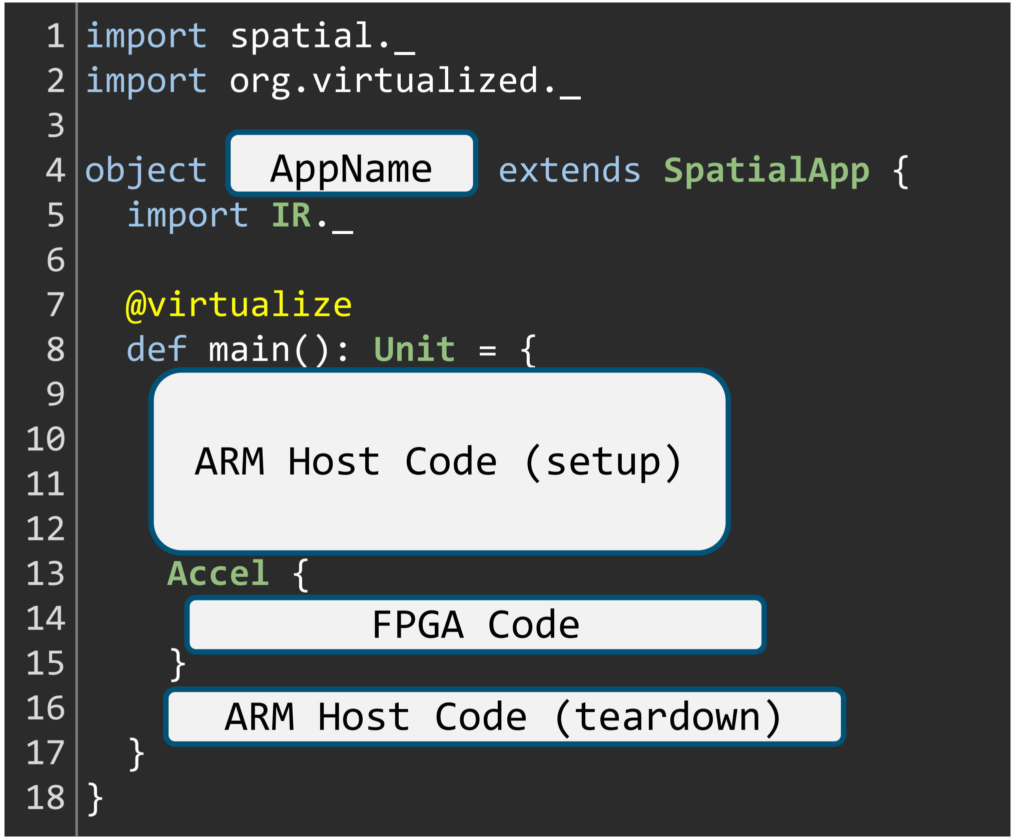
Spatial

Part 2: How to use Spatial?

SPATIAL PROGRAMMING BASICS

Spatial App Template



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

object  AppName  extends SpatialApp {
  import IR._

  @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 {
  import IR._

  @virtualize
  def main(): Unit = {

      -  Define FPGA peripherals
      -  Send data from ARM to FPGA

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

Hello Spatial!

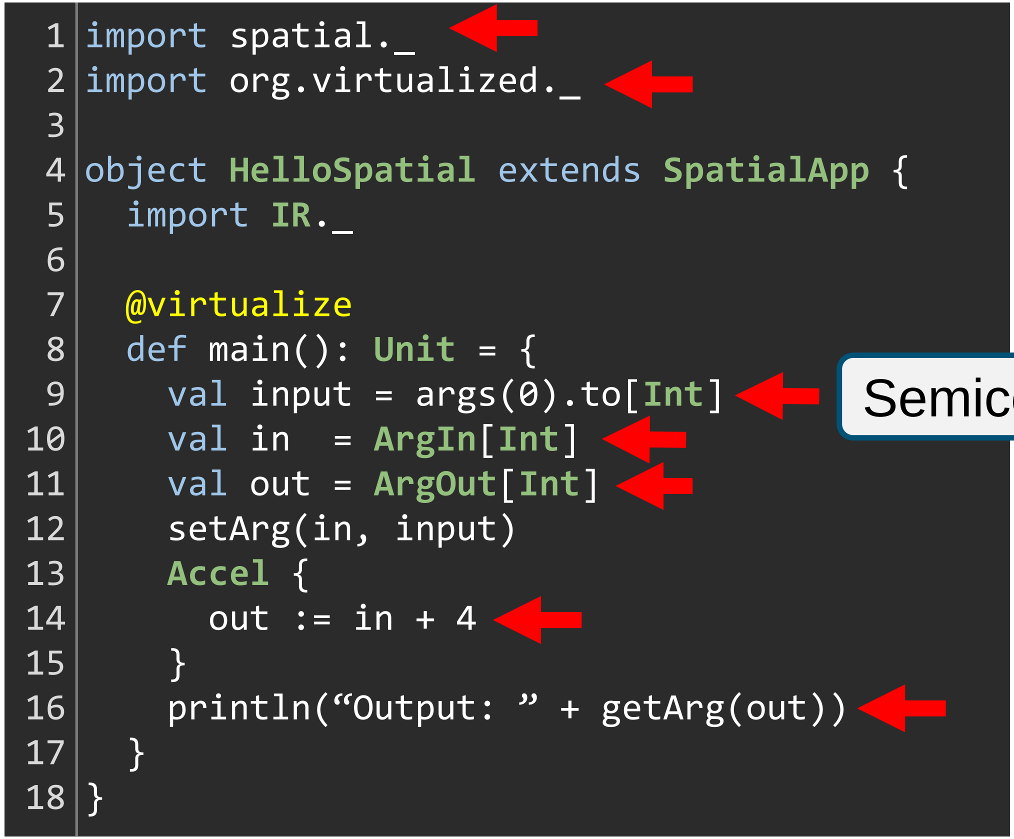
```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {
  import IR._

  @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**

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

object HelloSpatial extends SpatialApp {
  import IR._

  @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

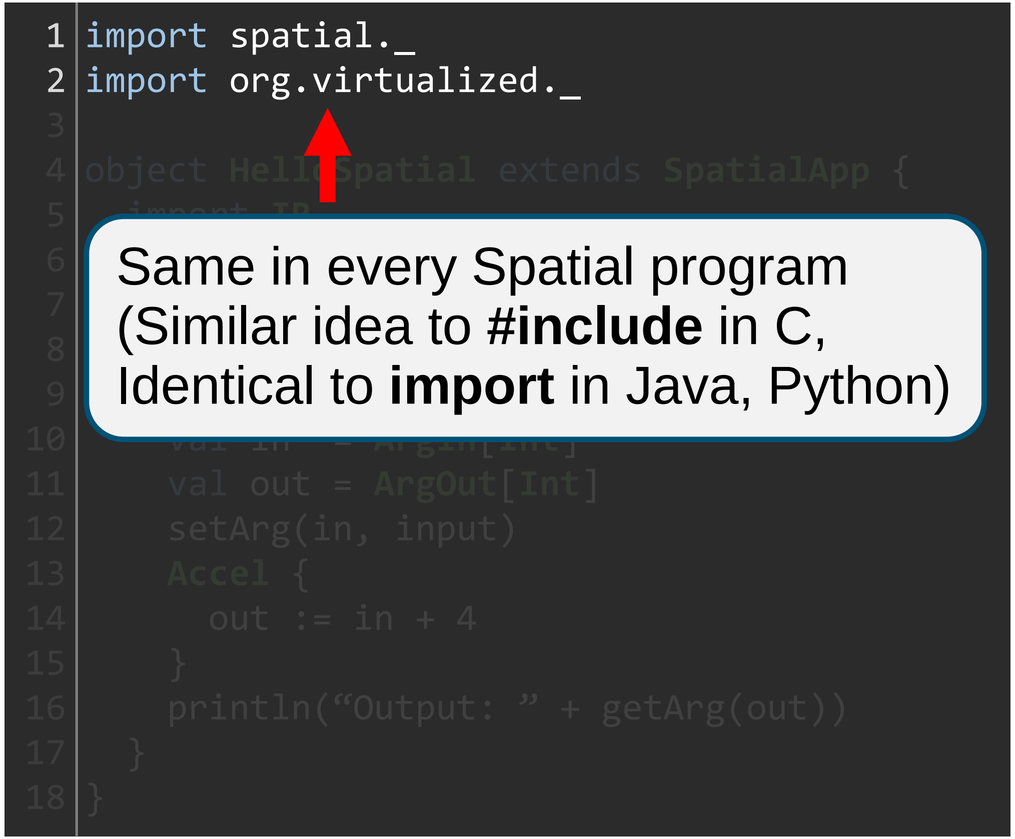
Spatial is Embedded in Scala



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

object HelloSpatial extends SpatialApp {
  import IR._

  @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

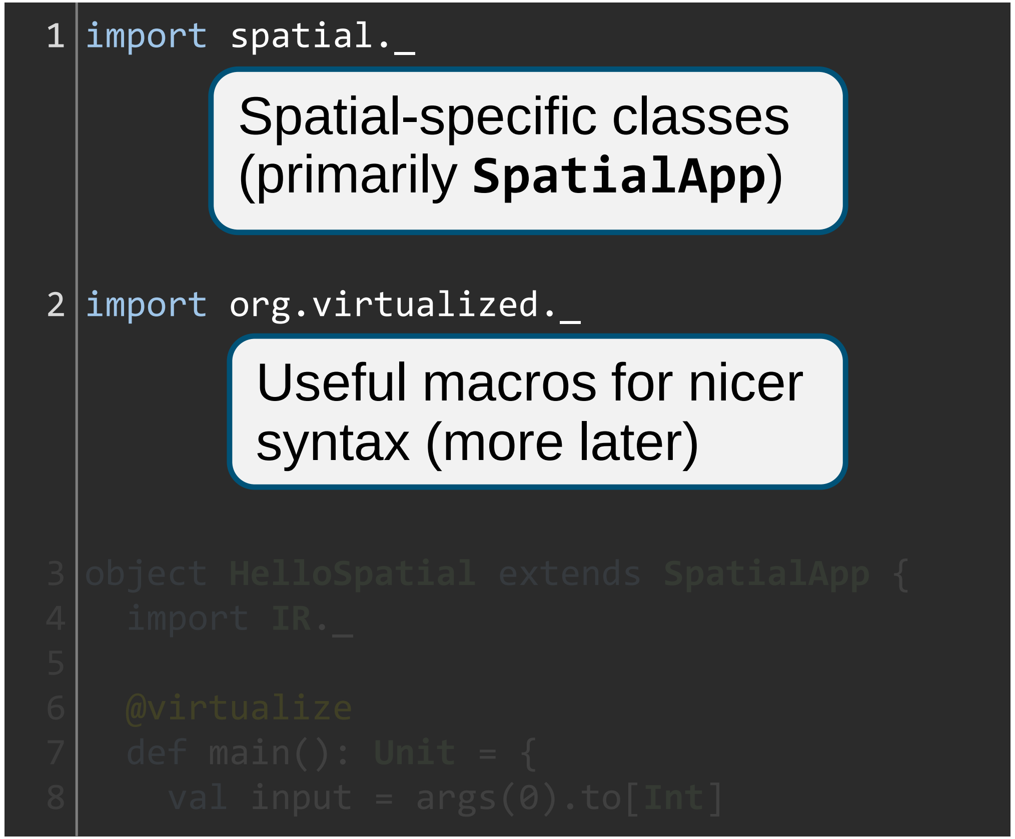
Import Statements



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

object HelloSpatial extends SpatialApp {
  import IR._
    val in = ArgIn[Int]
    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._
```

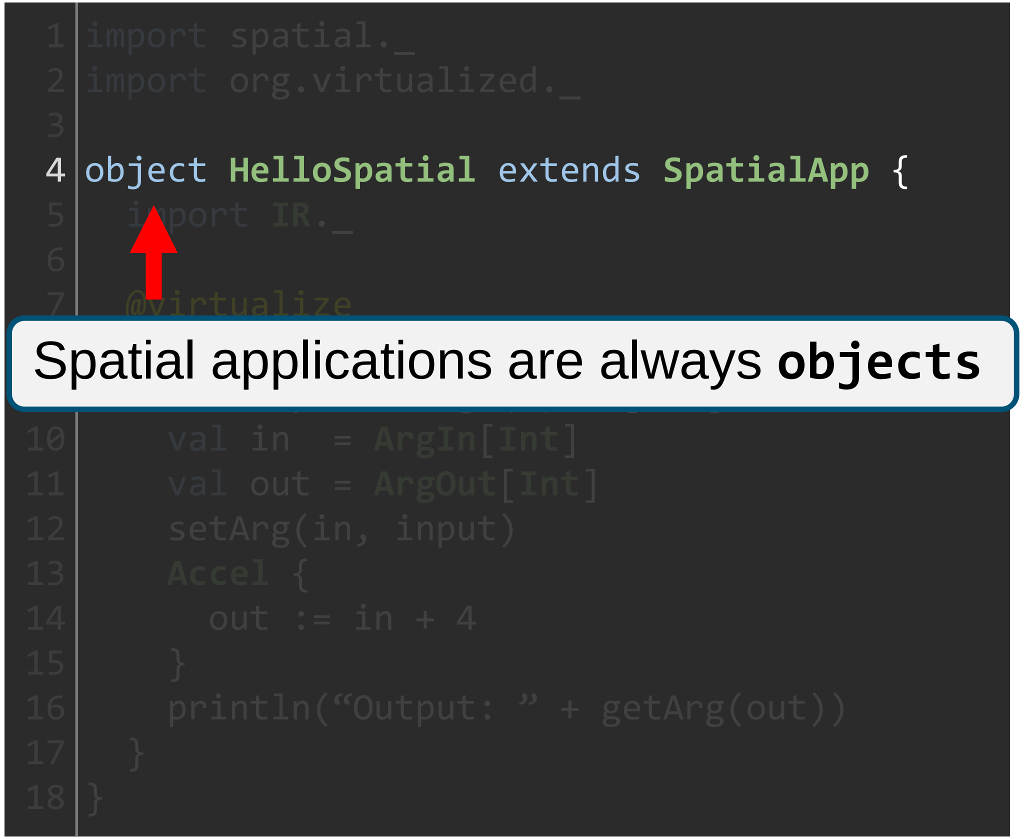
Spatial-specific classes (primarily **SpatialApp**)

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

Useful macros for nicer syntax (more later)

```
object HelloSpatial extends SpatialApp {
  import IR._

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

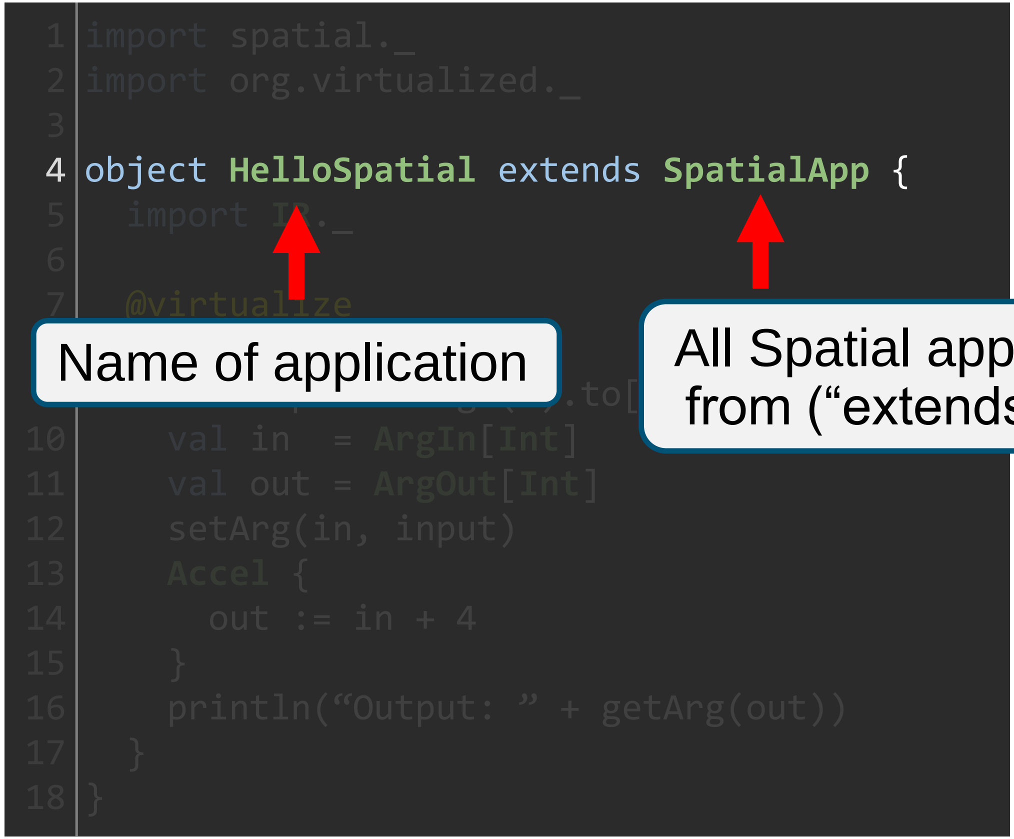
Application Object Declaration



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

object HelloSpatial extends SpatialApp {
  import IR._

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

Spatial applications are always **`objects`**

Application Object Declaration

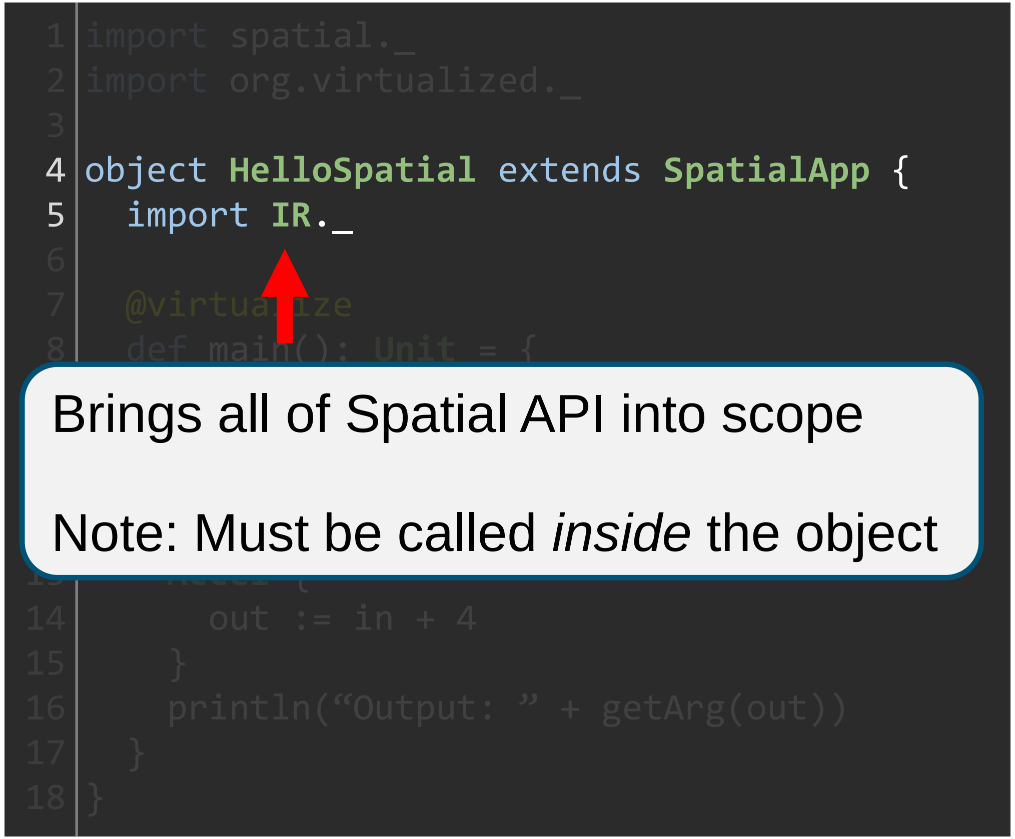


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {
  import IR._

  @virtualize
                            .to[
    val in  = ArgIn[Int]
    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**

Spatial API Import

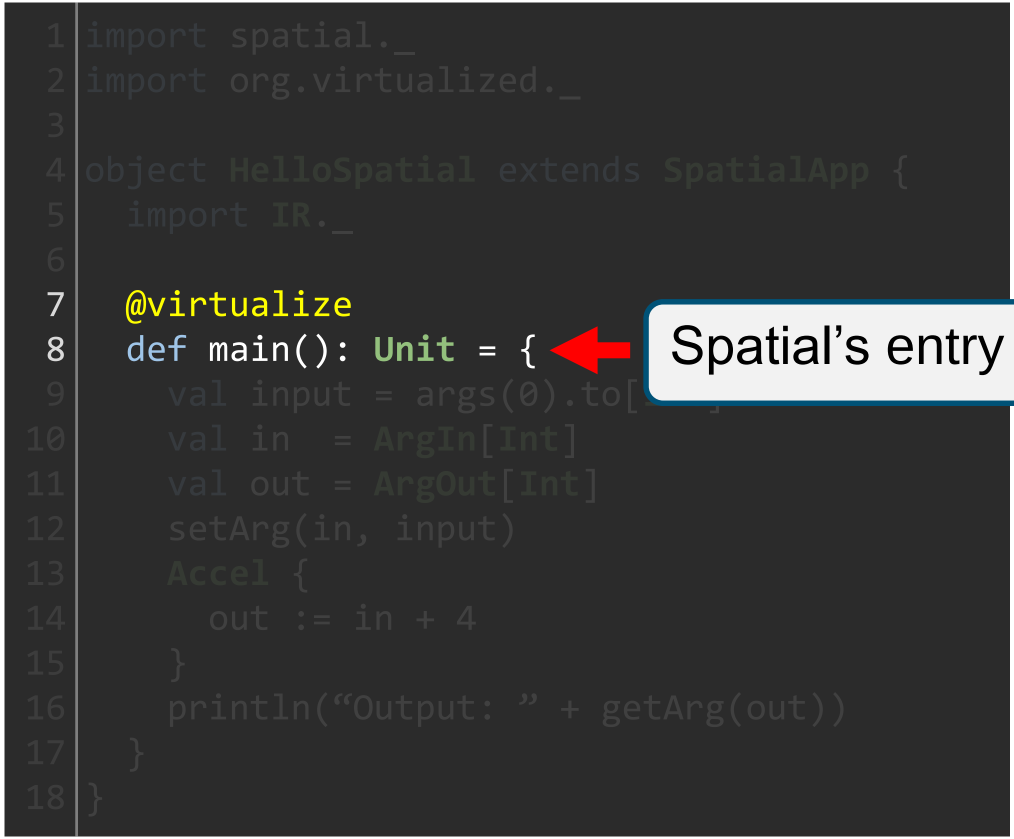


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {
  import IR._

  @virtualize
  def main(): Unit = {

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

Brings all of Spatial API into scope

Note: Must be called *inside* the object

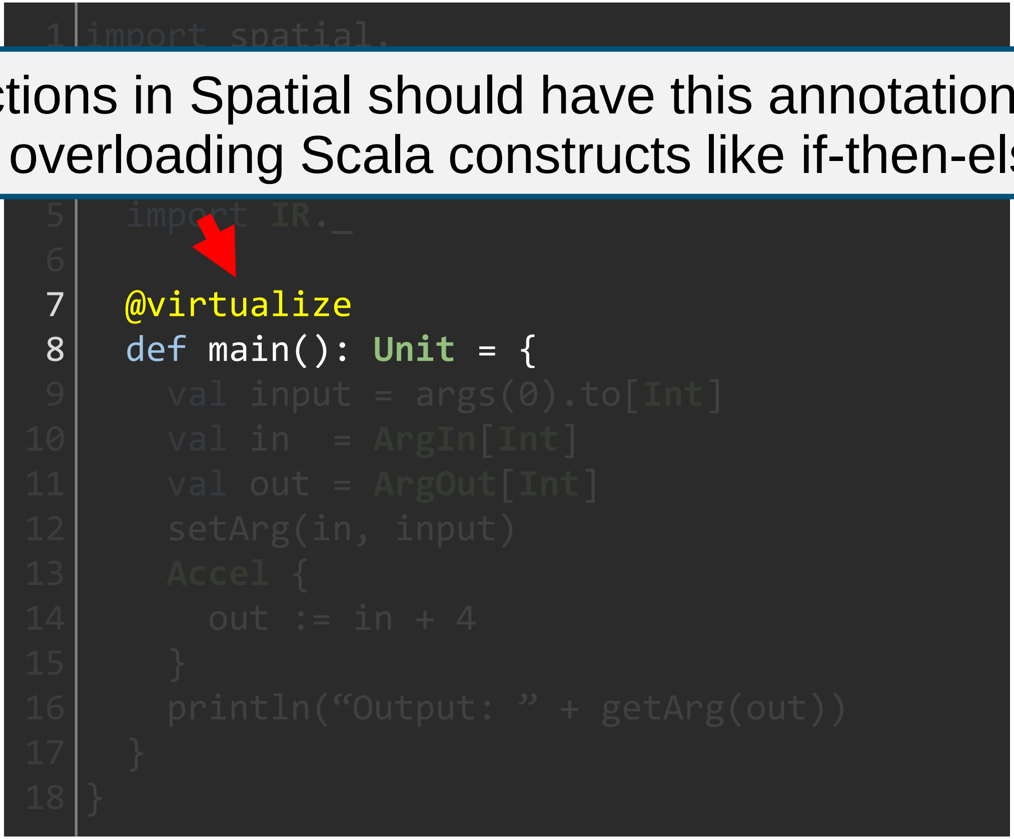
Spatial’s Entry Function: “main()”



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

object HelloSpatial extends SpatialApp {
  import IR._

  @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

“@virtualize” Annotation

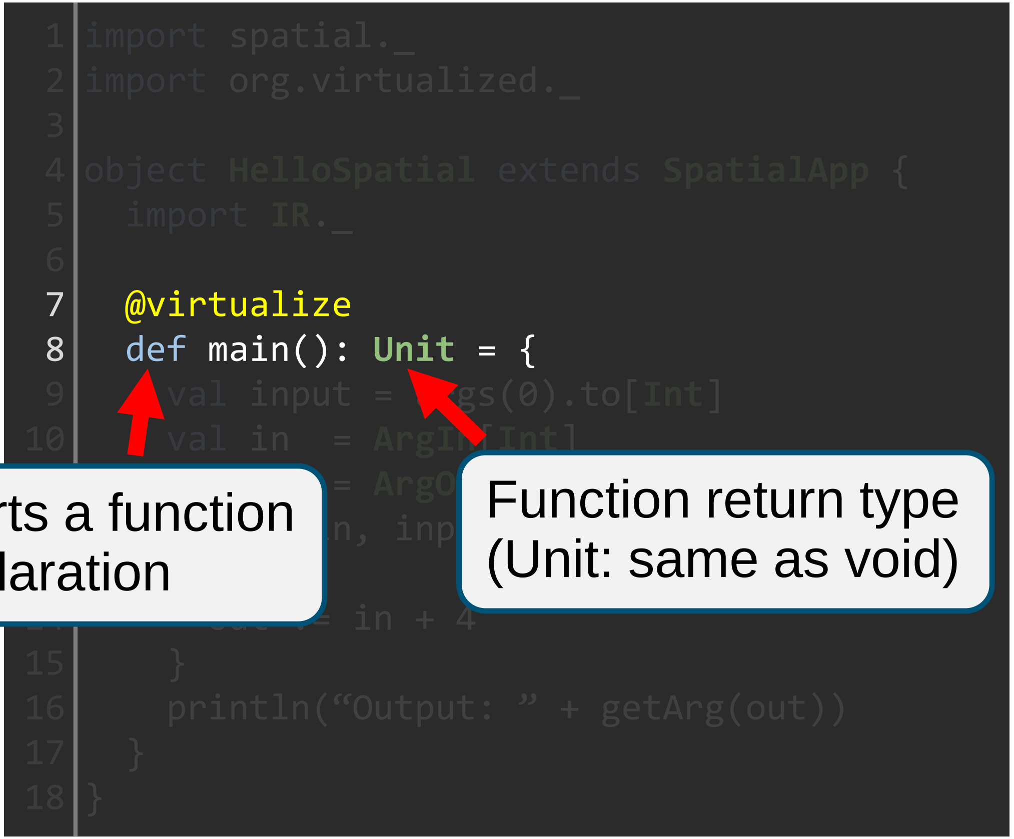


```
import spatial.
```

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

```
  import IR._

  @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()”

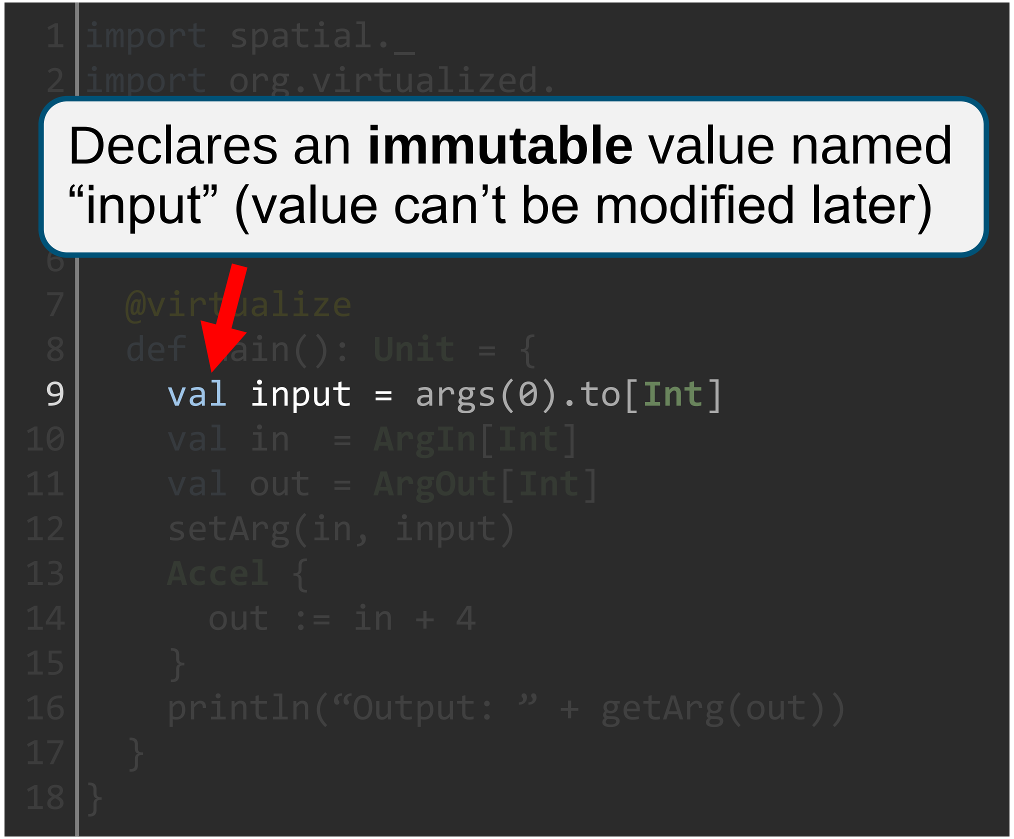


```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {
  import IR._

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

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

Starts a function declaration

Function return type (Unit: same as void)

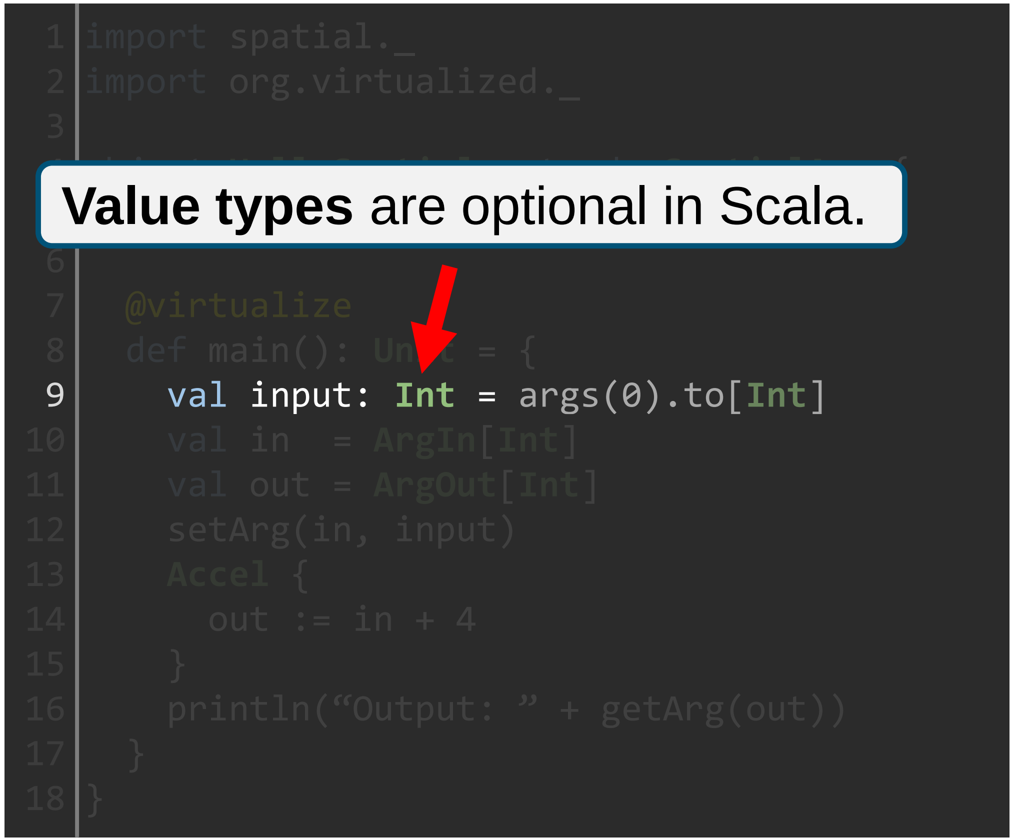
Val Definitions



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

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

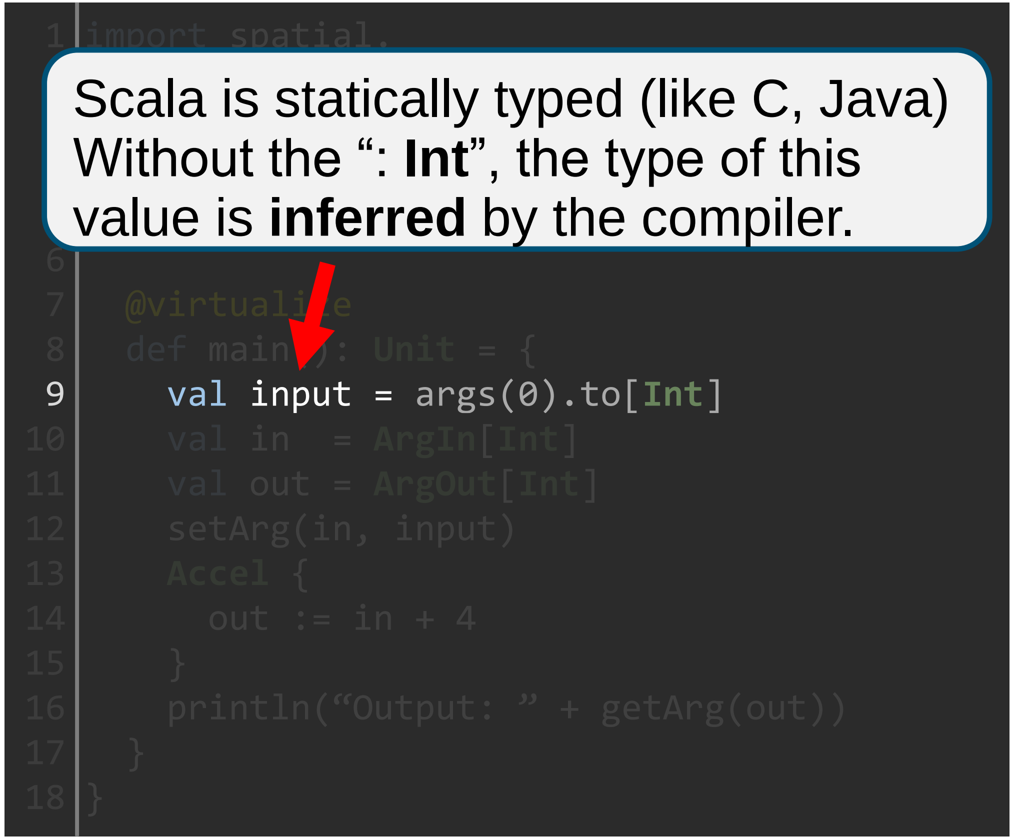
Val Definitions



Value types are optional in Scala.

