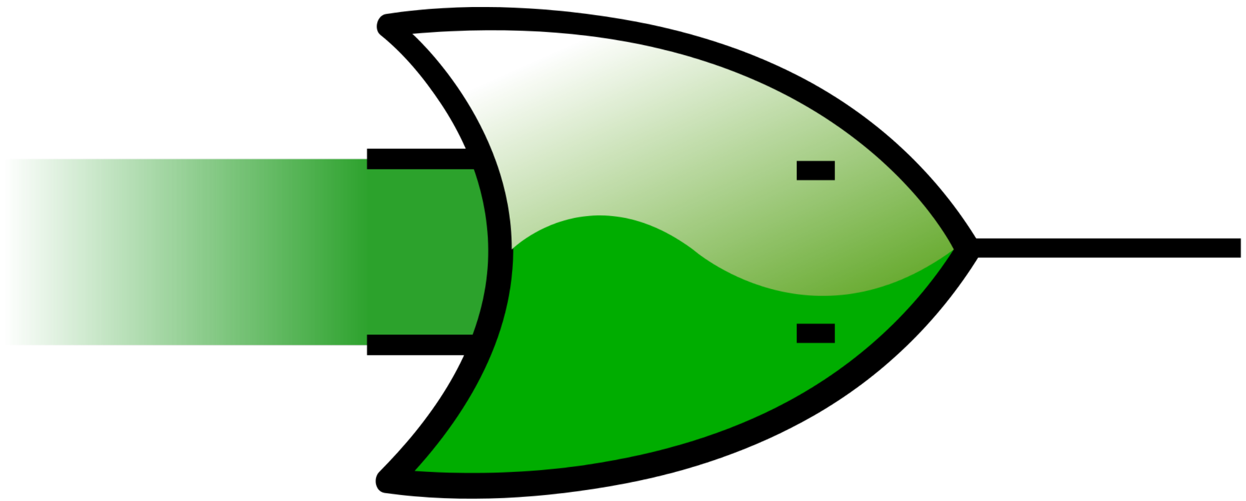

Migen manual

Release X

Sebastien Bourdeaudducq

August 12, 2012

CONTENTS

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Installing Migen . . . 2
- 1.3 Feedback . . . 2

2 The FHDL layer **3**
- 2.1 Expressions . . . 3
  - 2.1.1 Bit vector (BV) . . . 3
  - 2.1.2 Constant . . . 3
  - 2.1.3 Signal . . . 4
  - 2.1.4 Operators . . . 4
  - 2.1.5 Slices . . . 4
  - 2.1.6 Concatenations . . . 5
  - 2.1.7 Replications . . . 5
- 2.2 Statements . . . 5
  - 2.2.1 Assignment . . . 5
  - 2.2.2 If . . . 5
  - 2.2.3 Case . . . 6
  - 2.2.4 Arrays . . . 6
- 2.3 Special elements . . . 6
  - 2.3.1 Instances . . . 6
  - 2.3.2 Memories . . . 7
- 2.4 Fragments . . . 8
- 2.5 Conversion for synthesis . . . 8

3 Bus support **9**
- 3.1 Configuration and Status Registers . . . 9
  - 3.1.1 CSR-2 bus . . . 9
  - 3.1.2 Generating register banks . . . 10
  - 3.1.3 Generating interrupt controllers . . . 10
- 3.2 Advanced System Memory Infrastructure . . . 11
  - 3.2.1 Rationale . . . 11
  - 3.2.2 Topology . . . 11
  - 3.2.3 Signals . . . 11
  - 3.2.4 SDRAM burst length and clock ratios . . . 13
  - 3.2.5 Using ASMI with Migen . . . 13

4 Dataflow **15**
- 4.1 Actors . . . 15

4.1.1 Actors and endpoints 15
4.1.2 Busy signal 17
4.1.3 Common scheduling models 18
Combinatorial 18
N-sequential 18
N-pipelined 18
4.2 The Migen actor library 18
4.2.1 Plumbing actors 18
Buffer 19
Combinator 19
Splitter 19
4.2.2 Structuring actors 19
Cast 19
Unpack 19
Pack 19
4.2.3 Simulation actors 20
4.2.4 Arithmetic and logic actors 20
4.2.5 Bus actors 20
Wishbone reader 20
Wishbone writer 20
ASMI reader 21
ASMI writer 21
4.2.6 Miscellaneous actors 21
Integer sequence generator 21
4.3 Actor networks 21
4.3.1 Graph definition 21
4.3.2 Abstract and physical networks 22
4.3.3 Elaboration 22
4.3.4 Implementation 23
4.4 Performance tools 23
4.5 High-level actor description 23

