

PSL-AFU Interface Example: Vector Add

This project will go through the steps required to build an Accelerator Functional Unit (AFU) for the Coherent Attached Processor Interface (CAPI). A CAPI project consists of two main components:

- Host code to run on a Power processor
- RTL code to run on a FPGA accelerator

The two components communicate with one another using the Coherent Accelerator Processor Proxy (CAPP) available on the POWER processor, which extends coherency to the attached accelerator. The RTL design on the accelerator communicates with CAPP through the Power Service Layer (PSL). A host application on a POWER system attaches to an accelerator using the libcxl user library.

This project uses C for the host design and VHDL for the RTL on the FPGA. VHDL is chosen over Verilog due to the strong type casting provided by the language. For the most part, VHDL and Verilog are capable of producing the same design and the language used is primarily based on personal preferences. Some consider VHDL to have a larger learning curve due to the strong type casting. In contrast, Verilog has weak type casting and syntax similar to C. For beginners, this usually translates to multiple rounds of code revisions to pass a VHDL compiler error free. This may seem like an unnecessary burden, but some of the syntax errors caught by the compiler will also remove bugs from your design. For instance, the following is legal in Verilog:

```
module test;

reg [3:0] op_codes;

initial begin
    #5 op_codes = 'h3F;
    case (op_codes)
        'h3F: $display("success\n");
        default: $display("fail\n");
    endcase
    $finish;
```

```
end
```

```
endmodule
```

You may be surprised to see that the upper two bits are not being assigned to the `op_codes` register, likely changing your circuits desired behavior. This type of mistakes is not possible in VHDL and will result in a compilation error.

Getting Started

This guide uses a similar setup as Kenneth Wilke's [Tinkering with CAPI](#) blog.

We will start our CAPI design using the Power Service Layer Simulation Engine (PSLSE) along with either Xsim (Xilinx designs) or ModelSim (Altera designs) on an x86 workstation.

Requirements:

- Ubuntu 15.04 or newer
- Xilinx designs: Vivado 2015.3 or newer
- Altera designs: ModelSim Altera Starter Edition 10.4b or newer

Related Documents:

- Coherent Accelerator Processor Interface User's Manual (nallatech.com)
- Coherent Accelerator Processor Interface User's Manual, Xilinx Edition (alpha-data.com)
- PSL/AFU Interface Specification (OpenPOWERFoundation.org)
- Coherent Accelerator Interface Architecture (CAIA) Specification (OpenPOWERFoundation.org)

PSLSE Setup

Clone the pslse from github

```
git clone https://github.com/ibm-capi/pslse
```

Build the AFU Driver

The AFU driver communicates between the simulator (Xsim or ModelSim) and the PSLSE. This requires the `svdpi_user.h` header included with the simulator. For Xsim, the header file is located `<xilinx_rootdir>/data/xsim/include`. For ModelSim, the header file is located `<modelsim-install-point>/include`.

Use the environment variable `VPI_USER_H_DIR` to point the compiler to the correct header file. Additionally for Altera designs, the `afu_driver` needs to be built for a 32-bit platform to work with ModelSim.

Xilinx designs:

```
cd psl/afu_driver/src
export VPI_USER_H_DIR=$XILINX_ROOTDIR/data/xsim/include
make
```

Altera designs:

```
cd pslse/afu_driver/src
export VPI_USER_H_DIR=$MODELSIM_ROOT/include
BIT32=y make
```

You may need to install `libc6-dev-i386` if `afu_driver` does not build successfully.

Build PSLSE

PSLSE also needs to be 32-bit for ModelSim.

Xilinx designs:

```
cd ../../pslse
make
```

Altera designs:

```
cd ../../pslse
BIT32=y make
```

Build PSLSE version of libcx1

```
cd ../libcx1
make
```

NOTE: afu_driver and PSLSE will need to be rebuilt if switching from Xilinx to Altera and vice versa

Vector Addition AFU

The goal of this accelerator functional unit (AFU) is to demonstrate how to add two host vectors together on a FPGA accelerator using CAPI.

AFU Host code

We will describe how to build the host code with this Makefile:

```
CC=gcc

LIBCX1_DIR=/home/khill/pslse_src/pslse/libcx1

CFLAGS= -g -I${LIBCX1_DIR} -I.
LFLAGS= -L${LIBCX1_DIR} -lm -lcx1 -lpthread -lrt

SRCS=capi-vadd.c
OBSJS=$(SRCS:.c=.o)
TARGET=capi-vadd

all: vadd

.PHONY: vadd
vadd: $(TARGET)

$(TARGET): $(OBSJS)
    $(CC) $(OBSJS) $(LFLAGS) -o $@

%.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@

.PHONY: clean
clean:
    @rm -rf $(TARGET) $(OBSJS)
```

Notice that we are linking to the cx1 library (-lcx1) to be able to attach to CAPP. There are two version of the cx1 library that we will use with this project:

- libcx1 for the Power Service Layer Simulation Engine (PSLSE)

- libcxl on the POWER system

I will start off using the PSLSE by pointing the make variable `LIBCXL_DIR` to the PSLSE source directory. This line needs to be updated to point to your copy of PSLSE.

The next item created is the Work Element Descriptor (WED) for my AFU in *capi-vadd.h*. The WED is a single cache line that contains information needed by the AFU to complete its task. For this project, the AFU needs:

- the size of the input vectors
- pointer to the first input array
- pointer to the second input array
- pointer to the output array
- done flag to notify the host of completion
- return value for the number of clock cycles used during operation (AFU uses a 250 MHz clock)

The remaining reservedXX entries are to use the entire 128 byte cache line. The header file for libcxl is also included.

```
#ifndef __CAPI_VADD_H__
#define __CAPI_VADD_H__
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "libcxl.h"
// AFU data uses 128 byte alignment
#define CXL_ALIGNMENT 128

typedef struct {
// Problem size
    __u64 size;
// Input arrays
    void *input1;
    void *input2;
// Output arrays
    void *output;
// Done flag. Marked volatile to block compiler optimizations
    __u64 volatile done;
// Elapsed execution time in clock cycles (AFU uses 250 MHz clock)
    __u64 clock_count;
// reserve entire 128 byte cache line
    __u64 reserved01;
    __u64 reserved02;
    __u64 reserved03;
    __u64 reserved04;
    __u64 reserved05;
    __u64 reserved06;
    __u64 reserved07;
}
```

```

    __u64 reserved08;
    __u64 reserved09;
    __u64 reserved10;
} vadd_wed;
#endif

```

The host code for this example will do the following:

1) Allocating our WED for vector addition

NOTE: All data must be 128 byte aligned. `posix_memalign` is used in this example.

```

vadd_wed* create_wed(unsigned problem_size)
{
    vadd_wed *wed;
    posix_memalign((void**) &wed, CXL_ALIGNMENT, sizeof(*wed));
    // The datatype for this AFU is unsigned
    unsigned num_elements = CXL_ALIGNMENT/sizeof(unsigned);
    // Determine the number of cache lines needed for output data.
    // Write_na only supports data transfers of power of 2.
    // Reserving the last cache line for each array ensures no data corruption
    unsigned cache_lines = (problem_size%num_elements==0) ?
        (problem_size/num_elements):(problem_size/num_elements+1);
    wed->size = problem_size;
    posix_memalign(&wed->input1, CXL_ALIGNMENT, cache_lines*CXL_ALIGNMENT);
    posix_memalign(&wed->input2, CXL_ALIGNMENT, cache_lines*CXL_ALIGNMENT);
    posix_memalign(&wed->output, CXL_ALIGNMENT, cache_lines*CXL_ALIGNMENT);    //
    Set initial values for done flag and clock_count
    // The AFU will write to these values upon completion
    wed->done = 0;
    wed->clock_count = 0;
    return wed;
}

```

2) Setup the input arrays

Initializing the data that will be used by the accelerator. Notice that this data is located in main memory.

```

// Pointers to set the input arrays
// WED has void pointers. This AFU uses unsigned numbers
unsigned *input1 = (unsigned*) wed->input1;
unsigned *input2 = (unsigned*) wed->input2;
for (i=0; i<problem_size; i++)
{
    input1[i] = i;
    input2[i] = i;
}

```

3) Attach to the accelerator

```
// Attach to the AFU. WED is sent to the FPGA
cxl_afu_attach(afu, (__u64) wed);
```

4) Wait for the FPGA to set the done flag

```
// Wait for the AFU to write to done
printf("Waiting for done\n");
while (!wed->done) sleep(1);
```

5) Check the AFU results. Notice the accelerator directly transferred the data using the output pointer

```
// Check AFU output
unsigned errors = 0;
unsigned *output = wed->output;
printf("output: %p\n", output);
for (i=0; i<problem_size; i++)
{
    if(output[i] != 2*i)
    {
        printf("out%i: %X\n", i, output[i]);
        errors++;
    }
}
```

We can use make to check the host code for syntax errors and build our host binary.

