DHDL Documentation

Release 0.1

Stanford PPL

April 30, 2016

Contents

Contents:

CHAPTER 1

Introduction

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

Templates in DHDL 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, as shown in Figure 2. Every template is parameterized. A specific hardware design point is instantiated from a DHDL description by instantiating all the templates in the design with concrete parameter values passed to the program. DHDL heavily uses metaprogramming, so these values are passed in as arguments to the DHDL 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 2

Type Classes

2.1 Arith

<auto-generated stub>

2.1.1 Infix methods

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

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

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

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

2.2 Coll

<auto-generated stub>

2.2.1 Infix methods

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

```
def zeros(x: T): T
```

2.3 Mem

<auto-generated stub>

2.3.1 Infix methods

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

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

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

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 eql(x: T, y: T): Bit
```

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

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

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

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

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

CHAPTER 3

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(name: String, dims: Index*)(implicit ev0: Num[T]): BRAM[T]
```

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

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

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

3.1.2 Infix methods

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

Creates a tile store from a Tile of an OffChipMem to 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(y: Seq[Index], z: T): Unit
```

3.2 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.

3.2.1 Static methods

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

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

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

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

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

Creates an unnamed Counter with given min, max and step size

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

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

---

```
def apply(name: String, max: Index): Counter
```

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

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

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

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

Creates a named Counter with given min, max and step size

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

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

3.3 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.

3.3.1 Static methods

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

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

3.4 LoopRange

<auto-generated stub>

3.4.1 Static methods

```
def apply(x: Index, y: Index, z: Index): LoopRange
```

3.4.2 Infix methods

```
def by(y: Index): LoopRange
```

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

```
def par(y: Int): Counter
```

3.4.3 Implicit methods

```
def rangeToCounter(x: LoopRange): Counter
```

3.5 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). Other access patterns will be supported soon!

3.5.1 Static methods

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

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

---

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

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

3.5.2 Infix methods

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

Creates a reference to a 1D Tile of this 1D OffChipMem which can be loaded into on-chip BRAM.

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

Creates a reference to a 2D Tile of this 2D OffChipMem which can be loaded into on-chip BRAM.

```
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 on-chip BRAM.

```
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

3.6 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.

3.6.1 Static methods

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

Creates a register with type T and given name

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

Creates an unnamed register with type T

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

Creates a register of type T with given name and reset value

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

Creates an unnamed register with type T and given reset value

---

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

Creates a register of type T with given name and 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(name: String, reset: Float)(implicit ev0: Num[T]): Reg[T]`

Creates a register of type T with given name and 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(name: String, reset: Double)(implicit ev0: Num[T]): Reg[T]`

Creates a register of type T with given name and reset value

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

Creates an unnamed register with type T and given reset value

3.6.2 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

3.6.3 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

3.6.4 Related methods

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

Creates a named input argument from the host CPU

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

Creates an unnamed input argument from the host CPU

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

Creats a named output argument to the host CPU

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

Creats an unnamed output argument to the host CPU

3.7 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.

3.7.1 Infix methods

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

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

3.8 Tup2

<auto-generated stub>

3.8.1 Infix methods

```
def _1(): A
```

```
def _2(): B
```

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

3.8.2 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]
```

3.9 Tup3

<auto-generated stub>

3.9.1 Infix methods

```
def _1(): A
```

```
def _2(): B
```

```
def _3(): C
```

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

3.9.2 Related methods

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

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

3.10 Tup4

<auto-generated stub>

3.10.1 Infix methods

```
def _1(): A
```

```
def _2(): B
```

```
def _3(): C
```

```
def _4(): D
```

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

3.10.2 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]
```

3.11 Tup5

<auto-generated stub>

3.11.1 Infix methods

```
def _1(): A
```

```
def _2(): B
```

```
def _3(): C
```

```
def _4(): D
```

```
def _5(): E
```

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

3.11.2 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]
```

3.12 Tup6

<auto-generated stub>

3.12.1 Infix methods

```
def _1(): A
```

```
def _2(): B
```

```
def _3(): C
```

```
def _4(): D
```

```
def _5(): E
```

```
def _6(): F
```

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

3.12.2 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]
```

3.13 Tup7

<auto-generated stub>

3.13.1 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
```

3.13.2 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]
```

3.14 Tup8

<auto-generated stub>

3.14.1 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
```

3.14.2 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]
```

3.15 Tup9

<auto-generated stub>

3.15.1 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
```

3.15.2 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]
```

CHAPTER 4

Objects

4.1 Array

Unsynthesizable helper object for creating arrays on the CPU

4.1.1 Static methods

```
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 Static methods

```
def apply(x: Index*): Indices
```

4.2.2 Infix methods

```
def apply(y: Int): Index
```

4.2.3 Related methods

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

4.3 MetaPipe

<auto-generated stub>

4.3.1 Static methods

```
def apply(x: Counter)(y: (Index) => Unit): Unit
```

```
def apply(x: Counter, y: Counter)(z: (Index, Index) => Unit): Unit
```

```
def apply(x: Counter, y: Counter, z: Counter)(v: (Index, Index, Index) => Unit): Unit
```

```
def apply(x: Counter, y: C[T])(z: (Index) => T)(v: (T, T) => T)(implicit ev0: Mem[T,C],ev1
```

