From the bitstream to the netlist

Jean-Baptiste Note
Département d'informatique
École Normale Supérieure
45, rue d'Ulm, 75005 Paris
jean-baptiste.note@ens.fr

Éric Rannaud
Département d'informatique
École Normale Supérieure
45, rue d'Ulm, 75005 Paris
eric.rannaud@ens.fr

ABSTRACT

We present an in-depth analysis of the Xilinx bitstream format. The information gathered in this paper allows bitstream compilation and decompilation. While not actually compromising current bitstream security, the easiness of the decompilation process should raise awareness about bitstream security issues.

Available documentation from Xilinx and some custom assumptions about the bitstream format are presented and analyzed, so as to first gather a database mapping bitstream data to its related netlist elements, thanks to a suitable algorithm applied to a well-chosen bitstream.

This database is then used as input to an efficient program which can compile a bitstream from a low-level textual description or conversely decompile a bitstream to the same textual description for *any* subsequent input.

The whole process of database gathering and the decompilation of the bitstream format for a particular chip runs at about the speed of `bitgen` compilation. The sole process of compiling/decompiling a bitstream from/to its associated textual description runs two orders of magnitude faster.

Categories and Subject Descriptors

E.4 [**Data**]: Coding and Information Theory—*Nonsecret encoding schemes*; I.5.4 [**Pattern Recognition**]: Applications

General Terms

Algorithms, Documentation

Keywords

bitstream format

1. INTRODUCTION

1.1 Bitstream decompilation

In the software domain, great tools such as the GCC – tools which were once deemed too complex to be correctly implemented outside of the severe precincts of big companies – would never have come to life if microprocessor vendors had kept details about their instruction set architecture hidden.

In the hardware domain, lack of information from the vendors about the details of their bitstream format and overall FPGA architecture is still an impediment to research; likewise, it stifles wider adoption of FPGAs and cripples novel uses of reconfigurable computing.

It is widely believed that the analysis of bitstream formats is a daunting task. In fact, it is surprisingly easy. This paper presents a methodological approach to bitstream reversing illustrated with a case study on the Virtex FPGA lines from Xilinx.

Given a few assumptions about the bitstream format, we show in details how bitstream decompilation becomes both very simple and very fast. These assumptions are verified for Xilinx bitstreams. The paper is backed by proof-of-concept software [1] which can extract textual configuration information from the bitstreams used to program Xilinx's Virtex-II, Virtex-4 LXT and Virtex-5 LXT FPGAs.

1.2 Bitstream security

The full internal configuration state of SRAM-based programmable logic devices is contained in a programming file called a bitstream. It is commonly loaded from an external storage device on each power-up of the PLD, as the SRAM holding the configuration state inside the PLD is volatile. Therefore, it is often possible for an hostile end-user to intercept the bitstream during its transfer across the PCB.

Netlist security is of paramount importance for most FPGA hardware users. The programming file of the FPGA contains the netlist information, so FPGA vendors offer strong cryptography options for protecting bitstream contents. However, strong crypto support is only available on high-end FPGAs, and some implementations require the use of an external battery for holding the cryptographic keys [2].

Many real-world embedded projects cannot afford these very large, expensive and power-hungry FPGAs with built-in encryption. Furthermore, critical in-field projects cannot run

the risk of an external battery, which adds a sensitive single point of failure to the hardware.

For these users, the confidentiality of the netlist relies entirely upon the intricacies of the bitstream format with possibly some ad-hoc concealing layers added by the users themselves [3]. Security through obscurity schemes are known to be theoretically and practically flawed [4]. However, it is generally assumed that the reverse-engineering effort needed to get back to the netlist from a compiled bitstream is non-trivial for FPGAs [5].

1.3 Scope of the work

Bitstream decompilation has long been known to be possible. NeoCad provided alternative design tools for Xilinx chips. ClearLogic was able to break into Altera's bitstream format. To the best of our knowledge, it is however the first time that a very simple algorithm is put forward which realizes the decompilation task so simply.

We believe this is achieved without significantly compromising the security of any current in-field design. As will be shown in section 5.3.1, the textual representation obtained by a run of the software backing this paper is not a true netlist. Furthermore, assuming that the netlist stage could be reached, higher-level analysis tools would be required to actually structure and make sense of the data.

