Tydi language specification (pocket edition)

The source code of Tydi lang front end (Spoofax prototype version) is available here.

Literals

Comment

Comments in Tydi is similar to the C-style line comments. Block comments is currently not supported.

```
//This is a comment in Tydi-lang
```

Int

```
0 //normal integer 0
0x01234567894abcdef // hex integer
0x01234567894ABCDEF // hex integer is case insensitive
0o01234567 // octal integer
0b01010101 // binary integer
0b0000_0001 // you can use "_" as a separator anywhere except the beginning of the integer.
```

Float

```
1.0
22.00205
```

String

```
"123" // this is a string
```

Boolean literal

```
true
false // notice that "true" and "false" are case sensitive
```

Const values

Basic const values

You can define variables as many other programming languages to generate the hardware components. However, in Tydi-lang, all variables must be mutable, means they cannot be changed after definition. Tydi-lang support three built-in data types: `bool`, `int` and `string`.

```
const eight : int = 8;
const string : str = "Hello";
const flag : bool = true;
const f:float = 1.0;
```

Tydi-lang also supports type inference. For this case the type of the const will be inferenced from the type of literal.

```
const eight = 8;
const string = "Hello";
const flag = true;
const f = 1.0;
```

Const value expression

You can use arithmetic operations on const values to calculate new values. These calculation must be able to be performed on the front end stage. In other words, all values will be evaluated at the front end stage.

Arithmetic operations for integer

```
const number0 :int = 1;
const number1 :int = 2;

const add = number0 + number1; // add = 1+2 = 3
const minus = number1 - number0; // minus = 2-1 = 1
const times = number0 * number1; // times = 1*2 = 2
const div0 = number0 ./ number1; // div0 = 1/2 = 0, please notice that div follows the integer divide rules.
const div1 = number1 ./ number0; // div1 = 2/1 = 2

const mod = number0 % number1; // mod = 1%2 = 1
const left_shift = number0 << number1; // left_shift = 1<<2 = ob100 = 4
const right_shift = number0 >> number1; // right_shift = 1>>2 = 0

const power = number1 ^ number0; //power = 2^1 = 2

const bitwise_and = number0 & number1; //bitwise_and = 0b01 & 0b10 = 0b00 = 0
const bitwise_or = number0 | number1; //bitwise_and = 0b01 & 0b10 = 0b11 = 3
const bitwise_not = ~ (4) number0; //bitwise_not = ~ 0b0001 = 0b1110 = 14, notice that the expression "4" indicates the bit length of the data, becuase we don't specify the integer bit length when we declare it.
```

Arithmetic operations for floating point

Basic arithmetic operations such as addition, minus, multiplication, and power are also applicable for floating point numbers where the result type will be float.

```
const number0 :float = 1.0;
const number1 :float = 2.0;
const int0 : int = 1;

const add = number0 + number1; // add = 1.0+2.0 = 3.0
const add2 = int0 + number1; // add = 1+2.0 = 3.0
const minus = number1 - number0; // minus = 2.0-1.0 = 1.0
const times = number0 * number1; // times = 1.0*2.0 = 2.0

const power = number1 ^ number0; //power = 2.0^1.0 = 2.0
```

Logarithm computation is useful for calculating bit length. Tydi lang supports logarithm as following code.

```
const log = log 2 (8); //log_lower = log2(8) = 3.0
const log = log 2.0 (8.0); //log_lower = log2.0(8.0) = 3.0, the inputs data type can be int or
float, the output type is always float.
```

To convert a floating point number to an integer number, you can use following three converting functions.

```
const int0 = round(2.5); //int0 = 3
const int1 = round(2.4); //int1 = 2

const int2 = floor(2.5); //int2 = 2, round up
const int3 = ceil(2.5); //int3 = 3, round down
```

Logical operations

Logical operations only support bool values.

```
const flag0 = true;
const flag1 = false;

const logical_and = flag0 && flag1; // logical_and = true && false = false
const logical_or = flag0 || flag1; // logical_or = true || false = true
const logical_not = ! flag0;  // logical_and = !true = false
```