```
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))
  }
}
```

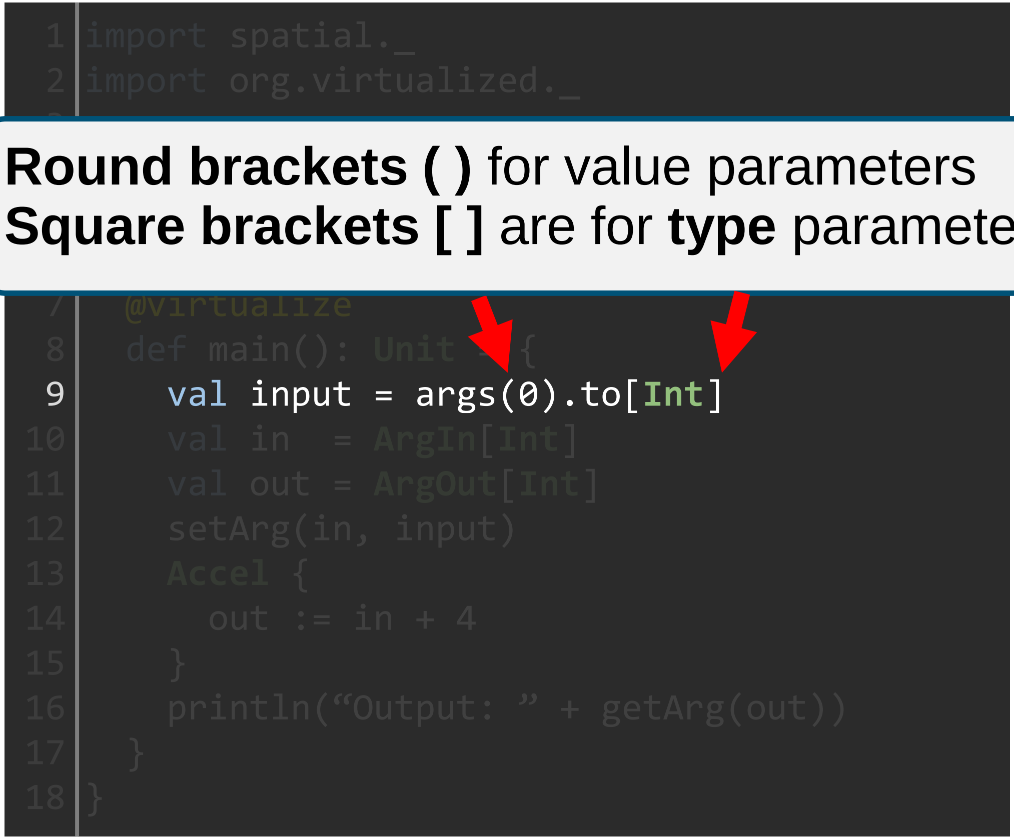
Val Definitions



```
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))
  }
}
```

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

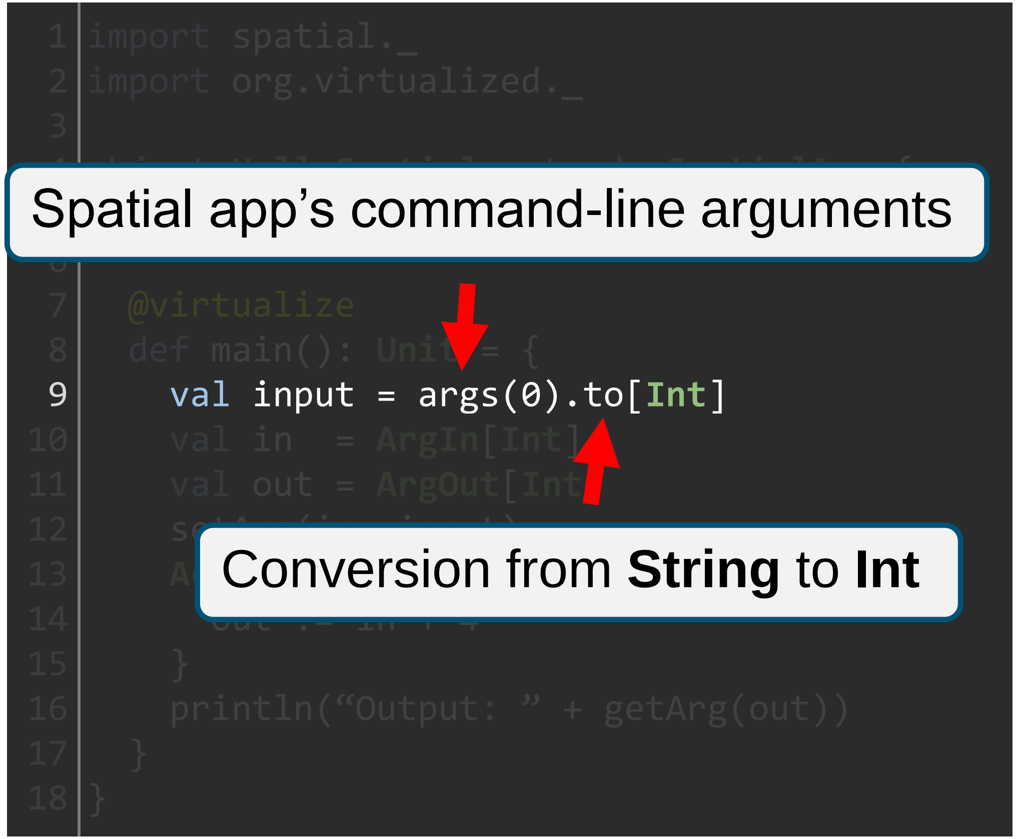
Method Calls



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

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

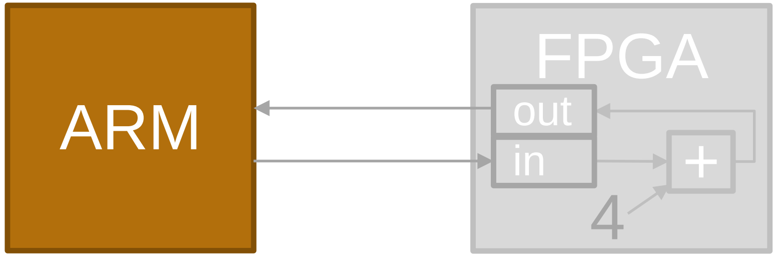
Spatial Command-Line Arguments



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

Spatial app’s command-line arguments

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

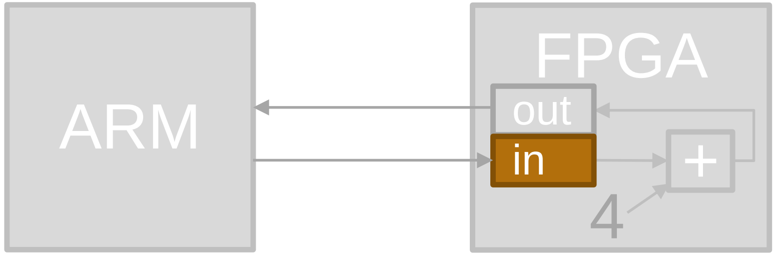


```
int main(int argc, char **argv) {
  int in = atoi(argv[1]);


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

Input Arguments (ArgIn)

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


  @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))
  }
}
```

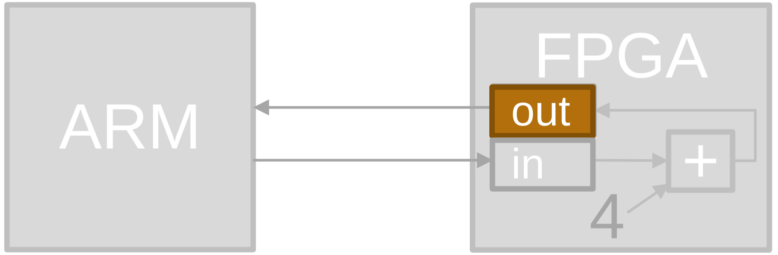
Creates a new register to capture a scalar argument *from* the ARM



Output Arguments (ArgOut)

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

object HelloSpatial extends SpatialApp {
[illegible]
    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))
  }
}
```

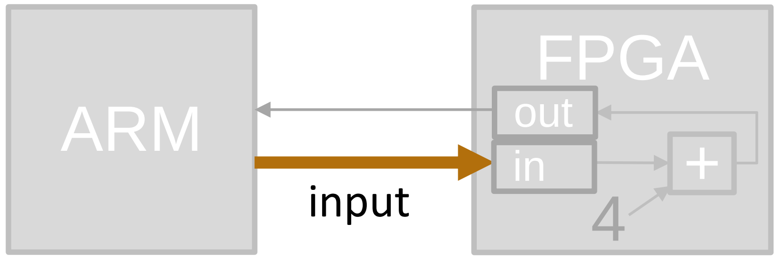
Creates a new scalar argument *to* the ARM *from* the FPGA



Scalar Transfers (ARM → FPGA)

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

object HelloSpatial extends SpatialApp {
  import IR._

  @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

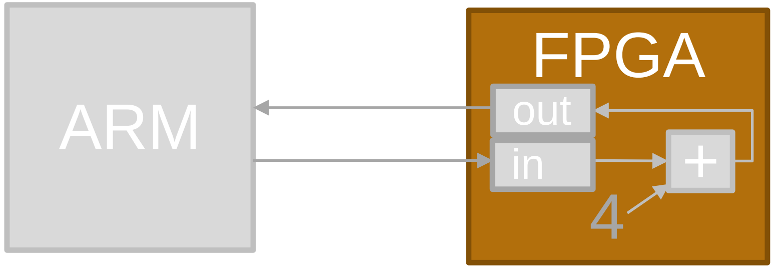


Accel Block

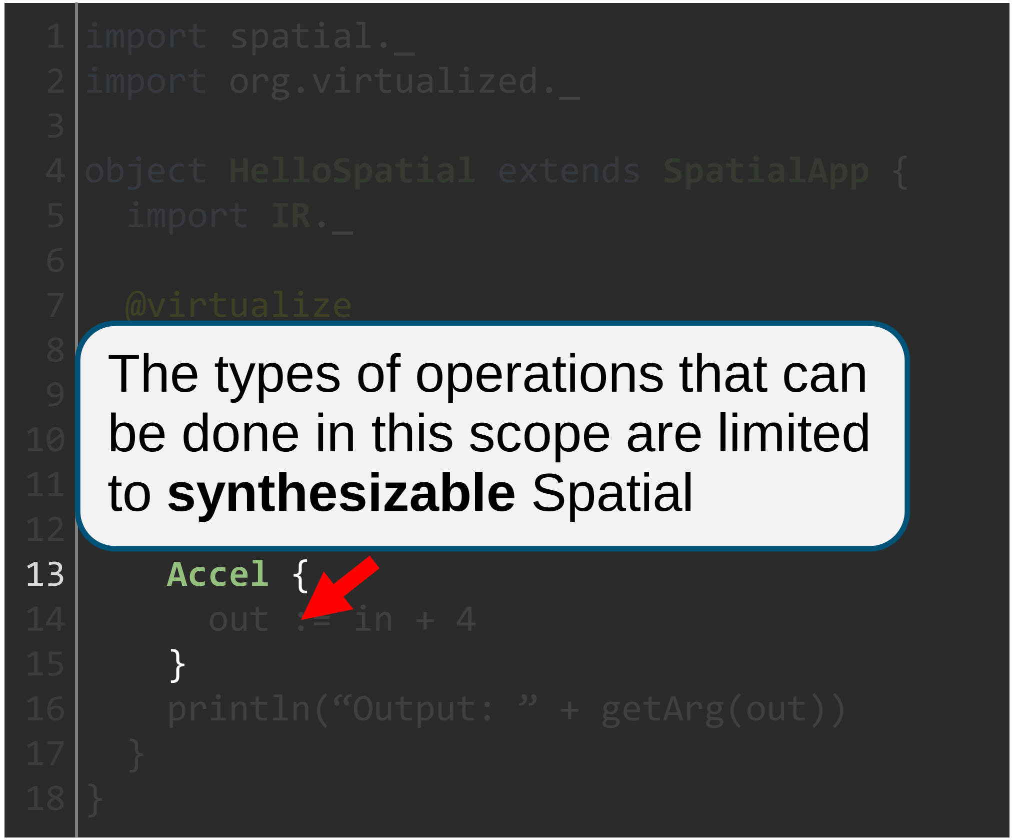
```
import spatial._
import org.virtualized._

object HelloSpatial extends SpatialApp {
  import IR._

  @virtualize
```

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

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

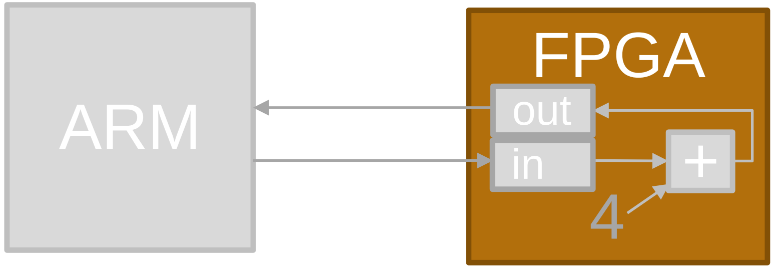


Accel Block



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

object HelloSpatial extends SpatialApp {
  import IR._

  @virtualize

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

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

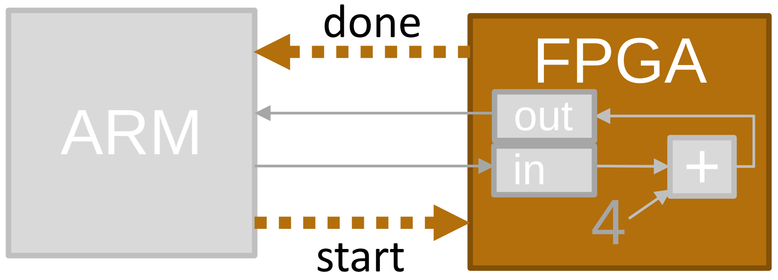


Accel Block

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

object HelloSpatial extends SpatialApp {
  import IR._
```

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

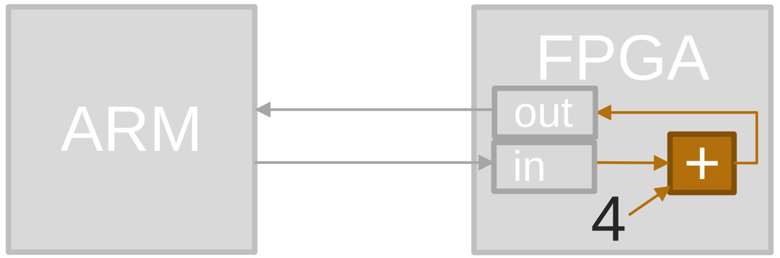
```
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```



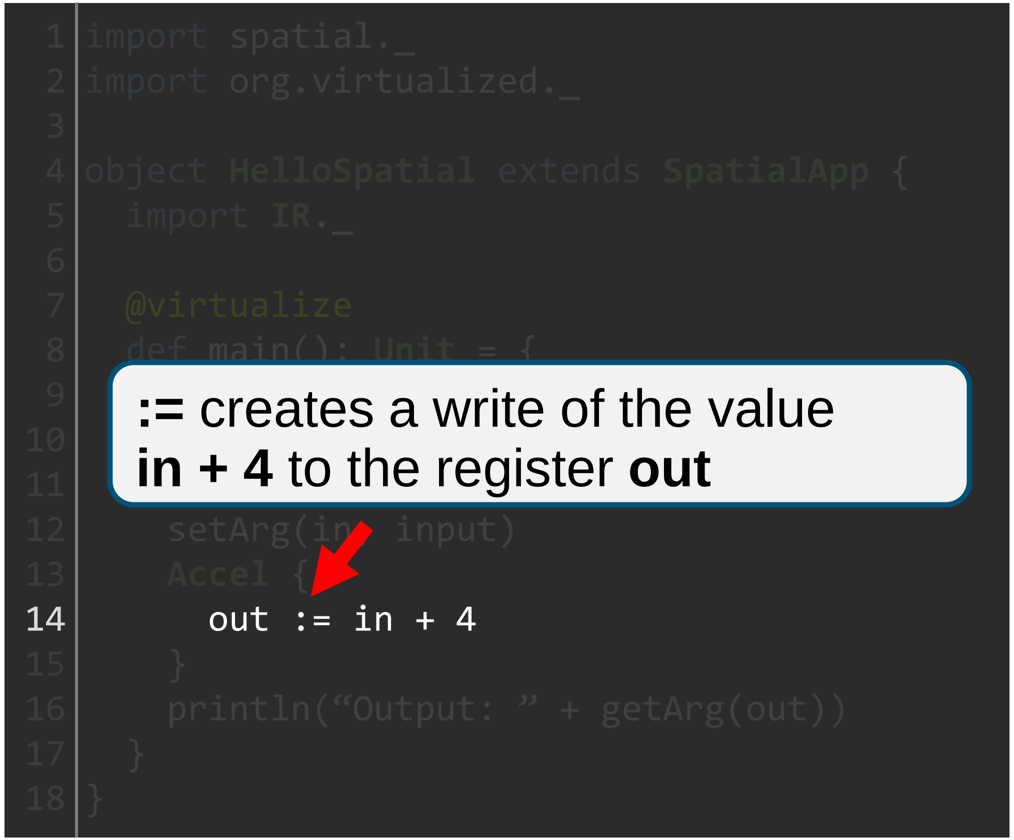
Implicit Register Reads

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

object HelloSpatial extends SpatialApp {
  import IR._

  @virtualize
  def main(): Unit = {



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

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



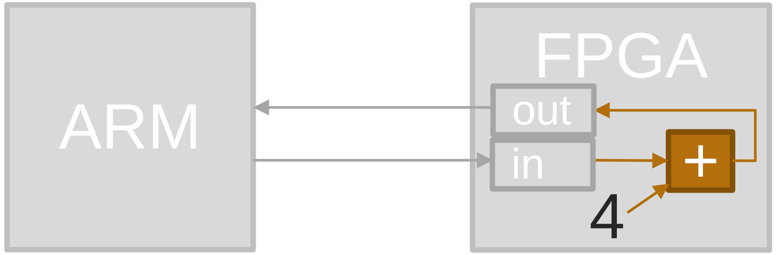
Register Writes



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

object HelloSpatial extends SpatialApp {
  import IR._

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

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

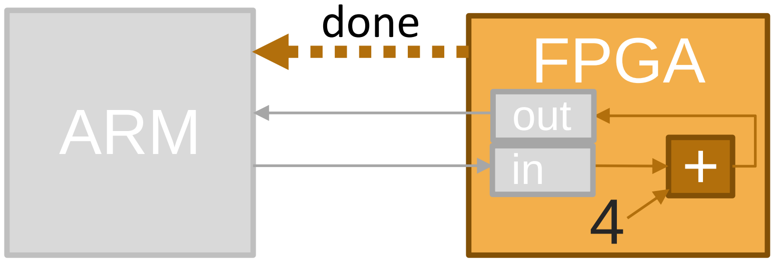
```
    setArg(in, input)
    Accel {
      out := in + 4
    }
    println("Output: " + getArg(out))
  }
}
```



Accel Block Scheduling

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

object HelloSpatial extends SpatialApp {
  import IR._

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

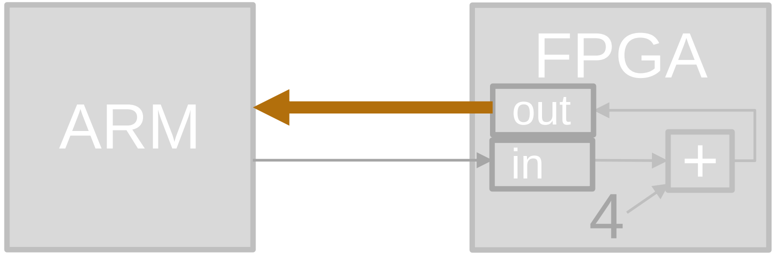
Accel guarantees that FPGA execution completes after all operations in this block complete



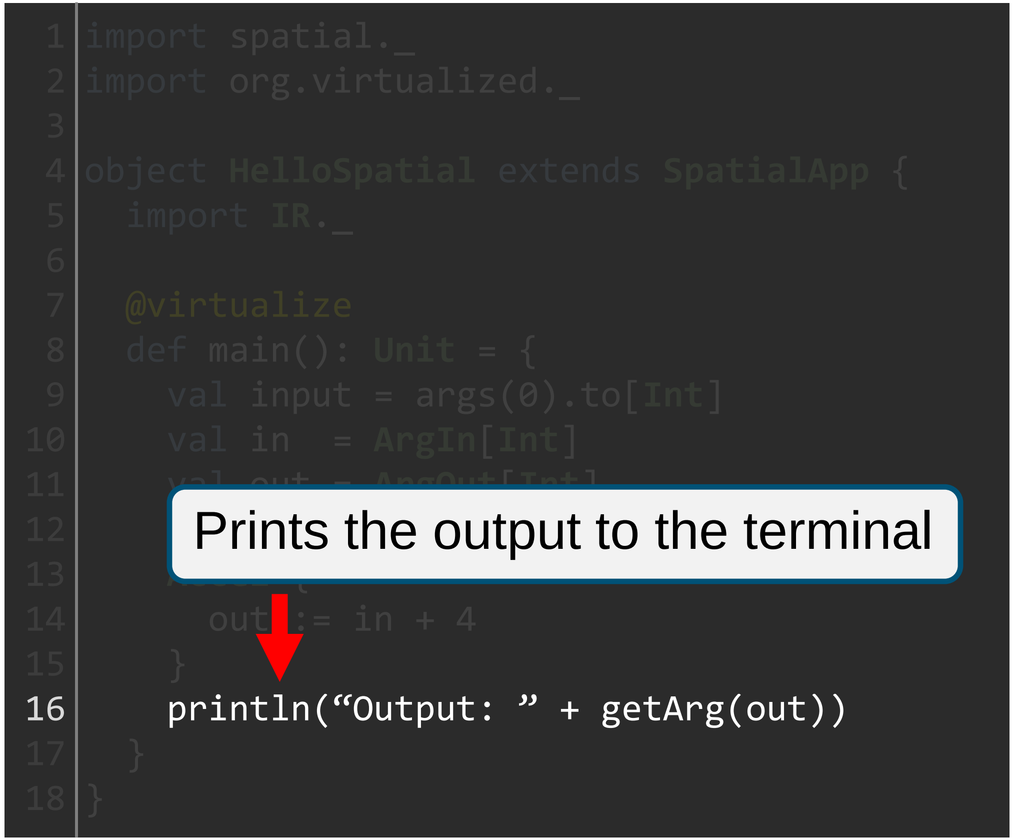
Scalar Transfers (FPGA → ARM)

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

object HelloSpatial extends SpatialApp {
  import IR._

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



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

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



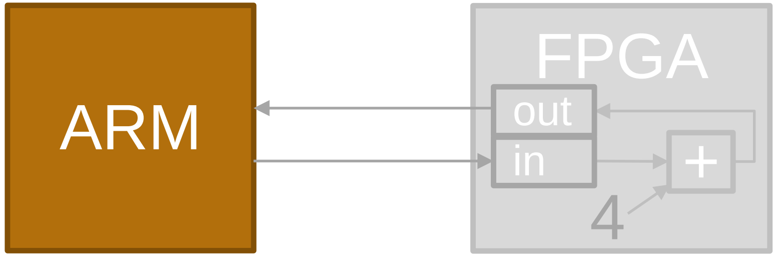
Printing in Spatial**



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

object HelloSpatial extends SpatialApp {
  import IR._

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

    Accel {
      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 {
  import IR._

  @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))
  }
}
```

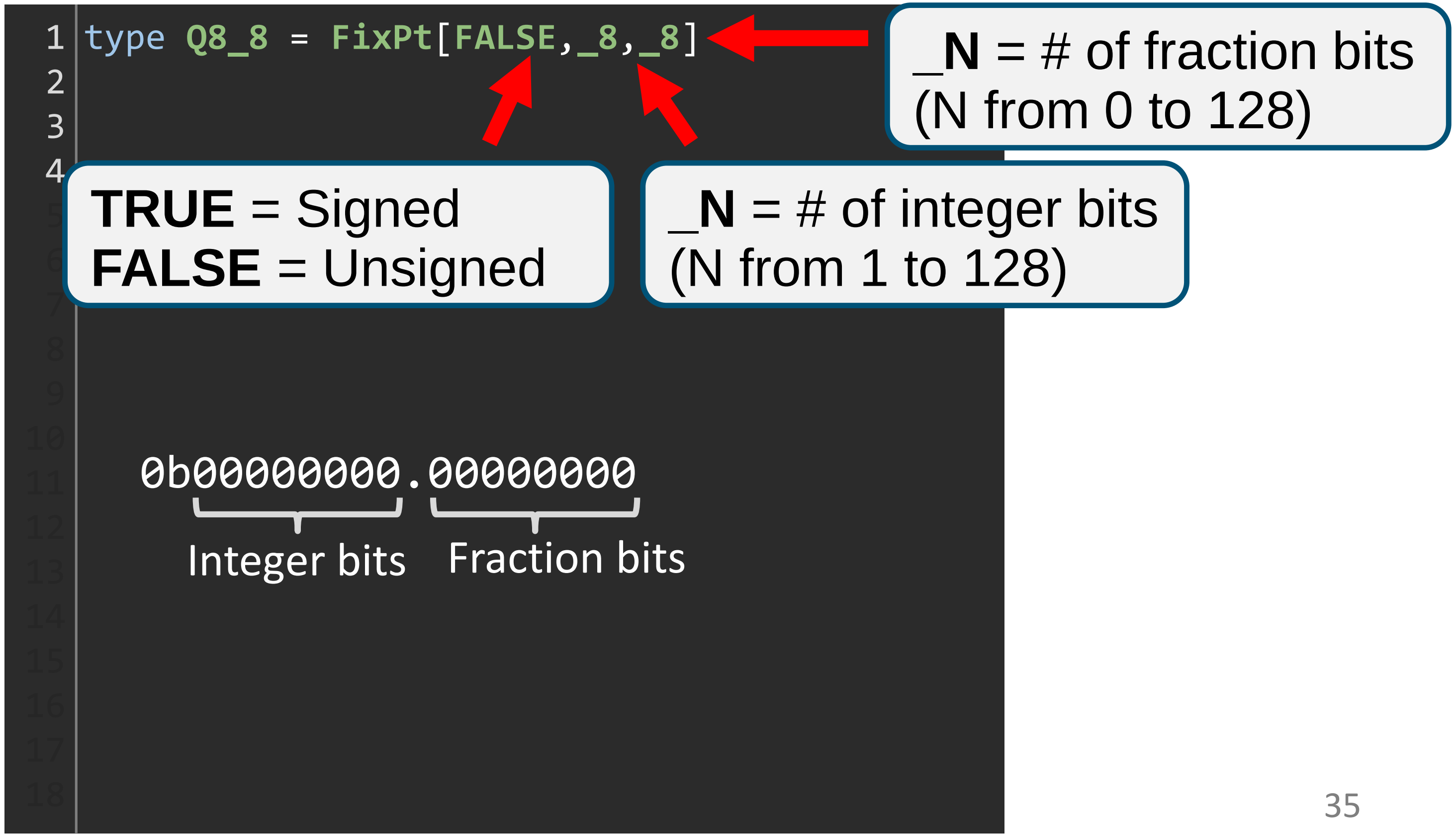
Custom Types in Spatial

- 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]
```

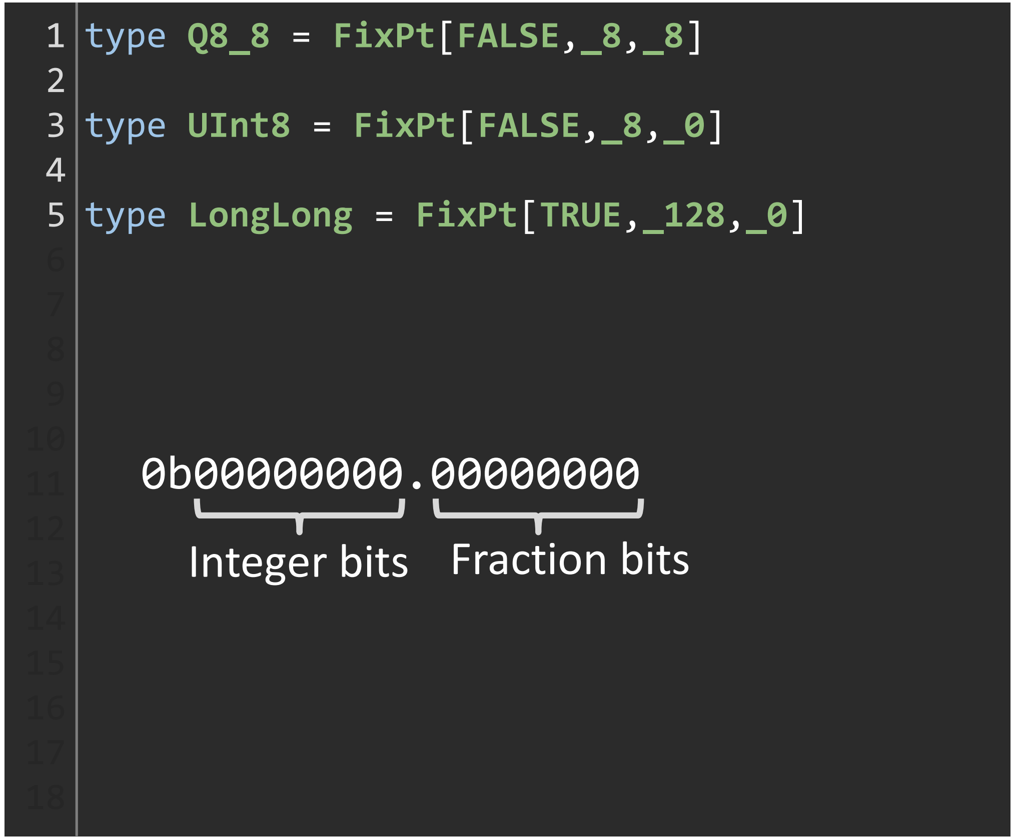
TRUE = Signed
FALSE = Unsigned

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

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

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

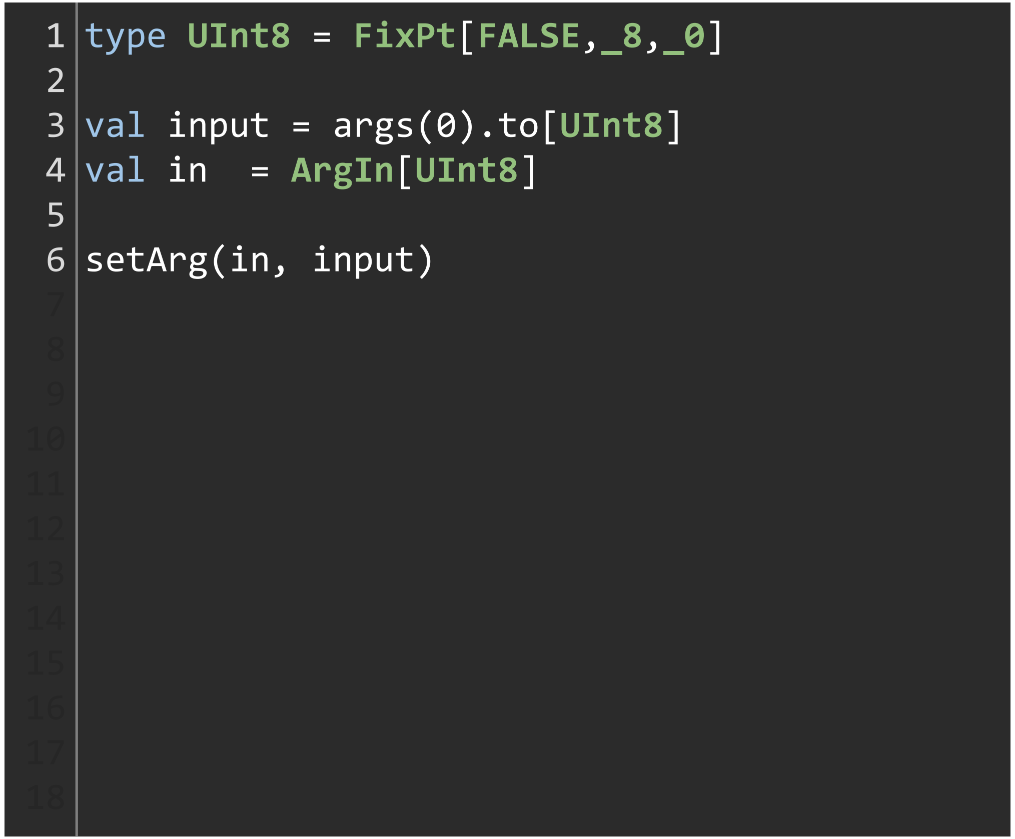
Custom Fixed Point Examples



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

type UInt8 = FixPt[FALSE,_8,_0]

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

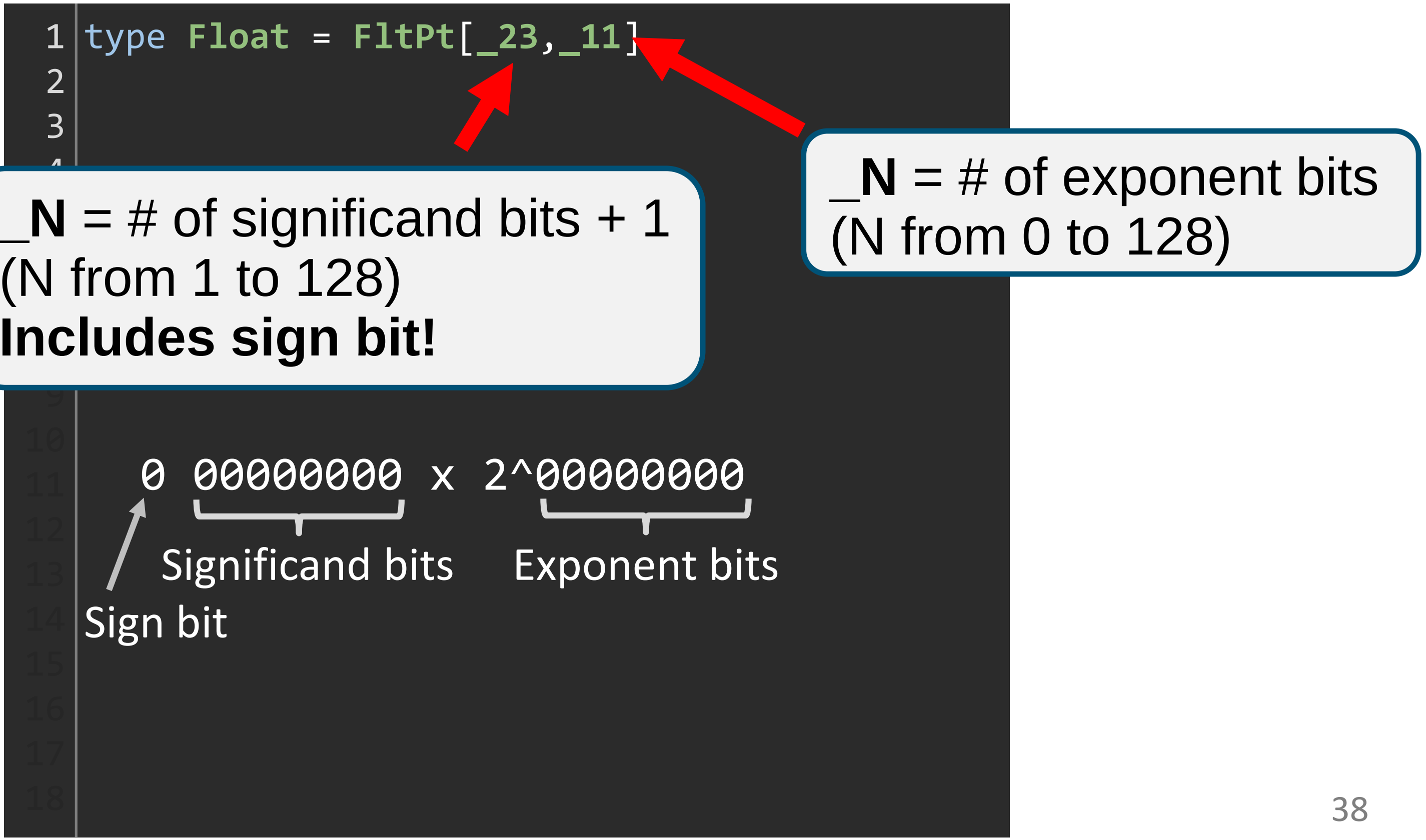
$$0b\underbrace{00000000}_{\text{Integer bits}}.\underbrace{00000000}_{\text{Fraction bits}}$$

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!

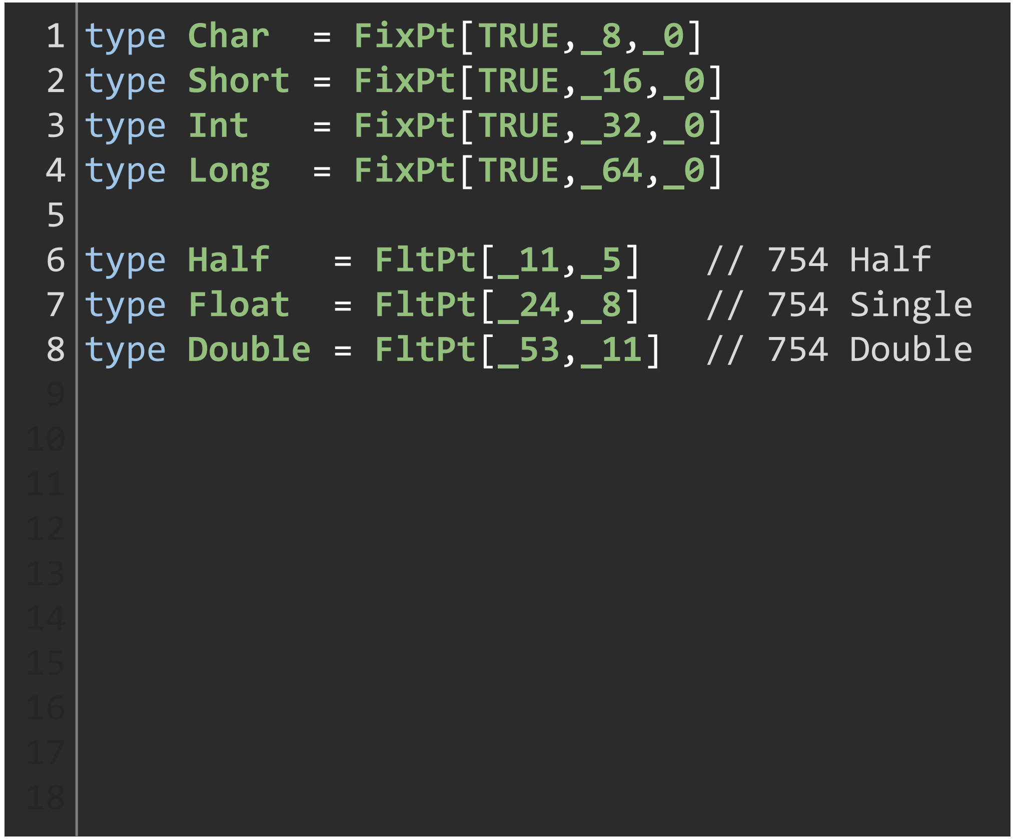
_**N** = # of exponent bits
(N from 0 to 128)

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

Custom Floating Point Types

```
type Full = FltPt[_24,_8]

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

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

```scala


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

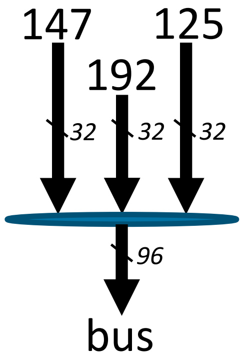
Uses “false” and “true” rather than 0 and 1

Custom Structs

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

Declares a new Struct type with the given list of fields

Custom Structs

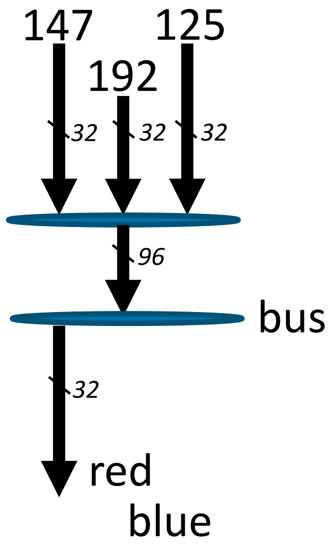


```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

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

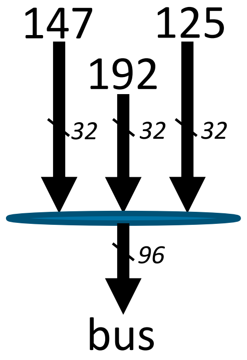
Custom Structs

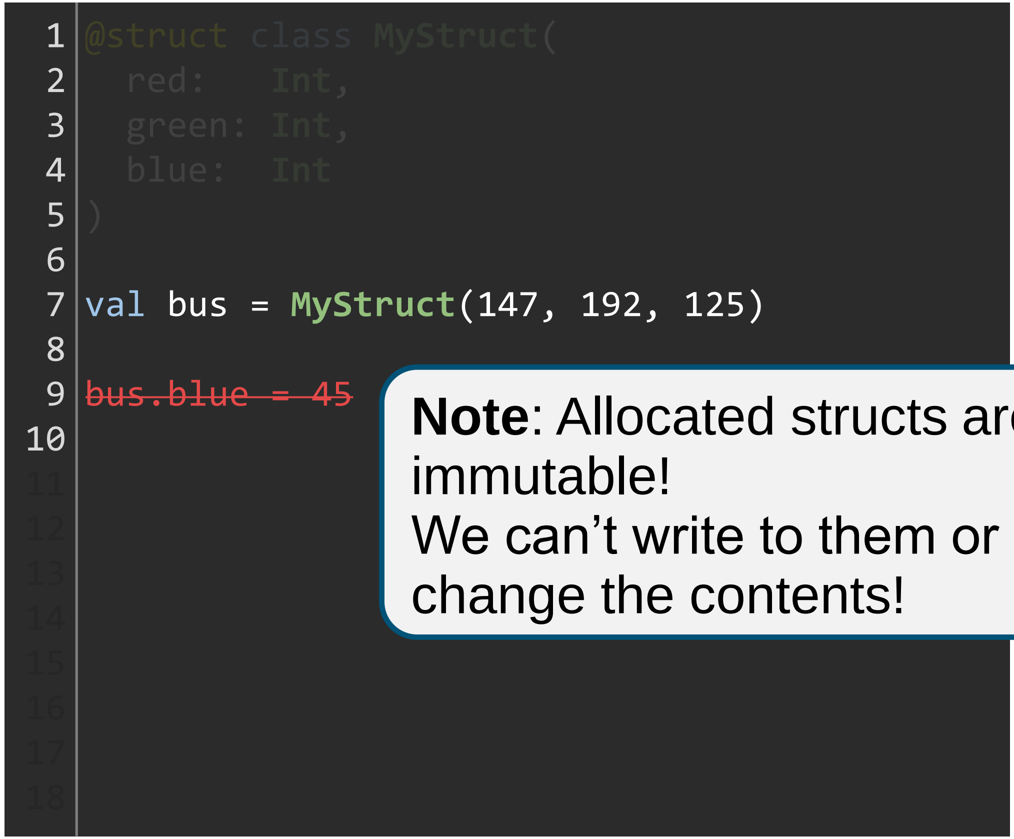


