            ...
        return res, flag
```

Subtract?



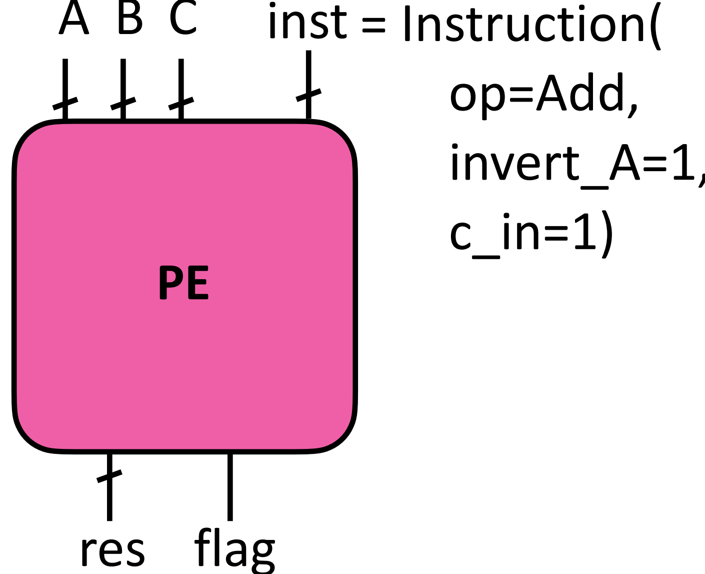
PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Instruction, A: Word,
      B: Word, C: Word) -> {“res”:Word, “flag”:Bit}:

      if inst.invert_A:
         A = ~A

      if inst.op == Opcode.Add:
         res, c_out = A.add(B, inst.c_in)
         flag = c_out
      elif inst.op == Opcode.Mul:
         res = A * B
         flag = (res == 0)
      elif ... :
         ...
      return res, flag
```

Subtract?



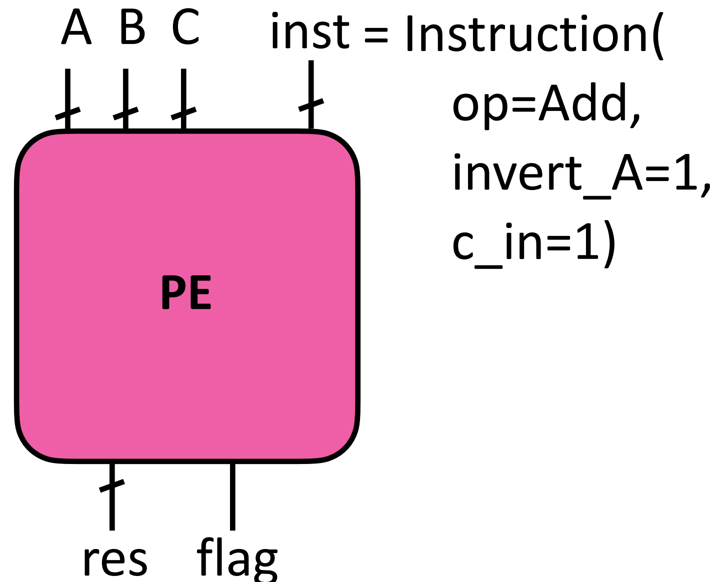
PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Instruction, A: Word,
      B: Word, C: Word) -> {"res":Word, "flag":Bit}:

      if inst.invert_A:
         A = ~A

      if inst.op == Opcode.Add:
         res, c_out = A.add(B, inst.c_in)
         flag = c_out
      elif inst.op == Opcode.Mul:
         res = A * B
         flag = (res == 0)
      elif ... :
         ...
      return res, flag
```

Subtract?



res = ~A + B + 1

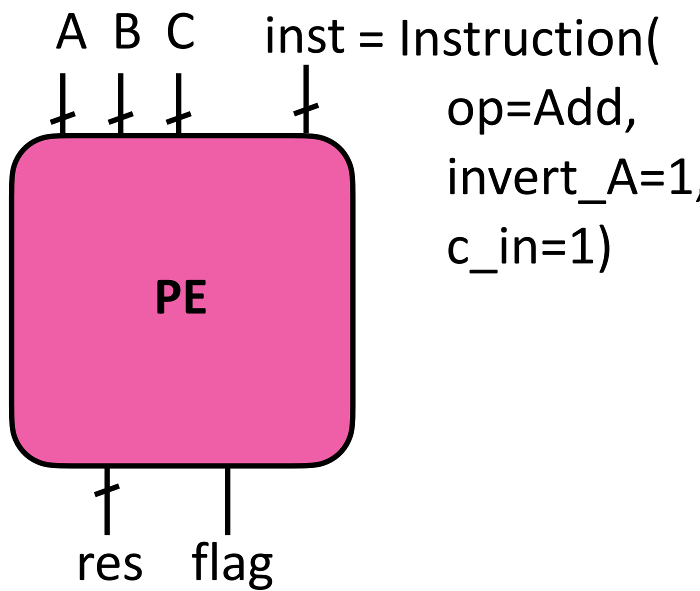
PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Instruction, A: Word,
      B: Word, C: Word) -> {"res":Word, "flag":Bit}:

      if inst.invert_A:
         A = ~A

      if inst.op == Opcode.Add:
         res, c_out = A.add(B, inst.c_in)
         flag = c_out
      elif inst.op == Opcode.Mul:
         res = A * B
         flag = (res == 0)
      elif ... :
         ...
      return res, flag
```

Subtract?



PE Functional Specification

```
class PE(Peak):
   def __call__(self, inst: Instruction, A: Word,
      B: Word, C: Word) -> {"res":Word, "flag":Bit}:

      if inst.invert_A:
         A = ~A

      if inst.op == Opcode.Add:
         res, c_out = A.add(B, inst.c_in)
         flag = c_out
      elif inst.op == Opcode.Mul:
         res = A * B
         flag = (res == 0)
      elif ... :
         ...
      return res, flag
```

res = ~A + B + 1 = B - A

RiscV Peak Specification

```
class RISCV(Peak):
    def __init__(self):
        self.rf = RegisterFile(32, Word)
        self.PC = Register(Data)
```

Define sub-components and state

RiscV Peak Specification

```
class RISCV(Peak):
   def __init__(self):
      self.rf = RegisterFile(32, Word)
      self.PC = Register(Data)
```

Define sub-components and state

```
   def __call__(self, inst: Instruction) ->{“next_PC”:Word}:

       #ID
       rs1_idx, rs2_idx, rd_idx, … = decode(inst)
       rs1_val, rs2_val = self.rf.read(rs1_idx, rs2_idx)

       #EX
       ...

        #MEM
        ...

        #WB
        self.rf.write(rd_val)
```

RiscV ISA Specification with Algebraic Data Types

32-bit RISC-V instruction formats

<table>
<tr><th rowspan="2">Format</th><th colspan="32">Bit</th></tr>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Register/register</td><td colspan="7">funct7</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Immediate</td><td colspan="12">imm[11:0]</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Upper immediate</td><td colspan="20">imm[31:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Store</td><td colspan="7">imm[11:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">imm[4:0]</td><td colspan="7">opcode</td></tr>
<tr><td>Branch</td><td>[12]</td><td colspan="6">imm[10:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="4">imm[4:1]</td><td>[11]</td><td colspan="7">opcode</td></tr>
<tr><td>Jump</td><td>[20]</td><td colspan="10">imm[10:1]</td><td>[11]</td><td colspan="8">imm[19:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
</table>

- **opcode (7 bits):** Partially specifies which of the 6 types of *instruction formats.*
- **funct7, and funct3 (10 bits):** These two fields, further than the *opcode* field, specify the operation to be performed.
- **rs1 (5 bits):** Specifies, by index, the register containing first operand (i.e., source register).
- **rs2 (5 bits):** Specifies the second operand register.
- **rd (5 bits):** Specifies the destination register to which the computation result will be directed.

