Migen manual

Release 0.5.dev0

Sebastien Bourdeauducq

Oct 08, 2017

Contents

CHAPTER 1

Introduction

Migen is a Python-based tool that aims at automating further the VLSI design process.

Migen makes it possible to apply modern software concepts such as object-oriented programming and metaprogramming to design hardware. This results in more elegant and easily maintained designs and reduces the incidence of human errors.

Background

Even though the Milkymist system-on-chip *[mm]* had many successes, it suffers from several limitations stemming from its implementation in manually written Verilog HDL:

1. The "event-driven" paradigm of today's dominant hardware descriptions languages (Verilog and VHDL, collectively referred to as "V*HDL" in the rest of this document) is often too general. Today's FPGA architectures are optimized for the implementation of fully synchronous circuits. This means that the bulk of the code for an efficient FPGA design falls into three categories:

   (a) Combinatorial statements

   (b) Synchronous statements

   (c) Initialization of registers at reset

   V*HDL do not follow this organization. This means that a lot of repetitive manual coding is needed, which brings sources of human errors, petty issues, and confusion for beginners:

   (a) wire vs. reg in Verilog

   (b) forgetting to initialize a register at reset

   (c) deciding whether a combinatorial statement must go into a process/always block or not

   (d) simulation mismatches with combinatorial processes/always blocks

   (e) and more...

A little-known fact about FPGAs is that many of them have the ability to initialize their registers from the bitstream contents. This can be done in a portable and standard way using an "initial" block in Verilog, and by affecting a value at the signal declaration in VHDL. This renders an explicit reset signal unnecessary in practice in some cases, which opens the way for further design optimization. However, this form of initialization is entirely not synthesizable for ASIC targets, and it is not easy to switch between the two forms of reset using V*HDL.

2. V*HDL support for composite types is very limited. Signals having a record type in VHDL are unidirectional, which makes them clumsy to use e.g. in bus interfaces. There is no record type support in Verilog, which means that a lot of copy-and-paste has to be done when forwarding grouped signals.
3. V*HDL support for procedurally generated logic is extremely limited. The most advanced forms of procedural generation of synthesizable logic that V*HDL offers are CPP-style directives in Verilog, combinatorial functions, and `generate` statements. Nothing really fancy, and it shows. To give a few examples:
   (a) Building highly flexible bus interconnect is not possible. Even arbitrating any given number of bus masters for commonplace protocols such as Wishbone is difficult with the tools that V*HDL puts at our disposal.
   (b) Building a memory infrastructure (including bus interconnect, bridges and caches) that can automatically adapt itself at compile-time to any word size of the SDRAM is clumsy and tedious.
   (c) Building register banks for control, status and interrupt management of cores can also largely benefit from automation.
   (d) Many hardware acceleration problems can fit into the dataflow programming model. Manual dataflow implementation in V*HDL has, again, a lot of redundancy and potential for human errors. See the Milkymist texture mapping unit *[mthesis] [mxcell]* for an example of this. The amount of detail to deal with manually also makes the design space exploration difficult, and therefore hinders the design of efficient architectures.
   (e) Pre-computation of values, such as filter coefficients for DSP or even simply trigonometric tables, must often be done using external tools whose results are copy-and-pasted (in the best case, automatically) into the V*HDL source.

Enter Migen, a Python toolbox for building complex digital hardware. We could have designed a brand new programming language, but that would have been reinventing the wheel instead of being able to benefit from Python's rich features and immense library. The price to pay is a slightly cluttered syntax at times when writing descriptions in FHDL, but we believe this is totally acceptable, particularly when compared to VHDL ;-)

Migen is made up of several related components:

1. the base language, FHDL
2. a library of small generic cores
3. a simulator
4. a build system

Installing Migen

Either run the `setup.py` installation script or simply set `PYTHONPATH` to the root of the source directory.

If you wish to contribute patches, the suggest way to install is;

1. Clone from the git repository at http://github.com/m-labs/migen
2. Install using `python3 ./setup.py develop --user`
3. Edit the code in your git checkout.

Alternative install methods

- Migen is available for the Anaconda Python distribution. The package can be found at at https://anaconda.org/m-labs/migen
- Migen can be referenced in a requirements.txt file (used for `pip install -r requirements.txt`) via `-e git+http://github.com/m-labs/migen.git#egg=migen`. See the pip documentation for more information.

Feedback

Feedback concerning Migen or this manual should be sent to the M-Labs developers' mailing list `devel` on lists.m-labs.hk.

CHAPTER 2

The FHDL domain-specific language

The Fragmented Hardware Description Language (FHDL) is the basis of Migen. It consists of a formal system to describe signals, and combinatorial and synchronous statements operating on them. The formal system itself is low level and close to the synthesizable subset of Verilog, and we then rely on Python algorithms to build complex structures by combining FHDL elements. The FHDL module also contains a back-end to produce synthesizable Verilog, and some structure analysis and manipulation functionality.

