Karuta Documentation

Yusuke Tabata

Jan 09, 2021

Index and topics:

Karuta is an object-oriented scripting language and its compiler to design logic circuits. The main objective of Karuta is to make logic circuit design more productive. This kind of software is usually known as HLS (High Level Synthesis).

Author: Yusuke Tabata (tabata.yusuke@gmail.com)

NOTE: There may be mistakes or glitches in this document due to my English skill. Please feel free to point out (or ignore...) them.

Source of this projects: <https://github.com/nlsynth/karuta> (see its docs/ directory for this document)

CHAPTER 1

Quick Introduction

1.1 Installation

If you are using Ubuntu, just do

```
$ sudo snap install karuta
```

Otherwise please see User's guide to build the binary.

1.2 Tutorial

This section illustrates some of Karuta's features using Xorshift32 method. A simple Xorshift32 code in Karuta is like this:

```
process main() {
  var y int = 1
  for var i int = 0; i < 10; ++i {
    y = y ^ (y << 13)
    y = y ^ (y >> 17)
    y = y ^ (y << 15)
    print(y)
  }
}
```

Save this to a file named xorshift32.karuta, then you can run this program like

```
$ karuta run xorshift32.karuta
print: default-isynth.karuta loaded
print: 268476417
print: 1157628417
print: 1158709409
 ...
```

I guess this looks pretty mundane to you, so let's start hardware design. So you can the run same *karuta* command again and will get xorshift32.v. The content should look like this.

```
$ karuta compile xorshift32.karuta
```

```
... 100~ lines of code in Verilog here. ...

module xorshift32(clk, rst);
  input clk;
  input rst;
  xorshift32_main xorshift32_main_inst(.clk(clk), .rst(rst));
endmodule
```

Then you can run this on a Verilog simulator with a testbench file to feed the clock and reset. If you think it is tedious, *sim* command will do most of the work;

```
$ karuta sim xorshift32.karuta
```

The output contains an enclosing module to generate clock and reset. So you can simply run iverilog (or your simulator) to see the results.

```
$ iverilog xorshift32.v
$ ./a.out
 268476417
1157628417
1158709409
 269814307
...
```

OK. Looks good? But there is a big problem. The code uses *$display()* which is useless on real FPGAs. The next step is to generate an output port from this design. With Karuta, you can annotate a method to make it an output port. The output value is updated when the method is called. For example,

```
output o int

process main() {
  var y int = 1
  for var i int = 0; i < 10; ++i {
    y = y ^ (y << 13); y = y ^ (y >> 17); y = y ^ (y << 15)
    o.write(y)
  }
}
```

The code above will be converted to a Verilog file like as follows. The top module xorshift32 has an output port *'o'*, so you can connect the port to other parts of your design.

```
... 100~ lines of code in Verilog here. ...

module xorshift32(clk, rst, o);
  input clk;
  input rst;
  output [31:0] o;
  mod_main mod_main_inst(.clk(clk), .rst(rst), .o(o));
endmodule
```

This can be tidied up a bit by factoring out update formulas.

```
// Member variable of the default object for this file.
shared y int

output o int

// Gets an argument t and returns an update value.
func update(t int) (int) {
  t = t ^ (t << 13); t = t ^ (t >> 17); t = t ^ (t << 15)
  return t
}

process main() {
  y = 1
  while true {
    y = update(y)
    o.write(y)
  }
}
```

The last example here illustrates some of the most important features of Karuta such as multiple threads and channels.

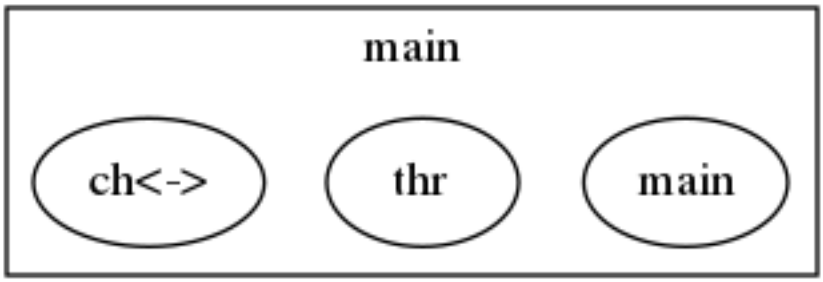
```
// Enclosing module { ... } is optional here and just to make it clear
// following members are in this module.
module {
  // This channel can be accessed like ch.write(v) or v = ch.read()
  channel ch int

  func update(t int) (int) {
    t = t ^ (t << 13); t = t ^ (t >> 17); t = t ^ (t << 15)
    return t
  }

  // Thread entry method.
  process main() {
    var y int = 1
    while true {
      y = update(y)
      ch.write(y)
    }
  }

  output o #0

  // Thread entry method.
  process thr() {
    var b #0 = 0
    while true {
      var v int = ch.read()
      // Flip the output on-off value when the generated random number is
      // below this number.
      if v < 10000 {
        b = ~b
        o.write(b)
      }
    }
  }
}
```

This code has 2 thread entry methods. One generates random numbers and the another reads the numbers via a channel. When the code is compiled, generated Verilog code will have 2 state machines (*'always'* blocks). You can deploy the code to an FPGA board, connect the output to an LED and see it flickers randomly.