The first aim of this work is to hint at FPGA vendors that, with proper security features available, they have everything to gain from openly releasing bitstream format specifications. More specifically, much more efficient software could certainly be developed outside of vendors' scope. Section 5.3.2 demonstrates that `bitgen`-like functionality could be much faster with a clean, optimized reimplementation freed of legacy code.

We believe that open-source third party tools may be achievable without vendor support – `debit`'s support for the latest Virtex-5 architecture tends to back this claim.

The second aim of this work is to raise awareness among FPGA users about the necessity to take security issues seriously. Vendors do provide extensive solution to secure bitstreams. Strong crypto has been mentioned; it is not the only option: Spartan3-AN line of FPGAs from Xilinx allows users of low-cost chips to avoid exposing their bitstream.

Lastly, we hope this work may be of use to academic researchers exploring novel FPGA architectures by revealing some internal structural details of current commercial FPGAs.

1.4 Paper structure

This paper is organized as follows. Section 2 reviews the Xilinx toolflow. Section 3 summarizes some important information about the bitstream file format. Based on this, section 4 shows the set-theoretic algorithm used to gather a database mapping the bitstream bits to their functional role. Section 5 goes into implementation details about how to take advantage of this database to actually produce a netlist description from any bitstream, thus effectively decompiling a bitstream. Section 6 takes a brief look at possible outcomes for this work and concludes.

2. XILINX TOOLCHAIN

2.1 Xilinx workflow

FPGA bitstream generation is currently only handled by proprietary vendor tools due to the closed nature of the bitstream format. In the Xilinx case, going from a netlist description of the circuit to the bitstream which embodies the netlist is a long process which involves various programs.

Figure 1 shows how to get from an EDIF file to the final bitstream (.bit file) which will be actually loaded into the FPGA. The software proceeds as follows:

- it converts the EDIF file to Xilinx's internal netlist format, the ngd file, through `ngdbuild`. It can do the same for a wide range of input netlist formats (e.g. HDLs);
- then it performs technology mapping of the .ngd file to a .ncd file through the `map` program. After this step, the compilation process knows precisely the logic primitives that will be used by the synthesized netlist;
- the next step is to place these primitives to actual physical locations onto the FPGA and configure the crossbars to correctly route the signals between primitives. This is achieved through the `par` program;
- the last step is to translate the fully-annotated .ncd file into the actual configuration data. The `bitgen` tool handles this mapping.

Once place and route has run successfully, the .ncd file contains logic configuration directives along with placement and routing annotations which completely describe the internal state of the FPGA. The action of `bitgen` is thus simply to encode into the bitstream a subset of the information available in the .ncd (it drops, for instance, the names of the wires).

The hard problem faced while reverse-engineering bitstreams is that most of these file formats are proprietary; furthermore, the most important of these processing steps, place and route, is very slow and not deterministic, making it difficult to reliably generate slightly altered bitstreams and observe incremental changes in order to infer the function of the altered locations.

2.2 Xilinx XDL file format

Xilinx does provide a window into these closed formats, in the form of the .xdl file.

The XDL file format is a clear-text representation of the ncd file: .xdl and .ncd files can be converted back and forth with the `xdl` program in recent versions[1] of the Xilinx toolchain. The designer can get access to a very low-level description of the FPGA's internal state through .xdl.

