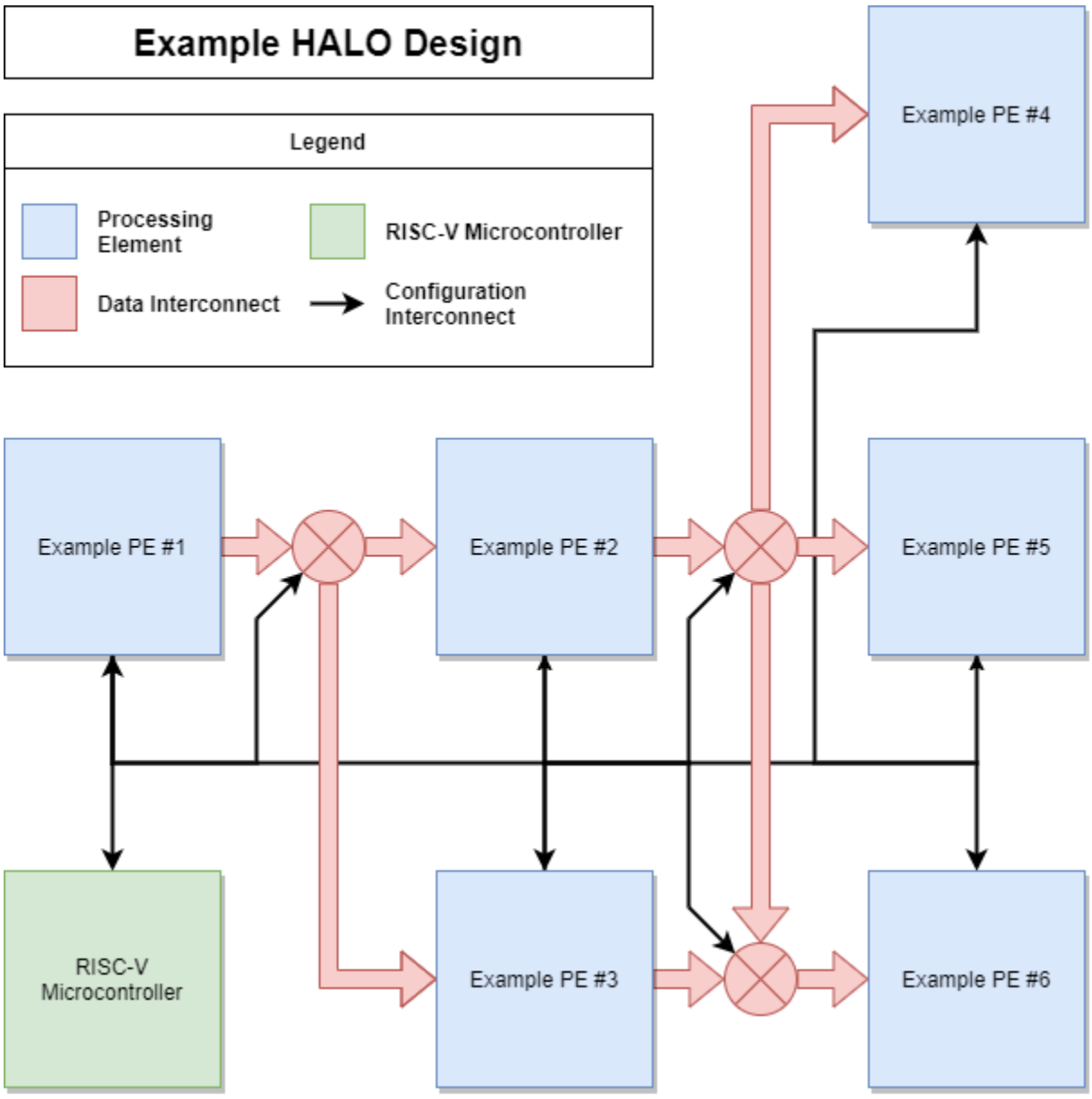
First HALO design

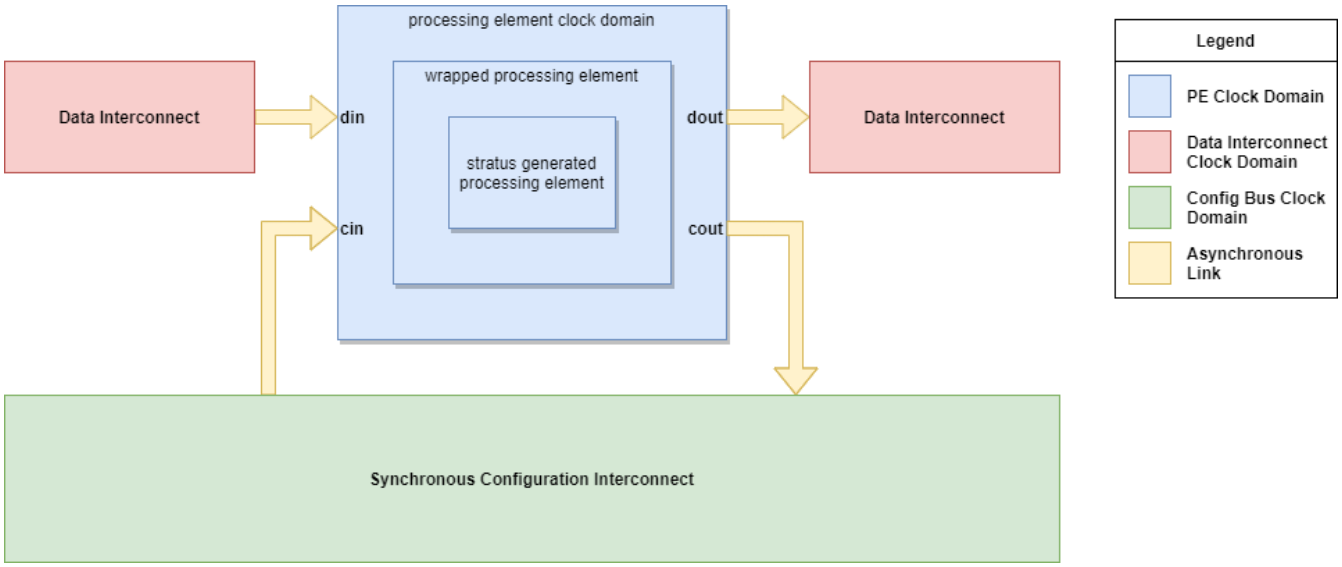
This is the first HALO design that contains a microcontroller + a collection of accelerators. A list