FHDL differs from MyHDL *[myhdl]* in fundamental ways. MyHDL follows the event-driven paradigm of traditional HDLs (see *Background*) while FHDL separates the code into combinatorial statements, synchronous statements, and reset values. In MyHDL, the logic is described directly in the Python AST. The converter to Verilog or VHDL then examines the Python AST and recognizes a subset of Python that it translates into V*HDL statements. This seriously impedes the capability of MyHDL to generate logic procedurally. With FHDL, you manipulate a custom AST from Python, and you can more easily design algorithms that operate on it.

FHDL is made of several elements, which are briefly explained below. They all can be imported directly from the `migen` module.

Expressions

Constants

The `Constant` object represents a constant, HDL-literal integer. It behaves like specifying integers and booleans but also supports slicing and can have a bit width or signedness different from what is implied by the value it represents.

`True` and `False` are interpreted as 1 and 0, respectively.

Negative integers are explicitly supported. As with MyHDL *[countin]*, arithmetic operations return the natural results.

To lighten the syntax, assignments and operators automatically wrap Python integers and booleans into `Constant`. Additionally, `Constant` is aliased to `C`. The following are valid Migen statements: `a.eq(0)`, `a.eq(a + 1)`, `a.eq(C(42)[0:1])`.

Signal

The signal object represents a value that is expected to change in the circuit. It does exactly what Verilog's "wire" and "reg" and VHDL's "signal" do.

The main point of the signal object is that it is identified by its Python ID (as returned by the `id()` function), and nothing else. It is the responsibility of the V*HDL back-end to establish an injective mapping between Python IDs and the V*HDL namespace. It should perform name mangling to ensure this. The consequence of this is that signal objects can safely become members of arbitrary Python classes, or be passed as parameters to functions or methods that generate logic involving them.

The properties of a signal object are:

- An integer or a (integer, boolean) pair that defines the number of bits and whether the bit of higher index of the signal is a sign bit (i.e. the signal is signed). The defaults are one bit and unsigned. Alternatively, the `min` and `max` parameters can be specified to define the range of the signal and determine its bit width and signedness. As with Python ranges, `min` is inclusive and defaults to 0, `max` is exclusive and defaults to 2.
- A name, used as a hint for the V*HDL back-end name mangler.
- The signal's reset value. It must be an integer, and defaults to 0. When the signal's value is modified with a synchronous statement, the reset value is the initialization value of the associated register. When the signal is assigned to in a conditional combinatorial statement (`If` or `Case`), the reset value is the value that the signal has when no condition that causes the signal to be driven is verified. This enforces the absence of latches in designs. If the signal is permanently driven using a combinatorial statement, the reset value has no effect.

The sole purpose of the name property is to make the generated V*HDL code easier to understand and debug. From a purely functional point of view, it is perfectly OK to have several signals with the same name property. The back-end will generate a unique name for each object. If no name property is specified, Migen will analyze the code that created the signal object, and try to extract the variable or member name from there. For example, the following statements will create one or several signals named "bar":

```
bar = Signal()
self.bar = Signal()
self.baz.bar = Signal()
bar = [Signal() for x in range(42)]
```

In case of conflicts, Migen tries first to resolve the situation by prefixing the identifiers with names from the class and module hierarchy that created them. If the conflict persists (which can be the case if two signal objects are created with the same name in the same context), it will ultimately add number suffixes.

Operators

Operators are represented by the `_Operator` object, which generally should not be used directly. Instead, most FHDL objects overload the usual Python logic and arithmetic operators, which allows a much lighter syntax to be used. For example, the expression:

```
a * b + c
```

is equivalent to:

```
_Operator("+", [_Operator("*", [a, b]), c])
```

Slices

Likewise, slices are represented by the `_Slice` object, which often should not be used in favor of the Python slice operation [x:y]. Implicit indices using the forms [x], [x:] and [:y] are supported. Beware! Slices work like Python slices, not like VHDL or Verilog slices. The first bound is the index of the LSB and is inclusive. The second bound is the index of MSB and is exclusive. In V*HDL, bounds are MSB:LSB and both are inclusive.

Concatenations

Concatenations are done using the `Cat` object. To make the syntax lighter, its constructor takes a variable number of arguments, which are the signals to be concatenated together (you can use the Python "*" operator to pass a list instead). To be consistent with slices, the first signal is connected to the bits with the lowest indices in the result. This is the opposite of the way the "{}" construct works in Verilog.

