Convert Xilinx FPGA/CPLD to C Source

Licence CPOL
First Posted **28 May 2011**

By **grilialex** | 28 May 2011 | Unedited contribution

Embedded | C | Microcontroller

Flow and tools to convert Xilinx bitstreams to C source code for programming FPGA/CPLD

Summary

In a previous article I presented the **AVRILOS** round-robin operating system. In this article I will describe the process and tools to generate FPGA configuration data for integration into your code and be able to configure FPGA devices without the need of an external serial or PROM/Flash. This will reduce chip count if you integrate FPGAs with a microcontroller as I often do. I will also present a makefile for compiling the Xilinx FPGA code. In a next article I will present how to download and use this code from **AVRILOS**.

Relative Links

- **AVRILOS** First Article (Introduction)
- FPGA/CPLD Design Flow for **AVRILOS**
- **AVRILOS** & FPGA

Introduction

I will present the process of how to generate the C file with the bitstream information for configuring the Xilinx Spartan FPGAs, although you can apply this technique to virtually any serially configured device. I will provide any necessary tools (and their code).

Background

Microcontrollers are very popular because they can easily control systems and they are flexible. Nowadays microcontrollers have bigger memory capacities and a bunch of peripherals that can do almost everything. Another class of devices that have similarities with microcontrollers is the programmable logic devices called CPLD (Complex Programmable Logic Device) or FPGA (Field Programmable Gate Arrays). These are devices that are programmed in hardware description languages (like VHDL, Verilog, System C) and their primitive functionality is down to flip-flop and gates, something which is a low level comparing to microcontrollers. The good thing about these devices is that they can be reprogrammed (like microcontrollers) and do easily functions that are better implemented in hardware. For example it is a nightmare to do a bit permutation in software, while in hardware this operation will need no more than wiring connections - not even gates! One drawback is that most SRAM based FPGAs need an external serial PROM/Flash that holds their configuration program (aka. Their code). This program is transferred during start-up to internal FPGA SRAM.

As I design both FPGAs and microcontrollers my favorite setup is to make combo designs. I use a microcontroller for my main logic

while I add up an FPGA for peripheral expansion. FPGAs offer logic gates, Flip-flops and many I/O which I can use to extend a conventional AVR DIP40 package. Because I have enough memory to the microcontroller I include the FPGA configuration inside the Flash eliminating this way the external serial PROM/Flash.

In this process presented here I assume that you know how to write, simulate, synthesize, place & route and generate the final bit-stream of the FPGA. Although I will present this technique for Xilinx Spartan devices, you can use virtually any device (and microcontroller of course).

Description

Aim of the project

Aim of the project is to provide or demonstrate a flow for generating C code from FPGA tools bitstreams that can be used to configure FPGAs from microcontrollers thus omitting specific external PROM/Flash.

Tools

The tools for FPGA Spartan development:

1. Xilinx Spartan4K development tools
2. External Synthesizer (I am using Xilinx Foundation edition synthesis)
3. WebISE package with simulator
4. CVTFPGA tool for integrating serial bitstreams of Xilinx Spartan FPGAs to my code.

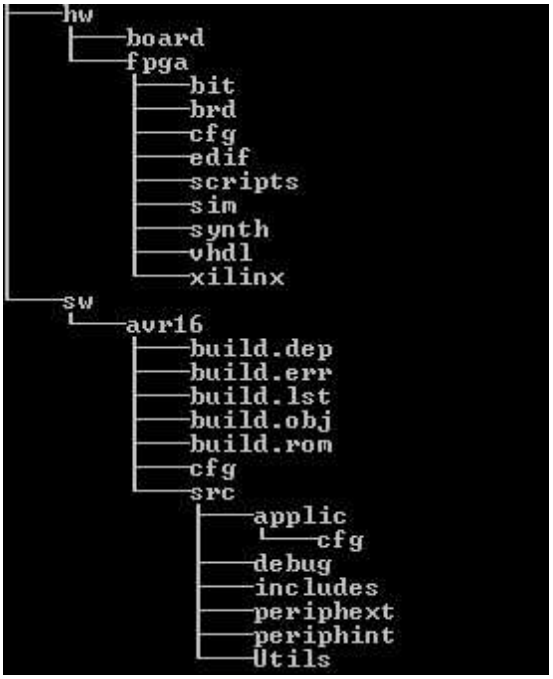
Other Tools that you might need:

1. GNUWIN32 (for makefiles if I don't use the WinAVR ie. Other compiler packages like MPLAB)
2. Anything else you can imagine and fits.

AVRILOS & FPGA

Directory Structure

The directory structure is as follows:



There are two major directories, HW and SW.