You can grab the code using:

```
git checkout host_code
```

Start of RTL design

Before starting the RTL design, let's review part of the CAPI User's Manual. Take note of the following conventions that are used:

- Bit significance: smallest bit number represents the most significant bit
- PSL only support big-endian byte-ordering

The AFU communicates with the PSL using five interfaces:

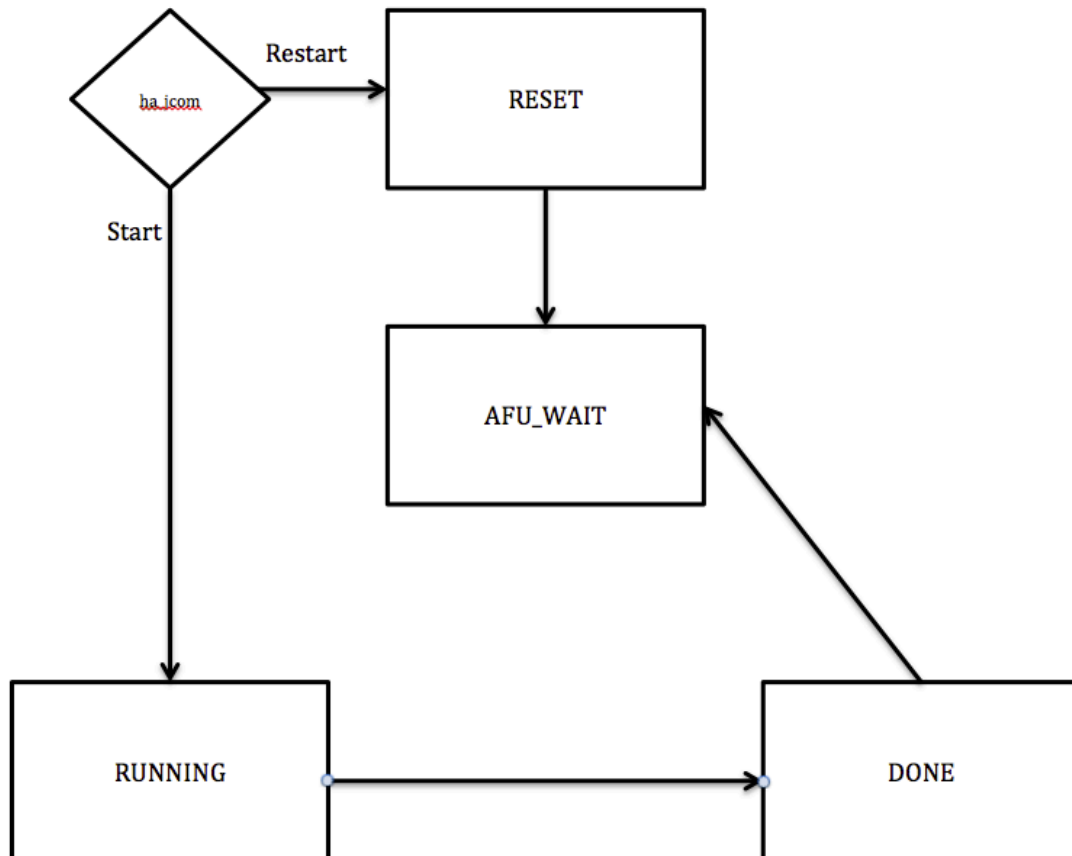
- Accelerator Command Interface
- Accelerator Buffer Interface
- PSL Response Interface
- Accelerator MMIO Interface
- Accelerator Control Interface

All of these interface need to be managed by the AFU for successful operation.

Accelerator Control Interface

The Accelerator Control Interfaced is used by the PSL to manage the AFU. All the signals in the AFU-PSL interface starting with ha are sourced from the PSL and ah are provided by the accelerator (RTL design). Currently the two supported PSL Control Commands are Start (0x90) and Reset (0x80). This interface will be managed by `job_control` in `job.vhd`. The port for `job_control` will contain all the signals in the Accelerator Control Interface.

The architecture of `job_control` creates a state machine to handle the Start and Reset commands. The states are: POWERON, RESET, AFU_WAIT, RUNNING, DONE.



Two processes are used to interface with the Accelerator Control Interface. The `MY_STATE` process manages which state will occur next and the `MY_SIGNALS` process sets the output signals. `ah_jcack`, `ah_jyield`, and `ah_tbreq` are currently not used and set according to the user guide.

Using PSLSE

We will use *afu.vhd* to represent our entire vector add AFU. For now, only the Control Interface will be used. The other 4 interfaces are unused and set to a default value.

The project up to this point:

```
git checkout afu_control
```

The toplevel design for our simulation is included with the PSLSE in the `afu_driver`. The most recent version of *top.v* from PSLSE should be used in the project

```
cp <path-to-pslse>/pslse/afu_driver/Verilog/top.v <path-to-project>
```

Xsim

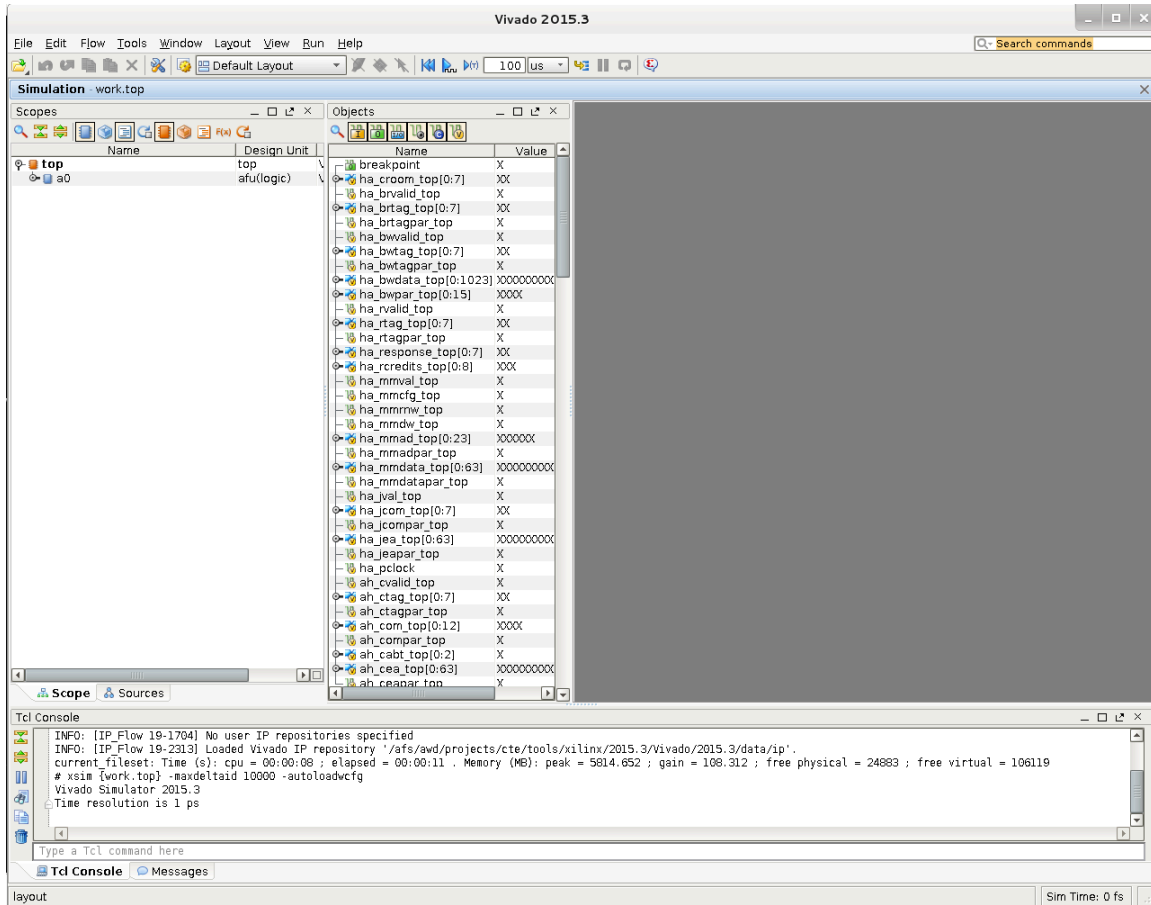
The *runXsim.sh* bash script located in `sim/xsim` directory will compile the source RTL code and start the simulation in Xsim.

```
#!/bin/bash
SRC=/home/khill/pslse_projects/vadd-walkthrough/vadd
ROOT_DIR=/home/khill/pslse_src/pslse/afu_driver/src
xvhd1 $SRC/vadd_pkg.vhd
xvhd1 $SRC/mmio.vhd
xvhd1 $SRC/job.vhd
xvhd1 $SRC/datapath.vhd
xvhd1 $SRC/afu.vhd
xelab -svlog $SRC/top.v -sv_root $ROOT_DIR -sv_lib libdpi -debug all
xsim -g work.top
```

The `SRC` and `ROOT_DIR` variables will need to be updated to point to the RTL source and `afu_driver` built in PSLSE. Additional source files can be added in the script using `xvhd1` for VHDL files and `xvlog` for Verilog/SystemVerilog. Source files should be included following their dependency order. Unused source files can be commented out with `#` throughout this tutorial.

