Stanford | ENGINEERING
Electrical Engineering
Computer Science

Spatial: A Language and Compiler for Application Accelerators

David Koeplinger

Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Tian Zhao

Ruben Fiszel, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, Kunle Olukotun

Spatial Resources

- Language documentation and tutorials:
  - spatial.stanford.edu
- Spatial Google Group:
  - groups.google.com/forum/#!forum/spatial-lang-users
- Spatial Github Repo:
  - github.com/stanford-ppl/spatial-lang

Increasing Demand for Reconfigurability

There is growing demand for **reconfigurable architectures** as **application accelerators** in data centers for performance and energy efficiency

Improvements in Performance, Energy Efficiency

Baidu 百度

Software Defined Accelerators for DNNs
Distributed real-time deep learning on FPGAs
[HotChips '14]

2 – 4x perf vs. CPU/GPU
2 – 3x perf/W vs. CPU/GPU

XPU: Programmable FPGA Accelerator
[HotChips '17]

64x perf vs. CPU
25x perf/W vs. CPU

Microsoft

Catapult "Configurable Cloud"
Distributed application and network acceleration
*"…already been deployed at **hyperscale** and is how most new Microsoft data center servers are configured."* [Caulfield et. al. Micro '16]

~10x perf vs. CPU
~ 9x perf/W vs. CPU

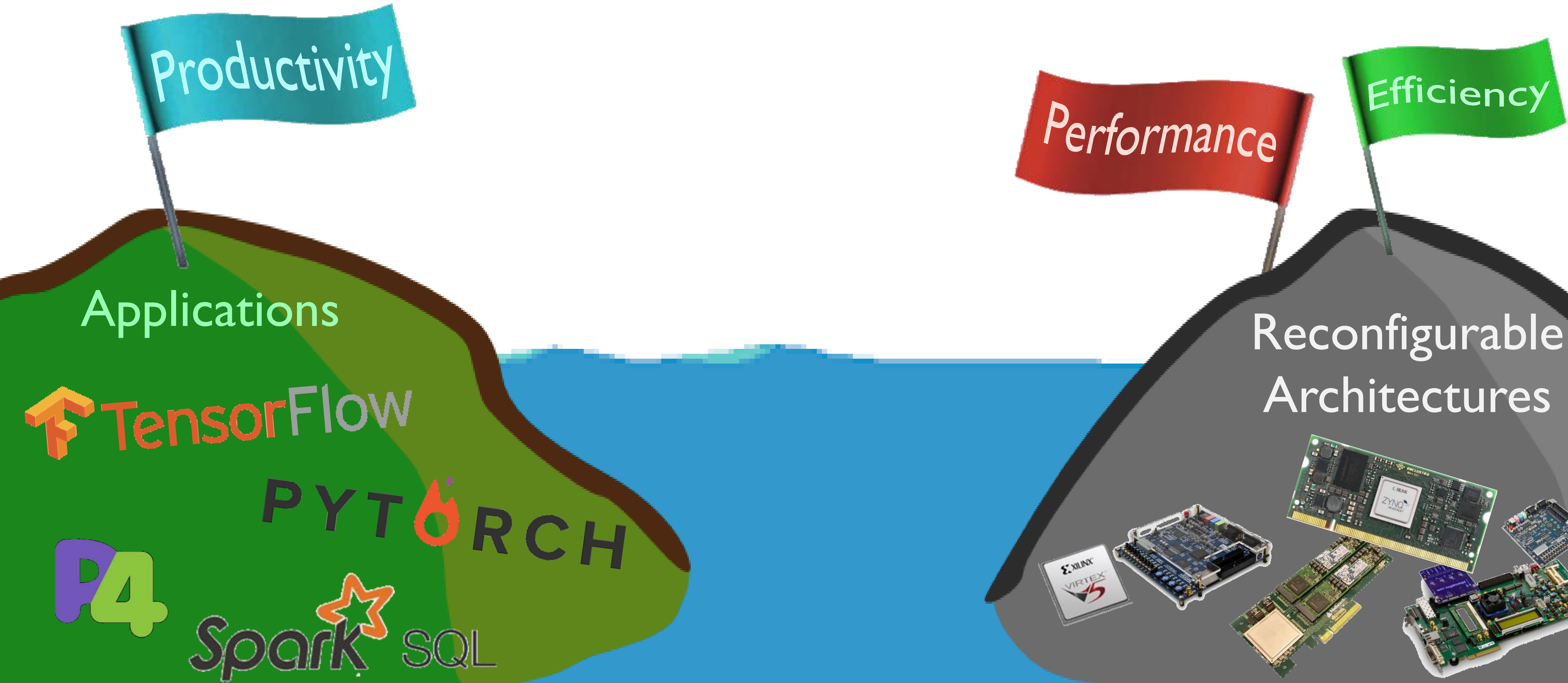
Project Brainwave
Distributed real-time deep learning on FPGAs
[HotChips '17]

PLASTICINE

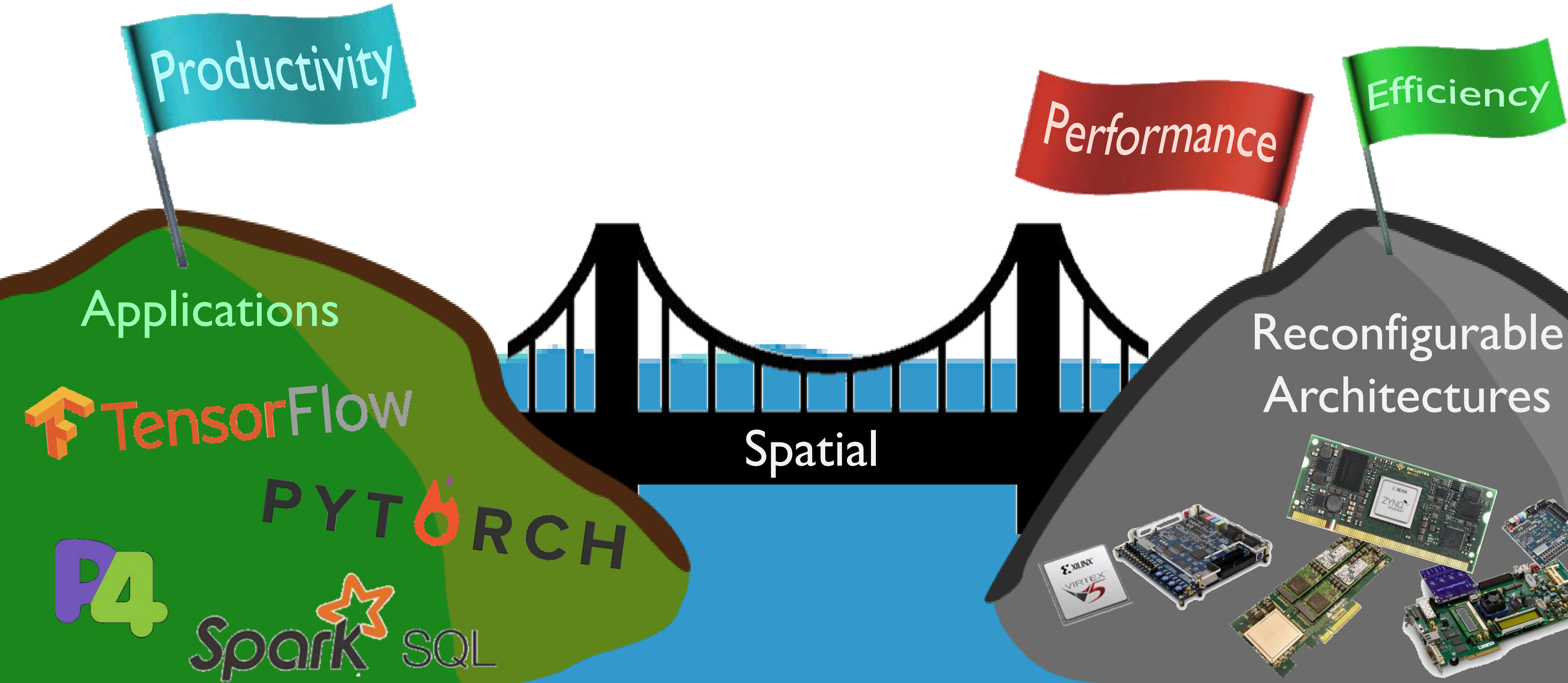
CGRA for Parallel Patterns
[ISCA '17]

Up to 95x perf vs. FPGA
Up to 75x perf/W vs. FPGA

Accessing Reconfigurable Architectures



Accessing Reconfigurable Architectures



Key Programming Challenges

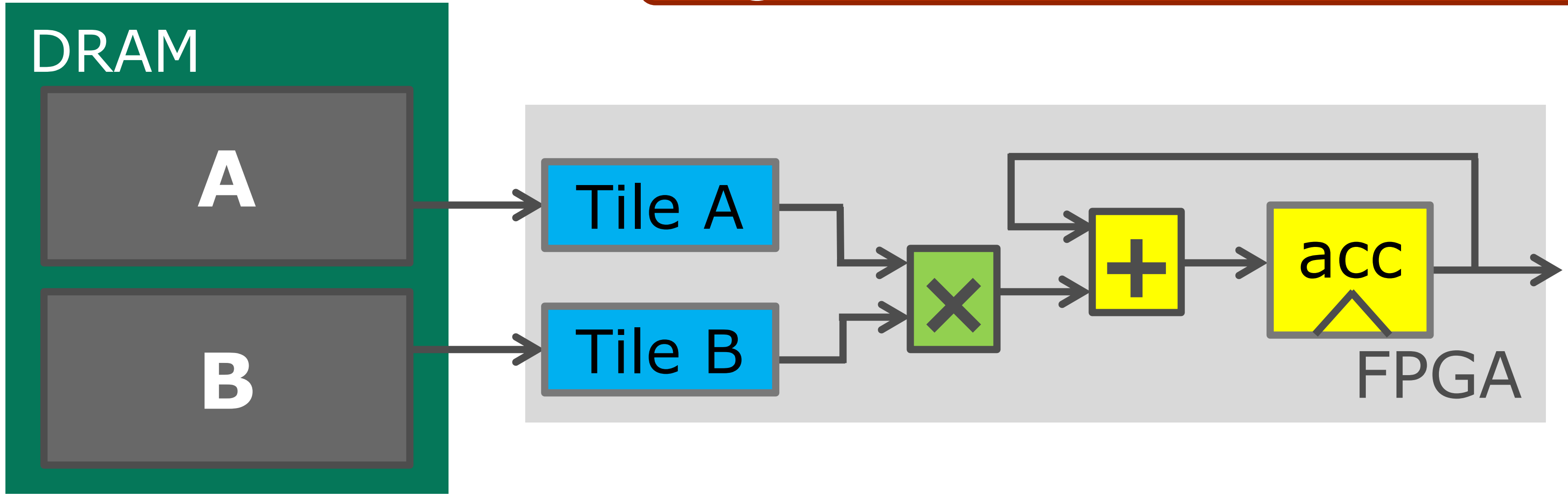
- Pipeline timing
  - Fine-grained timing of control signals
  - Management of multiple clock domains
- Disjoint memory spaces
  - No hardware-managed cache
  - Requires manual data partitioning and timing of memory operations
- Limited compute and memory resources
- Huge parameter design spaces
  - Grows exponentially - even relatively small designs can have very large spaces
  - Parameters are interdependent and can change runtime by orders of magnitude
  - Manual exploration is tedious and usually suboptimal

Key Programming Challenges (Continued..)

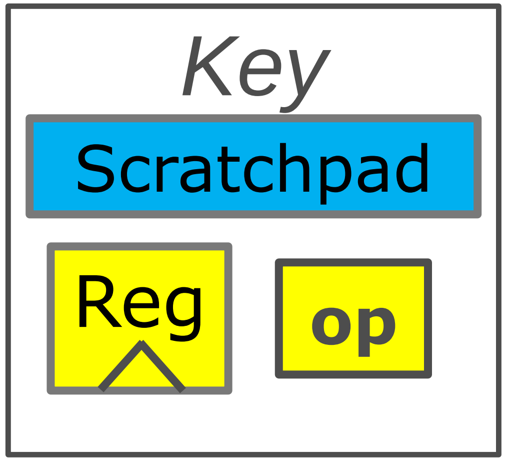
- Quick verification
  - Hardware synthesis is too slow to be part of the debug loop
  - Writing test benches is challenging and requires waveform debugging
- Managing host-accelerator interface
  - Tile transfers, register interfaces, data streams, etc.
- Incorporating IP cores into design

Design Space Example

Algorithm: Dot Product of Vectors A and B

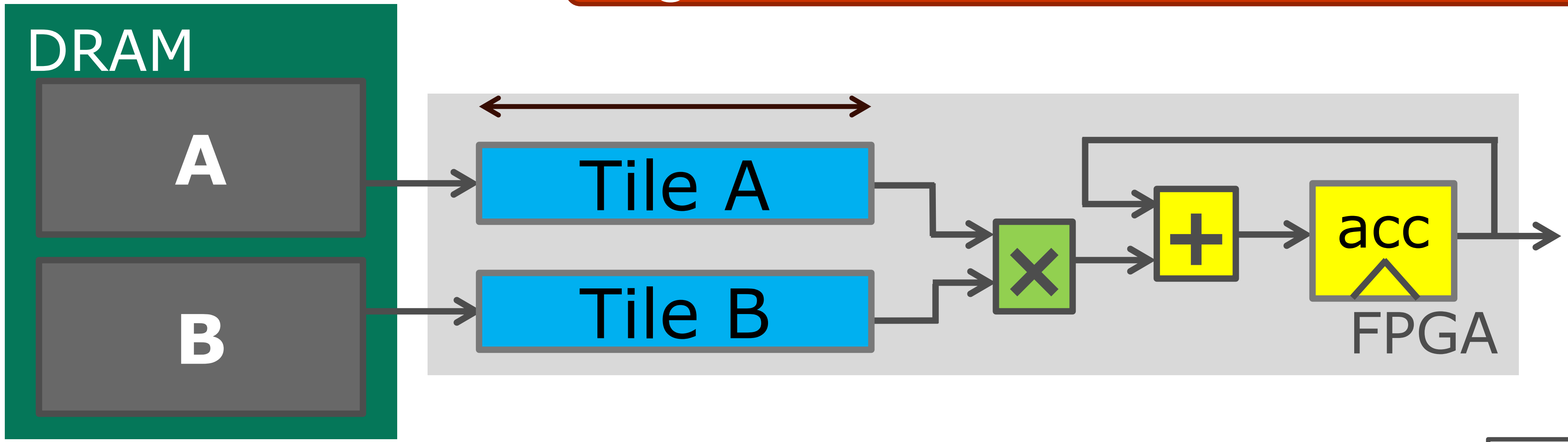


Small and simple, but slow!

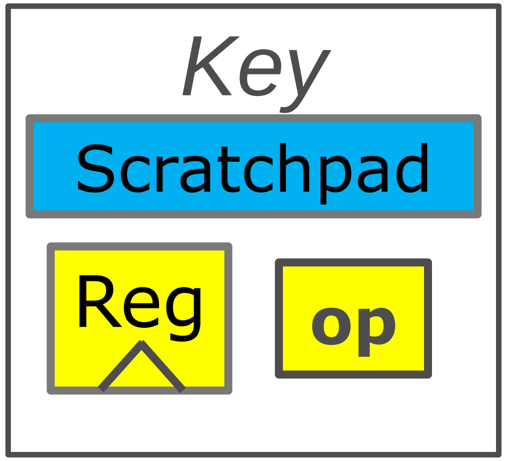


Important Parameters: Tile Sizes

Algorithm: Dot Product of Vectors A and B

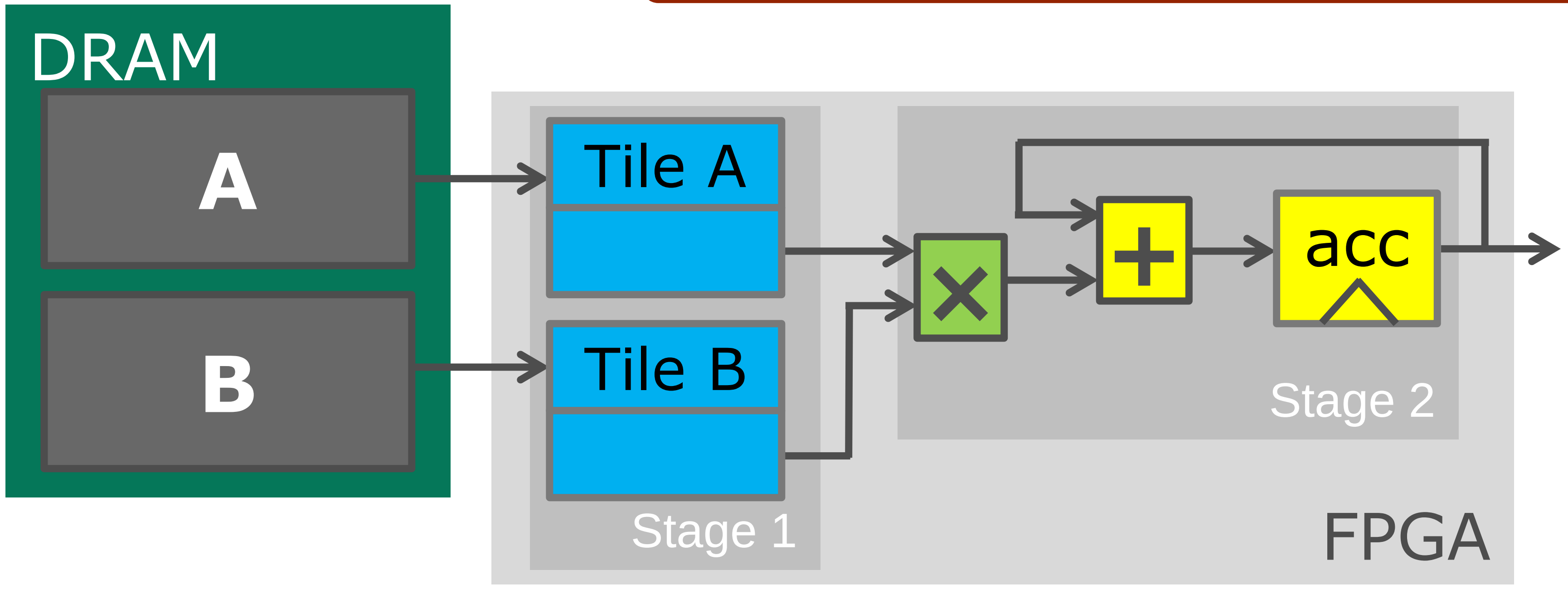


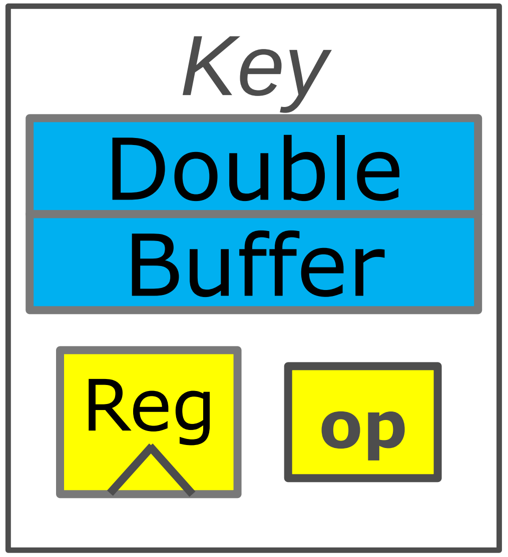
- Increases length of DRAM accesses ⬇ Runtime
- Increases exploited spatial locality ⬇ Runtime
- Increases local memory sizes ⬆ Area



Important Parameters: Pipelining

Algorithm: Dot Product of Vectors A and B

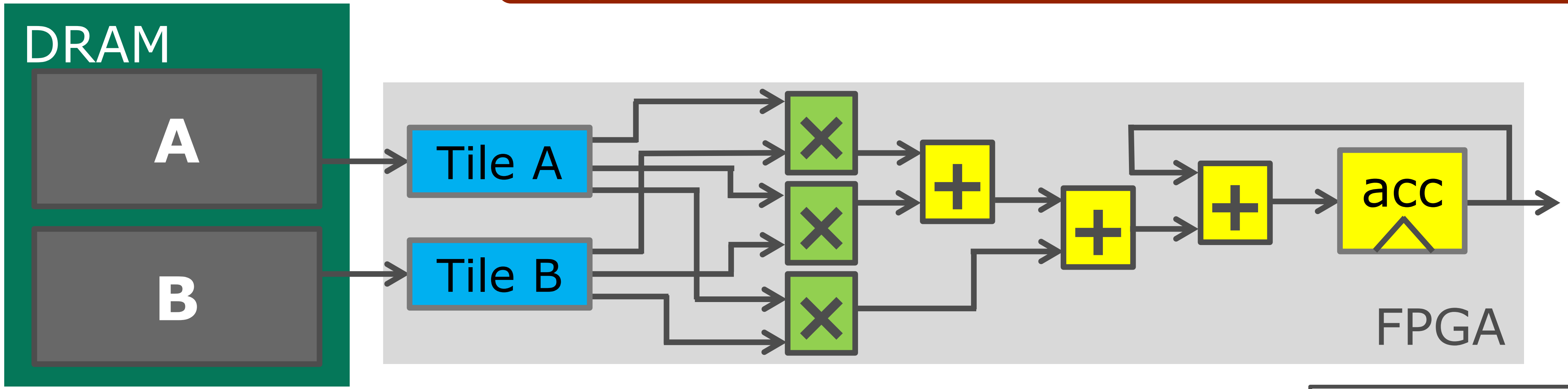




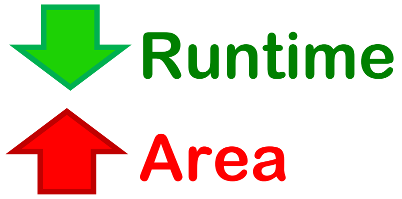
- Overlaps memory and compute ⬇ Runtime
- Increases local memory sizes ⬆ Area
- Adds synchronization logic ⬆ Area

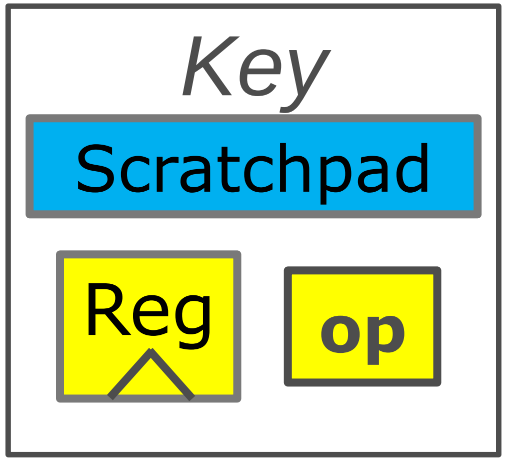
Important Parameters: Parallelization

Algorithm: Dot Product of Vectors A and B



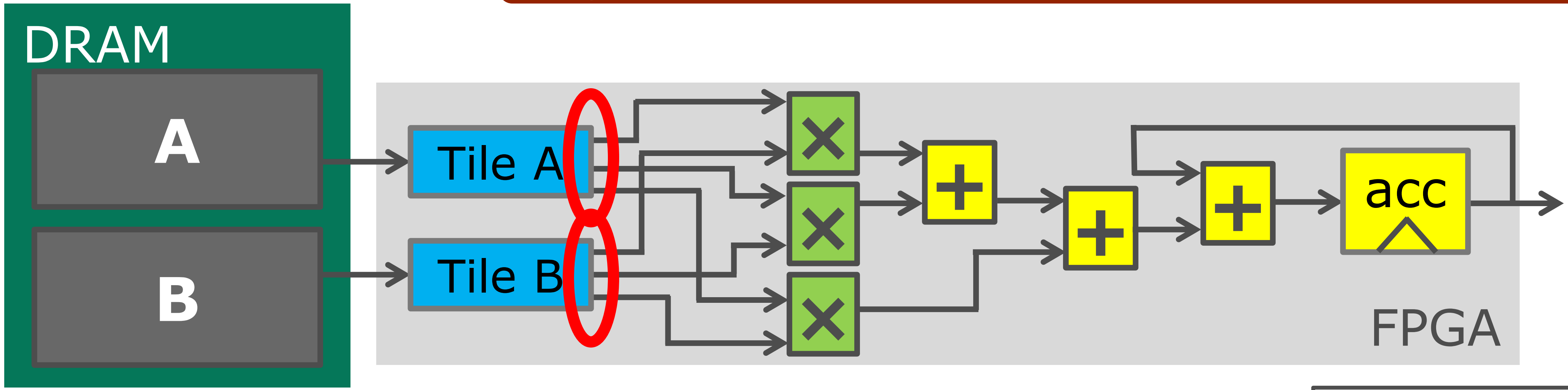
- Improves element throughput
- Duplicates compute resources



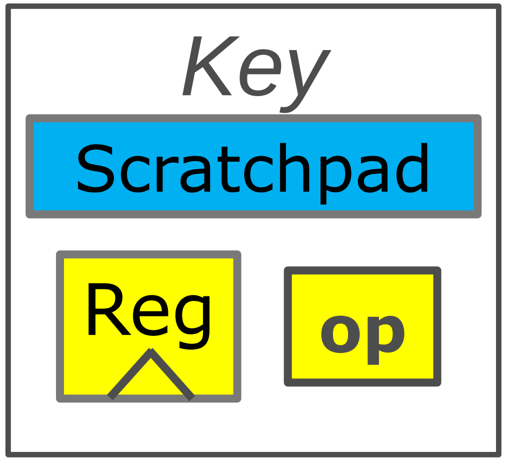


Important Parameters: Memory Banking

Algorithm: Dot Product of Vectors A and B

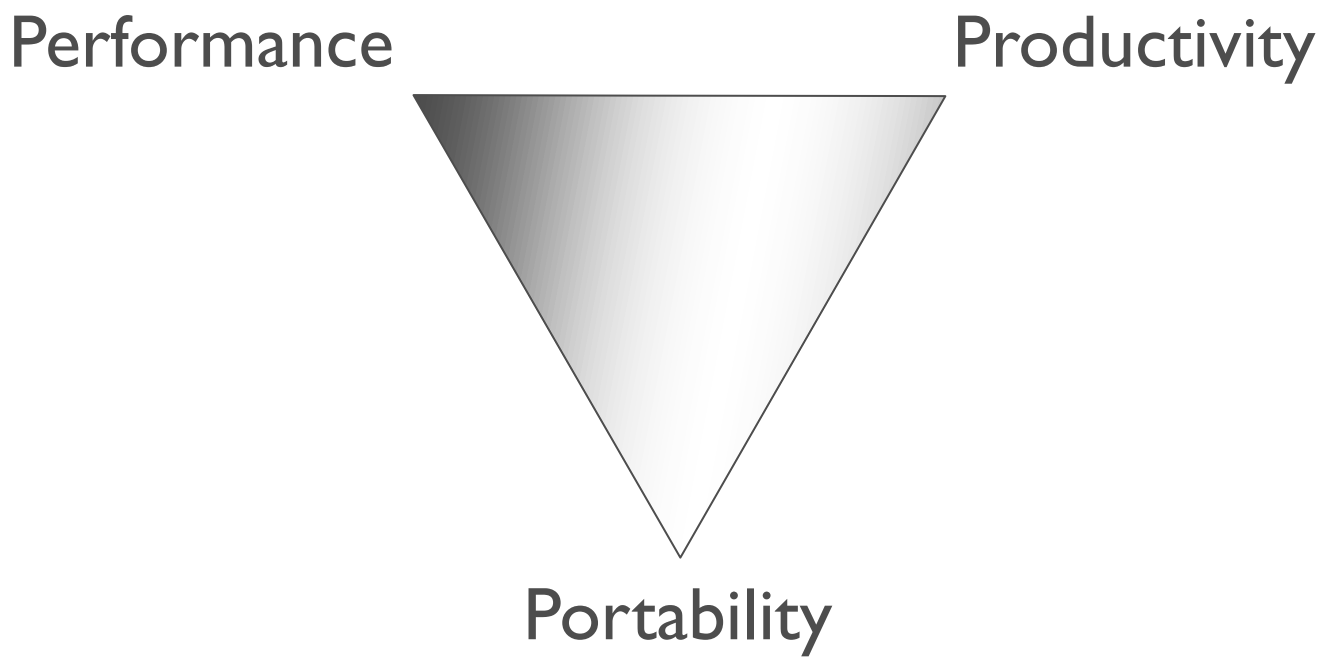


- Improves element throughput ⬇ Runtime
- May duplicate memory resources ⬆ Area



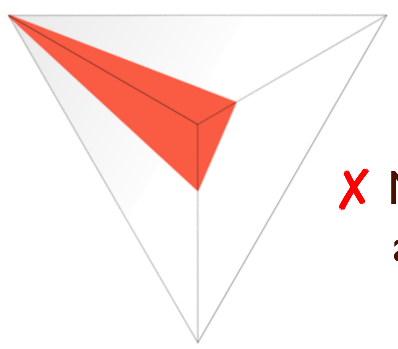
Language Requirements

1. Performs hardware optimizations for higher level frameworks
2. Productive language for “power” users (hardware programmers)
3. Produces efficient hardware



Language Comparisons

HDLs (Verilog, VHDL, Chisel, etc.)

