Spatial Documentation

Release 0.1

Stanford PPL

Oct 09, 2016

Contents:

CHAPTER

ONE

INTRODUCTION

Spatial is a domain-specific language for describing hardware datapaths. A Spatial program describes a dataflow graph consisting of various kinds of nodes connected to each other by data dependencies. Each node in a Spatial program corresponds to a architectural template. Spatial is represented in-memory as a parameterized, hierarchical dataflow graph.

Templates in Spatial capture parallelism, locality, and access pattern information at multiple levels. This dramatically simplifies coarse-grained pipelining and enables us to explicitly capture and represent a large space of designs which other tools cannot capture. Every template is parameterized. A specific hardware design point is instantiated from a Spatial description by instantiating all the templates in the design with concrete parameter values passed to the program. Spatial heavily uses metaprogramming, so these values are passed in as arguments to the Spatial program. The generated design instance is represented internally as a graph that can be analyzed to provide estimates of metrics such as area and cycle count. The parameters used to create the design instance can be automatically generated by a design space exploration tool.

This document was auto-generated using Sphinx. For corrections, post an issue on GitHub Issues .

CHAPTER

TWO

TYPE CLASSES

2.1 Arith

<auto-generated stub>

2.1.1 Infix methods

```
def divide(x: T, y: T): T
```

```
def minus(x: T, y: T): T
```

```
def plus(x: T, y: T): T
```

```
def times(x: T, y: T): T
```

2.2 Coll

<auto-generated stub>

2.2.1 Infix methods

```
def empty(): T
```

2.3 Mem

<auto-generated stub>

2.3.1 Infix methods

`def empty(x: C[T])(implicit ev0: Num[T]): C[T]`

`def flatIdx(x: C[T], y: Indices): Index`

`def iterator(x: C[T], y: List[Int]): CounterChain`

`def ld(x: C[T], y: Index): T`

`def st(x: C[T], y: Index, z: T): Unit`

`def zeroIdx(x: C[T]): Indices`

2.4 Num

<auto-generated stub>

2.4.1 Infix methods

`def zero(): T`

2.5 Order

<auto-generated stub>

2.5.1 Infix methods

`def equals(x: T, y: T): Bit`

`def greaterThan(x: T, y: T): Bit`

`def greaterThanOrEqual(x: T, y: T): Bit`

`def lessThan(x: T, y: T): Bit`

`def lessThanOrEqual(x: T, y: T): Bit`

`def notEqual(x: T, y: T): Bit`

CHAPTER

THREE

DATA STRUCTURES

3.1 BRAM

BRAMs are on-chip scratchpads with fixed size. BRAMs can be specified as multi-dimensional, but the underlying addressing in hardware is always flat. The contents of BRAMs are currently persistent across loop iterations, even when they are declared in an inner scope. BRAMs can have an arbitrary number of readers but only one writer. This writer may be an element-based store or a load from an OffChipMem.

3.1.1 Static methods

```
def apply(dims: Index*)(implicit ev0: Num[T]): BRAM[T]
```

Creates a BRAM with given dimensions. Dimensions must be statically known signed integers (constants or parameters).

3.1.2 Infix methods

```
def :=(tile: Tile[T])(implicit ev0: Num[T]): Unit
```

Stores a Tile of an OffChipMem to this BRAM.

```
def :=(tile: SparseTile[T]): Unit
```

Gathers the values from the supplied SparseTile into this BRAM

```
def apply(ii: Index*): T
```

Creates a read from this BRAM at the given multi-dimensional address. Number of indices given can either be 1 or the same as the number of dimensions that the BRAM was declared with.

- **ii** - multi-dimensional address

```
def update(i: Index, x: T): Unit
```

Creates a write to this BRAM at the given 1D address.

- **i** - 1D address
- **x** - element to be stored to BRAM

`def update(i: Index, j: Index, x: T): Unit`

Creates a write to this BRAM at the given 2D address. The BRAM must have initially been declared as 2D.

- **i** - row index
- **j** - column index
- **x** - element to be stored to BRAM

`def update(i: Index, j: Index, k: Index, x: T): Unit`

Creates a write to this BRAM at the given 3D address. The BRAM must have initially been declared as 3D.

- **i** - row index
- **j** - column index
- **k** - page index
- **x** - element to be stored to BRAM

`def update(ii: List[Index], x: T): Unit`

Creates a write to this BRAM at the given multi-dimensional address. The number of indices given can either be 1 or the same as the number of dimensions that the BRAM was declared with.

- **ii** - multi-dimensional index
- **x** - element to be stored to BRAM

3.2 CAM

CAMs (content addressable memories) are used for associative key-value stores.

3.2.1 Static methods

`def apply(size: Index)(implicit ev0: Num[K],ev1: Num[V]): CAM[K,V]`

Creates a CAM with given size. Size must be a statically known signed integer (constant or parameter).

3.2.2 Infix methods

`def apply(key: K): V`

Creates a key-value read for this CAM

`def update(key: K, value: V): Unit`

Creates a key-value store into this CAM

- **key** - the key to be used to address this value
- **value** - the value to be stored

3.3 Cache

Caches are on-chip caches for a specific off-chip memory/data structure. Caches allow loading with multi-dimentional address, whose dimensions are inherited from the cached off-chip memories. The multi-dimentional address is converted to a single flat address in hardware. The addressing scheme is word-based flat indexing of the offchip memory. Cache allows loading of a single element at a time. During a miss, a cache automatically loads from its off-chip memory and stalls the pipeline, and resumes pipeline when loading is complete.

3.3.1 Static methods

`def apply(name: OffChipMem[T])(implicit ev0: Num[T]): Cache[T]`

Creates a Cache with target OffChipMem. Dimensions is inherited from OffChipMem

3.3.2 Infix methods

`def apply(ii: Index, z: Index*): T`

Creates a read from this Cache at the given multi-dimensional address. Number of indices given can either be 1 or the same as the number of dimensions that the cached OffChipMem was declared with. During a miss, the innermost pipeline that contains current load will be stalled until data is loaded from offchip

- **ii** - multi-dimensional address

`def update(i: Index, x: T): Unit`

Creates a write to this Cache at the given 1D address. During a miss, the innermost pipeline that contains current load will be stalled until data is loaded from offchip

- **i** - 1D address
- **x** - element to be stored to Cache

`def update(i: Index, j: Index, x: T): Unit`

Creates a write to this Cache at the given 2D address. The cached OffChipMem must have initially been declared as 2D. During a miss, the innermost pipeline that contains current load will be stalled until data is loaded from offchip

- **i** - row index
- **j** - column index
- **x** - element to be stored to Cache

`def update(i: Index, j: Index, k: Index, x: T): Unit`

Creates a write to this Cache at the given 3D address. The cached OffChipMem must have initially been declared as 3D. During a miss, the innermost pipeline that contains current load will be stalled until data is loaded from offchip

- **i** - row index
- **j** - column index
- **k** - page index
- **x** - element to be stored to Cache

```
def update(y: Seq[Index], z: T): Unit
```

3.4 Counter

Counter is a single hardware counter with an associated minimum, maximum, step size, and parallelization factor. By default, the parallelization factor is assumed to be a design parameter. Counters can be chained together using CounterChain, but this is typically done implicitly when creating controllers.

