CS 217 Lecture 5

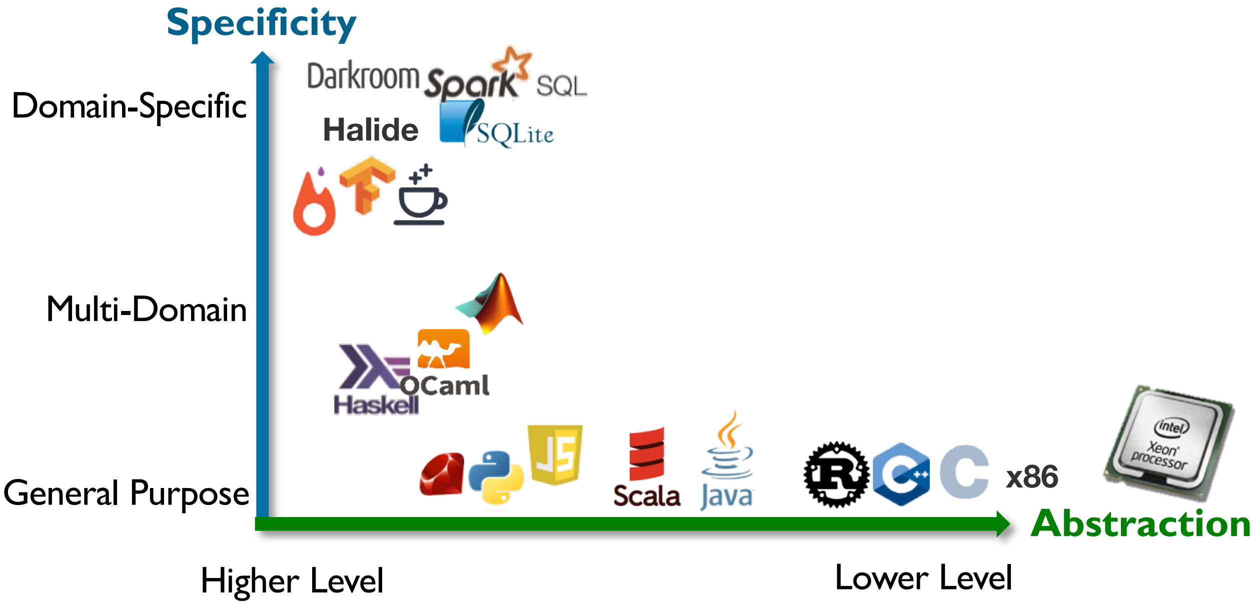
Introduction to Spatial

Fall 2018

Considerations for Designing ML HW

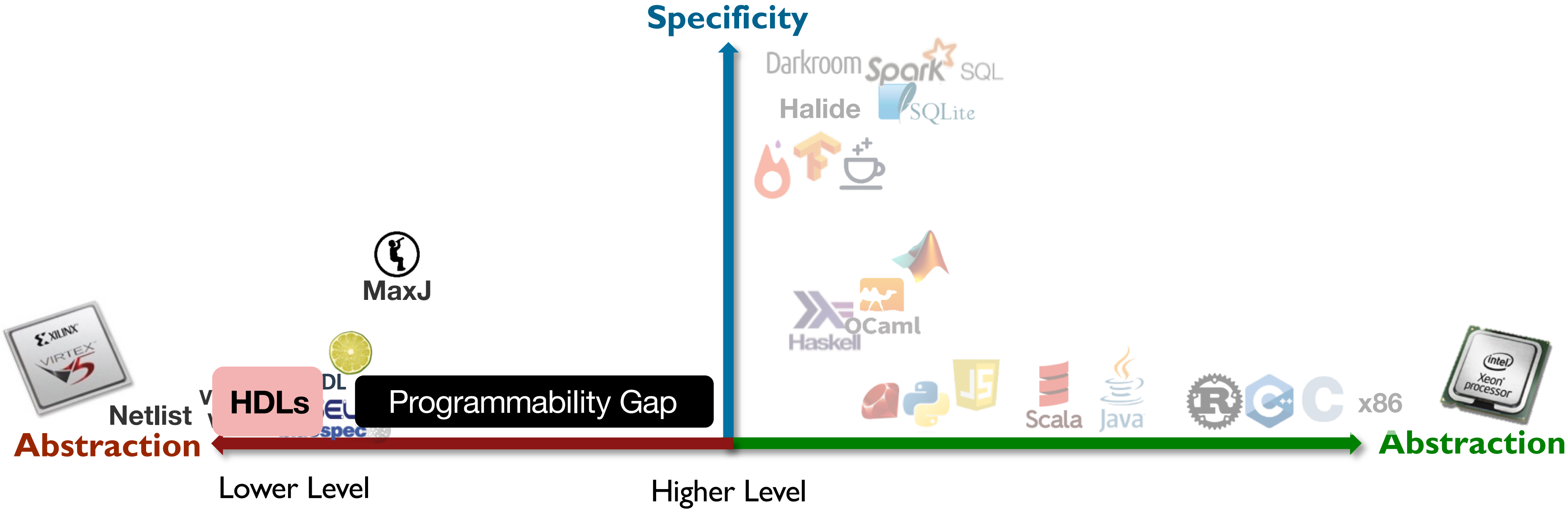
- Pipelining
  - For better performance, keep as much of the HW as busy as possible at once
- Custom memory hierarchy
  - Lots of room to specialize memory hierarchy for specific access patterns
  - But custom memory hierarchy requires custom partitioning, allocation, etc.
- Finite physical compute and memory resources
  - Modern deep ML models require significant amount of compute and memory
- Huge parameter design spaces
  - Different machine learning applications have application-specific bottlenecks
  - The bottlenecks could be removed by choosing the right set of design parameters

Language Taxonomy



Instruction-Based Architectures

Language Taxonomy



Specificity

Abstraction

Abstraction

Lower Level

Higher Level

Netlist

HDLs

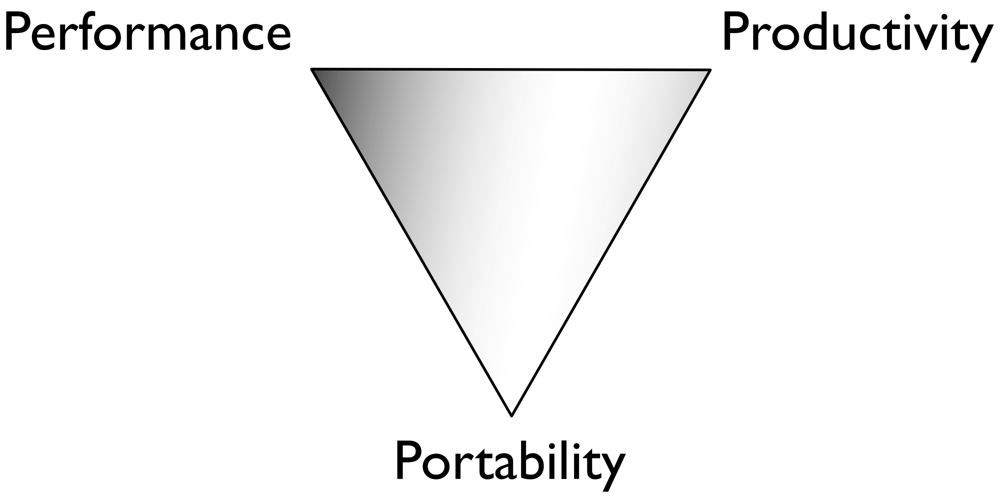
Programmability Gap

Reconfigurable Architectures

Instruction-Based Architectures

Language Requirements

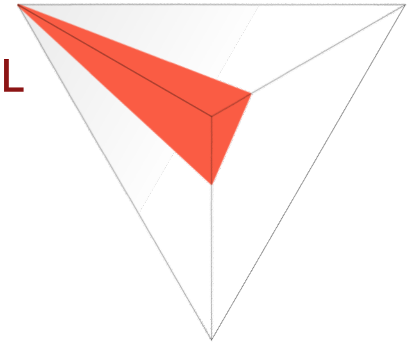
1. Performant: generates efficient hardware
2. Productive: High-level language for "power" users
3. Portable: Target-generic source



Language Comparisons

Hardware Description Languages (HDLs)

e.g. Verilog, VHDL, Chisel, Bluespec



Performance

✓ Arbitrary RTL

Productivity

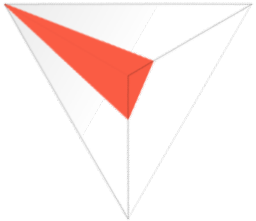
✗ No high-level abstractions

Portability

✗ Significant target-specific code

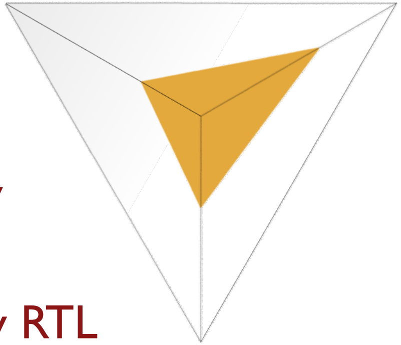
Language Comparisons

HDLs



High Level Synthesis Tools (HLS)

e.g. SDAccel, OpenCL



Performance

- ✗ No memory hierarchy
- ✗ No arbitrary pipelining
- ✗ No arbitrary RTL

Productivity

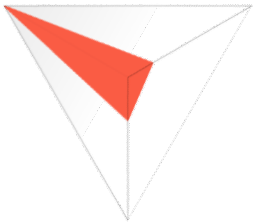
- ✓ Nested loops
- ✗ Difficult to tune
- ✗ Banking, unrolling require pragmas

Portability

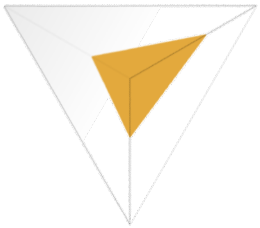
- ✓ Single vendor

Language Comparisons

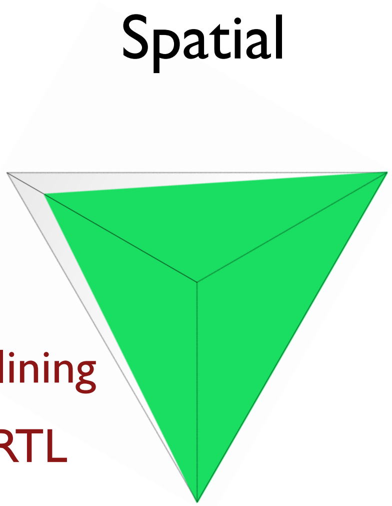
HDLs



HLS



Spatial



Performance

- ✓ Memory hierarchy
- ✓ Arbitrary pipelining
- ✗ No arbitrary RTL

Productivity

- ✓ Nested loops
- ✓ Automatic memory banking/buffering
- ✓ Implicit design parameters (unrolling, banking, etc.)
- ✓ Automated design tuning

Portability

- ✓ Target-generic source

Introducing Spatial

- Programming language to simplify accelerator design
  - Simple APIs to manage Host <-> Accelerator communication
  - Built-in 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

Things Spatial (the Language) Has

✓ Nest-able control structures (like software)

✓ Arbitrary pipelining of loops

✓ Memory hierarchy (SRAM vs. DRAM)

✓ Specialized memory types (e.g. FIFO, RegFile)

✓ Streaming abstractions (StreamIn, StreamOut)

✓ High level host/Accelerator interfaces

Things Spatial's Compiler Does

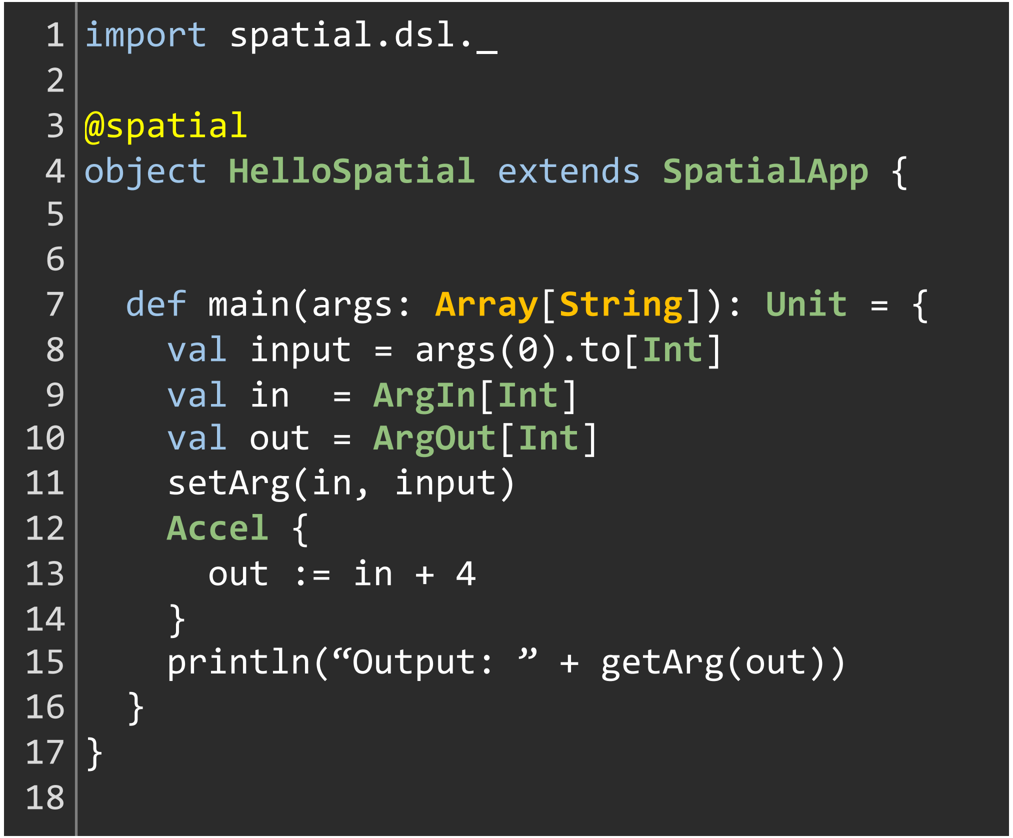
- ✓ Retiming (register insertion)
- ✓ Control scheduling
  - Scheduling of inner loops (using dependencies) is done
  - Scheduling of outer loops (also dependencies) soon!
- ✓ Multi-ported memory and buffer inference
  - Infers multiple ports for multiple concurrent accesses
  - Infers buffer depth for accesses across pipeline stages
- ✓ Some miscellaneous hardware optimizations
- ✓ Automated design tuning (updating soon)

Things Spatial's Compiler Doesn't (Yet)

- ✗ Automatic loop/data tiling
  - Splitting loops up
  - Inference of DRAM/SRAM + transfers
- ✗ Automatic work allocation
  - Breaking computation up into host/Accel regions
- ✗ Loop fusion
  - Merging of loops which have element-wise dependencies
  - Elimination of memories between such loops

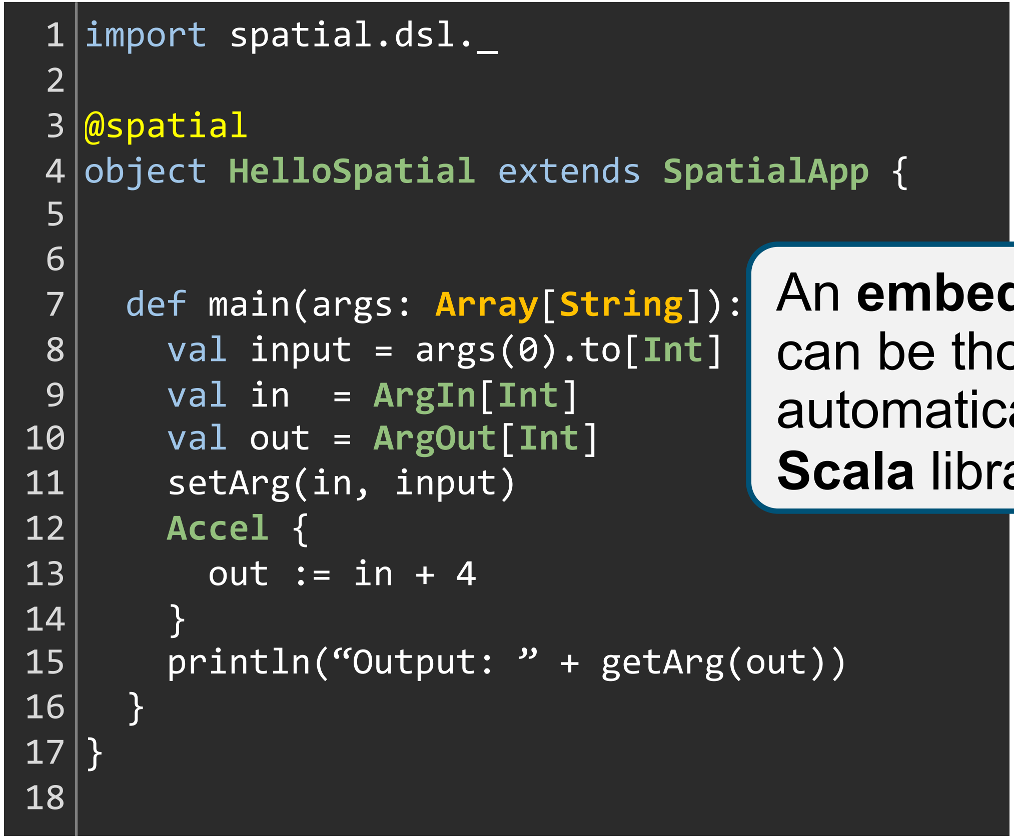
SPATIAL (AND SCALA) BASICS

Hello Spatial!



```scala
import spatial.dsl._

@spatial
object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): 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))
  }
}
```

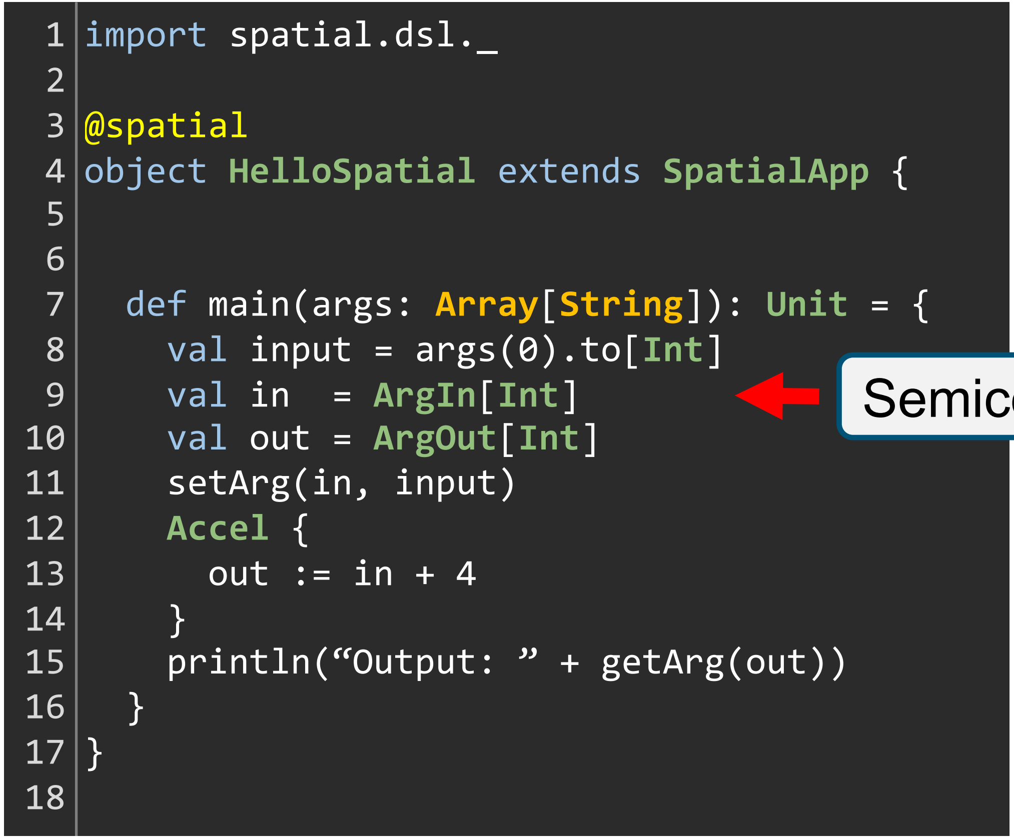
Spatial is an *Embedded DSL* in **Scala**



```
import spatial.dsl._

@spatial
object HelloSpatial extends SpatialApp {


  def main(args: Array[String]):
    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))
  }
}

```

An **embedded DSL** can be thought of as an automatically optimized **Scala** library.

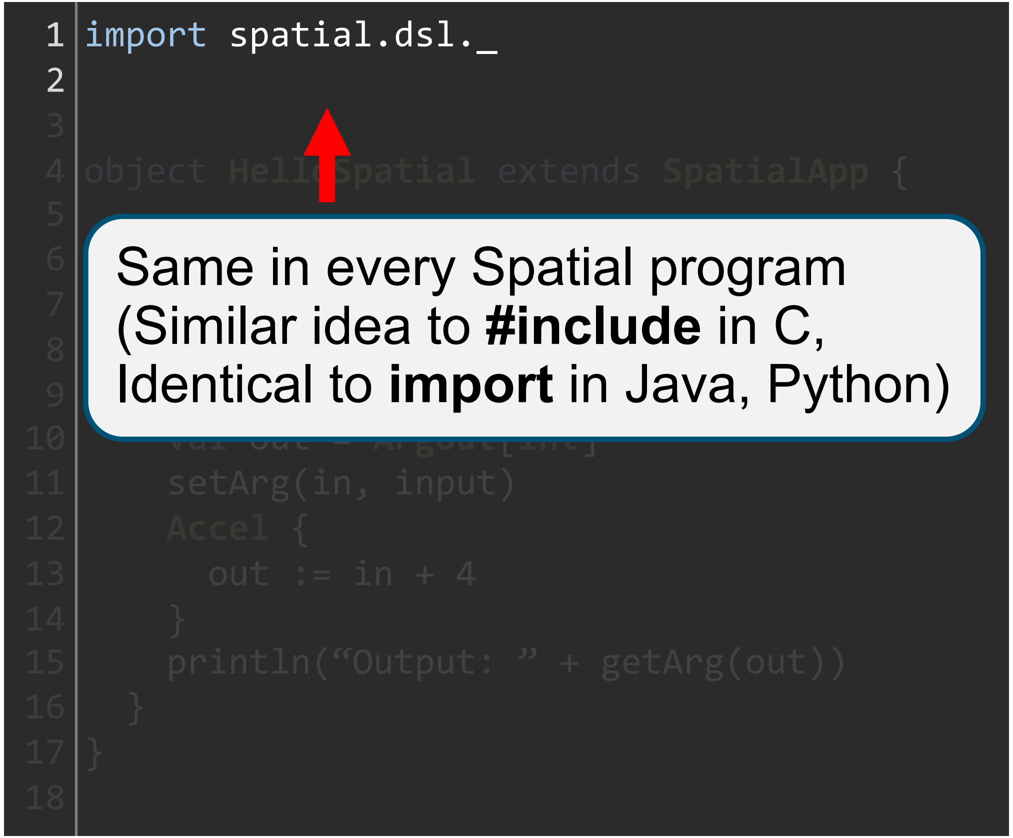
Scala Basics



```scala
import spatial.dsl._

@spatial
object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): 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

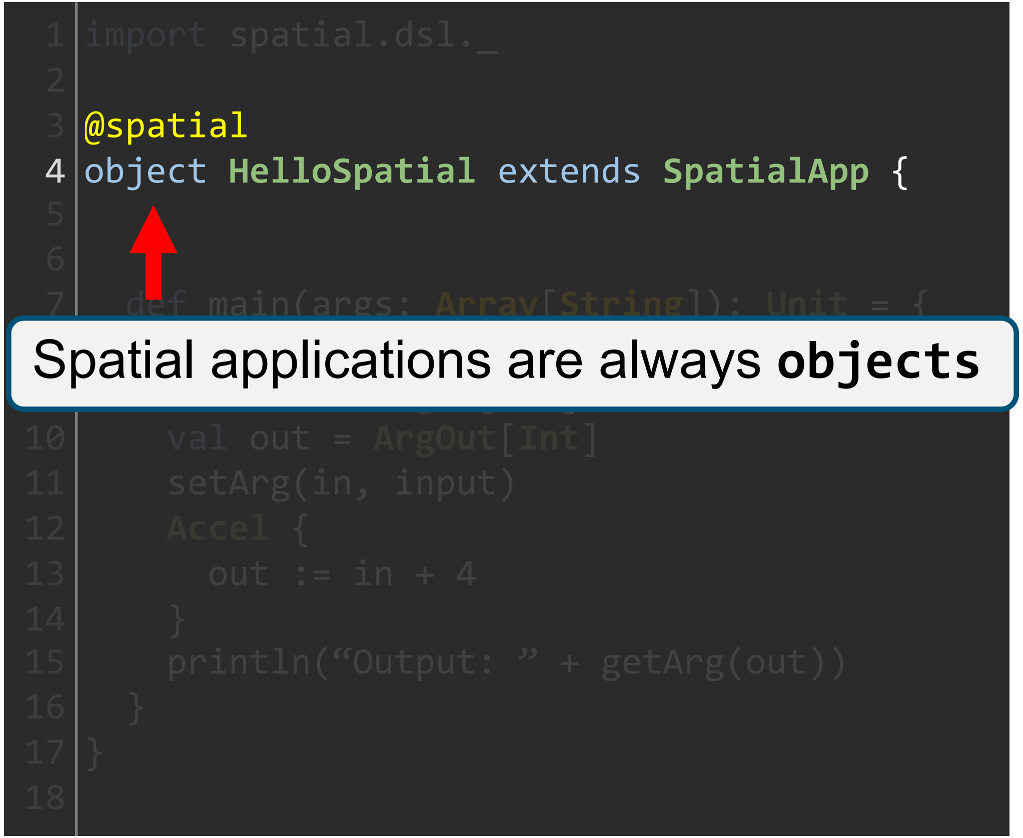
Import Statements



```
import spatial.dsl._


object HelloSpatial extends SpatialApp {





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

```

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

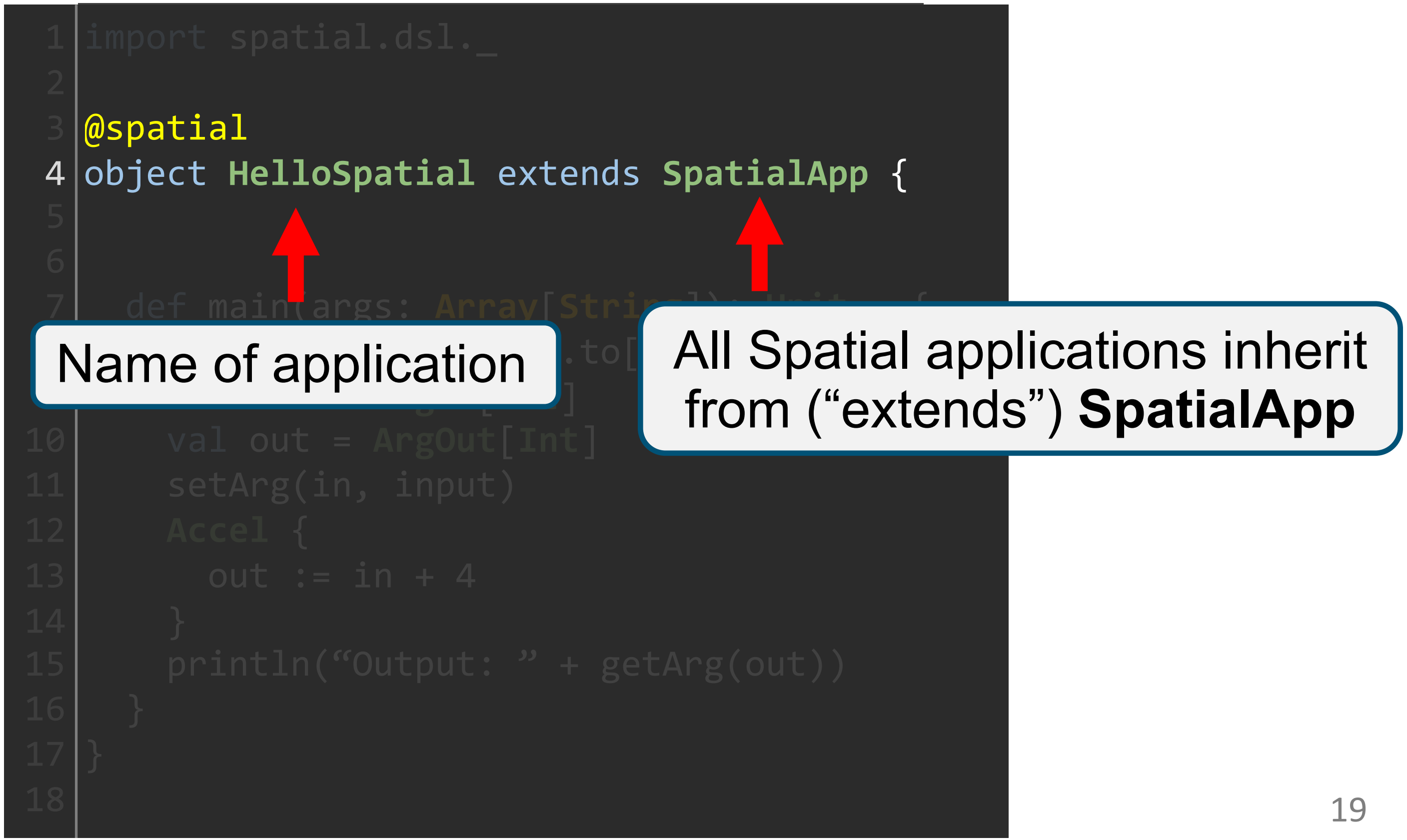
Application Object Declaration



```
import spatial.dsl._

@spatial
object HelloSpatial extends SpatialApp {


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

Spatial applications are always **objects**

Application Object Declaration

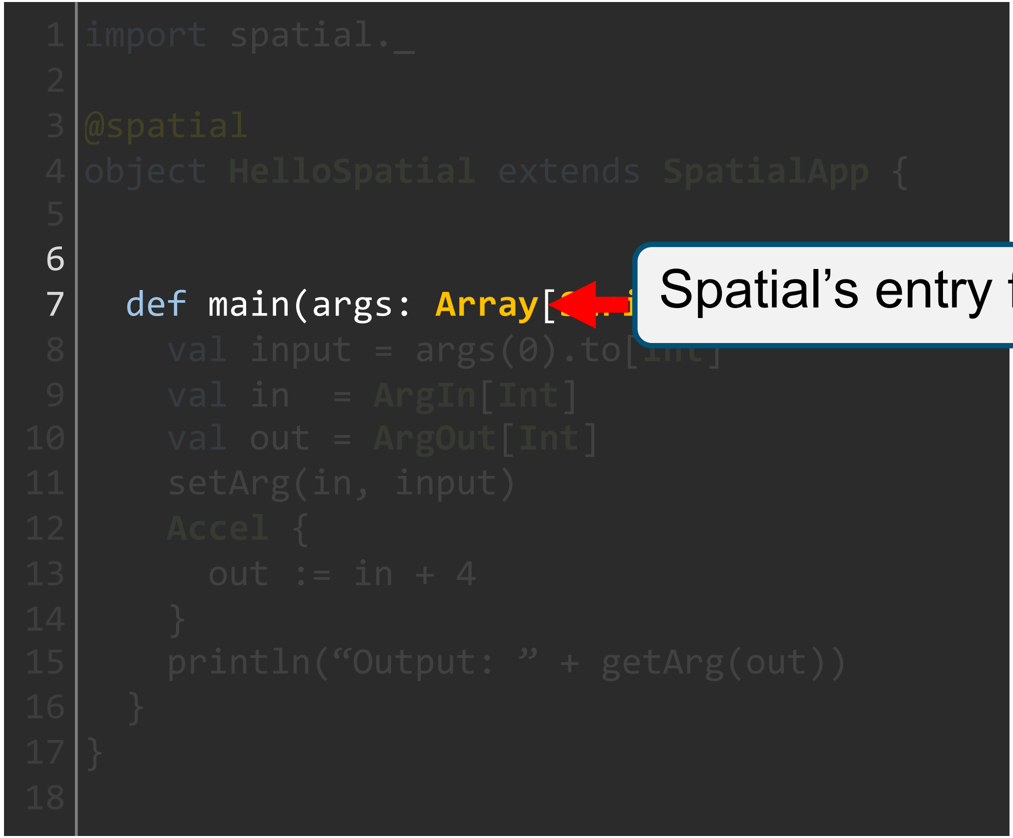


```
import spatial.dsl._

@spatial
object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): 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**

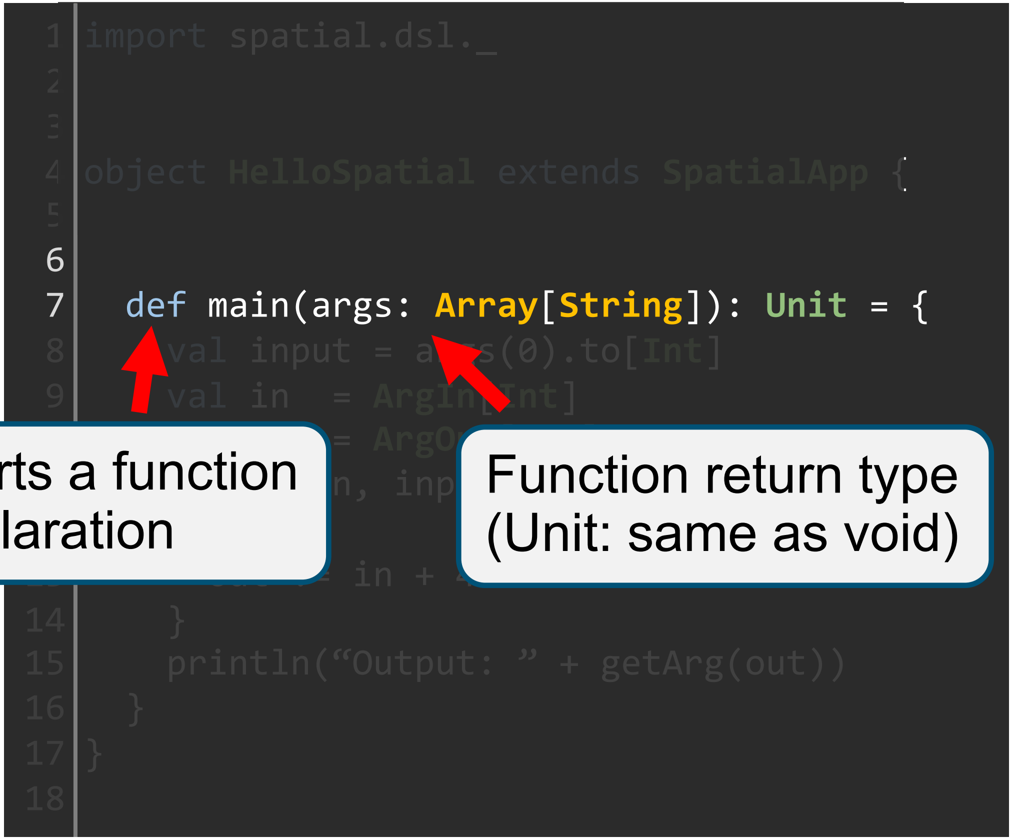
Spatial’s Entry Function: “main()”



```
import spatial._

@spatial
object HelloSpatial extends SpatialApp {

  def main(args: Array[
    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

Spatial’s Entry Function: “main()”

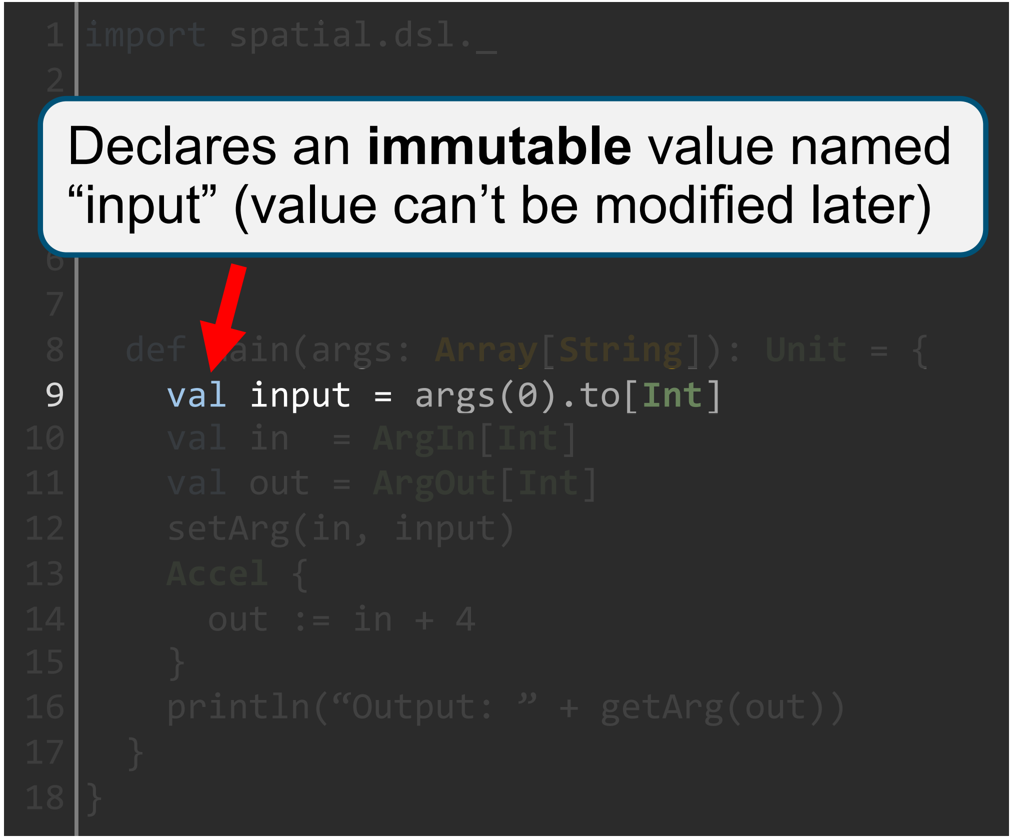


```
import spatial.dsl._


object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): Unit = {
    val input = args(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)

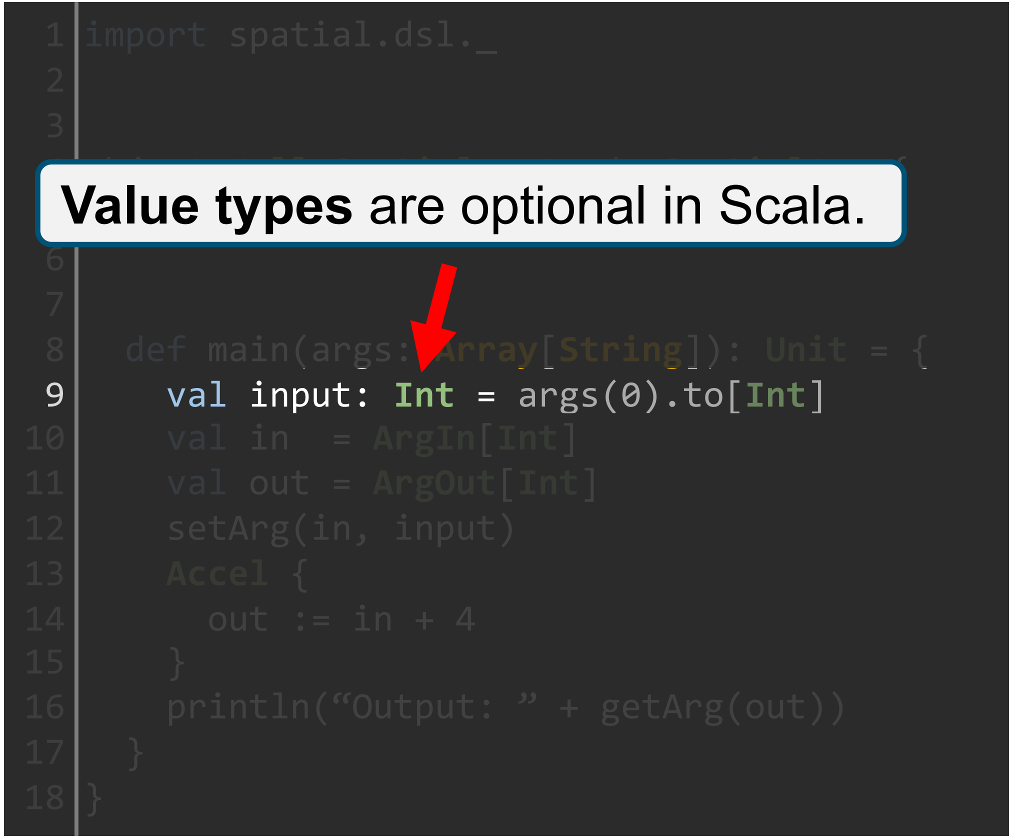
Val Definitions



```
import spatial.dsl._


  def main(args: Array[String]): 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)

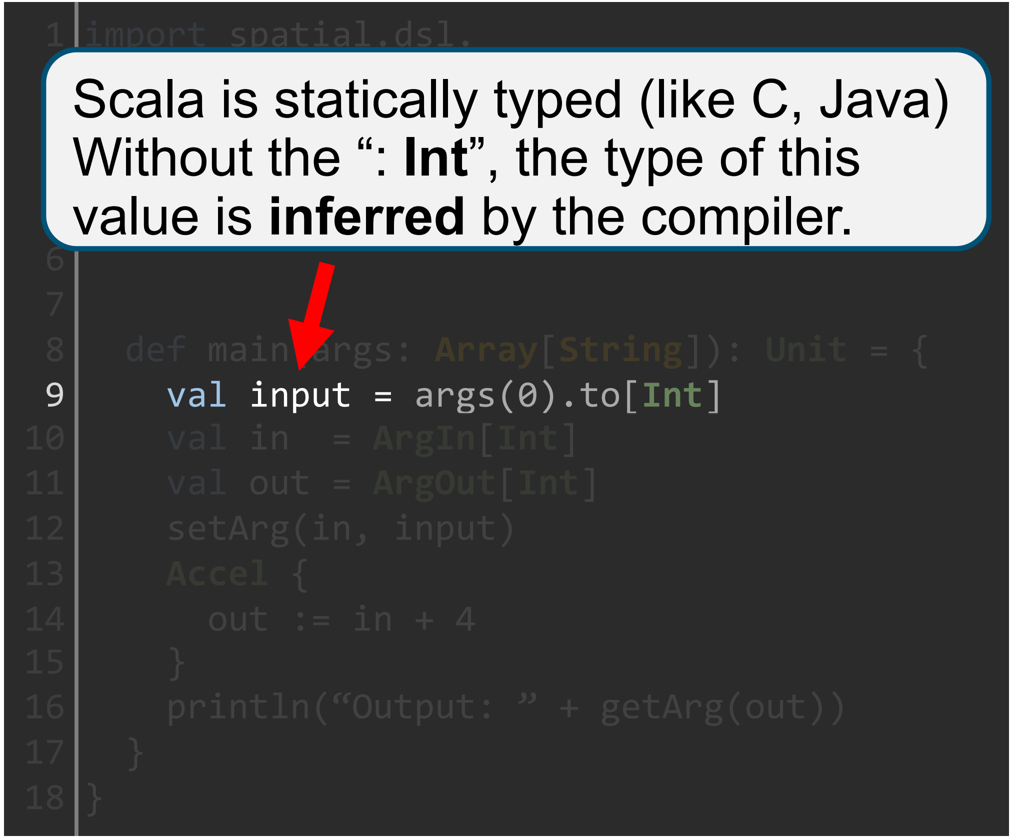
Val Definitions



```
import spatial.dsl._




  def main(args: Array[String]): 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.

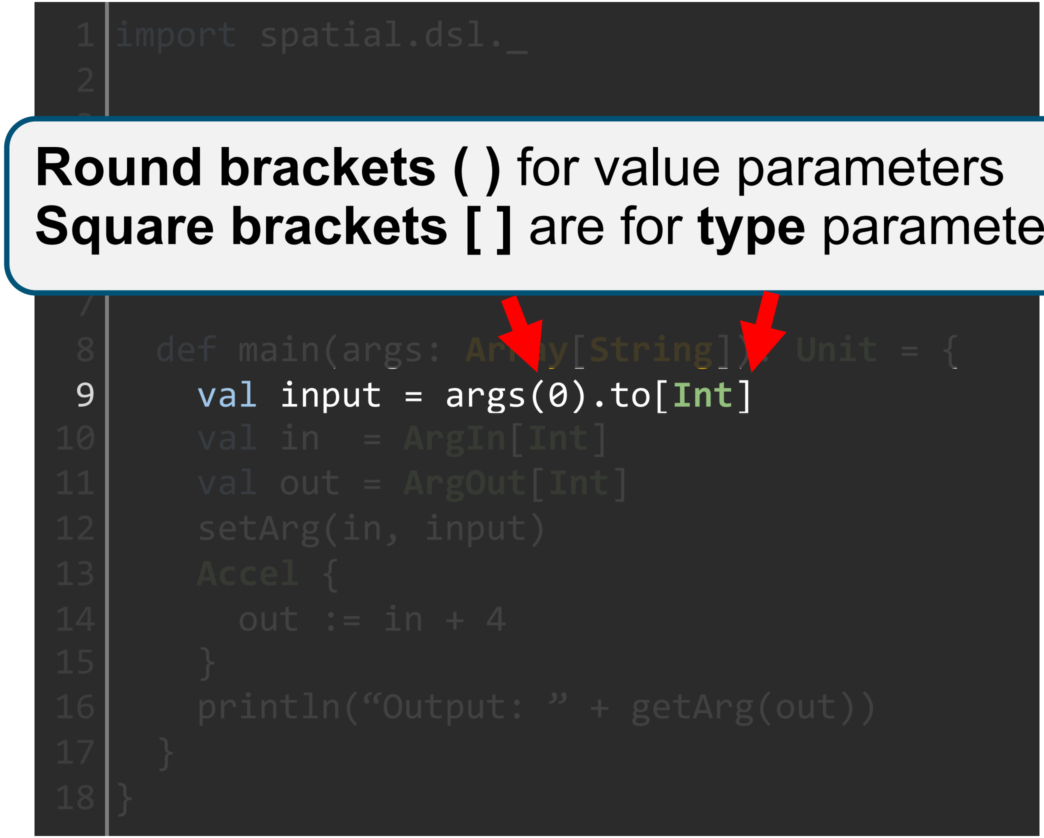
Val Definitions



```
import spatial.dsl.



  def main(args: Array[String]): 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.

