Yosys Application Note 010: Converting Verilog to BLIF

Clifford Wolf

November 2013

Abstract—Verilog-2005 is a powerful Hardware Description Language (HDL) that can be used to easily create complex designs from small HDL code. It is the preferred method of design entry for many designers[1].

The Berkeley Logic Interchange Format (BLIF) [6] is a simple file format for exchanging sequential logic between programs. It is easy to generate and easy to parse and is therefore the preferred method of design entry for many authors of logic synthesis tools.

Yosys [1] is a feature-rich Open-Source Verilog synthesis tool that can be used to bridge the gap between the two file formats. It implements most of Verilog-2005 and thus can be used to import modern behavioral Verilog designs into BLIF-based design flows without dependencies on proprietary synthesis tools.

The scope of Yosys goes of course far beyond Verilog logic synthesis. But it is a useful and important feature and this Application Note will focus on this aspect of Yosys.

I. Installation

Yosys written in C++ (using features from C++11) and is tested on modern Linux. It should compile fine on most UNIX systems with a C++11 compiler. The README file contains useful information on building Yosys and its prerequisites.

Yosys is a large and feature-rich program with a couple of dependencies. It is, however, possible to deactivate some of the dependencies in the Makefile, resulting in features in Yosys becoming unavailable. When problems with building Yosys are encountered, a user who is only interested in the features of Yosys that are discussed in this Application Note may deactivate TCL, Qt and MiniSAT support in the Makefile and may opt against building yosys-abc.

This Application Note is based on GIT Rev. e216e0e from 2013-11-23 of Yosys [1]. The Verilog sources used for the examples are taken from yosys-bigsim [2], a collection of real-world designs used for regression testing Yosys.

II. Getting Started

We start our tour with the Navré processor from yosys-bigsim. The Navré processor [3] is an Open Source AVR clone. It is a single module (softusb_navre) in a single design file (softusb_navre.v). It also is using only features that map nicely to the BLIF format, for example it only uses synchronous resets.

Converting softusb_navre.v to softusb_navre.blif could not be easier:

```
1  yosys -o softusb_navre.blif -S softusb_navre.v
```

Listing 1. Calling Yosys without script file

Behind the scenes Yosys is controlled by synthesis scripts that execute commands that operate on Yosys' internal state. For example, the -o softusb_navre.blif option just adds the command write_blif softusb_navre.blif to the end of the script. Likewise a file on the command line

---

[1]The other half prefers VHDL, a very different but – of course – equally powerful language.

– softusb_navre.v in this case – adds the command read_verilog softusb_navre.v to the beginning of the synthesis script. In both cases the file type is detected from the file extension.

Finally the option -S instantiates a built-in default synthesis script. Instead of using -S one could also specify the synthesis commands for the script on the command line using the -p option, either using individual options for each command or by passing one big command string with a semicolon-separated list of commands. But in most cases it is more convenient to use an actual script file.

III. Using a Synthesis Script

With a script file we have better control over Yosys. The following script file replicates what the command from the last section did:

```
1  read_verilog softusb_navre.v
2  hierarchy
3  proc; opt; memory; opt; techmap; opt
4  write_blif softusb_navre.blif
```

Listing 2. softusb_navre.ys

The first and last line obviously read the Verilog file and write the BLIF file.

The 2nd line checks the design hierarchy and instantiates parametrized versions of the modules in the design, if necessary. In the case of this simple design this is a no-op. However, as a general rule a synthesis script should always contain this command as first command after reading the input files.

The 3rd line does most of the actual work:

- The command opt is the Yosys' built-in optimizer. It can perform some simple optimizations such as const-folding and removing unconnected parts of the design. It is common practice to call opt after each major step in the synthesis procedure. In cases where too much optimization is not appreciated (for example when analyzing a design), it is recommended to call clean instead of opt.
- The command proc converts *processes* (Yosys' internal representation of Verilog always- and initial-blocks) to circuits of multiplexers and storage elements (various types of flip-flops).
- The command memory converts Yosys' internal representations of arrays and array accesses to multi-port block memories, and then maps this block memories to address decoders and flip-flops, unless the option -nomap is used, in which case the multi-port block memories stay in the design and can then be mapped to architecture-specific memory primitives using other commands.
- The command techmap turns a high-level circuit with coarse grain cells such as wide adders and multipliers to a fine-grain circuit of simple logic primitives and single-bit storage elements. The command does that by substituting the complex cells by circuits of simpler cells. It is possible to provide a custom set of rules for this process in the form of a Verilog source file, as we will see in the next section.