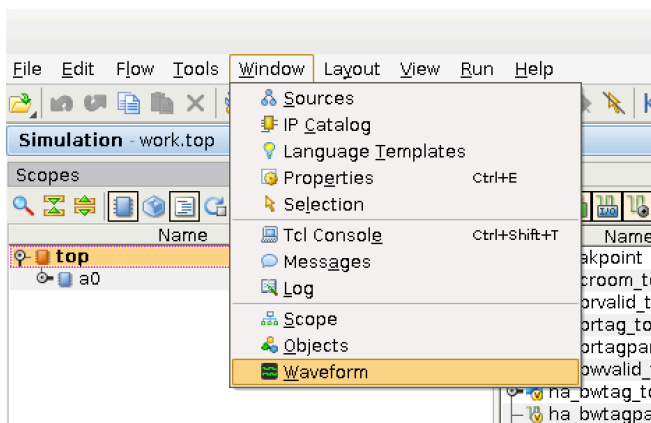
```
./runXsim.sh
```

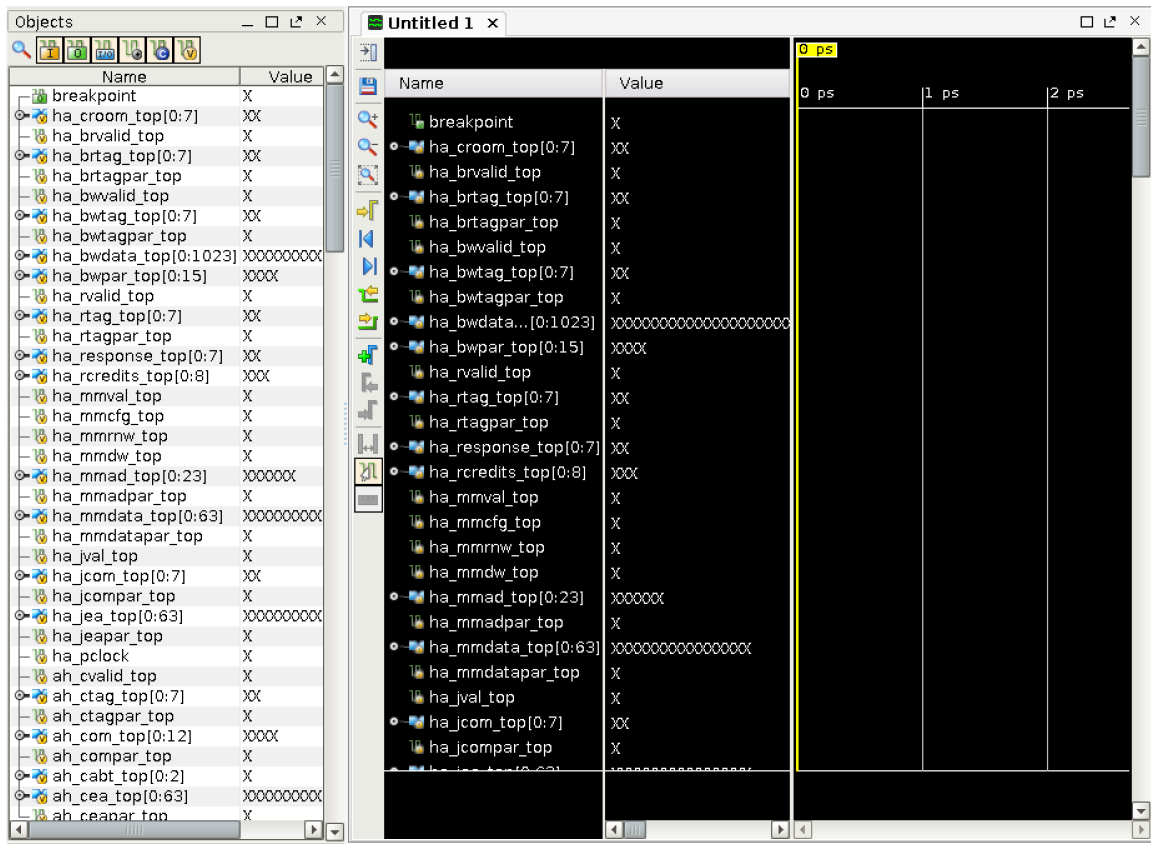
The above command will compile all the RTL source files and start the simulation in Xsim.



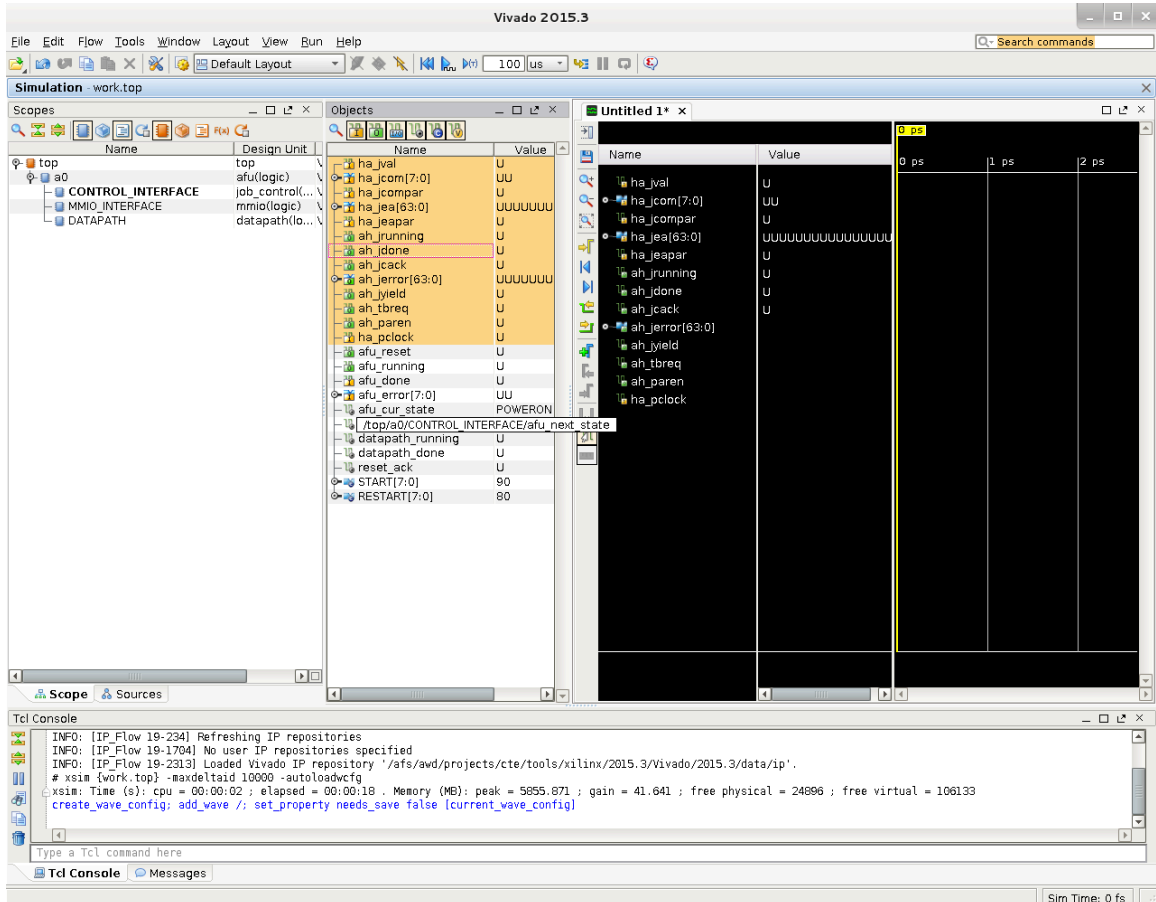
The Xsim GUI will open with the Scopes and Objects panes open by default.

The waveform can be opened with Window → Waveform





By default, the waveform will be populated with all the signals from *top.v*



Individual signals can be added to the waveform by selecting the component in the Scopes and dragging them from the Objects. The above figure is adding the Control Interface signals from `CONTROL_INTERFACE` component.

Signals can also be added to the waveform by sourcing the *.tcl scripts located at sim/xsim. Scripts can be sourced in the 'Tcl Console' of Xsim, which is located at the bottom of the window by default. The following command will add the Control Interface:

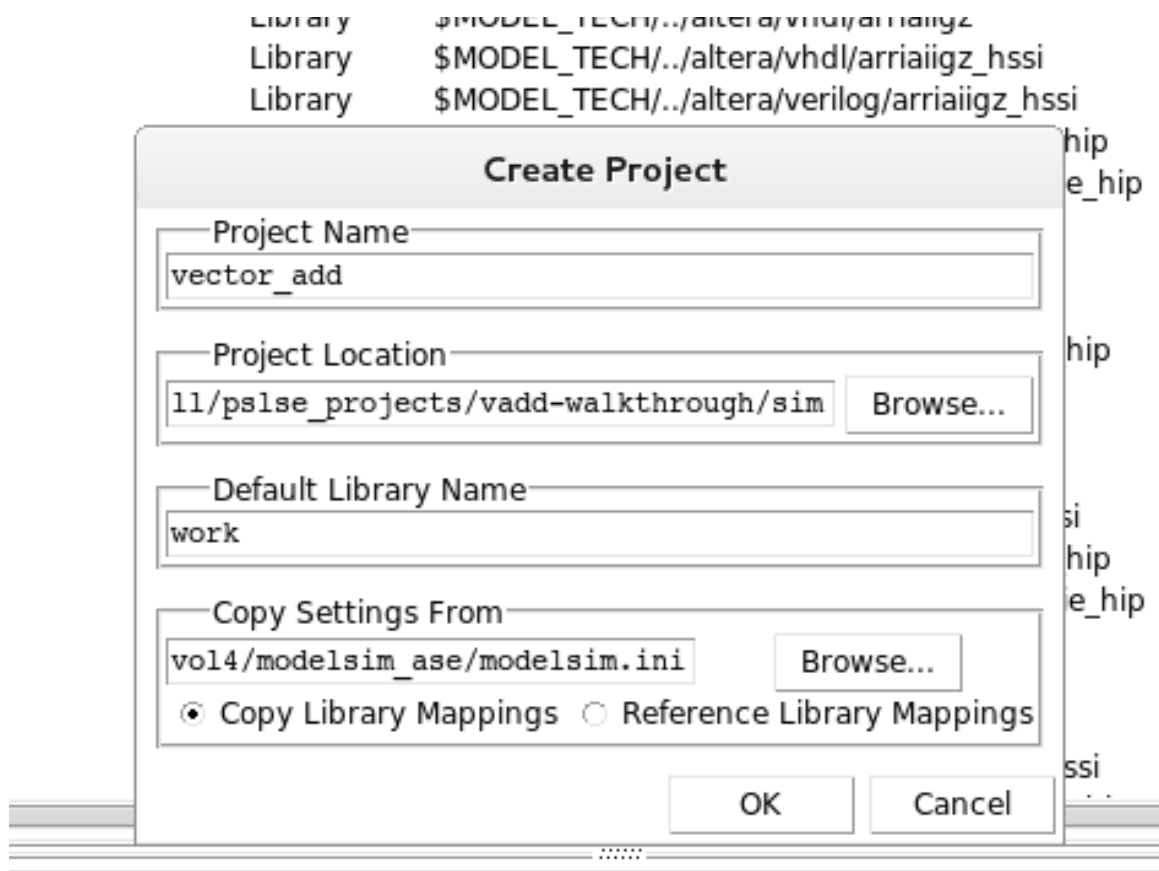
```
source control.tcl
```

NOTE: Script files may be removed by `git checkout <tag>`. You can make a local copy from the master branch (e.g. `cp <root-dir>/sim/xsim runXsim.sh <root-dir>/sim`). The master branch no longer tracks the `vadd/top.v` testbench file. You should always use the most recent version packaged with PSLSE found at `<pslse-root>/afu_driver/Verilog/top.v`. Git may require `vadd/top.v` be restored before changing to a different tag. You can restore a file in the current branch with `git checkout vadd/top.v` in the projects root directory (all modifications will be deleted!)

The simulation can be started using the blue arrows at the top center of the window. The simulation will appear to hang while waiting for the PSLSE to connect to Xsim. This process is described after the ModelSim section below.

