PyRTL

Timothy Sherwood

Apr 22, 2024

CONTENTS

A collection of classes providing simple RTL specification, simulation, tracing, and testing suitable for teaching and research. Simplicity, usability, clarity, and extensibility rather than performance or optimization is the overarching goal. With PyRTL you can use the full power of Python to describe complex synthesizable digital designs, simulate and test them, and export them to Verilog.

CHAPTER

ONE

QUICK LINKS

- Get an overview from the PyRTL Project Webpage
- Read through Example PyRTL Code
- File a Bug or Issue Report
- Contribute to project on GitHub

CHAPTER

TWO

INSTALLATION

Automatic installation:

```
pip install pyrtl
```

PyRTL is listed in PyPI and can be installed with **`pip`** or **`pip3`**. If the above command fails due to insufficient permissions, you may need to do `sudo pip install pyrtl` (to install as superuser) or `pip install --user pyrtl` (to install as a normal user).

PyRTL is tested to work with Python 3.8+.

CHAPTER

THREE

DESIGN, SIMULATE, AND INSPECT IN 15 LINES

```
import pyrtl

a = pyrtl.Input(8,'a')  # input "pins"
b = pyrtl.Input(8,'b')
q = pyrtl.Output(8,'q')  # output "pins"
gt5 = pyrtl.Output(1,'gt5')

result = a + b  # makes an 8-bit adder
q <<= result  # assigns output of adder to out pin
gt5 <<= result > 5  # does a comparison, assigns that to different pin

# simulate and output the resulting waveform to the terminal
sim = pyrtl.Simulation()
sim.step_multiple({'a':[0,1,2,3,4], 'b':[2,2,3,3,4]})
sim.tracer.render_trace()
```

After you have PyRTL installed, you should be able to cut and paste the above into a file and run it with Python. The result you should see, drawn right into the terminal, is the output of the simulation. While a great deal of work has gone into making hardware design in PyRTL as friendly as possible, please don't mistake that for a lack of depth. You can just as easily export to Verilog or other hardware formats, view results with your favorite waveform viewer, build hardware transformation passes, run JIT-accelerated simulations, design, test, and even verify hugely complex digital systems, and much more. Most critically of all it is easy to extend with your own approaches to digital hardware development as you find necessary.

CHAPTER

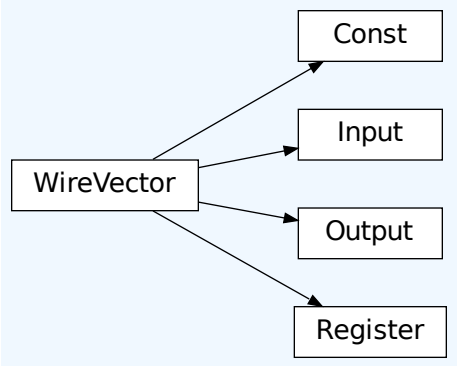
FOUR

OVERVIEW OF PYRTL

If you are brand new to PyRTL we recommend that you start with the PyRTL Code Examples which will show you most of the core functionality in the context of a complete design.

4.1 PyRTL Classes:

Perhaps the most important class to understand is `WireVector`, which is the basic type from which you build all hardware. If you are coming to PyRTL from Verilog, a `WireVector` is closest to a multi-bit *wire*. Every new `WireVector` builds a set of wires which you can then connect with other `WireVector` through overloaded operations such as *addition* or *bitwise or*. A bunch of other related classes, including `Input`, `Output`, `Const`, and `Register` are all derived from `WireVector`. Coupled with `MemBlock` (and `RomBlock`), this is all a user needs to create a functional hardware design.



After specifying a hardware design, there are then options to simulate your design right in PyRTL, synthesize it down to primitive 1-bit operations, optimize it, and export it to Verilog (along with a testbench).

4.1.1 Simulation

PyRTL provides tools for simulation and viewing simulation traces. Simulation is how your hardware is "executed" for the purposes of testing, and three different classes help you do that: *Simulation*, *FastSimulation* and *CompiledSimulation*. All three have *almost* the same interface and, except for a few debugging cases, can be used interchangeably. Typically one starts with *Simulation* and then moves up to *FastSimulation* when performance begins to matter.

Both *Simulation* and *FastSimulation* take an instance of *SimulationTrace* as an argument (or makes an empty *SimulationTrace* by default), which stores a list of the signals as they are simulated. This trace can then be rendered to the terminal with *WaveRenderer*, although unless there are some problems with the default configurations, most end users should not need to even be aware of *WaveRenderer*. The examples describe other ways that the trace may be handled, including extraction as a test bench and export to a VCD file.

4.1.2 Optimization

WireVector and *MemBlock* are just "sugar" over a core set of primitives, and the final design is built up incrementally as a graph of these primitives. *WireVectors* connects these "primitives", which connect to other *WireVectors*. Each primitive is a *LogicNet*, and a *Block* is a graph of *LogicNets*. Typically a full design is stored in a single *Block*. The function *working_block()* returns the block on which we are implicitly working. Hardware transforms may make a new *Block* from an old one. For example, see *PostSynthBlock*.

4.1.3 Errors

Finally, when things go wrong you may hit on one of two `Exceptions`, neither of which is likely recoverable automatically (which is why we limited them to only two). The intention is that `PyrtlError` is intended to capture end user errors such as invalid constant strings and mis-matched bitwidths. In contrast, `PyrtlInternalError` captures internal invariants and assertions over the core logic graph which should never be hit when constructing designs in the normal ways. If you hit a confusing `PyrtlError` or any `PyrtlInternalError` feel free to file an issue.

CHAPTER

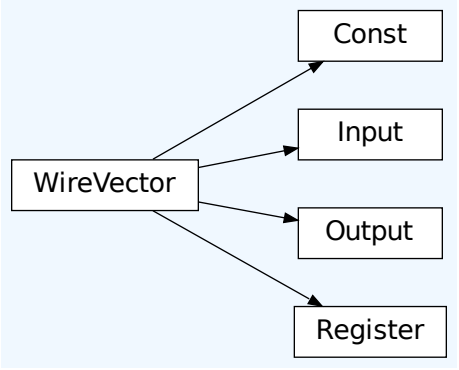
FIVE

REFERENCE GUIDE

5.1 Wires and Logic

Wires define the relationship between logic blocks in PyRTL. They are treated like normal wires in traditional RTL systems except the *Register* wire. Logic is then created when wires are combined with one another using the provided operators. For example, if `a` and `b` are both of type *WireVector*, then `a + b` will make an adder, plug `a` and `b` into the inputs of that adder, and return a new *WireVector* which is the output of that adder. *Block* stores the description of the hardware as you build it.

Input, *Output*, *Const*, and *Register* all derive from *WireVector*. *Input* represents an input pin, serving as a placeholder for an external value provided during simulation. *Output* represents an output pin, which does not drive any wires in the design. *Const* is useful for specifying hard-wired values and *Register* is how sequential elements are created (they all have an implicit clock).



5.1.1 WireVector

class pyrtl.wire.**WireVector**(*bitwidth=None*, *name=''*, *block=None*)

The main class for describing the connections between operators.

WireVectors act much like a list of wires, except that there is no "contained" type, each slice of a WireVector is itself a WireVector (even if it just contains a single "bit" of information). The least significant bit of the wire is at index 0 and normal list slicing syntax applies (i.e. `myvector[0:5]` makes a new vector from the bottom 5 bits of `myvector`, `myvector[-1]` takes the most significant bit, and `myvector[-4:]` takes the 4 most significant bits).

Operation	Syntax	Function
Addition	`a + b`	Creates an adder, returns WireVector
Subtraction	`a - b`	Subtraction (two's complement)
Multiplication	`a * b`	Creates an multiplier, returns WireVector
Xor	`a ^ b`	Bitwise XOR, returns WireVector
Or	`a | b`	Bitwise OR, returns WireVector
And	`a & b`	Bitwise AND, returns WireVector
Invert	`~a`	Bitwise invert, returns WireVector
Less Than	`a < b`	Less than, return 1-bit WireVector
Less or Eq.	`a <= b`	Less than or Equal to, return 1-bit WireVector
Greater Than	`a > b`	Greater than, return 1-bit WireVector
Greater or Eq.	`a >= b`	Greater or Equal to, return 1-bit WireVector
Equality	`a == b`	Hardware to check equality, return 1-bit WireVector
Not Equal	`a != b`	Inverted equality check, return 1-bit WireVector
Bitwidth	`len(a)`	Return bitwidth of the WireVector
Assignment	`a <<= b`	Connect from b to a (see note below)
Bit Slice	`a[3:6]`	Selects bits from WireVector, in this case bits 3,4,5

A note on <<= asssignment: This operator is how you "drive" an already created wire with an existing wire. If you were to do `a = b` it would lose the old value of `a` and simply overwrite it with a new value, in this case with a reference to WireVector `b`. In contrast `a <<= b` does not overwrite `a`, but simply wires the two together.

__add__(*other*)

Creates a LogicNet that adds two wires together into a single WireVector.

Return WireVector
Returns the result wire of the operation. The resulting wire has one more bit than the longer of the two input wires.

Addition is compatible with two's complement signed numbers.

Examples:

```
temp = a + b  # simple addition of two WireVectors
temp = a + 5  # you can use integers
temp = a + 0b110  # you can use other integers
temp = a + "3'h7"  # compatable verilog constants work too
```

__ilshift__(*other*)

Wire assignment operator (assign other to self).

Example:

```
i = pyrtl.Input(8, 'i')
t = pyrtl.WireVector(8, 't')
t <<= i
```

__init__(*bitwidth=None*, *name=''*, *block=None*)

Construct a generic WireVector.

Parameters

- **bitwidth** (*int*) – If no bitwidth is provided, it will be set to the minimum number of bits to represent this wire
- **block** (Block) – The block under which the wire should be placed. Defaults to the working block
- **name** (*str*) – The name of the wire referred to in some places. Must be unique. If none is provided, one will be autogenerated

Returns
a WireVector object

Examples:

```
data = pyrtl.WireVector(8, 'data')  # visible in trace as "data"
ctrl = pyrtl.WireVector(1)  # assigned tmp name, not visible in traces by␣
↪default
temp = pyrtl.WireVector()  # temporary with width to be defined later
temp <<= data  # this this case temp will get the bitwdith of 8 from data
```

__len__()

Get the bitwidth of a WireVector.

Return integer
Returns the length (i.e. bitwidth) of the WireVector in bits.

Note that WireVectors do not need to have a bitwidth defined when they are first allocated. They can get it from a <<= assignment later. However, if you check the `len` of WireVector with undefined bitwidth it will throw `PyrtlError`.

__mul__(*other*)

Creates a LogicNet that multiplies two different WireVectors.

Return WireVector
Returns the result wire of the operation. The resulting wire's bitwidth is the sum of the two input wires' bitwidths.

Multiplication is *not* compatible with two's complement signed numbers.

__sub__(*other*)

Creates a LogicNet that subtracts the right wire from the left one.

Return WireVector
Returns the result wire of the operation. The resulting wire has one more bit than the longer of the two input wires.

Subtraction is compatible with two's complement signed numbers.

property bitmask

A property holding a bitmask of the same length as this WireVector. Specifically it is an integer with a number of bits set to 1 equal to the bitwidth of the WireVector.

It is often times useful to "mask" an integer such that it fits in the the number of bits of a WireVector. As a convenience for this, the `bitmask` property is provided. As an example, if there was a 3-bit WireVector `a`, a call to `a.bitmask()` should return 0b111 or 0x7.

property name

A property holding the name (a string) of the WireVector, can be read or written. For example: `print(a.name)` or `a.name = 'mywire'`.

nand(*other*)

Creates a LogicNet that bitwise nands two WireVectors together to a single WireVector.

Return WireVector
Returns WireVector of the nand operation.

sign_extended(*bitwidth*)

Generate a new sign extended WireVector derived from self.

Return WireVector
Returns a new WireVector equal to the original WireVector sign extended to the specified bitwidth.

If the bitwidth specified is smaller than the bitwidth of self, then PyrtlError is thrown.

truncate(*bitwidth*)

Generate a new truncated WireVector derived from self.

Return WireVector
Returns a new WireVector equal to the original WireVector but truncated to the specified bitwidth.

If the bitwidth specified is larger than the bitwidth of self, then PyrtlError is thrown.

zero_extended(*bitwidth*)

Generate a new zero extended WireVector derived from self.

Return WireVector
Returns a new WireVector equal to the original WireVector zero extended to the specified bitwidth.

If the bitwidth specified is smaller than the bitwidth of self, then PyrtlError is thrown.

5.1.2 Input Pins

class pyrtl.wire.**Input**(*bitwidth=None*, *name=''*, *block=None*)

Bases: *WireVector*

A WireVector type denoting inputs to a block (no writers).

__init__(*bitwidth=None*, *name=''*, *block=None*)

Construct a generic WireVector.

Parameters

- **bitwidth** (*int*) – If no bitwidth is provided, it will be set to the minimum number of bits to represent this wire
- **block** (Block) – The block under which the wire should be placed. Defaults to the working block

- **name** (*str*) – The name of the wire referred to in some places. Must be unique. If none is provided, one will be autogenerated

Returns
a WireVector object

Examples:

```
data = pyrtl.WireVector(8, 'data')  # visible in trace as "data"
ctrl = pyrtl.WireVector(1)  # assigned tmp name, not visible in traces by␣
↪default
temp = pyrtl.WireVector()  # temporary with width to be defined later
temp <<= data  # this this case temp will get the bitwdith of 8 from data
```

5.1.3 Output Pins

class pyrtl.wire.**Output**(*bitwidth=None*, *name=''*, *block=None*)

Bases: *WireVector*

A WireVector type denoting outputs of a block (no readers).

Even though Output seems to have valid ops such as `__or__` , using them will throw an error.

__init__(*bitwidth=None*, *name=''*, *block=None*)

Construct a generic WireVector.

Parameters

- **bitwidth** (*int*) – If no bitwidth is provided, it will be set to the minimum number of bits to represent this wire
- **block** (Block) – The block under which the wire should be placed. Defaults to the working block
- **name** (*str*) – The name of the wire referred to in some places. Must be unique. If none is provided, one will be autogenerated

Returns
a WireVector object

Examples:

```
data = pyrtl.WireVector(8, 'data')  # visible in trace as "data"
ctrl = pyrtl.WireVector(1)  # assigned tmp name, not visible in traces by␣
↪default
temp = pyrtl.WireVector()  # temporary with width to be defined later
temp <<= data  # this this case temp will get the bitwdith of 8 from data
```

5.1.4 Constants

class pyrtl.wire.**Const**(*val*, *bitwidth=None*, *name=''*, *signed=False*, *block=None*)

Bases: *WireVector*

A WireVector representation of a constant value.

Converts from bool, integer, or Verilog-style strings to a constant of the specified bitwidth. If the bitwidth is too short to represent the specified constant, then an error is raised. If a positive integer is specified, the bitwidth can be inferred from the constant. If a negative integer is provided in the simulation, it is converted to a two's complement representation of the specified bitwidth.

__init__(*val*, *bitwidth=None*, *name=''*, *signed=False*, *block=None*)

Construct a constant implementation at initialization.

Parameters

- **val** (*int, bool, or str*) – the value for the const WireVector
- **bitwidth** (*int*) – the desired bitwidth of the resulting const
- **signed** (*bool*) – specify if bits should be used for two's complement

Returns
a WireVector object representing a const wire

Descriptions for all parameters not listed above can be found at *WireVector.__init__()*

For details of how constants are converted fron int, bool, and strings (for verilog constants), see documentation for the helper function `infer_val_and_bitwidth`. Please note that a constant generated with `signed=True` is still just a raw bitvector and all arthimetic on it is unsigned by default. The `signed=True` argument is only used for proper inference of WireVector size and certain bitwidth sanity checks assuming a two's complement representation of the constants.

5.1.5 Conditionals

Conditional assignment of registers and WireVectors based on a predicate.

The management of selected assignments is expected to happen through the "with" blocks which will ensure that the region of execution for which the condition should apply is well defined. It is easiest to see with an example:

```
r1 = Register()
r2 = Register()
w3 = WireVector()
with conditional_assignment:
    with a:
        r1.next |= i  # set when a is true
        with b:
            r2.next |= j  # set when a and b are true
    with c:
        r1.next |= k  # set when a is false and c is true
        r2.next |= k
    with otherwise:
        r2.next |= l  # a is false and c is false

    with d:
        w3.next |= m  # d is true (assignments must be independent)
```

This is equivalent to:

```
r1.next <<= select(a, i, select(c, k, default))
r2.next <<= select(a, select(b, j, default), select(c, k, l))
w3 <<= select(d, m, 0)
```

This functionality is provided through two instances: `conditional_update`, which is a context manager (under which conditional assignements can be made), and `otherwise`, which is an instance that stands in for a 'fall through' case. The details of how these should be used, and the difference between normal assignments and condtional assignments, described in more detail in the state machine example in `examples/example3-statemachine.py`.

There are instances where you might want a wirevector to be set to a certain value in all but certain with blocks. For example, say you have a processor with a PC register that is normally updated to PC + 1 after each cycle, except when the current instruction is a branch or jump. You could represent that as follows:

```
pc = pyrtl.Register(32)
instr = pyrtl.WireVector(32)
res = pyrtl.WireVector(32)

op = instr[:7]
ADD = 0b0110011
JMP = 0b1101111

with conditional_assignment(
    defaults={
        pc: pc + 1,
        res: 0
    }
):
    with op == ADD:
        res |= instr[15:20] + instr[20:25]
        # pc will be updated to pc + 1
    with op == JMP:
        pc.next |= pc + instr[7:]
        # res will be set to 0
```

In addition to the conditional context, there is a helper function *currently_under_condition()* which will test if the code where it is called is currently elaborating hardware under a condition.

pyrtl.conditional.**currently_under_condition**()

Returns True if execution is currently in the context of a `_ConditionalAssignment`.

pyrtl.**otherwise = <pyrtl.conditional._Otherwise object>**

Context providing functionality of PyRTL `otherwise`.

pyrtl.**conditional_assignment = <pyrtl.conditional._ConditionalAssignment object>**

5.2 Registers and Memories

5.2.1 Registers

The class *Register* is derived from *WireVector*, and so can be used just like any other *WireVector*. Reading a register produces the stored value available in the current cycle. The stored value for the following cycle can be set by assigning to property *next* with the <<= (*__ilshift__()*) operator. Registers reset to zero by default, and reside in the same clock domain.

class pyrtl.wire.**Register**(*bitwidth*, *name=''*, *reset_value=None*, *block=None*)

Bases: *WireVector*

A WireVector with a register state element embedded.

Registers only update their outputs on posedge of an implicit clock signal. The "value" in the current cycle can be accessed by just referencing the Register itself. To set the value for the next cycle (after the next posedge) you write to the property *next* with the <<= operator. For example, if you want to specify a counter it would look like: `a.next <<= a + 1`

__init__(*bitwidth*, *name=''*, *reset_value=None*, *block=None*)

Construct a register.

Parameters

- **bitwidth** (*int*) – Number of bits to represent this register.
- **name** (*str*) – The name of the wire. Must be unique. If none is provided, one will be autogenerated.
- **reset_value** – Value to initialize this register to during simulation and in any code (e.g. Verilog) that is exported. Defaults to 0, but can be explicitly overridden at simulation time.
- **block** – The block under which the wire should be placed. Defaults to the working block.

Returns

a WireVector object representing a register.

It is an error if the `reset_value` cannot fit into the specified bitwidth for this register.

property next

This property is the way to set what the WireVector will be the next cycle (aka, it is before the D-Latch)

5.2.2 Memories

class pyrtl.memory.**MemBlock**(*bitwidth*, *addrwidth*, *name=''*, *max_read_ports=2*, *max_write_ports=1*, *asynchronous=False*, *block=None*)

MemBlock is the object for specifying block memories. It can be indexed like an array for both reading and writing. Writes under a conditional are automatically converted to enabled writes. For example, consider the following examples where `addr`, `data`, and `we` are all WireVectors:

```
data = memory[addr]  # infer read port
memory[addr] <<= data  # infer write port
mem[address] <<= MemBlock.EnabledWrite(data, enable=we)
```

When the address of a memory is assigned to using an *EnabledWrite* object items will only be written to the memory when the enable WireVector is set to high (1).

class EnabledWrite(*data*, *enable*)

Allows for an enable bit for each write port, where data (the first field in the tuple) is the normal data address, and enable (the second field) is a one bit signal specifying that the write should happen (i.e. active high).

data

Alias for field number 0

enable

Alias for field number 1

__init__(*bitwidth*, *addrwidth*, *name=''*, *max_read_ports=2*, *max_write_ports=1*, *asynchronous=False*, *block=None*)

Create a PyRTL read-write memory.