3.4.1 Static methods

```
def apply(max: Index): Counter
```

Creates a Counter with min of 0, given max, and step size of 1

```
def apply(min: Index, max: Index): Counter
```

Creates a Counter with given min and max, and step size of 1

```
def apply(min: Index, max: Index, step: Index): Counter
```

Creates a Counter with given min, max and step size

```
def apply(min: Index, max: Index, step: Index, par: Int): Counter
```

Creates a Counter with given min, max, step size, and parallelization factor

3.5 CounterChain

CounterChain describes a set of chained hardware counters, where a given counter increments only when the counter below it wraps around. Order is specified as outermost to innermost.

3.5.1 Static methods

```
def apply(x: Counter*): CounterChain
```

Creates a chain of counters. Order is specified as outermost to innermost

3.6 FIFO

FIFOs are on-chip scratchpads with additional control logic for address-less push/pop operations. FIFOs can have an arbitrary number of readers but only one writer.

3.6.1 Static methods

`def apply(size: Index)(implicit ev0: Num[T]): FIFO[T]`

Creates a FIFO with given size. Size must be a statically known signed integer (constant or parameter).

3.6.2 Infix methods

`def :=(tile: Tile[T])(implicit ev0: Num[T]): Unit`

Streams a Tile of an OffChipMem to this FIFO.

`def count(): Index`

`def pop(): T`

Creates a pop (read) port to this FIFO

`def push(value: T): Unit`

Creates a push (write) port to this FIFO

- **value** - the value to be pushed into the FIFO

`def push(value: T, wren: Bit): Unit`

Creates a push (write) port to this FIFO with a write enable

- **value** - the value to be pushed into the FIFO
- **wren** - write enable

3.7 LoopRange

<auto-generated stub>

3.7.1 Infix methods

`def by(step: Index): LoopRange`

Changes given LoopRange's step to specified value

`def foreach(y: (Index) => Unit): Unit`

3.7.2 Implicit methods

`def rangeToCounter(x: LoopRange): Counter`

3.8 OffChipMem

OffChipMems are pointers to locations in the accelerators main memory to dense multi-dimensional arrays. They are the primary form of communication of data between the host and the accelerator. Data may be loaded to and from the accelerator in contiguous chunks (Tiles), by random access addresses, or by gather operations (SparseTiles).

3.8.1 Static methods

`def apply(dims: Index*)(implicit ev0: Num[T]): OffChipMem[T]`

Creates a reference to a multi-dimensional array in main memory with given dimensions

3.8.2 Infix methods

`def apply(cols: Range): Tile[T]`

Creates a reference to a 1D Tile of this 1D OffChipMem which can be loaded into local memory.

`def apply(rows: Range, cols: Range): Tile[T]`

Creates a reference to a 2D Tile of this 2D OffChipMem which can be loaded into local memory.

`def apply(rows: Range, cols: Range, pages: Range): Tile[T]`

Creates a reference to a 3D Tile of this 3D OffChipMem which can be loaded into local memory.

`def apply(row: Index, cols: Range): Tile[T]`

Creates a reference to a 1D row Tile of this 2D OffChipMem

`def apply(rows: Range, col: Index): Tile[T]`

Creates a reference to a 1D column Tile of this 2D OffChipMem

---

`def apply(row: Index, cols: Range, pages: Range): Tile[T]`

Creates a reference to a 2D column/page Tile of this 3D OffChipMem

---

`def apply(rows: Range, col: Index, pages: Range): Tile[T]`

Creates a reference to a 2D row/page Tile of this 3D OffChipMem

---

`def apply(rows: Range, cols: Range, page: Index): Tile[T]`

Creates a reference to a 2D row/column Tile of this 3D OffChipMem

---

`def apply(row: Index, col: Index, pages: Range): Tile[T]`

Creates a reference to a 1D page Tile of this 3D OffChipMem

```
def apply(row: Index, cols: Range, page: Index): Tile[T]
```

Creates a reference to a 1D column Tile of this 3D OffChipMem

---

```
def apply(rows: Range, col: Index, page: Index): Tile[T]
```

Creates a reference to a 1D row Tile of this 3D OffChipMem

```
def apply(addrs: BRAM[Index], size: Index, par: Int): SparseTile[T]
```

Sets up a sparse gather from this OffChipMem using *size* addresses from the supplied BRAM

- **addrs** - BRAM with addresses to load
- **size** - the number of addresses
- **par** - the number of elements to load in parallel

```
def apply(addrs: BRAM[Index], par: Int): SparseTile[T]
```

Sets up a sparse gather from this OffChipMem using *size* addresses from the supplied BRAM

- **addrs** - BRAM with addresses to load
- **par** - the number of elements to load in parallel

```
def apply(addrs: BRAM[Index], size: Index): SparseTile[T]
```

Sets up a sparse gather from this OffChipMem using *size* addresses from the supplied BRAM

- **addrs** - BRAM with addresses to load
- **size** - the number of addresses

```
def apply(addrs: BRAM[Index]): SparseTile[T]
```

Sets up a sparse gather from this OffChipMem using all addresses from the supplied BRAM

- **addrs** - BRAM with addresses to load

3.9 Reg

Reg defines a hardware register used to hold a scalar value. Regs have an optional name (primarily used for debugging) and reset value. The default reset value for a Reg is the numeric zero value for it's specified type. Regs can have an arbitrary number of readers but can only have one writer. By default, Regs are reset based upon the controller that they are defined within. A Reg defined within a Pipe, for example, is reset at the beginning of each iteration of that Pipe.

3.9.1 Static methods

`def apply()(implicit ev0: Num[T]): Reg[T]`

Creates a register with type T

`def apply(reset: Int)(implicit ev0: Num[T]): Reg[T]`

Creates an unnamed register with type T and given reset value

---

`def apply(reset: Long)(implicit ev0: Num[T]): Reg[T]`

Creates an unnamed register with type T and given reset value

---

`def apply(reset: Float)(implicit ev0: Num[T]): Reg[T]`

Creates an unnamed register with type T and given reset value

---

`def apply(reset: Double)(implicit ev0: Num[T]): Reg[T]`

Creates an unnamed register with type T and given reset value

3.9.2 Infix methods

`def :=(x: T): Unit`

Creates a writer to this Reg. Note that Regs and ArgOuts can only have one writer, while ArgIns cannot have any

---

`def value(): T`

Reads the current value of this register

3.9.3 Implicit methods

`def regBitToBit(x: Reg[Bit]): Bit`

Enables implicit reading from bit type Regs

`def regFixToFix(x: Reg[FixPt[S,I,F]]): FixPt[S,I,F]`

Enables implicit reading from fixed point type Regs

`def regFltToFlt(x: Reg[FltPt[G,E]]): FltPt[G,E]`

Enables implicit reading from floating point type Regs

3.9.4 Related methods

```
def ArgIn()(implicit ev0: Num[T]): Reg[T]
```

Creates an unnamed input argument from the host CPU

```
def ArgOut()(implicit ev0: Num[T]): Reg[T]
```

Creats an unnamed output argument to the host CPU

3.10 SparseTile

A SparseTile describes a sparse section of an OffChipMem which can be loaded onto the accelerator using a gather operation, or which can be updated using a scatter operation.