```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

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

Spatial Tutorial

Part 2: Spatial Memories and Control

VECTORS & BIT MANIPULATION

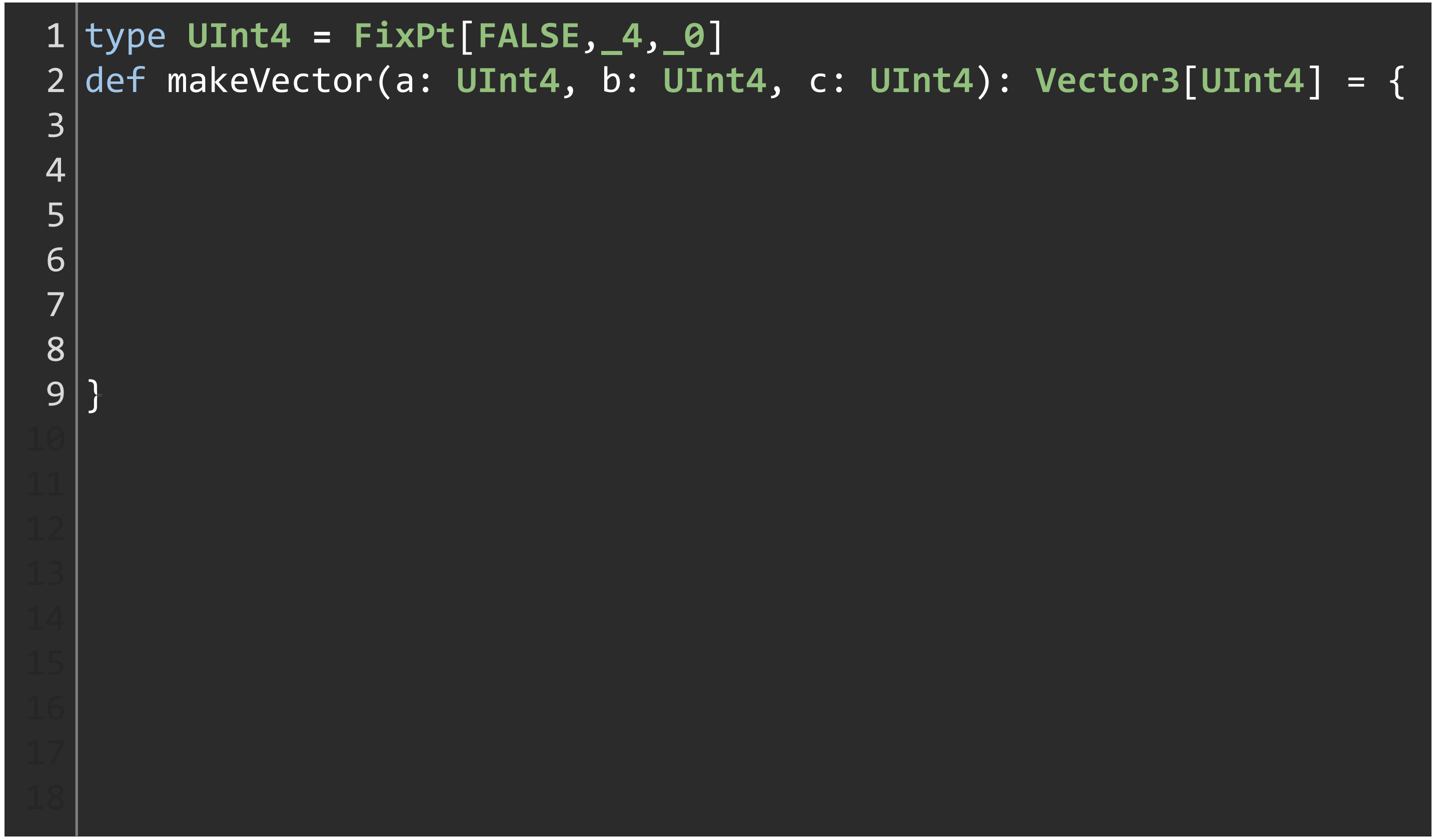
Vectors

- Another primitive type in Spatial
- Like structs, equivalent to a concatenation of wires
- Every word in the vector is the same type
- Number of words in vector must be statically known

Vector Types

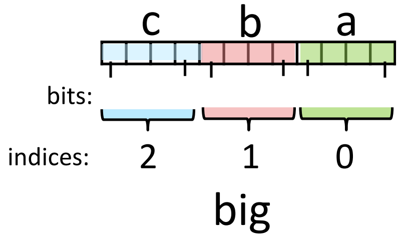
- Vector[T]
  - Vector1[T] to Vector128[T]
    - Width statically known by Spatial compiler
    - Width statically known by Scala compiler
  - VectorN[T]
    - Width statically known by Spatial compiler
    - Width *not* statically known by Scala compiler
    - Must be cast to a Vector###[T] type before further use

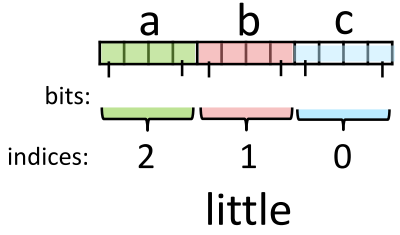
Vectors



```
type UInt4 = FixPt[FALSE,_4,_0]
def makeVector(a: UInt4, b: UInt4, c: UInt4): Vector3[UInt4] = {







}
```

Vectors: Big and Little Endian



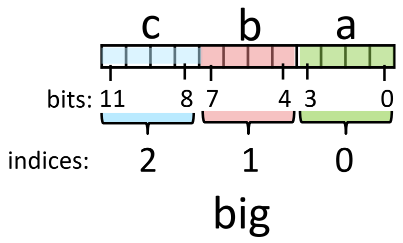


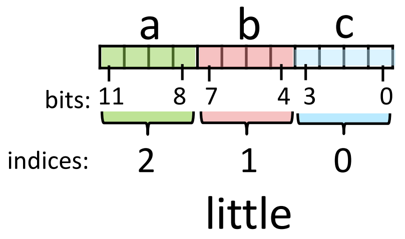
```
type UInt4 = FixPt[FALSE,_4,_0]
def makeVector(a: UInt4, b: UInt4, c: UInt4): Vector3[UInt4] = {
  val big = Vector.BigEndian(a, b, c)
                                        indices: 0   1   2


  val little = Vector.LittleEndian(a, b, c)
                                             indices: 2   1   0

}
```

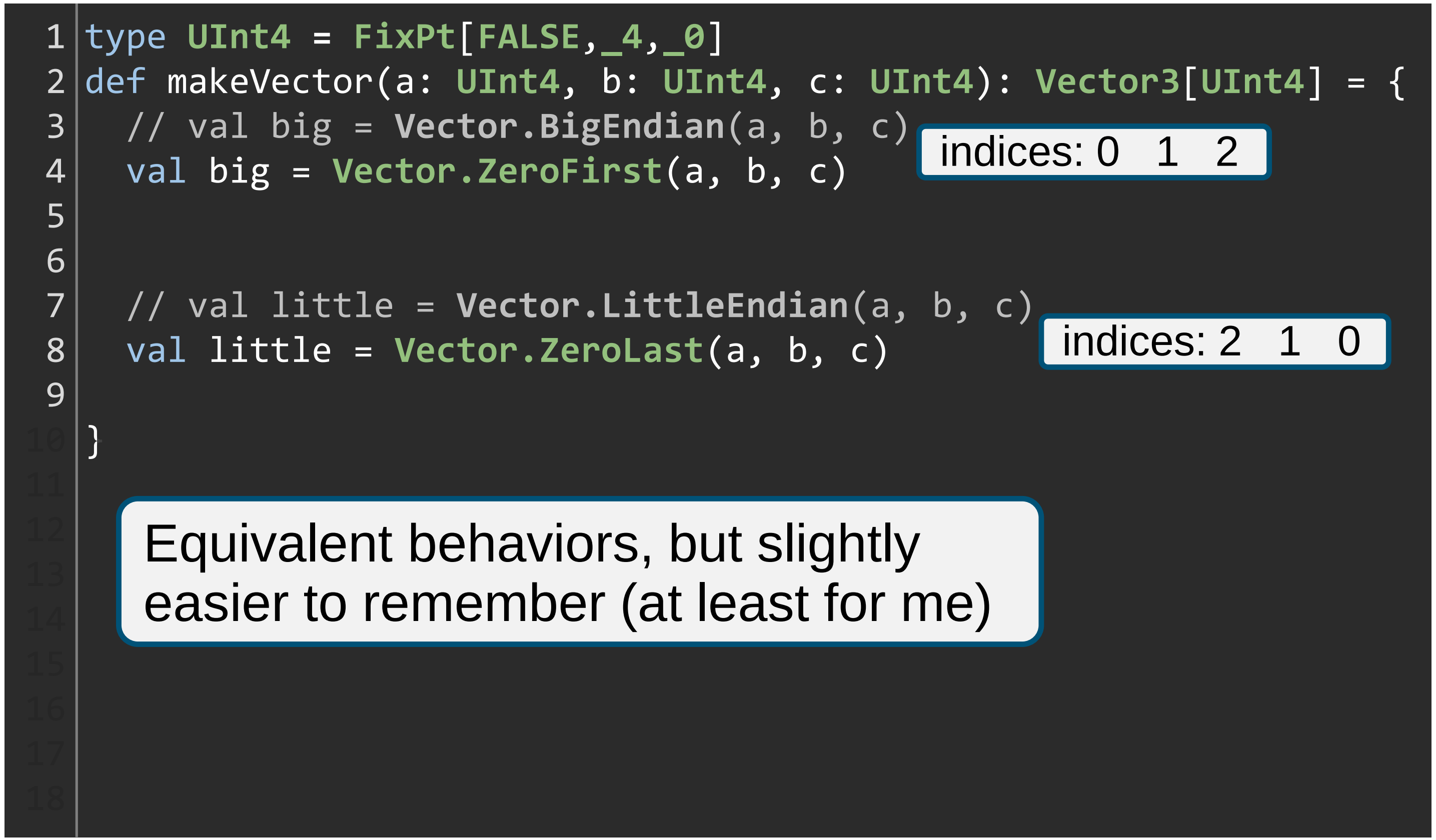
Both allocate a Vector3 with the given elements, but the order is different

Like structs, this is equivalent to a concatenation of wires

Aside: Equivalent Vector Mnemonics







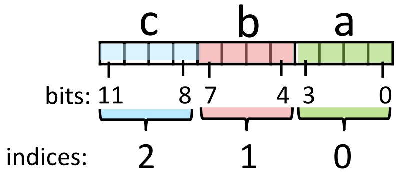
```
type UInt4 = FixPt[FALSE,_4,_0]
def makeVector(a: UInt4, b: UInt4, c: UInt4): Vector3[UInt4] = {
  // val big = Vector.BigEndian(a, b, c)
  val big = Vector.ZeroFirst(a, b, c)


  // val little = Vector.LittleEndian(a, b, c)
  val little = Vector.ZeroLast(a, b, c)

}
```

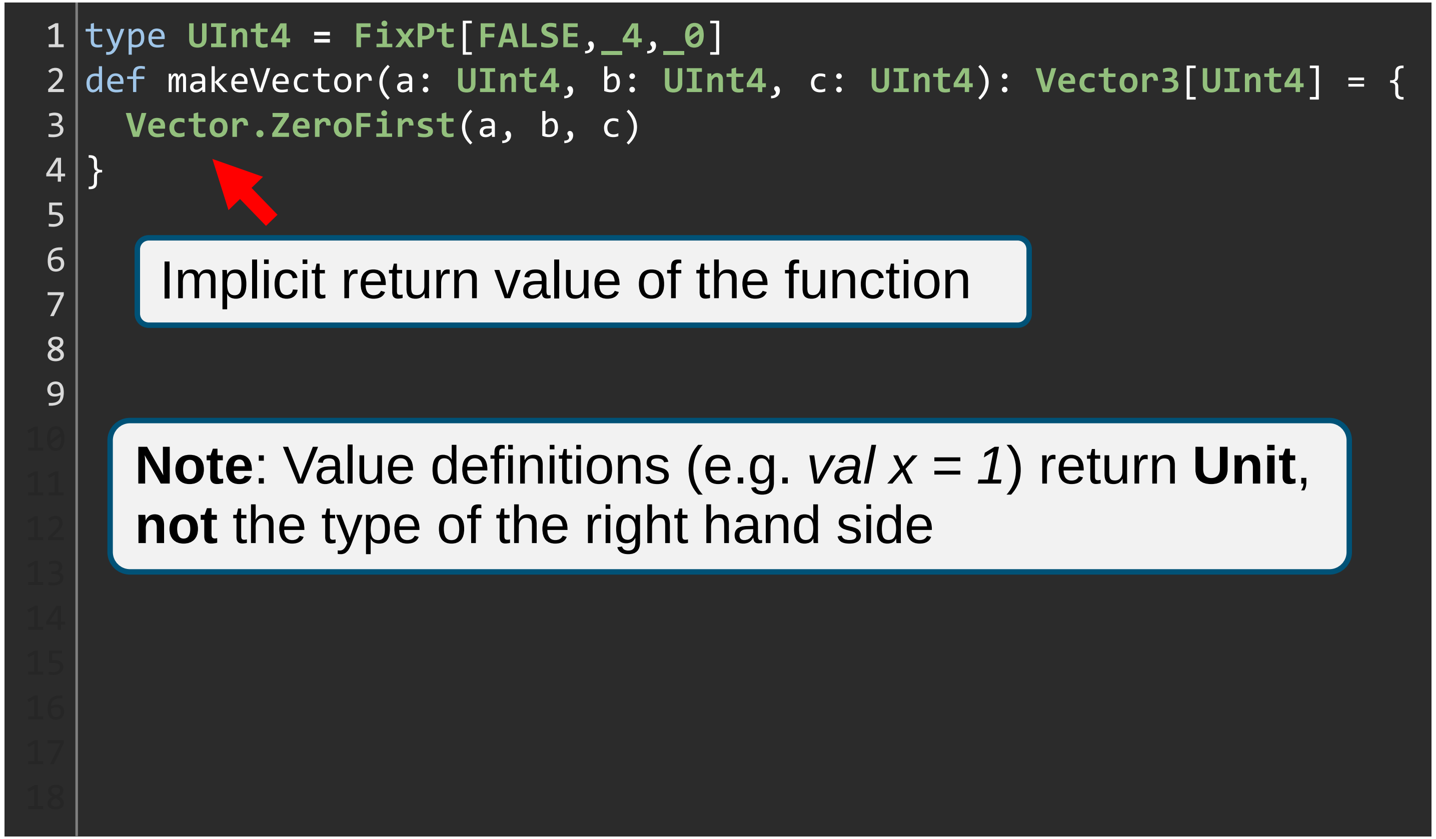
indices: 0 1 2

indices: 2 1 0

Equivalent behaviors, but slightly easier to remember (at least for me)

Simple Method



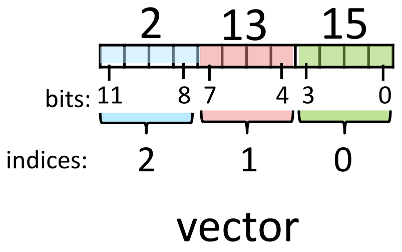


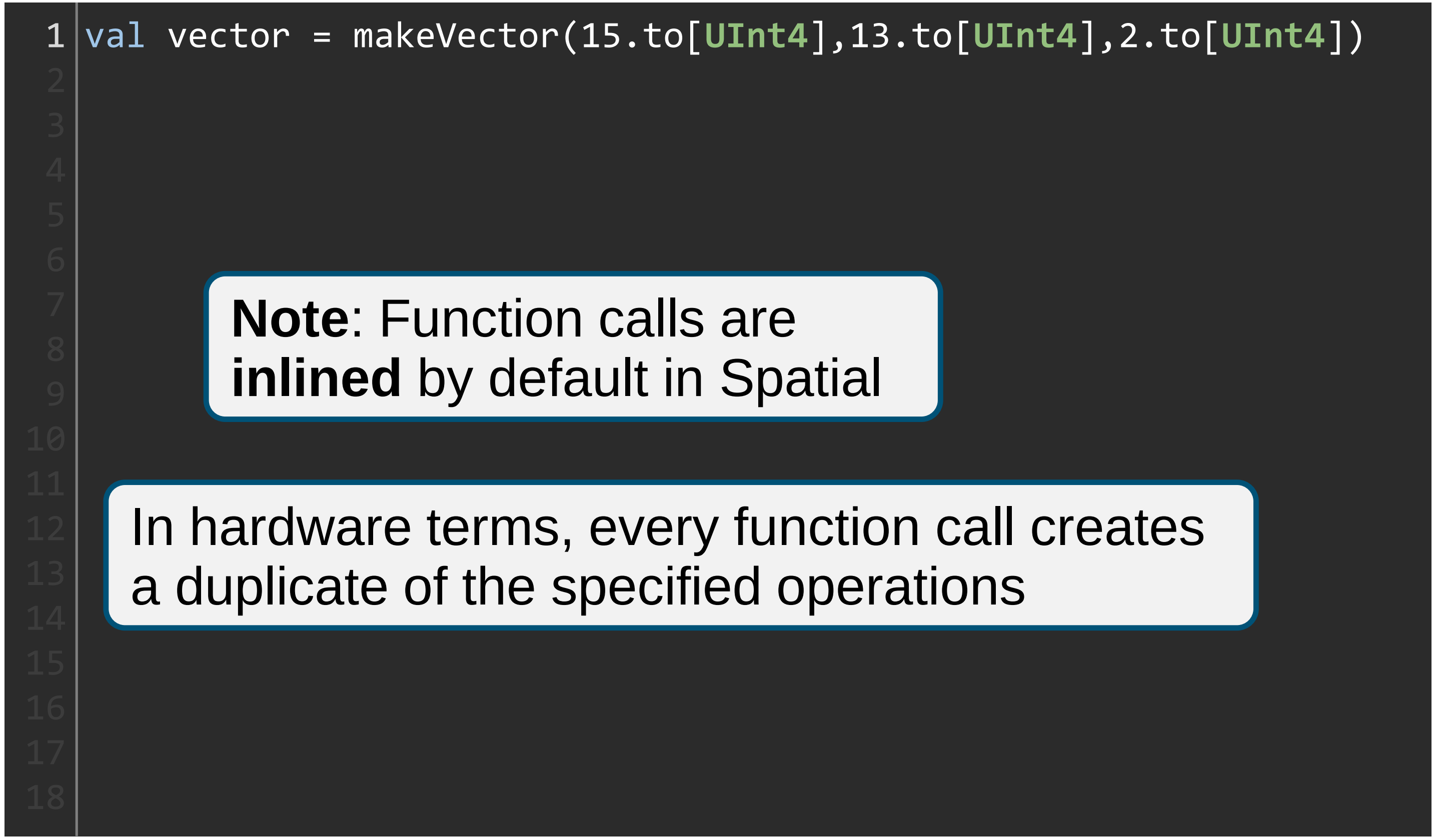
```
type UInt4 = FixPt[FALSE,_4,_0]
def makeVector(a: UInt4, b: UInt4, c: UInt4): Vector3[UInt4] = {
  Vector.ZeroFirst(a, b, c)
}
```

Implicit return value of the function

Note: Value definitions (e.g. *val x = 1*) return **Unit**, **not** the type of the right hand side

Method Call



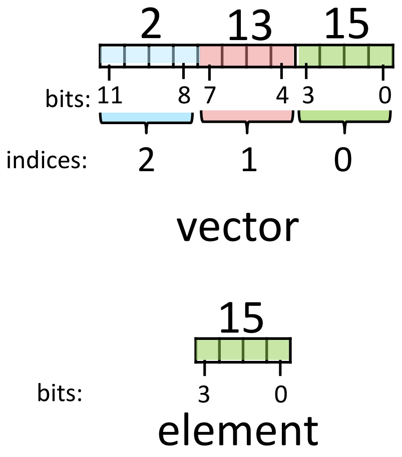


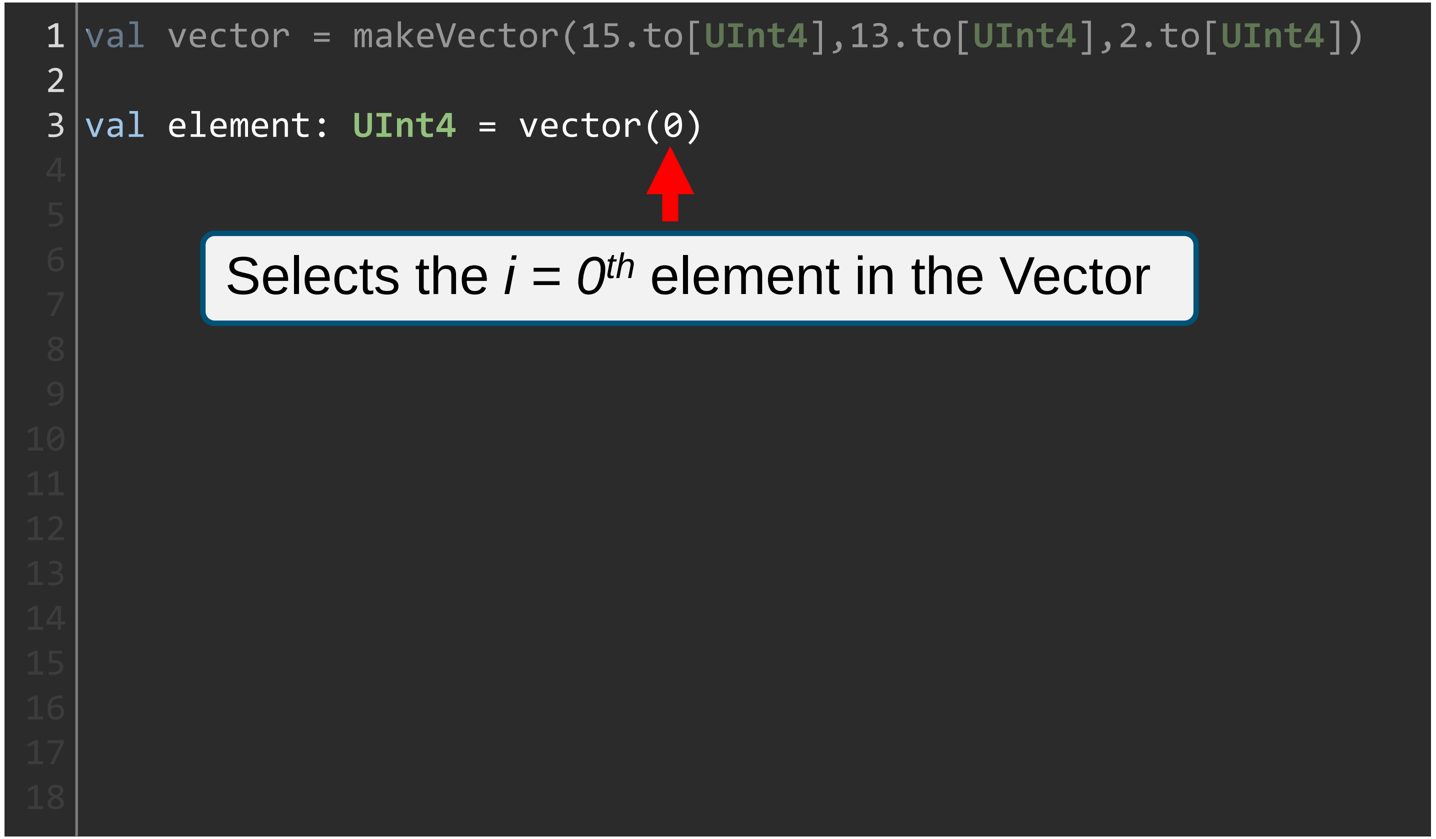
```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])
```

Note: Function calls are **inlined** by default in Spatial

In hardware terms, every function call creates a duplicate of the specified operations

Vector Operators: Element Select

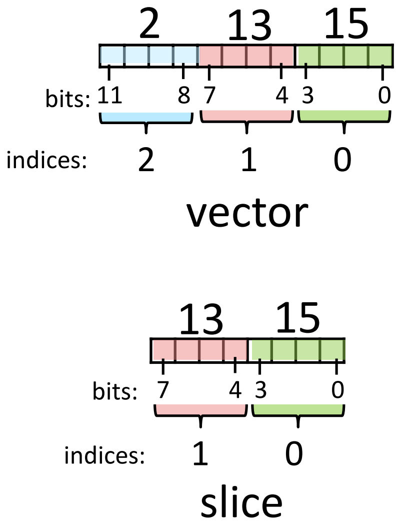


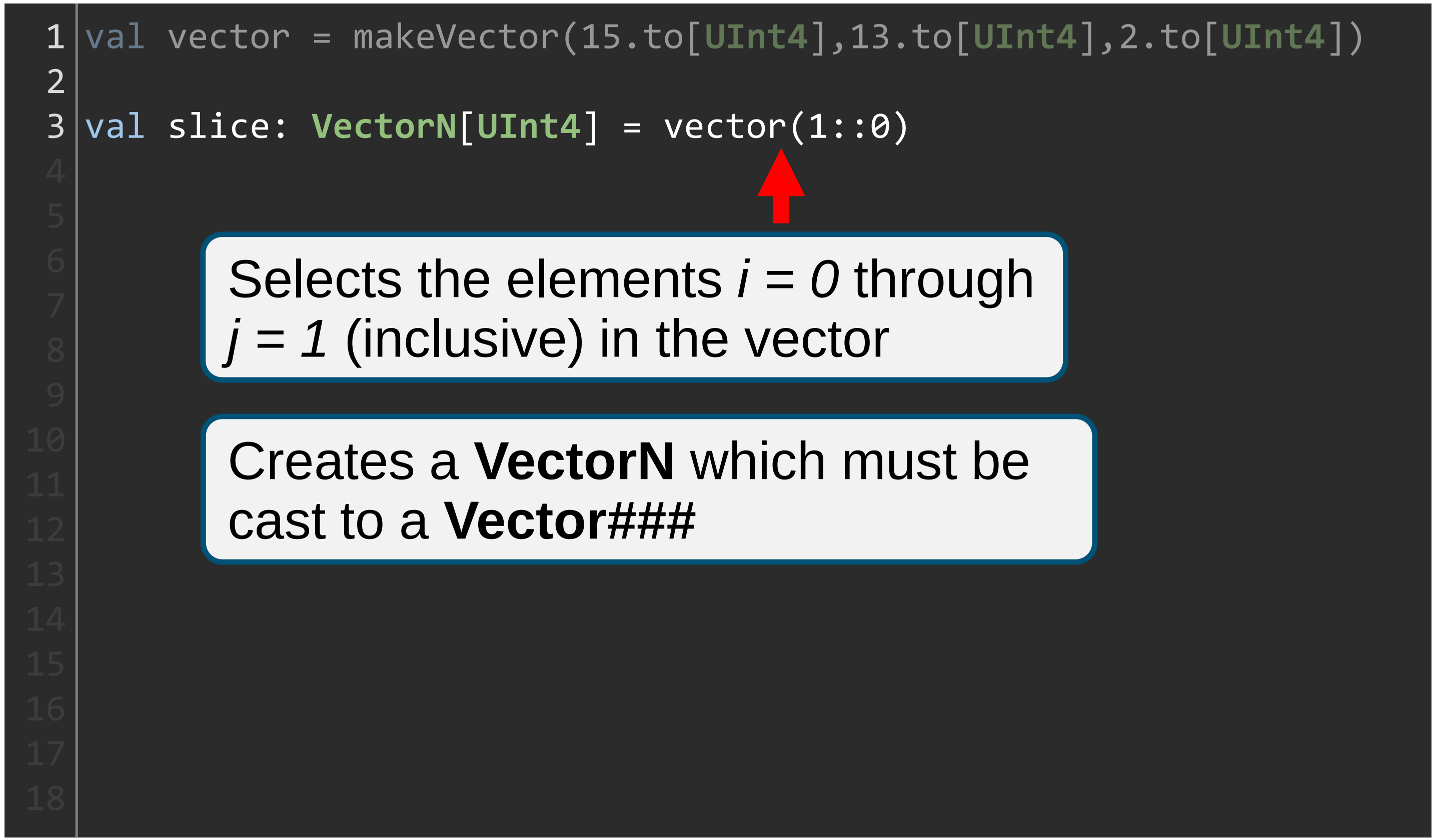


```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

val element: UInt4 = vector(0)
```

Selects the *i* = *0th* element in the Vector

Vector Operators: Element Slice



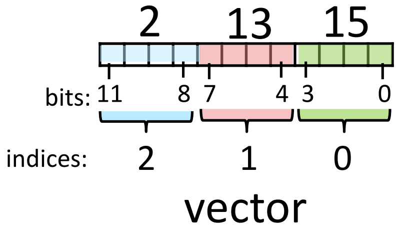


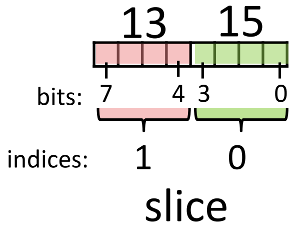
```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

val slice: VectorN[UInt4] = vector(1::0)
```

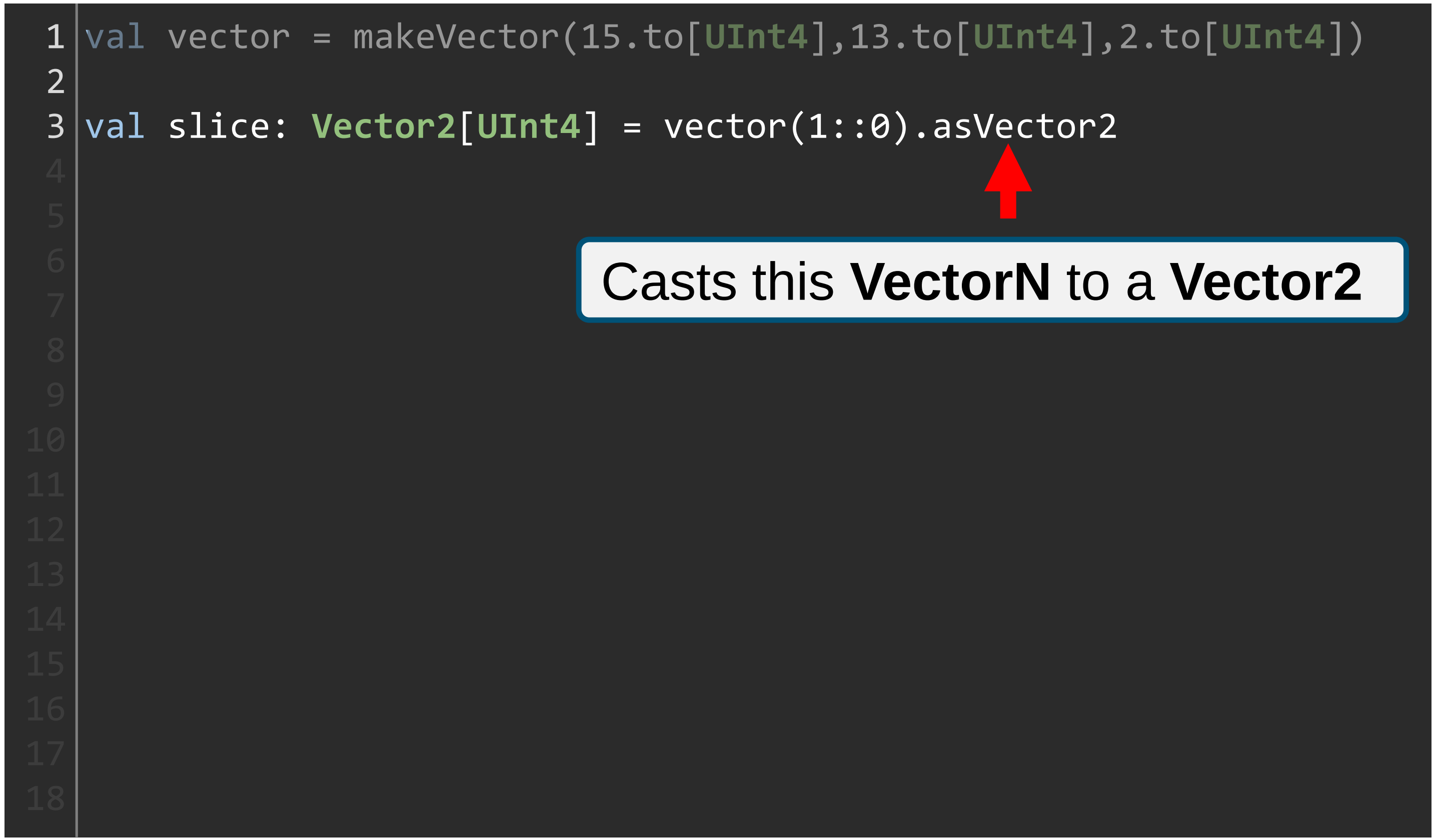
Selects the elements *i = 0* through *j = 1* (inclusive) in the vector

Creates a **VectorN** which must be cast to a **Vector###**

Vector Operators: Conversion



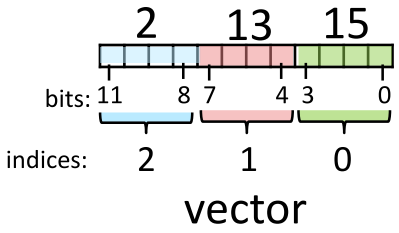


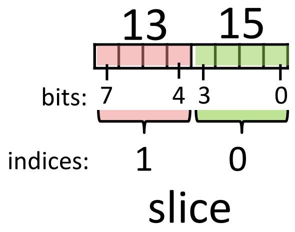


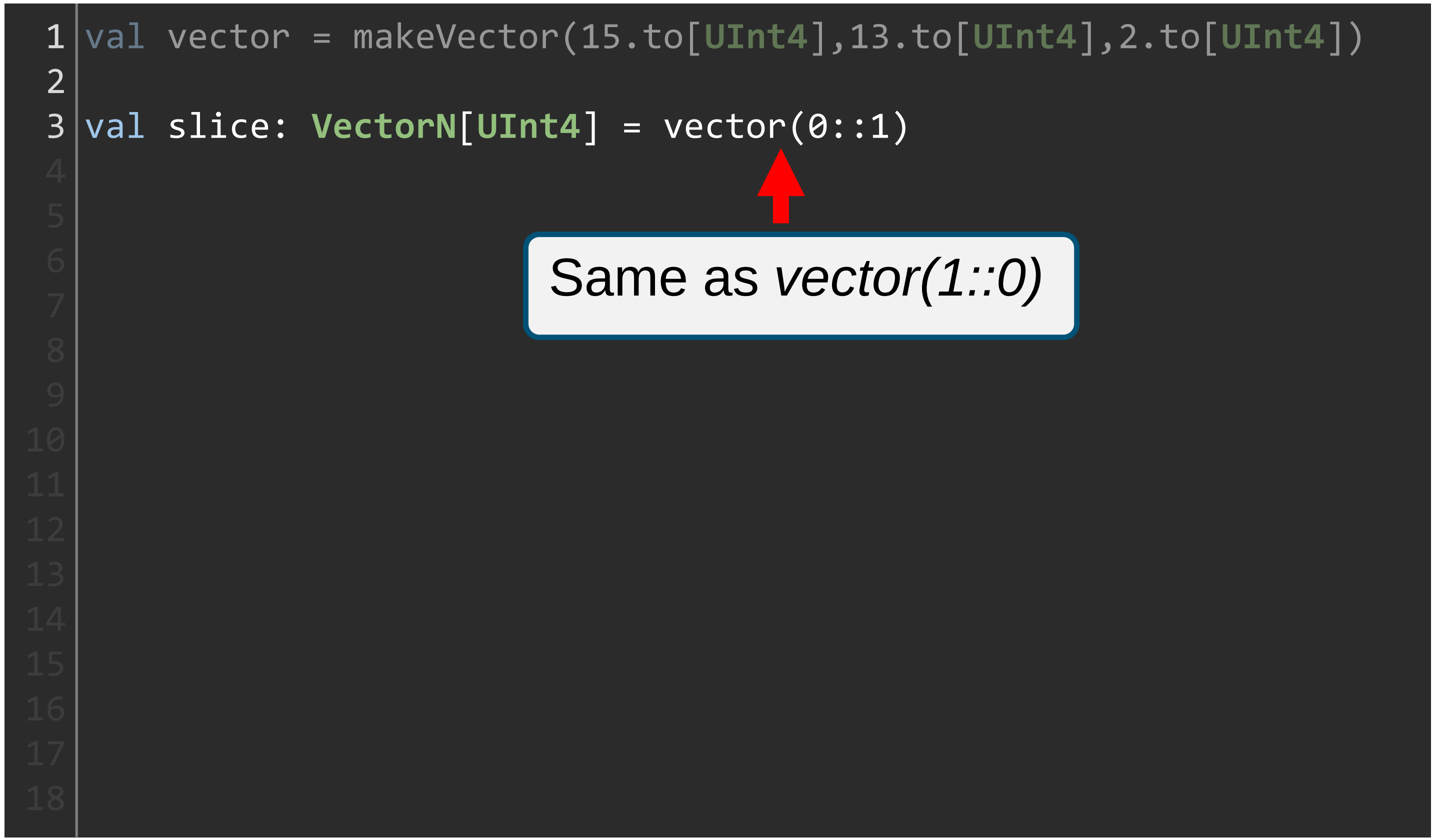
```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

val slice: Vector2[UInt4] = vector(1::0).asVector2
```

