Building SoCs with Migen and MiSoC

Sébastien Bourdeauducq

M-Labs Ltd, Hong Kong – http://m-labs.hk
M-Labs Limited

- Founded after Milkymist, similar to a small research institute
- Engineering contracts for physics are fun:
  - Purpose
  - Challenging problems, multidisciplinary, advanced technology
  - Often open source friendly
- Incorporated in Hong Kong in 2013, now 4 full-time staff
- Our HK office/lab contains many interesting devices (vacuum systems, cryocooler, TIG welder, ...)

Picture by Chong Kong
History of Migen

• Built Milkymist SoC in Verilog (2007-2011)
• Dataflow graphics pipeline, hardcoded
• Wanted a language for hardware dataflow
• Tried to implement on top of MyHDL, failed (2011)
• Developed Migen FHDL, based on metaprogramming
• Started implementing again on top of Migen FHDL
• Found out it was excellent for SoC, started MiSoC (2012)
• Migen dataflow is not used much these days
Basic idea: metaprogramming

- Use high level language (Python) to build code in low level language (HDL).
- Migen gives you Python objects to assemble to build your design.
- Contains hacks for syntactic sugar.
- Those objects assembled by your Python program are converted to Verilog so that third-party tools can synthesize the design.
A simple design

```verilog
da = Signal()
b = Signal()
x = Signal()
y = Signal()
module.comb += x.eq(a | b)
module.comb += _Assign(y, _Operator("+", [a, b]))
verilog.convert(module)
```
A simple design

```verilog
module top();

reg a = 1'd0;
reg b = 1'd0;
wire x;
wire y;

assign x = (a | b);
assign y = (a | b);

endmodule
```
Bus interfaces are free

class MySimpleBus:
    def __init__(self):
        self.stb = Signal()
        self.ack = Signal()
        self.we = Signal()
        self.adr = Signal(16)
        self.dat_w = Signal(16)
        self.dat_r = Signal(16)

bus = MySimpleBus()
module.comb += bus.stb.eq(...)
a = Signal()
b = Signal()
x = Signal()

# comb changed to sync
module.sync += x.eq(a | b)
verilog.convert(module)
module top(input sys_clk, input sys_rst);

reg a = 1'd0;
reg b = 1'd0;
reg x = 1'd0;

always @(posedge sys_clk) begin
    if (sys_rst) begin
        x <= 1'd0;
    end else begin
        x <= (a | b);
    end
end
endmodule
Finite state machines (FSMs)

```python
fsm = FSM()
fsm.act("IDLE",
    foo.eq(a & b),
    If(start_munging, NextState("MUNGING"))
)
fsm.act("MUNGING",
    foo.eq(c),
    If(back, NextState("IDLE"))
)
```
fsm = FSM()
fsm.act("IDLE",
    foo.eq(a & b),
    If(start_munging, NextState("MUNGING"))
)
fsm.act("MUNGING",
    foo.eq(c),
    If(load_one, NextValue(a, 1)),
    If(load_two, NextValue(a, 2)),
    If(inc, NextValue(b, b+1)),
    If(back, NextState("IDLE"))
)
The FSM module is not magical

It is implemented using regular Python and Migen FHDL

Memorizes all user actions (act calls), then finalization step issues FHDL calls. In that step it:

1. looks at all the states the user has referenced, encodes them, generates state register and next state signal
2. replaces NextState with assignments to the next state signal
3. looks at all uses of NextValue, generate load logic, replaces NextValue with assignments to load enable signals
4. generates combinatorial case statement on state with logic from the act calls (after replacements)

Read the source: `migen/genlib/fsm.py`
Bus decoding/arbitration

```python
cpu = LM32(...)
dma_engine = MungeAccelerator(...)
sdram = SDRAMController(...)
bus = BusCrossbar(
    # initiators
    [cpu.ibus, cpu.dbus, dma_engine.initiator],
    # targets
    [(0x10000000, sdram.bus),
     (0xc0000000, dma_engine.control)]
)

Again no magic - BusCrossbar is regular Python/FHDL
```
class MyCoolPeripheral(AutoCSR, Module):
    def __init__(self):
        self.enable = CSRStorage()
        self.fifo_level = CSRStatus(32)
        ...
        If(self.enable.storage, ...)
        ...
        self.comb += self.fifo_level.status.eq(...)

CSR* get automatic address assignment, generation of bus
interface logic, generation of C header file.
from migen import *
from migen.build.platforms import m1

plat = m1.Platform()
led = plat.request("user_led")

m = Module()
counter = Signal(26)
m.comb += led.eq(counter[25])
m.sync += counter.eq(counter + 1)

plat.build(m)

Runs synthesis+PAR (ISE/Quartus/Lattice\(^1\), Linux/Windows) and generates bitstream file. You may use e.g. OpenOCD for loading.