[1] using the `-ncd2xdl` and `-xdl2ncd` flags respectively

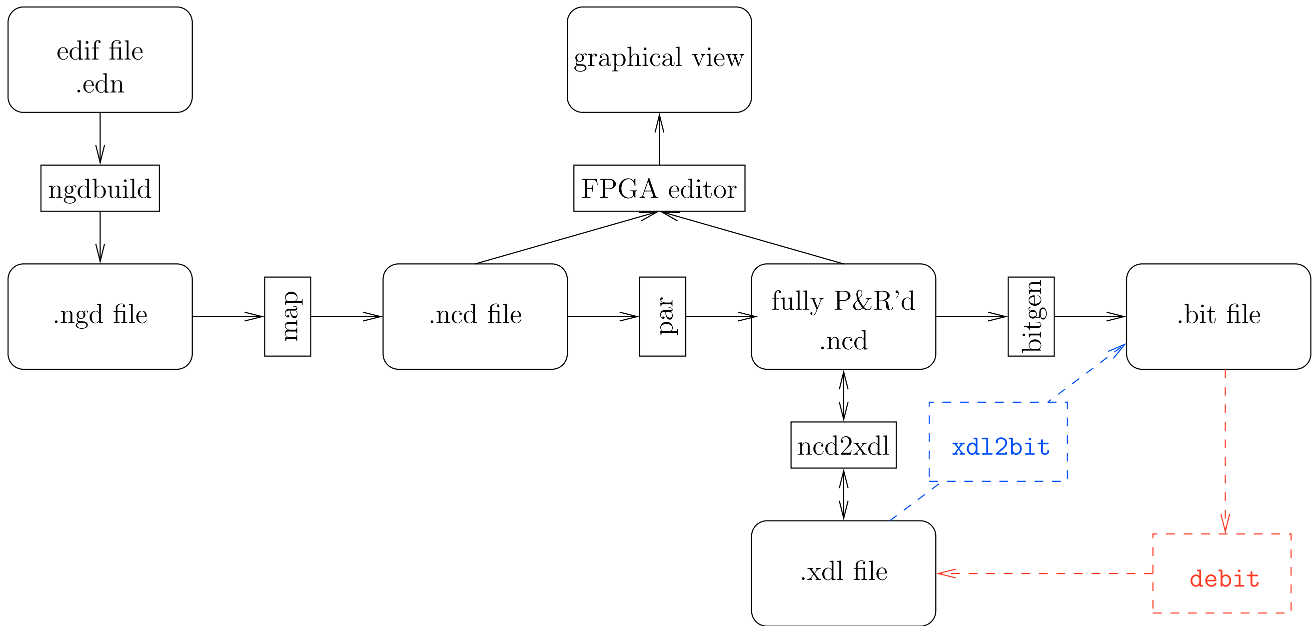

Figure 1: Xilinx tool flow

The basic idea of our reverse-engineering is to leverage the plain-text configuration data description available in the .xdl file in order to gain insight into the associated bitstream data. Therefore we need to understand in details the information available in the .xdl file.

The .xdl file format is not officially documented by Xilinx. However, the format is sufficiently simple that it can be understood directly. Additionally, any .xdl file generated by `xdl -xdl2ncd` contains comments which will help the user in interpreting the file. The following is a summary of the information contained in the .xdl.

2.2.1 Logic configuration information

Every configured slice in the FPGA has a corresponding instance in the .xdl file. This instance contains the various configurable elements of the slice, namely:

- the statically-configured muxes in the slice;
- the statically-configured inverters in the slice;
- the initial F & G LUT configuration data, if present;
- the configuration parameters for LUTs, registers, and possibly IO pads.

The slice description additionally contains information about wires going in and out of the slice through references to their name.

2.2.2 Wiring configuration information

The wires interconnecting the logic blocks above are described through nets. The FPGA contains static wires embedded in its fabric; the only configurable part in the routing infrastructure are *connexions* between these static wires. They are called PIPs according to Xilinx terminology, which stands for programmable interconnect points. These are logically regrouped in *switch-boxes* at the CLB level. Static wires, switch-boxes and PIPs can be seen and the latter individually configured with Xilinx's FPGA editor tool.

In the .xdl file, nets are described as the set of PIPs the electrical signal must go through to reach the outputs of the net from its input.

PIPs are *oriented* connections between wires: the electric signal of the net goes from the start wire of the PIP to the its end wire.

Each pip p_L is therefore described as a triplet which contains:

- the PIP's switch-box's location L;
- its start wire $S = S(p_L)$;
- its end wire $E = E(p_L)$;

and is written in the .xdl file as

```
pip L S -> E;
```

Wires are *locally* named, which means that wire names have to be prefixed by their site location to get a global meaning. As an example, the third south hex wire starts at the CLB numbered R4C5, then spans 6 CLBs to reach the R10C5 site. The wire is named S6BEG2 at the R4C5 site and is referred to as S6END2 at the R10C5 site. A net going through this wire will contain at least the following two PIPs:

```
pip R4C5  WIRE_X -> S6BEG2
pip R10C5 S6END2 -> WIRE_Y
```

2.3 Summary

The aim of the reverse-engineering effort is to find the function mapping the XDL information to the bitstream contents along with its reverse function, which will allow us to get back from the bitstream to an XDL-level representation of the netlist. Relating logic and PIP configuration settings to values in the bitstream is the step needed to achieve this goal.

