A Complete Open Source Design Flow for Gowin FPGAs

Pepijn de Vos
Symbiotic EDA
University of Twente
Enschede, Netherlands
pepijn@symbioticeda.com

Michael Kirchhoff
Group for Computer Architecture and Embedded Systems
Technische Universität Ilmenau
Ilmenau, Germany
michael.kirchhoff@tu-ilmenau.de

Daniel Ziener
Group for Computer Architecture and Embedded Systems
Technische Universität Ilmenau
Ilmenau, Germany
daniel.ziener@tu-ilmenau.de

***Abstract*—In this paper, we propose an open source design flow for the low cost FPGAs from the company Gowin. Open source design tools are opening the door for custom extensions and modification in the design flow. The proposed design flow which supports almost all Gowin FPGA resources, is based on well-known open source tools, like Yosys and nextpnr, as well as on our proposed open source bitstream generator. The necessary architectural details of the FPGA family are gathered by input fuzzing and comparisons with the vendor tool flow. Experimental results show an almost similar performance as the vendor tools.**

***Index Terms*—Field programmable gate arrays, Reverse engineering, Documentation, Open source software**

I. Introduction

The usage of FPGA design flows based on open source tools has many advantages. First, open source tools open the door for user defined modification in the design flow. This can be used to add, for example, routing constraints which can be useful for developing partial reconfigurable designs [1] or to control exactly the delay of certain nets. Furthermore by using open source tools, everyone can check the source code for possible backdoors or for the insertion of hardware Trojans [2]. This is very important for the development of FPGA designs for security applications, e.g., encryption. Moreover, it could be very useful for the education to have a transparent design flow in which every step can be explained by practical examples. Finally, older FPGA families are often not supported anymore in current vendor tools or the license of these families expired. Using open source tools could extend the lifetime of discontinued FPGA families in order to provide updates and fixes for older products. On the other hand, designs generated by open source design tools have often a lower performance (frequency, resource utilization) as the output of the vendor design suite.

Programming an FPGA consists of several steps. First a *hardware description language* (HDL) design is synthesized to a netlist by a tools such as *Synopsys Synplify* [3] or *Yosys* [4]. This step requires information about the FPGA architecture and available primitives. So adding a new FPGA family to open source tools like Yosys is fairly straightforward in most cases. After that a place and route step is done that maps the generated netlist to the available hardware on the FPGA. To develop an open source tool, understanding of the hardware blocks and interconnections between them are required. This information is only partially published for *Gowin* FPGAs through documentation and the floorplanner, but much of the needed information needs to be derived by other means. The final step is that a binary *bitstream* file needs to be generated to program the routed design onto the FPGA. The Gowin binary format is completely undocumented, and also needs to be derived through other means.

Until now a HDL design for Gowin FPGAs can be synthesized using open source tools like Yosys [4], but the resulting netlist has to be run through the vendor place and route tools. Furthermore, the Yosys synthesis also lacks support for some advanced resources such as *ALU carry chains* and *DSP blocks*, meaning they are slower and take up more space than needed. While this allows easier code sharing between FPGA architectures, proprietary software is still needed to do *place and route* (PnR) and to program the FPGA. The open source tools for place and route and bitstream generation for Gowin FPGAs are completely missing and have to be developed by inspecting the official vendor tools.

In the proposed approach, the Gowin Yosys support is improved, a place and route target is added by using *Nextpnr* [5], and a bitstream packer is written. The resulting tools are made open source in order that every one can use and contribute to improve the tools [6].

However for a full end-to-end open source flow, the internal FPGA architecture and bitstream format need to be analyzed and documented. This is one of the main contributions of this work. For extracting the information needed to do PnR and generate a bitstream, there are several methods. First and foremost is just collecting all the information the vendor tools give you. For many vendors this includes information about routing, but in the case of Gowin, hardly any useful information is generated in the PnR phase besides the actual bitstream. Therefore, *input fuzzing* was chosen to reveal the routing configurations.

The remainder of the paper is organized as follow: Section II gives background information and presents the related work. This is followed by a brief explanation of fuzzing in Section III and the Chip DB decoding approach in Section IV. Section V

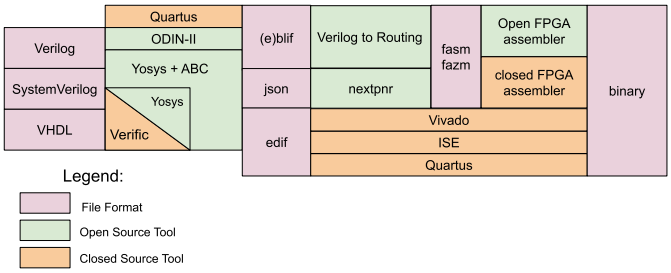

Fig. 1. An overview of different design flows with proprietary and open source tools to generate from HDL descriptions the configuration bitstream. [11]

covers all the necessary information about the used Gowin FPGA architecture. Based on this architecture Section VI discusses the place and route, especially the bitstream packer and unpacker as well as Netxtpnr. In Section VII are the experiments and the results shown. Section VIII concludes the paper and gives information about possible future work.

