Spatial Tutorial

Part 1: Introduction to Spatial

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

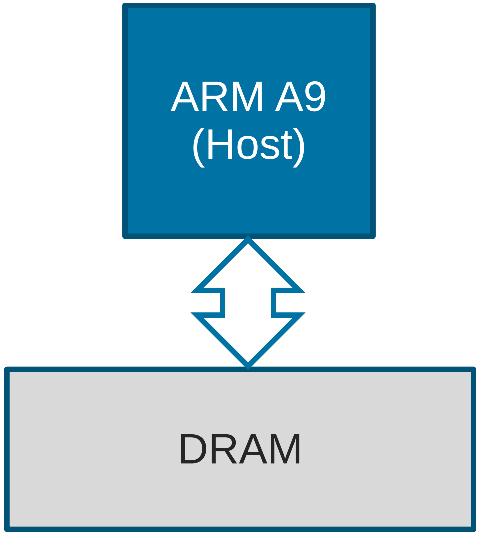
INTRODUCTION

Hardware Accelerator Design

- Hardware accelerators?
- What is involved in designing one?

Simple Example: ARM (CPU Host)

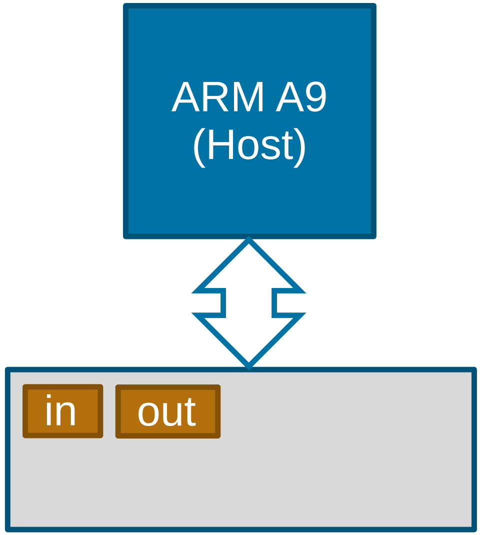
- Add '4' to an input integer



```
int main(int argc, char **argv) {
  int in = atoi(argv[1]);
  int out = in + 4;
  printf(“Output: %d\n”, out);
  return 0;
}
```

Simple Example: ARM A9 (Host)

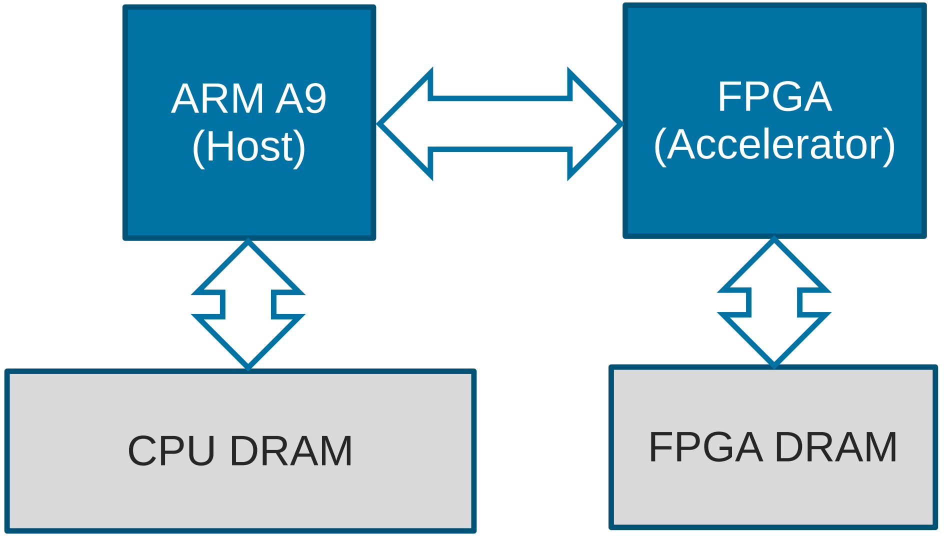
- Add '4' to an input integer



```
int main(int argc, char **argv) {
  int in = atoi(argv[1]);
  int out = in + 4;
  printf(“Output: %d\n”, out);
  return 0;
}
```

Simple Example: ARM + FPGA

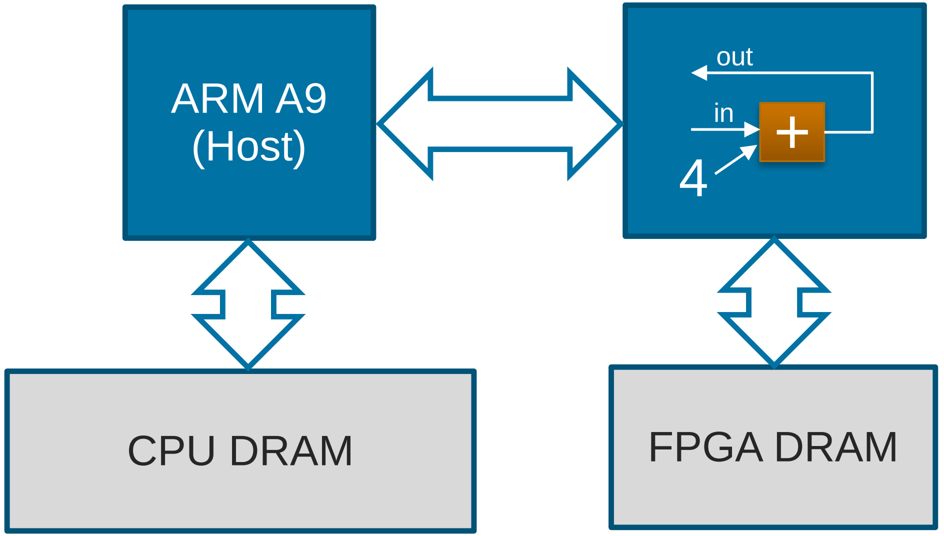
- Perform addition on FPGA



```
// Host Code: C
??

// FPGA Code: Verilog
??
```

Simple Example: ARM + FPGA

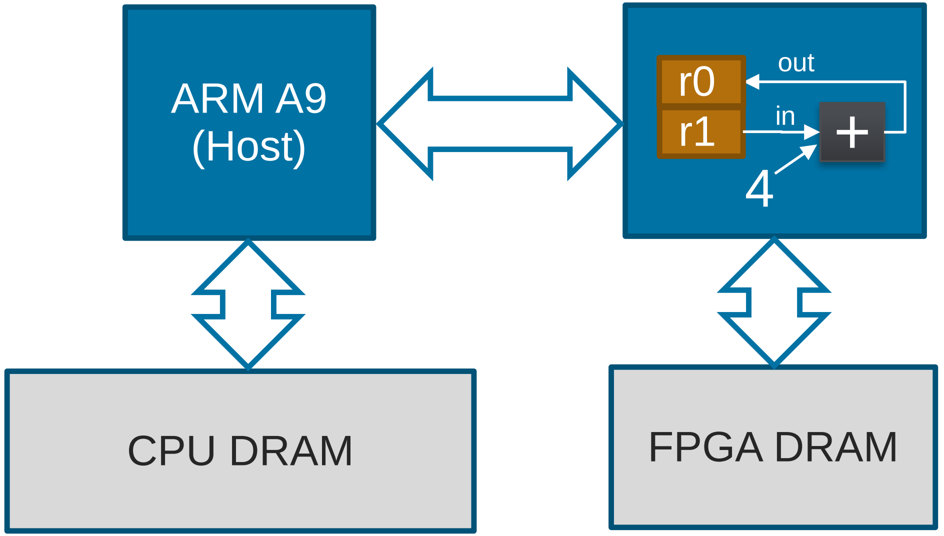
- Add '4' to an input integer



```
// FPGA Code: Verilog
// 1. Write addition module
module add4(
  input wire[31:0] in,
  output wire[31:0] out
);
  assign out = in + 31'h4;
endmodule
```

Simple Example: ARM + FPGA

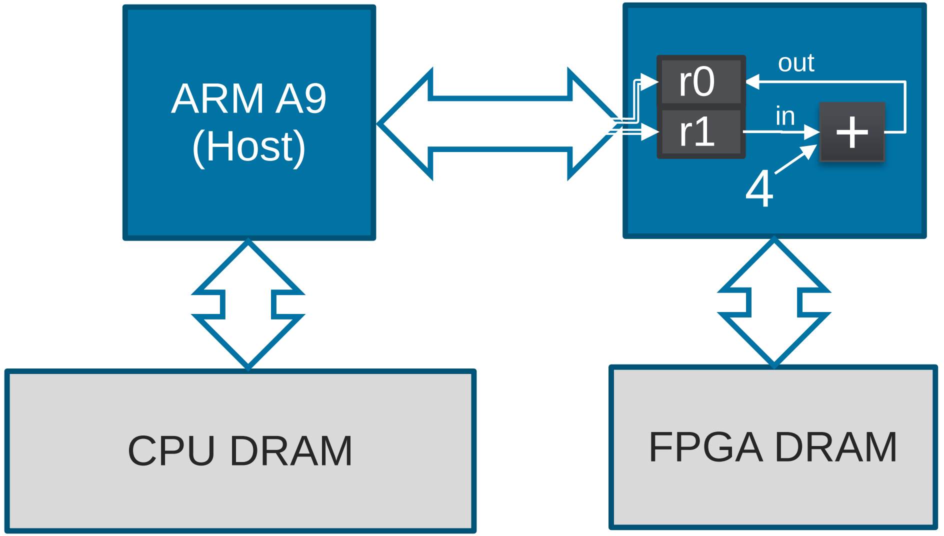
- Add '4' to an input integer



```
// FPGA Code: Verilog
// 1. Write addition module
// 2. Create registers
regFile #(2) rf(…)
```

Simple Example: ARM + FPGA

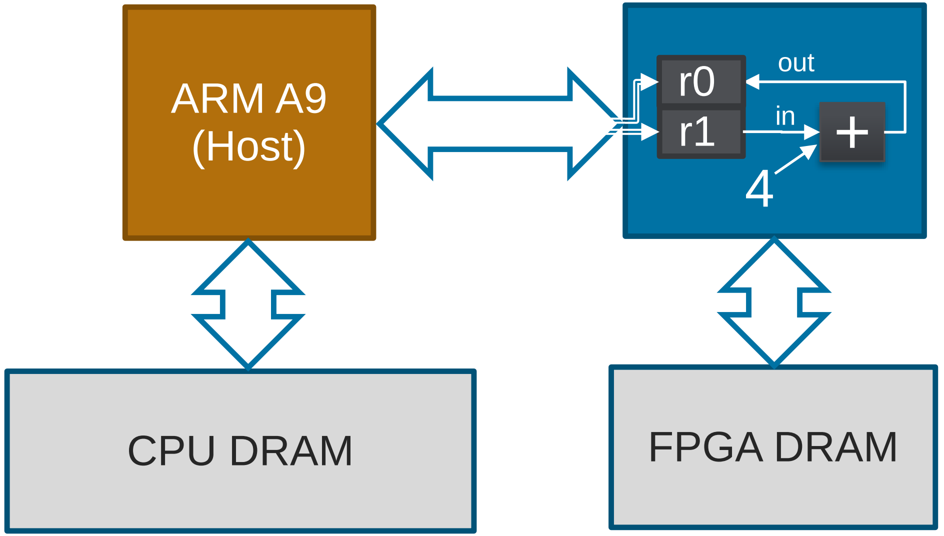
- Add '4' to an input integer



```
// FPGA Code: Verilog
// 1. Write addition module
// 2. Create registers
// 3. Connect to host bus
```

Simple Example: ARM + FPGA

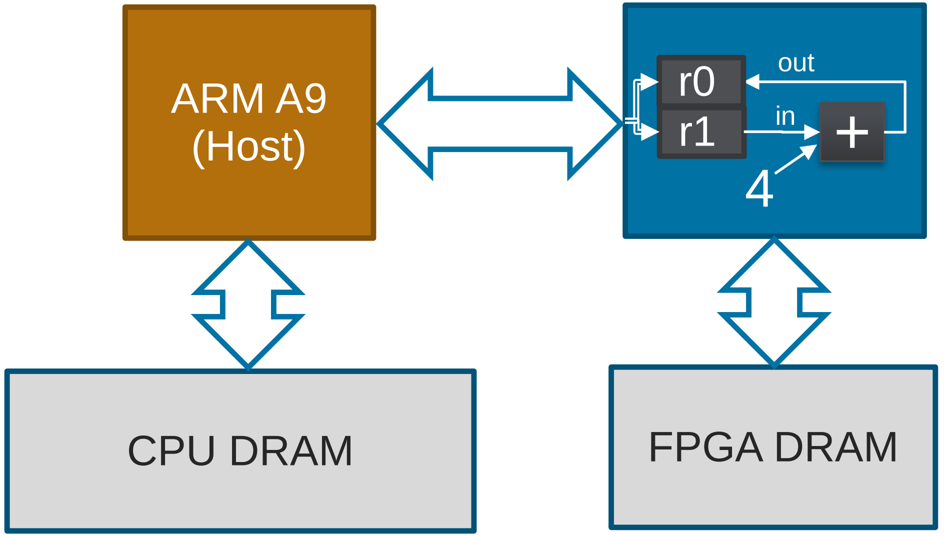
- Add '4' to an input integer



```
// FPGA Code: Verilog
// 1. Write addition module
// 2. Create registers
// 3. Connect to host bus

// Host Code: C
int main(int argc, char **argv)
{
  int in = atoi(argv[1]);
  int out = 0;

  // ???

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

Simple Example: ARM + FPGA

- Add '4' to an input integer

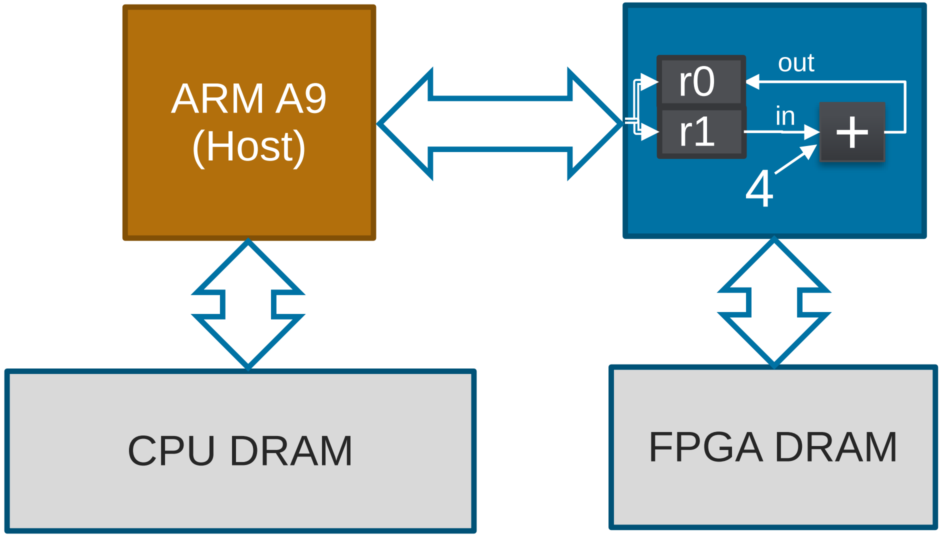


