Review on the Vortex project

Alessio Serra, Marco Simoni

May 2022

Abstract

In this work we provide a review of the Vortex GPU Project, starting from the paper *"Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics Research"* in which was introduced, and an analysis of the GPU's architecture and its specification. Moreover, we also provide an overview of the insights of the Vortex implementation, which is open-source and available in the official Github repository.

1 Introduction

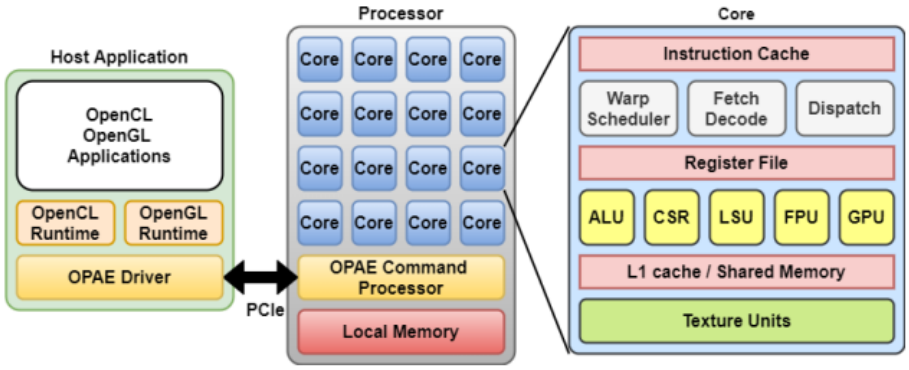

Figure 1: Vortex Framework overview.

General purpose GPUs (GPGPU) allows to use GPUs, which was dedicated to graphics rendering (Graphics Processing Units), for tasks that were formerly the domain of high-power CPUs with a boost of performance thanks to a **massive parallelism**.
They allow to address the power limitations and scalability of multi-core processor and have democratized High Performance Computing (HPC), allowing a much higher FLOP/$ rate.
There are several applications that has to process huge quantity of data and are predisposed to data-parallelism, like machine learning, graph analytics, cybersecurity and physics calculations.

However, there are many challenge to face for the implementation of a GPGPU and there is lack of open-source infrastructure.

The first aim of the Vortex project is to design an open-source software and hardware implementation of GPGPU on a FPGA (Field Programmable Gate Array) using the RISC-V ISA, i.e. a free, open and extensible ISA (Instruction Set Architecture) that allows a solid base for wide-range adoption of Vortex.
It supports both the OpenGL API for graphics programming and the OpenCL framework for heterogeneous computing.

The RISC-V ISA need to be extended to address the **execution model** used by GPGPU, that is the **SIMT** (Single Instruction, Multiple Thread), where single instruction, multiple data (SIMD) is combined with multithreading.
To do this efficiently they followed the RISC (Reduced Instruction Set Computer) principle trying to minimize the number of instructions to add for two main reasons:

1. To utilize as much of the existing open-source hardware and software ecosystem.
2. To provide a sustainable development ecosystem.

Their study led to the identification of six new instructions, explained in detail in section 2.1, that allows the Vortex processor to execute GPGPU applications and also accelerate the 3D graphics pipeline.

In addition to the standard SIMT microarchitecture components, Vortex:

- Implements a detailed *high-bandwidth non-blocking cache* subsystem optimized for FPGAs.
- Implements a robust compiler, driver, and application stack *supporting OpenCL*.
- Implementing *texture sampling units* to support 3D-graphics rendering.
- Integrates a *PCIe-based* command processor for communicating with a host processor like conventional GPGPUs.

2 Vortex Building blocks

Each component of the Vortex, both hardware and software, have been adapted to the **SIMT execution model**, which is used in GPU programming to **limit the instruction fetching overhead**, i.e. the latency that comes with memory access.
In fact, SIMT allows *multiple instruction streams to execute the same scalar instruction* through the creation of **warp** (a.k.a. wavefront), i.e. a logical cluster of Threads that execute the same program counter.

So Thread is the smallest unit of computation that execute in parallel and has its own register file (with 32 int and 32 fp registers).
However, when there are conditional **control flow instructions** the problem of **branch divergence** arises, in fact threads can execute different paths depending on the Boolean value of the condition.

2.1 Vortex ISA

The Vortex ISA extends the RISC-V ISA introducing instructions to handle wavefronts and their synchronization and to deal with the branch divergence problem, allowing **dynamic warp formation** that groups thread into warp **flexibly** to ensure that they **truly** execute the same instructions. Moreover was also added an instruction to support graphics processing.

The extension contains the following six instructions:

- **wspan:** activate a number of wavefronts at a specific program's PC value, enabling multiple instances of that program to execute independently. This is used for the **wavefront control.**
- **tmc:** activate or deactivate threads within a wavefront via a thread mask register. This is used for the **Thread control.**
- **split and join:** the former pushes information about the current state of the thread mask and the branch predication result for all threads into a hardware-immediate post dominator (IPDOM) stack, while the latter pops this out during reconvergence. These are used for **branch divergence**.
- **bar:** synchronizes wavefronts execution at barrier locations updating the information of the *barrier control module*, in particular a barrier is released when an expected number of wavefronts reach it. This is used for **synchronization.**
- **tex:** it is used for texture lookup depending on the parameters passed that identifies the coordinates of the source texel and the texture mipmap. This is used for **Texture sampling**.

All the previous instructions are of the RISC-V R-Type and fit in one opcode.

2.2 Vortex Microarchitecture

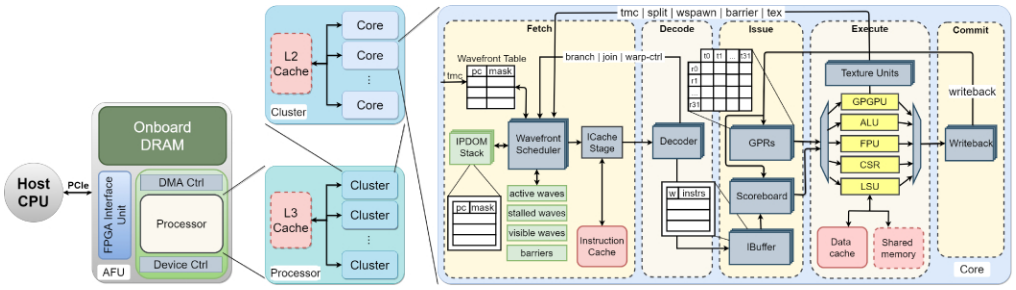

Figure 2: Overview of the Vortex Microarchitecture.

The Vortex microarchitecture includes:

- **A command processor (AFU),** which manages the onboard memory system and the communication with the host processor via PCIe.
- **A scalable processor** that can contain multiple clusters that share (optional) a L3 cache, each one contains many cores that share (optional) a L2 cache. The default configuration contains 1 cluster with 4 cores, each with 4 warps that groups 4 threads.

The core's architecture augments the standard five-stage in-order RISC-V pipeline (Fetch, Decode, Issue, Execute and Commit) with SIMT hardware component. In particular it includes:

- **Wavefront scheduler:** it decides, in the fetch stage, which wavefront to fetch from the *I-cache* depending on the wavefront information stored in the *wavefront table* and on a set of wavefront masks. The latter allows to determine each wavefront's state: active, stalled, barrier waiting and visible.
- **IPDOM stack:** it handles *branch divergence* by pushing wavefronts information after a *split* and popping them when there is a *join* instruction. In particular, it stores the thread mask before the divergence, in order to reset its value after the join, and another thread mask to identify threads with false predicate (*"the non-taken one"*) along with the program counter to resume the execution. So its function is to identify each potentially divergent branch and compute the nearest basic block on which all control paths from the divergent branch must ultimately converge.
- **Barrier control module:** which stores the counter of wavefront left to release the barrier and a mask to identify the stalled wavefronts. It is updated each time the *"bar"* instruction is executed.
- **Banked GPRs:** contains the general-purpose registers for each thread in each wavefront.

- **High-bandwidth caches:** with parallel access by the threads in the active wavefront.