Replications

The `Replicate` object represents the equivalent of {count{expression}} in Verilog. For example, the expression:

```
Replicate(0, 4)
```

is equivalent to:

```
Cat(0, 0, 0, 0)
```

Statements

Assignment

Assignments are represented with the `_Assign` object. Since using it directly would result in a cluttered syntax, the preferred technique for assignments is to use the `eq()` method provided by objects that can have a value assigned to them. They are signals, and their combinations with the slice and concatenation operators. As an example, the statement:

```
a[0].eq(b)
```

is equivalent to:

```
_Assign(_Slice(a, 0, 1), b)
```

If

The `If` object takes a first parameter which must be an expression (combination of the `Constant`, `Signal`, `_Operator`, `_Slice`, etc. objects) representing the condition, then a variable number of parameters representing the statements (`_Assign`, `If`, `Case`, etc. objects) to be executed when the condition is verified.

The `If` object defines a `Else()` method, which when called defines the statements to be executed when the condition is not true. Those statements are passed as parameters to the variadic method.

For convenience, there is also a `Elif()` method.

Example:

```
If(tx_count16 == 0,
    tx_bitcount.eq(tx_bitcount + 1),
    If(tx_bitcount == 8,
        self.tx.eq(1)
    ).Elif(tx_bitcount == 9,
        self.tx.eq(1),
        tx_busy.eq(0)
    ).Else(
        self.tx.eq(tx_reg[0]),
        tx_reg.eq(Cat(tx_reg[1:], 0))
    )
)
```

Case

The `Case` object constructor takes as first parameter the expression to be tested, and a dictionary whose keys are the values to be matched, and values the statements to be executed in the case of a match. The special value `"default"` can be used as match value, which means the statements should be executed whenever there is no other match.

Arrays

The `Array` object represents lists of other objects that can be indexed by FHDL expressions. It is explicitly possible to:

- nest `Array` objects to create multidimensional tables.
- list any Python object in a `Array` as long as every expression appearing in a module ultimately evaluates to a `Signal` for all possible values of the indices. This allows the creation of lists of structured data.
- use expressions involving `Array` objects in both directions (assignment and reading).

For example, this creates a 4x4 matrix of 1-bit signals:

```
my_2d_array = Array(Array(Signal() for a in range(4)) for b in range(4))
```

You can then read the matrix with (`x` and `y` being 2-bit signals):

```
out.eq(my_2d_array[x][y])
```

and write it with:

```
my_2d_array[x][y].eq(inp)
```

Since they have no direct equivalent in Verilog, `Array` objects are lowered into multiplexers and conditional statements before the actual conversion takes place. Such lowering happens automatically without any user intervention.

Any out-of-bounds access performed on an `Array` object will refer to the last element.

Specials

Tri-state I/O

A triplet (O, OE, I) of one-way signals defining a tri-state I/O port is represented by the `TSTriple` object. Such objects are only containers for signals that are intended to be later connected to a tri-state I/O buffer, and cannot be

used as module specials. Such objects, however, should be kept in the design as long as possible as they allow the individual one-way signals to be manipulated in a non-ambiguous way.

The object that can be used in as a module special is `Tristate`, and it behaves exactly like an instance of a tri-state I/O buffer that would be defined as follows:

```
Instance("Tristate",
  io_target=target,
  i_o=o,
  i_oe=oe,
  o_i=i
)
```

Signals `target`, `o` and `i` can have any width, while `oe` is 1-bit wide. The `target` signal should go to a port and not be used elsewhere in the design. Like modern FPGA architectures, Migen does not support internal tri-states.

A `Tristate` object can be created from a `TSTriple` object by calling the `get_tristate` method.

By default, Migen emits technology-independent behavioral code for a tri-state buffer. If a specific code is needed, the tristate handler can be overriden using the appropriate parameter of the V*HDL conversion function.

Instances

Instance objects represent the parametrized instantiation of a V*HDL module, and the connection of its ports to FHDL signals. They are useful in a number of cases:

- Reusing legacy or third-party V*HDL code.
- Using special FPGA features (DCM, ICAP, ...).
- Implementing logic that cannot be expressed with FHDL (e.g. latches).
- Breaking down a Migen system into multiple sub-systems.

The instance object constructor takes the type (i.e. name of the instantiated module) of the instance, then multiple parameters describing how to connect and parametrize the instance.

These parameters can be:

- `Instance.Input`, `Instance.Output` or `Instance.InOut` to describe signal connections with the instance. The parameters are the name of the port at the instance, and the FHDL expression it should be connected to.
- `Instance.Parameter` sets a parameter (with a name and value) of the instance.
- `Instance.ClockPort` and `Instance.ResetPort` are used to connect clock and reset signals to the instance. The only mandatory parameter is the name of the port at the instance. Optionally, a clock domain name can be specified, and the `invert` option can be used to interface to those modules that require a 180-degree clock or a active-low reset.

Memories

Memories (on-chip SRAM) are supported using a mechanism similar to instances.