Casts this **VectorN** to a **Vector2**

Vector Operators: Element Slice



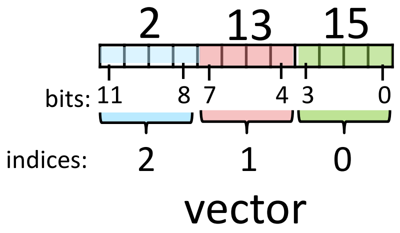


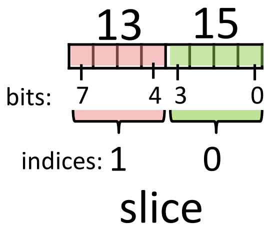


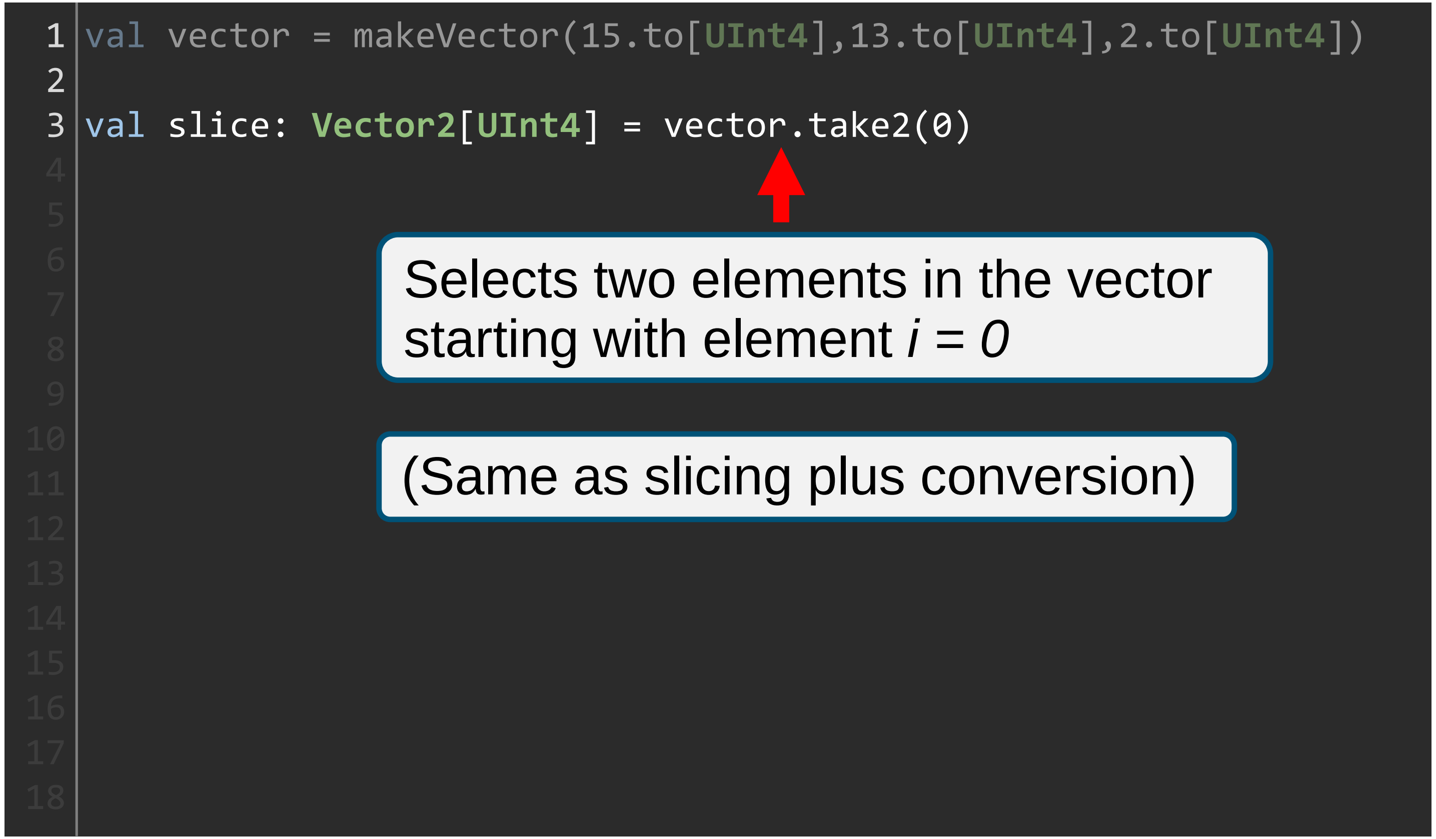
```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

val slice: VectorN[UInt4] = vector(0::1)
```

Same as *vector(1::0)*

Vector Operators: Element TakeN





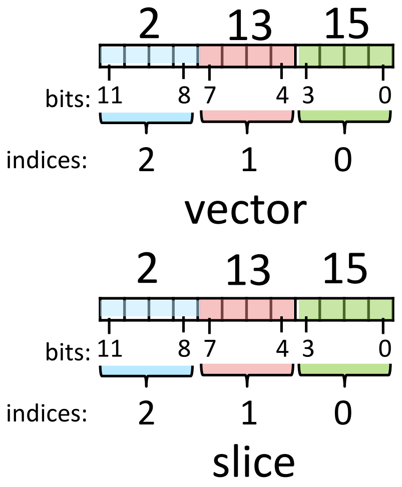


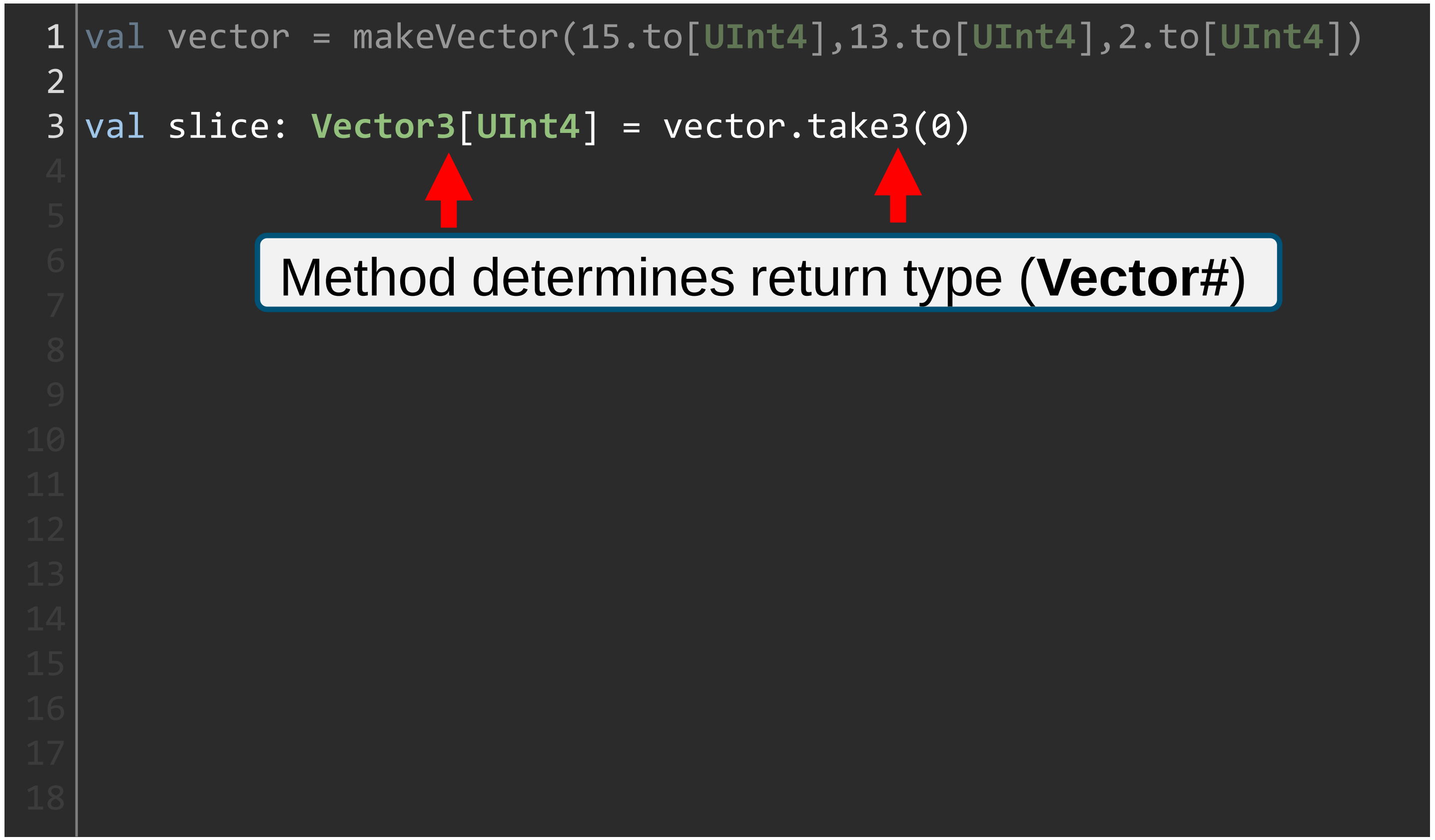
```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

val slice: Vector2[UInt4] = vector.take2(0)
```

Selects two elements in the vector starting with element *i = 0*

(Same as slicing plus conversion)

Vector Operators: Element TakeN

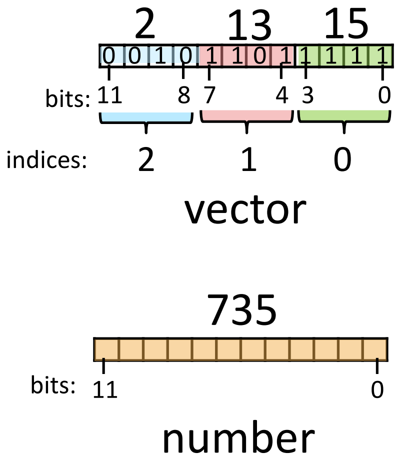


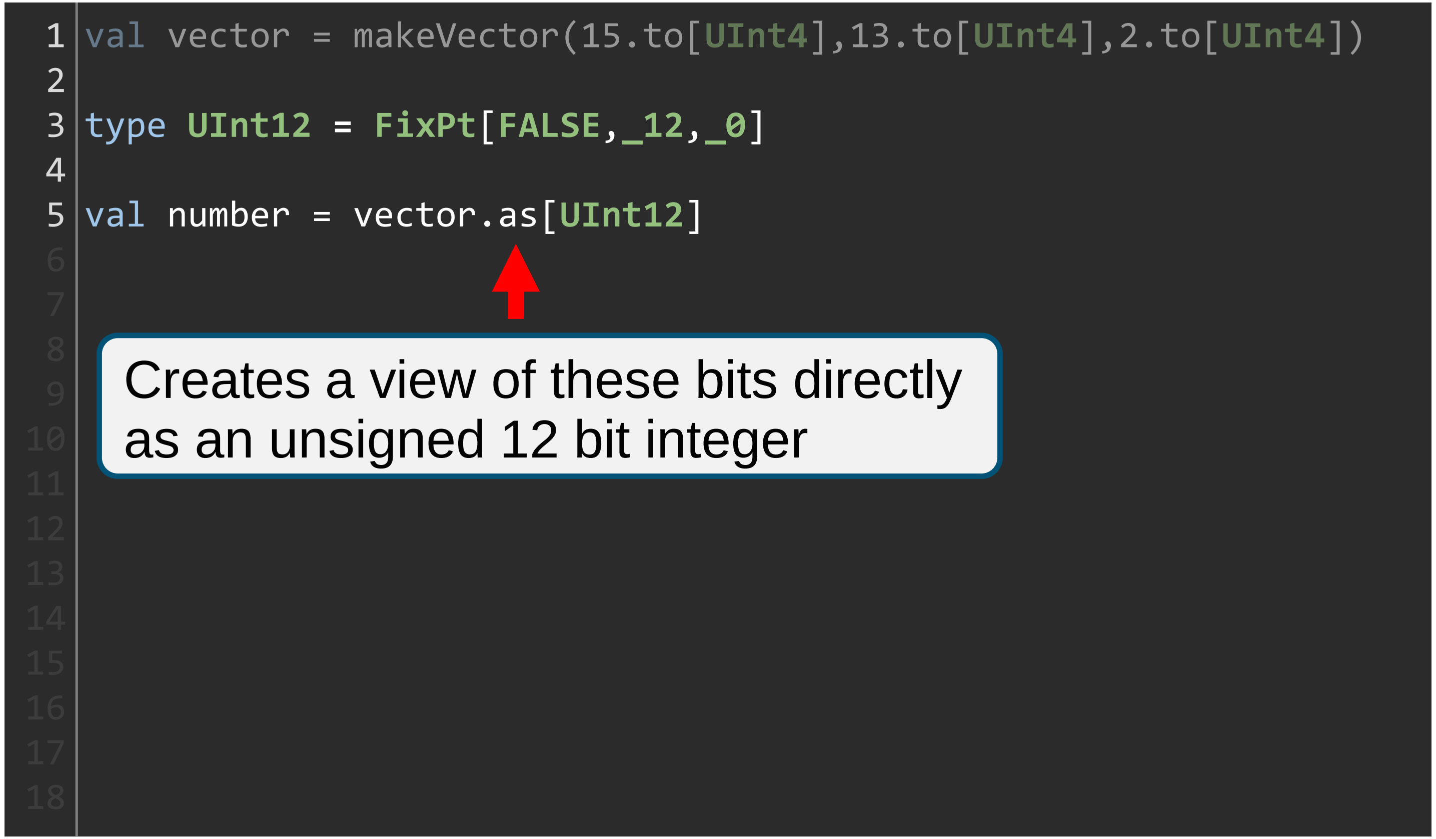


```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

val slice: Vector3[UInt4] = vector.take3(0)
```

Method determines return type (**Vector#**)

Vector Operators: Bit Casting

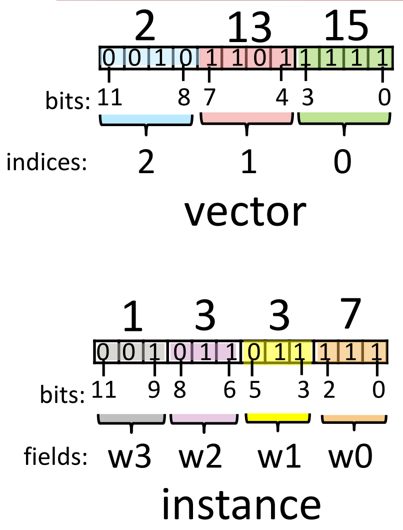


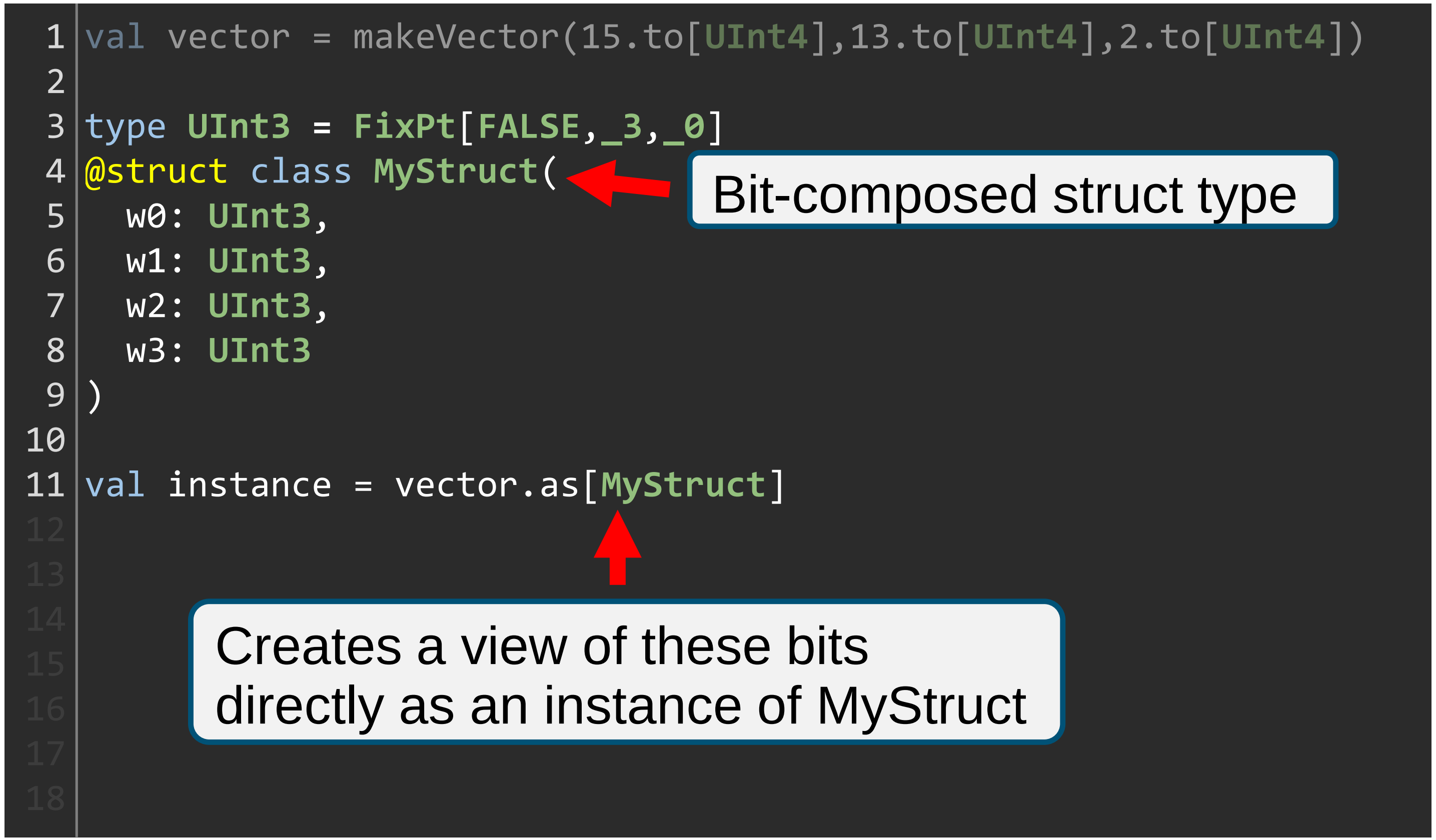


```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

type UInt12 = FixPt[FALSE,_12,_0]

val number = vector.as[UInt12]
```

Creates a view of these bits directly as an unsigned 12 bit integer

Vector Operators: Bit Casting



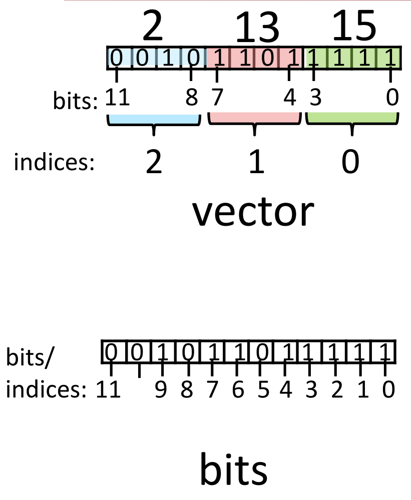


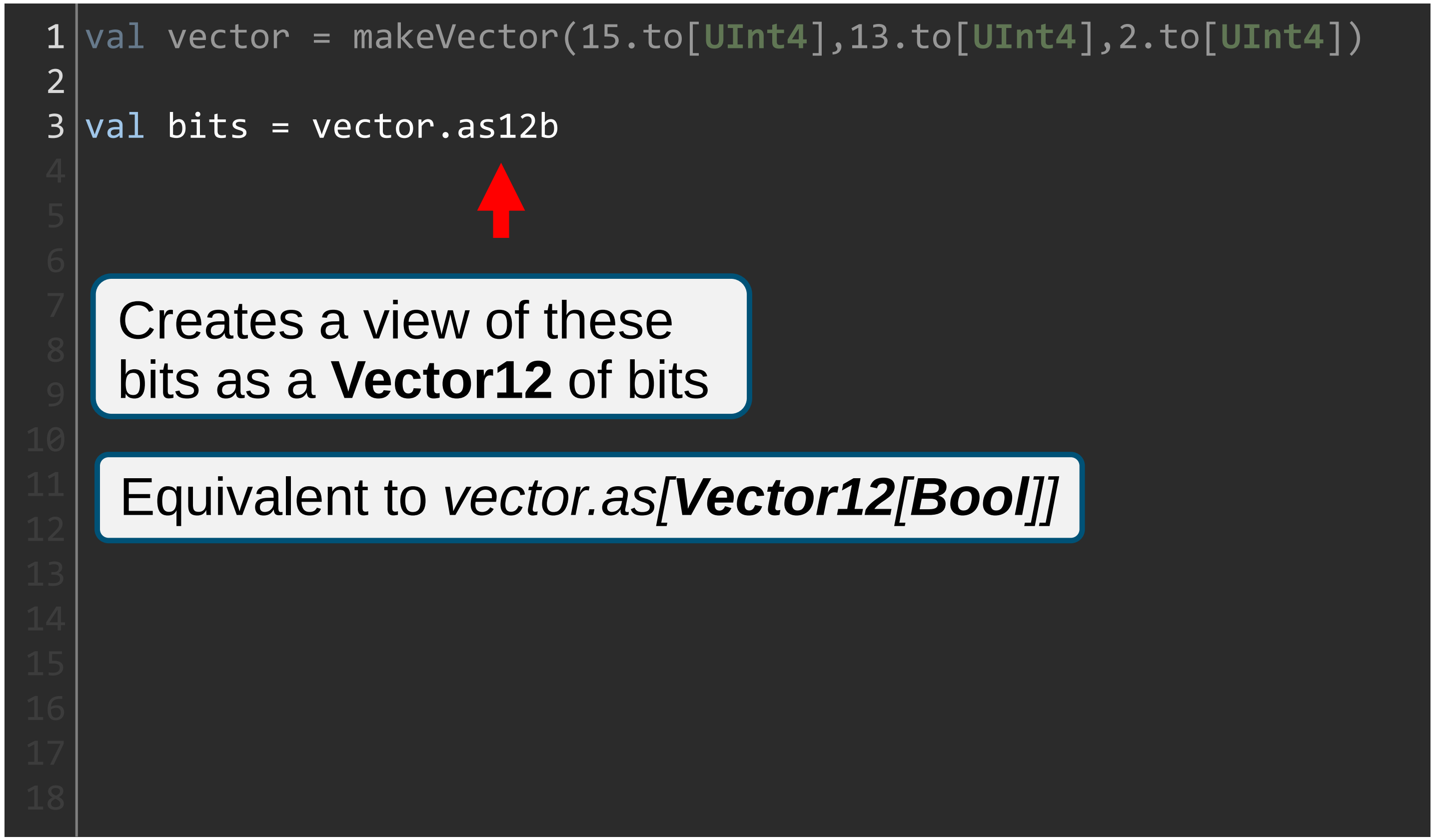
```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

type UInt3 = FixPt[FALSE,_3,_0]
@struct class MyStruct(
  w0: UInt3,
  w1: UInt3,
  w2: UInt3,
  w3: UInt3
)

val instance = vector.as[MyStruct]
```

Bit-composed struct type

Creates a view of these bits
directly as an instance of MyStruct

Vector Operations: As Vector Of Bits





```
val vector = makeVector(15.to[UInt4],13.to[UInt4],2.to[UInt4])

val bits = vector.as12b
```

Creates a view of these bits as a **Vector12** of bits

Equivalent to *vector.as[**Vector12[Bool]**]*

All Bit Primitives Are Bit Vectors

```
type UInt32 = FixPt[FALSE,_16,_0]
type UInt16 = FixPt[FALSE,_16,_0]
@struct class Split16(msByte: UInt16, lsByte: UInt16)

val a = 10.to[UInt32]

val bit3 = a(3)

val lsByte = a(17::2).asVector16

val bits = a.as32b

val split = a.as[Split16]

val a_again = split.as[UInt32]
```

4th least significant bit of *a*

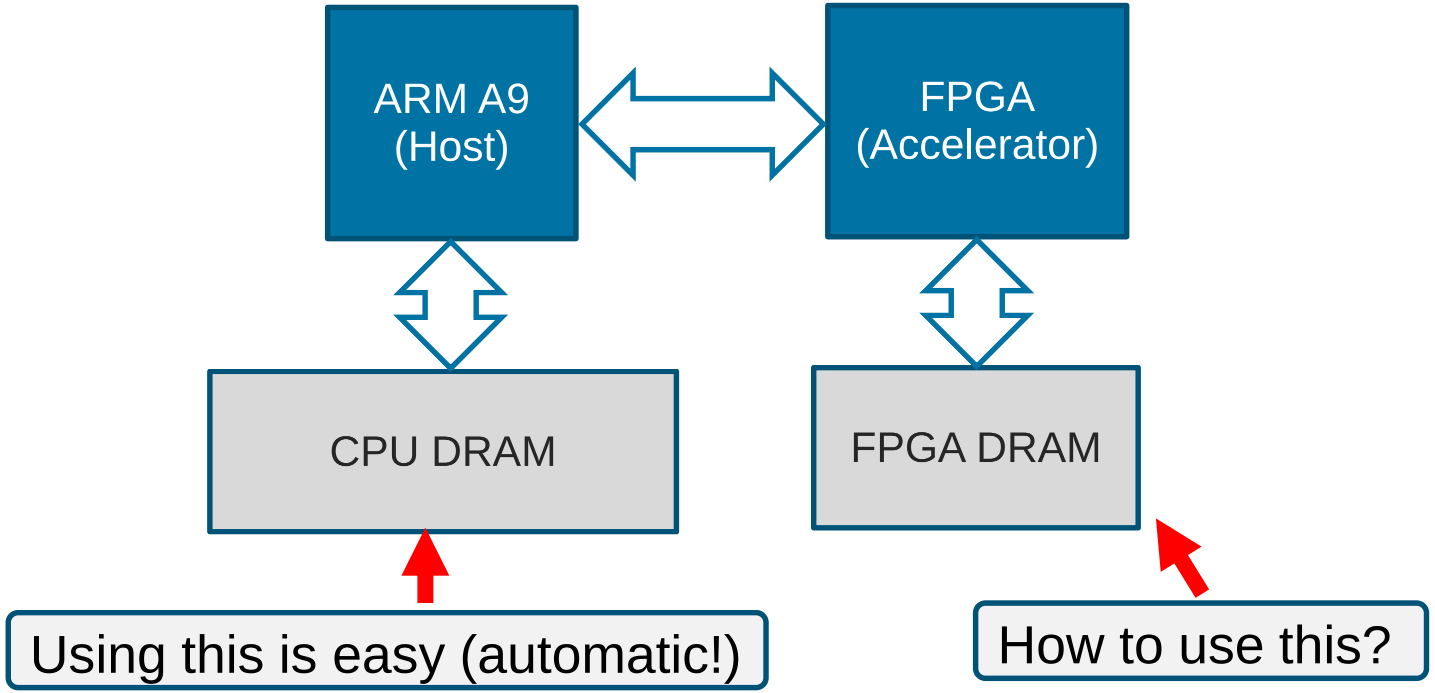
bit slice of *a*

vector view of all bits in *a*

Split16 view of *a*

OFF-CHIP MEMORIES

Previously on *Spatial*…

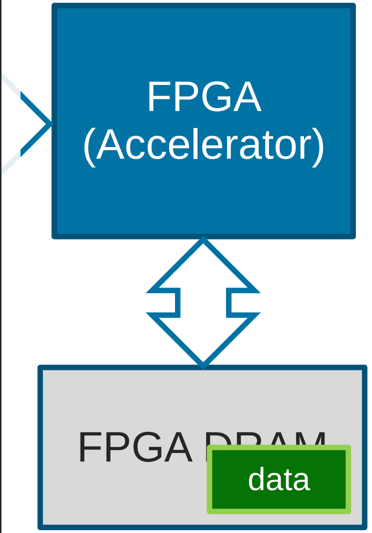


DRAM

```
val data = DRAM[Int](192, 192)

Accel {
  …
}
```

Declares a 192 x 192 (2D) block of memory in the FPGA DRAM with words of type **Int**

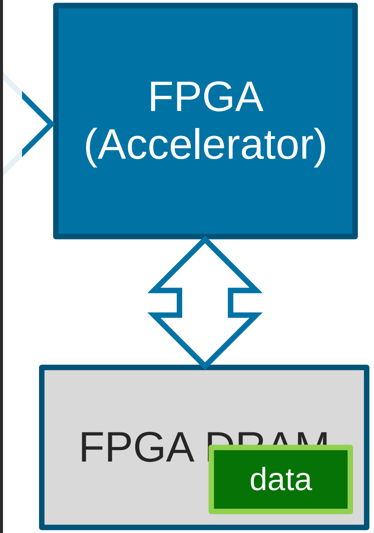
1 to 5 dimensions are currently supported



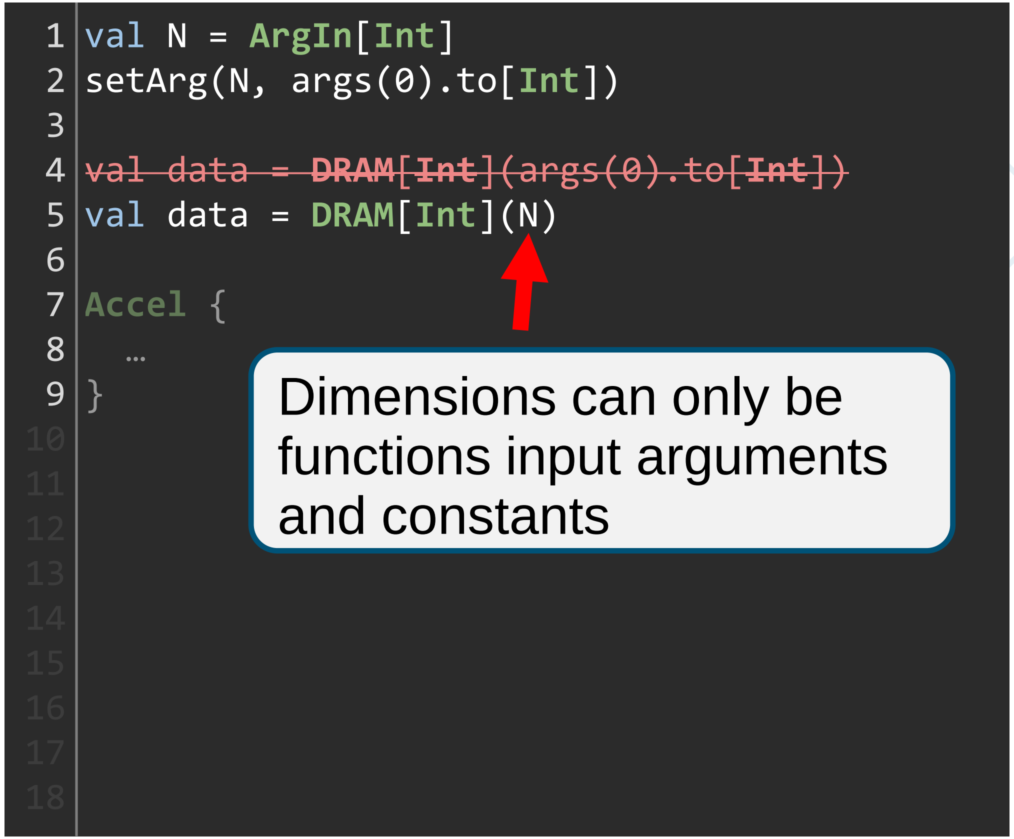
DRAM

```
val data = DRAM[Int](192, 192)

Accel {
  …
}
```

Must be used **outside** of the *Accel* scope

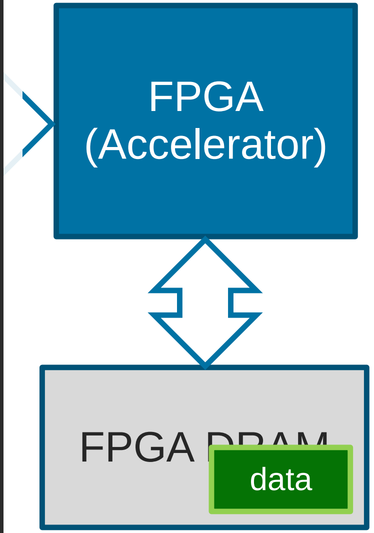


DRAM: Dimension Limitations



```
val N = ArgIn[Int]
setArg(N, args(0).to[Int])

~~val data = DRAM[Int](args(0).to[Int])~~
val data = DRAM[Int](N)

Accel {
   …
}
```

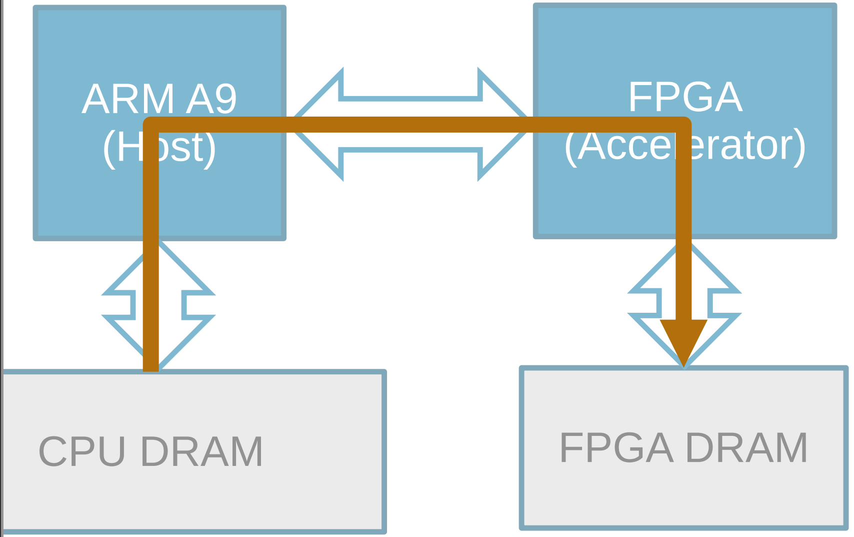
Dimensions can only be functions input arguments and constants



DRAM: Transfer from ARM to FPGA

```
val N = ArgIn[Int]
setArg(N, args(0).to[Int])
val data = DRAM[Int](N)

val armData: Array[Int] = … //Info soon!

setMem(data, array)

Accel {
  …
}
```

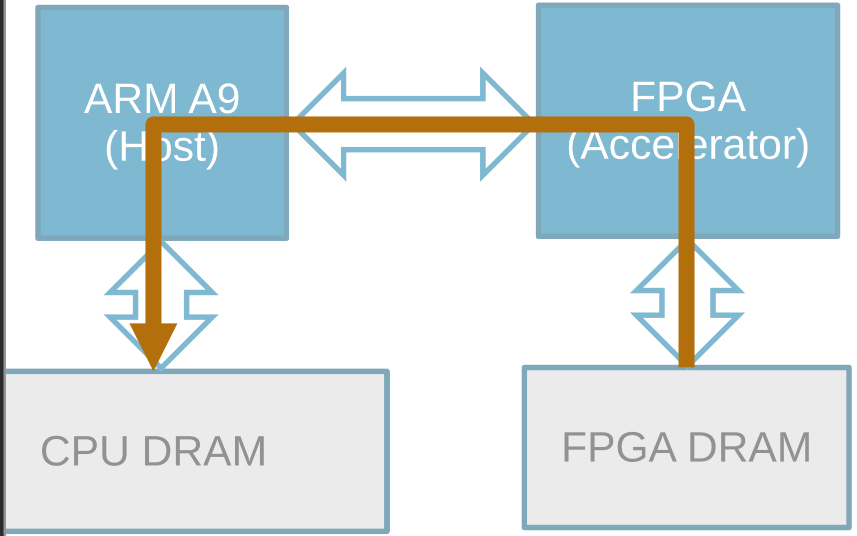
Copies data from ARM DRAM to FPGA DRAM



DRAM: Transfer from FPGA to ARM

```
val N = ArgIn[Int]
setArg(N, args(0).to[Int])
val data = DRAM[Int](N)

val armData: Array[Int] = … //Info soon!

setMem(data, array)

Accel {
  …
}

val outputData = getMem(data)
```

Copies data from FPGA DRAM to ARM DRAM

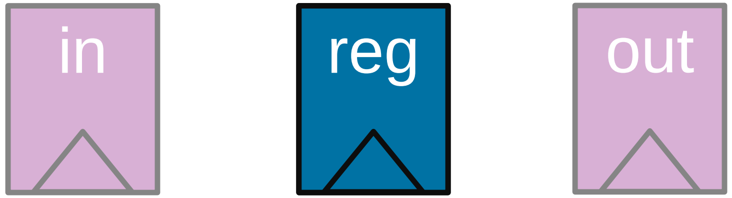


ON-CHIP MEMORIES

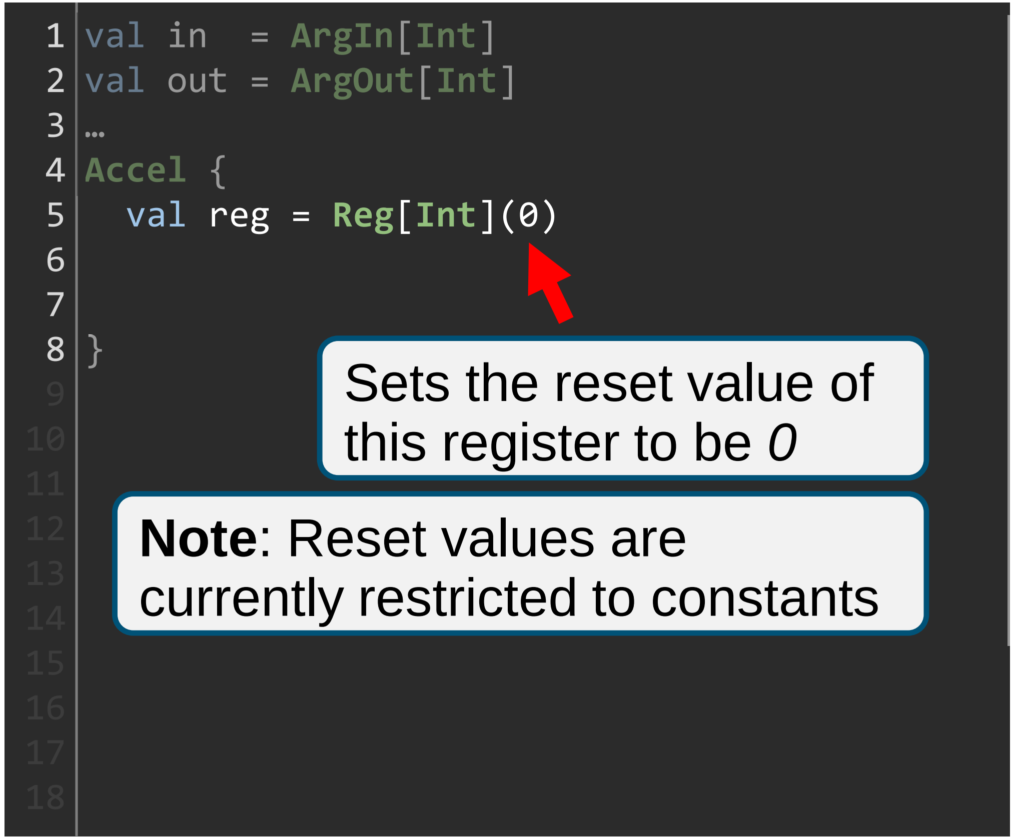
Reg

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int]


