AGWB - Address Generator for WishBone

Release 0.0.0

Wojciech M. Zabolotny

Jan 18, 2022

CONTENTS:

CHAPTER

ONE

OVERVIEW

Address Generator for WishBone (AGWB) is a tool for automatic management of local bus address space in complex FPGA systems. Its main advantage, comparing to other open source solutions, is the support for complex hierarchical systems. Fig. 1.1 shows an example of a hierarchical system.

1.1 Local bus

As the local control bus, the Wishbone bus was chosen. It is used in the classic single mode. In this mode, it may control both the Wishbone and IPbus slaves, which gives access to multiple open IP cores. It is possible to control the local bus from the IPbus master. Additionally, there are bridges providing control of the Wishbone bus from other busses like Avalon or AXI. Therefore, such selection of local bus ensures high versatility and flexibility of the created control infrastructure, which is desirable, even though it provides lower performance than pure AXI bus.

1.2 Allocation of adresses

To enable optimal implementation of address decoders the address space for each block requiring the K addresses, where $0 < K < 2^N$ is aligned to the 2^N boundary so that N bits are used for internal addressing in the block. To ensure efficient utilization of the address space, the required size of the address space for each block is calculated, traversing the system description from the most nested blocks to the top. After that, the blocks are ordered in the order of decreasing size of their address space, and their base addresses are set with the proper alignment.

1.3 Input

As an input AGWB accepts system registers structure described in *.xml* format. This is further described in *XML description* chapter.

1.4 Output products

AGWB always generates VHDL files appropriate to the defined blocks. User should expect following VHDL files to be generated.

1. *{top_name}_const_pkg.vhd* - package with constants defined in input *.xml* files.
2. *{block_name}_pkg.vhd* - package for given block. Packages for distinct blocks are generated into distinct files. These packages contain various constants, subtypes, types definitions and conversion functions related to given block.