Comparison operation

Comparison operations only support integer values.

```
const value0 = 1;
const value1 = 2;

const comparision0 = value0 == value1; //equal = (1==2) = false
const comparision1 = value0 != value1; //equal = (1!=2) = true
const comparision2 = value0 < value1; //equal = (1<2) = true
const comparision3 = value0 <= value1; //equal = (1<=2) = true
const comparision4 = value0 > value1; //equal = (1>2) = false
const comparision5 = value0 >= value1; //equal = (1>=2) = false

const comparision6 = 1.0==1.0;//Notice: comparision among floating point is possible but may
suffer from floating point error
```

Array const values

Defining an array is useful for generating parallelized hardware components. In Tydi-lang, you can define an array as follow. NOTICE: all expressions in an array must have the same type.

```
const array_of_int : int[] = {1,2,3};
const array_of_str : str[] = {"1","2","3"};
const flags : bool[] = {true,true,false};
```

Similarly, the type inference is also possible.

```
const array_of_int = {1,2,3}; // an array of integer
const array_of_str = {"1","2","3"}; // an array of string
const flags = {true,true,false}; // an array of bool
```

You can also use integer step to represent an array. An integer step contains three integer expressions, which are "start value", "step value" and "stop value". NOTICE: this representation only works for integer.

```
const int_array = 0-1->5; // int_array = {0,1,2,3,4}, an array starts from 0 until 5 with a step
of 1.
```

Array operations

To access an element of an array.

```
const a = int_array[0]; // the type of a will be infered as int
```

Append data to the array.

```
const a = int_array + 5; // a = {0,1,2,3,4,5}, append 5 at the end of the array
const b = 5 + int_array; // a = {5,0,1,2,3,4}, append 5 at the begin of the array
```

Const data operation across types

Tydi-lang also provides limited auto conversion between different const data types. For example, appending an integer or bool to a string is possible.

```
const str0 = " hello ";
const flag = true;
const str1 = str0 + 1; //str1 = "hello 1"
const str2 = 1 + str0; //str1 = "1 hello"
const str3 = flag + str0; //str1 = "true hello"
```

Logical data type

In Tydi-lang, you can define 5 logical types to represent the data structure. The details of each logical types are available in the Tydi paper and specification. The most important idea to describe complex data structures in Tydi-lang is using type alias. For example, `type a=b` defines an alias "a" for type "b".

Null

```
type null = Null;//we make an alias of type Null, Null itself represents the logical Null type.
```

Bit

```
type byte = Bit(8);//define byte as Bit(8)
```

Group

Syntax:

```
GROUP_TYPE = Group <ID> {
  <TYPE_ITEM+>
}

TYPE_ITEM = <ID> : <TYPE> ,
```

A Group example:

```
type location = Group location_ {
  x : Bit(32),
  y : Bit(32),
};
```

For simplicity, you can directly define a Group type without using alias.

```
type Group location {
  x : Bit(32),
  y : Bit(32),
};
```

Union

Union has a similar syntax as the Group syntax.

```
UNION_TYPE = Union <ID> {
  <TYPE_ITEM+>
}

TYPE_ITEM = <ID> : <TYPE> ,
```

An Union example:

```
type location = Union location_ {
  x : Bit(32),
  y : Bit(32),
};//type alias form

type Union location {
  x : Bit(32),
  y : Bit(32),
};//without type alias
```

Stream

Stream type can be defined by following syntax:

```
Stream ( <LOGICAL_TYPE> <STREAM_OPTION*> )
STREAM_OPTION = , d = <Exp> //define the dimension, the Exp must be an integer. Default: 0
STREAM_OPTION = , u = <LOGICAL_TYPE> //define the user type, the LOGICAL_TYPE should be a Null,
Bit, Group, Union or a composite of them (including an type alias). Default: Null
STREAM_OPTION = , t = <Exp> //define the throughput, the Exp must be a float. Default: 1
STREAM_OPTION = , s = <Exp> //define the synchronicity, the Exp must be a string. From Tydi
specification, the available options are "Sync", "Flatten", "Desync" and "FlatDesync". Default:
"Sync"
STREAM_OPTION = , c = <Exp> //define the complexity, the Exp must be an integer. Default: 7
(highest)
STREAM_OPTION = , r = <Exp> //define the direction, the Exp must be a string. Available options
are "Forward" and "Reverse". Default: Forward
STREAM_OPTION = , x = <Exp> //define the keep, the Exp must be a Bool. Default: False
```

Some examples about Stream:

```
type stream0 = Stream(Bit(4));
type stream1 = Stream(Bit(4), d=2, c=6);
```

Composite Examples

A small example to define a composite logical data structure:

```
type byte = Bit(8);
type word = Stream(byte, d=1);
type sentence = Stream(word, d=1);

type color_depth = Bit(10);
type Group pixel {
  r : color_depth,
  g : color_depth,
  b : color_depth,
};
type pixel_stream = Stream(pixel, d=2);
```

Streamlet

In Tydi-lang, streamlet describes the port of a component. Similar to the "entity" concept in VHDL(without the generic list).