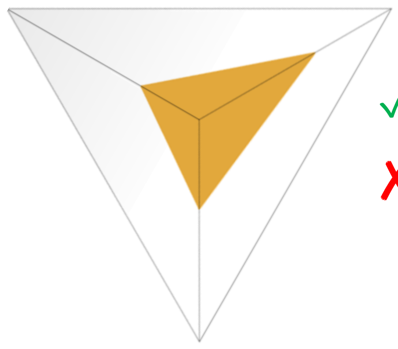


Performance
✓ Arbitrary RTL

Productivity
✗ No high-level abstractions

Portability

High Level Synthesis (C, OpenCL)

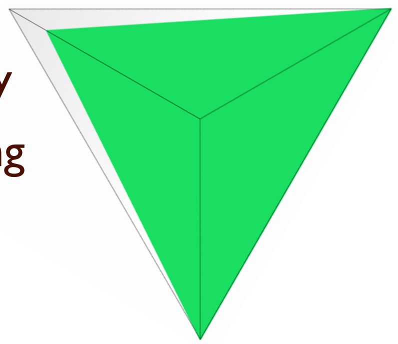


Performance
✗ No memory hierarchy
✗ No arbitrary pipelining

Productivity
✓ Nested loops
✗ Difficult to tune

Portability

Spatial

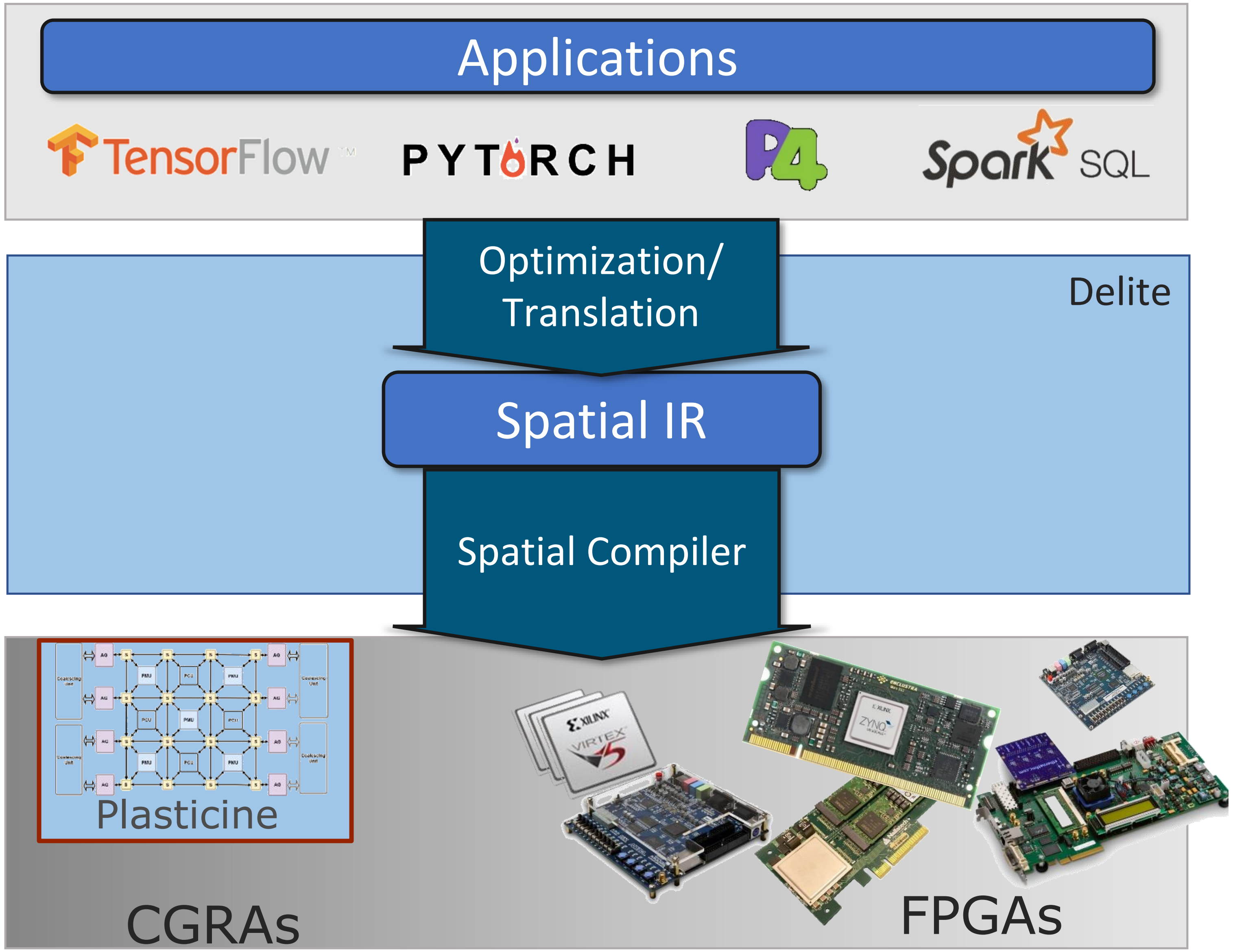


Performance
✓ Memory hierarchy
✓ Arbitrary pipelining
✗ Can't write arbitrary RTL

Productivity
✓ Nested loops
✓ Automatic memory banking/buffering
✓ Automated design tuning
✓ Simple host communication API

Portability
✓ Target-generic source

Spatial as an Intermediate Representation



The Spatial Language: Memory Abstractions

Typed storage templates

```
val accum  = Reg[Double]
val fifo   = FIFO[Float](D)
val lbuf   = LineBuffer[Int](R,C)
val pixels = ShiftReg[UInt8](R,C)
```

Explicit memory hierarchy

```
val buffer = SRAM[UInt8](C)
val image  = DRAM[UInt8](H,W)
```

Explicit transfers across memory hierarchy

Dense and sparse access

```
buffer load image(i, j::j+C)
out := x + y
buffer gather image(a, 10)
```

Streaming abstractions

```
val videoIn  = StreamIn[RGB]
val videoOut = StreamOut[RGB]
```

The Spatial Language: Control Abstractions

Blocking/non-blocking interaction with host

```
Accel { … }

Accel(*) { … }
```

Arbitrary state machine / loop nesting with implicit control signals

```
FSM[Int]{s => s != DONE }{
  case STATE0 =>
    Foreach(C by 1){j => … }
  case STATE1 => …
    Reduce(0)(C by 1){i => … }

}{s => nextState(s) }
```

The Spatial Language: Design Parameters

Spatial templates capture a variety of design parameters:

Implicit/Explicit parallelization factors

```
val P = 16 (1 → 32)
Reduce(0)(N by 1 par P){i =>
  data(i)
}{(a,b) => a + b}
```

Implicit/Explicit control schemes

```
Stream.Foreach(0 until N){i =>
  …
}
```

Explicit size parameters for stride and buffer sizes

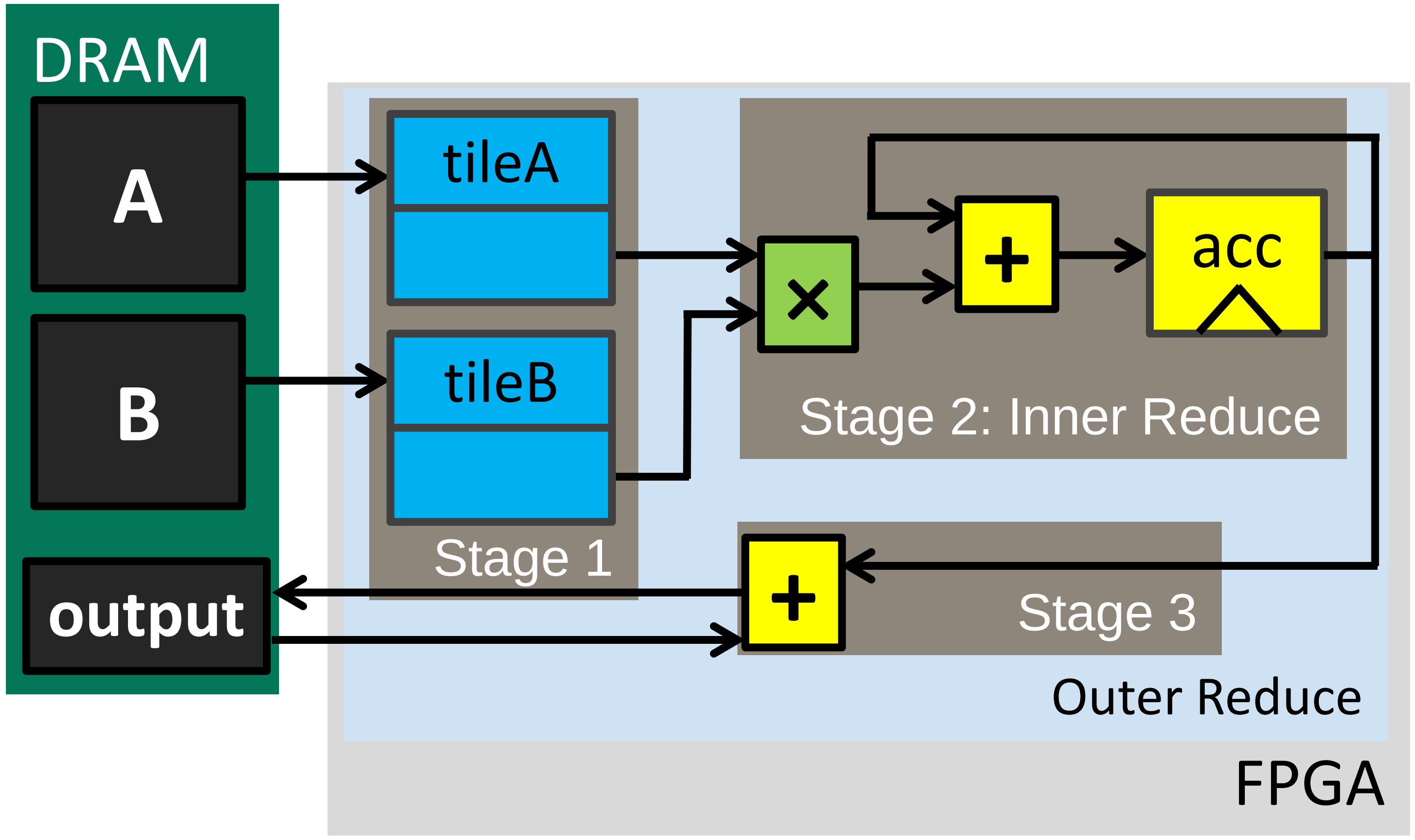
```
val B = 64 (64 → 1024)
val buffer = SRAM[Float](B)
Foreach(N by B){i =>
  …
}
```

Implicit memory banking and buffering schemes for parallelized access

```
Foreach(64 par 16){i =>
  buffer(i) // Parallel read
}
```

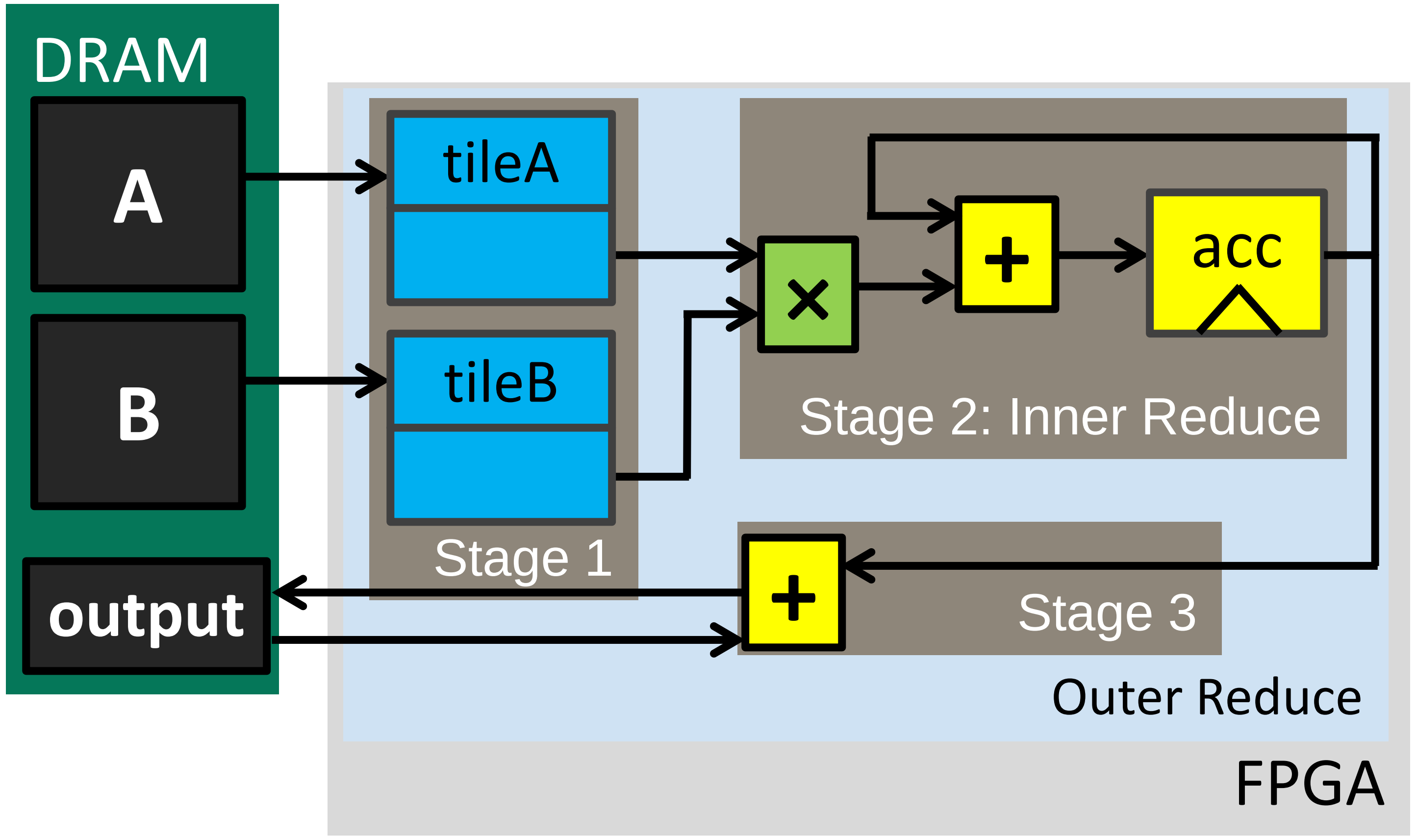
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



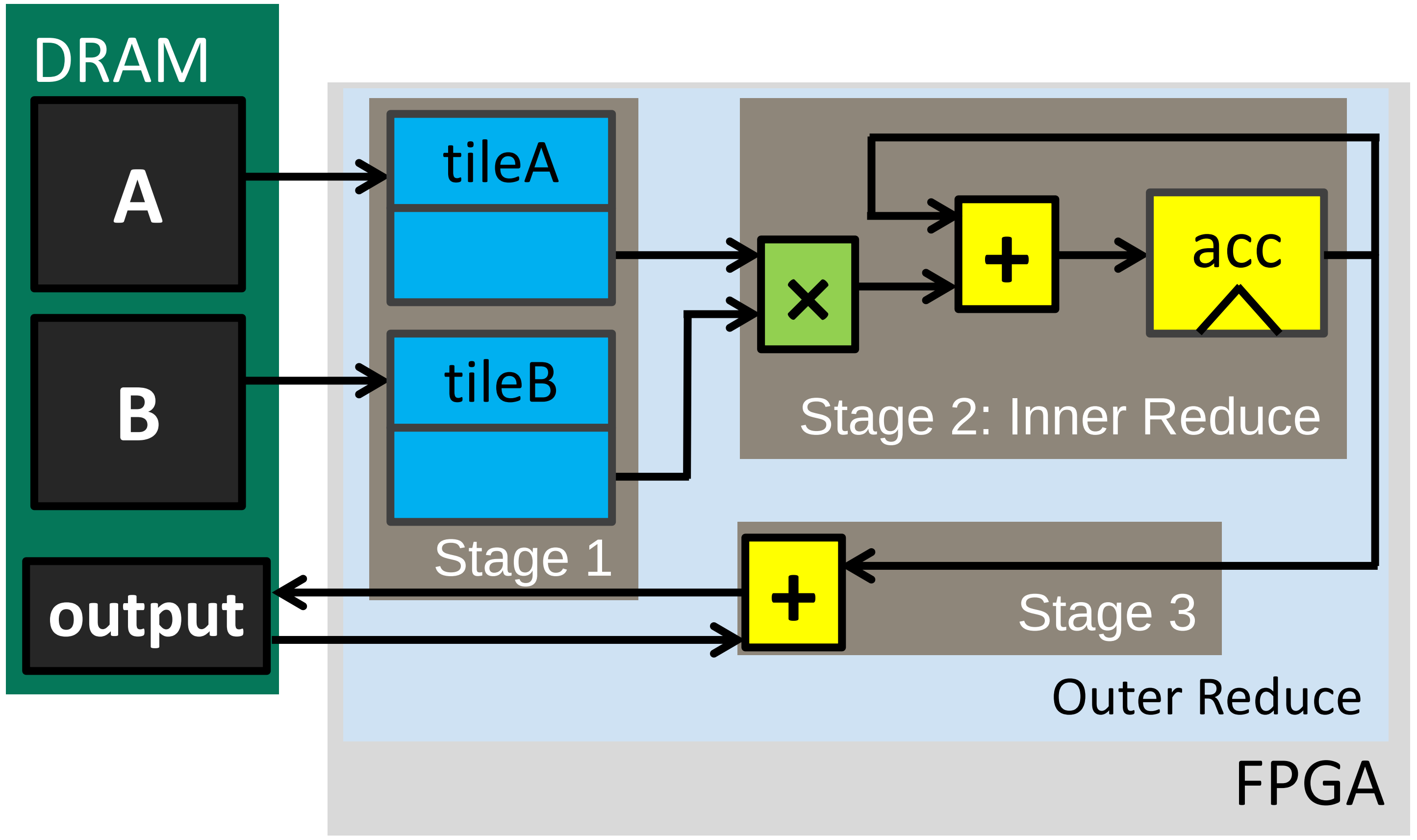
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



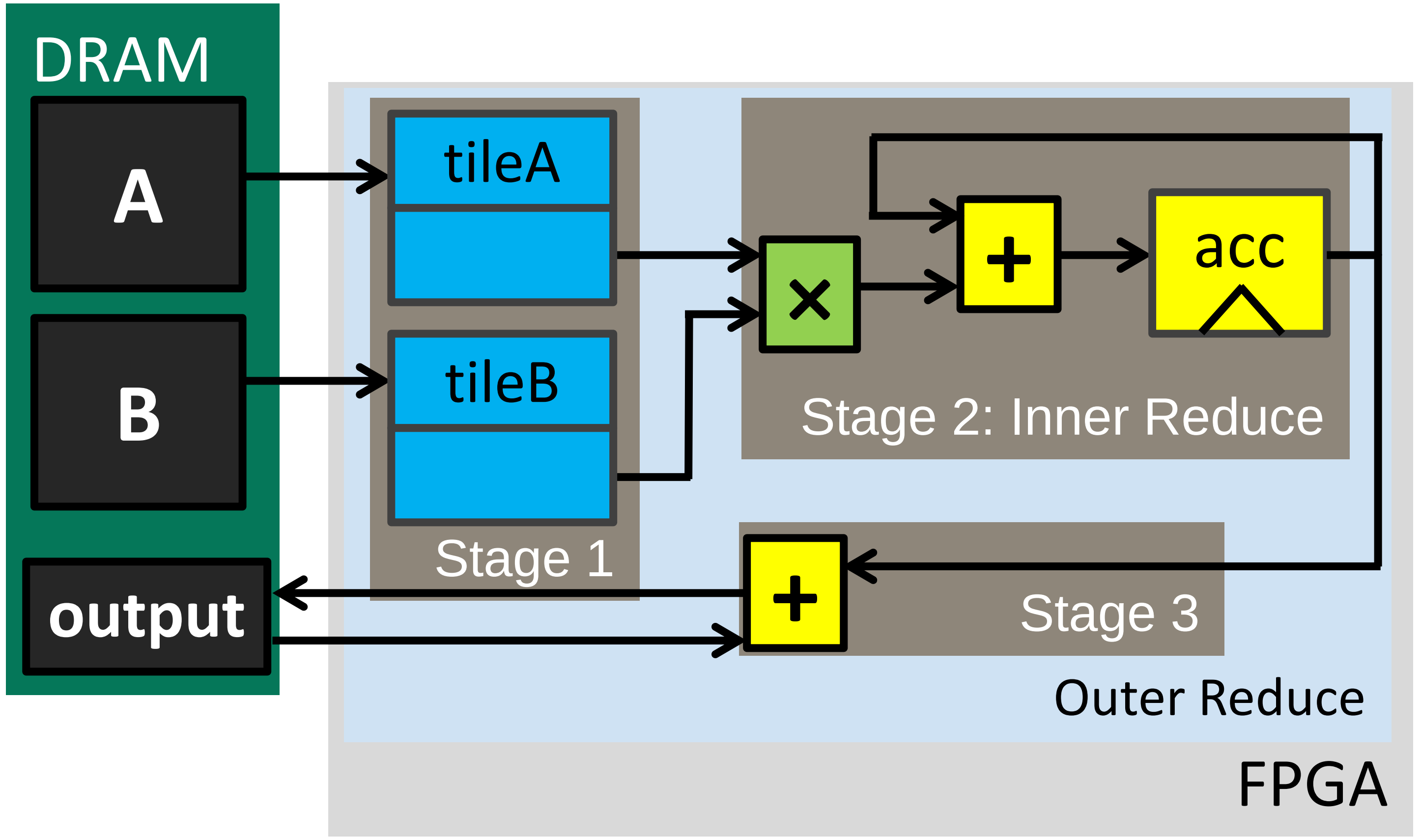
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



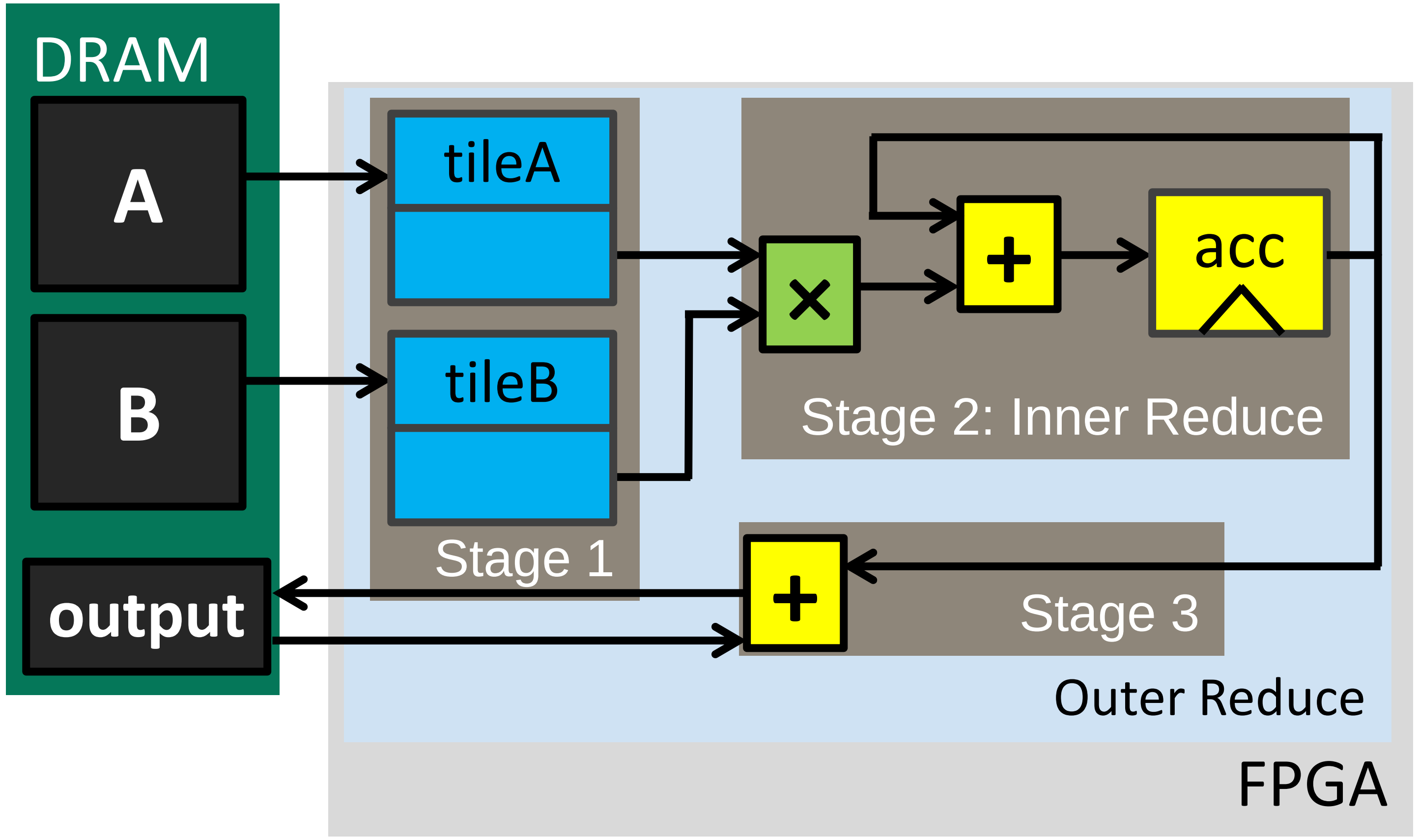
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