ModelSim

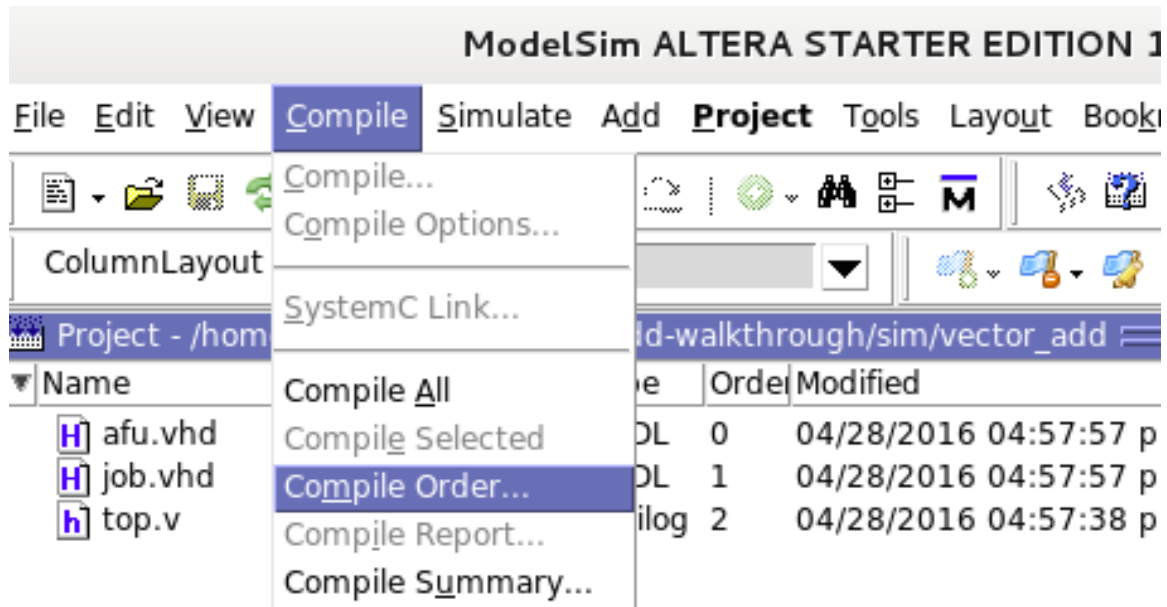
First we will setup a new project with File->New->Project



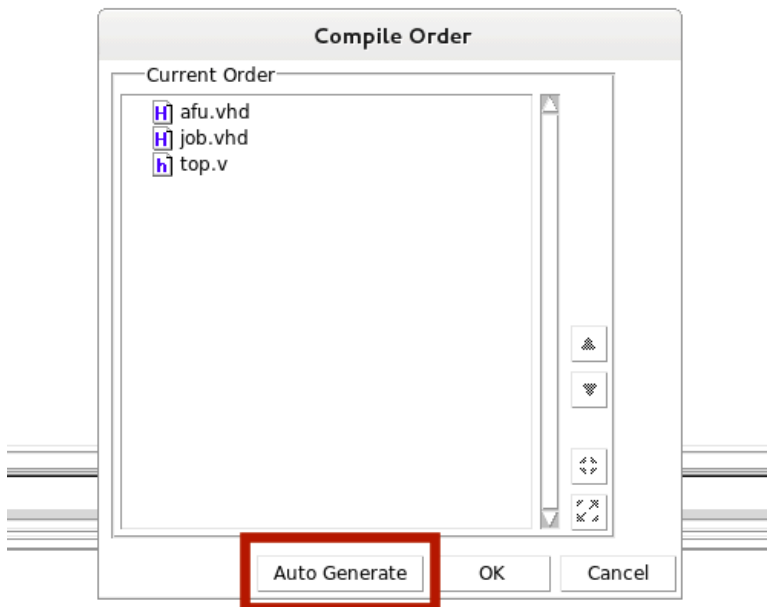
```
'proj/p3/cte/tools/altera/vol4/modelsim_ase/linuxaloem/./modelsim.i
```

Fill in the Project Name and Project Location and push OK. My project is vector_add and is located in sim directory. Be sure to leave the Default Library Name as work.

After the project opens, add the three design files by right-clicking on the project tab and Add to Project->Existing File



Select the Auto Generate button to automatically find the compilation order needed by the project

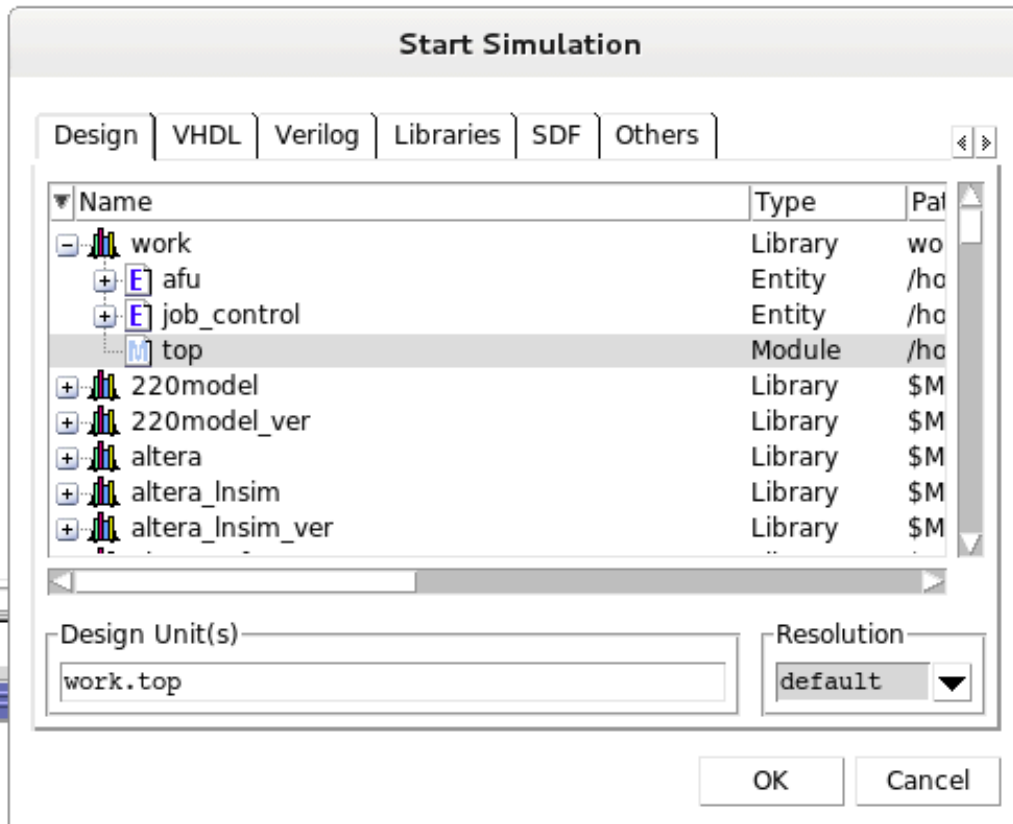


After the compilation order is determined, the project can be recompiled using the 'project compileall' command in the transcript window

```
Transcript :  
# Loading project vector_add  
# Compile of job.vhd was successful.  
# Compile of top.v was successful.  
# Compile of afu.vhd was successful.  
ModelSim> project compileall  
# Compile of job.vhd was successful.  
# Compile of top.v was successful.  
# Compile of afu.vhd was successful.  
# 3 compiles, 0 failed with no errors.  
ModelSim> |
```

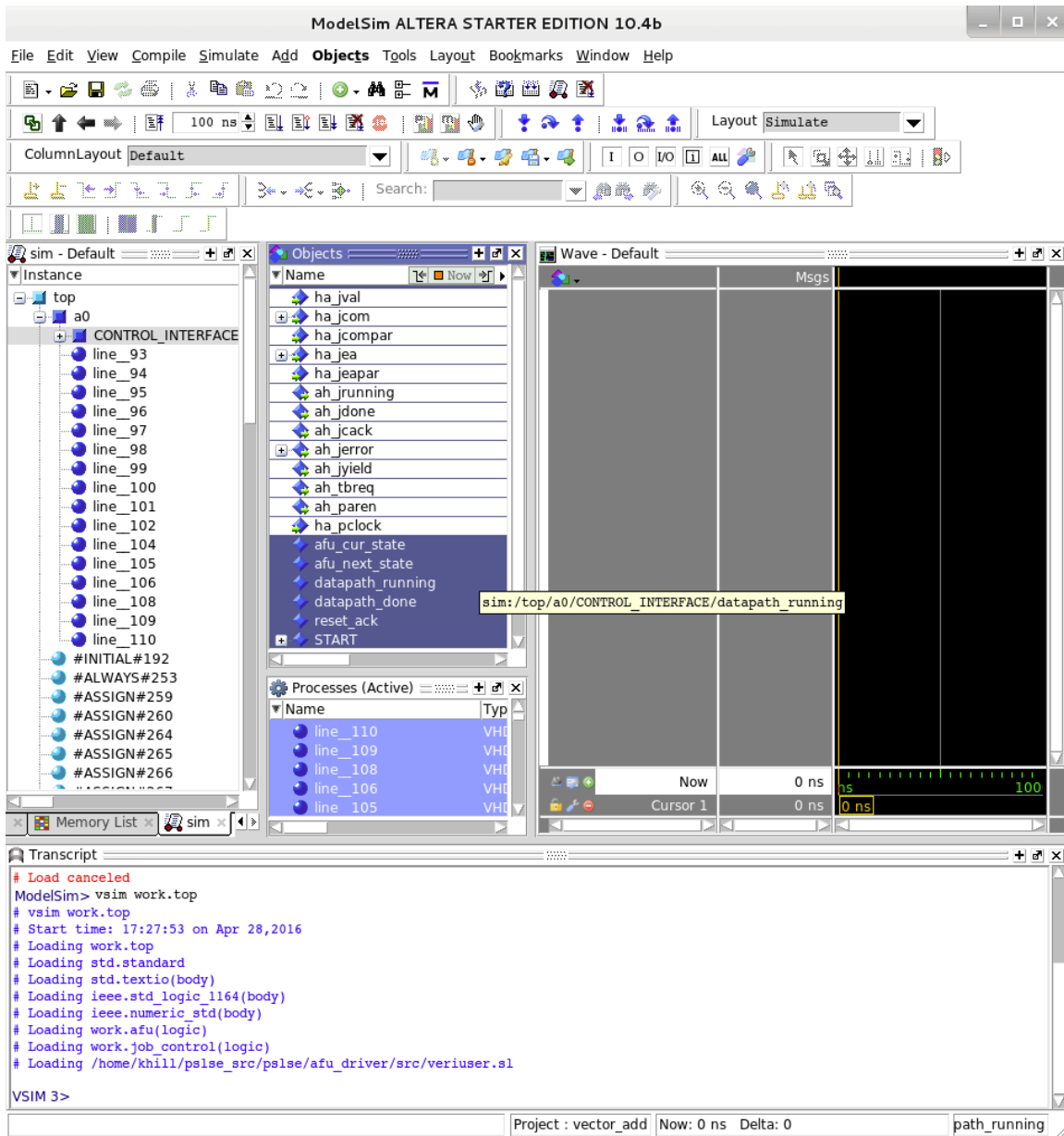
The simulation can now be started with Simulate->Start Simulation