Parameters

- **bitwidth** (*int*) – Defines the bitwidth of each element in the memory
- **addrwidth** (*int*) – The number of bits used to address an element of the memory. This also defines the size of the memory
- **name** (*basestring*) – The identifier for the memory
- **max_read_ports** – limits the number of read ports each block can create; passing *None* indicates there is no limit
- **max_write_ports** – limits the number of write ports each block can create; passing *None* indicates there is no limit
- **asynchronous** (*bool*) – If false make sure that memory reads are only done using values straight from a register. (aka make sure that the read is synchronous)
- **name** – Name of the memory. Defaults to an autogenerated name
- **block** – The block to add it to, defaults to the working block

It is best practice to make sure your block memory/fifos read/write operations start on a clock edge if you want them to synthesize into efficient hardware. MemBlocks will enforce this by making sure that you only address them with a register or input, unless you explicitly declare the memory as asynchronous with `asynchronous=True` flag. Note that asynchronous mems are, while sometimes very convenient and tempting, rarely a good idea. They can't be mapped to block RAMs in FPGAs and will be converted to registers by most design tools even though PyRTL can handle them with no problem. For any memory beyond a few hundred entries it is not a realistic option.

Each read or write to the memory will create a new *port* (either a read port or write port respectively). By default memories are limited to 2-read and 1-write port, but to keep designs efficient by default, but those values can be set as options. Note that memories with high numbers of ports may not be possible to map to physical memories such as block RAMs or existing memory hardware macros.

5.2.3 ROMs

class pyrtl.memory.**RomBlock**(*bitwidth*, *addrwidth*, *romdata*, *name=''*, *max_read_ports=2*, *build_new_roms=False*, *asynchronous=False*, *pad_with_zeros=False*, *block=None*)

Bases: *MemBlock*

PyRTL Read Only Memory.

RomBlocks are the read only memory block for PyRTL. They support the same read interface and normal memories, but they are cannot be written to (i.e. there are no write ports). The ROM must be initialized with some values and construction through the use of the `romdata` which is the memory for the system.

__init__(*bitwidth*, *addrwidth*, *romdata*, *name=''*, *max_read_ports=2*, *build_new_roms=False*, *asynchronous=False*, *pad_with_zeros=False*, *block=None*)

Create a Python Read Only Memory.

Parameters

- **bitwidth** (*int*) – The bitwidth of each item stored in the ROM
- **addrwidth** (*int*) – The bitwidth of the address bus (determines number of addresses)
- **romdata** – This can either be a function or an array (iterable) that maps an address as an input to a result as an output
- **name** (*str*) – The identifier for the memory
- **max_read_ports** – limits the number of read ports each block can create; passing `None` indicates there is no limit
- **build_new_roms** (*bool*) – indicates whether to create and pass new RomBlocks during `__getitem__` to avoid exceeding *max_read_ports*
- **asynchronous** (*bool*) – If false make sure that memory reads are only done using values straight from a register. (aka make sure that reads are synchronous)
- **pad_with_zeros** (*bool*) – If true, extend any missing romdata with zeros out until the size of the romblock so that any access to the rom is well defined. Otherwise, the simulation should throw an error on access of unintialized data. If you are generating verilog from the rom, you will need to specify a value for every address (in which case setting this to True will help), however for testing and simulation it useful to know if you are off the end of explicitly specified values (which is why it is False by default)
- **block** – The block to add to, defaults to the working block

5.3 Simulation and Testing

5.3.1 Simulation

class pyrtl.simulation.**Simulation**(*tracer=True*, *register_value_map={}*, *memory_value_map={}*, *default_value=0*, *block=None*)

A class for simulating blocks of logic step by step.

A Simulation step works as follows:

1. Registers are updated:
    1. (If this is the first step) With the default values passed in to the Simulation during instantiation and/or any reset values specified in the individual registers.
    2. (Otherwise) With their next values calculated in the previous step (`r` logic nets).
2. The new values of these registers as well as the values of block inputs are propagated through the combinational logic.
3. Memory writes are performed (`@` logic nets).
4. The current values of all wires are recorded in the trace.

5. The next values for the registers are saved, ready to be applied at the beginning of the next step.

Note that the register values saved in the trace after each simulation step are from *before* the register has latched in its newly calculated values, since that latching in occurs at the beginning of the *next* step.

In addition to the functions methods listed below, it is sometimes useful to reach into this class and access internal state directly. Of particular usefulness are:

- `.tracer`: stores the *SimulationTrace* in which results are stored
- `.value`: a map from every signal in the block to its current simulation value
- `.regvalue`: a map from register to its value on the next tick
- `.memvalue`: a map from memid to a dictionary of address: value

__init__(*tracer=True*, *register_value_map={}*, *memory_value_map={}*, *default_value=0*, *block=None*)

Creates a new circuit simulator.

Parameters

- **tracer** (SimulationTrace) – Stores execution results. Defaults to a new *SimulationTrace* with no params passed to it. If None is passed, no tracer is instantiated (which is good for long running simulations). If the default (true) is passed, Simulation will create a new tracer automatically which can be referenced by the member variable `.tracer`
- **register_value_map** (*dict[Register, int]*) – Defines the initial value for the registers specified; overrides the registers's `reset_value`.
- **memory_value_map** – Defines initial values for many addresses in a single or multiple memory. Format: {Memory: {address: Value}}. Memory is a memory block, address is the address of a value
- **default_value** (*int*) – The value that all unspecified registers and memories will initialize to (default 0). For registers, this is the value that will be used if the particular register doesn't have a specified `reset_value`, and isn't found in the *register_value_map*.
- **block** (Block) – the hardware block to be traced (which might be of type *PostSynthBlock*). Defaults to the working block

Warning: Simulation initializes some things when called with *__init__()*, so changing items in the block for Simulation will likely break the simulation.

inspect(*w*)

Get the value of a WireVector in the last simulation cycle.

Parameters

w (*str*) – the name of the WireVector to inspect (passing in a WireVector instead of a name is deprecated)

Returns

value of w in the current step of simulation

Will throw KeyError if w does not exist in the simulation.

Example:

```
sim.inspect('a') == 10  # returns value of wire 'a' at current step
```

inspect_mem(*mem*)

Get the values in a map during the current simulation cycle.

Parameters
mem – the memory to inspect

Returns
{address: value}

Note that this returns the current memory state. Modifying the dictonary will also modify the state in the simulator

step(*provided_inputs*)

Take the simulation forward one cycle.

Parameters
provided_inputs – a dictionary mapping WireVectors to their values for this step

A step causes the block to be updated as follows, in order:

1. Registers are updated with their *next* values computed in the previous cycle
2. Block inputs and these new register values propagate through the combinational logic
3. Memories are updated
4. The *next* values of the registers are saved for use in step 1 of the next cycle.

All input wires must be in the *provided_inputs* in order for the simulation to accept these values.

Example: if we have inputs named `a` and `x`, we can call:

```
sim.step({'a': 1, 'x': 23})
```

to simulate a cycle with values 1 and 23 respectively.

step_multiple(*provided_inputs={}*, *expected_outputs={}*, *nsteps=None*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *stop_after_first_error=False*)

Take the simulation forward N cycles, based on the number of values for each input

Parameters

- **provided_inputs** – a dictionary mapping WireVectors to their values for N steps
- **expected_outputs** – a dictionary mapping WireVectors to their expected values for N steps; use ? to indicate you don't care what the value at that step is
- **nsteps** – number of steps to take (defaults to None, meaning step for each supplied input value)
- **file** – where to write the output (if there are unexpected outputs detected)
- **stop_after_first_error** – a boolean flag indicating whether to stop the simulation after encountering the first error (defaults to False)

All input wires must be in the *provided_inputs* in order for the simulation to accept these values. Additionally, the length of the array of provided values for each input must be the same.

When *nsteps* is specified, then it must be *less than or equal* to the number of values supplied for each input when *provided_inputs* is non-empty. When *provided_inputs* is empty (which may be a legitimate case for a design that takes no inputs), then *nsteps* will be used. When *nsteps* is not specified, then the simulation will take the number of steps equal to the number of values supplied for each input.

Example: if we have inputs named `a` and `b` and output `o`, we can call:

```
sim.step_multiple({'a': [0,1], 'b': [23,32]}, {'o': [42, 43]})
```

to simulate 2 cycles, where in the first cycle `a` and `b` take on 0 and 23, respectively, and `o` is expected to have the value 42, and in the second cycle `a` and `b` take on 1 and 32, respectively, and `o` is expected to have the value 43.

If your values are all single digit, you can also specify them in a single string, e.g.:

```
sim.step_multiple({'a': '01', 'b': '01'})
```

will simulate 2 cycles, with `a` and `b` taking on 0 and 0, respectively, on the first cycle and 1 and 1, respectively, on the second cycle.

Example: if the design had no inputs, like so:

```
a = pyrtl.Register(8)
b = pyrtl.Output(8, 'b')

a.next <<= a + 1
b <<= a

sim = pyrtl.Simulation()
sim.step_multiple(nsteps=3)
```

Using `sim.step_multiple(nsteps=3)` simulates 3 cycles, after which we would expect the value of `b` to be 2.

5.3.2 Fast (JIT to Python) Simulation

class pyrtl.simulation.**FastSimulation**(*register_value_map={}*, *memory_value_map={}*, *default_value=0*, *tracer=True*, *block=None*, *code_file=None*)

A class for running JIT-to-python implementations of blocks.

A Simulation step works as follows:

1. Registers are updated:
    1. (If this is the first step) With the default values passed in to the Simulation during instantiation and/or any reset values specified in the individual registers.
    2. (Otherwise) With their next values calculated in the previous step (`r` logic nets).
2. The new values of these registers as well as the values of block inputs are propagated through the combinational logic.
3. Memory writes are performed (`@` logic nets).
4. The current values of all wires are recorded in the trace.
5. The next values for the registers are saved, ready to be applied at the beginning of the next step.

Note that the register values saved in the trace after each simulation step are from *before* the register has latched in its newly calculated values, since that latching in occurs at the beginning of the *next* step.

__init__(*register_value_map={}*, *memory_value_map={}*, *default_value=0*, *tracer=True*, *block=None*, *code_file=None*)

Instantiates a Fast Simulation instance.

The interface for FastSimulation and Simulation should be almost identical. In addition to the Simulation arguments, FastSimulation additionally takes:

Parameters
code_file – The file in which to store a copy of the generated Python code. Defaults to no code being stored.

Look at *Simulation.__init__()* for descriptions for the other parameters.

This builds the Fast Simulation compiled Python code, so all changes to the circuit after calling this function will not be reflected in the simulation.

inspect(*w*)

Get the value of a WireVector in the last simulation cycle.

Parameters
w (*str*) – the name of the WireVector to inspect (passing in a WireVector instead of a name is deprecated)

Returns
value of *w* in the current step of simulation

Will throw KeyError if *w* is not being tracked in the simulation.

inspect_mem(*mem*)

Get the values in a map during the current simulation cycle.

Parameters
mem – the memory to inspect

Returns
{address: value}

Note that this returns the current memory state. Modifying the dictonary will also modify the state in the simulator

step(*provided_inputs*)

Run the simulation for a cycle.

Parameters
provided_inputs – a dictionary mapping WireVectors (or their names) to their values for this step (eg: *{wire: 3, "wire_name": 17}*)

A step causes the block to be updated as follows, in order:

1. Registers are updated with their *next* values computed in the previous cycle
2. Block inputs and these new register values propagate through the combinational logic
3. Memories are updated
4. The *next* values of the registers are saved for use in step 1 of the next cycle.

step_multiple(*provided_inputs={}*, *expected_outputs={}*, *nsteps=None*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *stop_after_first_error=False*)

Take the simulation forward N cycles, where N is the number of
values for each provided input.

Parameters

- **provided_inputs** – a dictionary mapping WireVectors to their values for N steps
- **expected_outputs** – a dictionary mapping WireVectors to their expected values for N steps; use ? to indicate you don't care what the value at that step is

- **nsteps** – number of steps to take (defaults to None, meaning step for each supplied input value)
- **file** – where to write the output (if there are unexpected outputs detected)
- **stop_after_first_error** – a boolean flag indicating whether to stop the simulation after the step where the first errors are encountered (defaults to False)

All input wires must be in the *provided_inputs* in order for the simulation to accept these values. Additionally, the length of the array of provided values for each input must be the same.

When *nsteps* is specified, then it must be *less than or equal* to the number of values supplied for each input when *provided_inputs* is non-empty. When *provided_inputs* is empty (which may be a legitimate case for a design that takes no inputs), then *nsteps* will be used. When *nsteps* is not specified, then the simulation will take the number of steps equal to the number of values supplied for each input.

Example: if we have inputs named `a` and `b` and output `o`, we can call:

```
sim.step_multiple({'a': [0,1], 'b': [23,32]}, {'o': [42, 43]})
```

to simulate 2 cycles, where in the first cycle `a` and `b` take on 0 and 23, respectively, and `o` is expected to have the value 42, and in the second cycle `a` and `b` take on 1 and 32, respectively, and `o` is expected to have the value 43.

If your values are all single digit, you can also specify them in a single string, e.g.:

```
sim.step_multiple({'a': '01', 'b': '01'})
```

will simulate 2 cycles, with `a` and `b` taking on 0 and 0, respectively, on the first cycle and 1 and 1, respectively, on the second cycle.

Example: if the design had no inputs, like so:

```
a = pyrtl.Register(8)
b = pyrtl.Output(8, 'b')

a.next <<= a + 1
b <<= a

sim = pyrtl.Simulation()
sim.step_multiple(nsteps=3)
```

Using `sim.step_multiple(nsteps=3)` simulates 3 cycles, after which we would expect the value of `b` to be 2.

5.3.3 Compiled (JIT to C) Simulation

class pyrtl.compilesim.**CompiledSimulation**(*tracer=True*, *register_value_map={}*, *memory_value_map={}*, *default_value=0*, *block=None*)

Simulate a block, compiling to C for efficiency.

This module provides significant speed improvements over *FastSimulation*, at the cost of somewhat longer setup time. Generally this will do better than *FastSimulation* for simulations requiring over 1000 steps. It is not built to be a debugging tool, though it may help with debugging. Note that only *Input* and *Output* wires can be traced using CompiledSimulation. This code is still experimental, but has been used on designs of significant scale to good effect.

In order to use this, you need:

- A 64-bit processor
- GCC (tested on version 4.8.4)
- A 64-bit build of Python

If using the multiplication operand, only some architectures are supported:

- x86-64 / amd64
- arm64 / aarch64
- mips64 (untested)

default_value is currently only implemented for registers, not memories.

A Simulation step works as follows:

1. Registers are updated:
    1. (If this is the first step) With the default values passed in to the Simulation during instantiation and/or any reset values specified in the individual registers.
    2. (Otherwise) With their next values calculated in the previous step (`r` logic nets).
2. The new values of these registers as well as the values of block inputs are propagated through the combinational logic.
3. Memory writes are performed (`@` logic nets).
4. The current values of all wires are recorded in the trace.
5. The next values for the registers are saved, ready to be applied at the beginning of the next step.

Note that the register values saved in the trace after each simulation step are from *before* the register has latched in its newly calculated values, since that latching in occurs at the beginning of the *next* step.

__init__(*tracer=True*, *register_value_map={}*, *memory_value_map={}*, *default_value=0*, *block=None*)

inspect(*w*)

Get the latest value of the wire given, if possible.

inspect_mem(*mem*)

Get a view into the contents of a MemBlock.

run(*inputs*)

Run many steps of the simulation.

Parameters
inputs – A list of input mappings for each step; its length is the number of steps to be executed.

step(*inputs*)

Run one step of the simulation.

Parameters
inputs – A mapping from input names to the values for the step.

A step causes the block to be updated as follows, in order:

1. Registers are updated with their *next* values computed in the previous cycle
2. Block inputs and these new register values propagate through the combinational logic
3. Memories are updated

4. The `next` values of the registers are saved for use in step 1 of the next cycle.

step_multiple(*provided_inputs={}*, *expected_outputs={}*, *nsteps=None*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *stop_after_first_error=False*)

Take the simulation forward N cycles, where N is the number of values
for each provided input.

Parameters

- **provided_inputs** – a dictionary mapping wirevectors to their values for N steps
- **expected_outputs** – a dictionary mapping wirevectors to their expected values for N steps; use ? to indicate you don't care what the value at that step is
- **nsteps** – number of steps to take (defaults to None, meaning step for each supplied input value)
- **file** – where to write the output (if there are unexpected outputs detected)
- **stop_after_first_error** – a boolean flag indicating whether to stop the simulation after the step where the first errors are encountered (defaults to False)

All input wires must be in the *provided_inputs* in order for the simulation to accept these values. Additionally, the length of the array of provided values for each input must be the same.

When *nsteps* is specified, then it must be *less than or equal* to the number of values supplied for each input when *provided_inputs* is non-empty. When *provided_inputs* is empty (which may be a legitimate case for a design that takes no inputs), then *nsteps* will be used. When *nsteps* is not specified, then the simulation will take the number of steps equal to the number of values supplied for each input.

Example: if we have inputs named `a` and `b` and output `o`, we can call:

```
sim.step_multiple({'a': [0,1], 'b': [23,32]}, {'o': [42, 43]})
```

to simulate 2 cycles, where in the first cycle `a` and `b` take on 0 and 23, respectively, and `o` is expected to have the value 42, and in the second cycle `a` and `b` take on 1 and 32, respectively, and `o` is expected to have the value 43.

If your values are all single digit, you can also specify them in a single string, e.g.:

```
sim.step_multiple({'a': '01', 'b': '01'})
```

will simulate 2 cycles, with `a` and `b` taking on 0 and 0, respectively, on the first cycle and 1 and 1, respectively, on the second cycle.

Example: if the design had no inputs, like so:

```
a = pyrtl.Register(8)
b = pyrtl.Output(8, 'b')

a.next <<= a + 1
b <<= a

sim = pyrtl.Simulation()
sim.step_multiple(nsteps=3)
```

Using `sim.step_multiple(nsteps=3)` simulates 3 cycles, after which we would expect the value of `b` to be 2.

5.3.4 Simulation Trace

class pyrtl.simulation.**SimulationTrace**(*wires_to_track=None*, *block=None*)

Storage and presentation of simulation waveforms.

__init__(*wires_to_track=None*, *block=None*)

Creates a new Simulation Trace

Parameters

- **wires_to_track** – The wires that the tracer should track. If unspecified, will track all explicitly-named wires. If set to `'all'`, will track all wires, including internal wires.
- **block** – Block containing logic to trace

add_fast_step(*fastsim*)

Add the *fastsim* context to the trace.

add_step(*value_map*)

Add the values in *value_map* to the end of the trace.

print_perf_counters(**trace_names*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print performance counter statistics for *trace_names*.

Parameters

- **trace_names** (*str*) – List of trace names. Each trace must be a single-bit wire.
- **file** – The place to write output, defaults to stdout.

This function prints the number of cycles where each trace's value is one. This is useful for counting the number of times important events occur in a simulation, such as cache misses and branch mispredictions.

print_trace(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *base=10*, *compact=False*)

Prints a list of wires and their current values.

Parameters

- **base** (*int*) – the base the values are to be printed in
- **compact** (*bool*) – whether to omit spaces in output lines

print_vcd(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *include_clock=False*)

Print the trace out as a VCD File for use in other tools.

Parameters

- **file** – file to open and output vcd dump to.
- **include_clock** – boolean specifying if the implicit clk should be included.

Dumps the current trace to file as a value change dump file. The file parameter defaults to `stdout` and the *include_clock* defaults to False.

Examples:

```
sim_trace.print_vcd()
sim_trace.print_vcd("my_waveform.vcd", include_clock=True)
```

render_trace(*trace_list=None*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *renderer=<pyrtl.simulation.WaveRenderer object>*, *symbol_len=None*, *repr_func=<built-in function hex>*, *repr_per_name={}*, *segment_size=1*)

Render the trace to a file using unicode and ASCII escape sequences.

