Migen manual

Release X

Sebastien Bourdeauducq

March 13, 2014

CHAPTER

ONE

INTRODUCTION

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

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

1.1 Background

Even though the Milkymist system-on-chip [mm] is technically successful, it suffers from several limitations stemming from its implementation in manually written Verilog HDL:

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

   (a) Combinatorial statements

   (b) Synchronous statements

   (c) Initialization of registers at reset

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

   (a) wire vs. reg in Verilog

   (b) forgetting to initialize a register at reset

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

   (d) simulation mismatches with combinatorial processes/always blocks

   (e) and more...

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

2. V*HDL support for composite types is very limited. Signals having a record type in VHDL are unidirectional, which makes them clumsy to use e.g. in bus interfaces. There is no record type support in Verilog, which means that a lot of copy-and-paste has to be done when forwarding grouped signals.

3. V*HDL support for procedurally generated logic is extremely limited. The most advanced forms of procedural generation of synthesizable logic that V*HDL offers are CPP-style directives in Verilog, combinatorial functions, and `generate` statements. Nothing really fancy, and it shows. To give a few examples:
   (a) Building highly flexible bus interconnect is not possible. Even arbitrating any given number of bus masters for commonplace protocols such as Wishbone is difficult with the tools that V*HDL puts at our disposal.
   (b) Building a memory infrastructure (including bus interconnect, bridges and caches) that can automatically adapt itself at compile-time to any word size of the SDRAM is clumsy and tedious.
   (c) Building register banks for control, status and interrupt management of cores can also largely benefit from automation.
   (d) Many hardware acceleration problems can fit into the dataflow programming model. Manual dataflow implementation in V*HDL has, again, a lot of redundancy and potential for human errors. See the Milkymist texture mapping unit [mthesis] [mxcell] for an example of this. The amount of detail to deal with manually also makes the design space exploration difficult, and therefore hinders the design of efficient architectures.
   (e) Pre-computation of values, such as filter coefficients for DSP or even simply trigonometric tables, must often be done using external tools whose results are copy-and-pasted (in the best cases, automatically) into the V*HDL source.

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

Migen is made up of several related components, which are described in this manual.

1.2 Installing Migen

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

For simulation support, an extra step is needed. See *Installing the VPI module*.

1.3 Feedback

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

CHAPTER

TWO

THE FHDL LAYER

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

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

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

2.1 Expressions

2.1.1 Integers and booleans

Python integers and booleans can appear in FHDL expressions to represent constant values in a circuit. `True` and `False` are interpreted as 1 and 0, respectively.

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

2.1.2 Signal

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

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

The properties of a signal object are:

- An integer or a (integer, boolean) pair that defines the number of bits and whether the bit of higher index of the signal is a sign bit (i.e. the signal is signed). The defaults are one bit and unsigned. Alternatively, the `min` and

`max` parameters can be specified to define the range of the signal and determine its bit width and signedness. As with Python ranges, `min` is inclusive and defaults to 0, `max` is exclusive and defaults to 2.

- A name, used as a hint for the V*HDL back-end name mangler.
- The signal's reset value. It must be an integer, and defaults to 0. When the signal's value is modified with a synchronous statement, the reset value is the initialization value of the associated register. When the signal is assigned to in a conditional combinatorial statement (`If` or `Case`), the reset value is the value that the signal has when no condition that causes the signal to be driven is verified. This enforces the absence of latches in designs. If the signal is permanently driven using a combinatorial statement, the reset value has no effect.

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

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

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

2.1.3 Operators

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

```
a * b + c
```

is equivalent to:

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

2.1.4 Slices

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

2.1.5 Concatenations

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

2.1.6 Replications

The `Replicate` object represents the equivalent of {count{expression}} in Verilog.

2.2 Statements

2.2.1 Assignment

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

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

is equivalent to:

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

2.2.2 If

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

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

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

Example:

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

2.2.3 Case

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

2.2.4 Arrays

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

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

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

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

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

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

and write it with:

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

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

2.3 Specials

2.3.1 Tri-state I/O

A triplet (O, OE, I) of one-way signals defining a tri-state I/O port is represented by the `TSTriple` object. Such objects are only containers for signals that are intended to be later connected to a tri-state I/O buffer, and cannot be used as module specials. Such objects, however, should be kept in the design as long as possible as they allow the individual one-way signals to be manipulated in a non-ambiguous way.

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

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

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

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

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

2.3.2 Instances

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

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

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

These parameters can be:

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

2.3.3 Memories

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

A memory object has the following parameters:

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

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

Options to `get_port` are:

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

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

2.3.4 Inline synthesis directives

Inline synthesis directives (pseudo-comments such as `// synthesis attribute keep of clock_signal_name is true`) are supported using the `SynthesisDirective` object. Its constructor takes as parameters a string containing the body of the directive, and optional keyword parameters that are used to replace signal names similarly to the Python string method `format`. The above example could be represented as follows:

```
SynthesisDirective("attribute keep of {clksig} is true", clksig=clock_domain.clk)
```

2.4 Modules

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

2.4.1 Combinatorial statements

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

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

For example, the module below implements a OR gate:

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

    ###

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

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

2.4.2 Synchronous statements

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

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

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

2.4.3 Submodules and specials

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

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

2.4.4 Clock domains

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

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

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

2.4.5 Summary of special attributes

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

2.4.6 Clock domain management

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

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

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

2.4.7 Finalization mechanism

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

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

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

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

The clock domain management mechanism explained above happens during finalization.

2.4.8 Simulation

The `do_simulation` method of the `Module` class can be defined and will be executed at each clock cycle, or the generator-style API can be used by defining `gen_simulation` instead. The generator yields the number of cycles it wants to wait for. See *Simulating a Migen design* for more information on using the simulator.

Simulation of designs with several clock domains is not supported yet.

2.5 Conversion for synthesis

Any FHDL module (except, of course, its simulation functions) can be converted into synthesizable Verilog HDL. This is accomplished by using the `convert` function in the `verilog` module.

The Mibuild component provides scripts to interface third-party FPGA tools to Migen and a database of boards for the easy deployment of designs.

CHAPTER

THREE

BUS SUPPORT

Migen Bus contains classes providing a common structure for master and slave interfaces of the following buses:

- Wishbone [wishbone], the general purpose bus recommended by Opencores.
- CSR-2 (see *CSR-2 bus*), a low-bandwidth, resource-sensitive bus designed for accessing the configuration and status registers of cores from software.
- LASMIbus (see *Lightweight Advanced System Memory Infrastructure*), a bus optimized for use with a high-performance frequency-ratio SDRAM controller.
- DFI [dfi] (partial), a standard interface protocol between memory controller logic and PHY interfaces.

It also provides interconnect components for these buses, such as arbiters and address decoders. The strength of the Migen procedurally generated logic can be illustrated by the following example:

```
self.submodules.wbcon = wishbone.InterconnectShared(
    [cpu.ibus, cpu.dbus, ethernet.dma, audio.dma],
    [(lambda a: a[27:] == 0, norflash.bus),
     (lambda a: a[27:] == 1, wishbone2lasmi.wishbone),
     (lambda a: a[27:] == 3, wishbone2csr.wishbone)])
```

In this example, the interconnect component generates a 4-way round-robin arbiter, multiplexes the master bus signals into a shared bus, and connects all slave interfaces to the shared bus, inserting the address decoder logic in the bus cycle qualification signals and multiplexing the data return path. It can recognize the signals in each core's bus interface thanks to the common structure mandated by Migen Bus. All this happens automatically, using only that much user code.

3.1 Configuration and Status Registers

3.1.1 CSR-2 bus

The CSR-2 bus, is a low-bandwidth, resource-sensitive bus designed for accessing the configuration and status registers of cores from software.

