How to Build an Open Source FPGA Toolchain

Sébastien Bourdeauducq

Milkymist RTC

International Conference on Accelerator and Large Experimental Physics Control Systems

—

Open Hardware Workshop

—

October 9th 2011

Common issues

Problem

Instead of developing independent tools, you want to talk FPGA companies into opening their source code.

Common issues

Problem

Instead of developing independent tools, you want to talk FPGA companies into opening their source code.

Solution

They won't. Save your breath and start coding.

Common issues

Problem

You think that FPGA companies do not publish the information needed to write a toolchain and reverse engineering seems impossible.

Common issues

Problem

You think that FPGA companies do not publish the information needed to write a toolchain and reverse engineering seems impossible.

Solution

Find out that FPGA companies do **publish a lot of information** (netlist documentation, raw chip structure including interconnect, parts of the timing model). Look closer at the secret Xilinx bitstream format, it is surprisingly **straightforward and easy** to reverse engineer.

Common issues

Problem

You do not want to work for free for uncooperative FPGA companies. Reverse engineering seems to be a waste of time as FPGA chips may quickly become obsolete.

Common issues

Problem

You do not want to work for free for uncooperative FPGA companies. Reverse engineering seems to be a waste of time as FPGA chips may quickly become obsolete.

Solution A

Found your own FPGA startup.

Common issues

Problem

You do not want to work for free for uncooperative FPGA companies. Reverse engineering seems to be a waste of time as FPGA chips may quickly become obsolete.

Solution A

Found your own FPGA startup.

Solution B

Bite the bullet and consider that a very large part of the toolchain infrastructure can be **generic** and **portable** to many FPGA families – and hopefully to an open one someday.

Common issues

Problem

You are afraid of destroying FPGAs while developing your toolchain.

Common issues

Problem

You are afraid of destroying FPGAs while developing your toolchain.

Solution

- Risks of damaging modern FPGAs are **overrated**. Did you know that temporary internal short circuits occur even with the regular Xilinx flow?
- Test first on **cheap** FPGAs, some are easily available for less than $10.
- No risk, no fun!

Common issues

Problem

Building a FPGA toolchain seems impossibly hard and you don't know where to begin.

Common issues

Problem

Building a FPGA toolchain seems impossibly hard and you don't know where to begin.

Solution

- **Study** in detail how the existing FPGA toolchains work.
- Replace the proprietary tools **one by one** and consider **compatibility** with them.
- **Perfection is unattainable**. Accept being suboptimal in many cases (like the proprietary toolchain).
- Start with a **very crude** toolchain and improve it later.

Common issues

Problem

You know very well how FPGA toolchains work, and it still seems too large of a task for you.

Common issues

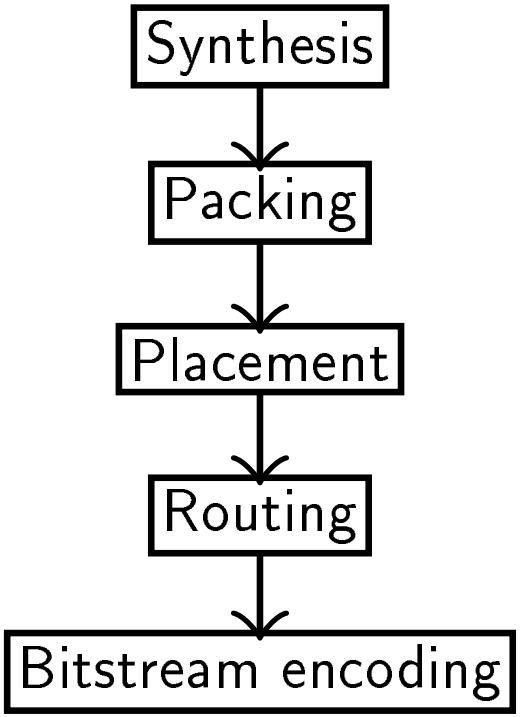
Problem

You know very well how FPGA toolchains work, and it still seems too large of a task for you.