3. BITSTREAM FORMAT

The bitstream format itself is not as obscure as may seem. Official documentation from Xilinx, such as the user guides [6, 7, 8], already provides significant information on the overall structure of the file format.

3.1 Bitstream loading state machine

The bitstream itself is actually microcode for a simple state machine inside the FPGA. It controls the many aspects of configuration data handling: integrity checking, initial and live loading, read-back. It also controls high-level boot-up operations.

The configuration data is loaded into the FPGA chip by blocks of data called "frames" in Xilinx's terminology. The first step in parsing a bitstream is extracting the raw configuration frames.

In fact, only a small fraction of the micro-controller specification needs to be implemented for successfully dumping frames from standard bitstreams. Our implementation fits in about one thousand lines of C code. It is capable of handling uncompressed, compressed and debug bitstreams, it also checks the embedded configuration data's CRCs.

3.2 Frame array and FAR decoding

The Frame Address Register (FAR) of the micro-controller directs each frame to a precise location in the configuration SRAM. The FAR value encodes the kind of site being configured, its location on the FPGA, and the ordinal position of the frame in the series configuring the site. Again, the encoding of the FAR value is no secret and thoroughly described in Xilinx's "configuration guides" documents [6, 9, 10].

However, in standard bitstreams the FAR's value is not encoded in the bitstream for each frame. In continuous data loading mode, the FAR is automatically incremented from one frame to the next, and its actual value is implicit. Though Xilinx clearly documents this "FAR auto-increment" feature, the sequence of values run through by the FAR is not documented.

The sequence of FAR values must be inferred from a bitstream in debug mode (`bitgen -g DebugBitstream:Yes`). In such bitstreams, frames are loaded in FAR auto-increment order, and the bitstream displays the FAR value for each frame in the bitstream. It is therefore very easy to understand the FAR's auto-incrementation logic.

Once the FAR's evolution has been understood, the program is able to store the configuration frame into a well-organized array indexed by the FAR value of each frame. The `-framedump` flag of the `debit` tool does indexed frame dumping.

3.3 Site array

The next step consists in understanding how the configuration frames relate to actual physical sites on the FPGA. Once again, the same official documentation hints at the site-level organization and allows the careful reader to organize the bitstream configuration data by relating it to the physical FPGA sites it configures.

The `-sitedump` flag of the `debit` tool allows site-wise configuration data dumping.

In a standard Xilinx bitstream for the Virtex-II, there are about 32 different types of sites, as Xilinx distinguishes between border sites at top, bottom, left, right and corners. For the Virtex-4 and Virtex-5 chips, however, the situation is clearer, with far fewer site types.

From now on, we'll concentrate on the CLB sites for the Virtex-II chips. They represent the vast majority of the sites on a Virtex-II, and the ones the reader is probably most familiar with. All methods presented can be applied to other site types; as a matter of fact, `debit` [1] successfully deals with all kinds of site present in the Virtex-II chip.

3.4 Intra-site information

The next step is to look *into* chip sites. The XDL format and the FPGA editor representation both concur to show that the CLB site configuration data contains configuration bits for the four slices embedded in the CLB along with PIP configuration data for the site's switch-box.

3.4.1 LUT configuration

The position of the LUT configuration bits is already widely known. For the Virtex-II line, see for instance [11]; very simple exploration techniques can be used to infer the LUT configuration bit positions for Virtex-4 and Virtex-5 devices.

The `-lutdump` flag of the `debit` tool allows LUT contents dumping.

3.4.2 Other slice configuration elements

Other logic configuration bits as listed in 2.2.1 can be inferred either by the same exhaustive exploration techniques or by generalizing the more subtle methods described further down in section 4 for PIP mapping.

3.4.3 Switch-box configuration

The number of bits taken up by logic configuration is no more than 10% of the total number of configuration bits. The bulk of the work lies in identifying the remaining 90% of interconnect configuration bits that handle the PIPs of the switch-boxes.