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

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

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

```
def foreach(x: CounterChain)(y: (Indices) => Unit): Unit
```

```
def reduce(x: CounterChain, y: C[T])(z: (Indices) => T)(v: (T, T) => T)(implicit ev0: Mem[
```

4.3.2 Related methods

```
def BlockReduce(x: Counter, y: BRAM[T])(z: (Index) => BRAM[T])(v: (T, T) => T): Unit
```

```
def BlockReduce(x: Counter, y: Counter, z: BRAM[T])(v: (Index, Index) => BRAM[T])(w: (T, T
```

```
def BlockReduce(x: Counter, y: Counter, z: Counter, v: BRAM[T])(w: (Index, Index, Index) =>
```

```
def BlockReduce(x: Counter, y: BRAM[T], z: Int)(v: (Index) => BRAM[T])(w: (T, T) => T): Uni
```

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

```
def block_reduce_create(x: CounterChain, y: Int, z: BRAM[T], v: (Indices) => BRAM[T], w: (
```

4.4 Pipe

<auto-generated stub>

4.4.1 Static methods

```
def apply(x: Counter)(y: (Index) => Unit): Unit
```

```
def apply(x: Counter, y: Counter)(z: (Index, Index) => Unit): Unit
```

```
def apply(x: Counter, y: Counter, z: Counter)(v: (Index, Index, Index) => Unit): Unit
```

```
def apply(x: Counter, y: C[T])(z: (Index) => T)(v: (T, T) => T)(implicit ev0: Mem[T,C],ev1
```

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

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

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

```
def foreach(x: CounterChain)(y: (Indices) => Unit): Unit
```

```
def reduce(x: CounterChain, y: C[T])(z: (Indices) => T)(v: (T, T) => T)(implicit ev0: Mem[
```

4.5 Sequential

<auto-generated stub>

4.5.1 Static methods

```
def apply(x: Counter)(y: (Index) => Unit): Unit
```

```
def apply(x: Counter, y: Counter)(z: (Index, Index) => Unit): Unit
```

```
def apply(x: Counter, y: Counter, z: Counter)(v: (Index, Index, Index) => Unit): Unit
```

```
def apply(x: Counter, y: C[T])(z: (Index) => T)(v: (T, T) => T)(implicit ev0: Mem[T,C],ev1
```

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

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

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

```
def foreach(x: CounterChain)(y: (Indices) => Unit): Unit
```

```
def reduce(x: CounterChain, y: C[T])(z: (Indices) => T)(v: (T, T) => T)(implicit ev0: Mem[
```

4.6 bound

<auto-generated stub>

4.6.1 Static methods

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

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

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

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

4.7 domainOf

<auto-generated stub>

4.7.1 Static methods

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

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

4.8 isDblBuf

<auto-generated stub>

4.8.1 Static methods

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

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

CHAPTER 5

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
```

5.2 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 **(y: Int): 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 /(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: Index): Range
```

```
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]
```

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

```
def by(y: Index): LoopRange
```

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

```
def to(): R
```

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

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

```
def until(y: Index): LoopRange
```

```
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]
```

5.3 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 **(y: Int): 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]): 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
```

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

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

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
```

```
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 6

Generic Methods

6.1 T

<auto-generated stub>

6.1.1 Infix methods

```
def +(y: String): String
```

CHAPTER 7

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
```

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

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

```
def mux(sel: Bit, a: T, b: CT)(implicit ev0: Num[T],ev1: Numeric[CT]): T
```

```
def mux(sel: Bit, a: CT, b: T)(implicit ev0: Num[T],ev1: Numeric[CT]): T
```

7.2 DHDLPrim

<auto-generated stub>

7.2.1 Related methods

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
def not_bit(x: Bit): Bit
```

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

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

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

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

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

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

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

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

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

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

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

7.3 ForgeArrayAPI

<auto-generated stub>

7.3.1 Related methods

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

7.4 DHDLMisc

7.4.1 Related methods

```
def param(name: String, default: T): T
```

Creates a design parameter with the given name and default value

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

```
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 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 bit_to_boolean(x: Bit): Boolean
```

```
def scala_boolean_to_bit(x: Boolean): Bit
```

```
def scala_float_to_fltpt(x: Float): Flt
```

```
def scala_int_to_fixpt(x: Int): SInt
```

```
def stage_int_to_fixpt(x: Int): SInt
```

7.7.2 Related methods

```
def bit(x: Boolean): Bit
```

```
def bit_to_bool(x: Bit): Boolean
```

```
def bit_to_string(x: Bit): String
```

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

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

```
def cast_string_to(x: String): R
```

```
def fixPt(x: T)(implicit ev0: Numeric[T]): FixPt[S,I,F]
```

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

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

```
def fltPt(x: T)(implicit ev0: Numeric[T]): FltPt[G,E]
```

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

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

```
def lift_to(x: T)(implicit ev0: Numeric[T]): R
```

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

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