of TODOs for the tape-out is at the end.

Structure Overview



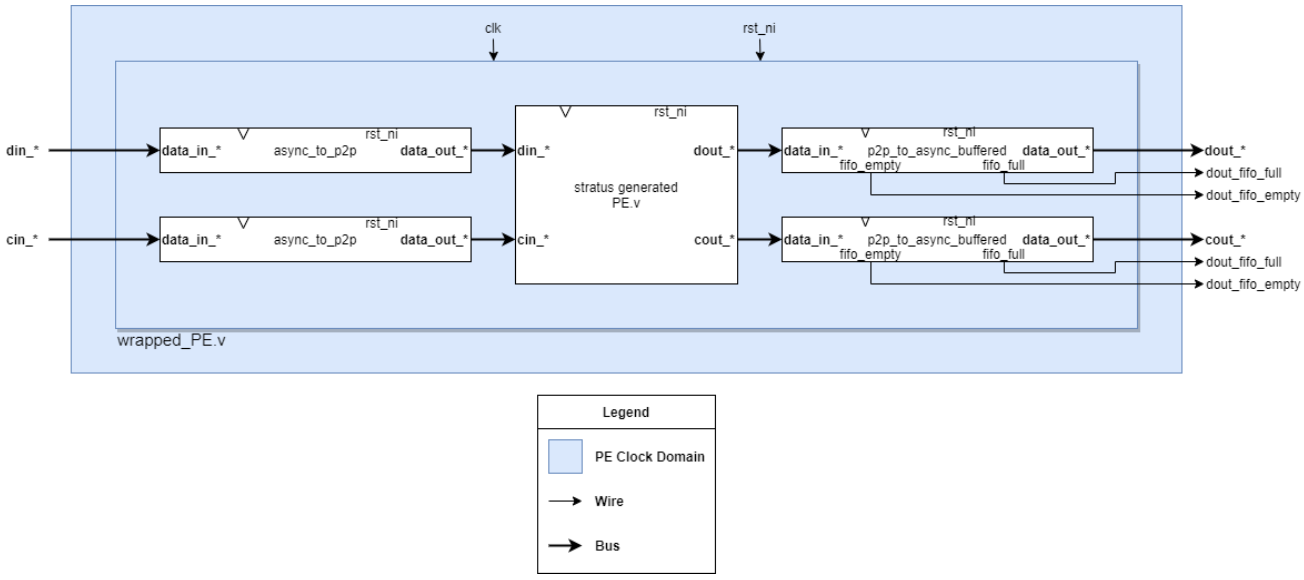
This diagram shows the overall SoC structure of HALO. This is an example implementation and not