2.3 High-Bandwidth Caches

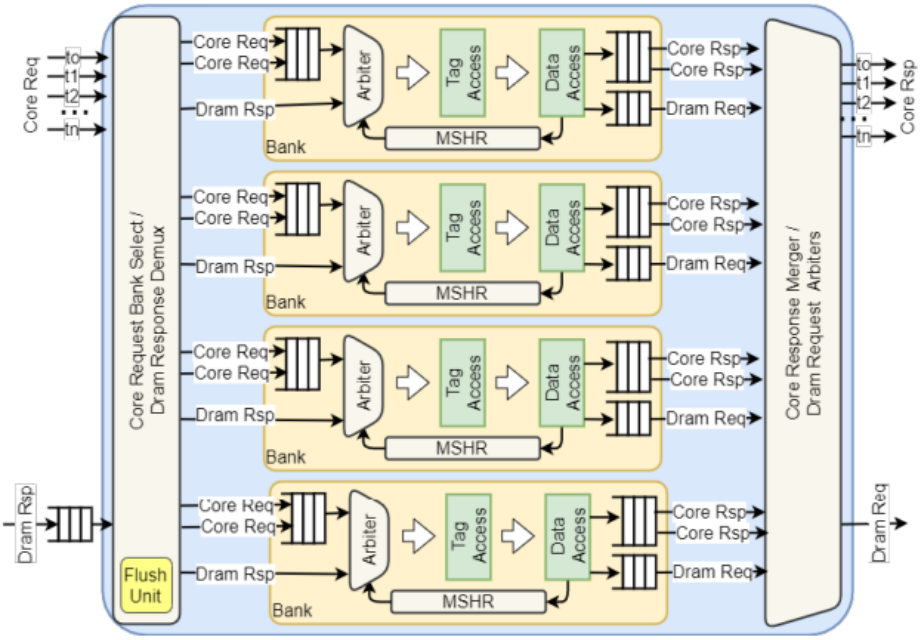

Figure 3: Overview of the High Bandwidth Cache.

One of the main elements of GPGPU architecture is represented by the **high bandwidth data cache**, which implements a Non-Blocking-High-Bandwidth approach, allowing the cache subsystem to *process multiple independent requests concurrently.*
In particular, the solution adopted by Vortex is based on the extension of **multi-banking** and the exploitation of **multiporting**. It is worth to specify that only data cache exploits multiporting since instruction cache executes one SIMT instruction per clock cycle.
As we can notice in the figure above, the cache is organized in different banks preceded by the *"bank selector"*; the latter is designed for assigning core requests to each bank according to their address. Once a request has been assigned to a specific bank, a proper schedule decides which input has to be processed among the following options:

- Core Request.
- Dram Response.

- Miss Status Holding Register (higher priority).

Just before the back-end which is able to coalesce the incoming core responses, we can spot the Data Access Partition. The coalescing operation is based on the tag value associated to the processed input element.

2.4 3D graphics acceleration

In 3D rendering we want to turn a 3D model to a 2D image and it has many applications fields, such as architecture, product design, advertising, video games and visual effects.
The main pipeline of a 3D rendering process includes first a **geometry stage** in which incoming vertices from the application are transformed to the rendering primitive, which is usually the triangle. Then there is a **rasterization stage** that receives the vector information, that can be basic point, line or triangle primitives, and convert it into a raster image composed of coloured pixel. Finally there is a **texturing stage** in which the pixel color is combined with the texture data, also called *texels*. The most compute and intense stage is the rasterization, especially the **texture sampling**.
To find a good trade-off between complexity, area occupied on the FPGA and speed-up, only the texture sampling is **accelerated**, while all the other stages are implemented in software. This was done because it is the most computational expensive operation and are usually a **performance bottleneck** in the software rendering pipeline.
In this way are followed the *OpenGL-ES specification* with the geometry processing running on the host processor and the rasterization pipeline running as a kernel on the Vortex parallel architecture.

2.4.1 Texture unit

The hardware implements configurable texture units to support **point or bilinear sampling** on 1D or 2D texture, that supports the OpenGL API.

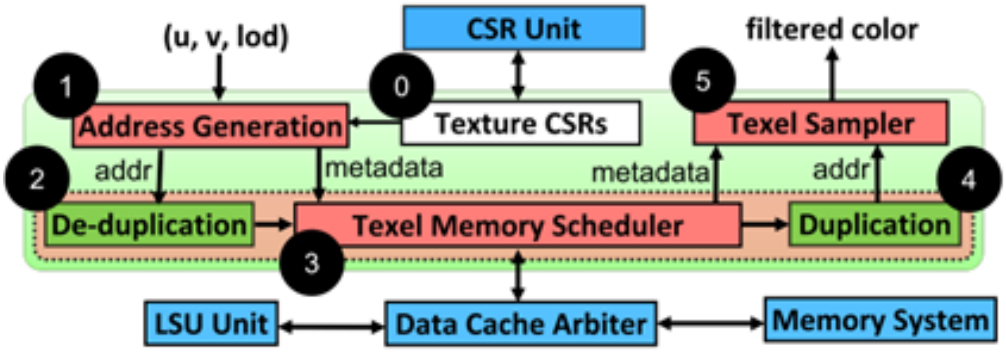

Figure 4: Texture unit microarchitecture.

In figure 6 is shown the microarchitecture of the texture unit which is configurable via the CSR and implements three main stages:

- Texture address generator.
- Texture memory system (composed by De-duplication, Texel memory scheduler and Duplication).
- The texture sampler.

As can be seen from the figure, the texture memory system communicates with the data cache arbiter to obtain from the cache the texel corresponding to the address received in input.

3 Vortex evaluation

The experiments were carried out using 3.5 GHz Intel Xeon E5-1650 for the host processor and tested the Vortex performance on a texture benchmark, composed of three synthetic benchmarks, for the 3-D graphics, and on a subset of the benchmark for heterogeneous computing for the GPGPU, composed both of memory bound and compute bound tests.

Core configuration consideration. The data parallelism can be increased by incrementing the number of threads per wavefront or the number of wavefronts of each core. This affects in different way the *area costs*:

- **Incrementing number of threads per wavefront:** increase the size of each GPR table, IPDOM stack and the logic in the cache and in the shared memory.
- **Incrementing number of wavefronts:** increase the number of GPR table, IPDOM stack, entry in the wavefront table and the logic in the wavefront scheduler.

Five different trade-off between number of threads and wavefront were tested, namely 4W-4T, 2W-8T, 8W-2T, 4W-8T and 8W-4T.
Considering the IPC (Instruction Per Cycle) as metric, the best configuration selected was the 4W-4T, which also allows to scale to 16/32 cores on the target FPGAs.
However, thanks to the simulation tools, were tested also more expensive configurations, which cannot fit on FPGA, in particular was tested a 6-core, 16-wavefront, 16-thread processor configuration.

Number of cores consideration. Were tested six different number of cores, i.e. 1, 2, 4, 5, 8, 16 and 32, to evaluate the differences in performance on the benchmark. All were tested on Intel Arria 10 FPGA, except for the 32 core configuration, which fits on Intel Stratix 10 FPGA.
Considering again the IPC as measure, the performance almost linearly increased with the number of cores for all the benchmarks, except one.

High Bandwidth Cache consideration. Considering a 4W-4T processor configuration, were tested several configuration of the Data cache, which as described in section 2.3. implements a virtual-multi-porting. It was considered the performance of a single core varying the number of virtual port from 1 to 4. Increasing the number of virtual ports led to:

- Increase in the logic area occupied.
- Increase of performances (were considered IPC and bank utilization, i.e. % of requests without bank conflicts).

The best trade-off found was the 2-port configuration.

3D-graphics evaluation. As mentioned before, were used three benchmark to test point sampling, bilinear sampling, and trilinear sampling. The Vortex accelerator performances (HW) were compared with a texture unit fully implemented in software (SW), i.e. without acceleration, obtaining different results on the three operations, due to their nature:

- **Point sampling:** the performances between HW and SW are almost the same, this is mainly due to the fact that point sampling shares the texel sampler implements only bilinear sampling (point is executed with blend value equal to 0).
- **Bilinear sampling:** there is a high speedup (x2) between HW and SW.
- **Trilinear sampling:** there are performance gains, but a lower speedup respect to bilinear sampling due to memory bandwidth since trilinear doubles the number of requeststo the memory.

The memory bandwidth problems also reduces the speedup between HW and SW *increasing the number of cores*, considering 1, 2, 4 and 8 cores.