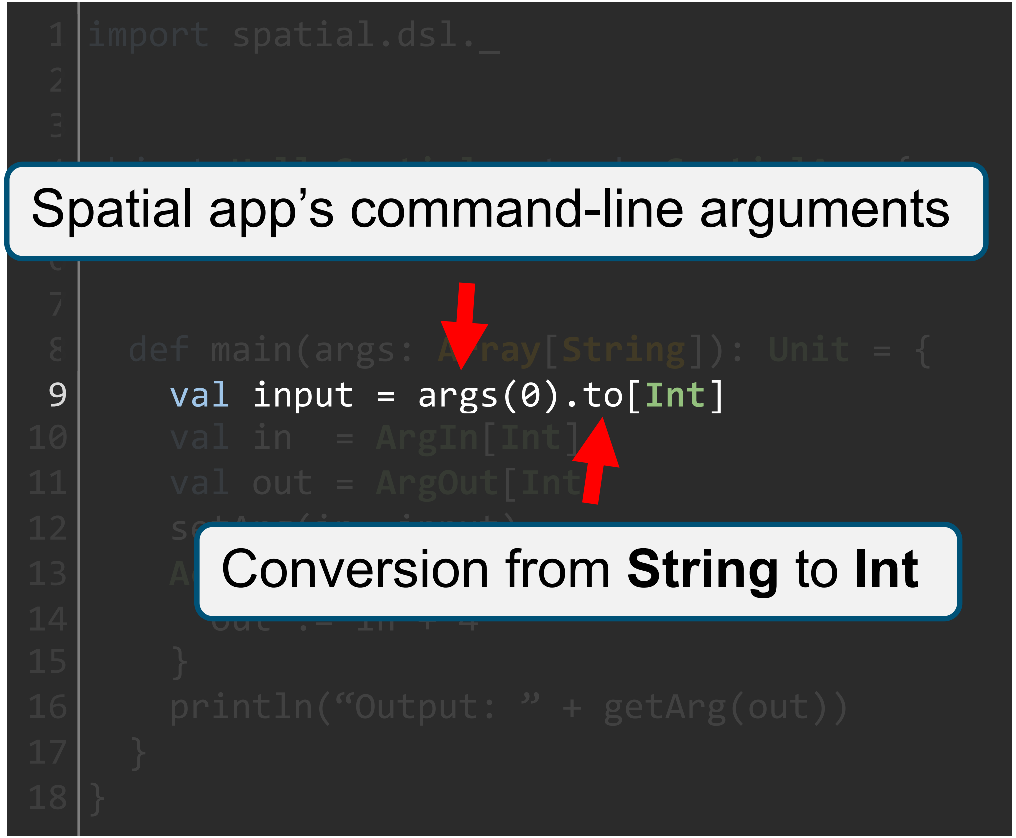
Method Calls



```
import spatial.dsl._


  def main(args: Array[String]): 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



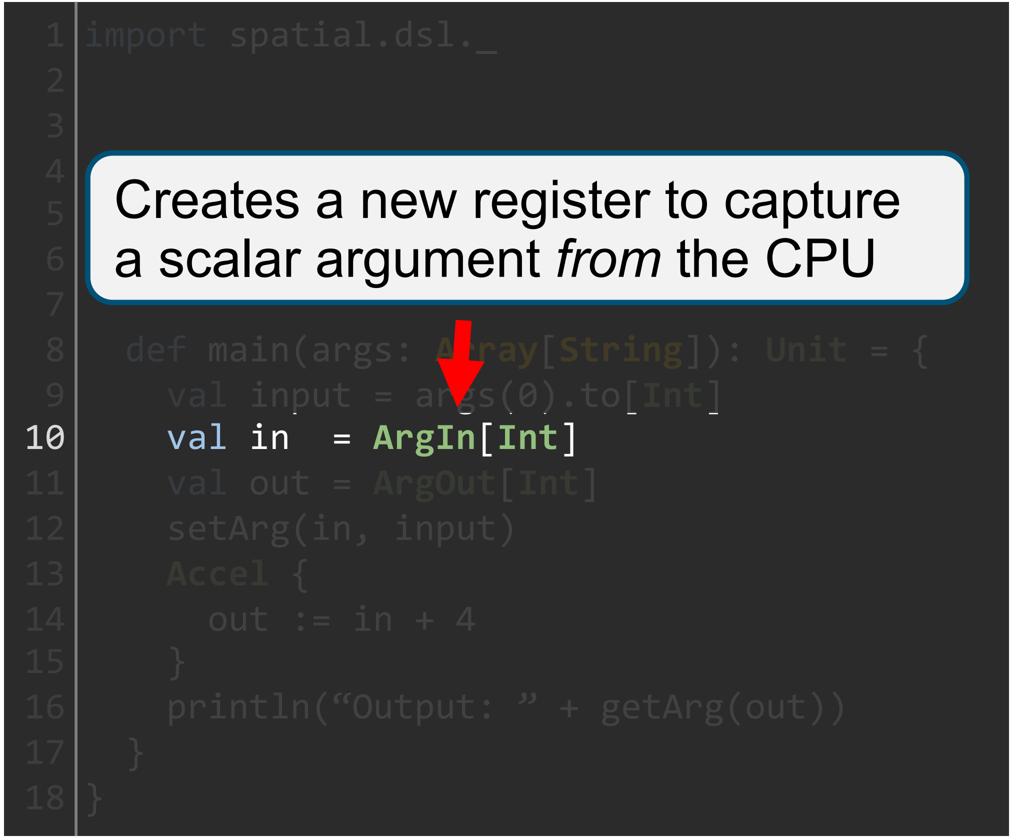
```
import spatial.dsl._



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

Spatial app’s command-line arguments

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

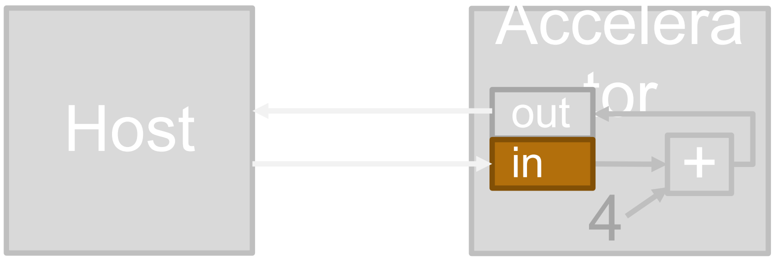
```
int main(int argc, char **argv) {
  int in = atoi(argv[1]);

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

Input Arguments (ArgIn)

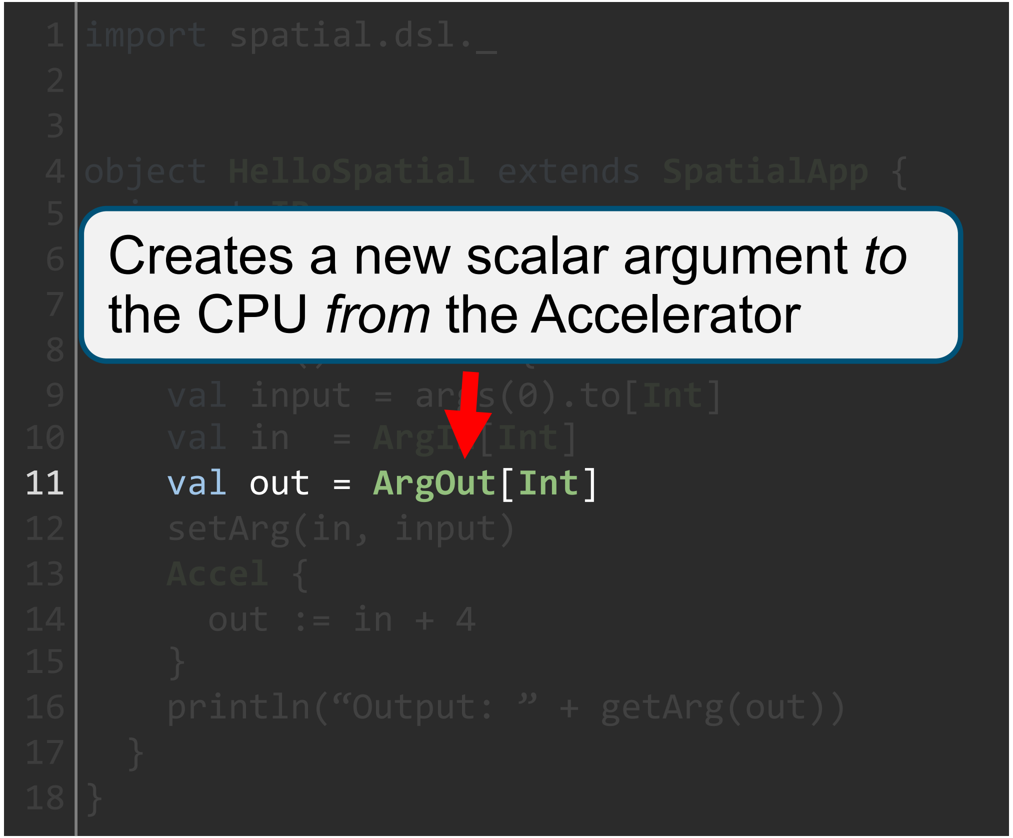


```
import spatial.dsl._



  def main(args: Array[String]): 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 CPU

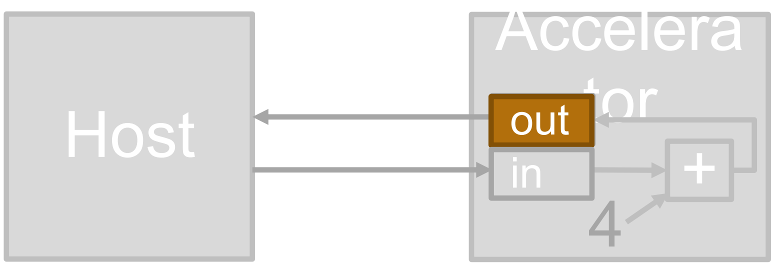


Output Arguments (ArgOut)

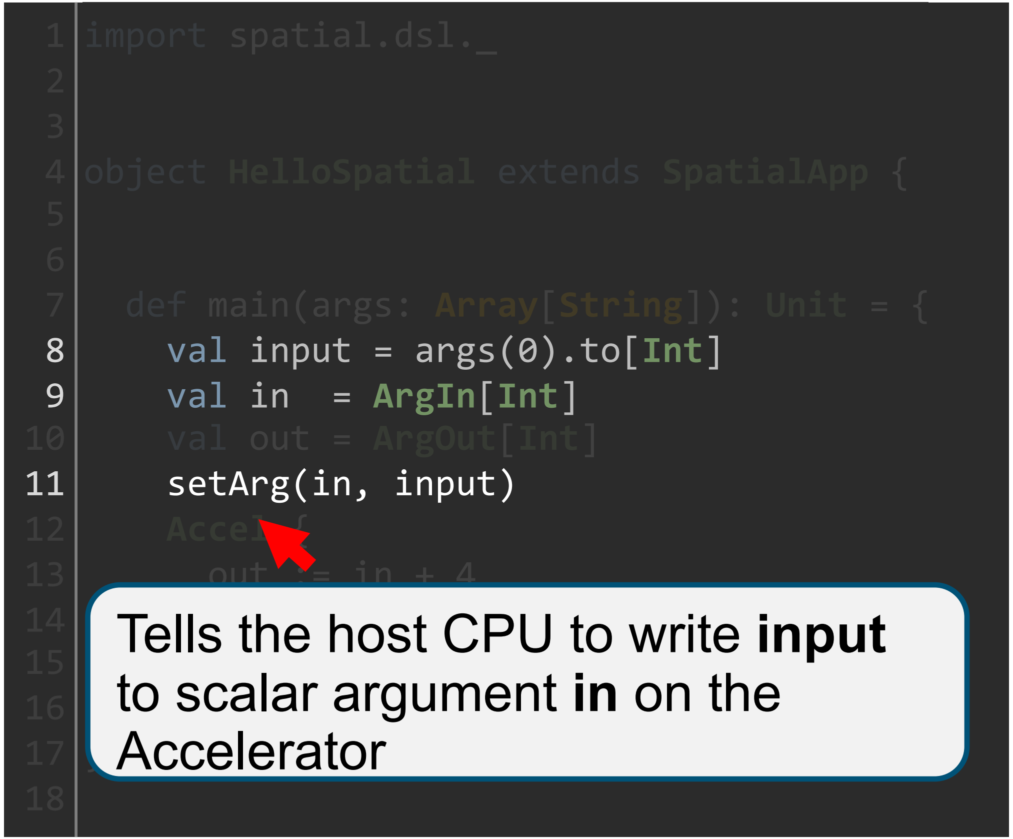


```
import spatial.dsl._


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

Creates a new scalar argument *to* the CPU *from* the Accelerator

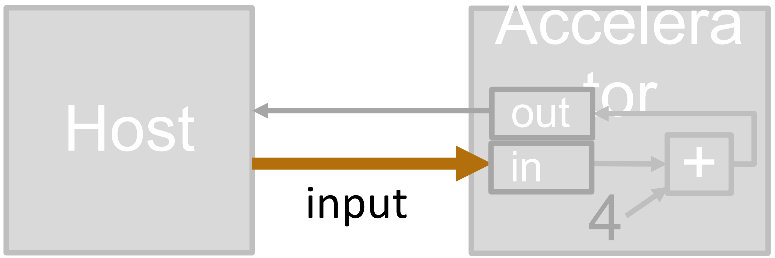


Scalar Transfers (CPU → Accelerator)

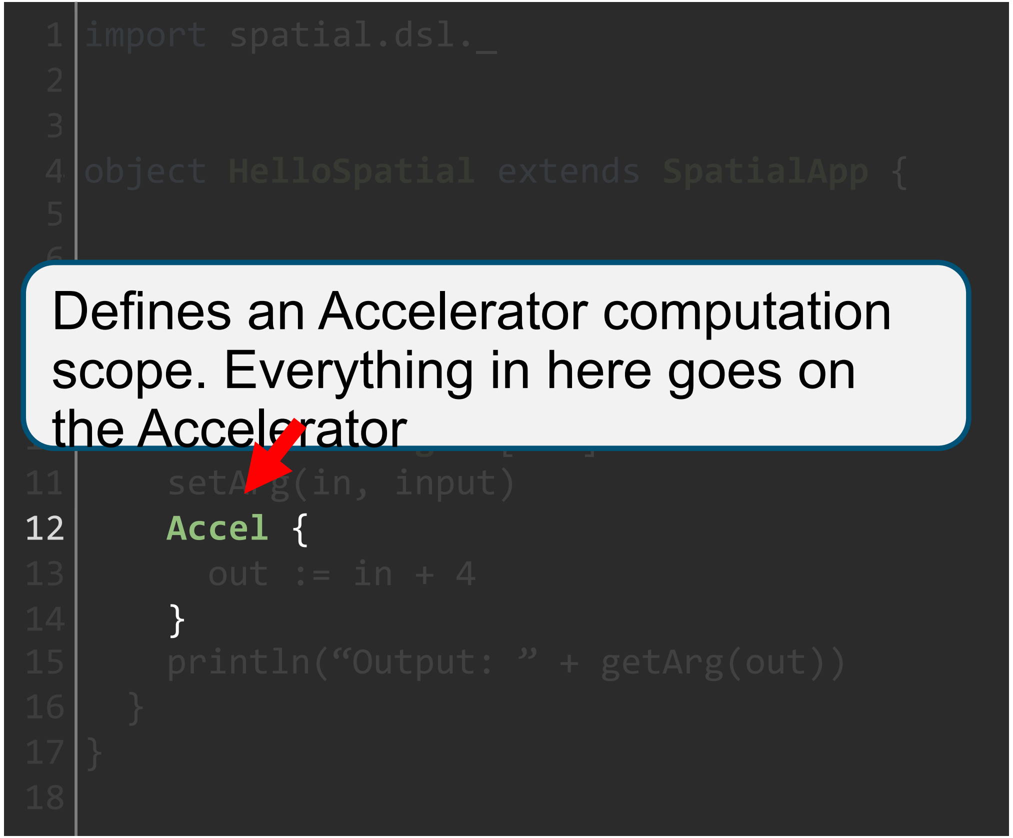


```
import spatial.dsl._


object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]
    val out = ArgOut[Int]
    setArg(in, input)
    Accel {
      out := in + 4
```

Tells the host CPU to write **input** to scalar argument **in** on the Accelerator

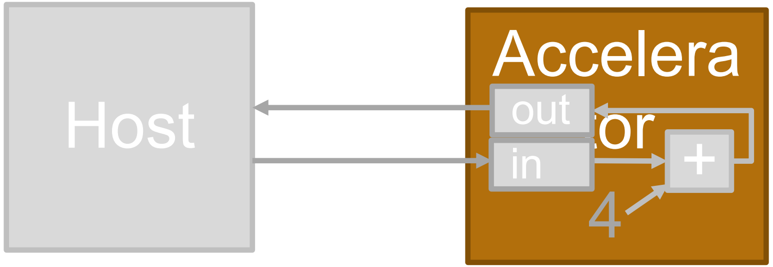


Accel Block

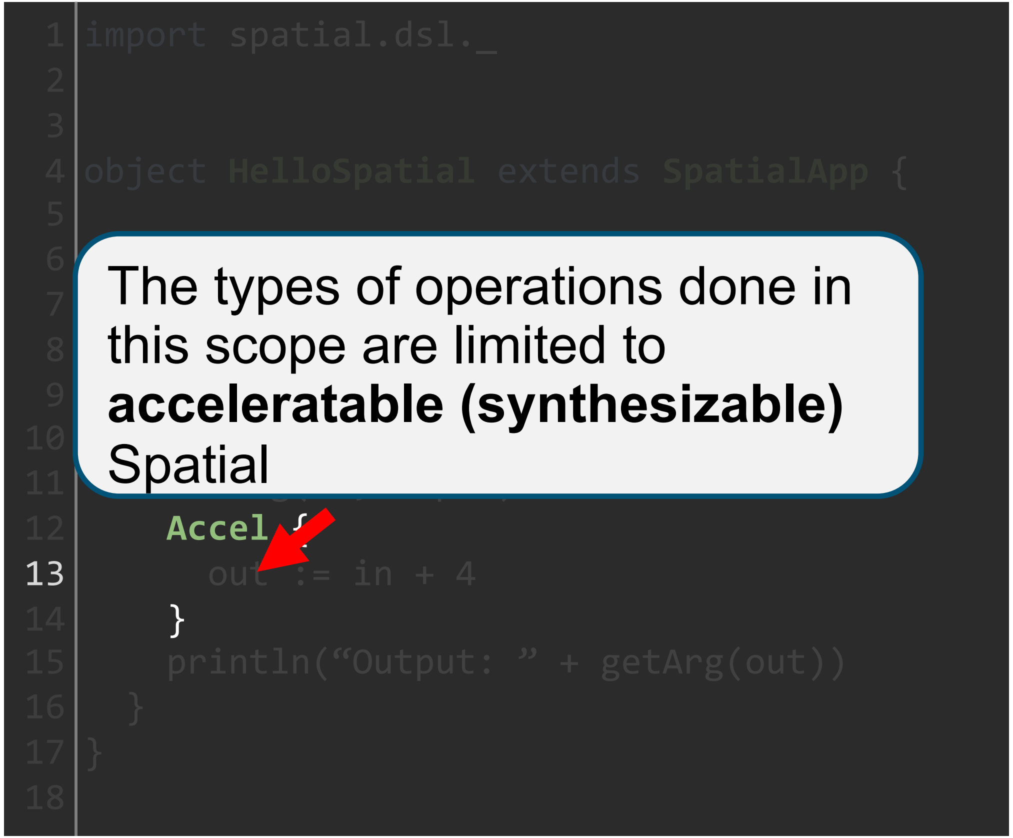


```
import spatial.dsl._


object HelloSpatial extends SpatialApp {

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

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

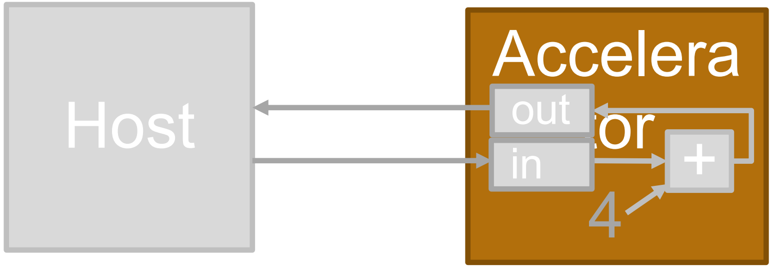


Acceleratable Code

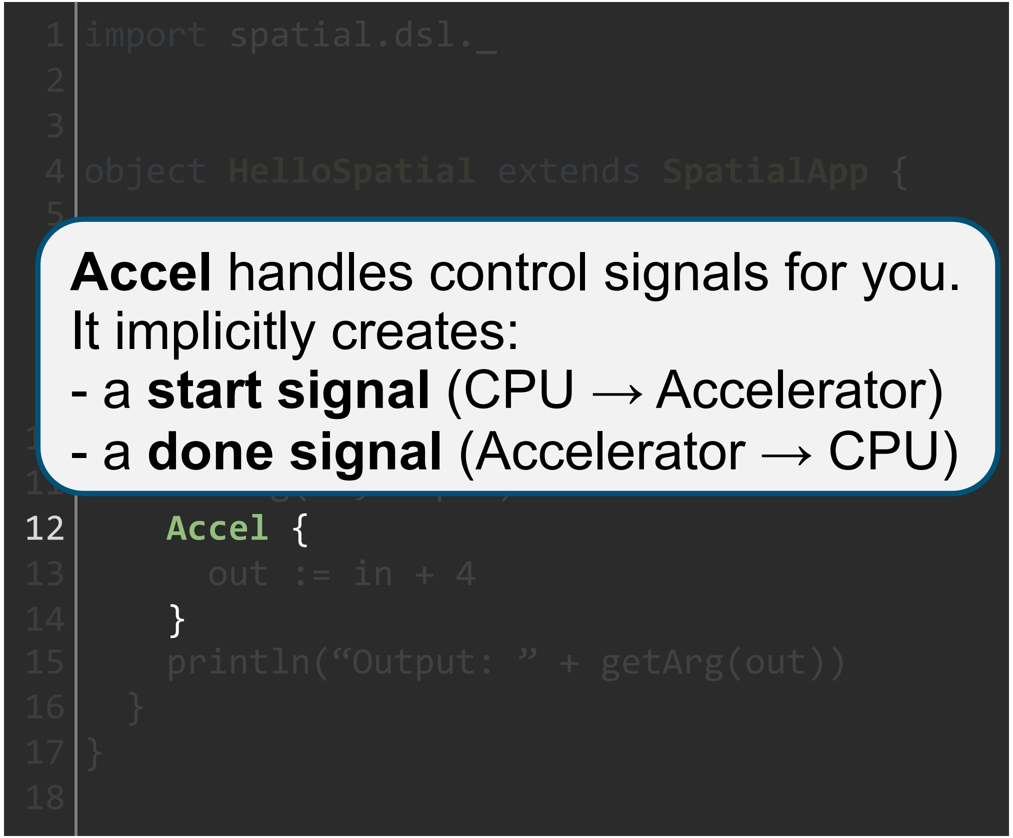


```
import spatial.dsl._


object HelloSpatial extends SpatialApp {

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

```

The types of operations done in this scope are limited to **acceleratable (synthesizable)** Spatial

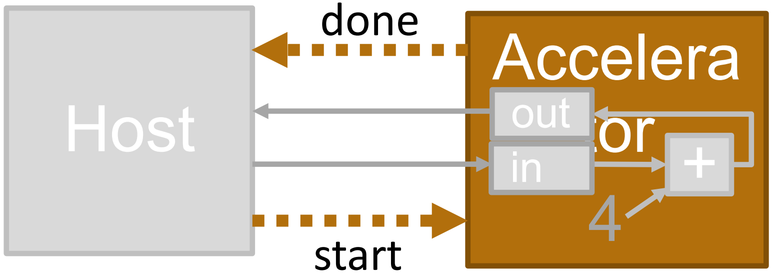


Accel Block

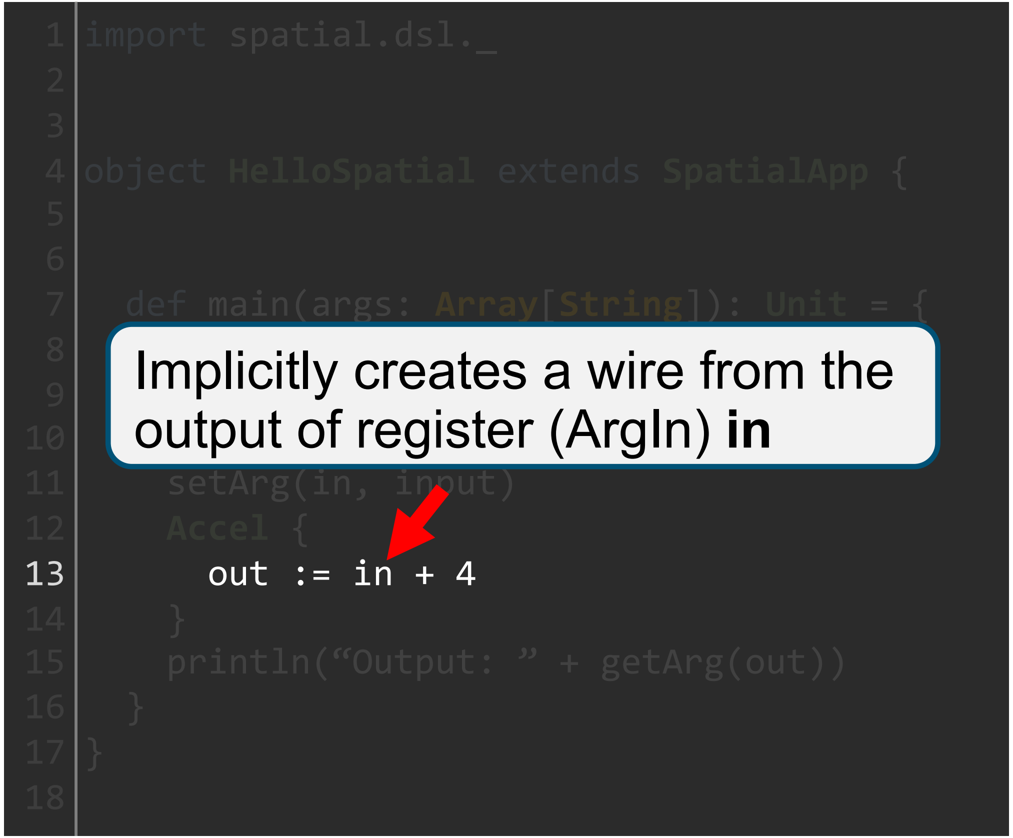


```
import spatial.dsl._


object HelloSpatial extends SpatialApp {

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

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

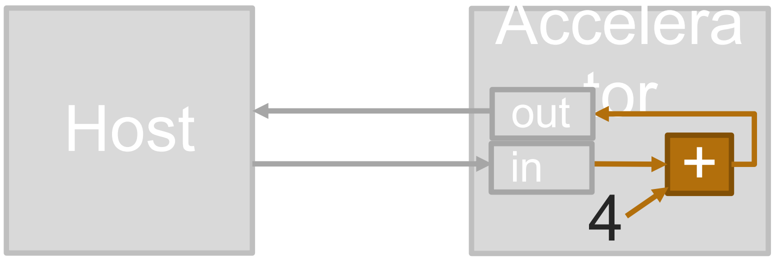


Implicit Register Reads

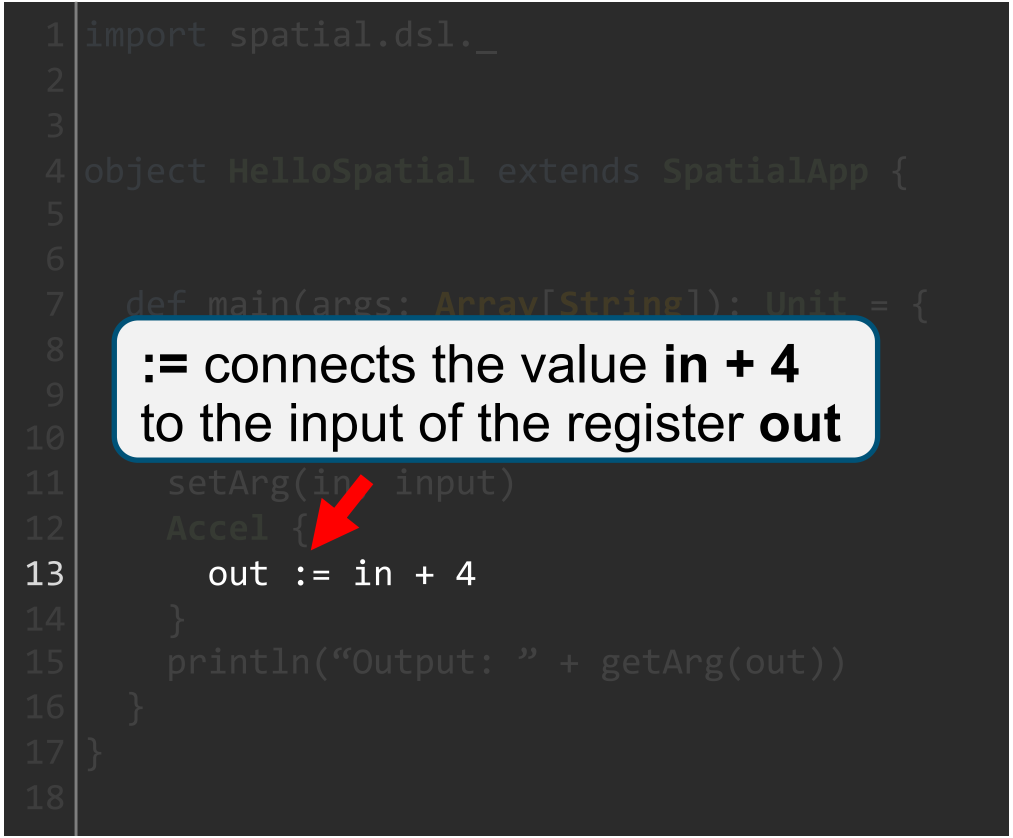


```
import spatial.dsl._


object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): Unit = {



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

```

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

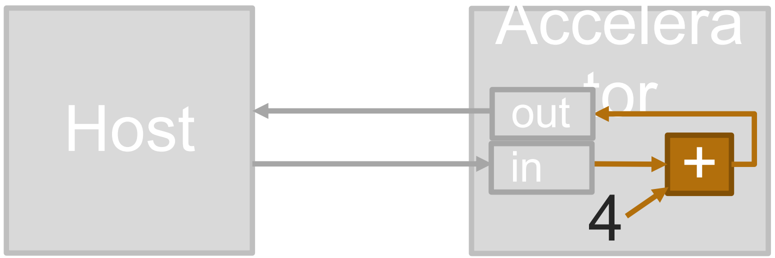


Implicit Register Reads

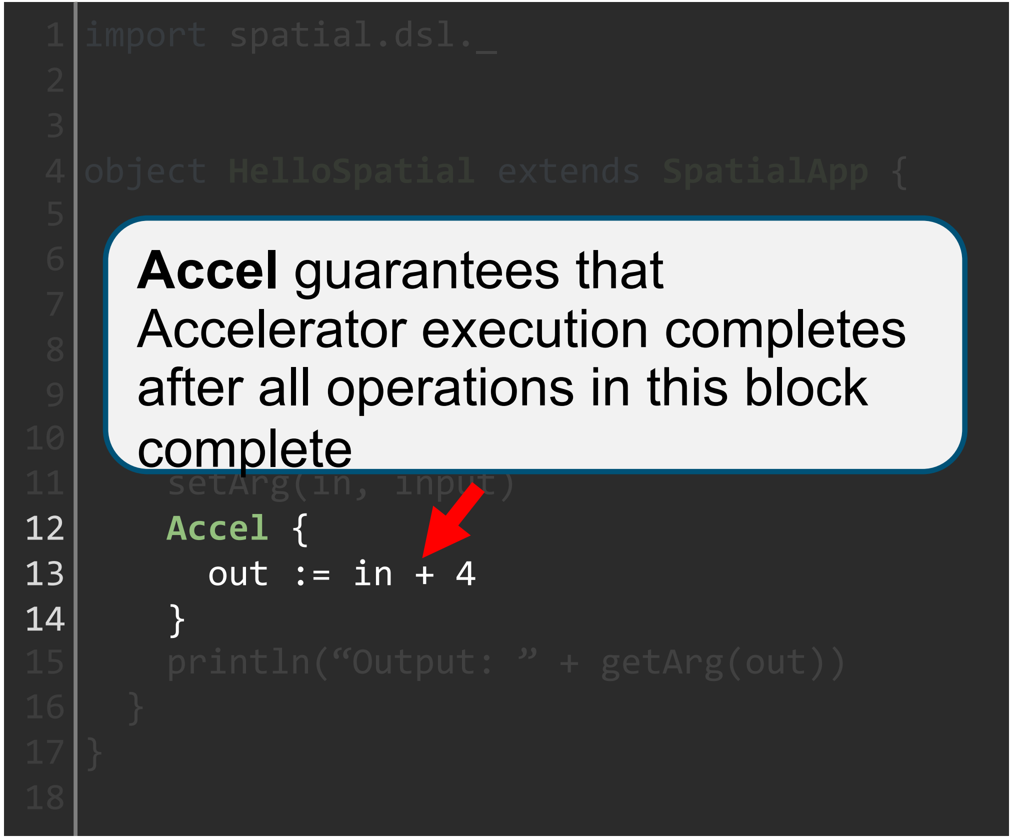


```
import spatial.dsl._


object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): Unit = {



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

:= connects the value **in + 4** to the input of the register **out**

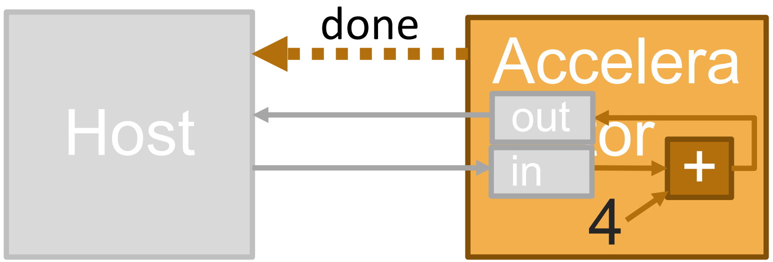


Accel Block Scheduling

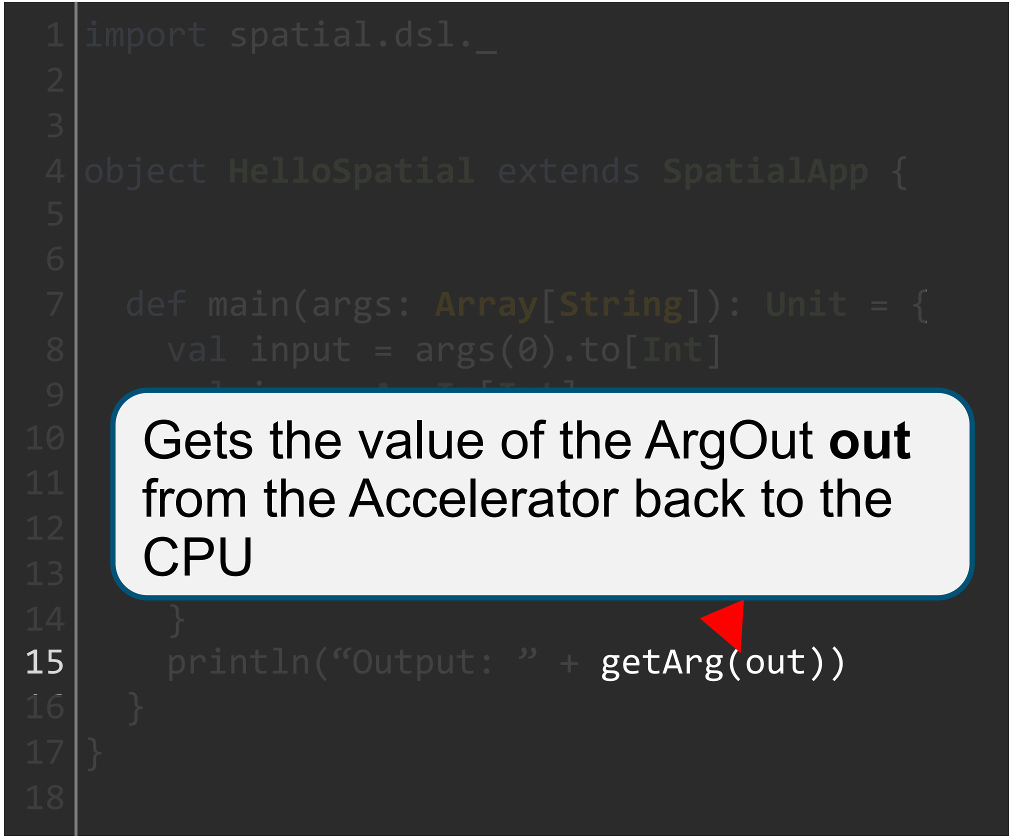


Accel guarantees that Accelerator execution completes after all operations in this block complete

```
import spatial.dsl._


object HelloSpatial extends SpatialApp {

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



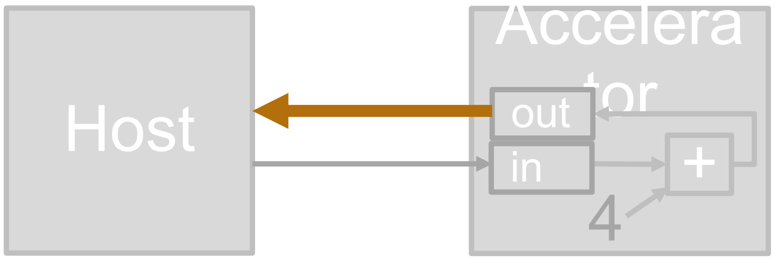
Scalar Transfers (Accelerator → CPU)



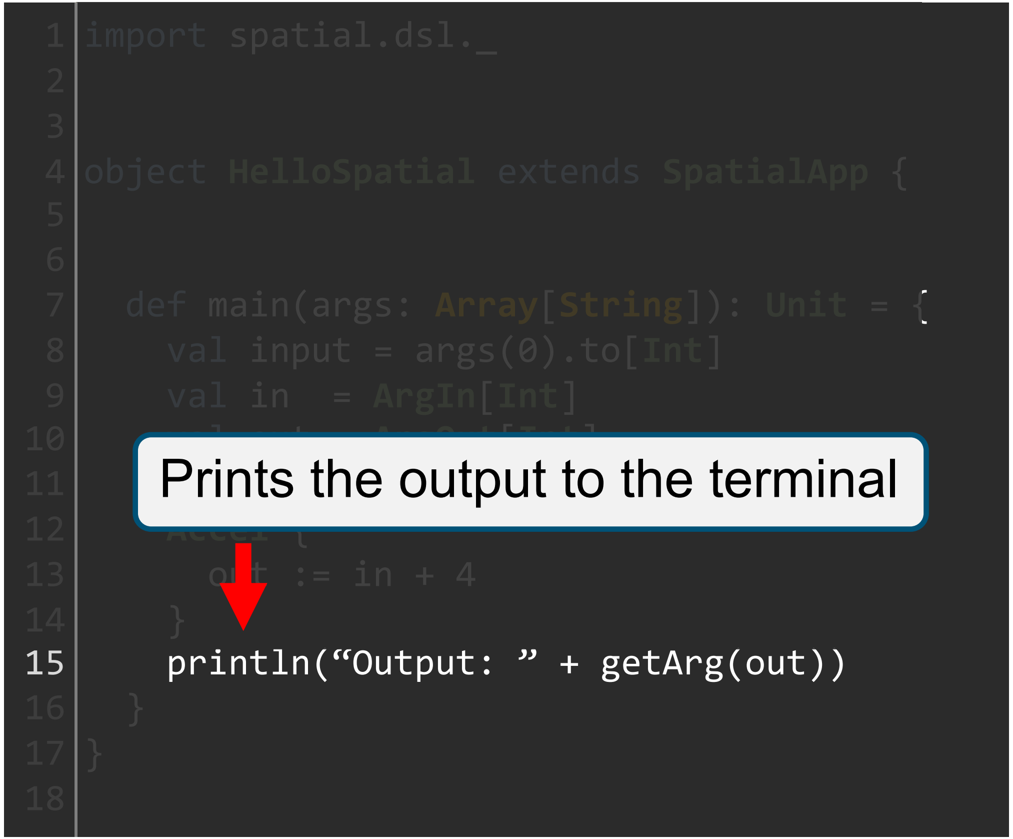
```
import spatial.dsl._


object HelloSpatial extends SpatialApp {


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

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

Gets the value of the ArgOut **out** from the Accelerator back to the CPU



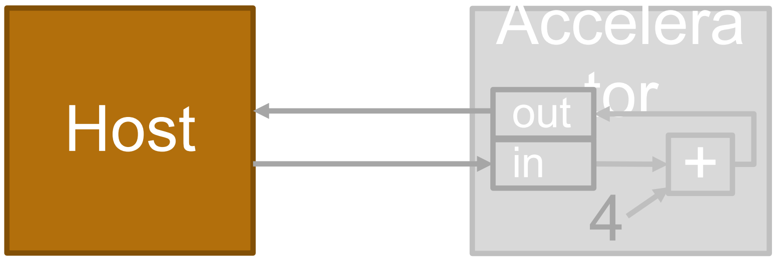
Printing in Spatial**



```
import spatial.dsl._


object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): Unit = {
    val input = args(0).to[Int]
    val in  = ArgIn[Int]


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