representative of the actual final design. Every blue box is a PE and may be hand-crafted or generated by stratus (with a suitable wrapper). The red arrows and switches indicate the main dataflow interconnect. The solid black arrows/lines indicate how the configuration bus interfaces with the rest of the chip.



This figure shows the overall structure of each PE. Each PE sits within it's own clock domain. There are asynchronous connections between each PE and the data and configuration interconnects. In the first tapeout, both the data and the configuration interconnects are synchronous in their own clock domains.

The cores of most PEs are generated by stratus (high level synthesis tool). The PEs are wrapped inside a wrapper which provides asynchronous interfaces.



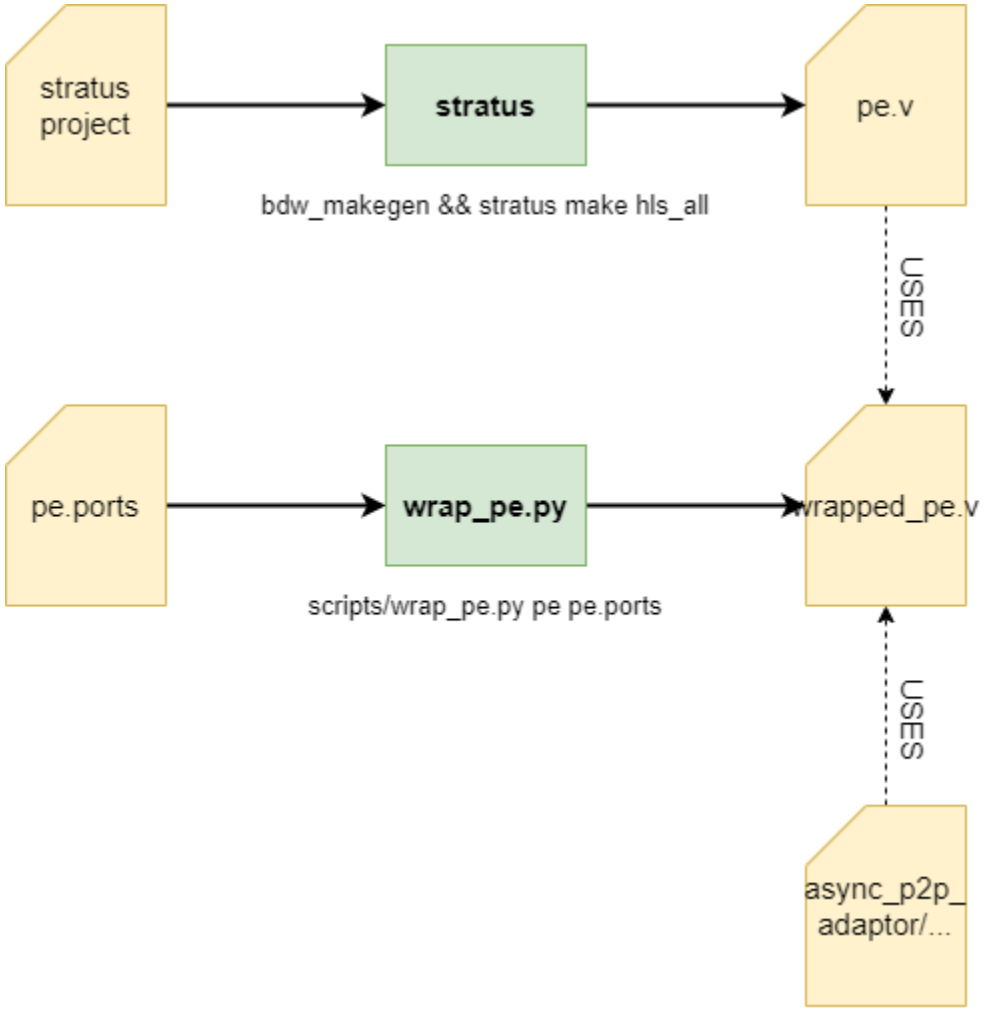
This figure shows the RTL structure of an individual PE in more detail. The core module is the PE that is generated by stratus. A number of adaptors (shims) are then added around the PE. There is one adaptor per port. The shims perform two functions:

- Act as protocol converters between the asynchronous and synchronous domains

- If the shim ends with *_buffered* then the adaptor also includes a FIFO

These adaptors are inserted automatically by the *wrap_pe.py* script in the scripts directory. There is no need to add these manually.

If an adaptor is buffered, then two special signals **_fifo_empty* and **_fifo_full* as also included. These signals indicate when the associated FIFO is empty and full respectively.



This figure explains the build process in more detail (replace pe with the name of the associated PE)

- *stratus* generates *pe.v* from the high level source code files
- *pe.ports* describes the modules input/output port configuration
- *wrap_pe.py* accepts the description of the modules ports and generates *wrapped_pe.v*
- *wrapped_pe.v* wraps the original PE, using adaptors contained with the "*async_p2p_adaptor*" directory

The asynchronous bus protocol, data bus width, and the depth of any FIFO for buffered adaptors are all specified in the *pe.ports* file.

Workflow

HALO is compiled into RTL from C/C++ code using stratus HLS, a high-level synthesis tool by Cadence. The source code can be found in a GitHub repo (https://github.com/ysarch-lab/neuronal-data-compression). If you don't have access please contact Karthik Sriram or Nicholas Lindsay (nick.lindsay@yale.edu) to be added.

To get started, first obtain a node on grace (Yale's HPC cluster). The following command will get you an interactive bash shell on a node in Rajit's group for 20.5 days (other PI groups for which you have been added should work also):

```
srun --pty -a manohar -p pi_manohar --time=20-12:00:00 bash
```

Navigate to a directory of your choosing. Then clone the *halo* repo using the following command:

```
git clone git@github.com:ysarch-lab/neuronal-data-compression.git
```

Change directory to the newly cloned repository, and then checkout the stratus branch using:

```
git checkout stratus
```

You can explore the various modules that have been implemented in HLS by exploring the various directories. For example, the identity directory contains code for a very simple PE which simply replicates the input at the output with a one-to-one mapping.

stratus Build Methodology

- **project** ï [*bdw_makegen*]ï **project with makefile** ï [*make hls_all*]ï **generated rtl**

There are a number of steps that must be followed to generate Verilog RTL from a stratus project. The first stage is to generate a makefile than will then be used to generate the actual Verilog. stratus HLS will generate the Makefile automatically based on the project specification file, project.tcl. The projects should already have the `project.tcl` file completed.

First, make sure that the Cadence tools are available in your environment using:

```
setup cadence
```

You can add this line simply to your .bashrc file or your shell equivalent.

