The VividSparks FPGA Co-simulation Development Flow

This document details the RacEr/UltrasoniK/SupersoniK setup and the examples provided.

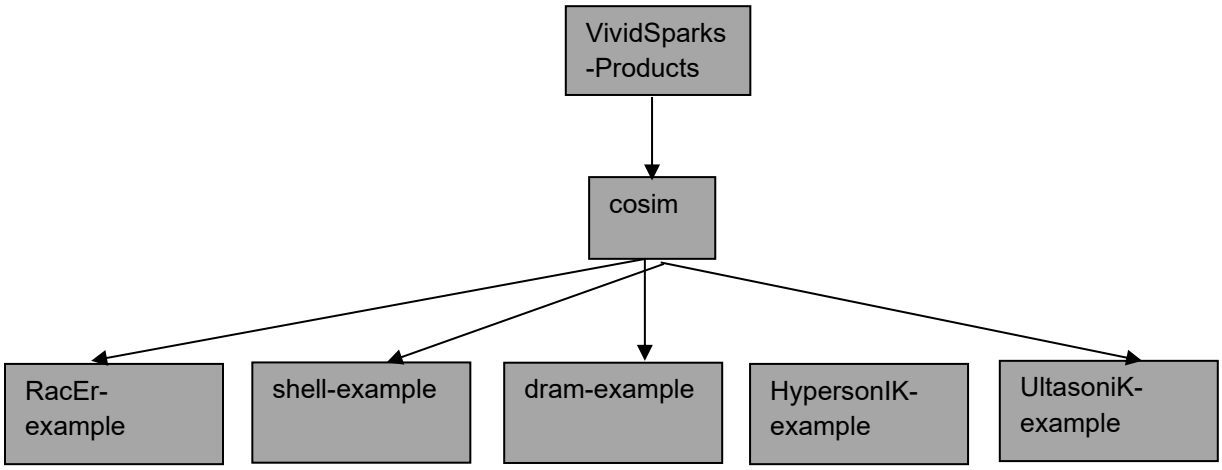
We demonstrate a methodology for prototyping accelerators and RISC-V cores like HypersoniK (HK), RacEr, UltrasoniK using Xilinx Zynq platforms with the following killer features:

- The entire system can be simulated in VCS before it is run on the FPGA.
- A standardized FPGA “shell” provides easy-to-use building blocks for communication between a C or Python program running on the host ARM core and the client accelerator or soft processor core. You can easily customize this interface.
- The ARM core (known as “PS: processor subsystem, in contrast to the PL: programmable logic), runs Linux and effectively replaces the $600 x86 server you would ordinarily have to buy to plug your FPGA board into. The total cost (for a VividSpark.tech board) is about $300-$800, making it more cost effective than Amazon F1 cloud FPGAs, and much easier to use and debug, and has much faster turn around time for builds (10 min versus 3 hrs.)
- The complex, error-prone, difficult to modify, and unsimulatable PCI-E connection traditionally used to connect to the FPGA board is replaced with the simple shell interface.
- We use the minimum amount of Xilinx IP, which tends to be buggy and not use best practices.
- We provide examples of how to use the Xilinx GUI (if necessary) to connect the Zynq core to your Xilinx “IPI block” and also how this can be scripted via Makefiles and generalizing the TCL outputted by the GUI when you prototype.
- The code base is extremely small compared to other infrastructures.
- It includes examples of how to integrate a Linux capable RISC-V processor.

This document also contains instructions on how to run HypersoniK, RacEr and UltrasoniK on the FPGA.

Design Examples

The figure below shows the VividSparks-Products directory structure. Generally speaking, hardware should be designed around optimizing debugging time, which means that rather than having one large design that integrates many components that each have potential problems, you should build up to that design. You should incrementally create specific designs that test each “risky” component in isolation. Having smaller designs means that you have a smaller needle-in-haystack problem and that the tool run times are minimized, allowing you to iterate and converge quickly.



The cosim directory contains the following design examples, each of which is worth looking at and understanding.

- **simple-example**: This example tests a simple logic which contains 4 read/write registers controlled by an AXI-lite client (slave) port, and shows how to use the GUI to connect to the Zynq ARM core. In many cases, it makes sense to skip this example.

- **shell-example**: This is the main example. It tests a single RacEr_zynq_pl_shell module. This is a good starting place to look at, handling all cases except DRAM interfacing. This module instantiates multiple basejump modules that implement a memory-mapped interface between the PS and PL. We have the following:
    - PS to PL registers: These registers can be read or written by the PS, but read-only by the PL
    - PL to PS registers: These registers can be read or written by the PL, but read-only by the PS
    - PS to PL FIFOs: These FIFOs are written by the PS and read by the PL.
    - PL to PS FIFOs: These FIFOs are written by the PL and read by the PS.