A memory object has the following parameters:

- The width, which is the number of bits in each word.
- The depth, which represents the number of words in the memory.
- An optional list of integers used to initialize the memory.

To access the memory in hardware, ports can be obtained by calling the `get_port` method. A port always has an address signal `a` and a data read signal `dat_r`. Other signals may be available depending on the port's configuration.

Options to `get_port` are:

- `write_capable` (default: `False`): if the port can be used to write to the memory. This creates an additional `we` signal.
- `async_read` (default: `False`): whether reads are asychronous (combinatorial) or synchronous (registered).
- `has_re` (default: `False`): adds a read clock-enable signal `re` (ignored for asychronous ports).
- `we_granularity` (default: `0`): if non-zero, writes of less than a memory word can occur. The width of the `we` signal is increased to act as a selection signal for the sub-words.
- `mode` (default: `WRITE_FIRST`, ignored for aynchronous ports). It can be:
  - `READ_FIRST`: during a write, the previous value is read.
  - `WRITE_FIRST`: the written value is returned.
  - `NO_CHANGE`: the data read signal keeps its previous value on a write.
- `clock_domain` (default: `"sys"`): the clock domain used for reading and writing from this port.

Migen generates behavioural V*HDL code that should be compatible with all simulators and, if the number of ports is <= 2, most FPGA synthesizers. If a specific code is needed, the memory handler can be overriden using the appropriate parameter of the V*HDL conversion function.

Modules

Modules play the same role as Verilog modules and VHDL entities. Similarly, they are organized in a tree structure. A FHDL module is a Python object that derives from the `Module` class. This class defines special attributes to be used by derived classes to describe their logic. They are explained below.

Combinatorial statements

A combinatorial statement is a statement that is executed whenever one of its inputs changes.

Combinatorial statements are added to a module by using the `comb` special attribute. Like most module special attributes, it must be accessed using the += incrementation operator, and either a single statement, a tuple of statements or a list of statements can appear on the right hand side.

For example, the module below implements a OR gate:

```
class ORGate(Module):
  def __init__(self):
    self.a = Signal()
    self.b = Signal()
    self.x = Signal()

    ###

    self.comb += self.x.eq(self.a | self.b)
```

To improve code readability, it is recommended to place the interface of the module at the beginning of the `__init__` function, and separate it from the implementation using three hash signs.

Synchronous statements

A synchronous statements is a statement that is executed at each edge of some clock signal.

They are added to a module by using the `sync` special attribute, which has the same properties as the `comb` attribute.

The `sync` special attribute also has sub-attributes that correspond to abstract clock domains. For example, to add a statement to the clock domain named `foo`, one would write `self.sync.foo += statement`. The default clock domain is `sys` and writing `self.sync += statement` is equivalent to writing `self.sync.sys += statement`.

Submodules and specials

Submodules and specials can be added by using the `submodules` and `specials` attributes respectively. This can be done in two ways:

1. anonymously, by using the `+=` operator on the special attribute directly, e.g. `self.submodules += some_other_module`. Like with the `comb` and `sync` attributes, a single module/special or a tuple or list can be specified.
2. by naming the submodule/special using a subattribute of the `submodules` or `specials` attribute, e.g. `self.submodules.foo = module_foo`. The submodule/special is then accessible as an attribute of the object, e.g. `self.foo` (and not `self.submodules.foo`). Only one submodule/special can be added at a time using this form.

Clock domains

Specifying the implementation of a clock domain is done using the `ClockDomain` object. It contains the name of the clock domain, a clock signal that can be driven like any other signal in the design (for example, using a PLL instance), and optionally a reset signal. Clock domains without a reset signal are reset using e.g. `initial` statements in Verilog, which in many FPGA families initalize the registers during configuration.

The name can be omitted if it can be extracted from the variable name. When using this automatic naming feature, prefixes `_`, `cd_` and `_cd_` are removed.

Clock domains are then added to a module using the `clock_domains` special attribute, which behaves exactly like `submodules` and `specials`.