Now Yosys can be run with the filename of the synthesis script as argument:

```
1  yosys softusb_navre.ys
```

Listing 3. Calling Yosys with script file

Now that we are using a synthesis script we can easily modify how Yosys synthesizes the design. The first thing we should customize is the call to the `hierarchy` command:

Whenever it is known that there are no implicit blackboxes in the design, i.e. modules that are referenced but are not defined, the `hierarchy` command should be called with the `-check` option. This will then cause synthesis to fail when implicit blackboxes are found in the design.

The 2nd thing we can improve regarding the `hierarchy` command is that we can tell it the name of the top level module of the design hierarchy. It will then automatically remove all modules that are not referenced from this top level module.

For many designs it is also desired to optimize the encodings for the finite state machines (FSMs) in the design. The `fsm` command finds FSMs, extracts them, performs some basic optimizations and then generate a circuit from the extracted and optimized description. It would also be possible to tell the `fsm` command to leave the FSMs in their extracted form, so they can be further processed using custom commands. But in this case we don't want that.

So now we have the final synthesis script for generating a BLIF file for the Navré CPU:

```
1  read_verilog softusb_navre.v
2  hierarchy -check -top softusb_navre
3  proc; opt; memory; opt; fsm; opt; techmap; opt
4  write_blif softusb_navre.blif
```

Listing 4. `softusb_navre.ys` (improved)

IV. Advanced Example: The Amber23 ARMv2a CPU

Our 2nd example is the Amber23 [4] ARMv2a CPU. Once again we base our example on the Verilog code that is included in yosys-bigsim [2].

The problem with this core is that it contains no dedicated reset logic. Instead the coding techniques shown in Listing 6 are used to define reset values for the global asynchronous reset in an FPGA

```
1  read_verilog a23_alu.v
2  read_verilog a23_barrel_shift_fpga.v
3  read_verilog a23_barrel_shift.v
4  read_verilog a23_cache.v
5  read_verilog a23_coprocessor.v
6  read_verilog a23_core.v
7  read_verilog a23_decode.v
8  read_verilog a23_execute.v
9  read_verilog a23_fetch.v
10 read_verilog a23_multiply.v
11 read_verilog a23_ram_register_bank.v
12 read_verilog a23_register_bank.v
13 read_verilog a23_wishbone.v
14 read_verilog generic_sram_byte_en.v
15 read_verilog generic_sram_line_en.v
16 hierarchy -check -top a23_core
17 add -global_input globrst 1
18 proc -global_arst globrst
19 techmap -map adff2dff.v
20 opt; memory; opt; fsm; opt; techmap
21 write_blif amber23.blif
```

Listing 5. `amber23.ys`

```
1  reg [7:0] a = 13, b;
2  initial b = 37;
```

Listing 6. Implicit coding of global asynchronous resets

implementation. This design can not be expressed in BLIF as it is. Instead we need to use a synthesis script that transforms this form to synchronous resets that can be expressed in BLIF.

(Note that there is no problem if this coding techniques are used to model ROM, where the register is initialized using this syntax but is never updated otherwise.)

Listing 5 shows the synthesis script for the Amber23 core. In line 17 the `add` command is used to add a 1-bit wide global input signal with the name `globrst`. That means that an input with that name is added to each module in the design hierarchy and then all module instantiations are altered so that this new signal is connected throughout the whole design hierarchy.

In line 18 the `proc` command is called. But in this script the signal name `globrst` is passed to the command as a global reset signal for resetting the registers to their assigned initial values.

Finally in line 19 the `techmap` command is used to replace all instances of flip-flops with asynchronous resets with flip-flops with synchronous resets. The map file used for this is shown in Listing 7. Note how the `techmap_celltype` attribute is used in line 1 to tell the techmap command which cells to replace in the design, how the `_TECHMAP_FAIL_` wire in lines 15 and 16 (which evaluates to a constant value) determines if the parameter set is compatible with this replacement circuit, and how the `_TECHMAP_DO_` wire in line 13 provides a mini synthesis-script to be used to process this cell.

```
1  (* techmap_celltype = "$adff" *)
2  module adff2dff (CLK, ARST, D, Q);
3
4  parameter WIDTH = 1;
5  parameter CLK_POLARITY = 1;
6  parameter ARST_POLARITY = 1;
7  parameter ARST_VALUE = 0;
8
9  input CLK, ARST;
10 input [WIDTH-1:0] D;
11 output reg [WIDTH-1:0] Q;
12
13 wire [1023:0] _TECHMAP_DO_ = "proc";
14
15 wire _TECHMAP_FAIL_ =
16     !CLK_POLARITY || !ARST_POLARITY;
17
18 always @(posedge CLK)
19         if (ARST)
20                 Q <= ARST_VALUE;
21         else
22                 Q <= D;
23
24 endmodule
```