4 Vortex Github REPO

The Directory structure is organised in the following way:

- **doc:** documentation.
- **hw:** hardware sources. This module is composed of three main different directories:
  1. The first concerns the DPI configuration settings.
  2. The second consists in the main RTL module, which will be explained in section 6.
  3. The third consists in the SYN module, which contains scripts for configuration and simulation execution.
- **driver:** host drivers repository. More details about this module will be presented in section 4.
- **runtime**: kernel Runtime software. The module is responsible to start the execution of kernel and manage system calls. Moreover, also the configuration of the dynamic linker is placed into this module.
- **sim:** simulators repository with a directory for each simulation environment, namely RTLSIM, SIMX and VLSIM.
- **tests:** tests repository with both memory and compute bounded kernels, like matrix and vector manipulation, which are placed in the OpenCL subdirectory.
- **ci:** continuous integration scripts.
- **miscs:** miscellaneous resources.

Furthermore, we can identify 4 different toolchain dependencies:

- **POCL**: a major goal of this project is improving interoperability of diversity of OpenCL-capable devices by integrating them to a single centrally orchestrated platform.
- **LLVM:** the major goal is to provide a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages.
- **RISCV-GNU-TOOLCHAIN**.
- **VERILATOR:** the major system/verilog simulator.

5 Driver Implementation

The Vortex driver handles the kernel interface to access the FPGA via the PCIe bus, an overview is shown in Figure 7.
The FPGA resources are seen as a set of features accessible by the host thanks to **OPAE** (Open Programmable Acceleration Engine), which is a lightweight user-space open-source C library.
For data transfer it is used the **CCI-P** (Core Cache Interface) protocol to assign a **shared memory space** which is accessible both by the host and the Accelerator Functional Unit (AFU):

- The host place data in the shared space and then the AFU writes them to the FPGA local memory.
- The FPGA is reset to start the execution, then the produced result is stored in the FPGA local memory.
- Finally the result is moved from the FPGA local memory to the shared space using MMIO (Memory Mapping I/O).

Memory mapped I/O is used for controlling the FPGA reading from/writing to registers or memory blocks of the hardware mapped to the system memory. The MMIO APIs allows for low level control over the FPGA.

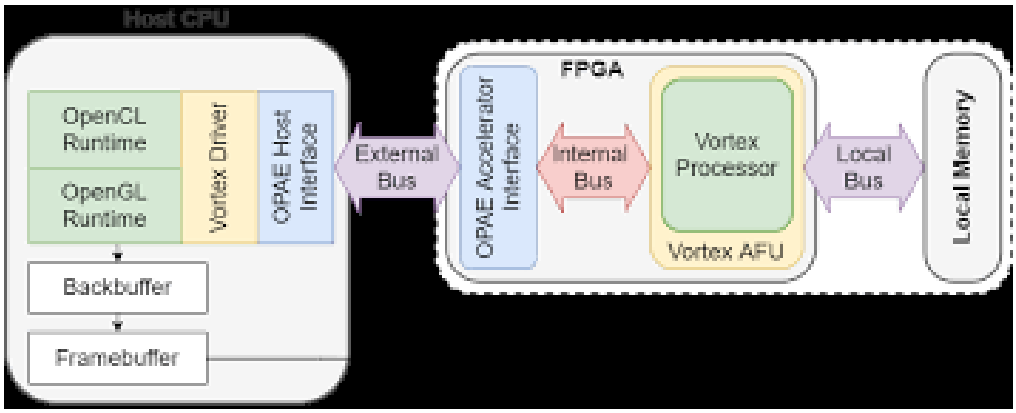

Figure 5: Overview of the Vortex Driver Stack.

5.1 OPAE

In the file *"vortex/driver/common/opae.cpp"* are defined several functions to access FPGA resources as a set of features accessible for software running on the host.
In particular it defines:

- **Vx_device class:** it is the most important class that represents the vortex device, which is characterized by the number of cores, of threads and of

warps and also by a memory allocator object to handle the memory that will be used.

- **AutoPerfDump class:** it is used only if DUMP_PERF_STATS is defined and it holds a list of vortex device which will be dumped, calling the *vx_dump_perf* function defined in *"vortex/driver/common/utils.cpp"* when is called its distructor.

- **vx_dev_caps():** returns the information on the device, which are identified by a certain ID. Possible device information are: version, number of cores, number of warps, number of threads, cache block size, local memory size, base address for allocation and startup address for the Kernel.

- **vx_dev_open():** it opens the vortex device or the simulated ones, in case USE_VLSIM is defined, calling the fpgaOpen() OPAE function to acquire its ownership. It also starts the scope, if SCOPE is defined, add the device to the dumped list holded by the AutoPerfDump class, if DUMP_PERF_STATS is defined, and initialize the structure that holds the device information by reading its feature from the shared memory space.

- **vx_dev_close():** it receives the device to close and before doing it, it possibly stops the scope and makes a dump if SCOPE and DUMP_PERF_STATS are defined, respectively.

- **vx_mem_alloc():** allocates an amount of memory in the vortex device at a specific address and of a certain size, depending on the input passed to the function, using the MemoryAllocator object.

- **vx_mem_free():** deallocates the memory in the vortex device which starts at the address given as input to the function, using the MemoryAllocator object.

- **vx_buf_alloc():** allocate a device buffer in the shared memory of a given size thanks to fpgaPrepareBuffer() OPAE function and passes its information initializing the buffer structure that is accessible with the pointer given as input to the function. This buffer will be used to place information that has to be passed from the vortex device to the host and vice versa.

- **vx_buf_free():** release the memory occupied by the device buffer in the shared memory thanks to the fpgaReleaseBuffer() OPAE function, which is given as input to the function.

- **vx_ready_wait():** wait for the conclusion of the command. It reads from the shared memory the status of the vortex device in a while loop until a maximum timeout. It is called always before giving a command to the vortex device to ensure that it is ready for a new command.

- **vx_copy_to_dev():** copy in the vortex local memory the content of the host memory starting from a certain offset and for a certain size. Both device memory address, host buffer, size and offset are given as input.
- **vx_copy_from_dev():** copy in a buffer in the host memory the content of the Vortex local memory starting from a certain offset and of a certain size. Both device memory address, buffer, size and offset are given as input.
- **vx_start:** gives the command RUN to the device given as input.

5.2 Memory allocation

For the allocation of memory in *"vortex/driver/common/vx_malloc.h"* is defined the MemoryAllocator class, which allows to handle a memory block contained between the addresses which are passed to the constructor, along with the alignment size of pages and blocks. It has just two public functions:

- **Allocate:** receives the size of the memory to be allocated and a pointer that will point to the block address that has been allocated. The block is chosen among the already used pages, in particular the smallest that fits with the allocation size. If there is not a free block in the used pages that fits, it is created a new Page. If the max address is reached during the creation of the new page -1 is returned.
- **Release:** receives the address of the block to release and when is found is also checked if there are free adjacent blocks to compact the memory, creating a single bigger block. In case the address is not found -1 is returned.

5.3 Scope handling

There are two main function defined in *"vortex/driver/common/vx_scope.cpp"* and declared in *"vortex/driver/common/vx_scope.h"*:

- **vx_scope_start:** start the recording at a given start time and until a certain stop time, unless the stop time is equal to -1. In particular, it sends the command CMD_SET_STOP and CMD_SET_START to the vortex device.
- **vx_scope_stop:** it forces the stop of the scope and write a dump of the Vortex scope trace, which also include a dump of the module and of taps and write it to the file "trace.vcd".

5.4 Utils function

In the file *"vortex/driver/common/utils.h"* are declared two functions, which are implemented in *"vortex/driver/common/utils.cpp"*:

- **aligned size:** given a size and an alignment, returns the aligned size.
- **is_aligned:** given an address and an alignment, returns (Boolean) whether the address is aligned or not.
- **vx_upload_kernel_bytes:** upload the kernel content, which is passed as input, starting from the device Kernel base address.
- **vx_upload_kernel_file:** it receives a file as input, which contains the kernel that has to be uploaded to the vortex device.
- **get_csr_64:** returns the content of the Control State Register.
- **vx_dump_perf:** it dumps the state of the vortex device writing the information in the file passed as input. It always dumps, for each core, the core id, instructions per core, cycles per core and IPC, which is obtained as (#instructions/#cycles). Moreover it also stores these last three information considering all the cores (the number of cycle is taken as the maximum among all cycle of cores).
  But if PERF_ENABLE is defined, it also stores many other information for each core, which are related to the ICache, the DCache, the SMEM and the memory.
  While if EXT_TEX_ENABLE is defined, also other information related to tex read and their latency is dumped.