It is the successor of the CSR bus used in Milkymist SoC 1.x, with two modifications:

- Up to 32 slave devices (instead of 16)
- Data words are 8 bits (instead of 32)

3.1.2 Generating register banks

Migen Bank is a system comparable to wishbone-gen [wbgen], which automates the creation of configuration and status register banks and interrupt/event managers implemented in cores.

Bank takes a description made up of a list of registers and generates logic implementing it with a slave interface compatible with Migen Bus.

The lowest-level description of a register is provided by the `CSR` class, which maps to the value at a single address on the target bus. The width of the register needs to be inferior or equal to the bus word width. All accesses are atomic. It has the following signal properties as interface to the user design:

- `r`, which contains the data written from the bus interface.
- `re`, which is the strobe signal for `r`. It is active for one cycle, after or during a write from the bus. `r` is only valid when `re` is high.
- `w`, which must provide at all times the value to be read from the bus.

Names of CSRs can be omitted if they can be extracted from the variable name. When using this automatic naming feature, prefixes _, `r_` and `_r_` are removed.

Compound CSRs (which are transformed into `CSR` plus additional logic for implementation) provide additional features optimized for common applications.

The `CSRStatus` class is meant to be used as a status register that is read-only from the CPU. The user design is expected to drive its `status` signal. The advantage of using `CSRStatus` instead of using `CSR` and driving `w` is that the width of `CSRStatus` can be arbitrary. Status registers larger than the bus word width are automatically broken down into several `CSR` registers to span several addresses. Be careful that the atomicity of reads is not guaranteed.

The `CSRStorage` class provides a memory location that can be read and written by the CPU, and read and optionally written by the design. It can also span several CSR addresses. An optional mechanism for atomic CPU writes is provided; when enabled, writes to the first CSR addresses go to a back-buffer whose contents are atomically copied to the main buffer when the last address is written. When `CSRStorage` can be written to by the design, the atomicity of reads by the CPU is not guaranteed.

A module can provide bus-independent CSRs by implementing a `get_csrs` method that returns a list of instances of the classes described above. Similary, bus-independent memories can be returned as a list by a `get_memories` method.

To avoid listing those manually, a module can inherit from the `AutoCSR` class, which provides `get_csrs` and `get_memories` methods that scan for CSR and memory attributes and return their list. If the module has child objects that implement `get_csrs` or `get_memories`, they will be called by the `AutoCSR` methods and their CSR and memories added to the lists returned, with the child objects' names as prefixes.

3.1.3 Generating interrupt controllers

The event manager provides a systematic way to generate standard interrupt controllers.

Its constructor takes as parameters one or several *event sources*. An event source is an instance of either:

- `EventSourcePulse`, which contains a signal `trigger` that generates an event when high. The event stays asserted after the `trigger` signal goes low, and until software acknowledges it. An example use is to pulse `trigger` high for 1 cycle after the reception of a character in a UART.
- `EventSourceProcess`, which contains a signal `trigger` that generates an event on its falling edge. The purpose of this event source is to monitor the status of processes and generate an interrupt on their completion. The signal `trigger` can be connected to the `busy` signal of a dataflow actor, for example.

- `EventSourceLevel`, whose `trigger` contains the instantaneous state of the event. It must be set and released by the user design. For example, a DMA controller with several slots can use this event source to signal that one or more slots require CPU attention.

The `EventManager` provides a signal `irq` which is driven high whenever there is a pending and unmasked event. It is typically connected to an interrupt line of a CPU.

The `EventManager` provides a method `get_csrs`, that returns a bus-independent list of CSRs to be used with Migen Bank as explained above. Each event source is assigned one bit in each of those registers. They are:

- `status`: contains the current level of the trigger line of `EventSourceProcess` and `EventSourceLevel` sources. It is 0 for `EventSourcePulse`. This register is read-only.
- `pending`: contains the currently asserted events. Writing 1 to the bit assigned to an event clears it.
- `enable`: defines which asserted events will cause the `irq` line to be asserted. This register is read-write.

3.2 Lightweight Advanced System Memory Infrastructure

3.2.1 Rationale

The lagging of the DRAM semiconductor processes behind the logic processes has led the industry into a subtle way of ever increasing memory performance.

Modern devices feature a DRAM core running at a fraction of the logic frequency, whose wide data bus is serialized and deserialized to and from the faster clock domain. Further, the presence of more banks increases page hit rate and provides opportunities for parallel execution of commands to different banks.

A first-generation SDR-133 SDRAM chip runs both DRAM, I/O and logic at 133MHz and features 4 banks. A 16-bit chip has a 16-bit DRAM core.

A newer DDR3-1066 chip still runs the DRAM core at 133MHz, but the logic at 533MHz (4 times the DRAM frequency) and the I/O at 1066Mt/s (8 times the DRAM frequency). A 16-bit chip has a 128-bit internal DRAM core. Such a device features 8 banks. Note that the serialization also introduces multiplied delays (e.g. CAS latency) when measured in number of cycles of the logic clock.

To take full advantage of these new architectures, the memory controller should be able to peek ahead at the incoming requests and service several of them in parallel, while respecting the various timing specifications of each DRAM bank and avoiding conflicts for the shared data lines. Going further in this direction, a controller able to complete transfers out of order can provide even more performance by:

1. grouping requests by DRAM row, in order to minimize time spent on precharging and activating banks.
2. grouping requests by direction (read or write) in order to minimize delays introduced by bus turnaround and write recovery times.
3. being able to complete a request that hits a page earlier than a concurrent one which requires the cycling of another bank.

The first two techniques are explained with more details in [drreorder].

Migen and MiSoC implement their own bus, called LASMIbus, that features the last two techniques. Grouping by row had been previously explored with ASMI, but difficulties in achieving timing closure at reasonable latencies in FPGA combined with uncertain performance pay-off for some applications discouraged work in that direction.

3.2.2 Topology and transactions

The LASMI consists of one or several memory controllers (e.g. LASMIcon from MiSoC), multiple masters, and crossbar interconnect.

Each memory controller can expose several bank machines to the crossbar. This way, requests to different SDRAM banks can be processed in parallel.

Transactions on LASMI work as follows:

1. The master presents a valid address and write enable signals, and asserts its strobe signal.
2. The crossbar decodes the bank address and, in a multi-controller configuration, the controller address and connects the master to the appropriate bank machine.
3. The bank machine acknowledges the request from the master. The master can immediately issue a new request to the same bank machine, without waiting for data.
4. The bank machine sends data acknowledgements to the master, in the same order as it issued requests. After receiving a data acknowldegement, the master must either:

- present valid data after a fixed number of cycles (for writes). Masters must hold their data lines at 0 at all other times so that they can be simply ORed for each controller to produce the final SDRAM write data.
- sample the data bus after a fixed number of cycles (for reads).

5. In a multi-controller configuration, the crossbar multiplexes write and data signals to route data to and from the appropriate controller.

When there are queued requests (i.e. more request acknowledgements than data acknowledgements), the bank machine asserts its `lock` signal which freezes the crossbar connection between the master and the bank machine. This simplifies two problems:

1. Determining to which master a data acknowledgement from a bank machine should be sent.
2. Having to deal with a master queuing requests into multiple different bank machines which may collectively complete them in a different order than the master issued them.

For each master, transactions are completed in-order by the memory system. Reordering may only occur between masters, e.g. a master issuing a request that hits a page may have it completed sooner than a master requesting earlier a precharge/activate cycle of another bank.

It is suggested that memory controllers use an interface to a PHY compatible with DFI [dfi]. The DFI clock can be the same as the LASMIbus clock, with optional serialization and deserialization taking place across the PHY, as specified in the DFI standard.

3.2.3 SDRAM burst length and clock ratios

A system using LASMI must set the SDRAM burst length B, the LASMIbus word width W and the ratio between the LASMIbus clock frequency Fa and the SDRAM I/O frequency Fi so that all data transfers last for exactly one LASMIbus cycle.