5 Simulating a Migen design **25**
5.1 Installing the VPI module 25
5.2 The generic simulator object 25
5.2.1 Creating a simulator object 25
5.2.2 Running the simulation 26
5.2.3 Reading and writing signals 26
5.2.4 Reading and writing memories 26
5.2.5 Initializing signals and memories 26
5.3 The external simulator runner 27
5.3.1 Role 27
5.3.2 Icarus Verilog support 27
5.4 The top-level object 27
5.4.1 Role of the top-level object 27
5.4.2 Role of the generated Verilog 27
5.4.3 The generic top-level object 28
5.5 Simulation examples 28
5.5.1 Most basic 28
5.5.2 A few more features 29
5.5.3 Memory access 30
5.5.4 A FIR filter 30
5.5.5 Abstract bus transactions 32
5.5.6 Dataflow simulation actors 34

6 Case studies 35
6.1 A VGA framebuffer core . . . 35
6.1.1 Purpose . . . 35
6.1.2 Architecture . . . 35
6.1.3 Frame initiator . . . 36
6.1.4 Pixel fetcher . . . 36
6.1.5 Video timing generator . . . 36
6.1.6 DAC driver . . . 36

Bibliography 37

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 Milkymist developers' mailing list at devel@lists.milkymist.org.

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 and encapsulating them in "fragments". The FHDL module also contains a back-end to produce synthesizable Verilog, and some basic analysis functions. It would be possible to develop a VHDL back-end as well, though more difficult than for Verilog - we are "cheating" a bit now as Verilog provides most of the FHDL semantics.

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.

2.1 Expressions

2.1.1 Bit vector (BV)

The bit vector (BV) object defines if a constant or signal is signed or unsigned, and how many bits it has. This is useful e.g. to:

- Determine when to perform sign extension (FHDL uses the same rules as Verilog).
- Determine the size of registers.
- Determine how many bits should be used by each value in concatenations.

2.1.2 Constant

This object should be self-explanatory. All constant objects contain a BV object and a value. If no BV object is specified, one will be made up using the following rules:

- If the value is positive, the BV is unsigned and has the minimum number of bits needed to represent the constant's value in the canonical base-2 system.
- If the value is negative, the BV is signed, and has the minimum number of bits needed to represent the constant's value in the canonical two's complement, base-2 system.

2.1.3 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" and "variable" 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:

- A bit vector description
- A name, used as a hint for the V*HDL back-end name mangler.
- A boolean "variable". If true, the signal will behave like a VHDL variable, or a Verilog reg that uses blocking assignment. This parameter only has an effect when the signal's value is modified in a synchronous statement.
- 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.

2.1.4 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])
```

2.1.5 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.

2.1.6 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.

2.1.7 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, then a variable number of lists describing the various cases.

Each list contains an expression (typically a constant) describing the value to be matched, followed by the statements to be executed when there is a match. The head of the list can be the an instance of the `Default` object.

2.2.4 Arrays

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

- nest `Array` objects to create multidimensional tables.
- list any Python object in a `Array` as long as every expression appearing in a fragment 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 Special elements

2.3.1 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 (asynchronous circuits, ...).
- Breaking down a Migen system into multiple sub-systems, possibly using different clock domains.

The properties of the instance object are:

- The type of the instance (i.e. name of the instantiated module).

- A list of output ports of the instantiated module. Each element of the list is a pair containing a string, which is the name of the module's port, and either an existing signal (on which the port will be connected to) or a BV (which will cause the creation of a new signal).
- A list of input ports (likewise).
- A list of (name, value) pairs for the parameters ("generics" in VHDL) of the module.
- The name of the clock port of the module (if any). If this is specified, the port will be connected to the system clock.
- The name of the reset port of the module (likewise).
- The name of the instance (can be mangled like signal names).

2.3.2 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.
- A list of port descriptions.

Each port description contains:

- The address signal (mandatory).
- The data read signal (mandatory).
- The write enable signal (optional). If the port is using masked writes, the width of the write enable signal should match the number of sub-words.
- The data write signal (iff there is a write enable signal).
- Whether reads are synchronous (default) or asynchronous.
- The read enable port (optional, ignored for asynchronous ports).
- The write granularity (default 0), which defines the number of bits in each sub-word. If it is set to 0, the port is using whole-word writes only and the width of the write enable signal must be 1. This parameter is ignored if there is no write enable signal.
- The mode of the port (default `WRITE_FIRST`, ignored for asynchronous 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.

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 generator function can be overriden using the `memory_handler` parameter of the conversion function.

2.4 Fragments

A “fragment” is a unit of logic, which is composed of:

- A list of combinatorial statements.
- A list of synchronous statements.
- A list of instances.
- A list of memories.
- A list of simulation functions (see *Simulating a Migen design*).

Fragments can reference arbitrary signals, including signals that are referenced in other fragments. Fragments can be combined using the “+” operator, which returns a new fragment containing the concatenation of each pair of lists.

Fragments can be passed to the back-end for conversion to Verilog.

By convention, classes that generate logic implement a method called `get_fragment`. When called, this method builds a new fragment implementing the desired functionality of the class, and returns it. This convention allows fragments to be built automatically by combining the fragments from all relevant objects in the local scope, by using the autofragment module.

2.5 Conversion for synthesis

Any FHDL fragment (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.

Migen does not provide support for any specific synthesis tools or ASIC/FPGA technologies. Users must run themselves the generated code through the appropriate tool flow for hardware implementation.

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.
- ASMIbus (see *Advanced System Memory Infrastructure*), a split-transaction bus optimized for use with a high-performance, out-of-order 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:

```
wbcon = wishbone.InterconnectShared(
    [cpu.ibus, cpu.dbus, ethernet.dma, audio.dma],
    [(0, norflash.bus), (1, wishbone2asmi.wishbone),
    (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, determines that the address decoding must occur on 2 bits, 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. The resulting interconnect logic can be retrieved using `wbcon.get_fragment()`, and combined with the fragments from the rest of the system.

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.

A register can be "raw", which means that the core has direct access to it. It also means that the register width must be less or equal to the bus word width. In that case, the register object provides the following signals:

- `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.

Registers that are not raw are managed by Bank and contain fields. If the sum of the widths of all fields attached to a register exceeds the bus word width, the register will automatically be sliced into words of the maximum size and implemented at consecutive bus addresses, MSB first. Field objects have two parameters, `access_bus` and `access_dev`, determining respectively the access policies for the bus and core sides. They can take the values `READ_ONLY`, `WRITE_ONLY` and `READ_WRITE`. If the device can read, the field object provides the r signal, which contains at all times the current value of the field (kept by the logic generated by Bank). If the device can write, the field object provides the following signals:

- `w`, which provides the value to be written into the field.
- `we`, which strobes the value into the field.

As a special exception, fields that are read-only from the bus and write-only for the device do not use the `we` signal. Instead, the device must permanently drive valid data on the `w` signal.

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.
- `EventSourceLevel`, 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.

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_registers`, that returns a list of registers to be used with Migen Bank. Each event source is assigned one bit in each of those registers. They are:

- `status`: contains the current level of the trigger line of `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 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].

To enable the efficient implementation of these mechanisms, a new communication protocol with the memory controller must be devised. Migen and Milkymist SoC (-NG) implement their own bus, called ASMIbus, based on the split-transaction principle.

3.2.2 Topology

The ASMI consists of a memory controller (e.g. ASMIcon) containing a hub that connects the multiple masters, handles transaction tags, and presents a view of the pending requests to the rest of the memory controller.

Each master has a number of dedicated transaction slots allocated inside the hub. Each slot is assigned a tag, that is later used in the data transfer to identify the slot the data belongs to.

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 ASMIbus clock, with optional serialization and deserialization taking place across the PHY, as specified in the DFI standard.

3.2.3 Signals

The ASMIbus consists of two parts: the control signals, and the data signals.

The control signals are used to issue requests.

- Master-to-Hub:

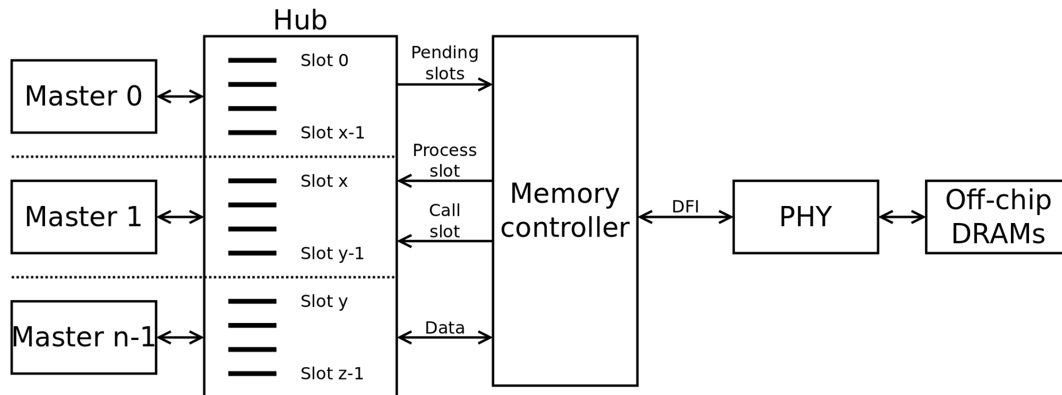

Figure 3.1: ASMI topology.

- `adr` communicates the memory address to be accessed. The unit is the word width of the particular implementation of ASMIbus.
- `we` is the write enable signal.
- `stb` qualifies the transaction request, and should be asserted until `ack` goes high.