To now generate the Makefile from the project settings, run:

```
bdw_makegen
```

You can now use various *make* commands to perform different actions. For example:

- *make hls_all* - Generate all HLS files
- *make sim_all* - Build and run all simulations
- *make ide* - Start the Stratus IDE
- *make help* - List all targets

After running a command (like *make hls_all*), the generated files will be available in certain subdirectories. In particular, the generated RTL for the module is found at:

```
bdw_work/modules/<MODULE_NAME>/<MODULE_CONFIG>/<MODULE>_rtl.v
```

This file can be used for synthesis/analysis/etc.

stratus Documentation

We have documentation for the Stratus HLS tools. Do not share the documentation publicly.

- Stratus HLS Getting Started Guide: stratus_hls_getting_started_guide.pdf
- Stratus HLS User Guide: stratus_hls_user_guide.pdf

Processing Element interface

Processing elements within their synchronous domain have the following ports:

- Per-domain clock and reset ports
- One or more data input ports
- One or more data output ports
- An input/output configuration port

Clock and Reset

clk is the clock signal for a particular PE. All PE interfaces are synchronized to the rising clock edge (posedge).

rst is an active-low synchronous reset signal. The PE is reset when rst is low at the posedge of the clock.

Data port interface

The data ports are used to connect the PE to the system's data interconnect.

Each data port uses the *cynw_p2p* protocol as generated by *stratus*.

The *cynw_p2p* interface has the following specification:

- Three connections:
  - **data**: Data connection from producer to consumer.
  - **valid**: Valid signal from producer to consumer indicating data is ready.
  - **busy**: Busy signal from consumer to producer indicating consumer is not ready to accept data.
- Data is only accepted on the clock edge iff **valid == 1** and **busy == 0**
- All operations within HALO are synchronized to the rising edge (posedge in Verilog)

Configuration port interface

The configuration port interface allows processing elements to be configured by the RISC-V microcontroller. The configuration interface is designed to support the following operations:

- Starting and stopping of each PE
- Configuring each PE
- Status monitoring
- Error recovery

The configuration bus is common to all PEs. The bus is connected to each PE through configuration ports. The bus uses a master-slave interface, where the RISC-V microcontroller is the master and PEs are the slaves. The bus defines the following signals:

- **cfg_addr**: M→S. Address of configuration register.
- **cfg_data_in**: M→S. Configuration data from master to PE.
- **cfg_data_out**: S→M. Status data from PE to master.
- **cfg_en**: M→S. Enable the bus for a read or write operation.
- **cfg_rw**: M→S. Indicates a write operation (cfg_rw=1) or read operation (cfg_rd=0).
- **cfg_accept**: S→M. PE indicates that the operation has finished. If the operation was a write, then data is available at the same posedge as cfg_accept is high.

The configuration interface is 16 bits. All addresses are at the word granularity. No port is byte addressable (the minimum transfer width is a 2 byte word).

Memory map for stratus components

Every PE generated by *stratus* has a private configuration address space with a structure common to all *stratus* generated PEs. Every PE reserves a configuration address space of 16 words (32 bytes) and is

16 word aligned. The lowest address is referred to as the BASE address. The layout of the configuration address space is given below:

```
+---------+-------------+----------------------------------+
| Addr    | Name        | Description                      |
+---------+-------------+----------------------------------+
| BASE+0  | PASSTHROUGH | PE specific commands and status. |
+---------+-------------+----------------------------------+
| BASE+1  | HARDCONFIG  | Generic PE control and status.   |
+---------+-------------+----------------------------------+
| BASE+2  | Reserved    | Reserved.                        |
|   ...   |             |                                  |
| BASE+15 |             |                                  |
+---------+-------------+----------------------------------+
```

Each *stratus* generated PE implements a minimum of two registers: PASSTHROUGH and HARDCONFIG. PASSTHROUGH is used to send *PE-specific* commands and to receive *PE-specific* status information. HARDCONFIG is used primarily to configure and reset the PE. HARDCONFIG shares the same interface across all PEs.