The toplevel for simulation is the top Design in the work Library

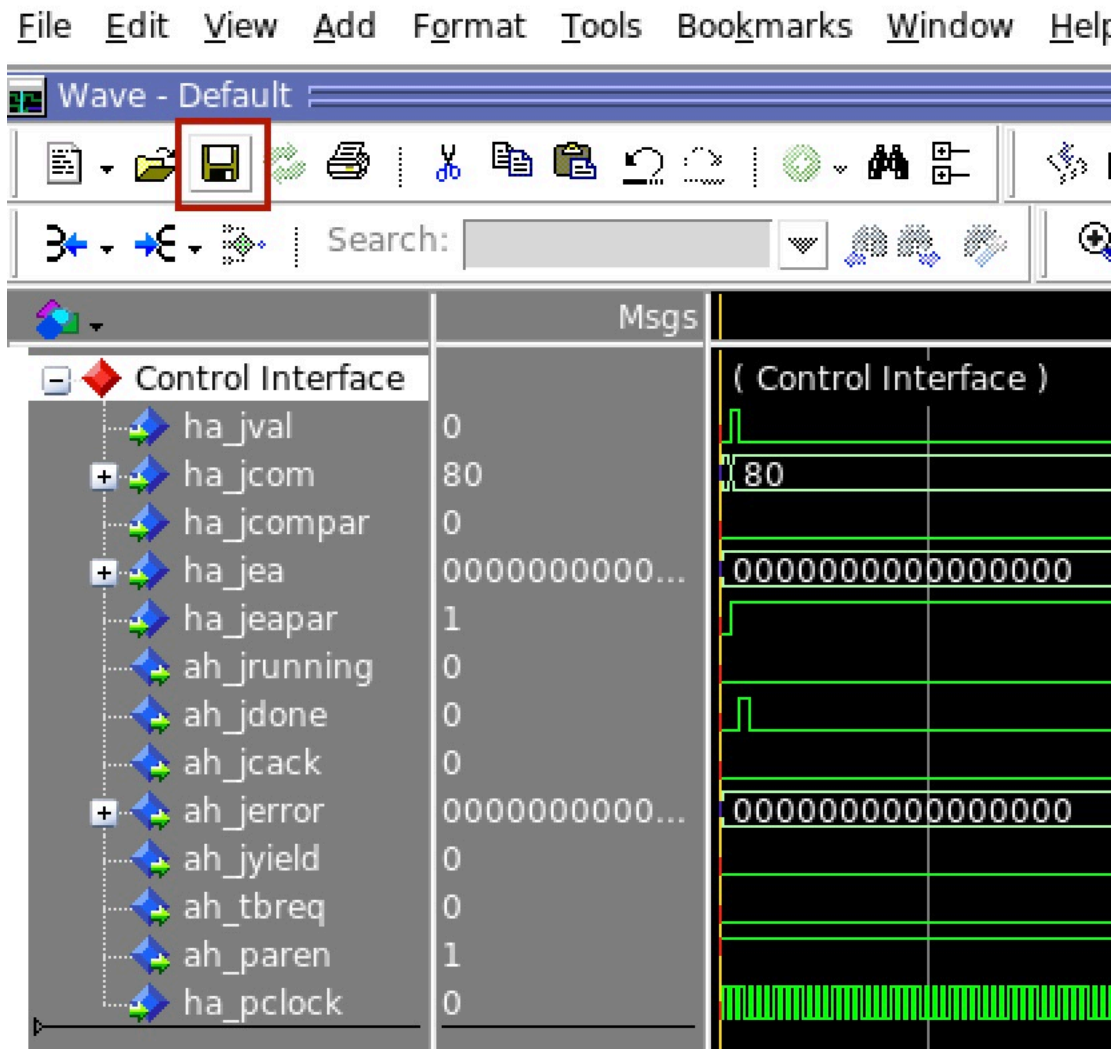


The simulation can also be started with 'vsim work.top' in the transcript window.

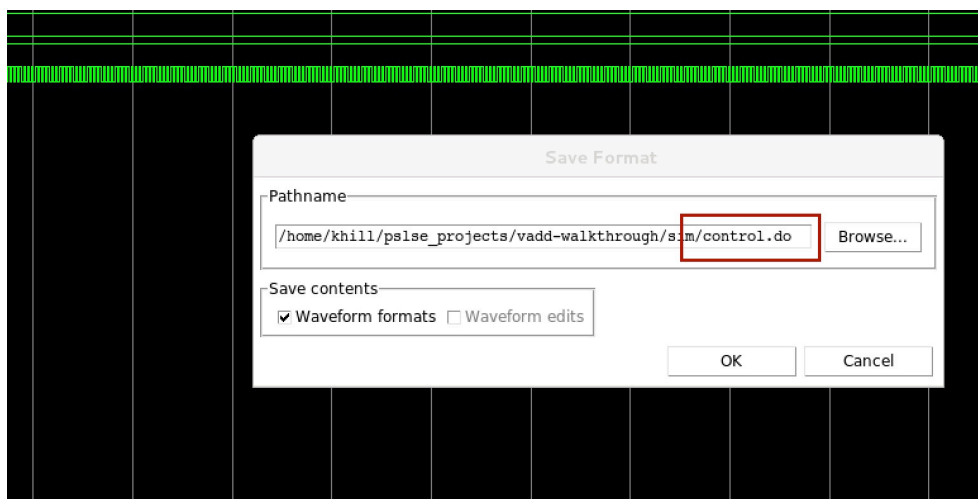
The simulation window will now open with the sim, Object, Process, and Wave tabs open. Individual designs can be selected in the sim tab to the left. The Object window will be populated with the signals from CONTROL_INTERFACE. This is the label used in *afu.vhd* to instantiate our `job_control` design. Individual signals can be added to the wave window by dragging from the objects tab.



The signals in the wave will be displayed and updated during simulation. The signals can be grouped together by highlighting and *right-click->group*.



The save button will write the signals and attributes for the current wave to a *.do file.



The command 'do <my_wave>.do' can be used to add the signal group to the Wave in the future. To start the simulation, use 'run 400ns' command in the transcript window. ModelSim may appear to hang while waiting to connect to PSLSE. Scripts for each AFU-PSL interface are located in sim/modelSim directory.

NOTE: Script files may be removed by `git checkout <tag>` . You can make a local copy from the master branch (e.g. `cp <root-dir>/sim/xsim runXsim.sh <root-dir>/sim`). The master branch no longer tracks the *vadd/top.v* testbench file. You should always use the most recent version packaged with PSLSE found at *<pslse-root>/afu_driver/Verilog/top.v* . Git may require *vadd/top.v* be restored before changing to a different tag. You can restore a file in the current branch with `git checkout vadd/top.v` in the projects root directory (all modifications will be deleted!)

PSLSE

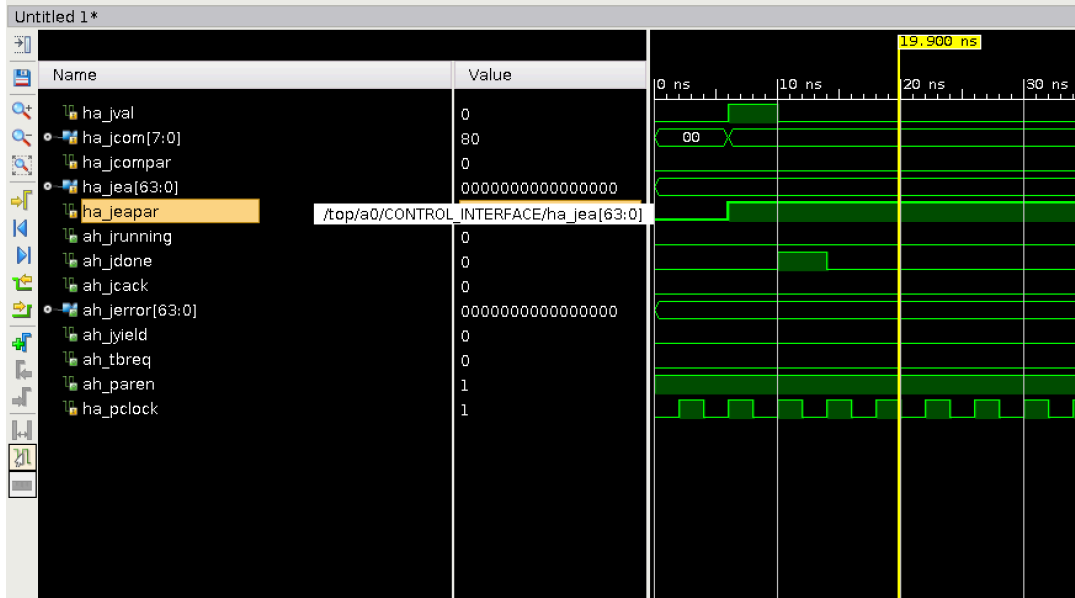
In a new terminal, go to the pslse directory and start pslse

```
cd <path-to-pslse>/pslse/pslse
./pslse
```

The terminal will look something like this

```
INFO:PSLSE version 1.002 compiled @ Apr 28 2016 16:22:18
INFO:PSLSE parm values:
    Seed      = 13
    Timeout   = 10 seconds
    Response  = 16%
    Paged     = 3%
    Reorder   = 86%
    Buffer     = 82%
INFO:Attempting to connect AFU: afu0.0 @ localhost:32768
PSL_SOCKET: Using PSL protocol level : 0.9908.1
INFO:Clocking afu0.0
```

NOTE: PSLSE will attempt to connect to the simulator using port 32768 by default. The simulator (Xsim or ModelSim) will prompt which port it is listening for PSLSE. The port number can be change by editing *shim_host.dat* in the pslse directory.