RiscV ISA Specification with Algebraic Data Types

```
class Register(Product):
    funct7 = Funct7Enum
    rs2 = BitVector[5]
    rs1 = BitVector[5]
    funct3 = Funct3Enum
    rd = BitVector[5]
    opcode= Opcode

class Immediate(Product):
    ...

class UImmediate(Product):
    ...
class Store(Product):
    ...
class Branch(Product):
    ...
class Jump(Product):
    ...
```

Instruction = Sum[Register, Immediate, UImmediate, Store, Branch, Jump]

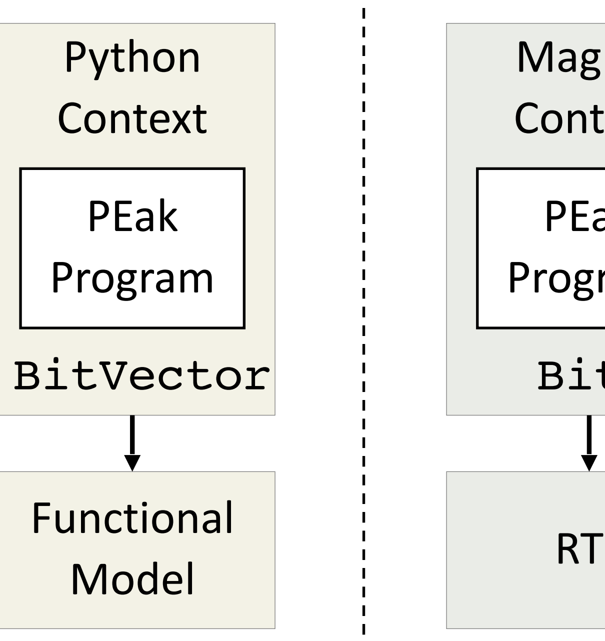
32-bit RISC-V instruction formats

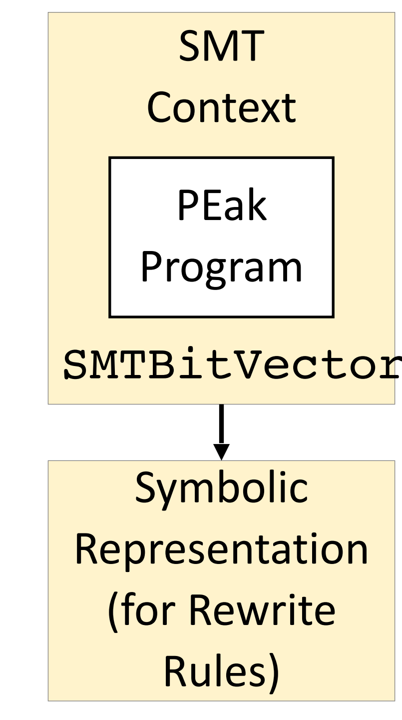
Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bits):** Partially specifies which of the 6 types of *instruction formats*.
- **funct7, and funct3 (10 bits):** These two fields, further than the *opcode* field, specify the operation to be performed.
- **rs1 (5 bits):** Specifies, by index, the register containing first operand (i.e., source register).
- **rs2 (5 bits):** Specifies the second operand register.
- **rd (5 bits):** Specifies the destination register to which the computation result will be directed.

Multiple Interpretations of PEak Specification

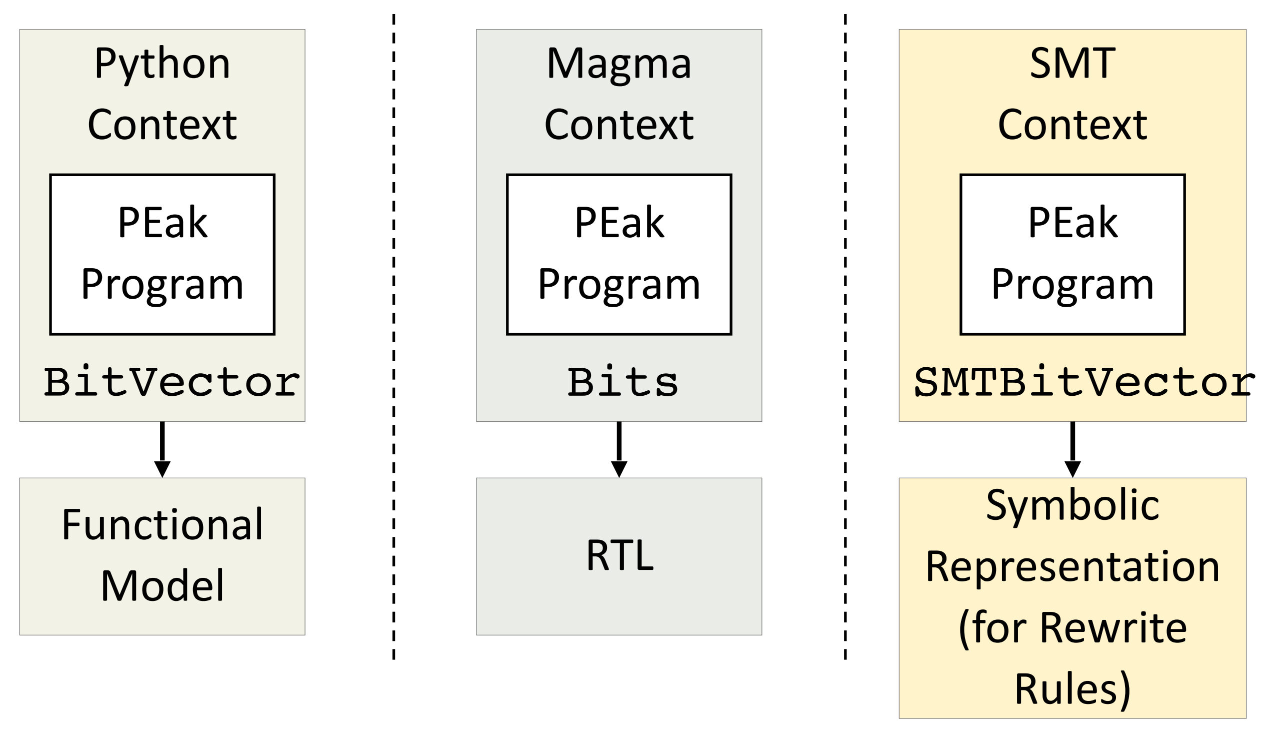
- PEak program uses abstract types provided by the PEak DSL such as `Bit`, `BitVector` etc.
- Each component of the PEak compiler provides a separate concrete implementation of these abstract types
- Multiple interpretations of a PEak specification in different contexts



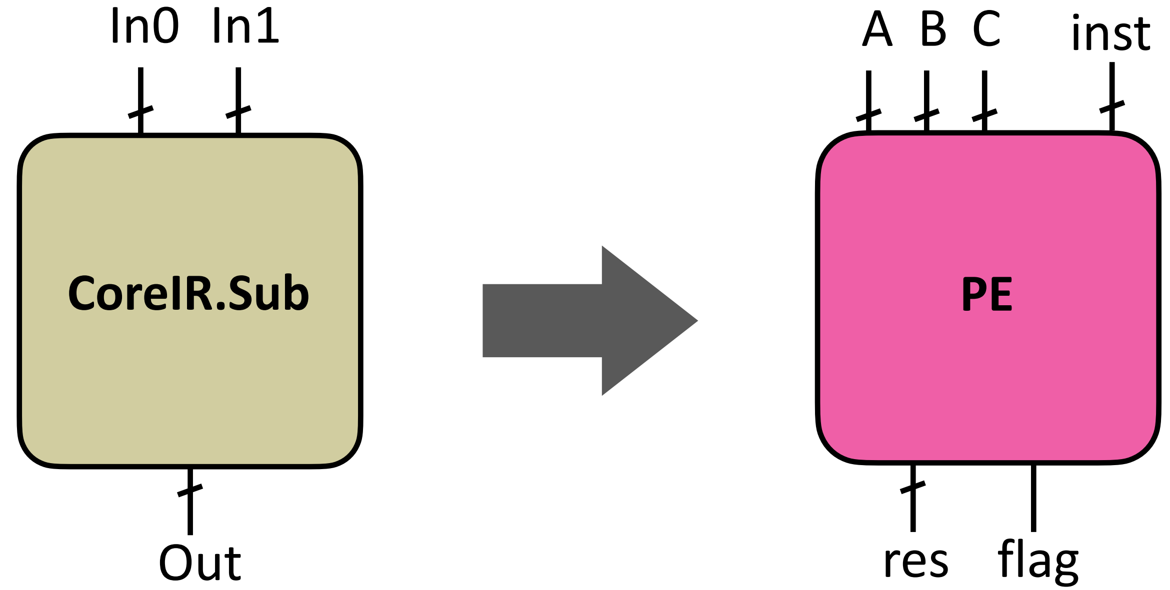


Multiple Interpretations of PEak Specification

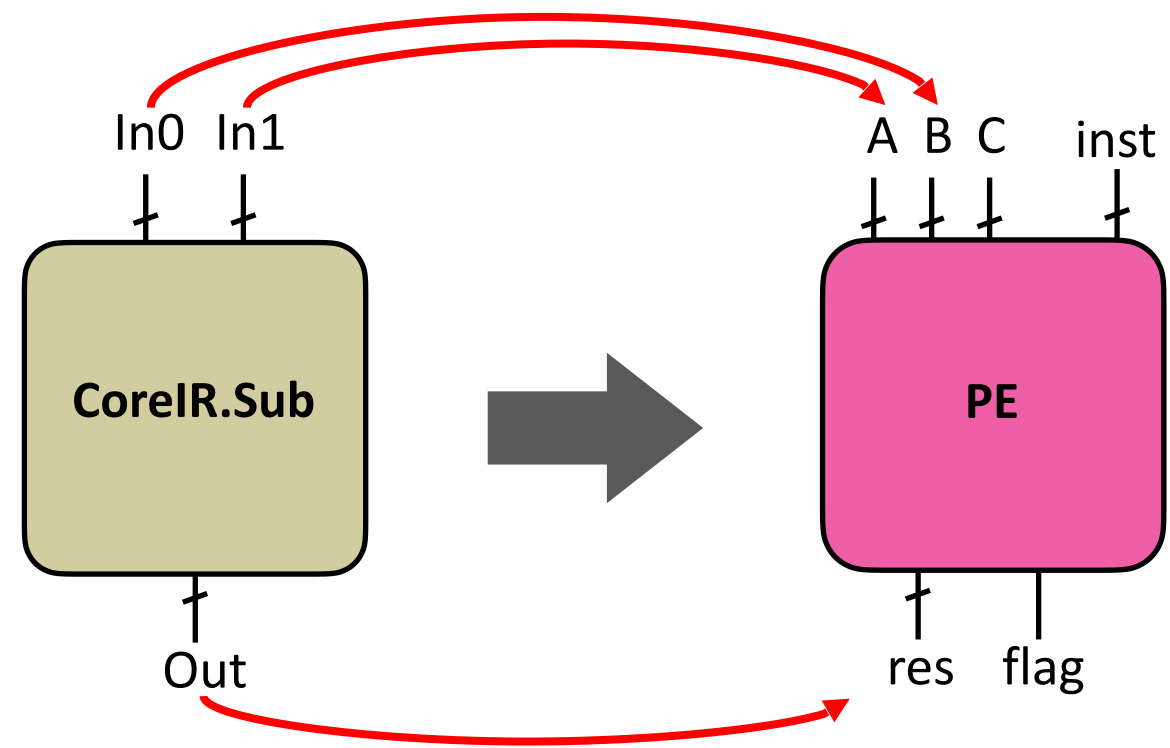
- PEak program uses abstract types provided by the PEak DSL such as `Bit`, `BitVector` etc.
- Each component of the PEak compiler provides a separate concrete implementation of these abstract types
- Multiple interpretations of a PEak specification in different contexts



