Performance Counter Design Variation in Rocket Chip via Feature-Oriented Programming

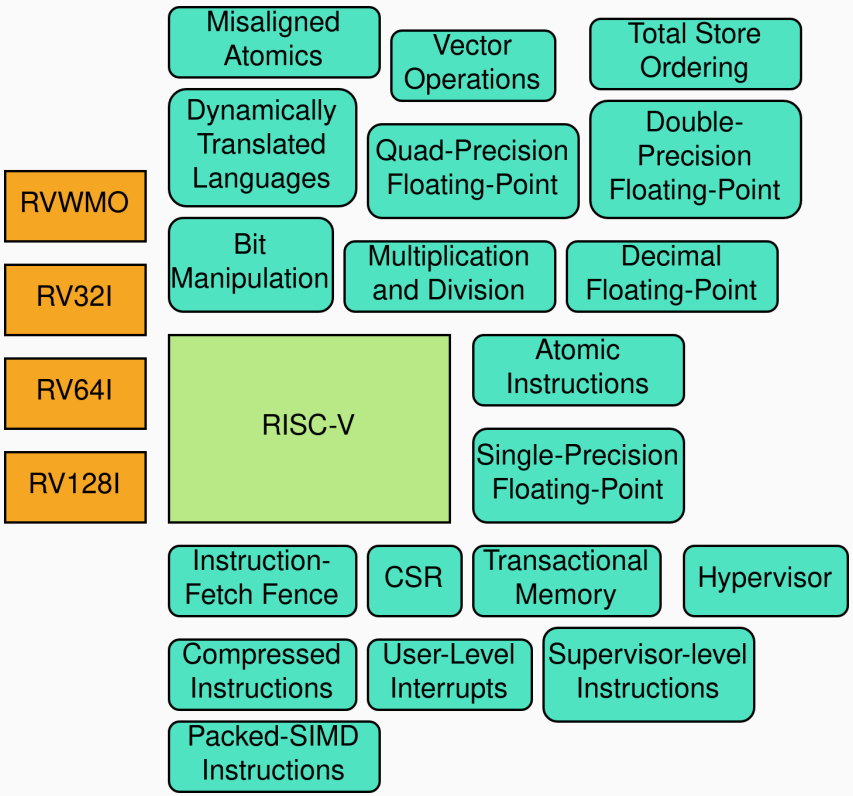
Justin Deters & Ron Cytron

Washington University in St. Louis
j.deters@wustl.edu

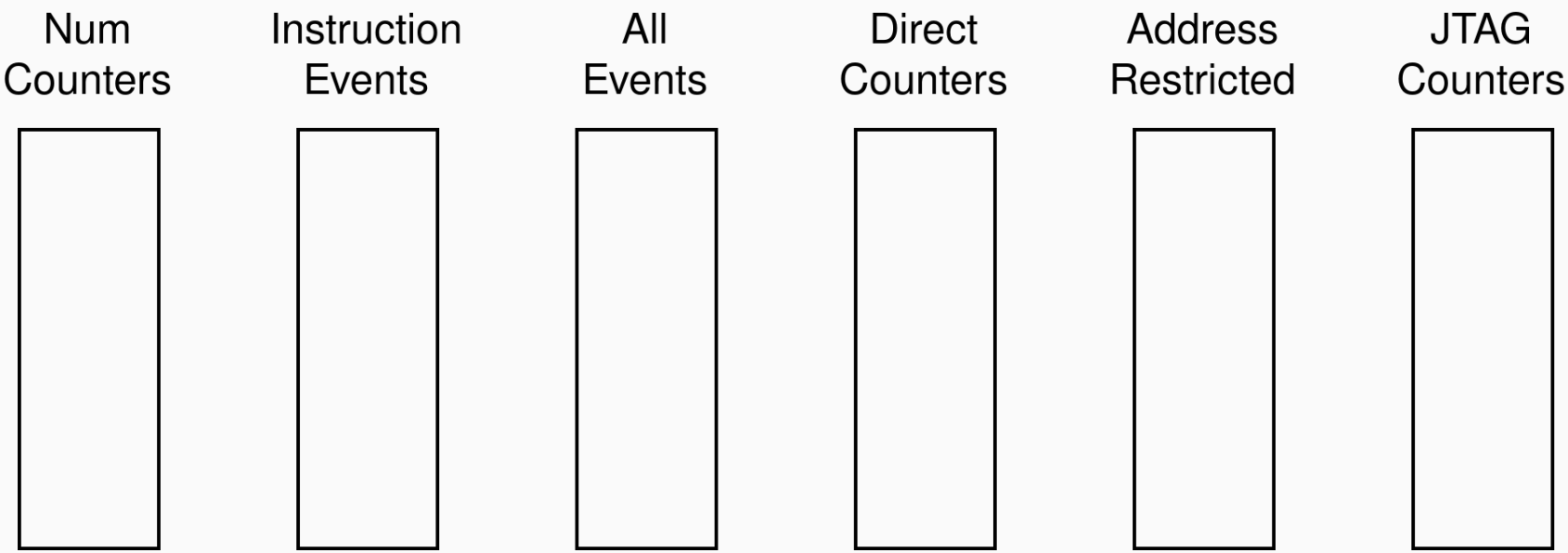
Supported under NSF CISE award CNS-1763503 Performant Architecturally Diverse Systems via Aspect-oriented Programming