4. PIP CONFIGURATION BIT MAPPING

4.1 Available data

At this point, given an .xdl file and its associated bitstream, the following information can be gathered for each CLB site L: on the one hand the configuration data for L extracted from the bitstream; and on the other hand the set of activated PIPs at L extracted from the .xdl file.

From this dataset the function which maps each PIP to its associated configuration bits at each site must be inferred. This problem is referred to as "PIP isolation" in the following paragraphs.

In other words, in order to carry out "PIP isolation", a set of pairs $(\mathcal{Q}_L, \mathcal{C}_L)$ has to be extracted from a bitstream and its associated .xdl file. For each site L, the first element $\mathcal{Q}_L$ of this pair represents the activated PIPs for L, extracted from the plain-text .xdl file, while the second element $\mathcal{C}_L$ represents the associated configuration data extracted from the bitstream for the same site L.

4.2 Coherency hypothesis

We assume that the configuration data layout is highly regular, in accordance with the highly regular structure of an FPGA. This is the most prominent hypothesis: intra-site PIP location is coherent across all CLB sites.

What this means is that:

- the set of available pips $\mathcal{Q}$ is the same in each and every CLB site L;
- the same PIP $p \in \mathcal{Q}$ when set in different sites will yield the exact same bit configuration pattern in both sites, which we write as $\mathcal{C}_p$.

Therefore, the $(\mathcal{Q}_L, \mathcal{C}_L)$ pairs of section 4.1 represent in fact a set of (element, image) pairs, $(\mathcal{Q}_L, f(\mathcal{Q}_L))$ instances of a *unique function* f which maps PIPs to their configuration data in the bitstream.

Our implementation shows this strong hypothesis to be correct *a posteriori*. It gives us the ability to cross-correlate the information gathered from all CLB sites to infer the value of the function f on individual PIPs.

4.3 Morphism hypothesis

The function f is defined from $\mathcal{P}(\mathcal{Q})$ to the set of configuration bits for CLB sites. It enjoys many simple but very useful properties.

First, the default value for the PIPs configuration bits is zero, which is translated as:

$$f(\emptyset) = \emptyset$$

Additionally, the configuration data for one site consists of the bits which are set by each enabled PIP.

$$f(A \cup B) = f(A) \cup f(B)$$

Together, these assumption simply state that f is a morphism with respect to the $\cup$ operator on sets.

Therefore, the function f can be reduced to its values on singletons, that is, isolated pips, and our problem can be expressed very easily as finding function f's values on individual PIPs; then its value on larger sets can be inferred by:

$$\forall L, \mathcal{C}_L = \bigcup_{p \in \mathcal{Q}_L} f(\{p\}) \tag{1}$$

Additionally, we reasonably assume that the continuity property holds for the $\cap$ operator, yielding:

$$f(A \cap B) \subseteq f(A) \cap f(B) \tag{2}$$

The fact that the equality may not hold stems from the fact that some bits in the configuration stream may configure several distinct PIPs. This is to be expected, for instance, for PIPs related to a multiplexor controlling a unique output wire.

As a particular case of (2), the pairs $(\mathcal{Q}_L, \mathcal{C}_L)$ of section 4.1 enjoy the following property:

$$\forall (L_1, L_2), f(\mathcal{Q}_{L_1} \cap \mathcal{Q}_{L_2}) \subseteq \mathcal{C}_{L_1} \cap \mathcal{C}_{L_2} \tag{3}$$

These assumptions are in fact verified *a posteriori* by a working implementation.

4.4 Restricted injectivity hypothesis

The hypothesis introduced here is much less important than the previous ones; it is also harder to justify *ex nihilo*. It was actually found with a much less refined version of the PIP isolation algorithm.

It has just been underlined that the function f is not an injection. Though one configuration bit is expected to be part of the image of several distinct PIPs, as in the multiplexor example of section 4.3, one may argue such collisions will only happen between PIPs sharing the same endpoint. Indeed, PIPs controlling the same wire are sure not to be set at the same site for electrical reasons; thus it makes sense to share some hardware or configuration bits between these. Conversely, sharing configuration bits for unrelated PIPs artificially reduces and complexifies the crossbar's connectivity.