II. Background and Related Work

To configure an FPGA, it is needed to configure all the LUT contents, MUX inputs, and other tiles. This is done by loading a bitstream, e.g., from a flash memory into the FPGA. These bitstreams are generated by the vendor design tool chains, and their format is usually not documented. The goal of this paper is to document the bitstream format for the whole *Gowin LittleBee* family.

First a circuit description written in a *hardware description language* (HDL) such as *Verilog*, *VHDL*, *Clash* [7], *Chisel* [8], *nMigen* [9], etc. needs to be synthesized to a netlist of these basic building blocks described above. This netlist then needs to be mapped to the available LUTs, DFFs, MUXes, and other resources available on the FPGA. And finally the annotated netlist needs to be converted to an actual bitstream.

Traditionally, all the functions, from synthesizing a netlist from a hardware description language (HDL) over place and route to the final step of generating bitstreams, are performed by proprietary software. Those software is usually produced by the vendor of a specific FPGA until recently. However, open source alternatives are on the rise, providing an unified feature set across all FPGA devices, and allowing endless customization and integration.

Fig. 1 shows how the various tools interoperate with each other using various intermediate representations. On the left are various HDL file formats, which can by synthesized using Yosys [4], Quartus [10], or any other synthesis tool, producing a netlist format such as *blif*, *json*, *edif*, etc. These can then be fed to open source PnR tools such as *Nextpnr*, or proprietary ones such as *Vivado*. The open source tools then produce a low-level description of all the primitives and locations used, which can again be programmed to the FPGA using either open source or proprietary tools.

Yosys is an open source synthesis tool originally developed by Claire Wolf for the Icestorm project [12]. It can output various netlist formats for usage with simulators, formal verification, vendor PnR tools, and the open source PnR tools. In the case of Nextpnr, used in the proposed approach, a JSON format is used.

For place and route, there are different open source tools. One example is VTR [13] (verilog to routing), a research tool to design and test new theoretical FPGA architectures. It uses an XML based format to describe an FPGA. The second available option is Arachne-PnR [14]. This is a fully functional FOSS FPGA PnR tool with industrial user base, but only supports ICE40.

Another PnR tool is called Nextpnr [5], which uses programmatic descriptions on FPGA architectures, and is written from the ground up with commercial FPGAs in mind.

Those PnR tools rely on separate bitstream documentation projects that document the architecture of a class of FPGAs, and produces a so called *ChipDB* that is a machine-readable description of an FPGA. Our proposed approach is such a project, which aims to provide a ChipDB for Gowin FPGAs. Other projects include IceStorm [12] for Lattice iCE40 FPGAs, Project Trellis [15] for Lattice ECP5 FPGAs, and Project X-Ray [16] for Xilinx 7 Series FPGAs.

III. Fuzzing

The main method to extract the information about the bitstream, that was used in this paper, is fuzzing [17]. Traditional fuzzing is used as an automated software testing technique that involves providing (pseudo) random data as inputs to a computer program, like [18], [19] or [20]. With this method, it is possible to realize security testing, like in [21] and it is used even with machine learning [22].

In contrast to the widely used fuzzing methods for software testing, we propose the usage of fuzzing to extract data about the FPGA configuration from the bitstream. This means that an automated system runs the vendor PnR tools on many different generated netlists and correlates the configuration of primitives in the netlist to bits in the bitstream.

The basic idea is to take a program, generate a bitstream, change one tiny element of the program and generate another bitstream. The difference between the two bitstreams then show which bits are related to that particular feature. In these generated bitstreams, extensive constraint files are used that explicitly specify the location of every primitive in order to obtain a reproducible result every time.

There are two different approaches to fuzzing. The first one is to determine the tile grid layout, and fuzz a single tile. The other just fuzzes every bit in the whole FPGA as efficiently as possible. The latter is more complex, but requires less prior knowledge and has the added benefit that it detects unexpected interdependencies that might not show up if you turn one feature on or off at the time.

We decided to realize the second approach, where the fuzzer is of the whole-FPGA kind, and completely written in Python. So the fuzzer was defined as a python class that uses a code generation module to generate netlists on the fly. These are then saved in temporary directories where the vendor tools

TABLE I
EXAMPLE USING BINARY CODES

Run\bit	A	B	C	D	E	F	G	H	B‖C
1	0	0	0	0	1	1	1	1	0
2	0	0	1	1	0	0	1	1	1
3	0	1	0	1	0	1	0	1	1
Unique	✗	✓	✓	✓	✓	✓	✓	✗	✗

TABLE II
EXAMPLE USING HAMMING CODES

Run\bit	A	B	C	D	E	F	G	H	B‖C
1	0	0	0	0	1	1	1	1	0
2	0	1	1	1	0	0	0	0	1
3	1	0	1	1	0	1	1	1	1
4	1	1	0	1	1	0	1	1	1
5	1	1	1	0	1	1	0	0	1
Unique	✓	✓	✓	✓	✓	✓	✓	✓	✓

are run in parallel to utilize all CPU cores. The results are then read and a report of all bitstream locations is generated.

