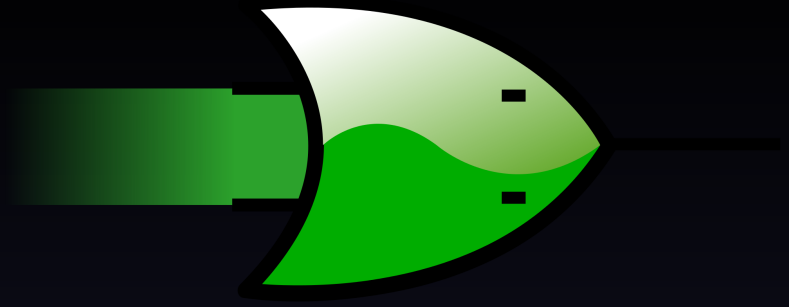
Migen

A Python toolbox for building complex digital hardware

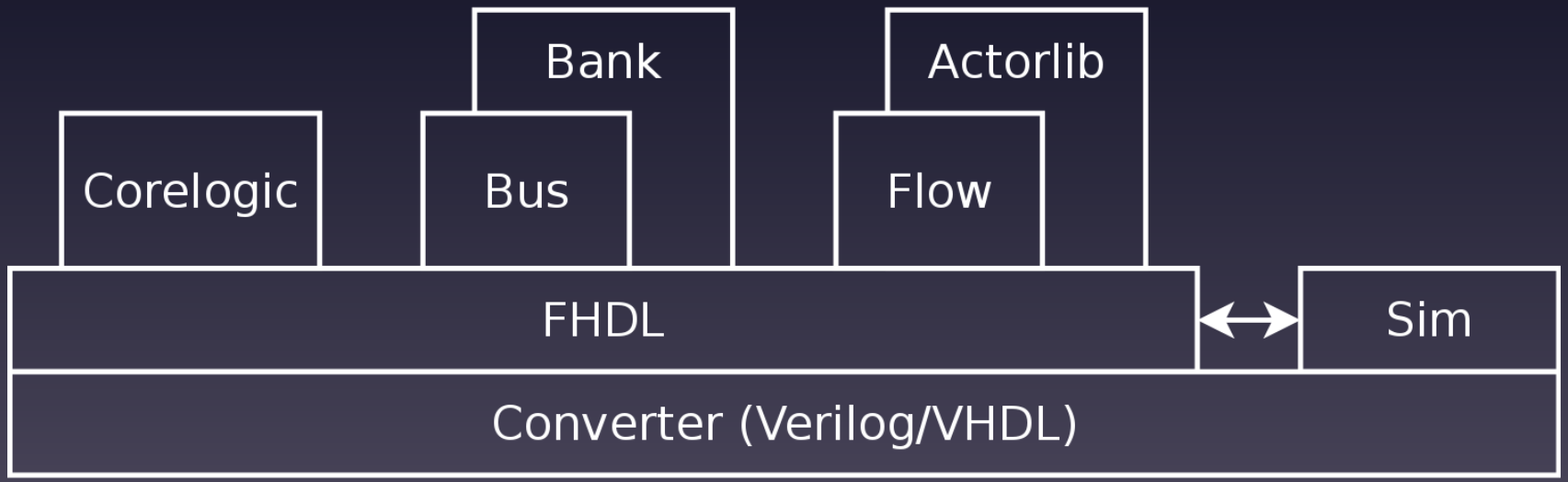
Sébastien Bourdeauducq

2012



migen

User applications and designs



FHDL

- Python as a meta-language for HDL
  - Think of a generate statement on steroids
- Restricted to synchronous circuits
- Designs are split into:
  - synchronous statements $\Longleftrightarrow$ always @(posedge clk)
    (VHDL: process(clk) begin if rising_edge(clk) then)
  - combinatorial statements $\Longleftrightarrow$ always @(*)
    (VHDL: process(all inputs) begin)
- Statements expressed using nested Python objects
  - Various syntax tricks to make them look nicer
    ("internal domain-specific language")

FHDL crash course

- Basic element is `Signal`.
  - Similar to Verilog `wire/reg` and VHDL `signal`.
- Signals can be combined to form expressions.
  - e.g. `(a & b) | c`
- Signals have a `eq` method that returns an assignment to that signal.
  - e.g. `x.eq((a & b) | c)`
- Assignments are collected into lists.
  - Control structures (`If`, `Case`) also supported.
- List of combinatorial assignments + list of synchronous assignments + "goodies" = `Fragment`
- Fragments can be combined to form a design to be converted for synthesis or simulated.