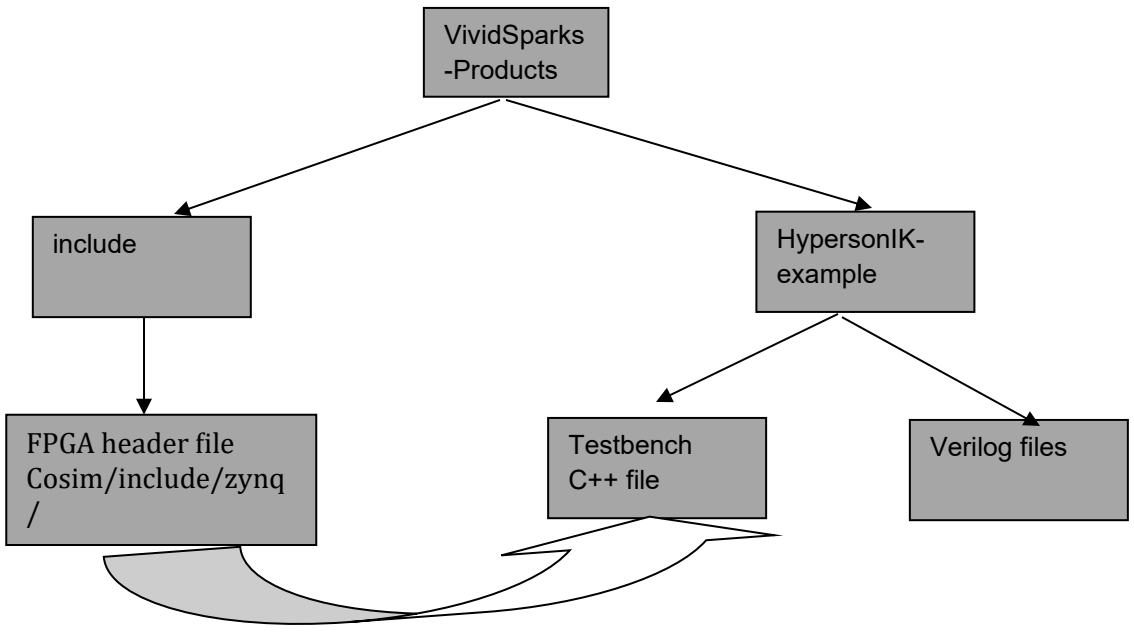
The module contains parameterization to control the number of each of the above elements as required. In the shell example, there are 4 PS to PL registers, 4 PS to PL FIFOs, 2 PL to PS FIFOs and 1 PL to PS register. In order to test all the elements, we loopback the PS to PL FIFOs to the PL to PS FIFOs, adding the outputs of a pair of PS to PL FIFOs to generate the value inserted into a pl to ps FIFO.

- **dram-example**: This example tests DRAM allocation on the board, by (on the ARM core in software) allocating DRAM, writing a stream of data and reading it back for comparison.

- **HypersonK-example**: This example demonstrating how to wire the HypersoniK Linux capable RISC-V core into Zynq. It includes having HK read/write to DRAM directly; and an AXI port that lets you write into HK's memory mapped control registers and memory space. This example is the full enchilada, and we focus on it in a dedicated section below. The other examples are a subset of the functionality.

- **UltrasoniK-example**: This is the most complex example, demonstrating how to wire RacEr and HypersoniK into Zynq. It includes having HK read/write to DRAM directly; and an AXI port that lets you write into HK's memory mapped control registers and memory space. This example is the full enchilada, and we focus on it in a dedicated section below. The other examples are a subset of the functionality. See more below!

Each example folder (except for dram-example) contains two other folders - zynq and and a testbench file *ps.cpp*.



The figure above shows the strategy used in cosimulation. We have two separate implementations of the HK_zynq_pl header files for the FPGA. These two files are located in *cosim/include*. When running the test in the corresponding folder, the correct header file is swapped into the build. These two files contain a class with largely the same methods, the common methods between the two files are as follows:

- *axil_write*: This method performs a write operation over an AXI port. In the simulation environment, the actual AXI4-Lite functionality is emulated in the function, whereas in the FPGA environment, it essentially becomes a write to a pointer that signifies the specific AXI port.
- *axil_read*: This method performs a read operation over an AXI port. In the simulation environment, the actual AXI4-Lite functionality is emulated in the function, whereas in the FPGA environment, it essentially becomes a read from a pointer that signifies the specific AXI port.
- *done*: This method signifies the end of testing.

There are a few additional methods and operations in the FPGA environment (eventually we will implement dummy variants of these):

- *allocate_dram*: The PS needs to allocate the required amount of DRAM for the PL to access. This method performs the allocation and returns the virtual and physical pointer to the first DRAM location.
- *free_dram*: This method frees the allocated DRAM and should be called at the end of testing.

Steps in Build and Running the Bitstream on VividSparks FPGA

You will be checking out two copies of the VividSparks_Products repo: one on an X86 server (for building your FPGA bit streams) and the other on the Zynq/VivdSparks board (for running the FPGA bit stream, and the code that interfaces to it on the Zynq ARM core.) It is helpful to browse through the Makefiles in both the verilator and FPGA folders in order to see what commands are available to run.

Go through the Makefiles in both the folders to run the commands for simulation and generating the bitstream. It is important to add the correct address bases and sizes for the two AXI ports in the Makefile before running.

FPGA build commands

The commands in the cosim/<example-name>/vivado directory's Makefile are generally executed in this order on the X86 server (you can just type `make` in the vivado directory to get a list of these commands):

- *fpga_build:* Builds the FPGA bitstream. Runs the following subrules, which in some bases may be useful to rerun individually:
    - *fpga_build_ip*: This just packages the HypersoniK IP without performing synthesis and the next steps.
    - *fpga_build_block*: Synthesize and generate bitstream for the entire system.
    - *fpga_report_build_block:* Print out errors in last build, and timing information.