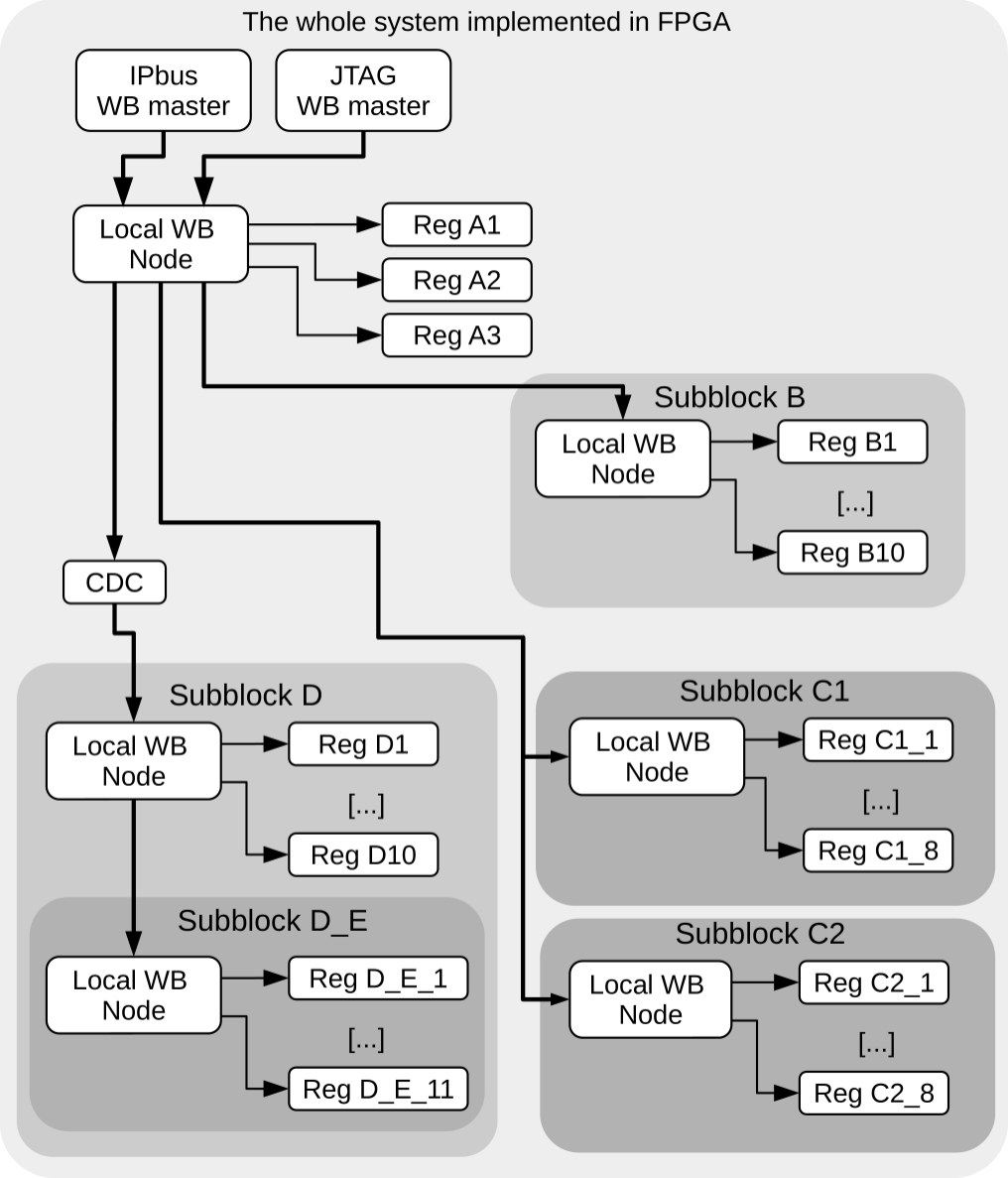

Fig. 1.1: The block diagram of an example design built in the FPGA using the AGWB. The CDC block provides the clock domain crossing functionality. It allows subblocks D and D_E to run with another clock than the rest of the system.

3. *{block_name}.vhd* - entity for given block. Entities for distinct blocks are generated into distinct files.

Depending on the input arguments AGWB can also generate following helper files.

1. IPbus compatible register files.
2. C header files for ???.
3. Python files for ???.
4. Forth files for ???.
5. HTML registers documentation file.

To get to know how to generate these files execute `python addr_gen_wb.py --help`.

1.5 License

The code is licensed under GPL v2 license. The generated code is free, and you can freely use it in your design.

CHAPTER

TWO

INSTALLATION AND USAGE

AGWB is not installable from PyPI via pip. To be able to use AGWB you have to clone the repository from https://github.com/wzab/addr_gen_wb and use `src/addr_gen_wb.py` in a script way.

Although AGWB is not installable from PyPI via pip it is possible to install some parts of it. Namely, the `agwb` Python package, which can be used for interaction with flashed FPGAs or simulation purposes. To install the package execute the following command in the repository root directory.

```
python setup.py install --user
```

2.1 AGWB and FuseSoc

AGWB can be used as a FuseSoc (https://github.com/olofk/fusesoc) generator. Following snippet from *.core* file can serve as an example.

```
CAPI=2:

# Choose whatever name you want.
# If you have only single AGWB description of the system
# within your design, then it is good to use 'agwb' name.
name: ::agwb

filesets:
  agwb_dep:
    depend:
      - wzab::addr_gen_wb

targets:
  default:
    generate:
      - agwb_regs
    filesets:
      - agwb_dep

generate:
  agwb_regs:
    generator: addr_gen_wb
    parameters:
      infile: top.xml
      # hdl parameter is optional. If you don't provide it
```

(continues on next page)

(continued from previous page)

```
    # VHDL files will be generated to the FuseSoc cache directory.
    hdl: ./optional/relative/path/for/generated/vhdl/files
    # Below commented parameters are programming language specific
    # and are also optional. Unlike 'hdl' parameter, if they are not
    # provided particular files will not be generated.
    # Paths for generated output files in all below parameters are relative.
    # header: c_headers/destination
    # html: html/destination
    # ipbus: ipbus_outputs/destination
    # python: python_raw/destination
    # fs: Forth_outputs/destination
```

CHAPTER

THREE

XML DESCRIPTION

AGWB uses `XML` as file format for describing system registers. There are predefined element names, that must be used for valid description. Each element requires at least one mandatory attribute and may accept some optional attributes.

3.1 Why XML?

You may be wondering why `XML`, why not someting more buzzy like `JSON` or `YAML`. Well, `XML` has something, that is particularly useful for describing hierarchical systems that neither `JSON` nor `YAML` have. Namely, it distinguishes block attributes and elements. Attributes function like a metadata of a block and elements are independent entities existing within outter element/block. With `JSON` or `YAML` one would have to imitate this structure and it would feel somehow less intuitive. `JSON` also does not support comments, and with `YAML` it is a bit harder to spot indentation mistakes in long blocks.

3.2 Valid Elements

3.2.1 blackbox

`blackbox` element can be used for incorporating registers or blocks not generated by the AGWB.