Rather than fuzzing one bit at a time, this fuzzer uses a system where N bits can be found in $\log_2(N)$ PnR runs, by changing each bit in an unique pattern. For example, a run with eight configuration bits could be done in three runs using patterns 00001111, 00110011, 01010101 as shown in TABLE I. However, in reality a fuzzer can test thousands of bits at a time, requiring about 20 PnR runs per batch to locate all of them. Multiple batches are needed to test incompatible fuzzers that use the same resources.

In the above example, the first feature would generate a bit pattern of 000 across the three runs, while the second would have the bit pattern 001, and the last one will have 111. The first and last ones are problematic because all bits that are not involved in the test are also all zero or all one. Another problem shows up with bits that are not 1:1 relations to program features. One example is I/O buffer bank enable that are set if any pin in that bank is used. These features generate different codes that may correspond to another unrelated feature.

To avoid all-zero and all-one codes and to detect codes that do not correspond 1:1 to a tested feature, the fuzzer uses constant Hamming weight codes [23] rather than straight binary. Hamming codes have a constant number of ones and zeros, which means that in most cases it can detect codes due to unexpected side effects. The same 8-bit example is displayed with Hamming codes in TABLE II. Note that it takes more runs, but the OR of bits B and C produce an unique code.

To correctly handle side-effects such as bank-enable bits on toggling an output pin, meta-fuzzers were introduced. A meta-fuzzer exposes a list of extra codes it expects given the configuration bits given to the fuzzer. It also produces groups of bits that toggle a particular extra bit, which are use to generate additional PnR runs to detect them. For example, D‖E would be 11111 and not be detected properly, so an additional run with both D and E zero needs to be generated.

This fuzzer is sufficient to find bits for most cell types, such as lookup tables, flip-flops, and I/O buffers.

However, a major challenge was hit in routing. Unlike most other vendors, the Gowin tools do not give any indication or control over which wires are in use, or even which wires exist at all. This makes fuzzing Gowin routing several orders of magnitude harder than previous efforts, despite the FPGA otherwise being relatively simple. Because of this, ChipDB decoding was used in addition to fuzzing as explained in the following Section IV.

After the ChipDB was decoded, a third fuzzer was written of the one-tile-at-a-time kind. Since the tile types and boundaries were now known, a very simple fuzzer could be written that fuzzes a single item per tile. Where the second fuzzer required complex code and long run-times, the third fuzzer is less than 200 lines of code and runs in under a minute. It also directly records results in a machine-readable database, together with information from the vendor files.

IV. CHIPDB DECODING

Another approach to obtain information about the PnR resources and bitstream format is through decoding the data files known as the chipDB that ship with the vendor tools. These files directly contain the resources, layout, and bitstream format of all FPGAs supported by then vendor.

To decode the chipDB, first GDB (*Gnu Debugger*) was used to set a catchpoint on the `openat` syscall to determine the functions where the files are read. This function was then decompiled with *Ghidra* [24] to see how it is read. From there GDB is used again. In one case data was read into a struct, and GDB watchpoints were used to connect getter methods to struct offsets. In another case a class-based archive format was used, where a GDB breakpoint on `fread` was used to connect class names to file offsets.

Once the format had been figured out, a Python parser was written to extract the data for further use. One of the first uses was to generate images in the same format as the fuzzer, so fuzzing results can be compared to chipDB results. Then other tools were written, directly using the vendor ChipDB.

Even with the decompiled parsing code, attaching meaning to the data was a big challenge. A major roadblock was that the routing information used wire IDs with unknown meanings. While some guesses could be made about their general usage, no exact routing table could be derived from the data file. Another foray into decompiled code was needed to locate the names of these wires. They were generated with some extremely hard to follow C++ code that wrote them to a vector. Eventually some long GDB command was used to extract the generated vector from the running program.

With these wire names in hand, some guesses could be made and then verified about the meaning of the data in those files. A table of wires was found, which for each wire contained the fuses to set. These fuses are indexes into a table, that combined with the tile type, give a 4-digit decimal number of the format `YYXX` which encodes the location of a bit to set relative to the top-left of the tile.

The same format was used for data tables that appear to contain info about LUTs, DFFs and IOBs. Since these results

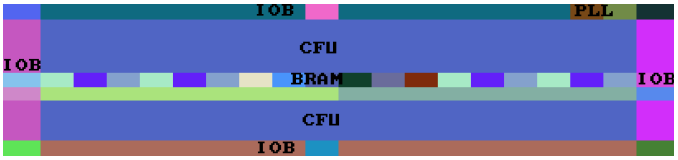

Fig. 2. Tile layout for GW1N-1 FPGA.

were previously known from fuzzing, they could be cross-checked for correctness. However, a very puzzling thing was that there were negative wire numbers and features in these tables. It was found that for wires, these indicate the default state of a MUX, so if the specified fuse is not set, then these wires are connected.

V. Gowin FPGA Architecture

Gowin FPGAs have a LUT4 architecture common to many smaller FPGA architectures. Larger LUTs, like LUT6 or LUT8, are realized with hardwired MUXes, meaning it is possible to combine for example two LUT4's into a LUT5, two LUT5's into a LUT6 and so on.

The FPGA consist of a grid of tiles with I/O buffers around the edges, rows of special-function blocks such as BRAM, and a large grid configurable logic units.