At this point, the AFU acknowledges the RESET command sent by the PSL by toggling the `ah_jdone` signal for one clock cycle. The next section will send the AFU Descriptor (described in the CAPI User's Guide) through the MMIO Interface.

Accelerator MMIO Interface

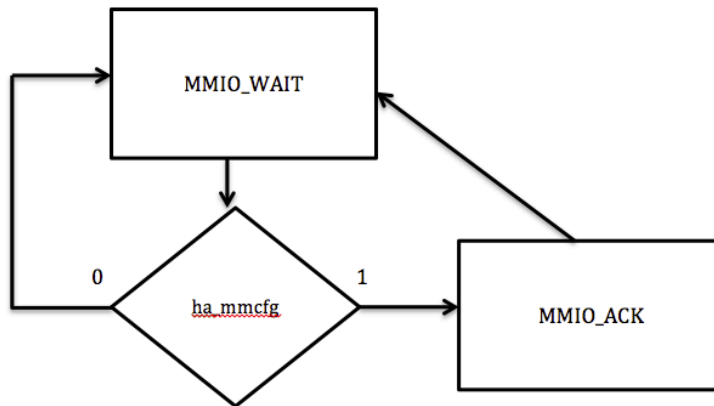
The PSL will request for the AFU Descriptor when it discovers an AFU. Table 4-1 of the CAPI User's Guide defines the AFU Descriptor. This simple example needs to set 'num_of_processes' and 'req_prog_model'. All other fields are set to '0'. This design uses the dedicated-process mode.

```

CASE mmio_cur_state IS
  WHEN MMIO_WAIT =>
    WHEN MMIO_ACK =>
      CASE ha_mmad IS
        WHEN AFU_DESC =>
          mmio_cfg_read <= X"0000" & X"0001" & X"0000" & X"8010";
        WHEN OTHERS =>
          mmio_cfg_read <= (OTHERS => '0');
      END CASE;
      ah_mmack <='1';
    END CASE;
END CASE;

```

The rest of the `mmio` design is in `mmio.vhd`. Notice that `mmio_cfg_read` is define as `std_logic_vector(0 to 63)` to match the bit significance of the AFU descriptor in Table 4-1. Similar to `job_control`, `mmio` uses a two-process design. The states for `mmio` are `MMIO_WAIT` and `MMIO_ACK`.



A user package, *vadd_pkg.vhd*, has also been added to handle odd parity and other utility functions for the PSL-AFU Interface. The *afu.vhd* has been updated to add the `mmio` module into the design.

The AFU will now respond with the AFU Descriptor to the PSL and can attach to a host process. The ModelSim project must be recompiled to reflect these changes. The compilation order may need to be recalculated to successfully compile the project.

The project at this point is:

```
git checkout afu_mmio
```

Use the most recent version of *top.v* from PSLSE:

```
cp <path-to-pslse>/pslse/afu_driver/Verilog/top.v <path-to-project>
```

Xilinx designs:

- Add *vadd_pkg.vhd* and *mmio.vhd* to *runXsim.sh*
- Start the simulation with './runXsim.sh'
- Add the MMIO Interface signals with 'source mmio.tcl'

Altera designs:

- Add *vadd_pkg.vhd* and *mmio.vhd* to ModelSim project
- Recompile ModelSim project

- Start the simulation with 'vsim work.top'
- Add the MMIO Interface signals with 'do mmio.do'



Run the simulation continuously and start the PSLSE in a new terminal:

```
cd <path-to-pslse>/pslse/pslse
./pslse
```

Open an additional terminal to start the AFU host code. Add 'pslse_server.dat' with the line 'localhost:16384' to the same directory as the source code. The LD_LIBRARY_PATH needs to be updated to point to libcxl included with pslse. For me:

```
cd <path-to-host-src>
make
LD_LIBRARY_PATH=/home/khill/pslse_src/pslse/libcxl ./capi-vadd
```

The host terminal will look something like this:

```
khill@sleight:~/pslse_projects/vadd-walkthrough ((no branch))$ LD_LIBRARY_PATH=/
home/khill/pslse_src/pslse/libcxl ./capi-vadd
Problem size: 128
INFO:Connecting to host 'localhost' port 16384
Creating Work Element Descriptor
WED: 0x674600
Size: 80
input1: 0x674700
input2: 0x674980
output: 0x674c00
Waiting for done
```

and PSLSE will show

hardware:

```
unsigned *in_ptr1;
unsigned *in_ptr2;
unsigned *out_ptr;

for (int curr_pos=0; curr_pos<problem_size; curr_pos++)
{
    unsigned input1_data = *in_ptr1++;
    unsigned input2_data = *in_ptr2++;
    unsigned output_data = input1_data + input2_data;
    *out_ptr++ = output_data;
}
```

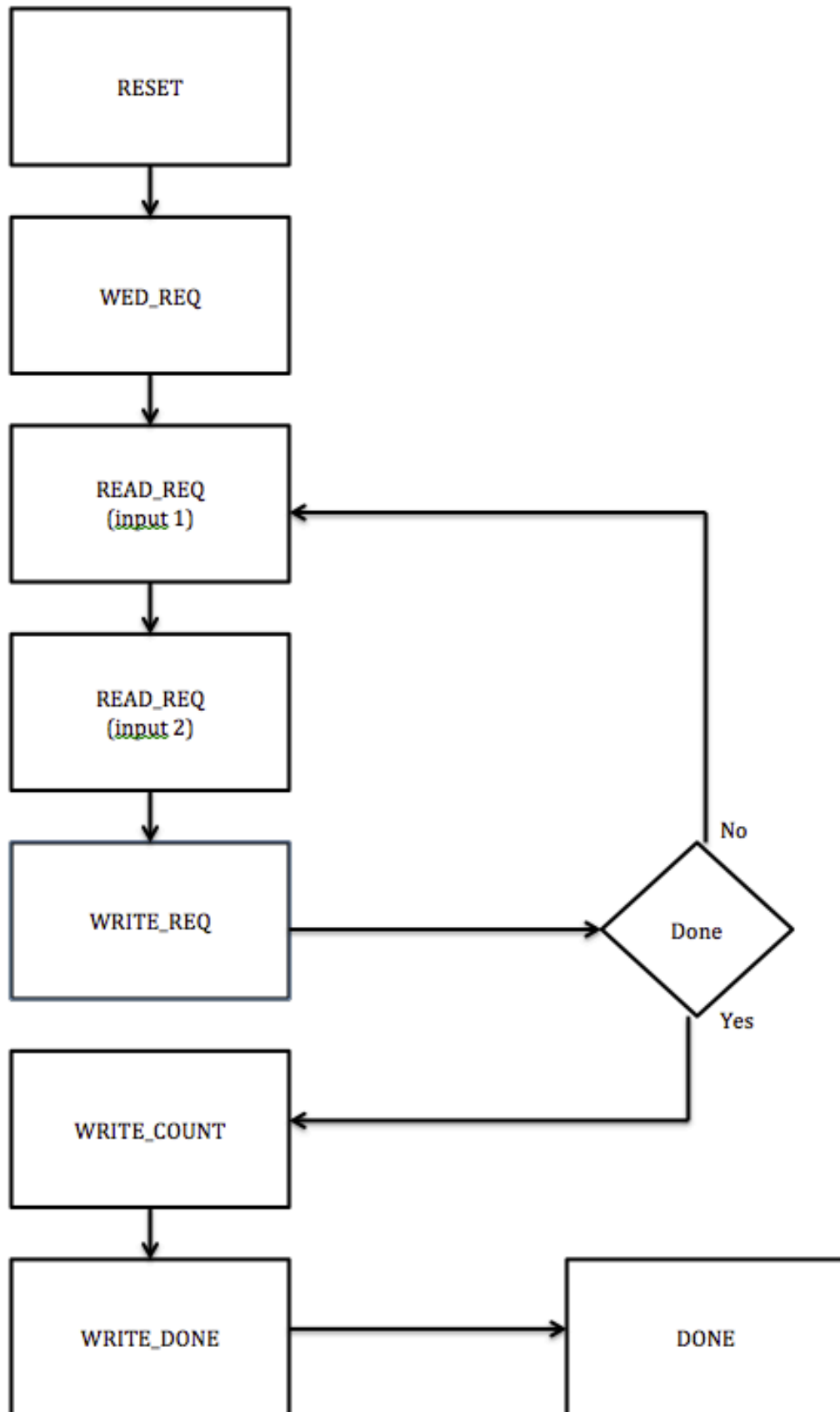
The `in_ptr1`, `in_ptr2`, `out_ptr`, and `problem_size` are assigned using the Work Element Descriptor (WED).

```
IF(wed_valid = '1' AND word_index = '0') THEN
    problem_size <= bwddata(SIZE_RANGE);
    in_ptr1 <= bwddata(IN1_RANGE);
    in_ptr2 <= bwddata(IN2_RANGE);
    out_ptr <= bwddata(OUT_RANGE);
```

The PSL will send/receive 128 byte cache lines to the datapath component for each read/write request using the Buffer Interface. The hardware pointers use an increment of 128 to read/write successive cache lines:

```
-- Increment input1 pointer
in_ptr1 <= STD_LOGIC_VECTOR(UNSIGNED(in_ptr1) +
    TO_UNSIGNED(128, in_ptr1'LENGTH));
```

The hardware will unroll the loop by a factor of $128\text{B}/\text{sizeof}(\text{unsigned}) = 32$. Each task in the loop will be performed on 32 values at a time. The states used in the datapath component are: RESET, WED_REQ, READ_REQ, WRITE_REQ, WRITE_DONE, WRITE_COUNT, WAIT_REQ, and DONE. The AFU will wait in the WAIT_REQ state for an acknowledgment on the Response Interface for each command sent on the Command Interface. The state machine for this component is:



The `my_signals` process sets the signals for the Command Interface. The tags used for the different commands are:

- WED_TAG
- INPUT1_TAG
- INPUT2_TAG
- OUTPUT_TAG
- DONE_TAG
- COUNT_TAG

For example, the WED_REQ state sets the following signals to read the WED from system memory:

```
WHEN WED_REQ =>
    ah_cvalid <= '1';
    ah_ctag <= WED_TAG;
    ah_ctagpar <= odd_parity(WED_TAG);
    ah_com <= READ_CL_NA;
    ah_compar <= odd_parity(READ_CL_NA);
    ah_cea <= ha_jea;
    ah_ceapar <= odd_parity(ha_jea);
```

This will issue a cache non-allocating read to the PSL to read the WED from main memory. The WED address (`ha_jea`) was sent on the Control Interface during the ‘Start’ command. Each tag has a similar definition to complete the specified read/write operation.