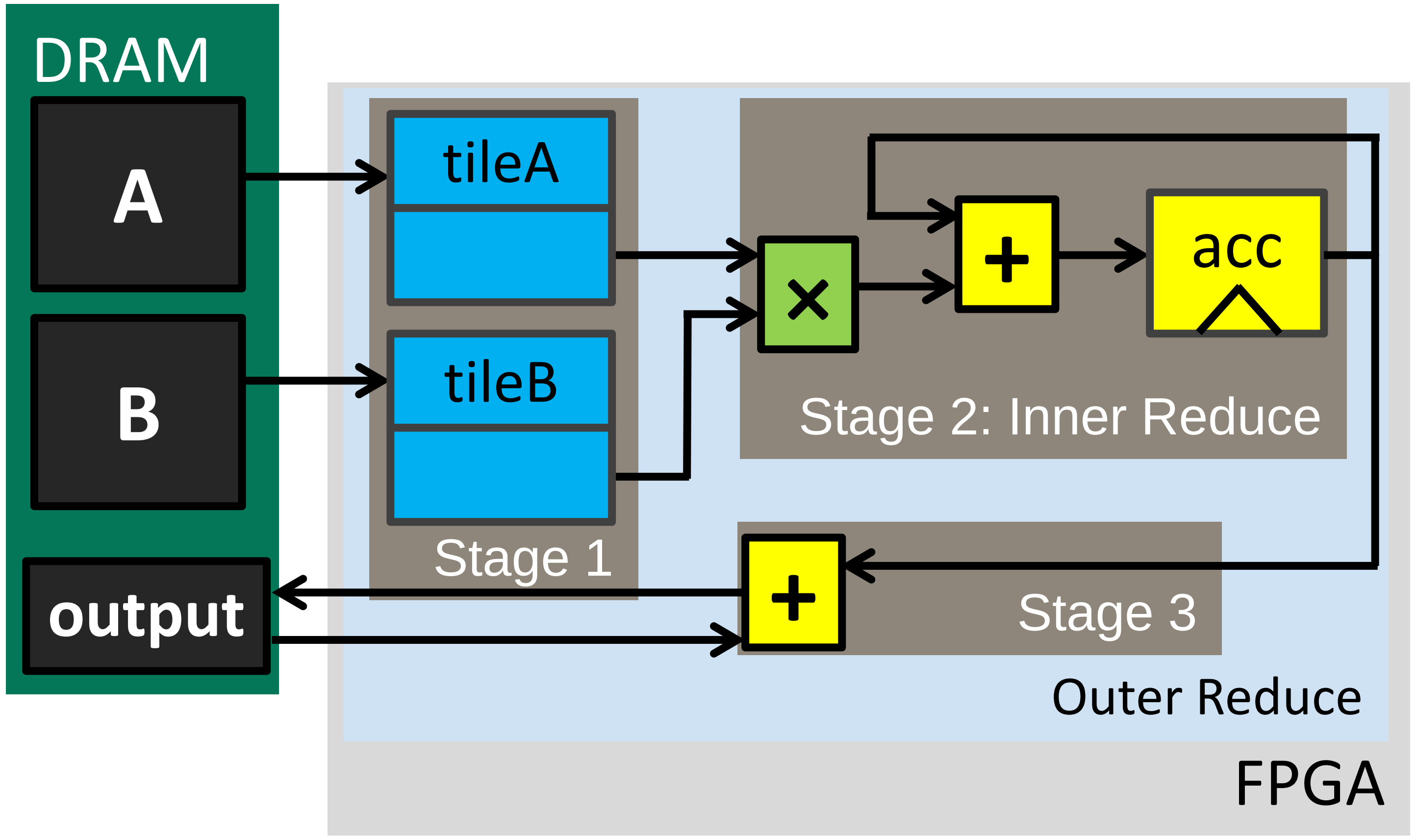
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



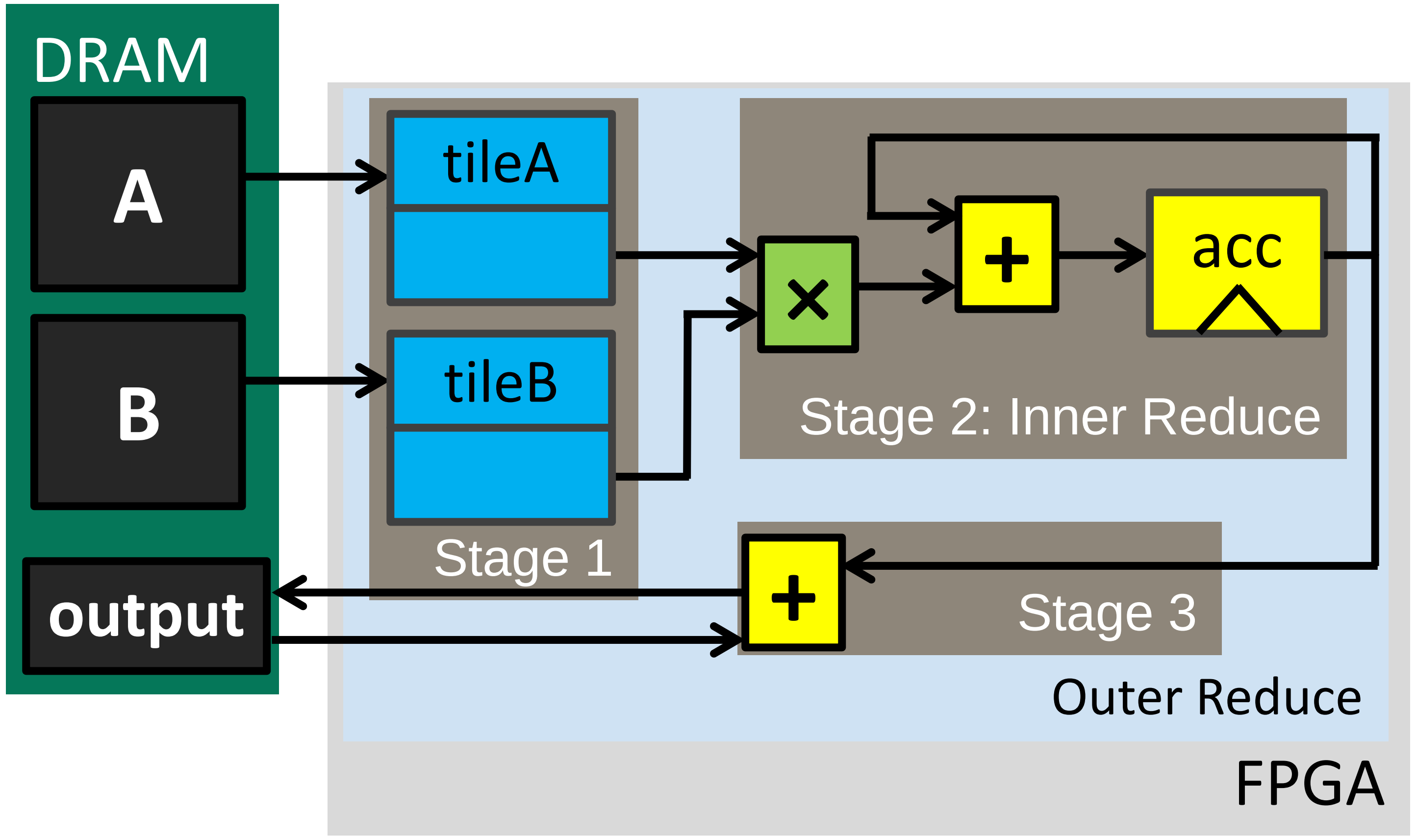
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



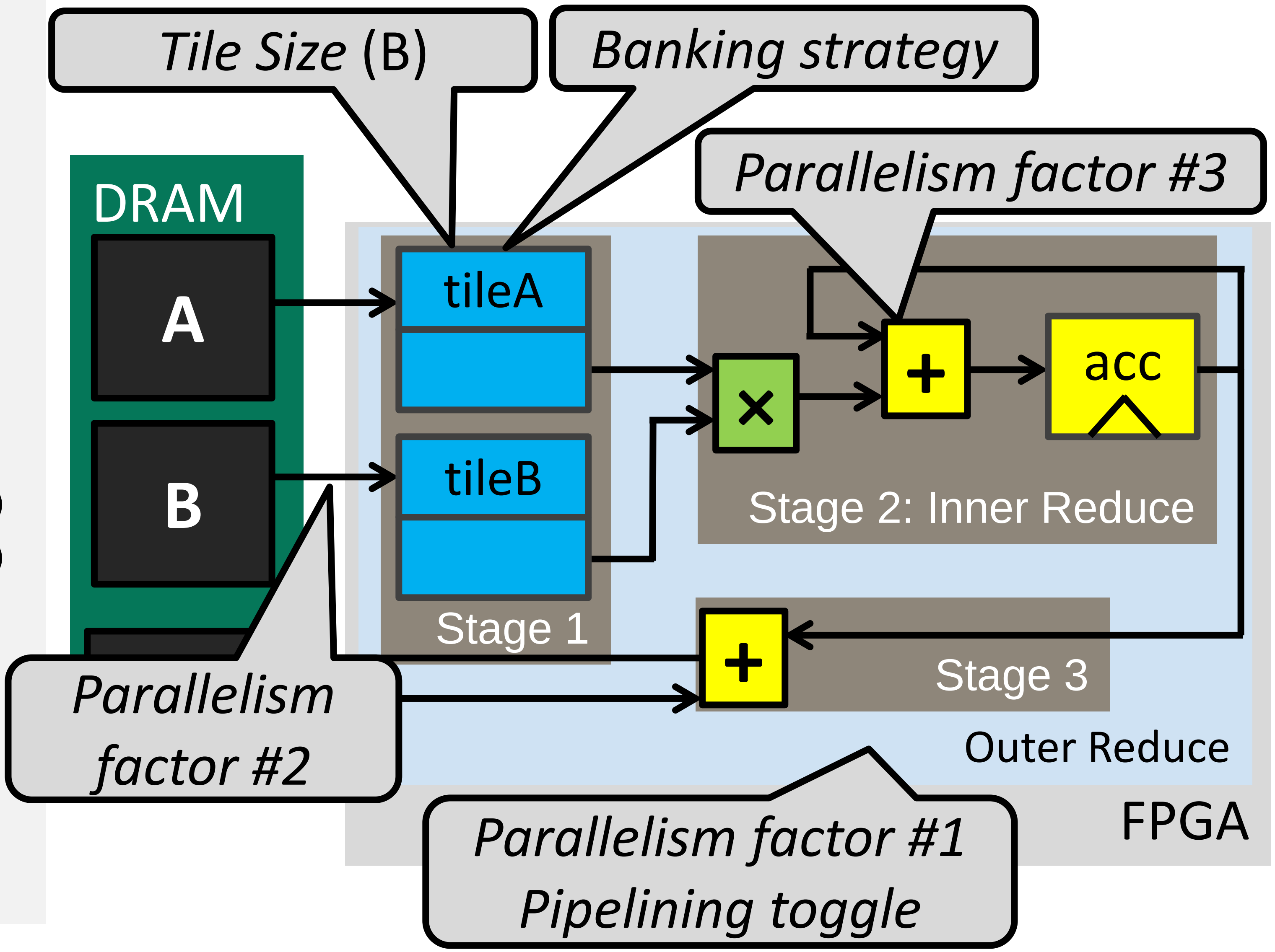
Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



Dot Product in Spatial

```
val output  = ArgOut[Float]
val vectorA = DRAM[Float](N)
val vectorB = DRAM[Float](N)

Accel {
   Reduce(output)(N by B){ i =>
      val tileA = SRAM[Float](B)
      val tileB = SRAM[Float](B)
      val acc   = Reg[Float]

      tileA load vectorA(i :: i+B)
      tileB load vectorB(i :: i+B)

      Reduce(acc)(B by 1){ j =>
         tileA(j) * tileB(j)
      }{a, b => a + b}
   }{a, b => a + b}
}
```



Design Tuning Performance

Spatial:

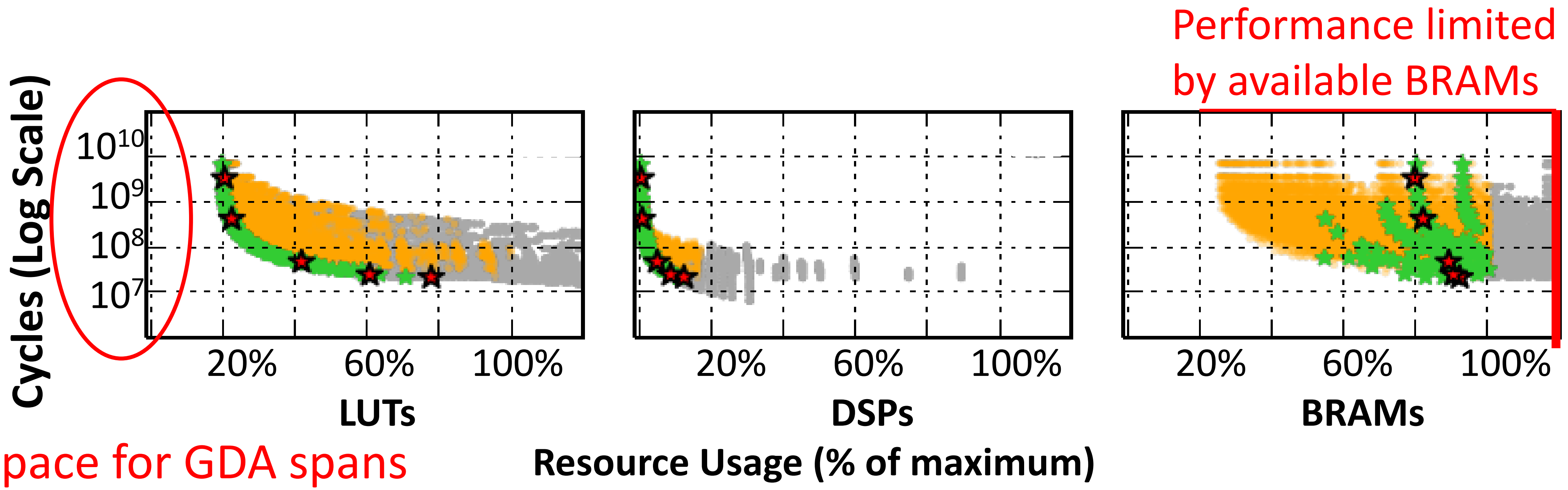
Benchmark	Designs	Search Time
Dot Product	5,426	**5.3 ms / design**
Outer Product	1,702	**30 ms / design**
TPCHQ6	5,426	**[illegible].2 ms / design**
Blackscholes	[illegible]	**[illegible]7 ms / design**
Matrix Multiply	[illegible]	**[illegible]1 ms / design**
K-Means	75,200	**20 ms / design**
GDA	42,800	**17 ms / design**

6500x Speedup Over HLS!

Vivado HLS:

	Designs	Search Time
GDA	250	**1.85 min / design**

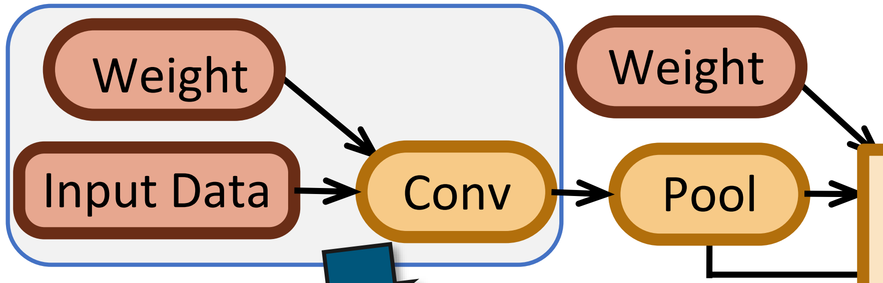
GDA Design Space



Space for GDA spans
four orders of magnitude

● Valid design point	★ Pareto-optimal (ALMs/cycles) design
● Invalid design point	★ Synthesized pareto design point

Preview: TensorFlow to Spatial



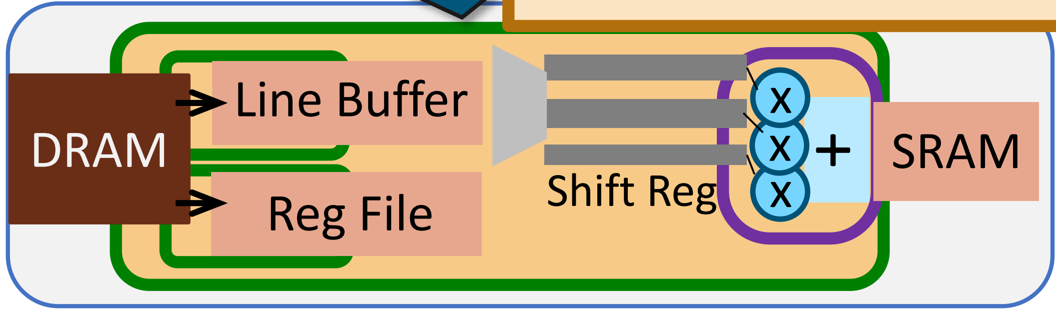
Dataflow graph of domain-specific operators

Each node in the DFG has a corresponding optimized Spatial template

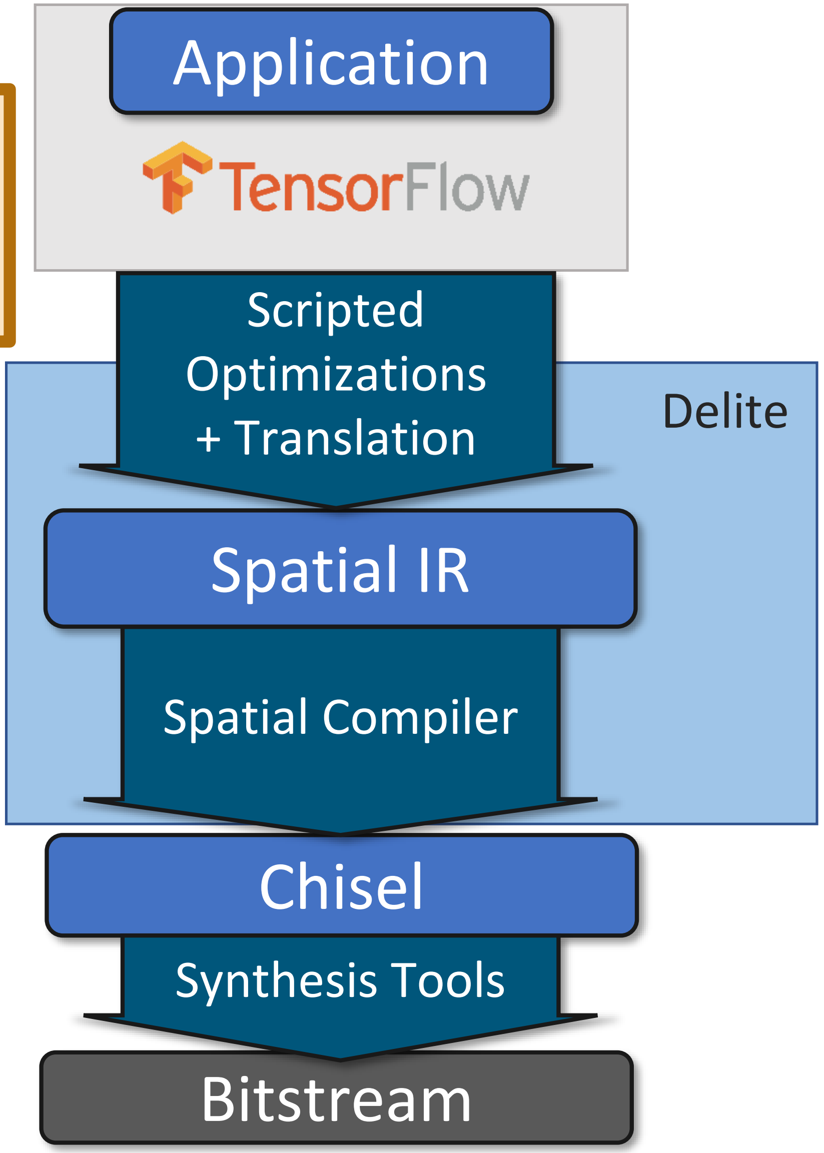
```
val data = LineBuffer[T](K, in_c)
val row = RegFile[T](K, K)
val weight_RF = RegFile[T](K, K)
weight_RF load w_DRAM(channel, 0::K, 0::K)
Foreach(0 until in_r){ r =>
 data load i0_DRAM(channel, r, 0::in_c)
 Foreach(0 until in_c){ c =>
  Foreach(0 until K){
   i => row(i, *) <<= da
  }
  val window = Reduce(0
   (i,j) => row(i,j) * w
  }{_+_}…
```

Hierarchical dataflow graph of **tiled pipelines** with memory hierarchy

Cross-node communication uses SRAM and registers to minimize DRAM bandwidth



Synthesizable hardware design



Preview: TensorFlow to Spatial

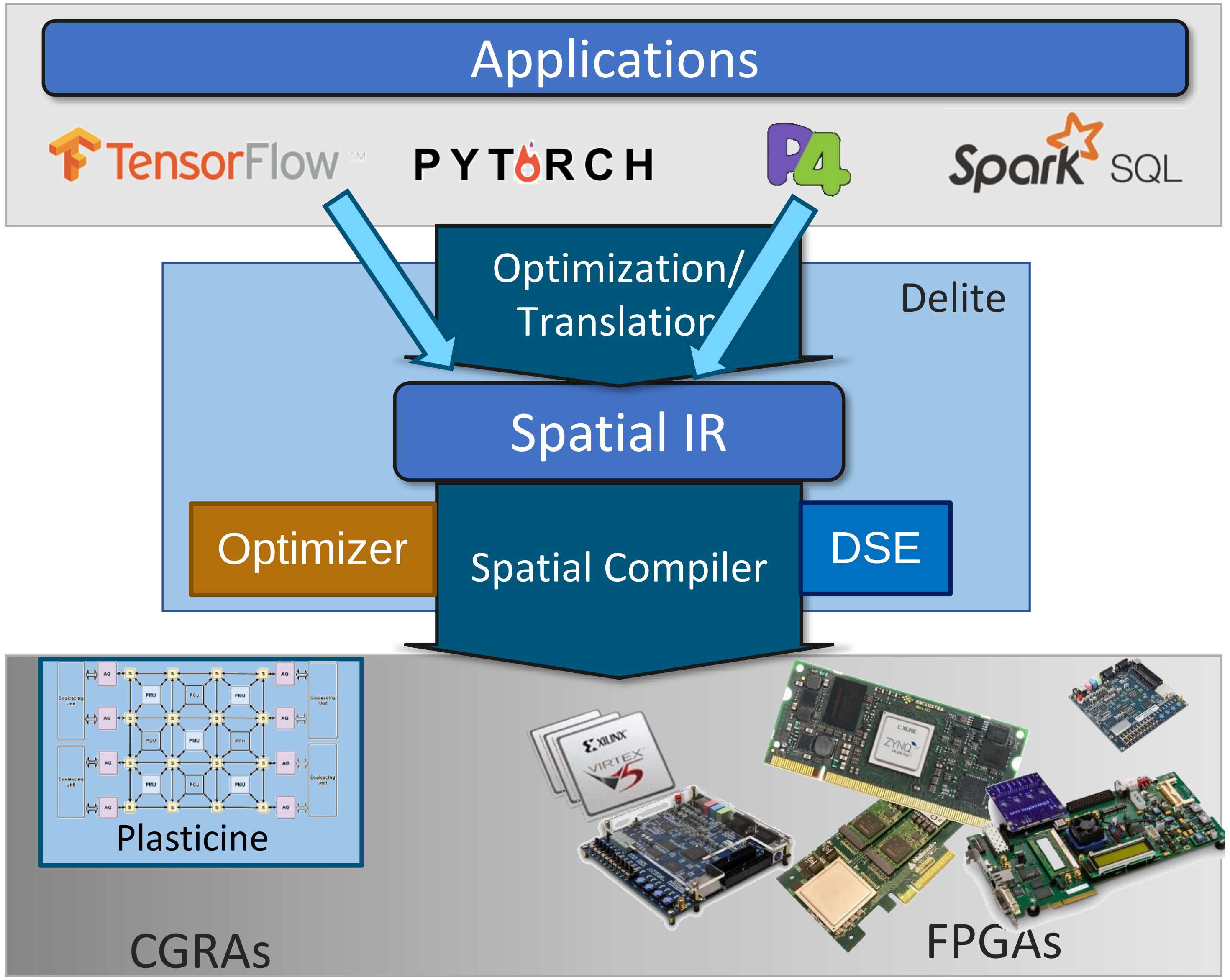
- Benefits
  - Makes Spatial easier to program
  - Automatically benefit from TensorFlow optimizations
    (e.g. constant folding, fusion, quantization for low precision)
- Current support:
  - Convolutional Neural Networks
  - Recurrent Neural Networks (GRU, LSTM)
  - General matrix algebra computations
- Same DFG mapping can be applied to other front-ends
  - Not specific to TensorFlow

Spatial's Advantages

- Automatic Design-Space Exploration
  - Parallelization selection to balance pipelines
  - Tile size selection to balance on-chip memory, bandwidth
- Automatic memory banking
- Agnostic to hardware vendor, Cloud/Embedded, FPGA/CGRA
- Preliminary results: within 20% of DNNWeaver (DNN hardware library)
  - For inferences/s of a small network (LeNet) on the Zynq ZC702
  - Ongoing work to close 20% gap by further optimizing templates

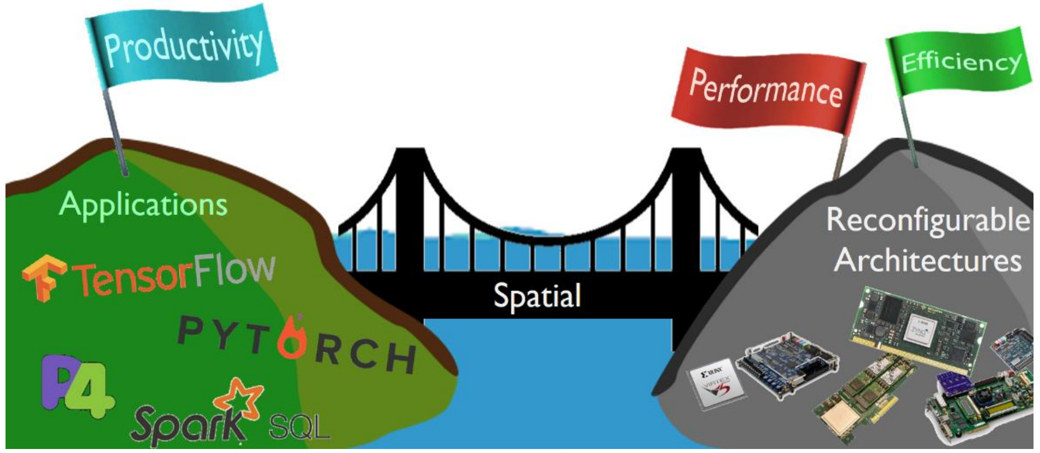
Future Work

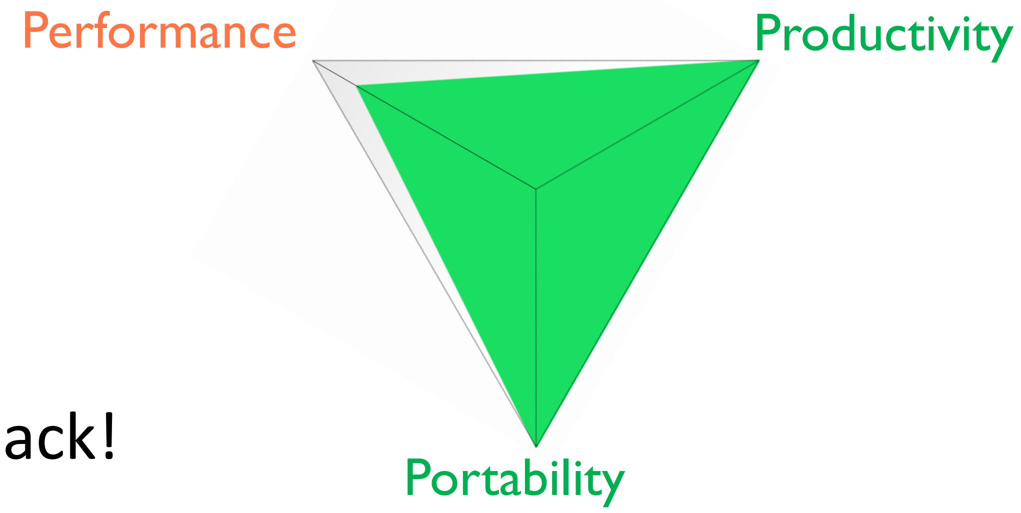
- Low precision DNN training
- Residual Networks
- Unify and support more frontends above Spatial
- Expand set of optimizations done by Spatial compiler
- Expand backends supported by Spatial



Conclusion

- **Orders of magnitude improvements** in performance and energy efficiency are possible with reconfigurable architectures
- Spatial's **intermediate representation** can be used as a bridge to reconfigurable architectures
- The Spatial **language** includes abstractions for **performance, productivity, and portability**
- Spatial's design tuning is **~6500x** faster than that possible with Vivado HLS
- Preliminary results **within 20% of manual HDL** on CNN implementations





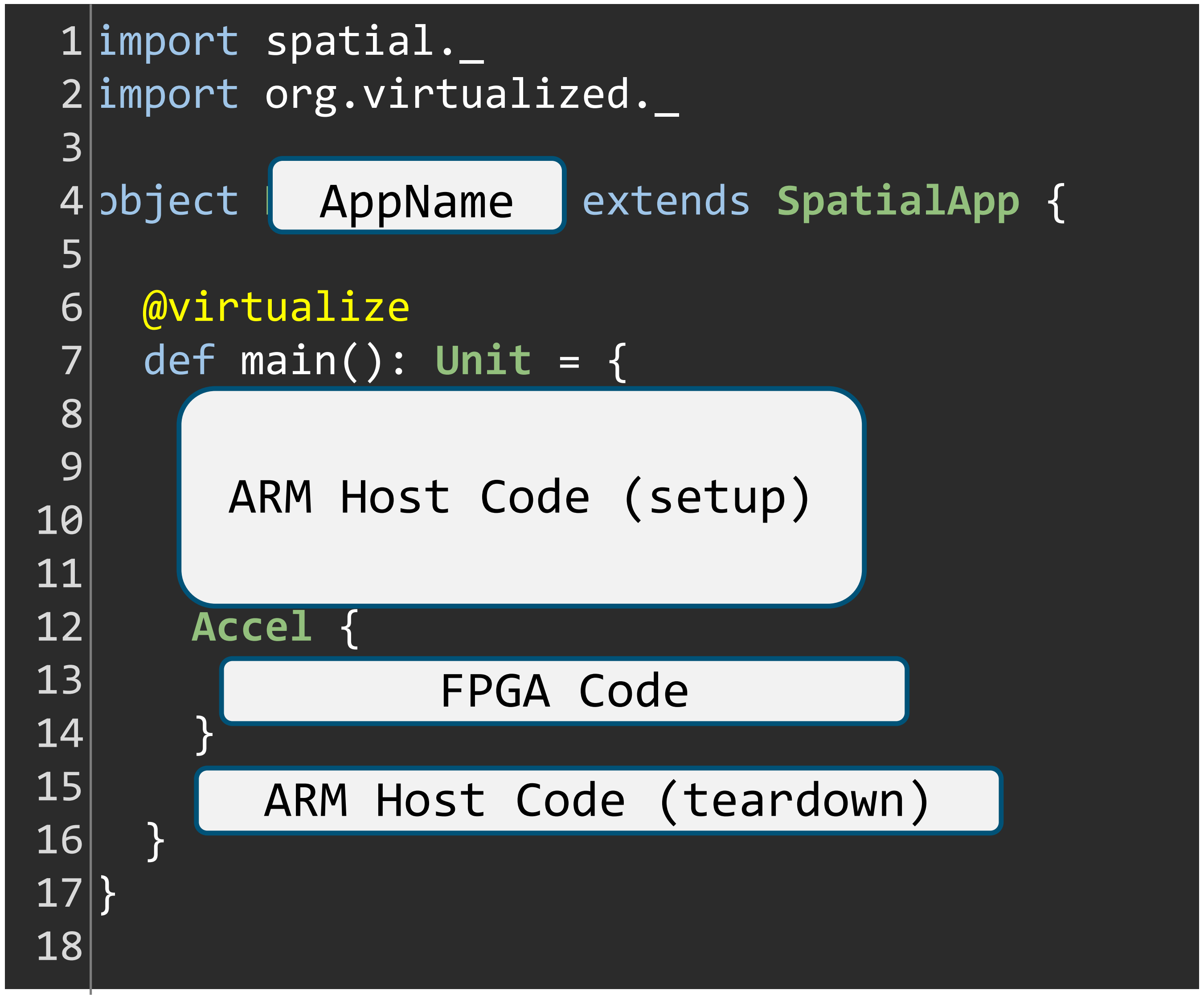
We're always looking for new users and feedback!