3.10.1 Infix methods

```
def :=(bram: BRAM[T]): Unit
```

Creates a store from the given on-chip BRAM to this SparseTile of off-chip memory

3.10.2 Related methods

```
def copySparse(tile: SparseTile[T], local: BRAM[T], isScatter: Boolean): Unit
```

3.11 Tile

A Tile describes a continguous slice of an OffChipMem which can be loaded onto the accelerator for processing or which can be updated with results once computation is complete.

3.11.1 Infix methods

```
def :=(bram: BRAM[T])(implicit ev0: Num[T]): Unit
```

Creates a store from the given on-chip BRAM to this Tile of off-chip memory

---

```
def :=(y: FIFO[T])(implicit ev0: Num[T]): Unit
```

3.11.2 Related methods

```
def streamTile(tile: Tile[T], fifo: FIFO[T], store: Boolean)(implicit ev0: Num[T]): Unit
```

3.12 Tup2

<auto-generated stub>

3.12.1 Infix methods

`def _1(): A`

`def _2(): B`

`def toString(): String`

3.12.2 Related methods

`def pack(t: Tuple2[A,B]): Tup2[A,B]`

`def pack(x: Tuple2[A,B]): Tup2[A,B]`

`def pack(x: Tuple2[A,B]): Tup2[A,B]`

`def pack(x: Tuple2[A,B]): Tup2[A,B]`

`def unpack(t: Tup2[A,B]): Tuple2[A,B]`

3.13 Tup3

<auto-generated stub>

3.13.1 Infix methods

`def _1(): A`

`def _2(): B`

`def _3(): C`

`def toString(): String`

3.13.2 Related methods

`def pack(t: Tuple3[A,B,C]): Tup3[A,B,C]`

`def unpack(t: Tup3[A,B,C]): Tuple3[A,B,C]`

3.14 Tup4

<auto-generated stub>

3.14.1 Infix methods

`def _1(): A`

`def _2(): B`

`def _3(): C`

`def _4(): D`

`def toString():` *String*

3.14.2 Related methods

`def pack(t: Tuple4[A,B,C,D]):` *Tup4*`[A,B,C,D]`

`def unpack(t:` *Tup4*`[A,B,C,D]): Tuple4[A,B,C,D]`

3.15 Tup5

<auto-generated stub>

3.15.1 Infix methods

`def _1(): A`

`def _2(): B`

`def _3(): C`

`def _4(): D`

`def _5(): E`

`def toString():` *String*

3.15.2 Related methods

`def pack(t: Tuple5[A,B,C,D,E]): Tup5[A,B,C,D,E]`

`def unpack(t: Tup5[A,B,C,D,E]): Tuple5[A,B,C,D,E]`

3.16 Tup6

<auto-generated stub>

3.16.1 Infix methods

`def _1(): A`

`def _2(): B`

`def _3(): C`

`def _4(): D`

`def _5(): E`

`def _6(): F`

`def toString(): String`

3.16.2 Related methods

`def pack(t: Tuple6[A,B,C,D,E,F]): Tup6[A,B,C,D,E,F]`

`def unpack(t: Tup6[A,B,C,D,E,F]): Tuple6[A,B,C,D,E,F]`

3.17 Tup7

<auto-generated stub>

3.17.1 Infix methods

`def _1(): A`

`def _2(): B`

`def _3(): C`

`def _4(): D`

`def _5(): E`

`def _6(): F`

`def _7(): G`

`def toString(): String`

3.17.2 Related methods

`def pack(t: Tuple7[A,B,C,D,E,F,G]): Tup7[A,B,C,D,E,F,G]`

`def unpack(t: Tup7[A,B,C,D,E,F,G]): Tuple7[A,B,C,D,E,F,G]`

3.18 Tup8

<auto-generated stub>

3.18.1 Infix methods

`def _1(): A`

`def _2(): B`

`def _3(): C`

`def _4(): D`

---

def _5(): E

def _6(): F

def _7(): G

def _8(): H

def toString(): *String*

3.18.2 Related methods

def pack(t: Tuple8[A,B,C,D,E,F,G,H]): *Tup8*[A,B,C,D,E,F,G,H]

def unpack(t: *Tup8*[A,B,C,D,E,F,G,H]): Tuple8[A,B,C,D,E,F,G,H]

3.19 Tup9

<auto-generated stub>

3.19.1 Infix methods

def _1(): A

def _2(): B

def _3(): C

def _4(): D

def _5(): E

def _6(): F

def _7(): G

def _8(): H

def _9(): I

def toString(): *String*

3.19.2 Related methods

`def pack(t: Tuple9[A,B,C,D,E,F,G,H,I]): Tup9[A,B,C,D,E,F,G,H,I]`

`def unpack(t: Tup9[A,B,C,D,E,F,G,H,I]): Tuple9[A,B,C,D,E,F,G,H,I]`

3.20 Vector

Vector defines a fixed size collection of scalar values.

3.20.1 Static methods

`def apply(elems: T*): Vector[T]`

Creates a new Vector containing the given elements

3.20.2 Infix methods

`def apply(i: Int): T`

Extracts the element of this vector at the given index

- **i** - the index of the element to extract

`def slice(start: Int, end: Int): Vector[T]`

Creates a subvector of this vector with elements [start, end)

- **start** - index of the first element to include in the subvector
- **end** - end index of the slice, non-inclusive

CHAPTER

FOUR

OBJECTS

4.1 Array

Unsynthesizable helper object for creating arrays on the CPU

4.1.1 Static methods

`def apply(x: T*): ForgeArray[T]`

Creates an array directly with apply function

`def empty(length: Index): ForgeArray[T]`

Creates an empty array with given length

`def fill(length: Index)(f:  => T): ForgeArray[T]`

```
Creates an array with given length whose elements are determined by the supplied function
```

`def tabulate(length: Index)(f: (Index) => T): ForgeArray[T]`

Creates an array with the given length whose elements are determined by the supplied indexed function

4.2 Indices

<auto-generated stub>

4.2.1 Infix methods

`def apply(y: Int): Index`

4.2.2 Related methods

`def indices_get_index(x: Indices, y: Int): Index`

4.3 Parallel

<auto-generated stub>

4.3.1 Static methods

```
def apply(body:  => Unit): Unit
```

Creates a parallel fork-join controller. Synchronizes all state machines in the body.