6 HDL Report

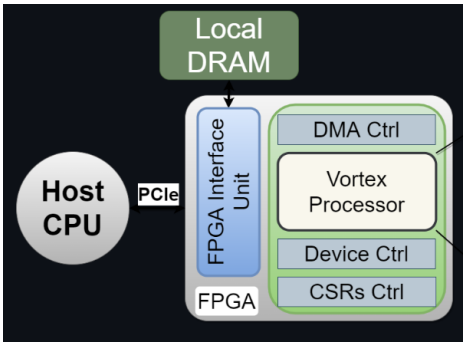

Figure 6: Overview of the Vortex System.

The figure above shows the complete overview of the system that we are going to exploit. Inside the vortex processor the following components could be spotted:

- L3 Cache
- Vortex Cluster

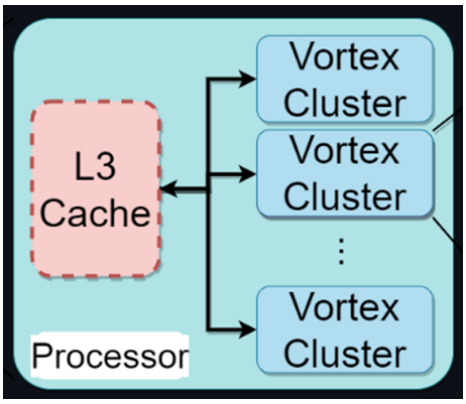

Figure 7: Overview of the Vortex Processor.

Starting from the architecture of the showed processor we can now proceed to illustrate each one of its components.

6.1 Cache

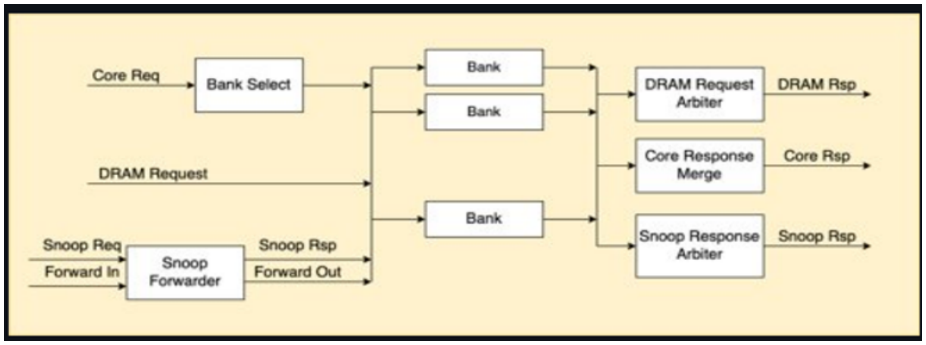

Figure 8: Overview of the Vortex Cache.

Inside the cache, there are several elements that must be configured; the most important ones are the size of the cache, the number of the banks and the bank's size. Once these parameters have been selected, besides the snooping signals, the I/O operations require to set the following wires too:

```
// Core request
input wire [NUM_REQS-1:0]                       core_req_valid,
input wire [NUM_REQS-1:0]                       core_req_rw,
input wire [NUM_REQS-1:0][`WORD_ADDR_WIDTH-1:0] core_req_addr,
input wire [NUM_REQS-1:0][WORD_SIZE-1:0]        core_req_byteen,
input wire [NUM_REQS-1:0][`WORD_WIDTH-1:0]      core_req_data,
input wire [NUM_REQS-1:0][CORE_TAG_WIDTH-1:0]   core_req_tag,
output wire [NUM_REQS-1:0]                      core_req_ready,

// Core response
output wire [`CORE_RSP_TAGS-1:0]                core_rsp_valid,
output wire [NUM_REQS-1:0]                      core_rsp_tmask,
output wire [NUM_REQS-1:0][`WORD_WIDTH-1:0]     core_rsp_data,
output wire [`CORE_RSP_TAGS-1:0][CORE_TAG_WIDTH-1:0] core_rsp_tag,
input  wire [`CORE_RSP_TAGS-1:0]                core_rsp_ready,

// Memory request
output wire                             mem_req_valid,
output wire                             mem_req_rw,
output wire [CACHE_LINE_SIZE-1:0]       mem_req_byteen,
output wire [`MEM_ADDR_WIDTH-1:0]       mem_req_addr,
output wire [`CACHE_LINE_WIDTH-1:0]     mem_req_data,
output wire [MEM_TAG_WIDTH-1:0]         mem_req_tag,
input  wire                             mem_req_ready,

// Memory response
input  wire                             mem_rsp_valid,
input  wire [`CACHE_LINE_WIDTH-1:0]     mem_rsp_data,
input  wire [MEM_TAG_WIDTH-1:0]         mem_rsp_tag,
output wire                             mem_rsp_ready
```

Figure 9: Overview of the cache's wire settings.

The next operation consists in enabling the selection of the banks by which the cache assigns valid and ready signals for each bank. Banks are composed by 3 different stages that corresponds respectively to Request Pick, Data Access and Request Solution.

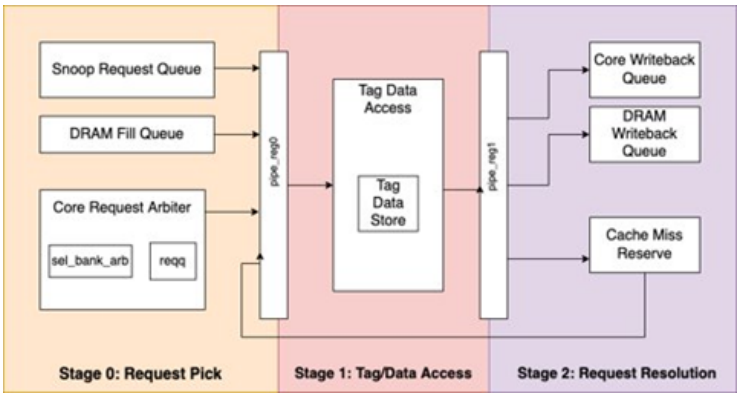

Figure 10: Overview of Bank's Stages.

After the activation of the banks, the next steps consist in the preparation of the cache response for the communication with the DRAM and the following setting of snoop response arbiter which prevents the system from violating cache coherency. The last operation that must be carried out consists in merging the response of the cores for trying to recombine the responses by thread ID due to the possibility that requests may not come back in the same responses.

6.2 Vortex Cluster

Even for this component, the first operations that are carried out correspond to set I/O signals which enable the memory responses in input and the memory requests in output.
The next operations are the activation of the L2 cache and setting of the number of cores per cluster. In fact, as we can notice from the figure, each cluster is composed by a L2 cache and multiple cores. The activation of the cores comprehends also the reset operation of each element and the further configuration of the I/O signals required by the cache. The latter ones are very similar from the signals required by the L3 cache; in both cases, we have memory response/requests and core responses/requests.

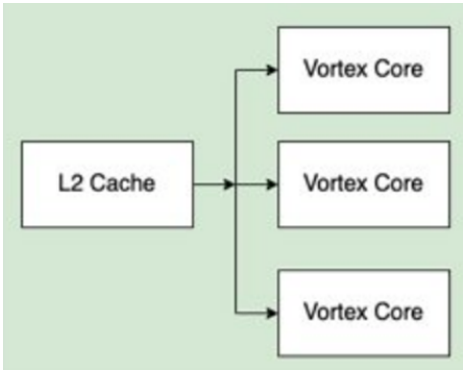

Figure 11: Overview of Clusters.

6.3 Vortex Core

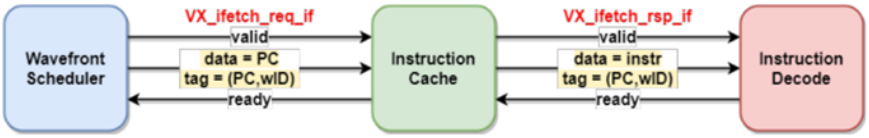

Figure 12: Overview of the pipeline.