Solution

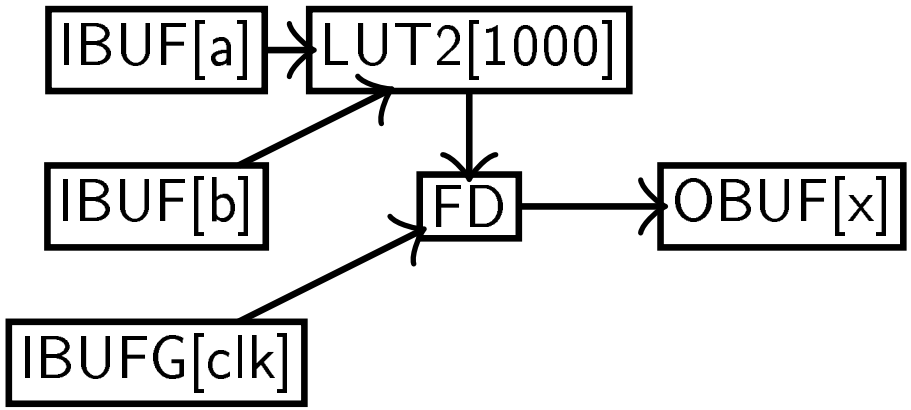
Projects like Linux and GNU have been in the same situation. Start **crude, small and modular**, and adopt an **efficient open source** development model.

The big picture



Synthesis

```
module foo(input clk,
  input a, input b,
  output x);
always @(posedge clk)
  x <= a & b;
endmodule
```



Synthesis transforms *HDL sources into a graph of *primitives*.

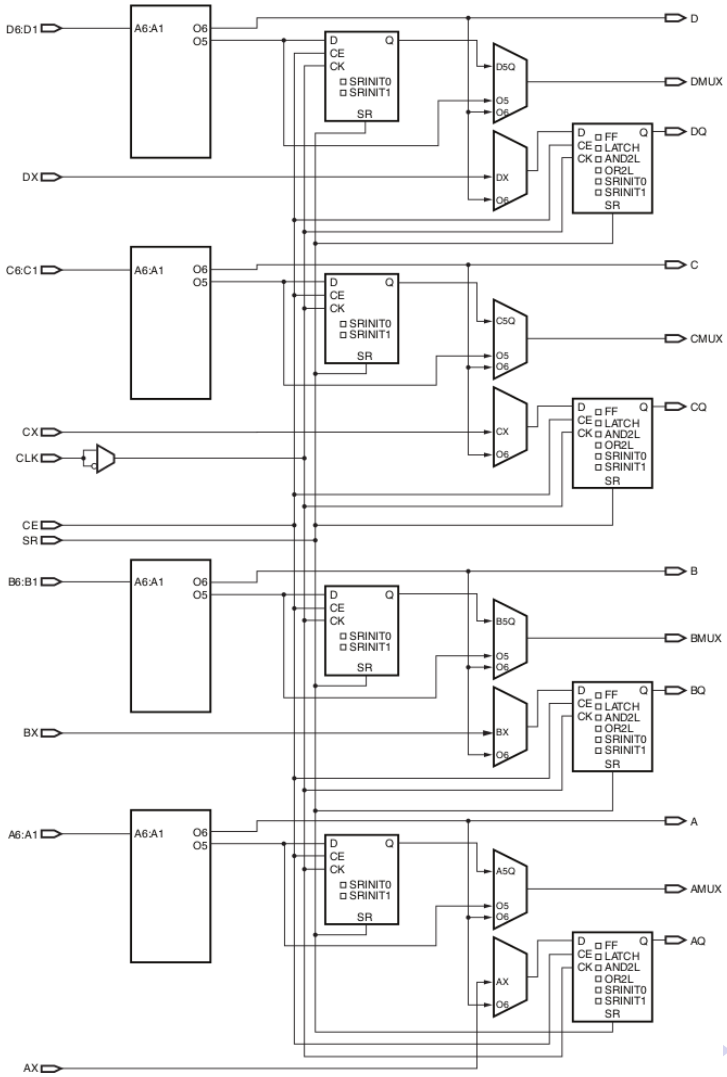
It is called the *netlist* and contains no information about the physical locations of logic and routes on the chip.