}
```

Creates a **Reg** which holds a value of type **Int**



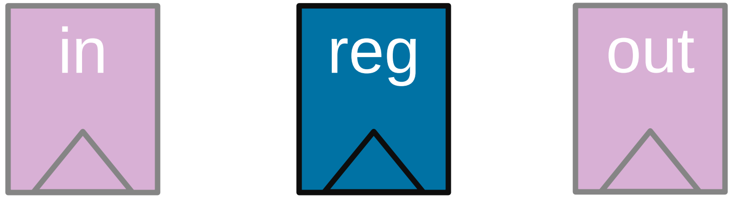
Reg Reset Value



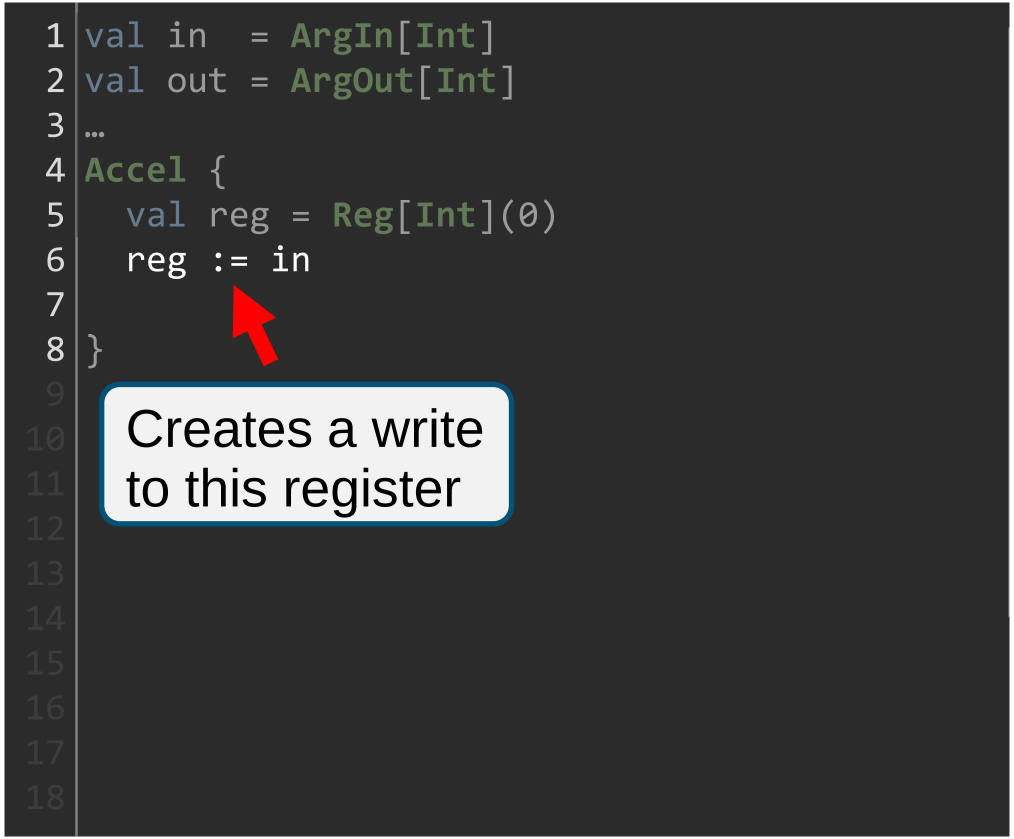
```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int](0)


}
```

Sets the reset value of this register to be *0*

Note: Reset values are currently restricted to constants

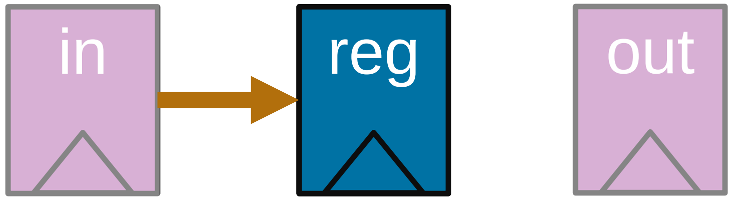


Reg Writing



```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int](0)
  reg := in

}
```

Creates a write to this register

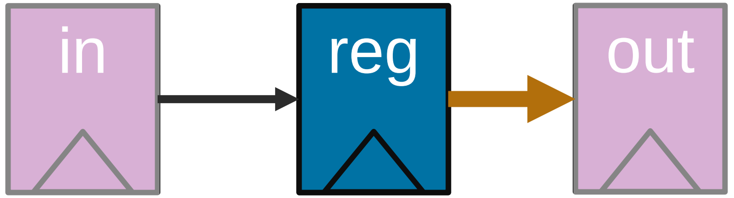


Reg Reading

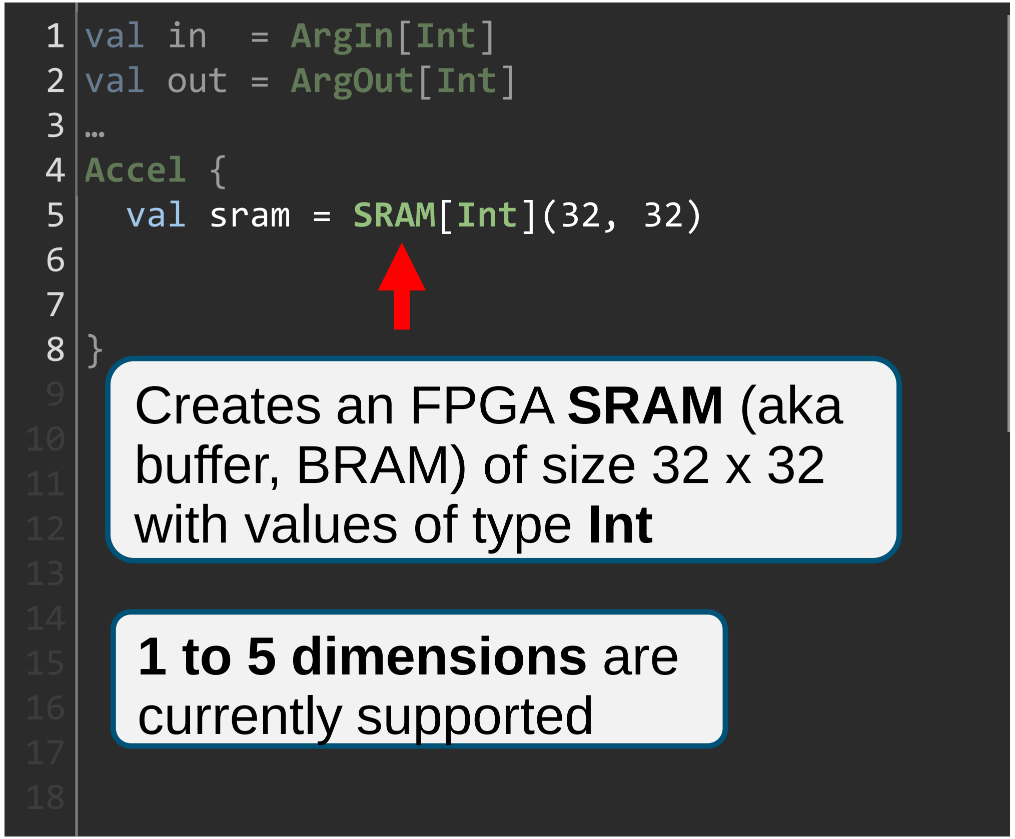
```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int](0)
  reg := in
  out := reg.value
}
```

Creates wires connected to the output of this register

Note: Register reads are normally implicit, but can be written explicitly



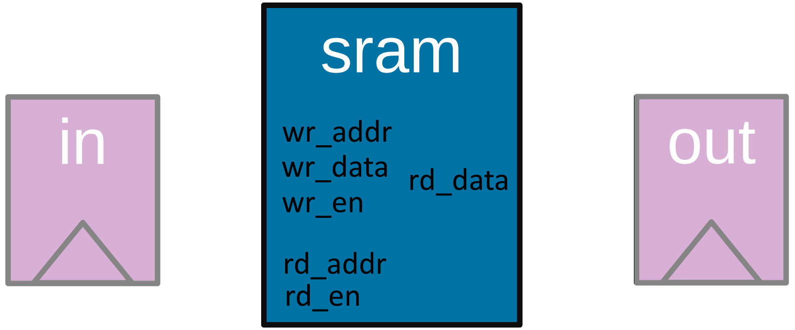
SRAM



```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val sram = SRAM[Int](32, 32)


}
```

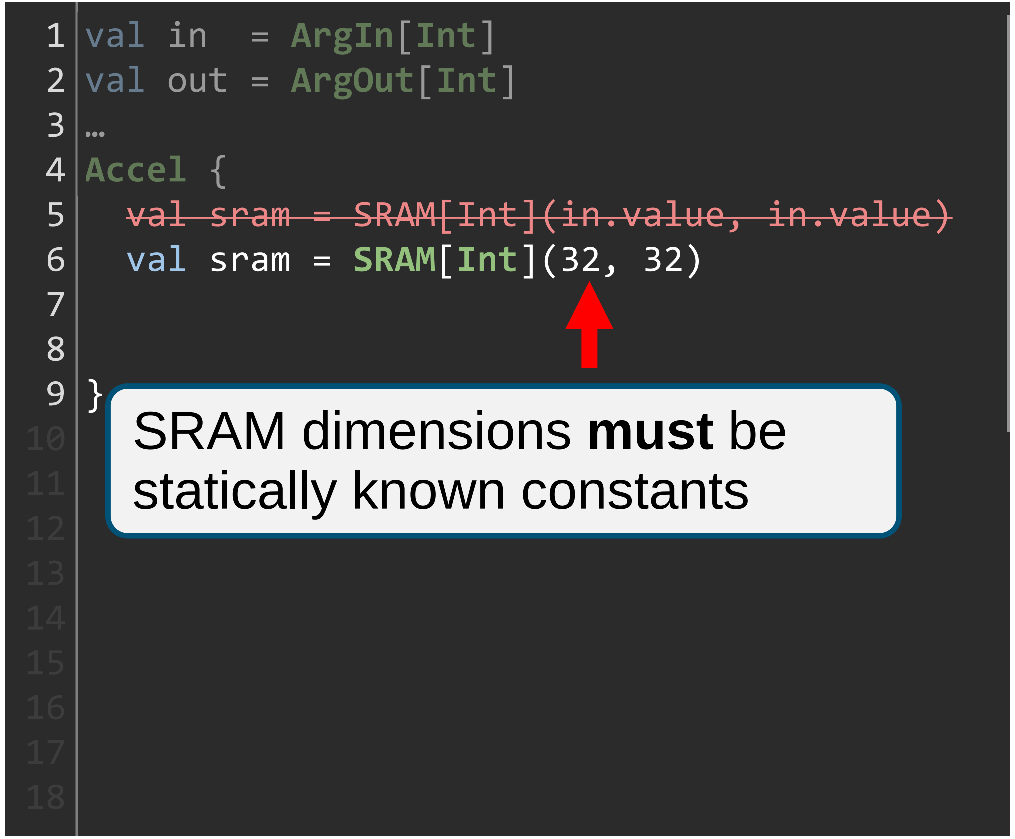
Creates an FPGA **SRAM** (aka buffer, BRAM) of size 32 x 32 with values of type **Int**

1 to 5 dimensions are currently supported



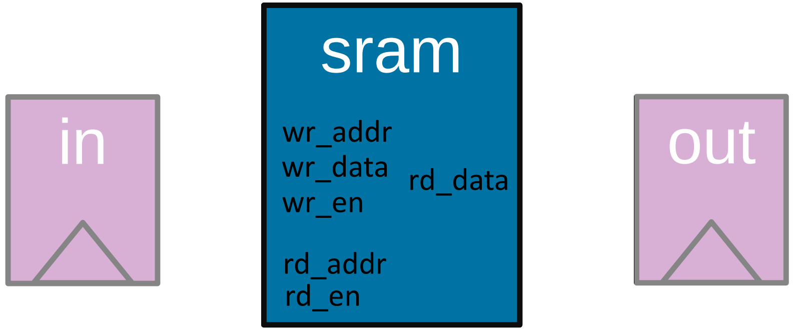
Note: SRAM is implemented on the DE1 SoC with FPGA M10Ks: **10Kb blocks with 40 bit ports**

SRAM



```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  ~~val sram = SRAM[Int](in.value, in.value)~~
  val sram = SRAM[Int](32, 32)


}
```

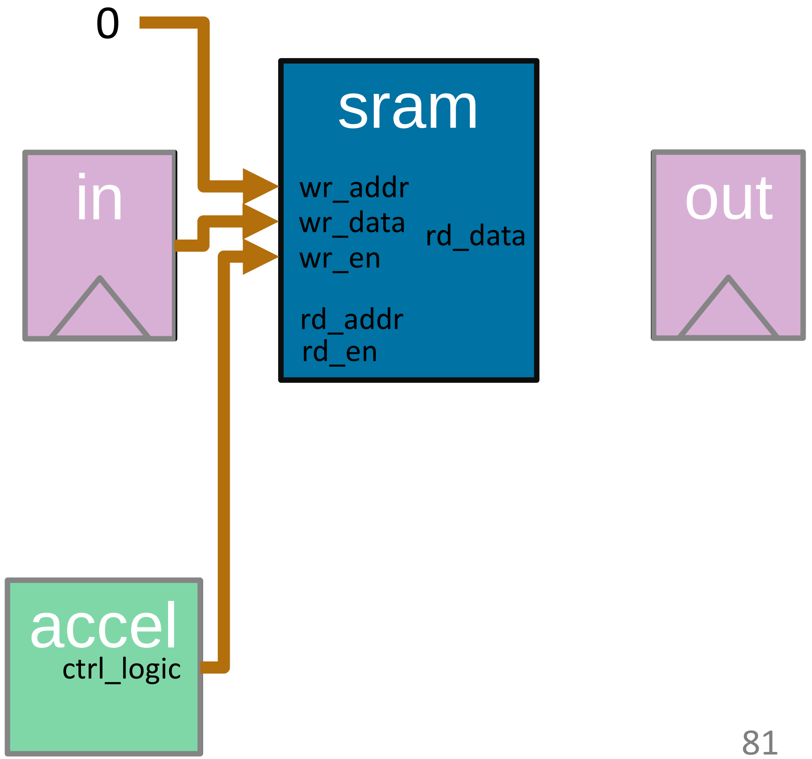
SRAM dimensions **must** be statically known constants



SRAM Writes

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val sram = SRAM[Int](32, 32)
  sram(0, 0) = in.value

}
```

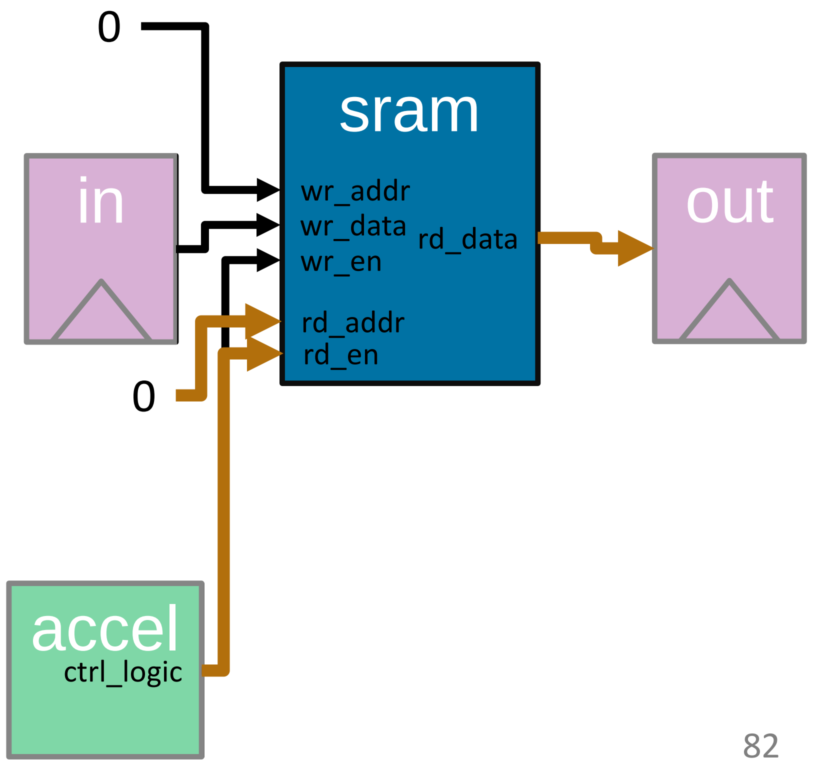
Creates an SRAM write port with data *in.value*, address *0* which is enabled by *Accel*



SRAM Reads

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val sram = SRAM[Int](32, 32)
  sram(0, 0) = in.value
  out := sram(0,0)
}
```

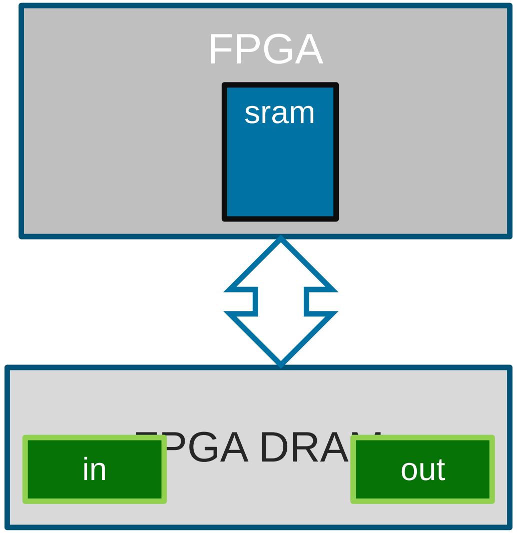
Creates an SRAM read port with address *0* which is enabled by *Accel*



SRAM: Interfacing with DRAM?

```
val in  = DRAM[Int](32)
val out = DRAM[Int](32)
…
Accel {
  val sram = SRAM[Int](16)


}
```

How can we copy data to/from FPGA DRAM?

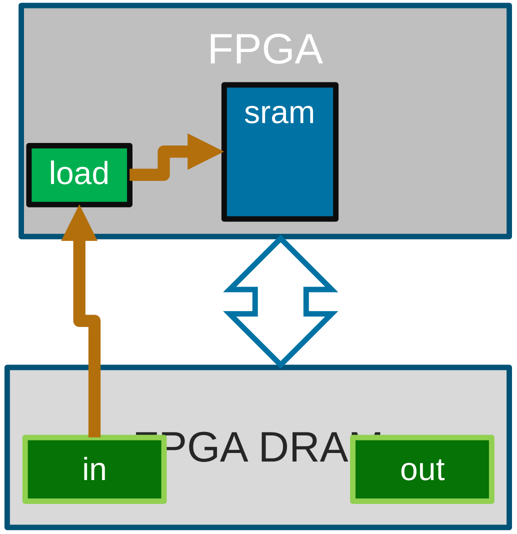


SRAM: Dense Loading from DRAM

```
val in  = DRAM[Int](32)
val out = DRAM[Int](32)
…
Accel {
  val sram = SRAM[Int](16)
  sram load in(0::16)

}
```

Creates logic which loads data within *in*, address range **0 until 16** (exclusive), to *sram*

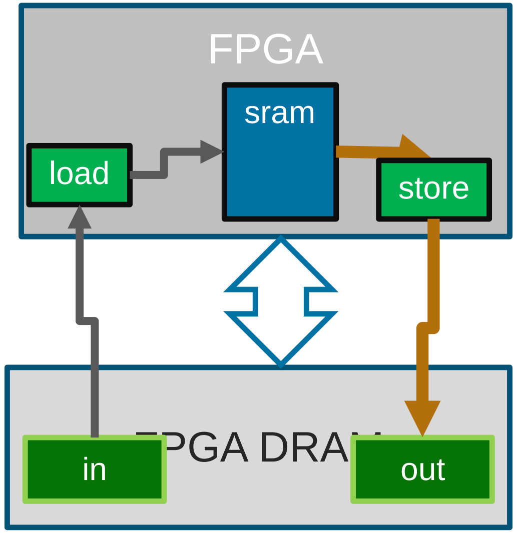
Note: The address range can be omitted if SRAM and DRAM are the same size



SRAM: Dense Storing to DRAM

```
val in  = DRAM[Int](32)
val out = DRAM[Int](32)
…
Accel {
  val sram = SRAM[Int](16)
  sram load in(0::16)
  out(0::16) store sram
}
```

Creates logic which stores contents of *sram* to *out's* address range **0 until 16** (exclusive)

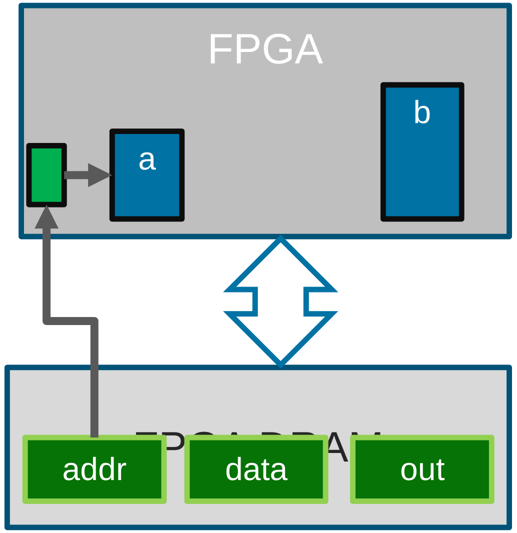


SRAM: Gather from DRAM

```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses

}
```

Equivalent C:

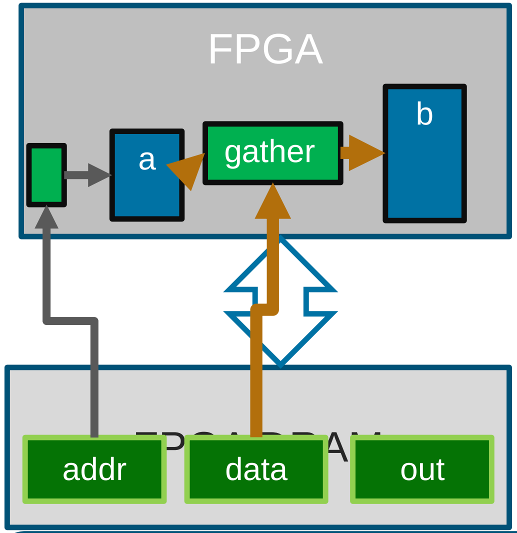
```
for (i=0; i<16; i++) {
  b[i] = data[a[i]]
}
```



SRAM: Gather from DRAM

```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  b gather data(a)
}
```

Creates logic which gathers elements in *data* at addresses in *a* into *b*

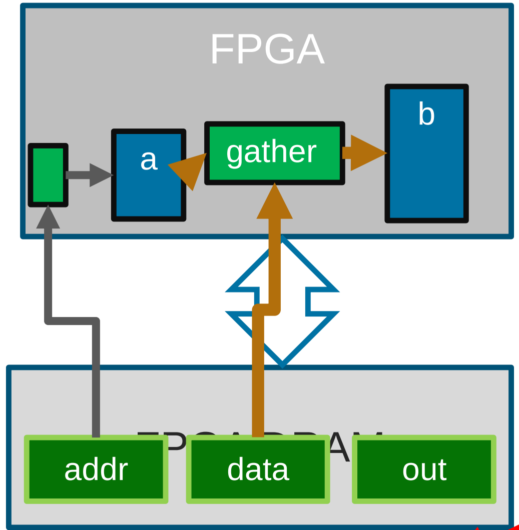


```
for (i=0; i<16; i++) {
  b[i] = data[a[i]]
}
```

SRAM: Gather from DRAM

```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  b gather data(a, 10)
}
```

Uses the first 10 elements in a only



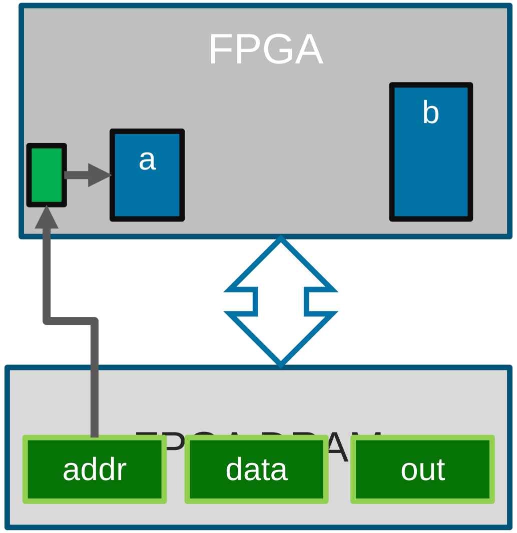
```
for (i=0; i<10; i++) {
  b[i] = data[a[i]]
}
```

SRAM: Scatter to DRAM

```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses

}
```

Equivalent C:

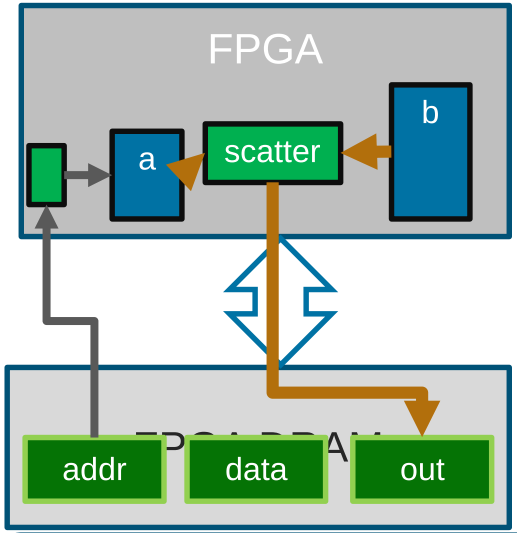
```
for (i=0; i<16; i++) {
  out[a[i]] = b[i]
}
```



SRAM: Scatter to DRAM

```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  out(a) scatter b
}
```

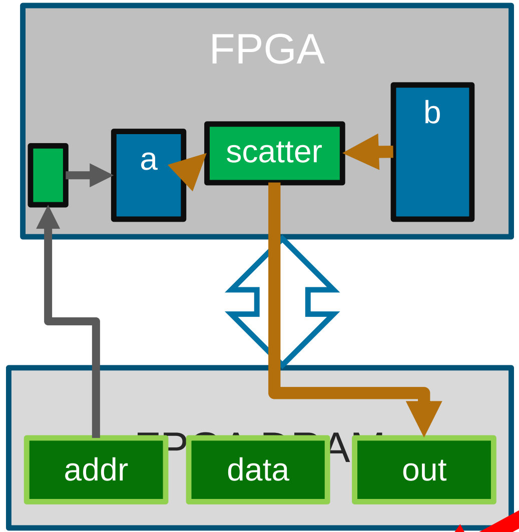
Creates logic which scatters elements in *b* into *out* at addresses in *a*



```
for (i=0; i<16; i++) {
  out[a[i]] = b[i]
}
```

SRAM: Scatter to DRAM

```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  out(a, 10) scatter b
}
```



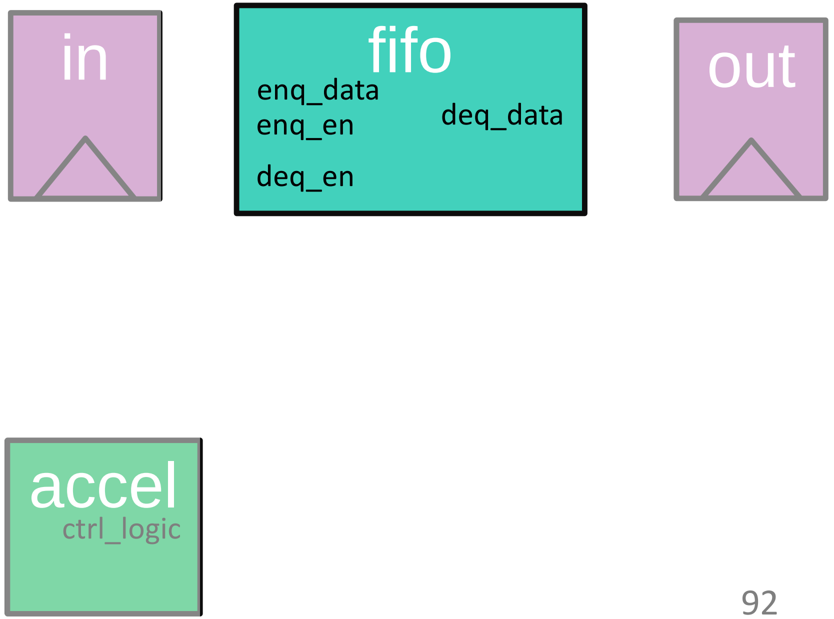
```
for (i=0; i<10; i++) {
  out[a[i]] = b[i]
}
```

FIFO

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val fifo = FIFO[Int](16)


}
```

FIFO with depth 16

Note: Depth must be statically known

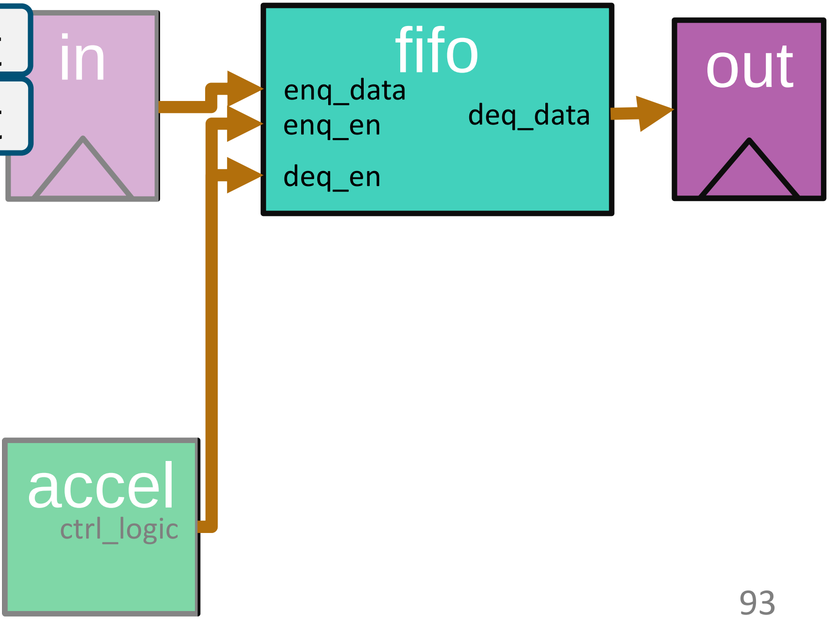


FIFO: Enqueueing / Dequeueing

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val fifo = FIFO[Int](16)
  a.enq(scalarIn)
  scalarOut := a.deq()
}
```

FIFO enqueue port

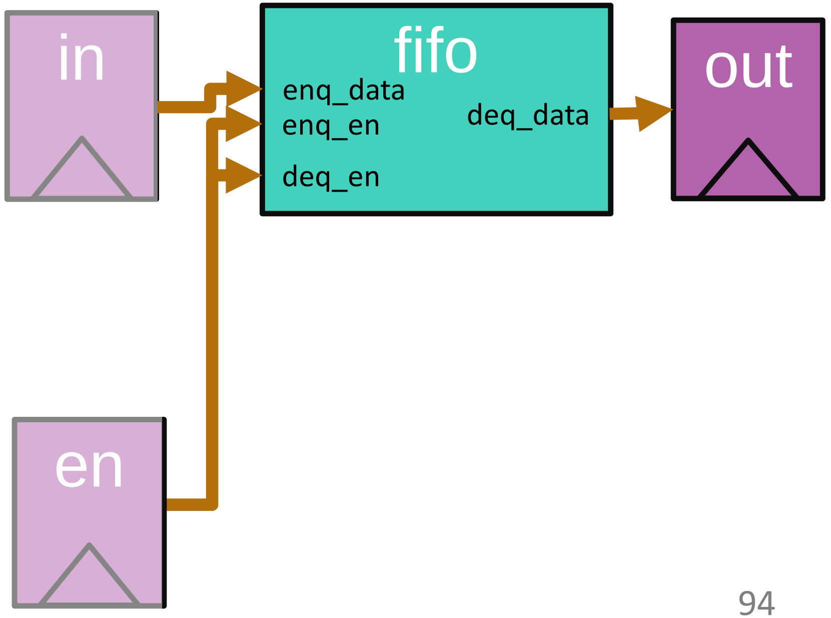
FIFO dequeue port



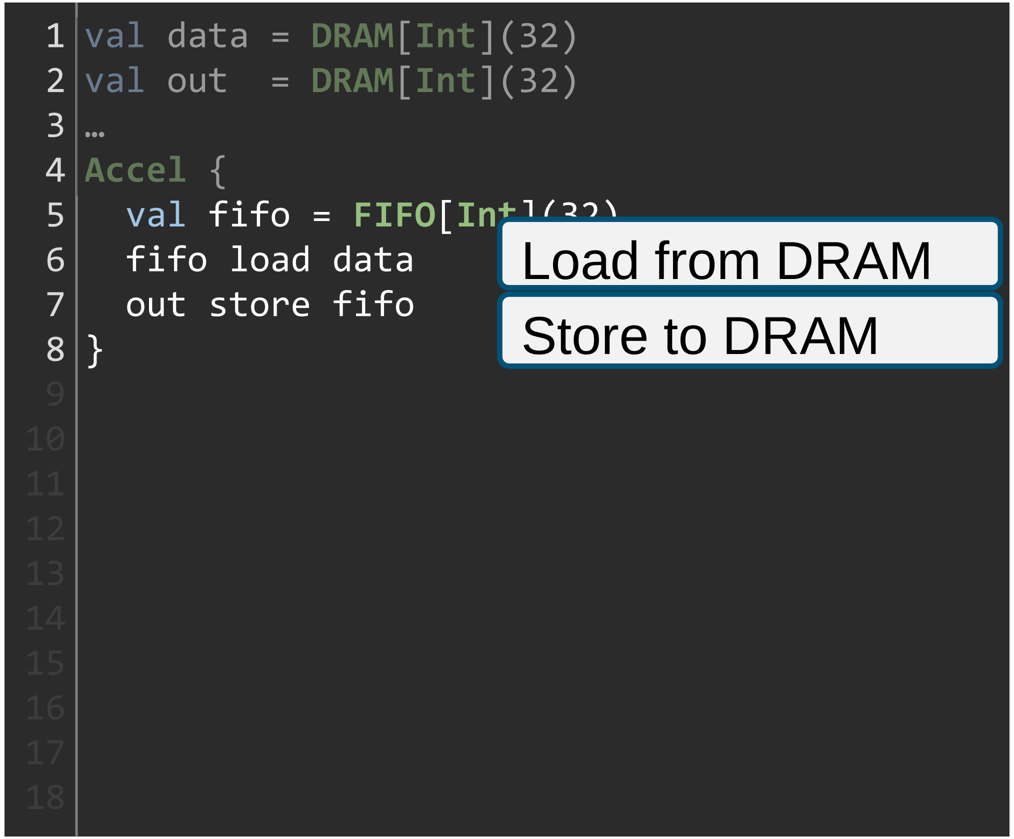
FIFO: Enabled Enqueuing/Dequeueing

```
val in  = ArgIn[Int]
val en  = ArgIn[Bool]
val out = ArgOut[Int]
…
Accel {
  val fifo = FIFO[Int](16)
  a.enq(scalarIn, en)
  scalarOut := a.deq(en)
}
```

Can also set data-dependent enables for enqueue/dequeue e.g. for data filtering



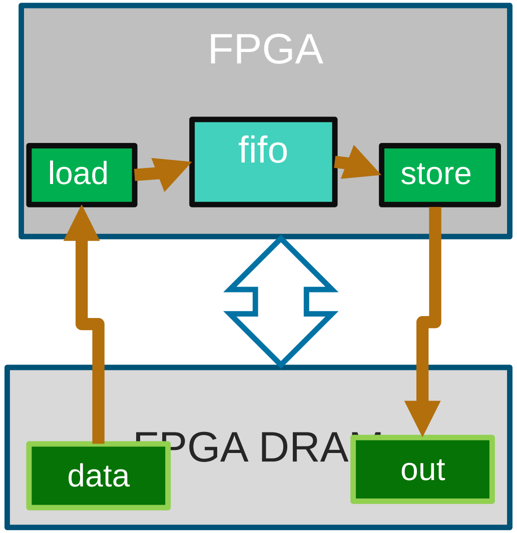
FIFO: Transfers to/from DRAM



```
val data = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val fifo = FIFO[Int](32)
  fifo load data
  out store fifo
}
```

Load from DRAM

Store to DRAM

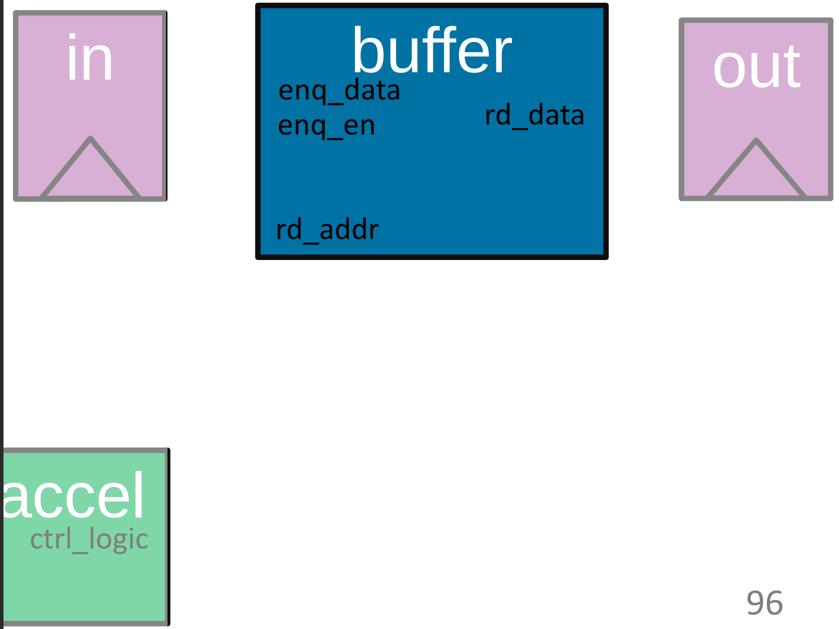


LineBuffer

```
val scalarIn  = ArgIn[Int]
val scalarOut = ArgOut[Int]
…
Accel {
  val buffer = LineBuffer[Int](3, 1024)


}
```

LineBuffer with 3 rows, each with 1024 columns

Note: Only 2-dimensional buffers currently supported. Dimensions must be statically known



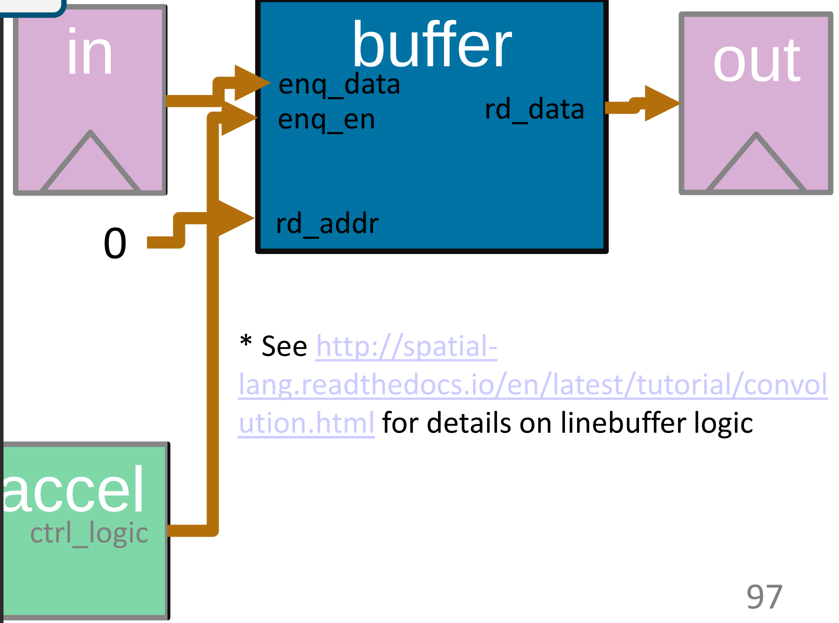
LineBuffer: Enqueueing / Reading

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val buffer = Li
  buffer.enq(in)
  out := buffer(0, 0)
}
```

Enqueue to **current row**

Addressed linebuffer read

Note: **LineBuffer** contains internal logic to increment **current row** based on the code structure*.