Parameters

- **trace_list** (list[str]) – A list of signal names to be output in the specified order.
- **file** – The place to write output, default to stdout.
- **renderer** (*WaveRenderer*) – An object that translates traces into output bytes.
- **symbol_len** (int) – The "length" of each rendered value in characters. If None, the length will be automatically set such that the largest represented value fits.
- **repr_func** (Callable) – Function to use for representing each value in the trace. Examples include hex, oct, bin, and str (for decimal), *val_to_signed_integer()* (for signed decimal) or the function returned by *enum_name()* (for IntEnum). Defaults to hex.
- **repr_per_name** (dict) – Map from signal name to a function that takes in the signal's value and returns a user-defined representation. If a signal name is not found in the map, the argument repr_func will be used instead.
- **segment_size** (int) – Traces are broken in the segments of this number of cycles.

The resulting output can be viewed directly on the terminal or looked at with **more** or **less -R** which both should handle the ASCII escape sequences used in rendering.

5.3.5 Wave Renderer

class pyrtl.simulation.**WaveRenderer**(*constants*)

Render a SimulationTrace to the terminal.

See examples/renderer-demo.py, which renders traces with various options. You can choose a default renderer by exporting the PYRTL_RENDERER environment variable. See the documentation for subclasses of RendererConstants.

__init__(*constants*)

Instantiate a WaveRenderer.

Parameters

constants – Subclass of RendererConstants that specifies the ASCII/Unicode characters to use for rendering waveforms.

pyrtl.simulation.**enum_name**(*EnumClass*)

Returns a function that returns the name of an enum value as a string.

Use enum_name as a repr_func or repr_per_name for *SimulationTrace.render_trace()* to display enum names, instead of their numeric value, in traces. Example:

```
class State(enum.IntEnum):
    FOO = 0
    BAR = 1
state = Input(name='state', bitwidth=1)
sim = Simulation()
sim.step_multiple({'state': [State.FOO, State.BAR]})
```

(continues on next page)

(continued from previous page)

```
# Generates a trace like:
#       |0  |1
#
# state FOO|BAR
sim.tracer.render_trace(repr_per_name={'state': enum_name(State)})
```

Parameters
EnumClass (type) – enum to convert. This is the enum class, like State, not an enum value, like State.FOO or 1.

Return type
Callable[[int], str]

Returns
A function that accepts an enum value, like State.FOO or 1, and returns the value's name as a string, like 'FOO'.

class pyrtl.simulation.**PowerlineRendererConstants**

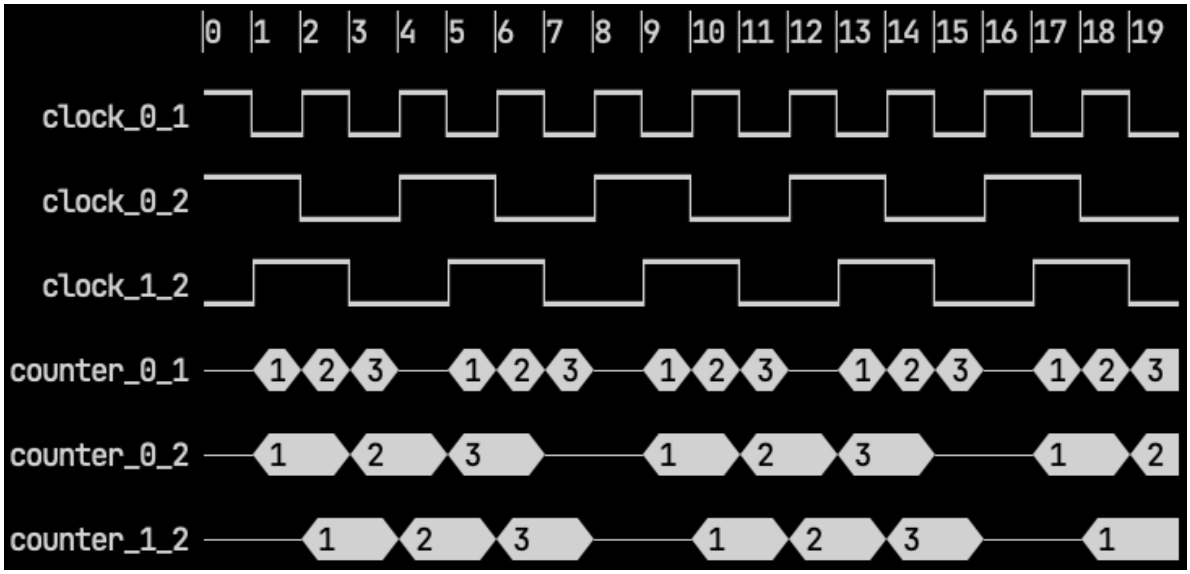
Bases: *Utf8RendererConstants*

Powerline renderer constants. Font must include powerline glyphs.

This render is closest to a traditional logic analyzer. Single-bit WireVectors are rendered as square waveforms, with vertical rising and falling edges. Multi-bit WireVector values are rendered in reverse-video hexagons.

This renderer requires a terminal font that supports Powerline glyphs

Enable this renderer by default by setting the PYRTL_RENDERER environment variable to powerline.



class pyrtl.simulation.**Utf8RendererConstants**

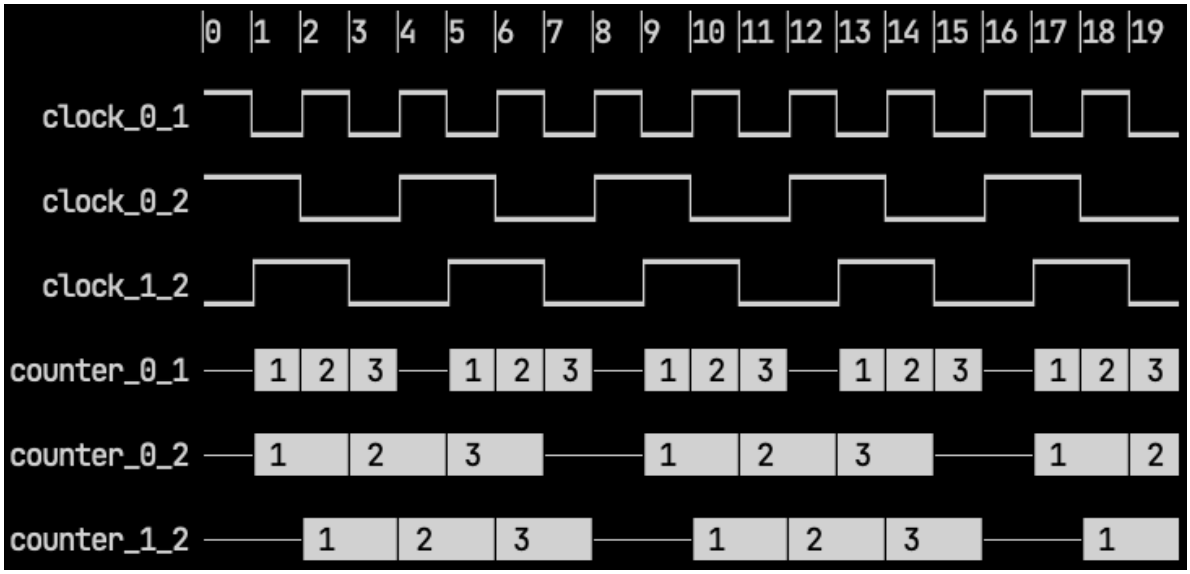
Bases: RendererConstants

UTF-8 renderer constants. These should work in most terminals.

Single-bit WireVectors are rendered as square waveforms, with vertical rising and falling edges. Multi-bit WireVector values are rendered in reverse-video rectangles.

This is the default renderer on non-Windows platforms.

Enable this renderer by default by setting the PYRTL_RENDERER environment variable to `utf-8`.



class pyrtl.simulation.**Utf8AltRendererConstants**

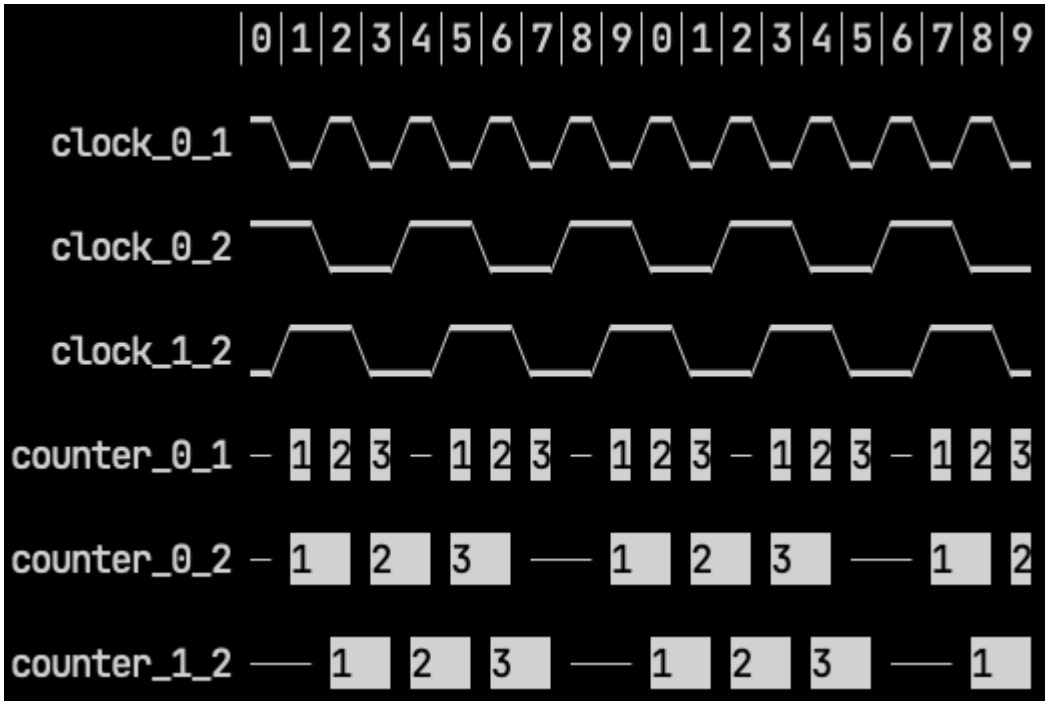
Bases: RendererConstants

Alternative UTF-8 renderer constants.

Single-bit WireVectors are rendered as waveforms with sloped rising and falling edges. Multi-bit WireVector values are rendered in reverse-video rectangles.

Compared to *Utf8RendererConstants*, this renderer is more compact because it uses one character between cycles instead of two.

Enable this renderer by default by setting the PYRTL_RENDERER environment variable to `utf-8-alt`.



class pyrtl.simulation.**Cp437RendererConstants**

Bases: RendererConstants

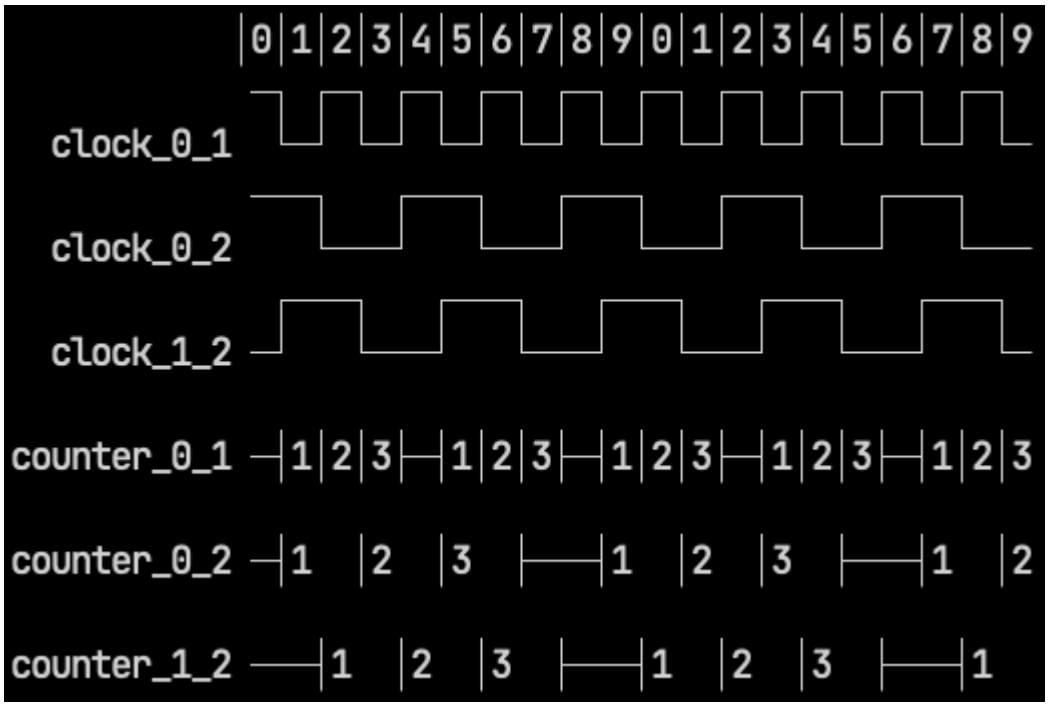
Code page 437 renderer constants (for windows cmd compatibility).

Single-bit WireVectors are rendered as square waveforms, with vertical rising and falling edges. Multi-bit WireVector values are rendered between vertical bars.

Code page 437 is also known as 8-bit ASCII. This is the default renderer on Windows platforms.

Compared to *Utf8RendererConstants*, this renderer is more compact because it uses one character between cycles instead of two, but the wire names are vertically aligned at the bottom of each waveform.

Enable this renderer by default by setting the PYRTL_RENDERER environment variable to cp437.



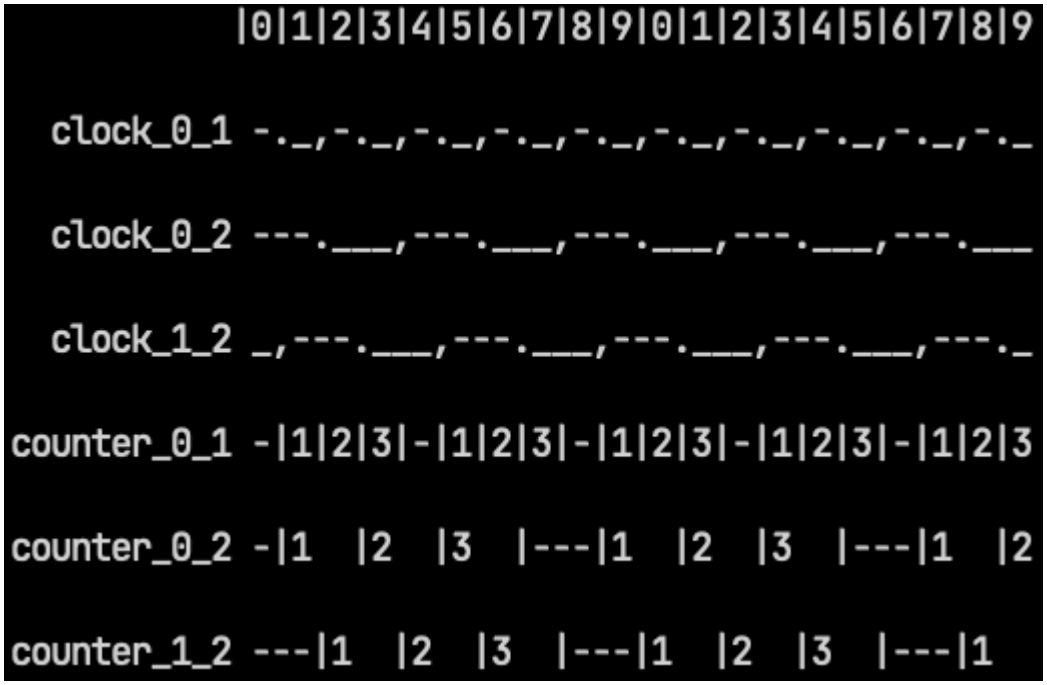
class pyrtl.simulation.**AsciiRendererConstants**

Bases: RendererConstants

7-bit ASCII renderer constants. These should work anywhere.

Single-bit WireVectors are rendered as waveforms with sloped rising and falling edges. Multi-bit WireVector values are rendered between vertical bars.

Enable this renderer by default by setting the PYRTL_RENDERER environment variable to ascii.



5.4 Logic Nets and Blocks

5.4.1 LogicNets

class pyrtl.core.**LogicNet**(*op*, *op_param*, *args*, *dests*)

The basic immutable datatype for storing a "net" in a netlist.

This is used for the internal representation that Python stores knowledge of what this is, and how it is used is only required for advanced users of PyRTL.

A 'net' is a structure in Python that is representative of hardware logic operations. These include binary operations, such as *and or* and *not*, arithmetic operations such as + and -, as well as other operations such as Memory ops, and concat, split, wire, and reg logic.

The details of what is allowed in each of these fields is defined in the comments of *Block*, and is checked by *Block.sanity_check()*

Logical Operations:

op	op_param	args	dests	
`&`	*None*	*a1, a2*	*out*	AND two wires together, put result into *out*
`|`	*None*	*a1, a2*	*out*	OR two wires together, put result into *out*
`^`	*None*	*a1, a2*	*out*	XOR two wires together, put result into *out*
`n`	*None*	*a1, a2*	*out*	NAND two wires together, put result into *out*
`~`	*None*	*a1*	*out*	invert one wire, put result into *out*
`+`	*None*	*a1, a2*	*out*	add *a1* and *a2*, put result into *out* `len(out) == max(len(a1), len(a2)) + 1` works with both unsigned and two's complement
`-`	*None*	*a1, a2*	*out*	subtract *a2* from *a1*, put result into *out* `len(out) == max(len(a1), len(a2)) + 1` works with both unsigned and two's complement
`*`	*None*	*a1, a2*	*out*	multiply *a1* & *a2*, put result into *out* `len(out) == len(a1) + len(a2)` assumes unsigned, but *signed_mult()* provides wrapper
`=`	*None*	*a1, a2*	*out*	check *a1* & *a2* equal, put result into *out* (0 | 1)
`<`	*None*	*a1, a2*	*out*	check *a2* greater than *a1*, put result into *out* (0 | 1)
`>`	*None*	*a1, a2*	*out*	check *a1* greater than *a2*, put result into *out* (0 | 1)
`w`	*None*	*w1*	*w2*	connects *w1* to *w2* directional wire with no logical operation

5.4.2 Blocks

class pyrtl.core.**Block**

Block encapsulates a netlist.

A Block in PyRTL is the class that stores a netlist and provides basic access and error checking members. Each block has well defined inputs and outputs, and contains both the basic logic elements and references to the wires and memories that connect them together.

The logic structure is primarily contained in `Block.logic` which holds a set of *LogicNets*. Each *LogicNet* describes a primitive operation (such as an adder or memory). The primitive is described by a 4-tuple of:

1) the op (a single character describing the operation such as + or `r`),
2) a set of hard-wired *op_params* (such as the constants to select from the "selection" op).
3) the tuple *args* which list the WireVectors hooked up as inputs to this particular net.
4) the tuple *dests* which list the WireVectors hooked up as output for this particular net.

Below is a list of the basic operations. These properties (more formally specified) should all be checked by the class method *Block.sanity_check()*.

- Most logical and arithmetic ops are pretty self explanatory. Each takes exactly two arguments, and they should perform the arithmetic or logical operation specified.

  OPS: `&`, `|`, `^`, `n`, ~, +, -, *.

  All inputs must be the same bitwidth. Logical operations produce as many bits as are in the input, while + and - produce n+1 bits, and * produces 2n bits.