A rather transparent process

- Xst netlists in proprietary NGC format can be exported to "human-readable" EDIF (`ngc2edif`).
- EDIF netlists can be converted back to NGC and used with all Xilinx tools (`edif2ngc`).
- Verilog and VHDL netlists (instantiating primitives) can be extracted from NGC files (`netgen`).
- EDIF and NGC netlists can be viewed in PlanAhead.
- Most primitives are documented by Xilinx.
- Behavioral Verilog models are published for all primitives (UNISIM).

A synthesizer should not be harder to write than a C compiler.

Why packing?


D6:D1
A6:A1
O6
O5
D
CE
CK
SRINIT0
SRINIT1
SR
Q
D5Q
DMUX
DX
DQ
FF
LATCH
AND2L
OR2L
C6:C1
C
C5Q
CMUX
CQ
CX
CLK
B6:B1
B
B5Q
BMUX
BQ
BX
A6:A1
A
A5Q
AMUX
AQ
AX

Packing challenges

- *Control set restrictions*: some signals (clock, reset, clock enable) are shared between all basic elements (LUTs, flip-flops) in the slice.
- Primitives packed together should be chosen wisely to reduce routing delays and maximize performance.

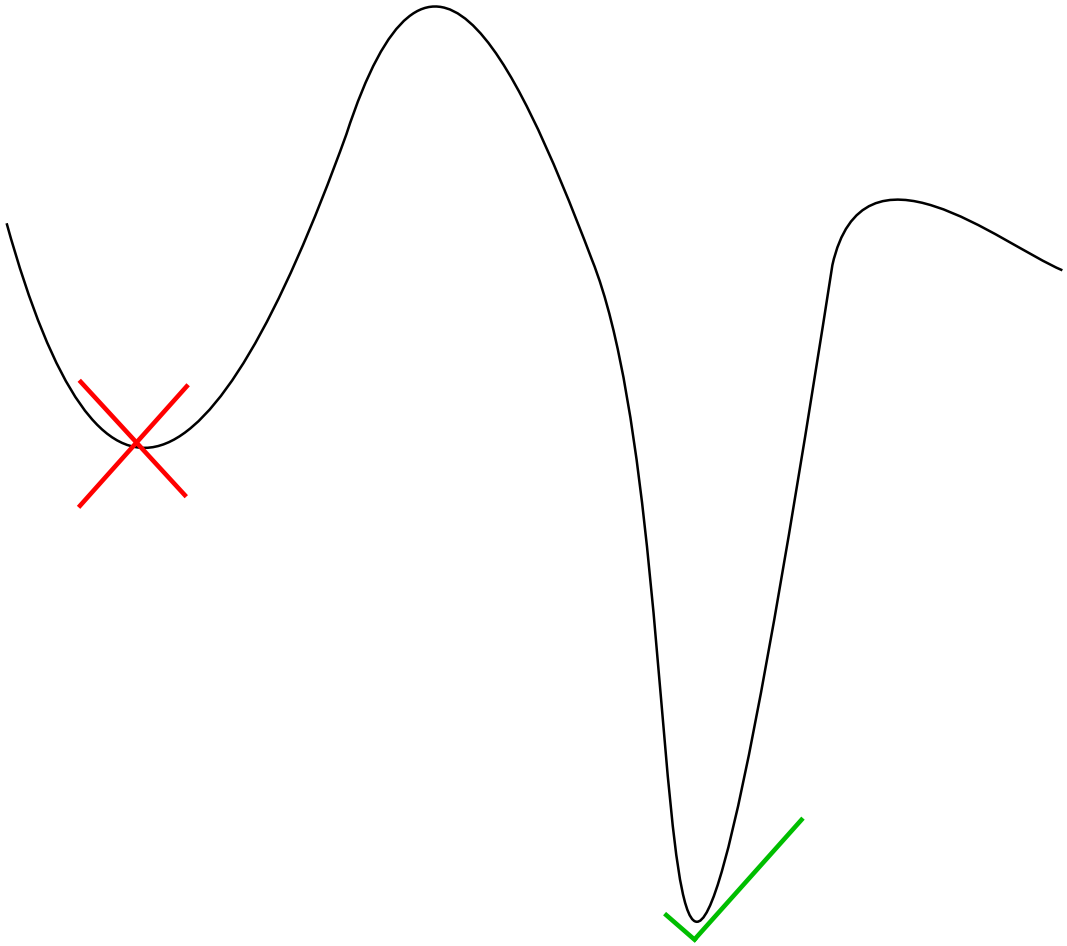
A simple packing algorithm

1. Split the input list L of unpacked LUT/FFs into lists $L_1 ... L_N$, according to control sets.
2. While at least one of $L_1 ... L_N$ is not empty:
   1. Pick a random LUT/FF from a random list L_x and pack it in a new slice S.
   2. While S is not full and L_x is not empty:
      1. For each LUT/FF in L_x, count the number of inputs it has in common with the LUT/FFs already packed in S.
      2. Pick the LUT/FF with the most common inputs and pack it in S. (NB: If there are too few common inputs, we may want to start a new slice instead)
3. Return the graph of interconnected slices.

Placement

- The next step is to place the different *sites* (slices, ...) on suitable locations on the chip.
  - For Xilinx, the detailed list of sites is available with `xdl -report <chip>` and FPGA Editor.
- Try to minimize routing length/delay.
- Since we do not know the routing yet, we have to use a heuristic.