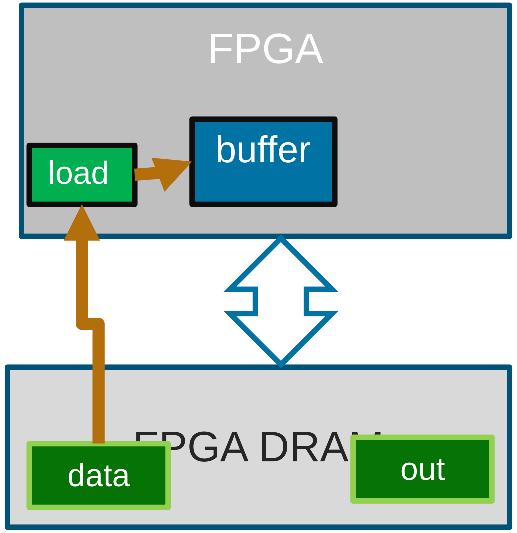
* See http://spatial-lang.readthedocs.io/en/latest/tutorial/convolution.html for details on linebuffer logic

LineBuffer: Loading from DRAM

```
val data = DRAM[Int](100,1024)
val out  = DRAM[Int](32)
…
Accel {
  val buffer = LineBuffer[Int](3,1024)
  buffer load data(0, 0::1024)

}
```

Load *data* row 0, columns 0 until 1024 to **current row** of buffer

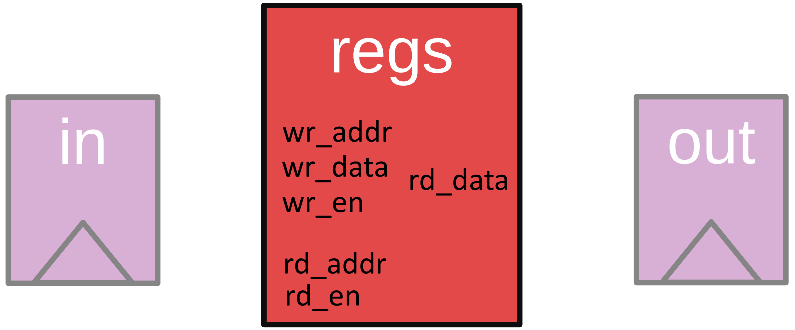
Note: Storing to **DRAM** from **LineBuffer** is currently unsupported



RegFile

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val regs = RegFile[Int](16, 16)


}
```

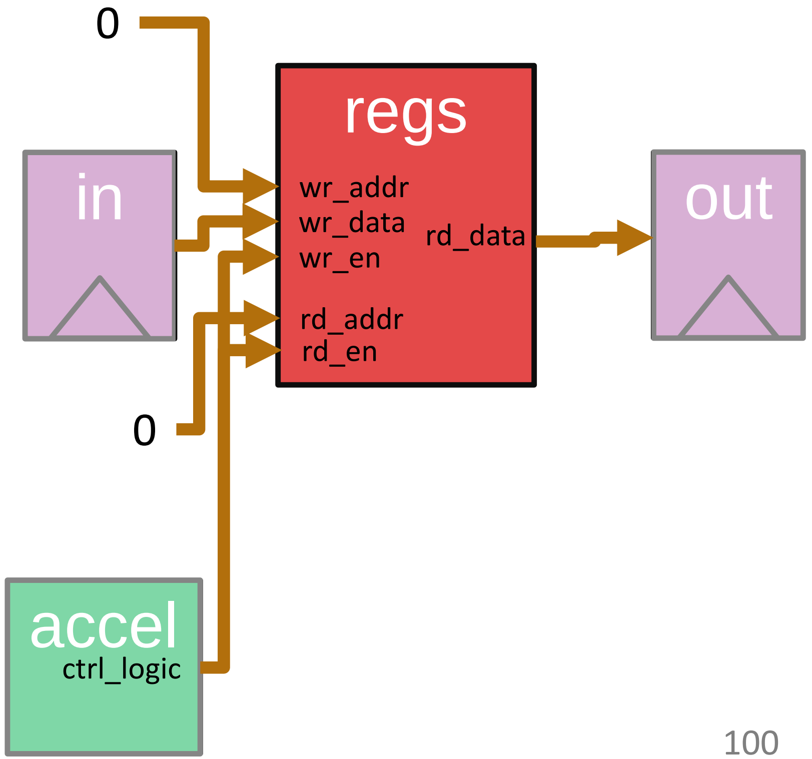
Creates register file (addressable array of registers) of size 16 x 16 with values of type **Int**



Note: Register files can be expensive! Use small ones, and only sparingly.

RegFile Reading/Writing

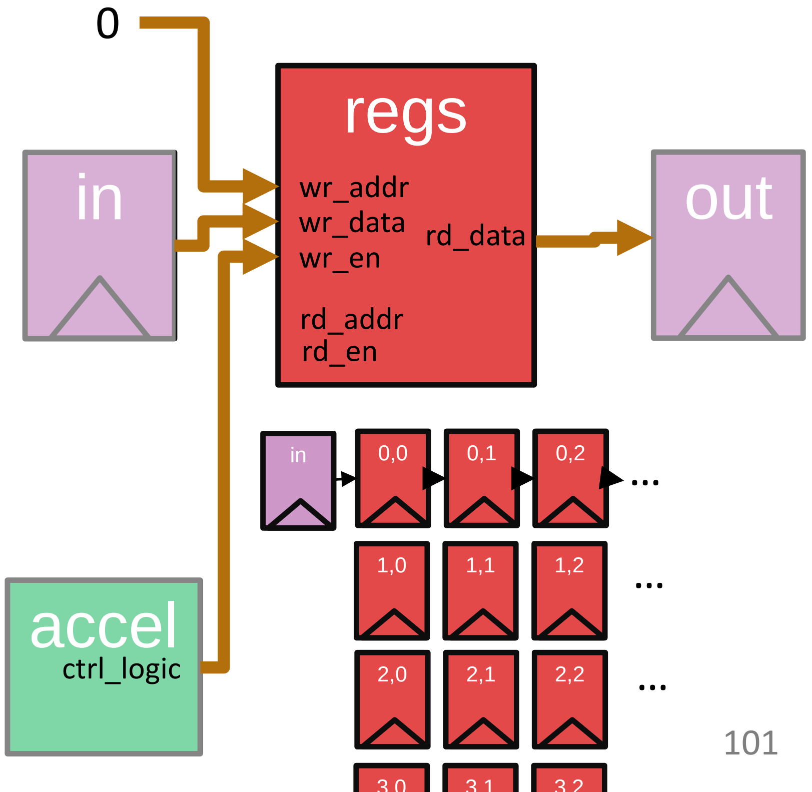
```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val regs = RegFile[Int](16, 16)
  regs(0, 0) = in.value
  out := regs(0, 0)
}
```



RegFile Shifting

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val regs = RegFile[Int](16, 16)
  regs(0, *) <<= in.value

}
```

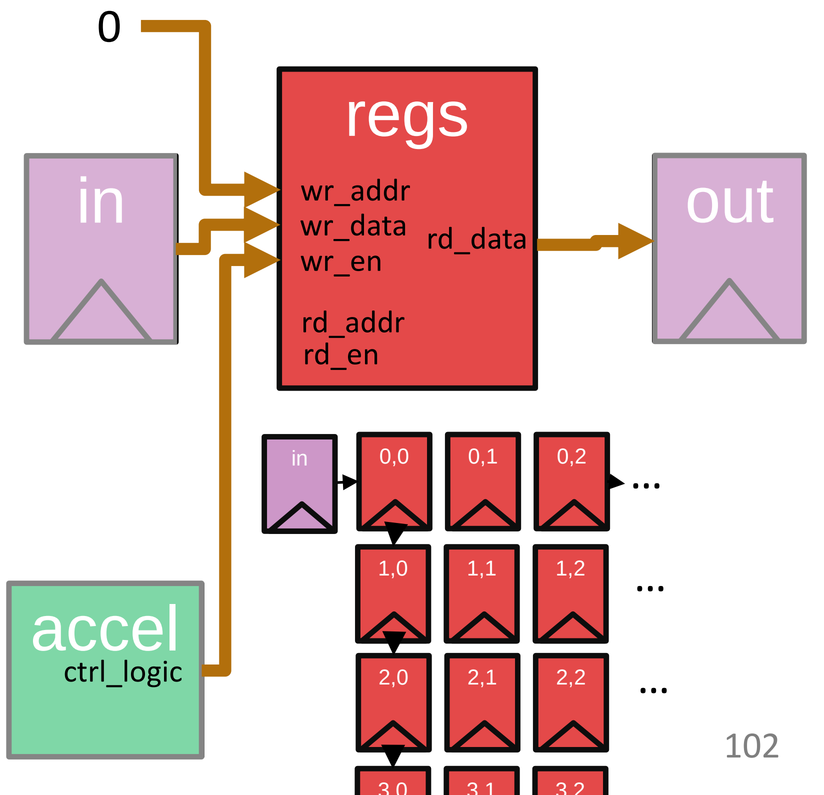
Shifts the given value into column 0 of the $i = 0^{th}$ row of *regs.* All other elements in this row move one column over



RegFile Shifting

```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val regs = RegFile[Int](16, 16)
  regs(*, 0) <<= in.value

}
```

Shifts the given value into row 0 of the *i* = *0th* column of *regs*. All other elements in this column move up by one position



A Note About Ports

- Spatial compiler makes best effort to minimize amount of resources needed to implement memory
- However, writing and reading from the same memory many times can be expensive!
- Be aware of how many times you read/write a given memory, and try to minimize the number of concurrent reads

CONTROLLERS

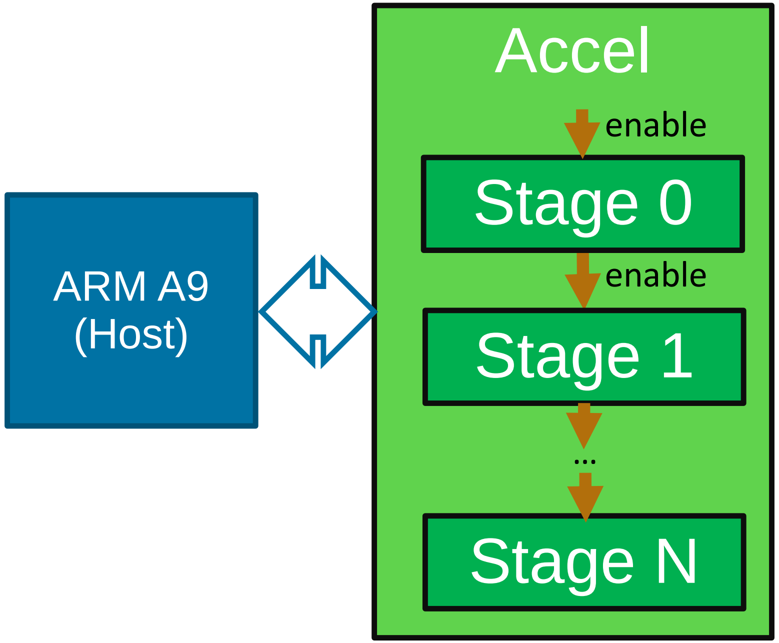
Accel

```
Accel {
  …
}
```

Receives **start** from ARM
Executes stages sequentially
Sends **done** to ARM

A **stage** is either:
- one primitive operation
- one controller

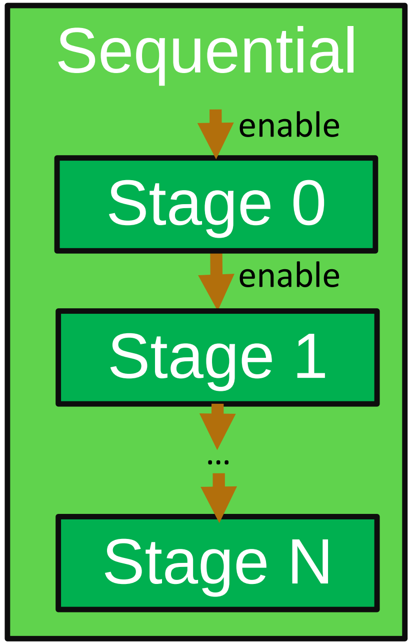
Note: May not be nested in any other controller



Sequential

```
Accel {
  …
  Sequential {
    …

  }
  …
}
```

Executes stages sequentially

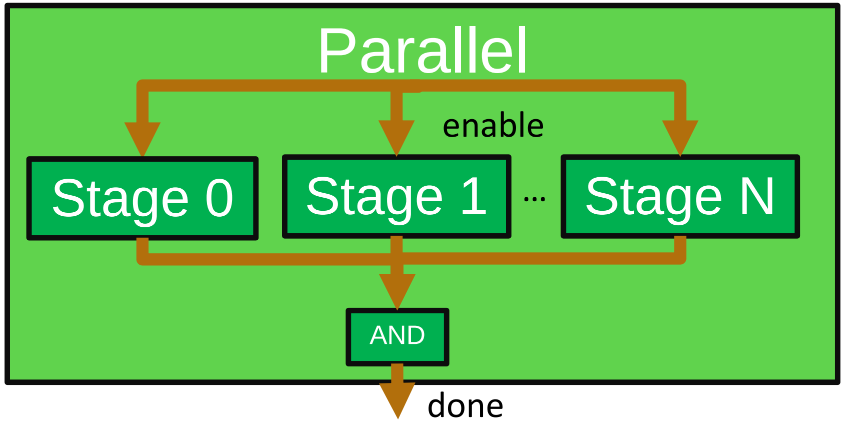


Parallel

```
Accel {
  …
  Parallel {
    …

  }
  …
}
```

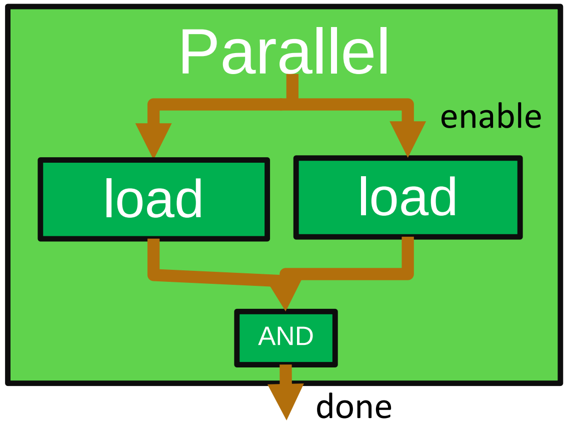
Executes stages in parallel
Completes when all stages finish

Note: Spatial will soon infer control structures for parallel execution automatically
But for now, use **Parallel** when you want to guarantee parallel execution

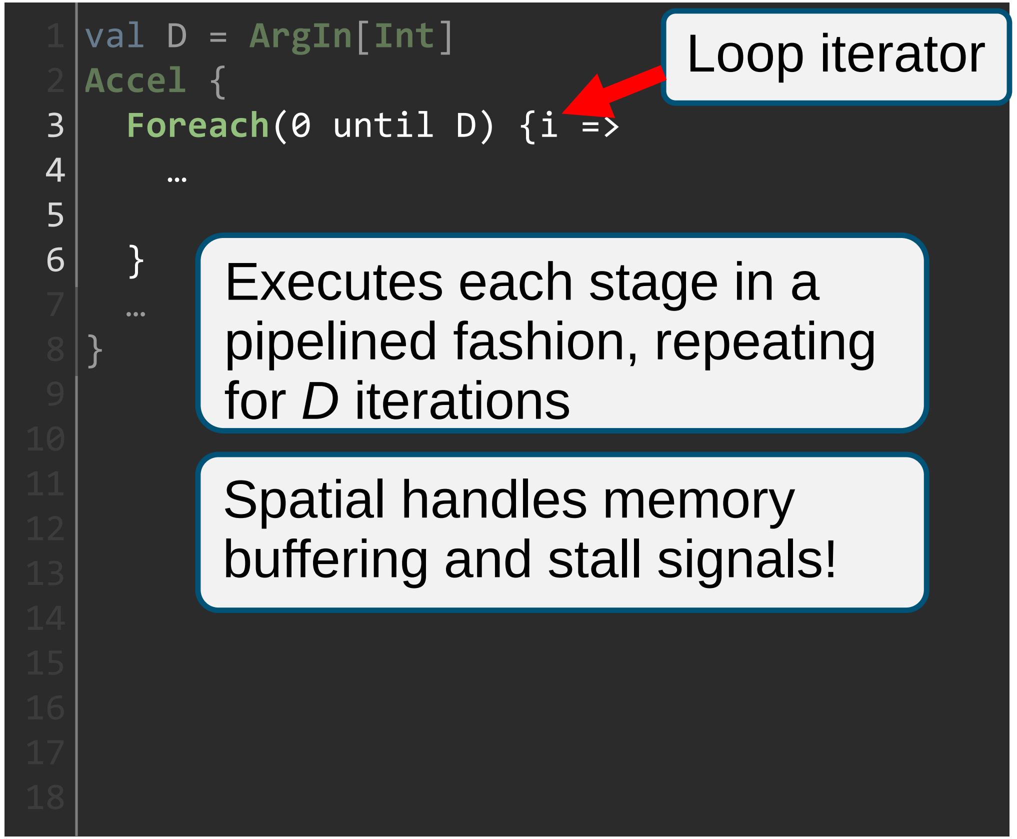


Parallel: Example

```
val dataA = DRAM[Int](1024)
val dataB = DRAM[Int](1024)

Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  Parallel {
    a load dataA(0::16)
    b load dataB(0::16)
  }
  …
}
```



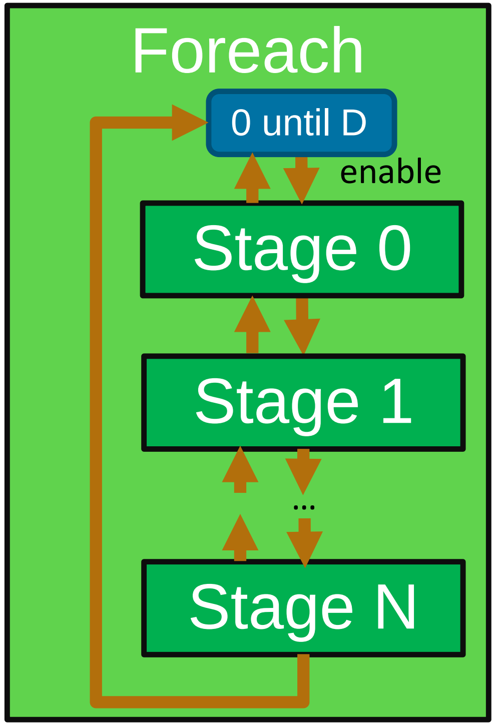
Foreach



```
val D = ArgIn[Int]
Accel {
  Foreach(0 until D) {i =>
     …

  }
  …
}
```

Executes each stage in a pipelined fashion, repeating for *D* iterations

Spatial handles memory buffering and stall signals!

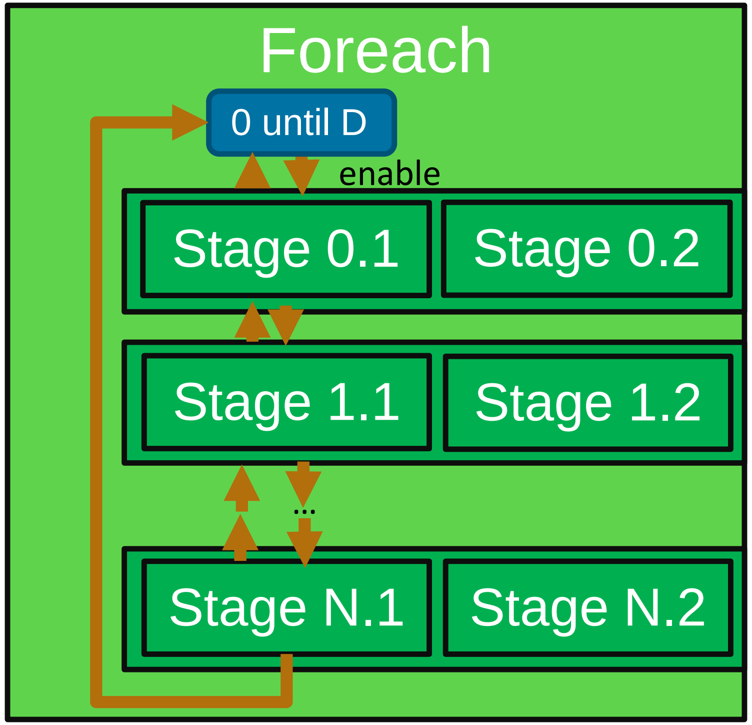


Foreach: Parallelization

```
val D = ArgIn[Int]
Accel {
  Foreach(0 until D par 2) {i =>
     …

  }
  …
}
```

Parallelizes the pipeline by duplicating the body

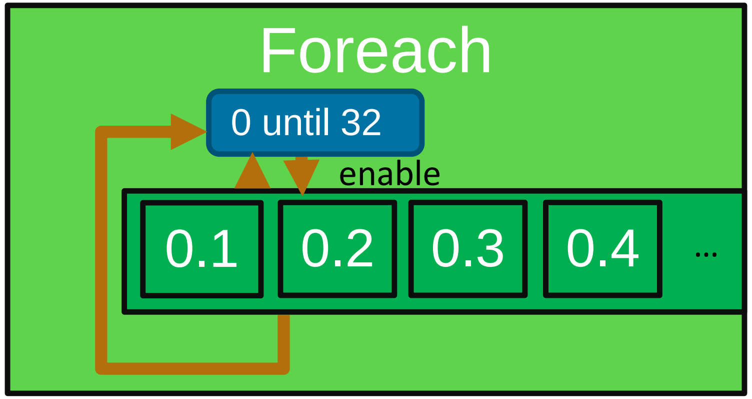
Spatial also handles memory banking for you!



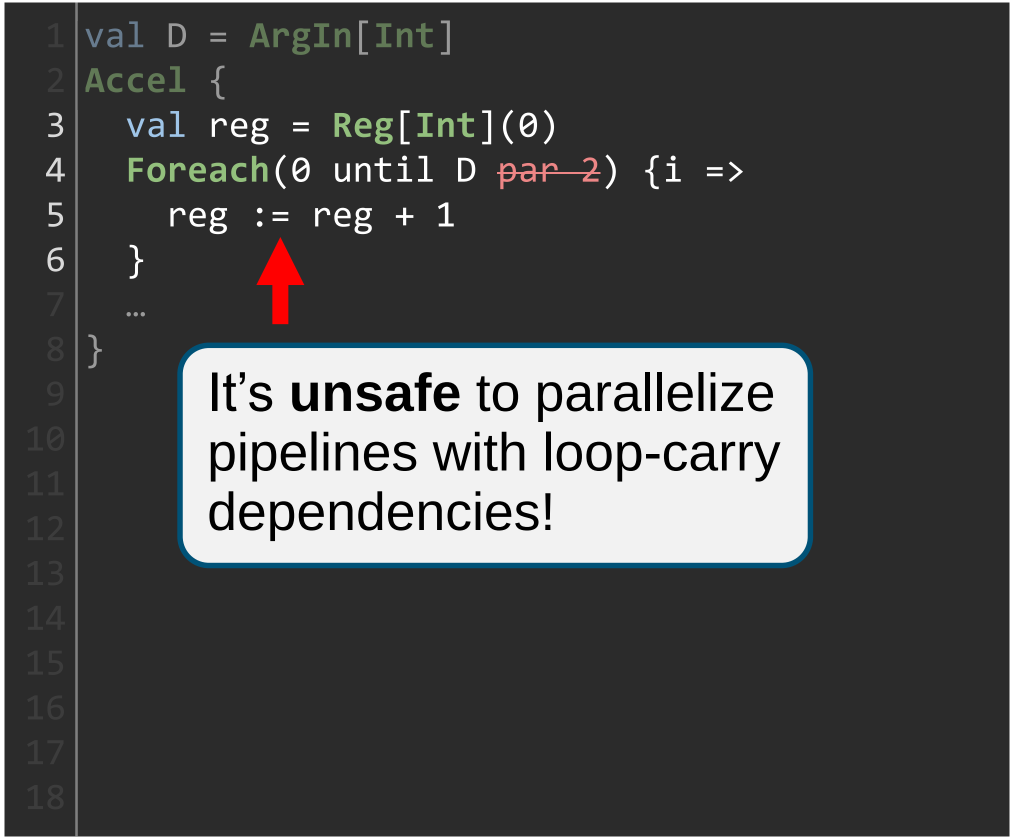
Foreach: Example

```
val data = DRAM[Int](32)
Accel {
  val input  = SRAM[Int](32)
  val output = SRAM[Int](32)
  input load data
  Foreach(0 until 32 par 16) {i =>
    output(i) = input(i) * 2
  }
  data store output
}
```

Multiply every element by 2, store back to DRAM

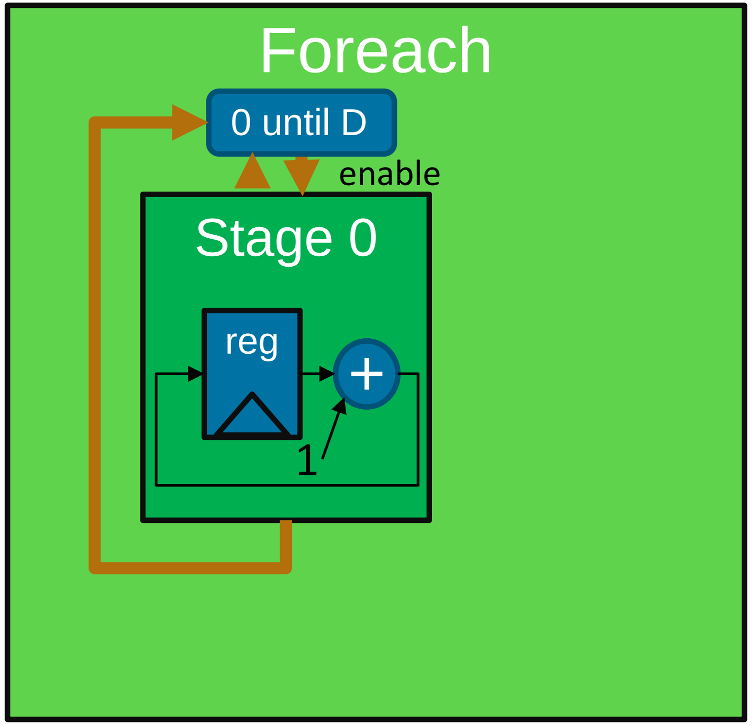


Foreach: Illegal Parallelization Cases



```
val D = ArgIn[Int]
Accel {
  val reg = Reg[Int](0)
  Foreach(0 until D ~~par 2~~) {i =>
    reg := reg + 1
  }
  …
}
```

It's **unsafe** to parallelize pipelines with loop-carry dependencies!



Reduce

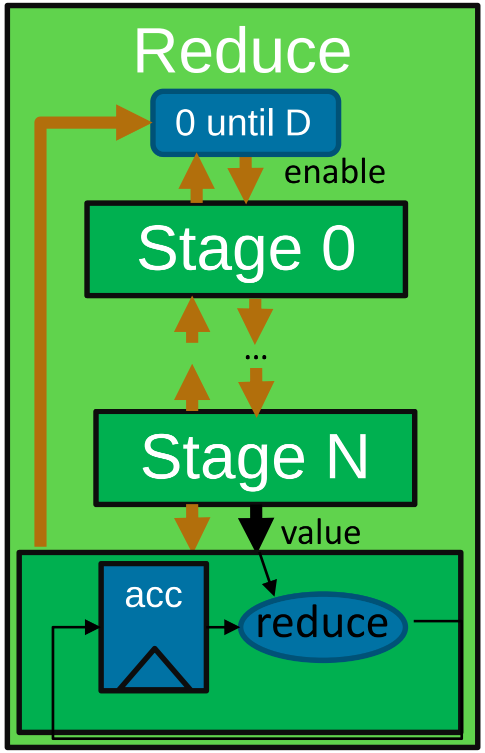
Zero value OR accumulator

Loop iterator

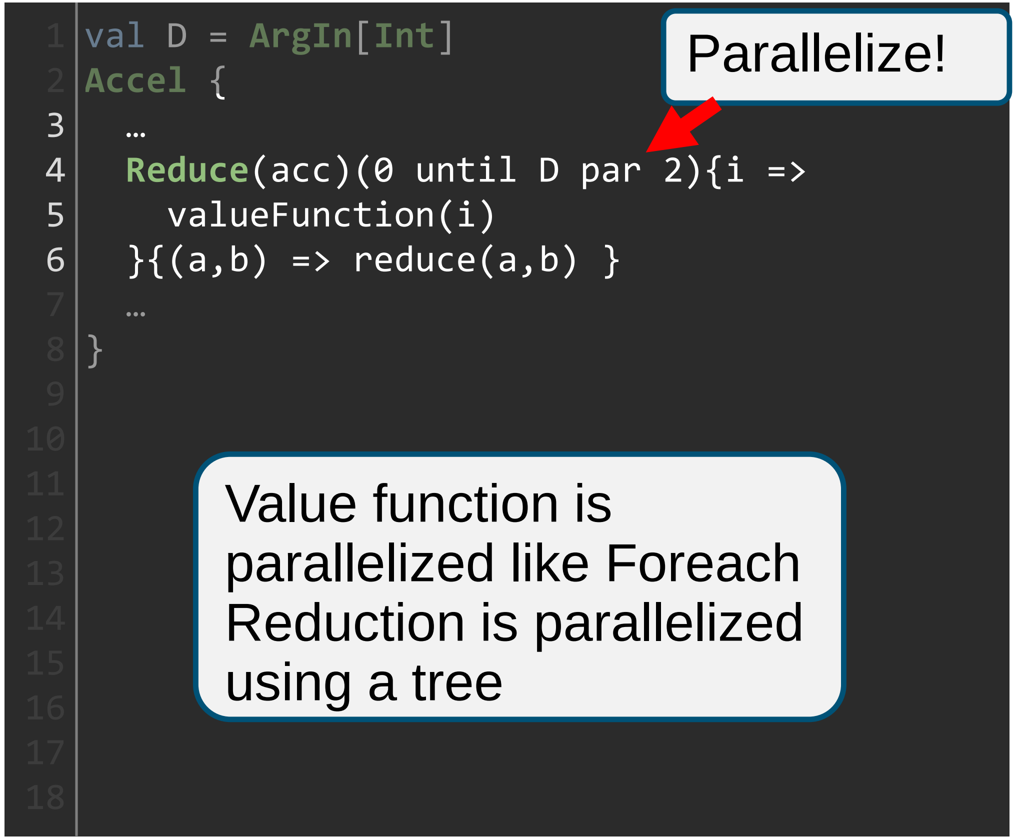
```
  …
  Reduce(acc)(0 until D){i =>

  }{(a,b) => reduce(a,b) }
  …
}
```

Value function (aka map)

Executes each stage in a pipelined fashion, repeating for *N* iterations.
Reduces the result of the value function into an accumulator



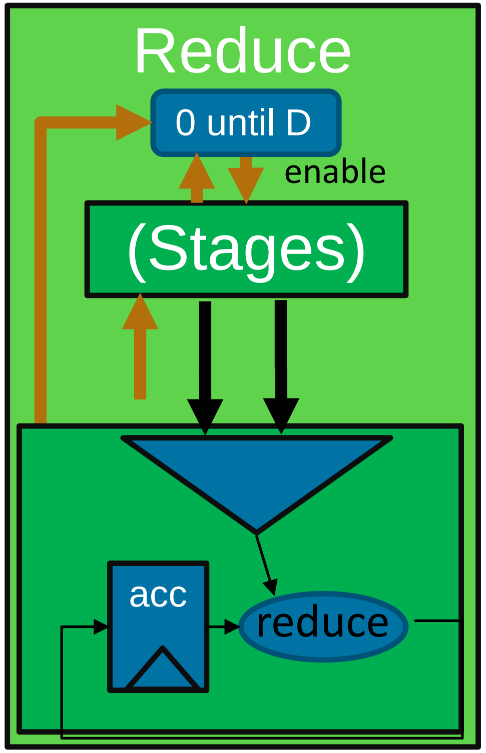
Reduce: Parallelization



```
val D = ArgIn[Int]
Accel {
  …
  Reduce(acc)(0 until D par 2){i =>
    valueFunction(i)
  }{(a,b) => reduce(a,b) }
  …
}
```

Parallelize!

Value function is parallelized like Foreach
Reduction is parallelized using a tree

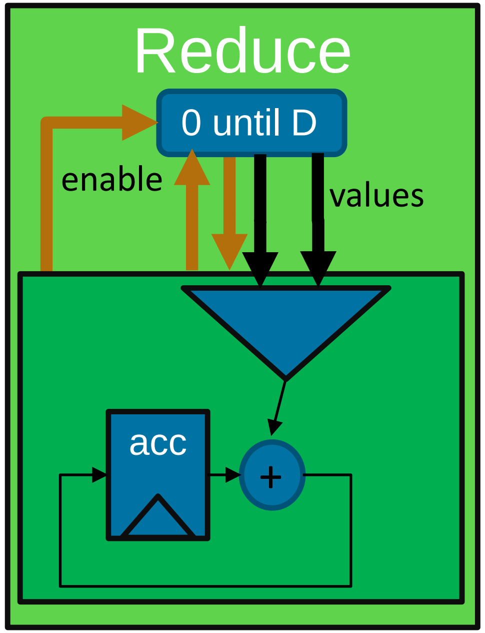


Reduce: Example

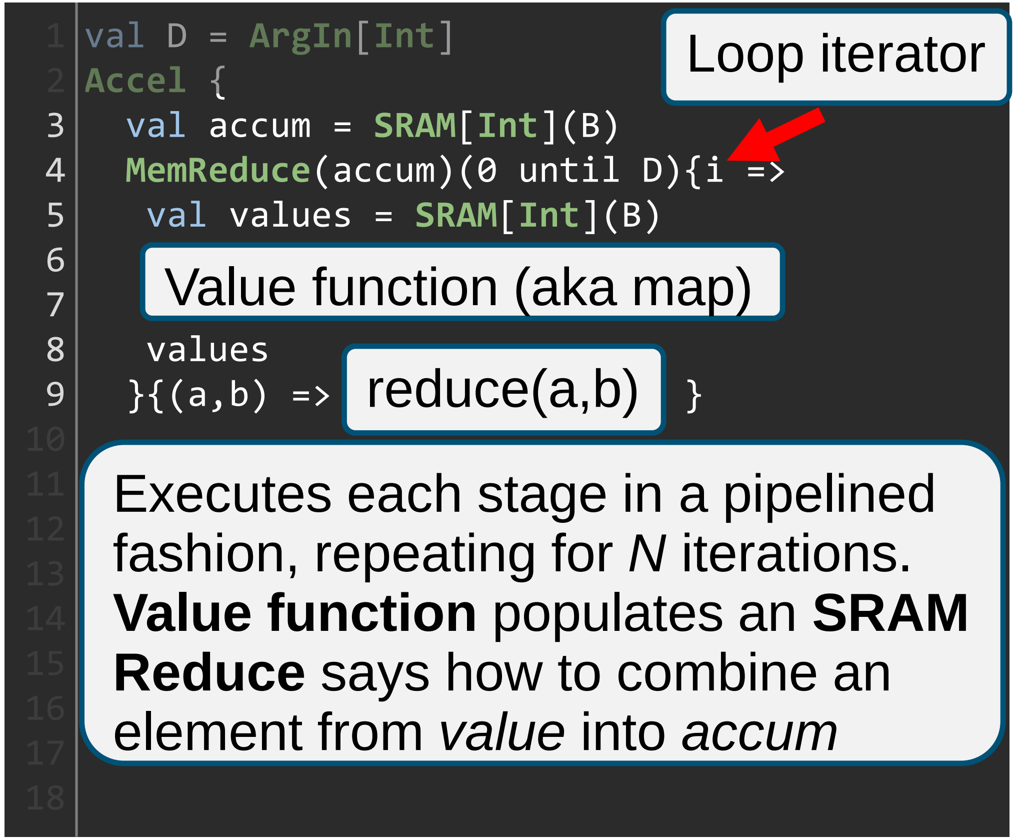
```
val D = ArgIn[Int]
val out = ArgOut[Int]
Accel {
  val acc = Reg[Int](0)
  Reduce(acc)(0 until D par 16){i =>
    i
  }{(a,b) => a + b }
  out := acc
}
```

Sum the values 0 until D, adding 16 values in parallel

accum contains the sum after the controller ends



MemReduce



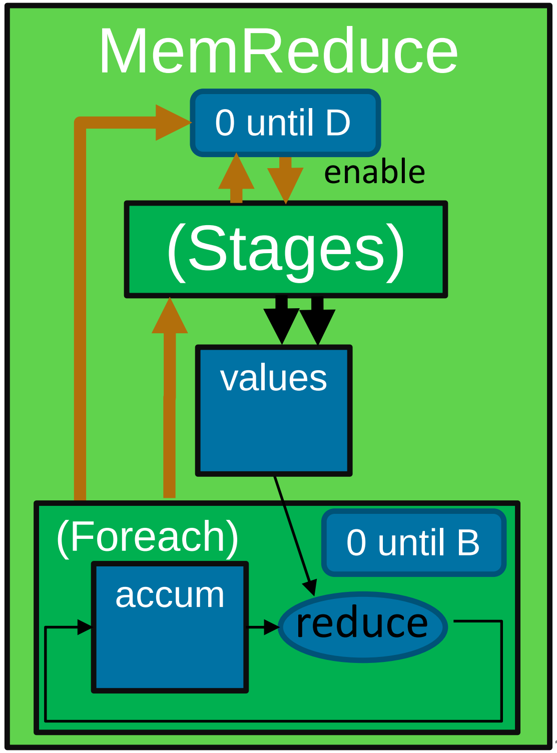
```
val D = ArgIn[Int]
Accel {
  val accum = SRAM[Int](B)
  MemReduce(accum)(0 until D){i =>
   val values = SRAM[Int](B)


   values
  }{(a,b) =>  }
```

Loop iterator

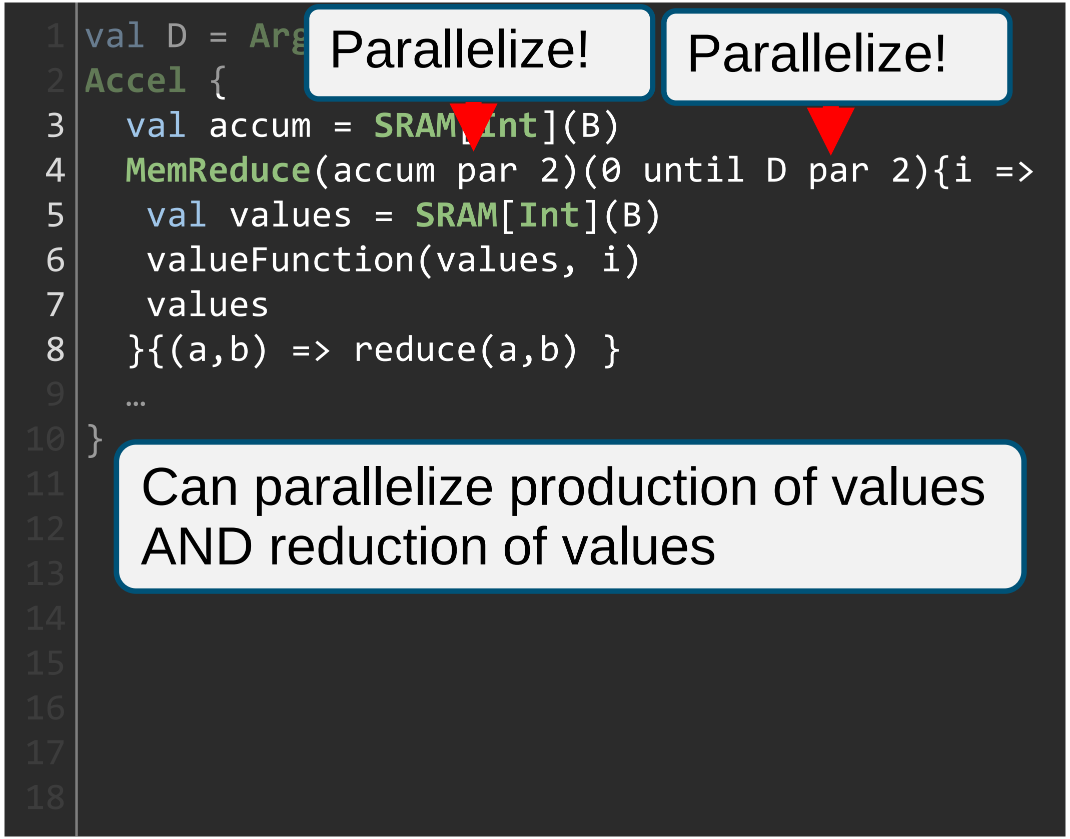
Value function (aka map)

reduce(a,b)

Executes each stage in a pipelined fashion, repeating for *N* iterations. **Value function** populates an **SRAM** **Reduce** says how to combine an element from *value* into *accum*



MemReduce: Parallelization

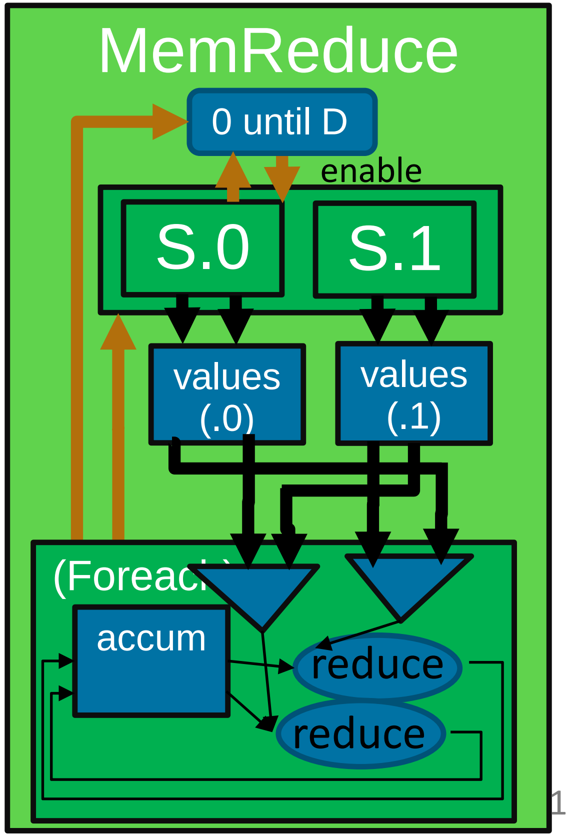


```
val D = Arg
Accel {
  val accum = SRAM[Int](B)
  MemReduce(accum par 2)(0 until D par 2){i =>
   val values = SRAM[Int](B)
   valueFunction(values, i)
   values
  }{(a,b) => reduce(a,b) }
  …
}
```

Parallelize!

