README.md

Device Tree Interconnect (DeTrI)

Pronounced dee-tree

Introduction

This package provides a VHDL library and supporting Python script to help implement a relatively dynamic register bus implementation. Addresses are assigned by the python script using information captured from the synthesis report after analysis and elaboration.

This package is related to the registerDef project that defines the contents of the register sets, but does not dependent on it.

The VHDL library provides an abstraction around a register bus implementation such as AvalonMM, AXI4, AXI4-lite, or Wishbone. Currently AvalonMM and a custom packet based transport register bus implementations are provided.

The DeTrI bus tree is assembled by hand in VHDL using the provided components:

1. **Master Access** - The access port to the register bus that implements a slave port from another protocol such as AvalonMM or AXI4. This would typically interface to a CPU.
2. **Register Bank** - an endpoint collection of registers that are contiguous in the memory map.
3. **RAM Bridge** - an endpoint bridge to a RAM (similar to a register bank but includes a sub-range address).
4. **AvalonMM Bridge** - an endpoint bridge (master) to an AvalonMM Slave, i.e. an Intel/Altera IP.
5. **Interconnect** - a register bus multiplexer used to connect many slaves to a single master (in a hierarchical tree-like arrangement).
6. **Partition** - a logical separator that collects many slaves into a common/limited address range. This includes an Interconnect. It may provide address translation services within its memory domain.
7. **Clock domain crossing** - a module that translates bus transactions between clock domains.

The DeTrI-bus signals consists of opaque records

1. a signal of type **t_detri_MOSI** (master out slave in) and
2. a signal of type **t_detri_SOMI** (slave out master in) and
3. a generic **g_INTERCONNECT_PARAMS**.

These signals are record types, so they can be easily changed to realise a concrete implementation. Users should not depend on any of the signals contained within, they may change without notice.

Implementations of the DeTrI bus provided are:

1. DeTrI-Pkt - a proprietary packet based transport for transactions, that uses heavy pipelining to achieve high clock rates.
2. Avalon-MM - standard Intel/Altera bus. Simple but clock rate limited by the number of registers fanning in a the wait-request signal.

The end-points retrieve their addresses from the address table that is passed around through the generic `g_INTERCONNECT_PARAMS`. A generic is used rather than a global constant so that two or more busses may be used in the same design. The address table(s) is updated by the python script, which scans the synthesis report to find all the master-access ports, end-points, and partitions within the design. Tools supported are:

1. Intel Quartus Prime Pro
2. Xilinx Vivado
3. Simulators:
   - Known to work: Modelsim.
   - Known not to work: None.

Addresses are all automatically assigned; manual assignment is *not* supported for philosophical reasons. Partitions can be used to group addresses into a limited and exclusive address space - for the purpose of support partial reconfiguration.

The python script can generate a number of output formats that contain the address map information:

1. a VHDL package that the VHDL end-points use to locate themselves on the bus.
2. a collection of device tree syntax (.dts, .dtsi) files that are compiled into a device tree blob (.dtb) file that is used by operating systems like linux to locate devices and their addresses.
3. a JSON formatted file that lists modules and address according to the system-map file. See below.
4. human friendly outputs to help visualize and debug the bus.
5. Hash maps (dictionaries) for use in scripting languages:
    i. Python,
    ii. Tcl.
6. a simulation testbench (basically the same as an embedded VHDL package).

Whenever end-points are added or removed from the VHDL design, the python script must be run after the analysis and elaboration step to update the address table. Analysis and elaboration must be re-started so that the correct addresses are used by the end-points. The python script exits with a return value of 1 when analysis and elaboration must be re-run.

How do I get set up?

- Install python3 and pip3 according to your operating system.
    - `$ sudo apt install python3 pip3`
- Install python3 dependencies
    - `$ sudo -H python3 -m pip install -r requirements.txt`

Software Integration

Passing the `detri_address_table.py` program both a synthesis report (`.syn.rpt`) file and a system map file (`.ipmap`) will associate the IP blocks identified by RTL hierarchy with simple names. With the `-json` flag this information will be output in a JSON format that can be consumed by the software system to associate the simple names with base addresses in the address map.

To map between the software name space and the firmware name space we use a system-map file.

System-Map File Syntax

The system-map file uses a simple syntax:

```
[HDL_hierarchy] => device_name [@ register_name]
```

where:

- `HDL_hierarchy` is the path to the registers in the VHDL, and is optional (in the case the IP does not yet exist in the RTL design).
- `device_name` is the name assigned to the module, it is required and must be unique.
- `register_name` is the name assigned to the register set within the device. This is optional, and will default to the a string defined in the RTL which is usually derived from the registerDef register-set mnemonic.