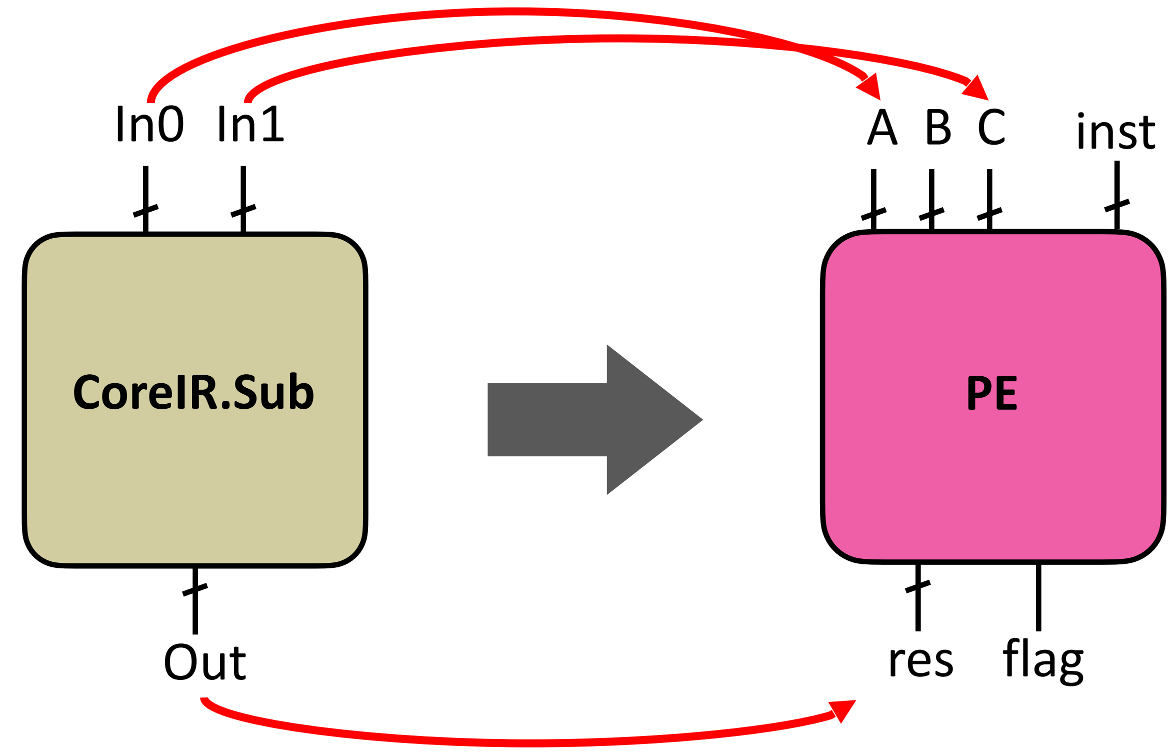
Discovering a Rewrite Rule



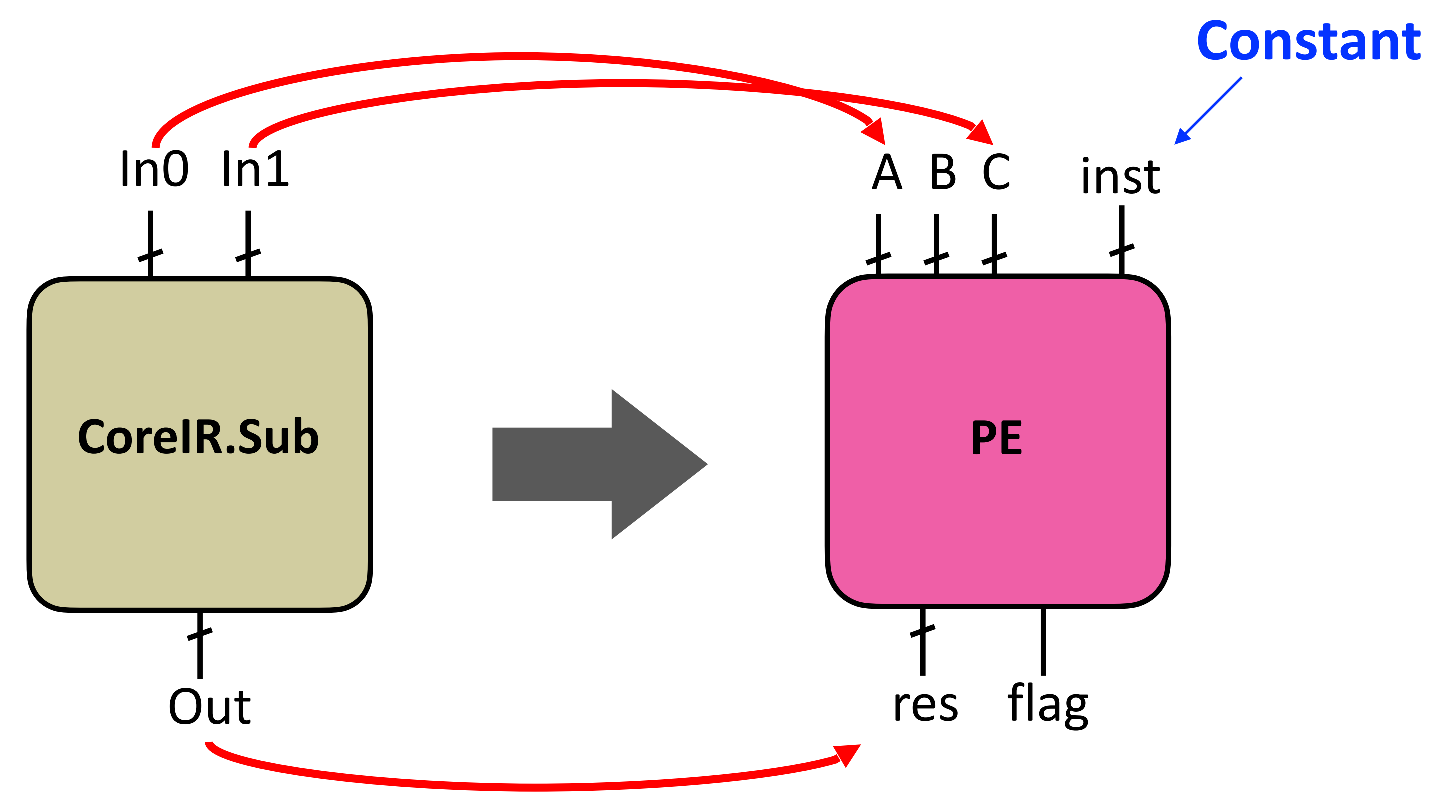
Input/Output Bindings



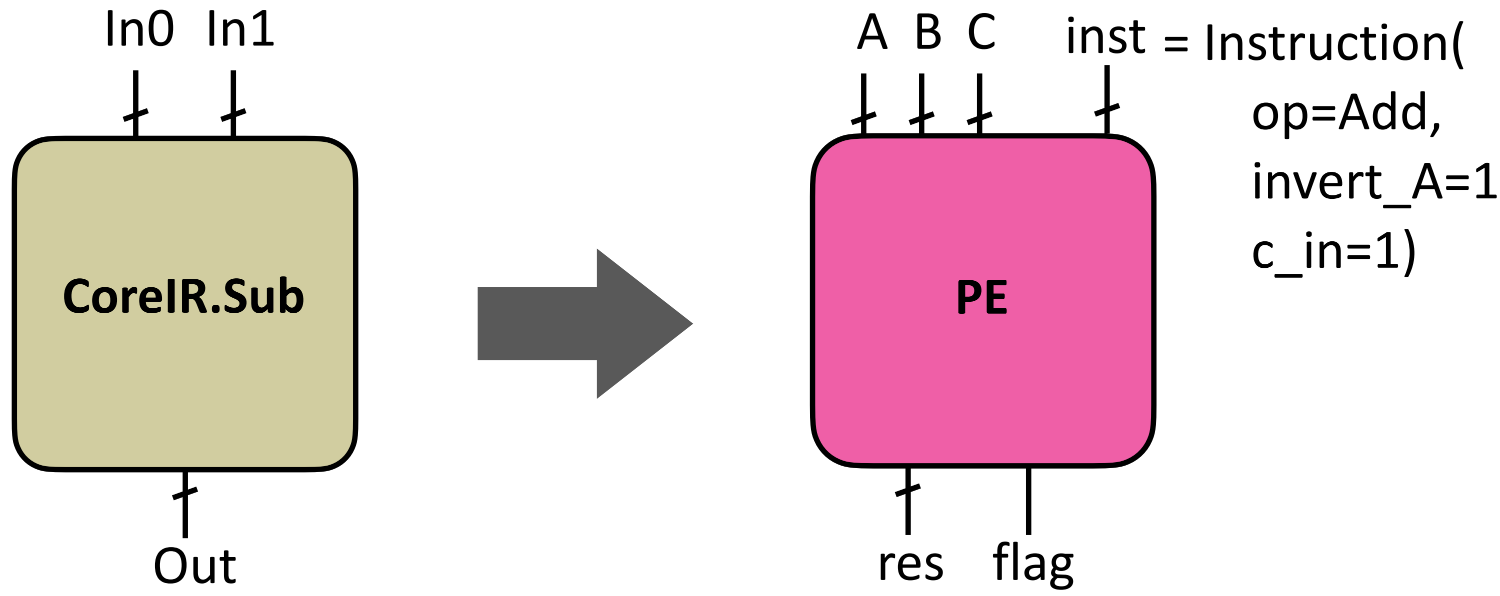
Input/Output Bindings



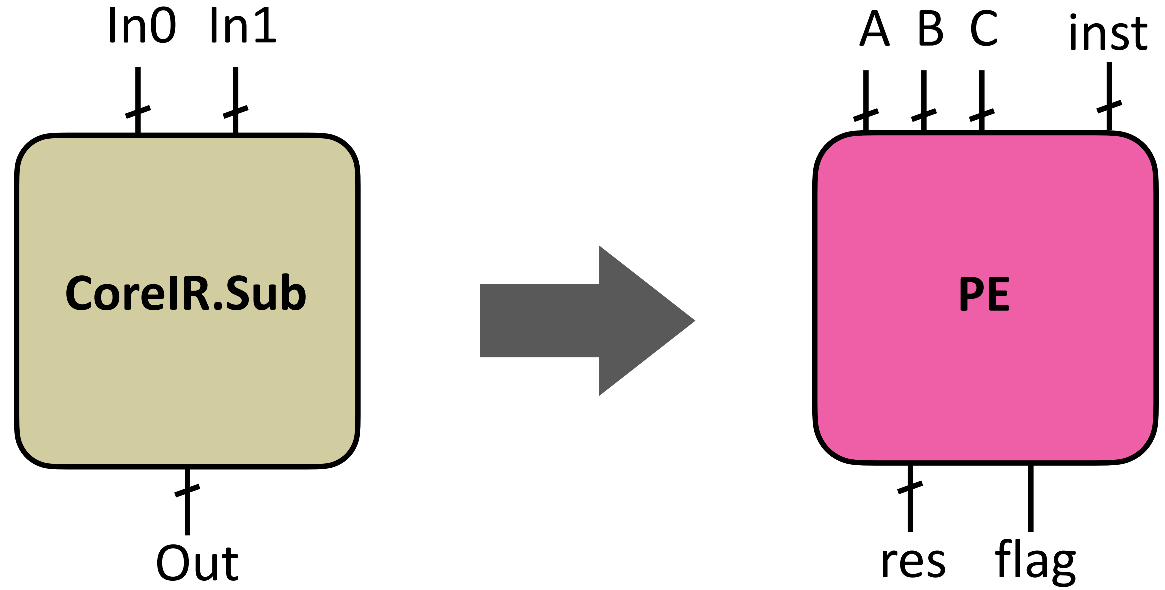
Input/Output Bindings