Parallelize!

Can parallelize production of values
AND reduction of values

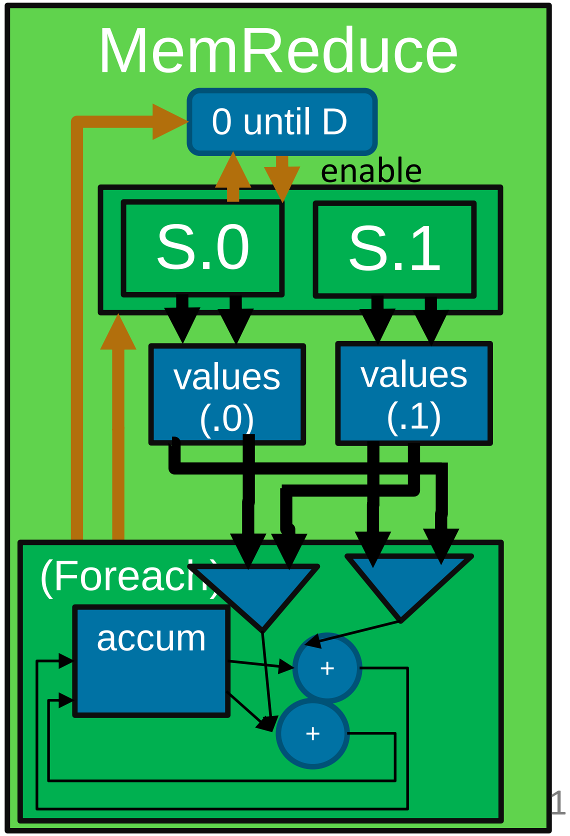


MemReduce: Example

```
val data = DRAM[Int](D)
val out  = DRAM[Int](16)
Accel {
  val accum = SRAM[Int](16)
  MemReduce(accum par 2)(D by 16 par 2){i =>
   val values = SRAM[Int](16)
   values load data(i::i+16)
   values
  }{(a,b) => a + b }

  out store accum
}
```

0 until D, strided by 16

Chunks up *data* into 16 element blocks and combines the blocks using element-wise addition

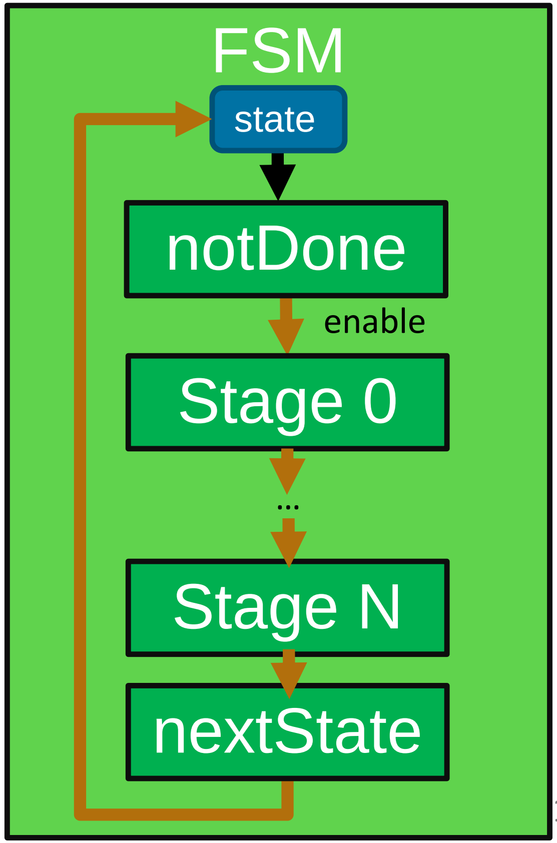


FSM (State Machine)

```
Accel {
  …
  FSM[T]{state => notDone(state) }{state =>

    action(state)

  }{state => nextState(state) }
  …
}
```

Type of state
(any bit-based type will work)

Checks the **notDone** condition
If notDone:
  Executes **action** states sequentially
  Executes the **nextState** logic
  Repeat

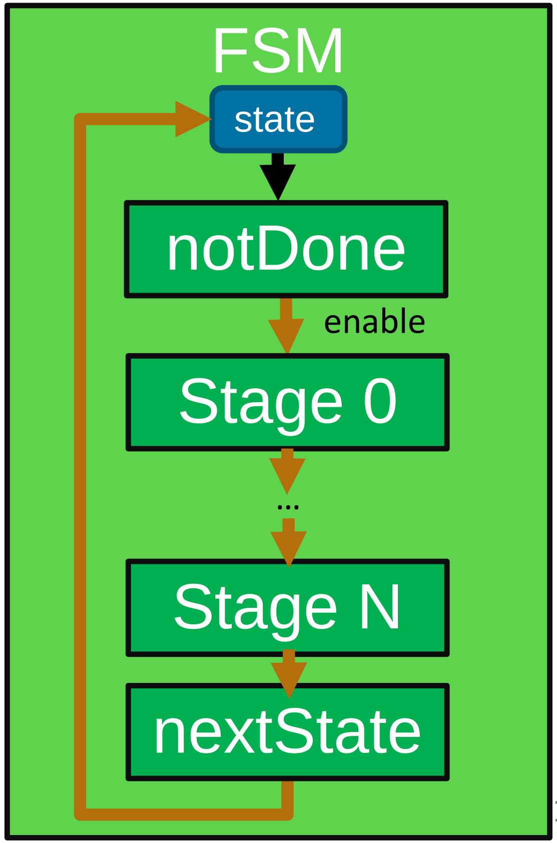


FSM (State Machine)

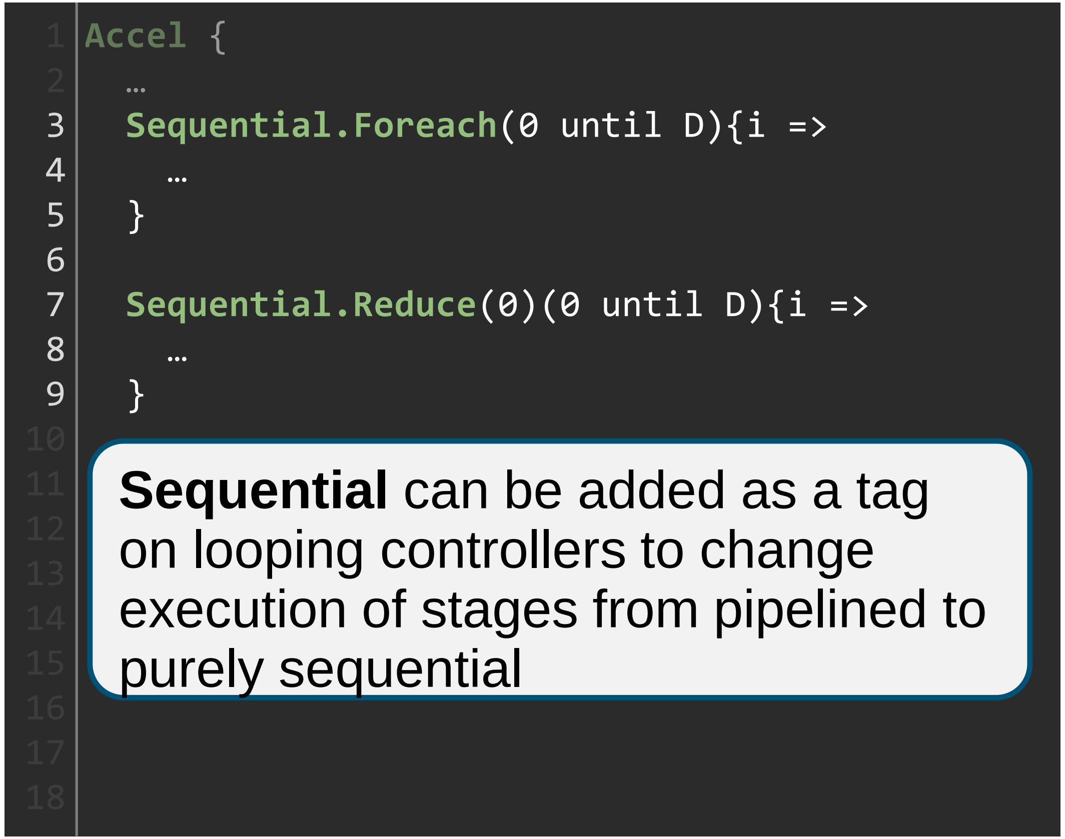
```
Accel {
  …
  FSM(init){state => notDone(state) }{state =>

    action(state)

  }{state => nextState(state) }
  …
}
```

Can also give an explicit initial state

(Otherwise initial state is zero)



Controller Tags



```
Accel {
  …
  Sequential.Foreach(0 until D){i =>
    …
  }

  Sequential.Reduce(0)(0 until D){i =>
    …
  }
```

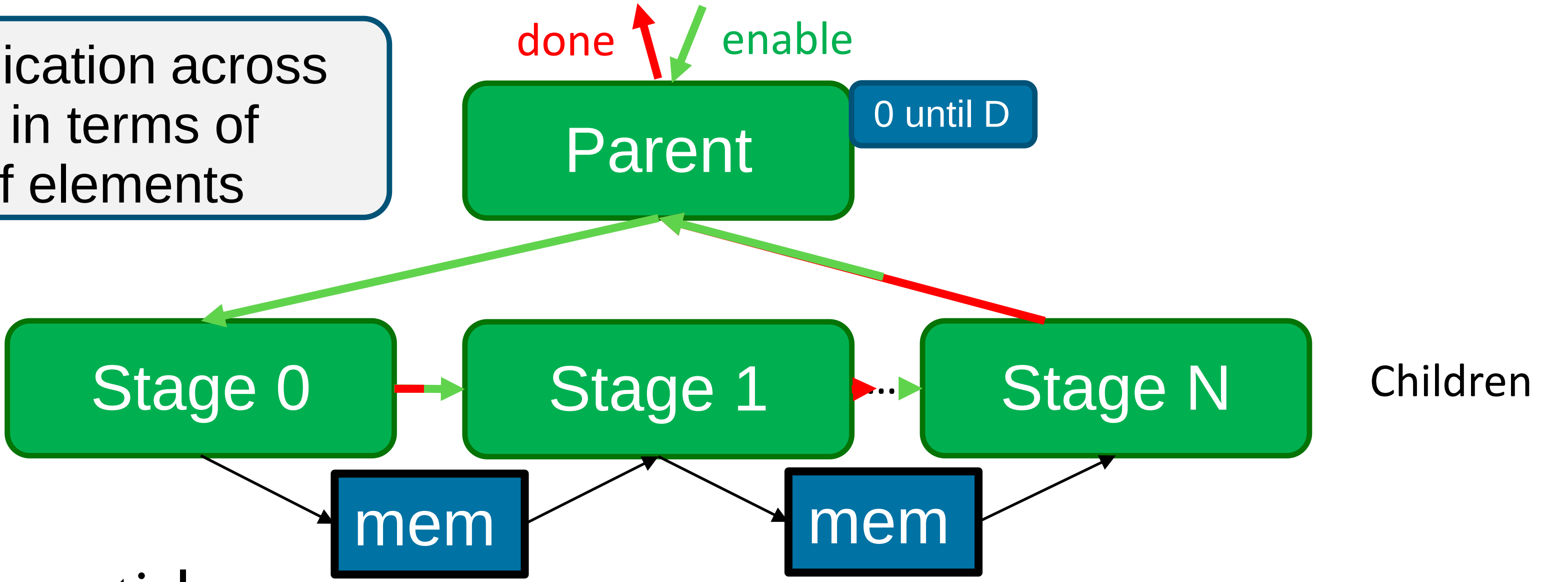
Sequential can be added as a tag on looping controllers to change execution of stages from pipelined to purely sequential

Spatial Tutorial

Part 3: Streaming and Debugging

Control Schemes: Sequential Execution

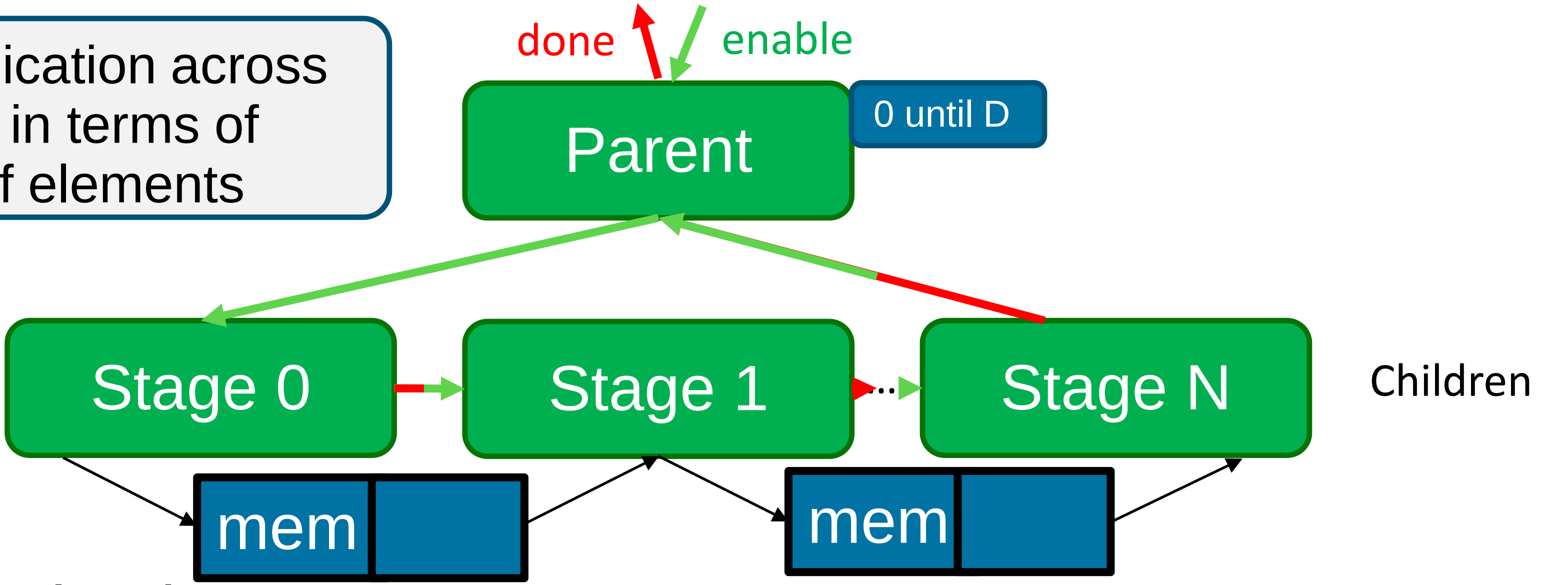
Communication across stages is in terms of **blocks** of elements



- Sequential
  - Parent enables Stage 0 when enabled, as long as counter < D
  - Stage K is enabled when Stage K–1 completes (K > 0)

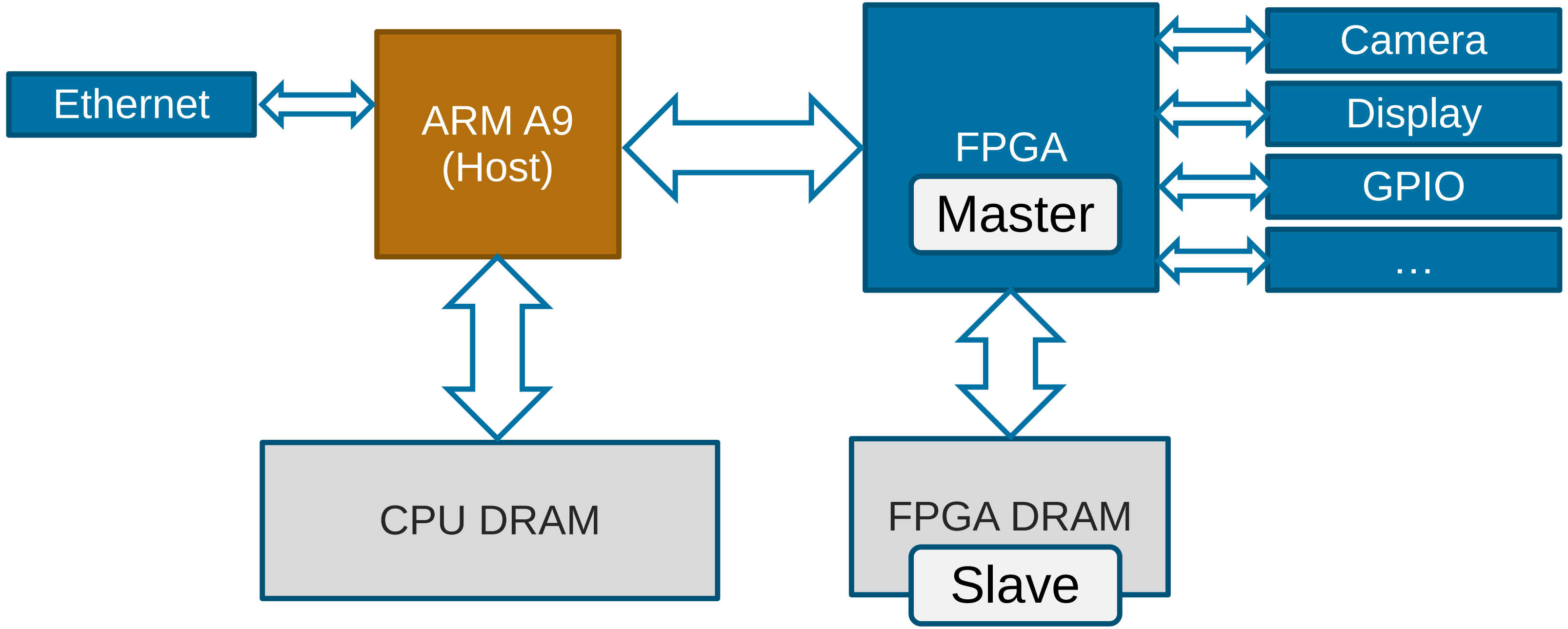
Control Schemes: Pipelined Execution

Communication across stages is in terms of **blocks** of elements



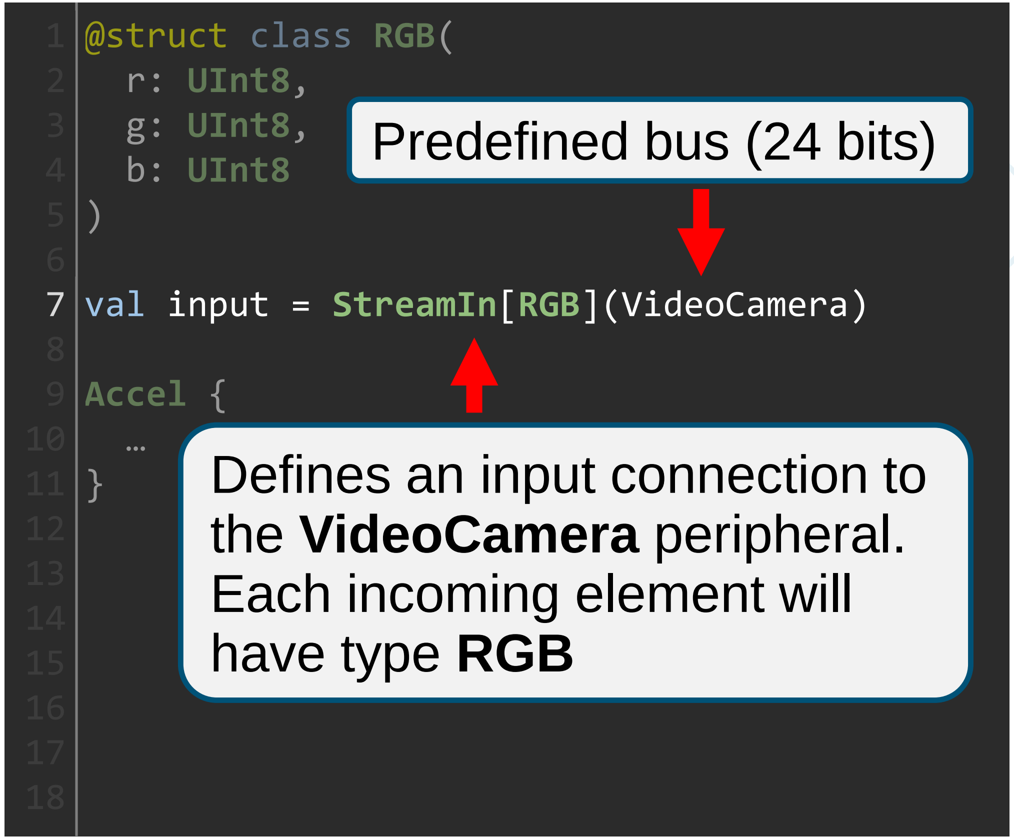
- Pipelined
  - Parent enables Stage 0 TWICE when enabled, as long as counter < D
  - Stage K is enabled when Stage K–1 completes (K > 0)

Previously on *Spatial*…



STREAMING

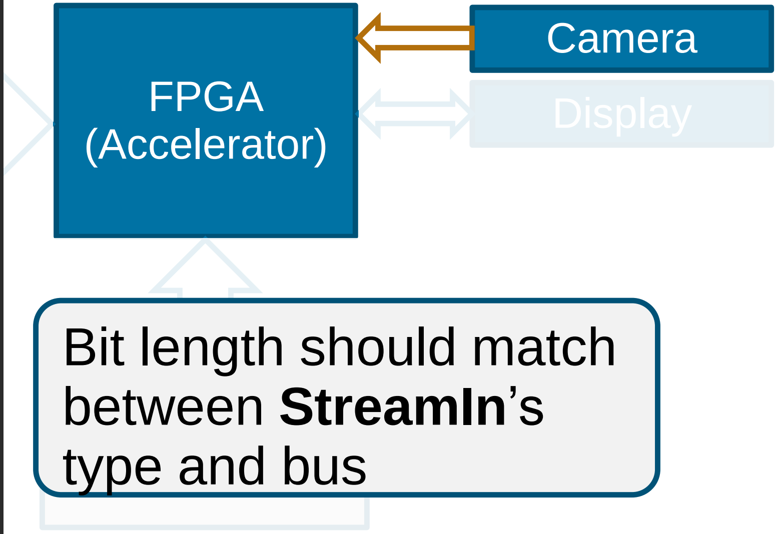
StreamIn



```
@struct class RGB(
  r: UInt8,
  g: UInt8,
  b: UInt8
)

val input = StreamIn[RGB](VideoCamera)

Accel {
  …
}
```

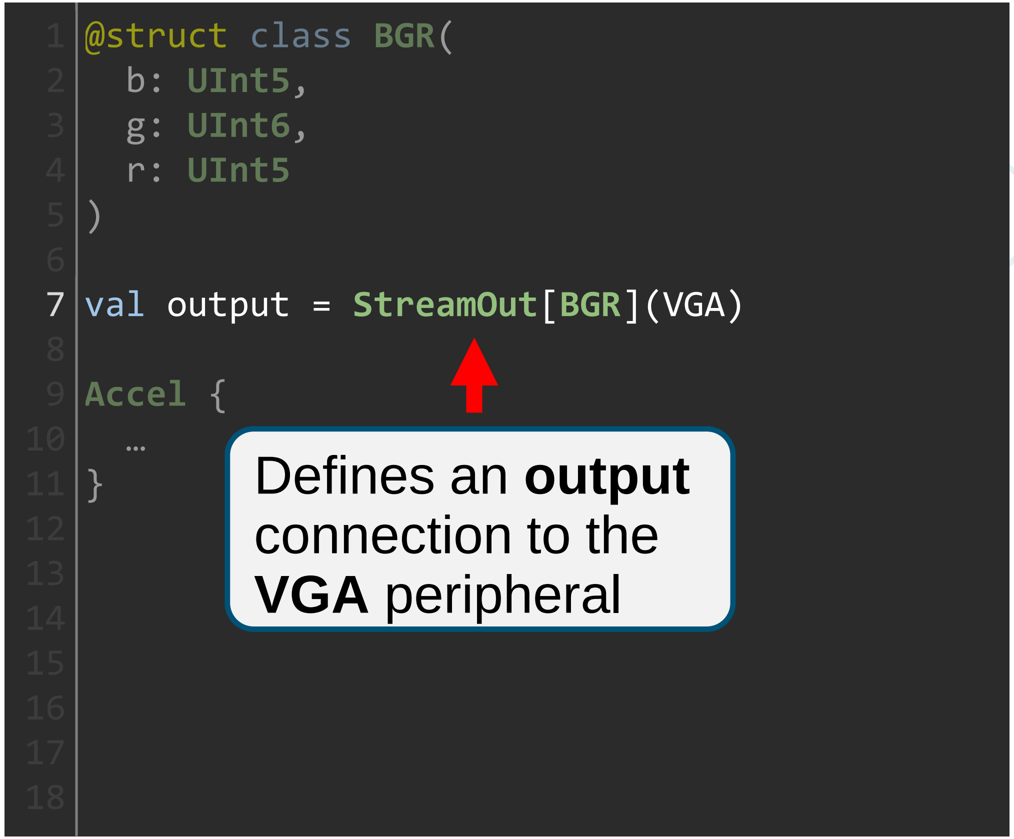
Predefined bus (24 bits)

Defines an input connection to the **VideoCamera** peripheral. Each incoming element will have type **RGB**



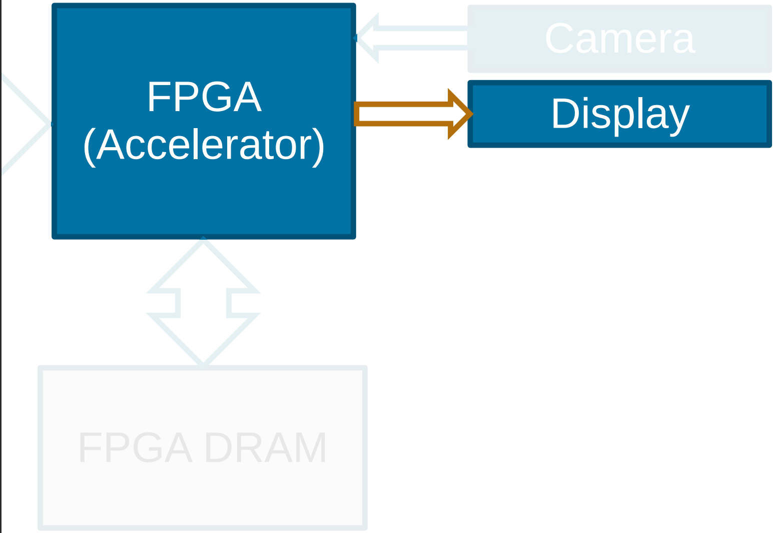
Bit length should match between **StreamIn**’s type and bus

StreamOut



```
@struct class BGR(
  b: UInt5,
  g: UInt6,
  r: UInt5
)

val output = StreamOut[BGR](VGA)

Accel {
  …
}
```

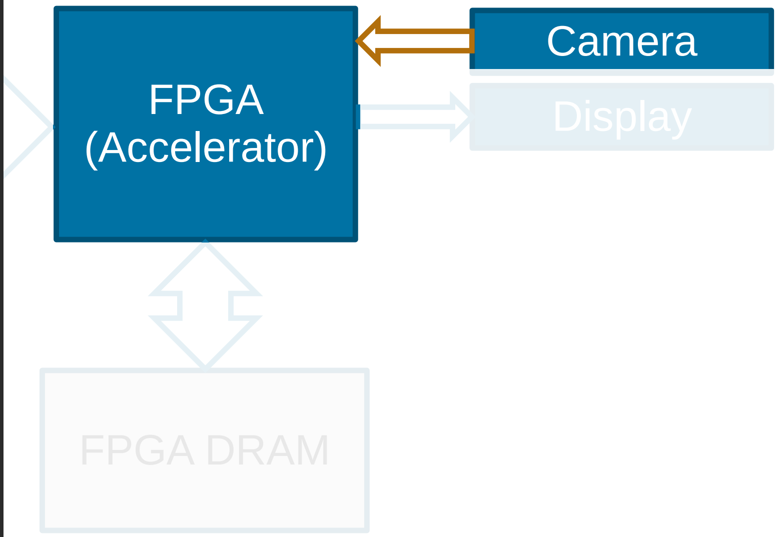
Defines an **output** connection to the **VGA** peripheral



Connecting to StreamIn

```
val input = StreamIn[RGB](VideoCamera)

val output = StreamOut[BGR](VGA)

Accel {
  val element = input.value

}
```

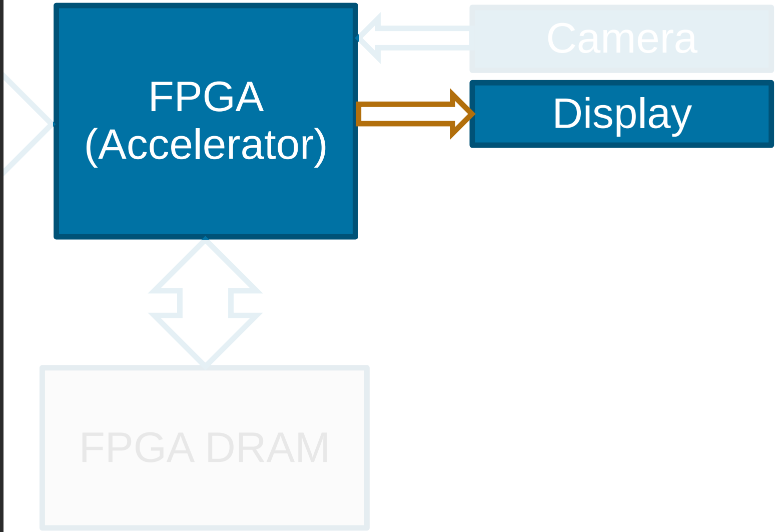
Creates a reference to the wires corresponding to this stream



Connecting to StreamOut

```
val input = StreamIn[RGB](VideoCamera)

val output = StreamOut[BGR](VGA)

Accel {
  val element = input.value
  output := BGR(32, 64, 32)
}
```

Sets the given wires as a driver for the **StreamOut**

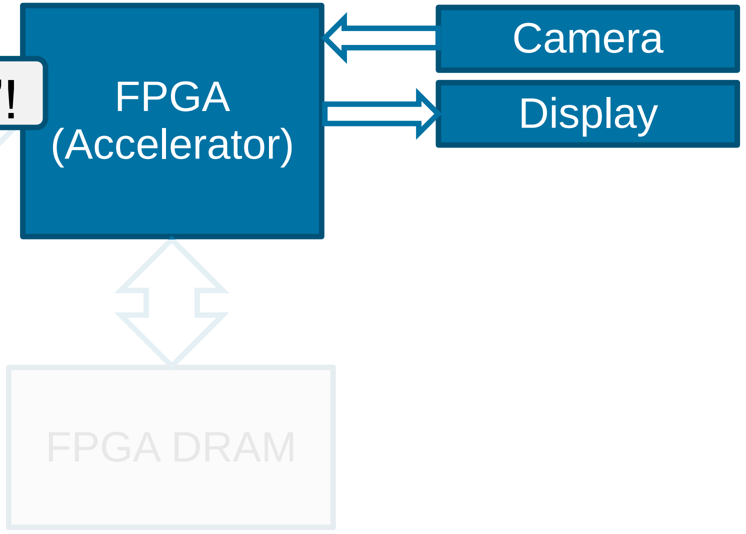


Problem?

```
val input = StreamIn[RGB](VideoCamera)

val output = StreamOut[BGR](VGA)

Accel {
  val element = input.value
  output := BGR(32, 64, 32)
}
```

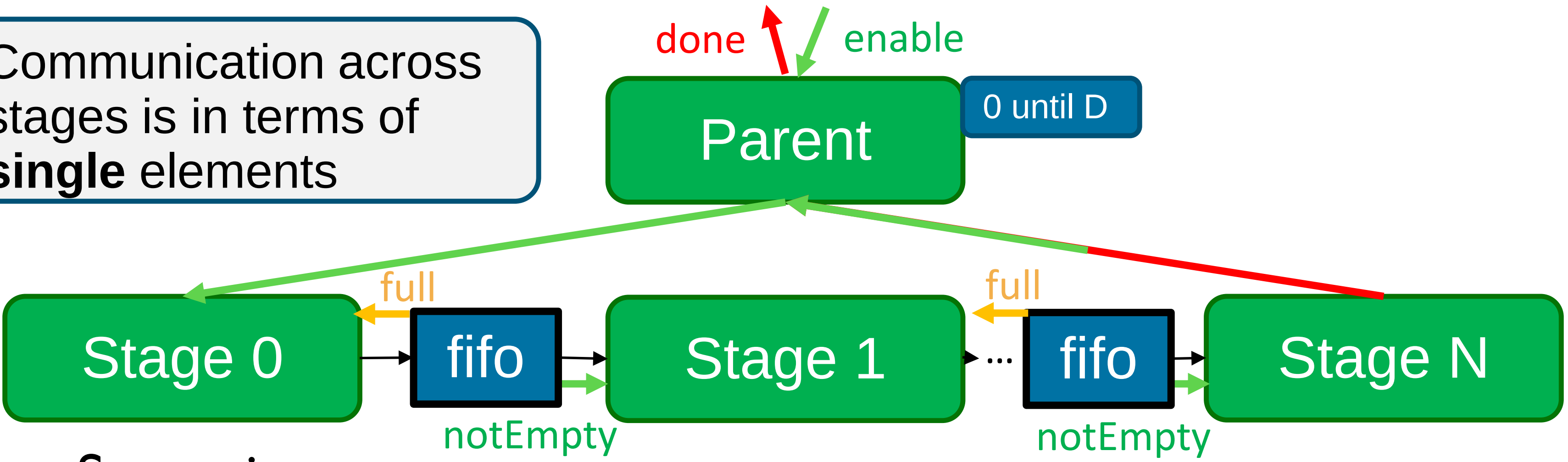
Only runs for one “iteration”!

Why won’t this do what we want?