Fig. 2 gives an overview of the tiles on a small Gowin FPGA. All the tiles labeled CFU are normal logic tiles. The row below BRAM are also logic tiles, but with different connections. All edges except the PLL and corner tiles are I/O buffers, but the IOB in the centers are a bit different. One BRAM is 3 tiles wide, hence the alternating pattern. The central tiles are in fact not BRAM and are used in global clock routing and other functions.

Each Configurable logic unit consists of 8 LUT4s grouped in 4 slices, of which 3 have data flip-flops. Each slice shares certain resources such as clock inputs and reset lines.

Each LUT4 has 4 inputs and one output that is directly connected to the data flip-flop input. The LUT output can be used independently, but the flip-flop is always used through the LUT. Each pair of flip flops has data in and out, clock, clock enable, and set/reset. Each pair of flip-flops can be configured for rising edge or falling edge sensitivity, and asynchronous or synchronous set or clear.

These tiles are connected with various multiplexers to adjacent tiles as well as global clock lines. Each tile has 8 tile-local wires, and in each direction it has 4 one-hop wires of which 2 are shared between north/south and east/west, 8 two-hop wires with one-hop taps, and 4 eight-hop wires with four-hop taps. An overview of all wires can be seen in Fig. 3.

Gowin outputs a bitstream in ASCII binary notation, making it possible to separate commands form data frames.

Data frames describe one row of bits on the FPGA tile grid. Frames are padded to full bytes, and verified with a CRC-16 checksum. These rows are stacked on top of each other to describe a bitmap that is overlaid on the FPGA tile grid.

The number of tiles on the grid depend on the specific FPGA model. A tile is roughly 60×24 bits, with I/O buffers

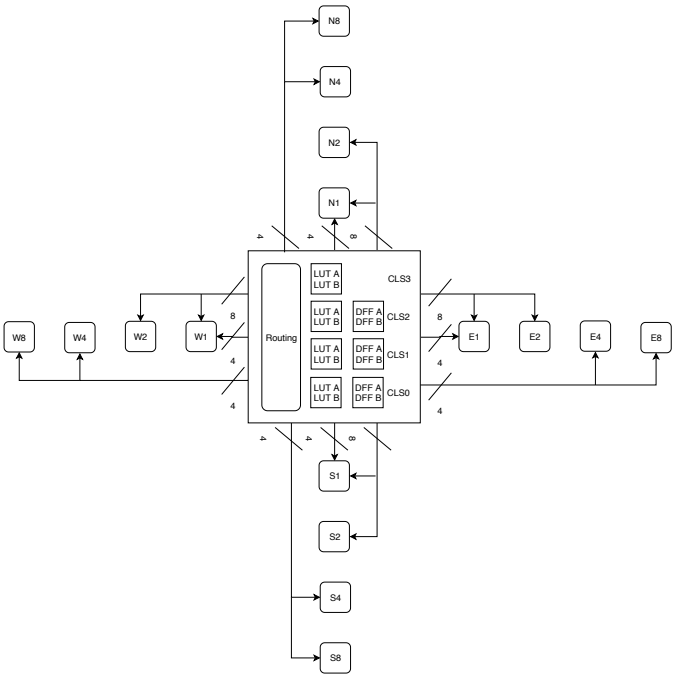

Fig. 3. Tile structure in details with wires.

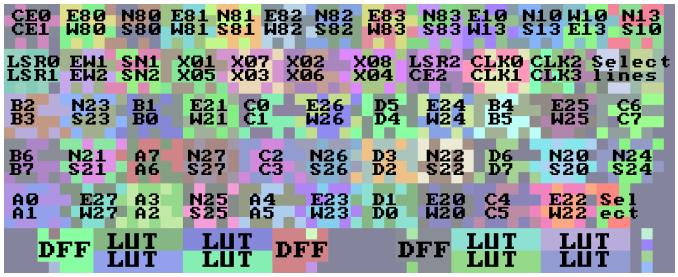

Fig. 4. Logic tile bits. Each colour corresponds to a single MUX or other primitive. MUX labels indicate the destination wire of the MUX.

and some special tiles being a few rows or columns larger. A common logic tile has the LUTs and flip-flops in the bottom 4 rows, with the top 20 rows being filled with multiplexers. An overview of the bitstream layout of LUTs, flip-flops, and multiplexers in a logic tile can be seen in Fig. 4.

VI. Place and Route & Bitstream Generation

Once the structure of the FPGA is understood, software can be written to parse and generate bitstreams.

The main building blocks that are needed are lookup tables for logic, flip-flops for registers, wires to connect primitive cells together, and I/O buffers to connect to external FPGA pins.

The first part of the Gowin bitstream tools that was written is the *unpacker*, which transforms a bitstream back to a netlist. This allows to validate our understanding of the bitstream by comparing post-synthesis netlists with unpacked netlists from bitstreams generated by the vendor PnR tools. It is also an important debugging tool for debugging the output of the open source PnR tools.