Setting Constants

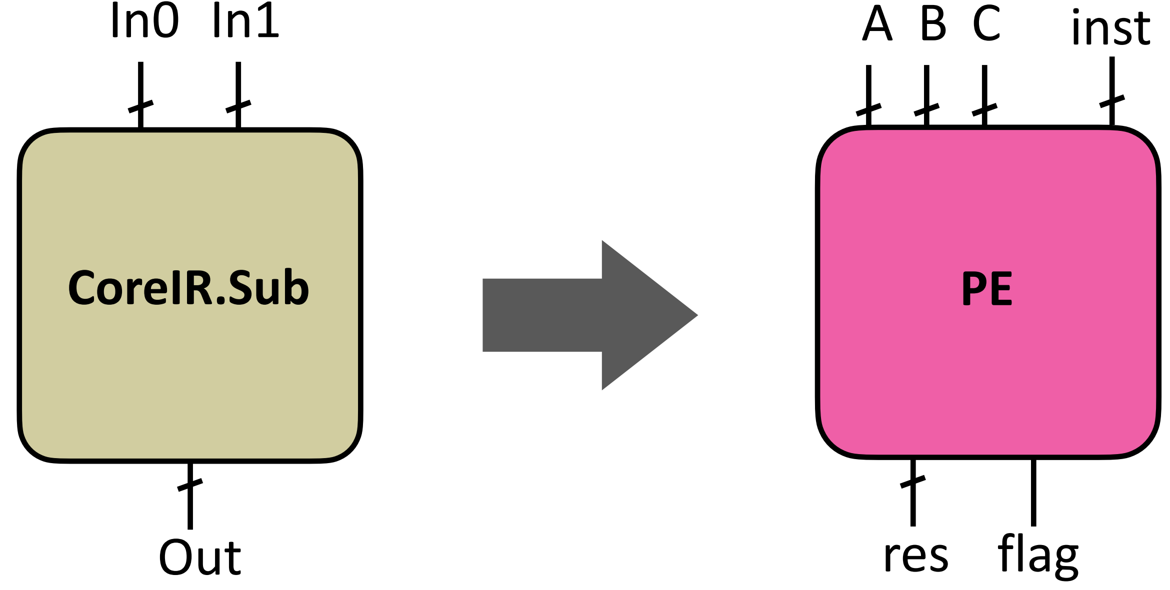


CoreIR.Sub(in0, in1) == PE(inst, input_binding(in0, in1))



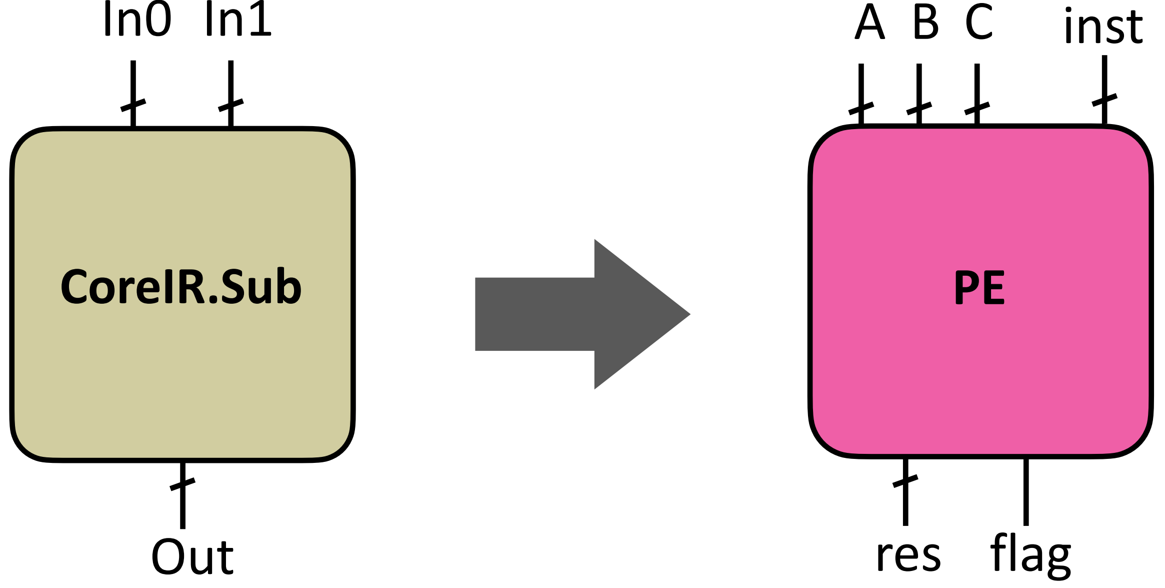
$\exists$(input_binding, inst) st $\forall$(in0, in1):

CoreIR.Sub(in0, in1) == PE(inst, input_binding(in0, in1))