Syntax

```
streamlet <ID> {
  <STREAMLET_PORT> ,
}
STREAMLET_PORT = <ID> : <LOGICAL_TYPE> <DIR> //define a single port, the LOGICAL_TYPE must be a stream type
STREAMLET_PORT = <ID> : <LOGICAL_TYPE> [ <Exp> ] <DIR> // define an array of port, the Exp must be an integer
DIR = "in"
DIR = "out"
```

An example:

```
package test; // define the package of the file
const num_stream = 2;
type stream0 = Stream(Bit(4));

streamlet test {
  in : stream0 in,
  out : stream0 out,

  in_array : stream0 [num_stream] in, //define 2 input ports of stream0
  out_array : stream0 [num_stream] out, //define 2 output ports of stream0
};
```

Streamlet template

Streamlet template can use logical types and const values as arguments to define a streamlet. This method is an abstraction of streamlet and will be useful in writing some libraries when library designers don't know how use will use it.

```
streamlet demux <mux_output:int> {
  in : Stream(Bit(1)) in,
  out : Stream(Bit(1)) [mux_output] out,

  select : Stream(Bit(ceil(log 2(mux_output)))) in,
}; // a demux with <mux_output> output ports

streamlet demux_data <t:type, mux_output:int> {
  in : Stream(t) in,
  out : Stream(t) [mux_output] out,

  select : Stream(Bit(ceil(log 2(mux_output)))) in,
}; // t is a logical type
```

A streamlet template can be instantiate by the following form.

```
demux<1>
const num = 2;
demux<num> //demux<2> is different from demux<1> because they might have completely different
port layout.
demux_data<type Bit(8), 2> //for template type arguments, you need to prefix a "type" to avoid
syntax ambiguous (it also might be a const)
```

Implement

Implement describes the internal layout (connections, sub components, etc) of a streamlet.

```
package test; // define the package of the file

type stream0 = Stream(Bit(4));

streamlet child {
  in : stream0 in,
  out : stream0 out,
}; // suppose we have a streamlet called child and its implement is temp

impl temp of child {
  //...
};

//now we want to describe a parent with following streamlet which contains 2 child streamlet.
streamlet parent {
  in : stream0 [2] in,
  out0 : stream0 out,
  out1 : stream0 out,
};

impl parent1 of parent {
  instance childs(temp) [2], //declare two "childs" instances with "impl temp", the port of
"childs" instance will be infered from the streamlet "child"
```

```
  childs[0].out => self.out0, // connect the "out" of first child to "out0" of parent1
  childs[1].out => self.out1,

  for x in 0-1->2 {
    self.in[x] => childs[x].in, //use for loop to describe connections
  }

  process{},
};
```

Reference system

In the above code, each port reference will be bind to its declaration. I use [[[]]] to represent their reference relationship where same number indicates they are referenced together.

```
package test; // define the package of the file

type [0[[stream0]]] = Stream(Bit(4));

streamlet child {
  [1[[in]]] : [0[[stream0]]] in,
  [2[[out]]] : [0[[stream0]]] out,
};

impl temp of child {
  //...
};

streamlet parent {
  [3[[in]]] : stream0 [2] in,
  [4[[out0]]] : stream0 out,
  [5[[out1]]] : stream0 out,
};

impl parent1 of parent {
  instance childs(temp) [2],

  childs[0].[2[[out]]] => self.[4[[out0]]], //the self will be resolved to the "parent"
streamlet
  childs[1].[2[[out]]] => self.[5[[out1]]],

  for x in 0-1->2 {
    self.[3[[in]]][x] => childs[x].[1[[in]]],
  }

  process{},
};
```

Declare connection

Two ports can only be connected only if they meet following rules:

- connect from an "output" port to an "input" port. The direction of the port is determined by its relative direction rather than the direction on the code. For example, the relative direction of an output port for a inner instance is "output", while an output port of the implement itself should be "input".eg. `childs[1].[2[[out]]] => self.[5[[out1]]],`
- The two port must have the same logical data type

Current prototype version only supports checking above two rules, the following rules only apply to future Tydi-lang version. TODO

- For stream type a => stream type b, the Dimension, the UserType and the ElementLane should equal.
- The source complexity should be smaller than sink complexity.