Conversion for synthesis

- FHDL is entirely convertible to synthesizable Verilog
- VHDL proposed, in development[1]

```
>>> from migen.fhdl.structure import *
>>> from migen.fhdl import verilog
>>> counter = Signal(BV(16))
>>> o = Signal()
>>> comb = [o.eq(counter == 0)]
>>> sync = [counter.eq(counter + 1)]
>>> f = Fragment(comb, sync)
>>> print(verilog.convert(f, ios={o}))
```

[1] https://github.com/peteut/migen

```
module top(input sys_rst, input sys_clk, output o);

reg [15:0] counter;

assign o = (counter == 1'd0);

always @(posedge sys_clk) begin
        if (sys_rst) begin
                counter <= 16'd0;
        end else begin
                counter <= (counter + 1'd1);
        end
end

endmodule
```

Simulation

- Fragments contain a list of Python functions to execute at each clock cycle during simulations.
- Simulator provide read and write methods that manipulate FHDL Signal objects.
- Powerful Python features, e.g. generators:

```
def my_generator():
    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
master1 = wishbone.Initiator(my_generator())
master2 = asmibus.Initiator(port, my_generator())
```

Verilog/VHDL interface

- Fragments can contain instances of external Verilog or VHDL modules.
- In development: Migen “embedded” mode[2]
  - Similar to PHP’s <? and ?>
  - Write most of your code in Verilog/VHDL, insert Python code between [- and -]

[2] https://github.com/Florent-Kermarrec/migen

Bus support

- Wishbone[3]
- SRAM-like CSR
- DFI [4]
- ASMI

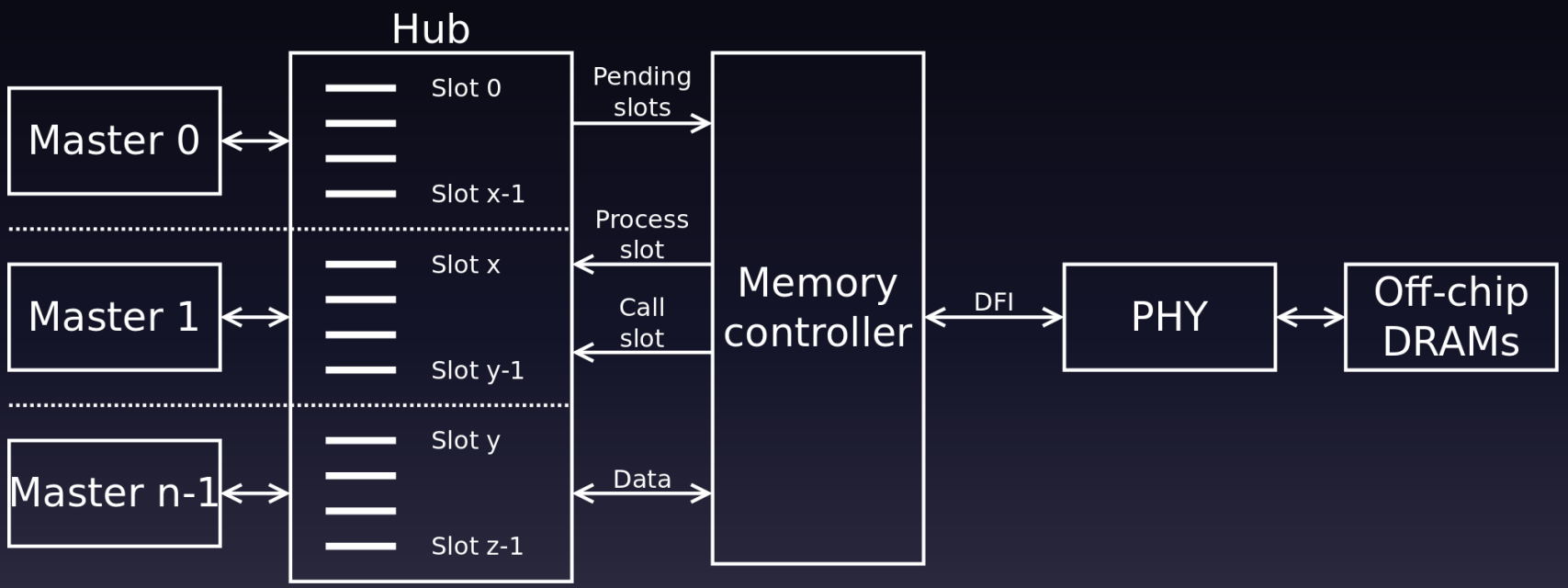

```
wishbonecon0 = wishbone.InterconnectShared(
        [cpu0.ibus, cpu0.dbus],
              [(binc("000"), norflash0.bus),
               (binc("001"), sram0.bus),
               (binc("011"), minimac0.membus),
               (binc("10"), wishbone2asmi0.wishbone),
               (binc("11"), wishbone2csr0.wishbone)])
```