- Hub-to-Master
  - `tag_issue` is an integer representing the transaction ("tag") attributed by the hub. The width of this signal is determined by the maximum number of in-flight transactions that the hub port can handle.
  - `ack` is asserted when `tag_issue` is valid and the transaction has been registered by the hub. A hub may assert `ack` even when `stb` is low, which means it is ready to accept any new transaction and will do as soon as `stb` goes high.

The data signals are used to complete requests.

- Hub-to-Master
  - `tag_call` is used to identify the transaction for which the data is "called". It takes the tag value that has been previously attributed by the hub to that transaction during the issue phase.
  - `call` qualifies `tag_call`.
  - `data_r` returns data from the DRAM in the case of a read transaction. It is valid for one cycle after CALL has been asserted and `tag_call` has identified the transaction. The value of this signal is undefined for the cycle after a write transaction data have been called.
- Master-to-Hub
  - `data_w` must supply data to the controller from the appropriate write transaction, on the cycle after they have been called using `call` and `tag_call`.
  - `data_wm` are the byte-granular write data masks. They are used in combination with `data_w` to identify the bytes that should be modified in the memory. The `data_wm` bit should be low for its corresponding `data_w` byte to be written.

In order to avoid duplicating the tag matching and tracking logic, the master-to-hub data signals must be driven low when they are not in use, so that they can be simply ORed together inside the memory controller. This way, only masters have to track (their own) transactions for arbitrating the data lines.

Tags represent in-flight transactions. The hub can reissue a tag as soon as the cycle when it appears on `tag_call`.

3.2.4 SDRAM burst length and clock ratios

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

3.2.5 Using ASMI with Migen

TODO: please document me!

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 derived from the the `migen.flow.actor.Actor` base class. The constructor of this base class takes a variable number of parameters, each describing one endpoint of the actor.

An endpoint description is a triple consisting of:

- The endpoint's name.

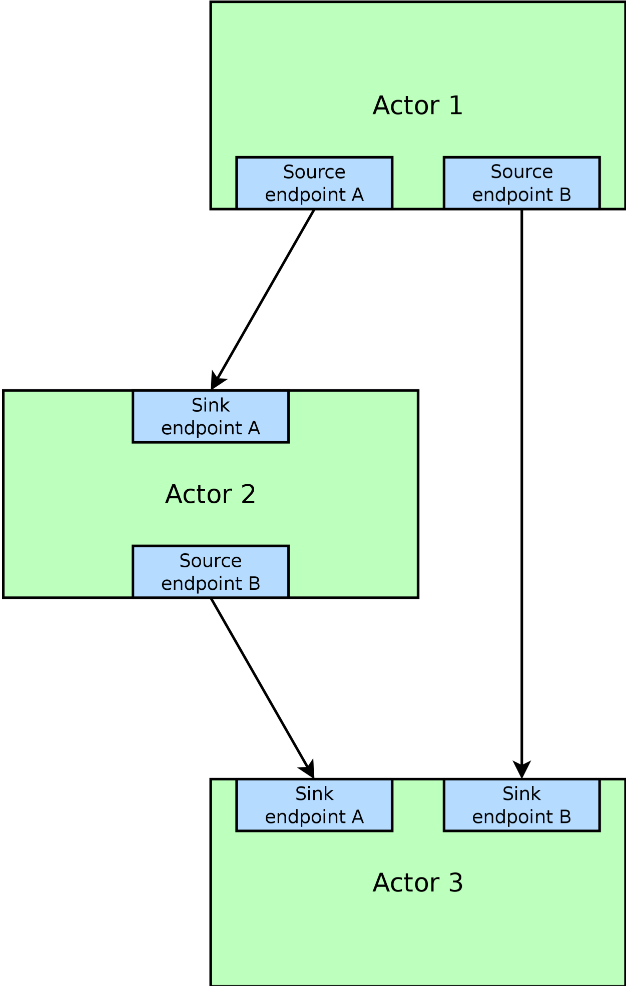

Figure 4.1: Actors and endpoints.

- A reference to the `migen.flow.actor.Sink` or the `migen.flow.actor.Source` class, defining the token direction of the endpoint.
- 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.
- Either a BV object (see *Bit vector (BV)*) if the field is a bit vector, or another record layout if the field is a lower-level record.