- In addition there are some operations for performing comparisons that should perform the operation specified. The = op is checking to see if the bits of the vectors are equal, while < and > do unsigned arithmetic comparison. All comparisons generate a single bit of output (1 for true, 0 for false).
- The `w` operator is simply a directional wire and has no logic function.
- The `x` operator is a multiplexer which takes a select bit and two signals. If the value of the select bit is 0 it selects the second argument; if it is 1 it selects the third argument. Select must be a single bit, while the other two arguments must be the same length.
- The `c` operator is the concatenation operator and combines any number of WireVectors (`a`, `b`, ..., `z`) into a single new WireVector with `a` in the MSB and `z` (or whatever is last) in the LSB position.
- The `s` operator is the selection operator and chooses, based on the *op_param* specified, a subset of the logic bits from a WireVector to select. Repeats are accepted.
- The `r` operator is a register and on posedge, simply copies the value from the input to the output of the register.
- The `m` operator is a memory block read port, which supports async reads (acting like combinational logic). Multiple read (and write) ports are possible to the same memory but each `m` defines only one of those. The *op_param* is a tuple containing two references: the mem id, and a reference to the MemBlock containing this port. The MemBlock should only be used for debug and sanity checks. Each read port has one *addr* (an arg) and one *data* (a dest).
- The `@` (update) operator is a memory block write port, which supports synchronous writes (writes are "latched" at positive edge). Multiple write (and read) ports are possible to the same memory but each `@` defines only one of those. The *op_param* is a tuple containing two references: the mem id, and a reference to the MemoryBlock. Writes have three args (*addr*, *data*, and write enable *we_en*). The dests should be an empty tuple. You will not see a written value change until the following cycle. If multiple writes happen to the same address in the same cycle the behavior is currently undefined.

The connecting elements (args and dests) should be WireVectors or derived from WireVector, and should be registered with the block using *Block.add_wirevector()*. Nets should be registered using *Block.add_net()*.

In addition, there is a member `Block.legal_ops` which defines the set of operations that can be legally added to the block. By default it is set to all of the above defined operations, but it can be useful in certain cases to only allow a subset of operations (such as when transforms are being done that are "lowering" the blocks to more primitive ops).

pyrtl.core.**working_block**(*block=None*)

Convenience function for capturing the current working block.

If a block is not passed, or if the block passed is None, then this will return the "current working block". However, if a block is passed in it will simply return that block instead. This feature is useful in allowing functions to "override" the current working block.

pyrtl.core.**reset_working_block**()

Reset the working block to be empty.

pyrtl.core.**set_working_block**(*block*, *no_sanity_check=False*)

Set the working block to be the block passed as argument. Compatible with the *with* statement.

Sanity checks will only be run if the new block is different from the original block.

pyrtl.core.**temp_working_block**()

Set the working block to be new temporary block.

If used with the *with* statement the block will be reset to the original value (at the time of call) at exit of the context.

pyrtl.core.Block.**add_wirevector**(*self*, *wirevector*)

Add a WireVector object to the block.

Parameters
wirevector (WireVector) – WireVector object added to block

pyrtl.core.Block.**remove_wirevector**(*self*, *wirevector*)

Remove a WireVector object from the block.

Parameters
wirevector (WireVector) – WireVector object removed from block

pyrtl.core.Block.**add_net**(*self*, *net*)

Add a net to the logic of the block.

Parameters
net (LogicNet) – LogicNet object added to block

The passed net, which must be of type LogicNet, is checked and then added to the block. No wires are added by this member, they must be added seperately with *Block.add_wirevector()*.

pyrtl.core.Block.**get_memblock_by_name**(*self*, *name*, *strict=False*)

Get a reference to a memory stored in this block by name.

Parameters

- **name** (*str*) – name of MemBlock object
- **strict** (*bool*) – Determines if PyrtlError or None is thrown on no match. Defaults to False.

Returns
a MemBlock object with specified name

By fallthrough, if a matching MemBlock cannot be found the value None is returned. However, if the argument strict is set to True, then this will instead throw a PyrtlError when no match is found.

This useful for when a block defines its own internal memory block, and during simulation you want to instantiate that memory with certain values for testing. Since the Simulation constructor requires a reference to the memory object itself, but the block you're testing defines the memory internally, this allows you to get the object reference.

Note that this requires you know the name of the memory block, meaning that you most likely need to have named it yourself.

Example:

```
def special_memory(read_addr, write_addr, data, wen):
    mem = pyrtl.MemBlock(bitwidth=32, addrwidth=5, name='special_mem')
    mem[write_addr] <<= pyrtl.MemBlock.EnabledWrite(data, wen & (write_addr > 0))
    return mem[read_addr]

read_addr = pyrtl.Input(5, 'read_addr')
write_addr = pyrtl.Input(5, 'write_addr')
data = pyrtl.Input(32, 'data')
wen = pyrtl.Input(1, 'wen')
res = pyrtl.Output(32, 'res')

res <<= special_memory(read_addr, write_addr, data, wen)

# Can only access it after the `special_memory` block has been instantiated/called
special_mem = pyrtl.working_block().get_memblock_by_name('special_mem')

sim = pyrtl.Simulation(memory_value_map={
    special_mem: {
        0: 5,
        1: 6,
        2: 7,
    }
})

inputs = {
    'read_addr':  '012012',
    'write_addr': '012012',
    'data':       '890333',
    'wen':        '111000',
}
expected = {
    'res': '567590',
}
sim.step_multiple(inputs, expected)
```

pyrtl.core.Block.**wirevector_subset**(*self*, *cls=None*, *exclude=()*)

Return set of WireVectors, filtered by the type or tuple of types provided as *cls*.

Parameters

- **cls** – Type of returned WireVector objects
- **exclude** – Type of WireVector objects to exclude

Returns

Set of WireVector objects that are both a cls type and not a excluded type

If no *cls* is specified, the full set of WireVectors associated with the Block are returned. If *cls* is a single type, or a tuple of types, only those WireVectors of the matching types will be returned. This is helpful for getting all inputs, outputs, or registers of a block for example.

Examples:

```
inputs = pyrtl.working_block().wirevector_subset(pyrtl.Input)
outputs = pyrtl.working_block().wirevector_subset(pyrtl.Output)

# returns set of all non-input WireVectors
non_inputs = pyrtl.working_block().wirevector_subset(exclude=pyrtl.Input)
```

pyrtl.core.Block.**logic_subset**(*self*, *op=None*)

Return set of LogicNets, filtered by the type(s) of logic op provided as op.

Parameters
op – Operation of LogicNet to filter by. Defaults to None.

Returns
set of LogicNets with corresponding op

If no *op* is specified, the full set of LogicNets associated with the Block are returned. This is helpful for getting all memories of a block for example.

pyrtl.core.Block.**get_wirevector_by_name**(*self*, *name*, *strict=False*)

Return the WireVector matching name.

Parameters

- **name** (*str*) – name of WireVector object
- **strict** (*bool*) – Determines if PyrtlError or None is thrown on no match. Defaults to False.

Returns
a WireVector object with specified name

By fallthrough, if a matching WireVector cannot be found the value None is returned. However, if the argument strict is set to True, then this will instead throw a PyrtlError when no match is found.

pyrtl.core.Block.**net_connections**(*self*, *include_virtual_nodes=False*)

Returns a representation of the current block useful for creating a graph.

Parameters
include_virtual_nodes (*bool*) – if enabled, the wire itself will be used to signal an external source or sink (such as the source for an Input net). If disabled, these nodes will be excluded from the adjacency dictionaries

Returns
Two dictionaries: one that maps WireVectors to the logic net that creates their signal (*wire_src_dict*) and one that maps WireVectors to a list of logic nets that use the signal (*wire_sink_dict*).

These dictionaries make the creation of a graph much easier, as well as facilitate other places in which one would need wire source and wire sink information.

Look at *net_graph()* for one such graph that uses the information from this function.

pyrtl.core.Block.**sanity_check**(*self*)

Check block and throw PyrtlError or PyrtlInternalError if there is an issue.

Should not modify anything, only check data structures to make sure they have been built according to the assumptions stated in the Block comments.

5.5 Helper Functions

5.5.1 Cutting and Extending WireVectors

The functions below provide ways of combining, slicing, and extending *WireVectors* in ways that are often useful in hardware design. The functions below extend those member functions of the *WireVector* class itself (which provides support for the Python builtin `len`, slicing e.g. `wire[3:6]`, *zero_extended()*, *sign_extended()*, and many operators such as addition and multiplication).

pyrtl.corecircuits.**concat**(**args*)

Concatenates multiple WireVectors into a single WireVector.

Parameters
args (WireVector) – inputs to be concatenated

Returns
WireVector with length equal to the sum of the args' lengths

You can provide multiple arguments and they will be combined with the right-most argument being the least significant bits of the result. Note that if you have a list of arguments to concat together you will likely want index 0 to be the least significant bit and so if you unpack the list into the arguments here it will be backwards. The function *concat_list()* is provided for that case specifically.

Example using `concat` to combine two bytes into a 16-bit quantity:

```
concat(msb, lsb)
```

pyrtl.corecircuits.**concat_list**(*wire_list*)

Concatenates a list of WireVectors into a single WireVector.

Parameters
wire_list (*list[*WireVector*]*) – list of WireVectors to concat

Returns
WireVector with length equal to the sum of the args' lengths

This take a list of WireVectors and concats them all into a single WireVector with the element at index 0 serving as the least significant bits. This is useful when you have a variable number of WireVectors to concatenate, otherwise *concat()* is prefered.

Example using `concat_list` to combine two bytes into a 16-bit quantity:

```
mylist = [lsb, msb]
concat_list(mylist)
```

pyrtl.corecircuits.**match_bitwidth**(**args*, ***opt*)

Matches the argument wires' bitwidth via zero or sign extension, returning new WireVectors

Parameters

- **args** (WireVector) – WireVectors of which to match bitwidths

- **opt** – Optional keyword argument `signed=True` (defaults to False)

Returns
tuple of args in order with extended bits

Example of matching the bitwidths of two WireVectors `a` and `b` with zero extension:

```
a, b = match_bitwidth(a, b)
```

Example of matching the bitwidths of three WireVectors `a`, `b`, and `c` with with sign extension:

```
a, b, c = match_bitwidth(a, b, c, signed=True)
```

pyrtl.helperfuncs.**truncate**(*wirevector_or_integer*, *bitwidth*)

Returns a WireVector or integer truncated to the specified bitwidth

Parameters

- **wirevector_or_integer** – Either a WireVector or an integer to be truncated.
- **bitwidth** (*int*) – The length to which the first argument should be truncated.

Returns
A truncated WireVector or integer as appropriate.

This function truncates the most significant bits of the input, leaving a result that is only *bitwidth* bits wide. For integers this is performed with a simple bitmask of size *bitwidth*. For WireVectors the function calls *WireVector.truncate()* and returns a WireVector of the specified *bitwidth*.

Examples:

```
truncate(9,3)  # returns 1  (0b1001 truncates to 0b001)
truncate(5,3)  # returns 5  (0b101 truncates to 0b101)
truncate(-1,3)  # returns 7  (-0b1 truncates to 0b111)
y = truncate(x+1, x.bitwidth)  # y.bitwdith will equal x.bitwidth
```

pyrtl.helperfuncs.**chop**(*w*, **segment_widths*)

Returns a list of WireVectors, each a slice of the original *w*.

Parameters

- **w** (*WireVector*) – The WireVector to be chopped up into segments
- **segment_widths** (*int*) – Additional arguments are integers which are bitwidths

Returns
A list of WireVectors each with a proper segment width

Return type
List[*WireVector*]

This function chops a WireVector into a set of smaller WireVectors of different lengths. It is most useful when multiple "fields" are contained with a single WireVector, for example when breaking apart an instruction. For example, if you wish to break apart a 32-bit MIPS I-type (Immediate) instruction you know it has an 6-bit opcode, 2 5-bit operands, and 16-bit offset. You could take each of those slices in absolute terms: `offset=instr[0:16]`, `rt=instr[16:21]` and so on, but then you have to do the arithmetic yourself. With this function you can do all the fields at once which can be seen in the examples below.

As a check, chop will throw an error if the sum of the lengths of the fields given is not the same as the length of the WireVector to chop. Note also that chop assumes that the "rightmost" arguments are the least signficant bits (just like *concat()*) which is normal for hardware functions but makes the list order a little counter intuitive.

Examples:

```
opcode, rs, rt, offset = chop(instr, 6, 5, 5, 16)  # MIPS I-type instruction
opcode, instr_index = chop(instr, 6, 26)  # MIPS J-type instruction
opcode, rs, rt, rd, sa, function = chop(instr, 6, 5, 5, 5, 5, 6)  # MIPS R-type
msb, middle, lsb = chop(data, 1, 30, 1) # break out the most and least significant␣
↪bits
```

pyrtl.helperfuncs.**wire_struct**(*wire_struct_spec*)

Decorator that assigns names to `WireVector` slices.

`@wire_struct` assigns names to *non-overlapping* `WireVector` slices. Suppose we have an 8-bit wide `WireVector` called `byte`. We can refer to all 8 bits with the name `byte`, but `@wire_struct` lets us refer to slices by name, for example we could name the high 4 bits `byte.high` and the low 4 bits `byte.low`. Without `@wire_struct`, we would refer to these slices as `byte[4:8]` and `byte[0:4]`, which are prone to off-by-one errors and harder to read.

The example `Byte` `@wire_struct` can be defined as:

```
@wire_struct
class Byte:
    high: 4  # 'high' is name for the 4 most significant bits.
    low: 4   # 'low' is name for the 4 least significant bits.
```

Construction

Once a `@wire_struct` class is defined, it can be instantiated by providing drivers for all of its wires. This can be done in two ways:

1. Provide a driver for *each* component wire, for example:

   ```
   byte = Byte(high=0xA, low=0xB)
   ```

   Note how the component names (`high`, `low`) are used as keyword args for the constructor. Drivers must be provided for *all* components.

2. Provide a driver for the entire `@wire_struct`, for example:

   ```
   byte = Byte(Byte=0xAB)
   ```

   Note how the class name (`Byte`) is used as a keyword arg for the constructor.

Accessing Slices

After instantiating a `@wire_struct`, the instance functions as a `WireVector` containing all the wires. For example, `byte` functions as a `WireVector` with bitwidth 8:

```
byte = Byte(Byte=0xAB)
print(byte.bitwidth)  # Prints 8.
```

The named slice can be accessed through the `.` operator (`__getattr__`), for example `byte.high` and `byte.low`, which both function as `WireVector` with bitwidth 4:

```
byte = Byte(Byte=0xAB)
print(byte.high.bitwidth)  # Prints 4.
print(byte.low.bitwidth)  # Prints 4.
```

Both the instance and the slices are first-class `WireVector`, so they can be manipulated with all the usual PyRTL operators.

Note: `len(byte)` returns the number of components in the `@wire_struct` (2), not the total bitwidth (8 == 4 + 4). To get the total bitwidth, use `byte.bitwidth` or `len(as_wires(byte))`.

Naming

A `@wire_struct` can be assigned a name in the usual way:

```
byte = Byte(name='b', high=0xC, low=0xD)
byte = Byte(name='b', Byte=0xCD)
```

When a `@wire_struct` is assigned a name (`b`), its components will be assigned dotted names (`b.high`, `b.low`):

```
print(byte.high.name)  # Prints 'b.high'.
print(byte.low.name)  # Prints 'b.low'.
```

Warning: All `@wire_struct` names are only set during construction. You can later rename a `@wire_struct` or its components, but those changes are local, and will not propagate to other `@wire_struct` components. Renaming a `@wire_struct` or its components is strongly discouraged.

Composition

`@wire_struct` can be composed with itself, and with `wire_matrix`. For example, we can define a `Pixel` that contains three `Byte`:

```
@wire_struct
class Pixel:
    red: Byte
    green: Byte
    blue: Byte
```

Drivers must be specified for all components, but they can be specified at any level. All these examples construct an equivalent `@wire_struct`:

```
pixel = Pixel(Pixel=0xABCDEF)
pixel = Pixel(red=0xAB, green=0xCD, blue=0xEF)
pixel = Pixel(red=Byte(high=0xA, low=0xB), green=0xCD, blue=0xEF)
pixel = Pixel(red=Byte(high=0xA, low=0xB),
              green=Byte(high=0xC, low=0xD),
              blue=0xEF)
```

Hierarchical `@wire_struct` components are accessed by composing `.` operators:

```
pixel
pixel.red
pixel.red.high
pixel.red.low
pixel.green
pixel.green.high
pixel.green.low
pixel.blue
pixel.blue.high
pixel.blue.low
```

`@wire_struct` can be composed with `wire_matrix`:

```
Word = wire_matrix(component_schema=8, size=4)

@wire_struct
class CacheLine:
    address: Word
    data: Word
    valid: 1

cache_line = CacheLine(address=0x01234567, data=0x89ABCDEF, valid=1)
```

Leaf-level components can be accessed by combining the `.` and `[]` operators, for example `cache_line.address[3]`.

Types

You can change the type of a `@wire_struct`'s components to a `WireVector` subclass like `Input` or `Output` with the `component_type` constructor argument:

```
# Generates Outputs named ``output_byte.low`` and ``output_byte.high``.
output_byte = Byte(name='output_byte', component_type=pyrtl.Output,
                   Byte=0xCD)
```

You can also change the type of the `@wire_struct` itself with the `concatenated_type` cnstructor argument:

```
# Generates an Input named ``input_byte``.
input_byte = Byte(name='input_byte', concatenated_type=pyrtl.Input)
```

Note: No values are specified for `input_byte` because its value is not known until simulation time.

`pyrtl.helperfuncs.`**`wire_matrix`**(*component_schema*, *size*)

Returns a class that assigns numbered indices to `WireVector` slices.

`wire_matrix` assigns numbered indices to *non-overlapping* `WireVector` slices. `wire_matrix` is very similar to *wire_struct()*, so read *wire_struct()*'s documentation first.

An example 32-bit `Word` `wire_matrix`, which represents a group of four bytes, can be defined as:

```
Word = wire_matrix(component_schema=8, size=4)
```

Note: `wire_matrix` returns a class, like `namedtuple`.

Construction

Once a `wire_matrix` class is defined, it can be instantiated by providing drivers for all of its wires. This can be done in two ways:

```
# Provide a driver for each component, most significant bits first.
word = Word(values=[0x89, 0xAB, 0xCD, 0xEF])

# Provide a driver for all components.
word = Word(values=[0x89ABCDEF])
```

Note: When specifying drivers for each component, the most significant bits are specified first.

After instantiating a `wire_matrix`, regardless of how it was constructed, the instance functions as a `WireVector` containing all the wires, so `word` functions as a `WireVector` with bitwidth 32. The named slice can be accessed with square brackets (`__getitem__`), for example `word[0]` and `word[3]`, which both function as `WireVector` with bitwidth 8. `word[0]` refers to the most significant byte, and `word[3]` refers to the least significant byte. Both the instance and the slices are first-class `WireVector`, so they can be manipulated with all the usual PyRTL operators.

Naming

A `wire_matrix` can be assigned a name in the usual way:

```
# The whole Word is named 'w', so the components will have names
# w[0], w[1], ...
word = Word(name='w', values=[0x89, 0xAB, 0xCD, 0xEF])
word = Word(name='w', values=[0x89ABCDEF])
```

Composition

`wire_matrix` can be composed with itself and `@wire_struct`. For example, we can define some multi-dimensional byte arrays:

```
Array1D = wire_matrix(component_schema=8, size=2)
Array2D = wire_matrix(component_schema=Array1D, size=2)
```

Drivers must be specified for all components, but they can be specified at any level. All these examples construct an equivalent `wire_matrix`:

```
array_2d = Array2D(values=[0x89AB, 0xCDEF])
array_2d = Array2D(values=[Array1D(values=[0x89, 0xAB]),
                           0xCDEF])
array_2d = Array2D(values=[Array1D(values=[0x89, 0xAB]),
                           Array1D(values=[0xCD, 0xEF])])
```

Accessing Slices

Hierarchical components are accessed by composing `[]` operators, for example:

```
print(array_2d[0][0].bitwidth)  # Prints 8.
print(array_2d[0][1].bitwidth)  # Prints 8.
```

When `wire_matrix` is composed with `@wire_struct`, components can be accessed by combining the `[]` and `.` operators:

```
@wire_struct
class Byte:
    high: 4
    low: 4
Array1D = wire_matrix(component_schema=Byte, size=2)
array_1d = Array1D(values=[0xAB, 0xCD])

print(array_1d[0].high.bitwidth)  # Prints 4.
```

Note: `len(array_1d)` returns the number of components in the `wire_matrix` (2), not the total bitwidth (16 == 2 * 8). To get the total bitwidth, use `array_1d.bitwidth` or `len(as_wires(array_1d))`.