1. HW is the directory where all my hardware development is done. This includes board schematic & PCB files as well as FPGA design.
2. SW is the software tree of the microcontroller, more on this you can find in my first article for **AVRILOS**.

Let's concentrate on this directory structure of FPGA.

1. Directory VHDL: Here are my source files for the FPGA.
2. Directory CFG: Contains FPGA configuration files like constrain files where pin-locking and timing directives are placed.
3. Directory Scripts: contains makefiles and other scripts.
4. Directory SIM: Here I copy the source files (from VHDL directory) where I perform the simulation. Also any simulation scripts or testbench are placed here as well.
5. Directory Synthesis: Here I copy the files for doing the logic synthesis.
6. Directory EDIF: The generated synthesis output is put here. Current scripts are using EDIF format but XNF can be used as well.
7. Directory Xilinx: Place & Route, and general Xilinx flow.
8. Directory BIT: The final bitstream is put here as well as its PROM hex or binary format as well.
9. Directory BRD: Contains the generated C output and its header file.

The root makefile (named makefile) resides on the scripts/ directory of FPGA.

Of course you can prefer the GUI offered by WebISE and avoid this hard core command line craziness. The beauty of makefile is that I do not need to remember where various GUI options exists after a few months or years where I need to do improvements or start complete new designs. It is more difficult for first-timers but after going that way you will not like to return… Each compilation will be exactly same today, tomorrow or next year on a different machine. If you upgrade the tools the script will break if a parameter meaning has changed and thus you will provide the needed attention to fix it (in a focused attention manner). Of

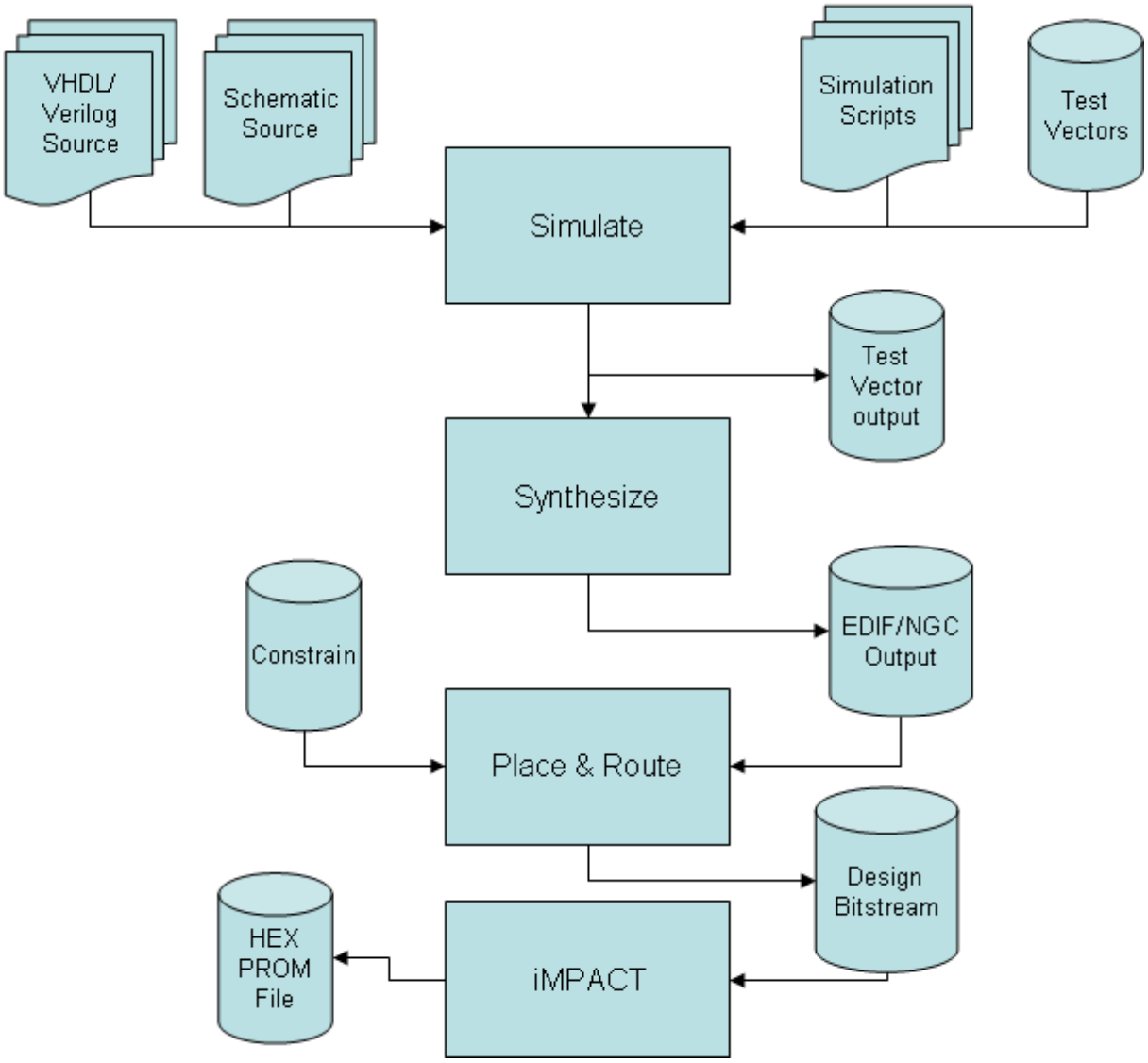
course you are free to go in a more manual and comfortable GUI way.