More explicitly, these relations must be verified:

B = Fi/Fa

W = B*[number of SDRAM I/O pins]

For DDR memories, the I/O frequency is twice the logic frequency.

CHAPTER

FOUR

DATAFLOW

Many hardware acceleration problems can be expressed in the dataflow paradigm. It models a program as a directed graph of the data flowing between functions. The nodes of the graph are functional units called actors, and the edges represent the connections (transporting data) between them.

Actors communicate by exchanging data units called tokens. A token contains arbitrary (user-defined) data, which is a record containing one or many fields, a field being a bit vector or another record. Token exchanges are atomic (i.e. all fields are transferred at once from the transmitting actor to the receiving actor).

4.1 Actors

4.1.1 Actors and endpoints

Actors in Migen are implemented in FHDL. This low-level approach maximizes the practical flexibility: for example, an actor can manipulate the bus signals to implement a DMA master in order to read data from system memory (see *Bus actors*).

Token exchange ports of actors are called endpoints. Endpoints are unidirectional and can be sources (which transmit tokens out of the actor) or sinks (which receive tokens into the actor).

The flow of tokens is controlled using two handshake signals (strobe and acknowledgement) which are implemented by every endpoint. The strobe signal is driven by sources, and the acknowledgement signal by sinks.

`stb`	`ack`	Situation
0	0	The source endpoint does not have data to send, and the sink endpoint is not ready to accept data.
0	1	The sink endpoint is ready to accept data, but the source endpoint has currently no data to send. The sink endpoint is not required to keep its `ack` signal asserted.
1	0	The source endpoint is trying to send data to the sink endpoint, which is currently not ready to accept it. The transaction is *stalled*. The source endpoint must keep `stb` asserted and continue to present valid data until the transaction is completed.
1	1	The source endpoint is sending data to the sink endpoint which is ready to accept it. The transaction is *completed*. The sink endpoint must register the incoming data, as the source endpoint is not required to hold it valid at the next cycle.

It is permitted to generate an `ack` signal combinatorially from one or several `stb` signals. However, there should not be any combinatorial path from an `ack` to a `stb` signal.

Actors are FHDL modules that have one or several endpoint attributes of the `migen.flow.actor.Sink` or `migen.flow.actor.Source` type, and a `busy` signal.

Endpoint constructors take as parameter the layout of the data record that the endpoint is dealing with.

Record layouts are a list of fields. Each field is described by a pair consisting of:

- The field's name.

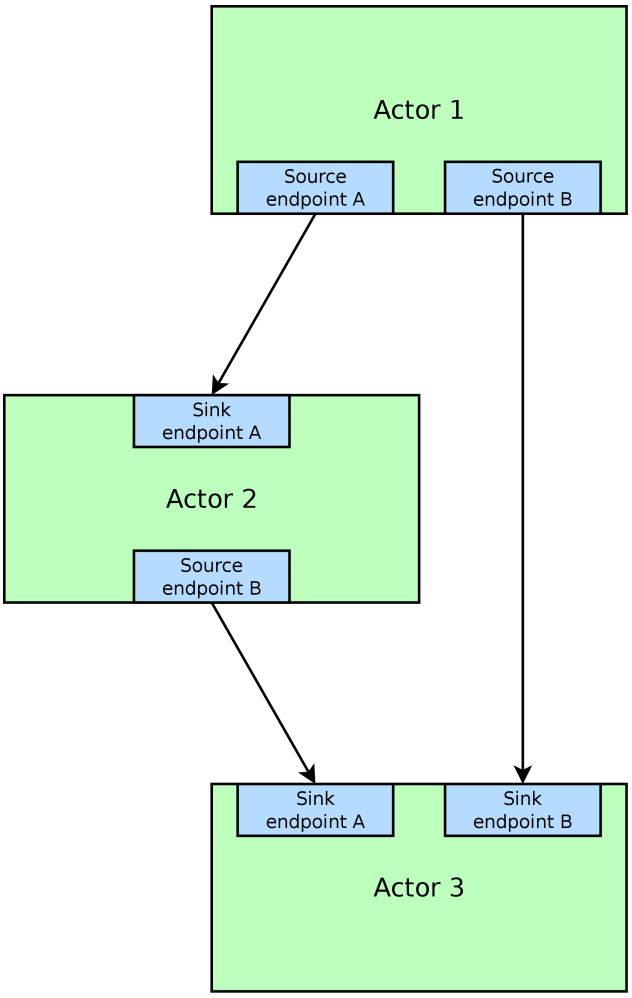

Figure 4.1: Actors and endpoints.

- Either a bit width or a (bit width, signedness) pair if the field is a bit vector, or another record layout if the field is a lower-level record.

For example, this code:

```
class MyActor(Module):
  def __init__(self):
    self.busy = Signal()
    self.operands = Sink([("a", 16), ("b", 16)])
    self.result = Source([("r", 17)])
```

creates an actor with:

- One sink named `operands` accepting data structured as a 16-bit field `a` and a 16-bit field `b`. Note that this is functionally different from having two endpoints `a` and `b`, each accepting a single 16-bit field. With a single endpoint, the data is strobed when *both* `a` and `b` are valid, and `a` and `b` are *both* acknowledged *atomically*. With two endpoints, the actor has to deal with accepting `a` and `b` independently. Plumbing actors (see *Plumbing actors*) and abstract networks (see *Actor networks*) provide a systematic way of converting between these two behaviours, so user actors should implement the behaviour that results in the simplest or highest performance design.
- One source named `result` transmitting a single 17-bit field named `r`.

Accessing the endpoints is done by manipulating the signals inside the `Source` and `Sink` objects. They hold:

- A signal object `stb`.
- A signal object `ack`.
- The data payload `payload`, which is a record with the layout given to the endpoint constructor.

4.1.2 Busy signal

The basic actor class creates a `busy` control signal that actor implementations should drive.

This signal represents whether the actor's state holds information that will cause the completion of the transmission of output tokens. For example:

- A "buffer" actor that simply registers and forwards incoming tokens should drive 1 on `busy` when its register contains valid data pending acknowledgement by the receiving actor, and 0 otherwise.
- An actor sequenced by a finite state machine should drive `busy` to 1 whenever the state machine leaves its idle state.
- An actor made of combinatorial logic is stateless and should tie `busy` to 0.

4.1.3 Common scheduling models

For the simplest and most common scheduling cases, Migen provides logic to generate the handshake signals and the busy signal. This is done through abstract actor classes that examine the endpoints defined by the user and add logic to drive their control signals (i.e. everything except the payload). The `__init__` method of the abstract scheduling class must be called after the user has created the endpoints. The `busy` signal is created by the abstract scheduling class.

These classes are usable only when the actor has exactly one sink and one source (but those endpoints can contain an arbitrary data structure), and in the cases listed below.

Combinatorial

The actor datapath is made entirely of combinatorial logic. The handshake signals pass through. A small integer adder would use this model.

This model is implemented by the `migen.flow.actor.CombinatorialActor` class. There are no parameters or additional control signals.

N-sequential

The actor consumes one token at its input, and it produces one output token after N cycles. It cannot accept new input tokens until it has produced its output. A multicycle integer divider would use this model.

This model is implemented by the `migen.flow.actor.SequentialActor` class. The constructor of this class takes as parameter the number of cycles N. The class provides an extra control signal `trigger` that pulses to 1 for one cycle when the actor should register the inputs and start its processing. The actor is then expected to provide an output after the N cycles and hold it constant until the next trigger pulse.

N-pipelined

This is similar to the sequential model, but the actor can always accept new input tokens. It produces an output token N cycles of latency after accepting an input token. A pipelined multiplier would use this model.

This model is implemented by the `migen.flow.actor.PipelinedActor` class. The constructor takes the number of pipeline stages N. There is an extra control signal `pipe_ce` that should enable or disable all synchronous statements in the datapath (i.e. it is the common clock enable signal for all the registers forming the pipeline stages).