Open source at: spatial.stanford.edu

Nuts and Bolts of a Full Spatial Application

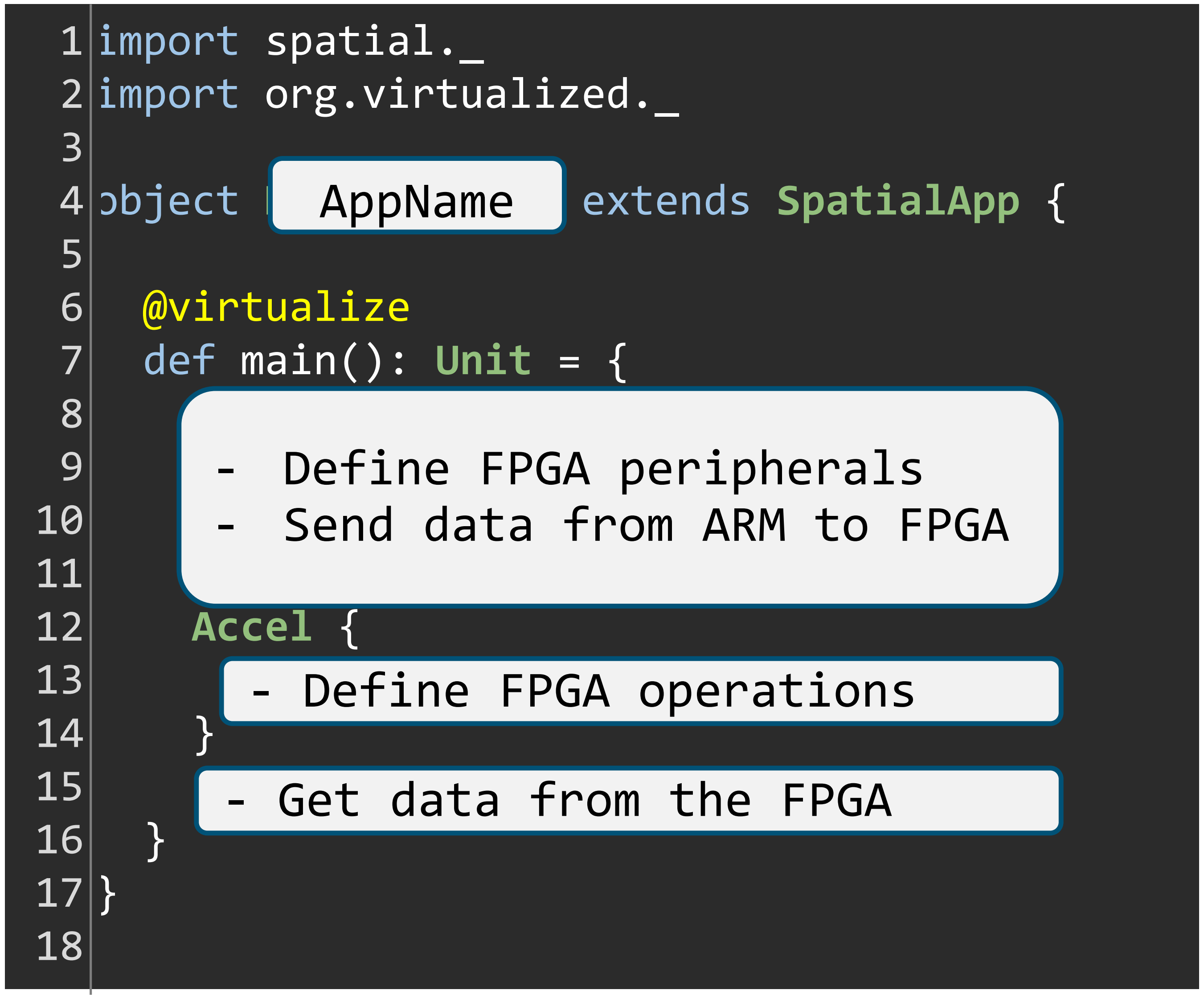
- Continue to see the bird's eye view of how to write a full application
- Visit the website tutorial for more detailed examples:
  - http://spatial-lang.readthedocs.io/en/latest/tutorial/starting.html

Spatial App Template



```
import spatial._
import org.virtualized._

object  AppName  extends SpatialApp {

  @virtualize
  def main(): Unit = {

      ARM Host Code (setup)

      Accel {
        FPGA Code
      }
      ARM Host Code (teardown)
  }
}
```

Spatial App Template



```
import spatial._
import org.virtualized._

object  AppName  extends SpatialApp {

  @virtualize
  def main(): Unit = {

      -  Define FPGA peripherals
      -  Send data from ARM to FPGA

     Accel {
       - Define FPGA operations
     }
     - Get data from the FPGA
  }
}

```

Hello Spatial!

```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}

```

Spatial is Embedded in **Scala**

```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}

```

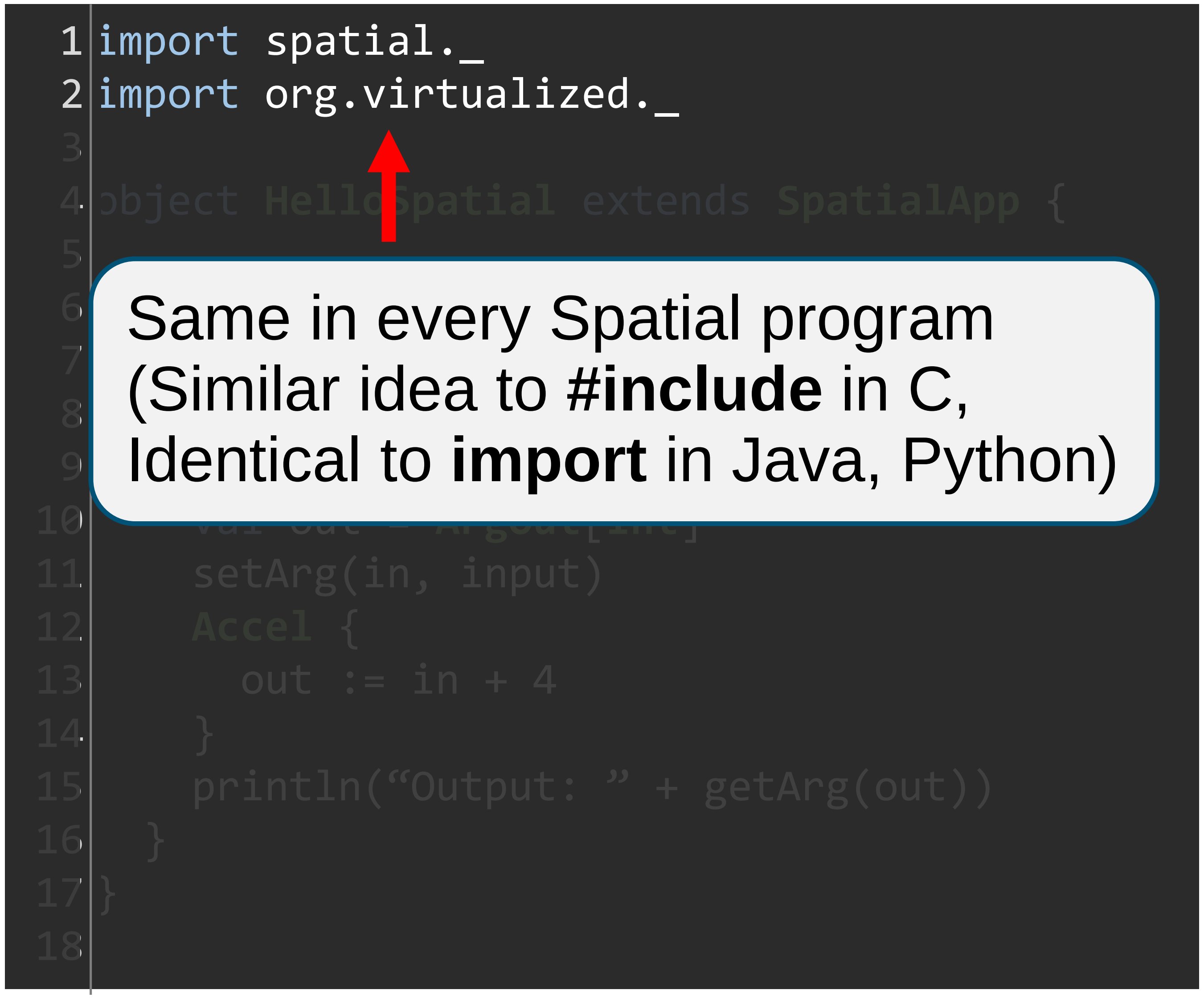
Spatial can be thought of as a **Scala** library

Spatial is Embedded in **Scala**

```scala
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}

```

Semicolons are optional

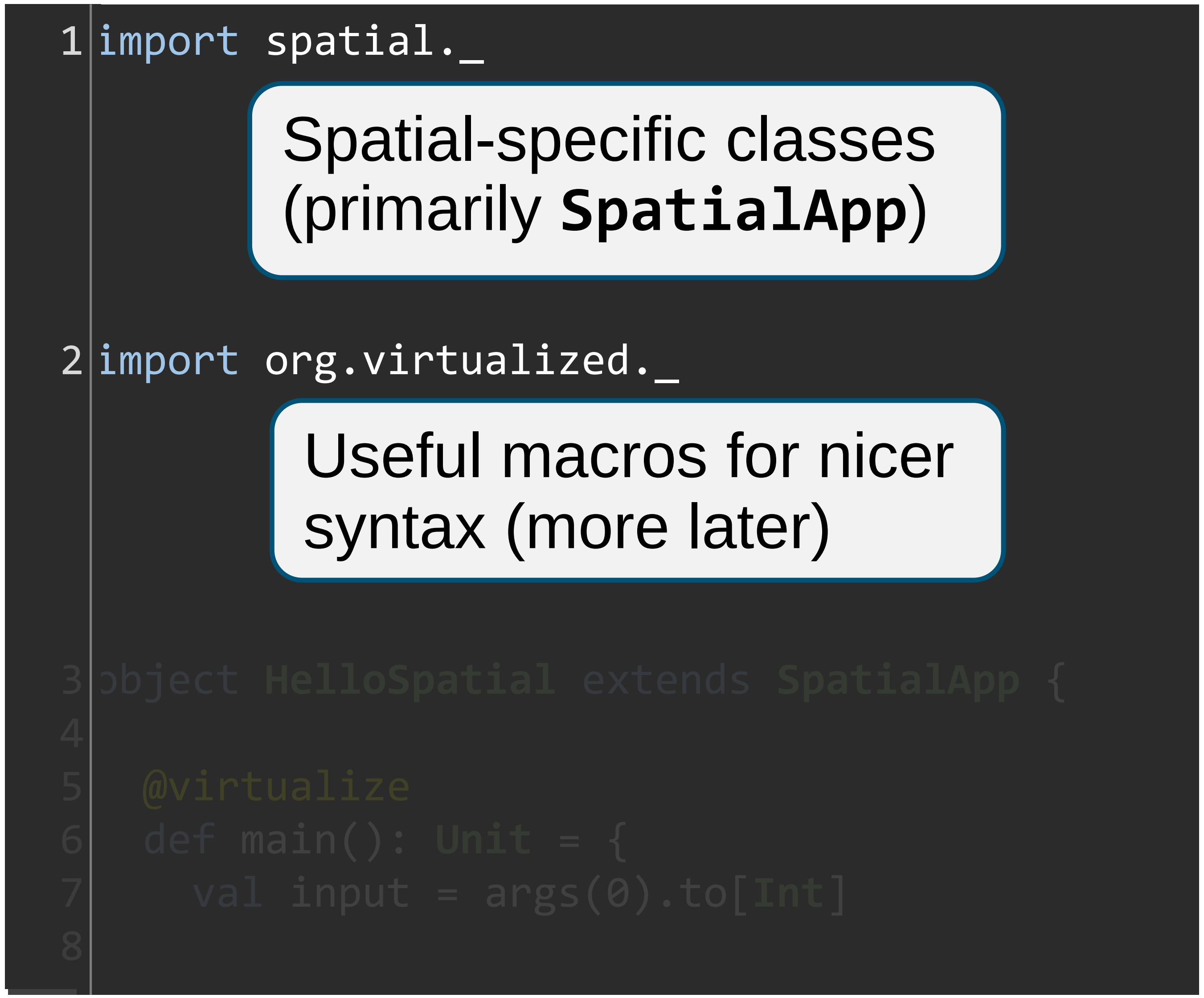
Import Statements



```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

Same in every Spatial program
(Similar idea to **#include** in C,
Identical to **import** in Java, Python)

Import Statements

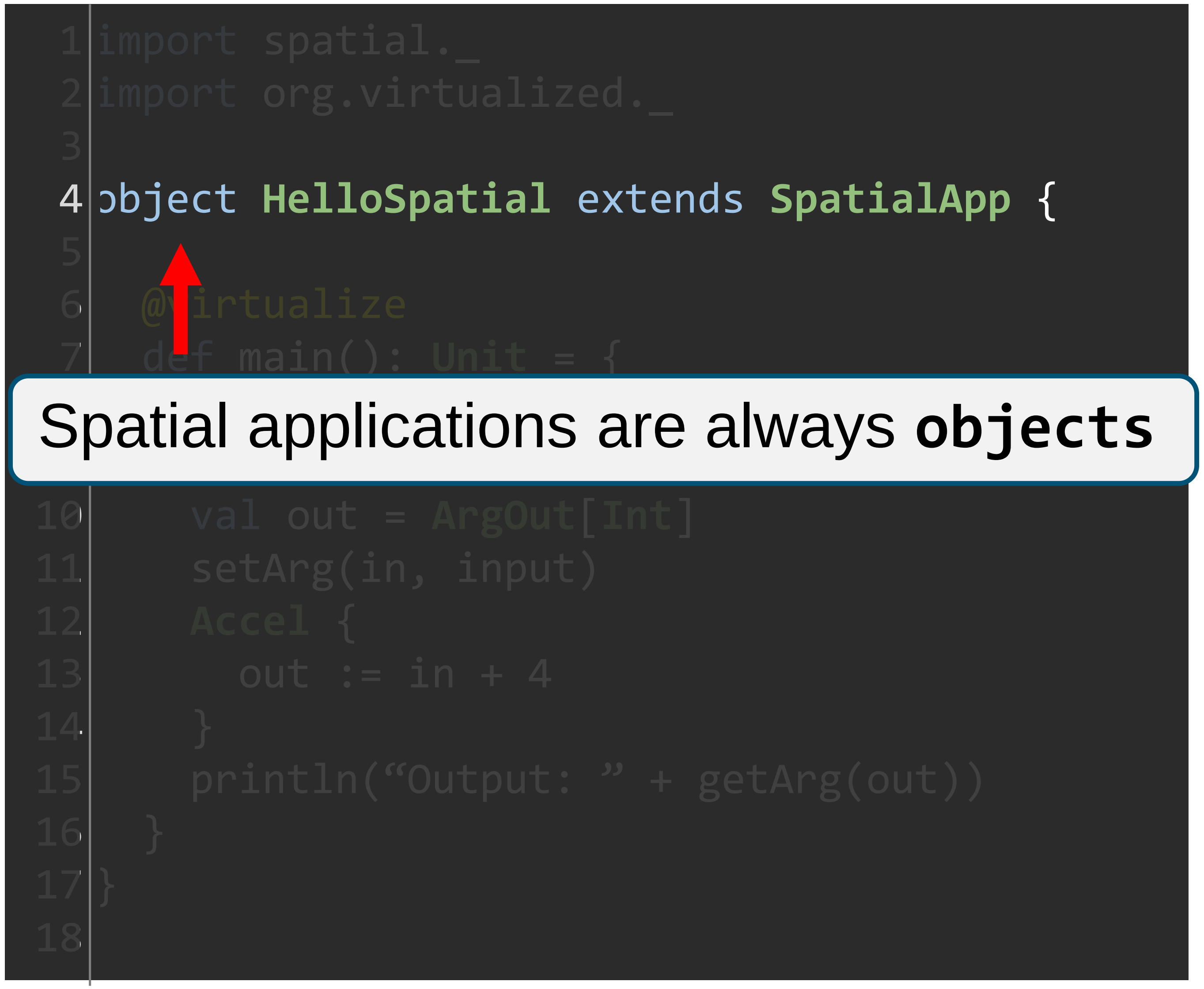


```
import spatial._
import org.virtualized._
object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]

```

Spatial-specific classes (primarily **`SpatialApp`**)

Useful macros for nicer syntax (more later)

Application Object Declaration

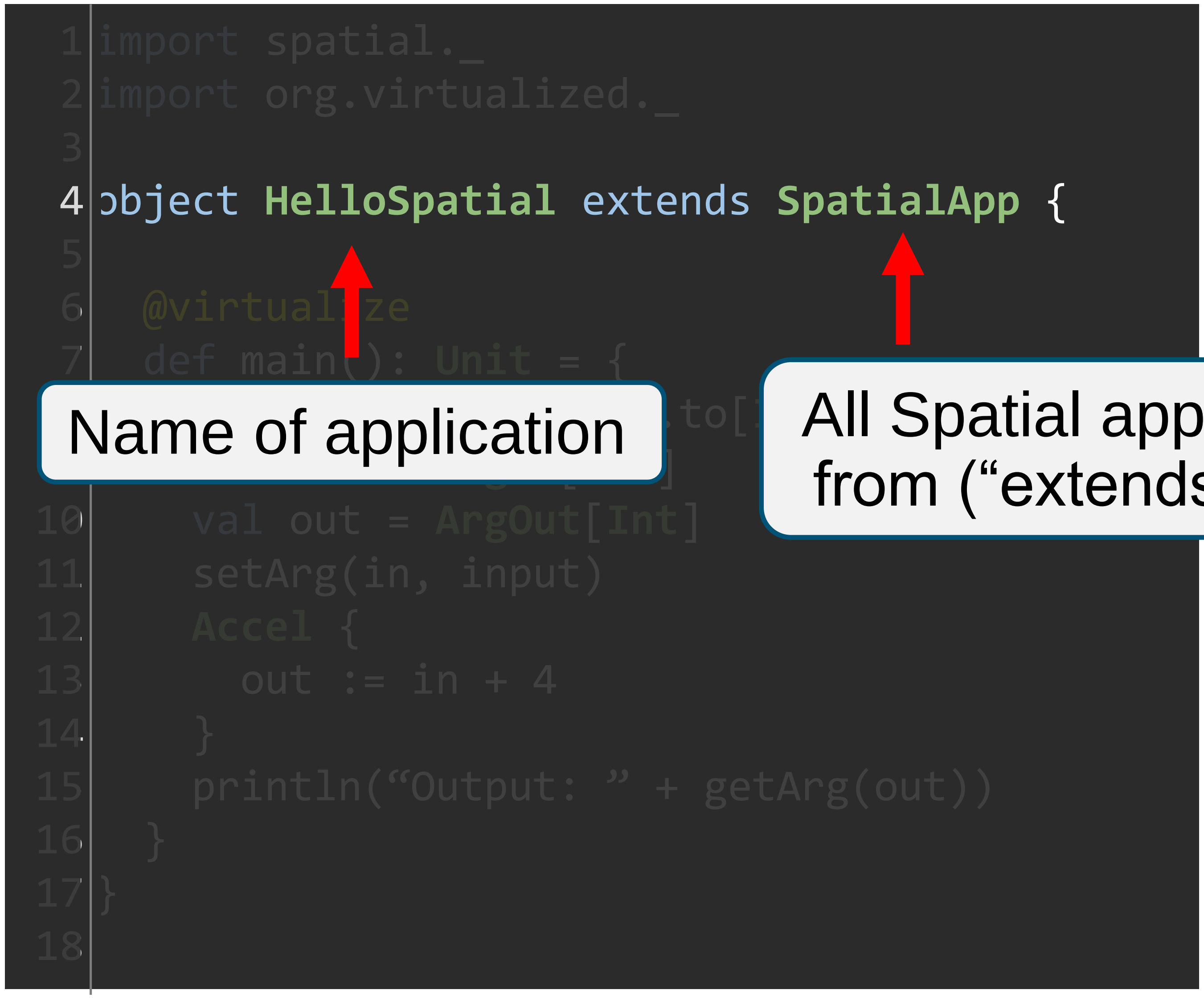


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
```

Spatial applications are always **`objects`**

```
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

Application Object Declaration

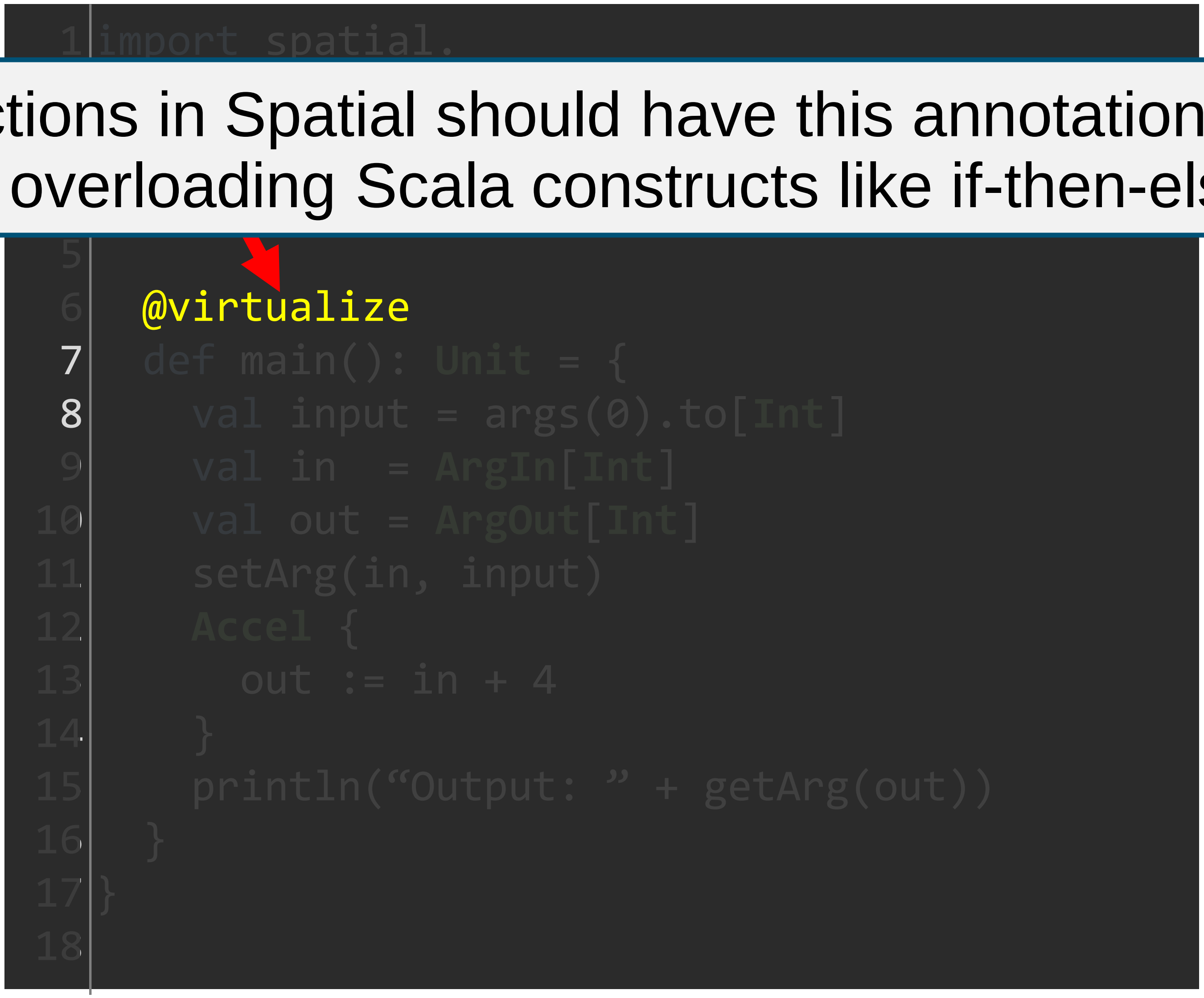


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
                         .to[
                           ]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}