Mandatory attributes:

1. `name` - name of the blackbox instance within outter block.
2. `type` - type of the blackbox.
3. `addrbits` - number of lower address bits used by blackbox for internal addressing.

Optional attributes:

1. `desc` - Text describing the block
2. `reps` - Number (or semicolon separated list of numbers, if you use *variants*) defining the number of repetitions (may contain "0" if in certain variant the block is not used). Presence of this attribute enforces implementation as the vector of blocks (even if the length 1 or 0).
3. `used` - Number (or semicolon separated list of numbers, if you use *variants*) defining if the block is used ('1') or not used ('0'). Other values are prohibited. This attribute replaces *reps* for objects that should not be converted into vectors.
4. `xmlpath` - relative path to *.xml* file with registers not generated by the AGWB.

3.2.2 block

`block` element is used for grouping registers or other blocks.

Mandatory attribute:

1. `name` - name of the block.

#. `name` - Optional attributes: #. `aggr_outs` - If set to one, all outputs of the block are aggregated into a single output named *out_regs*. That functionality is useful, if you need to route the output of the block to another VHDL entity (you route a single record signal instead of multiple signals). #. `reserved` - This optional argument reserves certain number of words at the begining of the address space. #. `testdev_ena` - If this attribute is set to the true (non-zero) value, it enables generation of a test device at the begining of the address space of the block. #. `desc` - Text describing the block #. `ignore` - This attribute informs that this block, and all its children should be ignored in certain backend. Currently only the 'forth' value has a meaning. It protects the Forth vocabulary against the overflow by multiple names that are not going to be used by the Forth CPU. The value may be overriden by *ignore* argument in a subblock or a child.

Example

```
<block name="top">
   <blackbox name="EXTHUGE" type="HTEST" addrbits="16" />
   <subblock name="LINKS" type="SYS1" reps="NSEL_MAX"/>
   <creg name="CTRL" desc="Control register in the top block" default="0x11">
      <field name="CLK_FREQ" width="4"/>
      <field name="PLL_RESET" width="1"/>
   </creg>
</block>
```

`creg` can contain `field` elements, see *field*.

3.2.3 constant

`constant` allows for definig constant number, which can later be used for parameterizing registers or blocks.

Mandatory attributes:

1. `name` - name of the constant.
2. `val`- value of the constant.
3. `desc` - extra description of the constant.

Example

```
<constant name="NUMBER_OF_BITS" val="3" />
```

3.2.4 creg

`creg` stands for *control register* and should be used to describe registers that are supposed to be both written and read by the software.

Mandatory attribute:

1. `name` - name of the control register.

Optional attributes:

1. `default` - default value stored in the register, this value is also applied after reset.

2. `desc` - extra description of the control register.
3. `mode` - special attribute, directly passed to the generated IPbus XML file.
4. `reps` - Number (or semicolon separated list of numbers, if you use *variants*) defining the number of repetitions (may contain "0" if in certain variant the block is not used). Presence of this attribute enforces implementation as the vector of registers (even if the length 1 or 0).
5. `used` - Number (or semicolon separated list of numbers, if you use *variants*) defining if the register is used ('1') or not used ('0'). Other values are prohibited. This attribute replaces *reps* for objects that should not be converted into vectors.
6. `stb` - setting this to 1 enables the *stb* signal, that is asserted for one clock pulse whenever the new value is written. Useful for FIFO write.
7. `stype` - Allows the user to define non-standard type name for the register. Otherwise the type is obtained from the register name, which may lead to collisions in the HDL namespace (if two blocks, have registers with the same name, but with different fields, width, or other properties).
8. `type` - type of the register. The default value is 'std_logic_vector'. May be also seto to 'signed' or 'unsigned'.
9. `width` - width of the register in bits.
10. `ignore` - This attribute informs that this register, and all its fields should be ignored in certain backend. Currently only the 'forth' value has a meaning. It protects the Forth vocabulary against the overflow by multiple names that are not going to be used by the Forth CPU. The value may be overriden by *ignore* argument in a field.

3.2.5 field

`field` element is used to define bit fields within register.

Mandatory attributes:

1. `name` - name of the field.
2. `width` - width of the field in bits.

Optional attribute:

1. `type` - type of the bit field. The default value is 'std_logic_vector'. May be also seto to 'signed' or 'unsigned'.
2. `desc` - description of the bit field.
3. `default` - defaut value of the bit field.
4. `ignore` - This attribute informs that this field. Currently only the 'forth' value has a meaning. It protects the Forth vocabulary against the overflow by multiple names that are not going to be used by the Forth CPU.
5. `trigger` - This attribute if set to true, informs that this field is a 'trigger'. It means, that if written, the written value is available only for one clock period. It is always read as zero. The 'trigger' fields should be used for launching certain actions in the hardware.

