Cheby User Guide

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	2018-09-12		C

Contents

Chapter 1

What is Cheby ?

Cheby is both a text description of the interface between hardware and software, and a tool to automatically generate code or documentation from the text files.

In Cheby, the hardware appears to the software as a block of address in the physical memory space. There might be other way to interface hardware and software (for example through a standard serial bus like USB or through a network).

The block of address is named the memory map of the hardware. The memory map is a map between addresses and hardware elements like registers or memories.

The hardware elements supported by Cheby are:

- Registers. A register uses one word of memory (usually 32 bits) or two (so 64 bits), and is divided into fields (a group of bit). Some bits of the registers can be unused. The difference between a register and a memory is that hardware has direct access to a register, there are wires between the register and the hardware so as soon as the software writes to a register the hardware 'can' see the new value. A register is usually read-write: the value of the register is defined by the last write (from the software) and the software always reads the last value. A register can also be read-only: the hardware defines the value that is read. It is also possible that a read triggers some changes in the hardware. Finally a register can be write-only, and usually a write triggers an action. In that case, a value read has no meaning.
- Memories. A memory is like a RAM memory except that hardware also has access to it (through a second port, hence the name dual port). Memories are used when a certain amount of data has to be transfered or to configure hardware for data transfers (like DMA descriptors). To avoid possible conflicts, memories are usually one direction: the software can read and the hardware can write, or the software can write and the hardwire can read.
- Submap. A submap is a sub-block of the memory map (an aligned continuous range of address) either defined by an external file or will be available to the hardware designer. Submaps make possible to create a hierarchy of blocks and to create custom blocks.

A fundamental feature of the Cheby text description is non-ambiguity: the memory map is defined by the file and there is only one way to assign addresses to hardware elements.

Once the text file is written it is possible to invoke the cheby tool to generate:

- C headers
- Device drivers
- HDL code
- HTML documentation

The automatic generation of these files avoid a tedious work and ensure coherency between them.

Chapter 2

User Starting Guide

Let's work on a very simple design: a counter. The hardware increments a countewr every cycle until it reaches a maximal value, then it starts again from 0.

As a designer, you have to implement the counter but you can use Cheby to generate the interface. The counter needs:

- A one bit register to enable/disable it.
- A 32-bit register containing the maximal value
- A 32-bit register with the current value.

In this user starting guide only the `cheby` command line tool is used, and the input file is created by any text editor.

Let's assume the design uses the wishbone bus, and create a Cheby file that describes the above elements.

```
memory-map:
  bus: wb-32-be
  name: counter
  description: A simple example of a counter
  children:
    - reg:
        name: control
        description: Counter control
        width: 32
        access: rw
        children:
          - field:
              name: enable
              description: Set to enable the counter
              range: 0
    - reg:
        name: value
        description: Maximum value of the counter
        width: 32
        access: rw
    - reg:
        name: counter
        description: Current value of the counter
        width: 32
        access: ro
```

The description of the file format is documented later in this guide.

For the hardware designer, cheby can generate the hardware interface: for VHDL this is an entity and its associated architecture and for verilog this is a module. The hardware interface contains the wishbone bus, the registers described in the file, the decoding logic, and ports for the registers.

To generate VHDL:

```
  cheby --gen-hdl=counter.vhdl -i counter.cheby
```

Here is the entity part generated by the tool:

```
entity counter is
  port (
    rst_n_i              : in    std_logic;
    clk_i                : in    std_logic;
    wb_cyc_i             : in    std_logic;
    wb_stb_i             : in    std_logic;
    wb_adr_i             : in    std_logic_vector(3 downto 2);
    wb_sel_i             : in    std_logic_vector(3 downto 0);
    wb_we_i              : in    std_logic;
    wb_dat_i             : in    std_logic_vector(31 downto 0);
    wb_ack_o             : out   std_logic;
    wb_err_o             : out   std_logic;
    wb_rty_o             : out   std_logic;
    wb_stall_o           : out   std_logic;
    wb_dat_o             : out   std_logic_vector(31 downto 0);

    -- Set to enable the counter
    control_enable_o     : out   std_logic;

    -- Maximum value of the counter
    value_o              : out   std_logic_vector(31 downto 0);

    -- Current value of the counter
    counter_i            : in    std_logic_vector(31 downto 0)
  );
end counter;
```

You can see the wishbone ports and the ports for the counter.

As an hardware designer, you have to write the HDL code for the counter logic. The enable and maximum value are given by the interface, and you should give the current value.