Also note:

- White space around the separators `=>` and `@` is ignored. This permits nicer formatting. Aligning `=>` vertically is recommended.
- Comments start with a `#` and continue to the end of the line.

A simple example with two IP blocks:

```
E_LED_CTRL|E_REGISTERS => base_led_ctrl  # Comments can go at the end of a line too.
E_SYS_ID|E_REGISTERS   => base_sys_id
```

For every end-point found in the synthesis report that matches the defined `HDL_hierarchy` pattern a device will be created in the ip-subsystem tree or JSON file.

Wild Cards

It is possible to extract fields from the HDL hierarchy using `{}` and use them to generate device names. For instance the above example can be collapsed into a single line:

```
E_{}|E_REGISTERS => base/{} # output the same as above example.
```

However, much more is possible, for example:

```
# Using a field named 'idx' that is decoded as a decimal number. Must use the name to refer to it.
G_VCC_INTERCONNECT|G_XCVR|{idx:d}|E_SLIM|E_TX|E_REGISTERS => base/tx_slim/{idx}

#Using a fixed position field. Can refer to it implicitly by position or by number. Note the device_name and link_name
G_VCC_INTERCONNECT|G_XCVR|{}|E_SLIM|E_RX|E_{}      => base/rx_siip/{}.{1}
```

The python *parse* library and *str.format()* function is used to implement this. See the parse documentation for the complete syntax of the mini-language. Using a named field is best practice, since the name can provide some context about the parameter and hopefully reduce the quantity of comments.

Register Names (@)

Appending the name with an @ allows a particular module's register name to be respecified. This is optional, and will default to the a string defined in the RTL which is usually derived from the registerDef register-set mnemonic.

This can be helpful when there are multiple register sets associated with one module and should be used together by a software device.

Unimplemented IP-Subsystem Devices

HDL_hierarchies that do not exist in the synthesis report will not match definitions in the system-map file, therefore the corresponding device will not be included in the device tree. However, the `HDL_hierarchy` is optional. This is used for devices that don't yet have a matching HDL instance, and for which an entry in the ip-subsystem device tree is desired.

The syntax for this case is:

```
=> base.unknown.tango.device
```

Engineering Debug Device for Unknown End-Points

When an end-point is found in the synthesis report that does not have a corresponding entry in the system-map file it is added to the `engineering_debug` device. Each unknown endpoint is appended to the link list with the name `eng_debug_#`, where `#` is an incrementing integer starting at zero.

A warning is printed for each missing register set in the system-map syntax so that it can be copied into the system-map file.

Using DeTrI in simulation environments

The provided simulation example uses the Vunit Verification Component Library as a base for the master interface. The `./vhdl/sim/detri_master.vhd` file provides a wrapper around a Vunit bus master. To use it first install Vunit. From PyPi:

```
$ sudo -H python3 -m pip install vunit_hdl
```

Then you can run the example testbench in `./tb/`

```
$ cd tb/
$ python3 vunit_run.py detri.interconnect_example_root_tb
```

After the simulation has elaborated it will have written a file `./vunit_out/<simulator>/<tb_entity>-regdef.txt` that has all of the information extracted from the detri_endpoints. This should occur even if the simulation does not run successfully.

The top level simulation file is `tb/interconnect_example_root_tb.vhd`. It contains both the Address Table and System-Map. For small simulations this is simpler than having to manage separate files and keeps all the information in one easy to reference place.

The address table can be populated/updated using the `interconnect_example_root_tb-regdef.txt` file created by the simulation elaboration and the System-Map comments in the testbench file.

To see this in action, first open the testbench file and delete the address table between (but not including) the comments `--[[[cog cog.out(address_table)]]]` and `--[[[end]]]`. Then run the simulation again.

```
$ python3 vunit_run.py detri.interconnect_example_root_tb
```

The simulation will have failed, but the `interconnect_example_root_tb-regdef.txt` file has been updated. To feedback this information of what registers exist into the simulation we use the python script to update the address table with the `--sim` flag.

```
$ python3 ../detri_address_table.py  vunit_out/modelsim/interconnect_example_root_tb-regdef.txt --sim interconnect_exa
```

Here we call the python script with,

- The `interconnect_example_root_tb-regdef.txt` file as our synthesis report (filename must end with `regdef.txt`).
- The option `--sim` with argument for the testbench file `interconnect_example_root_tb.vhd`. This file is used:
  i. as the system-map (which is a VHDL comment bounded by `-- IPMAP begin` and `-- end IPMAP`), and
  ii. the VHDL output (which must contain the cog comment `--[[[cog cog.out(address_table)]]] [[[end]]]` within the address table construct).