Thus we assume (and experiments show that this assumption is legitimate for Xilinx bitstreams) the following property:

$$\forall (p,q) \in \mathcal{Q}^2, \mathcal{C}_p \cap \mathcal{C}_q \neq \emptyset \Rightarrow E(p) = E(q)$$

which is equivalent to:

$$\forall (p,q) \in \mathcal{Q}^2, E(p) \neq E(q) \Rightarrow \mathcal{C}_p \cap \mathcal{C}_q = \emptyset \tag{4}$$

This property will be used to yield a much more efficient algorithm when cross-correlating information between sites.

4.5 Cross-correlation algorithm

Finding an instance of a site L where p is the only PIP set would be very convenient: in this case relating p to its configuration bits in L is trivial. However, finding such an occurrence for each possible $p \in \mathcal{Q}$ is virtually impossible.

By using the coherency hypothesis above, though, we can easily compute such isolated bit values from the random $(\mathcal{Q}_L, \mathcal{C}_L)_{L \in \mathcal{L}}$ set of section 4.1. Indeed, equation (3) can be used to try and intersect all sites containing a given PIP p so as to isolate it:

$$\begin{aligned} \forall p \in \mathcal{Q}, I_1(p) &= \bigcap_{L/p \in \mathcal{Q}_L} \mathcal{C}_L \\ &\supseteq f\left(\bigcap_{L/p \in \mathcal{Q}_L} \mathcal{Q}_L \right) \end{aligned} \tag{5}$$

Hopefully $\bigcap_{L/p \in \mathcal{Q}_L} \mathcal{Q}_L = \{p\}$, then PIP p is successfully isolated. In practice, the equality holds.

The algorithm can be refined by actually using the sites that do *not* contain p, subtracting them from the configuration. However, this is only possible provided the site L_k at hand do not contain configuration bits that would interfere with the configuration bits for p. Equation (4) shows this happens whenever no PIP in L_k shares the same endpoint as p.

We get the slightly more complex expression:

$$\begin{aligned} \forall p \in \mathcal{Q}, I_2(p) &= I_1(p) \cap \bigcup_{L/E(p) \notin E(\mathcal{Q}_L)} \neg\, \mathcal{C}_L \\ &\supseteq f\left(\bigcap_{L/p \in \mathcal{Q}_L} \mathcal{Q}_L \cap \bigcup_{L/E(p) \notin E(\mathcal{Q}_L)} \neg\, \mathcal{Q}_L \right) \end{aligned} \tag{6}$$

With this new formula, we increased our chances of isolating p, that is, getting the singleton $\{p\}$ in the end, which will yield

$$I_2(p) \supseteq f(\{p\})$$

and the equality holds in practice.

4.6 Database gathering and performance

Using the method of equation (5) on a single xc2v2000 bitstream of medium size yields 70% of the PIP database in about 5 seconds.

A much better result is achieved thanks to equation (6), which allows the gathering of all available pips in the bitstream in about 10 seconds.

The program `bitisolation` available in the `debit` suite implements —albeit without any optimization— both of these methods.

5. IMPLEMENTATION

5.1 Using the database

Once the database of section 4 has been gathered, decompiling the bitstream is still not straightforward. Sticking with the multiplexor example, one can easily understand that there is no direct mapping between the configuration bit patterns and the PIP configuration. Indeed, the bit patterns for a multiplexor controlled by two bits b_1 and b_2 could be akin to:

$$\{\{b_1, b_2\}, \{b_1\}, \{b_2\}\} \tag{7}$$

The fact that $\{b_1\} \subset \{b_1, b_2\}$ means that looking at the bit patterns in order to understand which PIP is configured at the multiplexor must be done carefully, so as not to include spurious PIPs. For instance, trying a blind match would include PIPs corresponding to the $\{b_1\}$ and $\{b_2\}$ configurations when the configured PIP is actually $\{b_1, b_2\}$. In this case, the value at bit b_1 is significant whenever we look at bit b_2. This problem arises every time pips collide in the configuration space, which can only happen, according to property (4), when pips share the same endpoint.