Because the interface is defined (and hopefully well documented), it is also possible for the software developer to start the software part. The SW developer needs to program the register and therefore to know the register map. So let's generate the corresponding C header:

```
#ifndef __CHEBY__COUNTER__H__
#define __CHEBY__COUNTER__H__

struct counter {
  /* [0x0]: REG Counter control */
  uint32_t control;

  /* [0x4]: REG Maximum value of the counter */
  uint32_t value;

  /* [0x8]: REG Current value of the counter */
  uint32_t counter;
};

#endif /* __CHEBY__COUNTER__H__ */
```

The absolute address of the counter is determined by the instantiation of this module, but you can refer directly to the registers.

In order for the end-user or a developper to have a better view of the design, it is better to read a documentation. A doc can be generated from the description file:

```
cheby --gen-doc=counter.html -i counter.cheby
```

For our example, the generated doc is:

2.1 Memory map summary

A simple example of a counter

HW address	Type	Name
0x0	REG	control
0x4	REG	value
0x8	REG	counter

2.2 Registers description

2.2.1 control

address: 0x0

Counter control

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	**enable**

2.2.2 value

address: 0x4

Maximum value of the counter

31	30	29	28	27	26	25	24
value[31:24]							
23	22	21	20	19	18	17	16
value[23:16]							
15	14	13	12	11	10	9	8
value[15:8]							
7	6	5	4	3	2	1	0
value[7:0]							

2.2.3 counter

address: 0x8

Current value of the counter

31	30	29	28	27	26	25	24
counter[31:24]							
23	22	21	20	19	18	17	16
counter[23:16]							
15	14	13	12	11	10	9	8
counter[15:8]							
7	6	5	4	3	2	1	0
counter[7:0]							

Chapter 3

Cheby File Format

3.1 General Structure

The Cheby file format represent a hierachy of nodes. A node contains a list of attribute and children. An attribute is designated by a name and has a value (a string, a boolean or an integer). The children are nodes, organized as a list.

The Cheby file format described in this manual is based on YAML, so that there is no new format to invent and many text editors have already support for it. However the file extension is usually `.cheby`.

The nodes are `memory-map`, `reg`, `field`, `array`, `block` and `submap`.

Some attributes are common to all nodes:

- `name`: The name of the node. This is required for all nodes. The name is also used to create HDL or C names in general files.
- `description`: This should be a short text that explain the purpose of the node. The description is copied into the code (as a comment) to make it more readable. This attribute is not required but it is recommended to always provide it.
- `comment`: This is a longer text that will be copied into the generated documentation.
- `children`: For nodes that have children, this is a list of the children.
- `address`: An optional address relative to the parent. The address must be correctly aligned. If not provided or it the value is `next`, then the address is computed using the previous one and the alignment. It is possible to go backward by providing explicit address (eg: the first child has address 4 and the second one has address 0), but this is not recommended and be a source of errors (in particular with automatic addresses that are always computed from the previous node). Overlapping addresses are detected by the tools.
- `x-NAME`: Extensions for tool or feature `NAME`. The Cheby file format is extensible so that new tools can be easily created without backward compatibility issues.
- `x-hdl`: Extensions for hdl generation.
- `x_gena`: Extensions for Gena compatibility.
- `x-wbgen`: Extensions for wbgen compatibility.

3.2 Header

A cheby file is an associative array named `memory-map`. The only purpose of this name is to easily refuse a random YAML file.

The `bus` attribute specifies which bus will be used to interface the CPU with the HW module. It can be:

- wb-32-be: non-pipelined wishbone with 32 bit of data using the big-endian convention
- axi4-lite-32: AIX4 lite bus with 32 bit of data
- cern-be-vme-SZ: CERN VME-like bus using `SZ` data bit. `SZ` can be 8, 16 or 32.

If you need to reserve area in a module, you can use the `size` attribute to specify the size (in bytes) of the memory space used by the module.

The following attributes under `x-hdl` are supported:

`busgroup`
: Use an input and an output record (in vhdl) for the bus.

`iogroup`
: Use an input and and output record (in vhdl) for the I/O. This will also generate a package to declare the records.

`reg_prefix`
: If false, discard register prefix and every prefix before register. This creates shorter names.

`block_prefix`
: If false, discard block prefix.

3.3 Registers

A register uses one (usual case) or two (for 64-bit registers) words address. It can be directly read or written by the CPU and each used bit or group of bits generates a port.