Before full-fledged support for all aspects of the FPGA are implemented, a proof of concept PnR flow was written using the Nextpnr generic target. This is a Python-based target that only supports very basic primitives, but is a much smaller time investment than a full-fledged C++ target. Using this target and the vendor data files, a simple flow can be written in a few dozen lines of Python. This flow only supports a very basic LUT4, flip-flop, and I/O buffer without any extra functions.

Once this was done, a packer was written to complement the unpacker to actually generate the bitstream from the Nextpnr output. The unpacker can be used in this process to verify that the packer output matches the original.

A. Bitstream unpacker

To confirm a correct understanding of the bitstream structure, an unpacker was written. The unpacker can be executed on a bitstream file produced by the Gowin IDE, and will then reconstruct the post-synthesis netlist from the bitstream file.

This unpacker currently only supports basic logic tiles, routing fabric, and I/O buffers, as explained in more detail later. Accordingly, the clock pin was constrained to not use a global clock net in the following design.

As an example, a simple clock divider was synthesized and routed using the Gowin IDE. The dot graph in Fig. 5 was then produced from the post-synthesis netlist using Yosys. After running the unpacker on the bitstream file produced by the Gowin IDE, Yosys was again used to produce the dot graph in Fig. 6.

Notable differences between the post-synthesis netlist and the recovered one are that a LUT4 and DFFCE are used. This is because in hardware there is no distinction between a LUT1 and a LUT4 with unused inputs (tied to arbitrary LUT outputs), and a DFFC and a DFFCE with the clock-enable tied high. Besides those differences, the netlist is reconstructed correctly.

B. Nextpnr

Nextpnr [5] presently has three targets, iCE40, ECP5, and the generic target. The final goal is to add a fourth Gowin target, implemented in C++. However, this will take thousands of lines of C++. As discussed previously, the generic target was used instead for a simple proof of concept.

The Nextpnr code includes a sample project that demonstrates how a simple imaginary FPGA could be implemented in Python. This was used as a basis for the simplified Gowin target. The major difference between the simple target and Gowin is that Gowin has much more wires and more powerful flip-flops.

This means that the reset and clock polarity features of the Gowin flip-flops are not used, and that all flip-flops within a tile share the same clock input. The generic packer packs a LUT and a DFF into a generic slice that gets mapped to the FPGA fabric, unfortunately the top two LUTs in a Gowin slice don't have a DFF, so were not used at all.

Only I/O buffer and logic tiles were used, and only in their most basic configuration. For example, buffers can also be

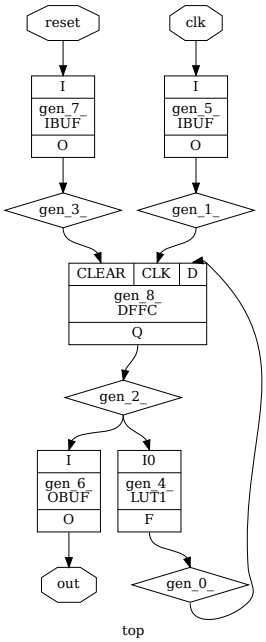

Fig. 5. Post-synthesis netlist of a simple clock divider.

configured for differential signaling, and logic tiles can also be configured as ALU or memory tiles. Other tile types such as BRAM, PLL, and DSP were not used at all.

The Python target is also extremely slow, taking dozens of seconds each run to load the data files. This is because in the C++ targets, these are C structures that are directly memory-mapped into RAM, whereas the Python target programmatically recreates the routing graph from the data files produced by the fuzzer.

To demonstrate the output of the PnR flow, a script was written that generates a netlist and post-placement file compatible with the vendor Floorplanner tool. This way the placement of tiles can be visualized. A simple blinky program used as proof proof of concept produced the output in Figure 7.

Since the PoC Nextpnr uses the generic target rather than a custom C++ target, no major contributions to Nextpnr have been made as of yet. Instead, the Python code for the generic target is published as part of Project Apicula [6].

C. Bitstream packer

The last major piece of the puzzle for an end-to-end HDL-to-bitstream flow is the bitstream packer that takes Nextpnr output and generates an FPGA bitstream. This was written in Python, reusing code from the unpacker.

Writing the packer required revisiting previously ignored details, such as the bitstream command structure and CRC verification. For the moment, an empty "donor" bitstream is used for the commands and bitstream "template", but a certain degree of understanding is required. Most importantly, each frame contains a CRC to verify the validity of the frame, but