  At each stage, it will interrupt, allowing you to type "start_gui" to fire up the gui and inspect the results. Just type "quit" to continue. The file "vivado-create-block.2022.1.tcl" contains the build instructions to the Vivado tool, including what frequency to target for the clock in the CONFIG.PCW_FPGA0_PERIPHERAL_FREQMHZ variable. There are other similarly named files in the directory, targeting different versions of Vivado.

- *pack_bitstream*: Pack the three files (.tcl, .bit, .hwh): tar, compress and perform BASE64 Encoding. You then copy this .b64 file over to corresponding /zynq directory in your checked out version of the VividSparks-Products on the PYNQ board. (An even more elegant way to do this is to exports the X86 filesystem over NFS (network file system) to the Pynq board, but that is not covered here.)

- *clean_fpga_build*: clean up Vivado build files

Then you will run these commands on the Zynq board in the FPGA directory::

- *unpack_bitstream*: Unpack the three files once you have copied the packed file onto the board. (Setup: check out the RacEr repo on the board, and enter the corresponding zynq directory.)

- *load_bitstream*: Load the bitstream onto the board using the PYNQ Overlay package.
- *run*: Load the bitstream and run the testbench with the FPGA include file.
- *print_clocks:* Print current clock frequency used by the PL
- *<freq>.set_clock*: Set the clock to <freq>
- *clean*: remove compilation of ps.cpp

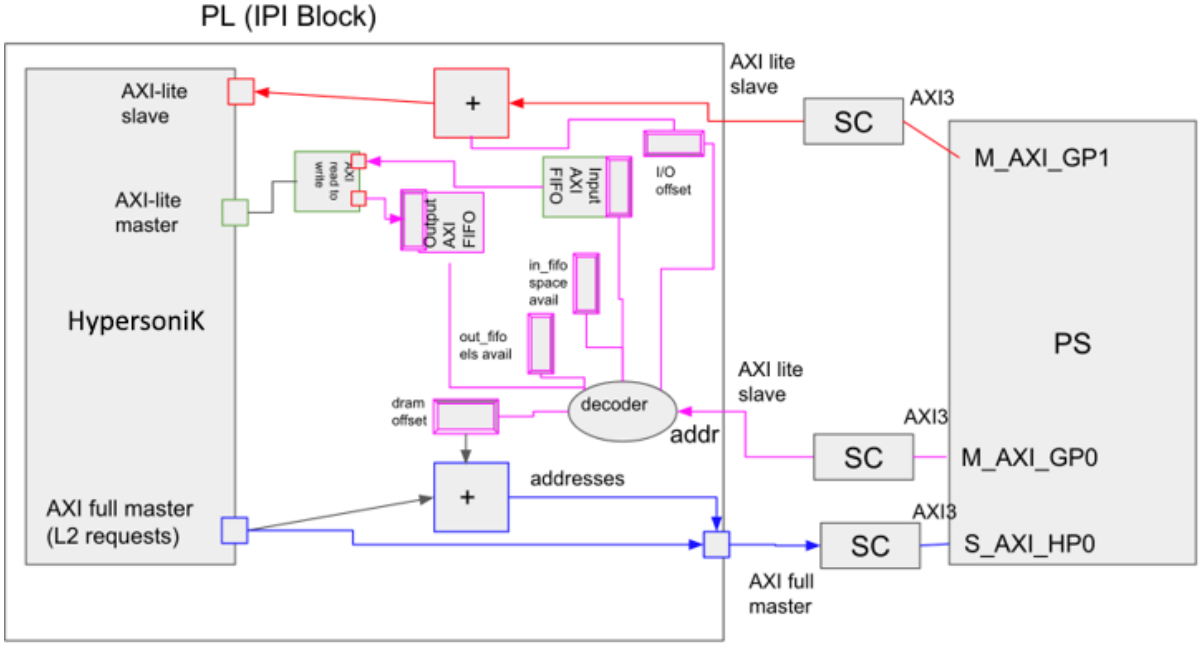
HypersoniK Design

This design contains a description of the HypersoniK microarchitecture. The design contains three components - Front End (FE), Back End (BE) and Memory End (ME). Communication between HypersoniK components is performed over a set of narrow interfaces. The interfaces are designed to allow the implementations of the Front End, Back End and Memory End to change independently of one another.