Features in RISC-V

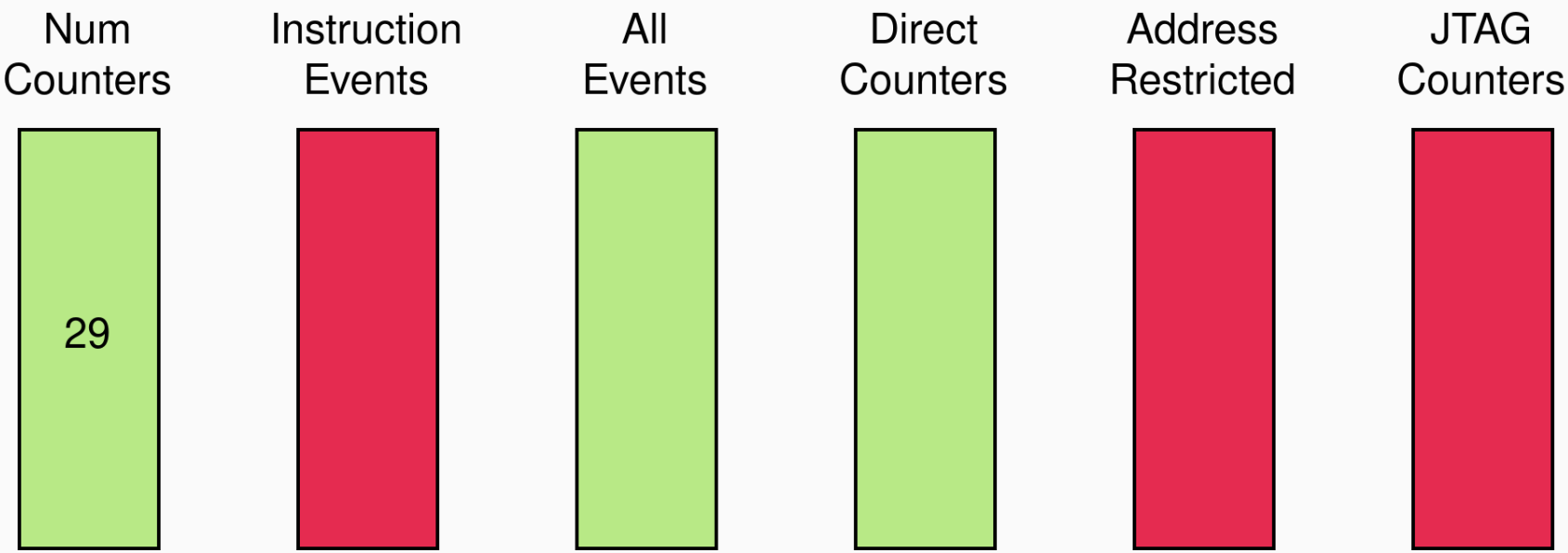
- RISC-V implementations need to be adaptable to be successful.
  - Not all features are needed all the time.
  - Sometimes we seek to augment features.
- We demonstrate this using Rocket Chip's performance counters.