```
// FPGA Code: Verilog
// 1. Write addition module
// 2. Create registers
// 3. Connect to host bus
```

```
// Host Code: C
int main(int argc, char **argv)
{
  int in = atoi(argv[1]);
  int out = 0;
  // Set up MMIO for r0, r1
  // Configure FPGA bitstream
  // Write 'in' to r1
  // Write '1' to command reg
  // Wait until FPGA status=1
  // Read r0 to 'out'
  printf("Output: %d\n", out);
  return 0;
}
```

Simple Example: ARM + FPGA

- Add '4' to an input integer



```
// FPGA Code: Verilog
// 1. Write addition module
// 2. Create registers
// 3. Connect to host bus

// Host Code: C
int main(int argc, char **argv)
{
   int in = atoi(argv[1]);
   int out = 0;
   init_fpga();
   config_fpga("add4.bin")
   write_to_fpga(1, in);
   run_fpga();
   read_from_fpga(0, &out);
   printf("Output: %d\n", out);
   return 0;
}
```

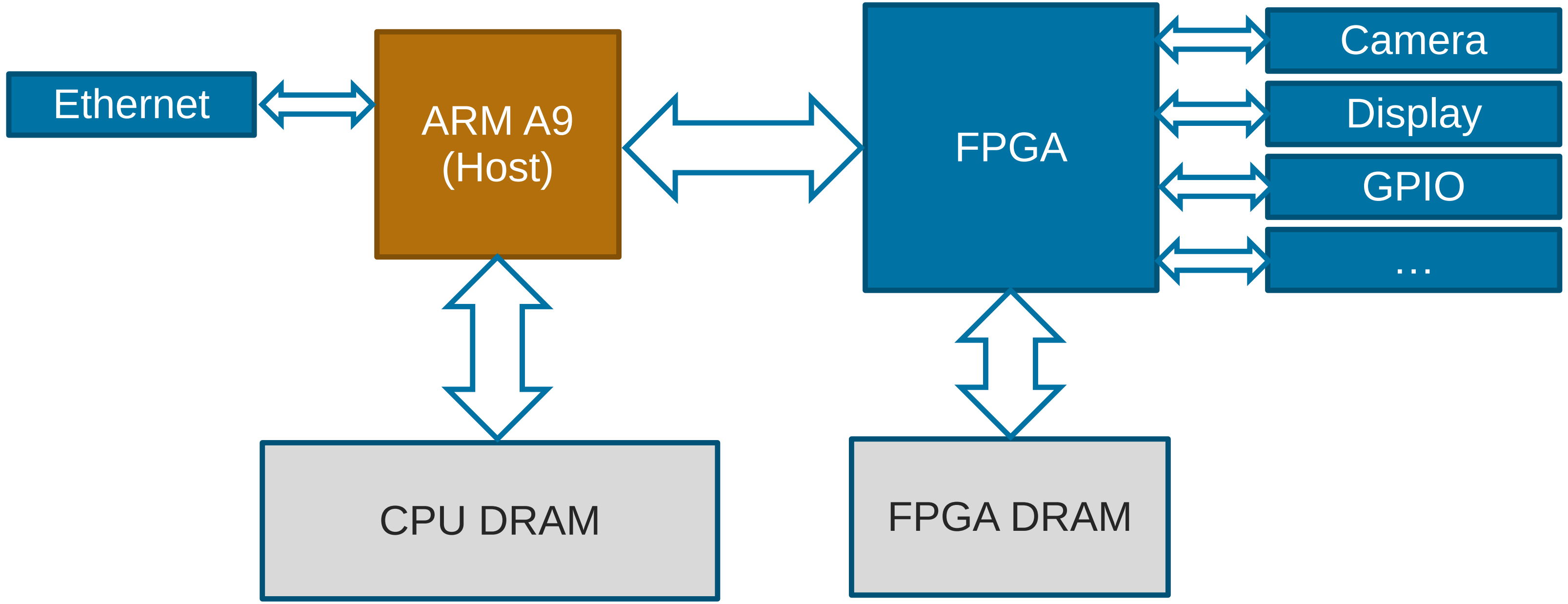
Simple Example: In a Nutshell

- Program starts on host, offloaded to accelerator
  - Interaction with host and peripherals
- Hardware data and control path design
  - Take advantage of parallelism and locality
  - Hierarchical Pipelining, on-chip memory banking, double buffering…
- Verification
- Requires understanding hw-sw interaction

Simple Example: Excluded details

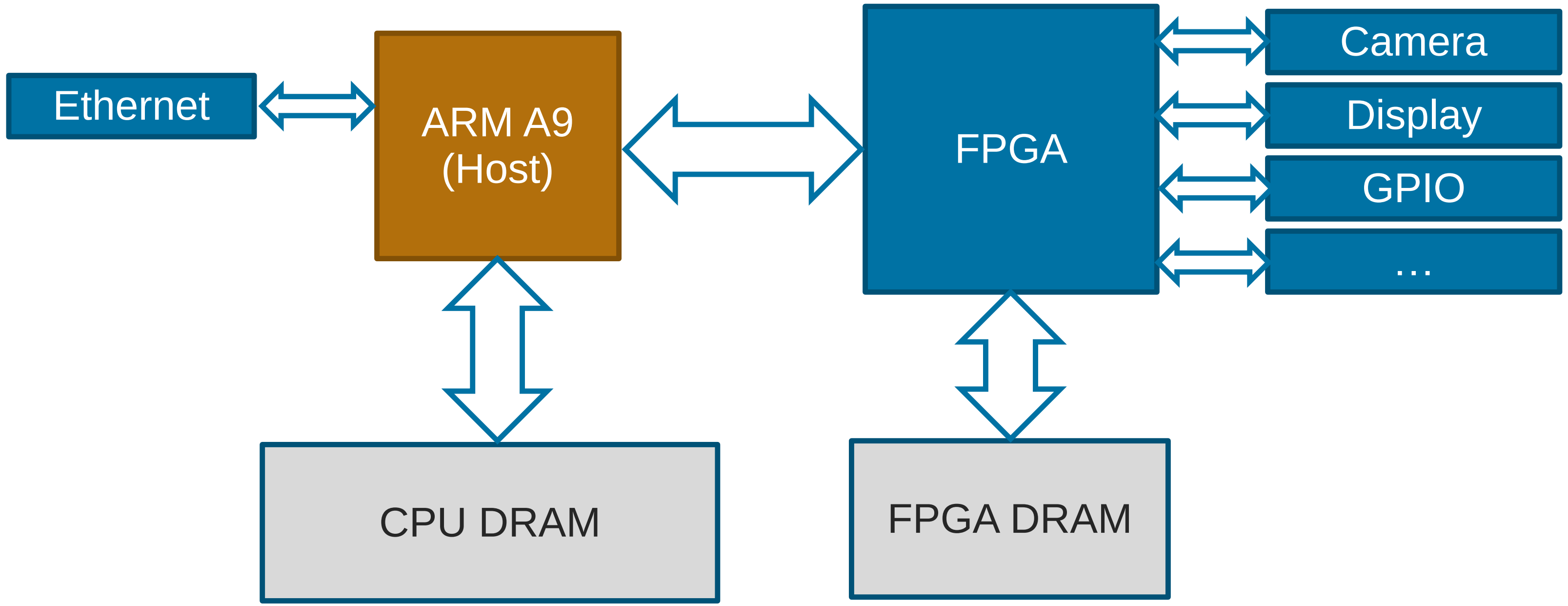
- How to test/verify code?
  - Logic synthesis takes minutes/hours to run, impractical for iterative code-run-debug cycles
  - How to write simulation testbenches?
- Low-level details
  - Instantiating reset controllers and other supporting IP
  - Implementing software APIs

Reality Is More Complicated



Reality Is More Complicated

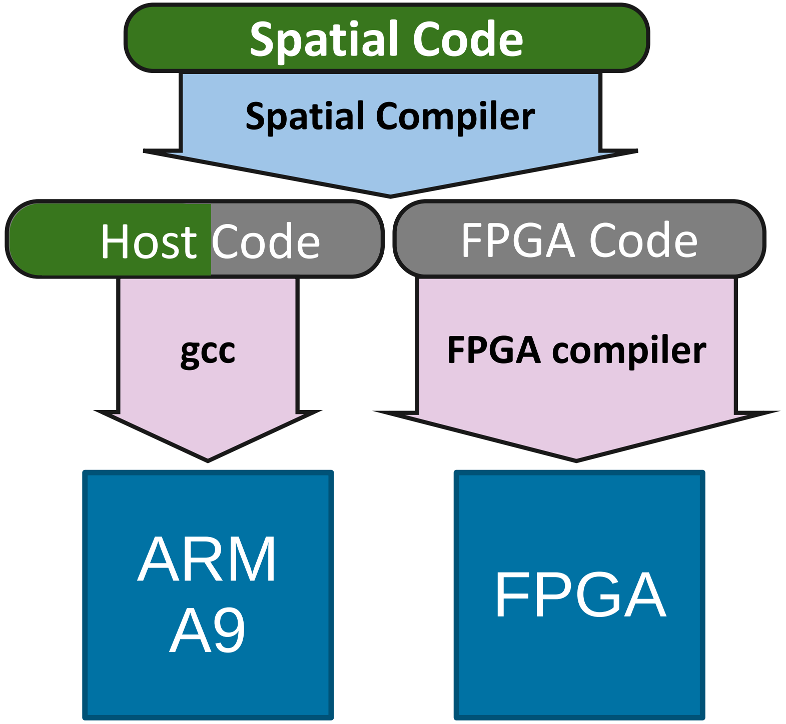
- **Hardware Video decoder + UDP ?**



Introducing Spatial

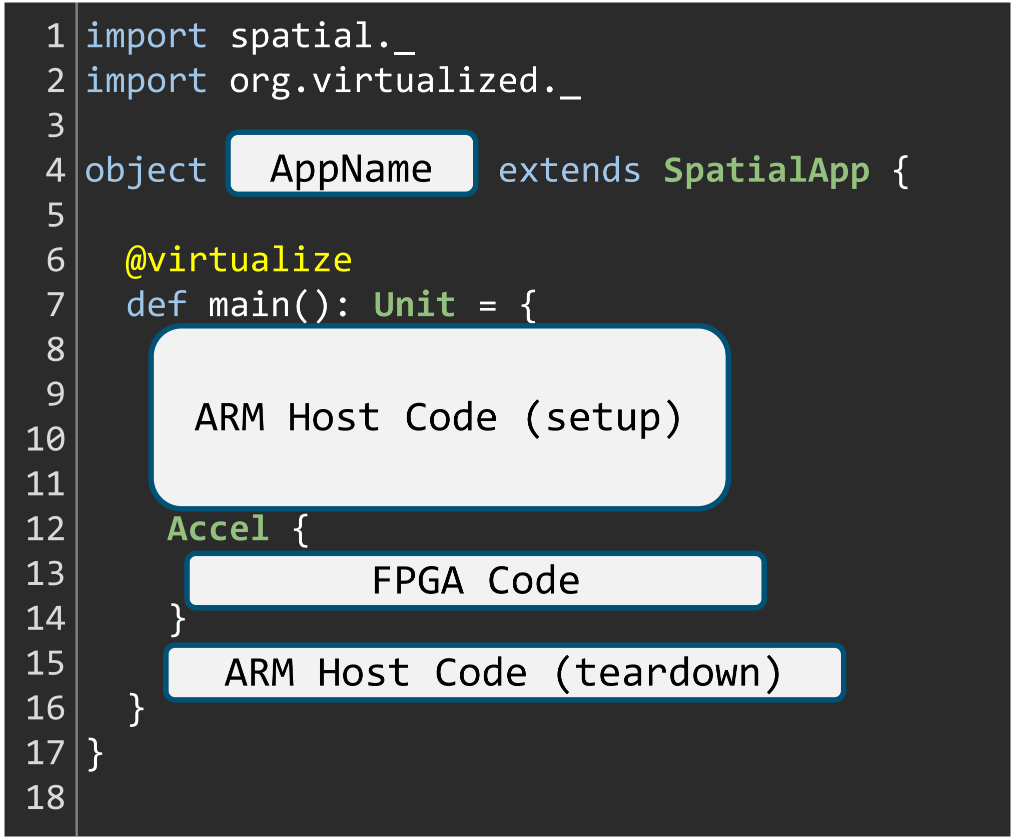
- Programming language to simplify accelerator design
  - Simple APIs to manage Host <-> FPGA communication
  - Peripherals exposed as streams i.e.: data, ready, valid
  - In-built constructs to express parallel datapaths, on-chip memories etc
  - Automatic functional and cycle-accurate simulation
- Focus on “interesting stuff” aka accelerator datapath and control design

Spatial Model



SPATIAL PROGRAMMING BASICS

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!

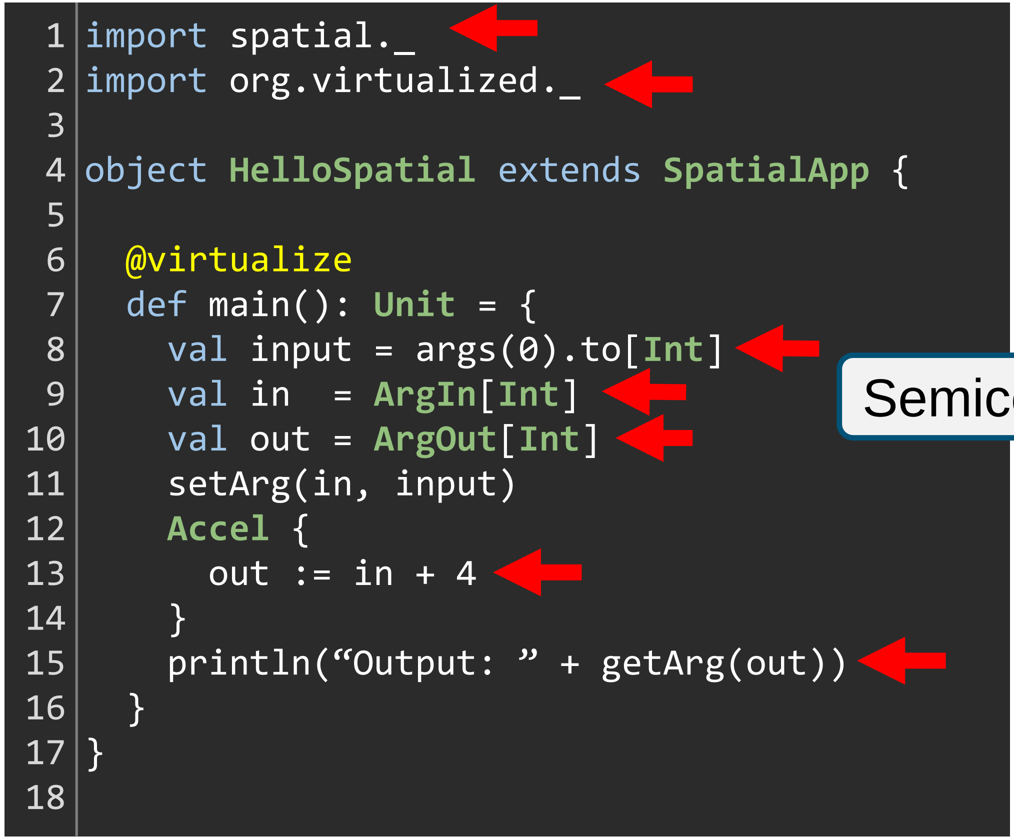
```
import spatial._
import org.virtualized._