Example

```
<sreg name="throughput">
   <field name="val" width="30" type="unsigned" />
   <field name="prev_missed" width="1" />
   <field name="read" width="1" />
</sreg>
```

3.2.6 include

`include` element allows including *.xml* files. This is very useful functionality, as different modules can be placed in different repositories and reused in different projects. Each module (entity) can have its own *.xml* file with block definition related strictly to this module.

Example

```
<include path="relative/path/to/block.xml"/>
```

3.2.7 sreg

`sreg` stands for *status register* and should be used to describe registers that are supposed to be read only by software.

Mandatory attribute:

1. `name` - name of the status register.

Optional attributes:

1. `ack` - setting this to 1 enables the *ack* signal, that is asserted for one clock pulse when the value is read.
2. `desc` - extra description of the register.
3. `mode` - special attribute, directly passed to the generated IPbus XML file.
4. `reps` - Number (or semicolon separated list of numbers, if you use *variants*) defining the number of repetitions (may contain "0" if in certain variant the block is not used). Presence of this attribute enforces implementation as the vector of registers (even if the length 1 or 0).
5. `used` - Number (or semicolon separated list of numbers, if you use *variants*) defining if the register is used ('1') or not used ('0'). Other values are prohibited. This attribute replaces *reps* for objects that should not be converted into vectors.
6. `stype` - Allows the user to define non-standard type name for the register. Otherwise the type is obtained from the register name, which may lead to collisions in the HDL namespace (if two blocks, have registers with the same name, but with different fields, width, or other properties).
7. `type` - type of the register. The default value is 'std_logic_vector'. May be also seto to 'signed' or 'unsigned'.
8. `width` - width of the register in bits.
9. `ignore` - This attribute informs that this register, and all its fields should be ignored in certain backend. Currently only the 'forth' value has a meaning. It protects the Forth vocabulary against the overflow by multiple names that are not going to be used by the Forth CPU. The value may be overriden by *ignore* argument in a field.

Example

```
<sreg name="my_reg" ack="1" default="0x0" desc="Some diagnostic registers." reps="8"␣
↪width="16" />
```

`sreg` can contain `field` elements, see *field*.

3.2.8 subblock

`subblock` element is used to include some block into another block.

Mandatory attributes:

1. `name`- name of the subblock instance within outter block.
2. `type`- type of the subblock. This is name of the subblock definition.

Optional attributes:

1. `desc` - extra description of the subblock.
2. `ignore` - This attribute informs that this register, and all its fields should be ignored in certain backend. Currently only the 'forth' value has a meaning. It protects the Forth vocabulary against the overflow by multiple names that are not going to be used by the Forth CPU. The value may be overriden by *ignore* argument in a field.
3. `reps` - Number (or semicolon separated list of numbers, if you use *variants*) defining the number of repetitions (may contain "0" if in certain variant the block is not used). Presence of this attribute enforces implementation as the vector of registers (even if the length 1 or 0).
4. `used` - Number (or semicolon separated list of numbers, if you use *variants*) defining if the register is used ('1') or not used ('0'). Other values are prohibited. This attribute replaces *reps* for objects that should not be converted into vectors.

Example

```
<block name="data_processing">
   <creg name="enable" width="1" />
   <sreg name="throughput" reps="9" ack="1">
      <field name="val" type="unsigned" width="30" />
      <field name="prev_missed" width="1" />
      <field name="read" width="1" />
   </sreg>
</block>

<block name="wfifo">
   <creg name="data" mode="non-incremental" stb="1" />
   <sreg name="unused" ack="1" type="unsigned" />
   <sreg name="valid_writes" type="unsigned" />
</block>

<block name="main">
   <subblock name="write_fifo" type="wfifo" desc="Some extra description." />
   <subblock name="dproc" type="data_processing" reps="2" />
</block>
```

3.2.9 sysdef

`sysdef` must be a root element.

Mandatory attribute:

1. `top` - designates the block which should be used as a top level for registers generation.

Optional attribute:

1. `masters` - number of Wishbone masters controlling the local bus (default value is 1).