Summary of special attributes

Syntax	Action
self.comb += stmt	Add combinatorial statement to current module.
self.comb += stmtA, stmtB self.comb += [stmtA, stmtB]	Add combinatorial statements A and B to current module.
self.sync += stmt	Add synchronous statement to current module, in default clock domain sys.
self.sync.foo += stmt	Add synchronous statement to current module, in clock domain foo.
self.sync.foo += stmtA, stmtB self.sync.foo += [stmtA, stmtB]	Add synchronous statements A and B to current module, in clock domain foo.
self.submodules += mod	Add anonymous submodule to current module.
self.submodules += modA, modB self.submodules += [modA, modB]	Add anonymous submodules A and B to current module.
self.submodules.bar = mod	Add submodule named bar to current module. The submodule can then be accessed using self.bar.
self.specials += spe	Add anonymous special to current module.
self.specials += speA, speB self.specials += [speA, speB]	Add anonymous specials A and B to current module.
self.specials.bar = spe	Add special named bar to current module. The special can then be accessed using self.bar.
self.clock_domains += cd	Add clock domain to current module.
self.clock_domains += cdA, cdB self.clock_domains += [cdA, cdB]	Add clock domains A and B to current module.
self.clock_domains.pix = ClockDomain() self.clock_domains._pix = ClockDomain() self.clock_domains.cd_pix = ClockDomain() self.clock_domains._cd_pix = ClockDomain()	Create and add clock domain pix to current module. The clock domain name is pix in all cases. It can be accessed using self.pix, self._pix, self.cd_pix and self._cd_pix, respectively.

Clock domain management

When a module has named submodules that define one or several clock domains with the same name, those clock domain names are prefixed with the name of each submodule plus an underscore.

An example use case of this feature is a system with two independent video outputs. Each video output module is made of a clock generator module that defines a clock domain `pix` and drives the clock signal, plus a driver module that has synchronous statements and other elements in clock domain `pix`. The designer of the video output module can simply use the clock domain name `pix` in that module. In the top-level system module, the video output submodules are named `video0` and `video1`. Migen then automatically renames the `pix` clock domain of each module to `video0_pix` and `video1_pix`. Note that happens only because the clock domain is defined (using ClockDomain objects), not simply referenced (using e.g. synchronous statements) in the video output modules.

Clock domain name overlap is an error condition when any of the submodules that defines the clock domains is anonymous.

Finalization mechanism

Sometimes, it is desirable that some of a module logic be created only after the user has finished manipulating that module. For example, the FSM module supports that states be defined dynamically, and the width of the state signal can be known only after all states have been added. One solution is to declare the final number of states in the FSM constructor, but this is not user-friendly. A better solution is to automatically create the state signal just before the FSM module is converted to V*HDL. Migen supports this using the so-called finalization mechanism.

Modules can overload a `do_finalize` method that can create logic and is called using the algorithm below:

1. Finalization of the current module begins.
2. If the module has already been finalized (e.g. manually), the procedure stops here.
3. Submodules of the current module are recursively finalized.
4. `do_finalize` is called for the current module.
5. Any new submodules created by the current module's `do_finalize` are recursively finalized.

Finalization is automatically invoked at V*HDL conversion and at simulation. It can be manually invoked for any module by calling its `finalize` method.

The clock domain management mechanism explained above happens during finalization.

Conversion for synthesis

Any FHDL module can be converted into synthesizable Verilog HDL. This is accomplished by using the `convert` function in the `migen.fhdl.verilog` module:

```
# define FHDL module MyDesign here

if __name__ == "__main__":
    from migen.fhdl.verilog import convert
    convert(MyDesign()).write("my_design.v")
```

The `migen.build` component provides scripts to interface third-party FPGA tools (from Xilinx, Altera and Lattice) to Migen, and a database of boards for the easy deployment of designs.

CHAPTER 3

Simulating a Migen design

Migen allows you to easily simulate your FHDL design and interface it with arbitrary Python code. The simulator is written in pure Python and interprets the FHDL structure directly without using an external Verilog simulator.

Migen lets you write testbenches using Python's generator functions. Such testbenches execute concurrently with the FHDL simulator, and communicate with it using the `yield` statement. There are four basic patterns:

1. Reads: the state of the signal at the current time can be queried using `(yield signal)`;
2. Writes: the state of the signal after the next clock cycle can be set using `(yield signal.eq(value))`;
3. Clocking: simulation can be advanced by one clock cycle using `yield`;
4. Composition: control can be transferred to another testbench function using `yield from run_other()`.

A testbench can be run using the `run_simulation` function from `migen.sim`; `run_simulation(dut, bench)` runs the generator function `bench` against the logic defined in an FHDL module `dut`.

Passing the `vcd_name="file.vcd"` argument to `run_simulation` will cause it to write a VCD dump of the signals inside `dut` to `file.vcd`.

Examples

For example, consider this module:

```
class ORGate(Module):
  def __init__(self):
    self.a = Signal()
    self.b = Signal()
    self.x = Signal()

    ###

    self.comb += self.x.eq(self.a | self.b)
```

It could be simulated together with the following testbench:

```
dut = ORGate()

def testbench():
  yield dut.a.eq(0)
  yield dut.b.eq(0)
  yield
  assert (yield dut.x) == 0

  yield dut.a.eq(0)
  yield dut.b.eq(1)
  yield
  assert (yield dut.x) == 1

run_simulation(dut, testbench())
```

This is, of course, quite verbose, and individual steps can be factored into a separate function:

```
dut = ORGate()

def check_case(a, b, x):
  yield dut.a.eq(a)
  yield dut.b.eq(b)
  yield
  assert (yield dut.x) == x

def testbench():
  yield from check_case(0, 0, 0)
  yield from check_case(0, 1, 1)
  yield from check_case(1, 0, 1)
  yield from check_case(1, 1, 1)

run_simulation(dut, testbench())
```

Pitfalls

There are, unfortunately, some basic mistakes that can produce very puzzling results.