Types

You can change the type of a `wire_matrix`'s components with the `component_type` constructor argument:

```
# Generates Outputs named ``output_word[0]``, ``output_word[1]``, ...
word = Word(name='output_word',
            component_type=pyrtl.Output,
            values=[0x89ABCDEF])
```

You can change the type of the `wire_matrix` itself with the `concatenated_type` cnstructor argument:

```
# Generates an Input named ``input_word``.
word = Word(name='input_word', concatenated_type=pyrtl.Input)
```

Note: No values are specified for `input_word` because its value is not known until simulation time.

5.5.2 Coercion to WireVector

In PyRTL there is only one function in charge of coercing values into *WireVectors*, and that is *as_wires()*. This function is called in almost all helper functions and classes to manage the mixture of constants and WireVectors that naturally occur in hardware development.

`pyrtl.corecircuits.`**`as_wires`**(*val*, *bitwidth=None*, *truncating=True*, *block=None*)

Return wires from *val* which may be wires, integers (including IntEnums), strings, or bools.

Parameters

- **`val`** – a WireVector-like object or something that can be converted into a Const

- **bitwidth** (*int*) – The bitwidth the resulting wire should be
- **truncating** (*bool*) – determines whether bits will be dropped to achieve the desired bitwidth if it is too long (if true, the most-significant bits will be dropped)
- **block** (Block) – block to use for wire

This function is mainly used to coerce values into WireVectors (for example, operations such as `x + 1` where 1 needs to be converted to a Const WireVector). An example:

```
def myhardware(input_a, input_b):
    a = as_wires(input_a)
    b = as_wires(input_b)
myhardware(3, x)
```

as_wires() will convert the 3 to Const but keep `x` unchanged assuming it is a WireVector.

5.5.3 Control Flow Hardware

pyrtl.corecircuits.**mux**(*index*, **mux_ins*, ***kwargs*)

Multiplexer returning the value of the wire from *mux_ins* according to *index*.

Parameters

- **index** (WireVector) – used as the select input to the multiplexer
- **mux_ins** (WireVector) – additional WireVector arguments selected when select>1
- **kwargs** (WireVector) – additional WireVectors, keyword arg "default" If you are selecting between fewer items than your index can address, you can use the *default* keyword argument to auto-expand those terms. For example, if you have a 3-bit index but are selecting between 6 options, you need to specify a value for those other 2 possible values of index (0b110 and 0b111).

Returns

WireVector of length of the longest input (not including *index*)

To avoid confusion, if you are using the mux where the index is a "predicate" (meaning something that you are checking the truth value of rather than using it as a number) it is recommended that you use *select()* instead as named arguments because the ordering is different from the classic ternary operator of some languages.

Example of multiplexing between `a0` and `a1`:

```
index = WireVector(1)
mux(index, a0, a1)
```

Example of multiplexing between `a0`, `a1`, `a2`, `a3`:

```
index = WireVector(2)
mux(index, a0, a1, a2, a3)
```

Example of *default* to specify additional arguments:

```
index = WireVector(3)
mux(index, a0, a1, a2, a3, a4, a5, default=0)
```

pyrtl.corecircuits.**select**(*sel*, *truecase*, *falsecase*)

Multiplexer returning *falsecase* when `sel == 0`, otherwise *truecase*.

Parameters

- **sel** (WireVector) – used as the select input to the multiplexer
- **truecase** (WireVector) – the WireVector selected if `sel == 1`
- **falsecase** (WireVector) – the WireVector selected if `sel == 0`

The hardware this generates is exactly the same as *mux()* but by putting the true case as the first argument it matches more of the C-style ternary operator semantics which can be helpful for readability.

Example of taking the min of `a` and 5:

```
select(a < 5, truecase=a, falsecase=5)
```

pyrtl.corecircuits.**enum_mux**(*cntrl*, *table*, *default=None*, *strict=True*)

Build a mux for the control signals specified by an enum.

Parameters

- **cntrl** – is a WireVector and control for the mux.
- **table** – is a dictionary of the form mapping enum to WireVector.
- **default** – is a WireVector to use when the key is not present. In addition it is possible to use the key *otherwise* to specify a default value, but it is an error if both are supplied.
- **strict** (*bool*) – when True, check that the dictionary has an entry for every possible value in the enum. Note that if a default is set, then this check is not performed as the default will provide valid values for any underspecified keys.

Returns

a WireVector which is the result of the mux.

Examples:

```
from enum import IntEnum

class Command(IntEnum):
    ADD = 1
    SUB = 2
enum_mux(cntrl, {Command.ADD: a + b, Command.SUB: a - b})
enum_mux(cntrl, {Command.ADD: a + b}, strict=False)  # SUB case undefined
enum_mux(cntrl, {Command.ADD: a + b, otherwise: a - b})
enum_mux(cntrl, {Command.ADD: a + b}, default=a - b)
```

pyrtl.corecircuits.**bitfield_update**(*w*, *range_start*, *range_end*, *newvalue*, *truncating=False*)

Return WireVector *w* but with some of the bits overwritten by *newvalue*.

Parameters

- **w** (WireVector) – a WireVector to use as the starting point for the update
- **range_start** (*int*) – the start of the range of bits to be updated
- **range_end** (*int*) – the end of the range of bits to be updated
- **newvalue** (*int*) – the value to be written in to the start:end range
- **truncating** (*bool*) – if true, silently clip newvalue to the proper bitwidth rather than throw an error if the value provided is too large

Given a WireVector *w*, this function returns a new WireVector that is identical to *w* except in the range of bits specified. In that specified range, the value *newvalue* is swapped in. For example:

```
bitfield_update(w, 20, 23, 0x7)
```

will return a WireVector of the same length as *w*, and with the same values as *w*, but with bits 20, 21, and 22 all set to 1.

Note that *range_start* and *range_end* will be inputs to a slice and so standard Python slicing rules apply (e.g. negative values for end-relative indexing and support for None).

```
w = bitfield_update(w, 20, 23, 0x7)  # sets bits 20, 21, 22 to 1
w = bitfield_update(w, 20, 23, 0x6)  # sets bit 20 to 0, bits 21 and 22 to 1
w = bitfield_update(w, 20, None, 0x7)  # assuming w is 32 bits, sets bits 31..20 =␣
↪0x7
w = bitfield_update(w, -1, None, 0x1)  # set the MSB (bit) to 1
w = bitfield_update(w, None, -1, 0x9)  # set the bits before the MSB (bit) to 9
w = bitfield_update(w, None, 1, 0x1)  # set the LSB (bit) to 1
w = bitfield_update(w, 1, None, 0x9)  # set the bits after the LSB (bit) to 9
```

pyrtl.corecircuits.**bitfield_update_set**(*w*, *update_set*, *truncating=False*)

Return WireVector *w* but with some of the bits overwritten by values in *update_set*.

Parameters

- **w** (WireVector) – a WireVector to use as the starting point for the update
- **update_set** – a map from tuples of integers (bit ranges) to the new values
- **truncating** (*bool*) – if true, silently clip new values to the proper bitwidth rather than throw an error if the value provided is too large

Given a WireVector *w*, this function returns a new WireVector that is identical to *w* except in the range of bits specified. When multiple non-overlapping fields need to be updated in a single cycle this provides a clearer way to describe that behavior than iterative calls to *bitfield_update()* (although that is, in fact, what it is doing).

```
w = bitfield_update_set(w, {
        (20, 23):    0x6,      # sets bit 20 to 0, bits 21 and 22 to 1
        (26, None):  0x7,      # assuming w is 32 bits, sets bits 31..26 to 0x7
        (None, 1):   0x0,      # set the LSB (bit) to 0
    })
```

pyrtl.helperfuncs.**match_bitpattern**(*w*, *bitpattern*, *field_map=None*)

Returns a single-bit WireVector that is 1 if and only if *w* matches the *bitpattern*, and a tuple containing the matched fields, if any. Compatible with the *with* statement.

Parameters

- **w** (WireVector) – The WireVector to be compared to the bitpattern
- **bitpattern** (*str*) – A string holding the pattern (of bits and wildcards) to match
- **field_map** – (optional) A map from single-character field name in the bitpattern to the desired name of field in the returned namedtuple. If given, all non-"1"/"0"/"?" characters in the *bitpattern* must be present in the map.

Returns

A tuple of 1-bit WireVector carrying the result of the comparison, followed by a named tuple containing the matched fields, if any.

This function will compare a multi-bit WireVector to a specified pattern of bits, where some of the pattern can be "wildcard" bits. If any of the `1` or `0` values specified in the bitpattern fail to match the WireVector during execution, a `0` will be produced, otherwise the value carried on the wire will be `1`. The wildcard characters can be any other alphanumeric character, with characters other than `?` having special functionality (see below). The string must have length equal to the WireVector specified, although whitespace and underscore characters will be ignored and can be used for pattern readability.

For all other characters besides `1`, `0`, or `?`, a tuple of WireVectors will be returned as the second return value. Each character will be treated as the name of a field, and non-consecutive fields with the same name will be concatenated together, left-to-right, into a single field in the resultant tuple. For example, `01aa1?bbb11a` will match a string such as `010010100111`, and the resultant matched fields are:

```
(a, b) = (0b001, 0b100)
```

where the `a` field is the concenation of bits 9, 8, and 0, and the `b` field is the concenation of bits 5, 4, and 3. Thus, arbitrary characters beside `?` act as wildcard characters for the purposes of matching, with the additional benefit of returning the WireVectors corresponding to those fields.

A prime example of this is for decoding instructions. Here we decode some RISC-V:

```
with pyrtl.conditional_assignment:
    with match_bitpattern(inst, "iiiiiiiiiiiirrrrr010ddddd0000011") as (imm, rs1,␣
↪rd):
        regfile[rd] |= mem[(regfile[rs1] + imm.sign_extended(32)).truncate(32)]
        pc.next |= pc + 1
    with match_bitpattern(inst, "iiiiiiirrrrrssssss010iiiii0100011") as (imm, rs2,␣
↪rs1):
        mem[(regfile[rs1] + imm.sign_extended(32)).truncate(32)] |= regfile[rs2]
        pc.next |= pc + 1
    with match_bitpattern(inst, "0000000rrrrrssssss111ddddd0110011") as (rs2, rs1,␣
↪rd):
        regfile[rd] |= regfile[rs1] & regfile[rs2]
        pc.next |= pc + 1
    with match_bitpattern(inst, "0000000rrrrrssssss000ddddd0110011") as (rs2, rs1,␣
↪rd):
        regfile[rd] |= (regfile[rs1] + regfile[rs2]).truncate(32)
        pc.next |= pc + 1
    # ...etc...
```

Some smaller examples:

```
m, _ = match_bitpattern(w, '0101')  # basically the same as w == '0b0101'
m, _ = match_bitpattern(w, '01?1')  # m will be true when w is '0101' or '0111'
m, _ = match_bitpattern(w, '??01')  # m be true when last two bits of w are '01'
m, _ = match_bitpattern(w, '??_0 1')  # spaces/underscores are ignored, same as␣
↪line above
m, (a, b) = match_pattern(w, '01aa1?bbb11a')  # all bits with same letter make up␣
↪same field
m, fs = match_pattern(w, '01aa1?bbb11a', {'a': 'foo', 'b': 'bar'})  # fields fs.foo,
↪ fs.bar
```

5.5.4 Creating Lists of WireVectors

pyrtl.helperfuncs.**input_list**(*names*, *bitwidth=None*)

Allocate and return a list of *Inputs*.

Parameters

- **names** – Names for the Inputs. Can be a list or single comma/space-separated string
- **bitwidth** (*int*) – The desired bitwidth for the resulting Inputs.

Returns
List of Inputs.

Return type
List[*Input*]

Equivalent to:

```
wirevector_list(names, bitwidth, wvtype=pyrtl.wire.Input)
```

pyrtl.helperfuncs.**output_list**(*names*, *bitwidth=None*)

Allocate and return a list of *Outputs*.

Parameters

- **names** – Names for the Outputs. Can be a list or single comma/space-separated string
- **bitwidth** (*int*) – The desired bitwidth for the resulting Outputs.

Returns
List of Outputs.

Return type
List[*Output*]

Equivalent to:

```
wirevector_list(names, bitwidth, wvtype=pyrtl.wire.Output)
```

pyrtl.helperfuncs.**register_list**(*names*, *bitwidth=None*)

Allocate and return a list of *Registers*.

Parameters

- **names** – Names for the Registers. Can be a list or single comma/space-separated string
- **bitwidth** (*int*) – The desired bitwidth for the resulting Registers.

Returns
List of Registers.

Return type
List[*Register*]

Equivalent to:

```
wirevector_list(names, bitwidth, wvtype=pyrtl.wire.Register)
```

pyrtl.helperfuncs.**wirevector_list**(*names*, *bitwidth=None*, *wvtype=<class 'pyrtl.wire.WireVector'>*)

Allocate and return a list of WireVectors.

Parameters

- **names** – Names for the WireVectors. Can be a list or single comma/space-separated string
- **bitwidth** (*int*) – The desired bitwidth for the resulting WireVectors.
- **wvtype** (WireVector) – Which WireVector type to create.

Returns
List of WireVectors.

Return type
List[*WireVector*]

Additionally, the *names* string can also contain an additional bitwidth specification separated by a `/` in the name. This cannot be used in combination with a *bitwidth* value other than 1.

Examples:

```
wirevector_list(['name1', 'name2', 'name3'])
wirevector_list('name1, name2, name3')
wirevector_list('input1 input2 input3', bitwidth=8, wvtype=pyrtl.wire.Input)
wirevector_list('output1, output2 output3', bitwidth=3, wvtype=pyrtl.wire.Output)
wirevector_list('two_bits/2, four_bits/4, eight_bits/8')
wirevector_list(['name1', 'name2', 'name3'], bitwidth=[2, 4, 8])
```

5.5.5 Interpreting Vectors of Bits

Under the hood, every single *value* a PyRTL design operates on is a bit vector (which is, in turn, simply an integer of bounded power-of-two size. Interpreting these bit vectors as humans, and turning human understandable values into their corresponding bit vectors, can both be a bit of a pain. The functions below do not create any hardware but rather help in the process of reasoning about bit vector representations of human understandable values.

pyrtl.helperfuncs.**val_to_signed_integer**(*value*, *bitwidth*)

Return value as intrepreted as a signed integer under two's complement.

Parameters

- **value** (int) – A Python integer holding the value to convert.
- **bitwidth** (int) – The length of the integer in bits to assume for conversion.

Return type
int

Returns
value as a signed integer

Given an unsigned integer (not a WireVector!) convert that to a signed integer. This is useful for printing and interpreting values which are negative numbers in two's complement.

```
val_to_signed_integer(0xff, 8) == -1
```

val_to_signed_integer can also be used as an repr_func for *SimulationTrace.render_trace()*, to display signed integers in traces:

```
bitwidth = 3
counter = Register(name='counter', bitwidth=bitwidth)
counter.next <<= counter + 1
sim = Simulation()
sim.step_multiple(nsteps=2 ** bitwidth)
```

(continues on next page)

(continued from previous page)

```
# Generates a trace like:
#        |0 |1 |2 |3 |4 |5 |6 |7
#
# counter --1 |2 |3 |-4|-3|-2|-1
sim.tracer.render_trace(repr_func=val_to_signed_integer)
```

pyrtl.helperfuncs.**val_to_formatted_str**(*val*, *format*, *enum_set=None*)

Return a string representation of the value given format specified.

Parameters

- **val** (*int*) – an unsigned integer to convert
- **format** (*str*) – a string holding a format which will be used to convert the data string
- **enum_set** – an iterable of enums which are used as part of the converstion process

Returns
a human-readable string representing *val*.

Return type
str

Given an unsigned integer (not a WireVector!) convert that to a human-readable string. This helps deal with signed/unsigned numbers (simulation operates on values that have been converted via two's complement), but it also generates hex, binary, and enum types as outputs. It is easiest to see how it works with some examples.

```
val_to_formatted_str(2, 's3') == '2'
val_to_formatted_str(7, 's3') == '-1'
val_to_formatted_str(5, 'b3') == '101'
val_to_formatted_str(5, 'u3') == '5'
val_to_formatted_str(5, 's3') == '-3'
val_to_formatted_str(10, 'x3') == 'a'
class Ctl(Enum):
    ADD = 5
    SUB = 12
val_to_formatted_str(5, 'e3/Ctl', [Ctl]) == 'ADD'
val_to_formatted_str(12, 'e3/Ctl', [Ctl]) == 'SUB'
```

pyrtl.helperfuncs.**formatted_str_to_val**(*data*, *format*, *enum_set=None*)

Return an unsigned integer representation of the data given format specified.

Parameters

- **data** (*str*) – a string holding the value to convert
- **format** (*str*) – a string holding a format which will be used to convert the data string
- **enum_set** – an iterable of enums which are used as part of the conversion process

Returns
data as a signed integer

Return type
int

Given a string (not a WireVector!) convert that to an unsigned integer ready for input to the simulation enviornment. This helps deal with signed/unsigned numbers (simulation assumes the values have been converted via

two's complement already), but it also takes hex, binary, and enum types as inputs. It is easiest to see how it works with some examples.

```
formatted_str_to_val('2', 's3') == 2  # 0b010
formatted_str_to_val('-1', 's3') == 7  # 0b111
formatted_str_to_val('101', 'b3') == 5
formatted_str_to_val('5', 'u3') == 5
formatted_str_to_val('-3', 's3') == 5
formatted_str_to_val('a', 'x3') == 10
class Ctl(Enum):
    ADD = 5
    SUB = 12
formatted_str_to_val('ADD', 'e3/Ctl', [Ctl]) == 5
formatted_str_to_val('SUB', 'e3/Ctl', [Ctl]) == 12
```

pyrtl.helperfuncs.**infer_val_and_bitwidth**(*rawinput*, *bitwidth=None*, *signed=False*)

Return a tuple (value, bitwidth) infered from the specified input.

Parameters

- **rawinput** – a bool, int, or verilog-style string constant
- **bitwidth** (*int*) – an integer bitwidth or (by default) None
- **signed** (*bool*) – a bool (by default set False) to include bits for proper two's complement

Returns

tuple of integers (*value*, *bitwidth*)

Return type

(int, int)

Given a boolean, integer, or verilog-style string constant, this function returns a tuple of two integers (*value*, *bitwidth*) which are infered from the specified *rawinput*. The tuple returned is, in fact, a named tuple with names *.value* and *.bitwidth* for fields 0 and 1 respectively. If *signed* is True, bits will be included to ensure a proper two's complement representation is possible, otherwise it is assume all bits can be used for standard unsigned representation. Error checks are performed that determine if the bitwidths specified are sufficient and appropriate for the values specified. Examples can be found below

```
infer_val_and_bitwidth(2, bitwidth=5) == (2, 5)
infer_val_and_bitwidth(3) == (3, 2)  # bitwidth infered from value
infer_val_and_bitwidth(3, signed=True) == (3, 3)  # need a bit for the leading zero
infer_val_and_bitwidth(-3, signed=True) == (5, 3)  # 5 = -3 & 0b111 = ..111101 &␣
↪0b111
infer_val_and_bitwidth(-4, signed=True) == (4, 3)  # 4 = -4 & 0b111 = ..111100 &␣
↪0b111
infer_val_and_bitwidth(-3, bitwidth=5, signed=True) == (29, 5)
infer_val_and_bitwidth(-3) ==> Error  # negative numbers require bitwidth or␣
↪signed=True
infer_val_and_bitwidth(3, bitwidth=2) == (3, 2)
infer_val_and_bitwidth(3, bitwidth=2, signed=True) ==> Error  # need space for sign␣
↪bit
infer_val_and_bitwidth(True) == (1, 1)
infer_val_and_bitwidth(False) == (0, 1)
infer_val_and_bitwidth("5'd12") == (12, 5)
infer_val_and_bitwidth("5'b10") == (2, 5)
infer_val_and_bitwidth("5'b10").bitwidth == 5
```

(continues on next page)

(continued from previous page)

```
infer_val_and_bitwidth("5'b10").value == 2
infer_val_and_bitwidth("8'B 0110_1100") == (108, 8)
```

pyrtl.helperfuncs.**log2**(*integer_val*)

Return the log base 2 of the integer provided.

Parameters
integer_val (*int*) – The integer to take the log base 2 of.