5.2 Database structure

In order to avoid this potential problem, the PIP mapping database was implemented as a two level lookup structure. The first level is an array D indexed by the output wires E of the pips. The array element $D[E]$ contains control information about all pips having E as their endpoint, namely the set:

$$\text{Control}(E) = \bigcup_{S/S \to E \in \mathcal{Q}} f(S \to E)$$

As a consequence of property (4), these sets are disjoint:

$$E \neq F \Rightarrow \text{Control}(E) \cap \text{Control}(F) = \emptyset$$

Additionally $D[E]$ contains the second level of the database, that is, the set D_E of all possible startpoints:

$$D_E = \bigcup_{S/S \to E \in \mathcal{Q}} \{S\}$$

along with, for each of these startpoints, the values of all bits over the set Control(E). In the multiplexor example, the programs looks at the values of the bits b_1 and b_2 in all cases, and is able to distinguish between the three cases of (7).

This method allows for a very compact database, in the range of 150KB for the whole database of Virtex-II chips, which includes values for *all* Virtex-II site types.

Running through the appropriate database for each site, the

Hardware config			5-DES, 201363 PIPs		10-DES, 431545 PIPs		20-DES, 883463 PIPs	
Processor	Clock-speed	RAM	`debit`	`bitgen`	`debit`	`bitgen`	`debit`	`bitgen`
Opteron	2.2GHz	2GB	0.65s	63.54s	0.91s	101.90s	1.47s	177.08s
P-IV	1.8GHz	1GB	1.14s	166.25s	1.56s	259.58s	2.29s	464.52s
PowerPC	500MHz	256MB	3.90s	N/A	6.60s	N/A	9.96s	N/A
ARM	220MHz	64MB	18.14s	N/A	23.86s	N/A	35.73s	N/A

Table 1: Speed comparison of `debit` and `bitgen`

software implementation can display all configured pips for all sites. This is the `-pipdump` flag of the `debit` tool.

Internal FPGA architecture. The database gathered certainly sheds some light into the internal structure of the FPGA. The PIP endpoints can be divided into two categories:

- standard endpoints are actually configured by two sets which partition the Control(E) set. An active connection is described by one active bit into each of these sets, yielding exactly two active bits to control each endpoint.
- multiplexed endpoints (essentially long wires) have a more compact control, as any subset of Control(E) can be activated to control the endpoint.

This information has been collected after a cursory glance through the database. More sense can certainly be made of this data; we hope FPGA architecture researchers will be able to interpret it more thoroughly.

5.3 Comparison to existing implementations

5.3.1 Features

Bitstream reversal. The set of configured elements (pips and logic) produced by `debit` are indeed sufficient to describe the *configurable* part of the FPGA. They lack, however, many details about the *static, non-configurable* internal architecture of the FPGA to become a fully-fledged netlist. For instance, nets should be reconstructed in order to get a real netlist. This can only be done by knowing how actual wires are laid out inside the chip between sites, so as to regroup pips into interconnected sets.

Bitstream synthesis. Xilinx provides the JBits [12] and JRoute [13] tools which are able to give low-level read/write access to the bitstream of Virtex-II devices.

JBits and JRoute are, as could be expected from DARPA-funded research, very advanced tools yielding very impressive results [14]; however they are not open-source, do not provide access to Virtex-4 and Virtex-5 bitstreams, and unfortunately seem unmaintained by Xilinx.

In order to test our internal bitstream manipulation API, we wrote the `xdl2bit` tool which attempts to convert an .xdl file to its bitstream. `xdl2bit` is by no way complete, but correctly writes all data associated to routing, LUT and logic configuration in the slices to form a bitstream. The results are encouraging, and we hope to enrich the tool sufficiently to reach feature parity with JBits. We hope more exciting work will be enabled by this software – in line with what JBits once allowed.

One key feature of `xdl2bit` is its very small memory footprint. Additionally, `xdl2bit` is written in plain C, allowing it to be ported to embedded platforms. Indeed, it has successfully been run on a small ARM processor. We do not foresee any major roadblock for a port to μBlaze, which has however not yet been attempted.

5.3.2 Speed

As a speed comparison with existing tools from Xilinx, we generated a bitstream with `bitgen`, and compared this to the time taken by `debit` to yield a pseudo-XDL form of the design. The design chosen is an array of DES kernels taken from the opencores [15] repository. Results are shown, in seconds, for the implementation of 5, 10 and 20 DES cores on an xc2v8000 chip. `bitgen`[2] was instructed to disable DRC and to only generate the bitstream, while `debit`[3] itself was instructed to dump PIPs, BRAM data contents, and LUT contents.