4.4 Pipe

<auto-generated stub>

4.4.1 Static methods

```
def apply(cchain: CounterChain)(func: (Indices) => Unit): Unit
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the ND domain defined by the supplied counterchain, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a pipelined fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **func** - the function to be executed each iteration

```
def apply(c0: Counter)(func: (Index) => Unit): Unit
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter specifying 1D index domain
- **func** - the function to be executed each iteration

```
def apply(c0: Counter, c1: Counter)(func: (Index, Index) => Unit): Unit
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the 2D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **func** - the function to be executed each iteration

```
def apply(c0: Counter, c1: Counter, c2: Counter)(func: (Index, Index, Index) => Unit): Uni
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the 3D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **c2** - counter for third dimension
- **func** - the function to be executed each iteration

```
def apply(body:  => Unit): Unit
```

Creates a "unit" pipeline. Used as a wrapper node around simple logic in the body.

```
def fold(cchain: CounterChain, accum: Int)(map: C[T])(reduce: (Indices) => C[T])(w: (T, T)
```

Multi-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied multi-dimensional domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **cchain** - counterchain specifying the index domain
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter)(accum: C[T])(map: (Index) => C[T])(reduce: (T, T) => T)(implicit ev0
```

1-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 1D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter specifying the 1D index domain
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter, c1: Counter)(accum: C[T])(map: (Index, Index) => C[T])(reduce: (T, T
```

2-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 2D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

`def fold(c0: Counter, c1: Counter, c2: Counter)(accum: C[T])(map: (Index, Index, Index) =>`

3-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 3D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **c2** - counter for the third dimension
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

`def foreach(cchain: CounterChain)(func: (Indices) => Unit): Unit`

Creates a pipelined, parallelizable state machine which iterates through the ND domain defined by the supplied counterchain, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a pipelined fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **func** - the function to be executed each iteration

`def foreach(c0: Counter)(func: (Index) => Unit): Unit`

Creates a pipelined, parallelizable state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter specifying 1D index domain
- **func** - the function to be executed each iteration

`def foreach(c0: Counter, c1: Counter)(func: (Index, Index) => Unit): Unit`

Creates a pipelined, parallelizable state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **func** - the function to be executed each iteration

```
def foreach(c0: Counter, c1: Counter, c2: Counter)(func: (Index, Index, Index) => Unit): U
```

Creates a pipelined, parallelizable state machine which iterates through the 3D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **c2** - counter for third dimension
- **func** - the function to be executed each iteration

```
def reduce(cchain: CounterChain)(zero: T)(map: (Indices) => T)(reduce: (T, T) => T)(implic
```

Multi-dimensional scalar fused map-reduce. Creates a state machine which iterates over the given multi-dimensional domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a pipelined fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

---

```
def reduce(c0: Counter)(zero: T)(map: (Index) => T)(reduce: (T, T) => T)(implicit ev0: Num
```

1-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 1D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter specifying the 1D index domain
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

---

```
def reduce(c0: Counter, c1: Counter)(zero: T)(map: (Index, Index) => T)(reduce: (T, T) =>
```

2-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 2D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def reduce(c0: Counter, c1: Counter, c2: Counter)(zero: T)(map: (Index, Index, Index) => T
```

3-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 3D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a pipelined fashion.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **c2** - counter for the third dimension
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

4.4.2 Related methods

```
def Fold(x: CounterChain, y: Int)(z: C[T], v: T)(w: (Indices) => C[T])(a: (T, T) => T)(impl
```

```
def Fold(x: Counter)(y: C[T], z: T)(v: (Index) => C[T])(w: (T, T) => T)(implicit ev0: Mem['
```

```
def Fold(x: Counter, y: Counter)(z: C[T], v: T)(w: (Index, Index) => C[T])(a: (T, T) => T)
```

```
def Fold(x: Counter, y: Counter, z: Counter)(v: C[T], w: T)(a: (Index, Index, Index) => C[
```

```
def Fold(x: Counter, y: Int)(z: C[T], v: T)(w: (Index) => C[T])(a: (T, T) => T)(implicit e
```

```
def Fold(x: Counter, y: Counter, z: Int)(v: C[T], w: T)(a: (Index, Index) => C[T])(b: (T,
```

```
def Fold(x: Counter, y: Counter, z: Counter, v: Int)(w: C[T], a: T)(b: (Index, Index, Inde
```

```
def Reduce(x: CounterChain)(y: T)(z: (Indices) => T)(v: (T, T) => T)(implicit ev0: Num[T])
```

`def Reduce(x: Counter)(y: T)(z: (Index) => T)(v: (T, T) => T)(implicit ev0: Num[T]): Reg[T`

`def Reduce(x: Counter, y: Counter)(z: T)(v: (Index, Index) => T)(w: (T, T) => T)(implicit`

`def Reduce(x: Counter, y: Counter, z: Counter)(v: T)(w: (Index, Index, Index) => T)(a: (T,`

4.5 Sequential

<auto-generated stub>

4.5.1 Static methods

`def apply(cchain: CounterChain)(func: (Indices) => Unit): Unit`

Shorthand form for foreach. Creates a sequential state machine which iterates through the ND domain defined by the supplied counterchain, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a sequential fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **func** - the function to be executed each iteration

`def apply(c0: Counter)(func: (Index) => Unit): Unit`

Shorthand form for foreach. Creates a sequential state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter specifying 1D index domain
- **func** - the function to be executed each iteration

`def apply(c0: Counter, c1: Counter)(func: (Index, Index) => Unit): Unit`

Shorthand form for foreach. Creates a sequential state machine which iterates through the 2D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **func** - the function to be executed each iteration

`def apply(c0: Counter, c1: Counter, c2: Counter)(func: (Index, Index, Index) => Unit): Uni`

Shorthand form for foreach. Creates a sequential state machine which iterates through the 3D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **c2** - counter for third dimension
- **func** - the function to be executed each iteration

```
def apply(body:  => Unit): Unit
```

Creates a “unit” pipeline. Used as a wrapper node around simple logic in the body.

```
def fold(cchain: CounterChain, accum: Int)(map: C[T])(reduce: (Indices) => C[T])(w: (T, T)
```

Multi-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied multi-dimensional domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **cchain** - counterchain specifying the index domain
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter)(accum: C[T])(map: (Index) => C[T])(reduce: (T, T) => T)(implicit ev0
```

1-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 1D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter specifying the 1D index domain
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter, c1: Counter)(accum: C[T])(map: (Index, Index) => C[T])(reduce: (T, T
```

2-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 2D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension

- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter, c1: Counter, c2: Counter)(accum: C[T])(map: (Index, Index, Index) =>
```

3-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 3D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **c2** - counter for the third dimension
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def foreach(cchain: CounterChain)(func: (Indices) => Unit): Unit
```

Creates a sequential state machine which iterates through the ND domain defined by the supplied counterchain, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a sequential fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **func** - the function to be executed each iteration

```
def foreach(c0: Counter)(func: (Index) => Unit): Unit
```

Creates a sequential state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter specifying 1D index domain
- **func** - the function to be executed each iteration