```

Name of application

All Spatial applications inherit from ("extends") **SpatialApp**

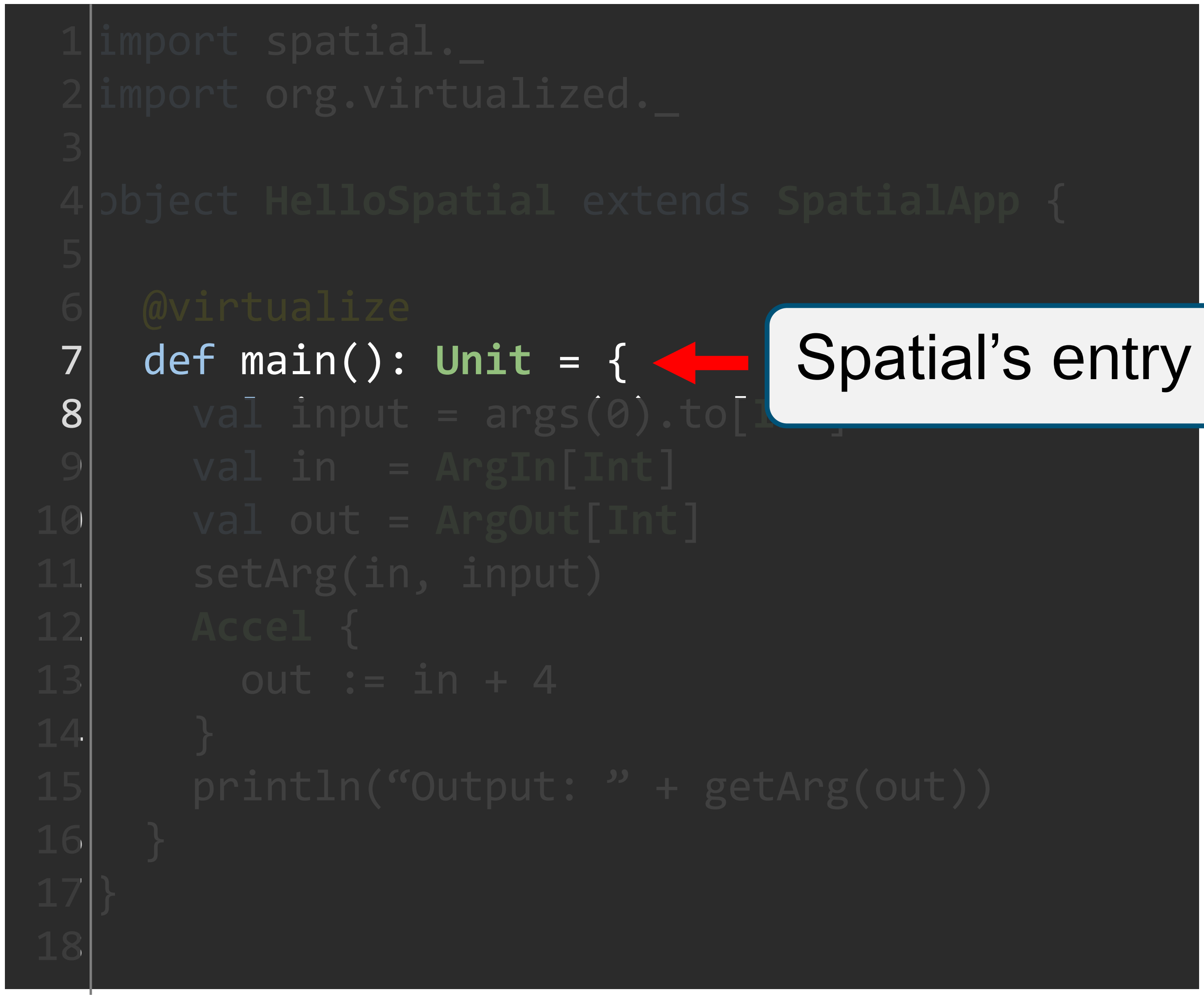
“@virtualize” Annotation



```
import spatial.

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

All functions in Spatial should have this annotation
(Allows overloading Scala constructs like if-then-else)

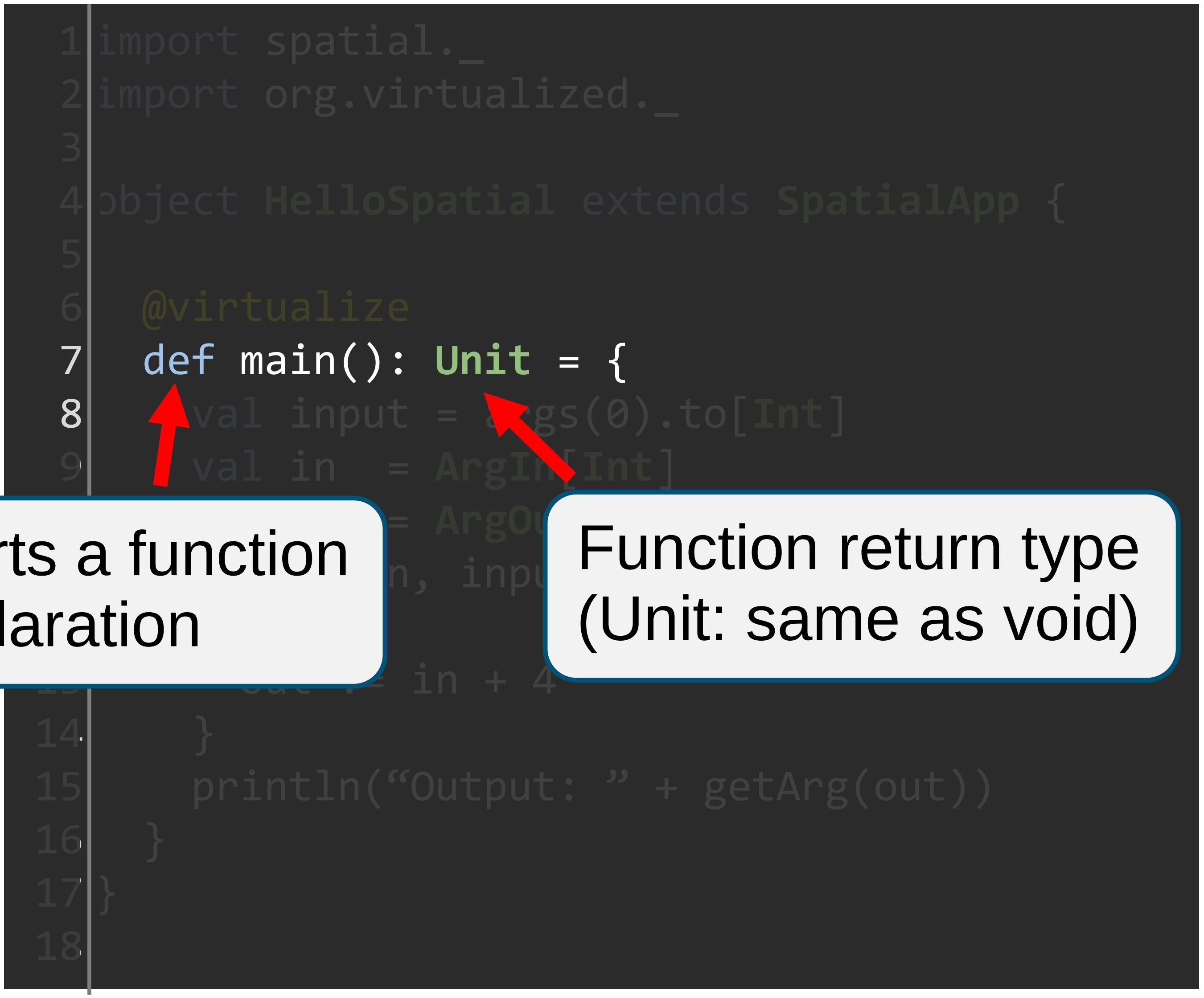
Spatial’s Entry Function: “main()”



```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[I
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

Spatial’s entry function

Spatial’s Entry Function: “main()”

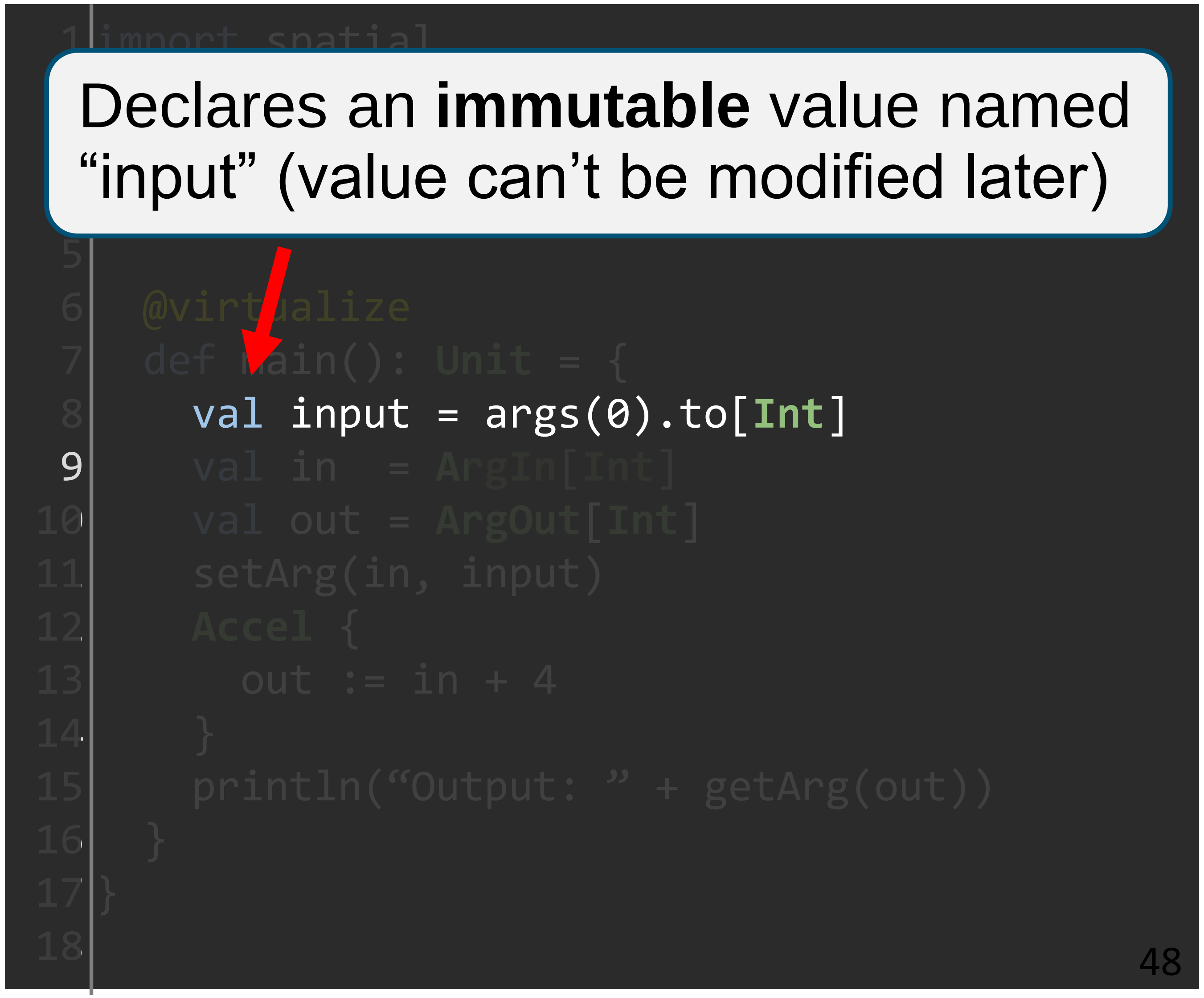


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in    = ArgIn[Int]
                = ArgOu
              n, inpu

             in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

Starts a function declaration

Function return type (Unit: same as void)

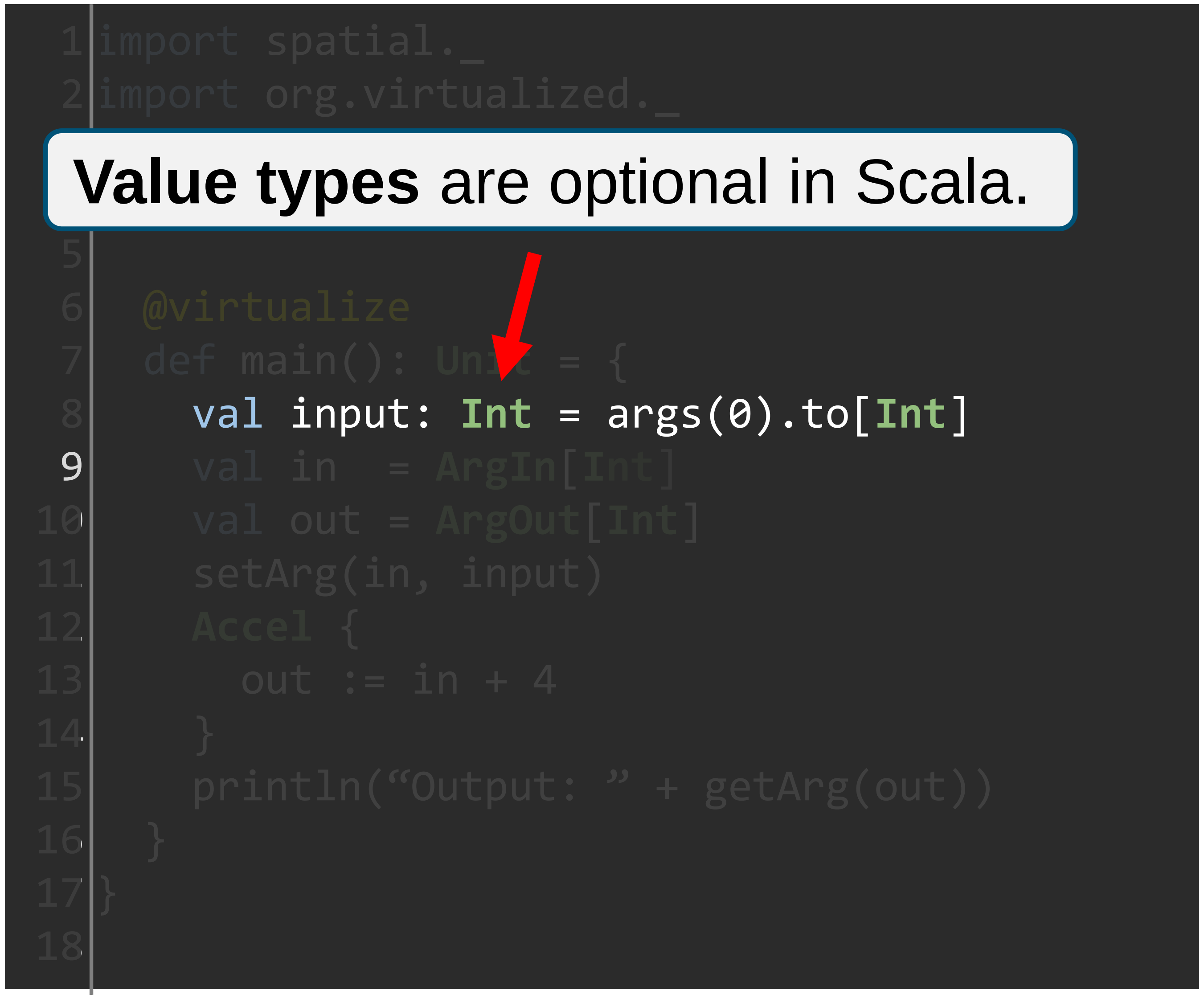
Val Definitions



Declares an **immutable** value named "input" (value can't be modified later)

```
import spatial

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

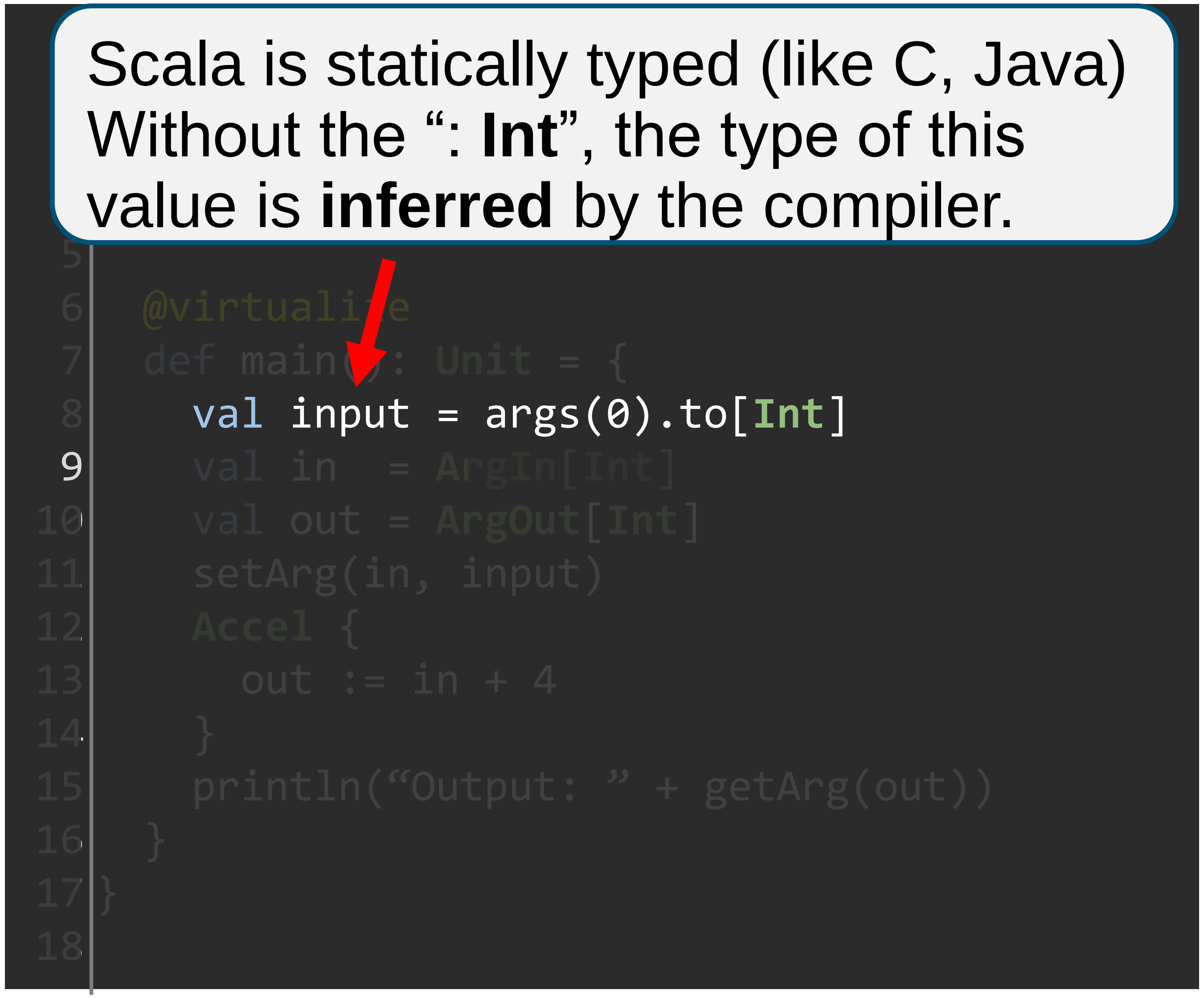
Val Definitions



```
import spatial._
import org.virtualized._


  @virtualize
  def main(): Unit = {
    val input: Int = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

Value types are optional in Scala.

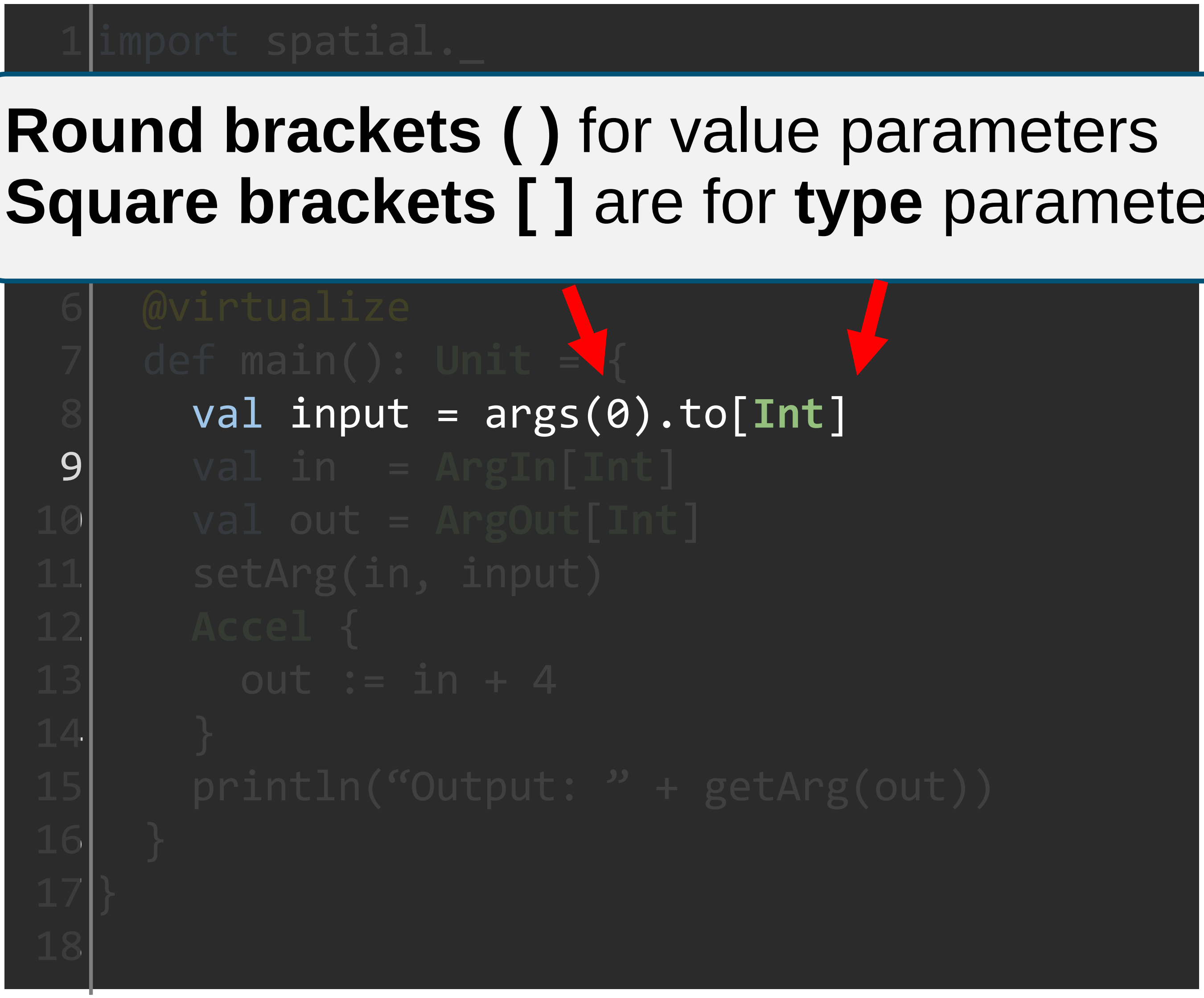
Val Definitions



Scala is statically typed (like C, Java)
Without the “: **Int**”, the type of this value is **inferred** by the compiler.

```
  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

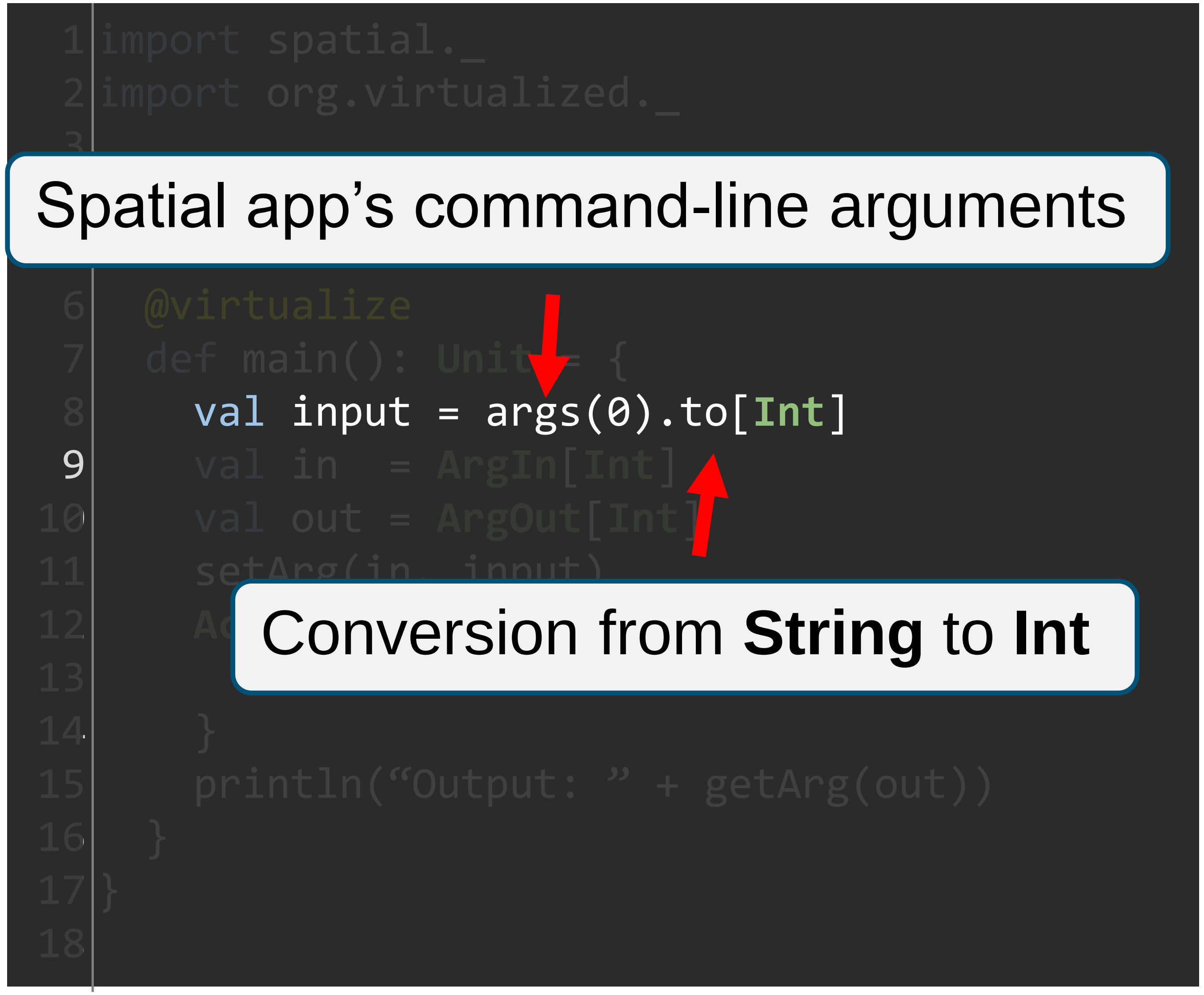
Method Calls



```
import spatial._
  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

Round brackets ( ) for value parameters
Square brackets [ ] are for **type** parameters

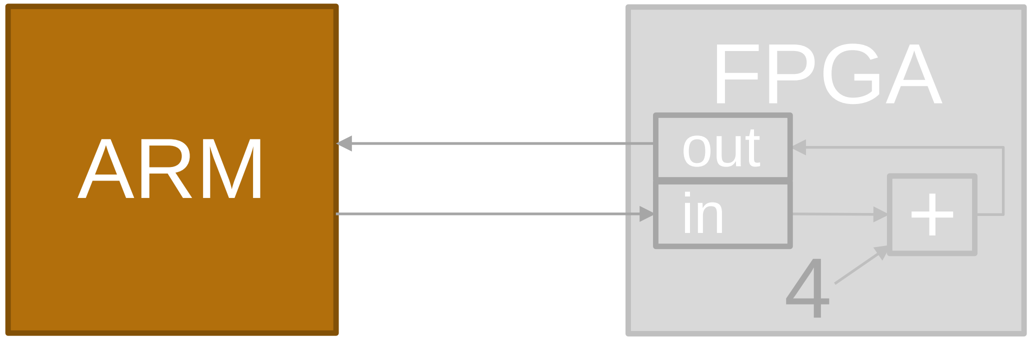
Spatial Command-Line Arguments



```
import spatial._
import org.virtualized._

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Ac

  }
  println("Output: " + getArg(out))
  }
}
```