Example

```
<sysdef top="foo" masters="2">
   <block name="foo">
      ...
   </block>

   <block name="bar">
      ...
   </block>
</sysdef>
```

3.3 Math within attribute value

The attribute values may be specified as a valid Python number, ar as a valid Python expression. The expressions are evaluated using the code based on https://stackoverflow.com/a/30516254/1735409 . Therefore, only certain subset of Python functions are available. The expression make make use of the constants defined in the system description XML. However, one must be aware, that as epxressions are stored in the XML file, so certain characters mas be escaped:

```
& with &amp;
< with &lt;
> with &gt;
" with &quot;
```

That may affect legibility of certain expressions. For example the expression: `1 << ADDRWIDTH` must be written as `1 << ADDRWIDTH`

3.4 Notes

3.4.1 reps attribute

The `reps` attribute is used for defining vectors of blocks/registers. It enforces the implementation of the particular instance to be treated as a vector even if the value equals 1 or 0. This is useful for parametrized designs, when sometimes the parameter describing the number of implemented blocks or registers may equal 1, and sometimes may equal value greater than 1. With such approach implemented codes are very flexible and need no modification.

3.4.2 ignore attribute

The `ignore` attribute is used for ignoring generation of definitions for certain blocks for particular backends. The atribute may be specified either in the definition of the block (igoring all its instances) or in the instantiation of the block, or in the definition of a register. Currently `ignore` attribute has effect only in case of Forth backend. It is possible to extend that functionality to other backends.

Example

```
<block name="my_block">
  <subblock name="links" type="sys1" reps="N_SEL_MAX" ignore="forth"/>
  <subblock name="olinks" type="sys1"/>
</block>
```

3.4.3 variants

To be described. At the moment, please look at https://github.com/wzab/agwb/wiki/Multiple-AGWB-trees for information about the reason for introducing the variants.

CHAPTER

FOUR

VHDL

AGWB generates VHDL files appropriate to the defined blocks (see *Output products*). When AGWB is used as a FuseSoc generator all auto generated VHDL files are put into separate `agwb` library. If user generates code using script directly (without FuseSoc), the generated files can be put into any library. However, it is recommended to always put auto generated VHDL files into dedicatd `agwb` library, even if FuseSoc is not used. This makes the design more readable and facilitates the maintenance.

4.1 Conversion functions

In VHDL there is often a need to convert objects to different types. AGWB automatically generates functions for converting to `std_logic_vector` and custom types defined in .xml files.

Example

Assume there is following block defined in the .xml file.

```
<block name="my_block">
   <creg name="my_creg">
      <field name="field_1" width="5"/>
      <field name="field_2" width="3"/>
   </creg>
</block>
```

Then following declarations and definitions will be automatically generated and available in the `my_block_pkg.vhd` file.

```
type t_my_creg is record
   field_1 : std_logic_vector(4 downto 0);
   field_2 : std_logic_vector(2 downto 0);
end record;

function to_my_creg(x : std_logic_vector) return t_my_creg;
function to_slv(x : t_my_creg) return std_logic_vector;

-- Definitions from the package body.
function to_my_creg(x : std_logic_vector) return t_my_creg is
variable res : t_my_creg;
begin
   res.field_1 := std_logic_vector(x(4 downto 0));
   res.field_2 := std_logic_vector(x(7 downto 5));
   return res;
```

(continues on next page)

(continued from previous page)

```
end function;

function to_slv(x : t_sx_mask_enc_mode) return std_logic_vector is
variable res : std_logic_vector(7 downto 0);
begin
   res(4 downto 0) := std_logic_vector(x.field_1);
   res(7 downto 5) := std_logic_vector(x.field_2);
   return res;
end function;
```

CHAPTER

FIVE

PYTHON

If `--python` argument is specified, AGWB generates special `agwb` package, which can be used for simulation or interaction with real hardware. To be able to import the package it must be in the path. It is left for the user how it is achieved.

The hardware-related structure of blocks and registers is represented as a nested structure of proper classes and attributes. The details are well abstracted from the user. Accessing a register feels exactly the same as accessing regular Python class attributes. Assume there is `top` block, which contains `foo` subblock, which containts `bar` status register. After instantiating the `top` class reading the `bar` register can be simply done with `top.foo.bar.read()`.

5.1 Register interface

Currently register interface supports following methods:

1. `read()`.
2. `read_fifo(count)` - read register *count* times.
3. `write(value)`.
4. `write_fifo(values)` - write register with *values*, where *values* is a list.

Both `read_fifo` and `write_fifo` are useful not only for interacting with real FIFOs. For example, `write_fifo([1, 0])` is a concise way for resetting modules (assuming required pulse width on a reset port can be shorter than single write operation within the FPGA).

5.2 Example

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search