```
def foreach(c0: Counter, c1: Counter)(func: (Index, Index) => Unit): Unit
```

Creates a sequential state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension

- **func** - the function to be executed each iteration

```
def foreach(c0: Counter, c1: Counter, c2: Counter)(func: (Index, Index, Index) => Unit): U
```

Creates a sequential state machine which iterates through the 3D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **c2** - counter for third dimension
- **func** - the function to be executed each iteration

```
def reduce(cchain: CounterChain)(zero: T)(map: (Indices) => T)(reduce: (T, T) => T)(implic
```

Multi-dimensional scalar fused map-reduce. Creates a state machine which iterates over the given multi-dimensional domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a sequential fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

---

```
def reduce(c0: Counter)(zero: T)(map: (Index) => T)(reduce: (T, T) => T)(implicit ev0: Num
```

1-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 1D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter specifying the 1D index domain
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

---

```
def reduce(c0: Counter, c1: Counter)(zero: T)(map: (Index, Index) => T)(reduce: (T, T) =>
```

2-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 2D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter for the first dimension

- **c1** - counter for the second dimension
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def reduce(c0: Counter, c1: Counter, c2: Counter)(zero: T)(map: (Index, Index, Index) => T
```

3-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 3D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a sequential fashion.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **c2** - counter for the third dimension
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

4.6 Stream

<auto-generated stub>

4.6.1 Static methods

```
def apply(cchain: CounterChain)(func: (Indices) => Unit): Unit
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the ND domain defined by the supplied counterchain, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a streaming fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **func** - the function to be executed each iteration

```
def apply(c0: Counter)(func: (Index) => Unit): Unit
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter specifying 1D index domain
- **func** - the function to be executed each iteration

```
def apply(c0: Counter, c1: Counter)(func: (Index, Index) => Unit): Unit
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the 2D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **func** - the function to be executed each iteration

```
def apply(c0: Counter, c1: Counter, c2: Counter)(func: (Index, Index, Index) => Unit): Uni
```

Shorthand form for foreach. Creates a pipelined, parallelizable state machine which iterates through the 3D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **c2** - counter for third dimension
- **func** - the function to be executed each iteration

```
def apply(body:  => Unit): Unit
```

Creates a "unit" pipeline. Used as a wrapper node around simple logic in the body.

```
def fold(cchain: CounterChain, accum: Int)(map: C[T])(reduce: (Indices) => C[T])(w: (T, T)
```

Multi-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied multi-dimensional domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **cchain** - counterchain specifying the index domain
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter)(accum: C[T])(map: (Index) => C[T])(reduce: (T, T) => T)(implicit ev0
```

1-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 1D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter specifying the 1D index domain
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter, c1: Counter)(accum: C[T])(map: (Index, Index) => C[T])(reduce: (T, T
```

2-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 2D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def fold(c0: Counter, c1: Counter, c2: Counter)(accum: C[T])(map: (Index, Index, Index) =>
```

3-dimensional fused map-reduce of memories. Creates a state machine which iterates over the supplied 3D domain, reducing the collection resulting from each iteration of the map using the supplied associative scalar reduction function. This state machine is always run as an outer loop of state machines. If the memory result of the map function has multiple elements (e.g. BRAMs), the reduction is run as an inner loop where the supplied associative reduction is used on each iteration. Supported memory types are: Regs and BRAMs.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **c2** - counter for the third dimension
- **accum** - accumulator for holding intermediate reduction values
- **map** - map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def foreach(cchain: CounterChain)(func: (Indices) => Unit): Unit
```

Creates a pipelined, parallelizable state machine which iterates through the ND domain defined by the supplied counterchain, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a streaming fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **func** - the function to be executed each iteration

```
def foreach(c0: Counter)(func: (Index) => Unit): Unit
```

Creates a pipelined, parallelizable state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter specifying 1D index domain
- **func** - the function to be executed each iteration

```
def foreach(c0: Counter, c1: Counter)(func: (Index, Index) => Unit): Unit
```

Creates a pipelined, parallelizable state machine which iterates through the 1D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **func** - the function to be executed each iteration

```
def foreach(c0: Counter, c1: Counter, c2: Counter)(func: (Index, Index, Index) => Unit): U
```

Creates a pipelined, parallelizable state machine which iterates through the 3D domain defined by the supplied counter, executing the specified function every iteration. If the function contains other state machines, this is executed as an outer as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter for first dimension
- **c1** - counter for second dimension
- **c2** - counter for third dimension
- **func** - the function to be executed each iteration

```
def reduce(cchain: CounterChain)(zero: T)(map: (Indices) => T)(reduce: (T, T) => T)(implic
```

Multi-dimensional scalar fused map-reduce. Creates a state machine which iterates over the given multi-dimensional domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a streaming fashion. Note that this is the general form for N-dimensional domains. Use the specialized 1, 2, or 3D forms when possible.

- **cchain** - counterchain determining index domain
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

---

```
def reduce(c0: Counter)(zero: T)(map: (Index) => T)(reduce: (T, T) => T)(implicit ev0: Num
```

1-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 1D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter specifying the 1D index domain
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

---

```
def reduce(c0: Counter, c1: Counter)(zero: T)(map: (Index, Index) => T)(reduce: (T, T) =>
```

2-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 2D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** the accumulator used in this reduction (identical to *accum*)

```
def reduce(c0: Counter, c1: Counter, c2: Counter)(zero: T)(map: (Index, Index, Index) => T
```

3-dimensional scalar fused map-reduce. Creates a state machine which iterates over the supplied 3D domain, reducing the scalar result of each iteration of the map using the supplied associative reduction function. If the map function contains other state machines, this is executed as an outer loop with each inner state machine run as a stage in a streaming fashion.

- **c0** - counter for the first dimension
- **c1** - counter for the second dimension
- **c2** - counter for the third dimension
- **zero** - identity value for this reduction function
- **map** - scalar map function
- **reduce** - associative reduction function
- **returns** a register containing the result of this reduction

4.7 bankOverride

<auto-generated stub>

4.7.1 Static methods

```
def apply(x: Any): Int
```

```
def update(x: Any, y: Int): Unit
```

4.8 bound

<auto-generated stub>

4.8.1 Static methods

```
def apply(x: Any): Option[Double]
```

```
 def update(x: Any, y: Double): Unit

Symbol bounds Tracks the maximum value for a given symbol, along with data about this bound
```

```
def update(x: Any, y: MBound): Unit
```

```
def update(x: Any, y: Option[MBound]): Unit
```

4.9 domainOf

<auto-generated stub>

4.9.1 Static methods

```
def apply(x: Any): Option[Tuple3[Int,Int,Int]]
```

```
def update(x: Any, y: Tuple3[Int,Int,Int]): Unit
```

4.10 hardcodeEnsembles

<auto-generated stub>

4.10.1 Static methods

```
def apply(x: Any): Boolean
```

```
def update(x: Any, y: Boolean): Unit
```

4.11 isDummy

<auto-generated stub>

4.11.1 Static methods