Returns
The log base 2 of *integer_val*, or throw PyRTL error if not power of 2

Return type
int

This function is useful when checking that powers of 2 are provided on inputs to functions. It throws an error if a negative value is provided or if the value provided is not an even power of two.

Examples:

```
log2(2)  # returns 1
log2(256)  # returns 8
addrwidth = log2(size_of_memory)  # will fail if size_of_memory is not a power of␣
↪two
```

5.5.6 Debugging

pyrtl.core.**set_debug_mode**(*debug=True*)

Set the global debug mode.

Parameters
debug (*bool*) – Optional boolean paramter to which debug mode will be set

This function will set the debug mode to the specified value. Debug mode is, by default, set to off to keep the performance of the system. With debug mode set to true, all temporary WireVectors created will be given a name based on the line of code on which they were created and a snapshot of the call-stack for those WireVectors will be kept as well.

pyrtl.helperfuncs.**probe**(*w*, *name=None*)

Print useful information about a WireVector when in debug mode.

Parameters

- **w** (WireVector) – WireVector from which to get info
- **name** (*str*) – optional name for probe (defaults to an autogenerated name)

Returns
original WireVector *w*

Return type
WireVector

Probe can be inserted into a existing design easily as it returns the original wire unmodified. For example `y <<= x[0:3] + 4` could be turned into `y <<= probe(x)[0:3] + 4` to give visibility into both the origin of `x` (including the line that WireVector was originally created) and the run-time values of `x` (which will be named and thus show up by default in a trace). Likewise `y <<= probe(x[0:3]) + 4`, `y <<= probe(x[0:3] + 4)`, and `probe(y) <<= x[0:3] + 4` are all valid uses of *probe*.

Note: *probe* does actually add an Output wire to the working block of *w* (which can confuse various post-processing transforms such as output to verilog).

pyrtl.helperfuncs.**rtl_assert**(*w*, *exp*, *block=None*)

Add hardware assertions to be checked on the RTL design.

Parameters

- **w** (WireVector) – should be a WireVector
- **exp** (*Exception*) – Exception to throw when assertion fails
- **block** (Block) – block to which the assertion should be added (default to working block)

Returns
the Output wire for the assertion (can be ignored in most cases)

Return type
Output

If at any time during execution the wire *w* is not *true* (i.e. asserted low) then simulation will raise *exp*.

pyrtl.helperfuncs.**check_rtl_assertions**(*sim*)

Checks the values in sim to see if any registers assertions fail.

Parameters
sim (Simulation) – Simulation in which to check the assertions

Returns
None

5.5.7 Reductions

pyrtl.corecircuits.**and_all_bits**(*vector*)

Returns WireVector, the result of "and"ing all items of the argument vector.

Parameters
vector (WireVector) – Takes a single arbitrary length WireVector

Returns
Returns a 1 bit result, the bitwise *and* of all of the bits in the vector to a single bit.

pyrtl.corecircuits.**or_all_bits**(*vector*)

Returns WireVector, the result of "or"ing all items of the argument vector.

Parameters
vector (WireVector) – Takes a single arbitrary length WireVector

Returns
Returns a 1 bit result, the bitwise *or* of all of the bits in the vector to a single bit.

pyrtl.corecircuits.**xor_all_bits**(*vector*)

Returns WireVector, the result of "xor"ing all items of the argument vector.

Parameters
vector (WireVector) – Takes a single arbitrary length WireVector

Returns
Returns a 1 bit result, the bitwise *xor* of all of the bits in the vector to a single bit.

pyrtl.corecircuits.**parity**(*vector*)

Returns WireVector, the result of "xor"ing all items of the argument vector.

Parameters
vector (WireVector) – Takes a single arbitrary length WireVector

Returns
Returns a 1 bit result, the bitwise *xor* of all of the bits in the vector to a single bit.

pyrtl.corecircuits.**rtl_any**(**vectorlist*)

Hardware equivalent of Python native `any`.

Parameters
vectorlist (WireVector) – all arguments are WireVectors of length 1

Returns
WireVector of length 1

Returns a 1-bit WireVector which will hold a '1' if any of the inputs are '1' (i.e. it is a big ol' OR gate). If no inputs are provided it will return a Const 0 (since there are no '1's present) similar to Python's `any` function called with an empty list.

Examples:

```
rtl_any(thing1, thing2, thing3)  # same as thing1 | thing2 | thing3
rtl_any(*[list_of_things])  # the unpack operator ("*") can be used for lists
rtl_any()  # returns Const(False) which comes up if the list above is empty
```

pyrtl.corecircuits.**rtl_all**(**vectorlist*)

Hardware equivalent of Python native `all`.

Parameters
vectorlist (WireVector) – all arguments are WireVectors of length 1

Returns
WireVector of length 1

Returns a 1-bit WireVector which will hold a '1' only if all of the inputs are '1' (i.e. it is a big ol' AND gate). If no inputs are provided it will return a Const 1 (since there are no '0's present) similar to Python's `all` function called with an empty list.

Examples:

```
rtl_all(thing1, thing2, thing3)  # same as thing1 & thing2 & thing3
rtl_all(*[list_of_things])  # the unpack operator ("*") can be used for lists
rtl_all()  # returns Const(True) which comes up if the list above is empty
```

5.5.8 Extended Logic and Arithmetic

The functions below provide ways of comparing and arithmetically combining *WireVectors* in ways that are often useful in hardware design. The functions below extend those member functions of the *WireVector* class itself (which provides support for addition, unsigned multiplication, unsigned comparison, and many others).

pyrtl.corecircuits.**signed_add**(*a*, *b*)

Return a WireVector for result of signed addition.

Parameters

- **a** (WireVector) – a WireVector to serve as first input to addition

- **b** (WireVector) – a WireVector to serve as second input to addition

Given WireVectors with length `n` and `m`, the result of the signed addition has length:

```
max(n, m) + 1
```

The inputs are two's complement sign extended to the same length before adding. If an integer is passed to either *a* or *b*, it will be converted automatically to a two's complement constant

pyrtl.corecircuits.**signed_mult**(*a*, *b*)

Return `a * b` where `a` and `b` are treated as signed values.

Parameters

- **a** (WireVector) – a wirevector to serve as first input to multiplication
- **b** (WireVector) – a wirevector to serve as second input to multiplication

If an integer is passed to either `a` or `b`, it will be converted automatically to a two's complement constant

pyrtl.corecircuits.**signed_lt**(*a*, *b*)

Return a single bit result of signed less than comparison.

pyrtl.corecircuits.**signed_le**(*a*, *b*)

Return a single bit result of signed less than or equal comparison.

pyrtl.corecircuits.**signed_gt**(*a*, *b*)

Return a single bit result of signed greater than comparison.

pyrtl.corecircuits.**signed_ge**(*a*, *b*)

Return a single bit result of signed greater than or equal comparison.

pyrtl.corecircuits.**shift_left_arithmetic**(*bits_to_shift*, *shift_amount*)

Shift left arithmetic operation.

Parameters

- **bits_to_shift** (WireVector) – WireVector to shift left
- **shift_amount** – WireVector or integer specifying amount to shift

Returns

WireVector of same length as *bits_to_shift*

This function returns a new WireVector of length equal to the length of the input *bits_to_shift* but where the bits have been shifted to the left. An arithemetic shift is one that treats the value as as signed number, although for left shift arithmetic and logic shift they are identical. Note that *shift_amount* is treated as unsigned.

pyrtl.corecircuits.**shift_right_arithmetic**(*bits_to_shift*, *shift_amount*)

Shift right arithmetic operation.

Parameters

- **bits_to_shift** (WireVector) – WireVector to shift right
- **shift_amount** – WireVector or integer specifying amount to shift

Returns

WireVector of same length as *bits_to_shift*

This function returns a new WireVector of length equal to the length of the input *bits_to_shift* but where the bits have been shifted to the right. An arithemetic shift is one that treats the value as as signed number, meaning the sign bit (the most significant bit of *bits_to_shift*) is shifted in. Note that *shift_amount* is treated as unsigned.

pyrtl.corecircuits.**shift_left_logical**(*bits_to_shift*, *shift_amount*)

Shift left logical operation.

Parameters

- **bits_to_shift** (WireVector) – WireVector to shift left
- **shift_amount** – WireVector or integer specifying amount to shift

Returns
WireVector of same length as *bits_to_shift*

This function returns a new WireVector of length equal to the length of the input *bits_to_shift* but where the bits have been shifted to the left. A logical shift is one that treats the value as as unsigned number, meaning the zeroes are shifted in. Note that *shift_amount* is treated as unsigned.

pyrtl.corecircuits.**shift_right_logical**(*bits_to_shift*, *shift_amount*)

Shift right logical operation.

Parameters

- **bits_to_shift** (WireVector) – WireVector to shift left
- **shift_amount** – WireVector or integer specifying amount to shift

Returns
WireVector of same length as *bits_to_shift*

This function returns a new WireVector of length equal to the length of the input *bits_to_shift* but where the bits have been shifted to the right. A logical shift is one that treats the value as as unsigned number, meaning the zeros are shifted in regardless of the "sign bit". Note that *shift_amount* is treated as unsigned.

5.6 Analysis and Optimization

Tools for analyzing and optimizing aspects of PyRTL designs.

5.6.1 Estimation

Contains functions to estimate aspects of blocks (like area and delay) by either using internal models or by making calls out to external tool chains.

class pyrtl.analysis.**PathsResult**

print(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Pretty print the result of calling *paths()*

Parameters
f – the open file to print to (defaults to stdout)

Returns
None

class pyrtl.analysis.**TimingAnalysis**(*block=None*, *gate_delay_funcs=None*)

Timing analysis estimates the timing delays in the block

TimingAnalysis has an `timing_map` object that maps wires to the 'time' after a clock edge at which the signal in the wire settles

__init__(*block=None*, *gate_delay_funcs=None*)

Calculates timing delays in the block.

Parameters

- **block** (Block) – PyRTL block to analyze
- **gate_delay_funcs** – a map with keys corresponding to the gate op and a function returning the delay as the value. It takes the gate as an argument. If the delay is negative (-1), the gate will be treated as the end of the block

Calculates the timing analysis while allowing for different timing delays of different gates of each type. Supports all valid presynthesis blocks. Currently doesn't support memory post synthesis.

critical_path(*print_cp=True*, *cp_limit=100*)

Takes a timing map and returns the critical paths of the system.

Parameters
print_cp (*bool*) – Whether to print the critical path to the terminal after calculation

Returns
a list containing tuples with the 'first' wire as the first value and the critical paths (which themselves are lists of nets) as the second

max_freq(*tech_in_nm=130*, *ffoverhead=None*)

Estimates the max frequency of a block in MHz.

Parameters

- **tech_in_nm** (*float*) – the size of the circuit technology to be estimated (for example, 65 is 65nm and 250 is 0.25um)
- **ffoverhead** (*float*) – setup and ff propagation delay in picoseconds

Returns
a number representing an estimate of the max frequency in Mhz

All params are optional and have reasonable default values. Estimation is based on Dennard Scaling assumption and does not include wiring effect – as a result the estimates may be optimistic (especially below 65nm).

max_length()

Returns the max timing delay of the circuit in ps.

The result assumes that the circuit is implemented in a 130nm process, and that there is no setup or hold time associated with the circuit. The resulting value is in picoseconds. If an proper estimation of timing is required it is recommended to us *TimingAnalysis.max_freq()* to determine the clock period as it more accurately considers scaling and setup/hold.

static print_critical_paths(*critical_paths*)

Prints the results of the critical path length analysis. Done by default by the *TimingAnalysis.critical_path()* function.

print_max_length()

Prints the max timing delay of the circuit

pyrtl.analysis.**area_estimation**(*tech_in_nm=130*, *block=None*)

Estimates the total area of the block.

Parameters
tech_in_nm (*float*) – the size of the circuit technology to be estimated (for example, 65 is 65nm and 250 is 0.25um)

Returns
tuple of estimated areas (logic, mem) in terms of mm^2

The estimations are based off of 130nm standard cell designs for the logic, and custom memory blocks from the literature. The results are not fully validated and we do not recommend that this function be used in carrying out science for publication.

pyrtl.analysis.**distance**(*src*, *dst*, *f*, *block=None*)

Calculate the 'distance' along each path from *src* to *dst* according to *f*

Parameters

- **src** (WireVector) – wire to start from
- **dst** (WireVector) – wire to end on
- **f** (*Callable[[LogicNet], int]*) – function from a net to number, representing the 'value' of a net that you want to sum across all nets in the path
- **block** (Block) – block to use (defaults to working block)

Returns
a map from each path (a tuple) to its calculated distance

This calls the given function *f* on each net in a path, summing the result.

pyrtl.analysis.**fanout**(*w*)

Get the number of places a wire is used as an argument.

Parameters
w (WireVector) – WireVector to check fanout for

Returns
integer fanout count

pyrtl.analysis.**paths**(*src=None*, *dst=None*, *dst_nets=None*, *block=None*)

Get the list of all paths from *src* to *dst*.

Parameters

- **src** (*Union[WireVector, Iterable[WireVector]]*) – source wire(s) from which to trace your paths; if None, will get paths from all Inputs
- **dst** (*Union[WireVector, Iterable[WireVector]]*) – destination wire(s) to which to trace your paths; if None, will get paths to all Outputs
- **dst_nets** (*dict[WireVector, LogicNet]*) – map from wire to set of nets where the wire is an argument; will compute it internally if not given via a call to pyrtl.net_connections()
- **block** (Block) – block to use (defaults to working block)

Returns
a map of the form *{src_wire: {dst_wire: [path]}}* for each *src_wire* in *src* (or all inputs if *src* is None), *dst_wire* in *dst* (or all outputs if *dst* is None), where *path* is a list of nets. This map is also an instance of *PathsResult*, so you can call *PathsResult.print()* on it to pretty print it.

You can provide *dst_nets* (the result of calling *net_connections()*, if you plan on calling this function repeatedly on a block that hasn't changed, to speed things up.

This function can accept one or more *src* wires, and one or more *dst* wires, such that it returns a map that can be accessed like so:

```
paths[src][dst] = [<path>, <path>, ...]
```

where *path* is a list of nets. Thus there can be multiple paths from a given *src* wire to a given *dst* wire.

If *src* and *dst* are both single wires, you still need to access the result via *paths[src][dst]*.

This also finds and returns the loop paths in the case of registers or memories that feed into themselves, i.e. *paths[src][src]* is not necessarily empty.

It does not distinguish between loops that include synchronous vs asynchronous memories.

`pyrtl.analysis.`**`yosys_area_delay`**(*library*, *abc_cmd=None*, *leave_in_dir=None*, *block=None*)

Synthesize with Yosys and return estimate of area and delay.

Parameters

- **`library`** – stdcell library file to target in liberty format
- **`abc_cmd`** – string of commands for **`yosys`** to pass to **`abc`** for synthesis
- **`dir`** – the directory where temporary files should be left
- **`block`** – PyRTL block to analyze

Returns

a tuple of numbers: area, delay

If *dir* is specified, that directory will be used to create any temporary files, and the resulting files will be left behind there (which can be useful for manual exploration or debugging)

The area and delay are returned in units as defined by the stdcell library. In the standard vsc 130nm library, the area is in a number of "tracks", each of which is about 1.74 square um (see area estimation for more details) and the delay is in ps.

http://www.vlsitechnology.org/html/vsc_description.html

May raise *PyrtlError* if **`yosys`** is not configured correctly, and *PyrtlInternalError* if the call to **`yosys`** was not successful

5.6.2 Optimization

`pyrtl.passes.`**`optimize`**(*update_working_block=True*, *block=None*, *skip_sanity_check=False*)

Return an optimized version of a synthesized hardware block.

Parameters

- **`update_working_block`** (*`bool`*) – Don't copy the block and optimize the new block (defaults to True)
- **`block`** (`Block`) – the block to optimize (defaults to working block)
- **`skip_sanity_check`** (*`bool`*) – Don't perform sanity checks on the block before/during/after the optimization passes (defaults to False). Sanity checks will always be performed if in debug mode.

Note: optimize works on all hardware designs, both synthesized and non synthesized

5.6.3 Synthesis

pyrtl.passes.**synthesize**(*update_working_block=True*, *merge_io_vectors=True*, *block=None*)

Lower the design to just single-bit "and", "or", "xor", and "not" gates.

Parameters

- **update_working_block** (*bool*) – Boolean specifying if working block should be set to the newly synthesized block.
- **merge_io_wirevectors** (*bool*) – if False, turn all N-bit IO wirevectors into N 1-bit IO wirevectors (i.e. don't maintain interface).
- **block** (Block) – The block you want to synthesize.

Returns

The newly synthesized block (of type *PostSynthBlock*).

Takes as input a block (default to working block) and creates a new block which is identical in function but uses only single bit gates and excludes many of the more complicated primitives. The new block should consist *almost* exclusively of the combination elements of `w`, `&`, `\|`, `^`, and `~` and sequential elements of registers (which are one bit as well). The two exceptions are for inputs/outputs (so that we can keep the same interface) which are immediately broken down into the individual bits and memories (read and write ports) which require the reassembly and disassembly of the wirevectors immediately before and after. These are the only two places where `c` and `s` ops should exist. If *merge_io_vectors* is False, then these individual bits are not reassembled and disassembled before and after, and so no `c` and `s` ops will exist. Instead, they will be named *<name>[n]*, where *n* is the bit number of original wire to which it corresponds.

The block that results from synthesis is actually of type *PostSynthBlock* which contains a mapping from the original inputs and outputs to the inputs and outputs of this block. This is used during simulation to map the input/outputs so that the same testbench can be used both pre and post synthesis (see documentation for *Simulation* for more details).

class pyrtl.core.**PostSynthBlock**

Bases: *Block*

This is a block with extra metadata required to maintain the pre-synthesis interface during post-synthesis.

It currently holds the following instance attributes:

***io_map*:**
a map from old IO WireVector to a list of new IO WireVectors it maps to; this is a list because for unmerged IO vectors, each old N-bit IO WireVector maps to N new 1-bit IO WireVectors.

***reg_map*:**
a map from old register to a list of new registers; a list because post-synthesis, each N-bit register has been mapped to N 1-bit registers

***mem_map*:**
a map from old memory block to the new memory block

5.6.4 Individual Passes

pyrtl.passes.**common_subexp_elimination**(*block=None*, *abs_thresh=1*, *percent_thresh=0*)

Common Subexpression Elimination for PyRTL blocks.

Parameters

- **block** (Block) – the block to run the subexpression elimination on
- **abs_thresh** (*float*) – absolute threshold for stopping optimization
- **percent_thresh** (*float*) – percent threshold for stopping optimization

pyrtl.passes.**constant_propagation**(*block*, *silence_unexpected_net_warnings=False*)

Removes excess constants in the block.

Note on resulting block: The output of the block can have WireVectors that are driven but not listened to. This is to be expected. These are to be removed by `_remove_unlistened_nets()`

pyrtl.passes.**nand_synth**(*net*)

Synthesizes a *PostSynthBlock* into one consisting of nands and inverters in place

Parameters
block (PostSynthBlock) – The block to synthesize.

pyrtl.passes.**and_inverter_synth**(*net*)

Transforms a decomposed block into one consisting of ands and inverters in place

Parameters
block (Block) – The block to synthesize

pyrtl.passes.**one_bit_selects**(*net*)

Converts arbitrary-sliced *selects* to concatenations of 1-bit *selects*.

Parameters
block (Block) – The block to transform

This is useful for preparing the netlist for output to other formats, like FIRRTL or BTOR2, whose *select* operation (*bits* and *slice*, respectively) require contiguous ranges. Python slices are not necessarily contiguous ranges, e.g. the range *[::2]* (syntactic sugar for *slice(None, None, 2)*) produces indices 0, 2, 4, etc. up to the length of the list on which it is used.

pyrtl.passes.**two_way_concat**(*net*)

Transforms a block so all n-way (n > 2) *concats* are replaced with series of 2-way *concats*.

Parameters
block (Block) – The block to transform

This is useful for preparing the netlist for output to other formats, like FIRRTL or BTOR2, whose *concatenate* operation (*cat* and *concat*, respectively), only allow two arguments (most-significant wire and least-significant wire).

5.7 Exporting and Importing Designs

5.7.1 Exporting Hardware Designs

pyrtl.importexport.**output_to_verilog**(*dest_file*, *add_reset=True*, *block=None*)

A function to walk the block and output it in Verilog format to the open file.