PASSTHROUGH register

For *stratus* PE's, the PASSTHROUGH register provides a request/response interface to the user. Requests are issued to the PE by writing to the PASSTHROUGH register. Requests are typically *commands* which affect the PE in some way. These commands are PE specific, and may include:

- Start and stop commands
- Soft reset commands
- Status request commands

Some commands may produce outputs. For example, a PE may contain a command which returns the number of bytes that have passed through the PE. These outputs can be read by reading the PASSTHROUGH register.

Commands that produce output are *asynchronous*. This is to prevent these commands from stalling the data flow through the PE. This is implemented by adding a FIFO to the output. Additionally, to prevent deadlock, attempts to read from the FIFO *will never stall*. This means that they will return a value *even if the output is not ready*. In such situations the value is undefined. To prevent this, you must poll the HARDCONFIG register until the FIFO Non-Empty bit is set. Once the FIFO Non-Empty bit is high, a read is guaranteed to return the value from the FIFO.

By default, this FIFO is 16 bytes. Once the FIFO is full, it *WILL* stall the PE. This situation must be avoided. In particular, be sure not to issue too many status requests such that the FIFO will be overfilled. If you are implementing a particular PE, make sure no command generates more than 16 bytes or be sure to increase the size of the FIFO (you'll need to modify wrap_pe.py to do this).

HARDCONFIG register

Reads from the HARDCONFIG register provide status information about the PE's configuration ports. The register has the following bit layout:

```
| Bit | 15 | 14 | 13 | 12 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
+-----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|  ID |                           Reserved                                | FF | NE |
+-----+-------------------------------------------------------------------+----+----+
HARDCONFIG Register Read Format
```

Bit description:

- NE: The PASSTHROUGH FIFO is Non-Empty - in other words, there is data that is ready to be read from the PASSTHROUGH register.
- FF: The PASSTHROUGH FIFO is Full and as a result is blocking the PE.

Writes to the HARDCONFIG register are usually used for error recovery. The supported commands are:

```
+----------+------------+-------------------------------------+
| Encoding | Command    | Description                         |
+----------+------------+-------------------------------------+
| 0xfffe   | FIFO_CLEAR | Clears the PASSTHROUGH output FIFO. |
+----------+------------+-------------------------------------+
| 0xffff   | HARD_RESET | Resets the PE and all interfaces.   |
+----------+------------+-------------------------------------+
```

FIFO_CLEAR force resets the PASSTHROUGH output FIFO. Any data in this FIFO will be lost. This is useful if the FIFO unexpectedly contains data (although this is *always* a result of an error, either in software or hardware, that must be corrected).

HARD_RESET performs a complete reset of the PE. This *will* result in the loss of data. This should be used as a last resort of the PE is definitely broken (e.g. through deadlock).

Programming Example

In this section are examples of instruction sequences that can be used to configure and operate the PEs.

Pseudocode example for configuring PE

```
# First, completely reset the PE
PE.HardConfig <- 0xffff
# Send configuration commands to PE
```

```
PE.Passthrough <- a
PE.Passthrough <- b
PE.Passthrough <- c
# Start the PE
PE.Passthrough <- start
```

Pseudocode for gathering status information from a PE

```
# Send the command to read status information
PE.Passthrough <- READ_STATUS_COMMAND
# Read all the data into a buffer
uint16_t buffer[8];
for (i = 0; i < 8; i++) {
  # poll the Non-Empty bit
  while (!PE.HardConfig & 0x1) {};
  buffer[i] <- PE.Passthrough
}
```

Configuration port implementation

The configuration port interface is implemented by a custom verilog wrapper (*cfg_wrapper.v*) which sits between the PE and the system-wide configuration interface. The system wide interface is the configuration port interface as described above. To accommodate configuration requests and responses, under the hood every PE (written in C/C++) contains two additional *cynw_p2p* interfaces:

- **cin**: A busy/valid input interface for accepting configuration requests.
- **cout**: A busy/valid output interface for outputting status information.