Description of flow

FPGA

There are two possible flows which are very similar. One for FPGA which is the original flow I was using for configuring FPGAs. In this flow I use the slave serial configuration interface to program the FPGA by the microcontroller. The microcontroller (**AVRILOS**) code for this flow was developed by me.

FPGA Flow



A very good document for programming Spartan FPGAs by using the slave serial configuration algorithm is shown in the next link:

Xilinx XAPP098.pdf

The information presented here can be used for most FPGAs that supports slave serial configuration (aka newest generations).

CPLD

The second flow is used for CPLD where you download the program only once to internal CPLD flash memory through the JTAG interface. I used this method successfully to program Xilinx

CPLD. However the microcontroller code was adopted by Xilinx Application Notes XAPP503 and XAPP058 available here:

- Xilinx XAPP058.pdf
- Xilinx XAPP502.pdf (relative design file XAPP502.ZIP for svf2xsvf502.exe converter, available after registration to Xilinx).
- Xilinx XAPP503.pdf

Code from Xilinx is available here:

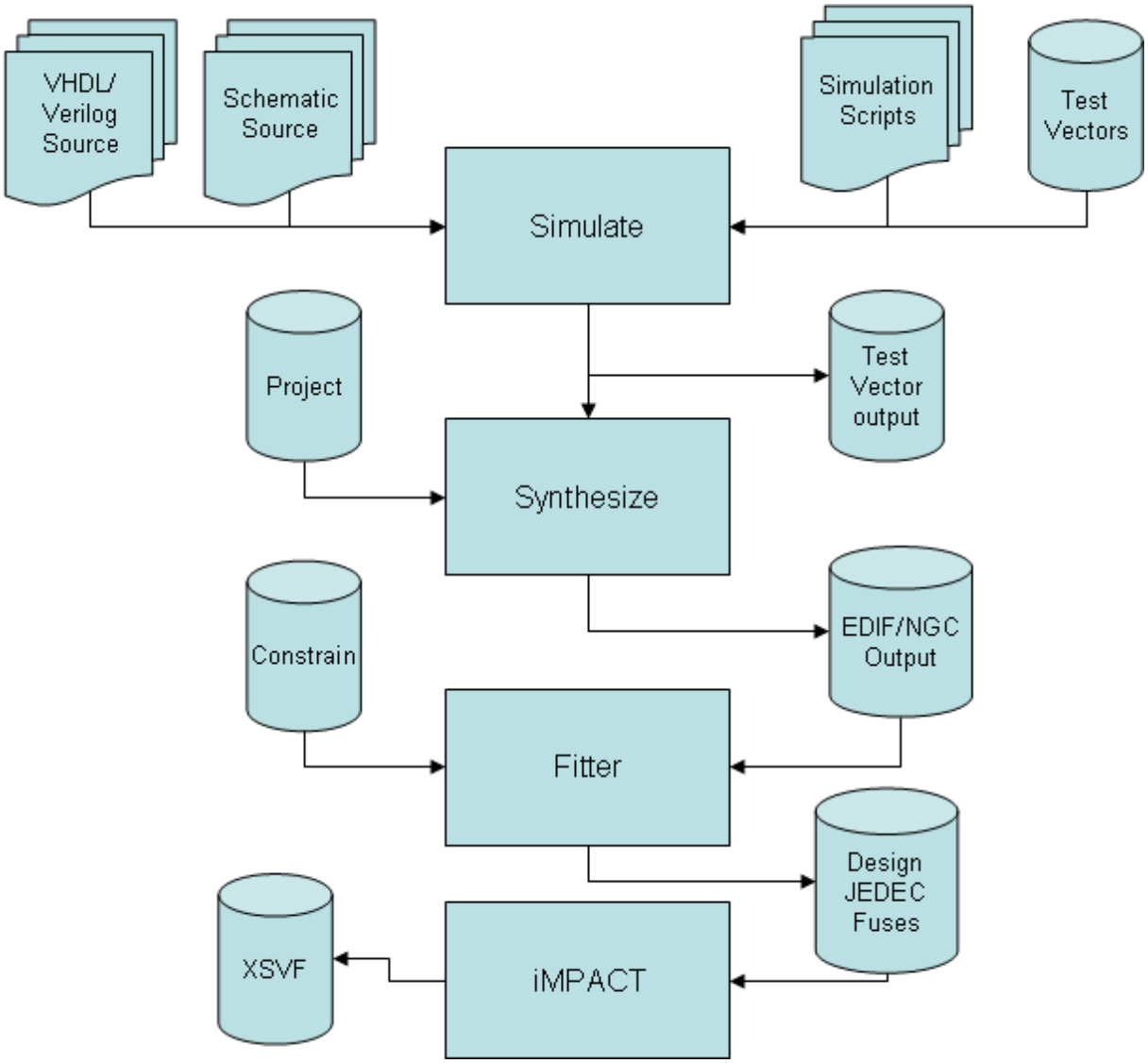
- Xilinx XAPP058.zip

Alternate sources (based on Xilinx code):

- http://www.ethernut.de/en/xsvfexec/

The flow for programming the CPLD is shown below:

CPLD Flow



Keep in mind that CPLD XSVF files are large (eg around 50K for XC9572XL).

You will also need a synthesis project file (.prj) which contains the VHDL files for the project. For Example: top_cpld.prj

`vhdl work "../vhdl/top_cpld.vhd"` (add all your project files).

Description of the makefile

Keep in mind that the flow presented here on, will relate with the FPGA design flow and not for CPLD, although process is similiar.

The makefile assumes you have synthesized your design and the netlist resides in the EDIF directory.Then it executes the Xilinx tool chain flow in sequence (or in a step by step manually):

1. `NGDBuild` (initial parsing) [`make NET`]
2. `MAP` (mapping logic to FPGA primitives) [`make MAP`]
3. `PAR` (Place and Route) [`make PAR`]
4. `TRCE` (Timing Analysis) [`make TIMING`]
5. Bitstream Generation [`make BITFLASH`]
6. AVR C code generation from bitstream. [`make AVR`]
7. [`make fpga`] does all these steps at once.
8. [`make cpld`] generates the corresponding xsvf files for CPLD download.