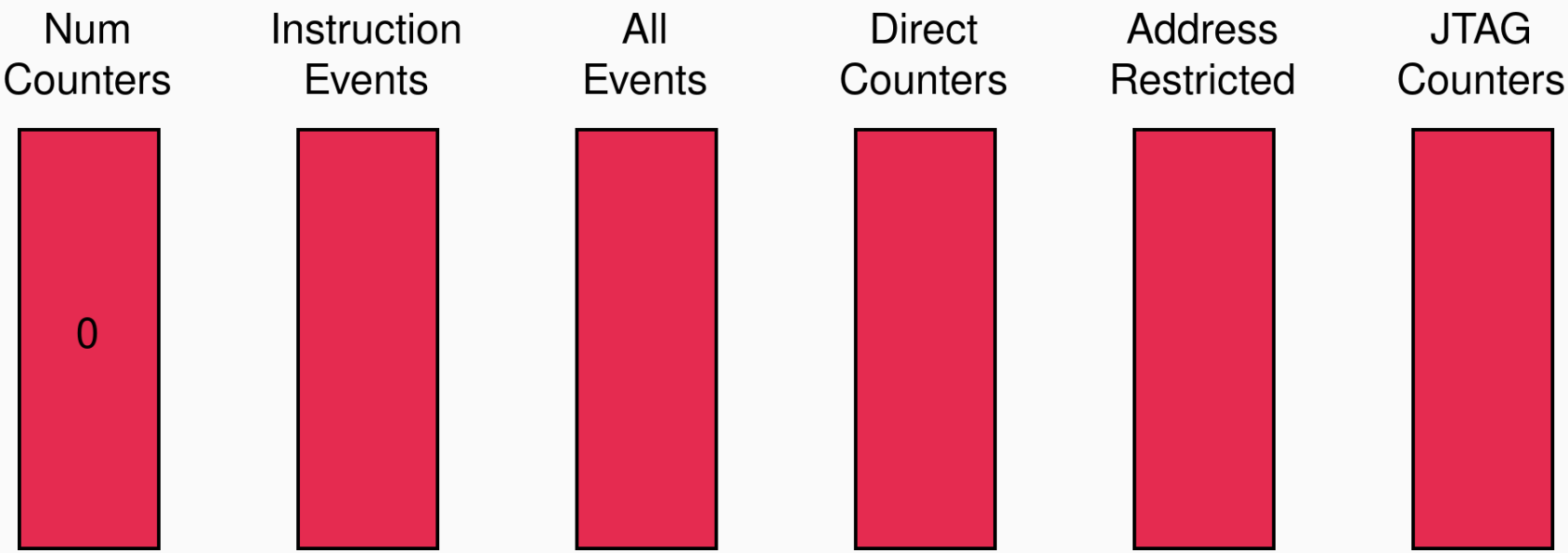
Feature Choices



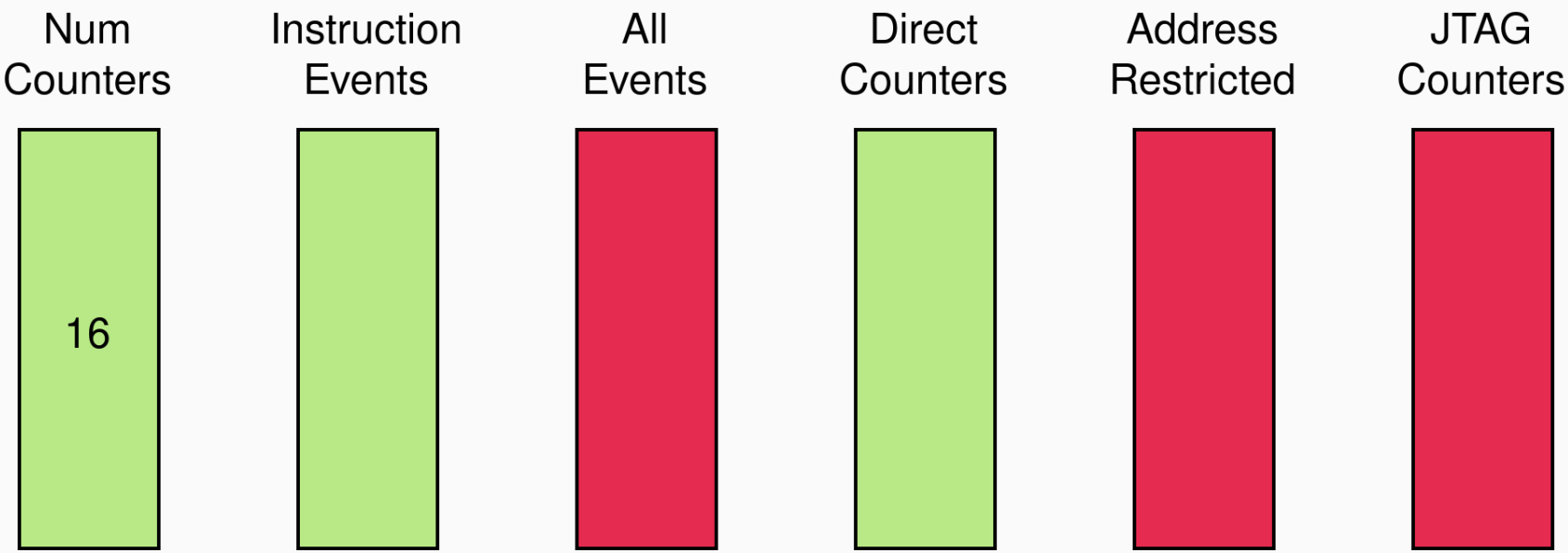
Standard Rocket Chip



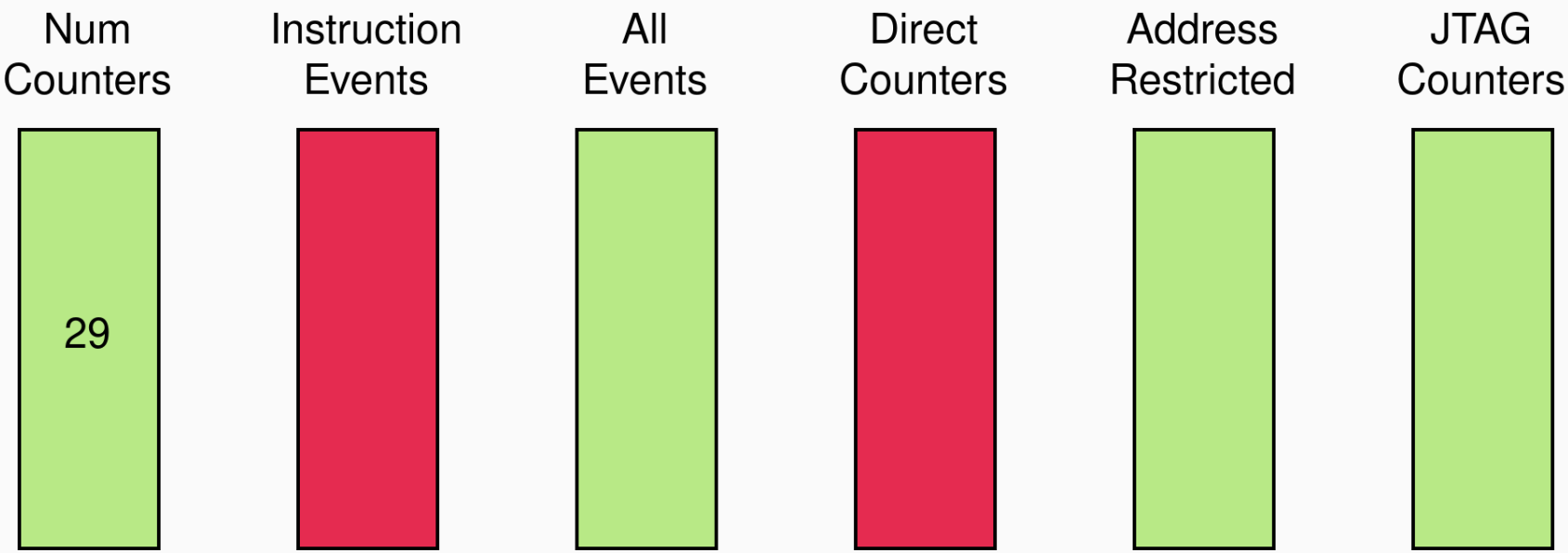
Embedded Core



Small Core


Num Counters
16
Instruction Events
All Events
Direct Counters
Address Restricted
JTAG Counters