Modules without configuration support must be modified to support this interface. This consists of the following:

1. Include the SystemC type declarations for the config interface by adding the following include to the "*_mod.h" file:
   - ```
     include "../common/types.h"
     ```
2. Add the two new configuration interfaces (**cin** and **cout**) by adding the following lines within the "SC_MODULE" declaration under the "*_mod.h" file, immediately after the already defined ports:
   - ```
     cynw_p2p < config_t >::in cin;
     cynw_p2p < config_t >::out cout;
     ```
3. Modify the main thread in the design to now poll the config port. The main thread at it's highest

level consists of a while loop which continuously polls the data input port(s). We need to now modify this function so that it also checks the configuration port for config commands. To accommodate this, we need to wrap the main thread function *t()* (which is defined in "*_mod.cc" files) with the following:

```
// Main execution loop
while (1)
{
    // Wait for an input to become available on either input
    sc_uint<2> avail = cynw_wait_any_can_get(din, cin);

    // Check initially for messages on the config port
    if (avail & 0x2) {
      // decode message type
      config_t msg;
      cin.nb_get(msg);

      switch (msg) {
        // insert configuration actions here
      }
    }

    // Check for inputs on the data port
    if (avail & 0x1) {
      // Get a value, process it, and put the result
      input_t in_val;
      din.nb_get(in_val);
      // The following line performs the computation on the input:
      output_t out_val = f(in_val);
      dout.put(out_val);
    }
}
```

4. Assuming everything goes well, the configuration port has now successfully been added to the module. You'll have to modify other files that instantiate the module to add the configuration port. The easiest way to find out which other files need to be modified is to simply run *make hls_all* and see which error messages arise. Files that will definitely need to be modified include *system.cc*, *system.h*, *tb.cc* and *tb.h*.
5. Now that the configuration interface has been added, it will be necessary to test that the interface works as expected. The first place to test this is at the module level itself using the provided *stratus* test facilities. The easiest way to do this is to send configuration commands in the testbench file (*tb.cc* and *tb.h*) and then verify the expected and actual responses.
6. Once the interface has been tested from within the HLS framework, the generated Verilog module can now be "wrapped" so that it can communicate with the system-wide configuration bus. Wrapped module files can be generated using the *wrap_pe.py* script within the *scripts* directory. This will produce a new module called *wrapped_<MODNAME>* which instantiates

both the module *<MODNAME>* and a shim that connects the modules configuration ports to the system-wide configuration bus. The new module retains the Busy/Valid-like data input/output connections but replaces the configuration FIFO interface with the addressable configuration bus.

7. You'll now need to write a testbench that uses the configuration bus for IO. Currently this process must be done manually using Verilog. Take a look inside the *rtl* directory for some examples of testbenches. To run the testbench with the generated files, you'll need to include the following files in the run command for your simulator:
   - *<MODNAME>.v* - the original module generated by *stratus*
   - *wrapped_<MODNAME>.v* - the module wrapped with the configuration interface (this file is generated by *wrap_pe.py*)
   - *cfg_wrapper.v* - the shim, internal to *wrapped_<MODNAME>.v*, that connects the config interface to the config FIFOs under the hood
8. Once the module has been tested, it will be ready for system integration.

Software Simulator for HALO

To produce the gold standard output to verify RTL and HLS, a software simulator for HALO has been created in the stratus branch. It's a simple software that can quickly generate the expected output for a particular HALO pipeline configuration. Note that simul doesn't take timing into account. This means that it will create one of the possible outputs for a configuration and input.

To build the binary, simply run (*make*) inside the simul folder. This will build the *simul* binary.