  - Example: sum of the Manhattan distances between connected endpoints.
- First approach: place sites arbitrarily, then try to improve incrementally (hillclimbing).

The problem with hillclimbing



Simulated annealing

- Allow some moves that **increase cost**, to be capable of **leaving local minima**.
- Progressively reduce the probability of acceptance of cost-increasing moves.
- Analogy with annealing in metallurgy: reduction of crystal defects by slow cooling.
- The parameter that controls the acceptance of cost-increasing moves is called *temperature*.
- $P_{accepted} = e^{-\frac{\Delta cost}{T}}$ if $\Delta cost > 0$, $P_{accepted} = 1$ otherwise.

A simple placement algorithm

- Place all sites randomly.
- While the solution is not good enough:
  - Pick a random move and compute its cost variation (i.e. difference in the heuristic function to be optimized).
  - If $P_{accepted}(\Delta cost, T) > random(0, 1)$, accept the move.
  - Update the temperature T.

For more details, see for example VPR.

Routing overview

- Purpose: establish the connections between the placed sites.
- **One output** pin of a site drives **one or several input** pins.
- Chips contain a network of *wires* and unidirectional switches called *PIPs (Programmable Interconnect Points)* can connect two wires together.
- Site I/Os are permanently attached to special *pinwires* (on which PIPs are present to continue routing).

Understanding the switch network

- All PIPs and wires can be listed with
  `xdl -report -pips <chip>`
- Very large text output (1GB for medium-small FPGAs).
- Fully expanded description of a quite **regular** architecture.
- Many switching **structures have equivalent** PIP/wire graphs.
- Need to **identify all types** of switching structures present in a FPGA family:
  - smaller "factored" database
  - faster tool runtime, less memory utilization
  - enables interconnect timing models to be built