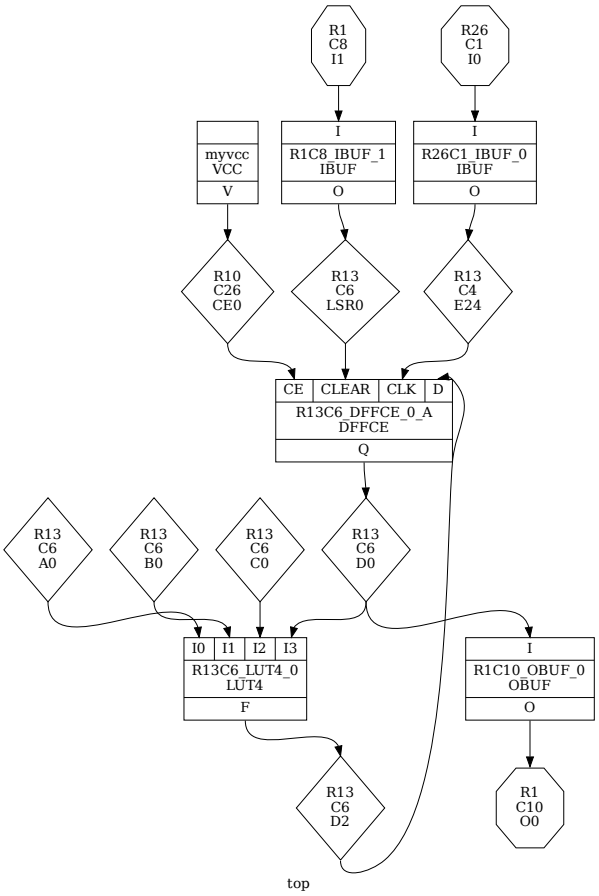

Fig. 6. Netlist unpacked from bitstream of a simple clock divider.

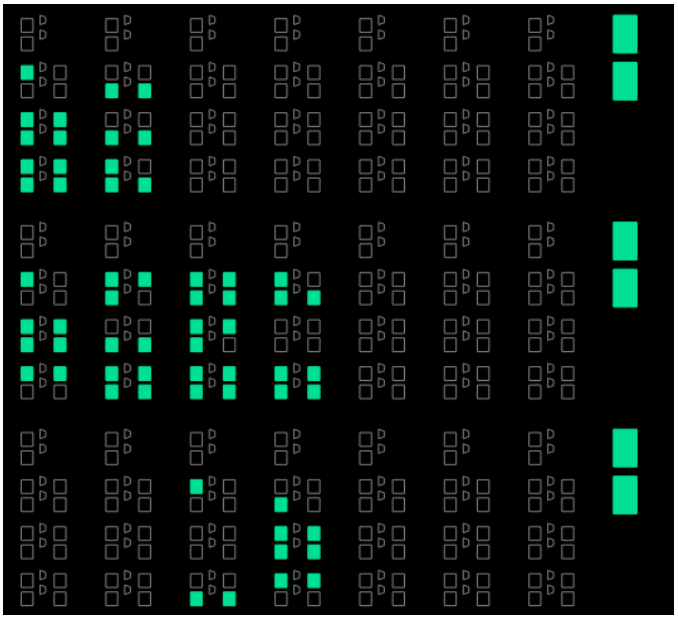

Fig. 7. Nextpnr results loaded in vendor Floorplanner tool.

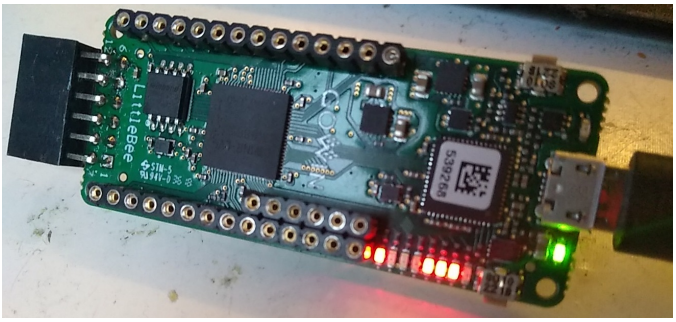

Fig. 8. First end-to-end open source blinky running on a Gowin FPGA

the first frame includes some command data. There is also a file checksum, but this appears to be ignored by the FPGA.

After verifying that a vendor bitstream can be fully decoded and re-encoded with identical results and valid CRC, work began on generating the actual frame data. Despite several bugs, this was a reasonably smooth process.

VII. Experiments and Results

In this section, the achieved end-to-end flow as well as a comparison between the achieved results and synthesis results of other flows are compared.

A. Demonstration of end-to-end flow

The resulting flow was first tested by applying constant outputs on the LEDs, then by wiring the user button directly to the LEDs, then adding an inverter gate, and finally the before mentioned blinky program. The first ever end-to-end open source blinky program running on a Gowin FPGA is shown in Fig. 8.

It was theorized that because global clock routing was not yet implemented, nontrivial designs would face severe clock skew leading to timing violations. On the other hand it was argued that these smaller, more low-power devices are a lot more forgiving than larger high-performance parts.

To put this hypothesis to the test, a RISV-V CPU was synthesized. For this task the PicoRV32 was chosen in the AttoSoC [25] configuration. This configuration is very minimal and does not have any memory besides the registers, making it suitable for the current flow that does not yet have block RAM.

For the software, a simple program was chosen that calculates all primes under 256 and displays them in binary on the LEDs of the FPGA. The whole AttoSoC design was synthesized with Yosys, using 5099 out of 6480 FPGA slices, 78% of the available resources, taking into account only 3/4 of the slices can be used in the current flow due to limitations in the `nextpnr-generic` packer.

The design was then placed and routed in Nextpnr, taking 234 placer iterations and 408985 routing interactions, taking several minutes to complete. The generated `.posp` file was then loaded in the Gowin floorplanner, shown in Fig. 9. As can be seen, almost the full FPGA is utilized by this design, but all of the DSP and BRAM blocks are left untouched.