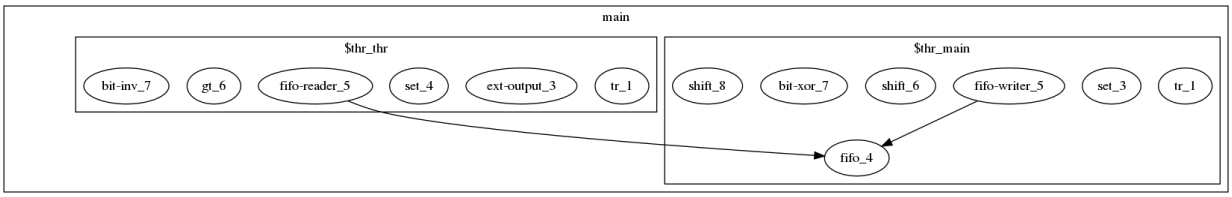
Did this work well? I hope you got the idea of Karuta's approach to hardware design.

1.2.1 Visualize the design

Karuta has features to visualize designs. They will help you to understand or explain complex designs in Karuta. One is to visualize the structure of objects in Karuta language.



Another one is to visualized the structure of modules and FSMs after synthesis.



1.3 Features

Karuta's 10 important features you might like...

- New scripting language with contemporary syntax designed primarily for hardware design
- Prototype based OOP
- Static thread concurrency
- Channels and mailboxes
- Attach AXI DMA controller to arrays
- Distance between objects can be specified
- Custom data width and numeric operations
- Interfaces to/from circuits outside
- HDL embedding
- Optimization techniques like PGO and SSA

CHAPTER 2

User's guide

Contents

2.1 Run or Compile

Assuming that you have a design like this and save to a file my_design.karuta

```
process thr() {
  // Possibly communicate with main() and other threads.
}

@Soft
process testThr() {
  // Code to generate stimulus to other threads.
  // (NOTE: a thread with @Soft will not be synthesized)
}

func f() {
  // Code which can be called from other methods or processes.
}

process main() {
  // Does interesting computation.
```

(continues on next page)

(continued from previous page)

```
}
```

Karuta generates synthesizable Verilog file, if *compile* command is specified.

```
$ karuta compile my_design.karuta
... karuta will output my_design.v
```

This is equivalent to call compile() and writeHdl("my_design.v") at the end of the code.

To run the threads described in the code, *run* command can be used. The example above has 3 thread entries *thr()*, *testThr()* and *main()*, so it will make 3 threads and wait for them to finish (or timeout).

```
$ karuta run my_design.karuta
... thr(), testThr() and main() will run
```

This is equivalent to call *run()* at the end of the code.

2.2 Language concepts

2.2.1 Process and function

```
// A function can be called by other functions and processes.
func f1() {
  // code
}

// Process is an entry point function starts to run from the beginning.
process p1() {
  // code
}

// 'always' is a syntax sugar to denote a process to repeat indefinitely.
always p2() {
  // code
}
// code above is equivalent to
process p2() {
  while true {
    // code
  }
}
```

2.2.2 Default file object

Karuta allocates an object for each source file and the object is used as the default object while executing the code in the file. The default object can be omitted or explicitly denoted as *self*.

```
// All self. are optional in this example.
reg self.m int
process self.main() {
}
```

(continues on next page)

(continued from previous page)

```
self.compile()
self.writeHdl("my_design.v")
```

2.2.3 Prototype-based object system

Karuta adopts prototype-base object oriented programming style. A new object can be created by cloning an existing base object and user's design is described by modifying object(s).

```
// Temporary object.
var o object = new()

// Adds 2 method f() and g()
func o.f() {
  print(g())
}
func o.g() (int) {
  return 1
}

// Makes 2 clones of the object `o` and set them as member objects of `self`.
shared self.o1 object = o.clone()
shared self.o2 object = o.clone()

// Modifies one of them a bit.
func o2.g() (int) {
  return 2
}

// `self` can access 2 objects and their methods.
func self.main() {
  o1.f()
  o2.f()
}
```

2.2.4 Member variables

Karuta is an object oriented language, so a design can be described as objects and their members. `shared`, `reg` and `ram` keyword is used to declare an member value of an object, integer or array (other kinds of member has different syntax).

```
// Just `shared o object` without `self.` is also ok.
shared self.o object = new()
// This declares a member of a member `o`.
reg self.o.v int

process self.main() {
  // Accesses a member of a member.
  o.v++
}

process self.o.f() {
  v = 0
}
```

2.2.5 Integer width

Bit width of data is important to use FPGAs efficiently while it is not cared so much for CPUs. Karuta allows arbitrary bit width.

```
// Variable declarations.
var x int  // default width is 32 bits.
var rgb #24  // specify 24 bits.

// This function takes a 32 bits argument (arg) and returns a 32 bits argument.
func bswap32(arg #32) (#32) {
  // [h:l] - bit slice operator
  // ::    - bit concatenation operator
  return arg[7:0] :: arg[15:8] :: arg[23:16] :: arg[31:24]
}
```

(Karuta also has features for user defined types (e.g. bfloat16). Document will be added later.)