Debug Core

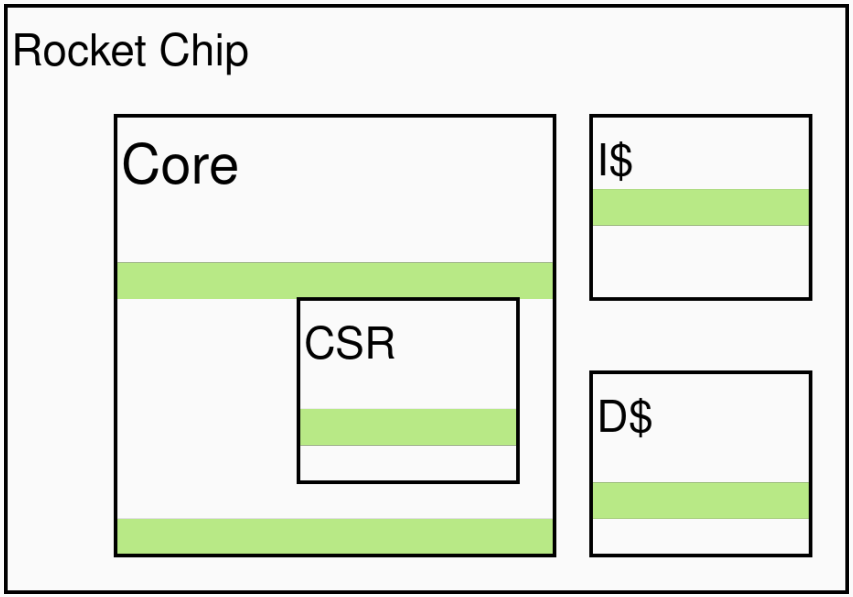

Num Counters
29
Instruction Events
All Events
Direct Counters
Address Restricted
JTAG Counters

How do we mix these features together?

What happens when we do this?

Current Monolithic Design

- The naive approach includes all features in *`If-Then-Else`* blocks.
- Including all features quickly becomes unmanageable.
- *Monolithic* design obscures where the system starts and ends.

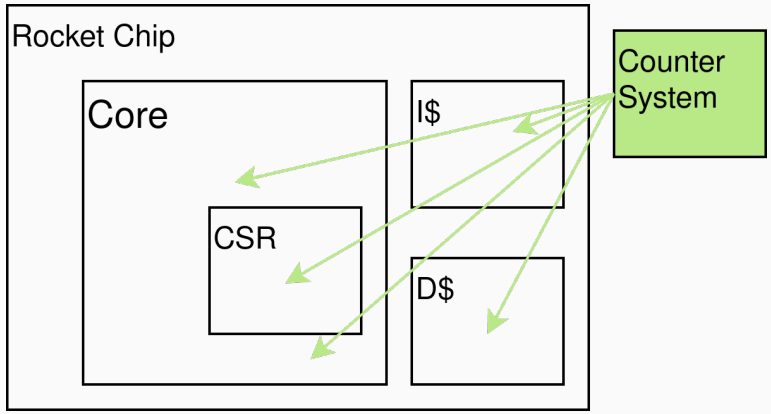


- Hard coding and entangling features complicates maintenance and extension.

Instead deconstructing a monolithic version, why not construct a version with only the features we need?

Feature-Oriented Programming

- We follow a feature-oriented approach to introduce features and their variations into a core implementation.
  - Obtain a foot print with only the features we need.
  - Structure the code to accommodate future variations easily.



- Instead of including everything, break the performance counter system into user selectable feature units.
- Use *aspect-oriented programming* to apply selected features.
  - Aspects capture what and where code should be added.
  - Conditionally apply aspects to “weave” desired features.

Contribution: Feature Application using Scala Trees (Faust)

- We modify the Scala abstract syntax trees with feature information.
- Faust can modify any part of the generator.
- We hook directly into the type system of Scala/Chisel.
- Faust packages features into aspects.

```
1  trait CSRHardware {
2    def buildDecode(): Unit
3    def buildMappings(): Unit
4  }
5
6  class CSRFile() with CSRHardware {
7
8    buildMappings()
9
10   buildDecode()
11
12   def buildMappings() = {
13     //mapping code
14   }
15
16   def buildDecode() = {
17     //decode code
18   }
19 }
20
21 abstract class PerfCounters()
22    extends CSRHardware {
23
24   def buildMappings() = {
25     //mapping code
26   }
27
28   def buildDecode() = {
29     //decode code
30   }
31 }
```