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

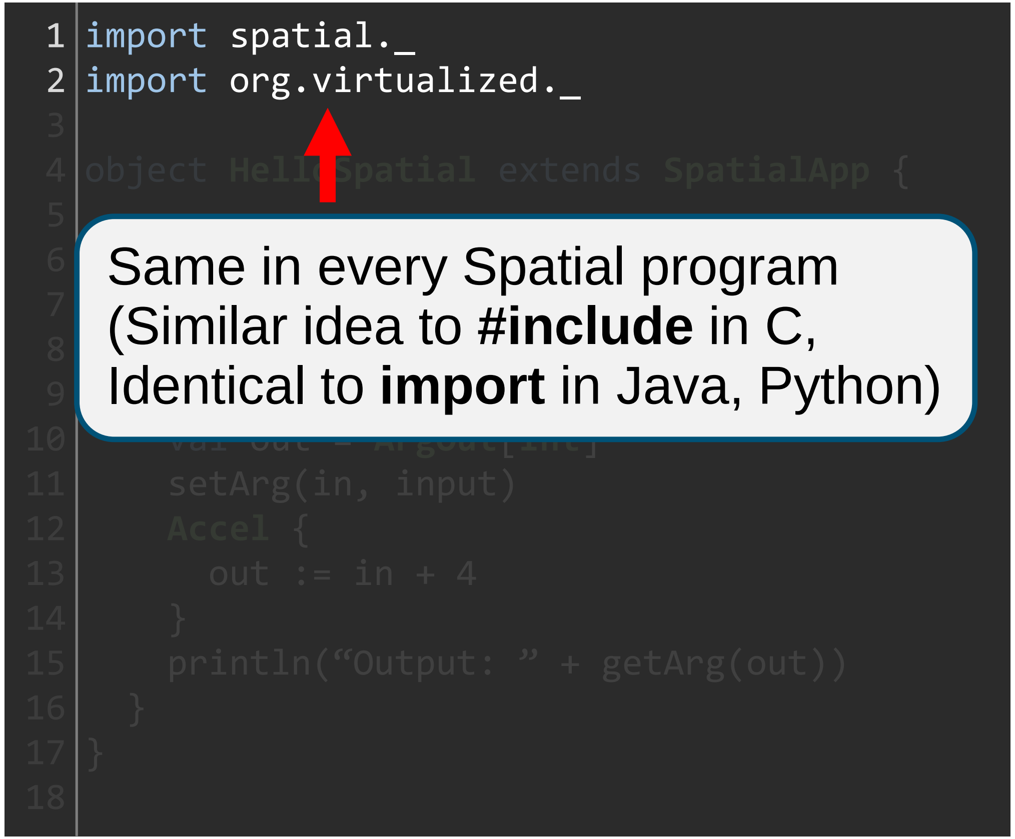
Spatial is Embedded in **Scala**



```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

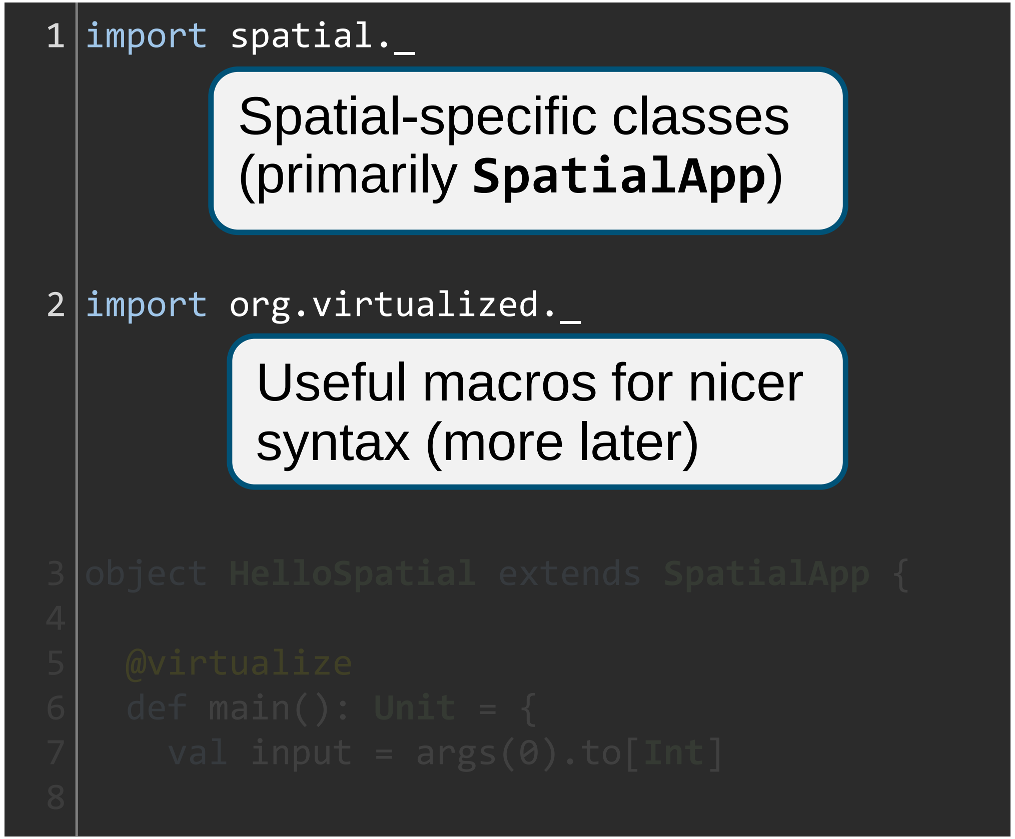
Import Statements



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

object HelloSpatial extends SpatialApp {





    var 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._
```

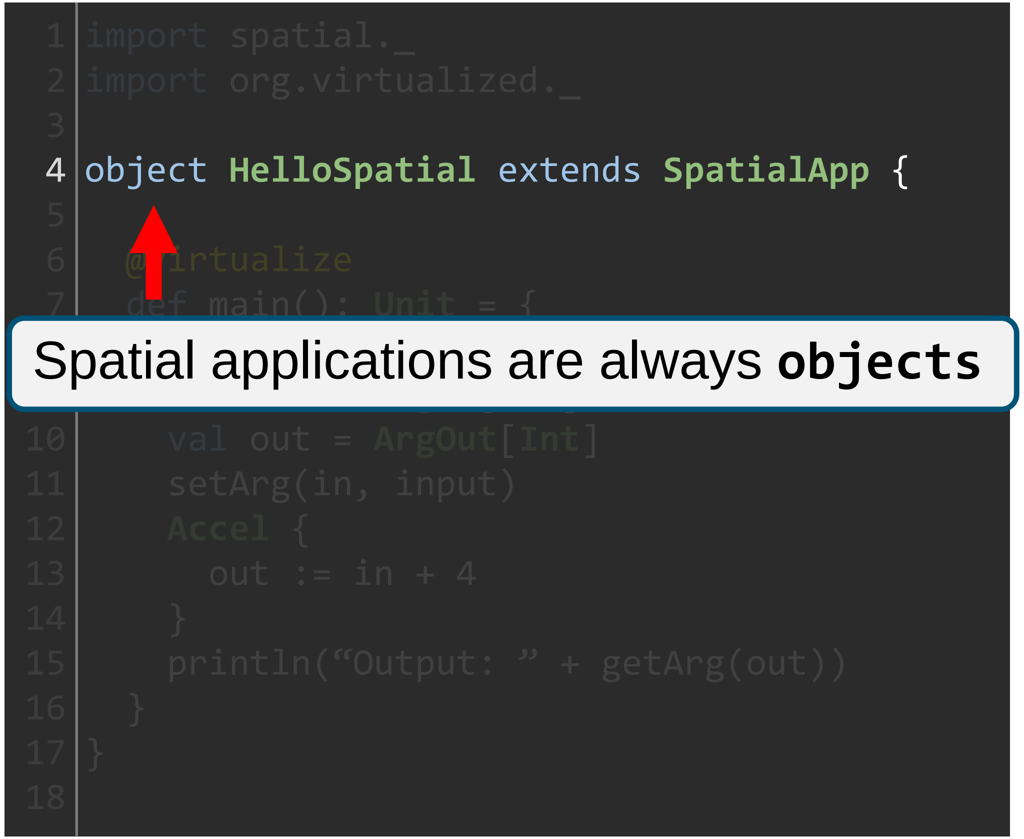
Spatial-specific classes (primarily **SpatialApp**)

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

Useful macros for nicer syntax (more later)

```
object HelloSpatial extends SpatialApp {

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

```

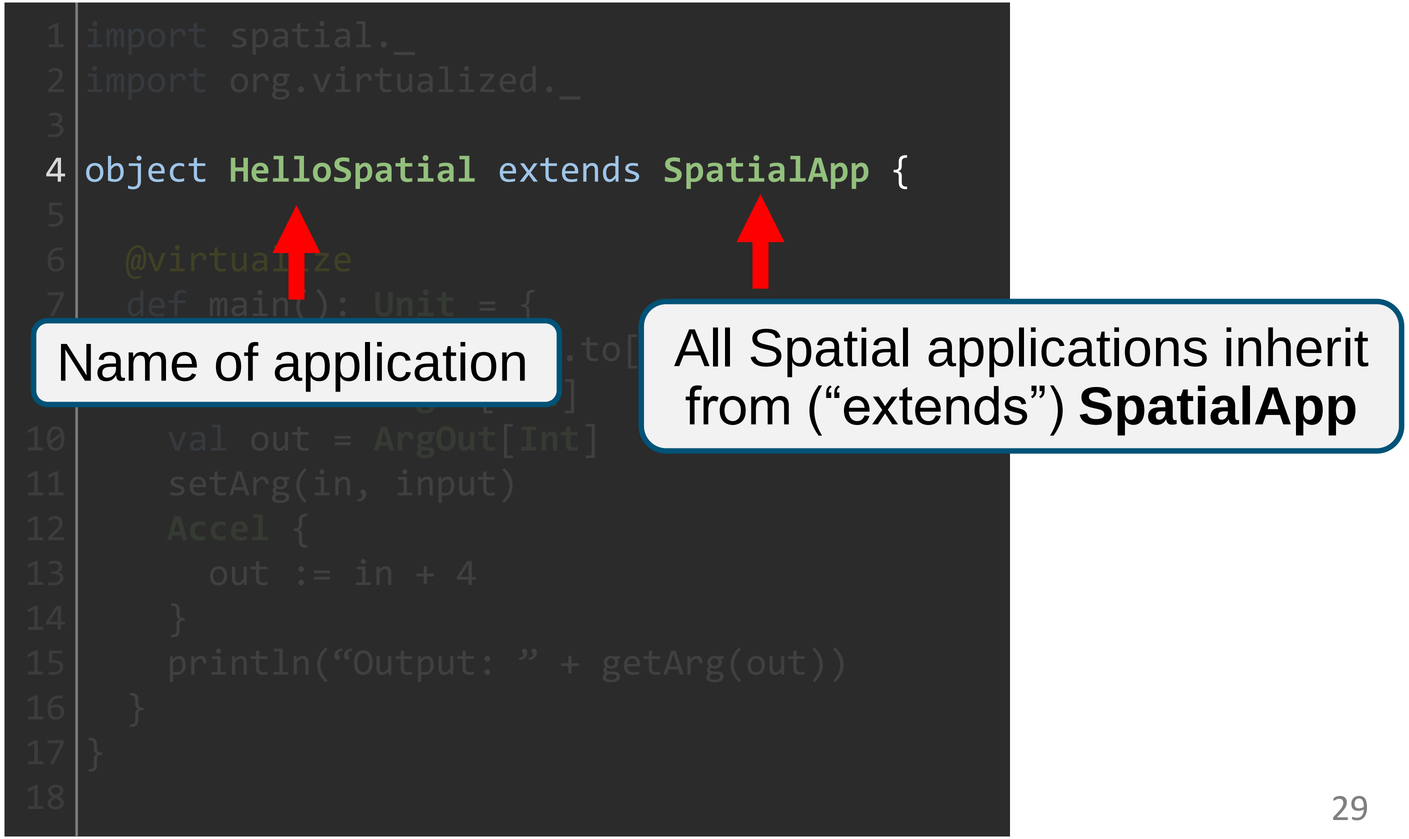
Application Object Declaration



```
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))
  }
}
```