2.2.6 Arrays

Arrays are really important to utilize FPGA, so Karuta has features to use arrays efficiently.

```
ram arr int[16]

func f(idx int) (int) {
  // This index wraps around by 16.
  return arr[idx - 1] + arr[idx] + arr[idx + 1]
}
```

One important difference from Karuta and other languages is that an array index wraps around by the length of the array.

Array images

Array images can be written to a file or read from a file.

```
ram arr int[16]

arr.saveImage("arr.image")
arr.loadImage("arr.image")
```

2.2.7 Threads

Method can be declared as a thread entry. A thread will be created when the code is executed or synthesized.

```
func f() {
  // Just a method.
}

func main() {
  // main() is automatically treated as a thread entry.
}
```

(continues on next page)

(continued from previous page)

```
process m1() {
  // This method will run as a thread.
}

@ThreadEntry
func m2() {
  // @ThreadEntry annotation starts the method as a thread entry.
}
```

2.2.8 Channel and mailbox

Communication between threads is really important for circuit design. While one simple way of communication is just to use shared registers or arrays, Karuta also supports channel and mailbox to communicate between threads.

This example this just write values and read them from other threads.

```
channel ch int

process th1() {
  ch.write(1)
  ch.write(1)
}

process th2() {
  ch.read()
}

// channel can be written or read by arbitrary number of threads.
process th3() {
  ch.read()
}
```

A mailbox is just a channel with one value.

```
mailbox mb int

process th1() {
  mb.put(1)
}

process th2() {
  mb.get()
}
```

But it can notify waiting threads.

```
mailbox mb int

process th1() {
  mb.notify(10)
}

process th2() {
  print(mb.wait())
}
```

2.3 Communication to external

2.3.1 I/O object

I/Os (e.g. LEDs, DIP switches, interrputs and so on) can be accessed with member variabels with *input* or *output* .

```
input i int
output o int
@(name="led")
output o2 #0

process p() {
  print(i.read())
  o.write(123)
  o2.write(1)
  // peek() returns previously written value.
  print(o2.peek())
  // wait() waits for the input value to change.
  print(i.wait())
}
```

2.3.2 I/O method

Another way to access I/Os is to annotate a method with *@ExtIO* annotation. Its argument when called is taken as the output value and return value is taken from the input value.

```
@ExtIO(output = "o")
func L.f(b bool) {
}

@ExtIO(input = "i")
func L.g() (bool) {
  return true
}
```

2.3.3 Mailbox writer

mailbox can be configured to accept writes from an external accessor.

```
// Signals "name", "name_wen", "name_notify", "name_put" and "name_put_ack"
// are genrated.
@Export(name="name", wen="wen", notify="notify", put="put")
mailbox mb int

process p1() {
  mb.wait()
}

process p2() {
  print(mb.get())
}
```

2.3.4 AXI interface

Either AXI master or slave interface can be attached to each array.

AXI master

When an array is declared with AXI master annotation, we can transfer data to/from external memory from/to the array by calling methods of the array.

```
// @AxiMaster(addrWidth = 64) // or 32 (default) to specify the width.
// @AxiMaster(sramConnection = "shared") // or "exclusive" (default).
@AxiMaster()
ram m int[16]

func f() {
  m.load(mem_addr, count, array_addr)
  m.store(mem_addr, count, array_addr)
}
```

AXI slave

When an array declared with AXI slave annotation, an AXI slave interface to outside of the design is generated and we can access the array from outside.

```
@AxiSlave()
ram s int[16]

func f() {
  while true {
    s.waitAccess()
    // Do something on access.
  }
}
```

`notifyAccess()` method can be used for testing.

2.3.5 SRAM interface

Internal SRAM

Similar to AXI slave interface, SRAM interface which can be accessed from outside of the design can be attached to a RAM.

```
@SramIf // or @Export
ram s int[16]
```

External SRAM

Antoher way to declare external RAM is to use @External annotation.

```
@External(name="sram")
ram r int[16]

process main() {
  r[0] = 123
}
```

The code above will generate sram interface ports

```
output reg [7:0] sram_s_addr,
input [3:0] sram_s_rdata,
output reg [31:0] sram_s_wdata,
output reg sram_s_wdata_en,
```

2.3.6 Method interface

Karuta supports the Method Interface <https://gist.github.com/ikwzm/bab67c180f2f1f3291998fc7dbb5fbf0> to communicate with external circuits.

```
// f() will be callable outside of the design.
@ExtEntry(name="e")
func f(x int) (int) {
  return 0
}

// Actual implementation of f() will be outside of the design.
@ExtStub(name="e")
func f(x int) (int) {
  return 0
}
```

