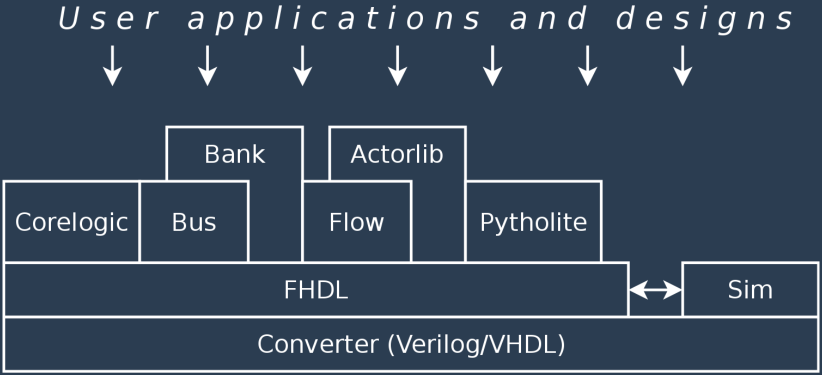
Migen

A Python toolbox for building complex digital hardware

Sébastien Bourdeaducq

2013

migen


User applications and designs
Bank
Actorlib
Corelogic
Bus
Flow
Pytholite
FHDL
Sim
Converter (Verilog/VHDL)

FHDL

- Python as a meta-language for HDL
  - Think of a `generate` statement on steroids
- Restricted to locally synchronous circuits (multiple clock domains are supported)
- 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`
    - arithmetic also supported, with user-friendly sign extension rules (à la MyHDL)
- Signals have a `eq` method that returns an assignment to that signal.
    - e.g. `x.eq((a & b) | c)`
- User gives an execution trigger (combinatorial or synchronous to some clock) to assignments, and makes them part of a `Module`.
    - Control structures (`If`, `Case`) also supported.
- Modules can be converted for synthesis or simulated.

Conversion for synthesis

- FHDL is entirely convertible to synthesizable Verilog

```
>>> from migen.fhdl.std import *
>>> from migen.fhdl import verilog
>>> counter = Signal(16)
>>> o = Signal()
>>> m = Module()
>>> m.comb += o.eq(counter == 0)
>>> m.sync += counter.eq(counter + 1)
>>> print(verilog.convert(m, ios={o}))
```

```
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 <= 1'd0;
        end else begin
                counter <= (counter + 1'd1);
        end
end

endmodule
```

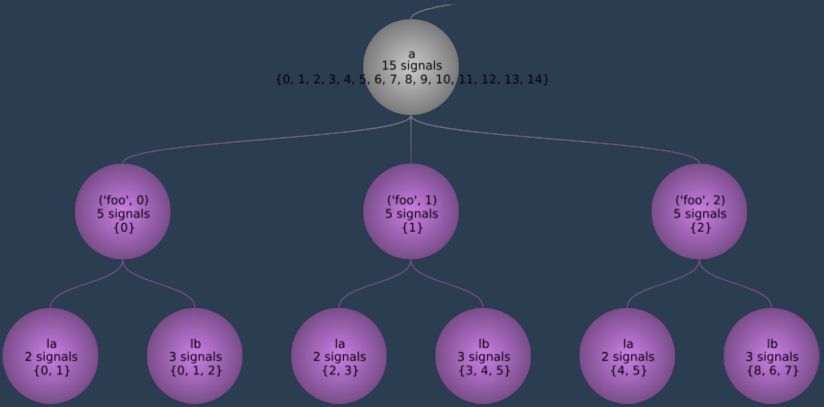
Name mangling

- Problem: how to map the structured Python `Signal` namespace to the flat Verilog namespace?
- Keep the generated code readable (e.g. for debugging or reading timing reports)
- Migen uses Python bytecode analysis and introspection to generate (often) meaningful names