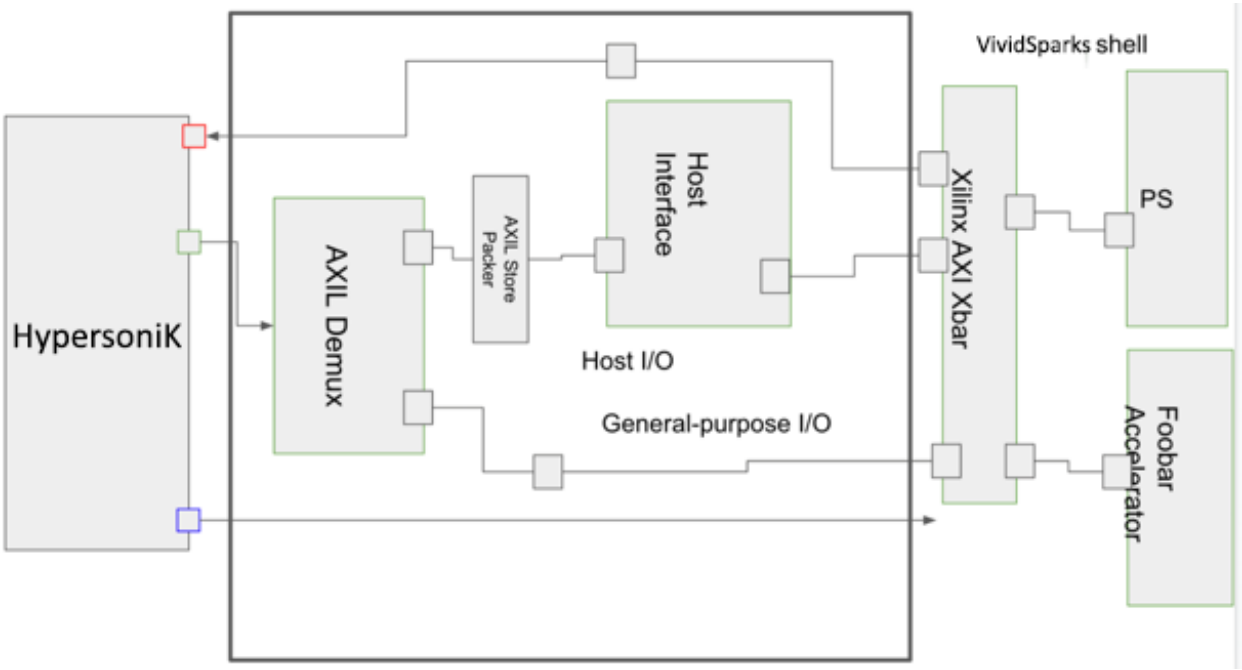
For the HypersoniK example, the NBF file currently used is for the hello_world program, this can be changed to run a different program instead, for example to do a sweep of the Spec2000 and Spec2006 benchmarks.

The HypersoniK Zynq Enablement

The HypersoniK Zynq Enablement along with the PS is shown in the below figure.



The Zynq 7020 ARM PS system uses ARM Physical Addresses 0x0000_0000-0x4000_0000 for DRAM, 0x4000_0000-0x8000_0000 for the AXI-lite master port 0, and 0x8000_0000-0xC000_0000 for the AXI-lite master port 1. The S_AXI_HP0 AXI-full client (aka slave) port provides direct access to the DRAM controlled by the ARM PS memory controller. Correspondingly the top-level HypersoniK interface module has two AXI-lite client ports and one AXI-full master port. The first AXI-lite port allows the ARM PS to access 512 MB of HK's address space, according to a hash function that maps



the address, via the HK input I/O interface. The ARM core stalls during this backdoor access to HK, so HK must service these requests without blocking. The second AXI-lite port controls the shell. The AXI-full port connects the backend of the HK L2 caches to the ARM DRAM controller, using a shell-programmed base register to translate from the standard RISC-V 0x8000_0000 base address to the physical address of a contiguous, uncached-by-PS dram space allocated by Linux on behalf of the user application, and then to the corresponding AXI address.

HypersoniK I/O Emulation

The remaining piece is how HK does I/O. In this case it communicates with the ARM PS to perform the I/O on HK's behalf. In this case, we need to have a way to connect two masters to each other (one AXI and one HK I/O). The approach we employ is to use the FIFOs and register interfaces provided by the shell. The FIFOs provide multiple

independent, reliable producer-consumer communication of sequences of data, and the register interfaces provide give-me-the-latest-value semantics. An alternative would have been to employ software-maintained shared FIFOs in the DRAM subsystem or in the Zynq 256KB on-die-memory. We hypothesize that the approach we have taken has less contention and lower HW/SW complexity than the alternatives, while using very few resources.

HypersoniK I/O takes the form of a stream of memory-mapped requests ie <read/write, address, data> tuples that are enqueued onto a shell fifo for the PS to read and process (and potentially to reply to.) So that the ARM core does not freeze and can still be used by the user to login via Ethernet and control the device, all of the ARM-side interfaces using polling interfaces to check if data is available, and never stall the processor if data is unavailable or if a fifo is full. The HK side is stalling, and HK will stall waiting for data or when writing to a full FIFO.

HypersoniK is integrated in the Zynq with an AXI wrapper. This provides a AXI4 master for DRAM accesses, an AXI-lite master for I/O and AXI-lite client for I/O. A module named *RacEr_axil_store_packer* convert raw AXI requests into a format used by the FIFO interface. This process is as follows:

- HypersoniK sends an I/O request (UART read/write, finish).
- The AXI-lite master sends the request to *RacEr_axil_store_packer* where it is converted into a 32-bit packet with a concatentation of $<$read/write (1-bit), address (23-bits), data (8-bits)$>$.
- *RacEr_axil_store_packer* enqueues this request onto the output FIFO and, if a read, waits for the response. If a write, it returns immediately
- The PS reads the packet, decodes it in software and performs the necessary operation
- If a read, the PS enqueues the data onto the response fifo
- *RacEr_axil_store_packer* reads the data from the response fifo and returns the data.

The PS currently supports requests to print, getchar or terminate the program.

The HK r/w to write module (*HK_to_axi_decoder* in the code) converts requests from the HypersoniK I/O output port into data that can then be enqueued onto the output FIFO. UART reads are currently unsupported, such a request will respond back to HypersoniK with zero data. Support will be added soon. The PS reads the request in the output FIFO, decodes it and performs the required operation (print to the screen or terminate the program).

The HypersoniK address space mapping and the required address translation is described in the file top_zynq.sv.

HypersoniK Control Program (ps.cpp)

Our Zynq cosim environment system is designed so that you can write a *control program* that runs on the PS and interacts with your accelerator or HypersoniK core running in PL. This program could be in C or interactive Python (with some additional porting of our C-based infrastructure to the Xilinx PYNQ environment.) Usually the control program will configure the soft accelerator / processor and then interact with it, performing IO.