The pipeline was designed following the **elastic design pattern**, which enables changes in the pipeline depth late in the design time with low effort, so increasing *flexibility*. In fact, one of the primary aim of the Vortex project is the **extensibility**, that make it easier to maintain and modify the hardware architecture for future reasearch. The first element of the 3 steps-pipeline is represented by the wavefront scheduler that is designed to alert the request of a warp to process the following instruction. The latter will be handled by the instruction cache which requires just two inputs from the scheduler: the Program Counter value of the instruction requested by the warp and the tag (composed by the Id of wavefront).
Once the instruction has been found the following step of the pipeline consists in the decodification of the selected instruction.

The next figure shows the micro-structure of each core. However, the main HDL module limits to the wiring of the I/O memory signals and the configuration of the parameters for the data-cache and instruction cache requests/responses. For both caches the parameters to be set are:

- Word Size
- Tag Width
- Number Of Request (just for data cache)

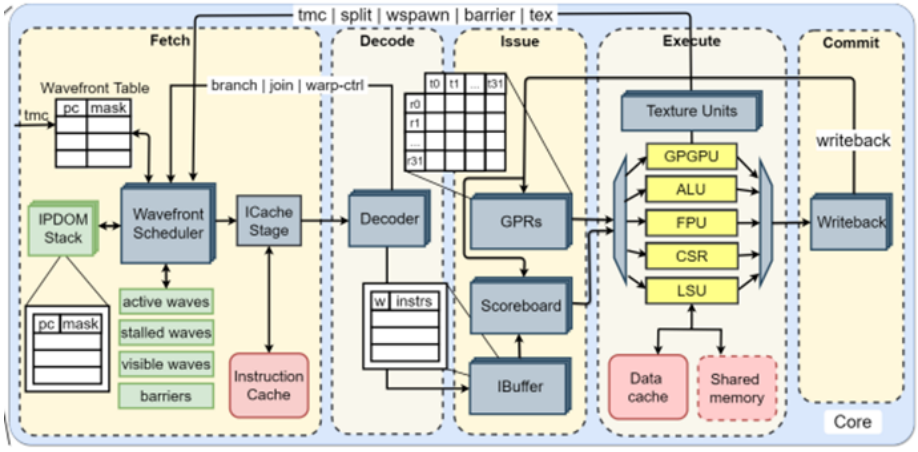

Figure 13: Overview of the Vortex Core.

In the following sections we are going to analyze the modules that describe the main components of the core architecture.

6.3.1 Fetch Section

Warp Scheduler
The first operation that needs to be carried out is represented by the setting of the "reg" for the active wavefronts, which is updated when a warp is activated or disabled, the reg for the number of the active wavefronts and the reg for the stalled wavefronts that is asserted when a brach/gpu instruction is issued. Going beyond the preceeding settings, two different instuctions must be configured:

- Configuration of the "wspawn" instruction which activates a variable number of warps to the same program counter.
- Configuration of the "bar" instruction which synchronizes wavefront execution at barrier locations.

The module is initialized leveraging one main "always" functions, which waits for the triggering event(positive edge) on the clk; in that case the following operations are exploited:

- In the case of "resetting" the gpu, all the declared regs are posed to 0, otherwise the barrier mask, the wspawn mask, the thread mask and the stalled warps of the interested warp's Ids are initialized to 1.
- The next step is represented by the management of the branch/join paradigm. However, at this point just the masks mentioned above are handled and so the pushing operations onto the stack don't appear at this point.

Once all of this has been set, the communication with the Ipdom Stack is configured. In particular, the procedure of split/join paradigm manages the operation of wiring "pop" and "push" instructions leveraging an iteration along all the warps.
The final operation that needs to be conducted consists in the scheduling of the next warp to execute. For this purpose, the selected warps have to be ready and they are collected through this formulation:

```
ready_warps = active_warps & ~(stalled_warps | barrier_stalls)
```

The last step coincides with the iteration along the ready warps activating the threads indicated by the respective mask and fetching the next instruction to execute. The latter will be broadcasted to each core of the available clusters.
The module concludes with the initialization of the Pipe register.

IPDOM
As mentioned before, two wires of these component are the *"push"* wire and the *"pop"* wire; besides them, one can also find the clock wire, the "pair" wire **warp-ctl-if.split.diverged**, the IPDOM data wire and the IPDOM Index wire.
The Ipdom data is composed by the pair Program counter value and thread mask; the following instruction shows how the Ipdom is built one frame at a time:

```
assign join_pc, join_tmask = ipdom_data[join_if.wid]
```

The main "always" function relays on the triggering edge of the clock and it accomplishes push and pop commands. The index instead, is used in the following way:

```
assign index = is_part[rd_ptr];
```

where the rd states for read pointer, intialized with the usage of localparam *ADDRW = $clog2(DEPTH)*:

```
reg [ADDRW_1:0] rd_ptr, wr_ptr;
```

```
assign join_else = ~ipdom_index[join_if.wid];
```

Icache Stage

The master slave interfaces for the Icache are reserved for requests and responses respectively while the fetching needs of this component are satisfied in the opposite direction of the two just mentioned. A request could be fired just in the case in which the Icache is ready and the request is actually valid.
The Icache requests needs two main elements:

- ```
  assign icache_req_if.addr = ifetch_req_if.PC[31:2];
  ```

- ```
  assign icache_req_if.tag = {ifetch_req_if.uuid, req_tag};
  ```

Where the req-tag consists in :

```
assign req_tag = ifetch_req_if.wid
```

The Icache can also be stalled out if the fetch response is not ready or not valid. The only *"always function"* that appears in this component is quite straightforward and it is composed just by these two checks:

```
if (icache_req_fire) begin
    dpi_trace("%d: I$%0d req: wid=%0d,
    PC=%0h (#%0d)\n",
    $time, CORE_ID, ifetch_req_if.wid, ifetch_req_if.PC,
    ifetch_req_if.uuid);
end

if (ifetch_rsp_if.valid && ifetch_rsp_if.ready) begin
    dpi_trace("%d: I$%0d rsp: wid=%0d, PC=%0h, data=%0h (#%0d)\n",
    $time, CORE_ID, ifetch_rsp_if.wid,
    ifetch_rsp_if.PC,
    ifetch_rsp_if.data, ifetch_rsp_if.uuid);
end
```

6.3.2 Decode Phase

Decoder
The slave of the decoder collects the response of the fetching operation coming from the ICACHE stage. The first 6 bit of the instruction corresponds to the **opcode** that indicates the operation to be exploited within the computational Unit. The structure of the instruction is quite similar to the one used by ARM architecture; in fact, also in this case we can find the field related to the *"function"*, which is leveraged in order to specify if we are going to deal with an immediate or a register, to set appropriate bit in the flag registers and to indicate the specific data-processing instruction. In the architecure, we can also spot 4 different registers of which just one is designed as *"destination"*.
Dealing with most common logic operations, in the case in which we are going to work with an immediate, the value of the first bit of *"func7"* is set to 0. In this specific scenario, there are two distinct possibilities:

- The operations are carried out with only one source register (rs1).
- The total number of source registers is equal to 2.

In the case in which we are going to deal with a register, an appropriate reg called op_mode is set to 2. The first ALU instructions in which both opcode and functions value are evaluated concern simple logic action; they are listed below:

- *XOR*
- *ADD*
- *AND*
- *OR*
- *LEFT/RIGHT SHIFT*
- *MULTIPLICATION/DIVISION*
- *MODULE*

Instruction concerning JAL (which branches the pc to a specific offset) or more complex actions such as *"load upper immediate"* are indicate with the only usage of the "opcode" without any specification deriving from the function field.
The usage of the function field is reintroduced in the development of the branch instructions where two source registers are used. The last instruction reserved for the ALU are the ones related to *"Break"* actions.
The next step consists in the specification of the LSU unit and also the FPU unit, which is designed to carry out operations that deal with floating points.
The last instructions that are specified concern the GPU; in this case both func3 and opcode are evaluated and the names of the developed instructions are the one explained in section 2.1, i.e. *wspawn, join, bar, split* and *tex* (for graphic purposes).

6.3.3 Issue Phase

This phase is devoted to issue instructions through which the downstream system warns the upstream one to be ready for processing another instruction.