Spatial app's command-line arguments

Conversion from **String** to **Int**

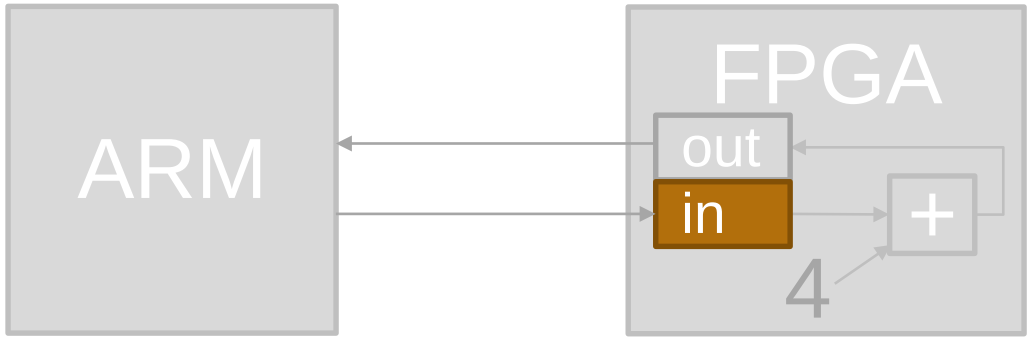


```
int main(int argc, char **argv) {
  int in = atoi(argv[1]);


  printf("Output: %d\n", out);
  return 0;
}
```

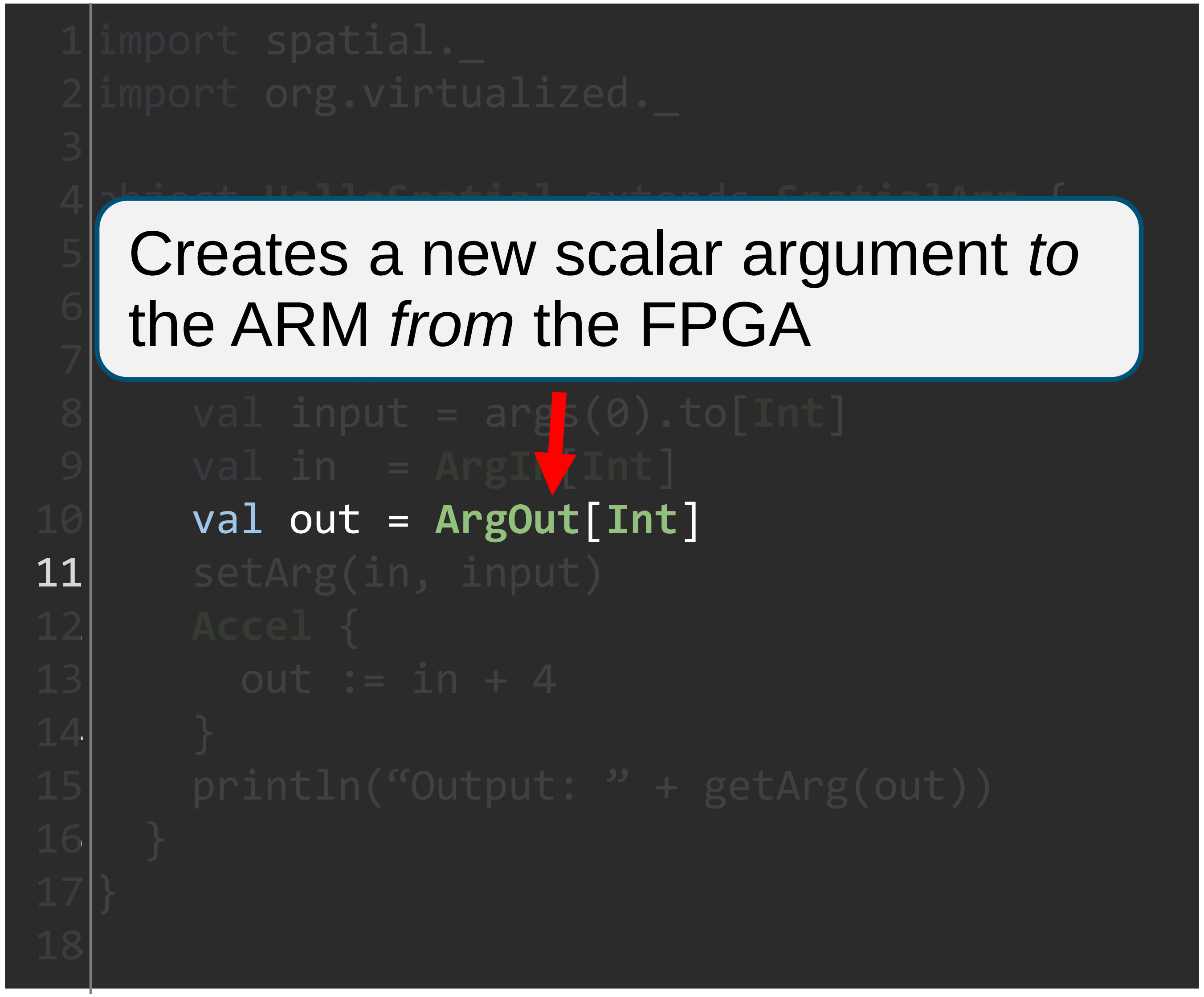
Input Arguments (ArgIn)

```
import spatial._
import org.virtualized._



  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

Creates a new register to capture a scalar argument *from* the ARM

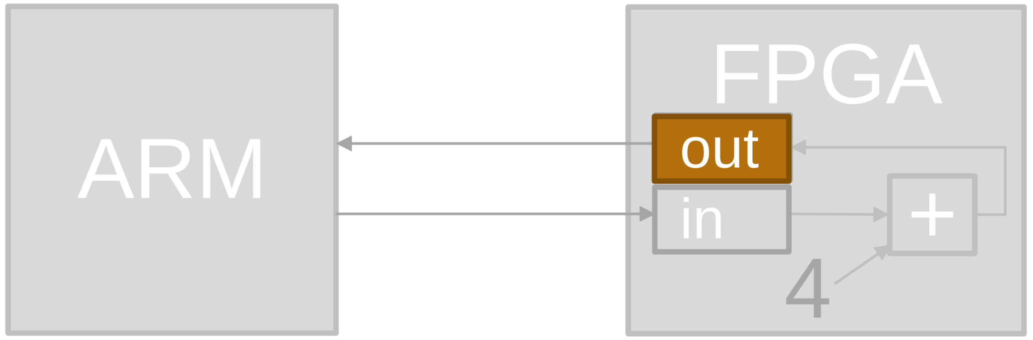


Output Arguments (ArgOut)

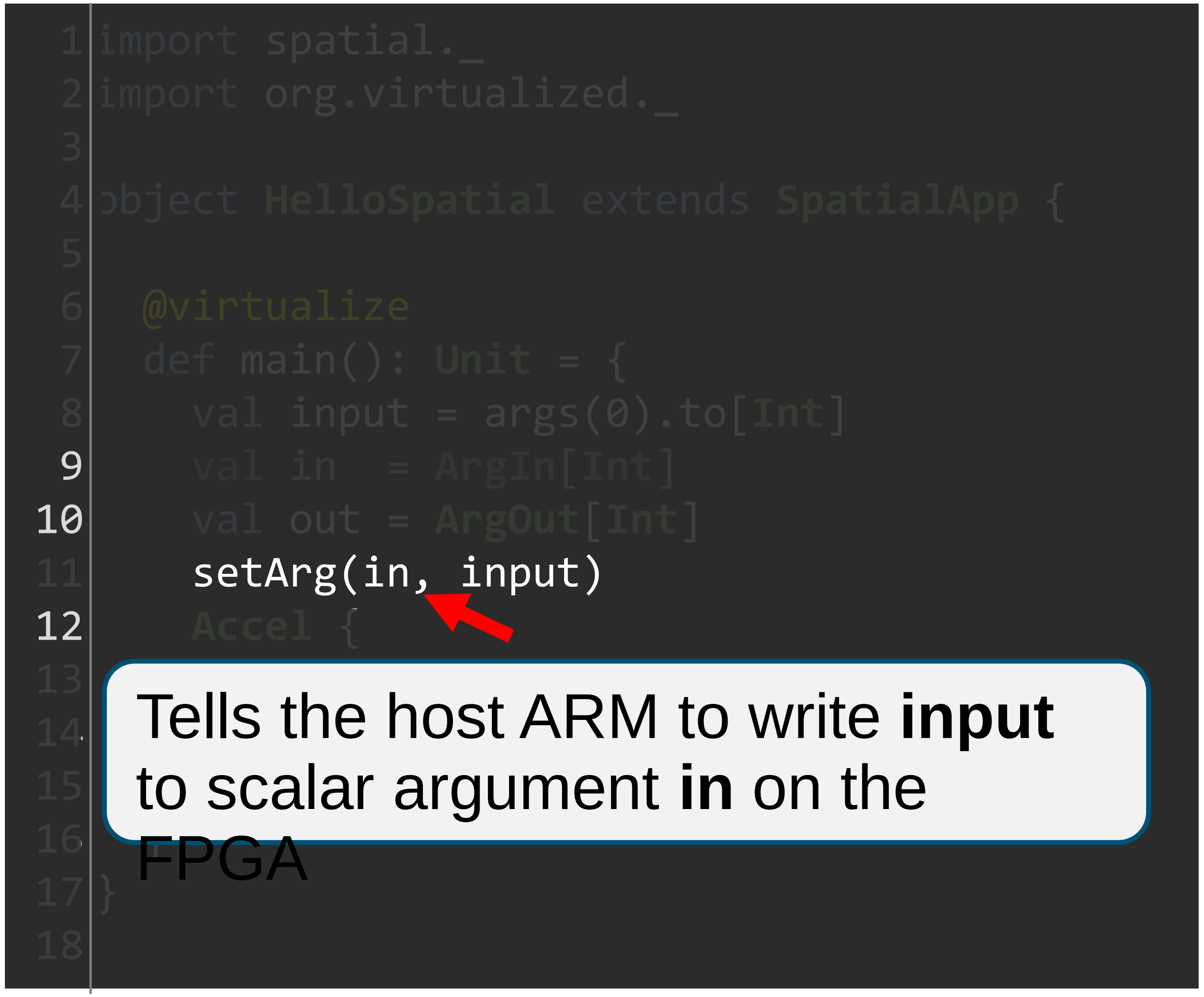


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {



    val input = args(0).to[Int]
    val in    = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

Creates a new scalar argument *to* the ARM *from* the FPGA

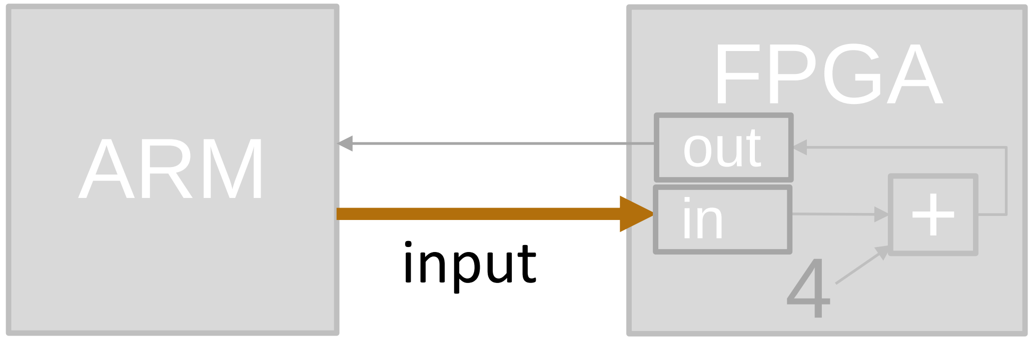


Scalar Transfers (ARM → FPGA)

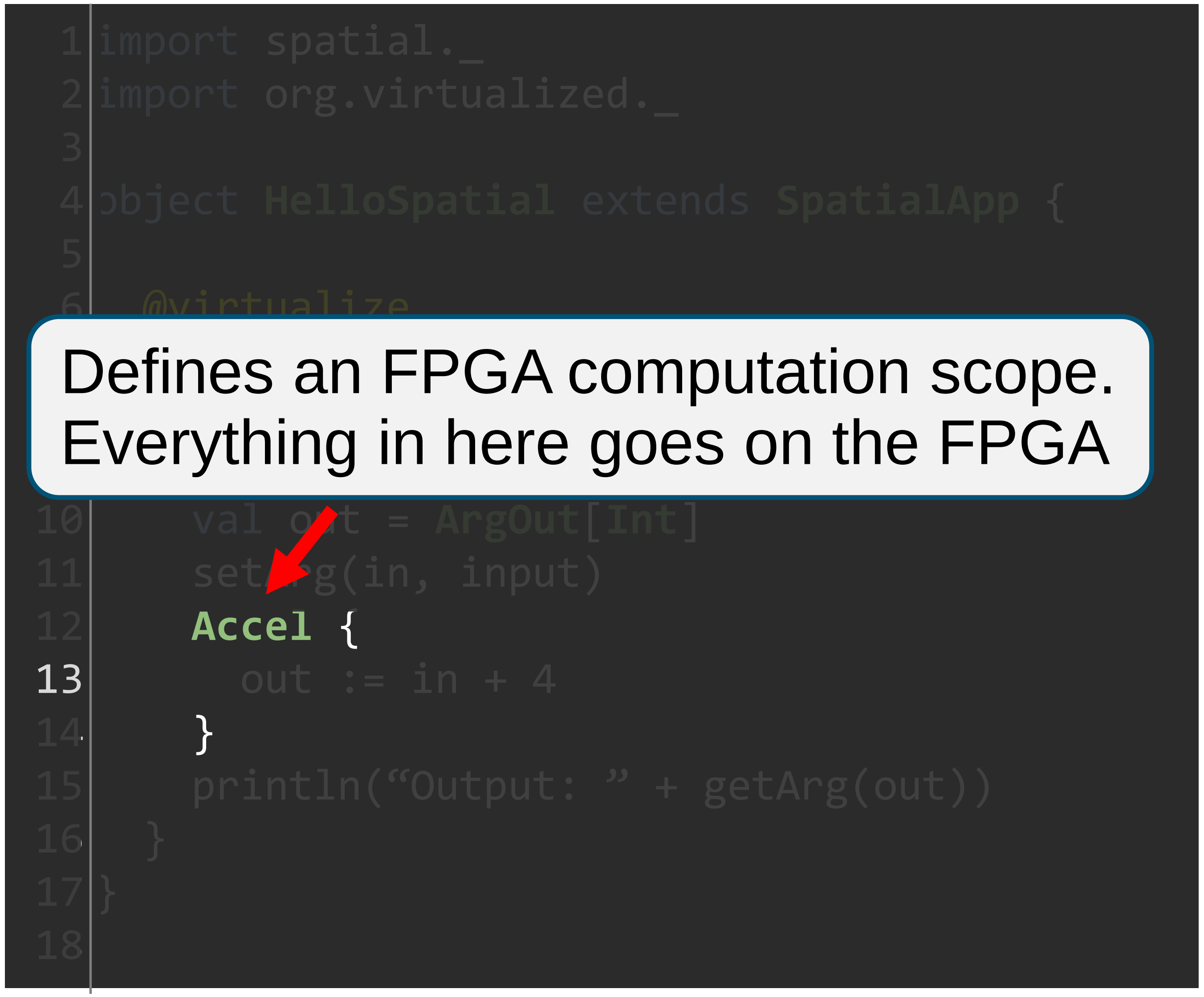


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {



}
```

Tells the host ARM to write **input** to scalar argument **in** on the FPGA

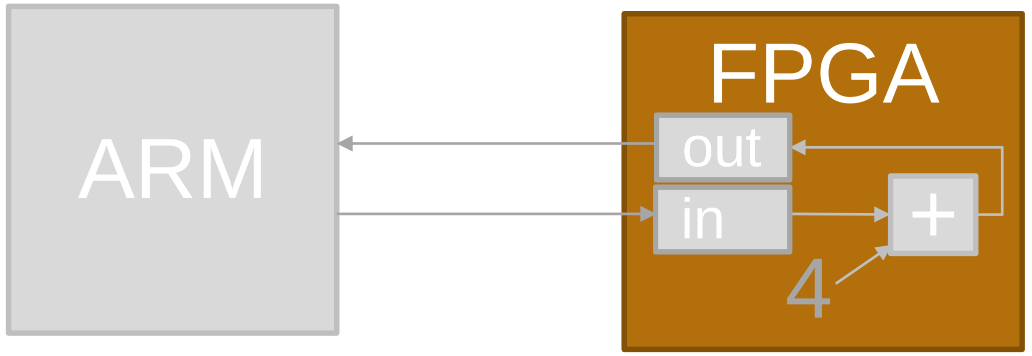


Accel Block



```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize

    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

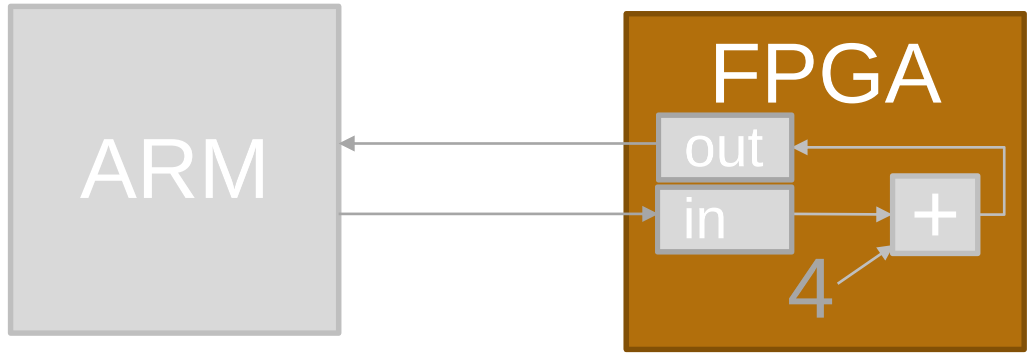
Defines an FPGA computation scope.
Everything in here goes on the FPGA



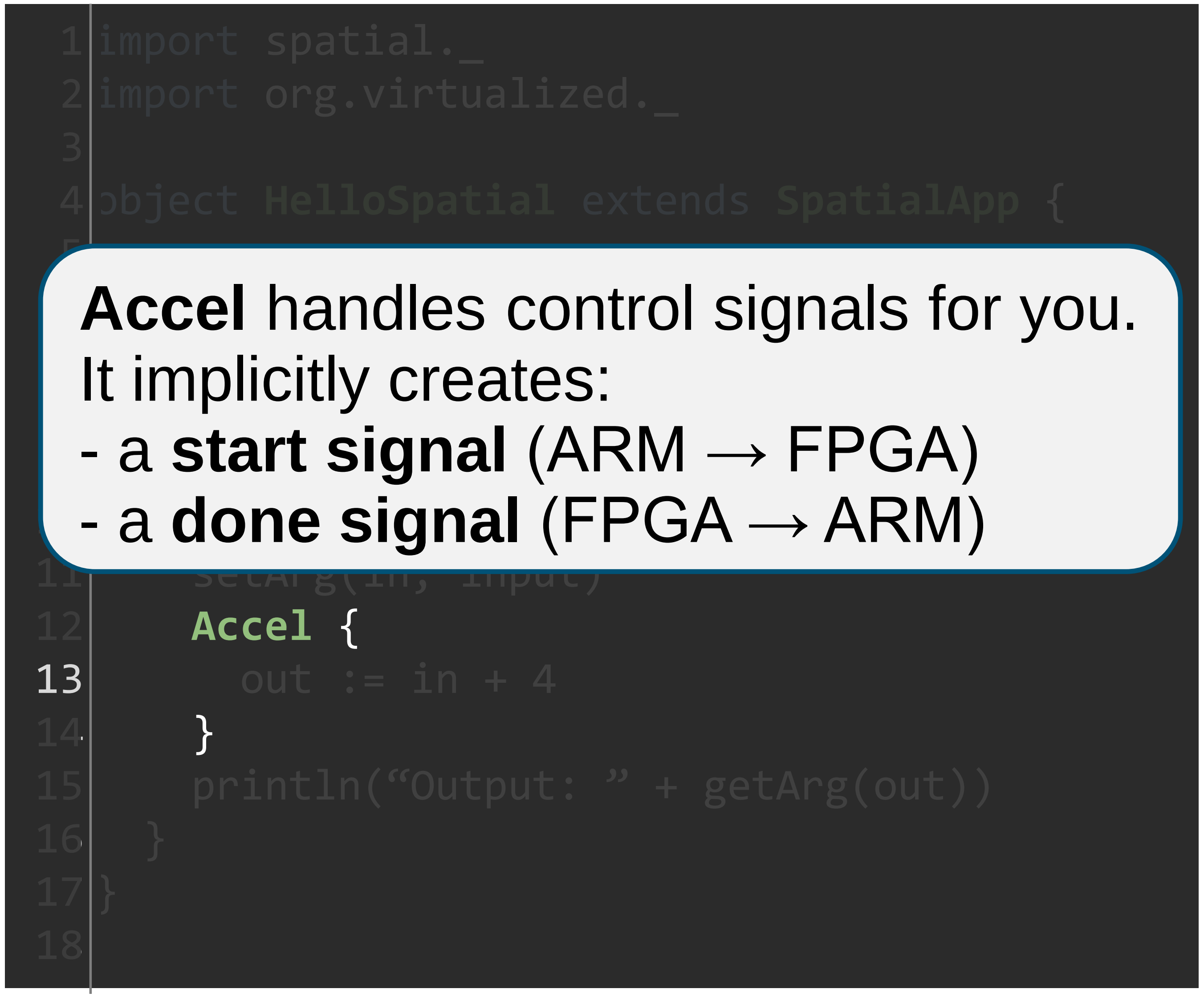
Accel Block

```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {


    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

The types of operations that can be done in this scope are limited to **synthesizable** Spatial



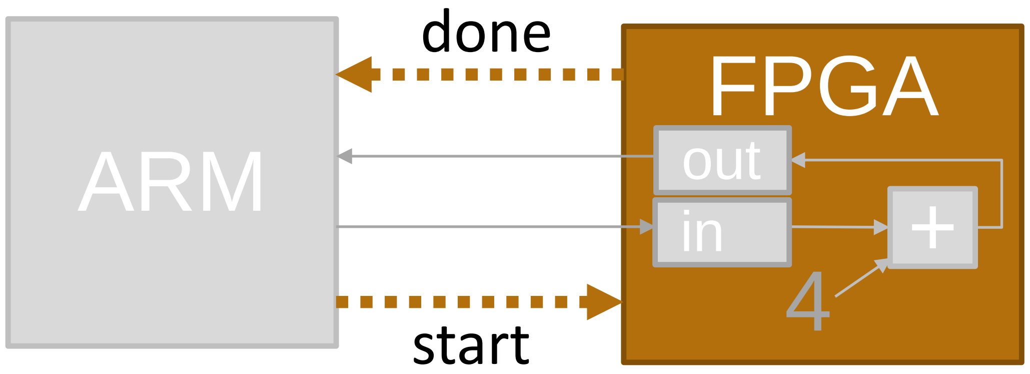
Accel Block



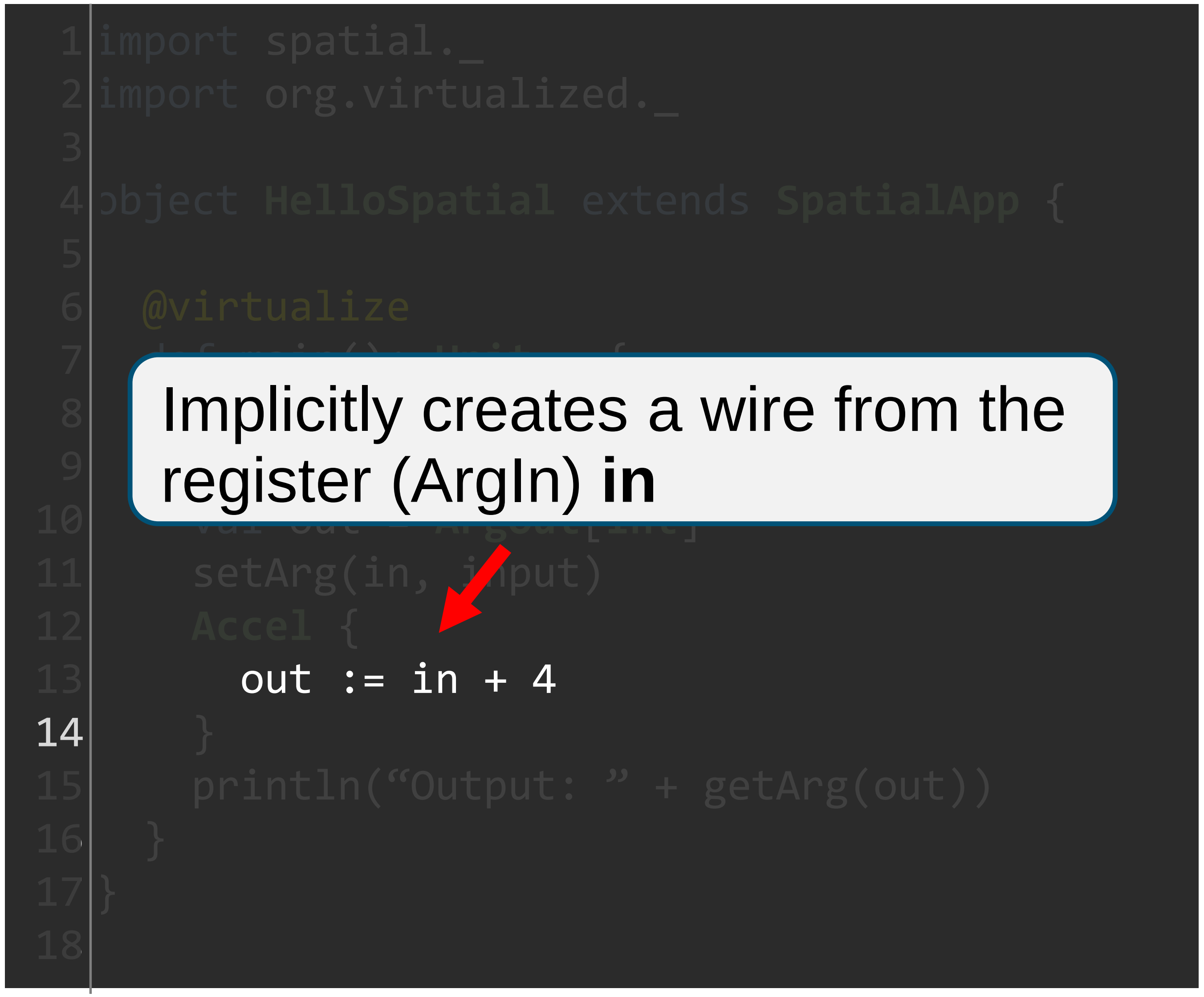
```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {
```

Accel handles control signals for you.
It implicitly creates:
- a **start signal** (ARM → FPGA)
- a **done signal** (FPGA → ARM)

```
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}
```