In the specific case of HypersoniK, we compile binaries with the RACER Panic Room environment, which supports the standard C library with the newlib lightweight implementation hosted on top of a DRAM-backed file system that is compiled into the program binary, allowing the program to be easily bundled with its own data and do basic (non-persistent) file I/O without requiring proxying of system calls by the control program. This replaces the relatively complex fesvr environment often used to do this by other RISC-V cores. Of course HypersoniK can still write to it's I/O address space, which will allow "real" communication to the PS.

The default control program for HypersoniK (which you are free to modify for your purposes!) runs on PS and instantiates an object of the HK_zynq_pl class described above and performs two operations: set the configuration registers inside HypersoniK and writes the program to the DRAM address via the HypersoniK L2 cache. Once these operations are complete, the testbench waits for data to arrive at the shell PL-to-PS FIFO described earlier, and decodes the data to perform the correct operation. The two functions specific to the HypersoniK testbench are described below:

- *nbf_load*: This function sets the configuration registers and loads the program into the DRAM, by using a .nbf (*network boot file*) file available in HypersoniK. The nbf file contains a packet with the address and data in each line. This function reads the file line by line, extracts the address and data, and calls the axil_write method of the object with these values.

- *decode_HK_output*: This function receives 32-bit data and decodes the contents. As mentioned in the hardware section, a packet sent by HypersoniK is a concatenation of <R/W, 23-bit address, 8-bit data>. If the packet is a UART write (R/W = 1 and Address = 0x101000), the data is printed onto the screen. If the

packet indicates a finish (R/W=0 and Address = 0x102000), the program is terminated. UART reads are currently not supported in this function, but can be added by polling for user input and writing the result to the input FIFO.

The default control program starts with multiple diagnostic tests that test each interface connection in both simulation and actual hardware. The goal is to easily isolate problems to help external users localize their problems in hooking HypersoniK up correctly. The following tests are performed before HypersoniK is initiated:

- Writing and reading the CSRs in the shell (tests the ARM PS GP0 port connection).
- Reading the `mtime` register repeatedly (testing reads from the ARM PS GP1 port to the HypersoniK CSR address space). It should increment slightly with each read.
- Read-Modify-Write-Read from the `mtimecmp` register inside HypersoniK (tests the GP1 connection for local CSR addresses inside HK).
- Writing to enough HypersoniK DRAM addresses via the ARM PS GP1 port that maximally triggers evictions and fills from the HypersoniK L2 cache, resulting in DRAM access from HK to the ARM PS DRAM controller via the ARM PS HP0 port.
- Reading these addresses directly on the ARM core ARM PS virtual addresses, bypassing HK and going directly to the DRAM on the hardware.
- Reading those addresses on the ARM core via the HypersoniK core using the ARM PS GP1 port, connecting through the HK input IO connection, through the internal network to the L2, and potentially over the HP0 port.

How to add your own instrumentation to the design and collect data as HypersoniK runs.

The top_zynq.sv file contains an example of how to connect an internal signal in HypersoniK to the shell module, so that the PS can read that value. This would be useful if you add any other counters inside HypersoniK and want to monitor the value in the testbench.

In the old days, if you wanted to expose a signal inside HypersoniK, you would have to wire those signals from their source inside HypersoniK, all the way up to the top-level, where you could attach it to the *csr_data_i* connection in the shell. Painful! Fortunately, Vivado recently added a feature (also available in VCS and Verilator), that is not available in ASIC Synthesis tools: synthesis of variables that are accessed hierarchically!

Take a look at the *csr_data_i* connection inside the `HypersoniK-example top_zynq.sv` shell instantiation inside the module:

```
,.csr_data_i(HypersoniK.unicore.unicore_lite.core_minimal.be.calculator.pipe_sys.csr.minstret_lo[31:0])
```

You can see, we are reaching deep into HypersoniK and pulling out the RISC-V instructions retired csr!

Of course, this is a ready made counter. What if you wanted to add your own counter? In that case, you could instantiate the counter inside the `top_zynq` register, and then pull the control signals from inside HypersoniK in order to read them. These versions use the SystemVerilog bind statement, which is not synthesizable and thus cannot work on FPGA. It would be not too much effort to leverage the knowledge from these modules to turn them into FPGA counters.

The Software Directory

In the RacEr/HypersoniK/UltrasoniK/software directory, we have a bunch of make commands to help you on your way; read the makefiles for details, this is just a summary:

The following make commands can be run on X86:

- `build_tools`: builds GCC for RISC-V, using the Panic Room environment. You can use this to build the beebs programs, or to build your own programs, to create small focused tests for programs.
- `build_beebs`: builds the beebs benchmark suite, generates a bunch of `.riscv` binaries. The beebs benchmarks are in cosim/import/sdk/beebs/src.
- `build_spec`: builds spec benchmarks. Because this uses the for-money spec benchmark suite, which we cannot distribute, we have tar'd the spec benchmarks and beebs and put them in /class/6.548/nbf.tar.xz; you can copy this into your software directory on both the zynq board and the class server.