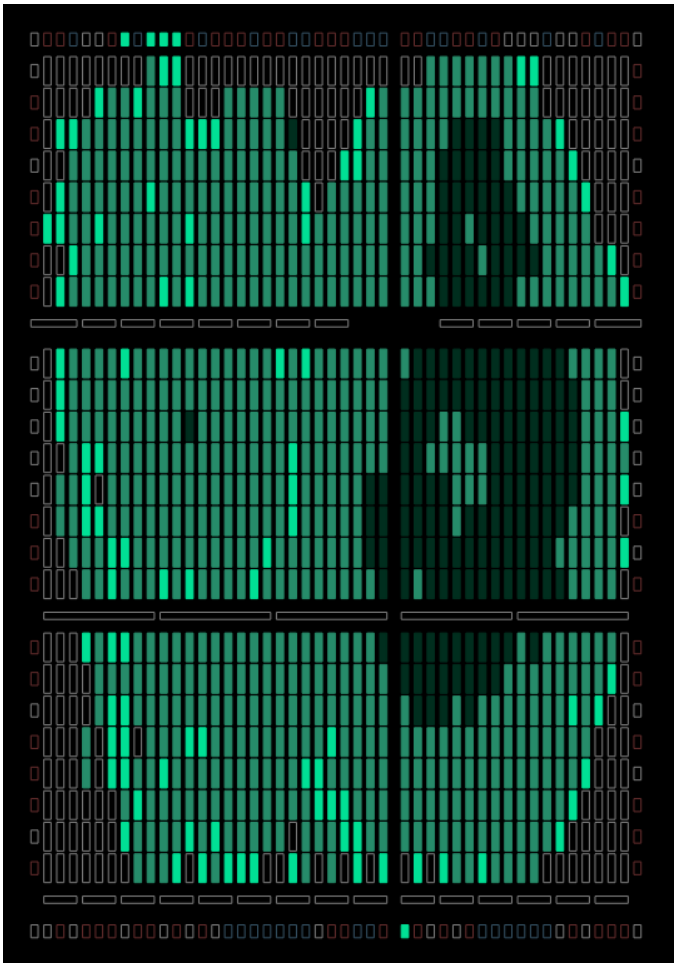

Fig. 9. PicoRV32 routed with Nextpnr shown in Gowin Floorplanner

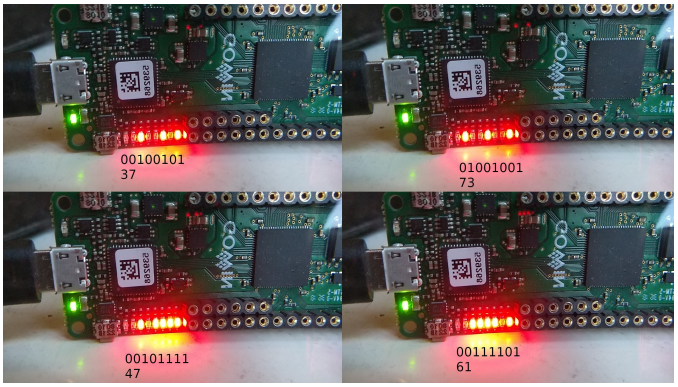

Fig. 10. End-to-end open source RISC-V prime calculation running on a Gowin FPGA.

The resulting JSON file was then packed into a bitstream, and programmed using the Gowin JTAG programmer. The design ran correctly at 12 MHz on the first try, showing that complex designs can be used with the current flow. In Fig. 10, the Gowin FPGA is seen displaying the numbers 37, 47, 61 and 73 in decimal, which are indeed prime.

The open source flow was tested on GW1N-1 and GW1NR-9 devices that were available as development board. In principle the flow should be compatible with any Gowin FPGA with only minimal effort, but this was not tested.

TABLE III
COMPARISON

	Frequency	LUT4s
Yosys (`abc9 -maxlut 8`)	98.4 Mhz	2247
Yosys (`abc9 -maxlut 6`)	89.4 Mhz	1746
GowinSynth	96.5 Mhz	1569

B. Comparison

In order to be able to classify the achieved results and to be able to evaluate the end-to-end flow, it is necessary to compare it to existing design flows. But because of the current trade war between the US and China and the resulting restriction in using US software (e.g., Synopsis) with Chinese hardware (Gowin), the comparison had to be adapted. Therefore, it is only possible to compare our results with Gowin's own synthesis results.

To compare the design flows, the same AttoSoC [25] code was used as for the end-to-end open source flow, but since `synth_gowin` does support BRAM, significantly less LUTs are used. Recent additions to `synth_gowin` enabled the use of timing-aware synthesis with ABC9 [26], as well as support for wide LUTs by combining several LUT4 into a larger LUT using a MUX2 primitive. The default synthesis behavior in Yosys is that after logic optimization the combinational logic is packed into generic lookup tables with 8 inputs (`abc9 -maxlut 8`). These LUT8s will be later decomposed during the technology mapping into up to 16 LUT4s with additional multiplexers. We optimized this behavior by forcing Yosys to use only lookup tables with 6 inputs as generic lookup tables (`abc9 -maxlut 6`).