A simple routing algorithm

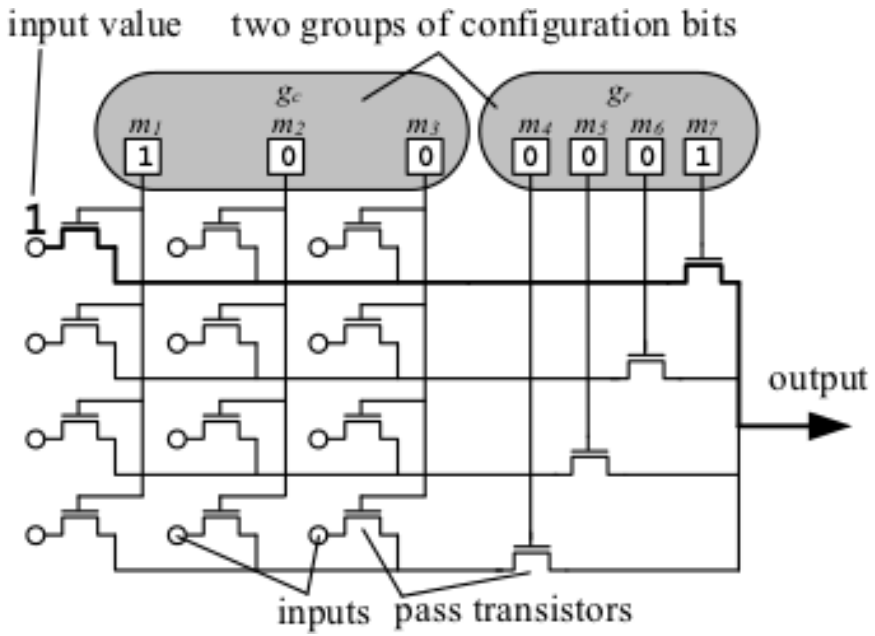
1. $P_s = pinwire(source); P_d = pinwires(destinations);$
2. $R = \{P_s\}$
3. While $P_d \not\subset R$:
   1. For all wires w in R, for all PIPs p departing from w with $destination(p) \notin R$ and $destination(p)$ is available:
      1. $R = R \cup \{destination(p)\}$
      2. $H[destination(p)] = p$
4. $L = \{\}$
5. For all pinwires d in P_d:
   1. $c = d$; while $c \neq P_s$:
      1. $L = L \cup \{H[c]\}$
      2. $c = source(H[c])$
6. Return the set of PIPs to use L.

Bitstream format

- General structure is regular and sparsely documented.
- Site configuration encoding:
  - Straightforward black-box reverse engineering.
  - Change with FPGA Editor or `xdl`, examine difference in output.
- Interconnect configuration encoding:
  - DRC won't let you turn on individual PIPs.
  - Crack software to remove this restriction (if possible).
  - Or use set operations to isolate bits (Debit method).

PIP to bit mapping

Simple basic rule: each wire is driven by a single multiplexer controlled by a 1-hot, 2-hot or 3-hot word.



ODIN II, ABC

Academic tools for synthesis, open source licenses.
Only target theoretic architectures.
`http://www.eecg.toronto.edu/vtr/`

VPR

Academic tool for packing, placement and routing. Non-free, but can serve as inspiration.
Only targets theoretic architectures.
`http://www.eecg.utoronto.ca/vpr/`

RapidSmith

Academic software similar to FPGA Editor. Open source but written in Java.
Based on XDL and can be used with real chip and the proprietary Xilinx bitstream generator.
`http://rapidsmith.sourceforge.net/`

Debit

A pioneering attempt at making sense of the Xilinx bitstream format.

`http://www.ulogic.org/`

ReCoBus tools

ReCoBus has made more thorough bitstream format analyses.
Software is non-free, but very interesting publications.
`http://www.recobus.de/`

Leading an open source project

Linus Torvalds's opinions are generally interesting...
http://h30565.www3.hp.com/t5/Feature-Articles/Linus-Torvalds-s-Lessons-on-Software-Development-Management/ba-p/440

Conclusion

An open source FPGA toolchain is possible, it just needs some work.

These slides online and more:
`http://www.milkymist.org/fpgatools`
Thank you for your attention!

Going further

- Synthesis
  - High performance LUT mappers
  - Resource sharing
  - Retiming
  - FSM extraction
  - Memory extraction
  - Carry chains (arithmetic, comparators, ...)
  - Multiplier blocks
- Physical implementation
  - Timing model
  - Timing-driven routing
  - Congestion management
  - Post-placement packing
  - Analytical global placers
  - Relative placement constraints (e.g. carry chains)
  - Floorplanning
  - Partial reconfiguration

Let's worry about those later...