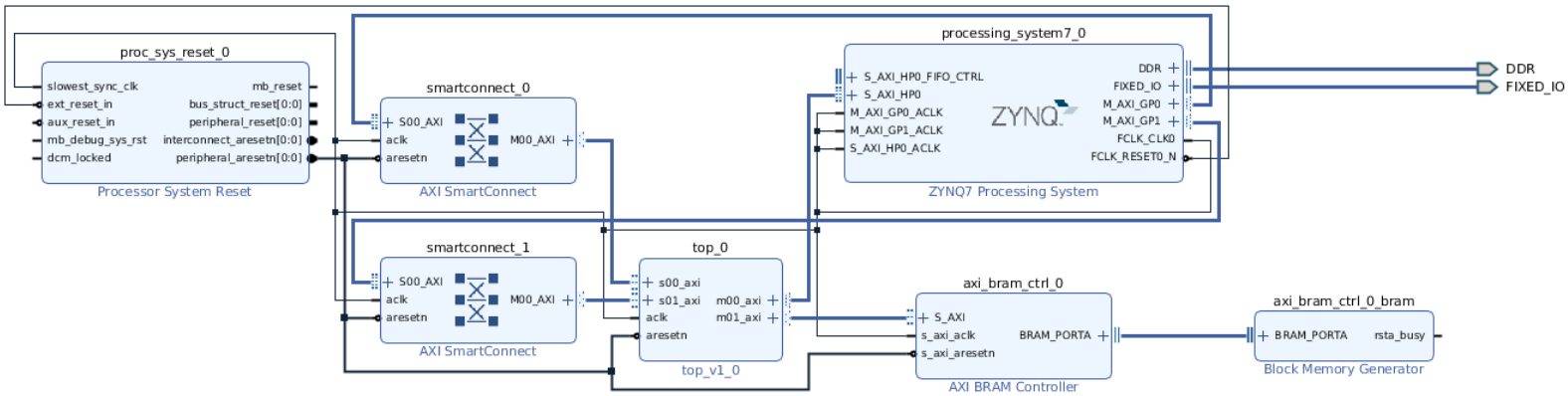
- `generate_and_packup_nbf:` converts the `.riscv` files into our .nbf format, creates tar file.
- `run_all_verilator_nbf:` runs all .nbf files in Verilator
- `nbf/<testname>.verilator.log`: runs <testname> on verilator (see nbf/*.nbf)

The following make commands can be run on Zynq:

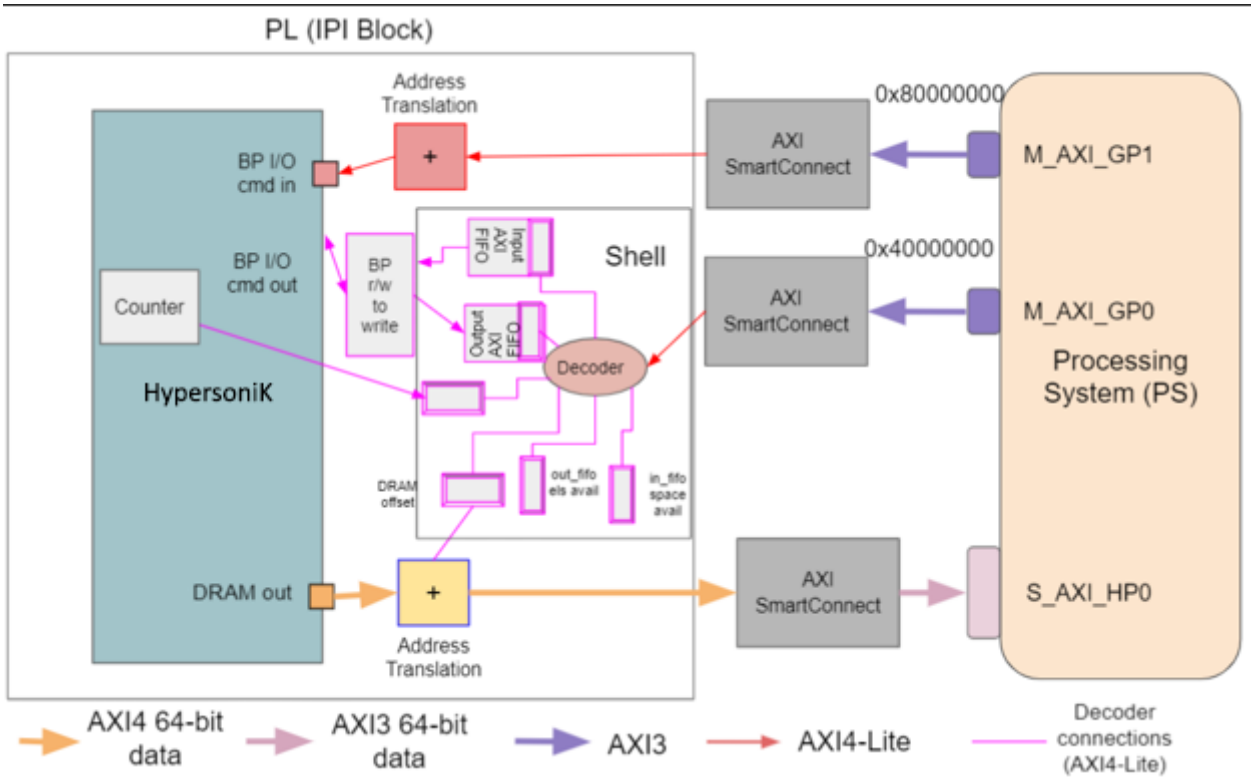
- `unpack_nbf`: run this after you copy the nbf tar file to the software directory of a checked out RacEr repo (the multiple_axi_ports branch) in the Zynq board. We suggest you deleted the bootrom.nbf file in the nbf/ directory, it appears to hang the board.
- `load_bitstream:` run this once after you have turned on board, to load bitstream. Make sure to place your bitstream in `cosim/HypersoniK-example/zynq` and unpack with make `unpack_bitstream`.
- `reload_bitstream_keep_dram`: run this to load a new bitstream, if you have already loaded the bitstream before.
- `run_all_fpga_nbf:` runs all .nbf files in zynq (see nbf/*.log for log files)
- `nbf/<testname>.fpga.rerun`: runs <testname> on fpga (see nbf/*.nbf for programs to run) (see nbf/*.log for log files)
- `summarize:` See which beebs benchmarks don't pass, or fail; greps nbf/*.log
- `clean_logs:` deletes log files