```

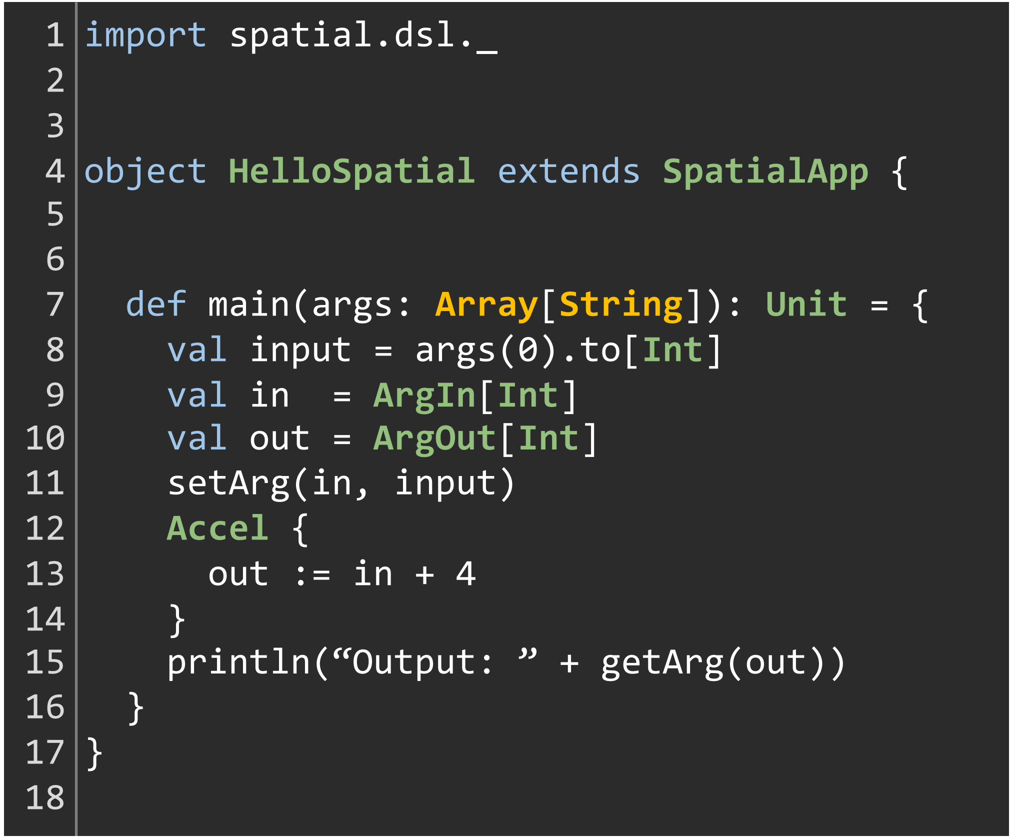
Prints the output to the terminal

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



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

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

Hello Spatial!



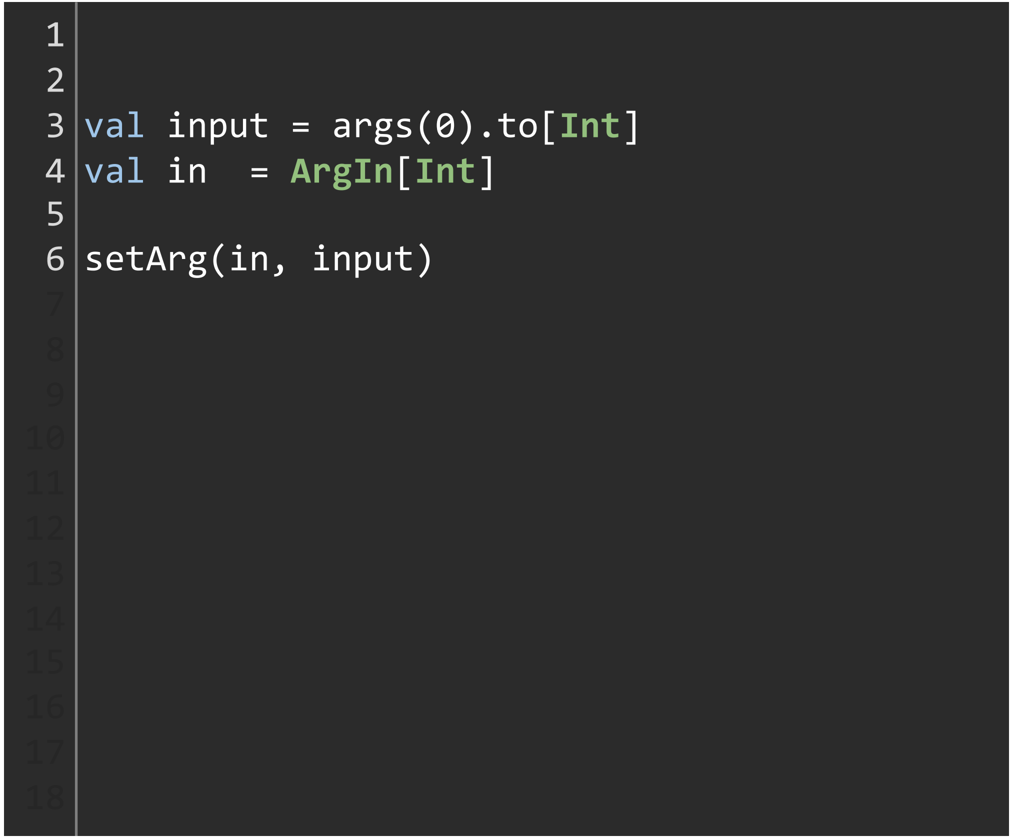
```
import spatial.dsl._


object HelloSpatial extends SpatialApp {


  def main(args: Array[String]): 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))
  }
}

```

SIMPLE TYPES

Custom Types in Spatial

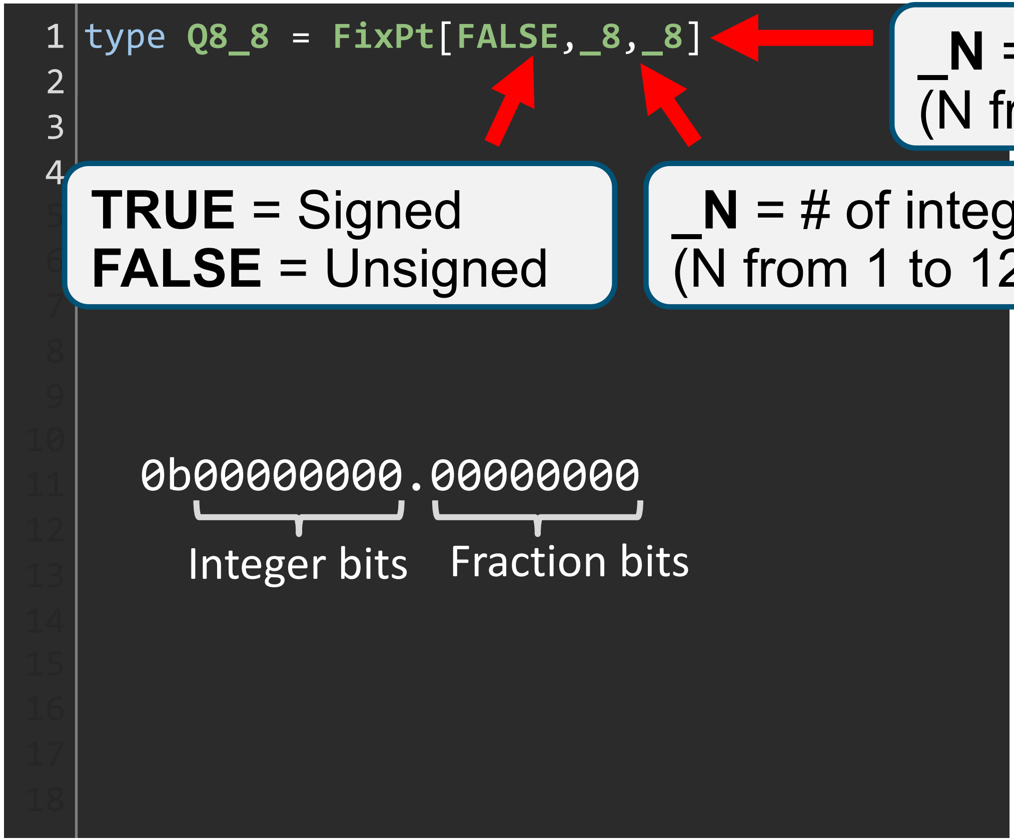
- Now what if we want an ArgIn value that isn't an Int?
- Other options:
  - Custom fixed point types
  - Custom floating point types
  - Structs
  - Vectors

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]
```

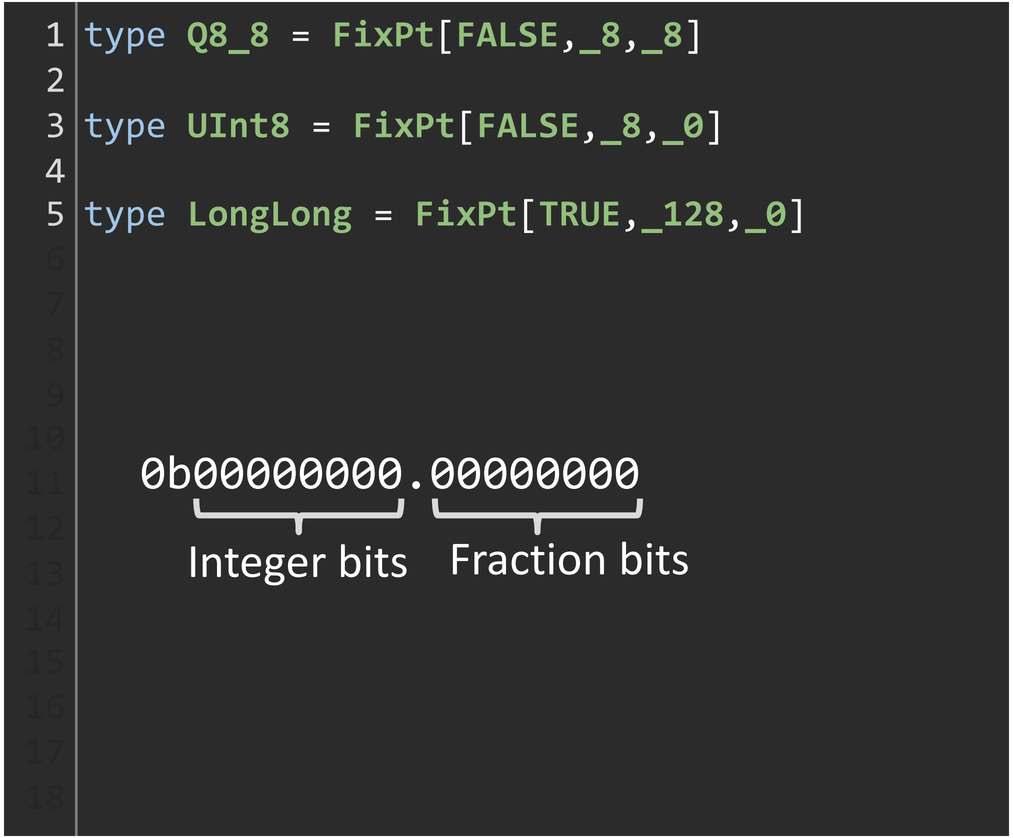
```
0b00000000.00000000
```

TRUE = Signed
FALSE = Unsigned

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

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

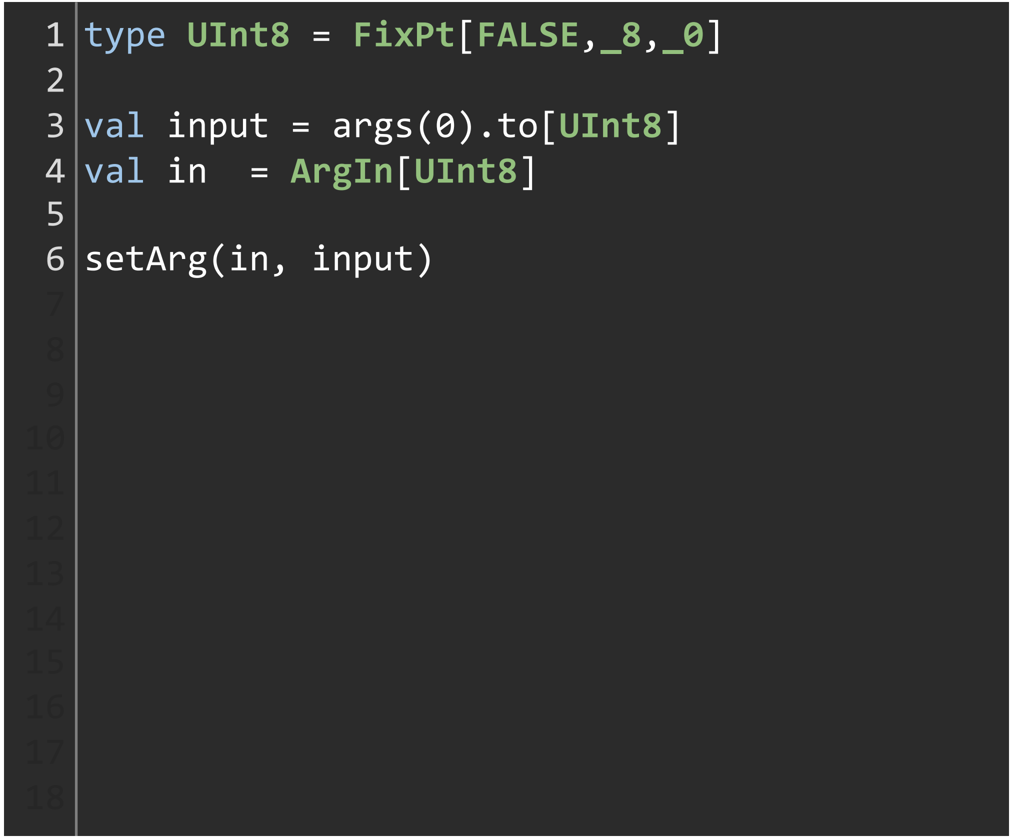
Custom Fixed Point Examples



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

type UInt8 = FixPt[FALSE,_8,_0]

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

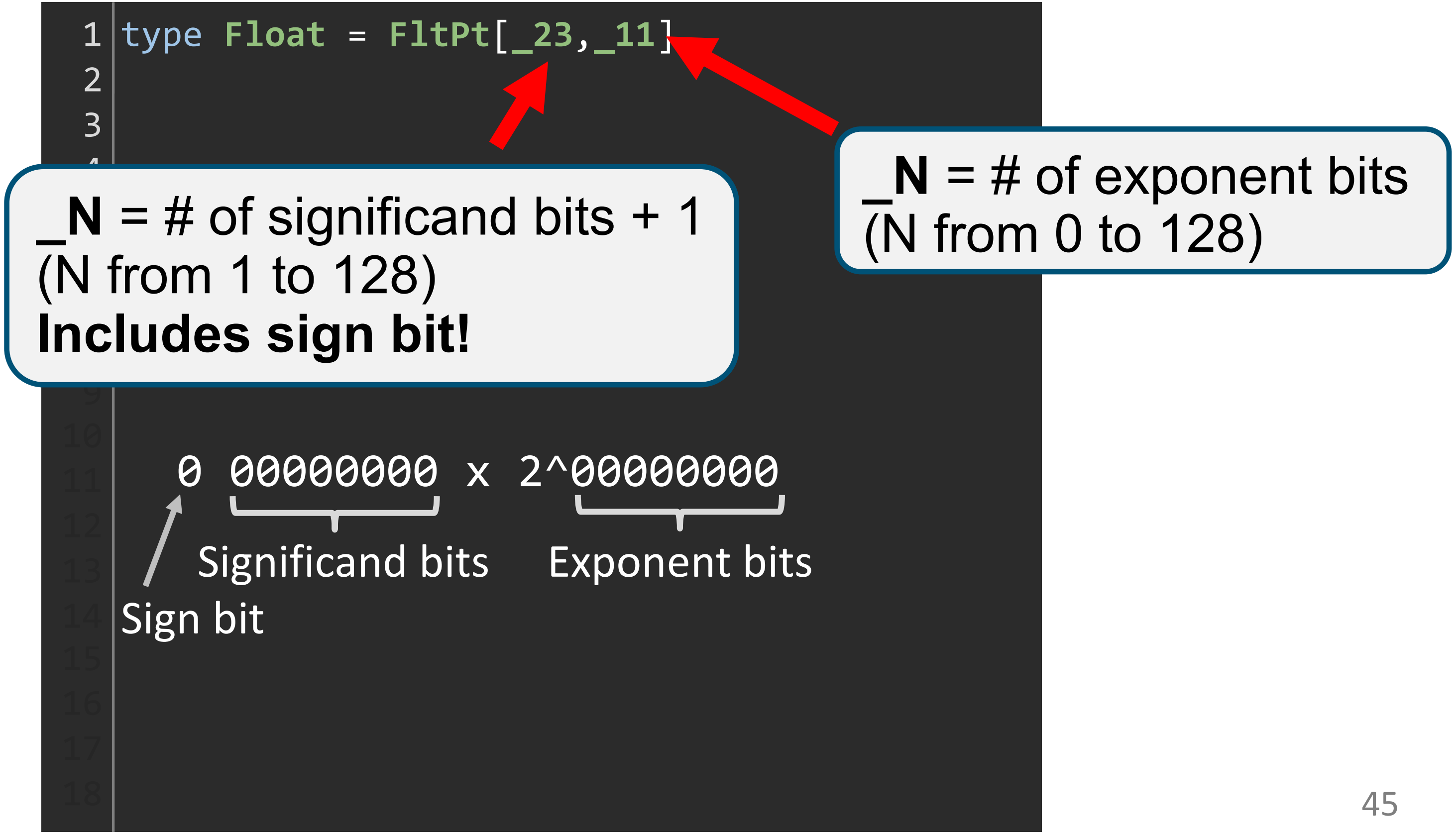
$$\texttt{0b}\underbrace{\texttt{00000000}}_{\text{Integer bits}}\texttt{.}\underbrace{\texttt{00000000}}_{\text{Fraction bits}}$$

Custom Fixed Point Types



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

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

setArg(in, input)
```

Custom Floating Point Types



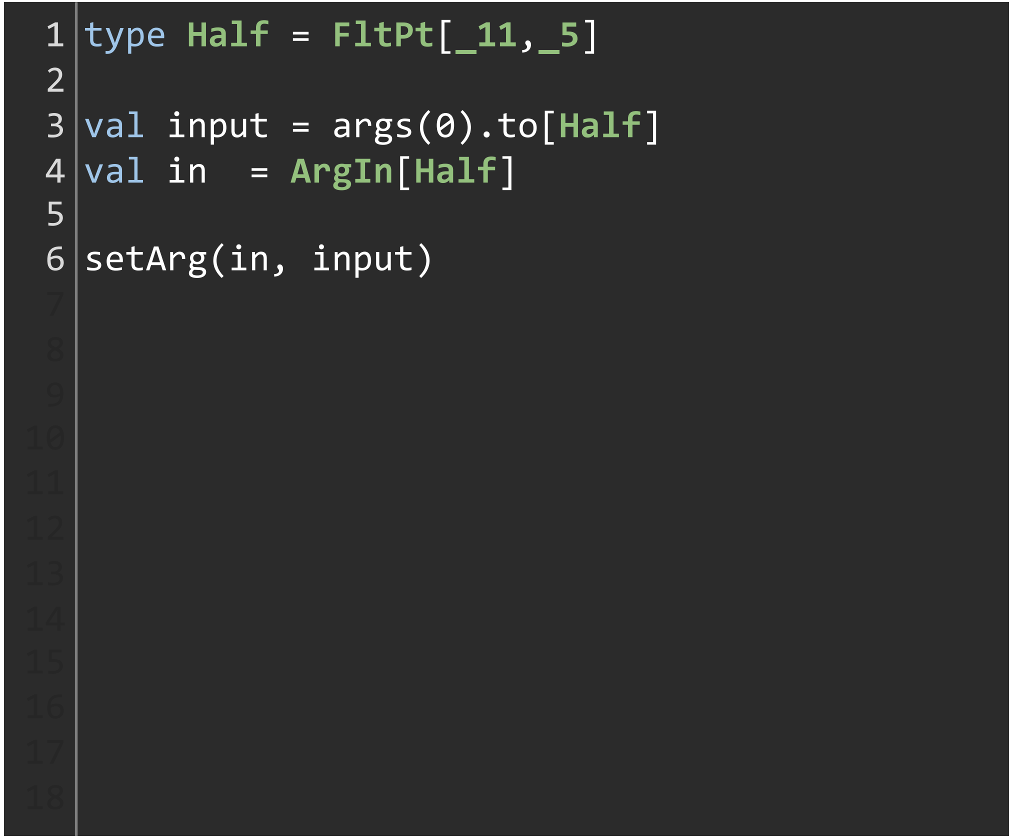
```
type Float = FltPt[_23,_11]
```

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

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

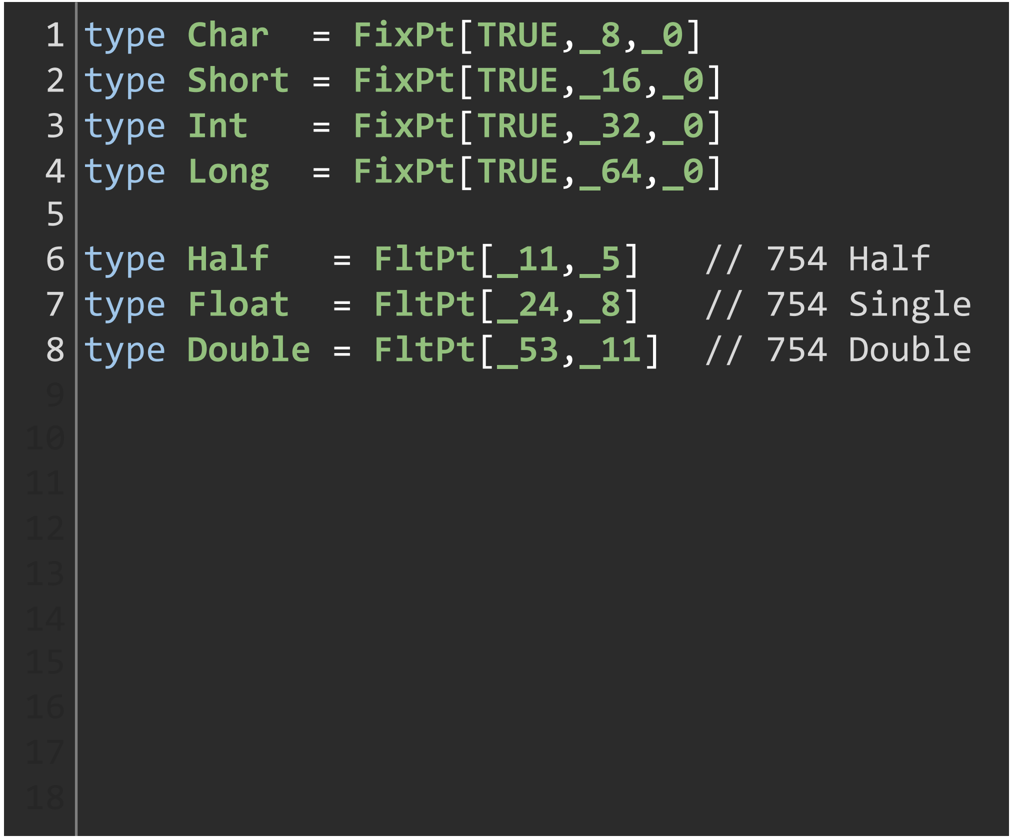
```
0 00000000 x 2^00000000
```

Custom Floating Point Types



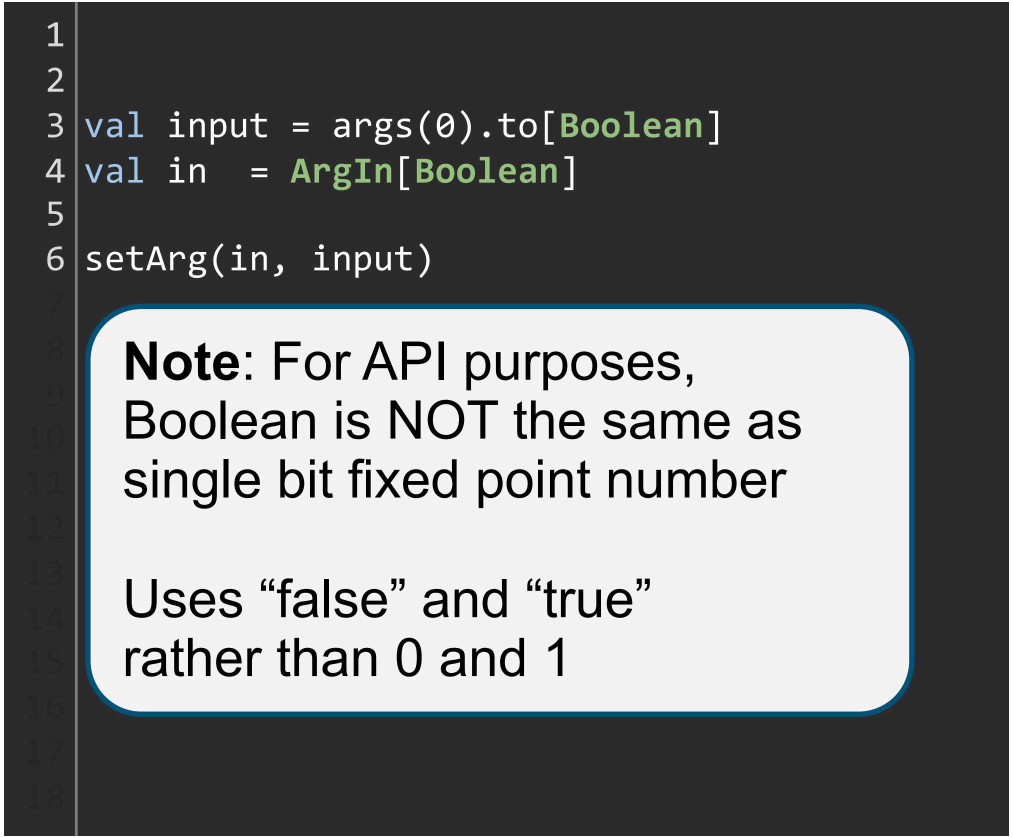
```
type Half = FltPt[_11,_5]

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

setArg(in, input)
```

Predefined Type Aliases



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

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

Note About Booleans

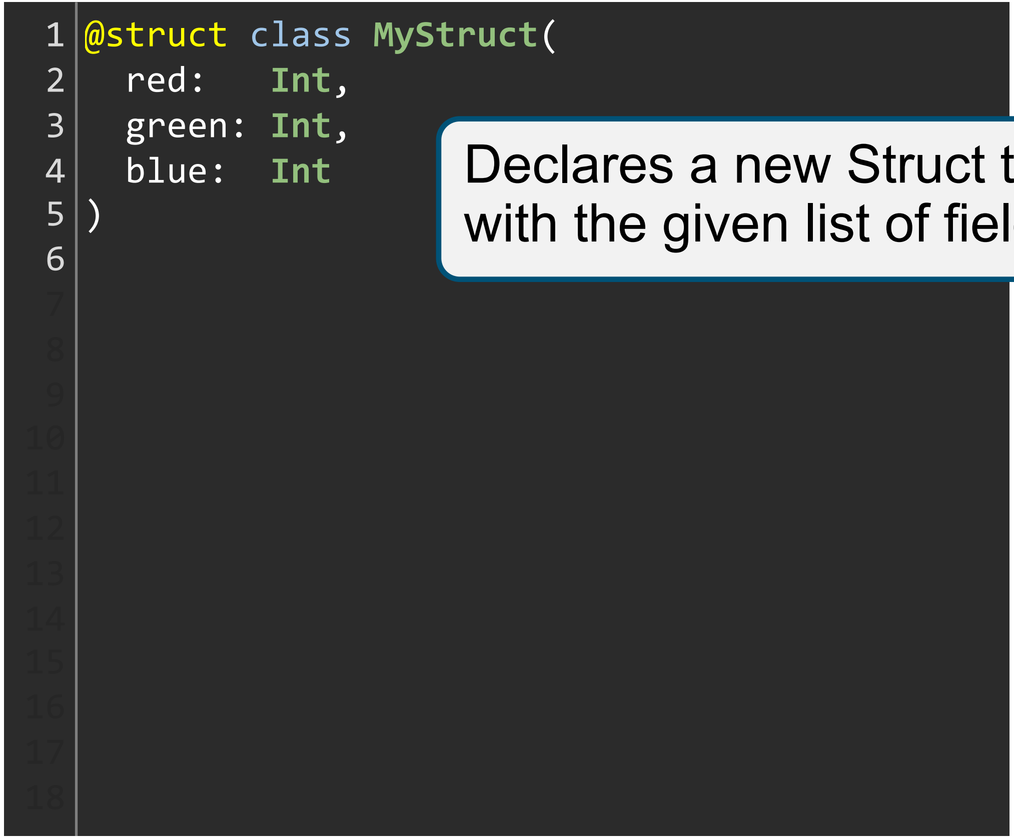


```
val input = args(0).to[Boolean]
val in  = ArgIn[Boolean]

setArg(in, input)
```

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

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

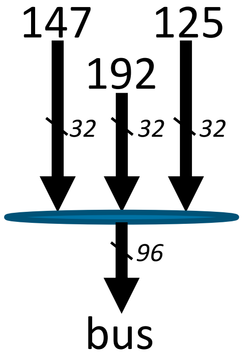
Custom Structs

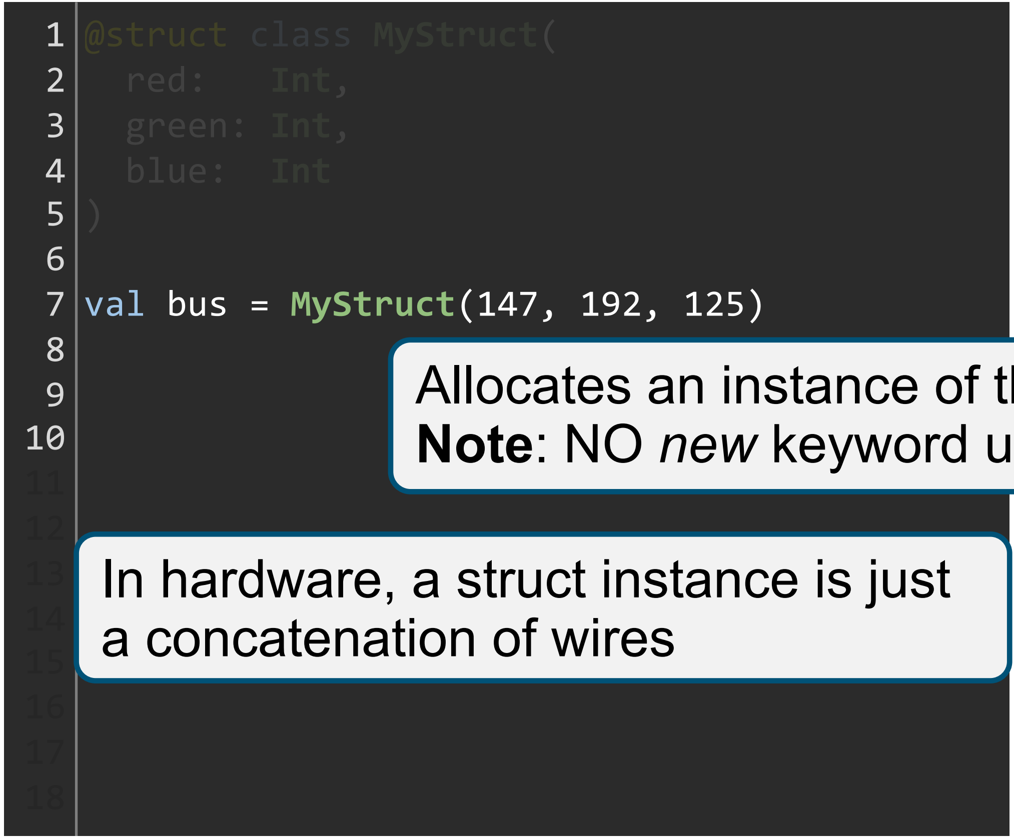


```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)
```

Declares a new Struct type with the given list of fields

Custom Structs



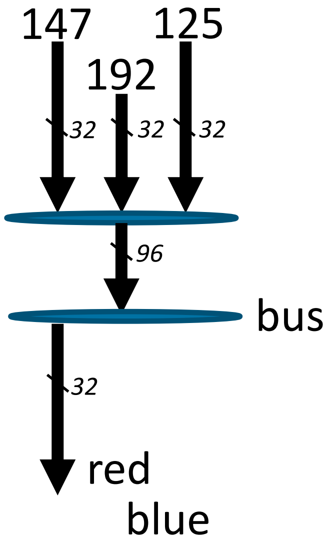


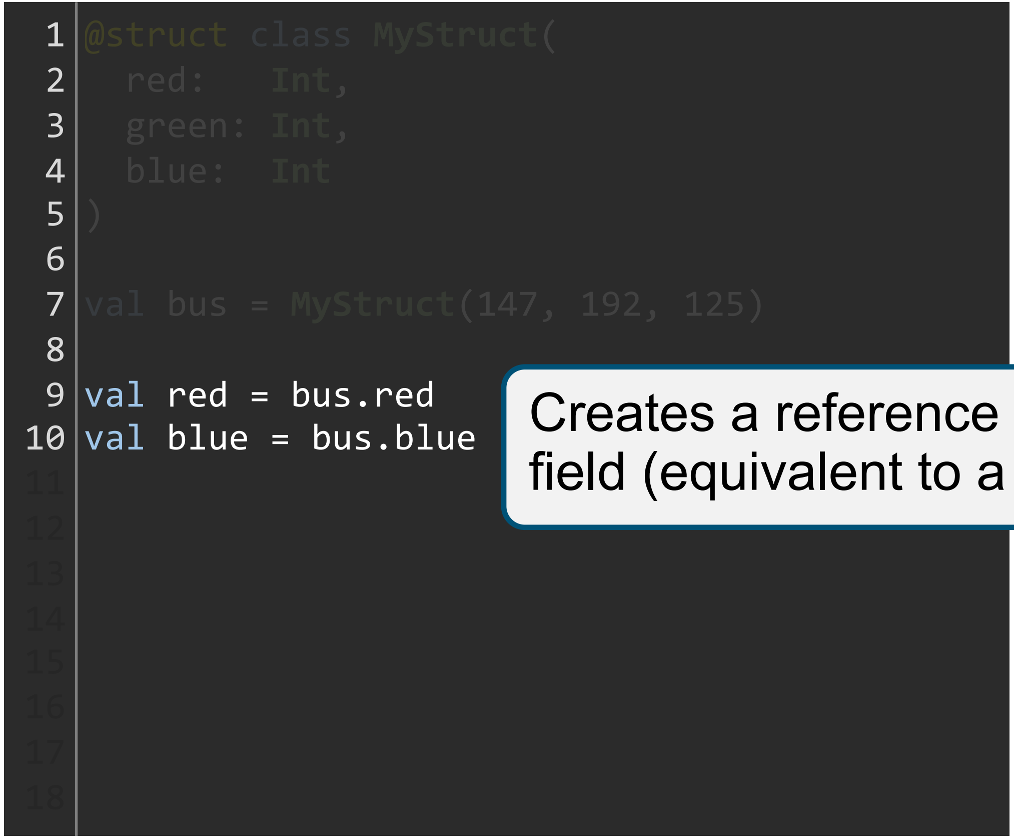
```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

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

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

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

Custom Structs

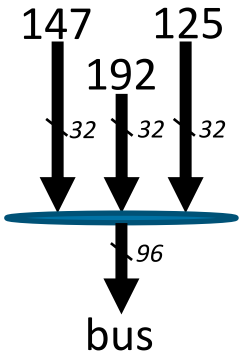


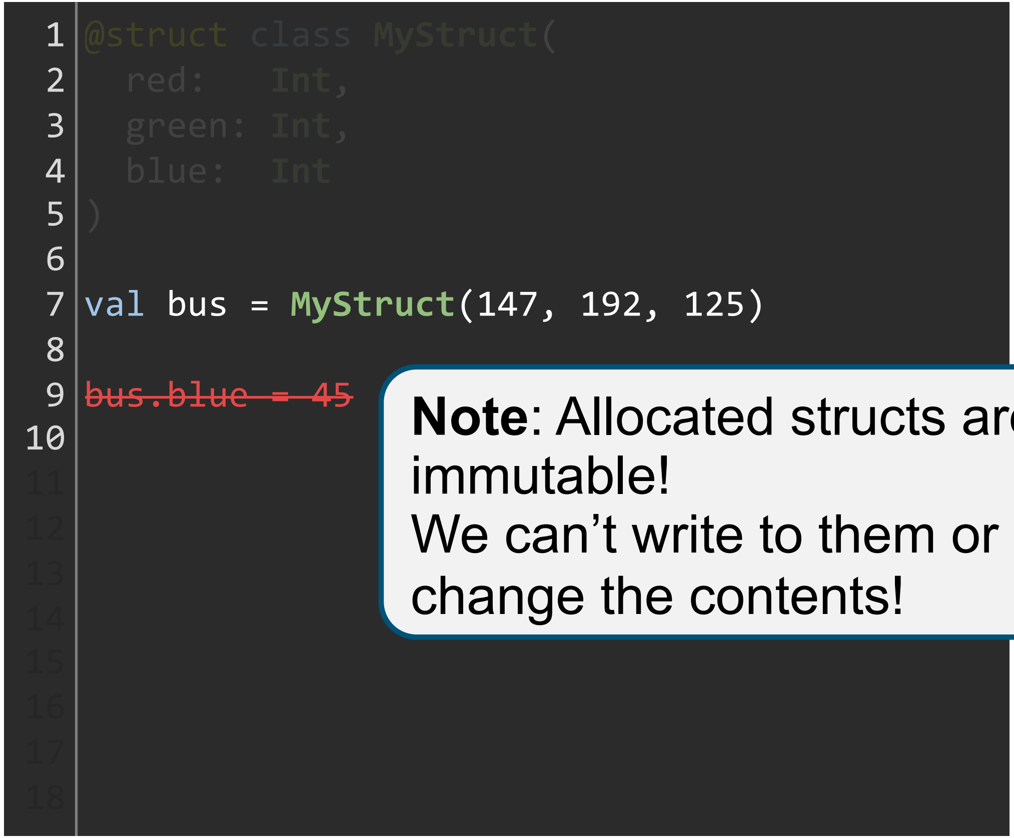


```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

val bus = MyStruct(147, 192, 125)

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

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

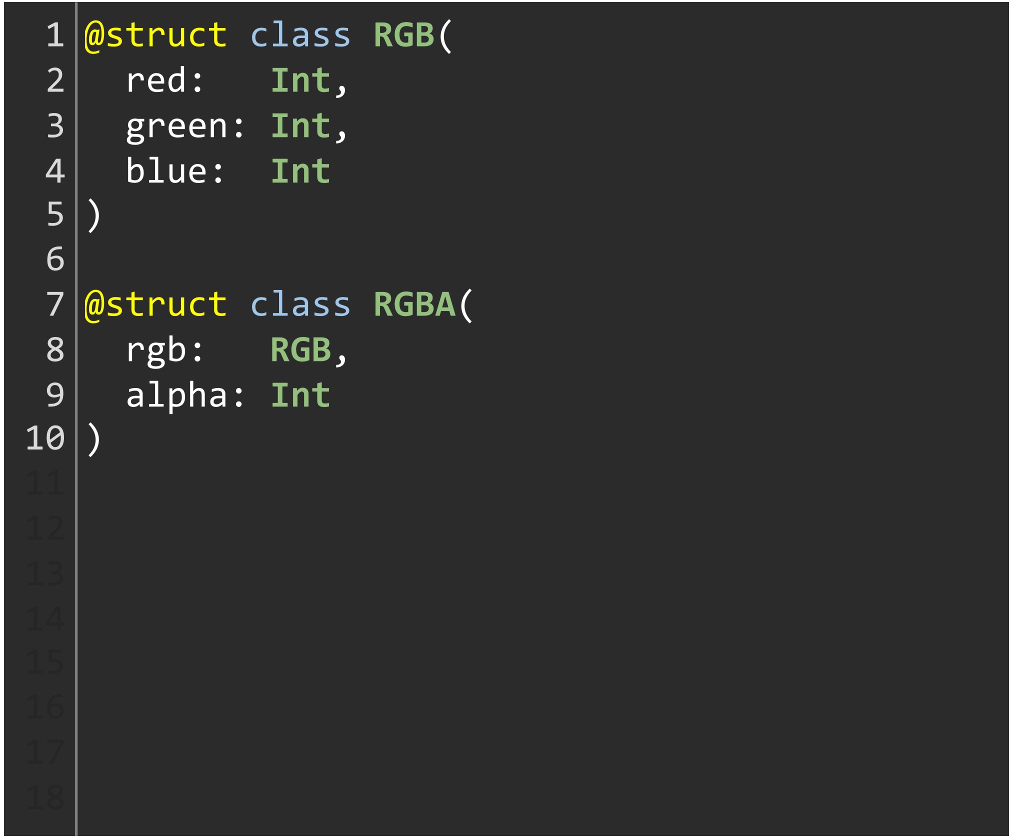
Custom Structs




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

val bus = MyStruct(147, 192, 125)

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

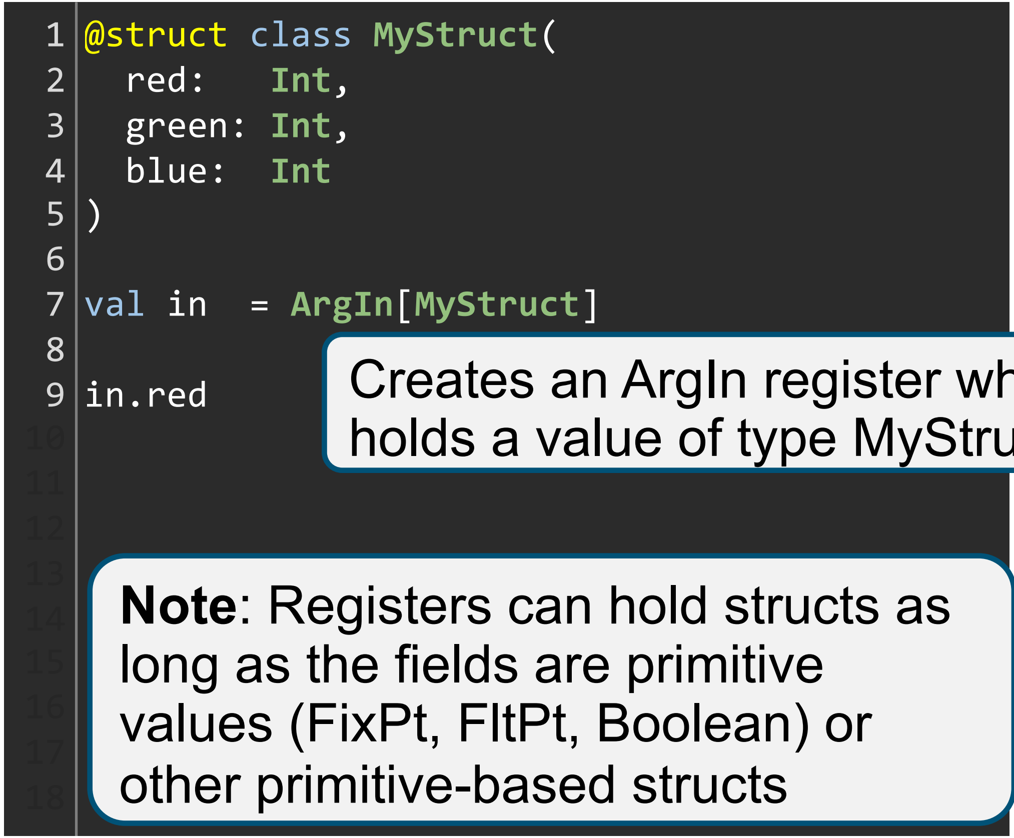
Note: Allocated structs are immutable!
We can’t write to them or change the contents!