The access mode is defined from the point of view of the software. It slightly change the generated hardware:

- rw (read/write): This generates flip-flops whose value is directly available to the hardware. The software can write to modify the value or read the get the current value. The hardware cannot change the value.
- wo (write-only): Like `rw`, but the software cannot read the current value.
- ro (read-only): This creates no hardware but just a port. The software can read the current value of the port, and cannot modify it.

The size (in bits) of the register can be specified by the `width` attribute. The size cannot be larger than two words.

It is possible to have fields in a register. A field is a group of consecutive bits and has a name.

If there is no field, this is a plain register.

The following attributes under `x-hdl` are supported:

`type`
: if set to `wire` no register is created. This is the default when the access type is `ro`.

`write-strobe`
: True to generate an additional signal that is asserted when the register is written by the host. The name of the signal is '_wr_o' appended to the name of the register.

`read-strobe`
: True to generate an additional signal that is asserted when the register is read by the host. The name of the signal is '_rd_o' appended to the name of the register.

3.3.1 Plain Registers

A plain register has a type, which could be `unsigned` (the default), `signed` or `float`. The type has no impact on the hardware, but changes the software view.

It is possible to define the initial (just after a reset) value of a register using the `preset` attribute.

Example of a plain register:

```
- reg:
    name: reg1
    description: a reg without fields
    width: 32
    access: rw
    type: unsigned
    preset: 0x123
```

3.3.2 Fields

There can be several fields in a register, and all of them have the same access right. Bits used by a field are specified by the `range` attribute. The range is a single number if the field is 1 bit, or in the form of `lo-hi` where `lo` is the lowest bit and `hi` is the highest bit. Bits are numbered using the little endian convention.

It is possible to define the initial value of a field using the `preset` attribute.

Example of a register with two fields:

```
- reg:
    name: reg0
    description: a normal reg with some fields
    width: 32
    access: rw
    children:
      - field:
          name: field0
          description: 1-bit field
          range: 1
      - field:
          name: field1
          description: a field with a preset value
          range: 10-8
          preset: 2
```

3.4 Blocks

A block is simply a group of elements (registers, rams, submaps or blocks). This is used only to create a hierarchy, but also offers the possibility to specify an address or an alignment.

It is possible to reserve space at the end of a block with the `size` attribute. It specifies the size (in bytes) of the block.

Example of a block:

```
- block:
    name: block1
    description: A block of registers
    children:
    - reg:
       name: b1reg0
       access: wo
       width: 32
```

3.5 RAMs

A RAM is represented by an array of a register. The number of elements is specified by the `repeat` attribute (usually it should be a power of 2).

The `align` attribute must be set to `true`.

Example of a ram:

```
- array:
    name: ram_ro1
    repeat: 32
    children:
      - reg:
          name: value
          access: rw
          width: 32
```

3.6 Repetition

It is possible to replicate elements using an `array` with the `align` attribute set to `false`.

Be careful that such a repetition can generate a lot of hardware.

Example of a replication:

```
- array:
    name: arr1
    repeat: 2
    align: False
    children:
      - reg:
          name: areg1
          access: rw
          width: 32
```

3.7 Submap

If the `filename` attribue is not present, then this is a generic submap and a bus port is generated in the HDL. The size of the submap is required.

Example of a generic submap:

```
- submap:
    name: sub3
    size: 0x1000
    description: A bus
    interface: wb-32-be
```

If the `filename` attribute is present, the `size` attribute is not allowed as the size of the submap is defined by the memory map given by the file. If the `interface` attribute is present and set to `include`, then the memory map described by the file is included directly, otherwise a bus interface is generated in the HDL.

Example of a normal submap:

```
- submap:
    name: sub1
    description: A normal submap
    filename: demo_all_sub.cheby
```

Example of an included submap:

```
  - submap:
      name: sub2
      description: An included submap
      filename: demo_all_sub.cheby
      interface: include
```

Chapter 4

Generated HDL

For all buses, only word accesses are supported. Sub-word (byte or half-word) accesses are considered as word accesses (which can lead to error when writting).

Addresses are always byte addresses.

4.1 AXI4-Lite

There are several restrictions from the AMBA AXI standard:

- There can be no combinatorial paths between input and output signals (A3.2.1). So there must be at least one register.
- A source is not permitted to wait until READY is asserted before asserting VALID. Likewise, a destination is permitted to wait for VALID to be assert before asserting the corresponding READY (A3.2.1).