Feature DSL

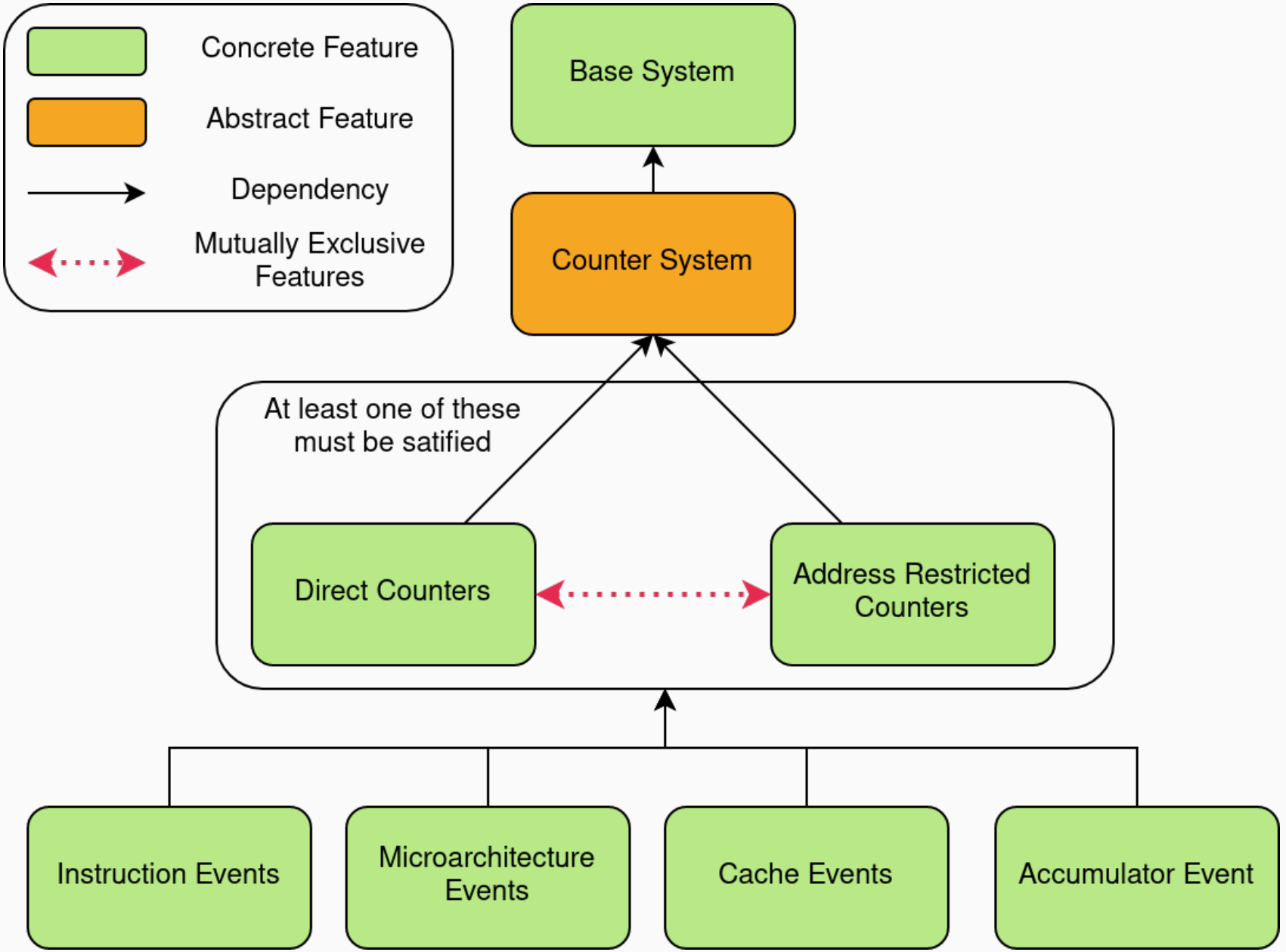
- Faust borrows syntax from aspect languages.
- Users just need to extend the *Feature* class.

Example

```
1 class CounterSystemFeature (numCounters: Int) extends Feature {
2   before (q"buildMappings()") insert (q"val numRealCounters =
      $numCounters") in (q"class CSRFile") register
3
4   after(q"buildMappings()") insert q"performanceCounters.
      buildMappings()" in (q"class CSRFile") register
5
6   before (q"buildDecode()") insert (q"performanceCounters.
      buildDecode()") in (q"class CSRFile") register
7 }
```

- Easily package features and add them to Faust.