```
def apply(x: Any): Boolean
```

```
def update(x: Any, y: Boolean): Unit
```

4.12 levelOf

<auto-generated stub>

4.12.1 Static methods

```
def apply(x: Any): Int
```

```
def update(x: Any, y: Int): Unit
```

4.13 memoryIndexOf

<auto-generated stub>

4.13.1 Static methods

```
def apply(x: Any): Int
```

```
def get(x: Any): Option[Int]
```

```
def update(x: Any, y: Int): Unit
```

4.14 parFactorOf

<auto-generated stub>

4.14.1 Static methods

```
def apply(x: Any): Int
```

```
def update(x: Any, y: Int): Unit
```

4.15 parFactorsOf

<auto-generated stub>

4.15.1 Static methods

```
def apply(x: Any): List[Int]
```

```
 def update(x: Any, y: List[Int]): Unit
```

```
Parallelization factors Isolated symbol parallelization factors (TODO: Clarify usage) User
```

4.16 unrollFactorsOf

<auto-generated stub>

4.16.1 Static methods

```
def apply(x: Any): List[Int]
```

```
 def update(x: Any, y: List[Int]): Unit
```

```
Unrolling factors The total set of unrolling factors for a given node. Defined such that, g
```

CHAPTER

FIVE

PRIMITIVES

5.1 Bit

Bit represents a single bit, equivalent to a Boolean

5.1.1 Infix methods

```
def !=(y: Bit): Bit
```

```
def &&(y: Bit): Bit
```

```
def ^(y: Bit): Bit
```

```
def mkString(): String
```

```
def toString(): String
```

```
def unary_!(): Bit
```

```
def ||(y: Bit): Bit
```

5.1.2 Related methods

```
def __equal(x: Bit, y: Bit): Bit
```

5.2 FixPt

FixPt[S,I,F] represents an arbitrary precision fixed point representation. FixPt values may be signed or unsigned. Negative values, if applicable, are represented in twos complement.

The type parameters for FixPt are:

S	Signed or unsigned representation	(Signed/Unsign)
I	Number of integer bits	(B0 - B64)
F	Number of fractional bits	(B0 - B64)

Note that numbers of bits use the B- prefix as integers cannot be used as type parameters in Scala

5.2.1 Type Aliases

type	SInt	*FixPt*[Signed,B32,B0]	Signed 32 bit integer
type	Index	*FixPt*[Signed,B32,B0]	Signed 32 bit integer (indexing)
type	UInt	*FixPt*[Unsign,B32,B0]	Unsigned 32 bit integer

5.2.2 Infix methods

```
def !=(y: FixPt[S,I,F]): Bit
```

```
def !=(y: Int): Bit
```

```
def !=(y: Long): Bit
```

```
def !=(y: Float): Bit
```

```
def !=(y: Double): Bit
```

```
def %(y: FixPt[S,I,B0]): FixPt[S,I,B0]
```

```
def %(y: Int): FixPt[S,I,B0]
```

```
def %(y: Long): FixPt[S,I,B0]
```

```
def %(y: Float): FixPt[S,I,B0]
```

```
def %(y: Double): FixPt[S,I,B0]
```

```
def &(y: FixPt[S,I,F]): FixPt[S,I,F]
```

```
def &(y: Int): FixPt[S,I,F]
```

```
def &(y: Long): FixPt[S,I,F]
```

```
def &(y: Float): FixPt[S,I,F]
```

```
def &(y: Double): FixPt[S,I,F]
```

```
def *(y: FixPt[S,I,F]): FixPt[S,I,F]
```

```
def *(y: Int): FixPt[S,I,F]
```

```
def *(y: Long): FixPt[S,I,F]
```

```
def *(y: Float): FixPt[S,I,F]
```

```
def *(y: Double): FixPt[S,I,F]
```

```
def **(x: Int): FixPt[S,I,F]
```

Integer power

- **n** - exponent, currently must be an integer greater than zero

---

```
def +(y: FixPt[S,I,F]): FixPt[S,I,F]
```

```
def +(y: Int): FixPt[S,I,F]
```

```
def +(y: Long): FixPt[S,I,F]
```

```
def +(y: Float): FixPt[S,I,F]
```

```
def +(y: Double): FixPt[S,I,F]
```

```
def -(y: FixPt[S,I,F]): FixPt[S,I,F]
```

`def -(y: Int): FixPt[S,I,F]`

`def -(y: Long): FixPt[S,I,F]`

`def -(y: Float): FixPt[S,I,F]`

`def -(y: Double): FixPt[S,I,F]`

`def /(y: FixPt[S,I,F]): FixPt[S,I,F]`

`def /(y: Int): FixPt[S,I,F]`

`def /(y: Long): FixPt[S,I,F]`

`def /(y: Float): FixPt[S,I,F]`

`def /(y: Double): FixPt[S,I,F]`

`def ::(end: Index): Range`

Creates a range with specified start (inclusive) and end (noninclusive)

`def <(y: FixPt[S,I,F]): Bit`

`def <(y: Int): Bit`

`def <(y: Long): Bit`

`def <(y: Float): Bit`

`def <(y: Double): Bit`

`def <<(y: FixPt[S,I,B0]): FixPt[S,I,F]`

`def <<(y: Int): FixPt[S,I,B0]`

```
def <=(y: FixPt[S,I,F]): Bit
```

```
def <=(y: Int): Bit
```

```
def <=(y: Long): Bit
```

```
def <=(y: Float): Bit
```

```
def <=(y: Double): Bit
```

```
def >(y: FixPt[S,I,F]): Bit
```

```
def >(y: Int): Bit
```

```
def >(y: Long): Bit
```

```
def >(y: Float): Bit
```

```
def >(y: Double): Bit
```

```
def >=(y: FixPt[S,I,F]): Bit
```

```
def >=(y: Int): Bit
```

```
def >=(y: Long): Bit
```

```
def >=(y: Float): Bit
```

```
def >=(y: Double): Bit
```

```
def >>(y: FixPt[S,I,B0]): FixPt[S,I,F]
```

Arithmetic shift right

```
def >>(y: Int): FixPt[S,I,B0]
```

`def by(end: Index): LoopRange`

Creates a LoopRange with start of 0 (inclusive), specified end (noninclusive) and step

---

`def mkString(): String`

`def to(): R`

Creates a conversion of this value to the given type

`def toString(): String`

`def unary_-(): FixPt[S,I,F]`

`def until(start: Index): LoopRange`

Creates a LoopRange with specified start (inclusive) and end (noninclusive) and step 1

`def |(y: FixPt[S,I,F]): FixPt[S,I,F]`

`def |(y: Int): FixPt[S,I,F]`

`def |(y: Long): FixPt[S,I,F]`

`def |(y: Float): FixPt[S,I,F]`

`def |(y: Double): FixPt[S,I,F]`

5.2.3 Related methods

`def __equal(x: FixPt[S,I,F], y: FixPt[S,I,F]): Bit`

`def __equal(x: Int, y: FixPt[S,I,F]): Bit`

`def __equal(x: FixPt[S,I,F], y: Int): Bit`

`def __equal(x: Long, y: FixPt[S,I,F]): Bit`

`def __equal(x: FixPt[S,I,F], y: Long): Bit`