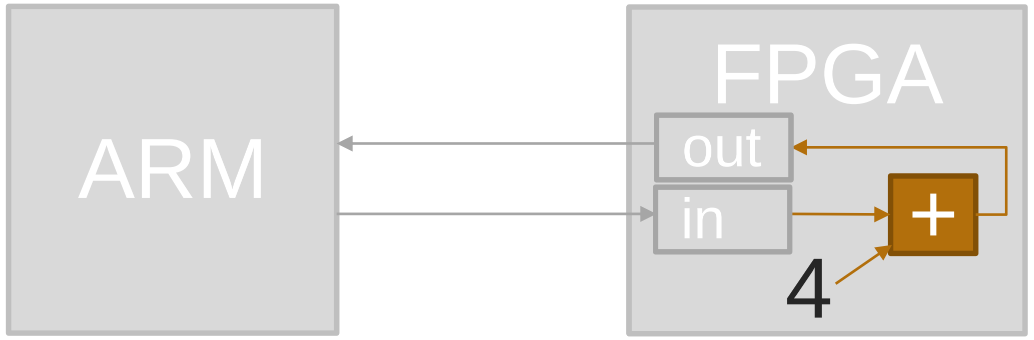


Implicit Register Reads

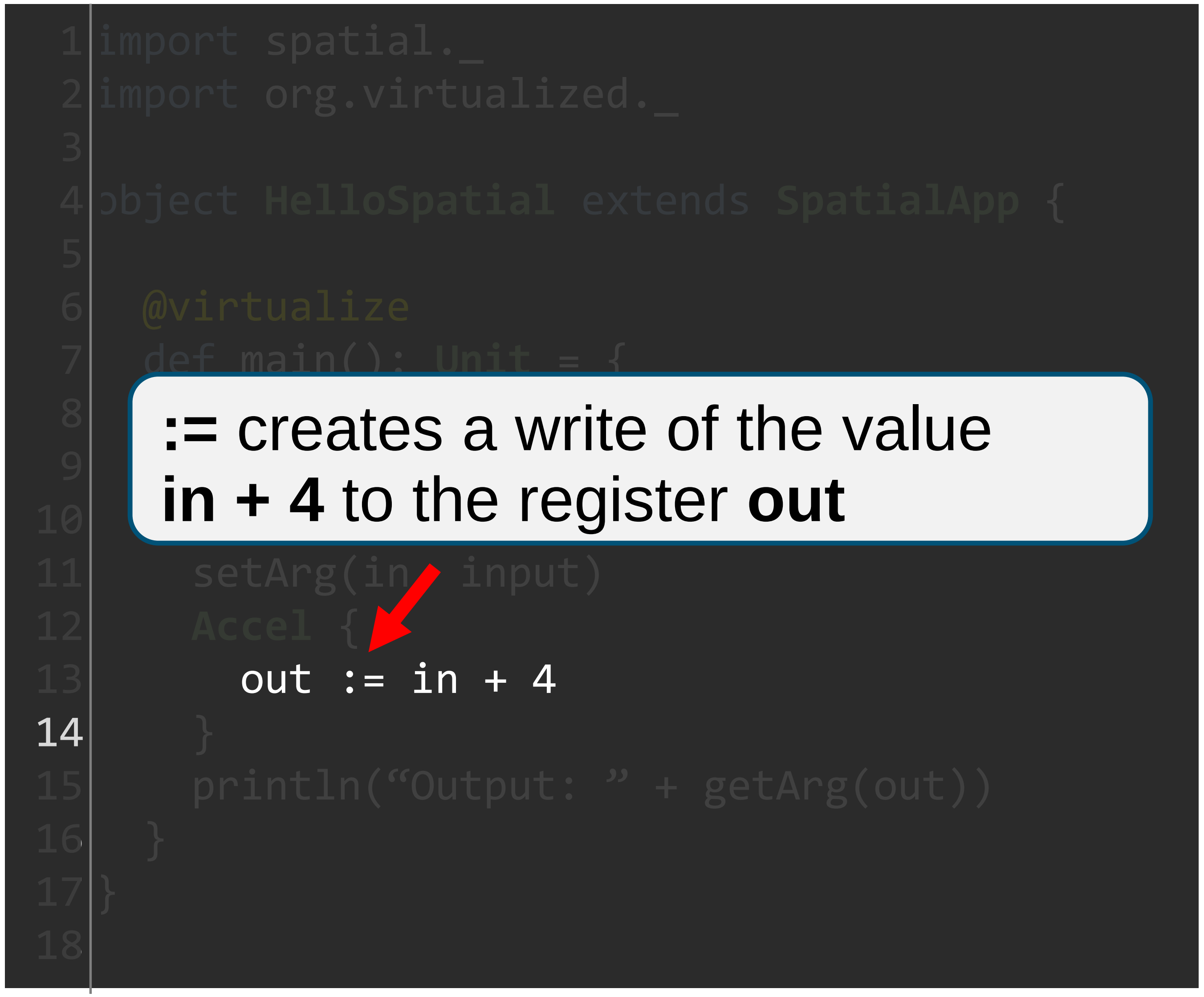


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {


    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}

```

Implicitly creates a wire from the register (ArgIn) **in**

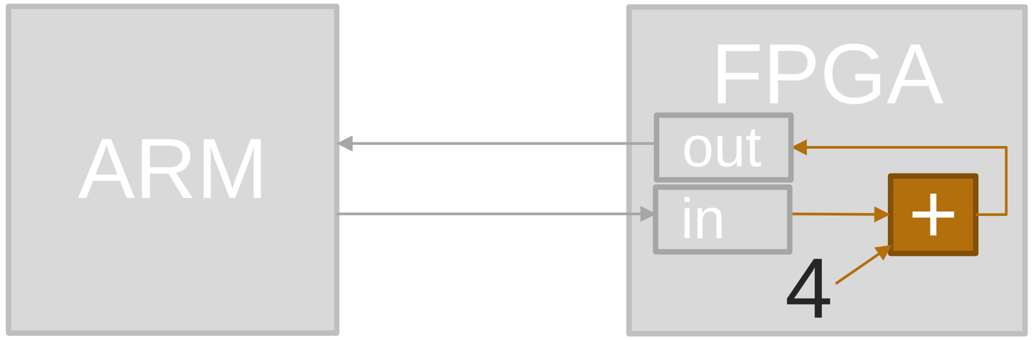


Register Writes

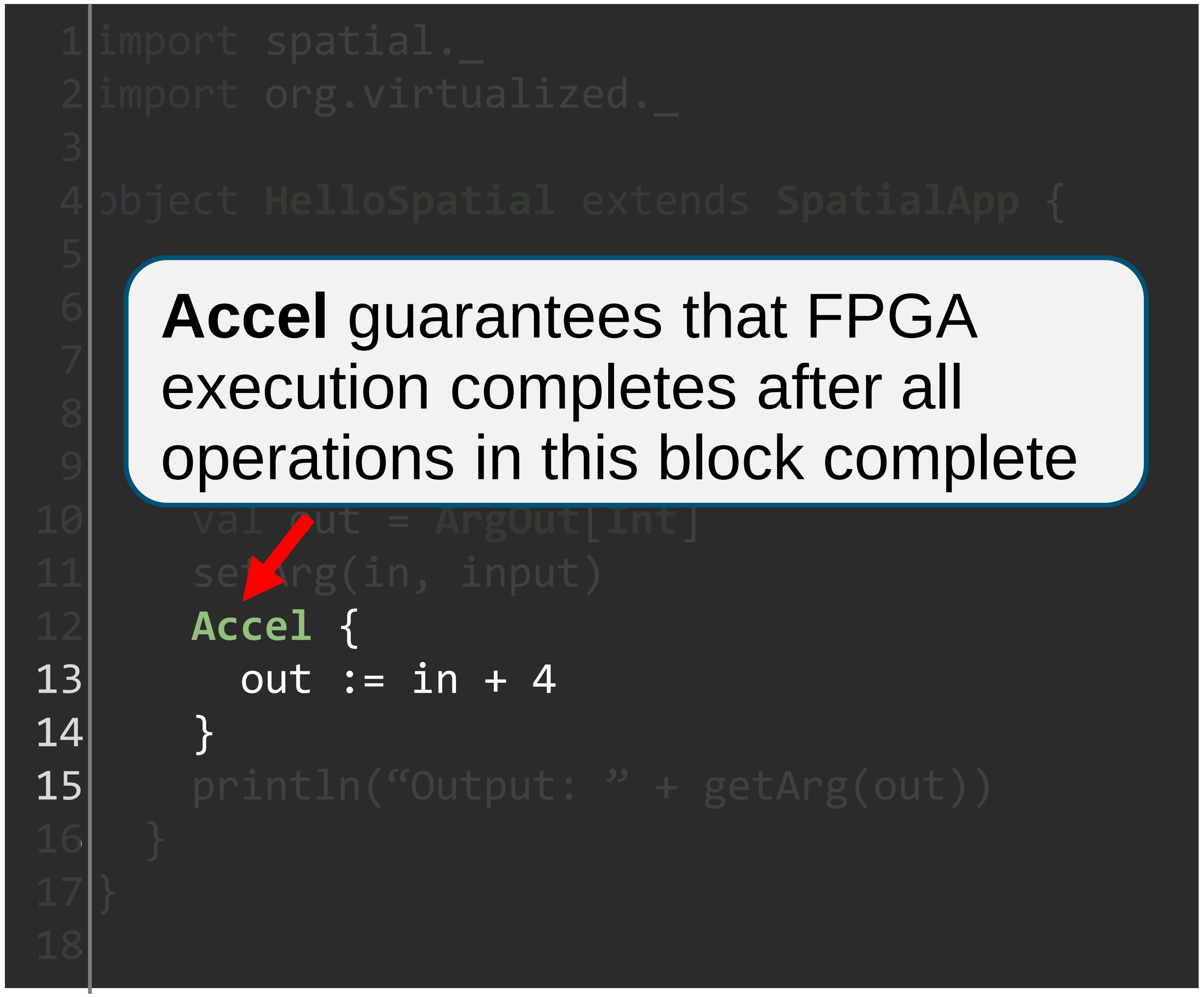


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {



    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}

```

:= creates a write of the value **in + 4** to the register **out**

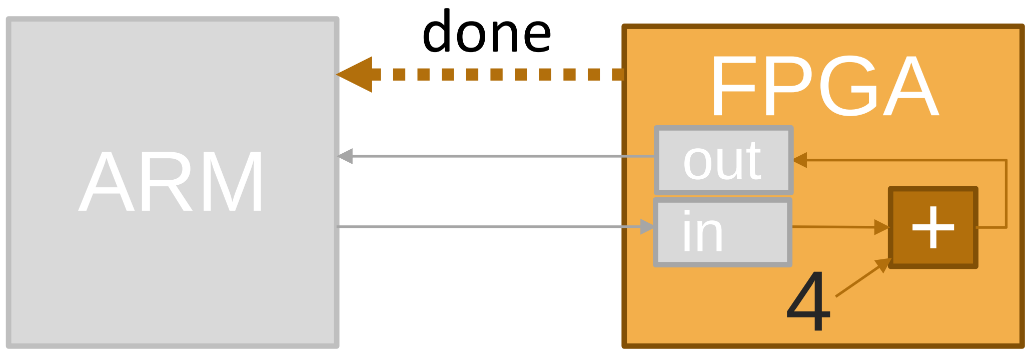


Accel Block Scheduling



```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

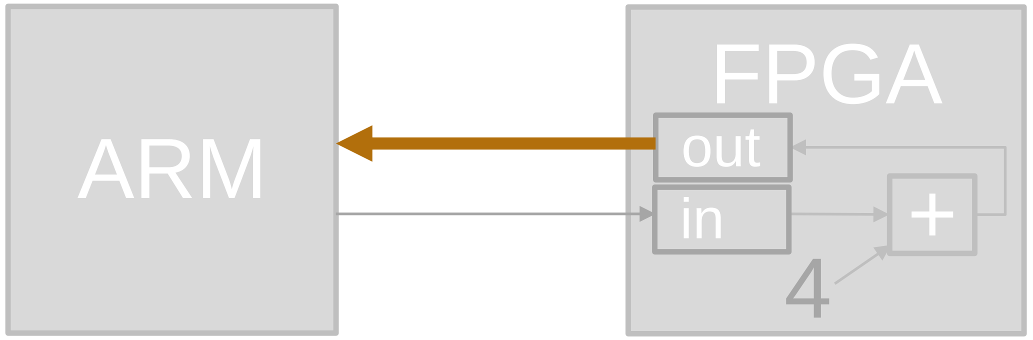
Accel guarantees that FPGA execution completes after all operations in this block complete



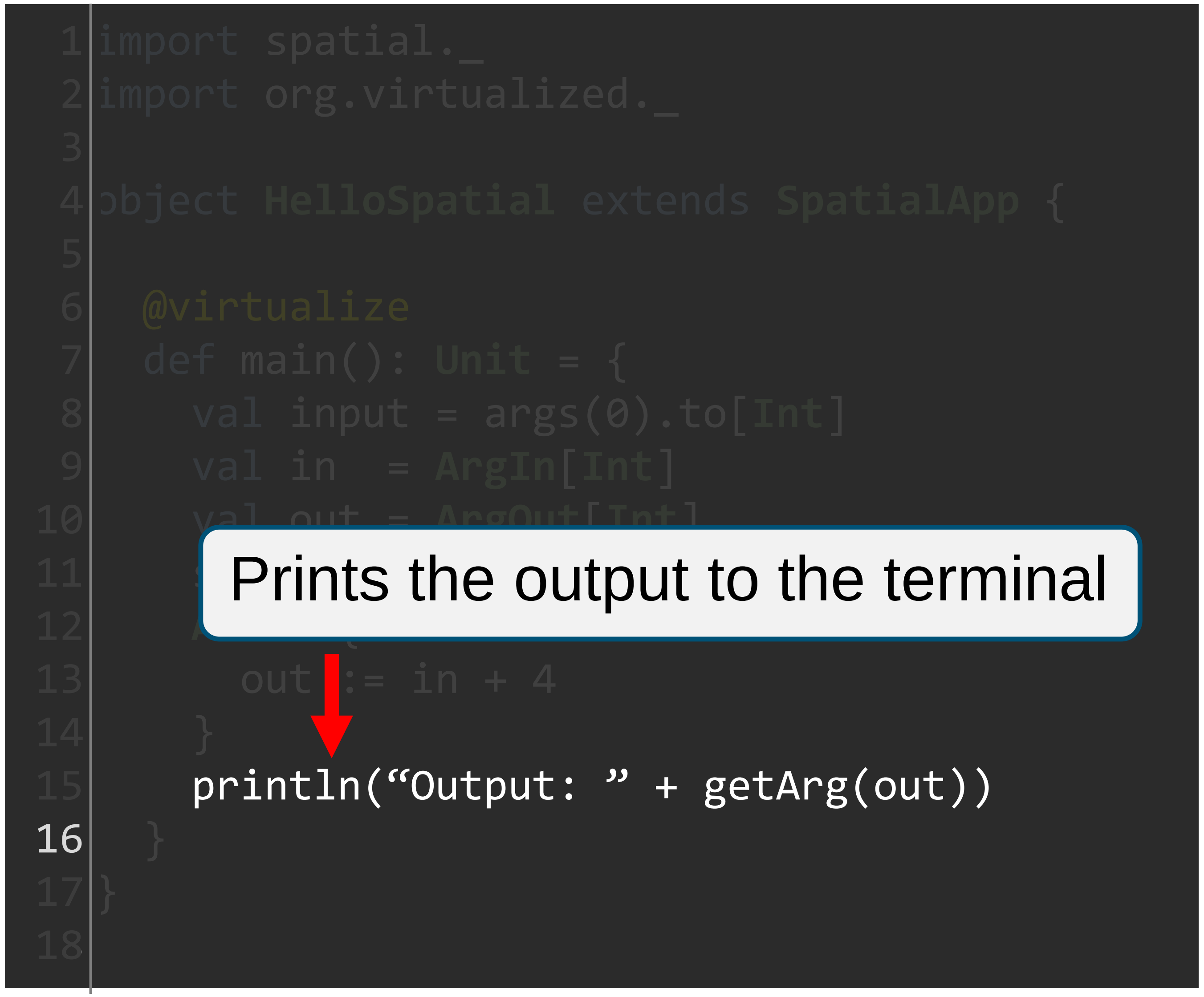
Scalar Transfers (FPGA → ARM)

```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]



      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}

```

Gets the value of the ArgOut **out** from the FPGA back to the ARM



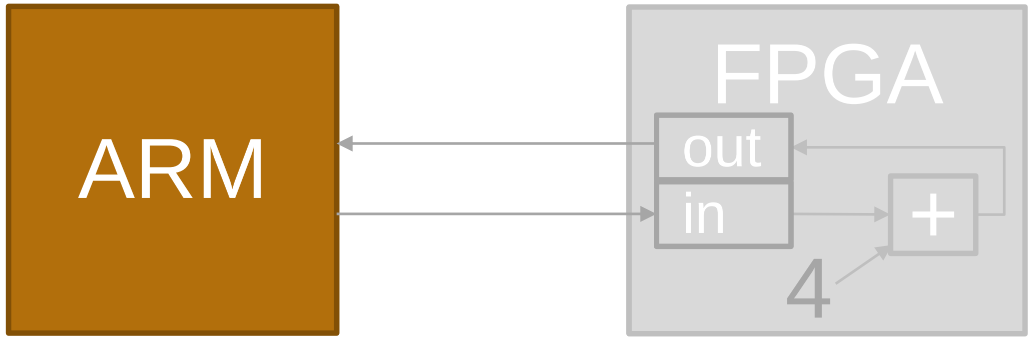
Printing in Spatial**



```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]

      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```

Prints the output to the terminal

** Printing in Spatial isn't synthesizable, but it can be used in **host code** and in **debugging** (more in future lectures)



```
int main(int argc, char **argv) {
  int in = atoi(argv[1]);
  …

  printf("Output: %d\n", out);
  return 0;
}
```

Hello Spatial!

```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println(“Output: ” + getArg(out))
  }
}

```

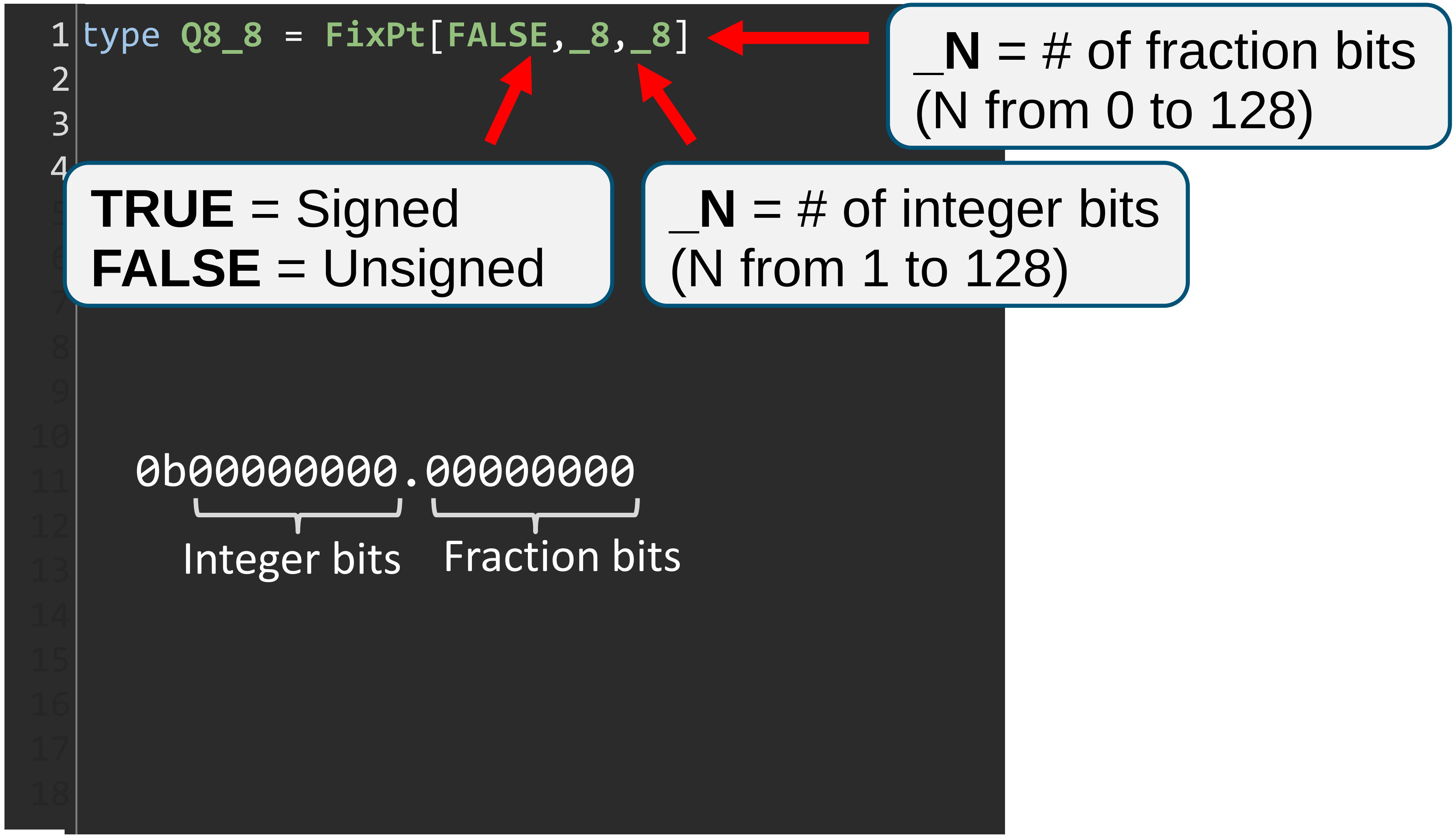
Custom Types in Spatial

- Now what if we want an ArgIn value that isn’t an Int?
- Other options:
  - Custom fixed point types
  - Custom floating point types
  - Structs
  - Vectors

Custom Types

```
val input = args(0).to[Int]
val in  = ArgIn[Int]

setArg(in, input)
```

Custom Fixed Point Types



```
type Q8_8 = FixPt[FALSE,_8,_8]
```

_N = # of fraction bits
(N from 0 to 128)

TRUE = Signed
FALSE = Unsigned

_N = # of integer bits
(N from 1 to 128)

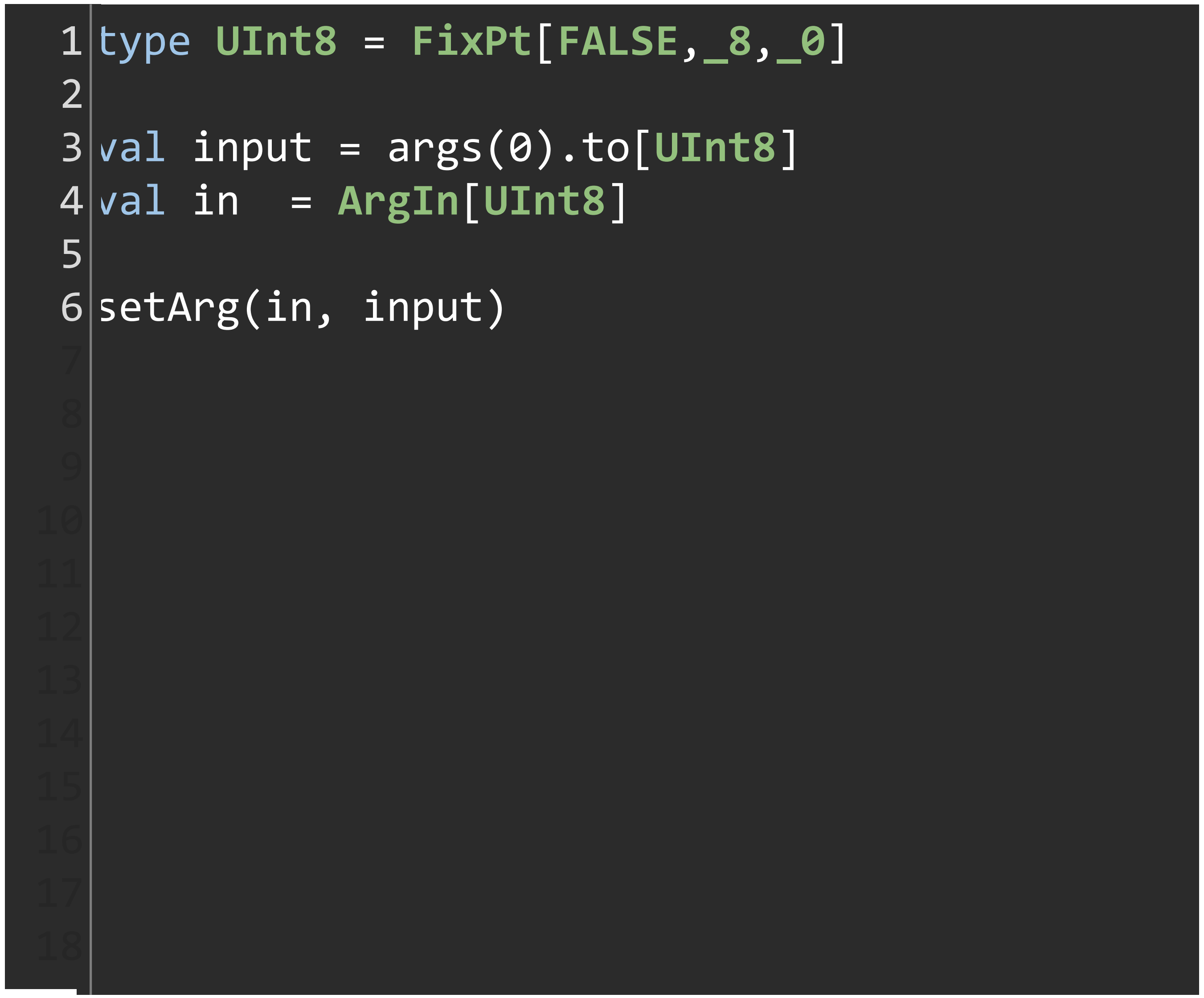
$\underbrace{\texttt{0b00000000}}_{\text{Integer bits}}\texttt{.}\underbrace{\texttt{00000000}}_{\text{Fraction bits}}$

Custom Fixed Point Examples

```
type Q8_8 = FixPt[FALSE,_8,_8]

type UInt8 = FixPt[FALSE,_8,_0]

type LongLong = FixPt[TRUE,_128,_0]
```

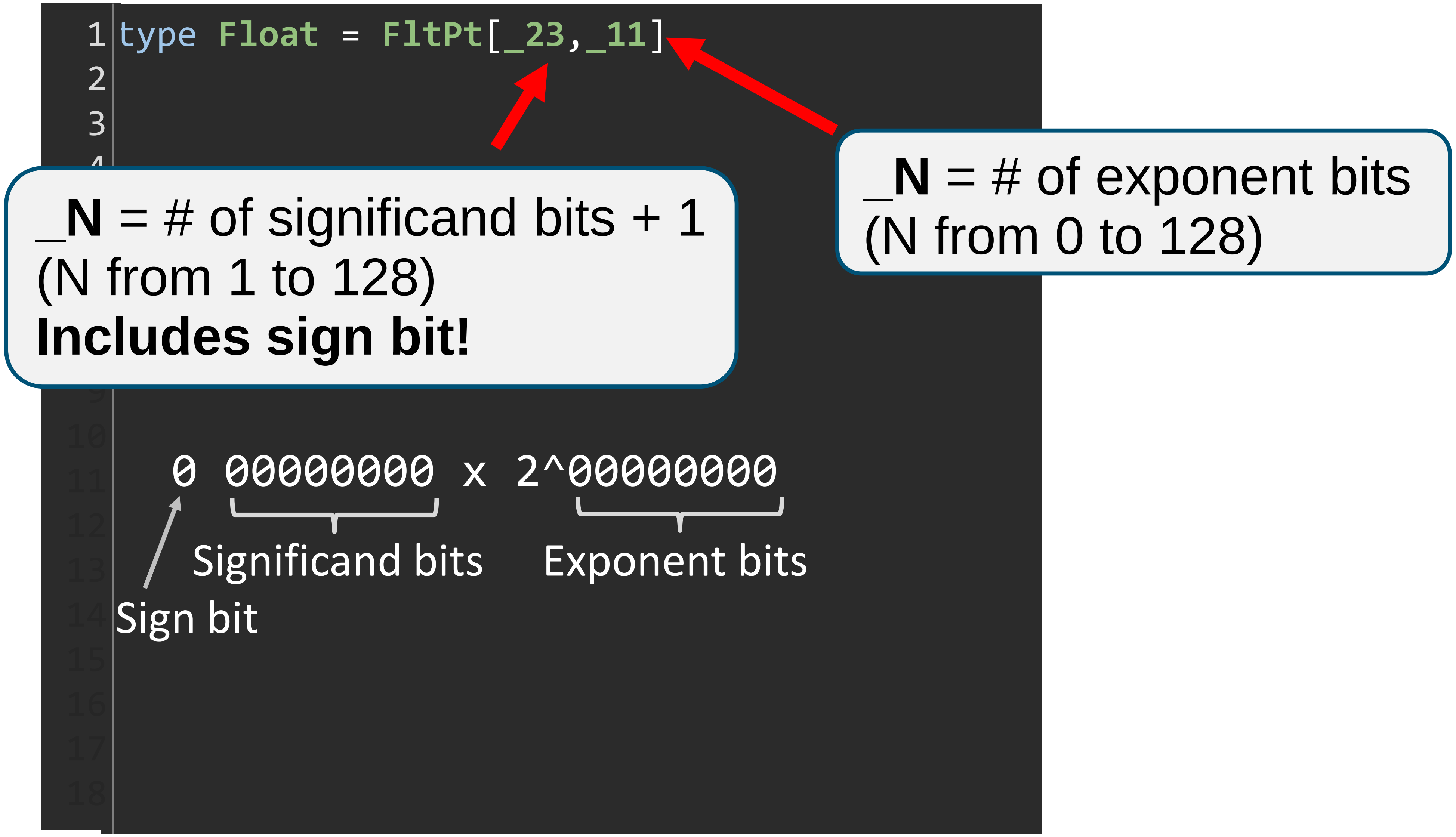
$$\texttt{0b}\underbrace{\texttt{00000000}}_{\text{Integer bits}}\texttt{.}\underbrace{\texttt{00000000}}_{\text{Fraction bits}}$$

Custom Fixed Point Types



```
type UInt8 = FixPt[FALSE,_8,_0]

val input = args(0).to[UInt8]
val in  = ArgIn[UInt8]

setArg(in, input)
```

Custom Floating Point Types



```
type Float = FltPt[_23,_11]
```

_**N** = # of significand bits + 1
(N from 1 to 128)
Includes sign bit!