2.3.7 Embedded combinational logic

A combinational logic in a Verilog module can be embedded in a function of Karuta by specifying the file name and module name by @ExtCombinational annotation.

```
@ExtCombinational(resource = "a", verilog = "resource.v", file="copy", module="my_
↪logic")
func f(x #32) (#32) {
  // This code is used by the interpreter, but Verilog module in resource.v
  // is used in synthesized code.
  return x + 1
}
```

Embedded Verilog module has input arguments arg_0, arg_1,, arg_N and output arguments ret_0, ret_1,, ret_N. The number of inputs and outputs should match with the original function.

```
module my_logic(input clk, input rst, input [31:0] arg_0, output [31:0] ret_0);
  assign ret_0 = arg_0 + 1;
endmodule
```

2.4 Basic features

2.4.1 Method call

```
shared m object = new()
func m.f() {
}

func g() {
}

process th1() {
  // Does handshake and arbitration
  m.f()
  // Inlined for this thread.
  g()
}

process th2() {
  // Does handshake and arbitration
  m.f()
  // Different inlined instance for this thread.
  g()
}
```

2.4.2 Type object

Karuta allows to implement user defined numeric types. An object describes user define numeric operations can be attached to each numeric declaration.

```
shared Numerics.Int32 object = Object.clone()
func Numerics.Int32.Build(arg #32) (#32) {
  return arg
}

func Numerics.Int32.Add(lhs, rhs #32) (#32) {
  return lhs + rhs
}

// NOTE: Type object can't be accessed from top level environment.
func f() {
  var x #Int32
  x = Numerics.Int32.Build(1)
  print(x + x)
}

// Add a method for the type.
func Numerics.Int32.IsZero(arg #32) (bool) {
  return arg == 0
}

func g() {
  var x #Int32
  x = Numerics.Int32.Build(1)
```

(continues on next page)

(continued from previous page)

```
  print(x.IsZero())
  x + x
}
```

2.4.3 Thread local variable

Multiple threads can be created from an entry method by specifying *num*= parameter.

```
@ThreadLocal()
shared M.x int

@(num=2)
process M.thr(idx int) {
  // 2 copies of this thread runs and the index is given as the method
  // argument. idx = 0, 1.

  // x is a per thread variable.
  x = x + idx
}
```

Custom data type with Verilog

Type object and embedded combinational logic can be used to build a custom type with staged operations (e.g. FP16, complex num, RGB and so on).

```
func Numerics.MyType.Add(lhs, rhs #32) (#32) {
  // 3 stage (clocks) operation.
  return add_st3(add_st2(add_st1(lhs, rhs)))
}

@ExtCombinational(resource = "my_type", verilog = "my_type.v", file="copy", module=
↪"my_logic_st1")
func add_st1(lhs, rhs #32) (#32, #32) {
  return rhs, lhs
}
// add_st2 and add_st3 here.
```

2.4.4 Building object hierarchy

The basic way to build an object hierarchy is to add new member objects and modify them.

```
shared x object = new()
shared x.y object = new()
func x.f() {
  y.g()
}
func x.y.g() {
  print(1)
}
func main() {
  x.f()
}
```

This structure can be a more cleanly described with `module` block.

```
shared x object = new()
shared x.y object = new()
module x {
  module y {
    func g() {
      print(1)
    }
  }
  func f() {
    y.g()
  }
}
func main() {
  x.f()
}
```

When `module` block is used, the member object can access its enclosing object by `parent` keyword.

```
shared x object = new()
module x {
  func f() {
    parent.g()
  }
}
func g() {
}
```

```
// Just use the default object.
module {
  ...
}

// Member object name or base file name of current file
// (e.g. "m" for "m.karuta").
module m {
  ...
}
```

2.4.5 Loop Unrolling

A *for* loop with fixed loop count can be unrolled by specifying the number of copies.

```
@(num=2)
for var i int = 0; i < 8; ++i {
  // does computation
}
```

WIP.

```
@Pipeline(num=2)
for var i int = 0; i < 8; ++i {
  // does computation
}
```

2.4.6 Profile Guided Optimization (PGO)

One of the most important points of optimization is to know which part of the design is a good target of optimization. A technique called PGO (Profile Guided Optimization) can be used to obtain the information.