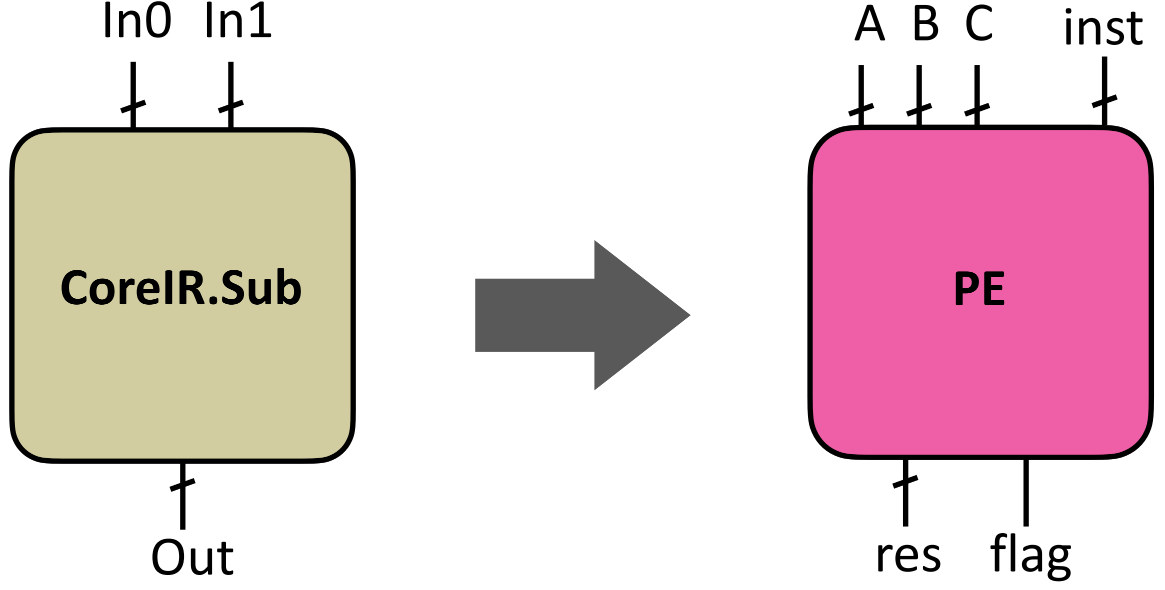
∃(input_binding, inst) st ∀(in0, in1):

CoreIR.Sub(in0, in1) == PE(inst, input_binding(in0, in1))[‘res’]

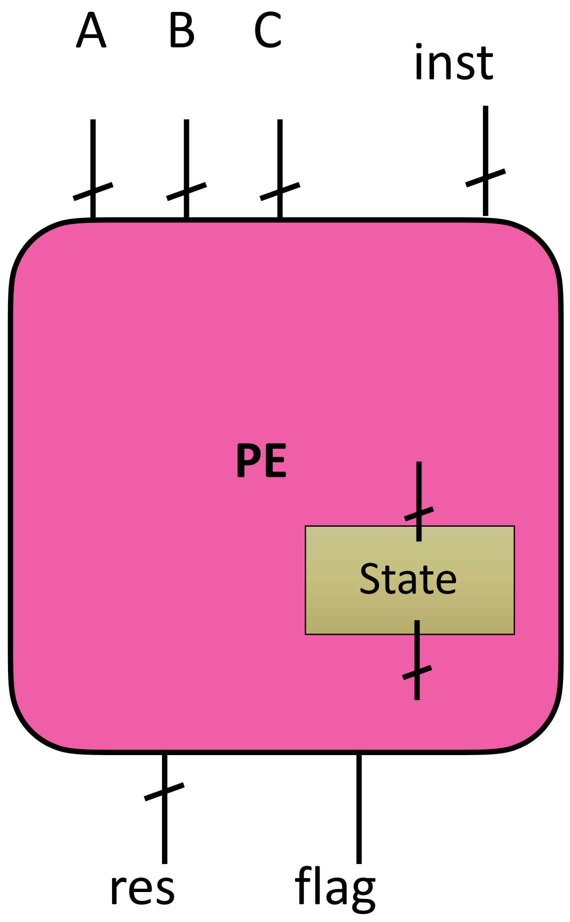


∃(input_binding, inst) st ∀(in0, in1, **other**):

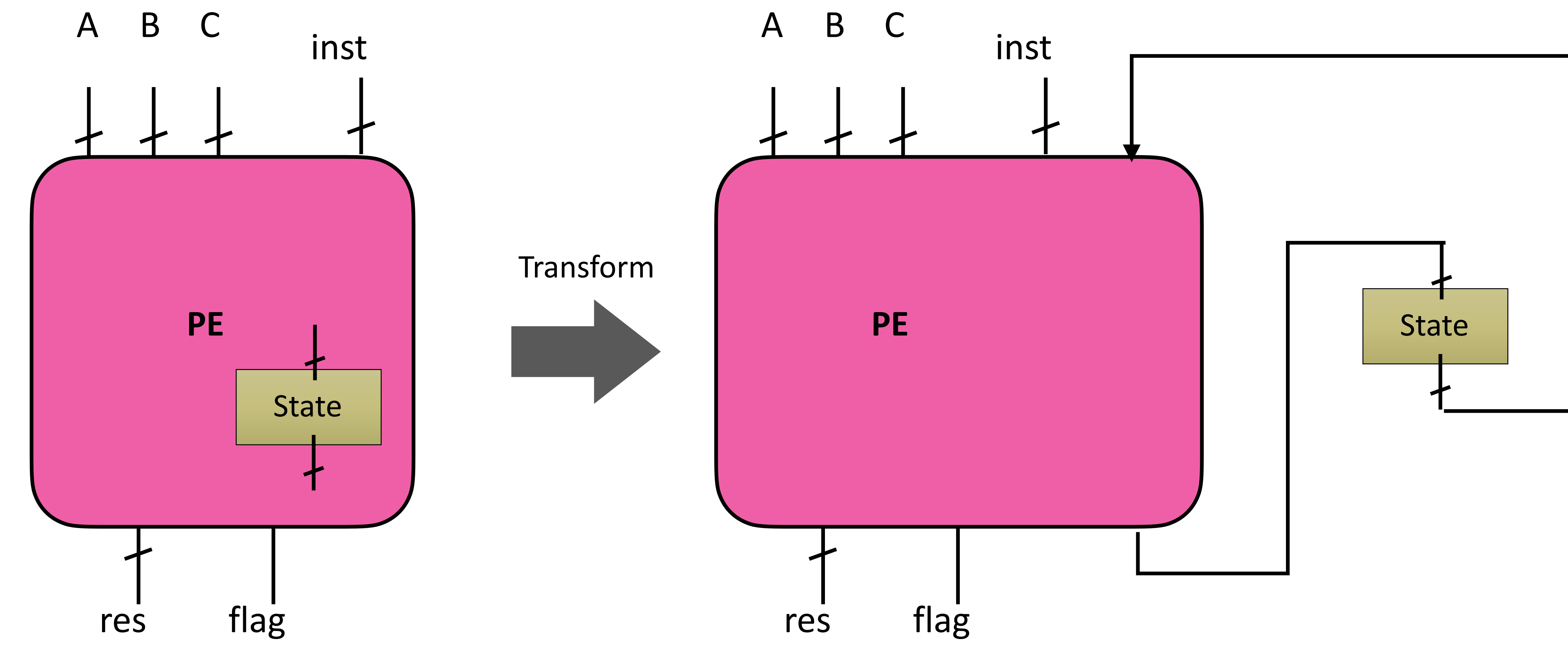
CoreIR.Sub(in0, in1) == PE(inst, input_binding(in0, in1, **other**))[‘res’]