Simul usage and options

simul needs two options to run: a pipeline configuration file, and an input file. To specify these, use the following options-

- –ifile, -i <filname> to specify input file
- –cfile, -c <filename> to specify pipeline configuration
- –ofile, -o <filename> to write the output to a file rather than stdout
- –hex-format, -x to write output in hex
- –verbose,-v to increase verbosity of output (useful for debugging)
- –help, -h to print this message

The configuration file declares the modules to be used, and the connections between them, specified in a simple format. Examples can be found under simul/config/ . Here's one example -

```
Neo
ADC
Sink
ADC->Neo
```

```
Neo->Sink
```

Here, a simple pipeline is made connecting the ADC to the Neo Processing Element which is then connected to a Sink. Note that modules are declared at the top with newlines while connections are declared by using declared module names with a → (without spaces) in between. You can connect any module with any other module, but be careful to not create a cyclic graph which causes an infinite loop. You can find a list of implemented modules under (*src/module*).

The ADC and the Sink module names are special. Only one ADC module should exist. All input read from the input file are sent to the ADC module, which then passes it to all connected modules.

Sink modules on the other hand can be instantiated multiple times, although they still need a unique name. Any module name starting with Sink will be instantiated as a unique Sink module (eg *Sink-ADC*, *Sink-LZ* etc). Any data sent to the sink will be printed to the output.

Steps to create a pipeline configuration

1. First, create a config file.
   a. List an ADC module and all modules you intend to use at the top. For any output you wish to monitor, create as many Sink modules with unique names.
      - *Example*

```
ADC
LZ
LIC
Sink-LZ
Sink-LIC
```

      - Here, two Sink module have been created with unique names to identify them.
   b. Next, in the same file, specify the connections between your modules. Connections can be many-to-many. However, it is undefined to use the ADC as an output and to use the Sink as an input.
      - *Example*

```
ADC->LZ
LZ->LIC
LZ->Sink-LZ
LIC->Sink-LIC
```

      - Here, the output of both LZ and LIC is being monitored
2. Now, create an input file. This is simply a file with inputs on new lines.
   - *Example*

```
240
103
17
```

- config parameters can be sent as well by using the following format:
- *c [ModName] <parameter>*
- *Example*

```
c LZ 0x9
```

3. Now run the simul binary with these files as the input (and any other option you want to turn on)

This will create a combined output for all specified Sinks. To create a file for a specific sink, one can specify a single sink. It is also sufficient to simply use *grep* to filter out unneeded output.

Todo

There are many tasks that need to be done before the chip tapeout is ready. These include:

- ~~Making the configuration port interface non-blocking. Currently the configuration ports internal to each module block until the output is accepted. To overcome this, we should place a FIFO at each output. This can be done in two ways - adding a FIFO to *cfg_wrapper*, or by adding a FIFO within each *stratus* PE. I favour the HLS approach since it requires less Verilog to be written manually. However, I could not get the provided *cynw_fifo_direct_out<T,N,L>* template class to compile correctly within *stratus*.~~
- ~~Add a status port to the config interface. Currently, the config interface blocks until data is produced at the config port. Therefor, a config read without previously issuing a command could result in a deadlock, since the read operation never returns a result. To avoid this, we should add another read-only register to each PE which indicates whether or not the output FIFO contains data. Reading from a FIFO with data should return the data, and reading from a FIFO without data should result in undefined behaviour. This will prevent deadlocks in the design which could potentially be fatal.~~
- ~~Make the config interface non-blocking. If the FIFO output is full, we should probably drop the packet and set an error bit in the status register to prevent the data pipeline from stalling.~~ Code to instantiate a FIFO has been added to cfg_wrapper, which adds a FIFO to the configuration output of all PEs. This will still block the data flow through the PE, but only if the FIFO is full. By default, the FIFO stores 16 words (32 bytes).
- ~~Document the config interface in more detail.~~ Documented on this page.
- Come up with a list of standardized config commands for each PE. For example, we should probably have common command codes for:
  - Starting the PE
  - Stopping the PE

    - Soft resetting the PE (handled within the PE)
    - ~~Hard resetting the PE (handled externally to the PE by asserting the PE-wide reset signal)~~ - Implemented in HARDCONFIG.
    - ~~Clearing the config output FIFO~~ - Implemented in HARDCONFIG.
- Rewrite modules to use the common data types specified in *common/types.h*. This more closely aligns with the data types specified in the HALO PE Interface Document.