Following example illustrates how to enable profiling. Profiling is enabled between the calls of *Env.enableProfile()* and *Env.disableProfile()*, so the profile information will be collected while running main(). compile() takes the profile information into account to perform optimization.

```
process main() {
  // Does some computation and I/O.
}

Env.clearProfile()
Env.enableProfile()

// Run actual code here on the interpreter.
main()

Env.disableProfile()

compile()
writeHdl("my_design.v")
```

2.4.7 Importing file

```
// Just reads and executes the file.
import "filename_1.karuta"

// Reads the file and assigns a local variable `m`.
import "filename_2.karuta" as m

// Now you can access m.
m.dump()
```

2.4.8 Object distance

Elements of designs are placed onto the physical area of an FPGA and there are physical distances between them. So Karuta has a feature to specify number of clocks to propagate signals for communication.

```
// Object distance between `self` and `m` is 10 clocks.
@_(distance=10)
shared self.m object = new()
reg self.m.v int

func self.m.f() {
  v = v + 1
}

func self.f() {
  // These takes 10(+basic overhead) clocks.
  m.v = 1
  m.f()
}
```

2.4.9 Clock ticker

A ticker object keeps a counter incremented by the clock. Each ticker object has *.getCount()* method to get current count and *.decrementCount(n)* to decrement the value.

```
module {
  shared ticker object = Env.newTicker()
  output o #0

  process p1() {
    // Resets the counter to (almost) 0.
    ticker.decrementCount(ticker.getCount())
    var b #0 = 0
    while true {
      o.write(b)
            b = ~b
            // Makes 10000 clocks interval.
            wait(10000 - ticker.getCount())
            ticker.decrementCount(10000)
    }
  }
}
```

2.4.10 Testing

Features for object oriented programming can be used to test designs as well. One key idea is to create an enclosing tester object for the design (There may be other ways).

```
// design.karuta
func f(arg int) (int) {
  return arg + 1
}
```

```
// test.karuta
// imports the design file and assigns the object to a local object `d`.
import "design.karuta" as d

// assigns to a member object.
shared design object = d

func main() {
  assert(design.f(10) == 11)
}

run()
```

2.4.11 Visualize designs

Karuta can visualize following 3 aspects of input designs.

(1) Structure of objects in Karuta.

(2) Structure of modules and FSMs.

(3) Details of each FSM.

Output can be either in HTML or DOT (format for Graphviz <https://www.graphviz.org/>)

Type	Format	Usage
Structure of objects	DOT	–dot option and call synth()
Structure of modules and FSMs	DOT	writeHdl() with file name .dot
Details of each FSM	HTML	writeHdl() with file name .html

- Structure of objects is generated when the script calls synth() method if command line option '–dot' is specified.

```
# synth() is called in design.karuta
$ karuta design.karuta --dot
# karuta generates design.0.dot file. Use 'dot' command to generate a png image file.
$ dot -Tpng design.0.dot -o design.png
```

- Structure of modules and FSMs and (3) Details of each FSM can be generated by specifying appropriate file name suffix.

```
// Outputs Verilog.
writeHdl("design.v")
// Outputs (2) Structure of modules and FSMs in DOT format.
writeHdl("design.dot")
// Outputs (3) Details of each FSM in HTML format.
writeHdl("design.html")
```

2.4.12 Platform description

Karuta can specify the name of target hardware to use its specific parameters.

```
// Default parameters. Platform defintion for actual chips will be available.
setSynthParam("platformFamily", "generic-platform")
setSynthParam("platformName", "default")
```

2.4.13 Using generated Verilog file

Each output Verilog file will have one top module with the basename of the output file name (e.g. *abc* for *abc.v*).

Each output Verilog file also contains placeholder code to instantiate the design for users' convenience.

```
// NOTE: Please copy the follwoing line to your design.
// abc abc_inst(.clk(), .rst(), ... other ports ...);
```

If the design is an AXI IP on Vivado, –flavor=vivado-axi option to karuta command can be used to add corresponding wire names like .s00_AWSIZE(s00_axi_awsize).

2.5 Installation

If you are using Ubuntu, just do

```
$ sudo snap install karuta
```

Installing Karuta from its source code requires a C++ compiler (namely g++ or clang++), python, gyp-next and make.

NOTE: gyp is a Makefile generator. Please use maintained gyp-next instead of the original gyp.

```
$ pip install gyp-next
install pip on Debian or Ubuntu.
$ sudo apt install python3-pip
```

```
# Get the source code.
$ git clone --recursive https://github.com/nlsynth/karuta

# Do build.
$ ./configure
$ make

# Compile an example.
$ cd examples
$ ../karuta top.karuta

# Test the output from the example.
$ iverilog tb_top.v top.v
$ ./a.out
```

CHAPTER 3

Reference

Contents

3.1 Command line flags

$ karuta <command> [Files]

- Commands(optional)
  - run - starts runnable threads (same as –run)
  - compile - compiles objects for specified files (same as –compile)
  - sim - same as `sim` and also with self contained shell (same as –compile and –with_shell)