This will populate the address table in the testbench file with ip_names defined by the system map. Running the simulation again will recompile the testbench and elaborate with the assigned addresses for each register.

```
$ python3 run.py
```

In the `P_MAIN` process of the testbench are the test cases. These demonstrate writing and checking registers. Other procedures for reading, burst transactions, and with blocking and non-blocking behaviour are also provided by the Vunit Verification library interface. The base addresses used to access the registers are looked-up in the address table using the ip_name as defined in the system-map. This allows the addresses to change without breaking the testbench.

The example testbench can and should be used as the basis of testbenches by replacing the DUT and adjusting the system-map rules.

Concrete Register Bus Implementations

The internal register bus implementation is selectable by including one of the implementation directories that contain the VHDL architectures for the base entities defined in the `rtl/` directory.

The current implementation directories are:

1. `rtl/detri_pkt`,

2. `rtl/avalon_mm` .

Using the `Manifest.py` system, these can be selected by setting the detri_arch variable to the directory name; one of `detri_pkt` or `avalon_mm` . If the `detri_arch` variable is not set, then a warning is printed and the default selected.

The rest of the information in this section is on the internal architecture of the DeTrI register bus, you shouldn't need to know about this to use DeTrI.

Custom Packet-based register bus Implementation

This is the default implementation, selected with the `detri_arch = "detri_pkt"` Manifest.py assignment. Currently it has only an AXI4 master access interface.

This implementation is designed primarily to allow the packet bus to pipeline throughout a large design and improve the static timing. The tradeoff is that additional pipelining increases the latency, and that latency ultimately affects the read bandwidth. Only AXI4 features that are used by the Stratix10 ARM CPU (HPS) have been included. DMA features such as fixed and wrap burst-modes are not provided.

The write bus in the master to endpoint direction is defined as

```
type t_DETRI_WRITE_BUS  is record
    path        : t_detri_path;
    hdr_dat     : std_logic_vector(127 downto 0);
    hdr_vld     : std_logic;
    dat_vld     : std_logic;
    dat_last    : std_logic;
end record t_DETRI_WRITE_BUS;
```

The bus `hdr_dat` has two phases:

1. A header phase with addressing information marked with the `hdr_vld` bit, lasting 1 cycle and,
2. A write data phase marked with the `dat_vld` bit, lasting as many cycles as required being terminated by the `dat_last` bit.

The header phase contains either a *write* transaction or a *read* transaction.

The read bus in the endpoint to master direction is defined as

```
    type t_DETRI_READ_BUS is record
        dat         : std_logic_vector(127 downto 0);
        dat_vld     : std_logic;
        dat_last    : std_logic;
        virt_chan   : unsigned(3 downto 0);
    end record t_DETRI_READ_BUS;
```

The data is transported over the `dat` signal as a packet delimited by `dat_vld` and `dat_last` . The signal `virt_chan` indicates the phase of the read bus time-division-multiplexing (TDM) into virtual channels. *Currently only a single virtual channel is implemented, so this is unused.*

Write Transactions

A write transaction has a header defined as

```
+-----------------------------+-----+--------------+---------------+
|       32 bit Byte Address   | '0' | 4b low_empty | 4b high_empty |
+-----------------------------+-----+--------------+---------------+
```

where,

1. The `address` is a 32b byte address,

2. The `is_rd` flag is '0' indicating a write transaction,
3. The `low_empty` field specifies how many of the lowest bytes of the first word of the bus are empty (i.e. byte-enable = '0'),
4. The `high_empty` field specifies how many of the highest bytes of the last word of the bus are empty (i.e. byte-enable = '0').

The `low_empty` and `high_empty` fields allow a transaction that is not aligned to the bus width be specified. These are typically translated to byte enables at the end-point. If the length of the transaction is one word, that is the first word is also the last word, then `low_empty` and `high_empty` apply to the same word and produce a byte-enable that permits smaller writes than the bus width. For example, a one byte write to address 0xFF01 can be specified as:

```
+-----------------------------+-----+-------------+---------------+
|         0xFF00              | '0' |      0x1    |      0x2      |
+-----------------------------+-----+-------------+---------------+
```

In the data phase the bus transports the data words to be written to the registers. This permits burst writes of "unlimited" length to be written. Practically write bursts must be limited to bursts of a maximum length as specified by `g_INTERCONNECT_PARAMS.burst_length`. This specifying the bit width of counters in the endpoint. Typically bursts are also limited to the address range of the endpoint, and by the master driving the bus. *For the Stratix10 HPS, this appears to be limited to 16 bytes (128b) for transactions originating at the CPUs. This may be different for DMA driven transactions.* The data words are assumed to be aligned to the bus width. Coincident with the last word on the bus the `dat_last` signal is asserted.