The acknowledgments from the Response Interface are set by:

```
-- Response tags received on Response Interface
in1_response <= '1' WHEN ha_rtag = INPUT1_TAG AND ha_rvalid = '1' ELSE '0';
in2_response <= '1' WHEN ha_rtag = INPUT2_TAG AND ha_rvalid = '1' ELSE '0';
out_response <= '1' WHEN ha_rtag = OUTPUT_TAG AND ha_rvalid = '1' ELSE '0';
reset_response <= '1' WHEN ha_rtag = RESET_TAG AND ha_rvalid = '1' ELSE '0';
```

The datapath will move on to the next operation when receiving a `DONE` from the Response Interface. The appropriate pointers will be incremented by 128 bytes when `in1_response`, `in2_response`, or `out_response` are asserted. The datapath will send a ‘restart’ command on the Command Interface when receiving `FLUSHED` or `PAGED` response. The datapath will then retry the previous issued command.

```
-- Decode response codes from Response Interface
-- DONE: Successful command completion
-- FLUSHED: PSL in flush state. Need to send reset command
-- PAGED: O/S has requested AFU to continue. Need to send reset command
-- All other responses indicate an error for this AFU
```

```

response_done <= '0';
response_retry <= '0';
response_error <= '0';
IF(ha_rvalid = '1') THEN
    CASE ha_response IS
        WHEN DONE_RESP =>
            response_done <= '1';
        WHEN PAGED_RESP =>
            response_retry <= '1';
        WHEN FLUSHED_RESP =>
            response_retry <= '1';
        WHEN OTHERS =>
            response_error <= '1';
    END CASE;
END IF;

```

All others responses from the Response Interface are considered errors and will cause the AFU to terminate early. The `ah_jerror` signal on the Control Interface is forwarded the response causing the failure.

The Buffer Interface is managed with:

```

-- Data valid for writes on Buffer Interface
wed_valid <= '1' WHEN ha_bwtag = WED_TAG AND ha_bwvalid = '1' ELSE '0';
in1_valid <= '1' WHEN ha_bwtag = INPUT1_TAG AND ha_bwvalid = '1' ELSE '0';
in2_valid <= '1' WHEN ha_bwtag = INPUT2_TAG AND ha_bwvalid = '1' ELSE '0';
-- Power Service Layer is Big-endian
bwdata <= endian_swap(ha_bwdata);

-- Select output for read on Buffer Interface
output_line <= (OTHERS => '1') WHEN ha_brtag = DONE_TAG
    ELSE STD_LOGIC_VECTOR(SHIFT_LEFT(RESIZE(UNSIGNED(clock_count),
output_line'LENGTH), 320))
    WHEN ha_brtag = COUNT_TAG
    ELSE vadd(input1_data, input2_data);
-- Power Service Layer is Big-endian
ah_brdata <= endian_swap(output_data);
ah_brpar <= odd_parity_64dw(output_data);

```

Data sent to the AFU during ‘read_na’ commands are buffered 64 bytes each clock cycle (half cache line).

```

IF(in1_valid = '1') THEN
    -- Grab entire cache line for input1 read command
    CASE word_index IS
        WHEN '0' =>
            input1_data(511 DOWNT0 0) <= bwdata;
        WHEN '1' =>
            input1_data(1023 DOWNT0 512) <= bwdata;
        WHEN OTHERS =>
            -- Do nothing
    END CASE;
END IF;
IF(in2_valid = '1') THEN
    -- Grab entire cache line for input2 read command
    CASE word_index IS
        WHEN '0' =>
            input2_data(511 DOWNT0 0) <= bwdata;
        WHEN '1' =>
            input2_data(1023 DOWNT0 512) <= bwdata;
        WHEN OTHERS =>
            -- Do nothing
    END CASE;
END IF;

```

```

        WHEN '1' =>
            input2_data(1023 DOWNT0 512) <= bwdata;
        WHEN OTHERS =>
            END CASE;
    END IF;

```

Data written from the AFU to the host during a 'write_na' command are delayed to match the latency specified by ah_brlat.

```

-- Register output data to match read latency of Buffer Interface
IF(brad = "000001") THEN
    output_data <= output_line(1023 DOWNT0 512);
ELSE
    output_data <= output_line(511 DOWNT0 0);
END IF;

```

The input1_data and input2_data are added together using the vadd function from vadd_pkg:

```

FUNCTION vadd(input1 : STD_LOGIC_VECTOR; input2 : STD_LOGIC_VECTOR)      RETURN
STD_LOGIC_VECTOR IS
    VARIABLE result      : STD_LOGIC_VECTOR(input1'RANGE);
    VARIABLE DATA_SIZE  : INTEGER := 4*8;
    VARIABLE NUM_ADD     : INTEGER := input1'LENGTH/(DATA_SIZE);
BEGIN
    FOR i IN 0 TO NUM_ADD-1 LOOP
        result((i+1)*DATA_SIZE-1 DOWNT0 i*DATA_SIZE) := STD_LOGIC_VECTOR(
            UNSIGNED(input1((i+1)*DATA_SIZE-1 DOWNT0 i*DATA_SIZE)) +
            UNSIGNED(input2((i+1)*DATA_SIZE-1 DOWNT0 i*DATA_SIZE)));
    END LOOP;
    RETURN result;
END vadd;

```

This function will add 32 numbers at a time for each 128B cache line stored in the input1_data and input2_data registers.

The final project is:

```
git checkout afu_complete
```

The sim directory contains *.tcl and *.do files to update the signals displayed by Xsim or ModelSim with each PSL-AFU interface.

Xilinx designs:

```

./runXsim.do
source mmio.tcl
source response.tcl
source control.tcl

```

```
source command.tcl
source buffer.tcl
```

Altera designs:

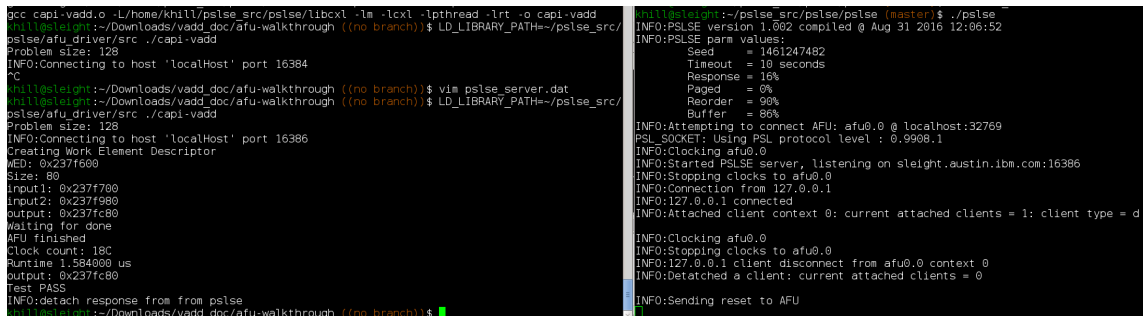
```
project compileall
vsim work.top
do datapath.do
```

In PSLSE terminal:

```
cd <path-to-pslse>
./pslse
```

In host terminal:

```
cd <path-to-project>
LD_LIBRARY_PATH=<pslse-root>/afu_driver/src:$LD_LIBRARY_PATH ./capi-vadd
```