- Debug options
  - -db debug byte code compiler
  - -dp debug parser
  - -ds debug scanner
  - -dt debug types
- -l
  - Enables info logging.
  - Comma separated list of modules to enable for specific files.
- –compile
  - Compiles the file object and writes to a Verilog file.
  - Calls compile() and writeHdl(name.v) at the end of execution.
- –duration
  - Maximum duration of the simulation.
- –iroha_binary [binary]
  - Specifies an alternative iroha binary.
- –module_prefix=[mod]
  - Module name prefix.
  - File name without suffix (“a” for “a.v”) will be used if this is not specified.
  - This can be used to get fixed module name for testing.
- –output_marker=[marker]
  - Marker string to be output before output file name.
  - Karuta server uses this to generate links from output log.
- –flavor=[flavor]
  - Sets a specific flavor to the generated HDL files.
  - e.g. –flavor=vivado-axi
- –print_exit_status
  - Shows exit status at the end of execution.
  - Test uses this to check if karuta isn’t aborted.
- –root
  - Prefix for file output name.
  - Karuta server uses this to isolate each run.
- –run
  - Runs every runnable threads in the source file.
  - Calls run() at the end of execution.
- –timeout
  - Timeout of karuta command execution.

  - Avoid infinite loop to run forever for test or Karuta server.
- –vanilla
  - Doesn't read lib/default-isynth.karuta.
- –vcd
  - Generates vcd from generated verilog file.
- –version
  - Print version number.
- –with_shell
  - Generates a top level module to feed clock and reset.

3.2 Program structure

A program is comprised of top level environment and method environment. Methods and various things can be declared only in top level environment.

```
// * Top level environment
// Modifying objects by declaring member variables.
//
func f() {
  // * Method environment
  // Object modification is not allowed (hence synthesizable).
}
```

3.2.1 Methods

A method is declared as a member of an object. If no object name is given in the declaration, the method will belong to the current file object.

```
// This method belong to the current file object.
func f() {
}

@Annotation(key1="value1", key2="value2")
func mod.f(x, y #32) (#16, #16) {
  return (0, 0)
}
```

A declaration can have an argument list and return value list. A declaration begin with a keyword either *func* or *process*.

3.2.2 Variable declaration

Syntax to declare a variable is: [variable type] [name] [data type] e.g. ram a int[16]

Variable type	Data type	Note
shared	scalar or array	Equivalent to ram or reg
ram	array	BRAM/SRAM
reg	scalar	reg
mailbox	scalar	mailbox
channel	scalar	channel
var	scalar	Local variable

Data types of a scalar variable are:

- int - 32 bits
- #0 - boolean
- #10 - 10 bits
- #T - User define type T
- object, module - Object

A vector type is declared by [scalar type][length] like int[16].

```
// top level local variable.
var x int
// member variable.
reg y int
reg self.z int

func f() {
  // method local variable
  var a int
  a = y + z
}
```

3.2.3 Basic Syntax

Comments

```
// Comment
/* Comment too */
# is alloed at the beginning of a file. This is for #! for shells.
```

Literals

```
// Just a number.
123
0xf00d
0b1010
// A number with explicit width
123#32
// string
"abc"
```

Method definition

```
// func name(arguments) (return values) { ... }
// (return values) can be omitted if there is no arguments.
func funcName(arg1, arg2 #16, arg3 int) (int, int) {
  return arg1, arg3
}
```

Declarations

```
//
var x int
var x #32
var x #MyType
var x object
//
var x, y int
var x int = 0
//
channel c int
mailbox m int
//
var s string = "abc"
// var for an array is not allowed
shared a int[32]
shared m.a #16[4] = {1,2,3,4}
```

Expressions

```
//
name
__name // reserved for the implementation
//
a + b
a - b
a * b
a = b
a, b
(a)
f(x)
a = f(x,y)
(a, b) = f(x,y)
obj.a
obj.f()
```

Operators

```
a + b
a - b
a * b
// TODO: Describe the limitations
a / b
// shift amount should be constant
a >> b
a << b
// bit concat
a :: b
// range should be constant
a[l:r]
```

Statements

```
if a > b {
} else {
}

for var x = 0; x < 10; ++x {
}
```

Control

```
if cond {
}

for init; cond; update {
}
```

TODO: switch/case statement

module block

```
module {
  shared obj object = new()
  func bar() {
    obj.x[0] = 0
    obj.foo()
  }
}

module obj {
  // same as obj.foo()
  func foo() {
  }
  // same as obj.x
  shared x int[16]
}
```

3.3 Built in objects and methods

3.3.1 Objects

- Object
  - Array - Internal SRAM
  - Env - Profiling
  - Global - Placeholder for global variables
  - Kernel - Base object for user's design
  - Memory - External SRAM
  - Numerics - Registry for user defined types.

