CERN control group cores and tools

Tristan Gingold

CERN BE-CEM-EDL

tristan.gingold@cern.ch

1st FPGA Developers' Forum
June 11, 2024

Presentation Overview

1. Introduction
2. HDLmake
3. Cheby
4. General Cores

Foreword

The slides could be used as a quick introduction to the HDLmake and Cheby tools.

To be understood without the one-man show presentation, the text is mostly written with sentences

So contrary to guidelines or tutorials, the slides are not drawn to support the presentation

Who is BE-CEM-EDL ?

Part of the control group(s)

- Electronic and low-level software for control
- Previously BE-CO-HT

Main projects:

- White-Rabbit (sub-ns synchronization over Ethernet)
- DI-OT (crate and kit for radiation-exposed and radiation-free areas)
- `ohwr.org` Open-Hardware Repository
- CERN OHL licenses

Many less well-known projects (SVEC, SPEC, FMCs, VME core...)

Particularities of the WR project

- Long project: over 15 years of existence
- **Multi-target**: Xilinx (ISE and Vivado), Altera
  - Cannot use methodology from a vendor
- HDL + Software
- IP but also full design (switch)
- Many asynchronous blocks

HDLmake

Describing sources

All HDL tools (simulator, synthesizers) have a notion of project to define:

- list of source files (HDL, constraints, block diagram, ...)
- options, flags, target...
- actions to perform

They all know TCL (so 90s)

But syntax/commands, options, and actions differ

Describing sources

Describing the source files is not that simple:

- Probably you know your own files
- But you'd like to organize freely (dirs and subdirs)
- You may know your own reusable IPs
- Maybe less external IPs

The source files of an IP should be declared in the IP

The source files must be declared once (and not once per tool)

HDLmake Manifest.py

Manifest.py

```
files = [
    'board_map.vhd', 'host_map.vhd',
]
```

It is a python file, ideally using only declarative syntax. Some variables like `files` are then used by `HDLmake`

HDLmake Manifest.py

Python statements may still be useful:

Manifest.py

```
if (syn_device[0:4]=="xc6s"): # Spartan6
    files.extend(["spartan6/wr_gtp_phy.vhd"])
elif (syn_device[0:4]=="xc6v"): # Virtex6
    files.extend(["virtex6/wr_gtx_phy.vhd"])
]
```

HDLmake modularity

In one `Manifest.py`, it is possible to refer to another one:

Manifest.py

```
modules = {
  'local': ['../top', '../my-lib'],
}
```

`HDLmake` will consider all the source files

HDLmake top-level

Action and options are in the top-level `Manifest.py`:

Manifest.py

```
action = "synthesis"

syn_device = "xczu4cg"
syn_package = "-sfvc784"
syn_top = "my_top"
syn_project = "my_project"
syn_tool = "vivado"

files = [ 'constraints.xdc' ]

modules = { 'local': ['../top'], }
```

HDLmake usage

Shell

```
$ hdlmake
```

`HDLmake` reads the `Manifest.py` and generate a `Makefile`

To do generate a bit-stream or to analyze the files:

Shell

```
$ make
```

To simply create the project file (ready to use by FPGA tool):

Shell

```
$ make project
```

IP cores with HDLmake

Can specify sources from an external repository:

Manifest.py

```
modules = {
 'local': ['../../top/vme'],
 "git" : [
   "https://ohwr.org/project/general-cores.git" ]
}
```

Use `$ hdlmake fetch` to clone or update the repository

Will be cloned in the `fechto` directory, which is a variable defined in the top `Manifest.py`

IP cores with HDLmake (our trick)

We use git submodules instead and manage the IP cores directly using `git` command

For `HDLmake`, the modules will always be ready to use

Additional trick: you can override the value of `fechto`:

Manifest.py

```
$ hdlmake --fetchto=../../..
```

If you have many projects, you can easily share the IPs, no need to clone them several times

HDLmake - conclusion

- It gathers source filenames and create project files
- Can also be used with your own TCL script just to gather the source files
- Automatically discard unused sources (useful with external IPs)
- Do not enforce a particular methodology
- Highly flexible (eg: generate sha1 file to embed)

Similar tools: `FuseSoC`, `Hog`

Cheby

Memory map

A very common problem:

- Many designs have a (or several) register maps
- It is tedious to write
- Needs a software view (usually `#define` or a `struct`)
- Keep HDL and software views in synch

Several tools have been written, including at CERN: Cheburashka, wbgen

When you have several tools for the same purpose, usually a new is created to merge the existing ones...