Issue Module
In the module, slaves come from the decoder and the writeback whether masters are represented by each computational unit that is available in the system: *ALU*, *FPU*, *GPU*, *LSU* and *CSR*.
Once masters and slaves have been configured, the next operation to exploit consists in the configuration of all the interfaces that fall into this section that are **SCOREBOARD**, **IBUFFER** and **GPR**.
The only elements that are set for the GPR are 3 different registers and the warp Id; each one of this component is initialized to the corresponding IBUFFER element.
As we can see from the last figure, the ScoreBoard is the only component of the section whose inputs came from both the writeback and the IBUFFER.
An instruction is issued leveraging the following expression:

```
assign ibuffer-if.ready = scoreboard-if.ready &&
dispatch-if.ready;
```

The only always function that appears in the module is fired with the positive edge of the clock and it is showed in the snippet below, whose mean is quite strightforward.

```
always @(posedge clk) begin
  if (reset) begin
        perf-ibf-stalls <= 0;
        perf-scb-stalls <= 0;
        perf-alu-stalls <= 0;
        perf-lsu-stalls <= 0;
        perf-csr-stalls <= 0;
        perf-gpu-stalls <= 0;
    `ifdef EXT-F-ENABLE
        perf-fpu-stalls <= 0;
    `endif
    end else begin
        if (decode-if.valid & ~decode-if.ready) begin
            perf-ibf-stalls <= perf-ibf-stalls  + `PERF-CTR-BITS'd1;
        end
        if (scoreboard-if.valid & ~scoreboard-if.ready) begin
            perf-scb-stalls <= perf-scb-stalls  + `PERF-CTR-BITS'd1;
        end
        if (dispatch-if.valid & ~dispatch-if.ready) begin
            case (dispatch-if.ex-type)
            `EX-ALU: perf-alu-stalls <= perf-alu-stalls +
            `PERF-CTR-BITS'd1;
        `ifdef EXT-F-ENABLE
            `EX-FPU: perf-fpu-stalls <= perf-fpu-talls +
            `PERF-CTR-BITS'd1;
        `endif
            `EX-LSU: perf-lsu-stalls <= perf-lsu-stalls +
            `PERF-CTR-BITS'd1;
            `EX-CSR: perf-csr-stalls <= perf-csr-stalls +
            `PERF-CTR-BITS'd1;
            //`EX-GPU:
            default: perf-gpu-stalls <= perf-gpu-stalls +
            `PERF-CTR-BITS'd1;
            endcase

        end

    end

end
```

Scoreboard

The main work accomplished by the scoreboard is to create a centralized method for dynamically scheduling instructions so that they can execute out of order when there are no conflicts and the hardware is available.

The scoreboard has two slaves coming from both the writeback component and

the Ibuffer. Moreover, two different registers are wired: *the reserve register, coming from the buffer and the release register, coming from the writeback.* The scoreboard is also composed by 4 register whose main purpose is to collect warp Id value and value of the corresponding register in the Ibuffer unit.

IBuffer

The input of the Ibuffer comes from the decoding phase and so the only one slave of this component is interfaced with the preceeding phase.
As we can see from the figure above, *the ibuffer consists in a table that collects pairs composed by the warp and the instruction that the corresponding warp is going to execute.* For each warps that we are going to handle both reading and writing wires are implemented; the latter is the result of an AND-logic gate that checks, for the analyzed warp, if the input is valid and the decoder is ready, the former is the result of an AND-logic gate that checks, for the analyzed warp, if the buffer is ready and the output of the buffer is valid.
Moreover, the IBuffer is based on the usage of 3 registers (whose length is exactly the same of the total number of warps) which indicate if a selected warp is **"Empty"**, **"Full"** or **"Almost Empty"**. Inside the first *"Always"* function, the management of these 3 regs is carried out for each available warps; in the resetting case, these regs are set to 0 otherwise there are two options:

- In the writing phase, for the selected warp, set the reg full to 1 and the reg empty to 0.
- In the reading phase, for the selected warp, set the reg full to 0 and the reg empty to 1.

The next step coincides with the constitution of the reg *"Valid Table"*, whose length is another time equal to the number of warps. In the case of writing phase, the entry of the warp that we are going to analyze is set to 1 and in the reading phase, it is set to a value that is the opposite of the "almost ready" reg.
The next step consist in organizing the scheduling of the next instruction to issue leveraging a **"Round-Robin"** approach and the end we also need to dispose scoreboard forewarding.

GPR

GPR has two inputs (only one come from the writeback phase) and only one single output; in the module there is only one local parameter that has to be declared that corresponds to the **"RAM size"** which is set to be equal to the product of the number of warps and the number of regs.
Furthermore, we are going to enabling writing operation for each thread that is present in a warp according to the thread mask associated with the thread.

Dispatch

The dispatch module has two slave inputs that comes from the Ibuffer and from the gpr and 5 outputs that are headed to the 5 computational units downstream.
The next program counter value is indicated by the one coming from the Ibuffer plus 4.
For each one of the computational units that will appear in the next phase,within the Dispatch module there is a skid buffer through which the corresponding computational component is represented; the structure of the buffer is the following one.

```
// ALU unit

wire alu_req_valid = ibuffer_if.valid && (ibuffer_if.ex_type == `EX_ALU);
wire [`INST_ALU_BITS-1:0] alu_op_type = `INST_ALU_BITS'(ibuffer_if.op_type);

VX_skid_buffer #(
    .DATAW   (`UUID_BITS + `NW_BITS + `NUM_THREADS + 32 + 32 + `INST_ALU_BITS + `INST_MOD_BITS
    .OUT_REG (1)
) alu_buffer (
    .clk       (clk),
    .reset     (reset),
    .valid_in  (alu_req_valid),
    .ready_in  (alu_req_ready),
    .data_in   ({ibuffer_if.uuid, ibuffer_if.wid, ibuffer_if.tmask, ibuffer_if.PC, next_PC,
    .data_out  ({alu_req_if.uuid, alu_req_if.wid, alu_req_if.tmask, alu_req_if.PC, alu_req_if.n
    .valid_out (alu_req_if.valid),
    .ready_out (alu_req_if.ready)
);
```

Figure 14: Buffer Structure

Where the value for the DATAW parameter can change with respect to the unit that we are considering.

6.3.4 Execution Phase

ALU
The main computational unit take care of Single-cycle operations and Branch instructions (Share ALU resources). It is composed by an input slave coming from the GPR (it is called "alu-request") and two masters, which refer to the branch output and the commit output. For each thread, a reg is set to contain the result of the ALU and 3 different wires for the **"ADD"**, **"SUB"** and **"SHR"** operations are configured. Furthermore, multiple wires are configured in order to understand the following things:

- *The arriving packet is a branch.*
- *The arriving packet is an ALU operation.*
- *The class of the operation to carry out.*
- *The operation is a subtraction.*

Before considering branch operations, in the alu module, we can spot two "always" functions that act at the beginning event. Both these two functions iterate all over the complete number of threads and the former takes care to check the value of the operation to execute while the latter takes care to check the class of the operation.

```
for (genvar i = 0; i < `NUM_THREADS; i++) begin
    always @(*) begin
        case (alu_op)
            `INST_ALU_AND: msc_result[i] = alu_in1[i] & alu_in2_imm[i];
            `INST_ALU_OR:  msc_result[i] = alu_in1[i] | alu_in2_imm[i];
            `INST_ALU_XOR: msc_result[i] = alu_in1[i] ^ alu_in2_imm[i];
            //`INST_ALU_SLL,
            default:  msc_result[i] = alu_in1[i] << alu_in2_imm[i][4:0];
        endcase
    end
end
```

Figure 15: Alu Value Operations

```
for (genvar i = 0; i < `NUM_THREADS; i++) begin
    always @(*) begin
        case (alu_op_class)
            2'b00: alu_result[i] = add_result[i];              // ADD, LUI, AUIPC
            2'b01: alu_result[i] = {31'b0, sub_result[i][32]}; // SLTU, SLT
            2'b10: alu_result[i] = is_sub ? sub_result[i][31:0] // SUB
                                          : shr_result[i];      // SRL, SRA
            // 2'b11,
            default: alu_result[i] = msc_result[i];            // AND, OR, XOR, SLL
        endcase
    end
end
```