In the beginning of the makefile you will see the following definitions:

The TOP design is the base name of the design. The netlist should have this as it's a basename. In this case the top_fpga.edif will be the netlist required for compilation.

```
#Name of TOP Design
BASENAME=top_fpga
```

The part to compile for, is stated on the next variable:

```
#Part To Place & Route
PART=xcs05-4-pc84
```

The selected part stated is a Spartan device, PLCC 84 package, -4 propagation delay. Consult your Xilinx manual for other specific devices. This makefile is used with Spartan4K devices. For newer devices you should consult your manual for commands and options.

I am using the first generation of Spartan devices because I could use them with simple PCBs (I could use a through hole PLCC socket on experimental PCBs) and they use 5V which is compatible with many peripherals and the AVR CPU as well.

In the cfg/ directory you will find the corresponding constrain file that locks the individual pins to specific locations. If you have already a PCB or you do a design iteration you will need to specify pin locations for your I/O. Example definitions of pin locking inside this file are shown below:

```
NET "Clk"                       LOC =  "P29"; #SysClk
NET "MCUClk"                    LOC =  "P13";
NET "MCUData"                   LOC =  "P14";
NET "MCUFrame"                  LOC =  "P18";
NET "MCUData"                   PULLUP;

NET "pulseout<0>"       LOC =  "P19";
NET "pulseout<1>"       LOC =  "P20";
NET "pulseout<2>"       LOC =  "P23";
```

Also in the same directory you will find the configuration options for the bitstream generation at the file with .ut extension.

Description of the CVTFPGA tool

In the past I have done many variations of this tool, but none was complete. I relied on third party tools for Intel Hex translation to binary and the main converter tool was not so friendly.

In this release I support both direct binary files as well as Intel Hex files as input.

So you can manually convert a .hex file to .bin and then input the .bin file the converter. Or you can directly use the .hex file for C code translation. File formats (and extensions) accepted:

1. .hex, .mcs (Xilinx FPGA PROM generator for Slave Serial)
2. .xsvf (Xilinx iMPACT generation for JTAG programming, like CPLDs)
3. .bin (any binary file you may need to convert), same format as .xsvf.