When calling other testbenches, it is important to not forget the `yield from`. If it is omitted, the call would silently do nothing.

When writing to a signal, it is important that nothing else should drive the signal concurrently. If that is not the case, the write would silently do nothing.

CHAPTER 4

Synthesizing a Migen design

[To be written]

CHAPTER 5

API reference

fhdl.structure module

class migen.fhdl.structure.**Array**
Bases: list

Addressable multiplexer

An array is created from an iterable of values and indexed using the usual Python simple indexing notation (no negative indices or slices). It can be indexed by numeric constants, *_Value* s, or *Signal* s.

The result of indexing the array is a proxy for the entry at the given index that can be used on either RHS or LHS of assignments.

An array can be indexed multiple times.

Multidimensional arrays are supported by packing inner arrays into outer arrays.

Parameters **values** (*iterable of ints, _Values, Signals*) – Entries of the array. Each entry can be a numeric constant, a *Signal* or a *Record*.

Examples

```
>>> a = Array(range(10))
>>> b = Signal(max=10)
>>> c = Signal(max=10)
>>> b.eq(a[9 - c])
```

migen.fhdl.structure.**C**
alias of *Constant*

class migen.fhdl.structure.**Case**(*test*, *cases*)
Bases: migen.fhdl.structure._Statement

Case/Switch statement

Parameters

- **test** (*_Value, in*) – Selector value used to decide which block to execute
- **cases** (*dict*) – Dictionary of cases. The keys are numeric constants to compare with *test*. The values are statements to be executed the corresponding key matches *test*. The dictionary may contain a string key *"default"* to mark a fall-through case that is executed if no other key matches.

Examples

```
>>> a = Signal()
>>> b = Signal()
>>> Case(a, {
...     0:         b.eq(1),
...     1:         b.eq(0),
...     "default": b.eq(0),
... })
```

makedefault(*key=None*)

Mark a key as the default case

Deletes/substitutes any previously existing default case.

Parameters **key** (*int, Constant or None*) – Key to use as default case if no other key matches. By default, the largest key is the default key.

class migen.fhdl.structure.**Cat**(**args*)

Bases: migen.fhdl.structure._Value

Concatenate values

Form a compound *_Value* from several smaller ones by concatenation. The first argument occupies the lower bits of the result. The return value can be used on either side of an assignment, that is, the concatenated value can be used as an argument on the RHS or as a target on the LHS. If it is used on the LHS, it must solely consist of *Signal* s, slices of *Signal* s, and other concatenations meeting these properties. The bit length of the return value is the sum of the bit lengths of the arguments:

```
len(Cat(args)) == sum(len(arg) for arg in args)
```

Parameters ***args** (*_Values or iterables of _Values, inout*) – *_Value* s to be concatenated.

Returns Resulting *_Value* obtained by concatentation.

Return type *Cat*, inout

class migen.fhdl.structure.**ClockDomain**(*name=None*, *reset_less=False*)

Bases: object

Synchronous domain

Parameters

- **name** (*str or None*) – Domain name. If None (the default) the name is inferred from the variable name this *ClockDomain* is assigned to (stripping any *"cd_"* prefix).
- **reset_less** (*bool*) – The domain does not use a reset signal. Registers within this domain are still all initialized to their reset state once, e.g. through Verilog *"initial"* statements.

clk
Signal, inout – The clock for this domain. Can be driven or used to drive other signals (preferably in combinatorial context).

rst
Signal or None, inout – Reset signal for this domain. Can be driven or used to drive.

rename(*new_name*)
Rename the clock domain

Parameters **new_name** (*str*) – New name

class migen.fhdl.structure.**ClockSignal**(*cd='sys'*)
Bases: migen.fhdl.structure._Value

Clock signal for a given clock domain

ClockSignal s for a given clock domain can be retrieved multiple times. They all ultimately refer to the same signal.

Parameters **cd** (*str*) – Clock domain to obtain a clock signal for. Defaults to *"sys"*.

class migen.fhdl.structure.**Constant**(*value*, *bits_sign=None*)
Bases: migen.fhdl.structure._Value

A constant, HDL-literal integer _*Value*

Parameters

- **value** (*int*) –
- **bits_sign** (*int or tuple or None*) – Either an integer *bits* or a tuple *(bits, signed)* specifying the number of bits in this *Constant* and whether it is signed (can represent negative values). *bits_sign* defaults to the minimum width and signedness of *value*.

class migen.fhdl.structure.**DUID**
Bases: object

Deterministic Unique IDentifier

class migen.fhdl.structure.**If**(*cond*, **t*)
Bases: migen.fhdl.structure._Statement

Conditional execution of statements

Parameters

- **cond** (*_Value(1), in*) – Condition
- ***t** (*Statements*) – Statements to execute if *cond* is asserted.