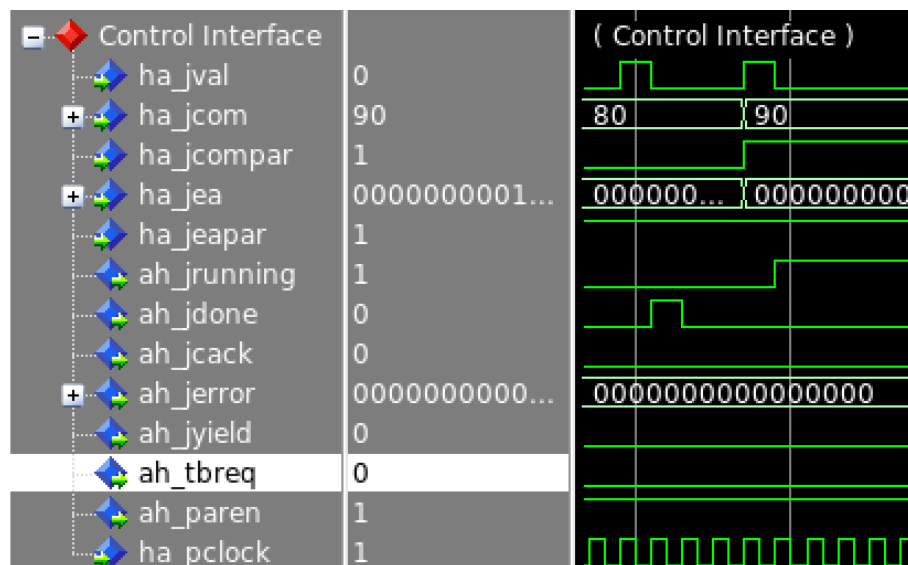
The image shows two terminal windows side-by-side. The left window shows the execution of the 'capi-vadd' program, which connects to the PSLSE on localhost:16384 and runs a test. The right window shows the PSLSE terminal output, which displays the PSLSE version, parameters, and the connection from the host.

```
gcc capi-vadd.o -L/home/khill/pslse_src/pslse/libcxl -lm -lcxl -lpthread -lrt -o capi-vadd
khill@slight:~/Downloads/vadd doc/afu-walkthrough ((no branch))$ LD_LIBRARY_PATH=~/.pslse_src/pslse/afu_driver/src ./capi-vadd
Problem size: 128
INFO:Connecting to host 'localhost' port 16384
^C
khill@slight:~/Downloads/vadd doc/afu-walkthrough ((no branch))$ vim pslse_server.dat
khill@slight:~/Downloads/vadd doc/afu-walkthrough ((no branch))$ LD_LIBRARY_PATH=~/.pslse_src/pslse/afu_driver/src ./capi-vadd
Problem size: 128
INFO:Connecting to host 'localhost' port 16386
Creating Work Element Descriptor
MED: 0x237f600
Size: 80
Input1: 0x237f700
Input2: 0x237f980
Output: 0x237fc80
Waiting for done
AFU finished
Clock counts: 18C
Runtime 1.584008 us
Output: 0x237fc80
Test PASS
INFO:detach response from from pslse
khill@slight:~/Downloads/vadd doc/afu-walkthrough ((no branch))$
```

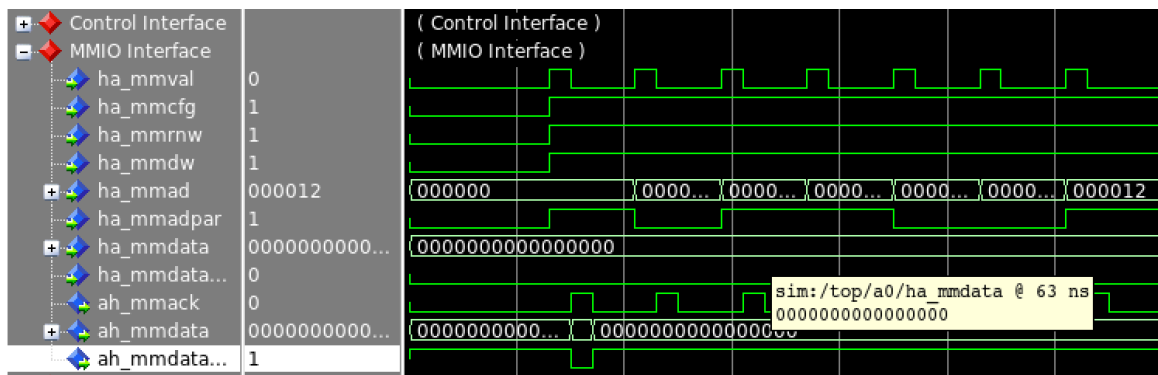
```
khill@slight:~/pslse_src/pslse/pslse (master)$ ./pslse
INFO:PSLSE version 1.002 compiled @ Aug 31 2016 12:06:52
INFO:PSLSE para values:
Seed = 1461247482
Timeout = 10 seconds
Response = 10%
Paged = 0%
Reorder = 90%
Buffer = 80%
INFO:Attempting to connect AFU: afu0.0 @ localhost:32769
PSL_SOCKET: Using PSL protocol level : 0.9908.1
INFO:Clocking afu0.0
INFO:Started PSLSE server, listening on slight.austin.ibm.com:16386
INFO:Stopping clocks to afu0.0
INFO:Connection from 127.0.0.1
INFO:127.0.0.1 connected
INFO:Attached client context 0: current attached clients = 1: client type = d
INFO:Clocking afu0.0
INFO:Stopping clocks to afu0.0
INFO:127.0.0.1 client disconnect from afu0.0 context 0
INFO:Detached a client: current attached clients = 0
INFO:Sending reset to AFU
```

The PSLSE can be killed after the capi-vadd detaches from the PSLSE.

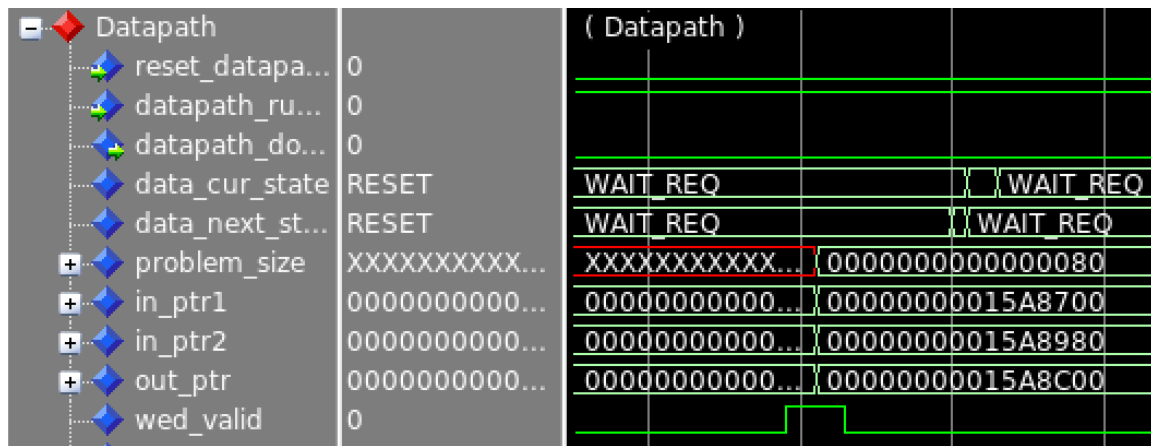
On the Control Interface, the AFU acknowledges a RESET (0x80) command and accepts the START (0x90) command.



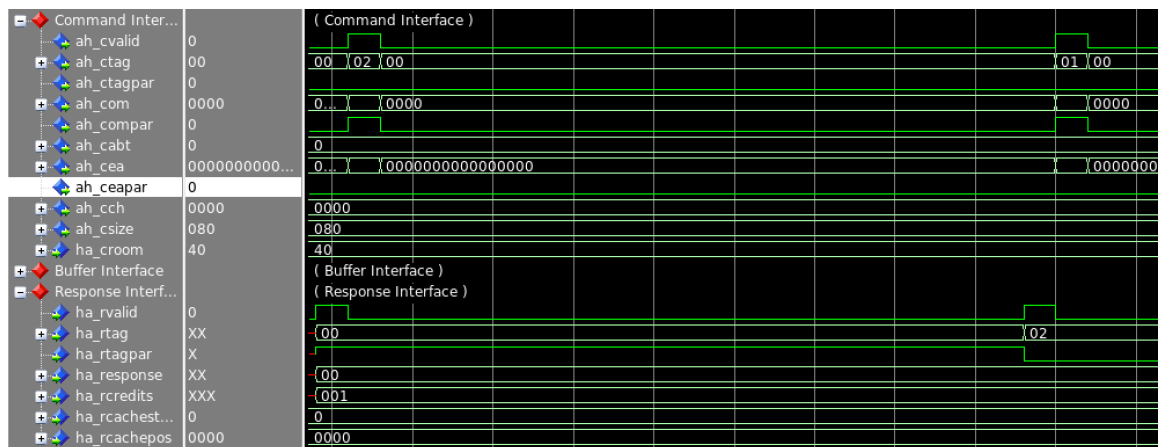
On the MMIO Interface, the AFU sends over the AFU Descriptor to the processor.



Here the WED is being saved into the `problem_size`, `in_ptr1`, `in_ptr2`, and `out_ptr` registers.



On the Command Interface, the AFU is issuing commands with tag 0x02 and 0x01. The response for tag 0x02 is shown on the Response Interface.



Build Project for FPGA

For Xilinx, complete steps 1-10 of Section 8.4.2 in the IBM CAPI Users Guide Xilinx edition. The Xilinx build flow sets up the Vivado project using perl scripts. On my machine, I had to call the scripts with 'perl ./write_vivado_prj_files ...'.

For Altera, complete steps 1-6 of Section 8.4.2 in the IBM CAPI Users Guide Altera edition. Note for step 3, afu0.qip will need to be updated using '-name VHDL_FILE' flag.

Configure FPGA with CAPI Flash script on POWER8

On x86 build machine:

1) Clone capi-flash-script with:

```
git clone https://github.com/kenhill/capi-flash-script.git
```

2) Check that quartus_pgm or vivado are included in the PATH variable

```
which quartus_pgm    or    which vivado
```

3) Create CAPI flash file:

```
cd <path-to-capi-flash>/x86  
./create_flash_file.sh
```

4) Copy .dat file from x86 build machine:

```
scp <afu>.dat <power-user>@<power-ip>:~/<path-to-walkthrough>
```

On POWER machine:

1) Clone capi-flash-script:

```
git clone https://github.com/kenhill/capi-flash-script.git
```

2) Configure FPGA on POWER machine with:

```
sudo <path-to-capi-flash>/capi-flash-fpga.sh <path-to-dat-file>/<afu>.dat
```

Run on POWER

Clone and build libcxl for the POWER system:

```
git clone https://github.com/ibm-capi/libcxl.git  
cd libcxl  
make
```

Build and run project host code

```
cd <path-to-walkthrough>
make
LD_LIBRARY_PATH=<path-to-libcxl>:$LD_LIBRARY_PATH ./capi-vadd
```

Conclusions

This project demonstrates how to use the PSL-AFU interface for CAPI and is not optimized for performance. One item ignored by this project is the `ha_croom` signal in the Command Interface. The `ha_croom` signal represents the number of commands that can be queued up at a time with PSL. This vector addition example only has 1 command pending at any given time (as opposed to 0x40 or 64 commands reported by the PSLSE). A more optimal design would queue up as many commands as possible and use the `ha_rcredits` returned by the Response Interface as a flow control.