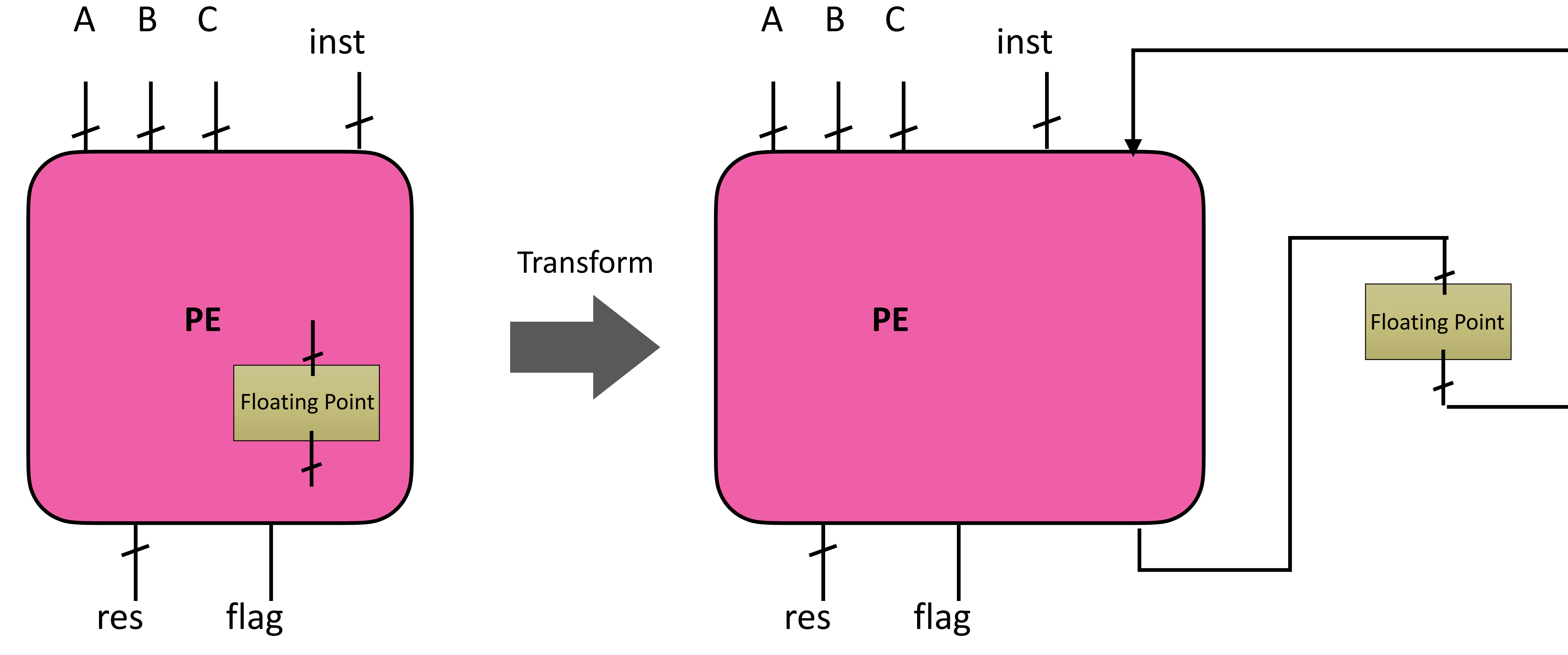
How to Handle State?



How to Handle State?



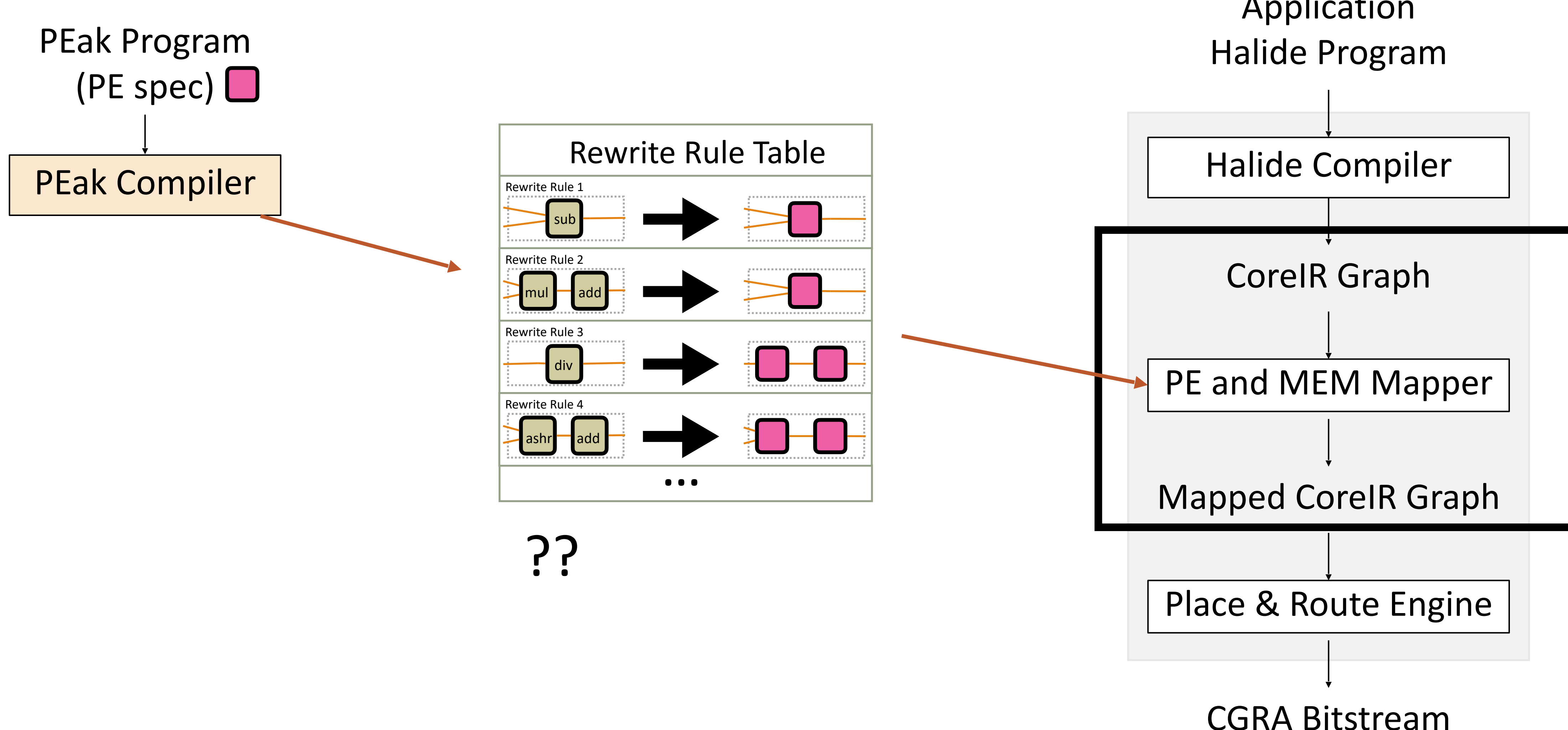
Floating Point?



Performance of Rewrite Rule Generator

- Problem: Universally Quantified SMT queries can take a long time
- Solutions:
  - It is okay to be slightly slow (unless doing DSE!)
  - Different ways to encode the final formula
  - Different techniques for solving Quantified Expression
- Recent results:
  - ~ 1 minute to solve 20 rewrite rules on our current CGRA.

What patterns to use in the rewrite rule table?

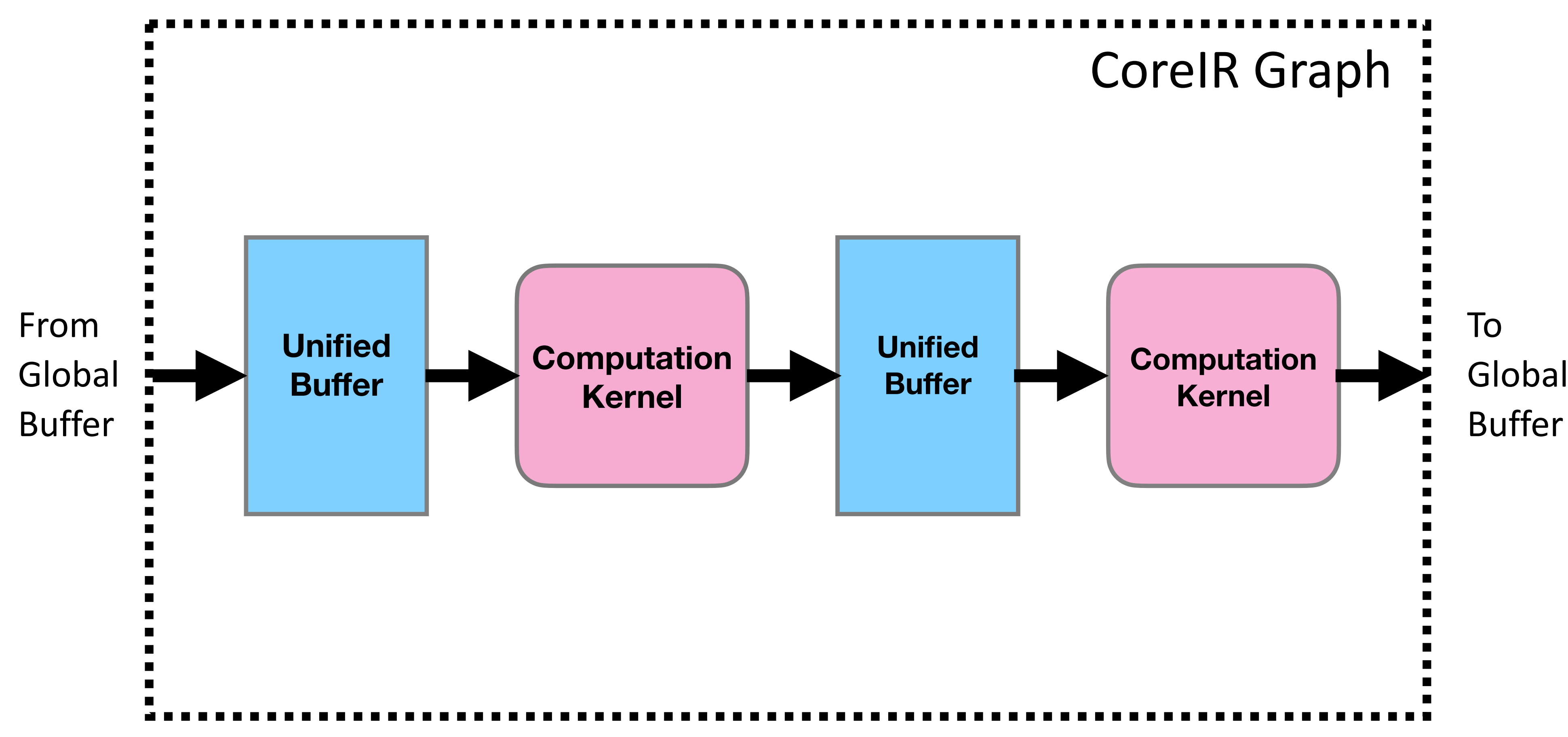


Which Patterns?