4.2 The Migen actor library

4.2.1 Plumbing actors

Plumbing actors arbitrate the flow of data between actors. For example, when a source feeds two sinks, they ensure that each sink receives exactly one copy of each token transmitted by the source.

Most of the time, you will not need to instantiate plumbing actors directly, as abstract actor networks (see *Actor networks*) provide a more powerful solution and let Migen insert plumbing actors behind the scenes.

Buffer

The `Buffer` registers the incoming token and retransmits it. It is a pipelined actor with one stage. It can be used to relieve some performance problems or ease timing closure when many levels of combinatorial logic are accumulated in the datapath of a system.

When used in a network, abstract instances of `Buffer` are automatically configured by Migen (i.e. the appropriate token layout is set).

Combinator

This actor combines tokens from several sinks into one source.

For example, when the operands of a pipelined multiplier are available independently, the `Combinator` can turn them into a structured token that is sent atomically into the multiplier when both operands are available, simplifying the design of the multiplier actor.

Splitter

This actor does the opposite job of the `Combinator`. It receives a token from its sink, duplicates it into an arbitrary number of copies, and transmits one through each of its sources. It can optionally omit certain fields of the token (i.e. take a subrecord).

For example, an Euclidean division actor generating the quotient and the remainder in one step can transmit both using one token. The `Splitter` can then forward the quotient and the remainder independently, as integers, to other actors.

4.2.2 Structuring actors

Cast

This actor concatenates all the bits from the data of its sink (in the order as they appear in the layout) and connects them to the raw bits of its source (obtained in the same way). The source and the sink layouts must contain the same number of raw bits. This actor is a simple "connect-through" which does not use any hardware resources.

It can be used in conjunction with the bus master actors (see *Bus actors*) to destructure (resp. structure) data going to (resp. coming from) the bus.

Unpack

This actor takes a token with the fields `chunk0` ... `chunk[N-1]` (each having the same layout L) and generates N tokens with the layout L containing the data of `chunk0` ... `chunk[N-1]` respectively.

Pack

This actor receives N tokens with a layout L and generates one token with the fields `chunk0` ... `chunk[N-1]` (each having the same layout L) containing the data of the N incoming tokens respectively.

4.2.3 Simulation actors

When hardware implementation is not desired, Migen lets you program actor behaviour in "regular" Python.

For this purpose, it provides a `migen.actorlib.sim.SimActor` class. The constructor takes a generator as parameter, which implements the actor's behaviour. The user must derive the `SimActor` class and add endpoint attributes. The `busy` signal is provided by the `SimActor` class.

Generators can yield `None` (in which case, the actor does no transfer for one cycle) or one or a tuple of instances of the `Token` class. Tokens for sink endpoints are pulled and the "value" field filled in. Tokens for source endpoints are pushed according to their "value" field. The generator is run again after all transactions are completed.

The possibility to push several tokens at once is important to interact with actors that only accept a group of tokens when all of them are available.

The `Token` class contains the following items:

- The name of the endpoint from which it is to be received, or to which it is to be transmitted. This value is not modified by the transaction.

- A dictionary of values corresponding to the fields of the token. Fields that are lower-level records are represented by another dictionary. This item should be set to `None` (default) when receiving from a sink.

See `dataflow.py` in the examples folder of the Migen sources for a demonstration of the use of these actors.

4.2.4 Bus actors

Migen provides a collection of bus-mastering actors, which makes it possible for dataflow systems to access system memory easily and efficiently.

Wishbone reader

The `migen.actorlib.dma_wishbone.Reader` takes a token representing a 30-bit Wishbone address (expressed in words), reads one 32-bit word on the bus at that address, and transmits the data.

It does so using Wishbone classic cycles (there is no burst or cache support). The actor is pipelined and its throughput is only limited by the Wishbone stall cycles.

Wishbone writer

The `migen.actorlib.dma_wishbone.Writer` takes a token containing a 30-bit Wishbone address (expressed in words) and a 32-bit word of data, and writes that word to the bus.

Only Wishbone classic cycles are supported. The throughput is limited by the Wishbone stall cycles only.

LASMI reader

The `migen.actorlib.dma_lasmi.Reader` requires a LASMI master port at instantiation time. This port defines the address and data widths of the actor and how many outstanding transactions are supported.

Input tokens contain the raw LASMI address, and output tokens are wide LASMI data words.

LASMI writer

Similarly, Migen provides a LASMI writer actor that accepts tokens containing an address and write data (in the same format as a LASMI word).

4.2.5 Miscellaneous actors

Integer sequence generator

The integer sequence generator either:

- takes a token containing a maximum value N and generates N tokens containing the numbers 0 to N-1.
- takes a token containing a number of values N and a offset O and generates N-O tokens containing the numbers O to O+N-1.

The actor instantiation takes several parameters:

- the number of bits needed to represent the maximum number of generated values.
- the number of bits needed to represent the maximum offset. When this value is 0 (default), then offsets are not supported and the sequence generator accepts tokens which contain the maximum value alone.

The integer sequence generator can be used in combination with bus actors to generate addresses and read contiguous blocks of system memory (see *Bus actors*).

4.3 Actor networks

4.3.1 Graph definition

Migen represents an actor network using the `migen.flow.network.DataFlowGraph` class. It is derived from `MultiDiGraph` from the NetworkX [networkx] library.

Nodes of the graph are either:

- An existing actor (*physical actor*).
- An instance of `migen.flow.network.AbstractActor`, containing the actor class and a dictionary (*abstract actor*). It means that the actor class should be instantiated with the parameters from the dictionary. This form is needed to enable optimizations such as actor duplication or sharing during elaboration.

Edges of the graph represent the flow of data between actors. They have the following data properties:

- `source`: a string containing the name of the source endpoint, which can be `None` (Python's `None`, not the string `"None"`) if the transmitting actor has only one source endpoint.
- `sink`: a string containing the name of the sink endpoint, which can be `None` if the transmitting actor has only one sink endpoint.
- `source_subr`: if only certain fields (a subrecord) of the source endpoint should be included in the connection, their names are listed in this parameter. The `None` value connects all fields.
- `sink_subr`: if the connection should only drive certain fields (a subrecord) of the sink endpoint, they are listed here. The `None` value connects all fields.

Compared to NetworkX's `MultiDiGraph` it is based on, Migen's `DataFlowGraph` class implements an additional method that makes it easier to add actor connections to a graph:

```
add_connection(source_node, sink_node,
  source_ep=None, sink_ep=None, # default: assume nodes have 1 source/sink
                                # and use that one
  source_subr=None, sink_subr=None) # default: use whole record
```

4.3.2 Abstract and physical networks

A network (or graph) is abstract if it cannot be physically implemented by only connecting existing records together. More explicitly, a graph is abstract if any of these conditions is met:

1. A node is an abstract actor.
2. A subrecord is used at a source or a sink.
3. A single source feeds more than one sink.

The `DataFlowGraph` class implements a method `is_abstract` that tests and returns if the network is abstract.

An abstract graph can be turned into a physical graph through *elaboration*.

4.3.3 Elaboration

The most straightforward elaboration process goes as follows:

1. Whenever several sources drive different fields of a single sink, insert a `Combinator` plumbing actor. A `Combinator` should also be inserted when a single source drive only certain fields of a sink.
2. Whenever several sinks are driven by a single source (possibly by different fields of that source), insert a `Splitter` plumbing actor. A `Splitter` should also be inserted when only certain fields of a source drive a sink.
3. Whenever an actor is abstract, instantiate it.

This method is implemented by default by the `elaborate` method of the `DataFlowGraph` class, that modifies the graph in-place.