Nesting Structs



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

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

Registers of Custom Types

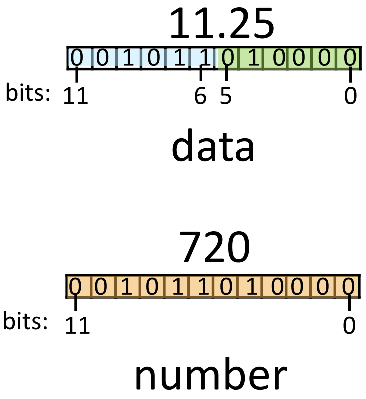


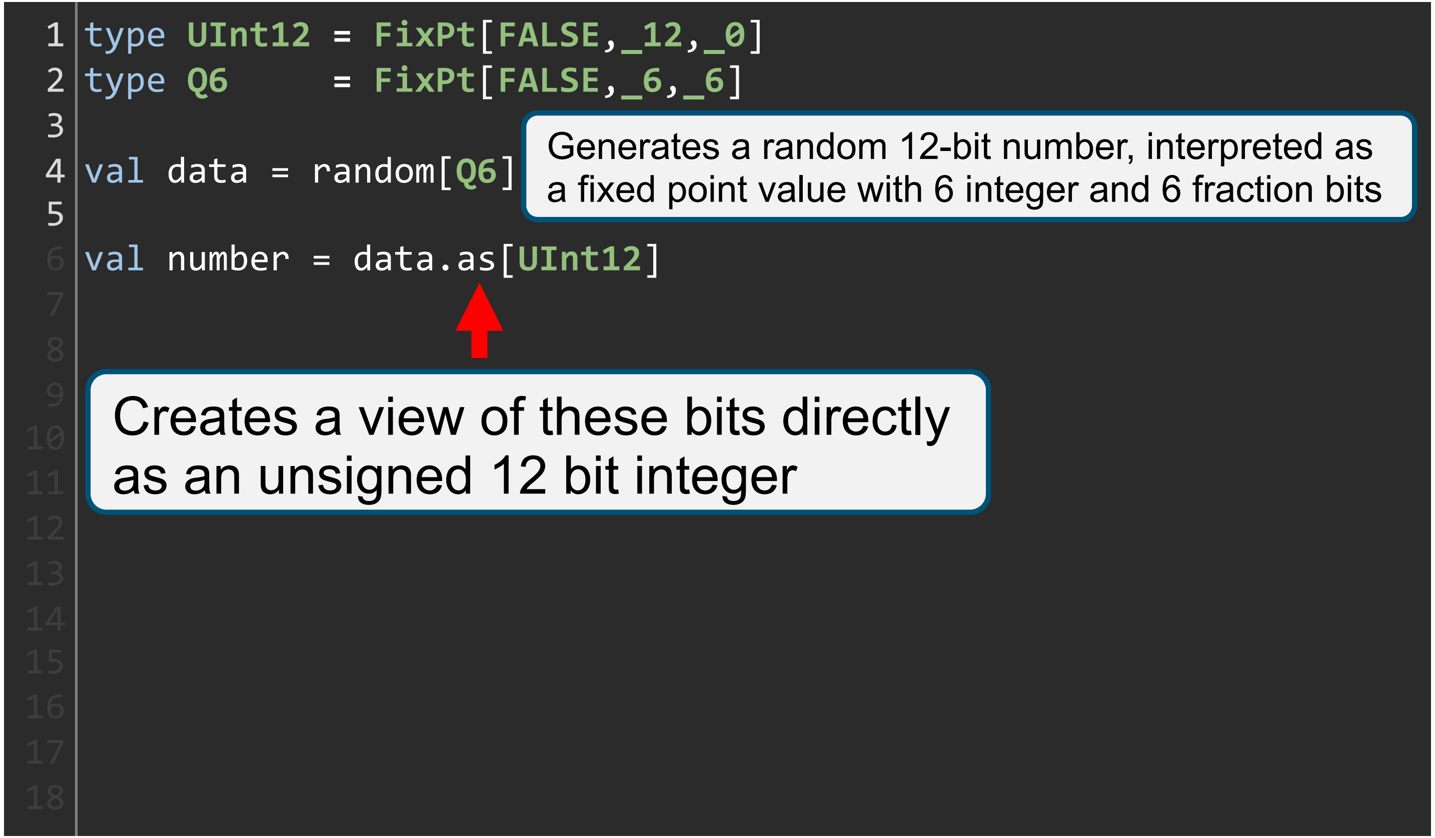
```
@struct class MyStruct(
  red:   Int,
  green: Int,
  blue:  Int
)

val in  = ArgIn[MyStruct]

in.red
```

Creates an ArgIn register which holds a value of type MyStruct

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

Bit Casting



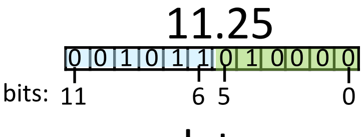


```
type UInt12 = FixPt[FALSE,_12,_0]
type Q6     = FixPt[FALSE,_6,_6]

val data = random[Q6]

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

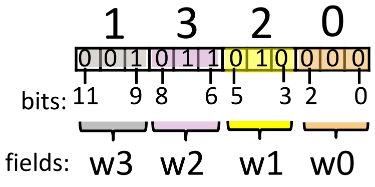
Generates a random 12-bit number, interpreted as a fixed point value with 6 integer and 6 fraction bits

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

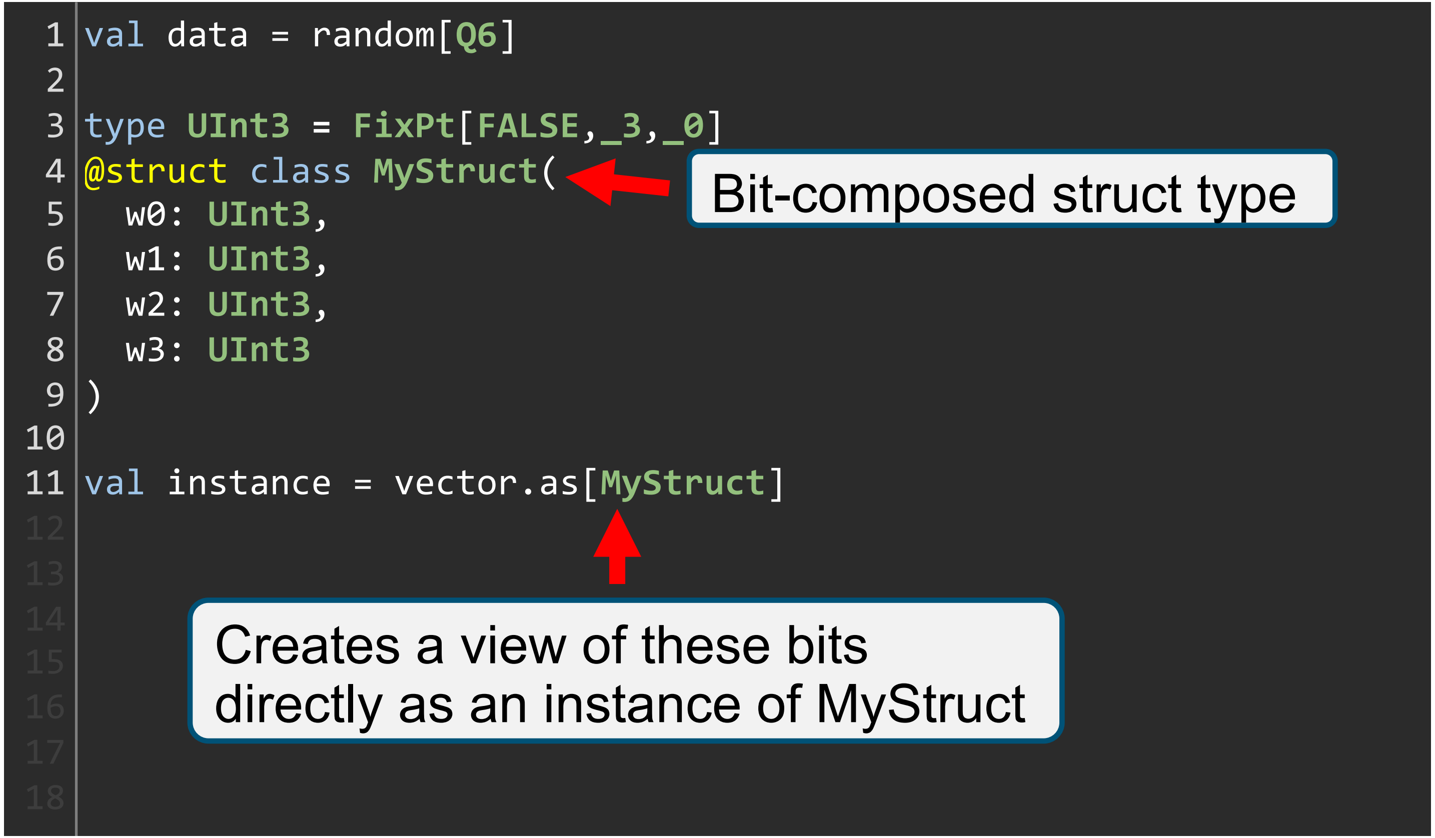
Bit Casting



data



instance

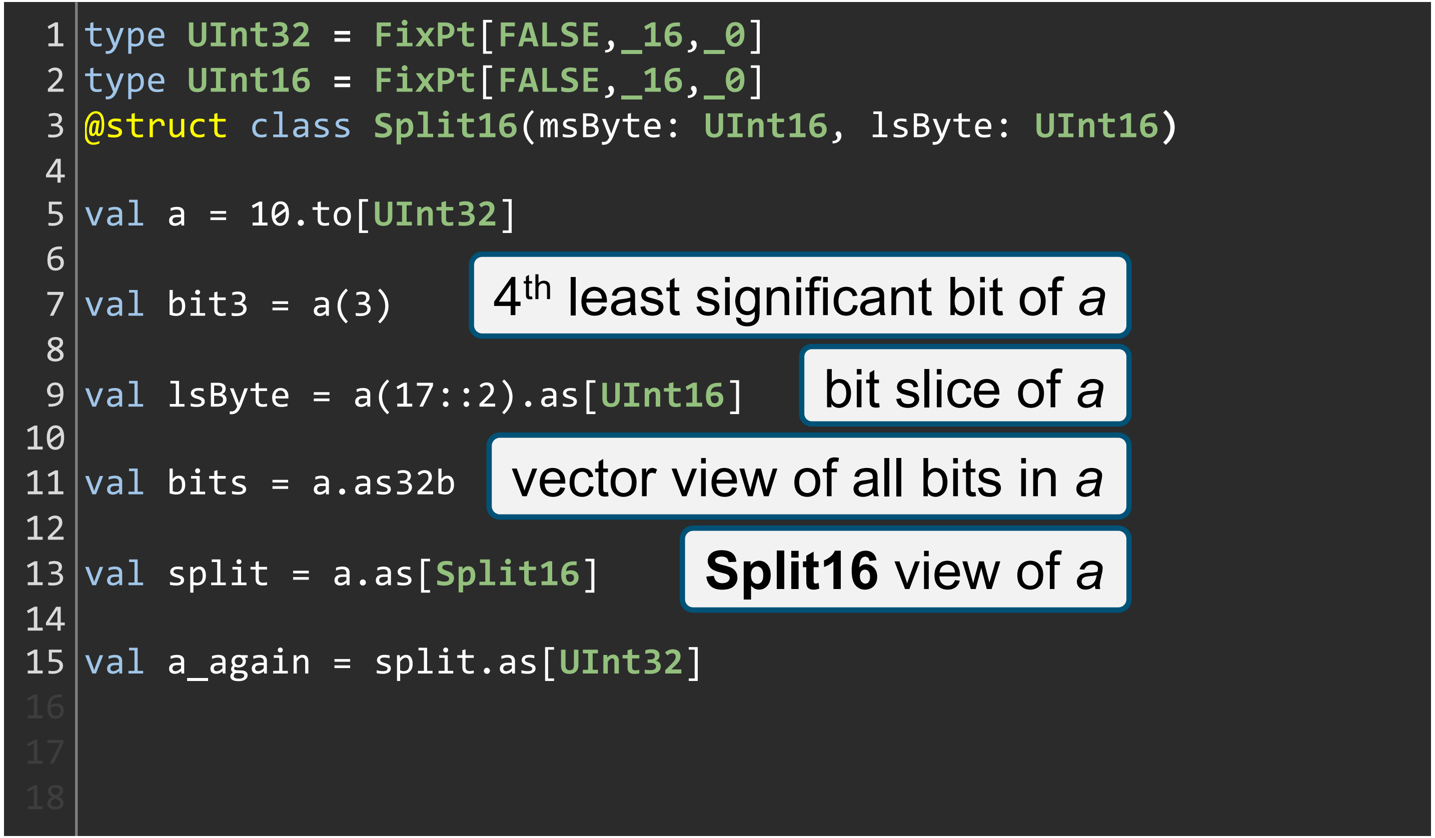


```
val data = random[Q6]

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

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).as[UInt16]

val bits = a.as32b

val split = a.as[Split16]

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

4th least significant bit of *a*

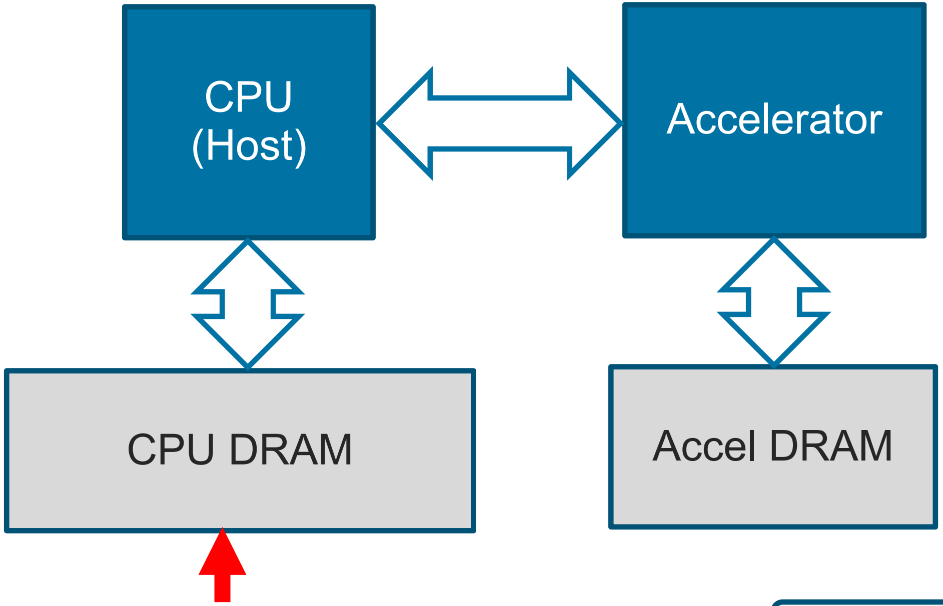
bit slice of *a*

vector view of all bits in *a*

Split16 view of *a*

OFF-CHIP MEMORIES

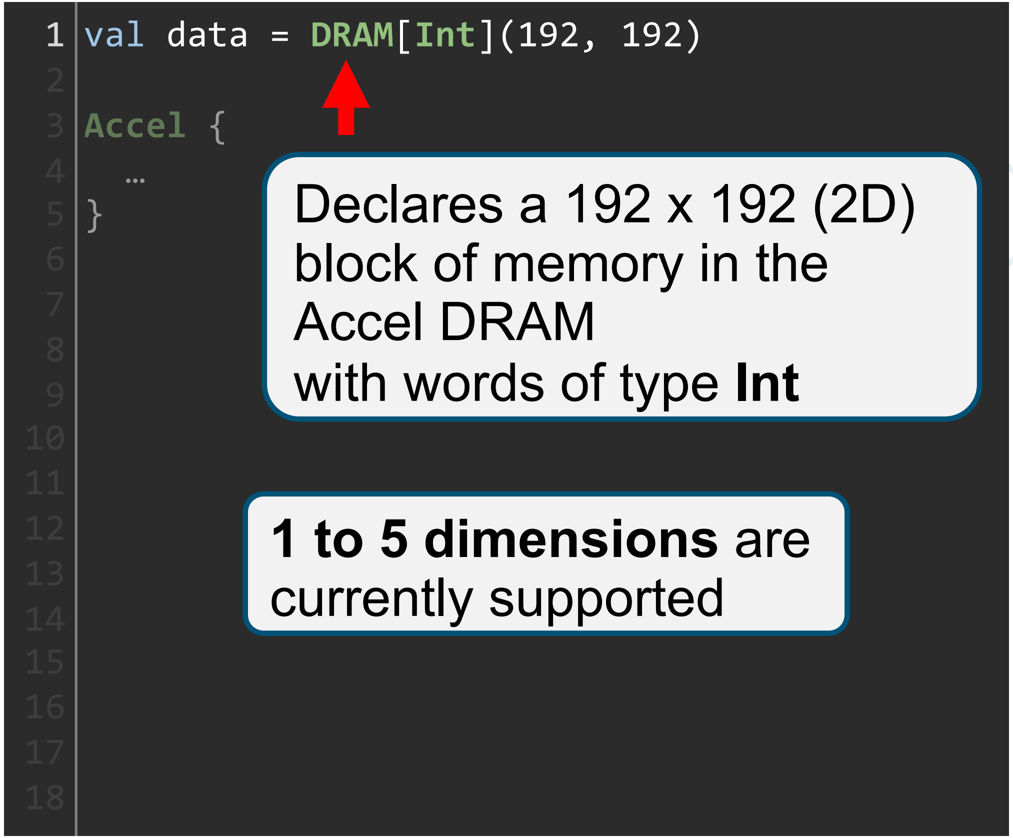
Basic Machine Model



Using this is easy (automatic!)

How to use this?

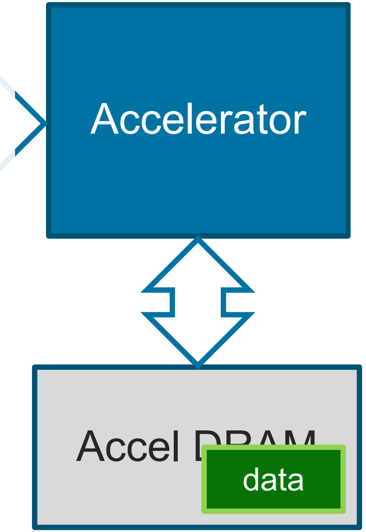
DRAM



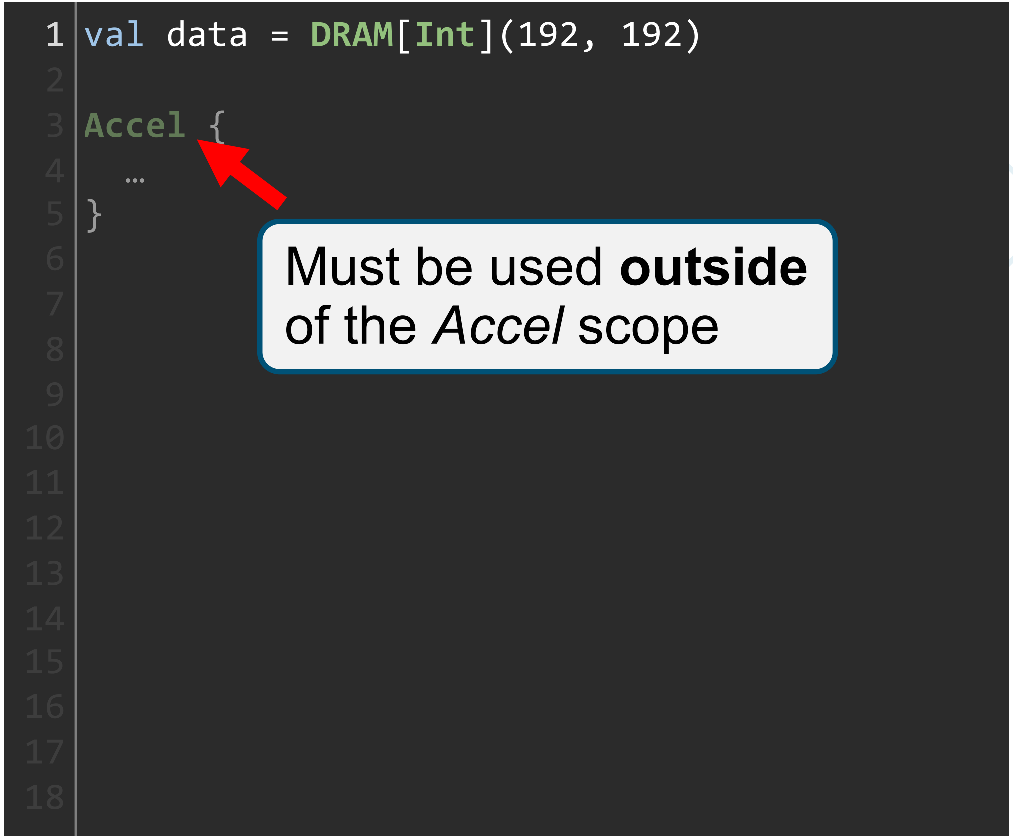
```
val data = DRAM[Int](192, 192)

Accel {
   …
}
```

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

1 to 5 dimensions are currently supported

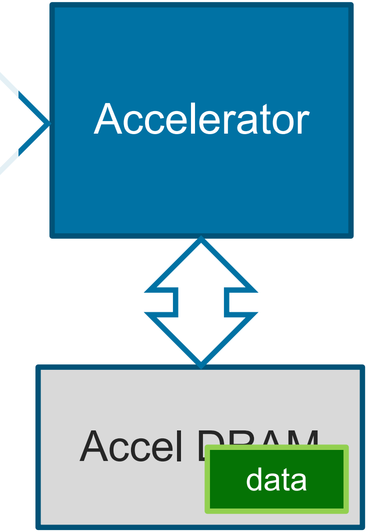


DRAM

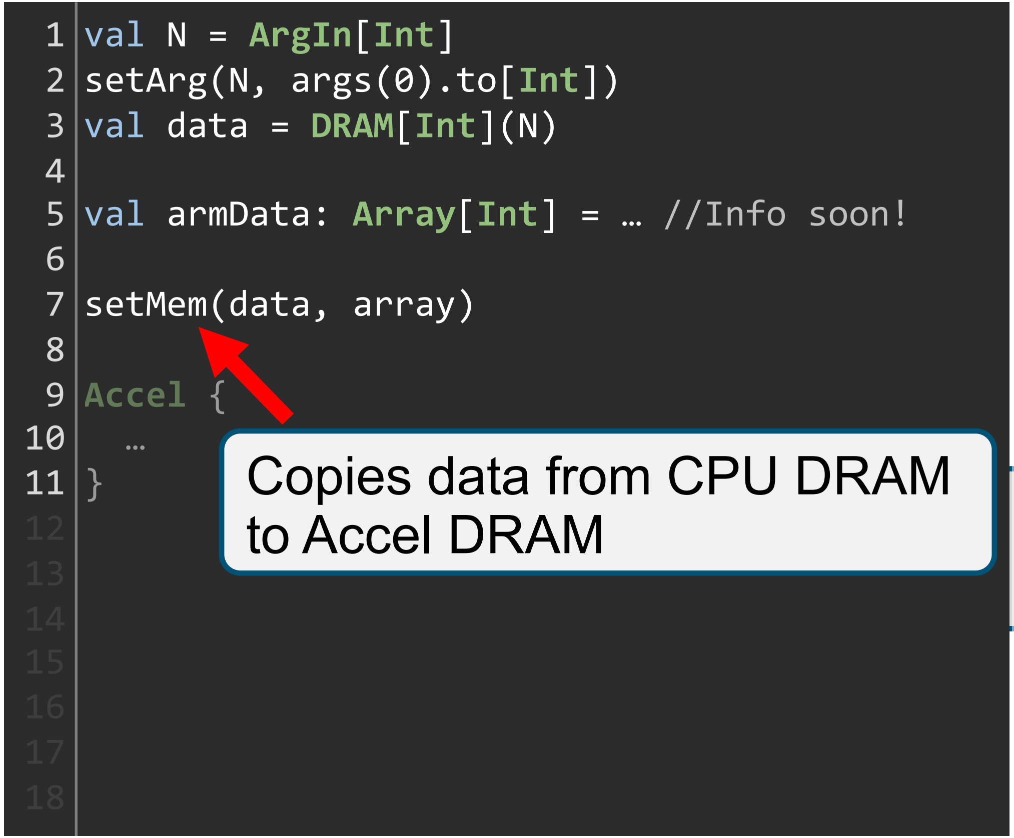


```
val data = DRAM[Int](192, 192)

Accel {
   …
}
```

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

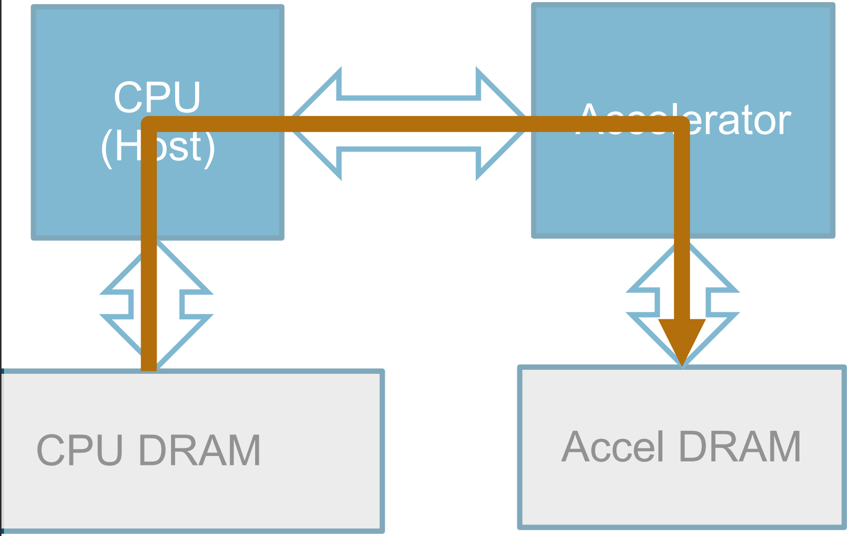


DRAM: Transfer from CPU to Accelerator

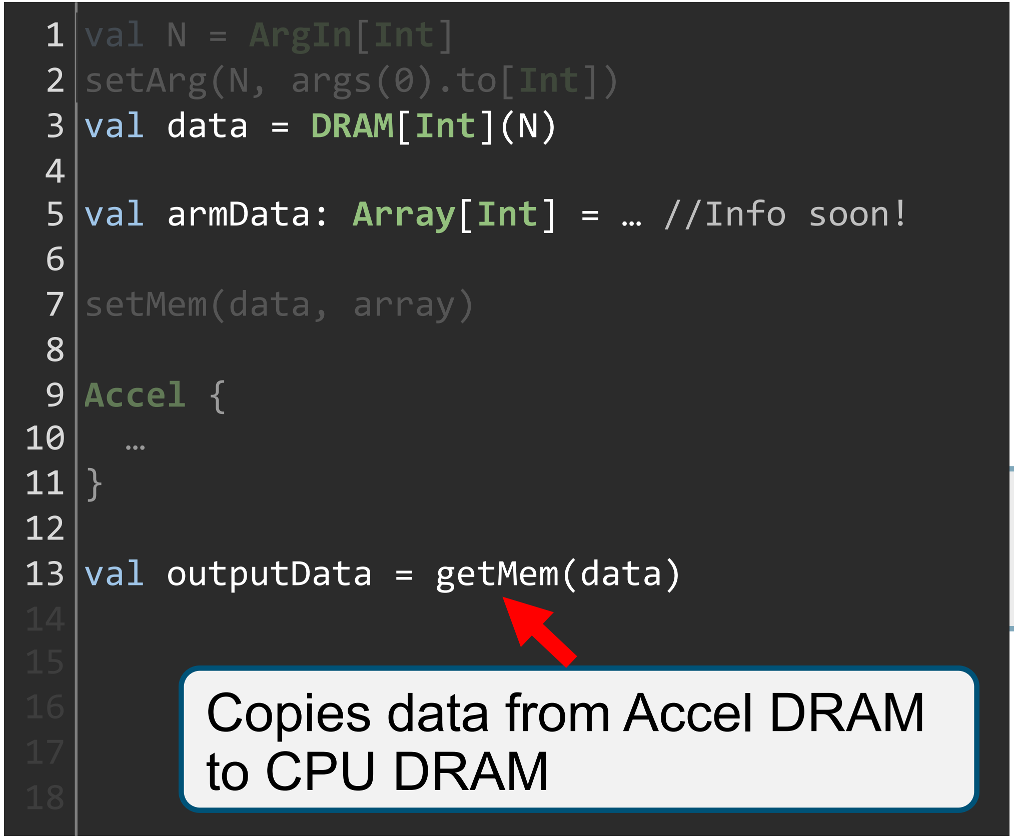


```
val N = ArgIn[Int]
setArg(N, args(0).to[Int])
val data = DRAM[Int](N)

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

setMem(data, array)

Accel {
   …
}
```

Copies data from CPU DRAM to Accel DRAM

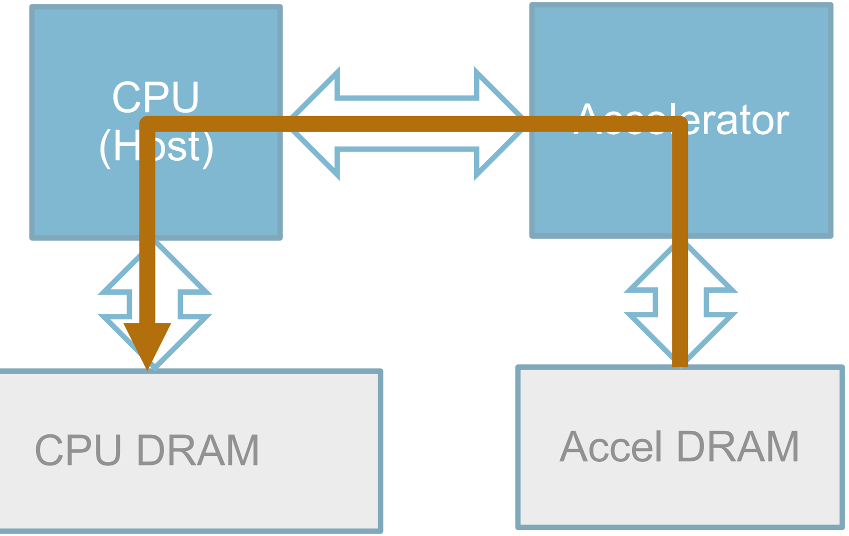


DRAM: Transfer from Accelerator to CPU



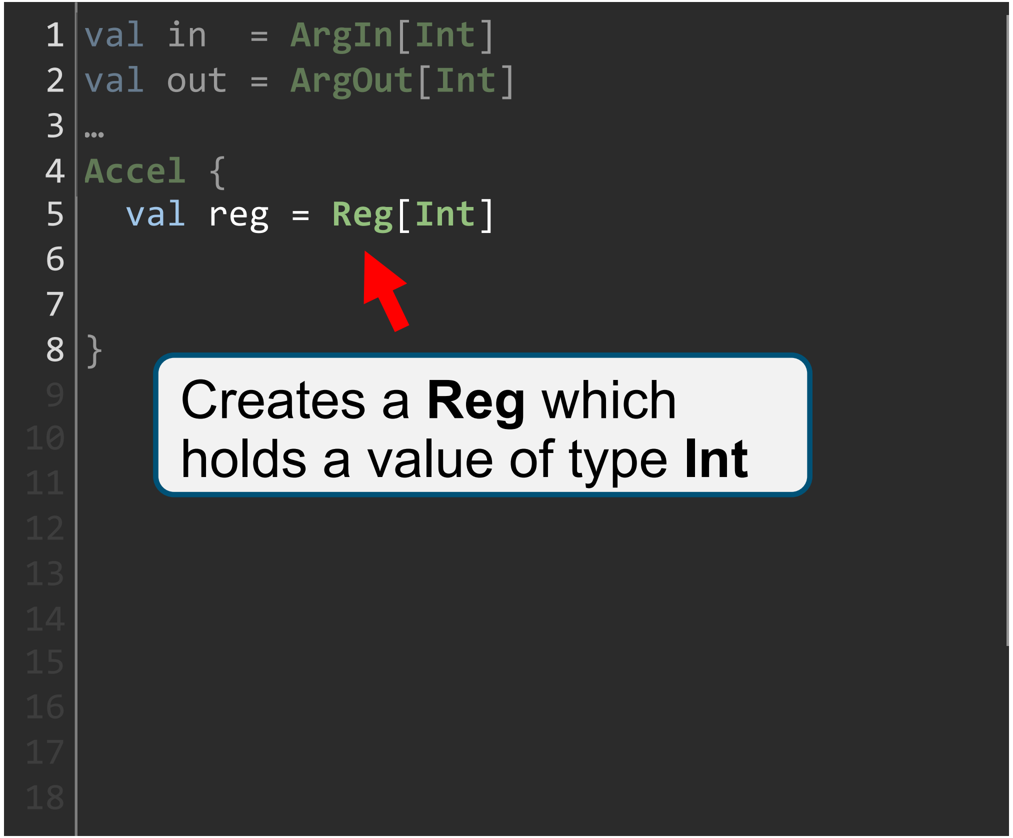
```
val N = ArgIn[Int]
setArg(N, args(0).to[Int])
val data = DRAM[Int](N)

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

setMem(data, array)

Accel {
  …
}

val outputData = getMem(data)
```

Copies data from Accel DRAM to CPU DRAM



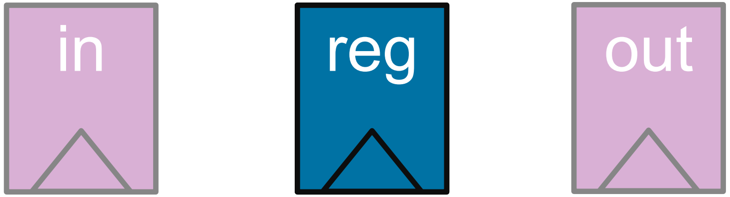
ON-CHIP MEMORIES

Reg

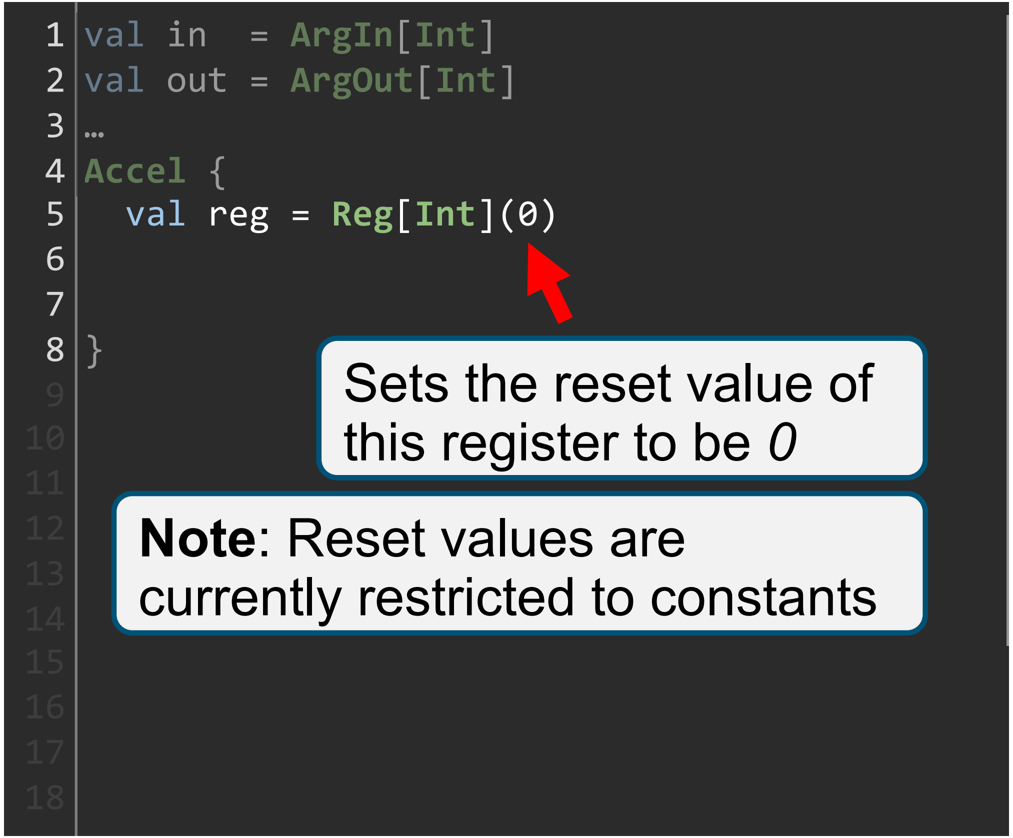


```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int]


}
```