Spatial applications are always **`objects`**

Application Object Declaration

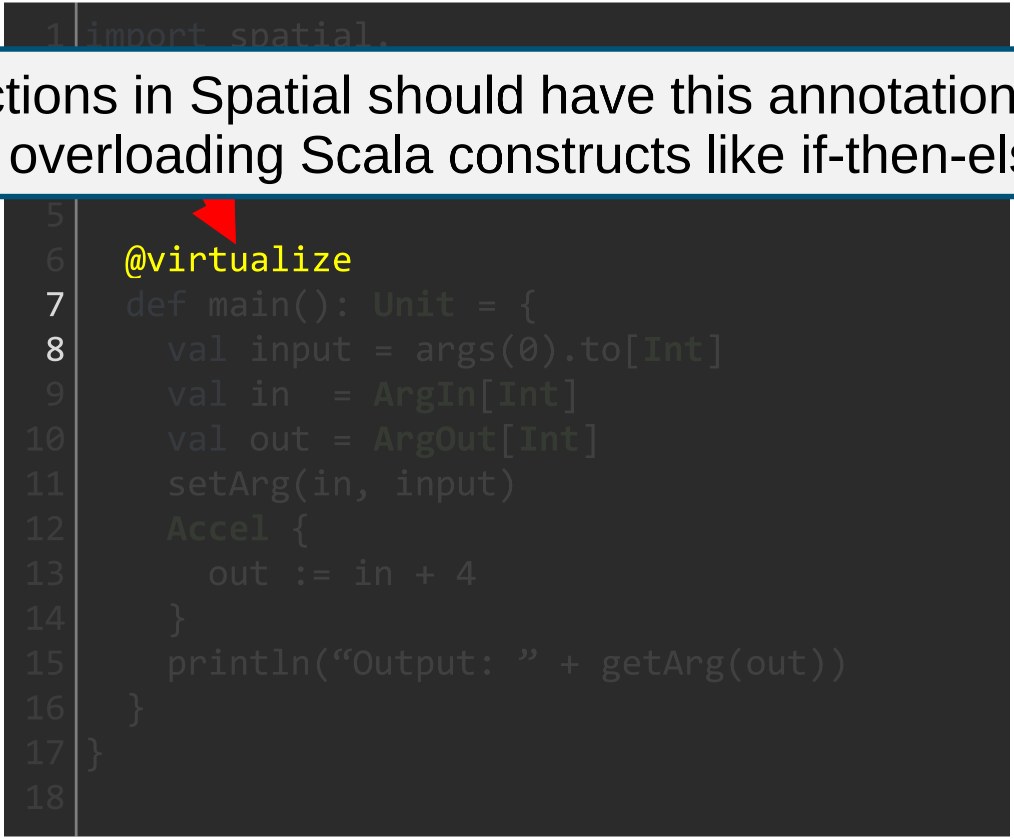


```
import spatial._
import org.virtualized._

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**

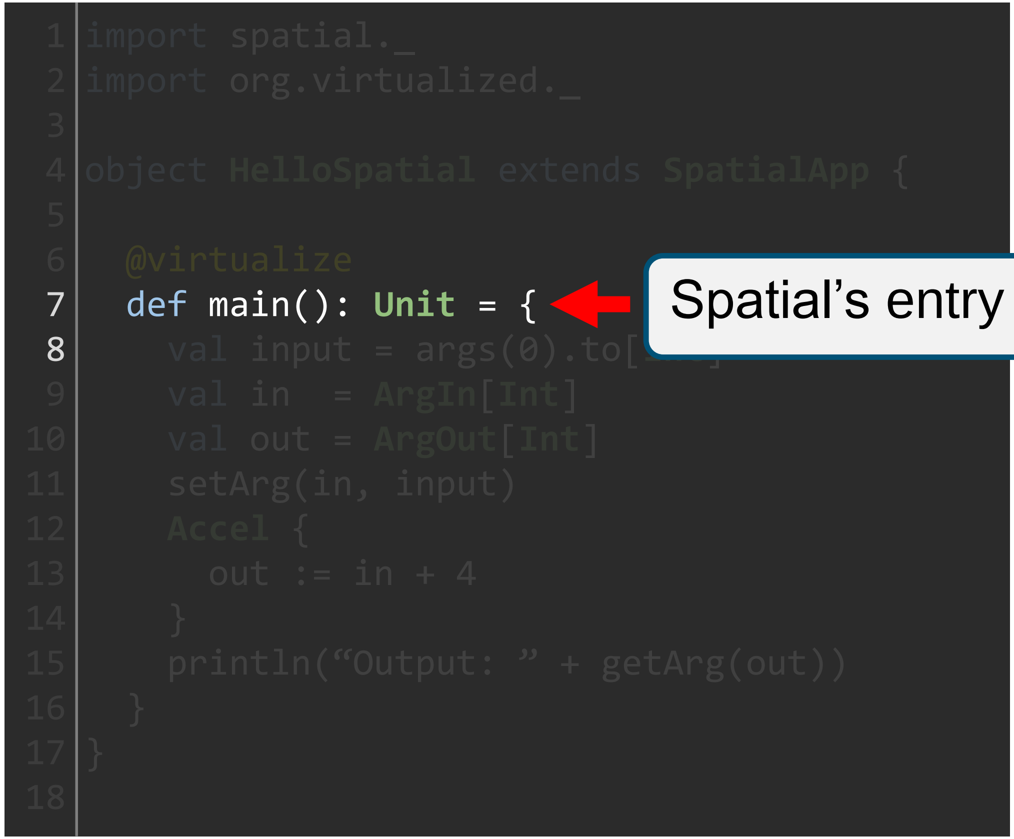
“@virtualize” Annotation

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



```
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))
  }
}
```

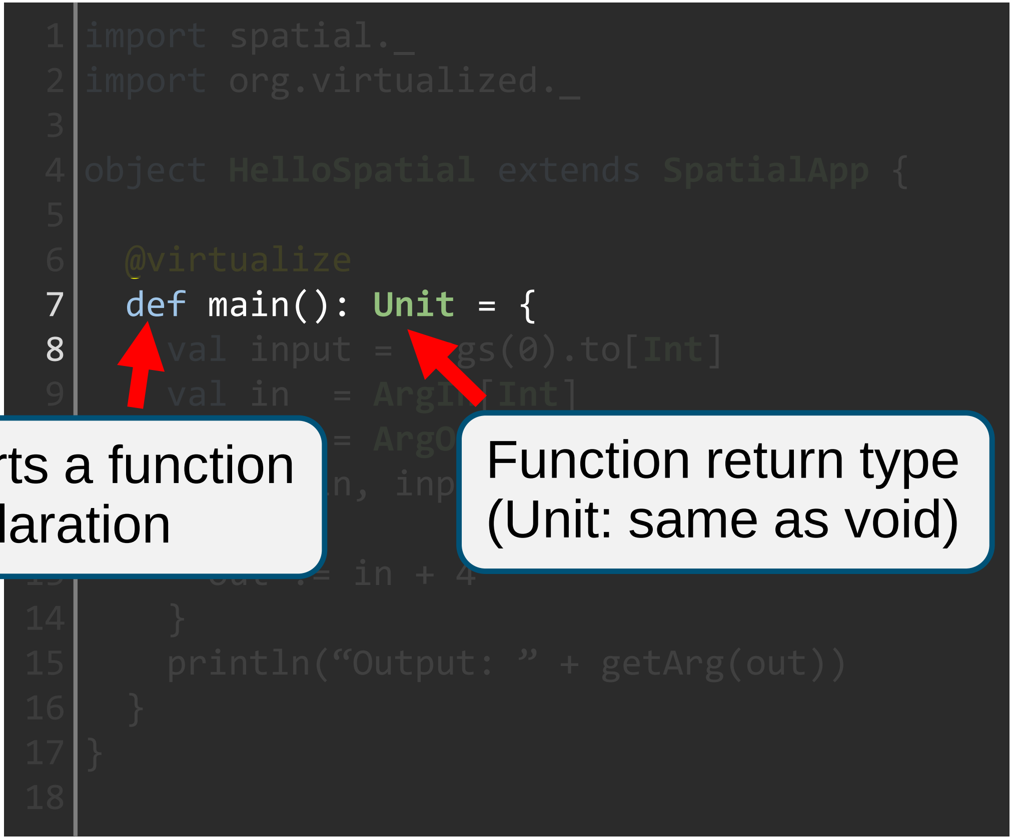
Spatial’s Entry Function: “main()”



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

object HelloSpatial extends SpatialApp {

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

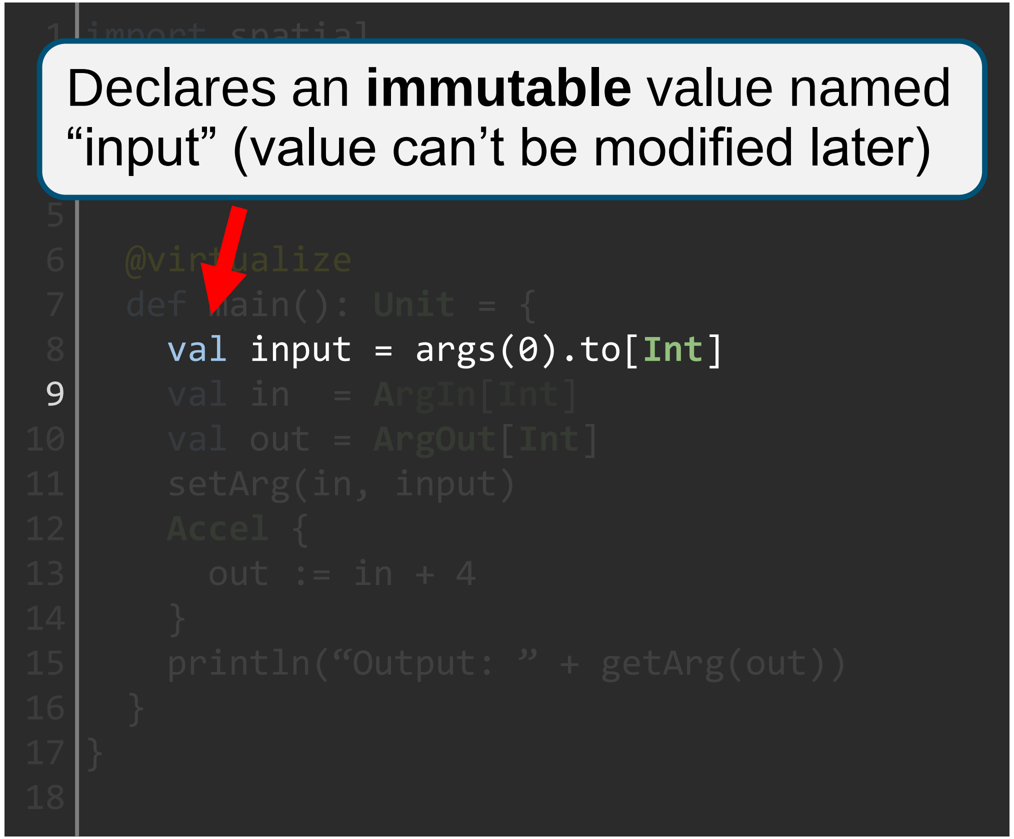


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {

  @virtualize
  def main(): Unit = {
    val input =    gs(0).to[Int]
    val in   = ArgI  [Int]
             = ArgO
           n, inp

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

Starts a function declaration

Function return type (Unit: same as void)

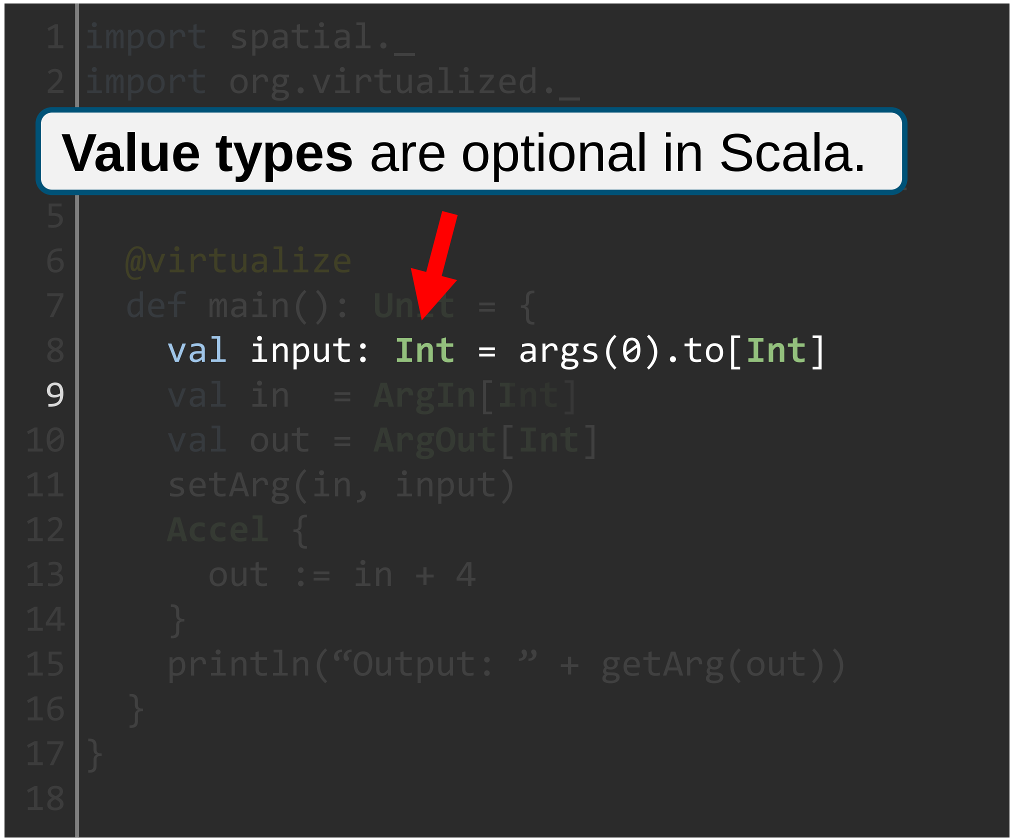
Val Definitions



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))
  }
}
```