As can be seen in TABLE III, Yosys actually produces a faster design, and gives you more control over the output but performs worse in area. So its f_{max} is actually higher than the vendor tools, but uses more area. The reason LUT8 uses more area is that Yosys is very eager to use them for speed gains, while they are twice as big as a LUT7 and four times as big as a LUT6. Looking into the vendor results, it uses only one or two LUT8, while Yosys uses dozens. This does in fact produce speed gains, but at a big area cost. As already mentioned, the control over output is far better, so it is possible to limit Yosys' ability to use the LUT8 and as can be seen it uses less area in the process. So it is possible to customize the designs to the very need of the task.

VIII. Conclusion and Future Work

With the goal of adding support for Gowin FPGAs to Yosys and Nextpnr in mind, this paper demonstrates the progress that has been made documenting the Gowin FPGA architecture and bitstream format and writing open source tools for these FPGAs.

Yosys support for Gowin FPGAs was extended to the point where it almost covers all primitives that the vendor supports, with the exception of latches, DSP, and some primitives that

cannot be automatically inferred by Yosys. This support has been merged upstream and can be used already with the vendor tools for PnR.

The basic architecture of the FPGA and the structure of the bitstream were described, including the topology of inter-tile wires, the layout of the tiles, and bit locations controlling basic primitives. An unpacker was written to decode simple designs from a bitstream file to show that the basic logic and routing are correctly understood.

A proof of concept Nextpnr target and bitstream packer were written to demonstrate a fully open source flow capable of synthesizing and running various programs. It was shown that the open source flow is capable of synthesizing a full RISC-V CPU that works correctly on a GW1NR-9 FPGA. It is planned to synthesize complex programs and designs, like [27], in the future as well. This demonstrates that even at this early stage, the open source flow can be used for relatively complex software, as long as it does not place high requirements on memory and DSP.

Everything achieved so far works perfectly for the whole Gowin LittleBee family and should in theory work for the Gowin Aurora family as well. One of the future steps is to prove this theory and conduct experiments with the Aurora family. All modifications to Yosys and nextpnr as well as the bitstream packer and unpacker are made open source [6]. The tools can be freely used and modified by the research community.

At this moment, many features of the FPGAs remain unexplored. Up to this point, the main focus has been on understanding the configurable logic tiles and routing fabric. In this area, global clock nets, long wires, ALU modes, and DRAM memory modes remain unexplored.

Other tile types have received a lot less focus. For IOB tiles, only a bare minimum is supported. Much work remains to be done for more complicated usage such as tri-state outputs, differential pairs, and different drive strengths and levels. Other tiles such as DSP, BRAM and PLL, are completely unexplored.

So far, a lot has been achieved, but a long road lays ahead before mature Gowin support is implemented in Yosys and Nextpnr.

ACKNOWLEDGMENT

This research was partially funded by Symbiotic EDA.

REFERENCES

[1] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go ahead: A partial reconfiguration framework. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 37–44. IEEE, 2012.

[2] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 27(1):10–25, 2010.

[3] Synplify Pro. Synplicity inc, 2006.

[4] Claire Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[5] Nextpnr. https://github.com/YosysHQ/nextpnr.

[6] Pepijn de Vos. Project apicula. *URl: https://github.com/YosysHQ/apicula*, 2019.

[7] Christiaan Baaij. Cλash : from haskell to hardware, December 2009.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. 06 2012.

[9] M-Labs. nmigen: A refreshed python toolbox for building complex digital hardware. https://github.com/m-labs/nmigen.

[10] William Kleitz. *Digital Electronics with VHDL, Quartus II Version*. Pearson Prentice Hall, 2006.

[11] SymbiFlow. Open source FPGA tooling for rapid innovation. https://symbiflow.github.io/, 2018. accessed 20 November 2019.

[12] Claire Wolf and Mathias Lasser. Project icestorm. http://www.clifford.at/icestorm/.

[13] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. The vtr project: Architecture and cad for fpgas from verilog to routing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 77–86, New York, NY, USA, 2012. ACM.

[14] Cotton Seed. Arachne-pnr. *URl: https://github. com/cseed/arachnepnr*, 2018.

[15] David Shah. Project Trellis. https://github.com/SymbiFlow/prjtrellis.

[16] Google. Project X-Ray. https://github.com/SymbiFlow/prjxray.

[17] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[18] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, 2009.

[19] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[20] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[21] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012.

[22] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn & fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59, 2017.

[23] S. Arno and F. S. Wheeler. Signed digit representations of minimal hamming weight. *IEEE Transactions on Computers*, 42(8):1007–1010, 1993.

[24] Roman Rohleder. Hands-on ghidra - a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, SPRO'19, page 77–78, New York, NY, USA, 2019. Association for Computing Machinery.

[25] David Shah. Attosoc. https://github.com/SymbiFlow/prjtrellis/blob/master/examples/soc_ecp5_evn/attosoc.v.

[26] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/.

[27] Michael Kirchhoff, Philipp Kerling, Detlef Streitferdt, and Wolfgang Fengler. A real-time capable dynamic partial reconfiguration system for an application-specific soft-core processor. *International Journal of Reconfigurable Computing*, 2019.