\(^1\)There is partial support for Yosys, but no one is testing it.
def foo():
    for i in range(10):
        yield 10*i

x = foo()
print(next(x))  # 0
print(next(x))  # 10
print(next(x))  # 20
print(next(x))  # 30
...

Concurrency with generators

```python
def foo(n):
    for i in range(10):
        print(n*i)
        yield
x = foo(100)
y = foo(1000)
next(x)  # 0
next(y)  # 0
next(x)  # 100
next(y)  # 1000
next(x)  # 200
next(y)  # 2000
```
Simulation

Yield statement used to synchronize generators to the clock tick

def munge1(dut):
    # ...manipulate signals in cycle 0...
    yield
    # ...manipulate signals in cycle 1...
    yield
    # ...manipulate signals in cycle 2...

def munge2(dut):
    # ...manipulate signals in cycle 0...
    yield
    # ...manipulate signals in cycle 1...

dut = DUT()
run_simulation(dut, {munge1(dut), munge2(dut)})
Maintaining determinism

- The result of a simulation must not depend on the order that the simulator chooses to restart the generators.
- Semantics of signal transactions provide this:
  - reads happen \textit{before} the clock tick
  - writes happen \textit{after} the clock tick
- This is similar to the semantics of the non-blocking assignment \((a <= b)\) in Verilog.
- This is also why careless use of the blocking assignment \((a = b)\) causes obscure simulation bugs.
  - Xilinx application notes are brimming with such bugs.
- VHDL users: non-blocking assignment \(=\) assignment to a signal, blocking assignment \(=\) assignment to a variable. Restricted scope of variables prevents those bugs.
class MySimpleBus:
    ...
    def read(self, address):
        ...
        yield
        ...
    def write(self, address, data):
        ...
        yield
        ...

def my_test(dut):
    yield from dut.bus.write(0x02, 0x1234)
x = yield from dut.bus.read(0x04)
assert x == 0x5678
MiSoC

- Provides high level classes for bus interconnect and MMIO:
  - Wishbone
  - CSR (as above)
  - streaming (ex-dataflow) interfaces

- Provides many cores:
  - Processors (wrapped Verilog): LM32, mor1kx (a better OpenRISC)
  - SDRAM controllers and PHYs (SDR, DDR1-3, fastest open source DDR3 controller @64Gbps)
  - UART, timer, SPI, 10/100/1000 Ethernet
  - VGA/DVI/HDMI framebuffer, DVI/HDMI sampler
MiSoC

- Provides bare-metal software (bootloader, low-level libraries) for your SoC.
- Provides SoC integration template classes.
- Provides basic and extensible SoC ports to FPGA boards.
- If those do not fit you, you can import the cores only and integrate yourself.
Installing Migen/MiSoC

- Known to run on Linux and Windows
- Requires Python 3.3+
- Migen and MiSoC are regular Python packages (setuptools)
- We also provide Anaconda packages
- C compiler for SoC (GCC or Clang) must be installed separately
After Migen/MiSoC are installed

```
python3 -m misoc.targets.kc705
    [--cpu-type lm32/or1k]
```

- Creates `misoc_basesoc_kc705` folder in current directory
- Builds software and bitstream there
- All compilation happens out-of-tree in that folder
- Concurrent builds supported
from migen import *
from misoc.targets import BaseSoC
from misoc.cores import gpio

class MySoC(BaseSoC):
    csr_map = {
        "my_gpio": 13,
    }
    csr_map.update(BaseSoC.csr_map)
    def __init__(self, *args, **kwargs):
        BaseSoC.__init__(self, *args, **kwargs)
        self.submodules.my_gpio = gpio.GPIOOut(Cat(
            self.platform.request("user_led", 0),
            self.platform.request("user_led", 1)))
from misoc.integration.builder import *

if __name__ == "__main__":
    Builder(MySoC()).build()

You may want to use argparse to reinstate support for CPU switching, toolchain options, etc.
LTE base station

- PCIe x1 generic SDR board (Artix7 with AD9361: 70MHz to 6GHz)
- Almost 100% Migen/MiSoC code (the only exception is the PCIe transceiver wrapper)
- Designed to be coupled together for MIMO 4x4
- With software LTE stack: allows affordable LTE BaseStation (10x cheaper than traditionnal solutions)
- > 50 boards already produced.
LTE base station

A few benefits of using Migen/MiSoC:

- Increased productivity compared with VHDL/Verilog.
- Developing a PCIe core would have been too expensive with traditional solutions, it has been done as part of this project.
- C header files that describes the hardware (registers/flags/interrupts) automatically generated.
- Kintex-7 KC705 prototyping board and Artix final board share most of the code.
SATA 1.5/3/6G core

- Connect hard drives to FPGAs, 6Gbps per drive.
- Used in research project at University of Hong Kong.
- Kintex-7 FPGA (KC705).
- All Migen, including transceiver block instantiation.