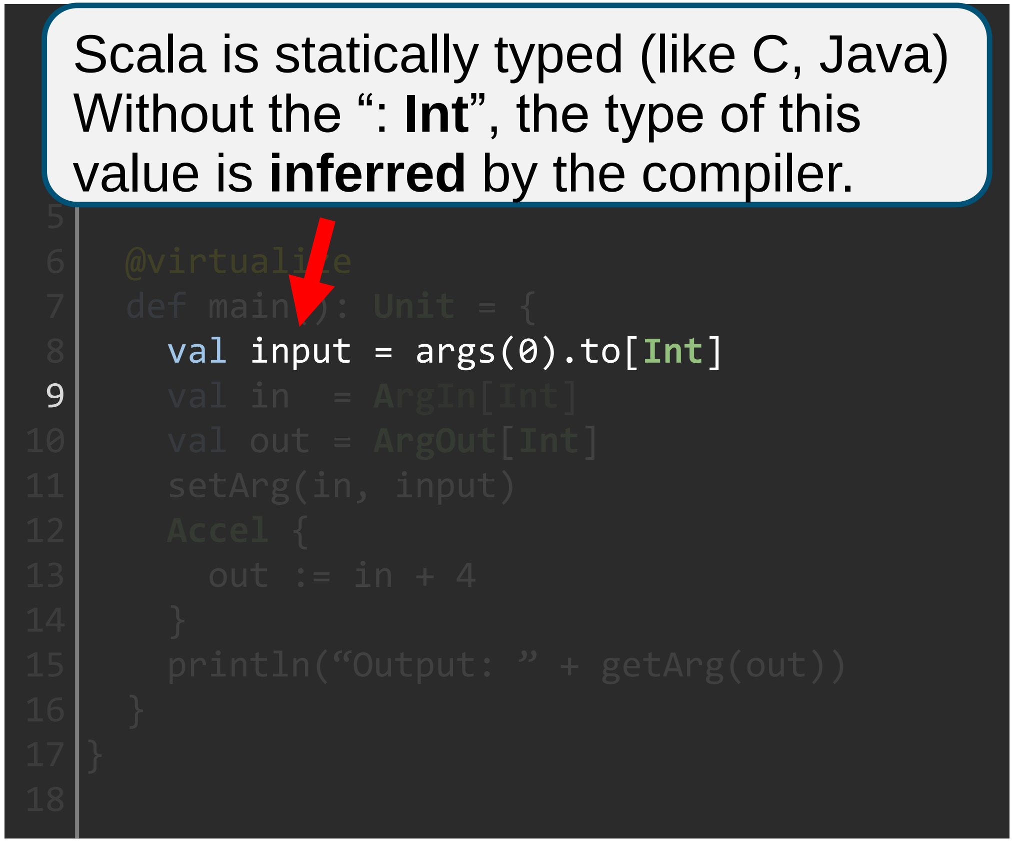
Val Definitions



```
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

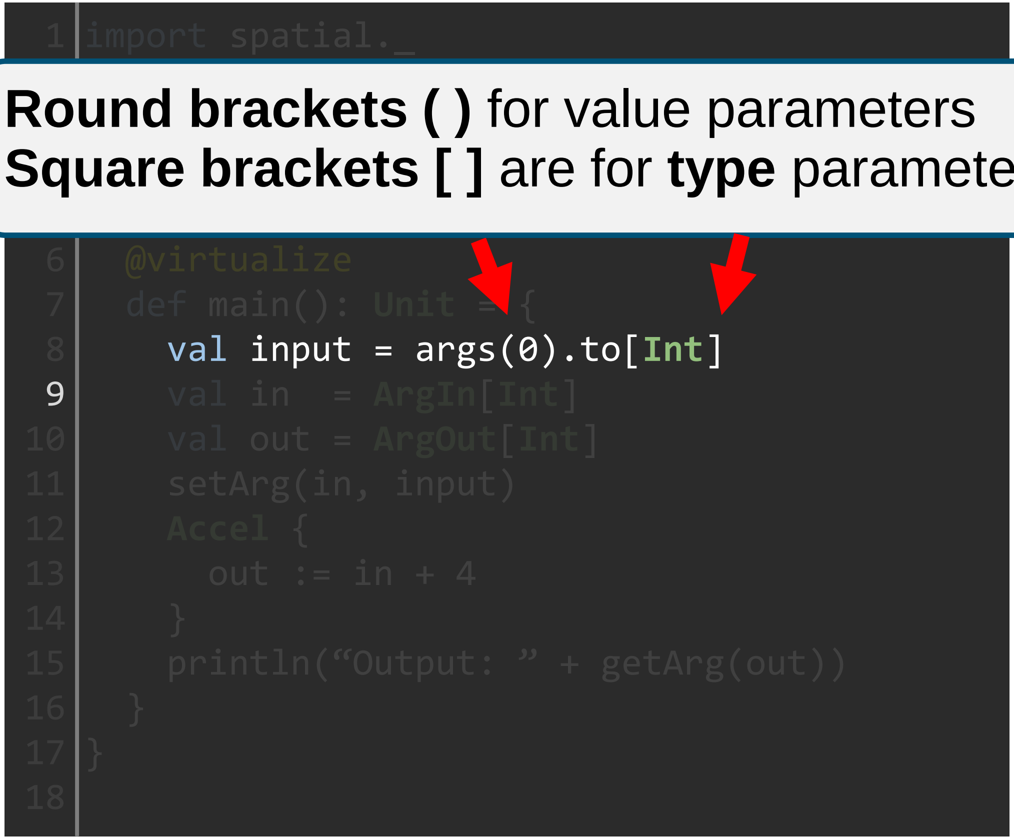


Scala is statically typed (like C, Java)
Without the ": **Int**", the type of this value is **inferred** by the compiler.

```
  @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))
  }
}
```

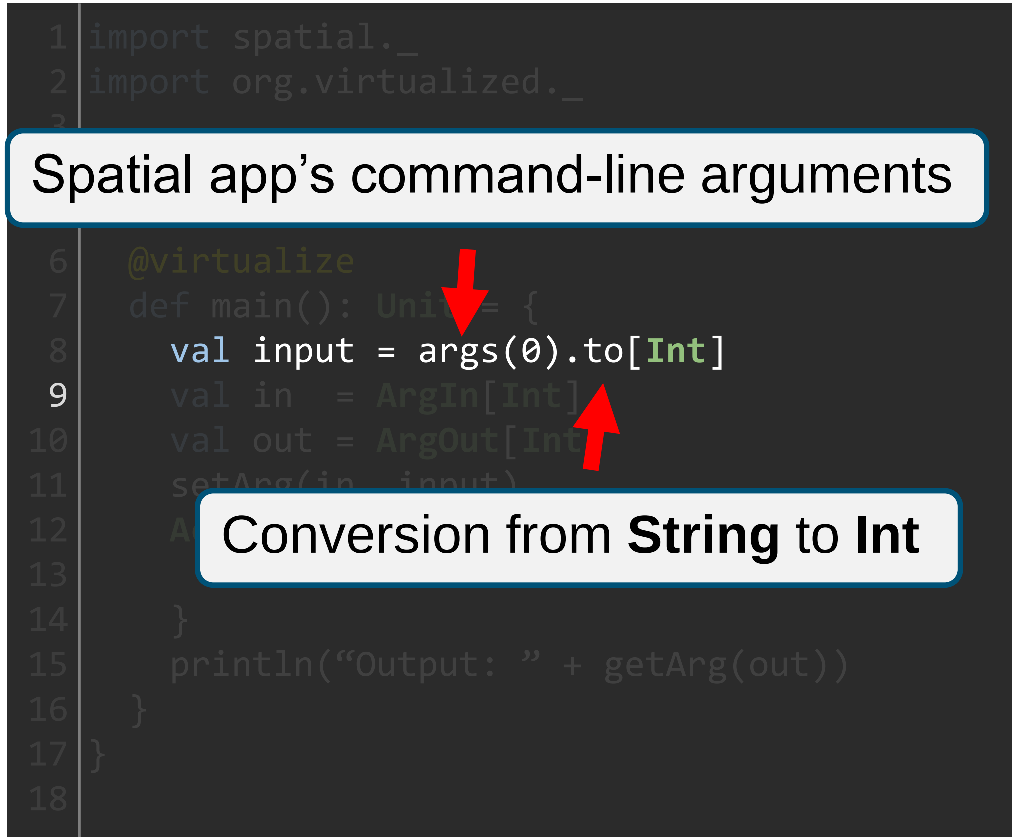
Method Calls

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



```
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))
  }
}
```

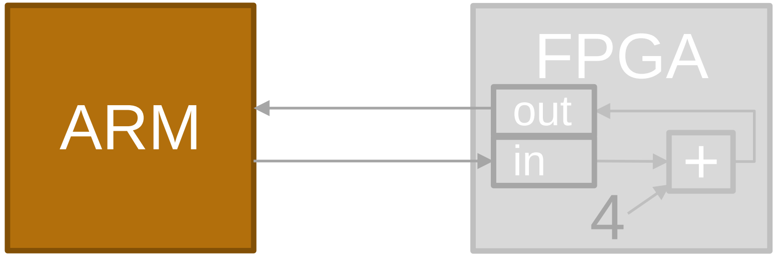
Spatial Command-Line Arguments



```
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)
    A

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

Spatial app's command-line arguments

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

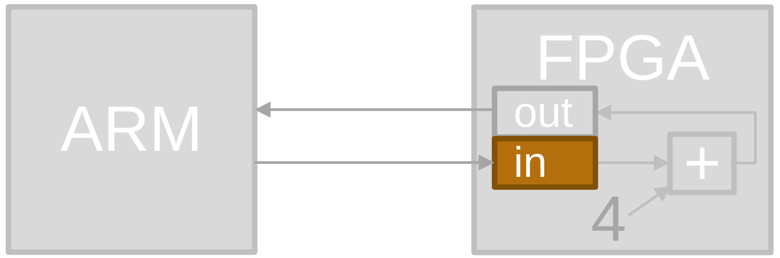


```
int main(int argc, char **argv) {
  int in = atoi(argv[1]);

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

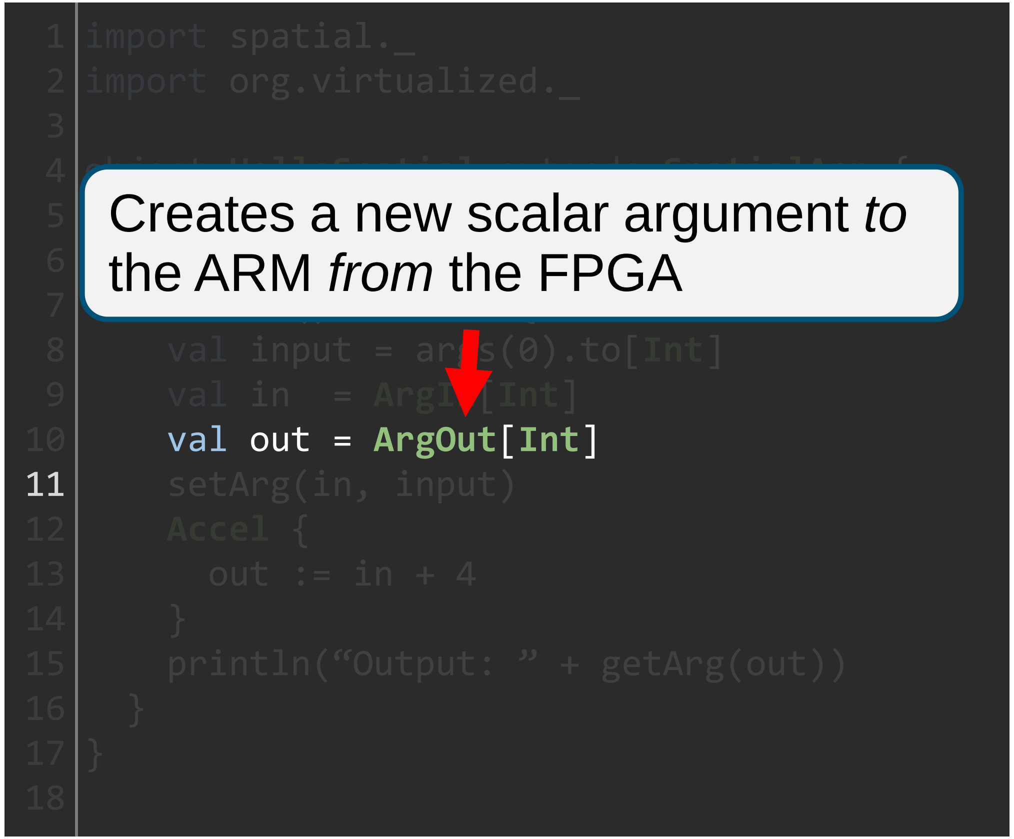
Input Arguments (ArgIn)

```
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

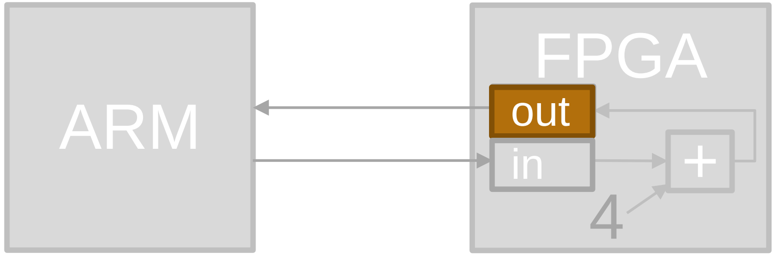


Output Arguments (ArgOut)



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


    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

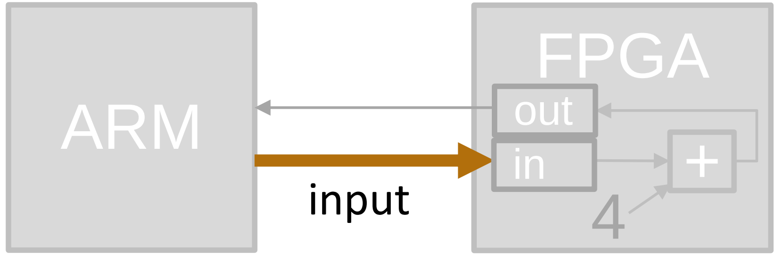


Scalar Transfers (ARM → FPGA)