For example, this code:

```
Actor(
  ("operands", Sink, [("a", BV(16)), ("b", BV(16))]),
  ("result", Source, [("r", BV(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`.

Implementing the functionality of the actor can be done in two ways:

- Overloading the `get_fragment` method.
- Overloading both the `get_control_fragment` and `get_process_fragment` methods. The `get_control_fragment` method should return a fragment that manipulates the control signals (strobes, acknowledgements and the actor's busy signal) while `get_process_fragment` should return a fragment that manipulates the token payload. Overloading `get_control_fragment` alone allows you to define abstract actor classes implementing a given scheduling model. Migen comes with a library of such abstract classes for the most common schedules (see *Common scheduling models*).

Accessing the endpoints is done via the `endpoints` dictionary, which is keyed by endpoint names and contains instances of the `migen.flow.actor.Endpoint` class. The latter holds:

- A signal object `stb`.
- A signal object `ack`.
- The data payload `token`. The individual fields are the items (in the Python sense) of this object.

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 overload `get_control_fragment` only, and the user should overload `get_process_fragment` to implement the actor's payload.

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, and a list of endpoints (similarly to the base `migen.flow.actor.Actor` class). The generator implements the actor's behaviour.

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 simulation actors* for an example demonstrating the use of these actors.

4.2.4 Arithmetic and logic actors

The `migen.actorlib.ala` module provides arithmetic and logic actors for the usual integer operations.

If complex operation combinations are needed, the `ComposableNode` class can be used. It overloads Python operators to make them instantiate the arithmetic and logic actors and connect them into an existing network. This creates a small internal domain-specific language (DSL).

The `ComposableNode` class is a derivative of the `ActorNode` class (see *Actor networks*) and should be used in the place of the latter when the DSL feature is desired.

4.2.5 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.

ASMI reader

The `migen.actorlib.dma_asmi.Reader` requires a ASMI 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 ASMI address, and output tokens are wide ASMI data words.

If more than one slot are assigned to the port, the reader actor implements a reorder buffer (so that the order of the output tokens matches that of the input tokens even if the memory system completes transactions out-of-order) and is capable of supporting as many outstanding transactions as there are slots.

ASMI writer

TODO

4.2.6 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 instances of the `migen.flow.network.ActorNode` class. The latter can represent actors in two ways:

- A reference to an existing actor (*physical actor*).
- An class and a dictionary (*abstract actor*), meaning 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.*).

4.5 High-level actor description

Warning: Not implemented yet, just an idea.

It is conceivable that a CAL [cal] to FHDL compiler be implemented at some point, to support higher level descriptions of some actors and reuse of third-party RVC-CAL applications. [orcc] [orcapps] [opendf]

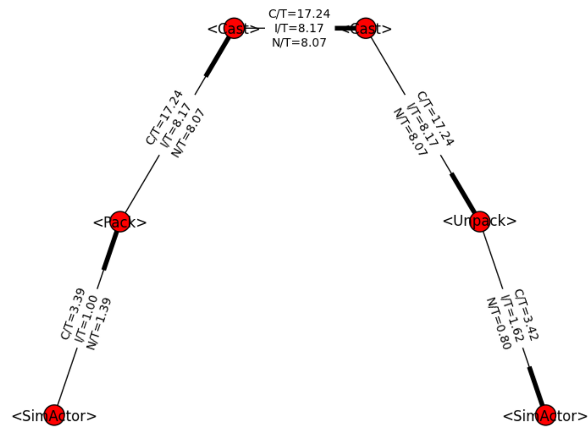

Figure 4.2: Actor network with performance data from a simulation run.

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 fragment to simulate. The fragment can (and generally does) contain both synthesizable code and a non-synthesizable list of simulation functions.

2. A simulator runner object (see *The external simulator runner*).
3. A top-level object (see *The top-level object*). With the default value of `None`, the simulator creates a default top-level object itself.
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.

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 -1 sets no cycle limit.

The simulation runs until the maximum number of cycles is reached, or a simulation function sets the property `interrupt` to `True` in the `Simulator` object.

At each clock cycle, the `Simulator` object runs in turn all simulation functions listed in the fragment. Simulation functions must take exactly one parameter which is used by the instance of the `Simulator` object to pass a reference to itself.

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.

5.2.3 Reading and writing signals

Reading and writing signals is done by calling the `Simulator` object's methods `rd` and `wr` (respectively) from simulation functions.

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.

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.

5.2.4 Reading and writing memories

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.

5.2.5 Initializing signals and memories

A simulation function can access (and typically initialize) signals and memories during the reset cycle if it has its property `initialize` set to `True`.

In this case, it will be run once at the beginning of the simulation with a cycle counter value of -1 indicating the reset cycle.

5.3 The external simulator runner

5.3.1 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 fragment.

5.3.2 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.

5.4 The top-level object

5.4.1 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.

5.4.2 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.

5.4.3 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).

5.5 Simulation examples

5.5.1 Most basic

```
# Copyright (C) 2012 Vermeer Manufacturing Co.
# License: GPLv3 with additional permissions (see README).

from migen.fhdl.structure import *
from migen.sim.generic import Simulator
from migen.sim.icarus import Runner