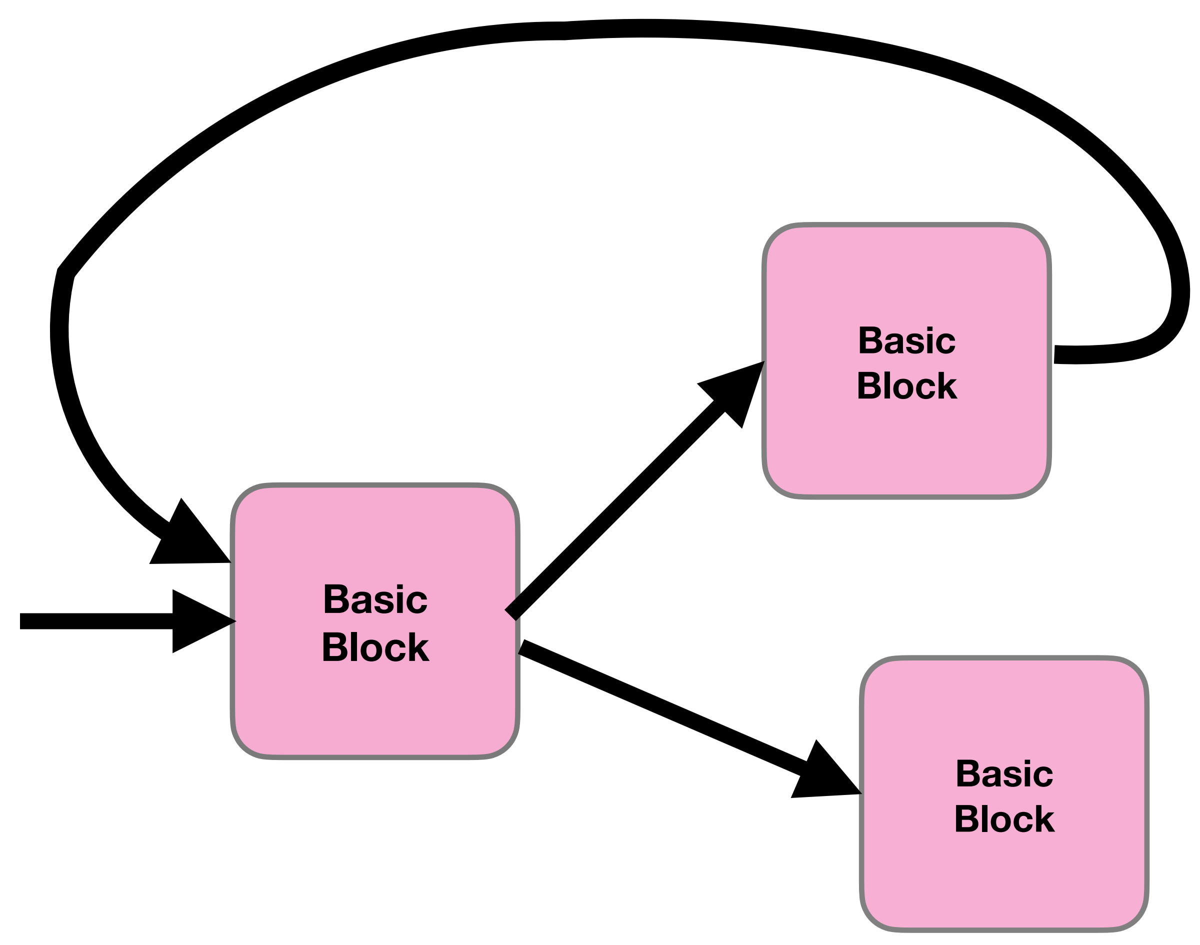
- Enumerate all possible patterns up to a size
  - Lots of uncommon patterns
  - Bloated Rewrite Rule Table
    - Slower instruction selection
- Analyze target domain’s applications for common subgraphs
  - Approach used for our upcoming DSE paper
- Only very basic patterns
  - Use peephole optimization/packing after instruction selection

CPU Instruction Selection

CGRA Compilation



Control Flow Graph



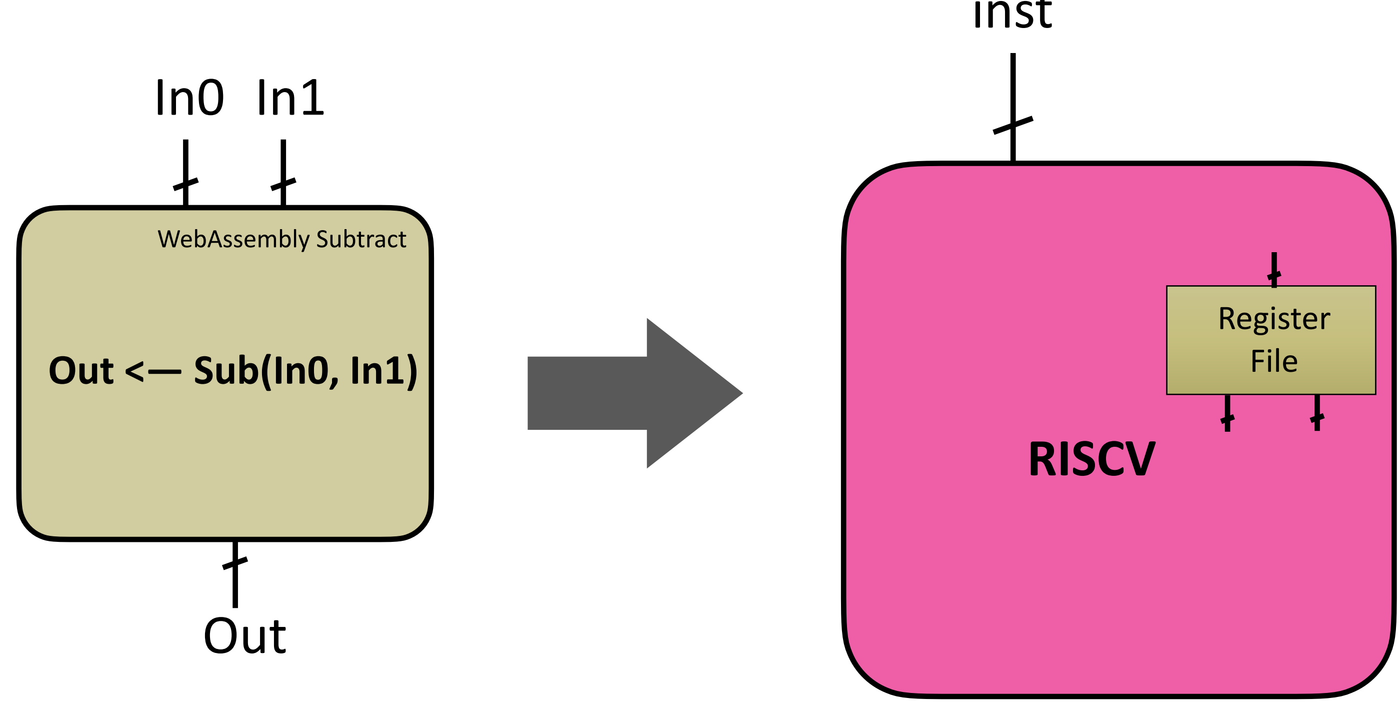
Basic Block

(Machine independent)