Parameters accepted are:

1. `-o AVRILOS` : This will generate **AVRILOS** compatible definition in the .C file. This will eliminate any further editing from your side. You will need only to copy and rename the file to the appropriate applic/cfg directory (see next related article on **AVRILOS**).
2. `-o CFILOS` : This is for my ColdFire OS (similar and based to **AVRILOS**) .
3. `-f hexval` : This command will select a different fill value than 0xFF which is the default for unused areas. Ie. `-f 0x55` will fill unused areas with 0x55. This is mostly applicable for .hex files as binary files are continuous in memory.

You can use whichever option suits best your needs. This tool might be used to other relative purposes for binary to C translation as well.

The first section performs the OS setting checks and sets the internal variable `v_os` to the corresponding OS requested. Default is **AVRILOS**.

Next section determines the extension of the input files. The determination of the input destination base name is done by reverse scanning the filename and getting the last three letters. Buffer overflow is also checked (filename should be less than `c_DESTLEN` [200] chars). This is done in order to avoid cases where the file path is something like "./file.c" where the dot character is seen multiple times in the file path.

After the input file extensions are known the converter determines the file format by the extension. In case of input files of type .bin or .xsvf the actual type is raw binary. In case of .hex or .mcs the input file is a Hex file and the hex file parser is used.

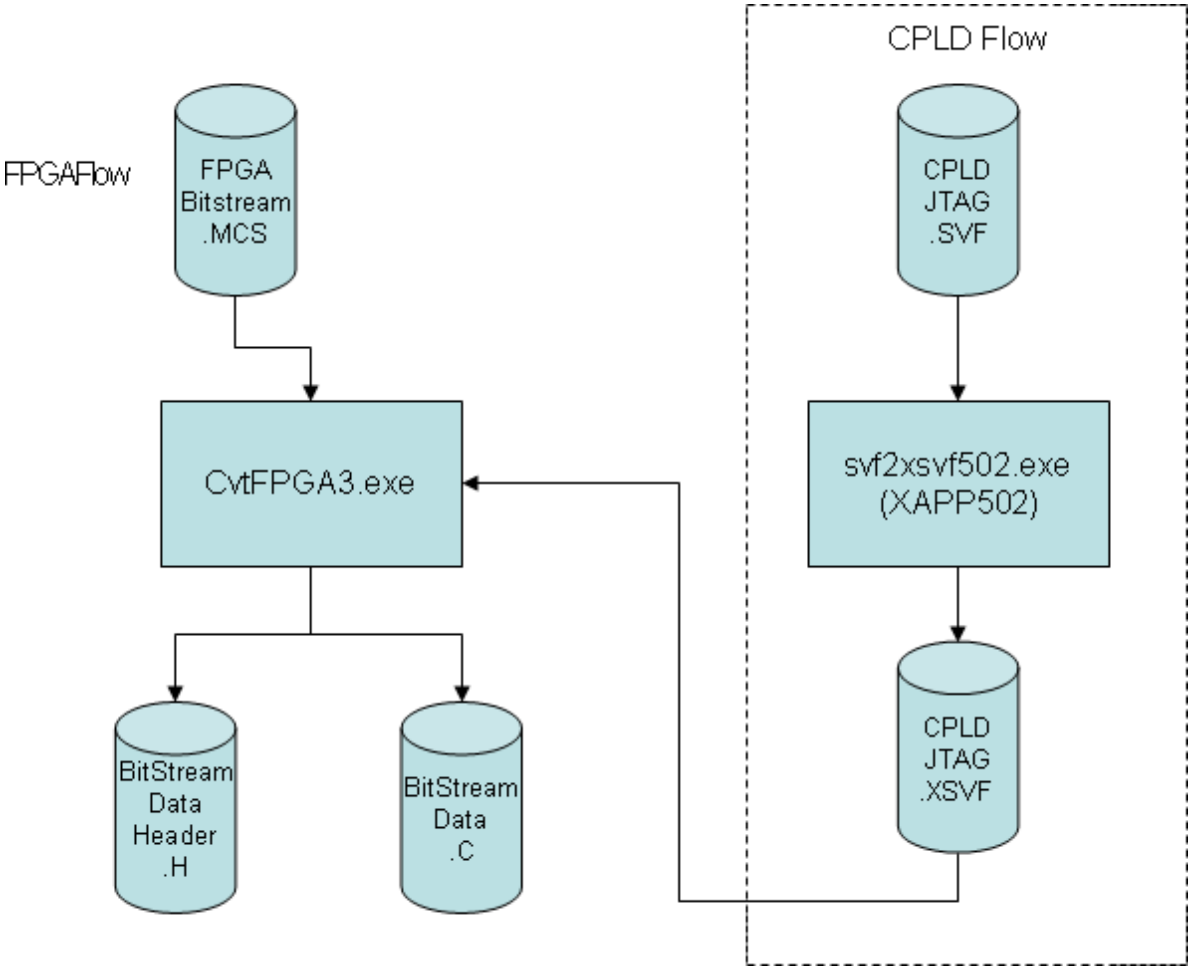
After filename determination the corresponding input and output files are opened. First we write the header file which depending of the OS requested could be a little different just for various MCU architectures and compilers.