Thanks to abstract actors, there are optimization possibilities during this stage:

- Time-sharing an actor to reduce resource utilization.
- Duplicating an actor to increase performance.
- Promoting an actor to a wider datapath to enable time-sharing with another. For example, if a network contains a 16-bit and a 32-bit multiplier, the 16-bit multiplier can be promoted to 32-bit and time-shared.
- Algebraic optimizations.
- Removing redundant actors whose output is only used partially. For example, two instances of divider using the restoring method can be present in a network, and each could generate either the quotient or the remainder of the same integers. Since the restoring method produces both results at the same time, only one actor should be used instead.

None of these optimizations are implemented yet.

4.3.4 Implementation

A physical graph can be implemented and turned into a synthesizable or simulable fragment using the `migen.flow.network.CompositeActor` actor.

4.4 Performance tools

The module `migen.flow.perftools` provides utilities to analyze the performance of a dataflow network.

The class `EndpointReporter` is a simulation object that attaches to an endpoint and measures three parameters:

- The total number of clock cycles per token (CPT). This gives a measure of the raw inverse token rate through the endpoint. The smaller this number, the faster the endpoint operates. Since an endpoint has only one set of synchronous control signals, the CPT value is always superior or equal to 1 (multiple data records can however be packed into a single token, see for example *Structuring actors*).
- The average number of inactivity cycles per token (IPT). An inactivity cycle is defined as a cycle with the `stb` signal deasserted. This gives a measure of the delay between attempts at token transmissions ("slack") on the endpoint.
- The average number of stall cycles per token (NPT). A stall cycle is defined as a cycle with `stb` asserted and `ack` deasserted. This gives a measure of the "backpressure" on the endpoint, which represents the average number of wait cycles it takes for the source to have a token accepted by the sink. If all tokens are accepted immediately in one cycle, then NPT=0.

In the case of an actor network, the `DFGReporter` simulation object attaches an `EndpointReporter` to the source endpoint of each edge in the graph. The graph must not be abstract.

The `DFGReporter` contains a dictionary `nodepair_to_ep` that is keyed by `(source actor, destination actor)` pairs. Entries are other dictionaries that are keyed with the name of the source endpoint and return the associated `EndpointReporter` objects.

`DFGReporter` also provides a method `get_edge_labels` that can be used in conjunction with NetworkX's `draw_networkx_edge_labels` function to draw the performance report on a graphical representation of the graph (for an example, see *Actor network with performance data from a simulation run.*).

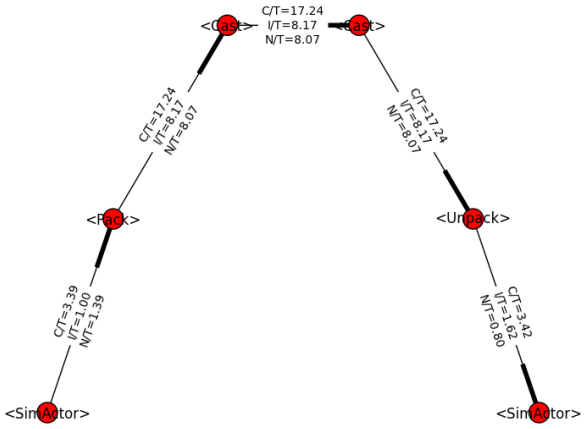

Figure 4.2: Actor network with performance data from a simulation run.

4.5 High-level actor description

Actors can be written in a subset of Python and automatically compiled into FHDL by using the Pytholite component. This functionality is still very limited for now.

CHAPTER

FIVE

SIMULATING A MIGEN DESIGN

Migen allows you to easily simulate your FHDL design and interface it with arbitrary Python code.

To interpret the design, the FHDL structure is simply converted into Verilog and then simulated using an external program (e.g. Icarus Verilog). This is is intrinsically compatible with VHDL/Verilog instantiations from Migen and maximizes software reuse.

To interface the external simulator to Python, a VPI task is called at each clock cycle and implement the test bench functionality proper - which can be fully written in Python.

Signals inside the simulator can be read and written using VPI as well. This is how the Python test bench generates stimulus and obtains the values of signals for processing.

5.1 Installing the VPI module

To communicate with the external simulator, Migen uses a UNIX domain socket and a custom protocol which is handled by a VPI plug-in (written in C) on the simulator side.

To build and install this plug-in, run the following commands from the `vpi` directory:

```
make [INCDIRS=-I/usr/...]
make install [INSTDIR=/usr/...]
```

The variable `INCDIRS` (default: empty) can be used to give a list of paths where to search for the include files. This is useful considering that different Linux distributions put the `vpi_user.h` file in various locations.

The variable `INSTDIR` (default: `/usr/lib/ivl`) specifies where the `migensim.vpi` file is to be installed.

This plug-in is designed for Icarus Verilog, but can probably be used with most Verilog simulators with minor modifications.

5.2 The generic simulator object

The generic simulator object (`migen.sim.generic.Simulator`) is the central component of the simulation.

5.2.1 Creating a simulator object

The constructor of the `Simulator` object takes the following parameters:

1. The module to simulate.
2. A top-level object (see *The top-level object*). With the default value of `None`, the simulator creates a default top-level object itself.

3. A simulator runner object (see *The external simulator runner*). With the default value of `None`, Icarus Verilog is used with the default parameters.
4. The name of the UNIX domain socket used to communicate with the external simulator through the VPI plug-in (default: "simsocket").
5. Additional keyword arguments (if any) are passed to the Verilog conversion function.

For proper initialization and clean-up, the simulator object should be used as a context manager, e.g.

```
with Simulator(tb) as s:
  s.run()
```

5.2.2 Running the simulation

Running the simulation is achieved by calling the `run` method of the `Simulator` object.

It takes an optional parameter that defines the maximum number of clock cycles that this call simulates. The default value of `None` sets no cycle limit.

5.2.3 The cycle counter

Simulation functions can read the current simulator cycle by reading the `cycle_counter` property of the `Simulator`. The cycle counter's value is 0 for the cycle immediately following the reset cycle.

5.2.4 Simplified simulation set-up

Most simulations are run in the same way and do not need the slightly heavy syntax needed to create and run a Simulator object. There is a function that exposes the most common features with a simpler syntax:

```
run_simulation(module, ncycles=None, vcd_name=None, keep_files=False)
```

5.3 Module-level simulation API

5.3.1 Simulation functions and generators

Whenever a `Module` declares a `do_simulation` method, it is executed at each cycle and can manipulate values from signal and memories (as explained in the next section).

Instead of defining such a method, `Modules` can declare a `gen_simulation` generator that is initialized at the beginning of the simulation, and yields (usually multiple times) to proceed to the next simulation cycle.

Simulation generators can yield an integer in order to wait for that number of cycles, or yield nothing (`None`) to wait for 1 cycle.

5.3.2 Reading and writing values

Simulation functions and generators take as parameter a special object that gives access to the values of the signals of the current module using the regular Python read/write syntax. Nested objects, lists and dictionaries containing signals are supported, as well as Migen memories, for reading and writing.

Here are some examples:

```
def do_simulation(self, selfp):
  selfp.foo = 42
  self.last_foo_value = selfp.foo
  selfp.dut.banks[2].bar["foo"] = 1
  self.last_memory_data = selfp.dut.mem[self.memory_index]
```

The semantics of reads and writes (respectively immediately before and after the clock edge) match those of the non-blocking assignment in Verilog. Note that because of Verilog's design, reading "variable" signals (i.e. written to using blocking assignment) directly may give unexpected and non-deterministic results and is not supported. You should instead read the values of variables after they have gone through a non-blocking assignment in the same `always` block.

Those constructs are syntactic sugar for calling the `Simulator` object's methods `rd` and `wr`, that respectively read and write data from and to the simulated design. The simulator object can be accessed as `selfp.simulator`, and for special cases it is sometimes desirable to call the lower-level methods directly.