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



Reg Reset Value

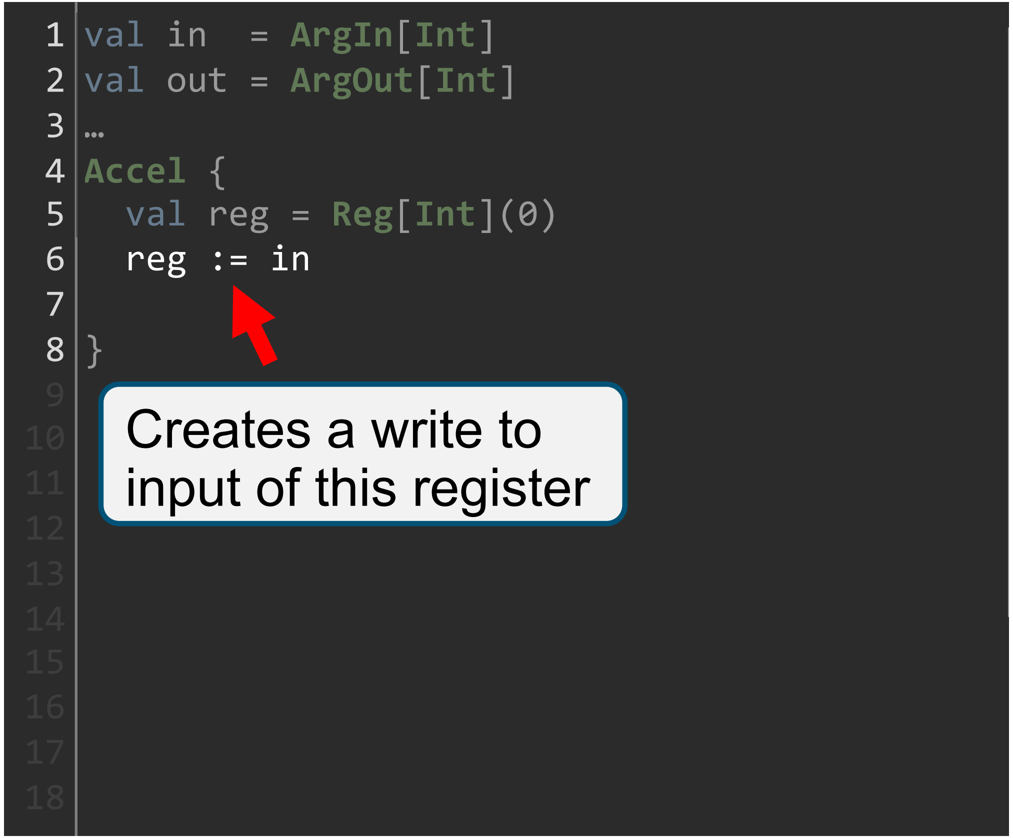


```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int](0)


}
```

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

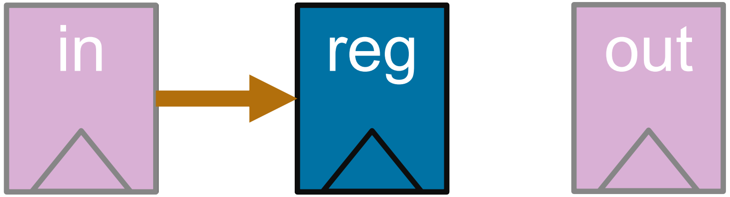
Note: Reset values are currently restricted to constants

Reg Writing

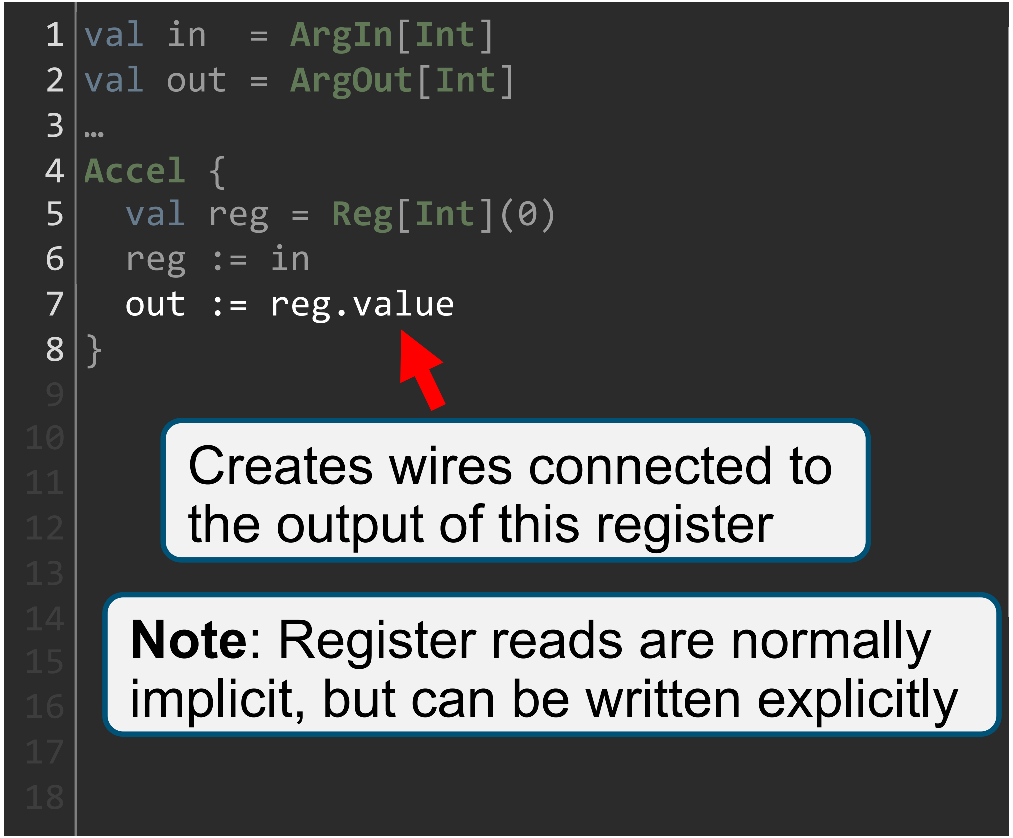


```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int](0)
  reg := in

}
```

Creates a write to input of this register



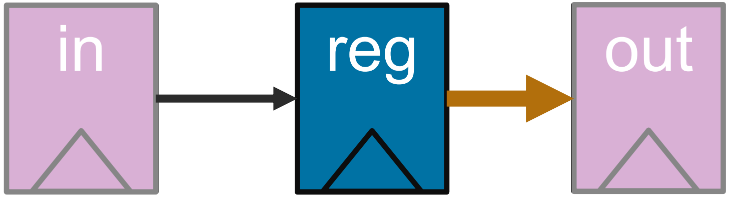
Reg Reading



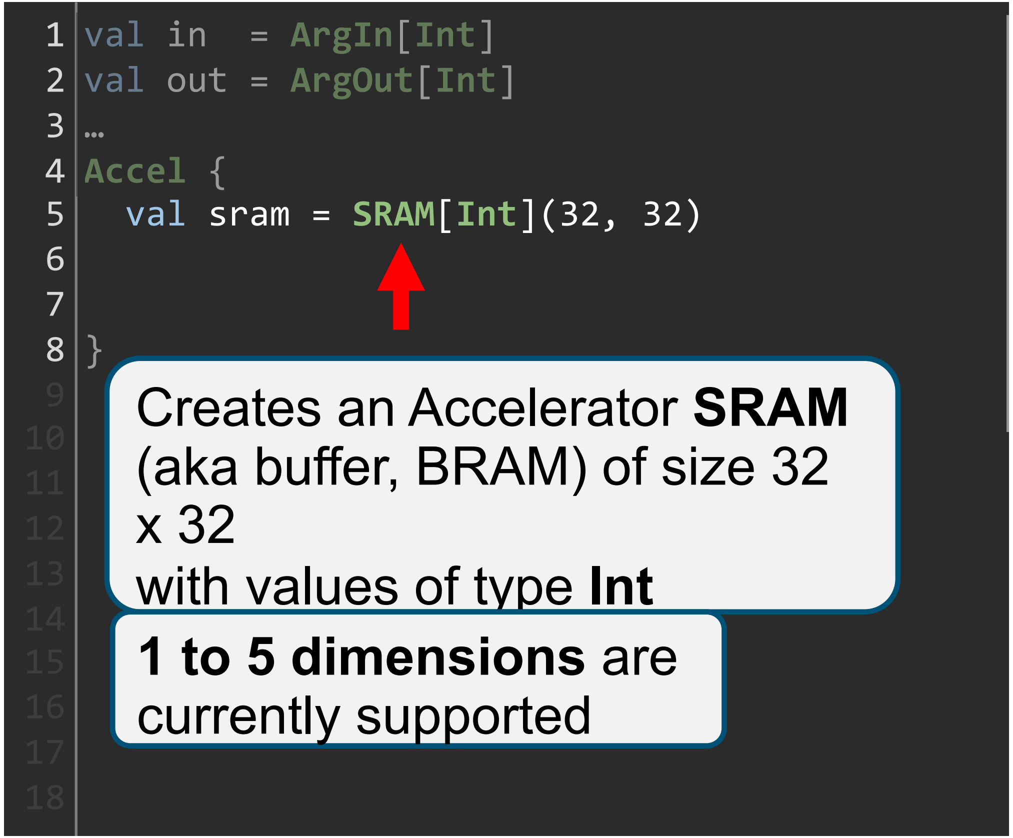
```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val reg = Reg[Int](0)
  reg := in
  out := reg.value
}
```

Creates wires connected to the output of this register

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



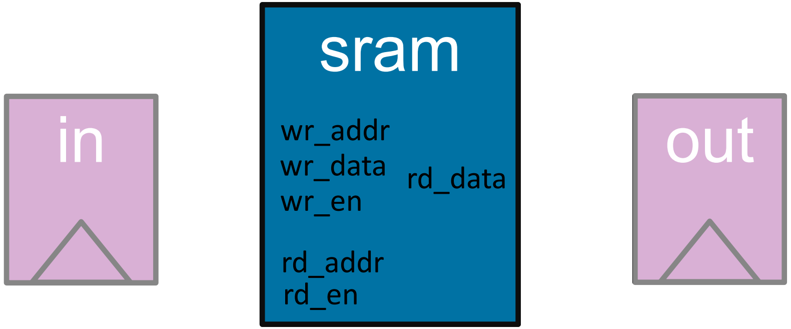
SRAM



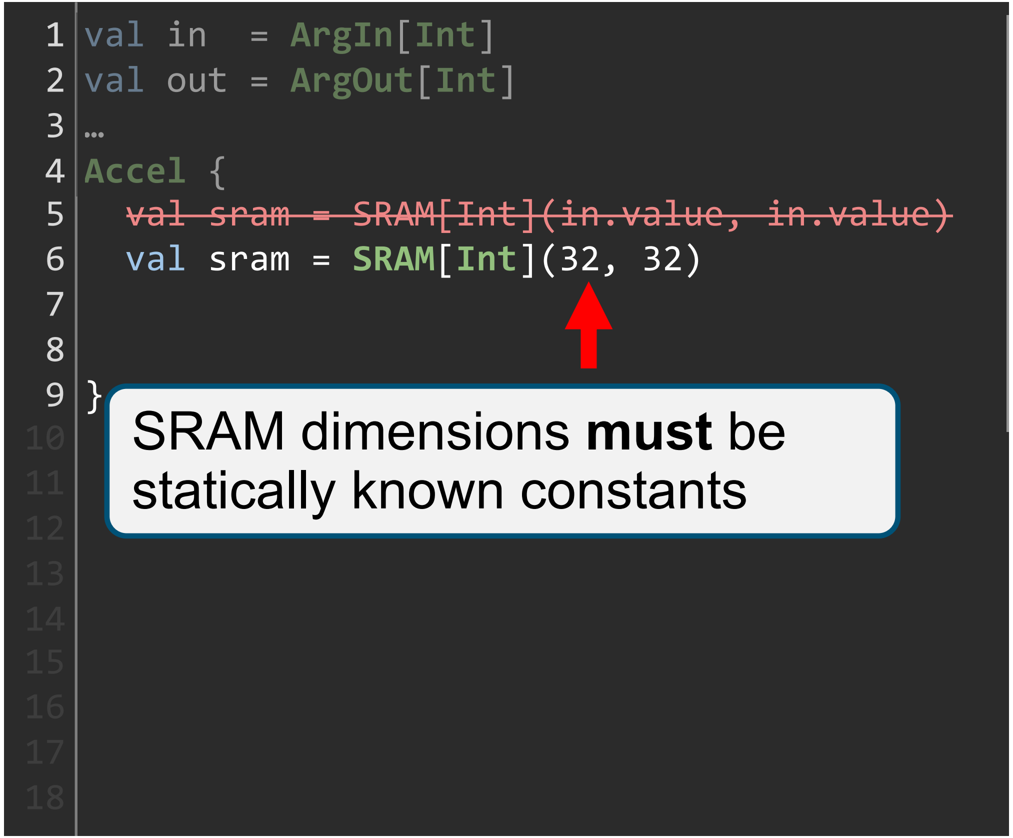
```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val sram = SRAM[Int](32, 32)


}
```

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

1 to 5 dimensions are currently supported

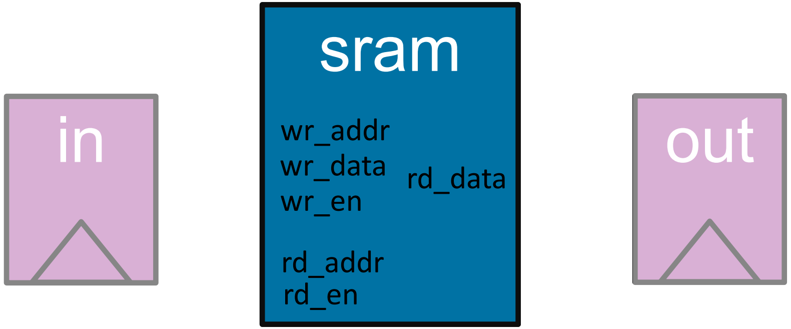


SRAM

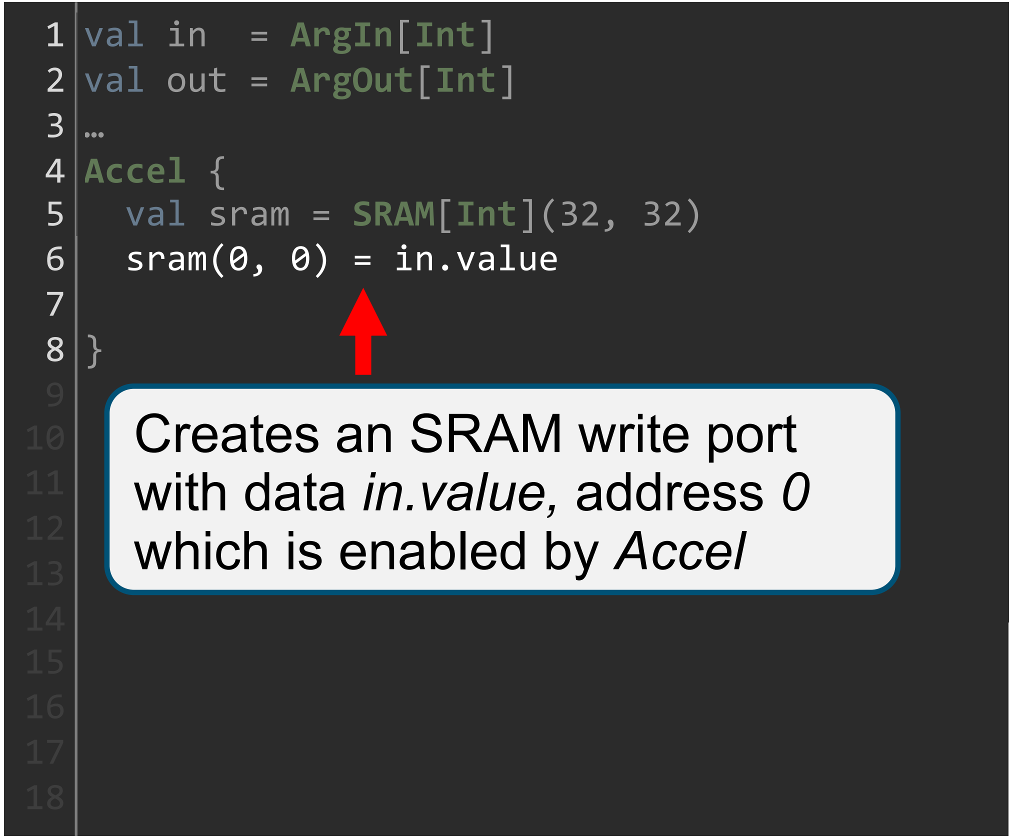


```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  ~~val sram = SRAM[Int](in.value, in.value)~~
  val sram = SRAM[Int](32, 32)


}
```

SRAM dimensions **must** be statically known constants

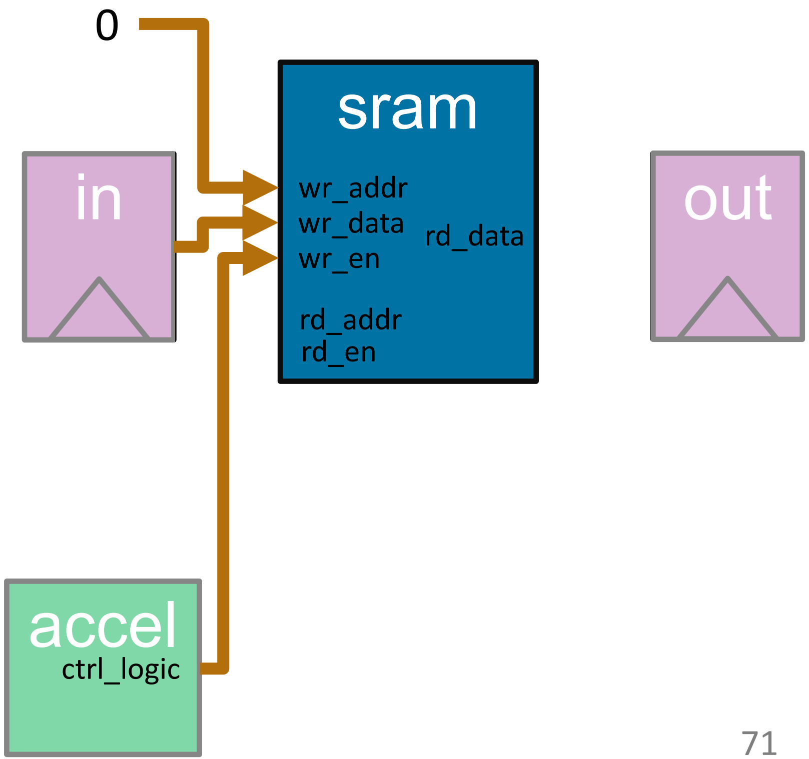


SRAM Writes

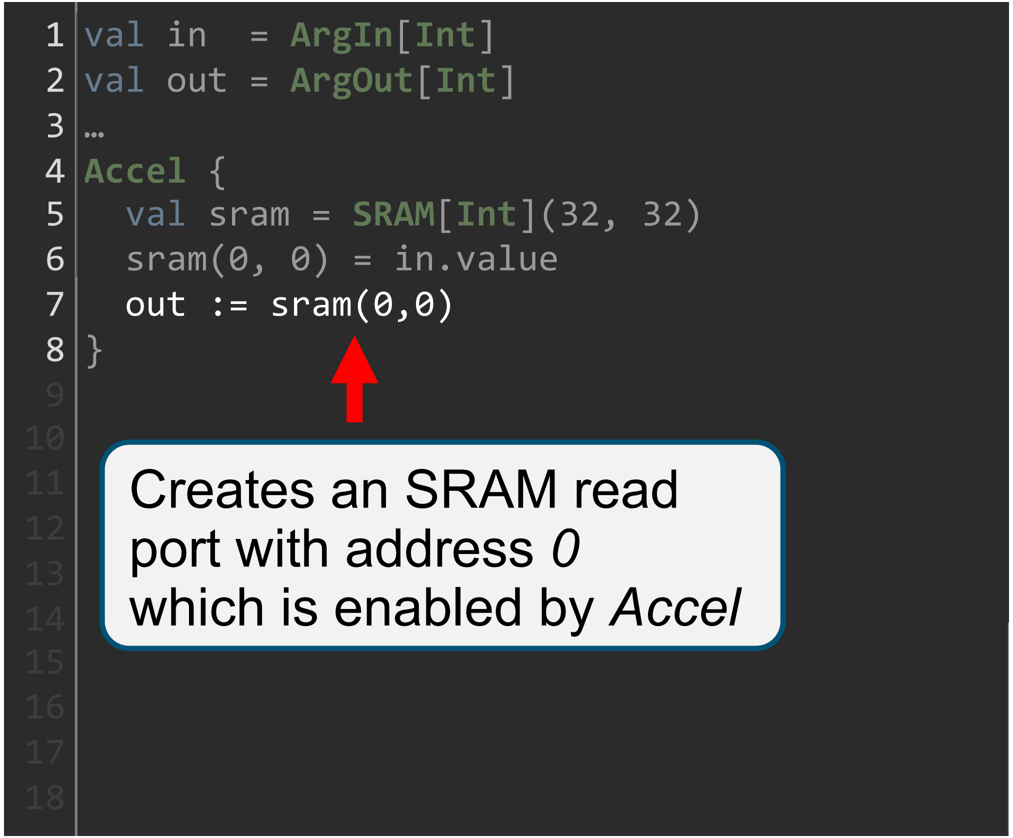


```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val sram = SRAM[Int](32, 32)
  sram(0, 0) = in.value

}
```

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

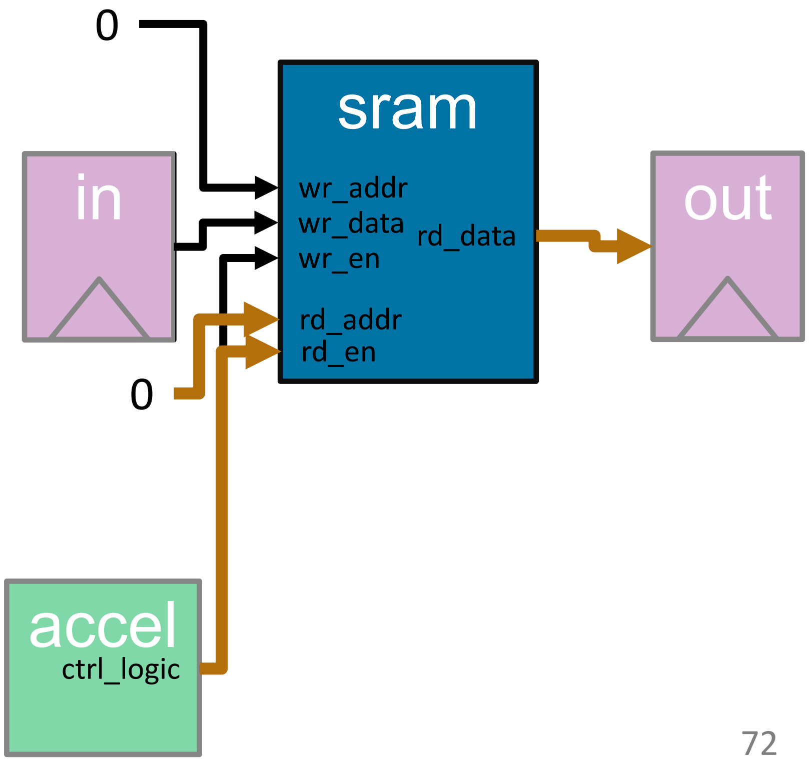


SRAM Reads

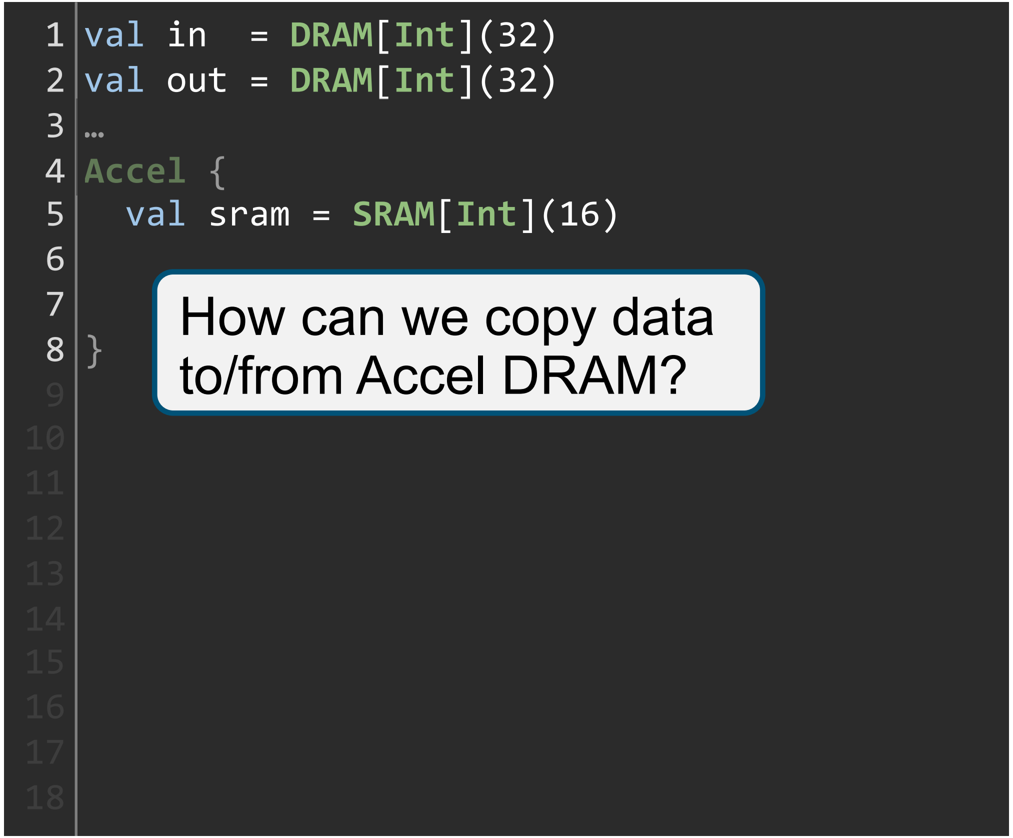


```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val sram = SRAM[Int](32, 32)
  sram(0, 0) = in.value
  out := sram(0,0)
}
```

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

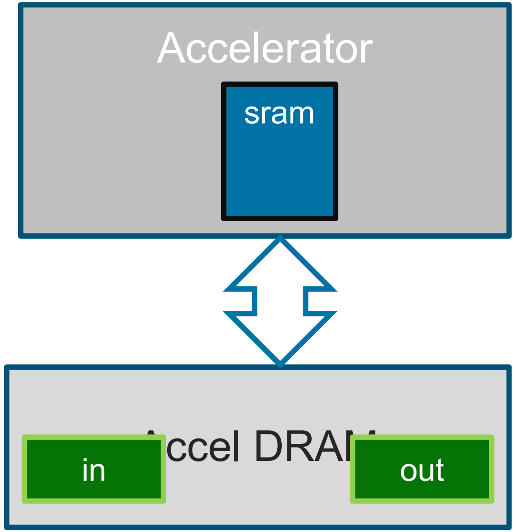


SRAM: Interfacing with DRAM?

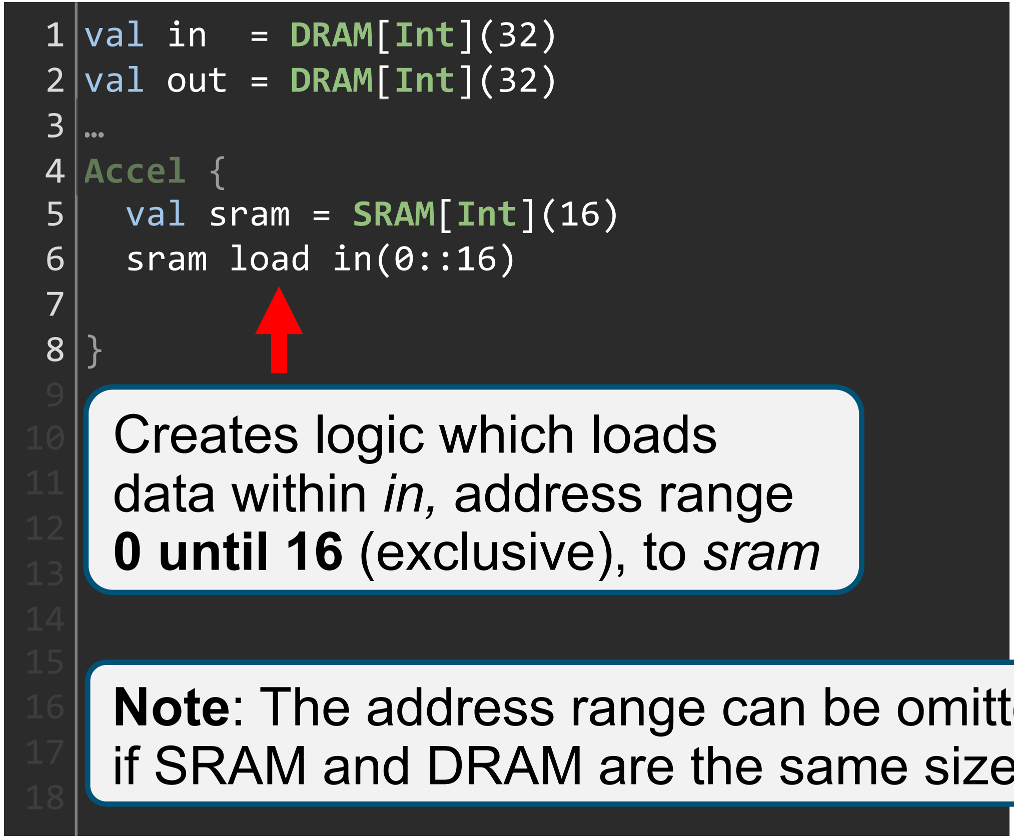


```
val in  = DRAM[Int](32)
val out = DRAM[Int](32)
…
Accel {
  val sram = SRAM[Int](16)


}
```

How can we copy data to/from Accel DRAM?

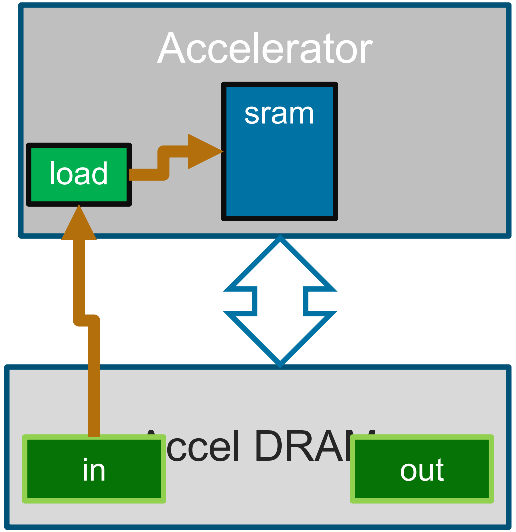


SRAM: Dense Loading from DRAM



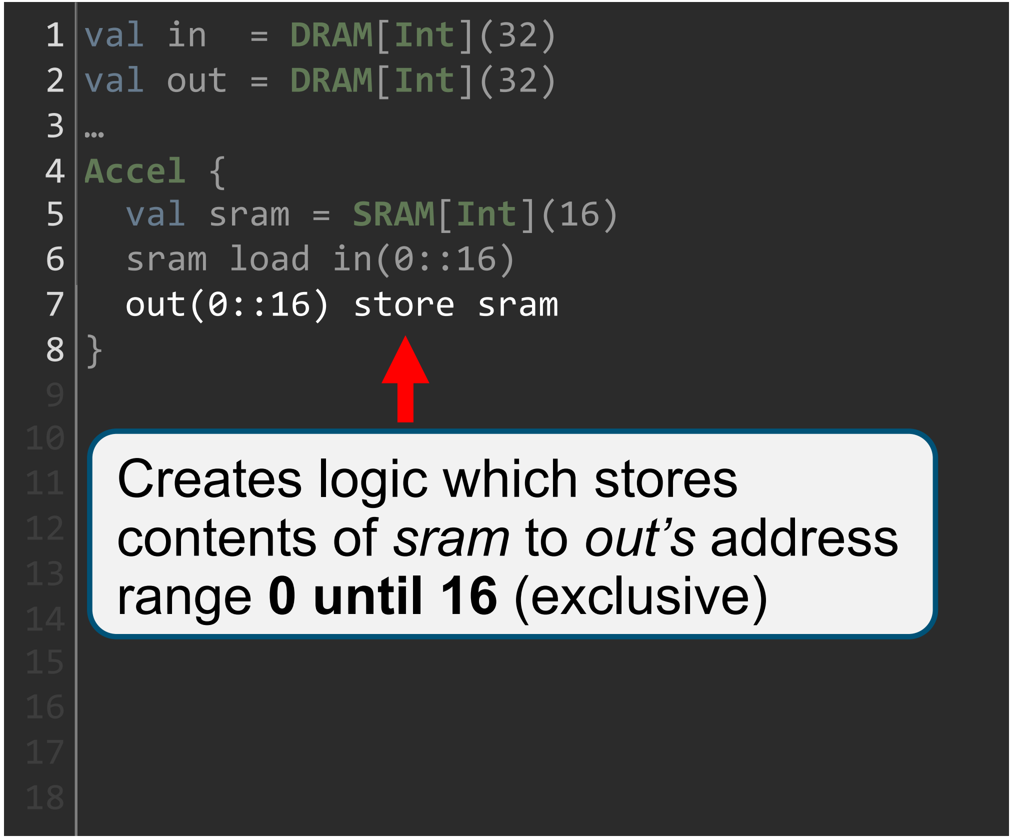
```
val in  = DRAM[Int](32)
val out = DRAM[Int](32)
…
Accel {
  val sram = SRAM[Int](16)
  sram load in(0::16)

}
```

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



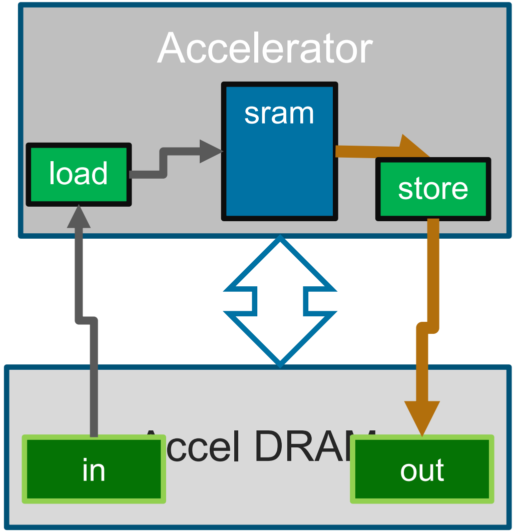
Note: The address range can be omitted if SRAM and DRAM are the same size

SRAM: Dense Storing to DRAM

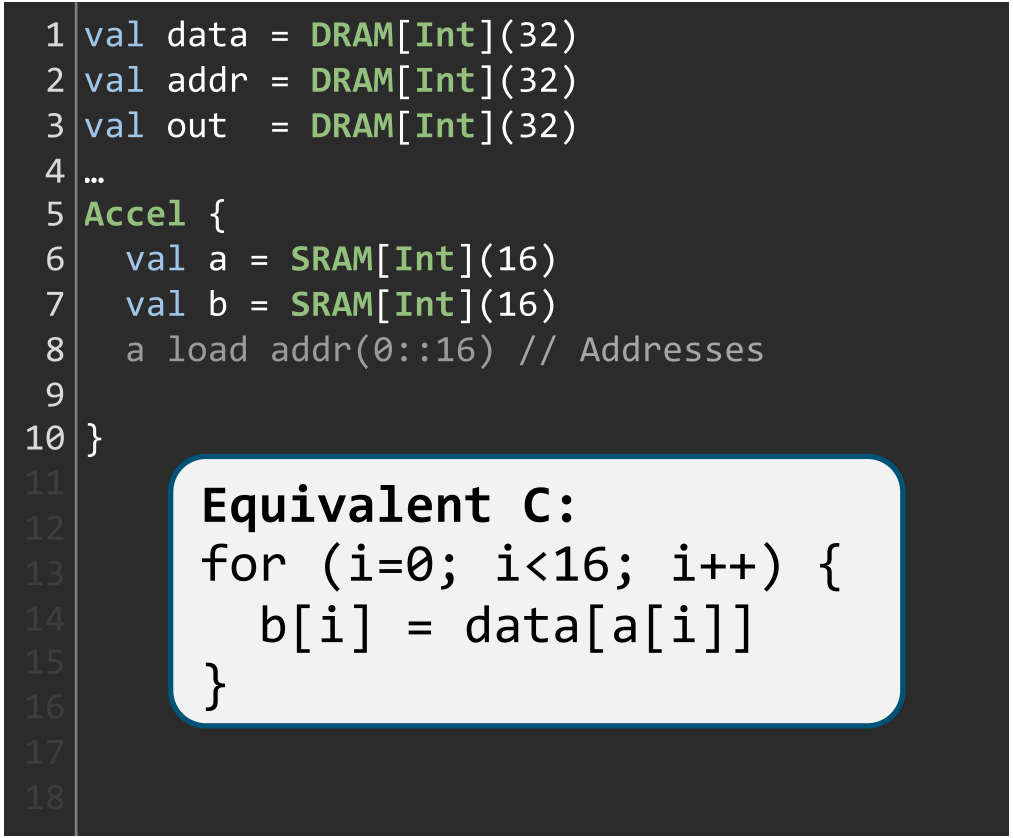


```
val in  = DRAM[Int](32)
val out = DRAM[Int](32)
…
Accel {
  val sram = SRAM[Int](16)
  sram load in(0::16)
  out(0::16) store sram
}
```

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



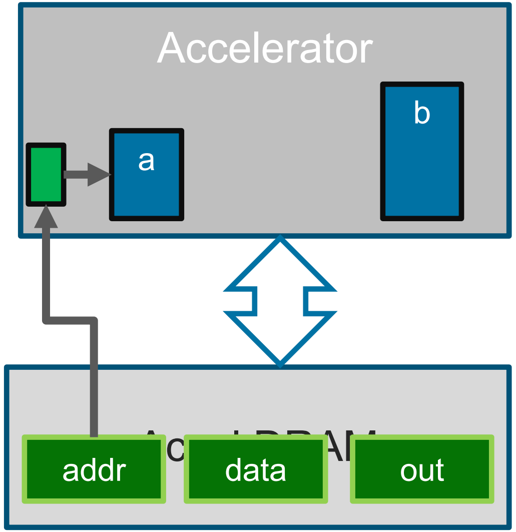
SRAM: Gather from DRAM



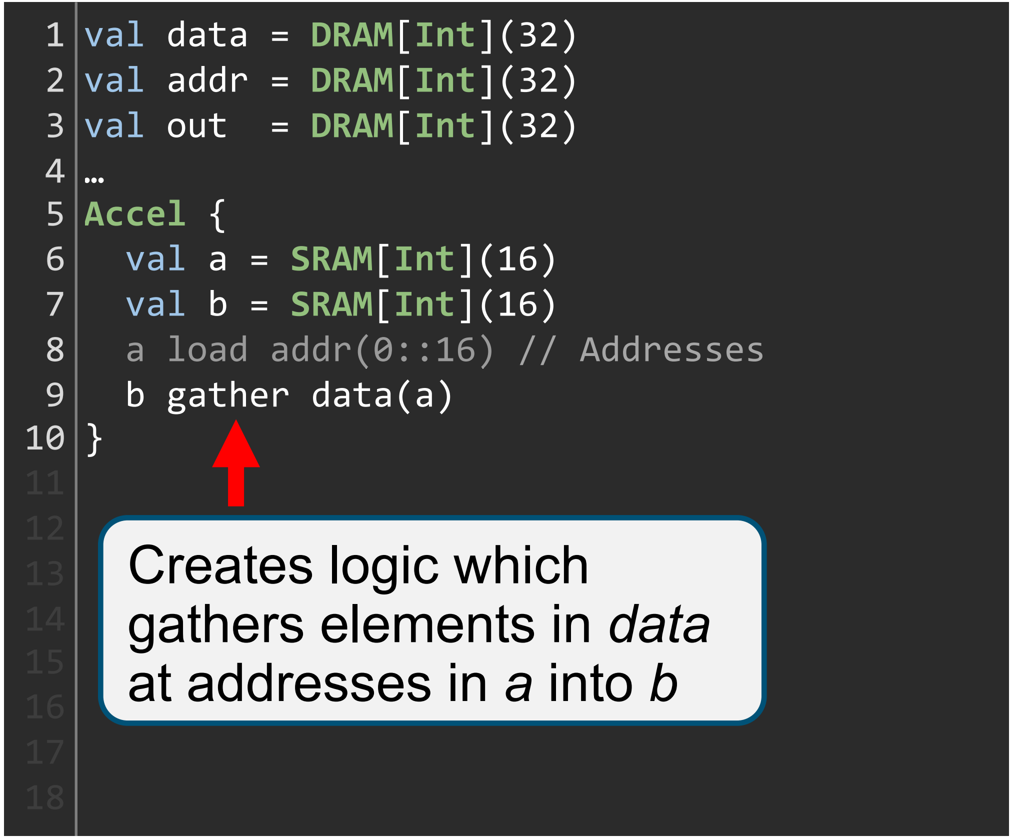
```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses

}
```

Equivalent C:

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

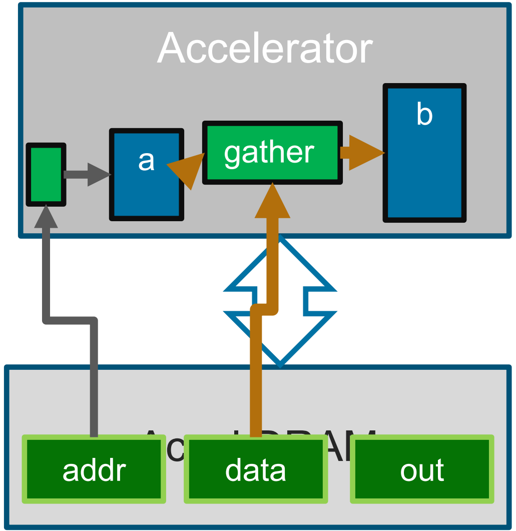


SRAM: Gather from DRAM



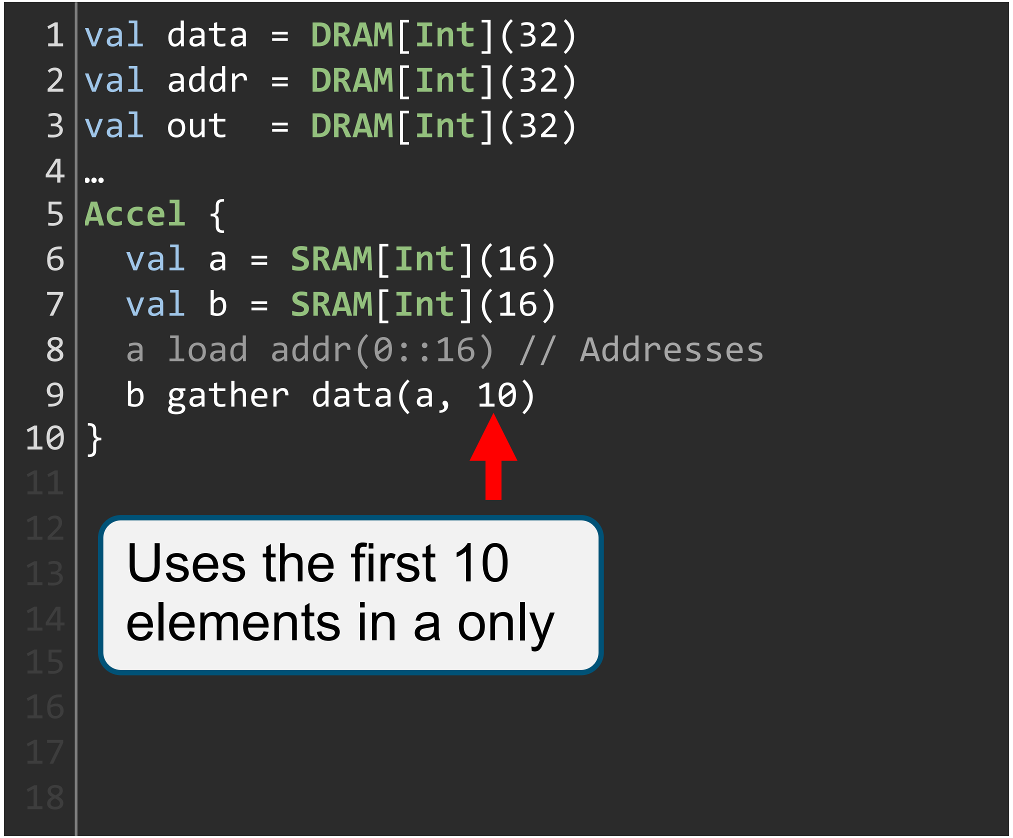
```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  b gather data(a)
}
```

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



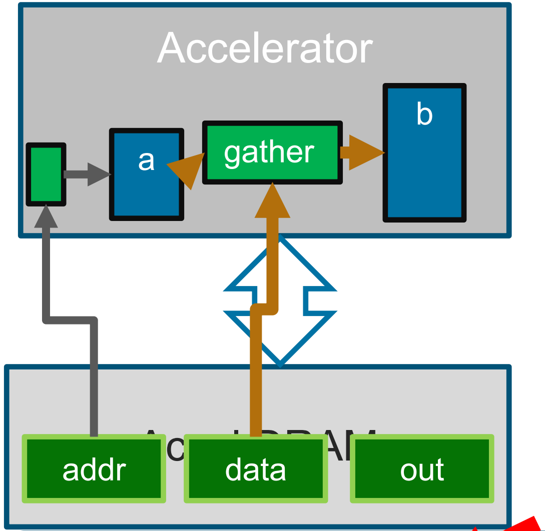
```
for (i=0; i<16; i++) {
  b[i] = data[a[i]]
}
```

SRAM: Gather from DRAM



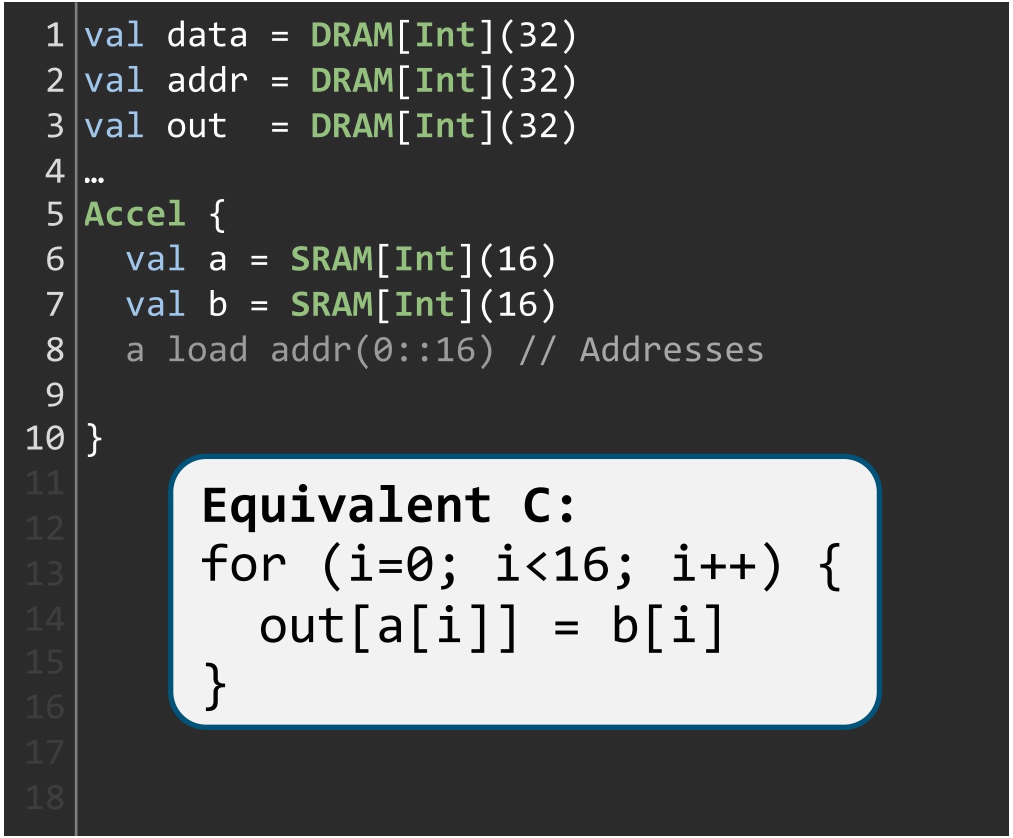
```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  b gather data(a, 10)
}
```

Uses the first 10 elements in a only



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

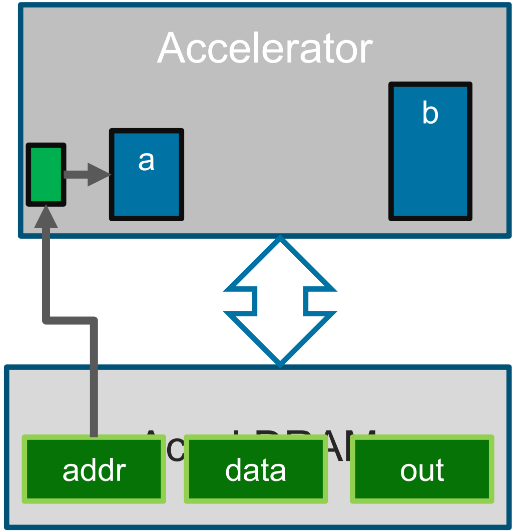
SRAM: Scatter to DRAM



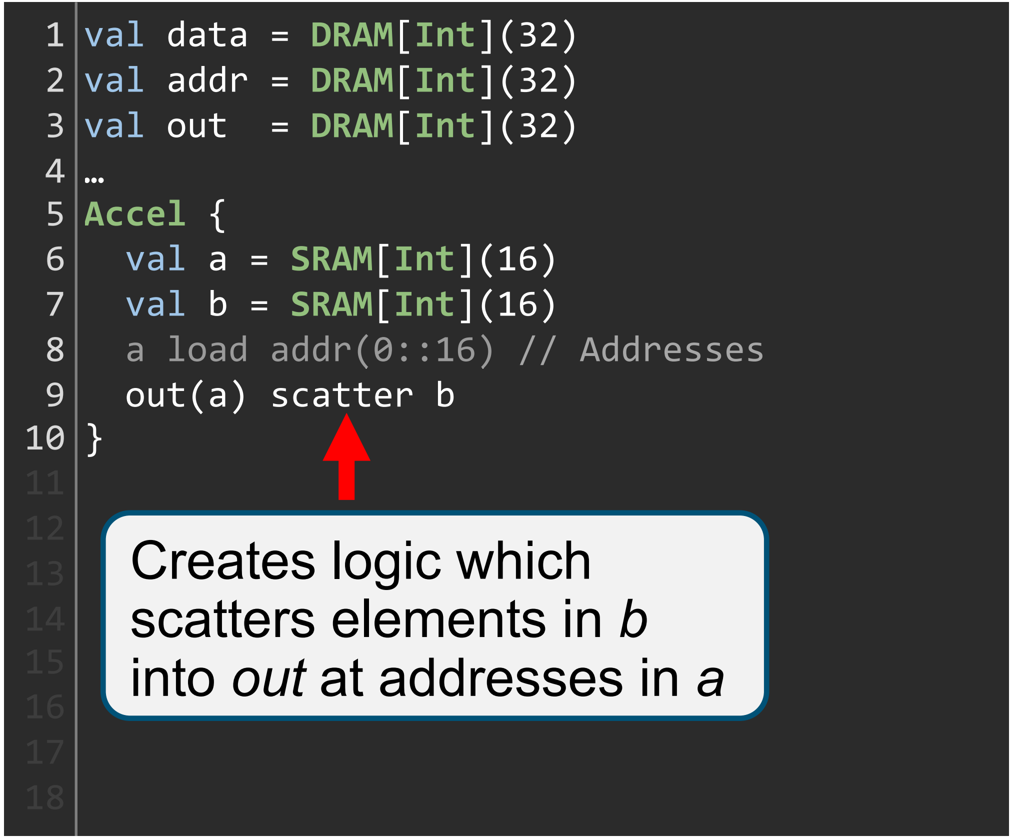
```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses

}
```

Equivalent C:

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

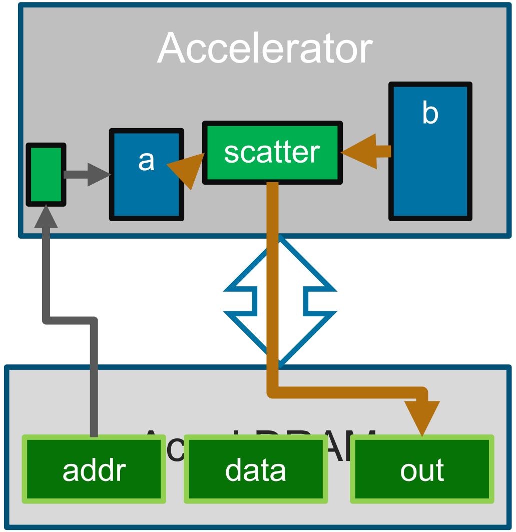


SRAM: Scatter to DRAM



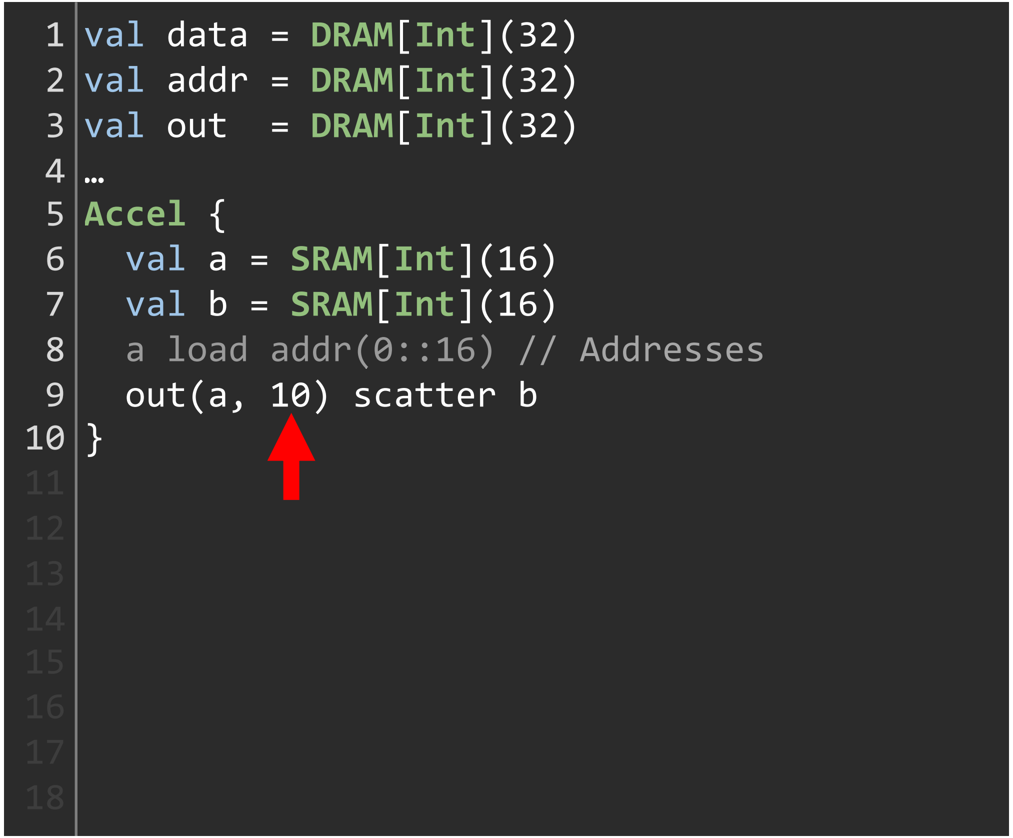
```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  out(a) scatter b
}
```

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

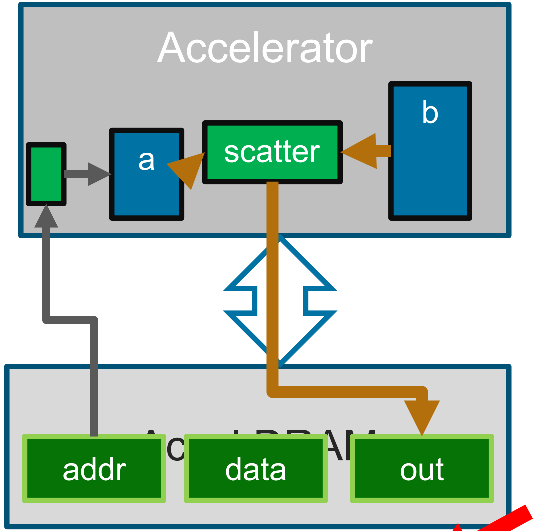


```
for (i=0; i<16; i++) {
  out[a[i]] = b[i]
}
```

SRAM: Scatter to DRAM

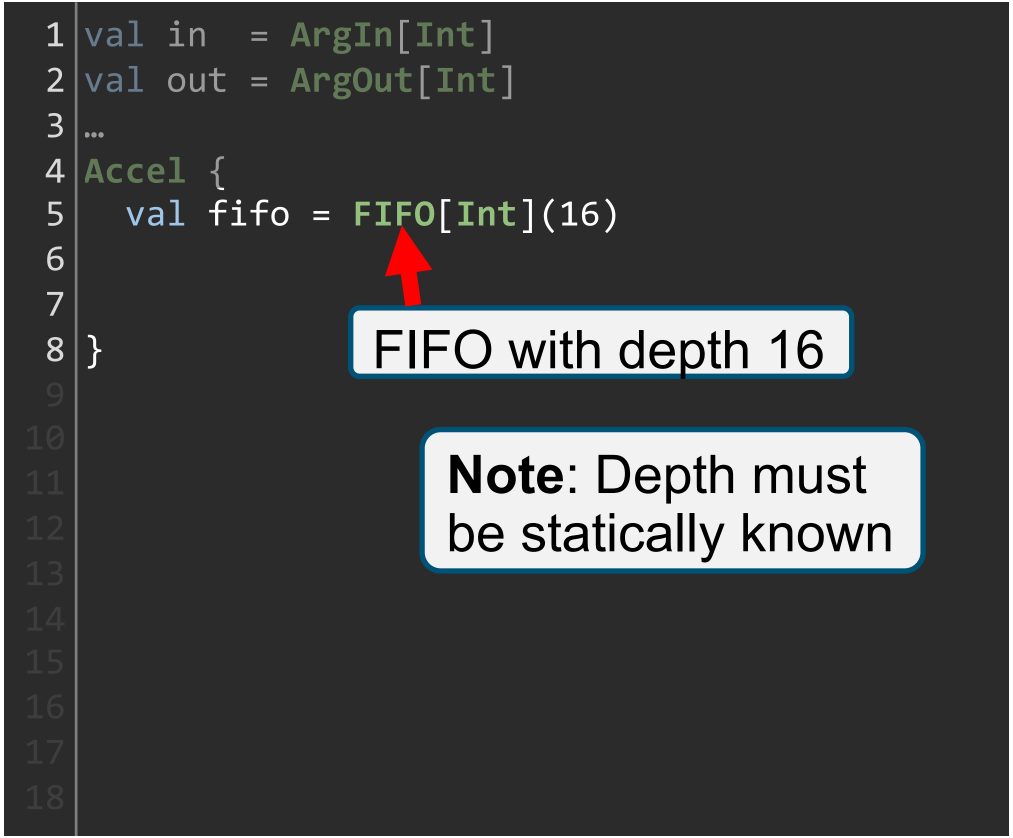


```
val data = DRAM[Int](32)
val addr = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val a = SRAM[Int](16)
  val b = SRAM[Int](16)
  a load addr(0::16) // Addresses
  out(a, 10) scatter b
}
```



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

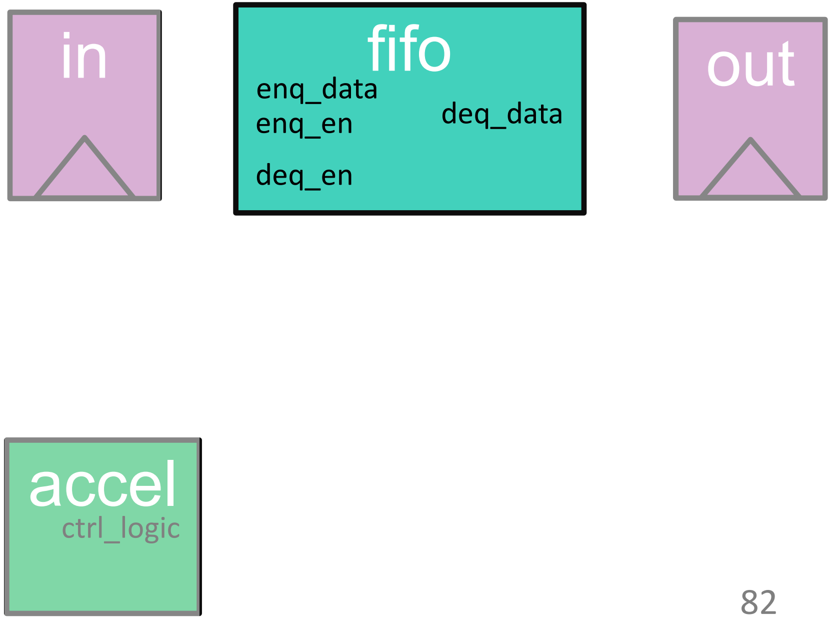
FIFO



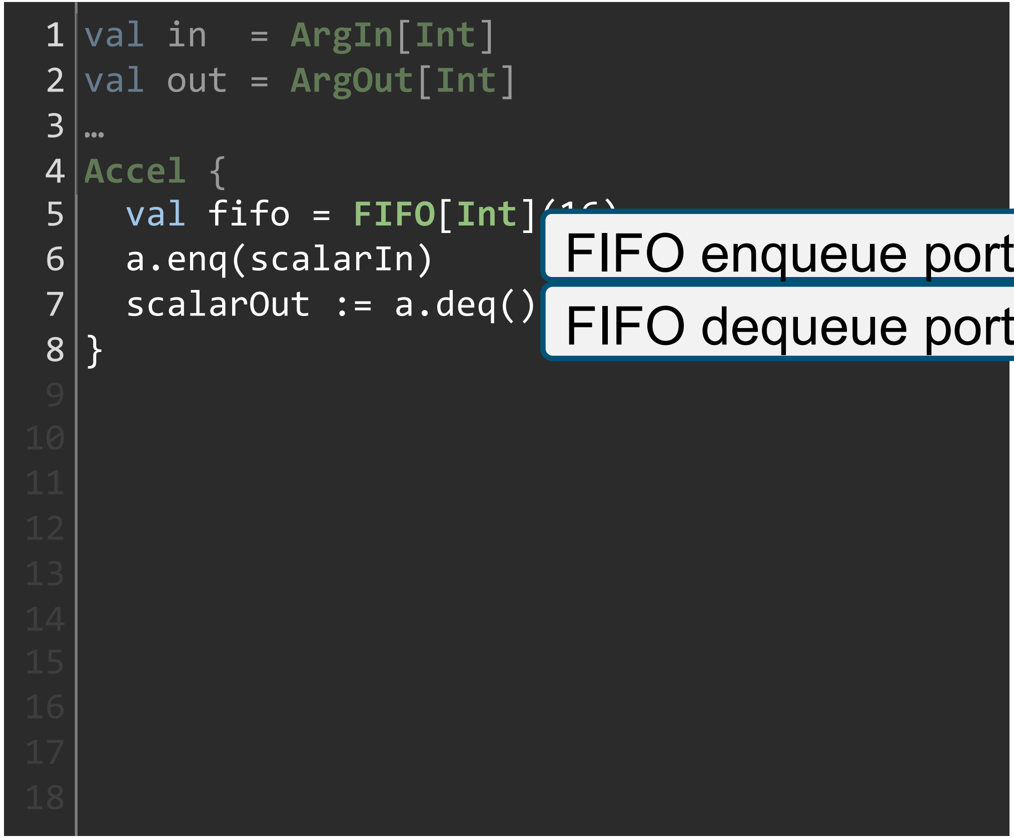
```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val fifo = FIFO[Int](16)


}
```

FIFO with depth 16

Note: Depth must be statically known



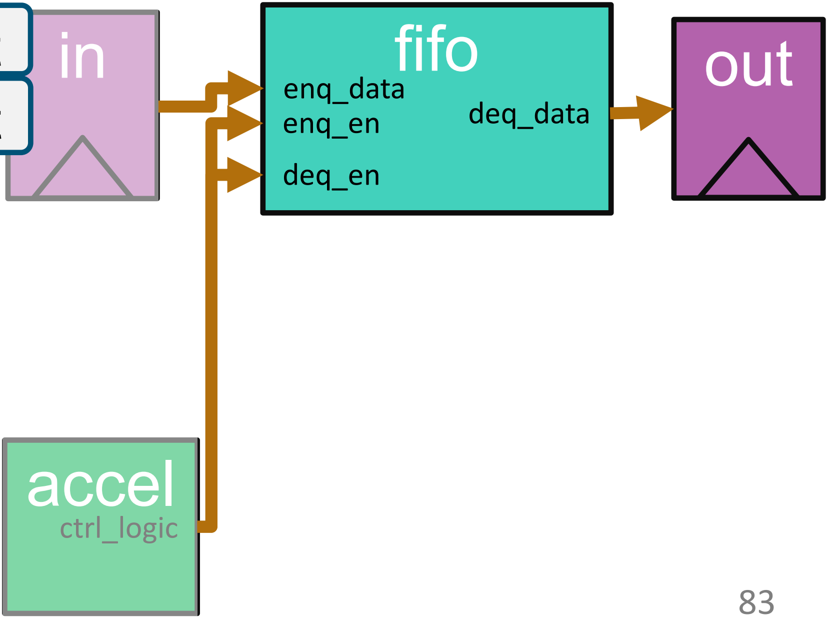
FIFO: Enqueueing / Dequeueing



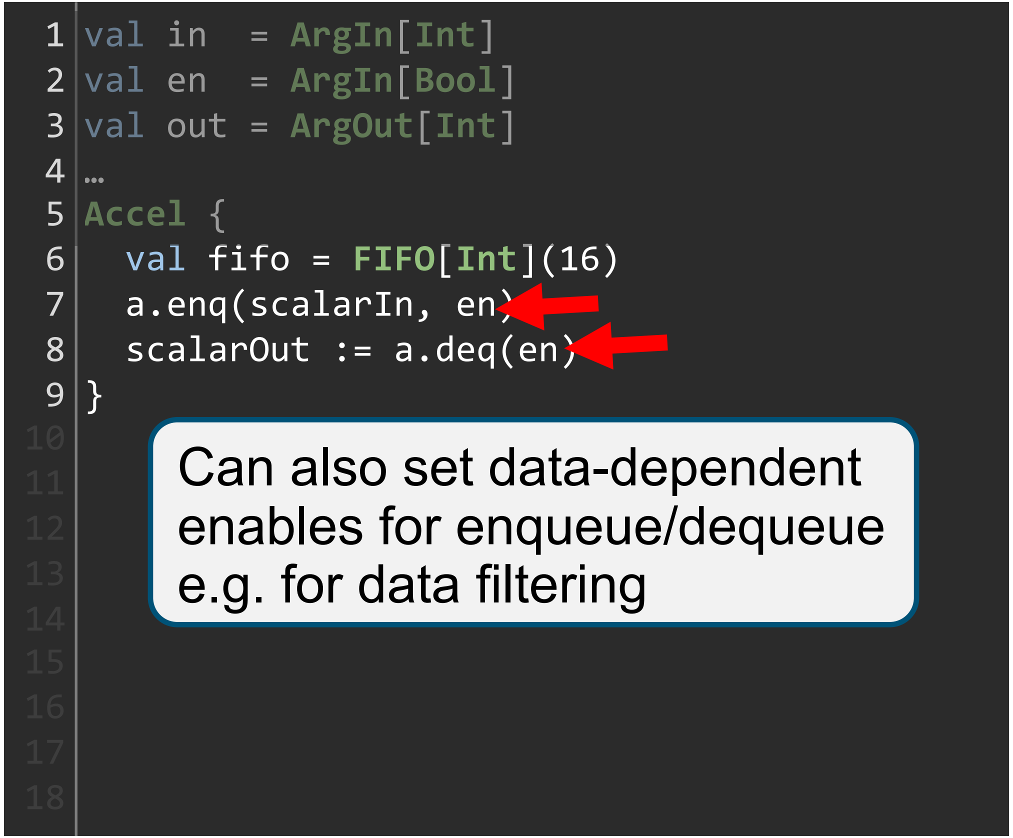
```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val fifo = FIFO[Int](16)
  a.enq(scalarIn)
  scalarOut := a.deq()
}
```

FIFO enqueue port

FIFO dequeue port

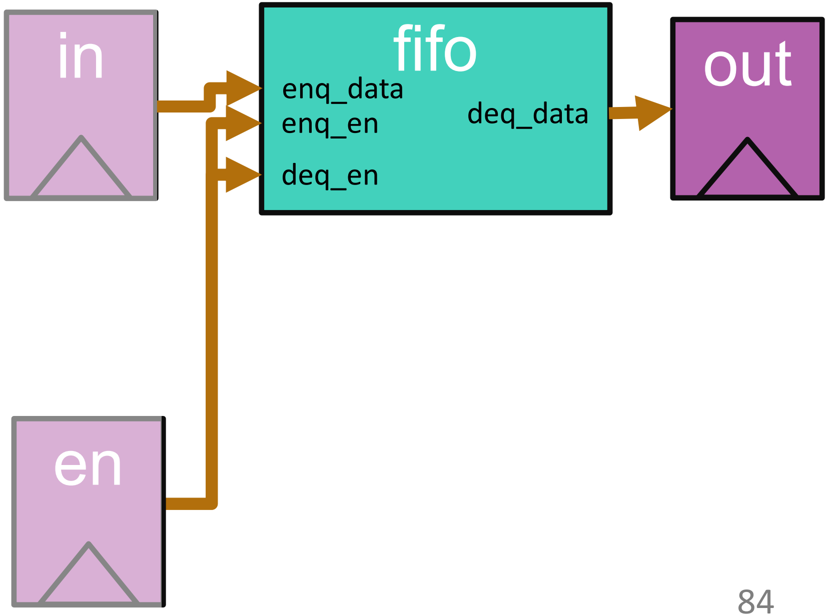


FIFO: Enabled Enqueuing/Dequeueing

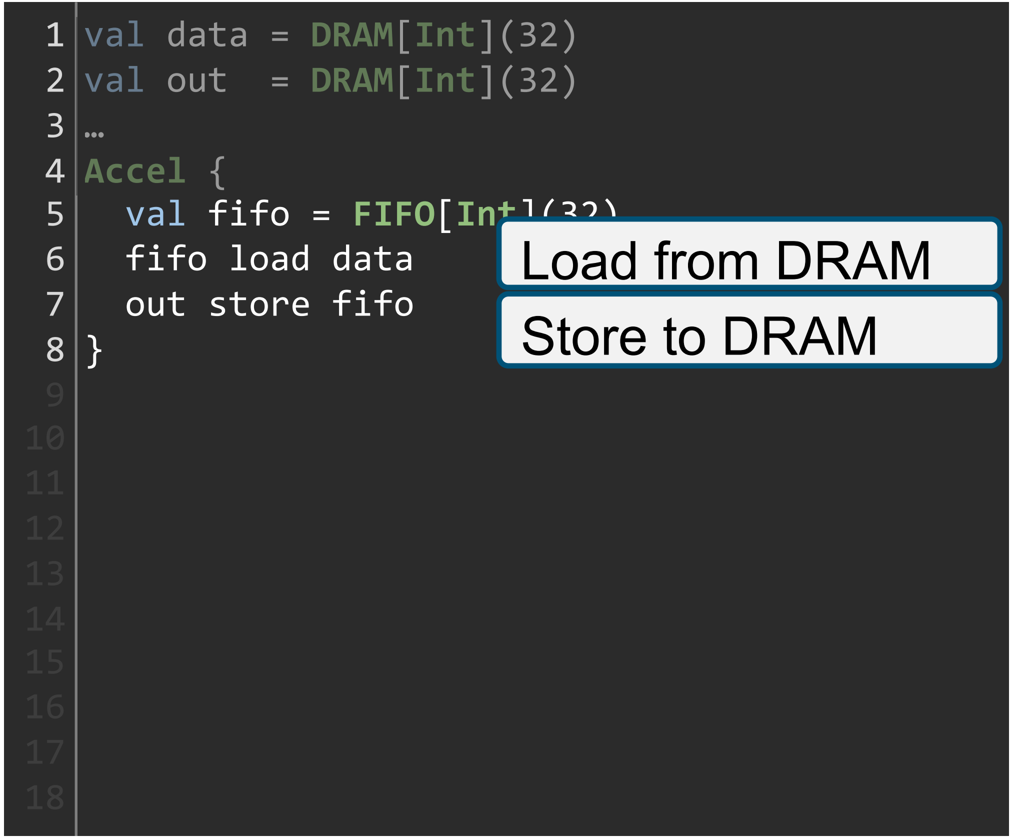


```
val in  = ArgIn[Int]
val en  = ArgIn[Bool]
val out = ArgOut[Int]
…
Accel {
  val fifo = FIFO[Int](16)
  a.enq(scalarIn, en)
  scalarOut := a.deq(en)
}
```

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



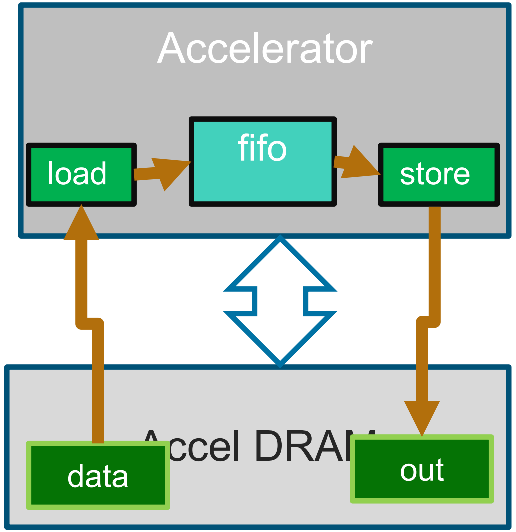
FIFO: Transfers to/from DRAM



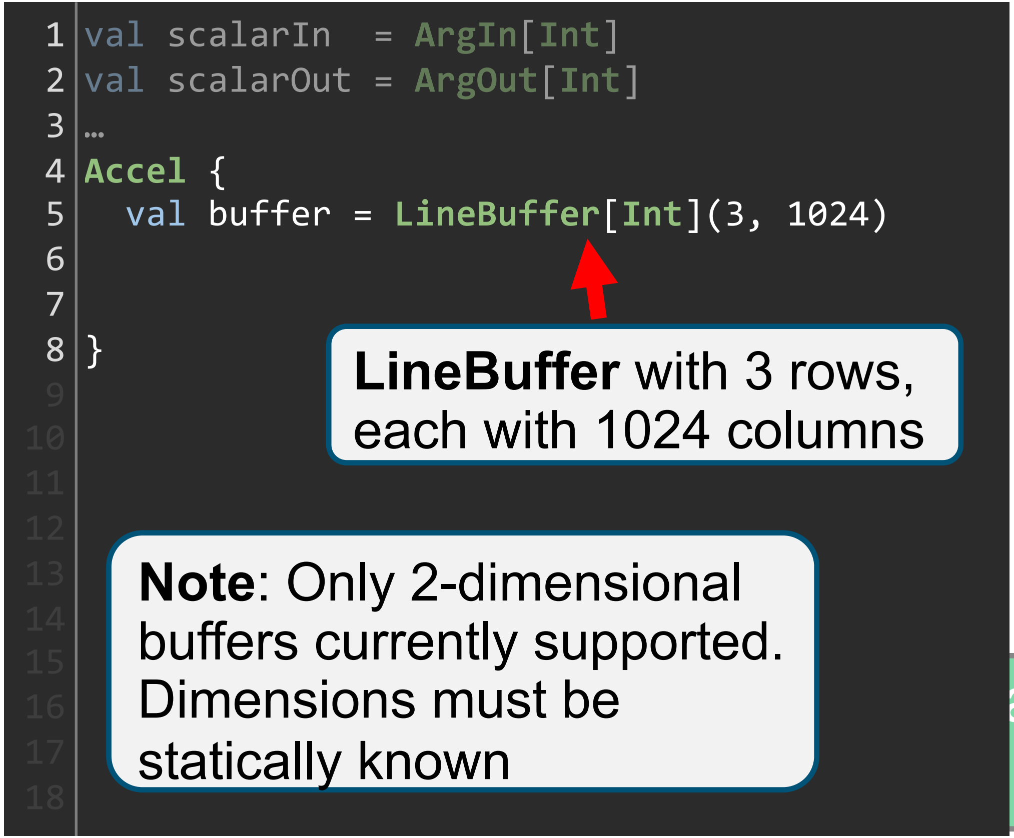
```
val data = DRAM[Int](32)
val out  = DRAM[Int](32)
…
Accel {
  val fifo = FIFO[Int](32)
  fifo load data
  out store fifo
}
```

Load from DRAM

Store to DRAM



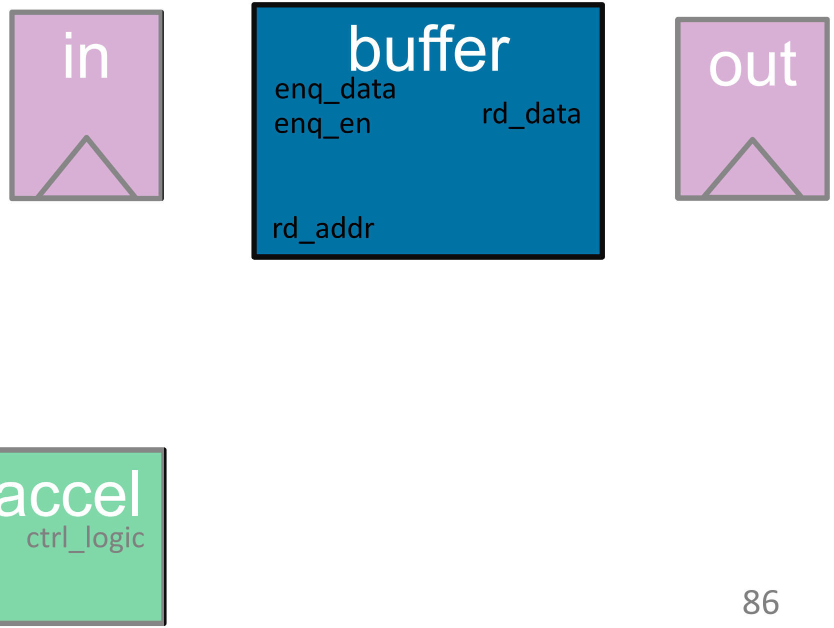
LineBuffer



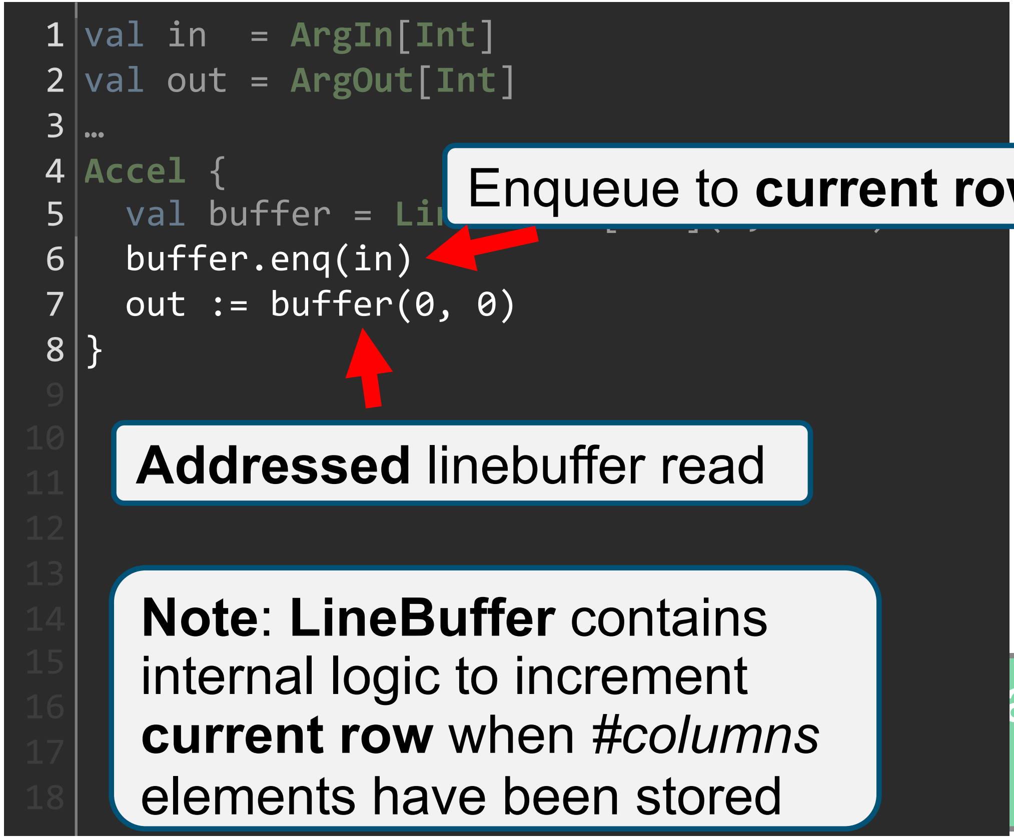
```
val scalarIn  = ArgIn[Int]
val scalarOut = ArgOut[Int]
…
Accel {
  val buffer = LineBuffer[Int](3, 1024)


}
```

LineBuffer with 3 rows, each with 1024 columns

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



LineBuffer: Enqueueing / Reading

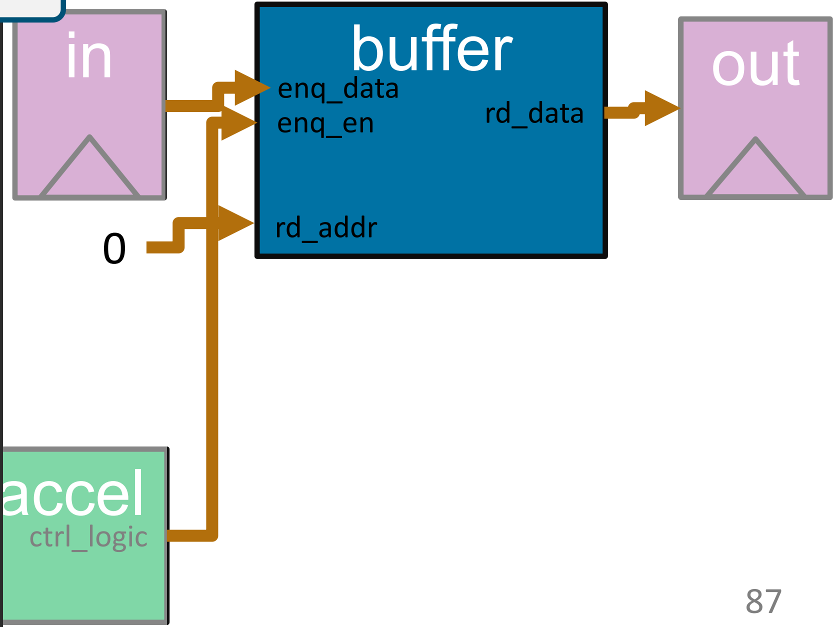


```
val in  = ArgIn[Int]
val out = ArgOut[Int]
…
Accel {
  val buffer = Li
  buffer.enq(in)
  out := buffer(0, 0)
}
```