Depending of the file type there are two parsers used. One for binary, which is very simple binary to ASCII converter. The second parser is more complex as it reads the Intel HEX format. Each line is parsed, the checksum is checked for errors, and then the record type is determined. This is accomplished by the function

`f_GetHexLine`. Also a virtual record pointer is updated depending of the record type and its address field. If an empty (unprogrammed) area is detected (address not continuous) then the fill value is used to fill this area. Default value is 0xFF.

Then the actual C source file is generated.

Convert FPGA / CPLD Flow



A header file is generated as well for reference that will match the C table declaration in the C file. In the above illustration we convert the CPLD SVF file to XSVF. However the makefile provided generates directly XSVF through iMPACT (The xilinx programming tool). Using this makefile the svf2xsvf502 converter is not needed.

Example Compilation

Open a command prompt at the <project>/hw/fpga/scripts.

Assuming you have synthesized you code (after simulation) and the edif/ncf netlist is ready, you can compile for Xilinx toolchain (NET, MAP,PAR, BITGEN, AVR).

Also i assume that the cvtfpga3.exe file is in your system path.

So in the script directory run the command:

```
make avr
```

This command will try to execute the full flow for generating the .c/.h final files.

An example output is shown below:

```
Netlist Parsing: top_fpga, Part: xcs05-4-pc84
cd ../xilinx/
ngdbuild -p xcs05-4-pc84 -uc ../cfg/top_fpga.ucf -dd
../xilinx ../edif/top_fpga.edf ../xilinx/top_fpga.ngd
Release 4.2.03i - ngdbuild E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

Command Line: ngdbuild -p xcs05-4-pc84 -uc ../cfg
/top_fpga.ucf -dd ../xilinx
../edif/top_fpga.edf ../xilinx/top_fpga.ngd

Launcher: Executing edif2ngd
"C:\Users\Ilias\work\OSS\AVRILOS_02\code\CodeProject_02
\hw\fpga\edif\top_fpga.ed
f"
"C:\Users\Ilias\work\OSS\AVRILOS_02\code\CodeProject_02
\hw\fpga\xilinx\top_fpga.
ngo"
INFO:NgdBuild - Release 4.2.03i - edif2ngd E.38
INFO:NgdBuild - Copyright (c) 1995-2001 Xilinx, Inc.  All
rights reserved.
Reading NCF file
"C:/Users/Ilias/work/OSS/AVRILOS_02/code/CodeProject_02
/hw/fpga/edif/top_fpga.nc
f"...
Writing the design to
"C:/Users/Ilias/work/OSS/AVRILOS_02/code/CodeProject_02
/hw/fpga/xilinx/top_fpga.
ngo"...
Reading NGO file
"C:/Users/Ilias/work/OSS/AVRILOS_02/code/CodeProject_02
/hw/fpga/xilinx/top_fpga.
ngo" ...
Reading component libraries for design expansion...

Annotating constraints to design from file "../cfg
/top_fpga.ucf" ...
WARNING:NgdBuild:383 - A case sensitive search for the
INST, PAD, or NET element
   refered to by a constraint entry in the UCF file that
accompanies this design
   has failed, while a case insensitive search is in
progress. The result of the
   case insenstive search will be used, but warnings will
accompany each and
   every use of a case insensitive result. Constraints are
case sensitive with
   respect to user-specified identifiers, which includes
names of logic elements
   in a design. For the sake of compatibility with
currently existing .xnf,
   .xtf, and .xff files, Software will allow a case
insensitive search for INST,
   PAD, or NET elements referenced in a .ucf file.
WARNING:NgdBuild:385 - Found case insensitive match for
NET name 'Clk'. NET is
   'clk'.
WARNING:NgdBuild:385 - Found case insensitive match for
NET name 'MCUClk'. NET
   is 'mcuclk'.
WARNING:NgdBuild:385 - Found case insensitive match for
NET name 'MCUData'. NET
   is 'mcudata'.
WARNING:NgdBuild:385 - Found case insensitive match for
NET name 'MCUFrame'. NET
   is 'mcuframe'.
WARNING:NgdBuild:385 - Found case insensitive match for
NET name 'MCUData'. NET
   is 'mcudata'.
Attached a PULLUP primitive to pad net mcudata

Checking timing specifications ...
Checking expanded design ...
```

```
NGDBUILD Design Results Summary:
  Number of errors:     0
  Number of warnings:   0

Writing NGD file "../xilinx/top_fpga.ngd" ...

Writing NGDBUILD log file "../xilinx/top_fpga.bld"...

NGDBUILD done.
Netlist Mapping: top_fpga, Part: xcs05-4-pc84
cd ../xilinx/
map -p xcs05-4-pc84 -o ../xilinx/map.ncd ../xilinx
/top_fpga.ngd ../xilinx/top_fpga.pcf