```
def __equal(x: Float, y: FixPt[S,I,F]): Bit
```

```
def __equal(x: FixPt[S,I,F], y: Float): Bit
```

```
def __equal(x: Double, y: FixPt[S,I,F]): Bit
```

```
def __equal(x: FixPt[S,I,F], y: Double): Bit
```

```
def abs(x: FixPt[S,I,F]): FixPt[S,I,F]
```

Absolute value

5.3 FltPt

FltPt[G,E] represents an arbitrary precision, IEEE-754-like representation. FltPt values are always assumed to be signed.

The type parameters for FltPt are:

G	Number of significand bits, including sign bit	(B2 - B64)
E	Number of exponent bits	(B1 - B64)

Note that numbers of bits use the B- prefix as integers cannot be used as type parameters in Scala

5.3.1 Type Aliases

type	Half	*FltPt*[B11,B5]	IEEE-754 half precision
type	Flt	*FltPt*[B24,B8]	IEEE-754 single precision
type	Dbl	*FltPt*[B53,B11]	IEEE-754 double precision

5.3.2 Infix methods

```
def !=(y: FltPt[G,E]): Bit
```

```
def !=(y: Int): Bit
```

```
def !=(y: Long): Bit
```

```
def !=(y: Float): Bit
```

```
def !=(y: Double): Bit
```

`def *(y: FltPt[G,E]): FltPt[G,E]`

`def *(y: Int): FltPt[G,E]`

`def *(y: Long): FltPt[G,E]`

`def *(y: Float): FltPt[G,E]`

`def *(y: Double): FltPt[G,E]`

`def **(x: Int): FltPt[G,E]`

Integer power

- **n** - exponent, currently must be an integer greater than zero

---

`def +(y: FltPt[G,E]): FltPt[G,E]`

`def +(y: Int): FltPt[G,E]`

`def +(y: Long): FltPt[G,E]`

`def +(y: Float): FltPt[G,E]`

`def +(y: Double): FltPt[G,E]`

`def -(y: FltPt[G,E]): FltPt[G,E]`

`def -(y: Int): FltPt[G,E]`

`def -(y: Long): FltPt[G,E]`

`def -(y: Float): FltPt[G,E]`

`def -(y: Double): FltPt[G,E]`

---

def /(y: *FltPt*[G,E]): *FltPt*[G,E]

def /(y: Int): *FltPt*[G,E]

def /(y: Long): *FltPt*[G,E]

def /(y: Float): *FltPt*[G,E]

def /(y: Double): *FltPt*[G,E]

def <(y: *FltPt*[G,E]): *Bit*

def <(y: Int): *Bit*

def <(y: Long): *Bit*

def <(y: Float): *Bit*

def <(y: Double): *Bit*

def <=(y: *FltPt*[G,E]): *Bit*

def <=(y: Int): *Bit*

def <=(y: Long): *Bit*

def <=(y: Float): *Bit*

def <=(y: Double): *Bit*

def >(y: *FltPt*[G,E]): *Bit*

def >(y: Int): *Bit*

def >(y: Long): *Bit*

`def >(y: Float): Bit`

`def >(y: Double): Bit`

`def >=(y: FltPt[G,E]): Bit`

`def >=(y: Int): Bit`

`def >=(y: Long): Bit`

`def >=(y: Float): Bit`

`def >=(y: Double): Bit`

`def mkString(): String`

`def to(): R`

Creates a conversion of this value to the given type

`def toString(): String`

`def unary_-(): FltPt[G,E]`

5.3.3 Related methods

`def __equal(x: FltPt[G,E], y: FltPt[G,E]): Bit`

`def __equal(x: Int, y: FltPt[G,E]): Bit`

`def __equal(x: FltPt[G,E], y: Int): Bit`

`def __equal(x: Long, y: FltPt[G,E]): Bit`

`def __equal(x: FltPt[G,E], y: Long): Bit`

`def __equal(x: Float, y: FltPt[G,E]): Bit`

`def __equal(x: FltPt[G,E], y: Float): Bit`

`def __equal(x: Double, y: FltPt[G,E]): Bit`

`def __equal(x: FltPt[G,E], y: Double): Bit`

`def abs(x: FltPt[G,E]): FltPt[G,E]`

Absolute value

`def exp(x: FltPt[G,E]): FltPt[G,E]`

Natural exponential (Euler's number, e, raised to the given exponent)

`def log(x: FltPt[G,E]): FltPt[G,E]`

Natural logarithm

`def sqrt(x: FltPt[G,E]): FltPt[G,E]`

Square root

5.4 ForgeArray

<auto-generated stub>

5.4.1 Infix methods

`def apply(i: Index): T`

Returns the element at the given index

`def flatten(): ForgeArray[T]`

`def length(): Index`

Returns the length of this Array

`def map(y: T => R): ForgeArray[R]`

`def mkString(y: String): String`

`def reduce(y: (T, T) => T)(implicit ev0: Coll[T]): T`

`def update(i: Index, x: T): Unit`

Updates the array at the given index

`def zip(y: ForgeArray[S])(z: (T, S) => R): ForgeArray[R]`

`def zipWithIndex(): ForgeArray[Tup2[T,:doc:Index <fixpt>]]`

5.5 String

<auto-generated stub>

5.5.1 Infix methods

`def contains(y: String): Boolean`

`def endsWith(y: String): Boolean`

`def fcharAt(y: Int): Char`

`def fsplit(y: String, numSplits: Int = 0): ForgeArray[String]`

`def getBytes(): ForgeArray[Byte]`

`def length(): Int`

`def replaceAllLiterally(y: String, z: String): String`

`def slice(y: Int, z: Int): String`

`def split(y: String, numSplits: Int = 0): ForgeArray[String]`

`def startsWith(y: String): Boolean`

`def substring(y: Int): String`

`def substring(y: Int, z: Int): String`

`def to(): R`

Converts this String to the specified type

`def toBoolean(): Boolean`

`def toDouble(): Double`

`def toFloat(): Float`

`def toInt(): Int`

`def toLong(): Long`

`def toLowerCase(): String`

`def toUpperCase(): String`

`def trim(): String`

CHAPTER

SIX

GENERIC METHODS

6.1 T

<auto-generated stub>

6.1.1 Infix methods

`def +(y: String): String`

CHAPTER SEVEN

OPERATIONS

7.1 BasicCtrl

<auto-generated stub>

7.1.1 Related methods

```
def max(a: T, b: T)(implicit ev0: Order[T],ev1: Num[T]): T
```

Selects the maximum of two given values. Implemented as a mux with a greater-than comparison

---

```
def min(a: T, b: T)(implicit ev0: Order[T],ev1: Num[T]): T
```

Selects the minimum of two given values. Implemented as a mux with a less-than comparison

---