Figure 16: Alu Class Operation

In the next step, the branch event is handled and in this case several preliminary configurations are set as we can see in the following snippet. The meaning of the showed wires are significative and they are appropriate to describe the check that they are supposed to implement.

```
wire is_jal = is_br_op && (br_op == `INST_BR_JAL || br_op == `INST_BR_JALR);
wire [`NUM_THREADS-1:0][31:0] alu_jal_result = is_jal ? {`NUM_THREADS{alu_req_if.next_PC}} : alu_result;

wire [31:0] br_dest    = add_result[alu_req_if.tid];
wire [32:0] cmp_result = sub_result[alu_req_if.tid];

wire is_less  = cmp_result[32];
wire is_equal = ~(| cmp_result[31:0]);
```

Figure 17: Alu Wiring Operation

After the configuration of the appropriate outputs, very few assignements are done; in particular, they concern the branch destination, the validity of the branch and also the ID of the interested warp. The following figure instead shows the inputs and outputs related to operations that concern multiplications and divisions; for this purpose, a multidiv structure has been declared and initialized.

```
VX_muldiv muldiv (
    .clk        (clk),
    .reset      (reset),

    // Inputs
    .alu_op     (mul_op),
    .uuid_in    (alu_req_if.uuid),
    .wid_in     (alu_req_if.wid),
    .tmask_in   (alu_req_if.tmask),
    .PC_in      (alu_req_if.PC),
    .rd_in      (alu_req_if.rd),
    .wb_in      (alu_req_if.wb),
    .alu_in1    (alu_req_if.rs1_data),
    .alu_in2    (alu_req_if.rs2_data),

    // Outputs
    .wid_out    (mul_wid),
    .uuid_out   (mul_uuid),
    .tmask_out  (mul_tmask),
    .PC_out     (mul_PC),
    .rd_out     (mul_rd),
    .wb_out     (mul_wb),
    .data_out   (mul_data),

    // handshake
    .valid_in   (mul_valid_in),
    .ready_in   (mul_ready_in),
    .valid_out  (mul_valid_out),
    .ready_out  (mul_ready_out)
);
```

Figure 18: Multiplication and Division

The last thing worth to be mentioned is that the ALU is available to process inputs data just in the case it declares itself ready leveraging the appropriate wire.

FPU

This component presents one slaves which signals the request made by floating point unit and two masters, one directed towards the csr and the other directed to the commit. One wire is used to verify if the unit is full or not while other 2 wires are leveraged in order to signal **"push"** and **"pop"** operations.
Moreover, in order to process the input, the first check that needs to be carried out is the following:

```
// CSR fflags Update
assign fpu_to_csr_if.write_enable = fpu_commit_if.valid && fpu_commit_if.ready && has_fflags_r;
assign fpu_to_csr_if.write_wid    = fpu_commit_if.wid;
assign fpu_to_csr_if.write_fflags = fflags_r;

// pending request
reg [`NUM_WARPS-1:0] pending_r;
always @(posedge clk) begin
    if (reset) begin
        pending_r <= 0;
    end else begin
        if (fpu_commit_if.valid && fpu_commit_if.ready) begin
            pending_r[fpu_commit_if.wid] <= 0;
        end
        if (fpu_req_if.valid && fpu_req_if.ready) begin
            pending_r[fpu_req_if.wid] <= 1;
        end
    end
end
assign pending = pending_r;
```

Figure 19: Pending Request Management

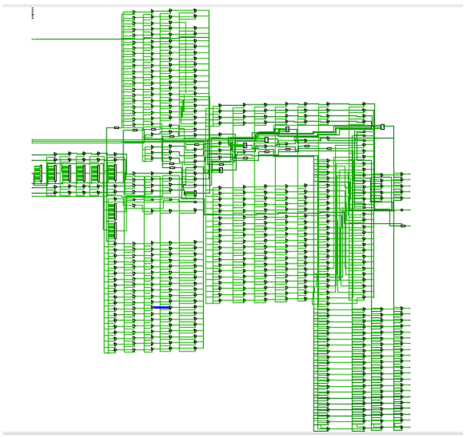

Figure 20: FPU Vivado Output

```
assign fpu_req_if.ready = ready_in && ~fpuq_full &&
```

```
!csr-pending[fpu-req-if.wid];
```

In the next steps, 3 different instantiantions are built: **FPU-DPI**, **FPU-NEWFP** and **FPU-FPGA**. The following snippets shows respectively the udpate of the CSR flag and the handlement of the pending requests.

LSU

This kind of computational unit is interfaced with the shared memory and the data cache. The slave in this case collects the dcache responses while the master collects the dcache requests.
The first operation that is worth to be mentioned is the duty of the unit to compute the full address; it is obtained summing a base address and an offset, iterating this computation for each thread. Once the address is assigned it is also checked to avoid duplication phenomena. The management of the full address can be analyzed in the following figure.

```
for (genvar i = 0; i < `NUM_THREADS; i++) begin
    // is non-cacheable address
    wire is_addr_nc = (full_addr[i][MEM_ASHIFT +: MEM_ADDRW] >= MEM_ADDRW'(`IO_BASE_ADDR >> MEM_ASHIFT));
    if (`SM_ENABLE) begin
        // is shared memory address
        wire is_addr_sm = (full_addr[i][MEM_ASHIFT +: MEM_ADDRW] >= MEM_ADDRW'((`SMEM_BASE_ADDR - `SMEM_SIZE) >> MEM_ASHIFT))
                        & (full_addr[i][MEM_ASHIFT +: MEM_ADDRW] < MEM_ADDRW'(`SMEM_BASE_ADDR >> MEM_ASHIFT));
        assign lsu_addr_type[i] = {is_addr_nc, is_addr_sm};
    end else begin
        assign lsu_addr_type[i] = is_addr_nc;
    end
end
```

Figure 21: Full Address Management

Accepting new requests depends on the result of this assignement:

```
assign lsu-req-if.ready = ~stall-in && ~fence-wait;
```

The snippet shows how LSU computes the request offsets starting from the address of the same requests for each thread.
The request made to the DCACHE is shown in the following figure.

```
for (genvar i = 0; i < `NUM_THREADS; i++) begin
    assign req_offset[i] = req_addr[i][1:0];
end
```

Figure 22: Dcache Request

CSR

The CSR unit core is shown in the figure below:

```
// CSR fflags Update
assign fpu_to_csr_if.write_enable = fpu_commit_if.valid && fpu_commit_if.ready && has_fflags_r;
assign fpu_to_csr_if.write_wid    = fpu_commit_if.wid;
assign fpu_to_csr_if.write_fflags = fflags_r;

// pending request
reg [`NUM_WARPS-1:0] pending_r;
always @(posedge clk) begin
    if (reset) begin
        pending_r <= 0;
    end else begin
        if (fpu_commit_if.valid && fpu_commit_if.ready) begin
            pending_r[fpu_commit_if.wid] <= 0;
        end
        if (fpu_req_if.valid && fpu_req_if.ready) begin
            pending_r[fpu_req_if.wid] <= 1;
        end
    end
end
assign pending = pending_r;
```

Figure 23: CSR Unit Core

Inside the module, just two "always" functions are present; the first is fired at the beginning event and it takes care to distinguish among 3 different operation types (CSR-RW, CSR-RS, CSR-RC) while the second is fired at the positive edge of the clock and it takes care of handling the pending requests.

```
// ensure all dependencies for the requests are resolved
wire req_dep_ready = (req_wb && ~(mbuf_full && is_req_start))
                  || (~req_wb && st_commit_if.ready);

// DCache Request

for (genvar i = 0; i < `NUM_THREADS; i++) begin

    reg [3:0]  mem_req_byteen;
    reg [31:0] mem_req_data;

    always @(*) begin
        mem_req_byteen = {4{req_wb}};
        case (`INST_LSU_WSIZE(req_type))
            0: mem_req_byteen[req_offset[i]] = 1;
            1: begin
                mem_req_byteen[req_offset[i]] = 1;
                mem_req_byteen[{req_offset[i][1], 1'b1}] = 1;
            end
            default : mem_req_byteen = {4{1'b1}};
        endcase
    end

    always @(*) begin
        mem_req_data = req_data[i];
        case (req_offset[i])
            1: mem_req_data[31:8]  = req_data[i][23:0];
            2: mem_req_data[31:16] = req_data[i][15:0];
            3: mem_req_data[31:24] = req_data[i][7:0];
            default:;
        endcase
    end
```

Figure 24: Pending Request Management

GPU

Inside this unit, the configuration of the instructions "wspawn", "split", "barrier" and "tmc" is implemented. Both the wspawn and the tmc instructions need an assignment to check validity and an assignment to set the thread mask and wspawn mask respectively (for the latter case the value for the wire is iterated along the total number of warps and wspawn needs a "program counter" assignment too). For the **"split"** instruction, we need to add a **"diverged mask"** assignment, a **"then tmask"** assignment, a **"else tmask"** assignment and a program counter assignment. For the barrier instead, it is worth to mention just the need to insert a barrier id assignement.