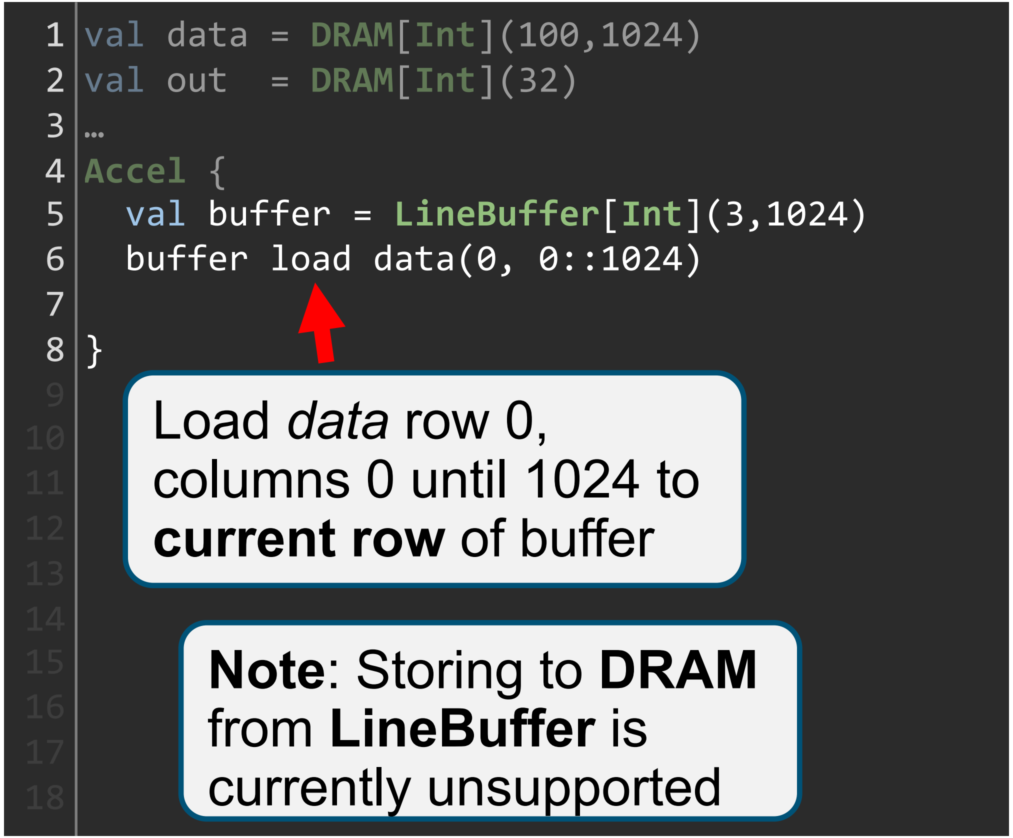
Enqueue to **current row**

Addressed linebuffer read

Note: **LineBuffer** contains internal logic to increment **current row** when *#columns* elements have been stored



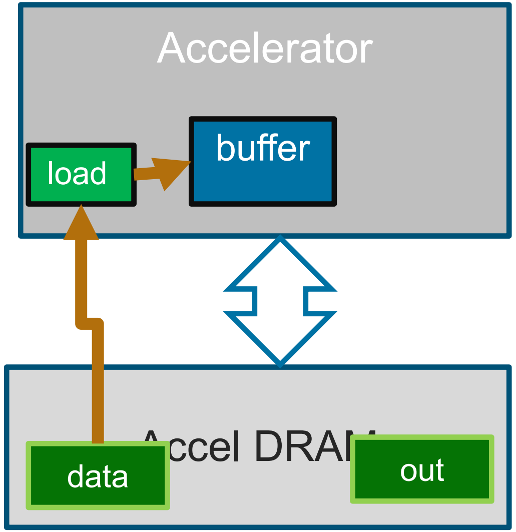
LineBuffer: Loading from DRAM



```
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

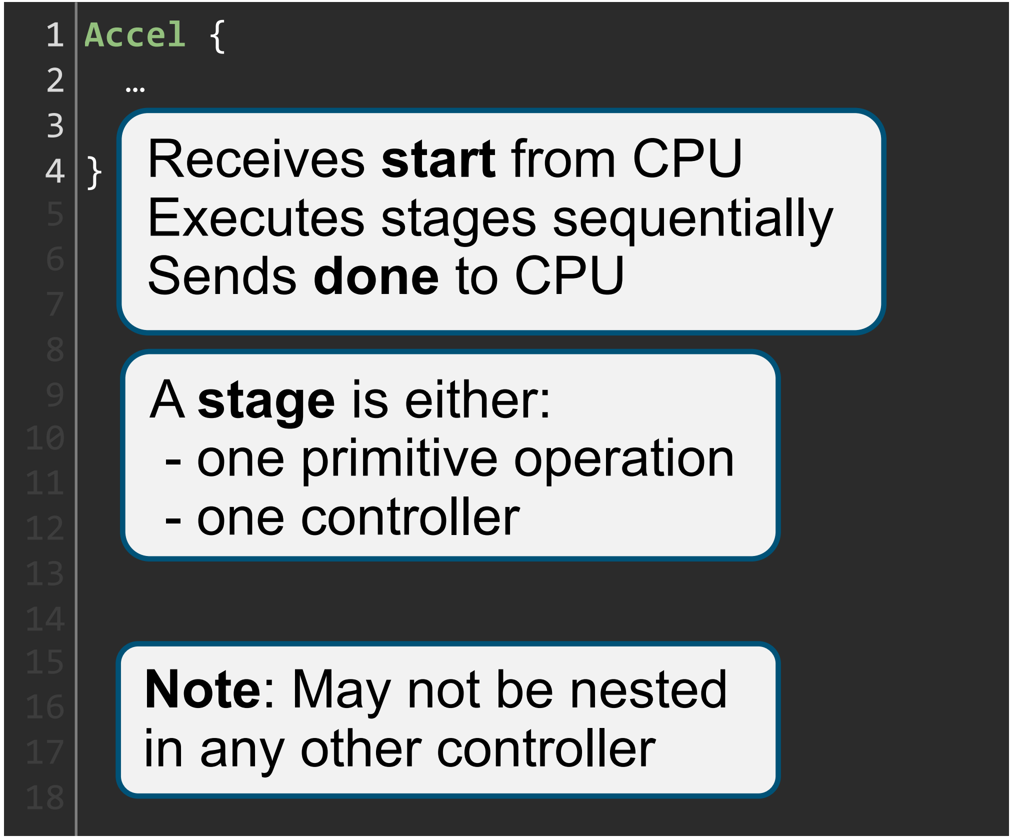


A Note About Ports

- Spatial compiler makes best effort to minimize amount of resources needed to implement logical memories
- 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

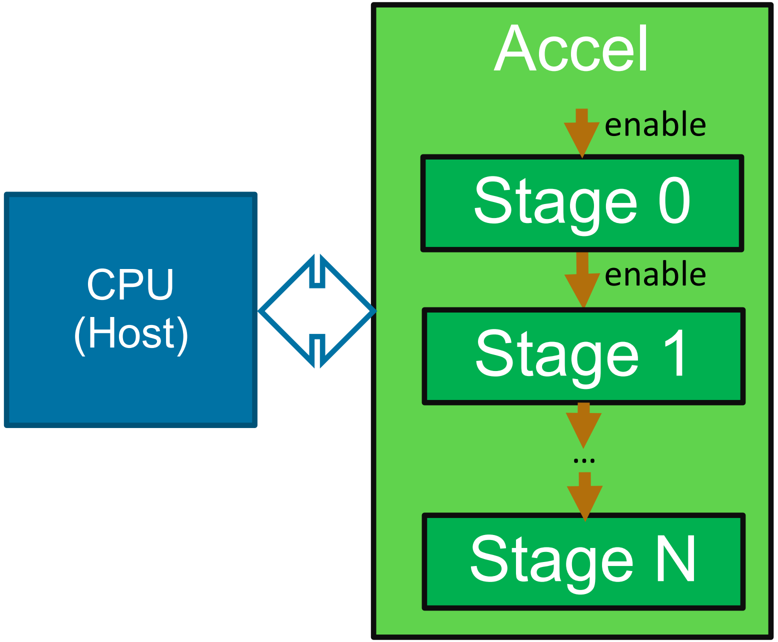


```
Accel {
  …

}
```

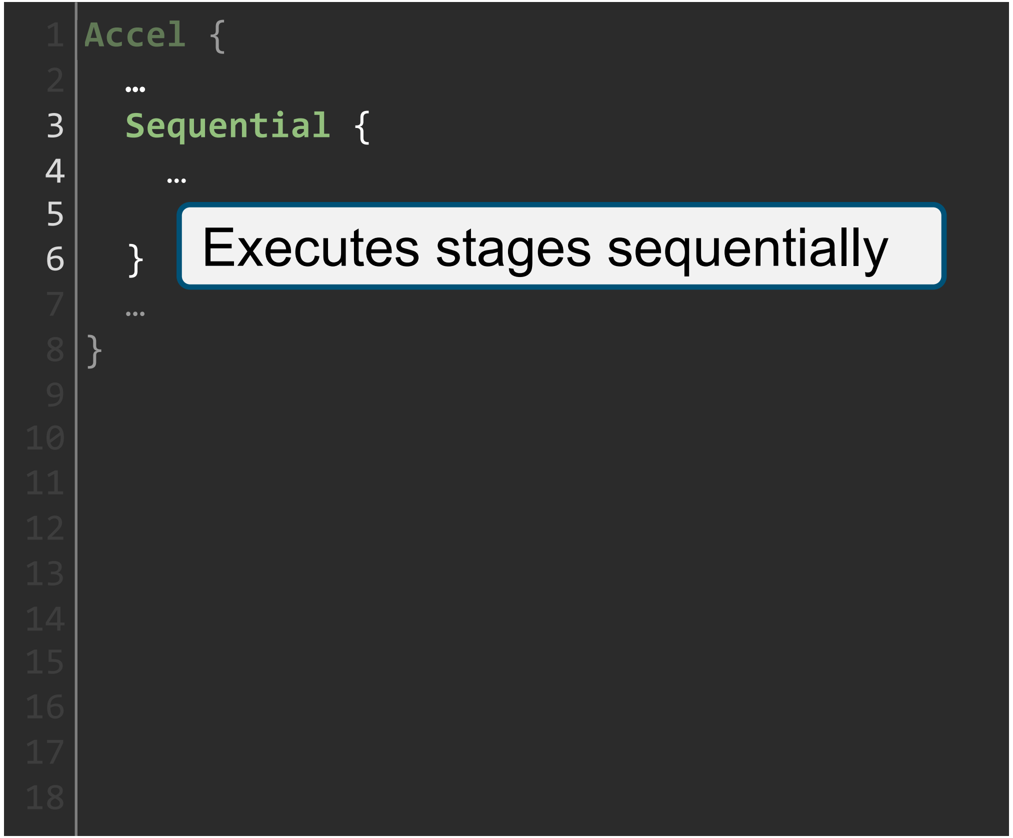
Receives **start** from CPU
Executes stages sequentially
Sends **done** to CPU

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

Note: May not be nested in any other controller

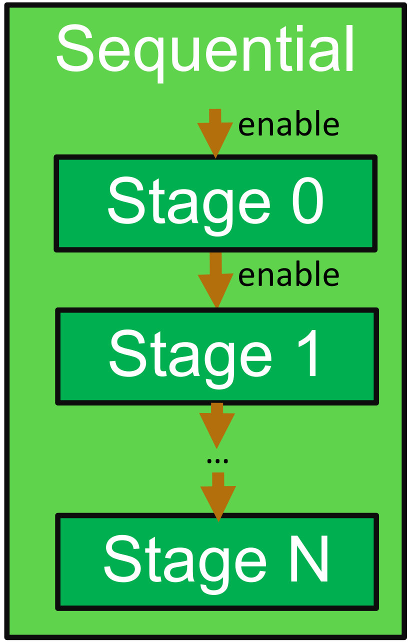


Sequential

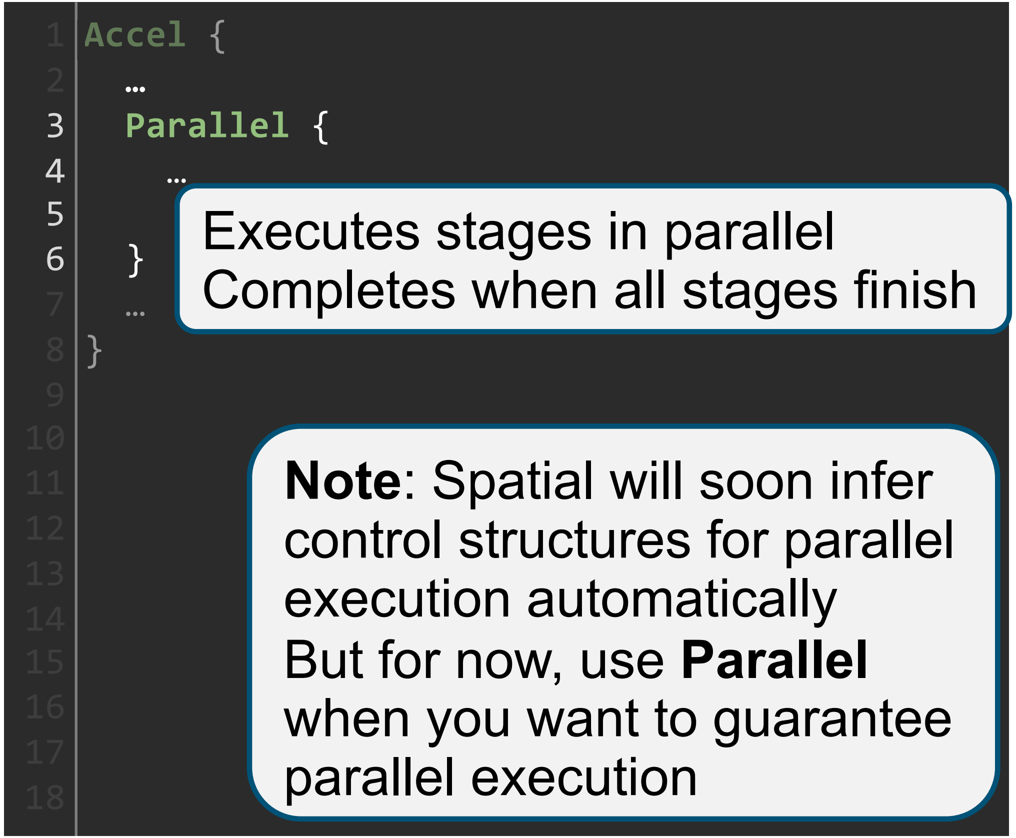


```
Accel {
  …
  Sequential {
    …

  }
  …
}
```

Executes stages sequentially



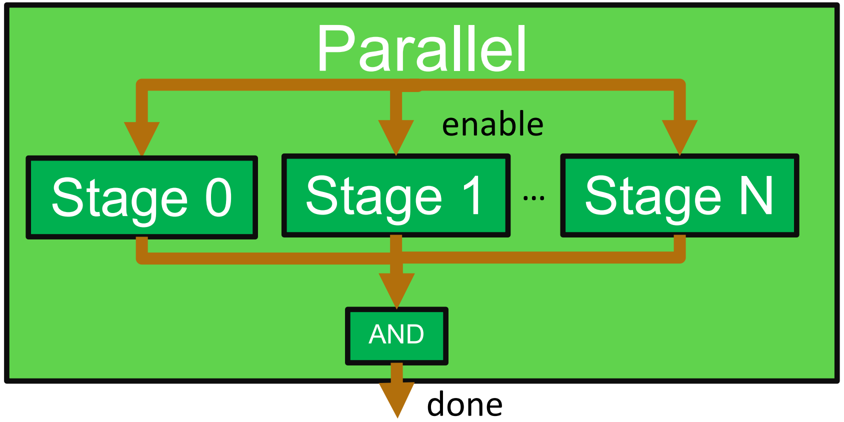
Parallel



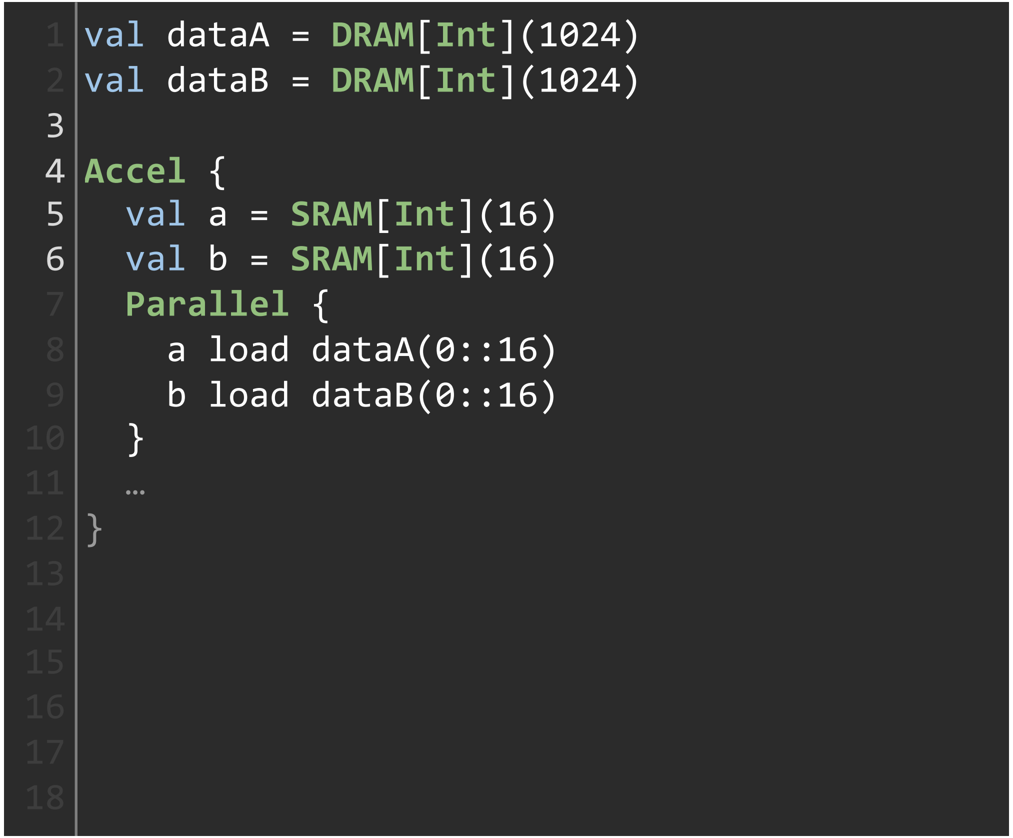
```
Accel {
  …
  Parallel {
    …
  }
  …
}
```

Executes stages in parallel
Completes when all stages finish

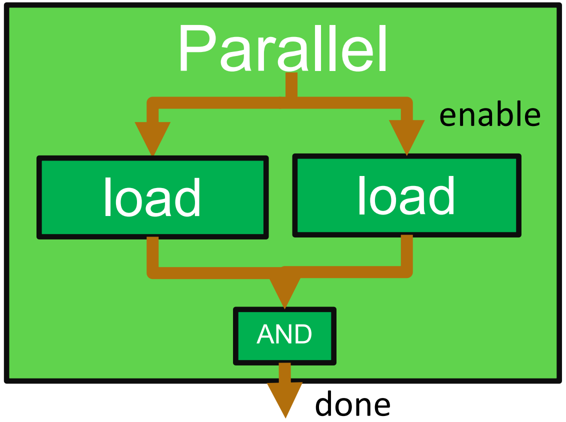
Note: Spatial will soon infer control structures for parallel execution automatically
But for now, use **Parallel** when you want to guarantee parallel execution



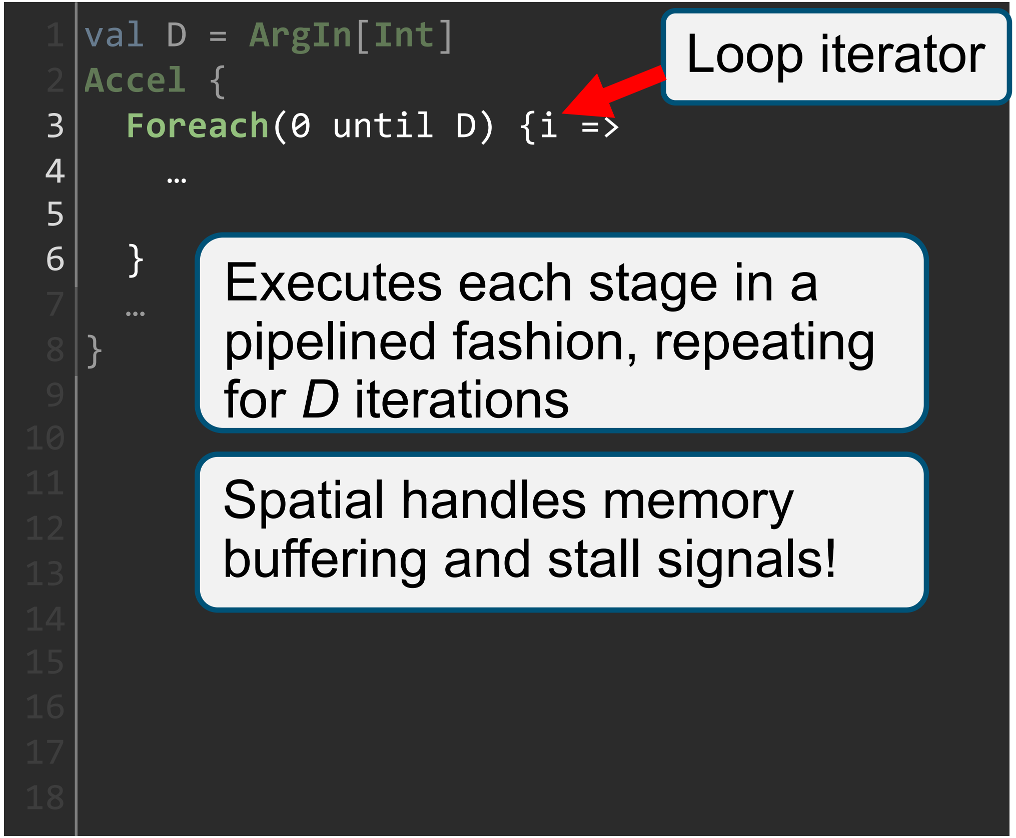
Parallel: Example



```
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

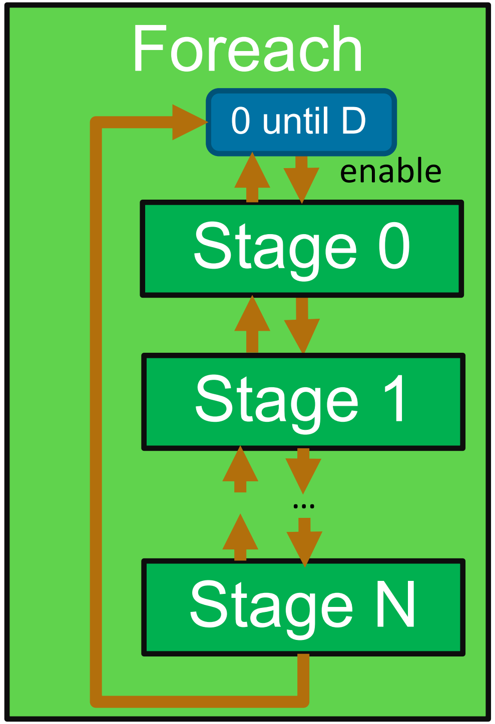


```
val D = ArgIn[Int]
Accel {
  Foreach(0 until D) {i =>
    …

  }
  …
}
```

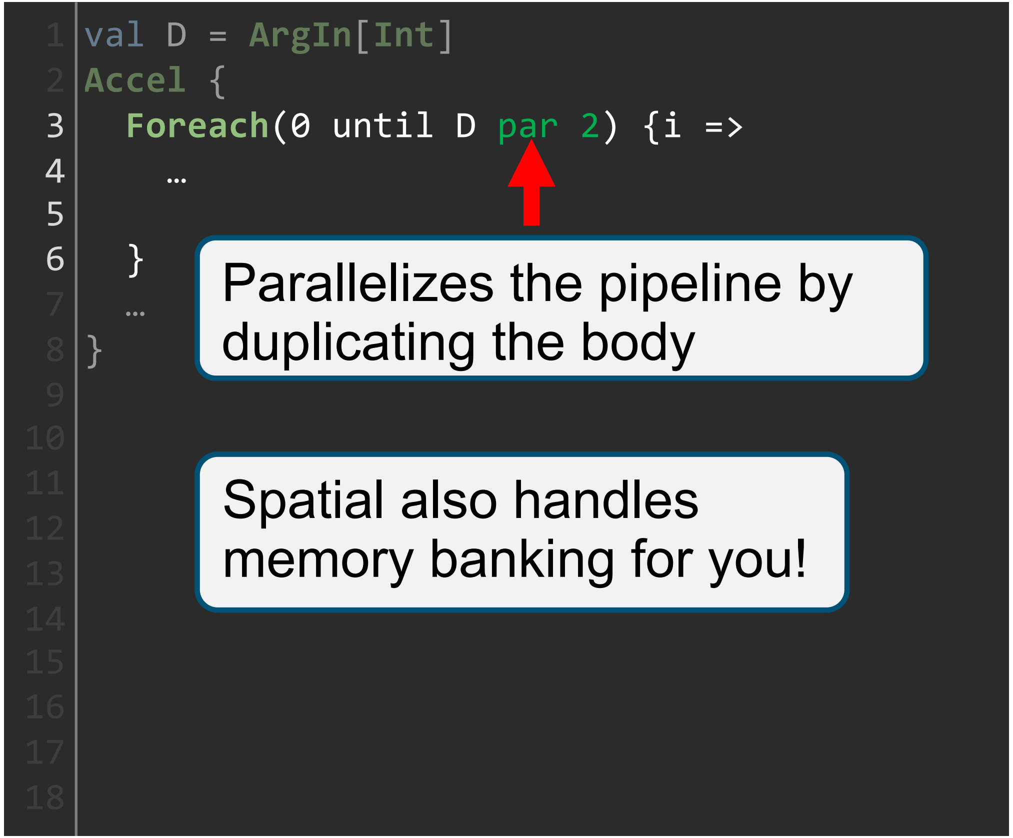
Loop iterator

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

Spatial handles memory buffering and stall signals!



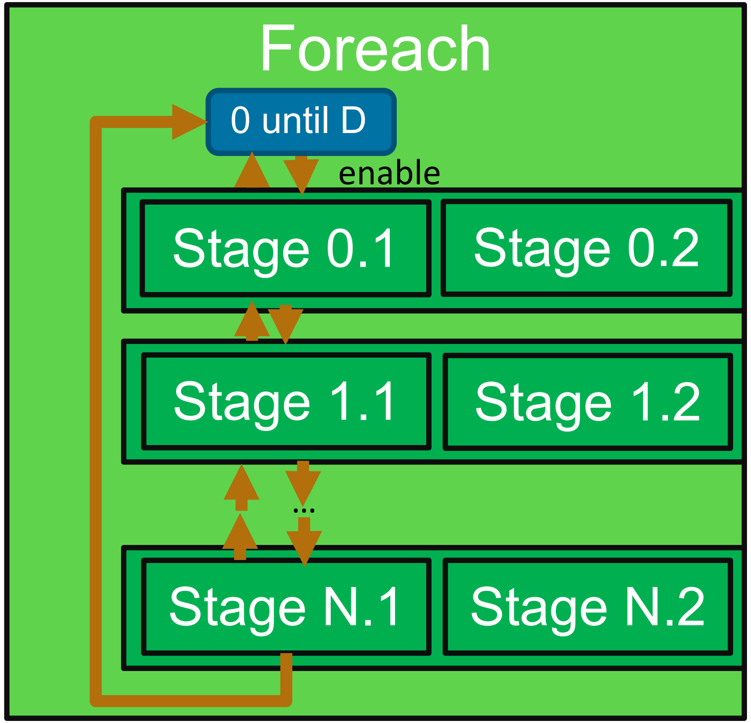
Foreach: Parallelization



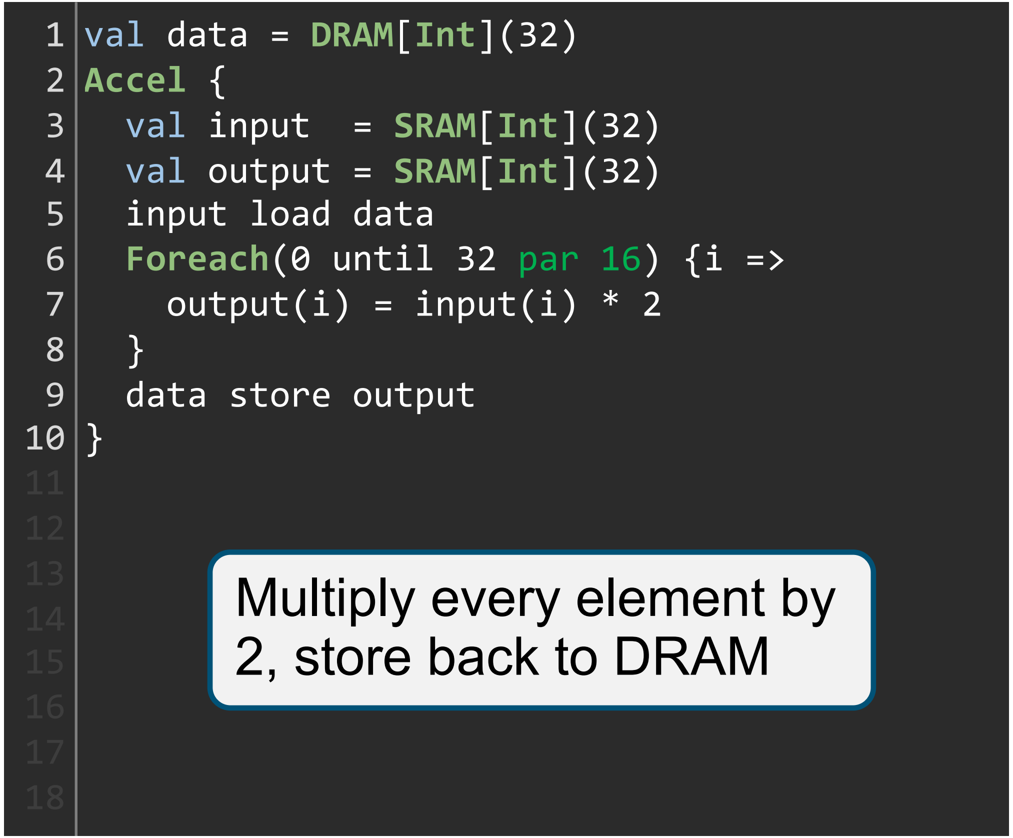
```
val D = ArgIn[Int]
Accel {
  Foreach(0 until D par 2) {i =>
    …

  }
  …
}
```

Parallelizes the pipeline by duplicating the body

Spatial also handles memory banking for you!

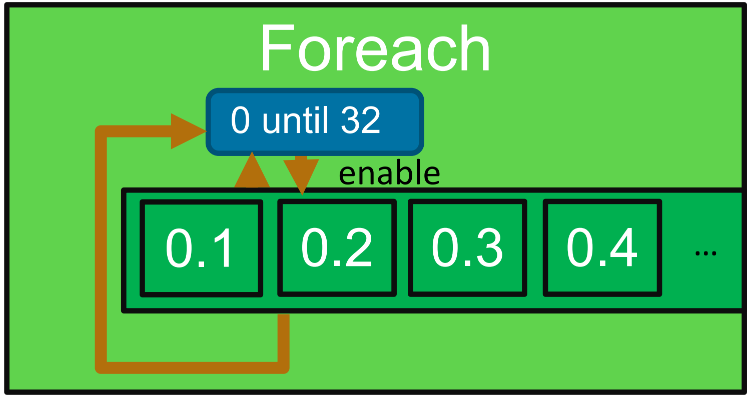


Foreach: Example

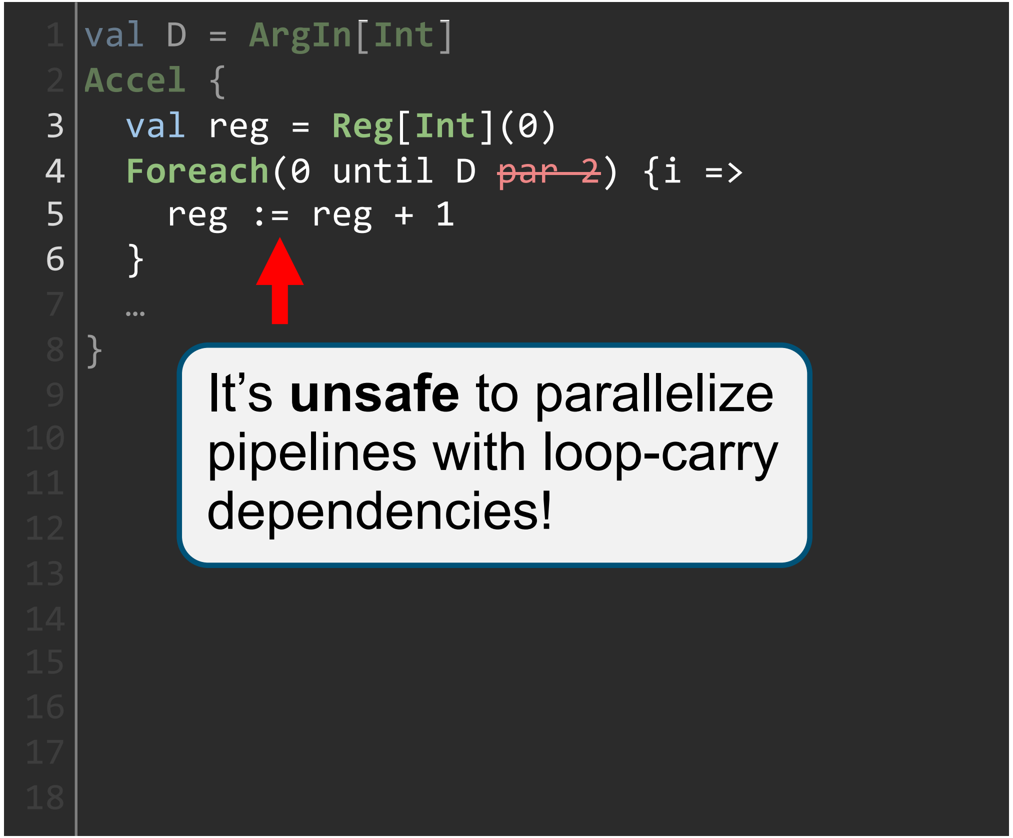


```
val data = DRAM[Int](32)
Accel {
  val input  = SRAM[Int](32)
  val output = SRAM[Int](32)
  input load data
  Foreach(0 until 32 par 16) {i =>
    output(i) = input(i) * 2
  }
  data store output
}
```

Multiply every element by 2, store back to DRAM

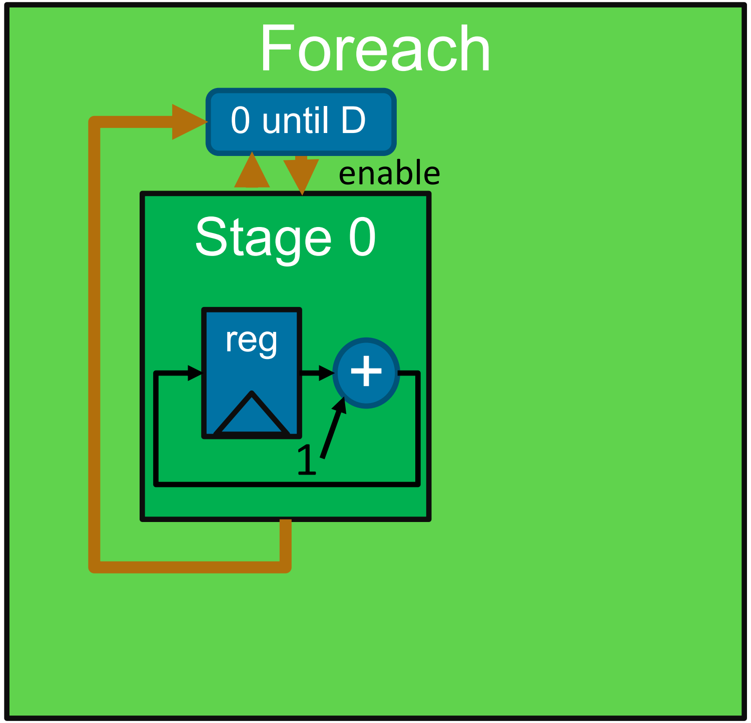


Foreach: Illegal Parallelization Cases

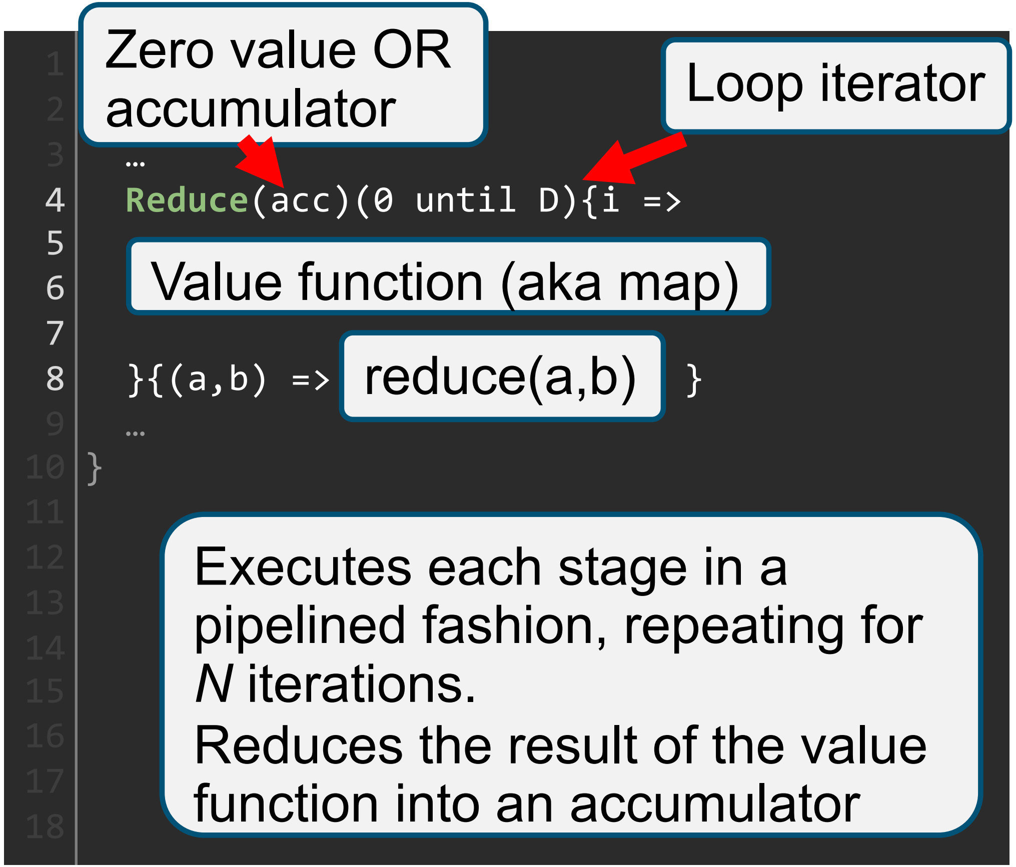


```
val D = ArgIn[Int]
Accel {
  val reg = Reg[Int](0)
  Foreach(0 until D ~~par 2~~) {i =>
    reg := reg + 1
  }
  …
}
```

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



Reduce

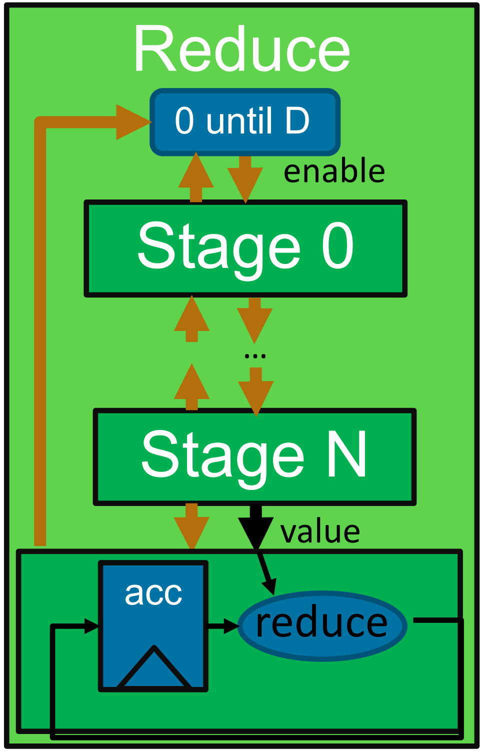


Zero value OR accumulator

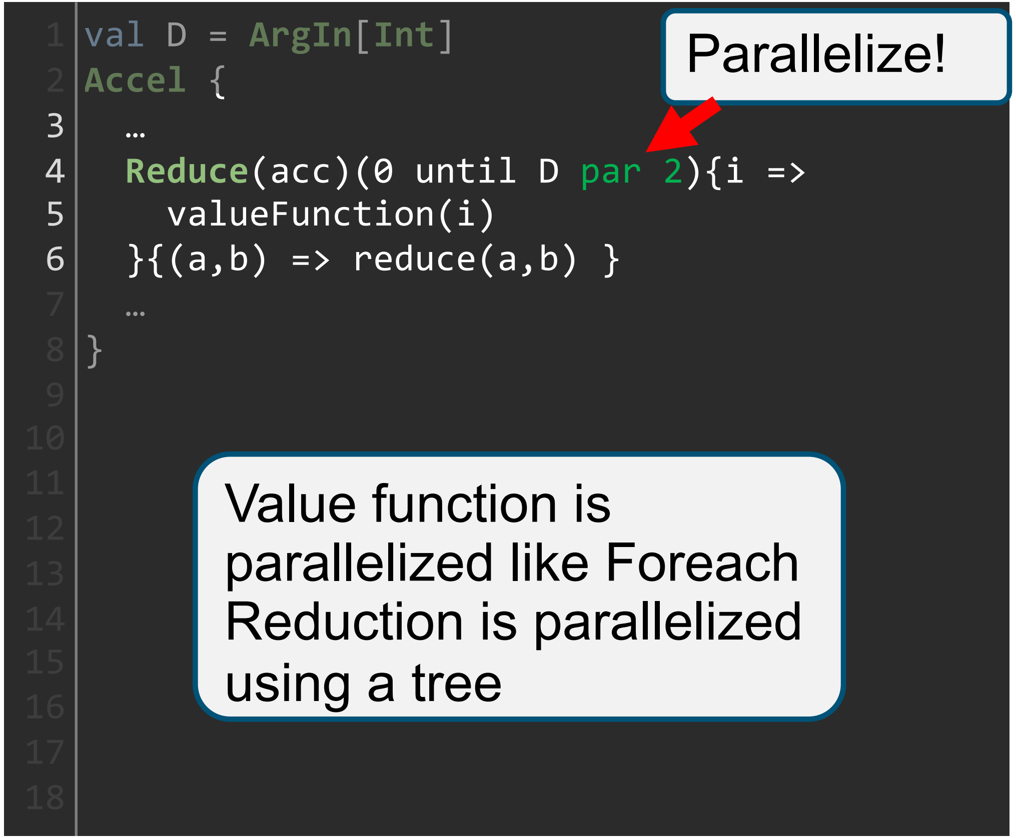
Loop iterator

```
  …
  Reduce(acc)(0 until D){i =>

    Value function (aka map)

  }{(a,b) => reduce(a,b) }
  …
}
```

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



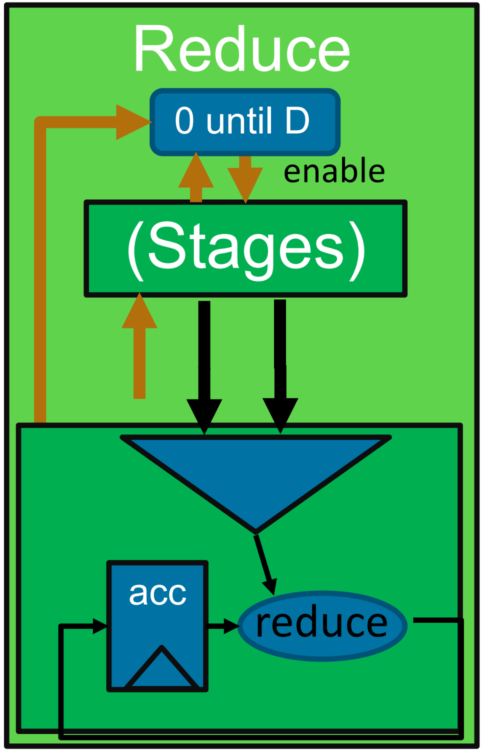
Reduce: Parallelization



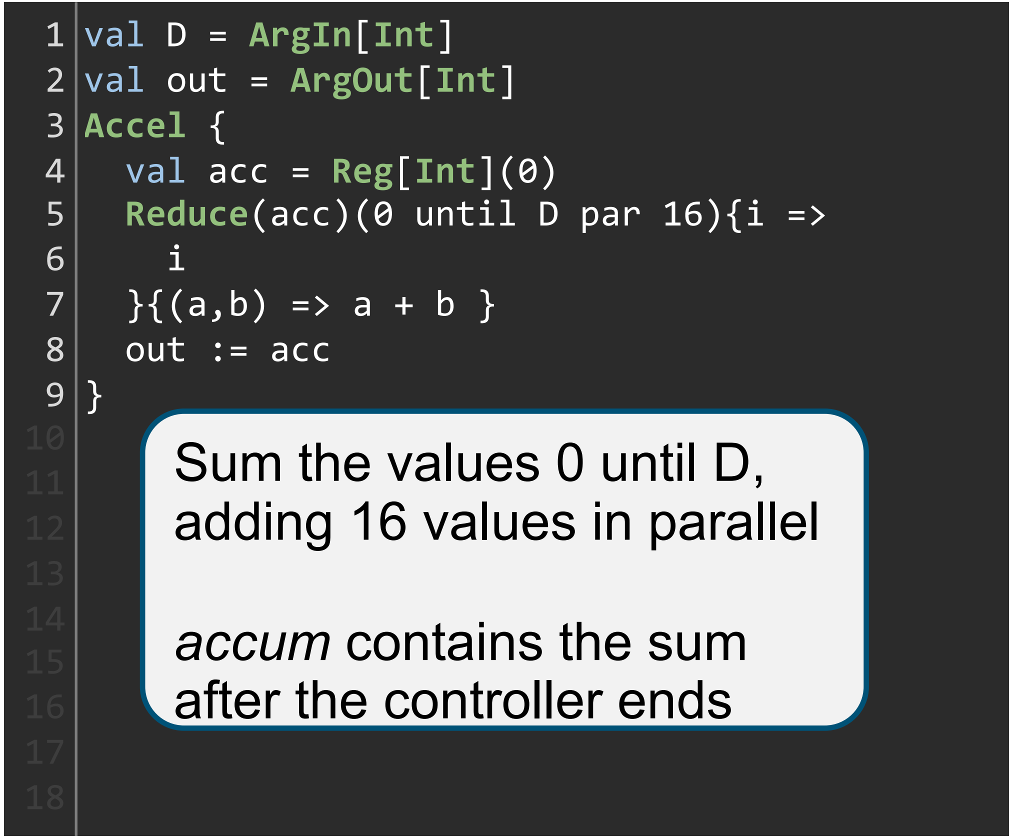
```
val D = ArgIn[Int]
Accel {
  …
  Reduce(acc)(0 until D par 2){i =>
    valueFunction(i)
  }{(a,b) => reduce(a,b) }
  …
}
```

Parallelize!