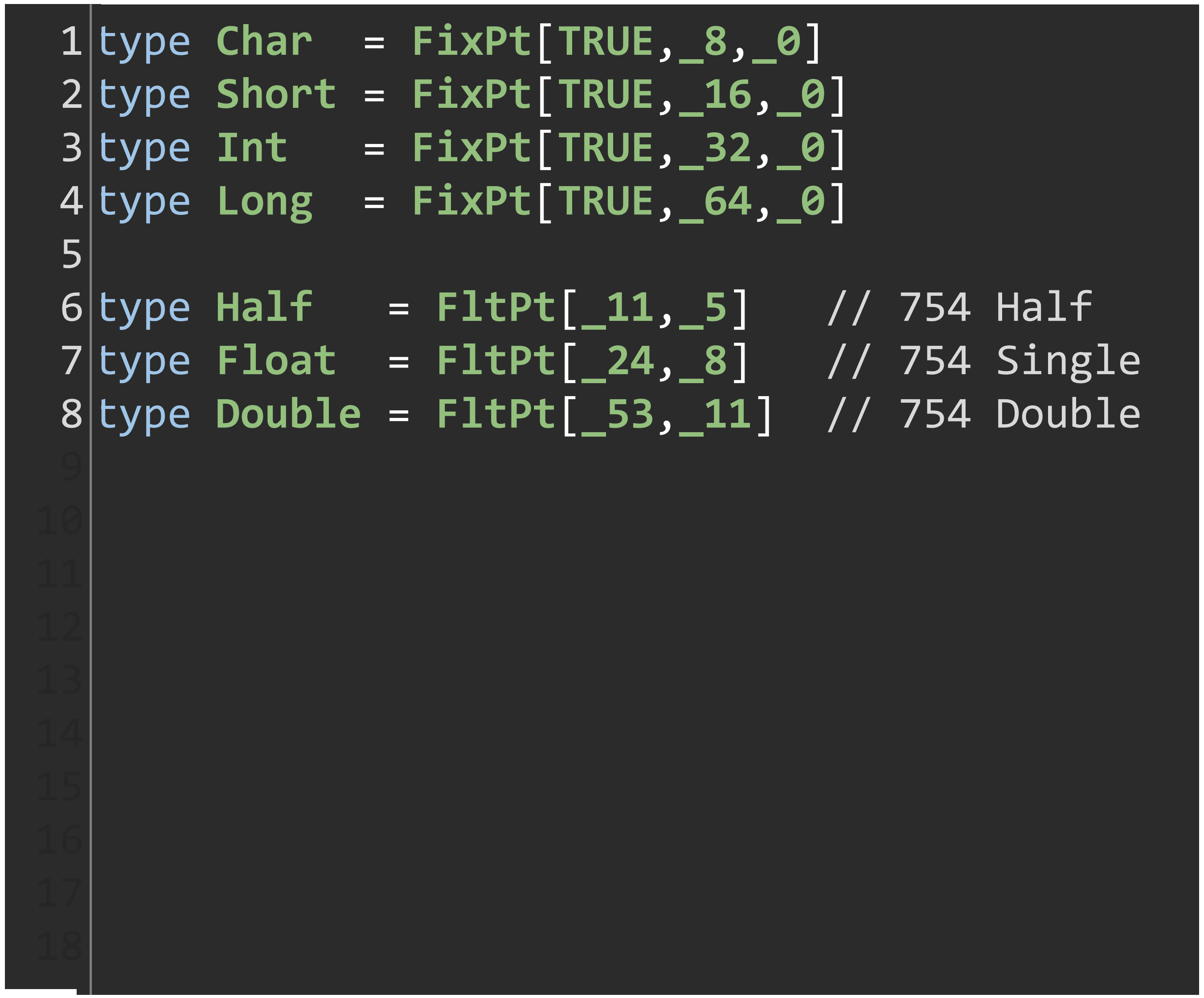
_**N** = # of exponent bits
(N from 0 to 128)

```
0 00000000 x 2^00000000
```

Custom Floating Point Types

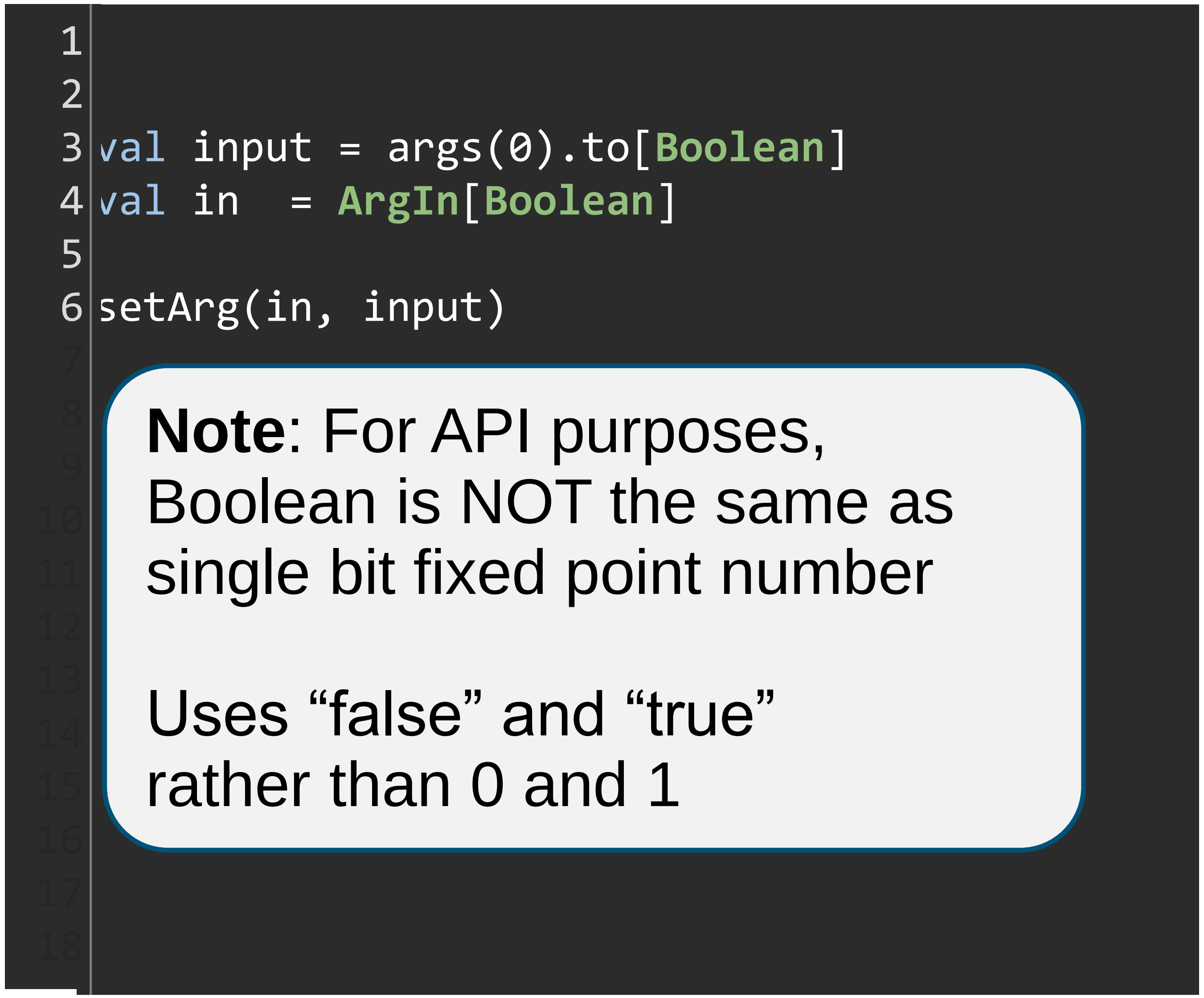
```
type Half = FltPt[_11,_5]

val input = args(0).to[Half]
val in   = ArgIn[Half]

setArg(in, input)
```

Predefined Type Aliases



```
type Char   = FixPt[TRUE,_8,_0]
type Short  = FixPt[TRUE,_16,_0]
type Int    = FixPt[TRUE,_32,_0]
type Long   = FixPt[TRUE,_64,_0]

type Half    = FltPt[_11,_5]   // 754 Half
type Float   = FltPt[_24,_8]   // 754 Single
type Double  = FltPt[_53,_11]  // 754 Double
```

Note About Booleans



```
val input = args(0).to[Boolean]
val in  = ArgIn[Boolean]

setArg(in, input)
```

Note: For API purposes, Boolean is NOT the same as single bit fixed point number

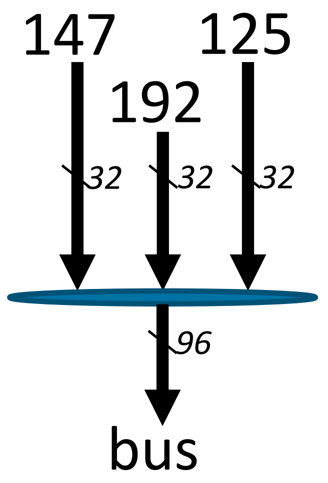
Uses “false” and “true” rather than 0 and 1

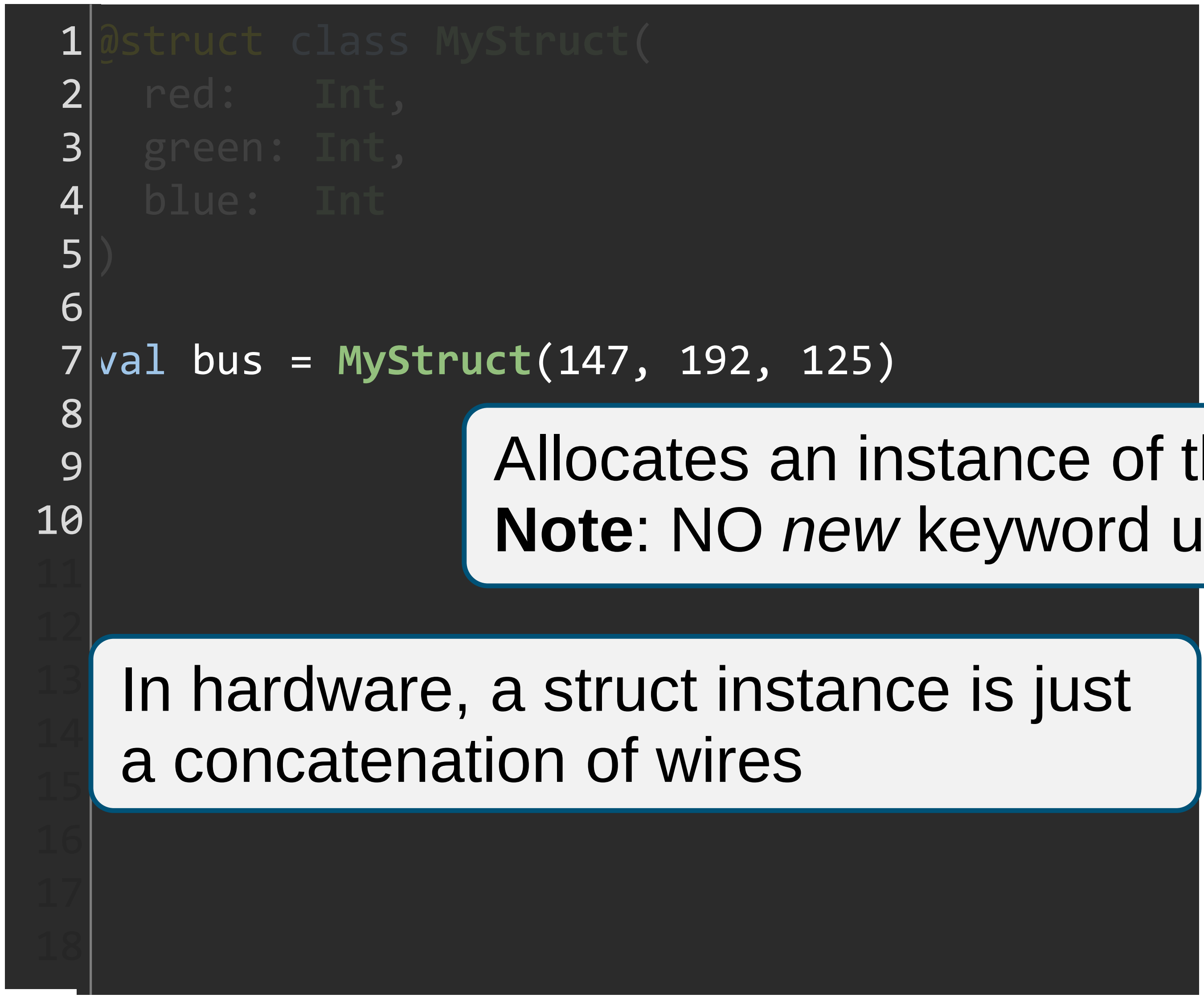
Custom Structs

```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)
```

Declares a new Struct type with the given list of fields

Custom Structs



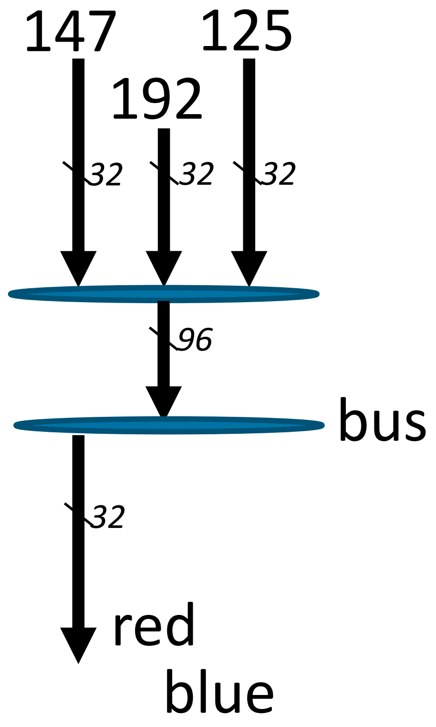


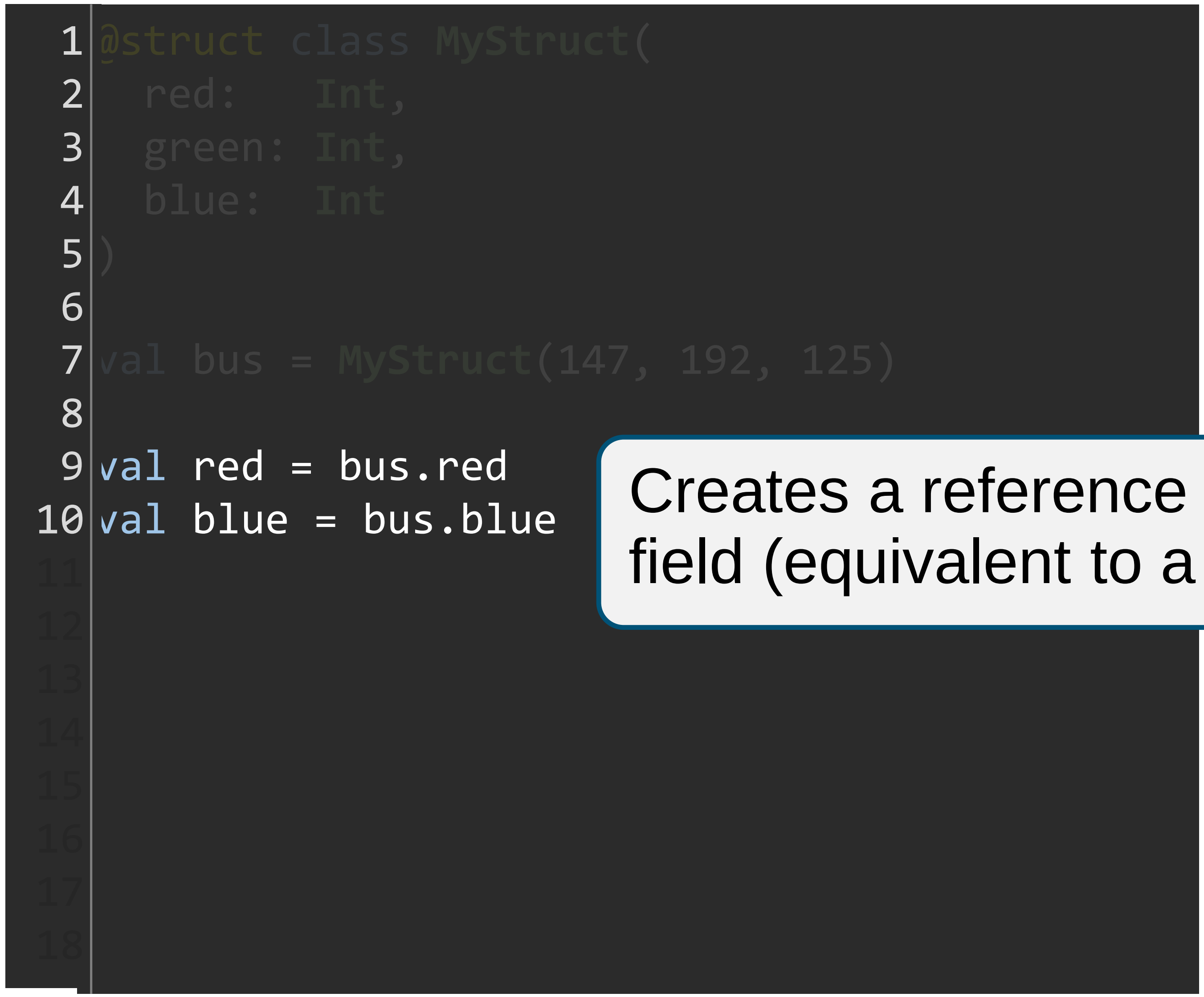
```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

val bus = MyStruct(147, 192, 125)
```

Allocates an instance of the struct.
Note: NO *new* keyword used

In hardware, a struct instance is just a concatenation of wires

Custom Structs

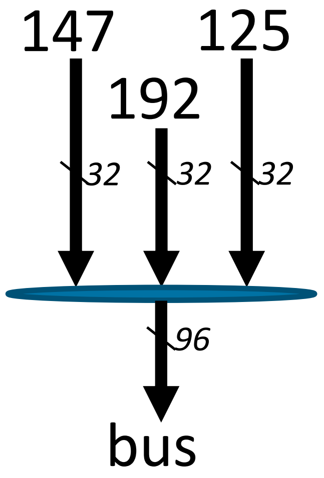


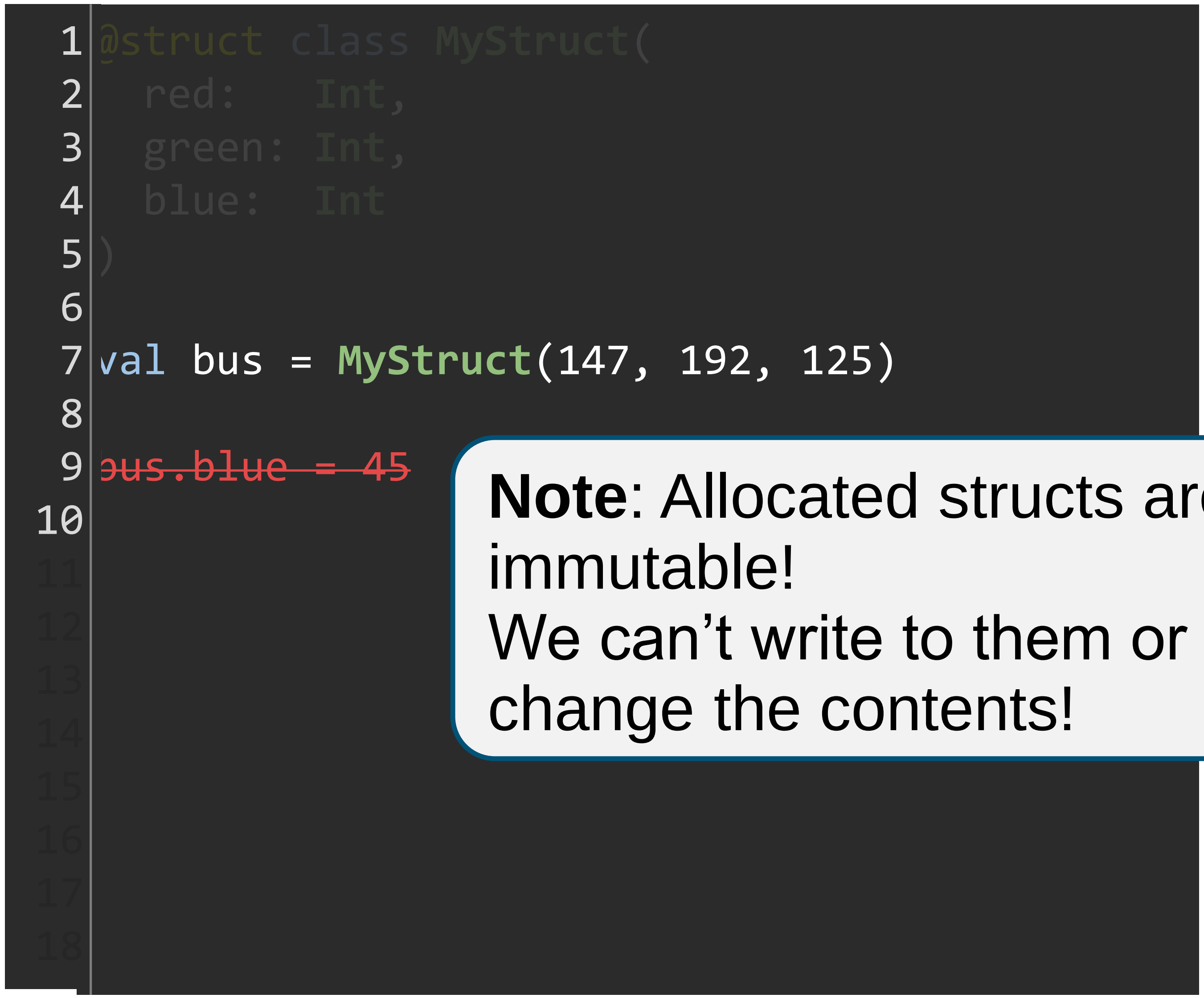


```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

val bus = MyStruct(147, 192, 125)

val red = bus.red
val blue = bus.blue
```

Creates a reference to the struct field (equivalent to a bit slice)

Custom Structs

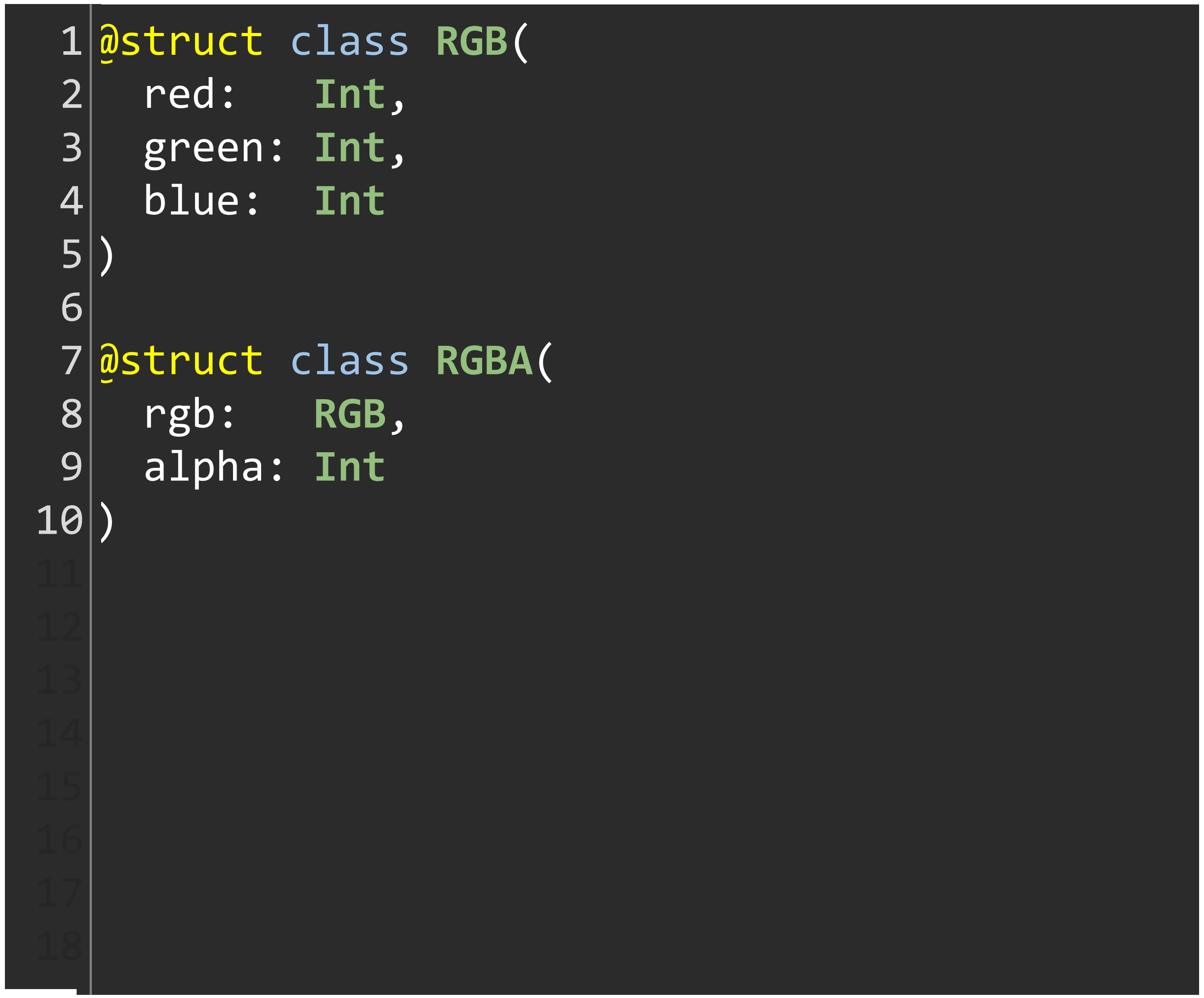




```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

val bus = MyStruct(147, 192, 125)

~~bus.blue = 45~~
```

Note: Allocated structs are immutable!
We can't write to them or change the contents!

Nesting Structs



```
@struct class RGB(
  red:   Int,
  green: Int,
  blue:  Int
)

@struct class RGBA(
  rgb:   RGB,
  alpha: Int
)
```

Registers of Custom Types

```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

val in  = ArgIn[MyStruct]

in.red
```

Creates an ArgIn register which holds a value of type MyStruct

Note: Registers can hold structs as long as the fields are primitive values (FixPt, FltPt, Boolean) or other primitive-based structs