The `rd` method takes the FHDL `Signal` object to read and returns its value as a Python integer. The returned integer is the value of the signal immediately before the clock edge.

The `wr` method takes a `Signal` object and the value to write as a Python integer. The signal takes the new value immediately after the clock edge.

References to FHDL `Memory` objects can also be passed to the `rd` and `wr` methods. In this case, they take an additional parameter for the memory address.

5.3.3 Simulation termination management

Simulation functions and generators can raise the `StopSimulation` exception. It is automatically raised when a simulation generator is exhausted. This exception disables the current simulation function, i.e. it is no longer run by the simulator. The simulation is over when all simulation functions are disabled (or the specified maximum number of cycles, if any, has been reached - whichever comes first).

Some simulation modules only respond to external stimuli - e.g. the `bus.wishbone.Tap` that snoops on bus transactions and prints them on the console - and have simulation functions that never end. To deal with those, the new API introduces "passive" simulation functions that are not taken into account when deciding to continue to run the simulation. A simulation function is declared passive by setting a "passive" attribute on it that evaluates to True. Raising `StopSimulation` in such a function still makes the simulator stop running it for the rest of the simulation.

5.4 The external simulator runner

5.4.1 Role

The runner object is responsible for starting the external simulator, loading the VPI module, and feeding the generated Verilog into the simulator.

It must implement a `start` method, called by the `Simulator`, which takes two strings as parameters. They contain respectively the Verilog source of the top-level design and the converted module.

5.4.2 Icarus Verilog support

Migen comes with a `migen.sim.icarus.Runner` object that supports Icarus Verilog.

Its constructor has the following optional parameters:

1. `extra_files` (default: `None`): lists additional Verilog files to simulate.
2. `top_file` (default: "migensim_top.v"): name of the temporary file containing the top-level.
3. `dut_file` (default: "migensim_dut.v"): name of the temporary file containing the converted fragment.
4. `vvp_file` (default: `None`): name of the temporary file compiled by Icarus Verilog. When `None`, becomes `dut_file + "vp"`.
5. `keep_files` (default: `False`): do not delete temporary files. Useful for debugging.

5.5 The top-level object

5.5.1 Role of the top-level object

The top-level object is responsible for generating the Verilog source for the top-level test bench.

It must implement a method `get` that takes as parameter the name of the UNIX socket the VPI plugin should connect to, and returns the full Verilog source as a string.

It must have the following attributes (which are read by the `Simulator` object):

- `clk_name`: name of the clock signal.
- `rst_name`: name of the reset signal.
- `dut_type`: module type of the converted fragment.
- `dut_name`: name used for instantiating the converted fragment.
- `top_name`: name/module type of the top-level design.

5.5.2 Role of the generated Verilog

The generated Verilog must:

1. instantiate the converted fragment and connect its clock and reset ports.
2. produce a running clock signal.
3. assert the reset signal for the first cycle and deassert it immediately after.
4. at the beginning, call the task `$migensim_connect` with the UNIX socket name as parameter.
5. at each rising clock edge, call the task `$migensim_tick`. It is an error to call `$migensim_tick` before a call to `$migensim_connect`.
6. set up the optional VCD output file.

5.5.3 The generic top-level object

Migen comes with a `migen.sim.generic.TopLevel` object that implements the above behaviour. It should be usable in the majority of cases.

The main parameters of its constructor are the output VCD file (default: `None`) and the levels of hierarchy that must be present in the VCD (default: 1).

CHAPTER

SIX

CASE STUDIES

6.1 A VGA framebuffer core

6.1.1 Purpose

The purpose of the VGA framebuffer core is to scan a buffer in system memory and generate an appropriately timed video signal in order to display the picture from said buffer on a regular VGA monitor.

The core is meant to be integrated in a SoC and is controllable by a CPU which can set parameters such as the framebuffer address, video resolution and timing parameters.

This case study highlights what tools Migen provides to design such a core.

6.1.2 Architecture

The framebuffer core is designed using the Migen dataflow system (see *Dataflow*). Its block diagram is given in the figure below:

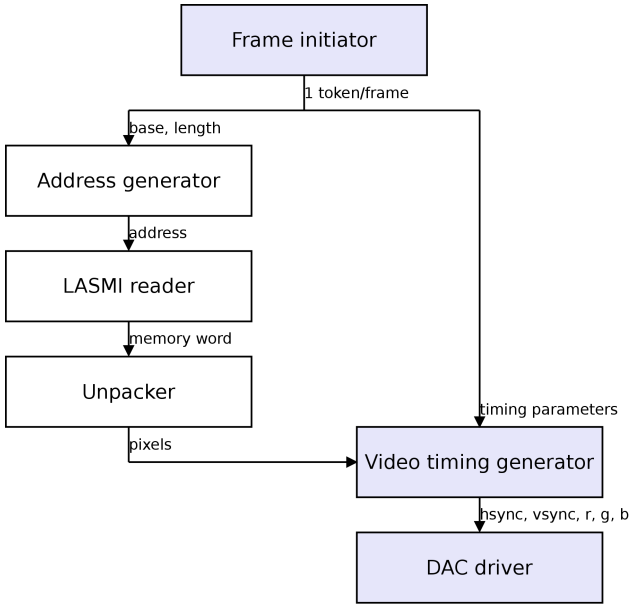

Figure 6.1: Data flow graph of the framebuffer core.

Actors drawn with a blue background are designed specifically for the framebuffer cores, the others are generic actors from the Migen library. Migen also provides the interconnect logic between the actors.

6.1.3 Frame initiator

The frame initiator generates tokens that correspond each to one complete scan of the picture (active video, synchronization pulses, front and back porches). The token contains the address and the length of the framebuffer used for the active video region, and timing parameters to generate the synchronization and porches.

Switching the framebuffer address (without tearing) is simply done by generating a token with the new address. Tearing will not occur since the new token will accepted only after the one from the previous frame has been processed (i.e. all addresses within the previous frame have been generated).

Video resolution can be changed in a similar way.

To interface with the CPU, the frame initiator uses Migen to provide a CSR bank (see *Generating register banks*).

6.1.4 Pixel fetcher

The pixel fetcher is made up of the address generator, the LASMI reader and the unpacker.

The address generator is a simple counter that takes one token containing the pair `(base, length)` and generates `length` tokens containing `base`, ..., `base+length-1`. It is implemented using a Migen library component (see *Integer sequence generator*).

Those addresses are fed into the LASMI reader (see *Bus actors*) that fetches the corresponding locations from the system memory. The LASMI reader design supports several outstanding requests, which enables it to sustain a high throughput in spite of memory latency. This feature makes it possible to utilize the available memory bandwidth to the full extent, and reduces the need for on-chip buffering.

LASMI memory words are wide and contain several pixels. The unpacking actor (see *Structuring actors*) takes a token containing a memory word and "chops" it into multiple tokens containing one pixel each.

6.1.5 Video timing generator

The video timing generator is the central piece of the framebuffer core. It takes one token containing the timing parameters of a frame, followed by as many tokens as there are pixels in the frame. It generates tokens containing the status of the horizontal/vertical synchronization signals and red/green/blue values. When the contents of those tokens are sent out at the pixel clock rate (and the red/green/blue value converted to analog), they form a valid video signal for one frame.

6.1.6 DAC driver

The DAC driver accepts and buffers the output tokens from the video timing generator, and sends their content to the DAC and video port at the pixel clock rate using an asynchronous FIFO.

CHAPTER

SEVEN

MIGEN API DOCUMENTATION

7.1 `fhdl.structure` Module

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

Addressable multiplexer

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

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

An array can be indexed multiple times.

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

Parameters **values** : iterable of ints, Values, Signals

Entries of the array. Each entry can be a numeric constant, a *Signal* or a *Record*.