# Our simple counter, which increments at every cycle
# and prints its current value in simulation.
class Counter:
        def __init__(self):
                self.count = Signal(BV(4))

        # This function will be called at every cycle.
        def do_simulation(self, s):
                # Simply read the count signal and print it.
                # The output is:
                # Count: 0
                # Count: 1
                # Count: 2
                # ...
                print("Count: " + str(s.rd(self.count)))

        def get_fragment(self):
                # At each cycle, increase the value of the count signal.
                # We do it with convertible/synthesizable FHDL code.
                sync = [self.count.eq(self.count + 1)]
                # List our simulation function in the fragment.
                sim = [self.do_simulation]
                return Fragment(sync=sync, sim=sim)

def main():
        dut = Counter()
        # Use the Icarus Verilog runner.
        # We do not specify a top-level object, and use the default.
        sim = Simulator(dut.get_fragment(), Runner())
```

```
        # Since we do not use sim.interrupt, limit the simulation
        # to some number of cycles.
        sim.run(20)

main()
```

5.5.2 A few more features

```
# Copyright (C) 2012 Vermeer Manufacturing Co.
# License: GPLv3 with additional permissions (see README).

from migen.fhdl.structure import *
from migen.sim.generic import Simulator, TopLevel
from migen.sim.icarus import Runner

# A slightly improved counter.
# Has a clock enable (CE) signal, counts on more bits
# and resets with a negative number.
class Counter:
        def __init__(self):
                self.ce = Signal()
                # Demonstrate negative numbers and signals larger than 32 bits.
                self.count = Signal(BV(37, True), reset=-5)

        def do_simulation(self, s):
                # Only assert CE every second cycle.
                # => each counter value is held for two cycles.
                if s.cycle_counter % 2:
                        s.wr(self.ce, 0) # This is how you write to a signal.
                else:
                        s.wr(self.ce, 1)
                print("Cycle: " + str(s.cycle_counter) + " Count: " + \
                        str(s.rd(self.count)))
        # Set the "initialize" property on our simulation function.
        # The simulator will call it during the reset cycle,
        # with s.cycle_counter == -1.
        do_simulation.initialize = True

        # Output is:
        # Cycle: -1 Count: 0
        # Cycle: 0 Count: -5
        # Cycle: 1 Count: -5
        # Cycle: 2 Count: -4
        # Cycle: 3 Count: -4
        # Cycle: 4 Count: -3
        # ...

        def get_fragment(self):
                sync = [If(self.ce, self.count.eq(self.count + 1))]
                sim = [self.do_simulation]
                return Fragment(sync=sync, sim=sim)

def main():
        dut = Counter()
        # Instantiating the generic top-level ourselves lets us
        # specify a VCD output file.
        sim = Simulator(dut.get_fragment(), Runner(), TopLevel("my.vcd"))
```

```
        sim.run(20)

main()
```

5.5.3 Memory access

```
# Copyright (C) 2012 Vermeer Manufacturing Co.
# License: GPLv3 with additional permissions (see README).

from migen.fhdl.structure import *
from migen.sim.generic import Simulator
from migen.sim.icarus import Runner

class Mem:
        def __init__(self):
                self.a = Signal(BV(12))
                self.d = Signal(BV(16))
                p = MemoryPort(self.a, self.d)
                # Initialize the beginning of the memory with integers
                # from 0 to 19.
                self.mem = Memory(16, 2**12, p, init=list(range(20)))

        def do_simulation(self, s):
                # Read the memory. Use the cycle counter as address.
                value = s.rd(self.mem, s.cycle_counter)
                # Print the result. Output is:
                # 0
                # 1
                # 2
                # ...
                print(value)
                # Demonstrate how to interrupt the simulator.
                if value == 10:
                        s.interrupt = True

        def get_fragment(self):
                return Fragment(memories=[self.mem], sim=[self.do_simulation])

def main():
        dut = Mem()
        sim = Simulator(dut.get_fragment(), Runner())
        # No need for a cycle limit here, we use sim.interrupt instead.
        sim.run()

main()
```

5.5.4 A FIR filter

```
# Copyright (C) 2012 Vermeer Manufacturing Co.
# License: GPLv3 with additional permissions (see README).

from math import cos, pi
from scipy import signal
import matplotlib.pyplot as plt
```

```
from migen.fhdl.structure import *
from migen.fhdl import verilog
from migen.corelogic.misc import optree
from migen.fhdl import autofragment
from migen.sim.generic import Simulator, PureSimulable
from migen.sim.icarus import Runner