Control Schemes: Streaming Execution

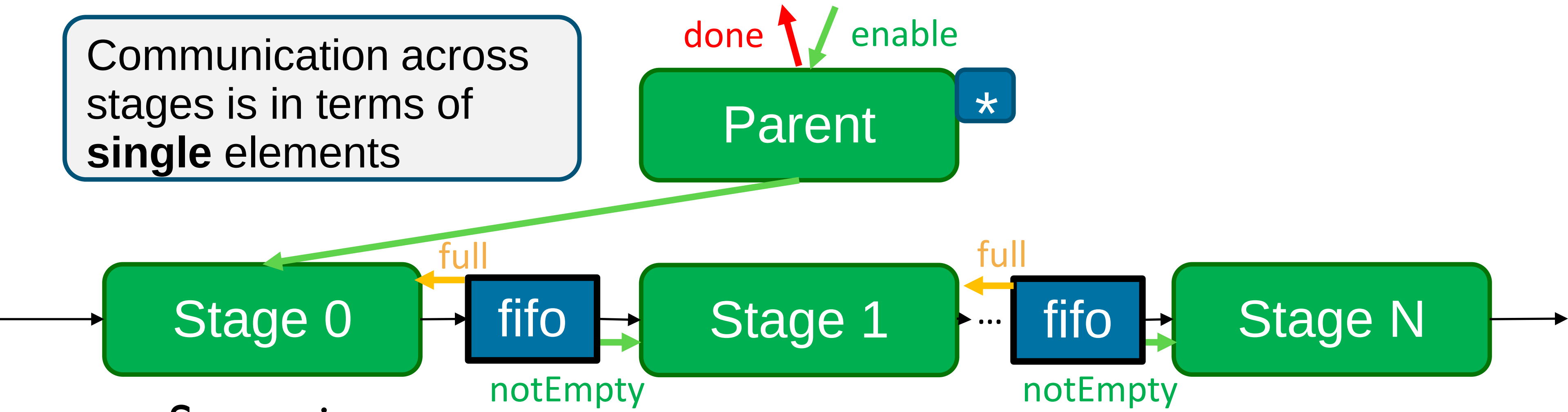
Communication across stages is in terms of **single** elements



- Streaming
  - Control is **data driven**
  - Stage 0 is enabled by the parent as long as counter < D
  - Stage K is enabled when data is available (K > 0)

Control Schemes: Streaming Forever

Communication across stages is in terms of **single** elements

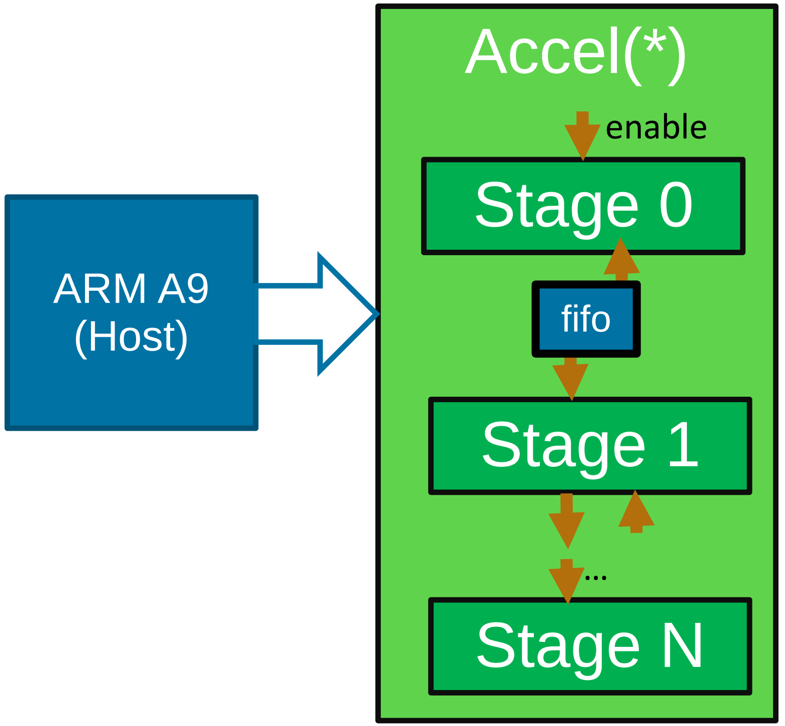


- Streaming
  - Control is **data driven**
  - Stage 0 is **initially** enabled by the parent
  - Stage K is enabled when data is available

Accel(*)

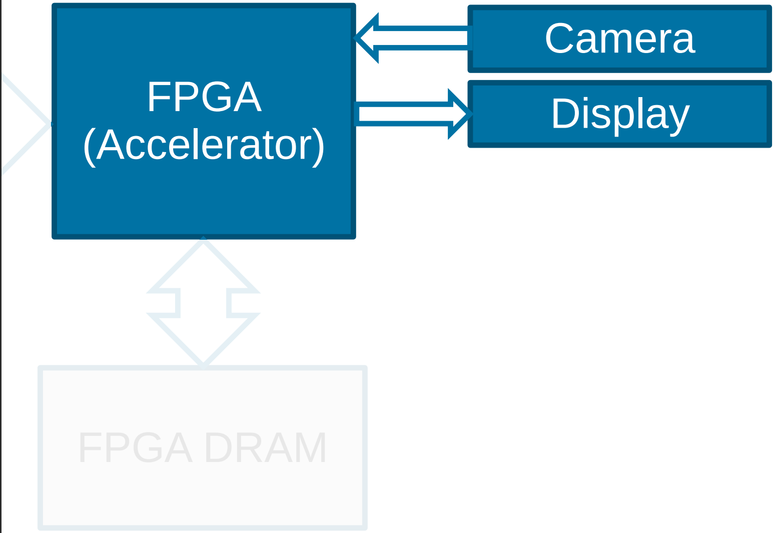
```
Accel(*) {
  …

}
```

Receives **start** from ARM
Executes stages in streaming mode
~~Sends **done** to ARM~~
RUNS FOREVER



Solution!

```
val input = StreamIn[RGB](VideoCamera)

val output = StreamOut[BGR](VGA)

Accel(*) {
  val element = input.value
  output := BGR(32, 64, 32)
}
```

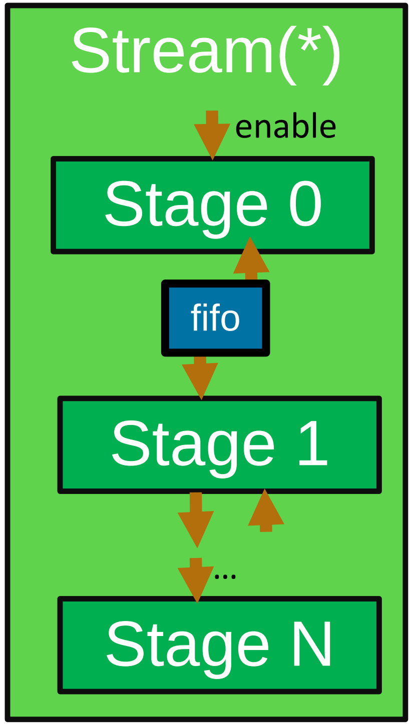


Stream(*)

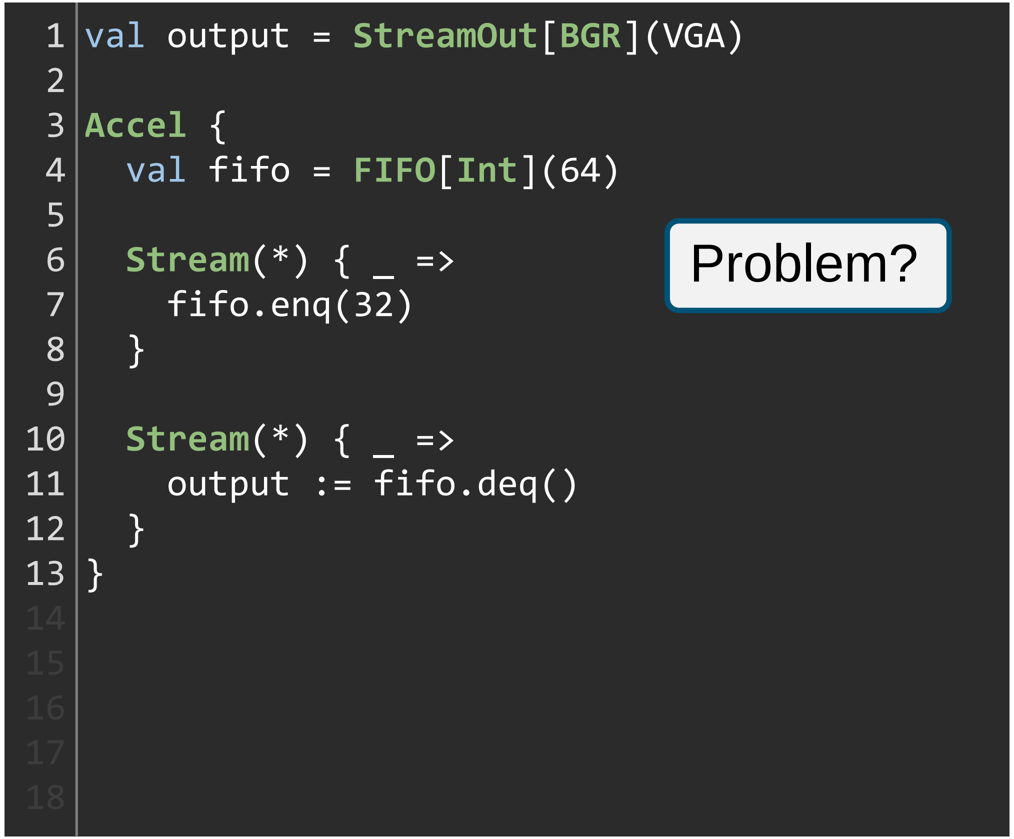
```
Accel {
  …
  Stream(*) { i =>
    …

  }
  …
}
```

Executes stages in streaming fashion, runs forever

i is technically a counter, but no guarantees about its value



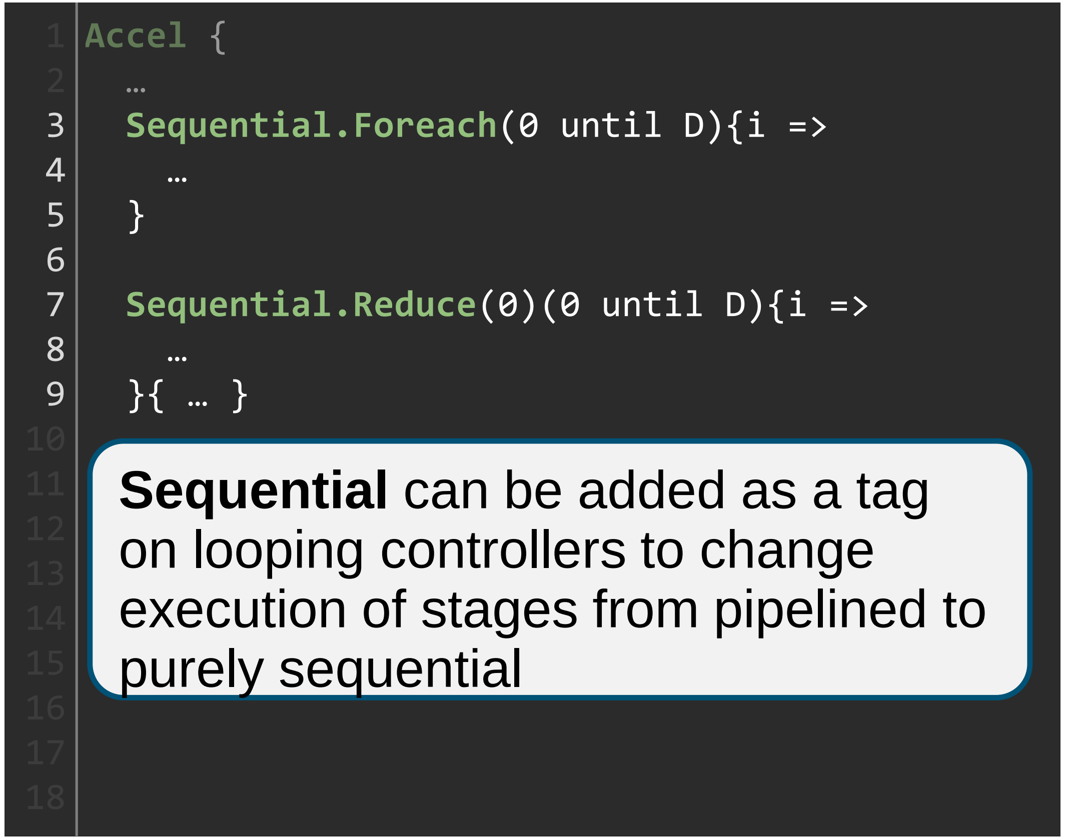
Notes on Streaming / Forever Loops



```
val output = StreamOut[BGR](VGA)

Accel {
  val fifo = FIFO[Int](64)

  Stream(*) { _ =>
    fifo.enq(32)
  }

  Stream(*) { _ =>
    output := fifo.deq()
  }
}
```

Problem?

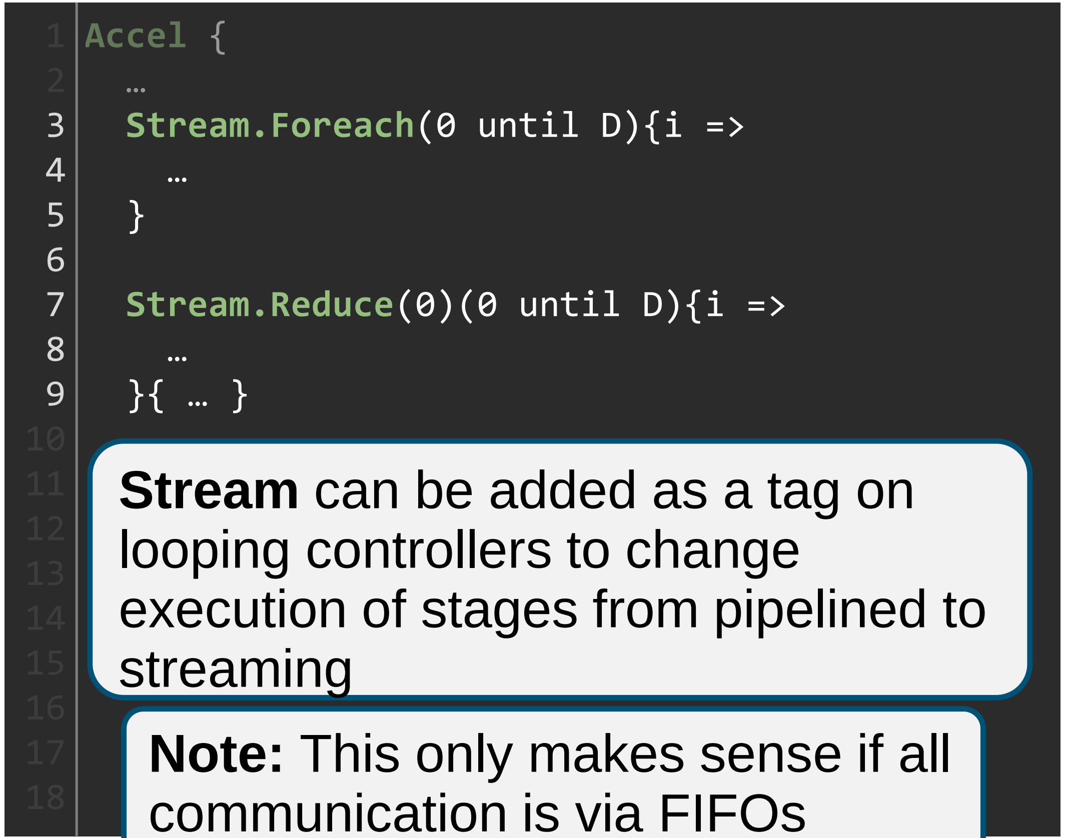
Controller Tags: Sequential



```
Accel {
  …
  Sequential.Foreach(0 until D){i =>
    …
  }

  Sequential.Reduce(0)(0 until D){i =>
    …
  }{ … }
```

Sequential can be added as a tag on looping controllers to change execution of stages from pipelined to purely sequential

Controller Tags: Stream

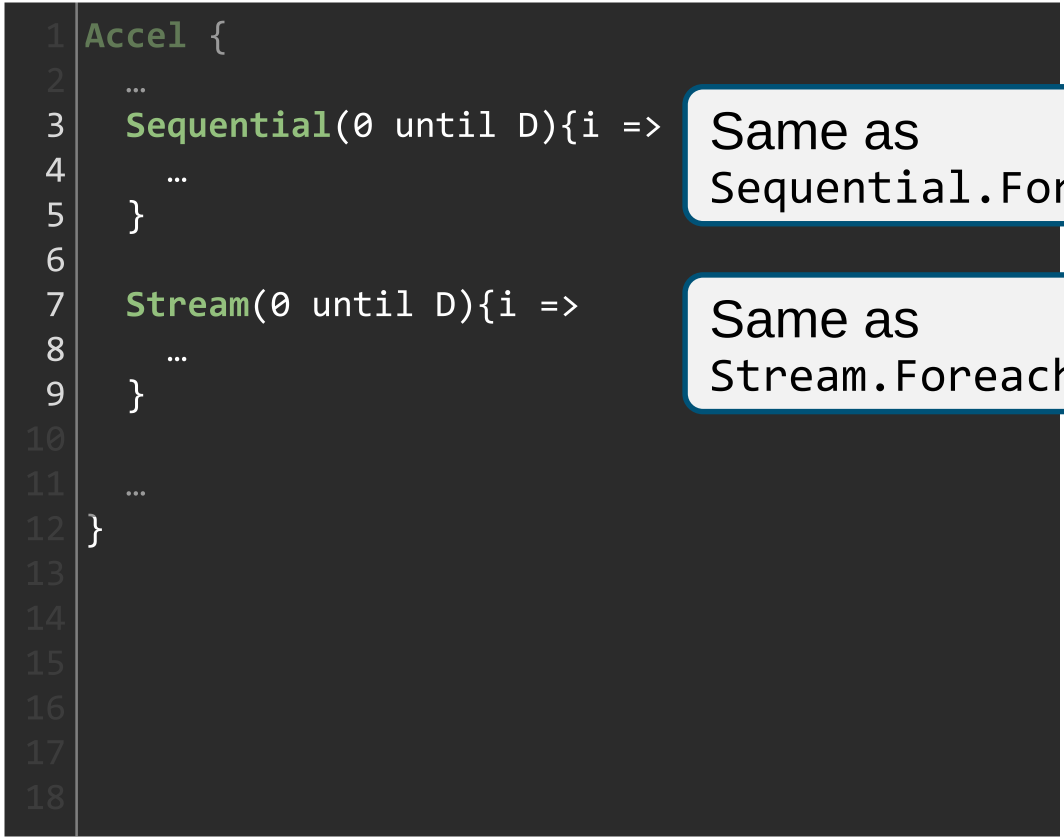


```
Accel {
  …
  Stream.Foreach(0 until D){i =>
    …
  }

  Stream.Reduce(0)(0 until D){i =>
    …
  }{ … }
```

Stream can be added as a tag on looping controllers to change execution of stages from pipelined to streaming

Note: This only makes sense if all communication is via FIFOs

Controller Tags: Foreach Shorthand



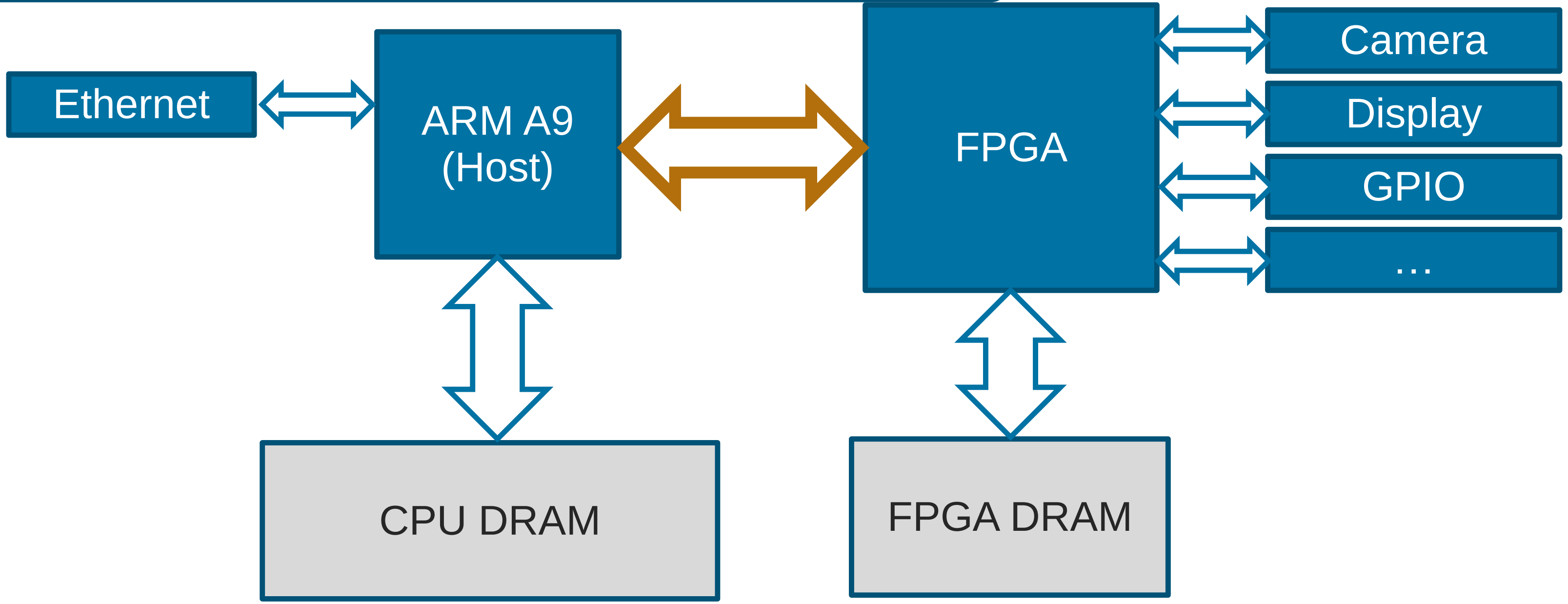
```
Accel {
  …
  Sequential(0 until D){i =>
    …
  }

  Stream(0 until D){i =>
    …
  }

  …
}
```

Same as
`Sequential.Foreach(0 until D)`

Same as
`Stream.Foreach(0 until D)`

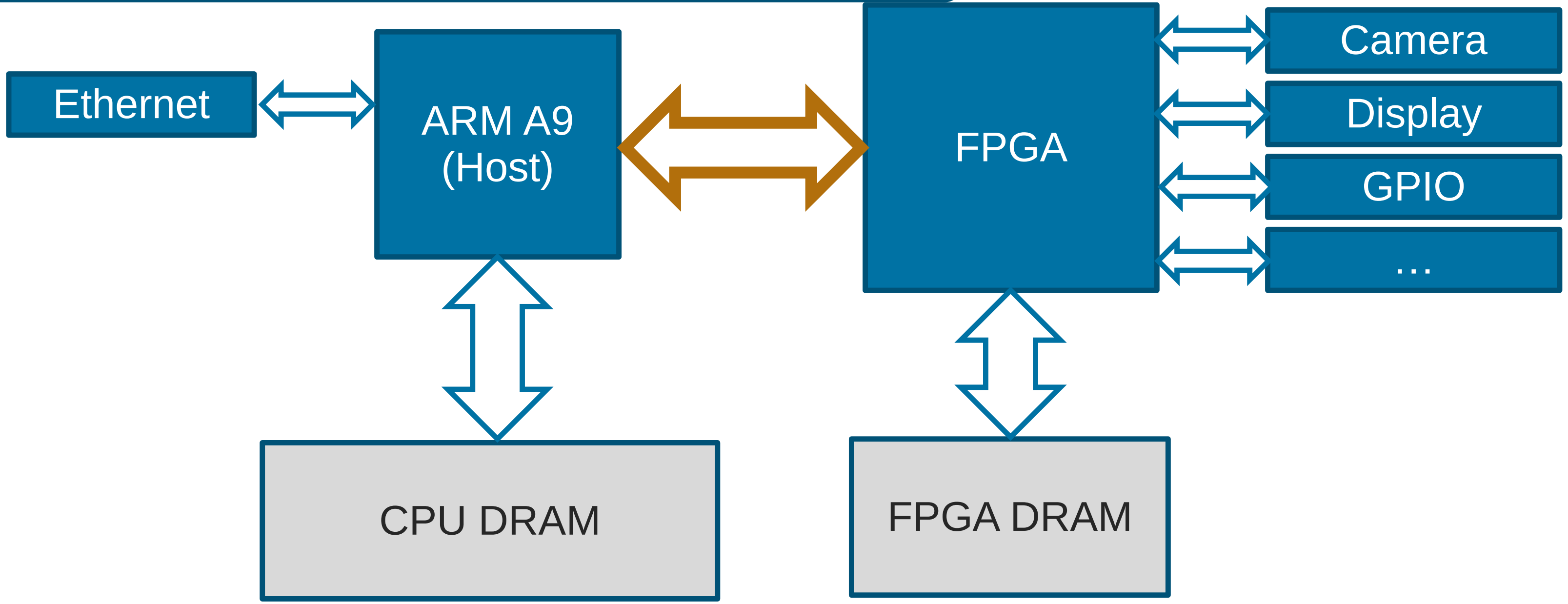
ARM <-> FPGA: Fine Grained Signaling

ArgIn and **ArgOut** are used for FPGA **initialization** and **returning** scalar results



ARM <-> FPGA: Fine Grained Signaling

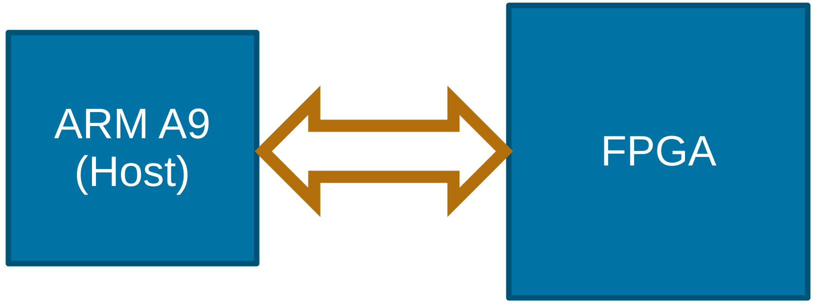
What to use for interaction between FPGA and host during non-blocking call?



HostIO

```
val signal = HostIO[Int]

Accel {
   …
}
```

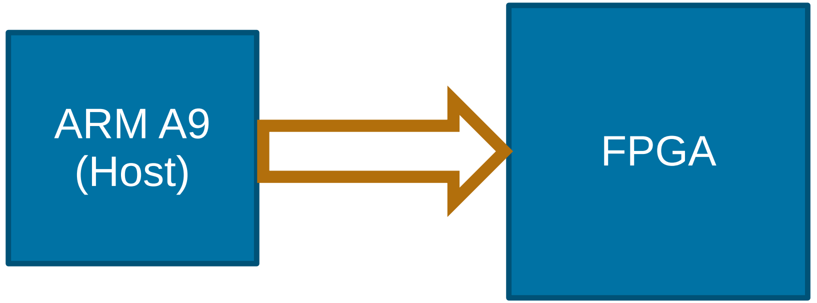
Defines a memory mapped register which the ARM can **write or read during** FPGA execution



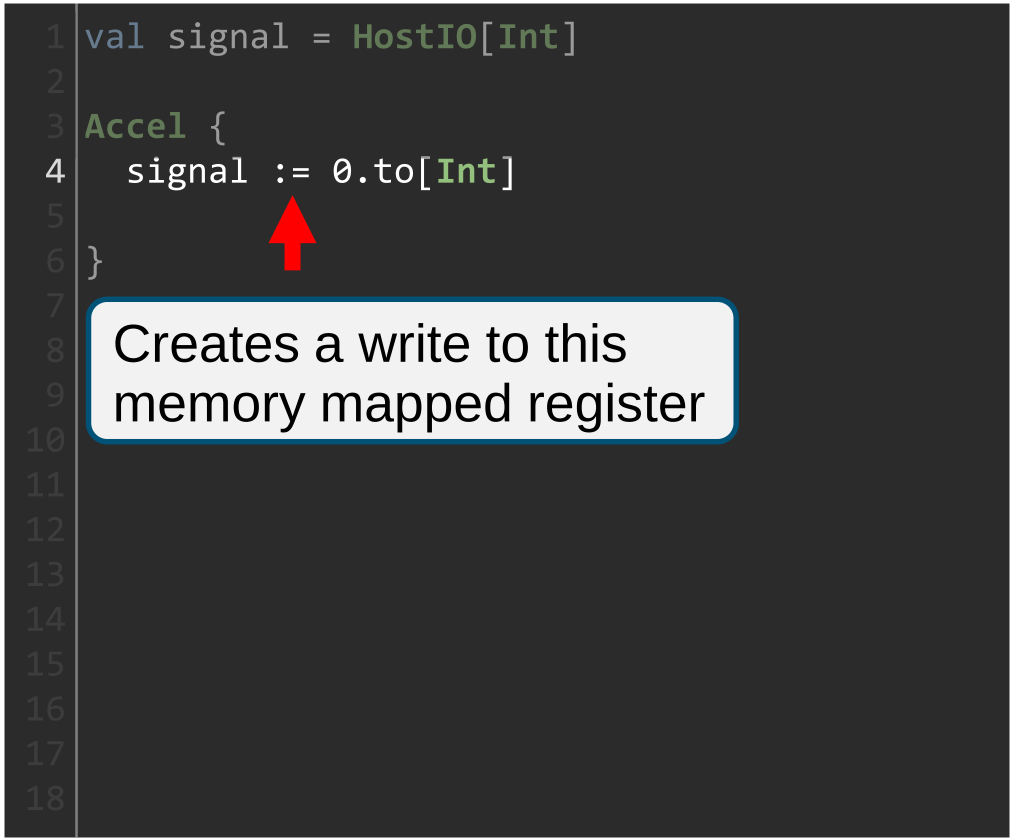
Reading from HostIO (on FPGA)

```
val signal = HostIO[Int]

Accel {
  val element = signal.value

}
```

Creates wires connected to this memory mapped register

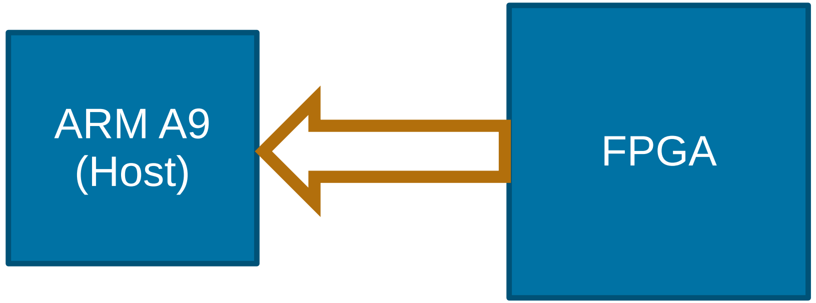


Writing to HostIO (on FPGA)



```
val signal = HostIO[Int]

Accel {
  signal := 0.to[Int]

}
```

Creates a write to this memory mapped register



Generally, it's good practice to use each **HostIO** for only one direction of communication

HOST CODE INTERFACE

Host Code (C++) Interface

- Spatial programs begin execution on host
  - Need a way to communicate with FPGA
- What kind of communication?
  - Initialize and configure the FPGA with bitstream
  - Read and write registers
  - Allocate FPGA DRAM
  - Move data between host and FPGA DRAM

FringeContext: C++ Execution Context

- FPGA exposed to host via **FringeContext**
  - To interact with FPGA, create a FringeContext object
- **FringeContext** implements necessary APIs
  - load(), malloc(), memcpy(), setArg(), getArg()..
  - Use same APIs for both simulation and on the board!

FringeContext: Example

```cpp
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int host_N = 10;
  int host_out = 0;
  int host_data = c1->malloc(sizeof(int) * host_N);

  c1->setArg(0, host_N);
  c1->setArg(1, host_data);

  c1->run();

  printf("out = %d\n", c1->getArg(0));
  return 0;
}

```

```scala
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}

```

FringeContext: Example

```
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int
  int
  int                                                    );

  c1->setArg(0, host_N);
  c1->setArg(1, host_data);

  c1->run();

  printf("out = %d\n", c1->getArg(0));
  return 0;
}

```

Create a new **FringeContext** with "path to bitstream"

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}

```

FringeContext: Example

```c
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int host_N = 10;
  int host_out = 0;
  int host_data = c1-

  c1->setArg(0, host_N);
  c1->setArg(1, host_data);

  c1->run();

  printf("out = %d\n", c1->getArg(0));
  return 0;
}

```

Reset and program the FPGA by loading the bitstream

```scala
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}

```

FringeContext: Example

```
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int host_N = 10;
  int host_out = 0;
  int host_data = c1->malloc(sizeof(int) * host_N);






  printf("out = %d\n", c1->getArg(0));
  return 0;
}

```

Create host params to be passed to FPGA

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}

```

FringeContext: Example

```
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int host_N = 10;
  int host_out = 0;
  int host_data = c1->malloc(sizeof(int) * host_N);

  c1->setArg(0, host_N);
  c1->setArg(1, host_data);




             ->getArg(0));

}

```

Set ArgIns, which includes pointers to DRAMs

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}

```

FringeContext: Example

```
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int host_N = 10;
  int host_out = 0;
  int host_data = c1->malloc(sizeof(int) * host_N);

  c1->setArg(0, host
  c1->setArg(1, host

  c1->run();

  printf("out = %d\r
  return 0;
}
```

Start **Accel** block on FPGA (code on right) with the given **FringeParams**

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}
```

FringeContext: Example

```c
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int host_N = 10;
  int host_out = 0;
  int host_data = c1->malloc(sizeof(int) * host_N);

  c1->setArg(0, host_N);
  c1->setArg(1, host_data);

  c1->run();

  printf("out = %d\n", c1->getArg(0));
  return 0;
}
```

Print results

```scala
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}
```

FringeContext: Example

```c
int main(int argc, char **argv) {
  FringeContext *c1 = new FringeContext("accel.bit");

  c1->load();

  int host_N = 10;
  int host_out = 0;
  int host_data = c1->malloc(sizeof(int) * host_N);

  c1->setArg(0, host_N);
  c1->setArg(1, host_data);

  c1->run();

  printf("out = %d\n", c1->getArg(0));
  return 0;
}

```

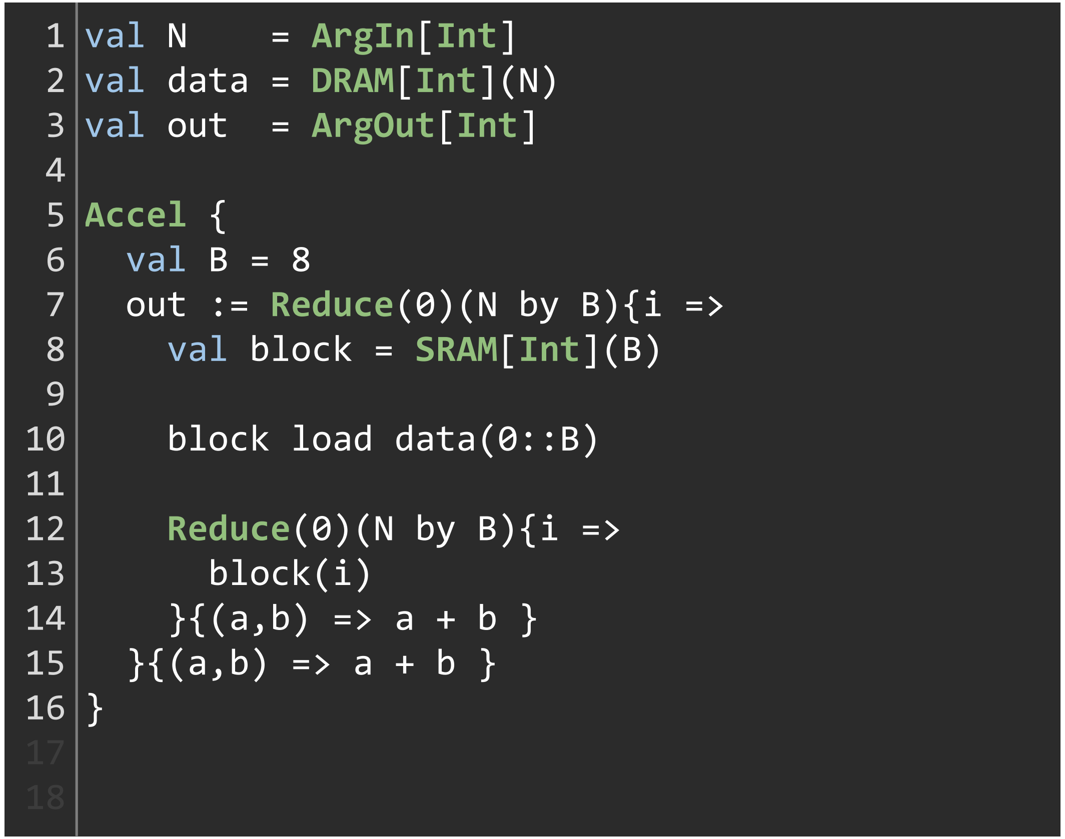
```scala
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::32)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}

```

DEBUGGING

Execution Modes: Functional Simulation

- Functional Simulation
  - Generates Scala code
  - NOT cycle accurate
  - Simulates sequential behavior of program
  - Executes on CPU only
  - `print/println` are fully supported

Buggy Spatial Program



```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 8
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(0::B)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}
```

Buggy Spatial Program: Debugging

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 8
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)
    block load data(0::32)
    print("[i = " + i + "] ")
    Foreach(B by 1){i => print(block(i) + " ") }
    println("")

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}
```

Printing works like any other primitive operation in functional simulation

Functional Simulation Running

```
$ bin/spatial --sim MyApp
$ ./MyApp.sim
$
$ [i = 0] 0 1 2 3 4 5 6 7
$ [i = 1] 0 1 2 3 4 5 6 7
$ [i = 2] 0 1 2 3 4 5 6 7
$ [i = 3] 0 1 2 3 4 5 6 7
  …
```

Fixed Spatial Program

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val B = 32
  out := Reduce(0)(N by B){i =>
    val block = SRAM[Int](B)

    block load data(i::i+B)

    Reduce(0)(N by B){i =>
      block(i)
    }{(a,b) => a + b }
  }{(a,b) => a + b }
}


```

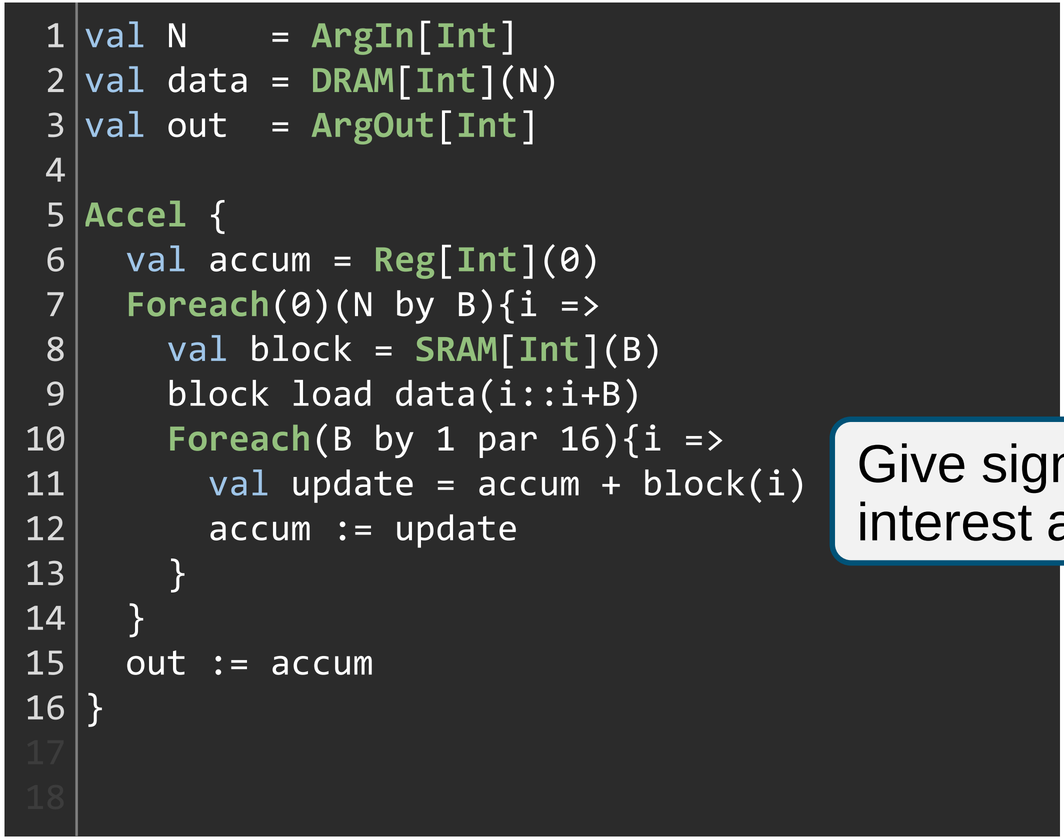
Execution Modes: Simulation with VCS

- Simulation with VCS
  - Spatial → Chisel/C++ → Verilog/C++
  - Cycle accurate with DRAM memory models
  - “Developer mode” with all control + data signals
  - “Named mode” where only named `vals` are shown
  - (`print/println` aren’t yet supported)

Buggy Spatial Program

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val accum = Reg[Int](0)
  Foreach(0)(N by B){i =>
    val block = SRAM[Int](B)
    block load data(i::i+B)
    Foreach(B by 1 par 16){i =>
      accum := accum + block(i)
    }
  }
  out := accum
}
```

Buggy Spatial Program



```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val accum = Reg[Int](0)
  Foreach(0)(N by B){i =>
    val block = SRAM[Int](B)
    block load data(i::i+B)
    Foreach(B by 1 par 16){i =>
      val update = accum + block(i)
      accum := update
    }
  }
  out := accum
}
```

Give signal of interest a name

Functional Simulation Running

```
$ bin/spatial --synth MyApp
$ ./MyApp.vcs
$
```

Makes the VCS project
Runs VCS cycle-accurate simulation
Starts GtkWave (waveform viewer)

Note: Requires VCS and GtkWave!

Fixed Spatial Program

```
val N    = ArgIn[Int]
val data = DRAM[Int](N)
val out  = ArgOut[Int]

Accel {
  val accum = Reg[Int](0)
  Foreach(0)(N by B){i =>
    val block = SRAM[Int](B)
    block load data(i::i+B)
    Foreach(B by 1 ~~par 16~~){i =>
      val update = accum + block(i)
      accum := update
    }
  }
  out := accum
}
```

Performance Debugging

- See http://spatial-lang.readthedocs.io/en/latest/tutorial/gemm.html for information about how to debug performance

Deploy to Various Targets

- Currently supported FPGA targets:
  - Amazon AWS F1 -