Write transactions are posted. There is no indication returned to the master indicating success or otherwise. *The AXI master b-channel is responded to as soon as the write is accepted, and posted to the DeTrI bus.*

Write transactions are executed in order. The AXI master should maintain the order of read and write transactions from the same master. This naturally flows from software, where the software process is stalled waiting for the read to complete before it issues further reads or writes.

Read Transactions

A read transaction has a header defined as

```
+-----------------------------+-----+--------------------+---------------+
|       32 bit Byte Address   | '1' | 4b Virtual Channel |   4b Length   |
+-----------------------------+-----+--------------------+---------------+
```

where,

1. The `address` is a 32b byte address,
2. The `is_rd` flag is '1' indicating a read transaction,
3. The `virt_chan` field specifies which virtual channel TDM slot to reply on.
4. The `length` field specifies how many consecutive words to read in the burst.

The read request transaction is dispatched on the write bus, and only consists of a header. This maintains strict ordering between write and read transactions to the same endpoint. *Transactions to different endpoints are not guaranteed to complete in order due to the variable latency through the interconnect.*

The endpoint replies on the read bus in the endpoint to master direction. The read bus is time-multiplexed, with each virtual channel being designated a time-slot. *Note: Currently only a single virtual channel is implemented.* With virtual channels multiple masters can issue independent read transactions to different endpoints, increasing system performance even though the read latency is fixed. Using time-slots simplifies merging of the bus, eliminating the need for FIFOs and reverse flow control.

Burst lengths are limited by the smaller of g_DETRI_INTERCONNECT_PARAMS.burst_length, and 16 (the maximum encodable in the width of the 4b `length` field). *For the Stratix10 HPS, this appears to be limited to 16 bytes (128b) for transactions originating at the CPUs. This may be different for DMA driven transactions.*

Bus Width

The actual bus width is one of 32, 64, or 128 bits as specified in the accompanying generic g_DETRI_INTERCONNECT_PARAMS. Increasing the bus width will linearly increase the write performance. Read performance is dominated by latency so will not increase much.

For the minimum 32-bit width, the high bits of the header are truncated limiting the usable address width to 22 bits. In the write header the `low_empty` and `high_empty` values are interpreted modulo the selected bus width in bytes.

Path Address and Network Address Translation (NAT)

The `path` field of the write bus specifies the path through the interconnect tree to reach the addressed endpoint. It is used to set the `hdr_vld` and `dat_vld` signals on the appropriate output path of the interconnect.

A NAT logic reads the address from the header and converts it to a path through the interconnect tree. The `path` is used to select the demultiplexers as it propagates though the interconnect. Each interconnect module takes the log2(N) lowest bits from `path`, and uses that to index one of the N slave ports. The path is left shifted by log2(N) and passed to the next interconnect modules in the tree.

The network address translation (NAT) logic (effectively a static content addressable memory (CAM)) is constructed from the tree.

1. Each endpoint populates a ROM by setting a valid flag in the index according to its position the the address table.
2. The ROM is passed to the interconnect module which pushes the slave port's index into the lower bits of the valid path(s).
3. The ROMs are then merged and passed up the tree to the next interconnect module.
4. When the ROM reaches a PR-partition, the static CAM logic is generated that converts the base address from the address table (ignoring the lower log2(X) "don't care" address bits according to the module's depth X) to the path value.

Critically, all of this NAT construction logic is combinatorial so will reduce to constants and optimise away during synthesis. When the address reaches the endpoint it only cares about the lower log2(X) bits to address its own registers.

If a read-request to an unknown address is received, then a default slave will reply on behalf. *Currently the default slave returns `0xDEADC0DE` as the read's value(s).* Write requests to unknown addresses are silently dropped.

Pipelining

The paths to endpoints are individually pipelined using the Stratix10 variable latency hyper-register pipelining feature. This inserts as many pipelines as are necessary to meet the static timing requirements, and transport the bus down and across the chip. This relaxes the path constraints from the register bus and allows the fitter greater freedom to place IP blocks according to data-path requirements rather than competing with control-path.

Clock Domain Crossing

Clock domain crossing is implemented using FIFOs in each direction.

Note, some signals that are intended to reduce to constants during synthesis are connected combinatorially, skipping the FIFOs. In particular the NAT table construction signals.

Contribution guidelines

Please do not commit to the master branch. Commit to a feature branch first so that the code can be reviewed before being merged.

- Writing tests
- Code review
- Other guidelines

Who do I talk to?

William Kamp, Ph.D. william.kamp@aut.ac.nz

High Performance Computing Research Lab, Auckland University of Technology, New Zealand.