Parameters

- **dest_file** – Open file where the Verilog output will be written
- **add_reset** (*Union[bool, str]*) – If reset logic should be added. Allowable options are: False (meaning no reset logic is added), True (default, for adding synchronous reset logic), and *asynchronous* (for adding asynchronous reset logic).
- **block** – Block to be walked and exported

The registers will be set to their *reset_value*, if specified, otherwise 0.

pyrtl.importexport.**output_to_firrtl**(*open_file*, *rom_blocks=None*, *block=None*)

Output the block as FIRRTL code to the output file.

Parameters

- **open_file** – File to write to
- **rom_blocks** – List of ROM blocks to be initialized
- **block** – Block to use (defaults to working block)

If ROM is initialized in PyRTL code, you can pass in the *rom_blocks* as a list *[rom1, rom2, ...]*.

5.7.2 Exporting Testbenches

pyrtl.importexport.**output_verilog_testbench**(*dest_file*, *simulation_trace=None*, *toplevel_include=None*, *vcd='waveform.vcd'*, *cmd=None*, *add_reset=True*, *block=None*)

Output a Verilog testbench for the block/inputs used in the simulation trace.

Parameters

- **dest_file** – an open file to which the test bench will be printed.
- **simulation_trace** (SimulationTrace) – a simulation trace from which the inputs will be extracted for inclusion in the test bench. The test bench generated will just replay the inputs played to the simulation cycle by cycle. The default values for all registers and memories will be based on the trace, otherwise they will be initialized to 0.
- **toplevel_include** (*str*) – name of the file containing the toplevel module this testbench is testing. If not None, an *include* directive will be added to the top.
- **vcd** (*str*) – By default the testbench generator will include a command in the testbench to write the output of the testbench execution to a .vcd file (via *$dumpfile*), and this parameter is the string of the name of the file to use. If None is specified instead, then no *dumpfile* will be used.
- **cmd** (*str*) – The string passed as cmd will be copied verbatim into the testbench just before the end of each cycle. This is useful for doing things like printing specific values out during testbench evaluation (e.g. `cmd='$display("%d", out);'` will instruct the testbench to print the value of *out* every cycle which can then be compared easy with a reference).

- **add_reset** (*Union[bool, str]*) – If reset logic should be added. Allowable options are: False (meaning no reset logic is added), True (default, for adding synchronous reset logic), and *asynchronous* (for adding asynchronous reset logic). The value passed in here should match the argument passed to *output_to_verilog()*.
- **block** (Block) – Block containing design to test.

If *add_reset* is not False, a *rst* wire is added and will passed as an input to the instantiated toplevel module. The *rst* wire will be held low in the testbench, because initialization here occurs via the *initial* block. It is provided for consistency with *output_to_verilog()*.

The test bench does not return any values.

Example 1 (writing testbench to a string):

```
with io.StringIO() as tbfile:
    pyrtl.output_verilog_testbench(dest_file=tbfile, simulation_trace=sim_trace)
```

Example 2 (testbench in same file as verilog):

```
with open('hardware.v', 'w') as fp:
    output_to_verilog(fp)
    output_verilog_testbench(fp, sim.tracer, vcd=None, cmd='$display("%d", out);')
```

5.7.3 Importing Verilog

pyrtl.importexport.**input_from_blif**(*blif*, *block=None*, *merge_io_vectors=True*, *clock_name='clk'*, *top_model=None*)

Read an open BLIF file or string as input, updating the block appropriately.

Parameters

- **blif** – An open BLIF file to read
- **block** (Block) – The block where the logic will be added
- **merge_io_vectors** (*bool*) – If True, Input/Output wires whose names differ only by a indexing subscript (e.g. 1-bit wires `a[0]` and `a[1]`) will be combined into a single Input/Output (e.g. a 2-bit wire `a`).
- **clock_name** (*str*) – The name of the clock (defaults to `clk`)
- **top_model** – name of top-level model to instantiate; if None, defaults to first model listed in the BLIF

If *merge_io_vectors* is True, then given 1-bit Input wires `a[0]` and `a[1]`, these wires will be combined into a single 2-bit Input wire `a` that can be accessed by name `a` in the block. Otherwise if *merge_io_vectors* is False, the original 1-bit wires will be Input wires of the block. This holds similarly for Outputs.

This assumes the following:

- There is only one single shared clock and reset
- Output is generated by Yosys with formals in a particular order

It currently supports multi-module (unflattened) BLIF, though we recommend importing a flattened BLIF with a single module when possible. It currently ignores the reset signal (which it assumes is input only to the flip flops).

5.7.4 Outputting for Visualization

pyrtl.visualization.**output_to_trivialgraph**(*file*, *namer=<function _trivialgraph_default_namer>*, *block=None*, *split_state=False*)

Walk the block and output it in trivial graph format to the open file.

Parameters

- **file** – Open file to write to
- **namer** – A function that takes in an object (a wire or LogicNet) as the first argument and a boolean *is_edge* as the second that is set True if the object is a wire, and returns a string representing that object.
- **block** (Block) – Block to use (defaults to current working block)
- **split_state** (*bool*) – if True, split connections to/from a register update net; this means that registers will be appear as source nodes of the network, and `r` nets (i.e. the logic for setting a register's next value) will be treated as sink nodes of the network.

pyrtl.visualization.**output_to_graphviz**(*file*, *block=None*, *namer=<function _graphviz_default_namer>*, *split_state=True*, *maintain_arg_order=False*)

Walk the block and output it in Graphviz format to the open file.

Parameters

- **file** – Open file to write to
- **block** (Block) – Block to use (defaults to current working block)
- **namer** – Function used to label each edge and node; see *block_to_graphviz_string()* for more information.
- **split_state** (*bool*) – If True, visually split the connections to/from a register update net.
- **maintain_arg_order** (*bool*) – If True, will add ordering constraints so that that incoming edges are ordered left-to-right for nets where argument order matters (e.g. <). Keeping this as False results in a cleaner, though less visually precise, graphical output.

The file written by the this function should be a directed graph in the format expected by the Graphviz package, specifically in the **dot** format. Once Graphviz is installed, the resulting graph file can be rendered to a .png file with:

```
dot -Tps output.dot > output.ps
```

pyrtl.visualization.**graphviz_detailed_namer**(*extra_node_info=None*, *extra_edge_info=None*)

Returns a detailed Graphviz namer that prints extra information about nodes/edges in the given maps.

Parameters

- **extra_node_info** – A dict from node to some object about that node (its string representation will be printed next to the node's label)
- **extra_edge_info** – A dict from edge to some object about that edge (its string representation will be printed next to the edge's label)

Returns

A function that knows how to label each element in the graph, which can be passed to *output_to_graphviz()* or *block_to_graphviz_string()*

If both dict arguments are None, the returned namer behaves identically to the default Graphviz namer.

pyrtl.visualization.**output_to_svg**(*file*, *block=None*, *split_state=True*)

Output the block as an SVG to the open file.

Parameters

- **file** – Open file to write to
- **block** (Block) – Block to use (defaults to current working block)
- **split_state** (*bool*) – If True, visually split the connections to/from a register update net.

pyrtl.visualization.**block_to_graphviz_string**(*block=None*, *namer=<function _graphviz_default_namer>*, *split_state=True*, *maintain_arg_order=False*)

Return a Graphviz string for the block.

Parameters

- **namer** – A function mapping graph objects (wires/logic nets) to labels. If you want a more detailed namer, pass in a call to *graphviz_detailed_namer()* (see below).
- **block** (Block) – Block to use (defaults to current working block)
- **split_state** (*bool*) – If True, split connections to/from a register update net; this means that registers will be appear as source nodes of the network, and `r` nets (i.e. the logic for setting a register's next value) will be treated as sink nodes of the network.
- **maintain_arg_order** (*bool*) – If True, will add ordering constraints so that that incoming edges are ordered left-to-right for nets where argument order matters (e.g. <). Keeping this as False results in a cleaner, though less visually precise, graphical output.

The normal namer function will label user-named wires with their names and label the nodes (logic nets or Input/Output/Const terminals) with their operator symbol or name/value, respectively. If custom information about each node in the graph is desired, you can pass in a custom namer function which must have the same signature as the default namer, `_graphviz_default_namer()`. However, we recommend you instead pass in a call to *graphviz_detailed_namer()*, supplying it with your own dicts mapping wires and nodes to labels. For any wire/node found in these maps, that additional information will be printed in parentheses alongside the node in the graphviz graph.

For example, if you wanted to print the delay of each wire and the fanout of each gate, you could pass in two maps to the *graphviz_detailed_namer()* call, which returns a namer function that can subsequently be passed to *output_to_graphviz()* or *block_to_graphviz_string()*.

```
node_fanout = {n: "Fanout: %d" % my_fanout_func(n) for n in working_block().logic}
wire_delay = {w: "Delay: %.2f" % my_delay_func(w) for w in working_block().
↪wirevector_set}

with open("out.gv", "w") as f:
    output_to_graphviz(f, namer=graphviz_detailed_namer(node_fanout, wire_delay))
```

pyrtl.visualization.**block_to_svg**(*block=None*, *split_state=True*, *maintain_arg_order=False*)

Return an SVG for the block.

Parameters

- **block** (Block) – Block to use (defaults to current working block)
- **split_state** (*bool*) – If True, visually split the connections to/from a register update net.
- **maintain_arg_order** (*bool*) – If True, will add ordering constraints so that that incoming edges are ordered left-to-right for nets where argument order matters (e.g. <). Keeping this as False results in a cleaner, though less visually precise, graphical output.

Returns
The SVG representation of the block

pyrtl.visualization.**trace_to_html**(*simtrace*, *trace_list=None*, *sortkey=None*, *repr_func=<built-in function hex>*, *repr_per_name={}*)

Return a HTML block showing the trace.

Parameters

- **simtrace** (SimulationTrace) – A SimulationTrace object
- **trace_list** (*list[str]*) – (optional) A list of wires to display
- **sortkey** – (optional) The key with which to sort the trace_list
- **repr_func** – function to use for representing the current_val; examples are `hex`, `oct`, `bin`, `str` (for decimal), or even the name of an IntEnum class you know the value will belong to. Defaults to `hex`.
- **repr_per_name** – Map from signal name to a function that takes in the signal's value and returns a user-defined representation. If a signal name is not found in the map, the argument *repr_func* will be used instead.

Returns
An HTML block showing the trace

pyrtl.visualization.**net_graph**(*block=None*, *split_state=False*)

Return a graph representation of the given block.

Parameters

- **block** (Block) – block to use (defaults to current working block)
- **split_state** (*bool*) – if True, split connections to/from a register update net; this means that registers will be appear as source nodes of the network, and `r` nets (i.e. the logic for setting a register's next value) will be treated as sink nodes of the network.

The graph has the following form:

```
{
  node1: { nodeA: [edge1A_1, edge1A_2], nodeB: [edge1B]},
  node2: { nodeB: [edge2B],             nodeC: [edge2C_1, edge2C_2]},
  ...
}
```

aka: `edges = graph[source][dest]`

Each node can be either a LogicNet or a WireVector (e.g. an Input, an Output, a Const or even an undriven WireVector (which acts as a source or sink in the network). Each edge is a WireVector or derived type (Input, Output, Register, etc.). Note that inputs, consts, and outputs will be both "node" and "edge". WireVectors that are not connected to any nets are not returned as part of the graph.

5.8 RTL Library

Useful circuits, functions, and testing utilities.

5.8.1 Adders

pyrtl.rtllib.adders.**carrysave_adder**(*a*, *b*, *c*, *final_adder=<function ripple_add>*)

Adds three wirevectors up in an efficient manner

Parameters

- **a** (WireVector) – a wire to add up
- **b** (WireVector) – a wire to add up
- **c** (WireVector) – a wire to add up
- **final_adder** (*Callable*) – The adder to use to do the final addition

Returns
a WireVector with length 2 longer than the largest input

pyrtl.rtllib.adders.**cla_adder**(*a*, *b*, *cin=0*, *la_unit_len=4*)

Carry Lookahead Adder

Parameters
la_unit_len (*int*) – the length of input that every unit processes

A Carry LookAhead Adder is an adder that is faster than a ripple carry adder, as it calculates the carry bits faster. It is not as fast as a Kogge-Stone adder, but uses less area.

pyrtl.rtllib.adders.**dada_reducer**(*wire_array_2*, *result_bitwidth*, *final_adder=<function kogge_stone>*)

The reduction and final adding part of a dada tree. Useful for adding many numbers together The use of single bitwidth wires is to allow for additional flexibility

Parameters

- **wire_array_2** (*[[Wirevector]]*) – An array of arrays of single bitwidth wirevectors
- **result_bitwidth** (*int*) – The bitwidth you want for the resulting wire. Used to eliminate unnecessary wires.
- **final_adder** – The adder used for the final addition

Returns
WireVector of length *result_bitwidth*

pyrtl.rtllib.adders.**fast_group_adder**(*wires_to_add*, *reducer=<function wallace_reducer>*, *final_adder=<function kogge_stone>*)

A generalization of the carry save adder, this is designed to add many numbers together in a both area and time efficient manner. Uses a tree reducer to achieve this performance

Parameters

- **wires_to_add** (*[WireVector]*) – an array of WireVectors to add
- **reducer** – the tree reducer to use
- **final_adder** – The two value adder to use at the end

Returns
a wirevector with the result of the addition

The length of the result is:

```
max(len(w) for w in wires_to_add) + ceil(len(wires_to_add))
```

pyrtl.rtllib.adders.**half_adder**(*a*, *b*)

pyrtl.rtllib.adders.**kogge_stone**(*a*, *b*, *cin=0*)

Creates a Kogge-Stone adder given two inputs

Parameters

- **a** (WireVector) – A WireVector to add up (bitwidths don't need to match)
- **b** (WireVector) – A WireVector to add up (bitwidths don't need to match)
- **cin** – An optimal carry in WireVector or value

Returns

a WireVector representing the output of the adder

The Kogge-Stone adder is a fast tree-based adder with *O(log(n))* propagation delay, useful for performance critical designs. However, it has *O(n log(n))* area usage, and large fan out.

pyrtl.rtllib.adders.**one_bit_add**(*a*, *b*, *cin=0*)

pyrtl.rtllib.adders.**ripple_add**(*a*, *b*, *cin=0*)

pyrtl.rtllib.adders.**ripple_half_add**(*a*, *cin=0*)

pyrtl.rtllib.adders.**wallace_reducer**(*wire_array_2*, *result_bitwidth*, *final_adder=<function kogge_stone>*)

The reduction and final adding part of a dada tree. Useful for adding many numbers together The use of single bitwidth wires is to allow for additional flexibility

Parameters

- **wire_array_2** (*[[Wirevector]]*) – An array of arrays of single bitwidth wirevectors
- **result_bitwidth** (*int*) – The bitwidth you want for the resulting wire. Used to eliminate unnecessary wires.
- **final_adder** – The adder used for the final addition

Returns

WireVector of length *result_bitwidth*

5.8.2 AES-128

A class for building a PyRTL AES circuit.

Currently this class only supports 128 bit AES encryption/decryption

Example:

```
import pyrtl
from pyrtl.rtllib.aes import AES

aes = AES()
plaintext = pyrtl.Input(bitwidth=128, name='aes_plaintext')
key = pyrtl.Input(bitwidth=128, name='aes_key')
aes_ciphertext = pyrtl.Output(bitwidth=128, name='aes_ciphertext')
```

(continues on next page)

(continued from previous page)

```
reset = pyrtl.Input(1, name='reset')
ready = pyrtl.Output(1, name='ready')
ready_out, aes_cipher = aes.encrypt_state_m(plaintext, key, reset)
ready <<= ready_out
aes_ciphertext <<= aes_cipher
sim_trace = pyrtl.SimulationTrace()
sim = pyrtl.Simulation(tracer=sim_trace)
sim.step ({
    'aes_plaintext': 0x00112233445566778899aabbccddeeff,
    'aes_key': 0x000102030405060708090a0b0c0d0e0f,
    'reset': 1
})
for cycle in range(1,10):
    sim.step ({
        'aes_plaintext': 0x00112233445566778899aabbccddeeff,
        'aes_key': 0x000102030405060708090a0b0c0d0e0f,
        'reset': 0
    })
sim_trace.render_trace(symbol_len=40, segment_size=1)
```

class pyrtl.rtllib.aes.**AES**

decryption(*ciphertext*, *key*)

Builds a single cycle AES Decryption circuit

Parameters

- **ciphertext** (WireVector) – data to decrypt
- **key** (WireVector) – AES key to use to encrypt (AES is symmetric)

Returns

a WireVector containing the plaintext

decryption_statem(*ciphertext_in*, *key_in*, *reset*)

Builds a multiple cycle AES Decryption state machine circuit

Parameters

reset – a one bit signal telling the state machine to reset and accept the current plaintext and key

Return ready, plain_text

ready is a one bit signal showing that the decryption result (*plain_text*) has been calculated.

encrypt_state_m(*plaintext_in*, *key_in*, *reset*)

Builds a multiple cycle AES Encryption state machine circuit

Parameters

reset – a one bit signal telling the state machine to reset and accept the current plaintext and key

Return ready, cipher_text

ready is a one bit signal showing that the encryption result (*cipher_text*) has been calculated.

encryption(*plaintext*, *key*)

Builds a single cycle AES Encryption circuit

Parameters

- **plaintext** (WireVector) – text to encrypt
- **key** (WireVector) – AES key to use to encrypt

Returns
a WireVector containing the ciphertext

5.8.3 Barrel

pyrtl.rtllib.barrel.**barrel_shifter**(*bits_to_shift*, *bit_in*, *direction*, *shift_dist*, *wrap_around=0*)

Create a barrel shifter that operates on data based on the wire width.

Parameters

- **bits_to_shift** – the input wire
- **bit_in** – the 1-bit wire giving the value to shift in
- **direction** – a one bit WireVector representing shift direction (0 = shift down, 1 = shift up)
- **shift_dist** – WireVector representing offset to shift
- **wrap_around** – ****currently not implemented****

Returns
shifted WireVector

5.8.4 Library Utilities

pyrtl.rtllib.libutils.**match_bitwidth**(**args*)

Matches the bitwidth of all of the input arguments.

Parameters
args (WireVector) – input arguments

Returns
tuple of *args* in order with extended bits

pyrtl.rtllib.libutils.**partition_wire**(*wire*, *partition_size*)

Partitions a wire into a list of N wires of size *partition_size*.

Parameters

- **wire** – Wire to partition
- **partition_size** – Integer representing size of each partition

The *wire*'s bitwidth must be evenly divisible by *parition_size*.

pyrtl.rtllib.libutils.**rev_twos_comp_repr**(*val*, *bitwidth*)

Takes a two's-complement represented value and converts it to a signed integer based on the provided *bitwidth*. For use with *Simulation.inspect()* etc. when expecting negative numbers, which it does not recognize

pyrtl.rtllib.libutils.**str_to_int_array**(*string*, *base=16*)

Converts a string to an array of integer values according to the base specified (int numbers must be whitespace delimited).

Example: `"13 a3 3c" => [0x13, 0xa3, 0x3c]`

Returns
[int]

pyrtl.rtllib.libutils.**twos_comp_repr**(*val*, *bitwidth*)

Converts a value to its two's-complement (positive) integer representation using a given bitwidth (only converts the value if it is negative).

Parameters

- **val** – Integer literal to convert to two's complement
- **bitwidth** – Size of val in bits

For use with *Simulation.step()* etc. in passing negative numbers, which it does not accept.

5.8.5 Multipliers

Multipliers contains various PyRTL sample multipliers for people to use

pyrtl.rtllib.multipliers.**complex_mult**(*A*, *B*, *shifts*, *start*)

Generate shift-and-add multiplier that can shift and add multiple bits per clock cycle. Uses substantially more space than *simple_mult()* but is much faster.

Parameters

- **A** (WireVector) – input wire for the multiplication
- **B** (WireVector) – input wire for the multiplication
- **shifts** (*int*) – number of spaces Register is to be shifted per clock cycle (cannot be greater than the length of *A* or *B*)
- **start** (*bool*) – start signal

Return [Register, bool]
Register containing the product and the *done* signal

pyrtl.rtllib.multipliers.**fused_multiply_adder**(*mult_A*, *mult_B*, *add*, *signed=False*, *reducer=<function wallace_reducer>*, *adder_func=<function kogge_stone>*)

Generate efficient hardware for `a * b + c`.