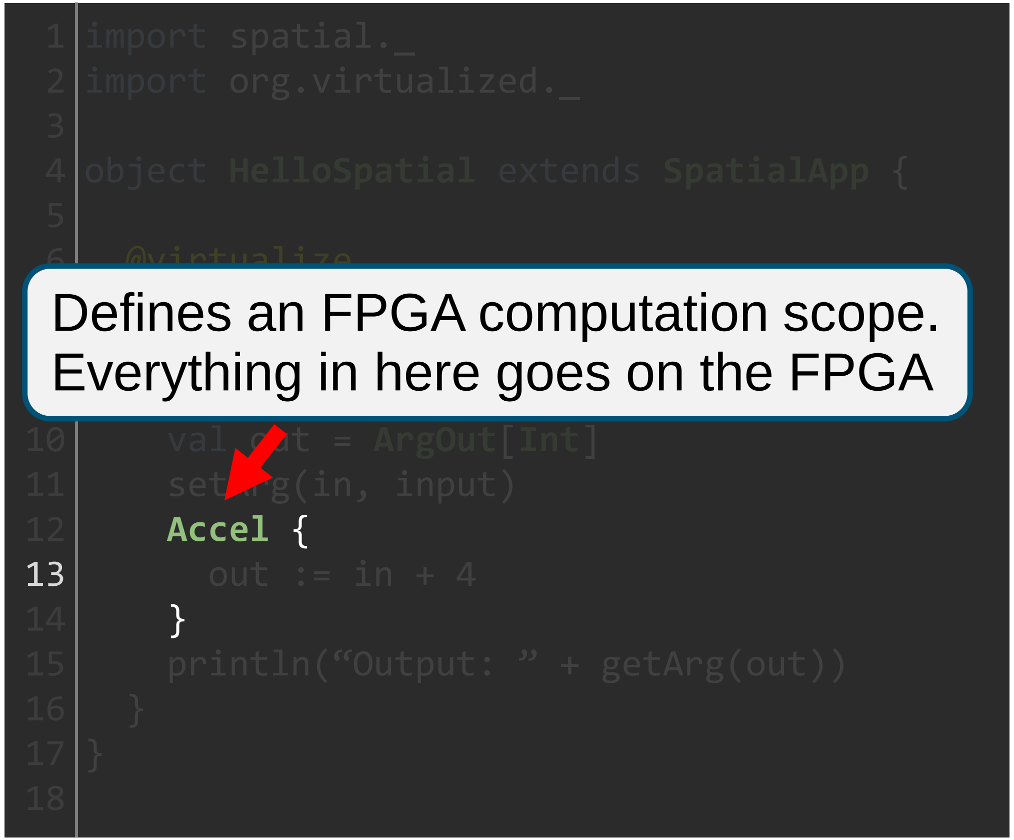
```
import spatial._
import org.virtualized._

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

```
  }
}
```

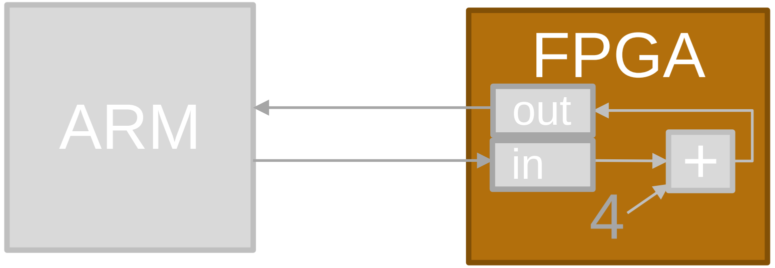


Accel Block



```
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))
  }
}
```

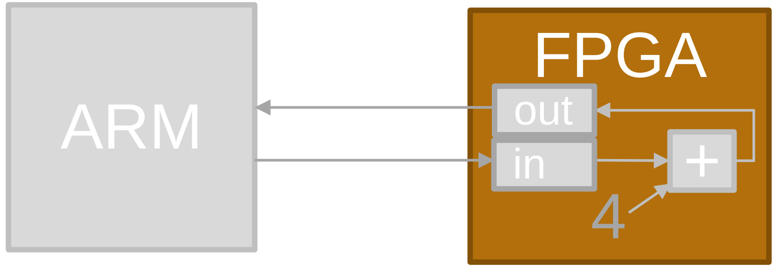
Defines an FPGA computation scope.
Everything in here goes on the FPGA



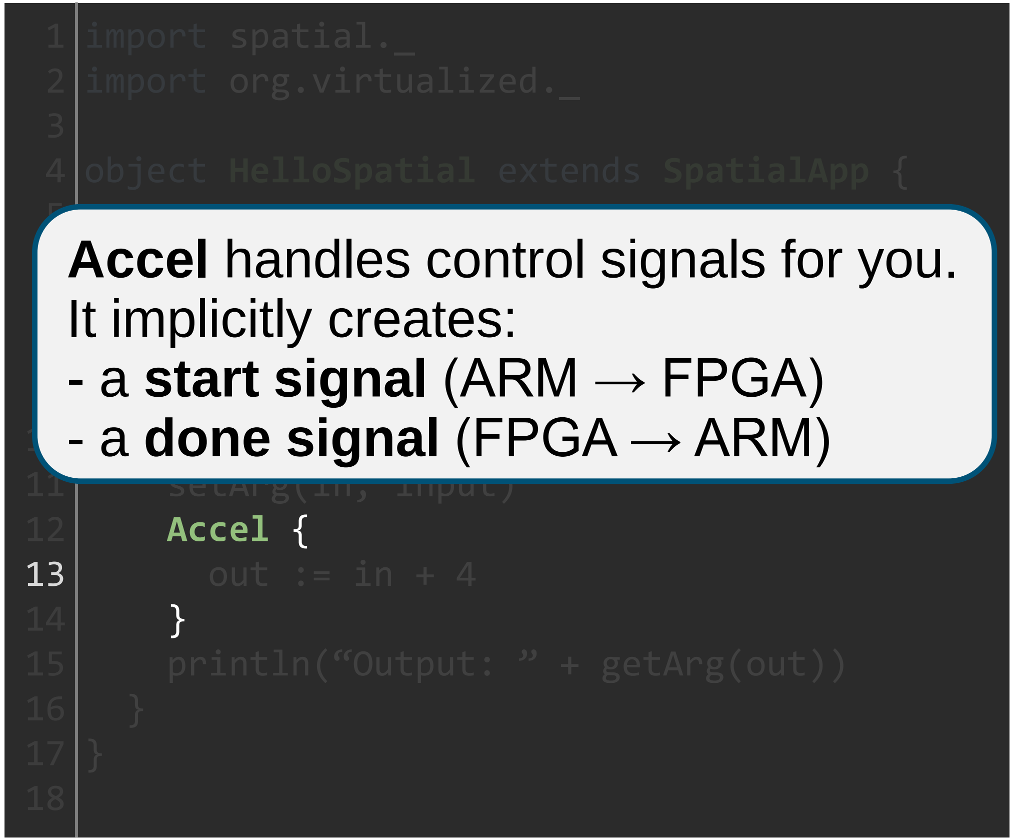
Accel Block

```
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

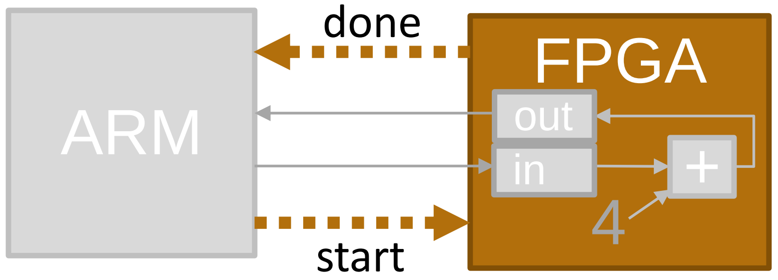


Accel Block



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

object HelloSpatial extends SpatialApp {

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

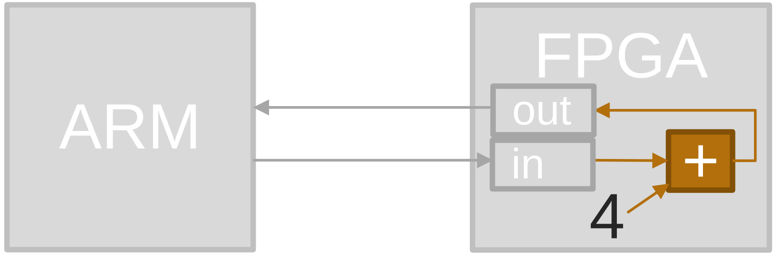
Accel handles control signals for you.
It implicitly creates:
- a **start signal** (ARM → FPGA)
- a **done signal** (FPGA → ARM)



Implicit Register Reads

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

object HelloSpatial extends SpatialApp {

  @virtualize



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

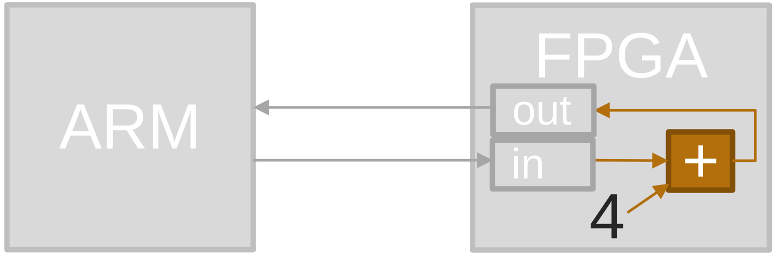
Implicitly creates a wire from the register (ArgIn) **in**



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))
  }
}

```

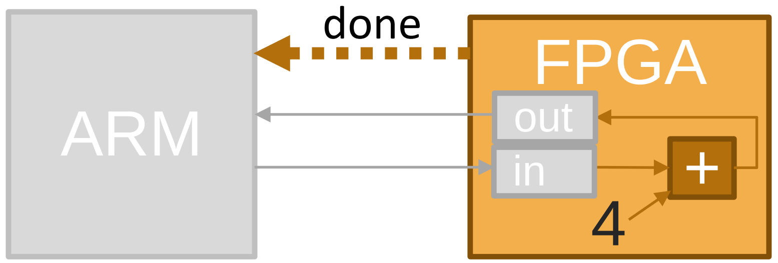
:= creates a write of the value **in + 4** to the register **out**



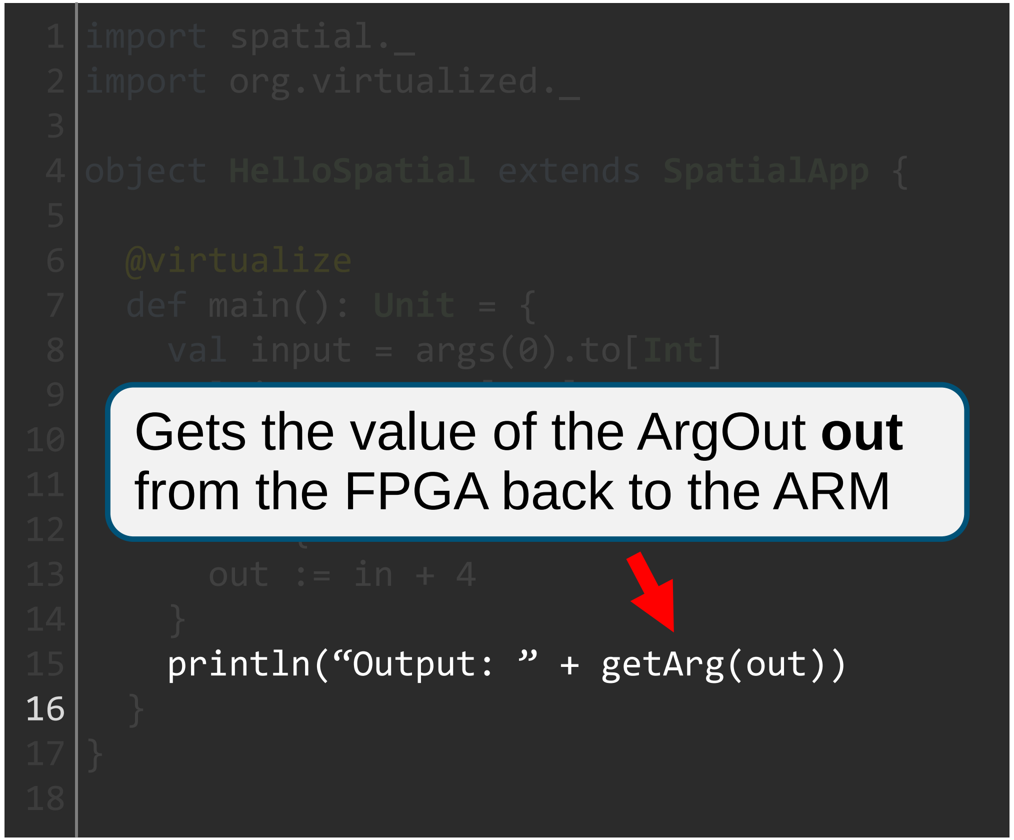
Accel Block Scheduling

```
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

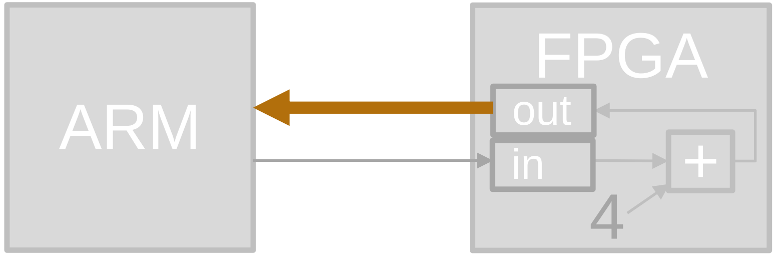


Scalar Transfers (FPGA → ARM)