Release 4.2.03i - Map E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.
Reading NGD file "../xilinx/top_fpga.ngd"...
Using target part "s05pc84-4".
MAP spartan directives:
   Partname = "xcs05-4-pc84".
   Covermode = "area".
   Pack Unrelated Logic into CLBs targeting 97% of CLB
resources.
Processing logical timing constraints...
Verifying F/HMAP validity based on pre-trimmed logic...
Removing unused logic...
Packing logic in CLBs...
   Running cover...
   Undirected packing...
Running physical design DRC...

Design Summary:
   Number of errors:        0
   Number of warnings:      2
   Number of CLBs:             97 out of   100   97%
      CLB Flip Flops:     154
      4 input LUTs:       168
      3 input LUTs:        59 (45 used as route-throughs)
   Number of bonded IOBs:      17 out of    61   27%
      IOB Flops:            4
      IOB Latches:          0
   Number of clock IOB pads:    1 out of     8   12%
   Number of secondary CLKs:    1 out of     4   25%
   Number of startup:           1 out of     1  100%
   15 unrelated functions packed into 11 CLBs.
   (11% of the CLBs used are affected.)
Total equivalent gate count for design: 2222
Additional JTAG gate count for IOBs:    816
Writing design file "../xilinx/map.ncd"...

Removed Logic Summary:
 145 block(s) removed
  43 block(s) optimized away
 126 signal(s) removed

Mapping completed.
See MAP report file "../xilinx/map.mrp" for details.
Device Place and Route: top_fpga, Part: xcs05-4-pc84
cd ../xilinx/
par  -w -ol 1 -x ../xilinx/map.ncd ../xilinx/top_fpga.ncd
../xilinx/top_fpga.pcf
Release 4.2.03i - Par E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.



Constraints file: ../xilinx/top_fpga.pcf

Loading design for application par from file ../xilinx
/map.ncd.
   "top_fpga" is an NCD, version 2.37, device xcs05,
package pc84, speed -4
Loading device for application par from file '4003e.nph'
in environment
c:/prog/xilinx.
```

```
Device speed data version:  D 1.3 FINAL.


Resolving physical constraints.
Finished resolving physical constraints.

Device utilization summary:

   Number of External IOBs            17 out of 80     27%
      Flops:                           4
      Latches:                         0
   Number of IOBs driving Global Buffers    1 out of
8      12%

   Number of CLBs                     97 out of 100    97%
      Total CLB Flops:               154 out of 200    77%
      4 input LUTs:                  168 out of 200    84%
      3 input LUTs:                   59 out of 100    59%

   Number of SEC-CLKs                  1 out of 4      25%
   Number of STARTUPs                  1 out of 1     100%



Overall effort level (-ol):   1 (set by user)
Placer effort level (-pl):    1 (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):    1 (set by user)
Extra effort level (-xe):     0 (default)

Starting initial Placement phase.  REAL time: 0 secs
Finished initial Placement phase.  REAL time: 0 secs

Starting Constructive Placer.  REAL time: 0 secs
Placer score = 96660
Placer score = 81240
Placer score = 72120
Placer score = 67080
Placer score = 63180
Placer score = 60240
Placer score = 57000
Placer score = 54420
Finished Constructive Placer.  REAL time: 0 secs

Dumping design to file ../xilinx/top_fpga.ncd.

Starting Optimizing Placer.  REAL time: 2 secs
Optimizing
Swapped 7 comps.
Xilinx Placer [1]   52980   REAL time: 2 secs

Finished Optimizing Placer.  REAL time: 2 secs

Dumping design to file ../xilinx/top_fpga.ncd.

Total REAL time to Placer completion: 2 secs
Total CPU time to Placer completion: 1 secs

0 connection(s) routed; 824 unrouted.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 2 secs
Starting iterative routing.
Routing active signals.
End of iteration 1
824 successful; 0 unrouted; (0) REAL time: 2 secs
Constraints are met.
Dumping design to file ../xilinx/top_fpga.ncd.
Starting cleanup
Improving routing.
End of cleanup iteration 1
824 successful; 0 unrouted; (0) REAL time: 3 secs
Dumping design to file ../xilinx/top_fpga.ncd.
Total REAL time: 3 secs
Total CPU  time: 3 secs
End of route.  824 routed (100.00%); 0 unrouted.
No errors found.
```

```
Completely routed.

The design submitted for place and route did not meet the