# A synthesizable FIR filter.
class FIR:
	def __init__(self, coef, wsize=16):
		self.coef = coef
		self.wsize = wsize
		self.i = Signal(BV(self.wsize, True))
		self.o = Signal(BV(self.wsize, True))

	def get_fragment(self):
		muls = []
		sync = []
		src = self.i
		for c in self.coef:
			sreg = Signal(BV(self.wsize, True))
			sync.append(sreg.eq(src))
			src = sreg
			c_fp = int(c*2**(self.wsize - 1))
			c_e = Constant(c_fp, BV(bits_for(c_fp), True))
			muls.append(c_e*sreg)
		sum_full = Signal(BV(2*self.wsize-1, True))
		sync.append(sum_full.eq(optree("+", muls)))
		comb = [self.o.eq(sum_full[self.wsize-1:])]
		return Fragment(comb, sync)

# A test bench for our FIR filter.
# Generates a sine wave at the input and records the output.
class TB(PureSimulable):
	def __init__(self, fir, frequency):
		self.fir = fir
		self.frequency = frequency
		self.inputs = []
		self.outputs = []

	def do_simulation(self, s):
		f = 2**(self.fir.wsize - 1)
		v = 0.1*cos(2*pi*self.frequency*s.cycle_counter)
		s.wr(self.fir.i, int(f*v))
		self.inputs.append(v)
		self.outputs.append(s.rd(self.fir.o)/f)

def main():
	# Compute filter coefficients with SciPy.
	coef = signal.remez(80, [0, 0.1, 0.1, 0.5], [1, 0])
	fir = FIR(coef)

	# Simulate for different frequencies and concatenate
	# the results.
	in_signals = []
	out_signals = []
	for frequency in [0.05, 0.07, 0.1, 0.15, 0.2]:
		tb = TB(fir, frequency)
```

```
                fragment = autofragment.from_local()
                sim = Simulator(fragment, Runner())
                sim.run(100)
                del sim
                in_signals += tb.inputs
                out_signals += tb.outputs

        # Plot data from the input and output waveforms.
        plt.plot(in_signals)
        plt.plot(out_signals)
        plt.show()

        # Print the Verilog source for the filter.
        print(verilog.convert(fir.get_fragment(),
                ios={fir.i, fir.o}))

main()
```

5.5.5 Abstract bus transactions

```
# Copyright (C) 2012 Vermeer Manufacturing Co.
# License: GPLv3 with additional permissions (see README).

from random import Random

from migen.fhdl.structure import *
from migen.fhdl import autofragment
from migen.bus.transactions import *
from migen.bus import wishbone, asmibus
from migen.sim.generic import Simulator
from migen.sim.icarus import Runner

# Our bus master.
# Python generators let us program bus transactions in an elegant sequential style.
def my_generator():
        prng = Random(92837)

        # Write to the first addresses.
        for x in range(10):
                t = TWrite(x, 2*x)
                yield t
                print("Wrote in " + str(t.latency) + " cycle(s)")
                # Insert some dead cycles to simulate bus inactivity.
                for delay in range(prng.randrange(0, 3)):
                        yield None

        # Read from the first addresses.
        for x in range(10):
                t = TRead(x)
                yield t
                print("Read " + str(t.data) + " in " + str(t.latency) + " cycle(s)")
                for delay in range(prng.randrange(0, 3)):
                        yield None

# Our bus slave.
class MyModel:
        def read(self, address):
```

```
                return address + 4

class MyModelWB(MyModel, wishbone.TargetModel):
        def __init__(self):
                self.prng = Random(763627)

        def can_ack(self, bus):
                # Simulate variable latency.
                return self.prng.randrange(0, 2)

class MyModelASMI(MyModel, asmibus.TargetModel):
        pass

def test_wishbone():
        print("*** Wishbone test")

        # The "wishbone.Initiator" library component runs our generator
        # and manipulates the bus signals accordingly.
        master = wishbone.Initiator(my_generator())
        # The "wishbone.Target" library component examines the bus signals
        # and calls into our model object.
        slave = wishbone.Target(MyModelWB())
        # The "wishbone.Tap" library component examines the bus at the slave port
        # and displays the transactions on the console (<TRead...>/<TWrite...>).
        tap = wishbone.Tap(slave.bus)
        # Connect the master to the slave.
        intercon = wishbone.InterconnectPointToPoint(master.bus, slave.bus)
        # A small extra simulation function to terminate the process when
        # the initiator is done (i.e. our generator is exhausted).
        def end_simulation(s):
                s.interrupt = master.done
        fragment = autofragment.from_local() + Fragment(sim=[end_simulation])
        sim = Simulator(fragment, Runner())
        sim.run()

def test_asmi():
        print("*** ASMI test")