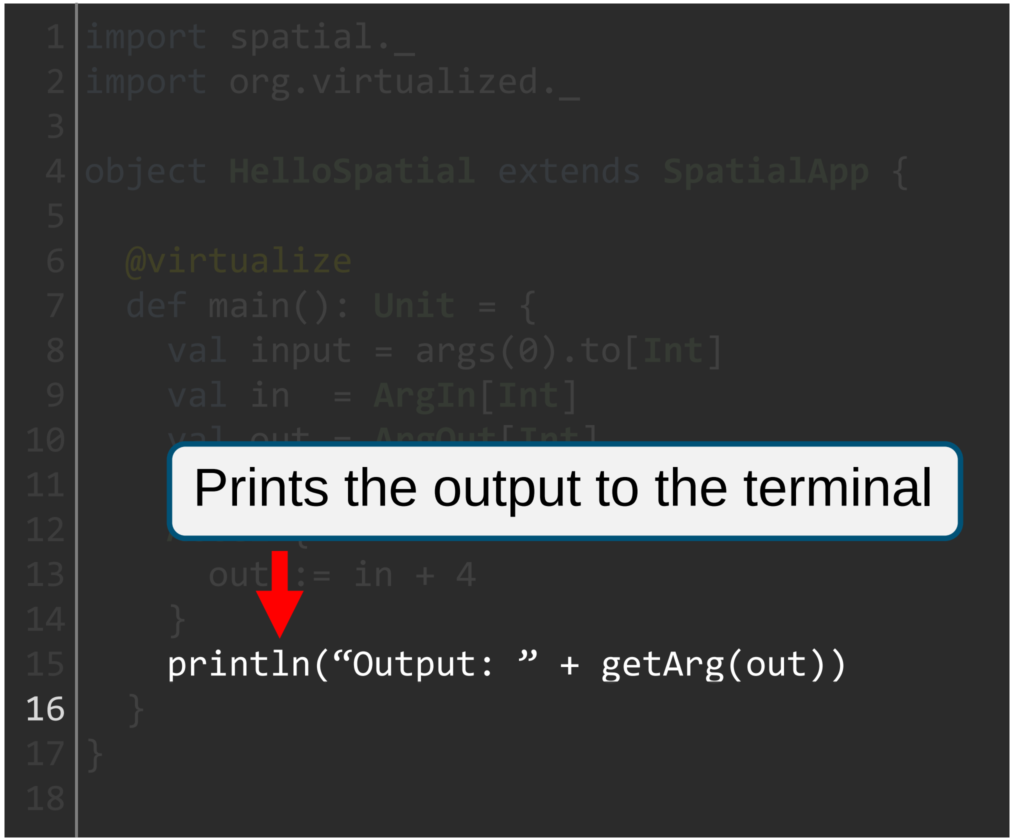


```
import spatial._
import org.virtualized._

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



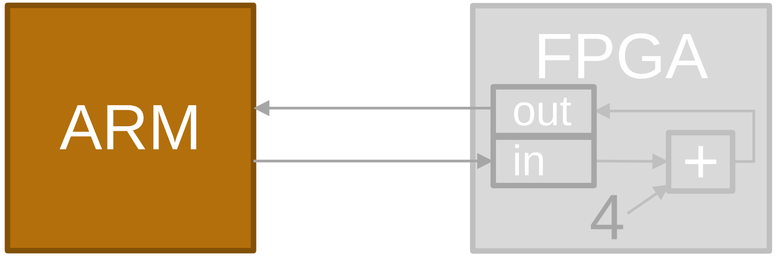
Printing in Spatial**



```
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]

    [illegible] {
      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!

```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))
  }
}

```

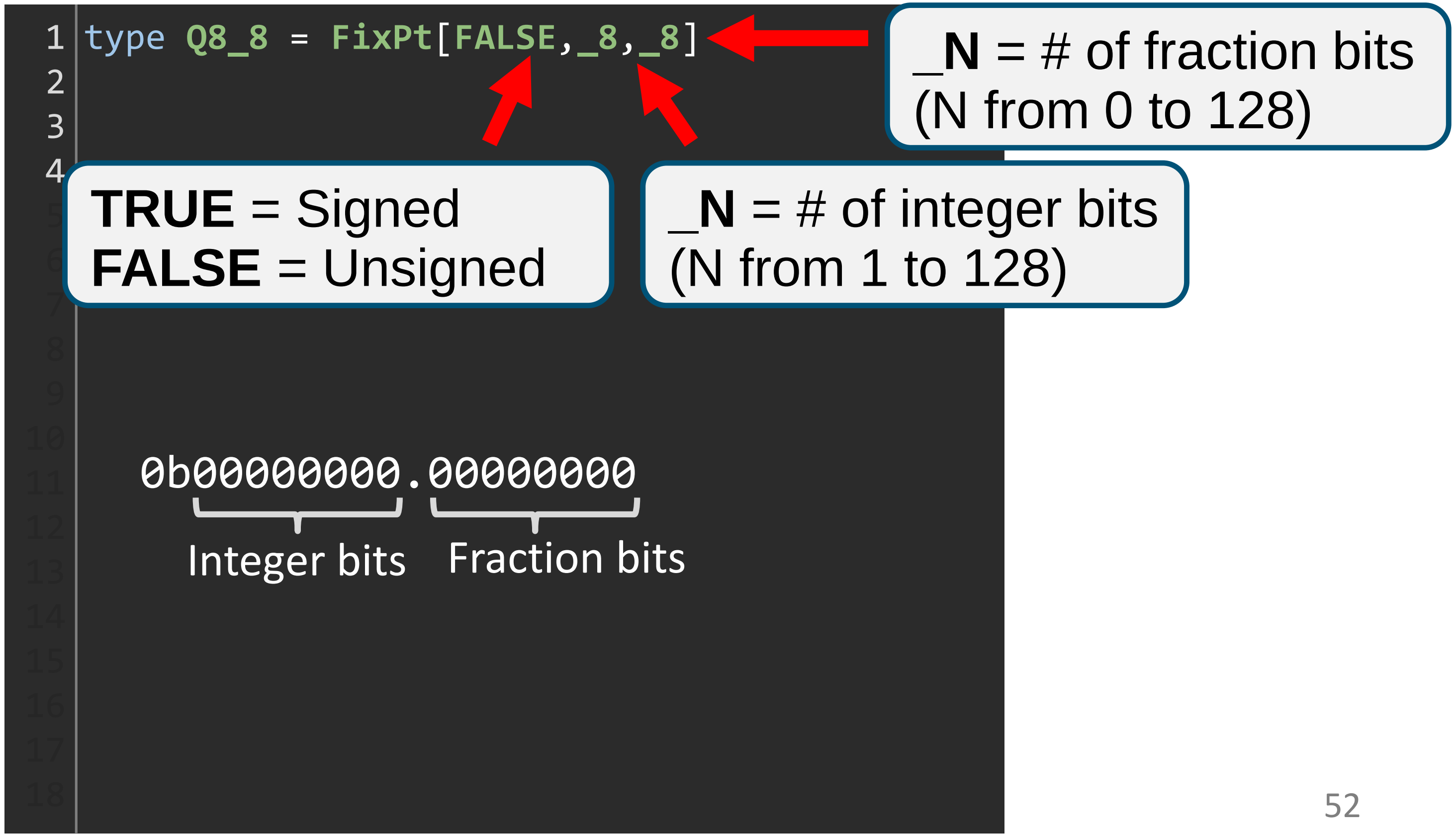
Custom Types in Spatial

- 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]
```

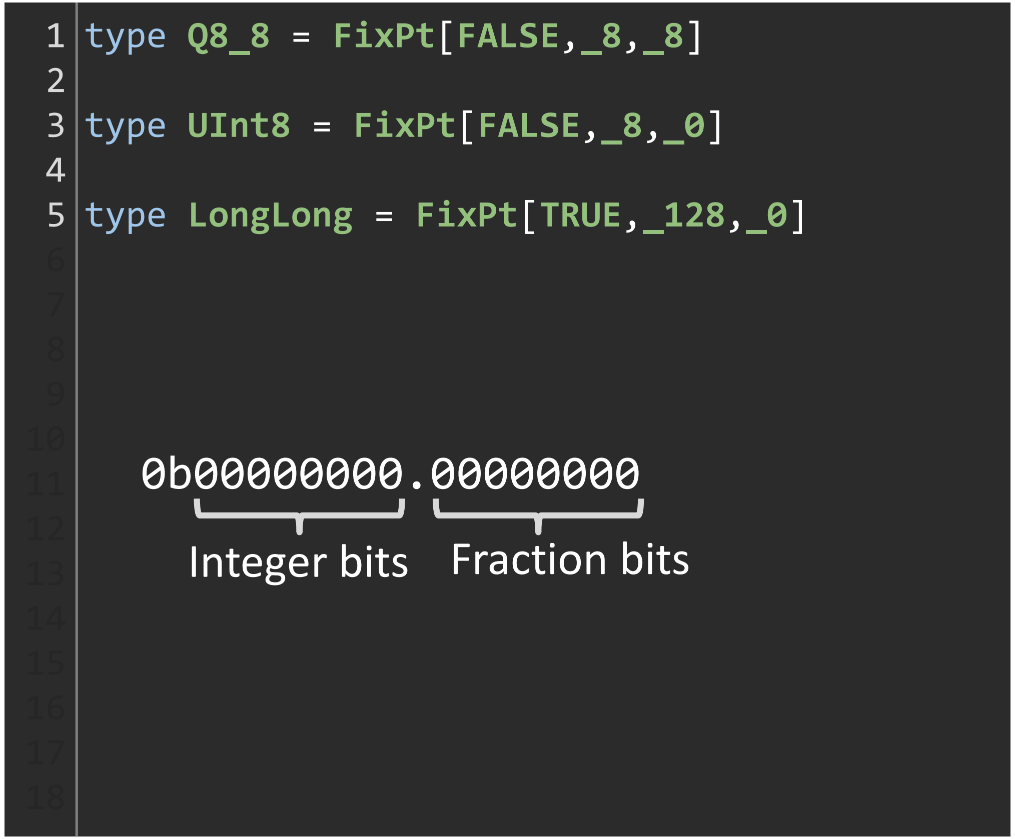
TRUE = Signed
FALSE = Unsigned

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

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

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

Custom Fixed Point Examples



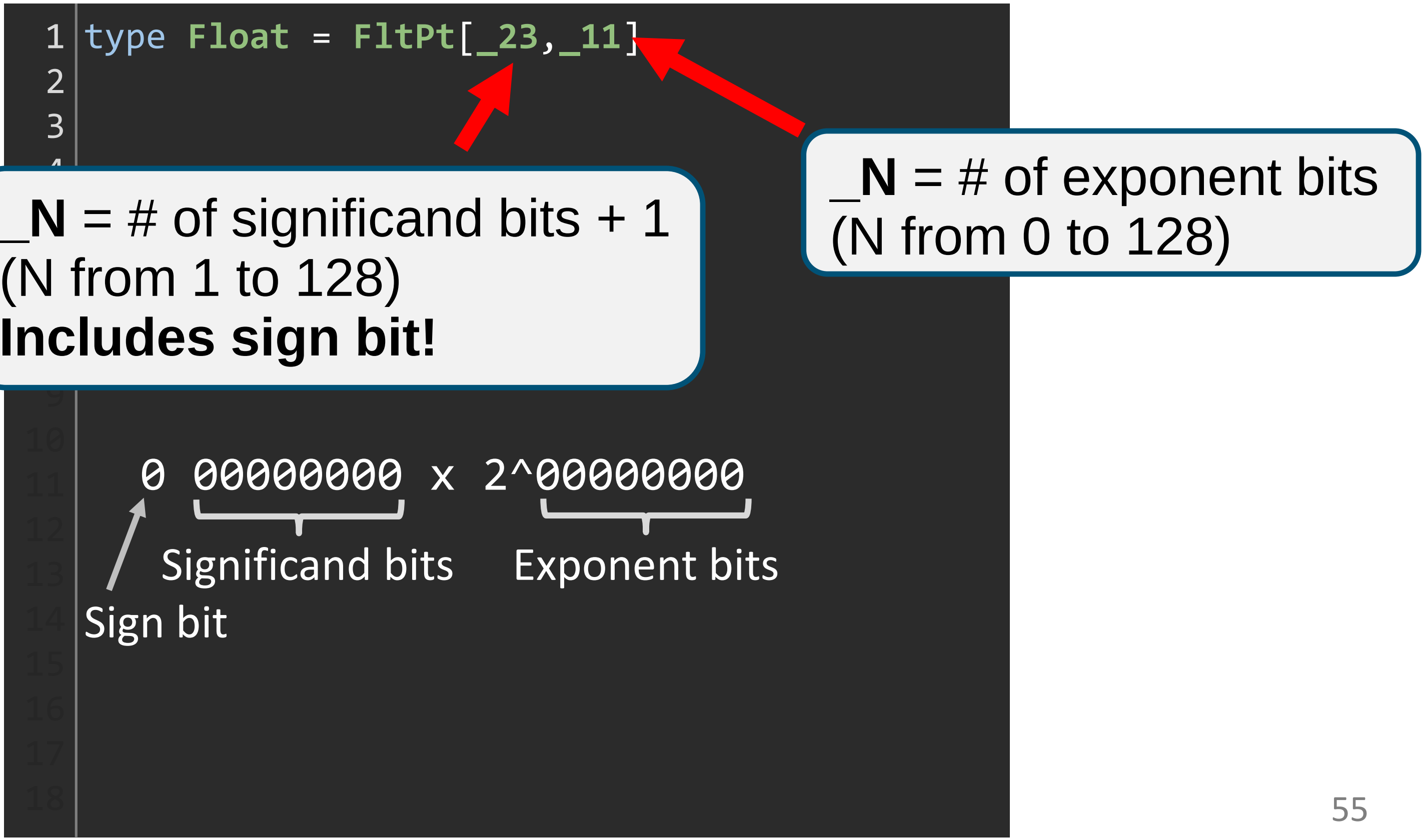
```
type Q8_8 = FixPt[FALSE,_8,_8]

type UInt8 = FixPt[FALSE,_8,_0]