3.3.2 Built in methods

- Object.clone()
- Object.dump()
- Object.run()
- Object.bool
- Object.false
- Object.true
- Object.Object
- Kernel.assert()
- Kernel.abort()
- Kernel.compile()
- Kernel.exit()
- Kernel.new()
- Kernel.print()
- Kernel.runIroha()
- Kernel.setDump()
- Kernel.setIROutput()
- Kernel.setIrohaPath()
- Kernel.setSynthParam()
- Kernel.synth()
- Kernel.wait()
- Kernel.widthof()
- Kernel.writeHdl()
- Kernel.yield()
- Kernel.Kernel_
- Kernel.Object
- Kernel.Module
- Kernel.parent
- Kernel.self
- Kernel.Array
- Kernel.Env
- Kernel.Global
- Kernel.Memory
- Kernel.Numerics
- Env.gc()
- Env.clearProfile()

- Env.disableProfile()
- Env.enableProfile()
- Env.isMain()
- Array axiLoad, axiStore, waitAccess, notifyAccess, saveImage, loadImage, read, write
- Memory setWidth
- Channel write, writeFast, read
- Mailbox width, put, notify, get, wait
- .$compiled_module
- .$dump_file_name
- setIrohaPath(p string)
- setIROuput(p string)
- runIroha(opts string)
  - e.g. runIroha("-v -S -o x.v")

3.4 Synthesis parameters

When compilation is requested by calling compile() method, the synthesizer takes a snapshot of the object and generates IR from the structure and computation.

```
setSynthParam("resetPolarity", 0) // set negative reset (e.g. rst_n).
setSynthParam("maxDelayPs", 10000) // 10ns
setSynthParam("platformFamily", "generic-platform")
setSynthParam("platformName", "generic")
```

3.5 Annotations

```
// Annotation for a method
@ThreadEntry() @ProcessEntry() @Process()
@SoftThreadEntry() @SoftProcess() @Soft()
@ExtEntry()
@ExtStub()
@ExtIO()
@ExtCombinational()
@DataFlowEntry()  // Might be removed
@ExtFlowStub()  // Might be removed
// Annotation for an array
@AxiMaster() @AxiMaster64() @AxiMaster32()
@AxiSlave() @AxiSlave64() @AxiSlave32()
@ThreadLocal() @ProcessLocal() @Local()
// channel parameters
depth=
// object parameters
distance=
// thread number
num=
```

CHAPTER 4

Design and Implementation

Contents

4.1 Overview of Karuta

As readers might know, there have been a good amount of efforts to improve efficiency to design digital circuits. One of the most significant achievement in this area is the introduction of RTL and languages which can describe in RTL. The introduction of RTL was so successful and most of recent designs are done in RTL.

Karuta is one of the efforts to make some of circuit designs more efficient. Karuta is a new programming designed for this purpose and its compiler. The language introduces higher level abstraction than RTL (so called HLS).

4.1.1 Why new language?

While most of other attempts to introduce higher level abstraction adopt existing programming languages for software, Karuta project designs a new language. This is because I believe following things:

1. Different assumptions on underlying hardware.
    - Most of languages for software have some assumptions that they run on CPUs with an instruction pointer, one global address space and so on.
2. Reuse is not easy.
    - Reusing an existing language still requires to get familiar with different semantics from that for software, while it is said to save efforts to learn a new language.
    - Anyway many of users have to learn a new language because there is no lingua franca nowadays.
3. Own language will make it easy to experiment new ideas and features.
    - It is also fun!

4.1.2 Considerations

With above hypotheses, Karuta's design took following considerations.

- Concurrency and communication

Use of concurrency is an essential issue for hardware designs. A whole design is placed over the area of an FPGA and computation can happen anywhere on it. So Karuta aims to make it easy to describe such a behavior by threads.

A thread on software is typically a natural unit of computation from its beginning to end and can be assigned to a CPU when it is available. This also means any thread with any computation can be assigned to a CPU.

On the other hand, Karuta assumes a thread is a piece of computation and corresponds to an FSM instance on an FPGA. So a whole design will consist of threads and their communications.

- Object system

To choose features to support object oriented programming, language designers usually have to consider two issues; (1) Users' convenience and (2) Runtime efficiency. So we have to be careful about the differences of these conditions for hardware design from software design.

Most of software languages uses class based object oriented programming. It works really well by assigning each class to code and each instance to data. On the other hand, Karuta's hypothesis is prototype based object oriented programming seems to fit better for hardware design. Karuta assumes it is more intuitive to place code as an FSM and make member variables as registers or RAMs.

- Arrays

Efficient use of RAM is also crucial for programs on FPGAs. While most of software runtime assumes one globally shared memory among threads, FPGAs allow to put RAMs near the place where the actual computation is done. This is important to achieve higher performance and energy efficiency.

Karuta allows to add arrays as member of an object and maps them to RAMs at synthesis.

- Data types

CPUs, GPUs and most of accelerators have fixed set of data width and operators. On the other hand, FPGAs can allow arbitrary set of data width and operators. Fixed data types can be inefficient if narrow data width or subset of operators are enough for the purpose. So Karuta is aiming to design to allow flexible data types.

In addition to allow arbitrary width, Karuta allows to define custom operators to variables. This makes it easy to implement data types like narrower/wider precision floating/fixed points or vector-ish data types like complex numbers, RGB or so on.

- Interface to outside of the design

Every design has some kinds of I/Os like master or slave interface of a certain bus protocol, GPIO, handshake or so on. Karuta supports them by annotations to a method or an array.