Examples

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

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

Case/Switch statement

Parameters **test** : Value, in

Selector value used to decide which block to execute

cases : dict

Dictionary of cases. The keys are numeric constants to compare with *test*. The values are statements to be executed the corresponding key matches *test*. The dictionary may contain a string key *"default"* to mark a fall-through case that is executed if no other key matches.

Examples

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

makedefault(*key=None*)

Mark a key as the default case

Deletes/Substitutes any previously existing default case.

Parameters **key** : int or None

Key to use as default case if no other key matches. By default, the largest key is the default key.

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

Bases: migen.fhdl.structure.Value

Concatenate values

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

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

Parameters ***args** : Values or iterables of Values, inout

Value s to be concatenated.

Returns Cat, inout

Resulting *Value* obtained by concatentation.

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

Bases: builtins.object

Synchronous domain

Parameters **name** : str or None

Domain name. If None (the default) the name is inferred from the variable name this *ClockDomain* is assigned to (stripping any *"cd_"* prefix).

reset_less : bool

The domain does not use a reset signal. Registers within this domain are still all initialized to their reset state once, e.g. through Verilog *"initial"* statements.

Attributes

clk	(Signal, inout) The clock for this domain. Can be driven or used to drive other signals (preferably in combinatorial context).
rst	(Signal or None, inout) Reset signal for this domain. Can be driven or used to drive.

rename(*new_name*)
Rename the clock domain

Parameters **new_name** : str

New name

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

Clock signal for a given clock domain

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

Parameters **cd** : str

Clock domain to obtain a clock signal for. Defaults to *"sys"*.

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

Conditional execution of statements

Parameters **cond** : Value(1), in

Condition

***t** : Statements

Statements to execute if *cond* is asserted.

Examples

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

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

Parameters **cond** : Value(1), in

Condition

***t** : Statements

Statements to execute if previous conditions fail and *cond* is asserted.

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

Parameters ***f** : Statements

Statements to execute if all previous conditions fail.

migen.fhdl.structure.**Mux**(*sel*, *val1*, *val0*)

Multiplex between two values

Parameters **sel** : Value(1), in

Selector.

val1 : Value(N), in

val0 : Value(N), in

Input values.

Returns Value(N), out

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

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

Bases: migen.fhdl.structure.Value

Replicate a value

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

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

Parameters **v** : Value, in

Input value to be replicated.

n : int

Number of replications.

Returns Replicate, out

Replicated value.

class migen.fhdl.structure.**ResetSignal**(*cd='sys'*)

Bases: migen.fhdl.structure.Value

Reset signal for a given clock domain

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

Parameters **cd** : str

Clock domain to obtain a reset signal for. Defaults to *"sys"*.

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

Bases: migen.fhdl.structure.Value

A *Value* that can change

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

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

Parameters **bits_sign** : int or tuple

Either an integer *bits* or a tuple *(bits, signed)* specifying the number of bits in this *Signal* and whether it is signed (can represent negative values). *signed* defaults to *False*.

name : str or None

Name hint for this signal. If *None* (default) the name is inferred from the variable name this *Signal* is assigned to. Name collisions are automatically resolved by prepending names of objects that contain this *Signal* and by appending integer sequences.

variable : bool

Deprecated.

reset : int

Reset (synchronous) or default (combinatorial) value. When this *Signal* is assigned to in synchronous context and the corresponding clock domain is reset, the *Signal* assumes the given value. When this *Signal* is unassigned in combinatorial context (due to conditional assignments not being taken), the *Signal* assumes its *reset* value. Defaults to 0.

name_override : str or None

Do not use the inferred name but the given one.

min : int or None

max : int or None

If *bits_sign* is *None*, the signal bit width and signedness are determined by the integer range given by *min* (inclusive, defaults to 0) and *max* (exclusive, defaults to 2).

related : Signal or None

class migen.fhdl.structure.**Value**

Bases: migen.fhdl.structure.HUID

Base class for operands

Instances of *Value* or its subclasses can be operands to arithmetic, comparison, bitwise, and logic operators. They can be assigned (eq()) or indexed/sliced (using the usual Python indexing and slicing notation).

Values created from integers have the minimum bit width to necessary to represent the integer.

eq(*r*)

Assignment

Parameters **r** : Value, in

Value to be assigned.

Returns _Assign

Assignment statement that can be used in combinatorial or synchronous context.

7.2 fhdl.bitcontainer Module

migen.fhdl.bitcontainer.**fiter**(*v*)

Bit iterator

Parameters **v** : int, bool or Value

Returns iter

Iterator over the bits in *v*

Examples

```
>>> list(fiter(f.Signal(2)))
[<migen.fhdl.structure._Slice object at 0x...>, <migen.fhdl.structure._Slice object at 0x...>]
>>> list(fiter(4))
[0, 0, 1]
```

migen.fhdl.bitcontainer.**flen**(*v*)

Bit length of an expression

Parameters **v** : int, bool or Value

Returns int

Number of bits required to store *v* or available in *v*

Examples

```
>>> flen(f.Signal(8))
8
>>> flen(0xaa)
8
```

migen.fhdl.bitcontainer.**freversed**(*v*)

Bit reverse

Parameters **v** : int, bool or Value

Returns int or Value

Expression containing the bit reversed input.

Examples

```
>>> freversed(f.Signal(2))
<migen.fhdl.structure.Cat object at 0x...>
>>> bin(freversed(0b1011))
'0b1101'
```

migen.fhdl.bitcontainer.**fslice**(*v*, *s*)

Bit slice

Parameters **v** : int, bool or Value

s : slice or int

Returns int or Value

Expression for the slice *s* of *v*.

Examples

```
>>> fslice(f.Signal(2), 1)
<migen.fhdl.structure._Slice object at 0x...>
>>> bin(fslice(0b1101, slice(1, None, 2)))
'0b10'
>>> fslice(-1, slice(0, 4))
```

```
1
>>> fslice(-7, slice(None))
9
```

7.3 genlib.fifo Module

class migen.genlib.fifo.**AsyncFIFO**(*width_or_layout*, *depth*)
Bases: migen.fhdl.module.Module, migen.genlib.fifo._FIFOInterface

Asynchronous FIFO (first in, first out)

Read and write interfaces are accessed from different clock domains, named *read* and *write*. Use *RenameClockDomains* to rename to other names.

Data written to the input interface (*din*, *we*, *writable*) is buffered and can be read at the output interface (*dout*, *re*, *readable*). The data entry written first to the input also appears first on the output.

Parameters **width_or_layout** : int, layout

Bit width or *Record* layout for the data.

depth : int

Depth of the FIFO.

Attributes

din	(in, width_or_layout) Input data either flat or Record structured.
writable	(out) There is space in the FIFO and *we* can be asserted to load new data.
we	(in) Write enable signal to latch *din* into the FIFO. Does nothing if *writable* is not asserted.
dout	(out, width_or_layout) Output data, same type as *din*. Only valid if *readable* is asserted.
readable	(out) Output data *dout* valid, FIFO not empty.
re	(in) Acknowledge *dout*. If asserted, the next entry will be available on the next cycle (if *readable* is high then).

class migen.genlib.fifo.**SyncFIFO**(*width_or_layout*, *depth*)
Bases: migen.fhdl.module.Module, migen.genlib.fifo._FIFOInterface

Synchronous FIFO (first in, first out)

Read and write interfaces are accessed from the same clock domain. If different clock domains are needed, use AsyncFIFO.

Data written to the input interface (*din*, *we*, *writable*) is buffered and can be read at the output interface (*dout*, *re*, *readable*). The data entry written first to the input also appears first on the output.

Parameters **width_or_layout** : int, layout

Bit width or *Record* layout for the data.

depth : int

Depth of the FIFO.