VividSpark.tech Block Design View inside Vivado:

This shows the connectivity of blocks inside the Zynq SoC, which is scripted in our infrastructure. Small changes may need to be made to this script if you need to go to a new version of Vivado. We include this picture in case you need to port to a new version of Vivado and want to know what it's supposed to look like. Note the BRAM is not actively used, and is just there to demonstrate a client device attached to HK via AXI.



VividSparks FPGA Block Design View inside Vivado:



This process is as follows:

- HypersoniK sends an I/O request (UART read/write, finish).

- The HK_to_axi_decoder module converts this request into a 32-bit packet with a concatenation of $<$read/write (1-bit), address (23-bits), data (8-bits)$>$.
- The 32-bit packet is enqueued onto the output FIFO.
- The PS reads the packet, decodes it in software, and performs the necessary operation.
- UART read requests are currently unsupported, support will be added soon.

VividSparks.tech Setup (PYNQ 3.0)

```
# update package list
sudo apt update
sudo apt-get install device-tree-compiler
sudo apt-get install emacs-nox
```

Changing VividSparks.tech memory config via device tree

By default the AXI ports on the ARM core are configured to be strongly ordered, as opposed to device ordered or cached. This means that the ARM core does one memory access at a time, waiting a full 24 cycles or so roundtrip time at the PL clock frequency. This is very slow. This can be accelerated by using NEON instructions, and increasing the PL clock frequency. However, this is just working around the fundamental problem: memory accesses should be device ordered, not strongly ordered, if the Linux kernel doesn't know about the memory range (i.e. it is not in the device tree), it uses strongly ordered for mmap'ing the region via /dev/mem. If it knows about the memory range, then it uses cached if /dev/mem is opened without O_SYNC and uncached if it is opened without O_SYNC. Unfortunately, neither of these settings correspond to device ordered. For device ordered, it seems that a modification to the Linux kernel is required,. It appears this is an artifact of Linux wanting to double map the address space into both kernel and user, and of kernel space defaulting to cached.

Unfortunately, there aren't many short-cuts, except to use the full Petalinux build to build an SD Card. To change the memory configuration, you need to change the device tree. There are actually two device trees, one appended to the end of the Linux kernel

(creating the file image.ub), and one in the boot loader (on the boot volume). The boot volume's device tree's memory settings override the one appended to the Linux kernel. So both BOOT.bin and the kernel package need to be updated. There are tricks for extracting and updating the device tree in image.ub, but that is not enough to change the memory setting for the kernel.

A. On X86 server, with PetaLinux build system installed, inside PYNQ repo:

Modify PYNQ/boards/Pynq-Z2/petalinux_bsp/meta-user/recipes-bsp/device-tree/files/system-user.dtsi:

```
/include/ "system-conf.dtsi"
/ {
        amba {

                ethernet@e000b000 {
                        phy-handle = <&ethernet_phy>;
                        ethernet_phy: ethernet-phy@1{
                                reg = <1>;
                        };
                };
        };
      usb_phy0: phy0 {
          compatible = "ulpi-phy";
          #phy-cells = <0>;
          reg = <0xe0002000 0x1000>;
          view-port = <0x170>;
          drv-vbus;
      };
        audio0:audio-codec-ctrl@43c00000 {
                compatible = "xlnk,audio-codec-ctrl-1.0";
                reg = <0x43c00000 0x10000>;
        };

        memory {
                reg = <0x0 0x20000000 0x40000000 0x1000>;
        };

        reserved-memory {
                #address-cells = <0x1>;
                #size-cells = <0x1>;
                ranges;
                reserved {
```

```
                        reg = <0x40000000 0x1000>;
                };
        };
};

&i2c0 {
        status = "okay";
};

&i2c1 {
        status = "okay";
};

&audio0 {
        compatible = "generic-uio";
};

&usb0 {
      usb-phy = <&usb_phy0>;
};
```

B. **Source /tools/Xilinx/Vitis/2022.1/settings64.sh and Petalinux/settings.sh[1]**
C. **Edit Makefile in PYNQ/sdbuild to set BOARDS to Pynq-Z2**
D. **In PYNQ/sdbuild: make clean; make**
E. **find the .img file and burn it to the SD card**
    a. tail -100 /var/log/syslog to see SDCARD device; e.g. /dev/sdb
    b. Note, you want the drive, not the partition (e.g. not /dev/sdb1) since you need to overwrite the boot. Watch out for Ubuntu automounting the card; you would want to umount /dev/sdb1 and /dev/sdb2
    c. sudo dd bs=1M if=output/Pynq-Z2-3.0.1.img of=<SDCARD device>; sync
F. Follow the VividSparks.tech Setup to install various applications
G. Upon booting, check the device tree:
    a.
    b. `dtc -I fs /proc/device-tree`