Name mangling in action

```
class Foo:
    def __init__(self):
        self.la = [Signal() for x in range(2)]
        self.lb = [Signal() for x in range(3)]
a = [Foo() for x in range(3)]
```

$\rightarrow$ foo0_la0, foo0_la1, foo0_lb0, foo0_lb1, foo1_la0, ..., foo1_lb0, ..., foo2_lb2


a
15 signals
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
('foo', 0)
5 signals
{0}
('foo', 1)
5 signals
{1}
('foo', 2)
5 signals
{2}
la
2 signals
{0, 1}
lb
3 signals
{0, 1, 2}
la
2 signals
{2, 3}
lb
3 signals
{3, 4, 5}
la
2 signals
{4, 5}
lb
3 signals
{8, 6, 7}

Simulation

- Modules can have a Python functions to execute at each clock cycle during simulations.
- Simulator provide read and write methods that manipulate FHDL `Signal` objects.
- Python libraries useful for DSP testbenches: scipy, matplotlib
- Python makes it easy to run multiple simulations with different parameters (including changes of IO timings)
- Writing self-checking testbenches is straightforward: reproduciblity, reusability

Simulation

- Powerful Python features, e.g. generators:

```
def my_generator():
        for x in range(10):
                t = TWrite(x, 2*x)
                yield t
                print("Latency: " + str(t.latency))
                for delay in range(prng.randrange(0, 3)):
                        yield None # inactivity
master1 = wishbone.Initiator(my_generator())
master2 = lasmibus.Initiator(my_generator(), port)
```

Pytholite

- Some of those generators are even synthesizable :)
- Output: FSM + datapath
- Lot of room for improvement (mapping, scheduling, recognized subset)
- One application today: high-speed control of the analog RF chain of a radar

```
def generator():
        for i in range(10):
                yield TWrite(i, 0)
bus_if = wishbone.Interface()
pl = make_pytholite(generator,
        buses={"def": bus_if})
... verilog.convert(pl) ...
```

Bus support

- Wishbone[1]
- SRAM-like CSR
- DFI [2]
- LASMI

```
wishbonecon0 = wishbone.InterconnectShared(
        [cpu0.ibus, cpu0.dbus],
        [(lambda a: a[26:29] == 0, norflash0.bus),
         (lambda a: a[26:29] == 1, sram0.bus),
         (lambda a: a[26:29] == 3, minimac0.membus),
         (lambda a: a[27:29] == 2, wishbone2lasmi0.wb),
         (lambda a: a[27:29] == 3, wishbone2csr0.wb)])
```

[1] http://www.opencores.org
[2] http://www.ddr-phy.org

CSR banks

Migen user:

```
self.baudrate = CSRStorage(16)
... If(counter == 0,
  counter.eq(self.baudrate.storage))
...
```

Migen generates address decoder and register logic.
Rhino-gateware does BORPH interfacing.
→ Software user:

```
/proc/[pid]/hw/ioreg/baudrate
```

LASMI

LASMI (Lightweight Advanced System Memory Infrastructure) key ideas

- Speed is beautiful: optimize for performance
- Operate several FSMs (*bank machines*) concurrently to manage each bank
- Crossbar interconnect between masters and bank machines
- Pipelining: new requests can be issued without waiting for data. Peak IO bandwidth (minus refresh) is attainable.
- In a frequency-ratio system, issue multiple DRAM commands from different bank FSMs in a single cycle

LASMIcon (milkymist-ng)

Memory controller operates several *bank machines* in parallel

- Each bank machine uses the page mode algorithm
- Tracks open row, detects page hits
- Ensures per-bank timing specifications are met (tRP, tRCD, tWR)
- Generates DRAM-level requests (PRECHARGE, ACTIVATE, READ, WRITE)

LASMIcon (milkymist-ng)

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!!!)
- Groups writes and reads to reduce turnaround times (reordering)
  - commands stay executed in-order for each bank machine: no reorder buffer needed on the master side
- Ensures no read-to-write conflict occurs on the shared bidirectional data bus
- Ensures write-to-read (tWTR) specification is met