        # Create a hub with one port for our initiator.
        hub = asmibus.Hub(32, 32)
        port = hub.get_port()
        hub.finalize()
        # Create the initiator, target and tap (similar to the Wishbone case).
        master = asmibus.Initiator(port, my_generator())
        slave = asmibus.Target(hub, MyModelASMI())
        tap = asmibus.Tap(hub)
        # Run the simulation (same as the Wishbone case).
        def end_simulation(s):
                s.interrupt = master.done
        fragment = autofragment.from_local() + Fragment(sim=[end_simulation])
        sim = Simulator(fragment, Runner())
        sim.run()

test_wishbone()
test_asmi()

# Output:
# <TWrite adr:0x0 dat:0x0>
```

```
# Wrote in 0 cycle(s)
# <TWrite adr:0x1 dat:0x2>
# Wrote in 0 cycle(s)
# <TWrite adr:0x2 dat:0x4>
# Wrote in 0 cycle(s)
# <TWrite adr:0x3 dat:0x6>
# Wrote in 1 cycle(s)
# <TWrite adr:0x4 dat:0x8>
# Wrote in 1 cycle(s)
# <TWrite adr:0x5 dat:0xa>
# Wrote in 2 cycle(s)
# ...
# <TRead adr:0x0 dat:0x4>
# Read 4 in 2 cycle(s)
# <TRead adr:0x1 dat:0x5>
# Read 5 in 2 cycle(s)
# <TRead adr:0x2 dat:0x6>
# Read 6 in 1 cycle(s)
# <TRead adr:0x3 dat:0x7>
# Read 7 in 1 cycle(s)
# ...
```

5.5.6 Dataflow simulation actors

```
from migen.fhdl.structure import *
from migen.flow.actor import *
from migen.flow.network import *
from migen.actorlib.sim import *
from migen.sim.generic import Simulator
from migen.sim.icarus import Runner

def source_gen():
        for i in range(10):
                print("Sending:  " + str(i))
                yield Token("source", {"value": i})

def sink_gen():
        while True:
                t = Token("sink")
                yield t
                print("Received: " + str(t.value["value"]))

def main():
        source = ActorNode(SimActor(source_gen(), ("source", Source, [("value", BV(32))])))
        sink = ActorNode(SimActor(sink_gen(), ("sink", Sink, [("value", BV(32))])))
        g = DataFlowGraph()
        g.add_connection(source, sink)
        comp = CompositeActor(g)
        def end_simulation(s):
                s.interrupt = source.actor.done
        fragment = comp.get_fragment() + Fragment(sim=[end_simulation])
        sim = Simulator(fragment, Runner())
        sim.run()

main()
```

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:

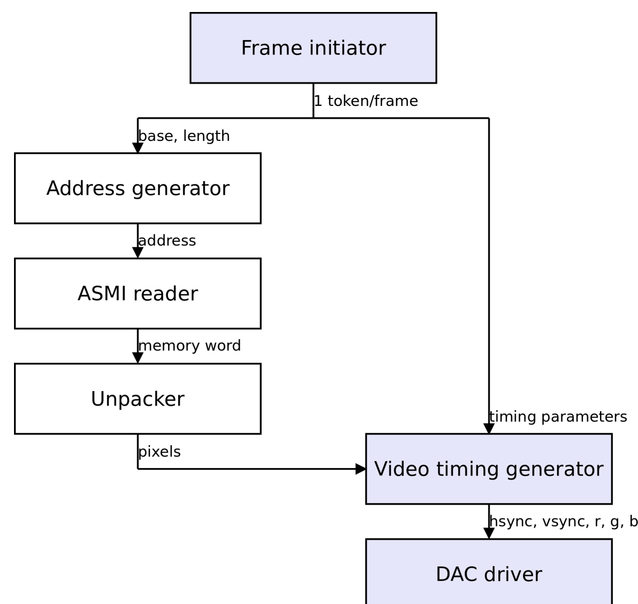

Figure 6.1: Data flow graph of the framebuffer core.

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 ASMI 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 ASMI reader (see *Bus actors*) that fetches the corresponding locations from the system memory. The ASMI reader design supports an arbitrary number of outstanding requests (which is equal to the number of slots in its ASMI port), which enables it to sustain a high throughput in spite of memory latency. The ASMI reader also contains a reorder buffer and generates memory word tokens in the order of the supplied address tokens, even if the memory system completes the transactions in a different order (see see *Advanced System Memory Infrastructure* for information about reordering). These features make it possible to utilize the available memory bandwidth to the full extent, and reduce the need for on-chip buffering.

ASMI 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.

BIBLIOGRAPHY

[mm] http://www.milkymist.org

[mthesis] http://milkymist.org/thesis/thesis.pdf

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

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

[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/

[cal] http://opendf.svn.sourceforge.net/viewvc/opendf/trunk/doc/GentleIntro/GentleIntro.pdf

[orcc] http://orcc.sourceforge.net/

[orcapps] http://orc-apps.sourceforge.net/

[opendf] http://opendf.sourceforge.net/