7 OpenCL examples

Vortex supports OpenCL, which is a standardized, cross-platform API designed to support portable parallel application development on heterogeneous computing systems.
The POCL open-source framework was used to implement:

- **The compiler**, which was modified to generate kernel programs adapted to the Vortex ISA.
- **The runtime**, which was modified to access the Vortex driver, enabling communication with the FPGA via PCIe.

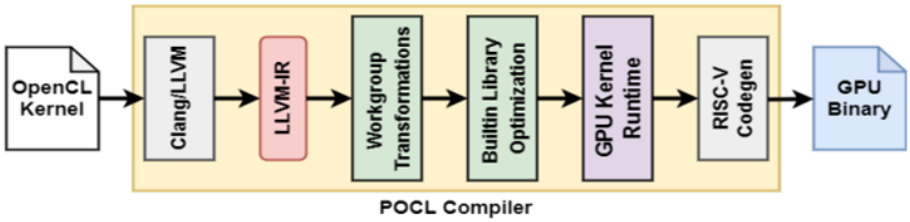

Figure 25: Overview of POCL Compiler.

In the directory *"vortex/tests/openCL/"* are present many examples written using this API:

- **BlackScholes:** contains an OpenCL implementation of the Black Scholes model for the computation of the evolution of an European option price.
- **DotProduct:** perfom the dot product of two float vectors, in particular it assigns to each work-item the dot product of 4 elements of a 256-dimension vector.
- **VectorHypot:** implements a element by element vector hypotenuse computation using 2 input float arrays and 1 output float array.
- **bfs (*):** implements the Breadth-First-Search in a graph that is specified in a text file.
- **convolution:** apply a convolution operation over an image using a 7x7 filter, to apply a 45 degree motion blur.
- **cutcp:** receives a PQR molecular structure file as input and execute Cutoff-limited Coulombic Potential (CUTCP), thus computing a short range component of this.
- **gaussian (*):** solves systems of equations given as input using the gaussian elimination method.

- **kmeans:** execute the kmeans algorithm on data given as input.
- **lbm:** it applies the Lattice Boltzmann methods (LBM) on input data.
- **mri-q:** create the Q data structure for fast convolution-based Hessian multiplication for arbitrary k-space trajectories.
- **nearn (*):** computes the nearest location to a specific latitude and longitude for a number of hurricanes.
- **oclprintf:** just print on the screen a message for each work item with the same message, except for a counter to differentiate output.
- **psort:** sort an array.
- **reduce:** perform a reduction operation on an array of values to produce a single value.
- **sad:** measures the sum of absolute differences (SAD), i.e. a measure of the similarity between image blocks that is calculated by taking the absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison.
- **saxpy (*):** multiply a vector by a certain factor.
- **sgemm (*):** matrix-matrix multiplication of two matrix given as input.
- **spmv:** perform a sparse matrix-vector multiplication.
- **stencil:** implements 7-point stencil code.
- **transpose:** perform the transpose of a matrix given as input.
- **vecadd (*):** return a float vector given by the sum of two float vectors given in input.

The examples marked with **(*)** were used as benchmark to evaluate the Vortex performance in section 5.

All the examples are composed by:

- A ***"kernel.cl"*** file, which contains the code of the kernel, within a function with the *"__kernel"* qualifier, which will run on the Vortex device.
- A ***"main.cpp/cc"*** file, in which are followed these steps:
  1. **Allocates memory** for host buffers and initializes them.
  2. **Gets platform and device information** using clGetPlatformIDs() and clGetDeviceIDs(), respectively.
  3. **Create the context** using clCreateContext().

4. **Create the command queue** using clCreateCommandQueue()) to keep track of different calls you do to the target device.
5. **Allocate memory buffers** on the vortex device using clCreateBuffer() that will be written using clEnqueueWriteBuffer().(*)
6. **Create a program from the kernel** source using clCreateProgramWithSource().
7. **Build the program** using clBuildProgram().
8. **Create the OpenCL kernel** using clCreateKernel and passing the previous program created.
9. **Set the arguments of the kernel** using clSetKernelArg().
10. **Enque the command to execute the kernel** using clEnqueueNDRangeKernel() and passing three key arguments, i.e. the kernel, the number of work-items you wish to execute (called the global size) and the number of work-items you wish to group into a work-group (called the local size). These size can also be of 2 or 3 dimension, e.g. in *"convolution/main.cpp"* are passed local and global size of 2 dimension.
11. **Read the results of the computation** usually stored in a buffer and read thanks to clEnqueueReadBuffer().
12. **Clean up** and wait for all the comands to complete using clFlush() and clFinish().
13. **Release all the host buffers and openCL allocated objects** using free() for the former and clReleaseContext(), ReleaseCommandQueue(), clReleaseMemObject(), clReleaseProgram() and clReleaseKernel() for the latter.

(*)Depending on the type of example can be allocated also different memory space on the vortex device, e.g. in *"convolution/main.cpp"* is used clCreateImage2D that will be written using clEnqueueWriteImage().

Thread mapping. The code in the kernel represents the algorithm to be applied to a single work-item. So the **granularity** of the work item is determined by the implementer through the **thread mapping**, in fact depending on how are used the information related to the thread id, the thread can access different data elements.
The thread hierarchy can have up to three dimension and the choice of dimensions should be guided by the size of the data structure of the task. In particular the functions that are important for the Thread mapping are:

- get_global_size()/get_local_size().
- get_global_id()/get_local_id().

7.1 Simulation

The simulation functionalities could be partitioned into 4 sections:

- Common functionalities
- Register Transfer Level functionalities
- Component functionalities
- Interfacing Functionalities

Common Functionalities
The first element belonging to the common partition, refers to the abstraction of the memory. In the module, we could spot the representation of different types of memory such as:

- **RAM device:** it 'push back' an input file and then make available read and write functions to operate on the data added through the 'push back'.
- **ROM device:** it implements just a write function in order to try to insert code into the device.
- Translation Lookaside Buffer: simulating the unit used by the MMU to speed up the translation of addresses
- **RAM:** differently from the device above, it implements reading and writing functionalities at specific addresses which are specified by the size of the partitions of the interested memory. Moreover, it also enables the load of the binary Image of the code and the hexadecimal Image of the code.

Besides the memory units, another element for the common functionalities is represented by the module 'rvfloats' which can guarantee the execution of each possible operation leveraging 32bit or 64bit floating point format.

Register Level Functionalities
The main component of this partition is represented by the processor abstraction where it is simulated in each one of its aspects. At the beginning, clock and mapping operations have been set in order to proceed with the reset of the state machine; besides that, this abstraction executes the program taking care of handling each one of connected device.
The processor abstraction, together with abstractions encountered in the previous section, is implemented within a main module which takes care to load both memory and computational components in order to exploit the execution of the selected program.

Component Functionalities
Inside this section, we could spot the simulation of each one of components that compose the different steps of the pipeline (inside the core component), starting from the Decoder to the implementation of the shared memory.

The only one element that, in some way, level-up to a high level of abstraction w.r.t the mentioned elements are represented by the simulation of the warps.

Interfacing Functionalities
This final module tries to simulate the component that takes care of communication and interfacing duties. We can spot two main different elements:

- FPGA representation: the first two operation that needs to be implemented coincide with the management of the buffer which needs to handle releasing data and preparing spaces. Furthermore, it implements also reading and writing operations from and to MMIO addresses
- OPAE representation: it implements all the elements that have been just mentioned for the FPGA and elements for the alignment of memory buffer.