[3] http://www.opencores.org
[4] http://www.ddr-phy.org

CSR banks and interrupt controllers

```
_rxtx = RegisterRaw("rxtx", 8)
_divisor = RegisterField("divisor", 16, reset=int(clk_freq/baud/16))
_tx_event = EventSourceLevel()
_rx_event = EventSourcePulse()
events = EventManager(_tx_event, _rx_event)
bank = csrgen.Bank([_rxtx, _divisor] + events.get_registers(),
    address=address)
```

ASMI



ASMI (Advanced System Memory Infrastructure) topology

For master devices

- Each master has one or several dedicated request slots
- At any slot, master presents valid address and WE and asserts request
- Slot number (*tag*) appears on shared bus
  - nb: this sharing is not a performance bottleneck, DRAM data pins already have exact same sharing
- For a read: followed by data from DRAM in the next cycle
- For a write: master is required to output valid data in the next cycle

Proposed memory controller

Memory controller operates several *bank machines* in parallel

- Each bank machine sees all outstanding requests for the DRAM bank it manages
- Tracks open row, detects page hits
- May reorder requests to optimize page hit rate
- Ensures per-bank timing specifications are met (tRP, tRCD, tWR)
- Generates DRAM-level requests (PRECHARGE, ACTIVATE, READ, WRITE)

Proposed memory controller

Command steering stage picks final requests

- In a frequency-ratio system, may issue multiple commands from several bank machines in a single cycle
  - PHY uses SERDES to handle I/O
  - FPGAs are horribly and painfully SLOW, so we need such tricks even for DDR333 (2002!!!)
- May group writes and reads to reduce turnaround times
- Ensures no read-to-write conflict occurs on the shared bidirectional data bus
- Ensures write-to-read (tWTR) specification is met