There are registers for each channel. The W and AW channels waits until both have a request, handle the request and then become ready when the B channel becomes ready.

Note that AXI4 addresses are byte addresses as specified in A3.4.1

4.2 Wishbone

The normal wishbone protocol is used (and not the pipelined one).

The `err` (error) and `rty` (retry) are always ignored (as inputs) and never asserted as output.

By default the wishbone interface uses one port per wishbone signal. It is possible to group all signals in one input and one output port by setting the `x-hdl.busgroup` attribute to `True`.

Chapter 5

Gena Compatibility

For Cheburashka/Gena users, there is a simple transition path. You can convert a regular Cheburashka XML file to the cheby format using `gena2cheby`:

```
$ gena2cheby FILE.xml
```

This tool writes on the standard output a cheby file. Note that this file contains several extensions (under `x_gena` arrays) so that all the feature of the XML file are kept.

It is possible to generate a VHDL file (that is very similar to the VHDL file generated by Gena) using cheby:

```
$ cheby --gen-gena-regctrl=OUTPUT.vhdl -i INPUT.cheby
$ cheby --gen-gena-memmap=OUTPUT.vhdl -i INPUT.cheby
```

Chapter 6

wbgen Compatibility

There is also a transition path for wbgen users. If you have a fully declarative and well formed wbgen file, you can convert it to the cheby file:

```
$ wbgen2cheby FILE.wb
```

This generate a cheby file on the standard output. Note that this file contains extensions using `x-wbgen` arrays.

It is possible to generate a VHDL file that is very similar to the VHDL file generated by wbgen using cheby:

```
$ cheby --gen-wbgen-hdl=OUTPUT.vhdl -i INPUT.cheby
```

Chapter 7

Cheby Command-Line Tool

The `cheby` tool can generate various files from an input file. The input file must be specified with the `-i` flag:

```
$ cheby ACTION1 ACTION2... -i INPUT.cheby
```

An action flag is a flag optionally followed by a file name. If the file name is not present, the result is sent to the standard output.

```
$ cheby --gen-hdl=output.vhdl -i input.cheby
$ cheby --gen-hdl -i input.cheby
```

7.1 Generating HDL

Either VHDL or verilog can be generated by `cheby`. You can specify the language (either `vhdl` or `verilog`) with the `--hdl` flag, the default being `vhdl`.

```
$ cheby --hdl=vhdl --gen-hdl=OUTPUT.vhdl -i INPUT.cheby
```

7.2 Generating EDGE file

You can generate an EDGE block definition (in CSV) with the `--gen-edge` flag. Note that you need to provide the other part of the files.

```
$ cheby --gen-edge -i INPUT.cheby
```

7.3 Generating C header

The definition of a C structure representing the layout of the design is generated using the `--gen-c` flag. This generates a header file and assumes that referenced designs (throught the submap feature of Cheby) have been generated.

```
$ cheby --gen-c -i INPUT.cheby
```

It is also possible to generate a C program that check the layout is the same as the layout seen by Cheby. This can be used as a consistency check.

```
$ cheby --gen-c-check-layout -i INPUT.cheby
```

7.4 Generating documentation

Documentation can be generated either in HTML or in markdown (tested with asciidoctor). The format is specified by the `--doc=FORMAT` flag and the format is either `html` or `md`.

```
$ cheby --doc=md --gen-doc=OUTPUT.md -i INPUT.cheby
```

7.5 Generating constants file

In order to write testbench, you can generate a verilog include files that defines the address of the registers, the offset and a mask for each fields.

```
$ cheby --gen-consts -i INPUT.cheby
```

7.6 Generating SILECS file

It is possible to generate SILECS (https://wikis.cern.ch/display/SIL/Design+document) XML file from a Cheby description. Not all features of Cheby are supported: only registers.

```
$ cheby --gen-silecs -i INPUT.cheby
```

7.7 Generating text files

It is possible to general a compact text file describing the layout of a cheby file with the `--print-memmap` flag. This could be useful to have a quick look on alignment effects.

```
$ cheby --print-memmap -i INPUT.cheby
```

To display the fields, use `--print-simple`:

```
$ cheby --print-simple -i INPUT.cheby
```

When a part of the design is replicated (using the `repeat` attribute) you can view the effect of it with the `--print-simple-expanded` flag.

```
$ cheby --print-simple-expanded -i INPUT.cheby
```

Finally to regenerated the initial file (properly indented but without the comments), you can use `--print-pretty` or `--print-pretty-expanded`.