Table 1 sums up the results; included times are user time in seconds as reported by the `time` command on a Linux platform. On low-memory platforms the actual running time of `bitgen` is much longer, as the system starts swapping heavily; quite to the contrary `debit` stays within 2MB of shared heap size. The table also presents performance results on platforms where `bitgen` is not available.

This comparison may be unfair as `bitgen` assumedly does some checking internally and carries information which `debit` simply discards. However, controlled environment such as embedded systems can do without the burden of this extraneous checking information.

6. CONCLUSION

The highly regular nature of Xilinx chips and their configuration space makes it very easy to relate the configuration space of the chip to its bitstream. This regularity, and all architectural properties used for bitstream analysis were at first only assumptions about the internal structure of Xilinx

[2]command-line is `bitgen -d -w -l`

[3]command-line is `debit --pipdumplut --lutdump --bramdump`

chips – which our work verified to be true. In itself, this is offering a deep insight into Xilinx's internal architecture. More information useful to FPGA architecture researchers can certainly be inferred from the mapping database.

From a software perspective, such a regularity has many positive aspects. In particular, it allows the implementation of very fast and low-memory footprint tools, as shown by the `debit` proof-of-concept compiler and decompiler. We hope `debit` will foster a wide range of software *à la* JBits to be built for more recent versions of Xilinx chips.

Acknowledgements. Mark Shand's deep knowledge of the Xilinx tools and Unix methodology as well as his overall benevolent guidance inspired us and framed the state of mind that allowed this work to happen.

7. REFERENCES

[1] Debit reference implementation. [Online]. Available: http://www.ulogic.org

[2] Xilinx, "Is your FPGA design secure?" May 2003. [Online]. Available: http://www.xilinx.com/publications/xcellonline

[3] T. Kean, "Secure configuration of field programmable gate arrays," in *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 2001, pp. 142–151.

[4] S. Garfinkel and G. Spafford, *Practical Unix and Internet security (2nd ed.)*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[5] T. Wollinger, J. Guajardo, and C. Paar, "Security on FPGAs: State-of-the-art implementations and attacks," *Trans. on Embedded Computing Sys.*, vol. 3, no. 3, pp. 534–574, 2004.

[6] Xilinx, "Virtex-2 Platform FPGA User Guide (UG002 version 2.0)," March 2005. [Online]. Available: http://www.xilinx.com/bvdocs/userguides/ug002.pdf

[7] ——, "Virtex-4 User Guide (UG070 version 2.0)," January 2007. [Online]. Available: http://www.xilinx.com/bvdocs/userguides/ug070.pdf

[8] ——, "Virtex-5 User Guide (UG070 version 2.1)," October 2006. [Online]. Available: http://www.xilinx.com/bvdocs/userguides/ug190.pdf

[9] ——, "Virtex-4 Configuration Guide (UG071 version 1.4)," January 2006. [Online]. Available: http://www.xilinx.com/bvdocs/userguides/ug071.pdf

[10] ——, "Virtex-5 Configuration User Guide (UG191 version 2.1)," October 2006. [Online]. Available: http://www.xilinx.com/bvdocs/userguides/ug191.pdf

[11] A. Upegui and E. Sanchez, "Evolving hardware by dynamically reconfiguring Xilinx FPGAs." in *ICES*, ser. Lecture Notes in Computer Science, J. M. Moreno, J. Madrenas, and J. Cosp, Eds., vol. 3637. Springer, 2005, pp. 56–65.

[12] S. Guccione and D. Levi, "XBI: A Java-based interface to FPGA hardware," in *SPIE Proceedings, Vol. 3526*, 1998, pp. 97–102.

[13] E. Keller, "JRoute: A run-time routing API for FPGA hardware." in *IPDPS Workshops*, ser. Lecture Notes in Computer Science, J. D. P. Rolim, Ed., vol. 1800. Springer, 2000, pp. 874–881.

[14] S. Singh and P. James-Roxby, "Lava and JBits: From HDL to bitstreams in seconds," in *IEEE Symposium on FPGAs for Custom Computing Machines*, April 2001. [Online]. Available: citeseer.ist.psu.edu/singh01lava.html

[15] The OpenCores project. [Online]. Available: http://www.opencores.org