Examples

```
>>> a = Signal()
>>> b = Signal()
>>> c = Signal()
>>> d = Signal()
>>> If(a,
...     b.eq(1)
... ).Elif(c,
...     b.eq(0)
... ).Else(
```

```
...     b.eq(d)
... )
```

Elif(*cond*, **t*)
Add an *else if* conditional block

Parameters

- **cond** (*_Value(1), in*) – Condition
- ***t** (*Statements*) – Statements to execute if previous conditions fail and *cond* is asserted.

Else(**f*)
Add an *else* conditional block

Parameters ***f** (*Statements*) – Statements to execute if all previous conditions fail.

migen.fhdl.structure.**Mux**(*sel*, *val1*, *val0*)
Multiplex between two values

Parameters

- **sel** (*_Value(1), in*) – Selector.
- **val1** (*_Value(N), in*) –
- **val0** (*_Value(N), in*) – Input values.

Returns Output *_Value*. If *sel* is asserted, the Mux returns *val1*, else *val0*.

Return type _Value(N), out

class migen.fhdl.structure.**Replicate**(*v*, *n*)
Bases: migen.fhdl.structure._Value

Replicate a value

An input value is replicated (repeated) several times to be used on the RHS of assignments:

```
len(Replicate(s, n)) == len(s)*n
```

Parameters

- **v** (*_Value, in*) – Input value to be replicated.
- **n** (*int*) – Number of replications.

Returns Replicated value.

Return type *Replicate*, out

class migen.fhdl.structure.**ResetSignal**(*cd='sys'*, *allow_reset_less=False*)
Bases: migen.fhdl.structure._Value

Reset signal for a given clock domain

ResetSignal s for a given clock domain can be retrieved multiple times. They all ultimately refer to the same signal.

Parameters

- **cd** (*str*) – Clock domain to obtain a reset signal for. Defaults to *"sys"*.

- **allow_reset_less** (*bool*) – If the clock domain is resetless, return 0 instead of reporting an error.

class migen.fhdl.structure.**Signal**(*bits_sign=None*, *name=None*, *variable=False*, *reset=0*, *reset_less=False*, *name_override=None*, *min=None*, *max=None*, *related=None*, *attr=None*)

Bases: migen.fhdl.structure._Value

A *_Value* that can change

The *Signal* object represents a value that is expected to change in the circuit. It does exactly what Verilog's *wire* and *reg* and VHDL's *signal* do.

A *Signal* can be indexed to access a subset of its bits. Negative indices (*signal[-1]*) and the extended Python slicing notation (*signal[start:stop:step]*) are supported. The indices 0 and -1 are the least and most significant bits respectively.

Parameters

- **bits_sign** (*int or tuple*) – Either an integer *bits* or a tuple *(bits, signed)* specifying the number of bits in this *Signal* and whether it is signed (can represent negative values). *signed* defaults to *False*.
- **name** (*str or None*) – Name hint for this signal. If *None* (default) the name is inferred from the variable name this *Signal* is assigned to. Name collisions are automatically resolved by prepending names of objects that contain this *Signal* and by appending integer sequences.
- **variable** (*bool*) – Deprecated.
- **reset** (*int*) – Reset (synchronous) or default (combinatorial) value. When this *Signal* is assigned to in synchronous context and the corresponding clock domain is reset, the *Signal* assumes the given value. When this *Signal* is unassigned in combinatorial context (due to conditional assignments not being taken), the *Signal* assumes its *reset* value. Defaults to 0.
- **reset_less** (*bool*) – If *True*, do not generate reset logic for this *Signal* in synchronous statements. The *reset* value is only used as a combinatorial default or as the initial value. Defaults to *False*.
- **name_override** (*str or None*) – Do not use the inferred name but the given one.
- **min** (*int or None*) –
- **max** (*int or None*) – If *bits_sign* is *None*, the signal bit width and signedness are determined by the integer range given by *min* (inclusive, defaults to 0) and *max* (exclusive, defaults to 2).
- **related** (*Signal or None*) –
- **attr** (*set of synthesis attributes*) –

classmethod like(*other*, ***kwargs*)

Create Signal based on another.

Parameters

- **other** (*_Value*) – Object to base this Signal on.
- **migen.fhdl.bitcontainer.value_bits_sign for details.** (*See*) –

migen.fhdl.structure.**wrap**(*value*)

Ensures that the passed object is a Migen value. Booleans and integers are automatically wrapped into Constant.

fhdl.bitcontainer module

migen.fhdl.bitcontainer.**value_bits_sign**(*v*)
Bit length and signedness of a value.