Migen support

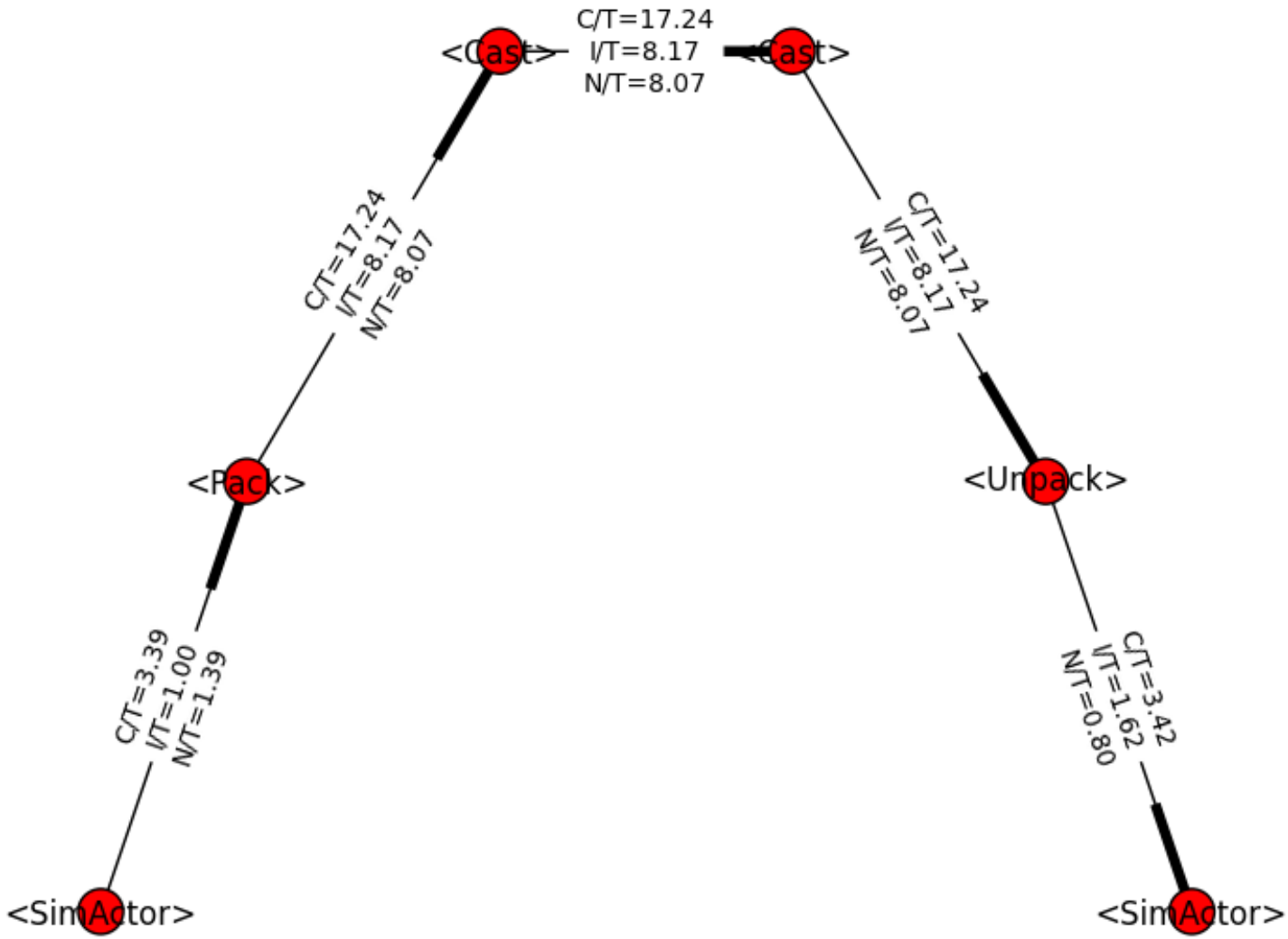
- Migen provides generic components only
- e.g. memory controller and PHY are not included
  - part of milkymist-ng[5]
  - reference design available for Milkymist One board (DDR333 + Spartan-6)
- Provides containers, bus simulation components, hub to manage requests, reorder buffer

[5] https://github.com/milkymist/milkymist-ng

Dataflow programming

- Representation of algorithms as a graph (network) of functional units
- Similar to LabVIEW or Pure Data
- Parallelizable and relatively intuitive
- Migen provides infrastructure for actors (functional units) written in FHDL
- Migen provides an actor library for arithmetic, DMA (Wishbone and ASMI), simulation, etc.
- Later: high level synthesis of actors? (e.g. c-to-verilog method)

Dataflow programming



Migen is open source!

- GPLv3 with instantiation permissions
- http://milkymist.org/3/migen.html
- http://github.com/milkymist/migen
- mailing list: http://lists.milkymist.org
- IRC: Freenode #milkymist