Building Petalinux with a kernel patch

See this link: https://support.xilinx.com/s/question/0D52E00006krJDSSA2/new-flow-for-patching-kernel?language=en_US And XILINX ug1144 pg 257

[1] Awkwardly unpacked at the top level of petalinux

pushd build/Pynq-Z2/petalinux_project/
petalinux-devtool is found here: tools/common/petalinux/bin/petalinux-devtool (should be in path after you source settings.sh)

```
petalinux-devtool modify linux-xlnx
```

cd components/yocto/workspace/sources/linux-xlnx
emacs arch/arm/mm/mmu.c

```
pgprot_t phys_mem_access_prot(struct file *file, unsigned long pfn,
                              unsigned long size, pgprot_t vma_prot)
{
        if (!pfn_valid(pfn))
          // for an address starting in 0x4???_????
          if (((pfn << PAGE_SIZE) >> 28) == 0x4)
            return pgprot_device(vma_prot);
          else
            return pgprot_noncached(vma_prot);
        else if (file->f_flags & O_SYNC)
                return pgprot_writecombine(vma_prot);
        return vma_prot;
}
```

```
e makefile
# modify extraversion to add tag to kernel name so you can make sure changes
# take
petalinux-devtool build linux-xlnx
# in PYNQ/sdbuild:
make
# goto step E above
```

See https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842412/Accessing+BRAM+In+Linux

inside ~xilinx:
git clone https://github.com/Xilinx/Pynq.git

OUT OF DATE

see https://gist.github.com/yunqu/827862e580a5f9b069eccdfcdcf70398

```
# extract device tree
dtc -I dtb -O dts system.dtb
# extract kernel
dumpimage -T flat_dt -p 0 -i image.ub zImage
```

```
# grab its file
cp ~xilinx/Pynq/sdbuild/boot/image_arm.its .
# remake image with device tree
mkimage -f image_arm.its image.ub
<reboot>
# print out device tree to verify
dtc -I fs /proc/device-tree
```

Unfortunately, this is not quite enough. This flow can specify memory reservations and other parameters, but not the memory section. Linux ignores the memory section in its device tree and instead uses the corresponding section in the separate device tree from u-boot. Therefore, it is necessary to also build u-boot.

- Download Petalinux, making the version of Xilinx tools installed
- Clone PYNQ (git@github.com:Xilinx/PYNQ.git)
- Edit patch file with memory section above:
  PYNQ/boards/Pynq-Z2/petalinux_bsp/meta-user/recipes-bsp/u-boot/files/0004-add-pynqz2-support.patch

- Follow these instructions:
  - https://pynq.readthedocs.io/en/latest/pynq_sd_card.html
  - “Using an existing Ubuntu OS; Building the Image From Source”

~/peta/PYNQ/sdbuild/build/Pynq-Z2/petalinux_project/images/linux/
sdbuild/Makefile

Petalinux-package

cd /home/mbt/peta/PYNQ/sdbuild/output/boot/Pynq-Z2 && petalinux-package --boot --fpga /home/mbt/peta/PYNQ/sdbuild/../boards/Pynq-Z2/base/base.bit --u-boot -p /home/mbt/peta/PYNQ/sdbuild/build/Pynq-Z2/petalinux_project --force
WARNING: Fpga Manager enabled, skipping bitstream to pack into BOOT.BIN
[INFO] Sourcing buildtools
INFO: Getting system flash information...
INFO: File in BOOT BIN: "/home/mbt/peta/PYNQ/sdbuild/build/Pynq-Z2/petalinux_project/images/linux/zynq_fsbl.elf"
INFO: File in BOOT BIN: "/home/mbt/peta/PYNQ/sdbuild/build/Pynq-Z2/petalinux_project/images/linux/u-boot.elf"
INFO: File in BOOT BIN: "/home/mbt/peta/PYNQ/sdbuild/build/Pynq-Z2/petalinux_project/images/linux/system.dtb"
INFO: Generating zynq binary package BOOT.BIN...

/home/mbt/peta/PYNQ/sdbuild/build/Pynq-Z2/petalinux_project/images/linux

~/peta/PYNQ/sdbuild/build/Pynq-Z2/petalinux_project$ more ./components/plnx_workspace/device-tree/device-tree/system-top.dts

sudo dd bs=4M if=/home/mbt/peta/PYNQ/sdbuild/output/Pynq-Z2-3.0.1.img of=/dev/sdb

These links talk about setting baremetal mode device memory:

https://support.xilinx.com/s/question/0D52E00006hpLnlSAE/axi4-lite-memory-mapped-performance-poor?language=en_US

https://support.xilinx.com/s/question/0D52E00006hpXGjSAM/interrupt-inconsistency-after-changing-settlbattributes-of-axilite-slave-address?language=en_US

Here is a link to Baremetal mode configuration of device memory:

https://adaptivesupport.amd.com/s/question/0D52E000079LMmiSAG/how-to-set-mmu-given-a-certain-axi-peripheral-slave-interface?language=en_US

Here is a call that gets 1 burst per 7 cycles: Call of Xil_SetTlbAttributes(XPAR_AXI_FIFO_0_BASEADDR, DEVICE_MEMORY)

- 70 ns (7 cycles) per transaction (57 MB/s) – baseline throughput

Can further optimize with SIMD writes and removing AXI3 to AXI4 converter.