Listing 7. `adff2dff.v`

```
1   #include <stdint.h>
2   #include <stdbool.h>
3
4   #define BITMAP_SIZE 64
5   #define OUTPORT 0x10000000
6
7   static uint32_t bitmap[BITMAP_SIZE/32];
8
9   static void bitmap_set(uint32_t idx) { bitmap[idx/32] |= 1 << (idx % 32); }
10  static bool bitmap_get(uint32_t idx) { return (bitmap[idx/32] & (1 << (idx % 32))) != 0; }
11  static void output(uint32_t val) { *((volatile uint32_t*)OUTPORT) = val; }
12
13  int main() {
14          uint32_t i, j, k;
15          output(2);
16          for (i = 0; i < BITMAP_SIZE; i++) {
17                  if (bitmap_get(i)) continue;
18                  output(3+2*i);
19                  for (j = 2*(3+2*i);; j += 3+2*i) {
20                          if (j%2 == 0) continue;
21                          k = (j-3)/2;
22                          if (k >= BITMAP_SIZE) break;
23                          bitmap_set(k);
24                  }
25          }
26          output(0);
27          return 0;
28  }
```

Listing 8. Test program for the Amber23 CPU (Sieve of Eratosthenes). Compiled using GCC 4.6.3 for ARM with `-Os -marm -march=armv2a -mno-thumb-interwork -ffreestanding`, linked with `--fix-v4bx` set and booted with a custom setup routine written in ARM assembler.

V. Verification of the Amber23 CPU

The BLIF file for the Amber23 core, generated using Listings 5 and 7 and the version of the Amber23 RTL source that is bundled with yosys-bigsim, was verified using the test-bench from yosys-bigsim. It successfully executed the program shown in Listing 8 in the test-bench.

For simulation the BLIF file was converted back to Verilog using ABC [5]. So this test includes the successful transformation of the BLIF file into ABC's internal format as well.

The only thing left to write about the simulation itself is that it probably was one of the most energy inefficient and time consuming ways of successfully calculating the first 31 primes the author has ever conducted.

VI. Limitations

At the time of this writing Yosys does not support multi-dimensional memories, does not support writing to individual bits of array elements, does not support initialization of arrays with $readmemb and $readmemh, and has only limited support for tristate logic, to name just a few limitations.

That being said, Yosys can synthesize an overwhelming majority of real-world Verilog RTL code. The remaining cases can usually be modified to be compatible with Yosys quite easily.

The various designs in yosys-bigsim are a good place to look for examples of what is within the capabilities of Yosys.

VII. Conclusion

Yosys is a feature-rich Verilog-2005 synthesis tool. It has many uses, but one is to provide an easy gateway from high-level Verilog code to low-level logic circuits.

The command line option `-S` can be used to quickly synthesize Verilog code to BLIF files without a hassle.

With custom synthesis scripts it becomes possible to easily perform high-level optimizations, such as re-encoding FSMs. In some extreme cases, such as the Amber23 ARMv2 CPU, the more advanced Yosys features can be used to change a design to fit a certain need without actually touching the RTL code.

References

[1] Clifford Wolf. The Yosys Open SYnthesis Suite.
http://www.clifford.at/yosys/
[2] yosys-bigsim, a collection of real-world Verilog designs for regression testing purposes.
https://github.com/cliffordwolf/yosys-bigsim
[3] Sebastien Bourdeauducq. Navré AVR clone (8-bit RISC).
http://opencores.org/project,navre
[4] Conor Santifort. Amber ARM-compatible core.
http://opencores.org/project,amber
[5] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification.
http://www.eecs.berkeley.edu/~alanmi/abc/
[6] Berkeley Logic Interchange Format (BLIF)
http://vlsi.colorado.edu/~vis/blif.ps