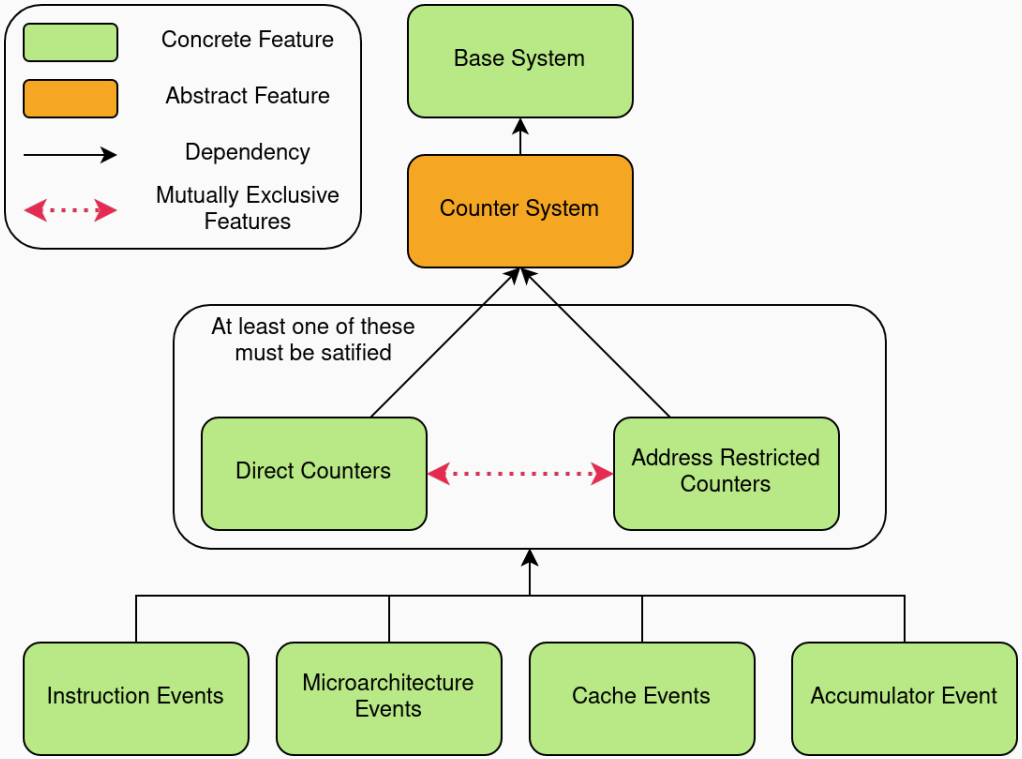
Dependency Management



When are events counted?

Direct Counters

- The standard way Rocket Chip collects event information.
- All events are counted at all times if configured.



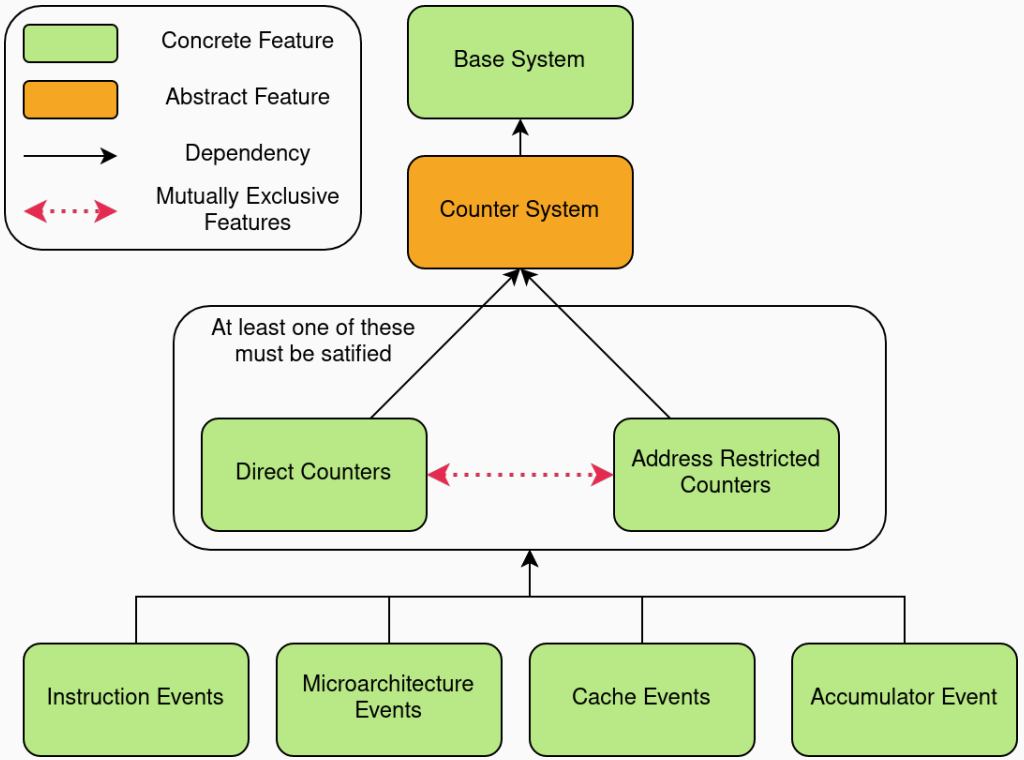
Address Restricted Counters

- Events are only counted when the PC is within a specific address range.
- Feature users can customize the address range.

Which events are counted?