Multiplies two WireVectors together and adds a third WireVector to the multiplication result, all in one step. By doing it this way (instead of separately), one reduces both the area and the timing delay of the circuit.

Parameters

- **signed** (*Bool*) – Currently not supported (will be added in the future) The default will likely be changed to True, so if you want the smallest set of wires in the future, specify this as False
- **reducer** – (advanced) The tree reducer to use
- **adder_func** – (advanced) The adder to use to add the two results at the end

Return WireVector
The result WireVector

pyrtl.rtllib.multipliers.**generalized_fma**(*mult_pairs*, *add_wires*, *signed=False*, *reducer=<function wallace_reducer>*, *adder_func=<function kogge_stone>*)

Generated an opimitized fused multiply adder.

A generalized FMA unit that multiplies each pair of numbers in *mult_pairs*, then adds the resulting numbers and the values of the *add_wires* all together to form an answer. This is faster than separate adders and multipliers because you avoid unnecessary adder structures for intermediate representations.

Parameters

- **mult_pairs** – Either None (if there are no pairs to multiply) or a list of pairs of wires to multiply: *[(mult1_1, mult1_2), ...]*
- **add_wires** – Either None (if there are no individual items to add other than the *mult_pairs*), or a list of wires for adding on top of the result of the pair multiplication.
- **signed** (*bool*) – Currently not supported (will be added in the future) The default will likely be changed to True, so if you want the smallest set of wires in the future, specify this as False
- **reducer** – (advanced) The tree reducer to use
- **adder_func** – (advanced) The adder to use to add the two results at the end

Return WireVector
The result WireVector

pyrtl.rtllib.multipliers.**signed_tree_multiplier**(*A*, *B*, *reducer=<function wallace_reducer>*, *adder_func=<function kogge_stone>*)

Same as tree_multiplier, but uses two's-complement signed integers

pyrtl.rtllib.multipliers.**simple_mult**(*A*, *B*, *start*)

Builds a slow, small multiplier using the simple shift-and-add algorithm. Requires very small area (it uses only a single adder), but has long delay (worst case is *len(A)* cycles). *start* is a one-bit input to indicate inputs are ready. *done* is a one-bit output signal raised when the multiplication is finished.

Parameters

- **A** (WireVector) – input wire for the multiplication
- **B** (WireVector) – input wire for the multiplication

Return [Register, bool]
Register containing the product and the *done* signal

pyrtl.rtllib.multipliers.**tree_multiplier**(*A*, *B*, *reducer=<function wallace_reducer>*, *adder_func=<function kogge_stone>*)

Build an fast unclocked multiplier for inputs A and B using a Wallace or Dada Tree.

Parameters

- **A** (WireVector) – input wire for the multiplication
- **B** (WireVector) – input wire for the multiplication
- **reducer** (*Callable*) – Reduce the tree using either a Dada reducer or a Wallace reducer determines whether it is a Wallace tree multiplier or a Dada tree multiplier
- **adder_func** (*Callable*) – an adder function that will be used to do the last addition

Return WireVector
The multiplied result

Delay is *O(log(N))*, while area is *O(N^2)*.

5.8.6 Muxes

class pyrtl.rtllib.muxes.**MultiSelector**(*signal_wire*, **dest_wires*)

The MultiSelector allows you to specify multiple wire value results for a single select wire.

Useful for processors, finite state machines and other places where the result of many wire values are determined by a common wire signal (such as a 'state' wire).

Example:

```
with muxes.MultiSelector(select, res0, res1, res2, ...) as ms:
    ms.option(val1, data0, data1, data2, ...)
    ms.option(val2, data0_2, data1_2, data2_2, ...)
```

This means that when the `select` wire equals the `val1` wire the results will have the values in `data0, data1, data2, ...` (all ints are converted to wires)

__enter__()

For compatibility with *with* statements, which is the recommended method of using a MultiSelector.

__init__(*signal_wire*, **dest_wires*)

default(**data_signals*)

finalize()

Connects the wires.

option(*select_val*, **data_signals*)

pyrtl.rtllib.muxes.**demux**(*select*)

Demultiplexes a wire of arbitrary bitwidth

Parameters
select (WireVector) – indicates which wire to set on

Return (WireVector, ...)
a tuple of wires corresponding to each demultiplexed wire

pyrtl.rtllib.muxes.**prioritized_mux**(*selects*, *vals*)

Returns the value in the first wire for which its select bit is 1

Parameters

- **selects** (*[WireVector]*) – a list of WireVectors signaling whether a wire should be chosen
- **vals** (*[WireVector]*) – values to return when the corresponding select value is 1

Returns
WireVector

If none of the *selects* are high, the last *val* is returned

pyrtl.rtllib.muxes.**sparse_mux**(*sel*, *vals*)

Mux that avoids instantiating unnecessary mux_2s when possible.

Parameters

- **sel** (WireVector) – Select wire, determines what is selected on a given cycle
- **vals** (*dict[int, WireVector]*) – dictionary of values at mux inputs

Returns
WireVector that signifies the change

This mux supports not having a full specification. Indices that are not specified are treated as don't-cares

It also supports a specified default value, SparseDefault

5.8.7 Matrix

class pyrtl.rtllib.matrix.**Matrix**(*rows*, *columns*, *bits*, *signed=False*, *value=None*, *max_bits=64*)

Class for making a Matrix using PyRTL.

Provides the ability to perform different matrix operations.

__init__(*rows*, *columns*, *bits*, *signed=False*, *value=None*, *max_bits=64*)

Constructs a Matrix object.

Parameters

- **rows** (*int*) – the number of rows in the matrix. Must be greater than 0
- **columns** (*int*) – the number of columns in the matrix. Must be greater than 0
- **bits** (*int*) – The amount of bits per WireVector. Must be greater than 0
- **signed** (*bool*) – Currently not supported (will be added in the future)
- **value** (*(WireVector/list)*) – The value you want to initialize the Matrix with. If a WireVector, must be of size *rows* * *columns* * *bits*. If a list, must have *rows* rows and *columns* columns, and every element must fit in *bits* size. If not given, the matrix initializes to 0
- **max_bits** (*int*) – The maximum number of bits each WireVector can have, even after operations like adding two matrices together results in larger resulting WireVectors

Returns
a constructed Matrix object

property bits

Gets the number of bits each value is allowed to hold.

Returns
an integer representing the number of bits

copy()

Constructs a deep copy of the Matrix.

Returns
a Matrix copy

flatten(*order='C'*)

Flatten the matrix into a single row.

Parameters
order (*str*) – C means row-major order (C-style), and F means column-major order (Fortran-style)

Returns
A copy of the matrix flattened in to a row vector matrix

put(*ind*, *v*, *mode='raise'*)

Replace specified elements of the matrix with given values

Parameters

- **ind** (*int/list[int]/tuple[int]*) – target indices
- **v** (*int/list[int]/tuple[int]/Matrix row-vector*) – values to place in matrix at target indices; if *v* is shorter than *ind*, it is repeated as necessary
- **mode** (*str*) – how out-of-bounds indices behave; `raise` raises an error, `wrap` wraps around, and `clip` clips to the range

Note that the index is on the flattened matrix.

reshape(**newshape*, ***order*)

Create a matrix of the given shape from the current matrix.

Parameters

- **newshape** (*int/ints/tuple[int]*) – shape of the matrix to return; if a single int, will result in a 1-D row-vector of that length; if a tuple, will use values for number of rows and cols. Can also be a varargs.
- **order** (*str*) – `C` means to read from self using row-major order (C-style), and `F` means to read from self using column-major order (Fortran-style).

Returns

A copy of the matrix with same data, with a new number of rows/cols

One shape dimension in newshape can be -1; in this case, the value for that dimension is inferred from the other given dimension (if any) and the number of elements in the matrix.

Examples:

```
int_matrix = [[0, 1, 2, 3], [4, 5, 6, 7]]
matrix = Matrix.Matrix(2, 4, 4, value=int_matrix)

matrix.reshape(-1) == [[0, 1, 2, 3, 4, 5, 6, 7]]
matrix.reshape(8) == [[0, 1, 2, 3, 4, 5, 6, 7]]
matrix.reshape(1, 8) == [[0, 1, 2, 3, 4, 5, 6, 7]]
matrix.reshape((1, 8)) == [[0, 1, 2, 3, 4, 5, 6, 7]]
matrix.reshape((1, -1)) == [[0, 1, 2, 3, 4, 5, 6, 7]]

matrix.reshape(4, 2) == [[0, 1], [2, 3], [4, 5], [6, 7]]
matrix.reshape(-1, 2) == [[0, 1], [2, 3], [4, 5], [6, 7]]
matrix.reshape(4, -1) == [[0, 1], [2, 3], [4, 5], [6, 7]]
```

to_wirevector()

Outputs the PyRTL Matrix as a singular concatenated WireVector.

Returns

a Wirevector representing the whole PyRTL matrix

For instance, if we had a 2 x 1 matrix `[[wire_a, wire_b]]` it would return the concatenated wire: `wire = wire_a.wire_b`

transpose()

Constructs the transpose of the matrix

Returns
a Matrix object representing the transpose

pyrtl.rtllib.matrix.**argmax**(*matrix*, *axis=None*, *bits=None*)

Returns the index of the max value of the matrix.

Parameters

- **matrix** (*Matrix/Wirevector*) – the matrix to perform argmax operation on. If it is a WireVector, it will return itself
- **axis** (*None/int*) – The axis to perform the operation on. None refers to argmax of all items. 0 is argmax of the columns. 1 is argmax of rows. Defaults to None
- **bits** (*int*) – The bits per value of the argmax. Defaults to bits of old matrix

Returns
A WireVector or Matrix representing the argmax value

NOTE: If there are two indices with the same max value, this function picks the first instance.

pyrtl.rtllib.matrix.**concatenate**(*matrices*, *axis=0*)

Join a sequence of matrices along an existing axis.

Parameters

- **matrices** (*list[Matrix]*) – a list of matrices to concatenate one after another
- **axis** (*int*) – axis along which to join; 0 is horizontally, 1 is vertically (defaults to 0)

Returns
a new Matrix composed of the given matrices joined together

This function essentially wraps hstack/vstack.

pyrtl.rtllib.matrix.**dot**(*first*, *second*)

Performs the dot product on two matrices.

Parameters

- **first** (Matrix) – the first matrix
- **second** (Matrix) – the second matrix

Returns
a PyRTL Matrix that contains the dot product of the two PyRTL Matrices

Specifically, the dot product on two matrices is:

- If either *first* or *second* are WireVectors/have both rows and columns equal to 1, it is equivalent to `Matrix.__mul__()`
- If both *first* and *second* are both arrays (have rows or columns equal to 1), it is inner product of vectors.
- Otherwise it is `Matrix.__matmul__()` between *first* and *second*

NOTE: Row vectors and column vectors are both treated as arrays

pyrtl.rtllib.matrix.**hstack**(**matrices*)

Stack matrices in sequence horizontally (column-wise).

Parameters
matrices (*list[Matrix]*) – a list of matrices to concatenate one after another horizontally

Return Matrix
a new Matrix, with the same number of rows as the original, with a bitwidth equal to the max of the bitwidths of all the matrices

All the matrices must have the same number of rows and same 'signed' value.

For example:

```
m1 = Matrix(2, 3, bits=5,  value=[[1,2,3],
                                  [4,5,6]])
m2 = Matrix(2, 1, bits=10, value=[[17],
                                  [23]])
m3 = hstack(m1, m2)
```

m3 looks like:

```
[[1,2,3,17],
 [4,5,6,23]]
```

pyrtl.rtllib.matrix.**list_to_int**(*matrix*, *n_bits*)

Convert a Python matrix (a list of lists) into an integer.

Parameters

- **matrix** (*list[list[int]]*) – a pure Python list of lists representing a matrix
- **n_bits** (*int*) – number of bits to be used to represent each element; if an element doesn't fit in *n_bits*, it truncates the most significant bits

Return int
a *N* * *n_bits* wide WireVector containing the elements of *matrix*, where *N* is the number of elements in *matrix*

Integers that are signed will automatically be converted to their two's complement form.

This function is helpful for turning a pure Python list of lists into a integer suitable for creating a Constant WireVector that can be passed in to as a Matrix constructor's *value* argument, or for passing into a Simulation's step function for a particular input wire.

For example, calling Matrix.list_to_int([3, 5], [7, 9], 4) produces 13,689, which in binary looks like this:

```
0011 0101 0111 1001
```

Note how the elements of the list of lists were added, 4 bits at a time, in row order, such that the element at row 0, column 0 is in the most significant 4 bits, and the element at row 1, column 1 is in the least significant 4 bits.

Here's an example of using it in simulation:

```
a_vals = [[0, 1], [2, 3]]
b_vals = [[2, 4, 6], [8, 10, 12]]

a_in = pyrtl.Input(4 * 4, 'a_in')
b_in = pyrtl.Input(6 * 4, 'b_in')
a = Matrix.Matrix(2, 2, 4, value=a_in)
b = Matrix.Matrix(2, 3, 4, value=b_in)
...

sim = pyrtl.Simulation()
sim.step({
```

(continues on next page)

(continued from previous page)

```
    'a_in': Matrix.list_to_int(a_vals)
    'b_in': Matrix.list_to_int(b_vals)
})
```

pyrtl.rtllib.matrix.**matrix_wv_to_list**(*matrix_wv*, *rows*, *columns*, *bits*)

Convert a wirevector representing a matrix into a Python list of lists.

Parameters

- **matrix_wv** (WireVector) – result of calling to_wirevector() on a Matrix object
- **rows** (*int*) – number of rows in the matrix *matrix_wv* represents
- **columns** (*int*) – number of columns in the matrix *matrix_wv* represents
- **bits** (*int*) – number of bits in each element of the matrix *matrix_wv* represents

Return list[list[int]]
a Python list of lists

This is useful when printing the value of a wire you've inspected during Simulation that you know represnts a matrix.

Example:

```
values = [[1, 2, 3], [4, 5, 6]]
rows = 2
cols = 3
bits = 4
m = Matrix.Matrix(rows, cols, bits, values=values)

output = Output(name='output')
output <<= m.to_wirevector()

sim = Simulation()
sim.step({})

raw_matrix = Matrix.matrix_wv_to_list(sim.inspect('output'), rows, cols, bits)
print(raw_matrix)

# Produces:
# [[1, 2, 3], [4, 5, 6]]
```

pyrtl.rtllib.matrix.**max**(*matrix*, *axis=None*, *bits=None*)

Returns the max value in a matrix.

Parameters

- **matrix** (*Matrix/Wirevector*) – the matrix to perform max operation on. If it is a WireVector, it will return itself
- **axis** (*None/int*) – The axis to perform the operation on None refers to max of all items. 0 is max of the columns. 1 is max of rows. Defaults to None
- **bits** (*int*) – The bits per value of the max. Defaults to bits of old matrix

Returns
A WireVector or Matrix representing the max value

pyrtl.rtllib.matrix.**min**(*matrix*, *axis=None*, *bits=None*)

Returns the minimum value in a matrix.

Parameters

- **matrix** (*Matrix/Wirevector*) – the matrix to perform min operation on. If it is a WireVector, it will return itself
- **axis** (*None/int*) – The axis to perform the operation on None refers to min of all item. 0 is min of column. 1 is min of rows. Defaults to None
- **bits** (*int*) – The bits per value of the min. Defaults to bits of old matrix

Returns
A WireVector or Matrix representing the min value

pyrtl.rtllib.matrix.**multiply**(*first*, *second*)

Perform the elementwise or scalar multiplication operation.

Parameters

- **first** (Matrix) – first matrix
- **second** (*Matrix/Wirevector*) – second matrix

Returns
a Matrix object with the element wise or scalar multiplication being performed

pyrtl.rtllib.matrix.**sum**(*matrix*, *axis=None*, *bits=None*)

Returns the sum of all the values in a matrix

Parameters

- **matrix** (*Matrix/Wirevector*) – the matrix to perform sum operation on. If it is a WireVector, it will return itself
- **axis** (*None/int*) – The axis to perform the operation on None refers to sum of all item. 0 is sum of column. 1 is sum of rows. Defaults to None
- **bits** (*int*) – The bits per value of the sum. Defaults to bits of old matrix

Returns
A WireVector or Matrix representing sum

pyrtl.rtllib.matrix.**vstack**(**matrices*)

Stack matrices in sequence vertically (row-wise).

Parameters
matrices (*list[Matrix]*) – a list of matrices to concatenate one after another vertically

Return Matrix
a new Matrix, with the same number of columns as the original, with a bitwidth equal to the max of the bitwidths of all the matrices

All the matrices must have the same number of columns and same 'signed' value.

For example:

```
m1 = Matrix(2, 3, bits=5,  value=[[1,2,3],
                                  [4,5,6]])
m2 = Matrix(1, 3, bits=10, value=[[7,8,9]])
m3 = vstack(m1, m2)
```

m3 looks like:

```
[[1,2,3],
 [4,5,6],
 [7,8,9]]
```

5.8.8 Testing Utilities

pyrtl.rtllib.testingutils.**an_input_and_vals**(*bitwidth*, *test_vals=20*, *name=''*, *random_dist=<function uniform_dist>*)

Generates an input wire and a set of test values for testing purposes

Parameters

- **bitwidth** – The bitwidth of the value to be generated
- **test_vals** (*int*) – number of values to generate per wire
- **name** – name for the input wire to be generated

Returns
tuple of *input_wire*, *test_values*

pyrtl.rtllib.testingutils.**generate_in_wire_and_values**(*bitwidth*, *test_vals=20*, *name=''*, *random_dist=<function uniform_dist>*)

Generates an input wire and a set of test values for testing purposes

Parameters

- **bitwidth** – The bitwidth of the value to be generated
- **test_vals** (*int*) – number of values to generate per wire
- **name** – name for the input wire to be generated

Returns
tuple of *input_wire*, *test_values*

pyrtl.rtllib.testingutils.**make_consts**(*num_wires*, *max_bitwidth=None*, *exact_bitwidth=None*, *random_dist=<function inverse_power_dist>*)

Returns
[Const_wires]; [Const_vals]

pyrtl.rtllib.testingutils.**make_inputs_and_values**(*num_wires*, *max_bitwidth=None*, *exact_bitwidth=None*, *dist=<function uniform_dist>*, *test_vals=20*)

Generates multiple input wires and sets of test values for testing purposes

Parameters
dist (*function*) – function to generate the random values

Returns
wires; list of values for the wires

The list of values is a list of lists. The interior lists represent the values of a single wire for all of the simulation cycles

pyrtl.rtllib.testingutils.**multi_sim_multicycle**(*in_dict*, *hold_dict*, *hold_cycles*, *sim=None*)

Simulates a circuit that takes multiple cycles to complete multiple times.

Parameters

- **in_dict** – *{in_wire: [in_values, ...], ...}*
- **hold_dict** – *{hold_wire: hold_value}* The hold values for the
- **hold_cycles** –
- **sim** –

Returns

pyrtl.rtllib.testingutils.**sim_and_ret_out**(*outwire*, *inwires*, *invals*)

Simulates the net using *inwires* and *invals*, and returns the output array. Used for rapid test development.

Parameters

- **outwire** – The wire to return the output of
- **inwires** – a list of wires to read in from (*[Input, ...]*)
- **invals** – a list of input value lists (*[[int, ...], ...]*)

Returns
a list of values from the output wire simulation result

pyrtl.rtllib.testingutils.**sim_and_ret_outws**(*inwires*, *invals*)

Simulates the net using *inwires* and *invals*, and returns the output array. Used for rapid test development.

Parameters

- **inwires** – a list of wires to read in from (*[Input, ...]*)
- **invals** – a list of input value lists (*[[int, ...], ...]*)

Returns
a list of values from the output wire simulation result

pyrtl.rtllib.testingutils.**sim_multicycle**(*in_dict*, *hold_dict*, *hold_cycles*, *sim=None*)

Simulation of a circuit that takes multiple cycles to complete.

Parameters

- **in_dict** –
- **hold_dict** –
- **hold_cycles** –
- **sim** –

Returns

CHAPTER

SIX

INDEX

- genindex

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z