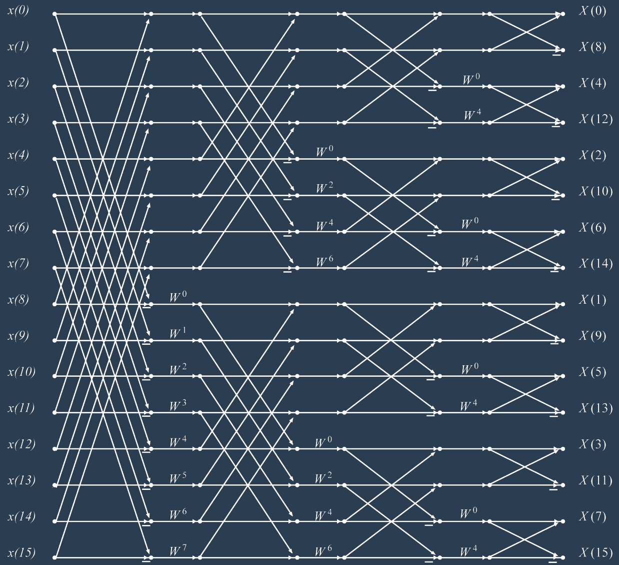
LASMIcon (milkymist-ng)

- Supports SDR, DDR, LPDDR, DDR2 and DDR3 (partial)
- Hardware tested:
  - Mixxeo digital video mixer (Spartan-6, DDR, 10Gbps)
  - Experiment control board from Paul-Drude-Institut Berlin (Spartan-6, DDR2, 4Gbps)
  - FPGA development boards: KC705 (Kintex-7, DDR3), LX9 Microboard (Spartan-6, LPDDR), Altera DE0Nano (Cyclone, SDR), ...

Dataflow programming

- Representation of algorithms as a graph (network) of functional units
- Similar to Simulink or LabVIEW
- Parallelizable and relatively intuitive
- Migen provides infrastructure for actors (functional units) written in FHDL
- Migen provides an actor library for DMA (Wishbone and LASMI), simulation, etc.

DF example: Fourier transform



Graph by C. Burrus “FFT Flowgraphs”

`http://cnx.org/content/m16352/latest/`

Mibuild

Interface between Migen and your FPGA board

```
from mibuild.platforms import roach
plat = roach.Platform()
m = YourModule(plat.request("clocks"),
  plat.request("adc"), plat.request("dac"))
plat.build_cmdline(m)
```

- Supports Xilinx: ISE and Vivado (including 7 series)
- Supports Altera: Quartus
- Runs on Linux and Windows

Current Migen users

- Mixxeo digital video mixer
- RHINO — software-defined radio
- Vermeer — radar
- NIST — trapped ion quantum computers
- Paul-Drude-Institut Berlin — experiment control
- A few semiconductor companies — ??? (fixing bugs)

Current and future works

- Dataflow system overhaul
  - Static scheduling when possible
  - Actor sharing
  - Better graph language
  - Unify with Pytholite?
- GUI
  - Build simple DF graphs more easily
  - For complex designs, Python programming is great
- Direct synthesis (Mist): Migen FHDL to EDIF netlist
  - Get rid of the Xst proprietary bloatware
  - Later: get rid of the P&R + Bitgen proprietary bloatware

Direct synthesis (Mist)

- Right now
  - basic logic operations
  - registers
  - IO
  - instantiation of pre-existing IP
- Working on
  - Arithmetic operations
  - BRAM
- To Do
  - Optimization

Migen is open source!

- BSD license
  - Compatible with proprietary designs.
  - Contributing what you can is *encouraged*.
- http://milkymist.org/3/migen.html
- http://github.com/milkymist/migen
- Mailing list: http://lists.milkymist.org
- IRC: Freenode #milkymist
- Commercial support available