Instruction, Microarchitectureal, & Cache Events

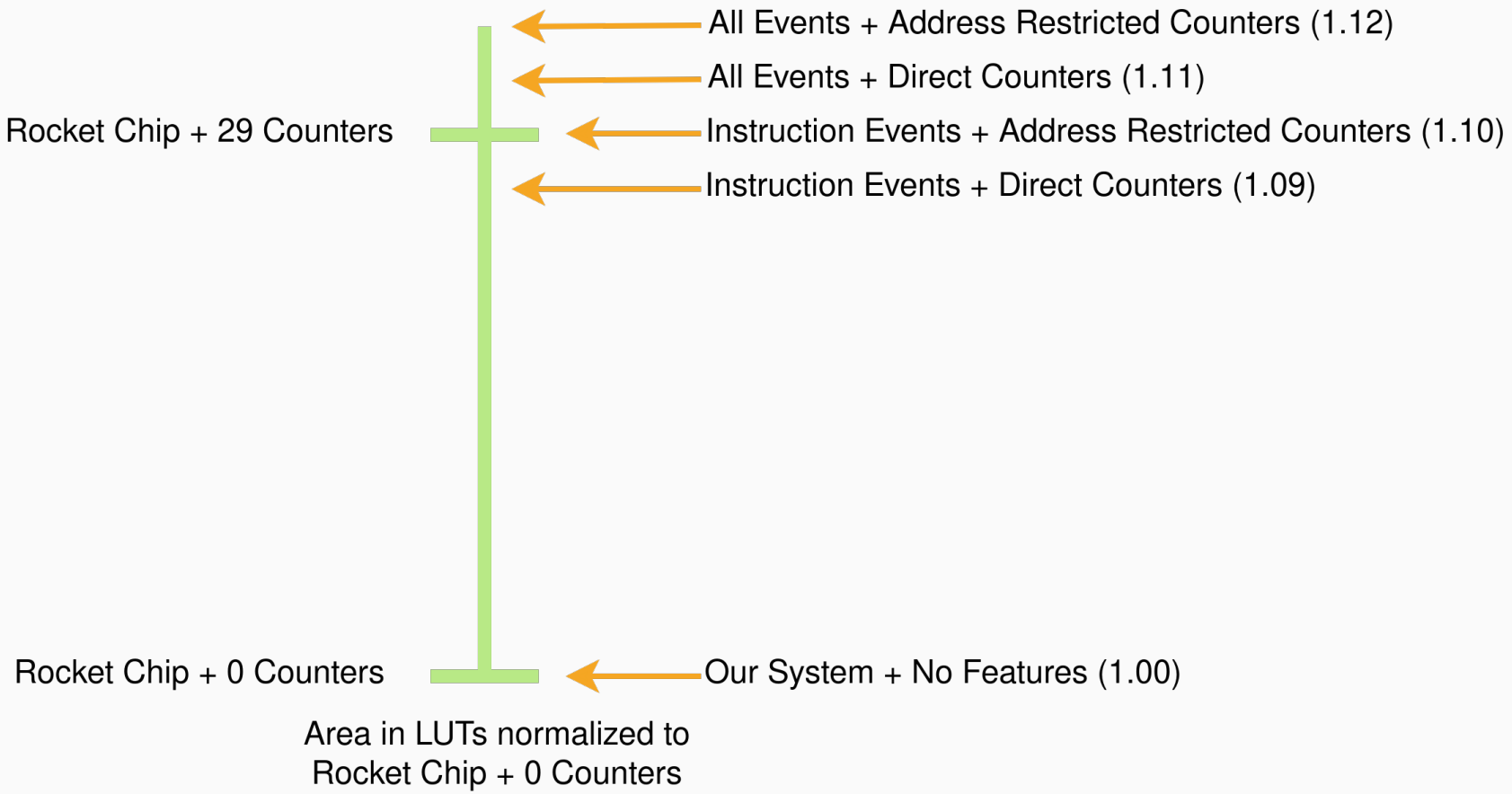
- These are the events provided by Rocket Chip.
- These groupings are arbitrary and could easily be more atomized.



Accumulator Event

- Simple event from the Accumulator RoCC accelerator.
- Any accelerator could be adapted to provide event information.

Endpoint Design Variations



- The base implementation has 24056 LUTs.
- Only pay for features that we actually want.
- Easily compare different design endpoints.

Our feature oriented design can save space!

Monolithic implementations leave space savings on the table and are tedious to start with.

Conclusions and Future Work

Our System

- Compossible
- Extendable
- Simple
- Cheap

Future Work

- Bring feature-oriented design to other parts of Rocket Chip.
- Work directly with Rocket Chip authors to improve the type system.

Feature-oriented design provides a viable path for RISC-V implementations to be tailored, extendable, and easy to understand.