4.1.3 Compiler and interpreter

Once the behavior of a design is written in Karuta language, `karuta` command can execute it as an interpreter or generate the RTL description of the behavior. So, users can simulate and test the behavior on Karuta's interpreter first before implementing it on real FPGAs.

4.1.4 Simple and familiar syntax

Karuta adopts syntax similar to recently popular programming languages. For example, declaring a variable is like `var x int = 123`. It also allows multiple return values from a function as other languages do. Karuta also takes some syntax from HDLs so bits can be sliced like `x[15:8]` and concatenated like `x :: y`.

4.1.5 Use of IR

Karuta adopts Iroha (Intermediate Representation Of Hardware Abstraction) as its IR and backend which borrowed concepts from LLVM. Karuta generates Iroha based IR and Iroha takes it as its input, optimizes and writes out HDL files.

4.1.6 HDL embedding

Trying to design everything in one HLS language is a terrible goal, so Karuta has features to embed Verilog code in users' design.

4.1.7 Synthesis friendly HDL

Karuta implements following features to support large scale hardware.

Object distance

Elements of design can be placed in distant positions in a chip, so Karuta lets users specify the latency between objects manually. This can allow place and route tool to work with reasonable constraints.

Tree of MUXes

Karuta aims to support designs with many FSMs and resources shared between them. To support many accessors to one resource, Karuta generates a tree of multiplexers to arbitrate accesses. The tree structure avoids too deep priority logic.

4.2 Architecture and source code structure

Karuta parses an input source code file, builds an AST (fe::Method), then compiles (by compiler::Compiler) into a bytecode sequence (vm::Method). A bytecode sequence can be executed (by vm::Executor) or synthesized to HDL.

To synthesize HDL from a bytecode sequence, Karuta uses Iroha library. Karuta builds Iroha's data structure (iroha::IDesign) and dumps the results into a file, then it invokes iroha command to perform some optimizations and conversion to HDL.

- src/
  - main.cpp and build related files.
- src/base
  - Basic utility code for other components.
- src/compiler
  - Karuta Script to bytecode.
- src/fe
  - Karuta Script parser.
- src/karuta
  - Common definitions for Karuta.
- src/synth
  - Bytecode to Iroha IR.
- src/vm
  - Bytecode executor.
- iroha/
  - Iroha backend.

CHAPTER 5

Examples (WIP)

Examples in this page are to understand concepts of Karuta language quickly (I hope everything is within your familiar concepts and paradigms).

Contents

5.1 Just a module

```
module {
  reg r int
  ram a int[16]

  process p() {
    a[r]++
  }
}
```

module keyword specifies an object and allows to define members inside it.

5.2 Communication

```
module {
  channel ch int

  process p1() {
    ch.write(1)
  }

  process p2() {
    ch.read()
  }
}
```

channels, regs, rams and most of member objects are accessible from multiple threads.

5.3 Module hierarchy

```
module {
  shared m module = new()

  process p() {
    m.ch.write(1)
  }
}

module m {
  channel ch int
  process p() {
    p.ch.read()
  }
}
```

5.4 Matrix multiplication

```
module m {
  ram a int[4][4]
  ram b int[4][4]
  ram x int[4][4]

  func mult() {
    for var i int = 0; i < 4; ++i {
      for var j int = 0; j < 4; ++j {
        var t int = 0
        for var k int = 0; k < 4; ++k {
          t += a[i][k] * b[k][j]
        }
        x[i][j] = t
      }
    }
  }
  process p() {
```

(continues on next page)

(continued from previous page)

```
    mult()
  }
}
```

CHAPTER 6

Experimental features

Contents

Karuta has some experimental and premature features to see their feasibility and implementation difficulty. This section describes such features.

They might be remove without notice, probably due to low convenience against high maintenance burden.

6.1 Data flow

```
@DataFlowEntry()
func f(x int) {
}
```

```
@ExtFlowStub(name="e")
func f(x int) (int) {
}
```

6.2 Thread member declaration

Threads can be declared as a member object instead of @ThreadEntry().

```
func f() {
}
thread Kernel.thr1 = f()
run()
```

6.3 External SRAM operations

Memory object represents an address space and can be accessed by read/write method. This assumes 32bit address/data for now.

```
Memory.read(addr)
Memory.write(addr, data)
```

6.4 Multi dimensional arrays

Multi dimensional array is just an additional view of a normal 1 dimensional array.

```
shared a int[4][4]

func main() {
  x[1][2]
}
```

CHAPTER 7

Glossary

- DFG
  - Data Flow Graph to represent internal RTL
- Bytecode
  - Compiled from Karuta Light Script and can be executed or synthesized
- Iroha
  - Intermediate Representation Of Hardware Abstraction
  - https://github.com/nlsynth/iroha
- Karuta
  - Name of Japanese playing cards.
  - This package.
- Soft thread
  - Threads which will not be synthesized (used for tests).