specified timing
requirements.  Please use the static timing analysis tools
(TRCE or Timing
Analyzer) to report which constraints were not met.  To
obtain a better result,
you may try the following:
  * Use the Re-entrant routing feature to run more router
iterations on the
design.
  * Check the timing constraints to make sure the design
is not
over-constrained.
  * Specify a higher placer effort level, if possible.
  * Specify a higher router effort level.
  * Use the Multi-Pass PAR (MPPR) feature.  This generates
multiple placement
trials from which the best (i.e., lowest design score)
placement can be used
with re-entrant routing to obtain a better result.

Please consult the Development System Reference Guide for
more detailed
information about the usage options pertaining to these
features.

Total REAL time to Router completion: 3 secs
Total CPU time to Router completion: 3 secs

Generating PAR statistics.
Dumping design to file ../xilinx/top_fpga.ncd.


All signals are completely routed.

Total REAL time to PAR completion: 4 secs
Total CPU time to PAR completion: 3 secs

Placement: Completed - No errors found.
Routing: Completed - No errors found.

PAR done.
Generating Configuration Bitstream for FLASH download:
top_fpga, Part: xcs05-4-pc84
cd ../xilinx/
bitgen ../xilinx/top_fpga.ncd  -m -w -f ../cfg
/bitgen_cclk.ut
Release 4.2.03i - Bitgen E.38
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.

Loading design for application Bitgen from file ../xilinx
/top_fpga.ncd.
   "top_fpga" is an NCD, version 2.37, device xcs05,
package pc84, speed -4
Loading device for application Bitgen from file
'4003e.nph' in environment
c:/prog/xilinx.
Opened constraints file ../xilinx/top_fpga.pcf.

Wed Apr 06 22:28:27 2011

Running DRC.
DRC detected 0 errors and 0 warnings.
Creating bit map...
Saving bit stream in "../xilinx/top_fpga.bit".
Creating bit mask...
Saving mask bit stream in "../xilinx/top_fpga.msk".
Bitstream generation is complete.
cp ../xilinx/top_fpga.bit ../bit/top_fpga_xcs05-4-
pc84_flash.bit
Generating AVR C Configuration file
promgen -u 0000 ../bit/top_fpga_xcs05-4-pc84_flash.bit -p
bin -c FF -o ../bit/top_fpga_xcs05-4-pc84_flash.bin -w
Release 4.2.03i - Promgen E.35
```

```
Copyright (c) 1995-2001 Xilinx, Inc.  All rights reserved.
0x1a5c (6748) bytes loaded up from 0x0
Using generated prom size of 8K
Writing file "../bit/top_fpga_xcs05-4-pc84_flash.bin".
Writing file "../bit/top_fpga_xcs05-4-pc84_flash.prm".
cvtfpga3.exe ../bit/top_fpga_xcs05-4-pc84_flash.bin ../brd
/top_fpga_xcs05-4-pc84_flash.c -t AVRILOS
Ilialex Research Lab, Convert FPGA V3.01
Convert a binary file (ie/ .bin, xsvf etc) to a C file
Convert an intel Hex file (ie/ .hex, mcs etc) to a C file
To be used for CPLD/FPGA in embedded systems
Input File: ../bit/top_fpga_xcs05-4-pc84_flash.bin
Output File: ../brd/top_fpga_xcs05-4-pc84_flash.c
Base: ../bit/top_fpga_xcs05-4-pc84_flash.bin, Ext: ../bit
/top_fpga_xcs05-4-pc84_flash.bin
Input File ok
Output File ok
Output Header File ok: ../brd/top_fpga_xcs05-4-pc84_flash.h
Total Bytes Written: 6748
```

After the makefile finishes you may copy the output files from brd/ to your software development tree. For **AVRILOS** you should copy the .c/.h files to the ../../sw/avr16/src/applic/cfg. But more on these steps we will discuss in the next article.

Conclusion

In this article I presented a simple flow for embedding FPGA/CPLD configuration data in C source code. I also provided an example makefile that controls compilation of Xilinx FPGA using command line only. Some pointers to relative documents and code are provided as a reference to build your own code. I also provided a consolidation tool that gets direct FPGA/CPLD tool output and generates the corresponding C file with proper definition of variables ready to integrate to **AVRILOS**. On the next article I will describe more in depth the **AVRILOS** & FPGA integration and control.

History

- Version 1.0, Initial Release.

License

About the Author

More than 10 year of Embedded Systems development designing both hardware & software for products, lab prototypes and Automated Testers. Have used numerous micro-controllers/processors, DSP & FPGAs.
More info you can find at my personal site: Ilialex