Parameters **v** (*Value*) –

Returns Number of bits required to store *v* or available in *v*, followed by whether *v* has a sign bit (included in the bit count).

Return type int, bool

Examples

```
>>> value_bits_sign(f.Signal(8))
8, False
>>> value_bits_sign(C(0xaa))
8, False
```

genlib.fifo module

genlib.coding module

Encoders and decoders between binary and one-hot representation

class migen.genlib.coding.**Decoder**(*width*)
Bases: migen.fhdl.module.Module

Decode binary to one-hot

If *n* is low, the *i* th bit in *o* is asserted, the others are not, else *o* == *0*.

Parameters **width** (*int*) – Bit width of the output

i
Signal(max=width), in – Input binary

o
Signal(width), out – Decoded one-hot

n
Signal(1), in – Invalid, no output bits are to be asserted

class migen.genlib.coding.**Encoder**(*width*)
Bases: migen.fhdl.module.Module

Encode one-hot to binary

If *n* is low, the *o* th bit in *i* is asserted, else none or multiple bits are asserted.

Parameters **width** (*int*) – Bit width of the input

i
Signal(width), in – One-hot input

o
Signal(max=width), out – Encoded binary

n
Signal(1), out – Invalid, either none or multiple input bits are asserted

class migen.genlib.coding.**PriorityEncoder**(*width*)
Bases: migen.fhdl.module.Module

Priority encode requests to binary

If *n* is low, the *o* th bit in *i* is asserted and the bits below *o* are unasserted, else *o == 0*. The LSB has priority.

Parameters **width** (*int*) – Bit width of the input

i
Signal(width), in – Input requests

o
Signal(max=width), out – Encoded binary

n
Signal(1), out – Invalid, no input bits are asserted

genlib.sort module

class migen.genlib.sort.**BitonicSort**(*n*, *m*, *ascending=True*)
Bases: migen.fhdl.module.Module

Combinatorial sorting network

The Bitonic sort is implemented as a combinatorial sort using comparators and multiplexers. Its asymptotic complexity (in terms of number of comparators/muxes) is O(n log(n)**2), like mergesort or shellsort.

http://www.dps.uibk.ac.at/~cosenza/teaching/gpu/sort-batcher.pdf

http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

http://www.myhdl.org/doku.php/cookbook:bitonic

Parameters

- **n** (*int*) – Number of inputs and output signals.
- **m** (*int*) – Bit width of inputs and outputs. Or a tuple of *(m, signed)*.
- **ascending** (*bool*) – Sort direction. *True* if input is to be sorted ascending, *False* for descending. Defaults to ascending.

i
list of Signals, in – Input values, each *m* wide.

o
list of Signals, out – Output values, sorted, each *m* bits wide.

genlib.fsm module

class migen.genlib.fsm.**FSM**(*reset_state=None*)
Bases: migen.fhdl.module.Module

Finite state machine

Any Python objects can be used as states, e.g. strings.

Parameters **reset_state** – Reset state. Defaults to the first added state.

Examples

```
>>> self.active = Signal()
>>> self.bitno = Signal(3)
>>>
>>> fsm = FSM(reset_state="START")
>>> self.submodules += fsm
>>>
>>> fsm.act("START",
...     self.active.eq(1),
...     If(strobe,
...         NextState("DATA")
...     )
... )
>>> fsm.act("DATA",
...     self.active.eq(1),
...     If(strobe,
...         NextValue(self.bitno, self.bitno + 1)
...         If(self.bitno == 7,
...             NextState("END")
...         )
...     )
... )
>>> fsm.act("END",
...     self.active.eq(0),
...     NextState("STOP")
... )
```

act(*state*, **statements*)
Schedules *statements* to be executed in *state*. Statements may include:

- combinatorial statements of form *a.eq(b)*, equivalent to *self.comb* += *a.eq(b)* when the FSM is in the given *state*;
- synchronous statements of form *NextValue(a, b)*, equivalent to *self.sync* += *a.eq(b)* when the FSM is in the given *state*;
- a statement of form *NextState(new_state)*, selecting the next state;
- *If*, *Case*, etc.

ongoing(*state*)
Returns a signal that has the value 1 when the FSM is in the given *state*, and 0 otherwise.

Bibliography

[mm] http://m-labs.hk

[mthesis] http://m-labs.hk/thesis/thesis.pdf

[mxcell] http://www.xilinx.com/publications/archives/xcell/Xcell77.pdf p30-35

[myhdl] http://www.myhdl.org

[countin] http://www.jandecaluwe.com/hdldesign/counting.html

Index