```
def mux(sel: Bit, a: T, b: T)(implicit ev0: Num[T]): T
```

2 input multiplexer

7.2 ForgeArrayAPI

<auto-generated stub>

7.2.1 Related methods

```
def __equal(x: ForgeArray[T], y: ForgeArray[T])(implicit ev0: Order[T]): Bit
```

7.3 GraphIO

<auto-generated stub>

7.3.1 Related methods

```
def genRandDirEdgeList(x: String, y: Index, z: Index, v: Boolean): Unit
```

```
def getEdgeList(x: scala.collection.mutable.HashMap[Int,scala.collection.mutable.ListBuffe
```

```
def getVertList(map: scala.collection.mutable.HashMap[Int,scala.collection.mutable.ListBuf
```

Generating an array of array representing vertices and their pointer to the edge list based on the adjacency list represented in the map. If explicitVert=true, withSize=true, returns an N by 3 Array of Array(vertex #, ptr, size) If explicitVert=true, withSize=false, returns an N by 2 Array of Array(vertex #, ptr) If explicitVert=false, withSize=true, returns an N by 2 Array of Array(ptr, size) If explicitVert=false, withSize=false, returns an N by 1 Array of Array(ptr) size is the number of edges for correponding vertex.

```
 def loadDirEdgeList(__arg0: String, __arg1: Index, __arg2: Boolean): ForgeArray[scala.col

Returns Array(smap,dmap), where smap is the adjacency list in hashMap for source nodes and

       * __arg0 - path full path to graph file
       * __arg1 - numVert number of vertices
       * __arg2 - dot whether is loading from a dot graph
```

```
 def loadDirEdgeList(__arg0: String, __arg1: Boolean): ForgeArray[scala.collection.mutable

Returns Array(smap,dmap), where smap is the adjacency list in hashMap for source nodes and

       * __arg0 - path full path to graph file
       * __arg1 - dot whether is loading from a dot graph
```

```
 def loadDirEdgeList(__arg0: String, __arg1: Index): ForgeArray[scala.collection.mutable.H

Returns Array(smap,dmap), where smap is the adjacency list in hashMap for source nodes and

       * __arg0 - path full path to graph file
       * __arg1 - numVert number of vertices
```

```
 def loadDirEdgeList(__arg0: String): ForgeArray[scala.collection.mutable.HashMap[Int,scal

Returns Array(smap,dmap), where smap is the adjacency list in hashMap for source nodes and

       * __arg0 - path full path to graph file
```

```
 def loadUnDirEdgeList(__arg0: String, __arg1: Index, __arg2: Boolean): scala.collection.m

Returns an adjacency list in HashMap

       * __arg0 - path full path to graph file
       * __arg1 - numVert number of vertices
       * __arg2 - dot whether is loading from a dot graph
```

```
def loadUnDirEdgeList(__arg0: String, __arg1: Boolean): scala.collection.mutable.HashMap[I
```

```
Returns an adjacency list in HashMap

       * __arg0 - path full path to graph file
       * __arg1 - dot whether is loading from a dot graph
```

```
def loadUnDirEdgeList(__arg0: String, __arg1: Index): scala.collection.mutable.HashMap[In
```

```
Returns an adjacency list in HashMap

       * __arg0 - path full path to graph file
       * __arg1 - numVert number of vertices
```

```
def loadUnDirEdgeList(__arg0: String): scala.collection.mutable.HashMap[Int,scala.collect
```

```
Returns an adjacency list in HashMap

       * __arg0 - path full path to graph file
```

```
def load_dir_edge_list(x: String, y: Option[Int], z: Option[Boolean]): ForgeArray[scala.co
```

```
def load_undir_edge_list(x: String, y: Option[Int], z: Option[Boolean]): scala.collection.
```

7.4 Math

<auto-generated stub>

7.4.1 Related methods

```
def pow(x: T, n: Int)(implicit ev0: Arith[T]): T
```

Integer power implemented as a simple reduction tree

- **n** - exponent, currently must be an integer greater than zero

```
def productTree(x: List[T])(implicit ev0: Arith[T]): T
```

Creates a reduction tree which calculates the product of the given symbols

```
def reduceTree(syms: List[T])(rFunc: (T, T) => T): T
```

Creates a reduction tree structure of the given list of symbols

- **syms** - List of symbols to reduce

- **rFunc** - Associative reduction function

`def sumTree(x: List[T])(implicit ev0: Arith[T]): T`

Creates a reduction tree which calculates the sum of the given symbols

---

`def zero()(implicit ev0: Num[T]): T`

7.5 Nosynth

<auto-generated stub>

7.5.1 Related methods

`def Accel(x:  => Unit): Unit`

`def __ifThenElse(x: Boolean, y:  => T, z:  => T): T`

`def __whileDo(x:  => Boolean, y:  => Unit): Unit`

`def assert(x: Bit): Unit`

`def getArg(x: Reg[T]): T`

`def getBram(bram: BRAM[T]): ForgeArray[T]`

Get content of BRAM in an array format (debugging purpose only)

`def getMem(x: OffChipMem[T]): ForgeArray[T]`

`def print(x: Any): Unit`

`def printBram(bram: BRAM[T]): Unit`

Print content of a BRAM (debugging purpose only)

`def printMem(bram: OffChipMem[T]): Unit`

Print content of a OffChip (debugging purpose only)

`def println(x: Any): Unit`

`def println(): Unit`

`def setArg(x: Reg[T], y: T): Unit`

`def setArg(x: Reg[T], y: Int): Unit`

`def setArg(x: Reg[T], y: Long): Unit`

`def setArg(x: Reg[T], y: Float): Unit`

`def setArg(x: Reg[T], y: Double): Unit`

`def setBram(bram: BRAM[T], array: ForgeArray[T]): Unit`

Set content of BRAM to an array (debugging purpose only)

`def setMem(x: OffChipMem[T], y: ForgeArray[T]): Unit`

7.6 Rand

Unsynthesizable group of operations for generating random data for testing

7.6.1 Related methods

`def random(x: A): A`

Returns a uniformly distributed random value of type A with the given maximum value

---

`def random(x: Int): A`

`def random(x: Long): A`

`def random(x: Float): A`

`def random(x: Double): A`

`def random(): A`

Returns a uniformly distributed random value of type A. Fixed point types are unbounded, while floating point types are between 0 and 1

7.7 Tpes

<auto-generated stub>

7.7.1 Implicit methods

`def insert_unit(x: Any): Unit`

7.7.2 Related methods

`def bit_to_bool(x: Bit): Boolean`

`def bit_to_string(x: Bit): String`

`def fix_to_int(x: FixPt[S,I,B0]): Int`

`def fixpt_to_string(x: FixPt[S,I,F]): String`

`def fltpt_to_string(x: FltPt[G,E]): String`

`def int_to_fix(x: Int): FixPt[S,I,B0]`

`def string_to_fixpt(x: String): FixPt[S,I,F]`

`def string_to_fltpt(x: String): FltPt[G,E]`