Declare connection with delay

```
childs[0].out =4=> self.out0, // the "out" signal will arrive "out0" after 4 time unit.(As for what is the time unit, it can be clock cycle or data sample (achieve by an FIFO), maybe we need more discussions here)
```

Control block

Tydi-lang provides two mechanisms to control the layout of an implement.

```
package test; // define the package of the file

const flag = true;
const num_instance = 8;
const num_stream = 2;

type stream0 = Stream(Bit(4));
type stream1 = Stream(Bit(8));

streamlet mux2<i:int, t:type> {
  in : stream0 in,
  out : stream0 out,

  in_ : stream1 in,

  in_array : stream0 [num_stream] in,
  out_array : stream0 [num_stream] out,
};

impl temp of mux2<8, type Bit(10)> {

};

impl tmux<n:int, t : type> of mux2<n, type stream0> {
  instance adders(temp) [8],
  instance and_gate(temp),

  and_gate.out => and_gate.in,
  self.in => self.out,
```

```
  and_gate.out => self.out,
  self.in => and_gate.in,

  and_gate.out_array[0] => and_gate.in_array[0],

  if (flag) { // if block, the if Exp is a bool expression
    self.in => and_gate.in,
  }
  elif (flag) { // else if block, the if Exp is a bool expression
    self.in => and_gate.in,
  }
  else{ // else block, the if Exp is a bool expression
    self.in => and_gate.in,
  }

  for inst_index in 0-1->num_instance {
    for stream_index in 0-1->num_stream {
      adders[inst_index].out_array[stream_index] => adders[inst_index].in_array[stream_index],
    }
  }
  and_gate.out_array[0] => and_gate.in_array[0],

  process{}, // just a place holder
};
```

Implement Template

Similarly to streamlet, you can also use an implement which is not yet defined to construct a new implement. Here is an example.

```
package test;

const flag = true;
const num_instance = 8;
const num_stream = 2;

type stream0 = Stream(Bit(4));

streamlet sl0<i:int, t:type> { //define a streamlet with templates: {int, type}
  in : stream0 in,
  out : stream0 out,

  in_array : t [num_stream] in,
  out_array : t [num_stream] out,
};

streamlet sl1 { //define a streamlet without templates
  in : stream0 in,
  out : stream0 out,

  in_array : stream0 [num_stream] in,

  out_array : stream0 [num_stream] out,
```

```
};

impl temp_impl of sl0<num_stream, type Bit(1)> { //implement temp_impl of sl0<2, type Bit(1)>

};
impl tmux<n: int, ts: impl of sl0<num_instance, type Bit(1)>> of sl0<n, type stream0> { // here we define ts, which is an implementation of sl0<2, type Bit(1)> but don't specify the implementation.
  instance test_inst(ts), // define an instance of ts, the port layout will be resolved in the streamlet sl0 scope

  test_inst.out => test_inst.in,

  process{},
};
impl test of sl0<2, type Bit(1)> {
  instance inst0(tmux<1, impl test>), //because tmux is an implement template, and test is an implement of sl0<2, type Bit(1)>, so it's acceptable here. TODO: But "impl test" uses "self" as a template argument which is not allowed. Current Tydi-lang didn't check it but the restriction will be applied in the future.
};
```

Template arguments checking rules

```
impl implement<ts: impl of [STREAMLET_NAME1]> of [STREAMLET_NAME2]
//ts will be an abstract implement type
//when instantiating the implement, the "ts" must be an implement of [STREAMLET_NAME1]. If the [STREAMLET_NAME1] is an instantiated streamlet, the "ts" also have the same streamlet template arguments.
```

Tydi-lang front end

All streamlet, implement templates will not present in the final Tydi IR. All transformations will be carried on by the following procedures (for future Rust version).