HDD picture by Evan-Amos, CC BY-SA 3.0
HDMI2USB project

- HDMI2USB: Open video capture hardware + firmware
- Created by the TimVideos project to enable every user group and conference to record and livestream
- Based around making hardware problems, software problems using FPGAs.
- Appears as a UVC webcam and CDC ACM serial port, allowing capture and control.
HDMI2USB project
Original firmware was hand coded mix of VHDL and Verilog.

- Had questionable license as used Xilinx Coregen for parts.
- Slow progress, took 2 years of development.
- Poor testing.
Conversion from VHDL/Verilog Firmware to Migen/MiSoC

Decided to attempt a rewrite based on the Migen+MiSoC

- Milkymist/Mixxeo had the similar Spartan 6 FPGA and support for most things needed - DDR, DVI/HDMI
- Funded Enjoy Digital to do the rewrite.
- Took about 4 weeks to re-implement everything apart from MJPEG core.
Conversion from VHDL/Verilog Firmware to Migen/MiSoC

New Migen+MiSoC firmware was much easier to use!

- Unambiguous, full FOSS licensing!
- VHDL/Verilog are very hard to use, Python is significantly faster to develop in. Softcore approach means much of code is C now.
- Already significantly more functionality then original firmware (Ethernet, Buffering, Multi-board support).
Numato Opsis hardware

- Firmware was originally developed on a commercial development board.
- Created our own hardware, the Numato Opsis.
- Created the hardware design in KiCad - hardware isn’t open if you can’t improve it.
- Our own hardware meant we could add new features such as DisplayPort!
- Successfully crowdfunded through CrowdSupply.
Numato Opsis hardware
ARTIQ

- ARTIQ is the Advanced Real-Time Infrastructure for Quantum physics.
- An integrated software/gateware/hardware system that controls many aspects of atomic physics experiments.
- Developed with the NIST Ion Storage Group (atomic clocks, quantum computing, quantum simulations)
- Managing/scheduling experiments, driving distributed devices, displaying/archiving results.
- Like in high-energy physics, timing is important.
at_mu(ttl_in.timestamp_mu())  # wait for input trigger
delay(1.5*us)
# first pulse precisely 1.5us after trigger
for i in range(3):
    # pulses as written, no delays from CPU/loop
ttl_out.pulse(17*ns)
delay(32*ns)

- Compromise between timing control and expressivity.
- We have developed a subset of Python with timing additions.
Implementation of the core language

- For low latency (microsecond): control loops implemented in CPU tightly coupled to IO
- For timing precision: IO connected to TDC/DTC system ("RTIO core")
- TTL IO uses SERDES and has 1ns resolution
- Other devices (e.g. DDS) can be connected at output of TDC/DTC with typ. 8ns resolution
- Python subset is processed by custom compiler (LLVM-based) and loaded dynamically into the device
Quantum Information Processor

Smart hardware to drive electrodes ("PDQ")

AD9726 DAC
AD8250 Amplifier
XC3S500E PQ208
USB Connector FT245RL
Board-to-board Interconnect

Spline interpolation in FPGA ("PDQ")

R. Bowler et al., Rev. Sci. Instrum. 84, 033108 (2013);

Migen/MiSoC advantages

- Automation, more productivity
- Portable SoC platform
- Factoring and reuse of code, e.g.
  - OOP to decouple generic SERDES-TDC logic from platform-dependent code
  - generic SoC base classes for ARTIQ core devices
- Physicists love their legacy hardware
  - We ended up supporting 4 different core devices
- Good management of different types of RTIO devices
- Lightweight
Conclusions

- Migen/MiSoC is a powerful solution to design, simulate and implement gateware
- Used successfully in several products
- Permissive open source licensing (BSD)
- A few words of warning:
  - Scarce documentation or tutorials (RTFS)
  - Some corner cases are not well handled (e.g. different directions in IO signal slice, slicing a slice)
  - No “stable” release yet (Git only), though this will change soon
Links

- Migen/MiSoC: http://m-labs.hk/gateware.html
- PCIe core: https://github.com/enjoy-digital/litepcie
- SATA 6G core: https://github.com/enjoy-digital/litesata
- HDMI2USB: https://hdmi2usb.tv
- PDQ: https://github.com/nist-ionstorage/pdq2
- ARTIQ: http://m-labs.hk/artiq