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



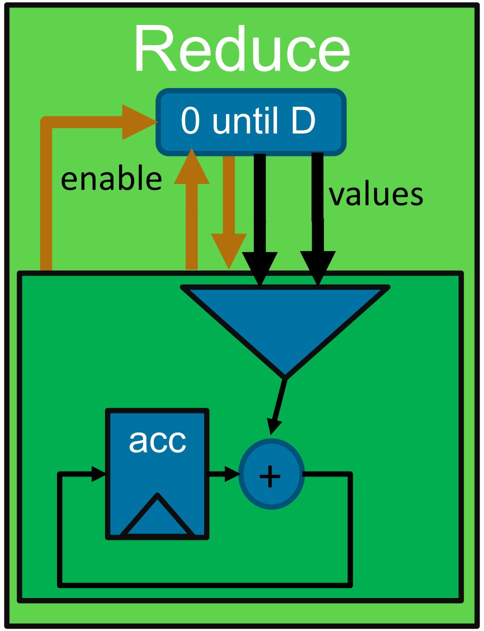
Reduce: Example



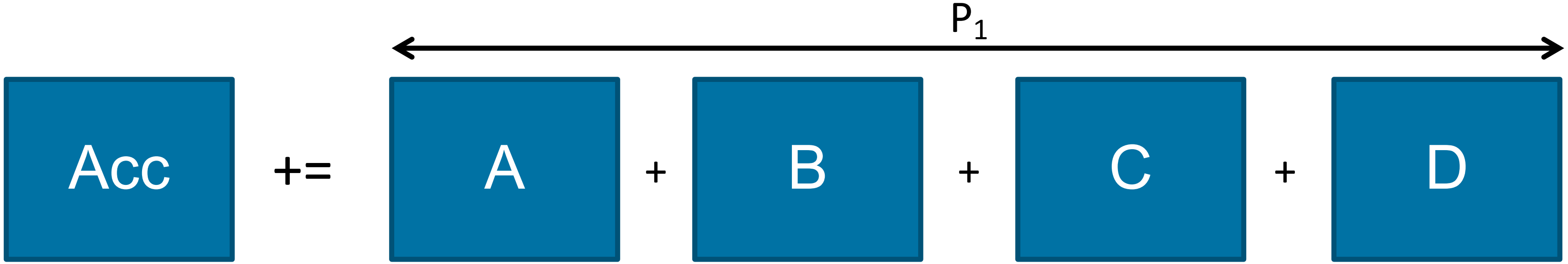
```
val D = ArgIn[Int]
val out = ArgOut[Int]
Accel {
  val acc = Reg[Int](0)
  Reduce(acc)(0 until D par 16){i =>
    i
  }{(a,b) => a + b }
  out := acc
}
```

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

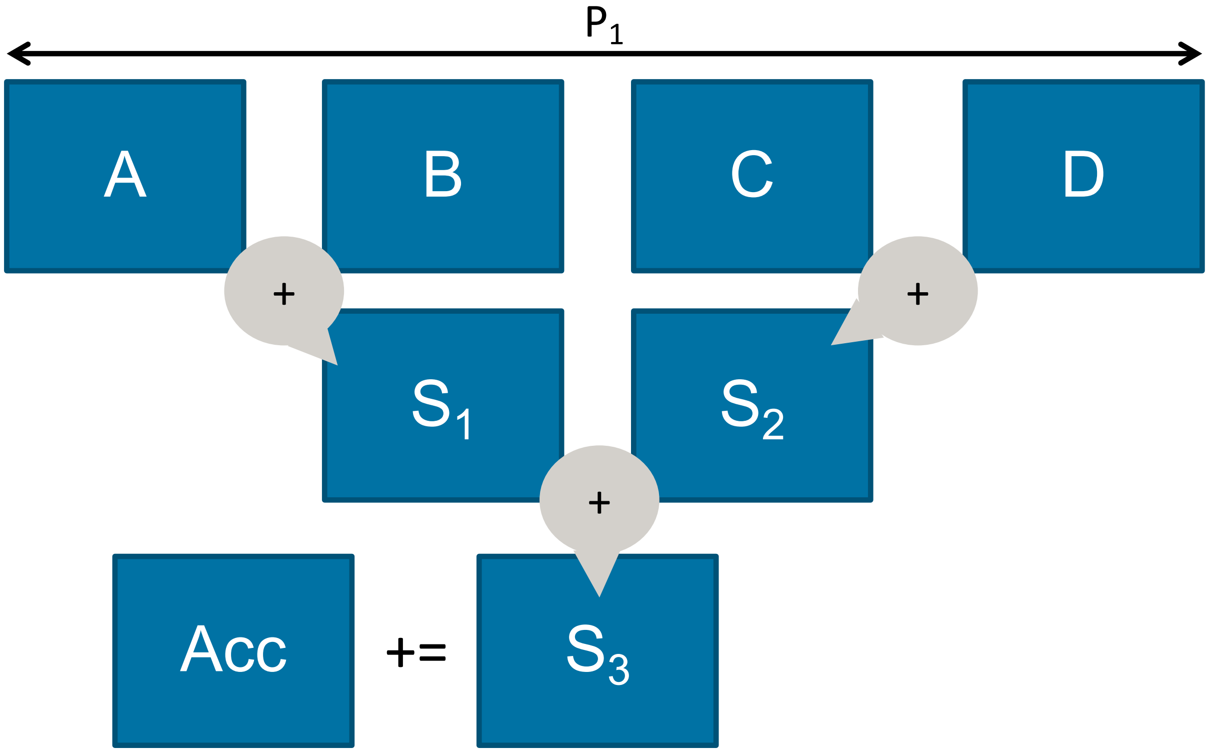
accum contains the sum after the controller ends



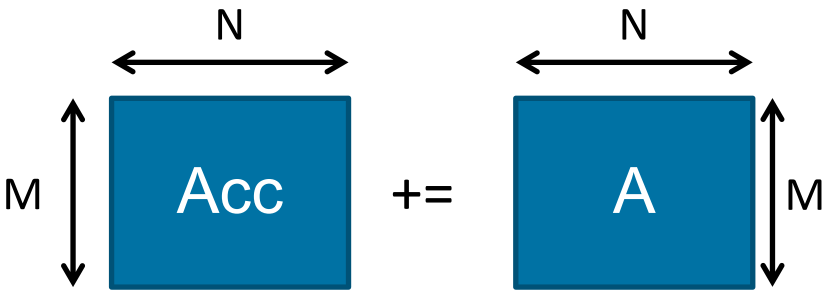
Accumulation



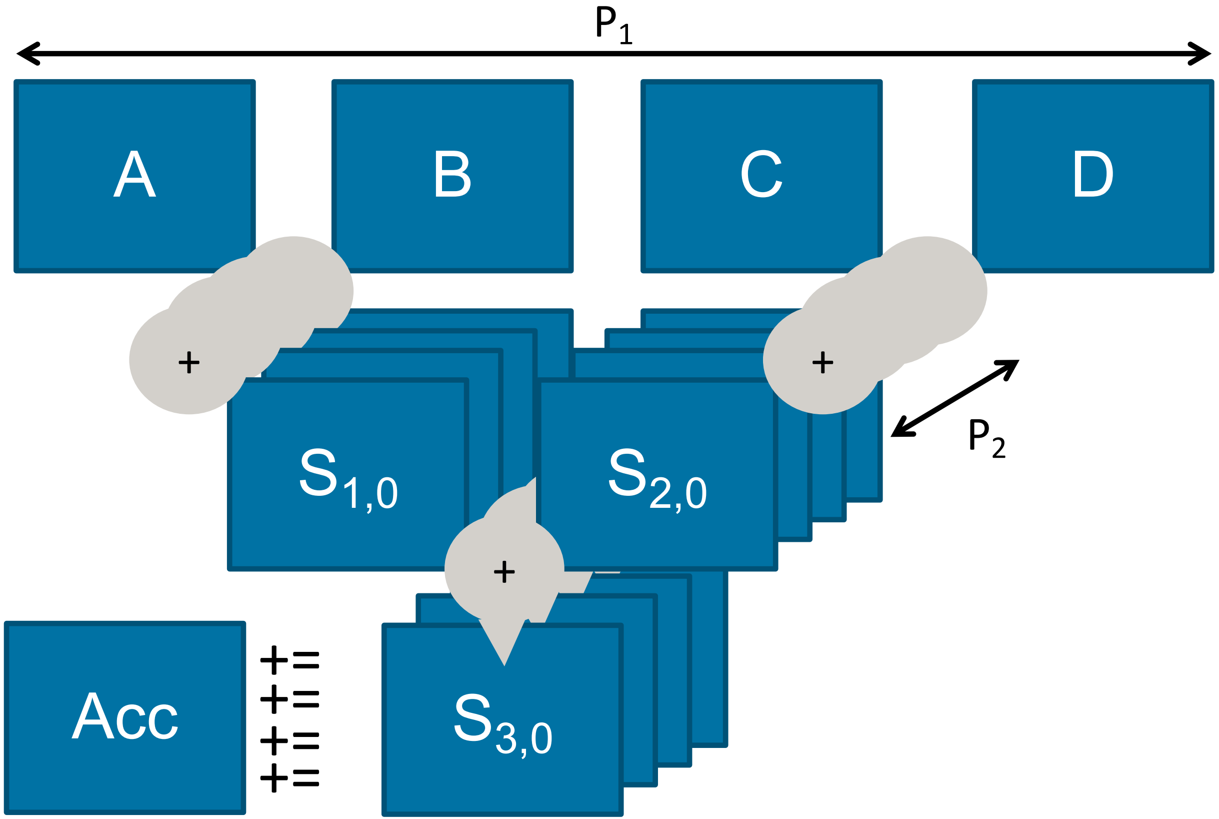
Reduce (Visualization)



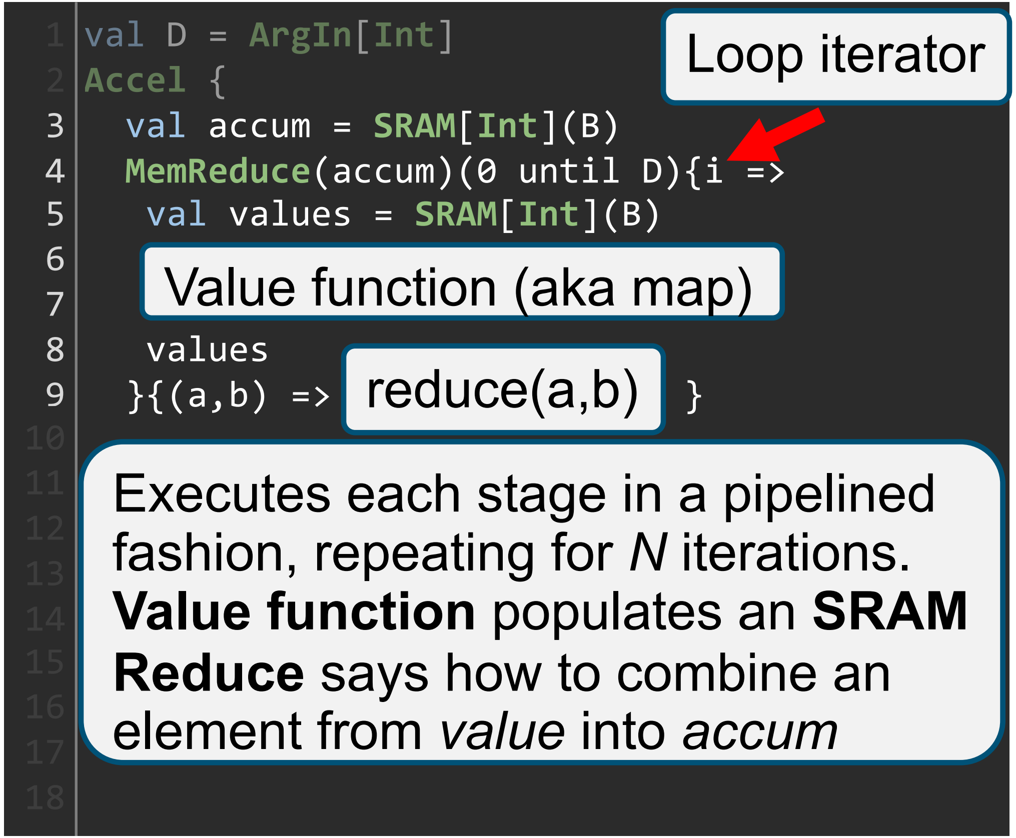
MemReduce



MemReduce



MemReduce



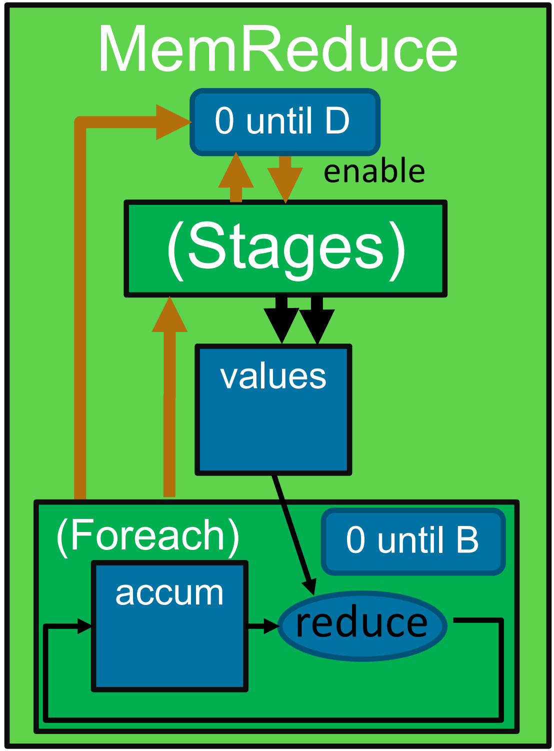
```
val D = ArgIn[Int]
Accel {
  val accum = SRAM[Int](B)
  MemReduce(accum)(0 until D){i =>
   val values = SRAM[Int](B)


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

Loop iterator

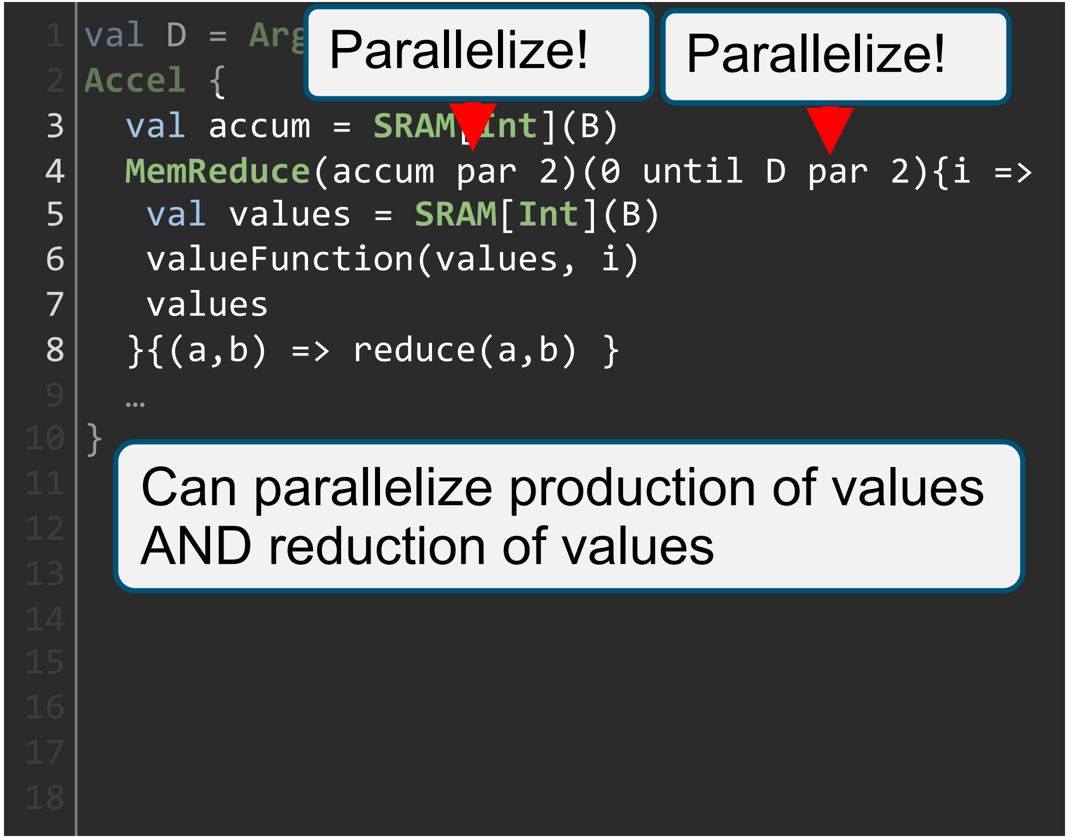
Value function (aka map)

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

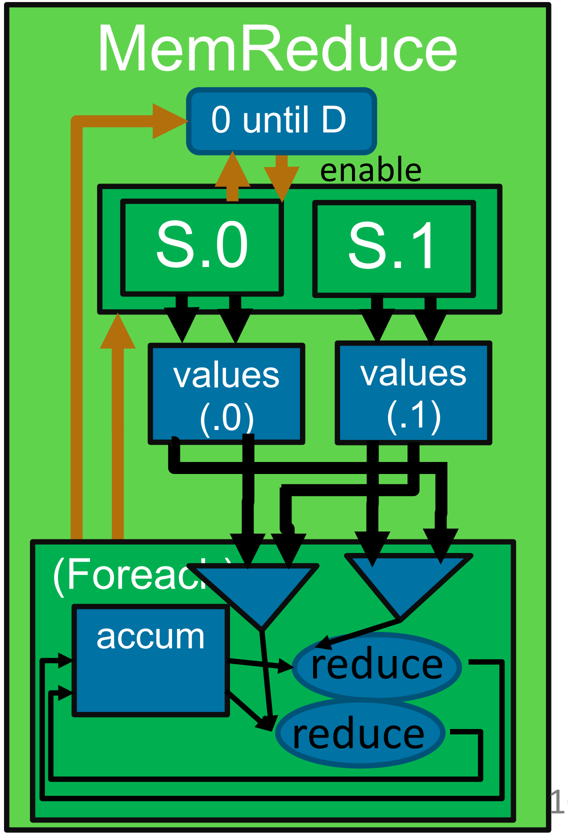


```
val D = Arg
Accel {
  val accum = SRAM[Int](B)
  MemReduce(accum par 2)(0 until D par 2){i =>
   val values = SRAM[Int](B)
   valueFunction(values, i)
   values
  }{(a,b) => reduce(a,b) }
  …
}
```

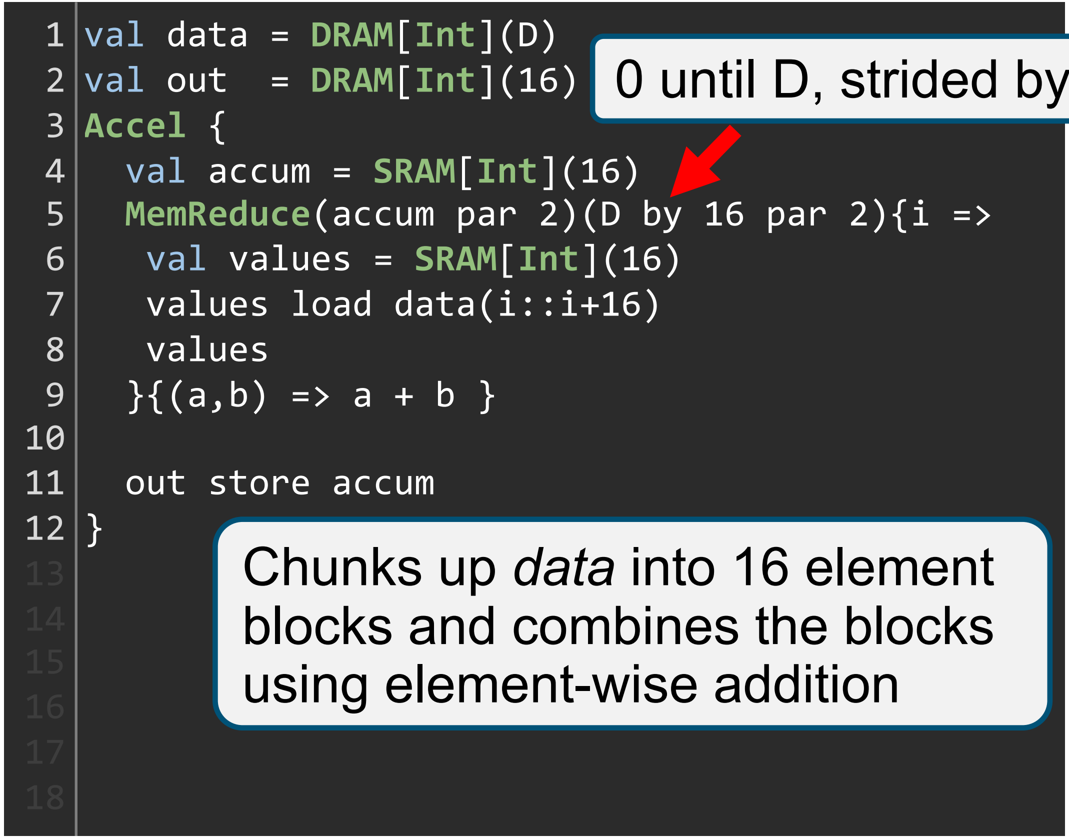
Parallelize!

Parallelize!

Can parallelize production of values AND reduction of values



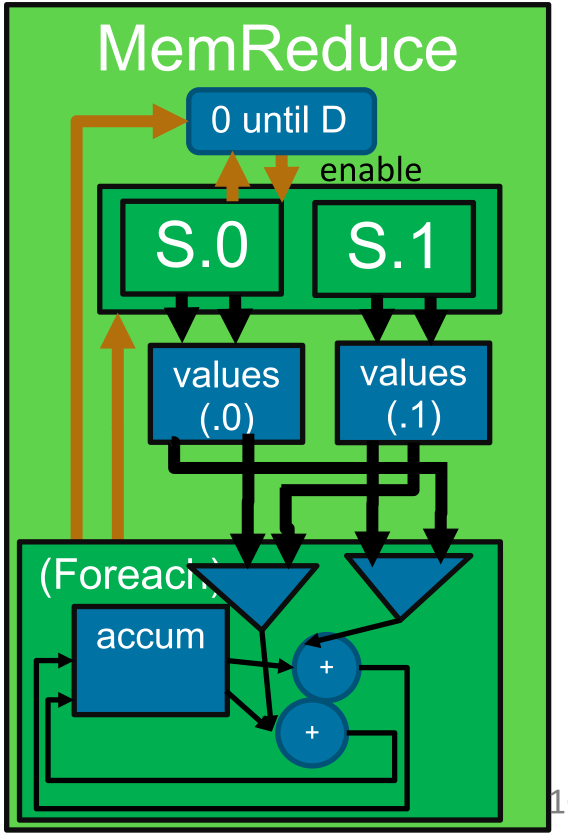
MemReduce: Example



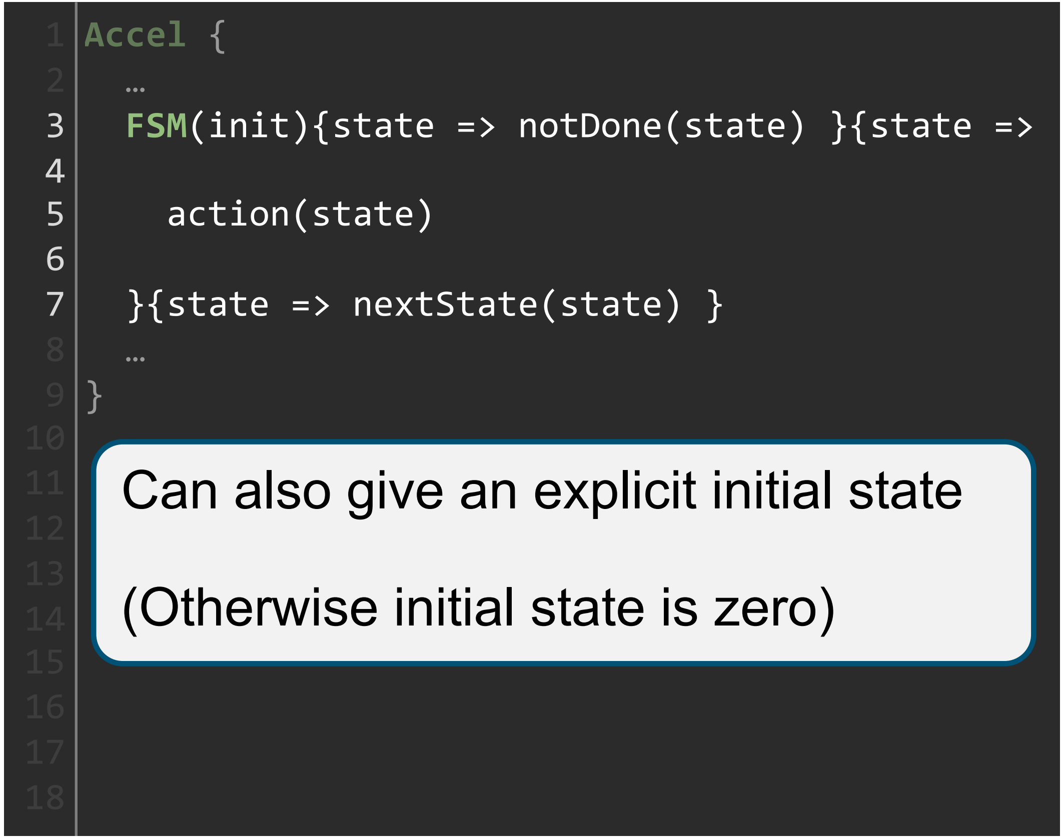
```
val data = DRAM[Int](D)
val out  = DRAM[Int](16)
Accel {
  val accum = SRAM[Int](16)
  MemReduce(accum par 2)(D by 16 par 2){i =>
   val values = SRAM[Int](16)
   values load data(i::i+16)
   values
  }{(a,b) => a + b }

  out store accum
}
```

0 until D, strided by 16

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



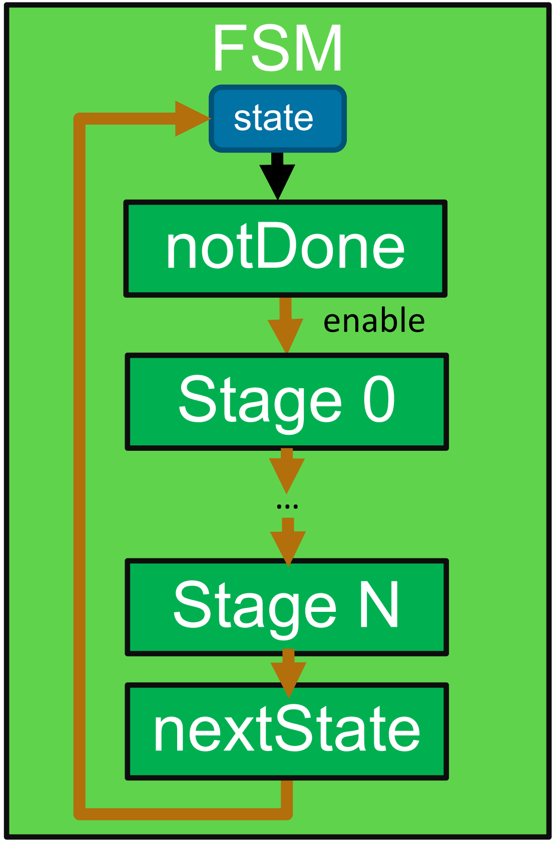
FSM (State Machine)



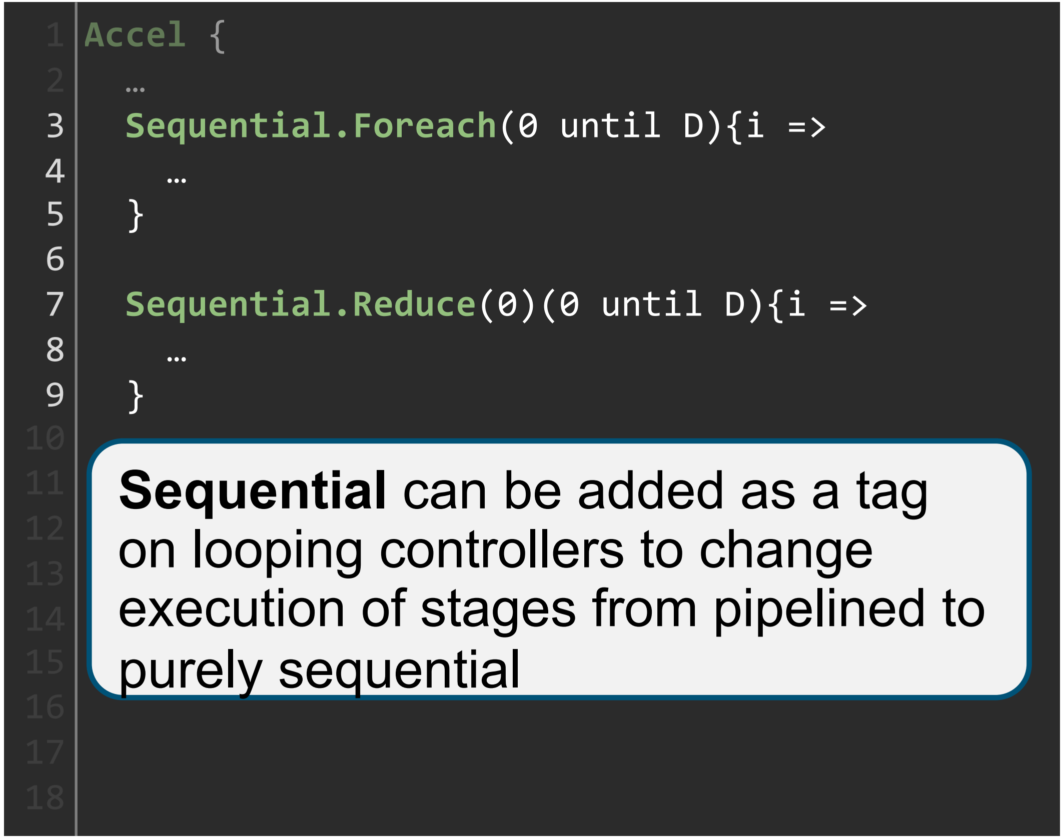
```
Accel {
  …
  FSM(init){state => notDone(state) }{state =>

    action(state)

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

Can also give an explicit initial state

(Otherwise initial state is zero)



Controller Tags

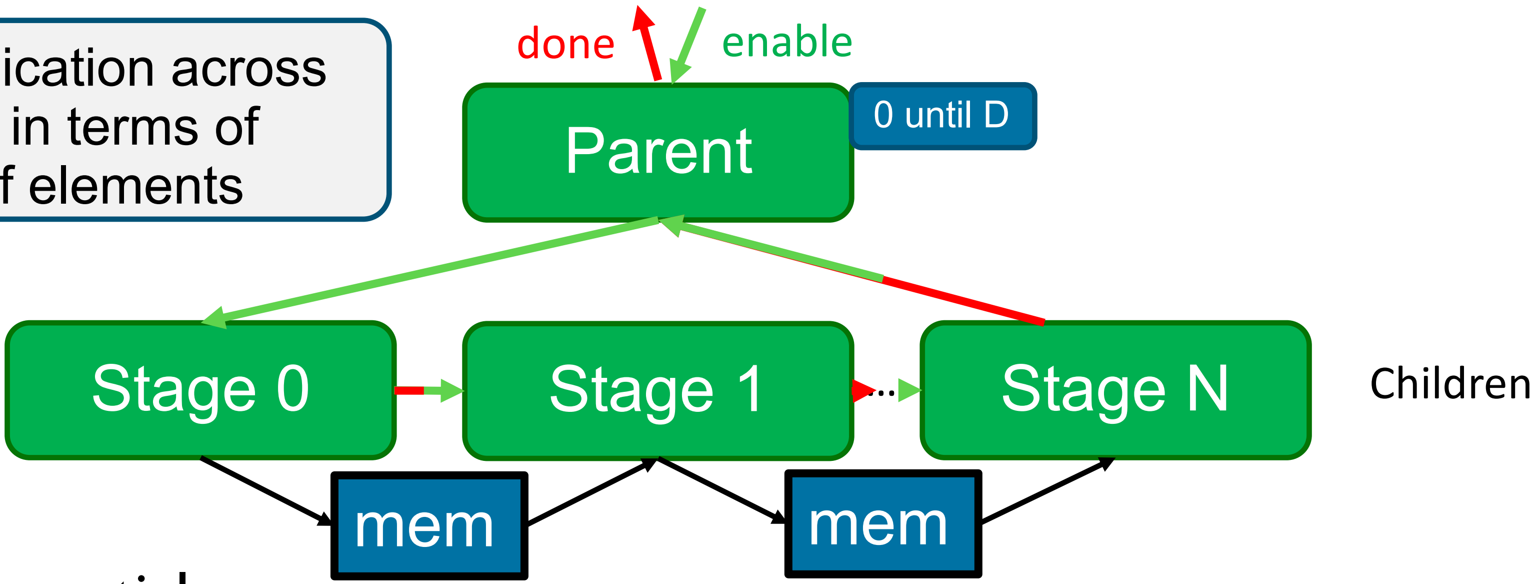


```
Accel {
  …
  Sequential.Foreach(0 until D){i =>
    …
  }

  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

Sequential Execution

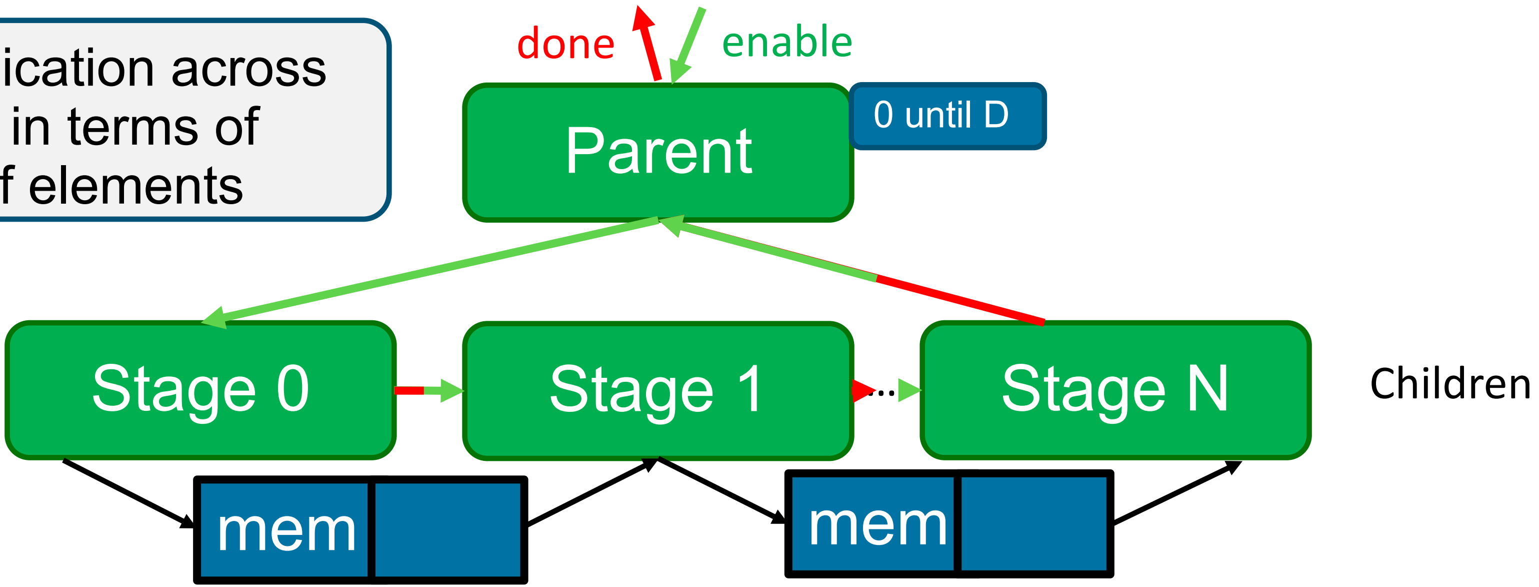
Communication across stages is in terms of **blocks** of elements



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

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)