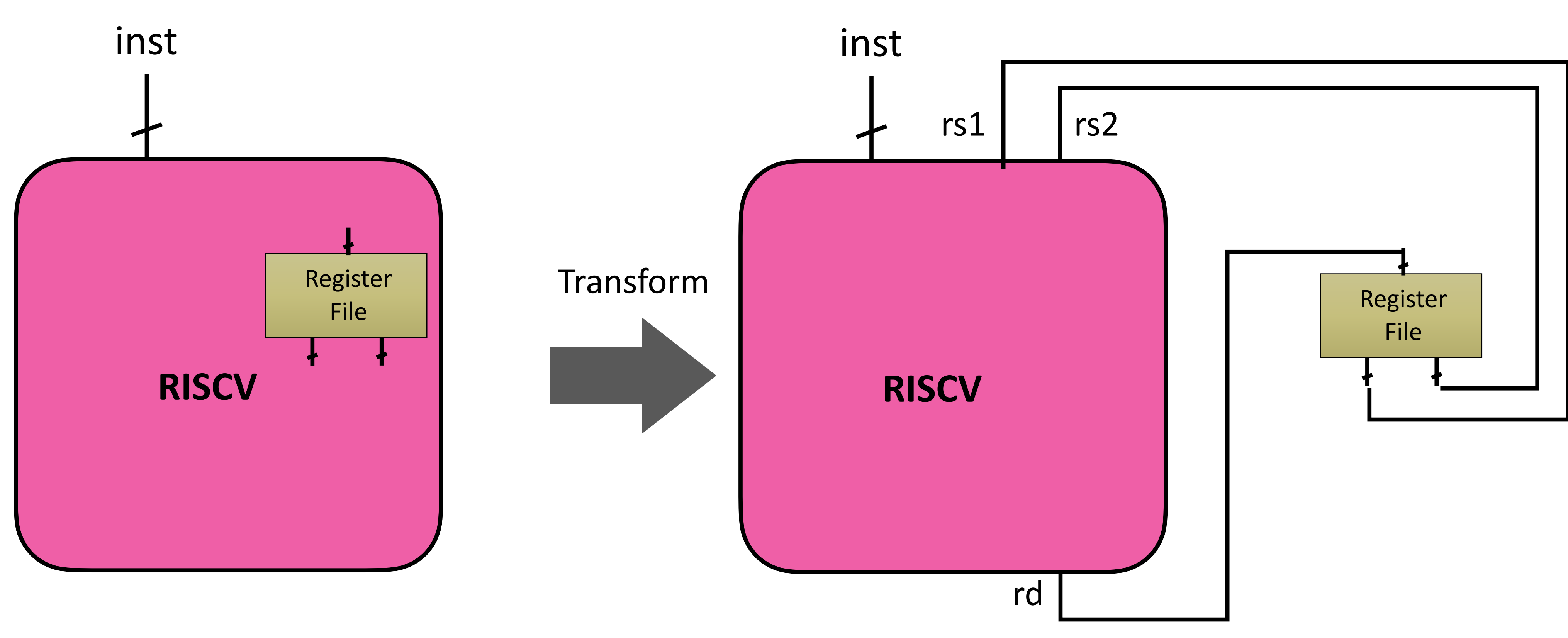
Basic Block

R2 <— Sub(R0, R1)
R3 <— M[R2]
M[R3] <— R1
R4 <— Add(R1, 0x50)
...

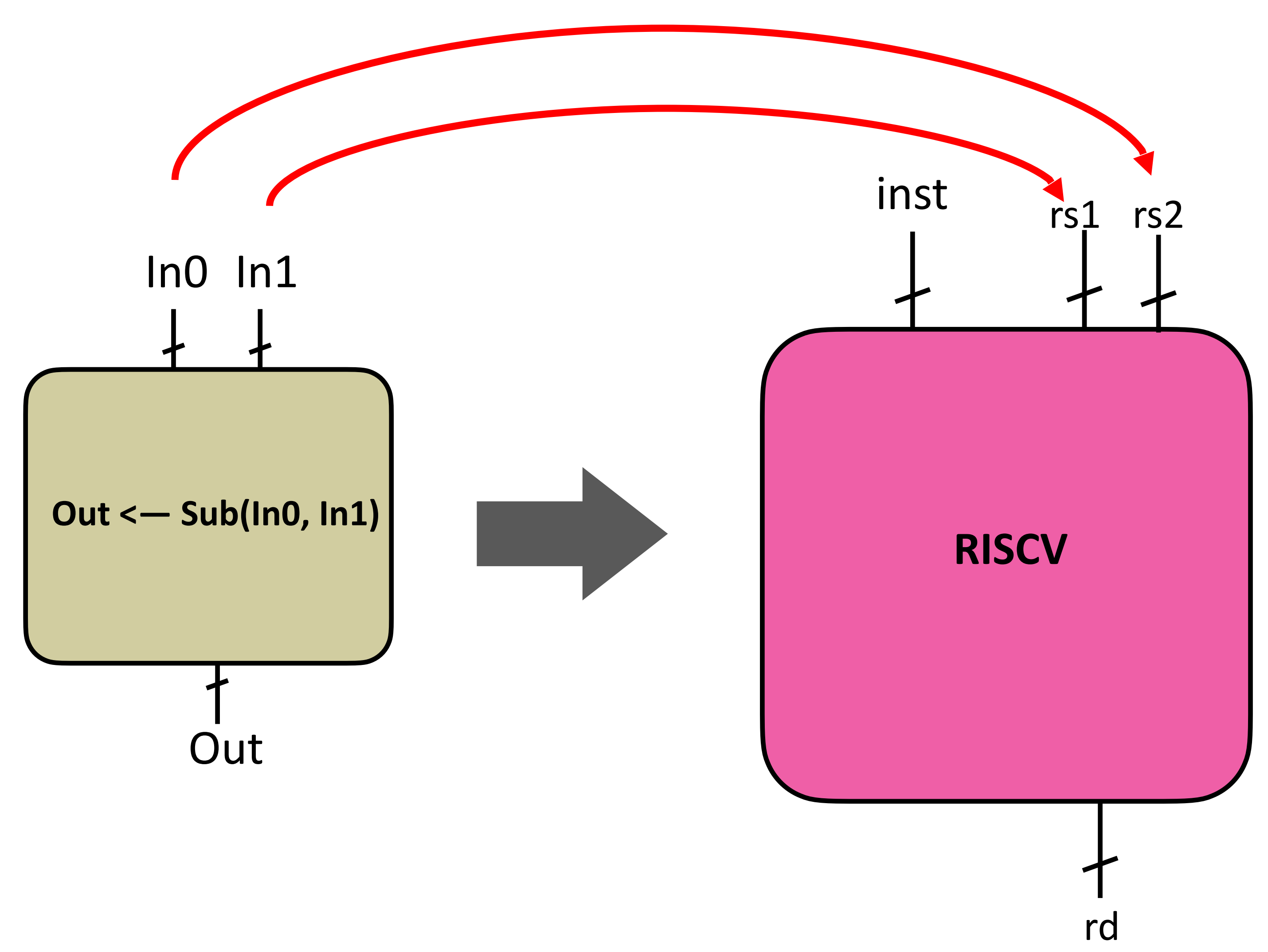
Compiling WebAssembly to RiscV?



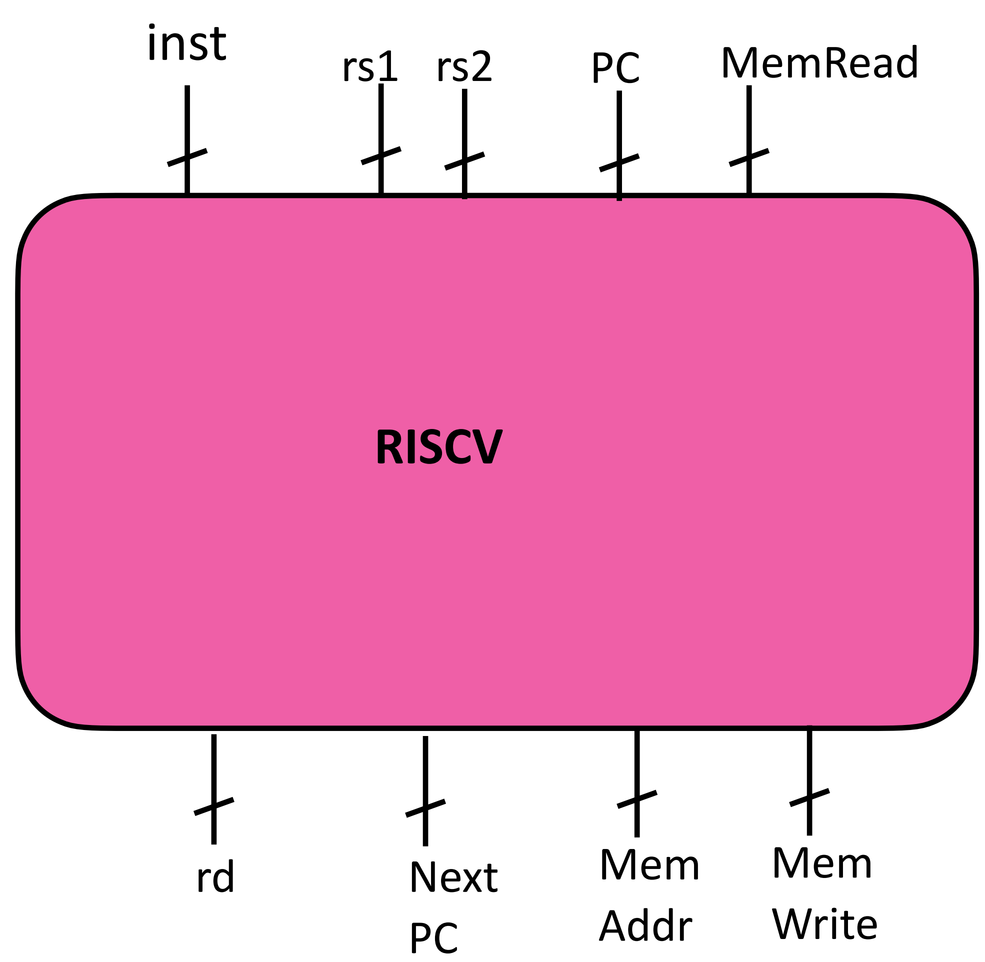
Transform RiscV to remove Register File



Discovering Subtract



Branch/Memory Instructions?



The Future

- Goal: Fully Automatic compiler generation for Accelerator Architectures

Thank You