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

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

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



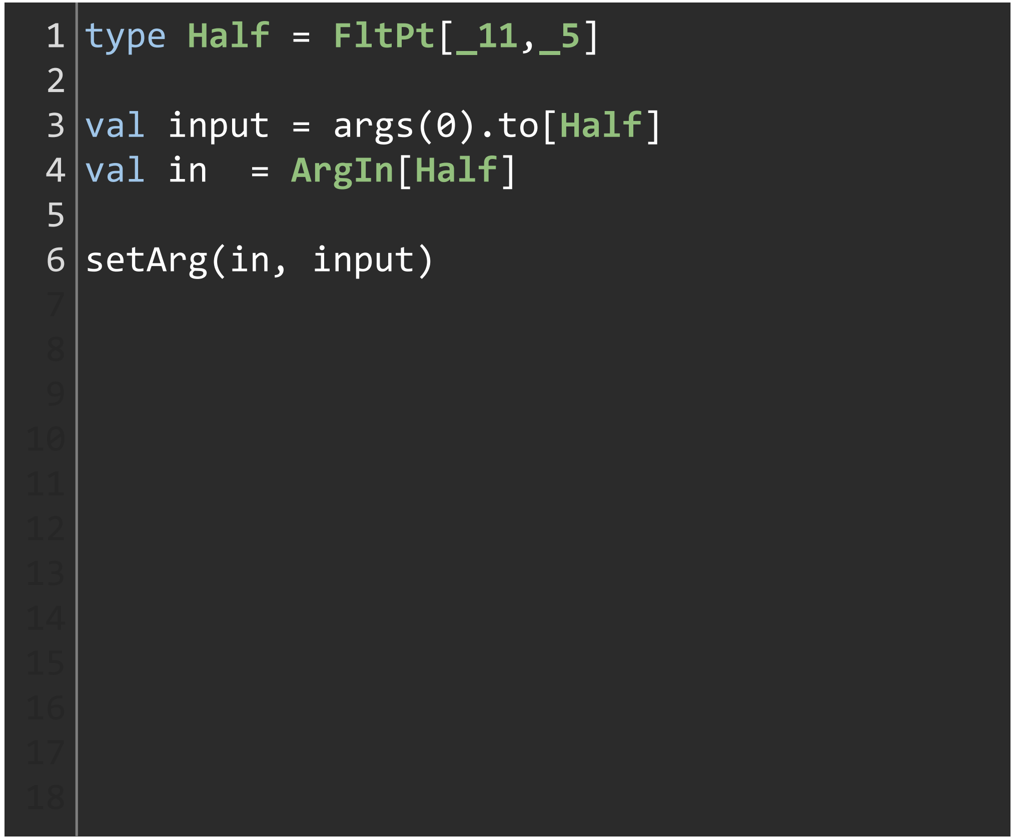
```
type Float = FltPt[_23,_11]
```

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

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

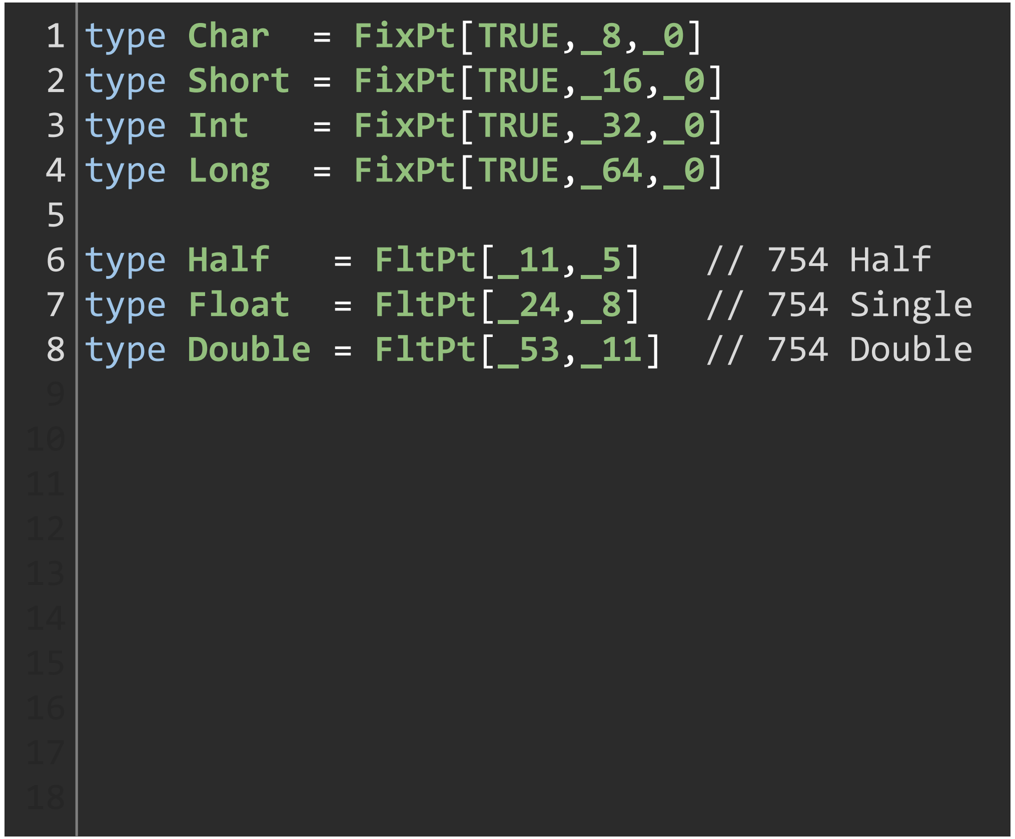
```
0 00000000 x 2^00000000
```

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

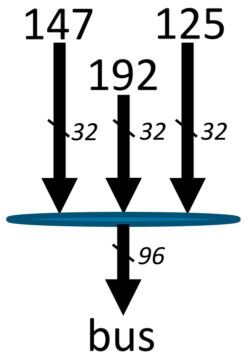
Uses "false" and "true" rather than 0 and 1

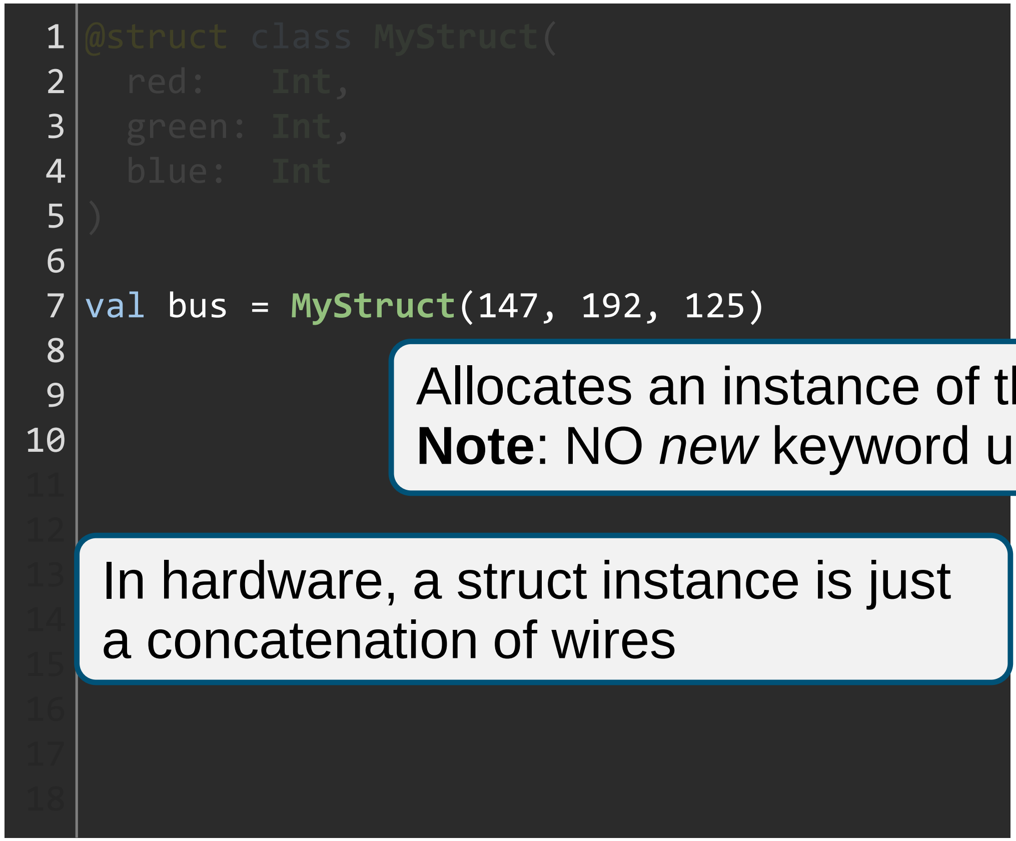
Custom Structs

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

Declares a new Struct type with the given list of fields

Custom Structs



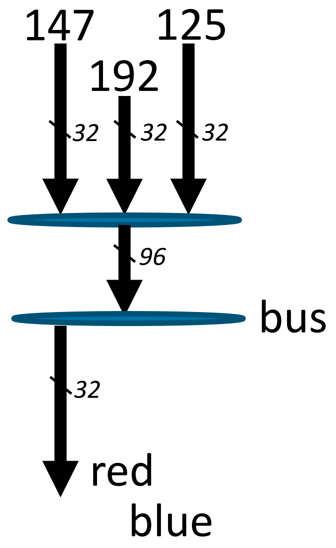


```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

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

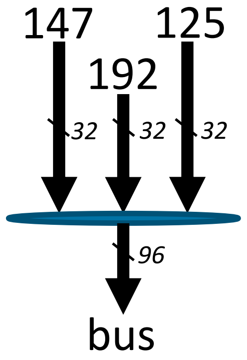
Custom Structs

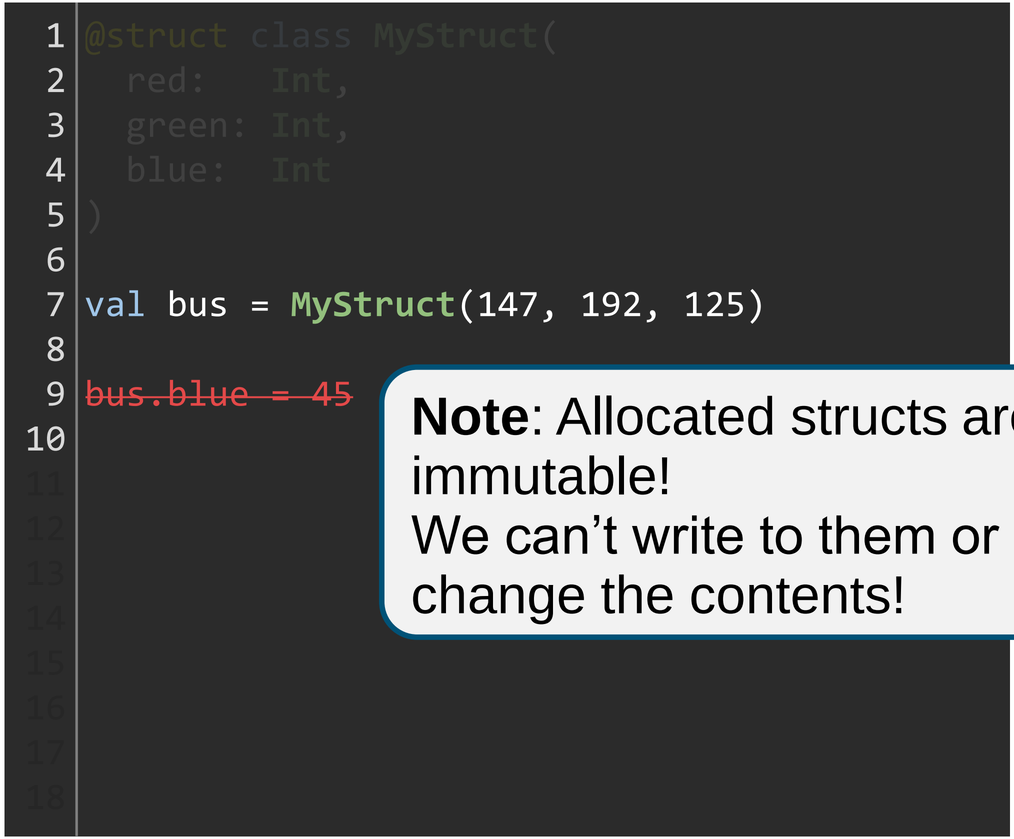


```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

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

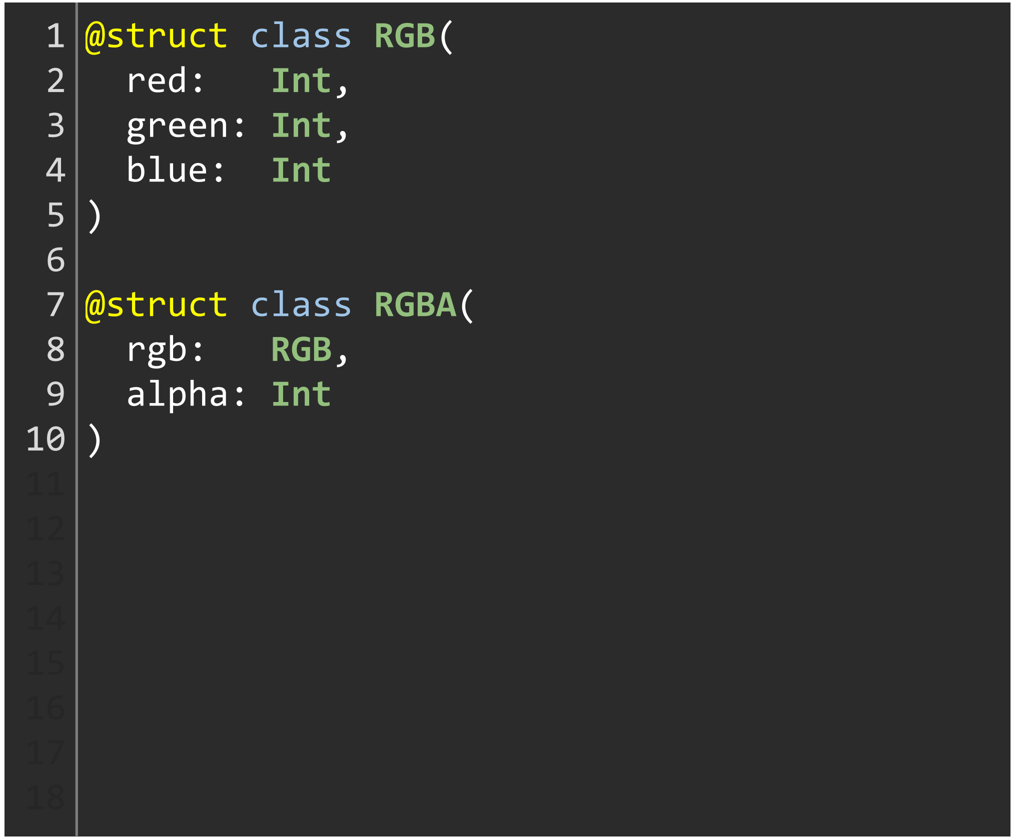




```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

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