Cheby - header

example.cheby

```
memory-map:
  bus: wb-32-be
  name: example
  description: An example of a cheby memory map

...
```

- The input file is YAML
- The structure of the file is regular (name, description, children)
- The header specifies the slave bus (WB, AXI4-Lite, Avalon...)
- ... width and endianness

Cheby - body

example.cheby

```
...

 children:
    - reg:
        name: regA
        description: The first register
        width: 32
        access: rw
```

- the map is composed of registers (other elements exist)
- the access mode is valid for the whole register

Cheby - fields

A register can have fields:

example.cheby

```
...
          children:
           - field:
               name: field0
               description: 1-bit field
               range: 1
```

- the range is explicit and define the width of the field
- fields cannot overlap

Cheby - multi bit fields

example.cheby

```
- reg:
    name: div
    comment: clock divider for the spi
    width: 32
    access: rw
    children:
    - field:
        name: val
        range: 7-0
```

Cheby - define behaviour

Usually a `reg` behaves like a register if the access mode is `wr` or like an input if the access mode is `ro`.

It is possible to refine the behaviour:

example.cheby

```
- reg:
    name: ident
    comment: magic number to identify the core
    width: 32
    access: ro
    preset: 0x52544430
    x-hdl:
      type: const
```

Cheby - extensions

Tags whose name start with `x-` are extensions

- they have no influence on the layout
- they are directive for a particular generator
- here `x-hdl` is for the HDL generate
- they can be ignored by other generators

This system makes the input file flexible

Cheby - attributes for reg/field

`type: autoclear` will automatically clear the field, so a pulse is generated (usually the range is a single bit)

example.cheby

```
    - field:
      name: start
      comment: Start transmission
      range: 0
      x-hdl:
        type: autoclear
```

`type: or-clr` allows to capture pulse. The field (usually one bit) is or-ed with the input and is cleared by writing a `1`

Cheby - full control

The attribute `type: wire` removes the flip-flop; you usually need to also add `write-strobe` to know when the register is addressed. You can also use `read-strobe`

example.cheby

```
- reg:
    name: dout
    comment: Data to send (bit 31 first)
    access: wo
    width: 32
    x-hdl:
      type: wire
      write-strobe: True
```

With this mechanism you have full control; you can define different behaviour for reads and for writes

Cheby - generation

From the memory map definition (single source of truth), the tool can generate:

- HDL: vhdl, verilog or SystemVerilog
- HDL constant files: addresses and offsets of the registers, useful when writing testbenches
- constant file in python or TCL: likewise
- C header file: either `#define` or `struct` that represent the memory map (for drivers)
- documentation: html, md, reset
- simple text output: useful for debugging or reviewing

Cheby - Extra elements

In addition to registers, a memory map can have:

- memory: either external RAMs or instantiated ones
- repetition: for multiple registers
- submap: re-use of an existing cheby memory map (or external bus)
- block: just a way to group elements

Cheby - submap

`submap` makes Cheby modular: you can integrate existing IP memory maps into your core.
Cheby will automatically create the bus conversions if needed.

example.cheby

```
children:
- submap:
    name: host
    address: 0x000000
    filename: host_map.cheby
```

General Cores

General Cores

Gather various small cores and infrastructure

- Wishbone (WB) and AXI4 records
- bus conversion (WB/AXI4)
- WB infrastructure (crossbar, adapter, clock domain)
- I2C, SPI, UART, OneWire
- synchronizer
- reset generator
- memories
- FIFO
- 8b10 decoder
- ECC and voters
- Cordic, FIR

General Cores

Why a single repository with many cores ?

- too small to deserve a separate repository
- much easier to add a new core (no need to create a new setup)
- easier to browse them
- easier to use them (add only one link)
- each core is stable
- `HDLmake` only adds referenced file in your project
- common style (suffix, names, ...)

RAM and FIFO from General Cores

- generic memories (width + size)
- implementation can be vendor specific (but hidden)
- dual port memories
- dual clock memories
- initialized memories
- likewise for FIFO (sync, async)

Synchronizers

Handle metastability for an asynchronous input

- two flip-flops in a raw
- with appropriate directives
- and script to generate constraints

Block to safely transfer a word across two clock domains

Reset generation for multiple clocks

Wishbone

We mainly use Wishbone as a SoC bus

- The specification is free, no licence
- Rather simple

Connecting buses wire by wire is tedious

- SystemVerilog has interfaces
- Records can be used in VHDL

Multiple cores for wishbone infrastructure

- crossbar
- adapters (protocol, clocks)
- pipeline

SPI, I2C...

- These cores come from OpenCore
- A wrapper has been added (for style)
- Some with modifications
- Linux drivers
- Testbench

The End

Questions? Comments?