Attributes

din	(in, width_or_layout) Input data either flat or Record structured.
writable	(out) There is space in the FIFO and *we* can be asserted to load new data.
we	(in) Write enable signal to latch *din* into the FIFO. Does nothing if *writable* is not asserted.
dout	(out, width_or_layout) Output data, same type as *din*. Only valid if *readable* is asserted.
readable	(out) Output data *dout* valid, FIFO not empty.
re	(in) Acknowledge *dout*. If asserted, the next entry will be available on the next cycle (if *readable* is high then).

7.4 `genlib.coding` Module

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

Decode binary to one-hot

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

Parameters **width** : int

Bit width of the output

Attributes

i	(Signal(max=width), in) Input binary
o	(Signal(width), out) Decoded one-hot
n	(Signal(1), in) Invalid, no output bits are to be asserted

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

Encode one-hot to binary

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

Parameters **width** : int

Bit width of the input

Attributes

i	(Signal(width), in) One-hot input
o	(Signal(max=width), out) Encoded binary
n	(Signal(1), out) Invalid, either none or multiple input bits are asserted

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

Priority encode requests to binary

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

Parameters **width** : int

Bit width of the input

Attributes

i	(Signal(width), in) Input requests
o	(Signal(max=width), out) Encoded binary
n	(Signal(1), out) Invalid, no input bits are asserted

7.5 genlib.cordic Module

class migen.genlib.cordic.**Cordic**(***kwargs*)
Bases: migen.genlib.cordic.TwoQuadrantCordic

Four-quadrant CORDIC

Same as TwoQuadrantCordic but with support and convergence for *abs(zi) > pi/2 in circular rotate mode or 'xi < 0* in circular vector mode.

class migen.genlib.cordic.**TwoQuadrantCordic**(*width=16*, *widthz=None*, *stages=None*, *guard=0*, *eval_mode='iterative'*, *cordic_mode='rotate'*, *func_mode='circular'*)
Bases: migen.fhdl.module.Module

Coordinate rotation digital computer

Trigonometric, and arithmetic functions implemented using additions/subtractions and shifts.

http://eprints.soton.ac.uk/267873/1/tcas1_cordic_review.pdf

http://www.andraka.com/files/crdcsrvy.pdf

http://zatto.free.fr/manual/Volder_CORDIC.pdf

The way the CORDIC is executed is controlled by *eval_mode*. If *"iterative"* the stages are iteratively evaluated, one per clock cycle. This mode uses the least amount of registers, but has the lowest throughput and highest latency. If *"pipelined"* all stages are executed in every clock cycle but separated by registers. This mode has full throughput but uses many registers and has large latency. If *"combinatorial"*, there are no registers, throughput is maximal and latency is zero. *"pipelined"* and *"combinatorial"* use the same number of sphifters and adders.

The type of trigonometric/arithmetic function is determined by *cordic_mode* and *func_mode*. g is the gain of the CORDIC.

- rotate-circular: rotate the vector *(xi, yi)* by an angle *zi*. Used to calculate trigonometric functions, *sin(), cos(), tan() = sin()/cos()*, or to perform polar-to-cartesian coordinate transformation:

$$x_o = g\cos(z_i)x_i - g\sin(z_i)y_i$$
$$y_o = g\sin(z_i)x_i + g\cos(z_i)y_i$$

- vector-circular: determine length and angle of the vector *(xi, yi)*. Used to calculate *arctan(), sqrt()* or to perform cartesian-to-polar transformation:

$$x_o = g\sqrt{x_i^2 + y_i^2}$$
$$z_o = z_i + \tan^{-1}(y_i/x_i)$$

- rotate-hyperbolic: hyperbolic functions of *zi*. Used to calculate hyprbolic functions, *sinh, cosh, tanh = cosh/sinh, exp = cosh + sinh*:

$$x_o = g\cosh(z_i)x_i + g\sinh(z_i)y_i$$
$$y_o = g\sinh(z_i)x_i + g\cosh(z_i)z_i$$

•vector-hyperbolic: natural logarithm *ln()*, *arctanh()*, and *sqrt()*. Use *x_i = a + b* and *y_i = a - b* to obtain *2* sqrt(a*b)* and *ln(a/b)/2*:

$$x_o = g\sqrt{x_i^2 - y_i^2}$$
$$z_o = z_i + \tanh^{-1}(y_i/x_i)$$

•rotate-linear: multiply and accumulate (not a very good multiplier implementation):

$$y_o = g(y_i + x_i z_i)$$

•vector-linear: divide and accumulate:

$$z_o = g(z_i + y_i/x_i)$$

Parameters **width** : int

Bit width of the input and output signals. Defaults to 16. Input and output signals are signed.

widthz : int

Bit with of *zi* and *zo*. Defaults to the *width*.

stages : int or None

Number of CORDIC incremental rotation stages. Defaults to *width + min(1, guard)*.

guard : int or None

Add guard bits to the intermediate signals. If *None*, defaults to *guard = log2(width)* which guarantees accuracy to *width* bits.

eval_mode : str, {“iterative”, “pipelined”, “combinatorial”}

cordic_mode : str, {“rotate”, “vector”}

func_mode : str, {“circular”, “linear”, “hyperbolic”}

Evaluation and arithmetic mode. See above.

Notes

Each stage *i* in the CORDIC performs the following operation:

$$x_{i+1} = x_i - md_i y_i r^{-s_{m,i}},$$
$$y_{i+1} = y_i + d_i x_i r^{-s_{m,i}},$$
$$z_{i+1} = z_i - d_i a_{m,i},$$

where:

- d_i: clockwise or counterclockwise, determined by *sign(z_i)* in rotate mode or *sign(-y_i)* in vector mode.
- r: radix of the number system (2)
- m: 1: circular, 0: linear, -1: hyperbolic
- $s_{m,i}$: non decreasing integer shift sequence
- $a_{m,i}$: elemetary rotation angle: $a_{m,i} = \tan^{-1}(\sqrt{m}s_{m,i})/\sqrt{m}$.

Attributes

xi, yi, zi	(Signal(width), in) Input values, signed.
xo, yo, zo	(Signal(width), out) Output values, signed.
new_out	(Signal(1), out) Asserted if output values are freshly updated in the current cycle.
new_in	(Signal(1), out) Asserted if new input values are being read in the next cycle.
zmax	(float) *zi* and *zo* normalization factor. Floating point *zmax* corresponds to *1<<(widthz - 1)*. *x* and *y* are scaled such that floating point *1* corresponds to *1<<(width - 1)*.
gain	(float) Cumulative, intrinsic gain and scaling factor. In circular mode *sqrt(xi**2 + yi**2)* should be no larger than *2**(width - 1)/gain* to prevent overflow. Additionally, in hyperbolic and linear mode, the operation itself can cause overflow.
interval	(int) Output interval in clock cycles. Inverse throughput.
latency	(int) Input-to-output latency. The result corresponding to the inputs appears at the outputs *latency* cycles later.

7.6 `genlib.sort` Module

class `migen.genlib.sort.`**`BitonicSort`**(*n*, *m*, *ascending=True*)

Bases: `migen.fhdl.module.Module`

Combinatorial sorting network

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

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

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

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

Parameters **n** : int

Number of inputs and output signals.

m : int

Bit width of inputs and outputs. Or a tuple of *(m, signed)*.

ascending : bool

Sort direction. *True* if input is to be sorted ascending, *False* for descending. Defaults to ascending.

Attributes

i	(list of Signals, in) Input values, each *m* wide.
o	(list of Signals, out) Output values, sorted, each *m* bits wide.

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

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

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

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

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

[wishbone] http://cdn.opencores.org/downloads/wbspec_b4.pdf

[dfi] http://www.ddr-phy.org/

[wbgen] http://www.ohwr.org/projects/wishbone-gen

[drreorder] http://www.xilinx.com/txpatches/pub/documentation/misc/improving%20ddr%20sdram%20efficiency.pdf

[networkx] http://networkx.lanl.gov/