Politecnico di Torino

Evaluating the impact of counters for the in-field test of GPUs

Master degree in Electronics Engineering

Advisors: Prof. Sonza Reorda Matteo
Prof. Sterpone Luca
Dr. Rodriguez Condia Josie Esteban

Author: Mohammadmahdi Mohammadi

April 4, 2021

Summary

Graphic Processing Units (GPUs) are a group of processor units being used beside CPUs with the aim of increasing the processing speed. These devices nowadays are being massively employed not only for graphical purposes but also in other domains such as DSP, bioinformatics, machine learning, etc. The fast computation speed of a huge amount of data as the main advantage of these processors makes them increasingly popular in different systems and applications. The GPUs being used beyond the predefined graphical applications are also called as GPGPUs (General-Purpose Graphics Processing Units).
Despite the undeniable benefits of GPUs, they suffer from faults that can cause application misbehaviors which is not acceptable in critical applications. On the other hand, the massive complexity of GPUs (especially the ones produced by the new advanced technologies) makes them difficult to be tested. Furthermore, after being mounted on a board, it is not possible to test a GPU using external testers such as the ones being used at the end of the production. To effectively face this challenging issue, a GPU should be provided with additional observability and fault detection capability. While various tests can be done on a device during different phases of production, the mentioned capability will give the user an opportunity to get a vision about interactions between the device and the real environment and check the correct behavior of the device during its operative life; this feature is known as in-field test feature. In the past several options have been proposed, including software, hardware, and hybrid approaches. An easy to implement efficient option to accomplish the mentioned goal is using a specific monitoring unit being accessed directly by simple test instructions.
The current work focuses on adding an in-field test structure to increase observability and allowing the identification of design errors and hardware faults in GPUs. For this purpose, an open-source GPGPU model (FlexGripPlus) is employed to validate and analyze the proposed solution. The selected model is a VHDL-based GPGPU that is basically an extension of the FlexGrip model which was developed for academic usages at the University of Massachusetts Amherst.
The testing approach presented here is based on including a performance monitoring unit in the mentioned model. The designed unit is in charge of observing and recording the activities on some specific signals taken from critical parts of the processor. The recorded results can be accessed by the user through the general-purpose registers and can be exploited to perform a fault analysis. The presented unit is fully accessible by the newly defined instruction.
As the first step, the performance monitoring unit (PMU) is designed. This unit includes four sections which are a decoder to deliver commands to the desired monitoring subunit, an array of sixteen monitoring subunits, each containing an FSM controller and a 32-bit

synchronous counter, a converter in charge of converting raw input vectors or bits into recordable single-bit signals for counters, and finally a 32-bit 16 to 1 mux to drive the output using the desired subunit's content. This unit is then individually tested, validated, and synthesized before being included in the model.

For the purpose of this thesis, the selection of input signals is done based on their toggling activities, meaning that the signals with the highest toggling activities are chosen to be monitored. The focus is on taking signals from critical units of the processor such as scheduler and execution unit. Although the designer is free to choose the input signals with any size, considering the complexity of the design, vectors are preferred to be monitored instead of single-bit signals.

As the next step the required instruction to activate, reset, and read the content of the monitoring subunits is defined. Since the actual assembly instruction format (SASS) is not available to the public, following the standard format is not possible and the definition is done manually. A single free op-code is chosen for this instruction and three different options to perform the mentioned operations are considered.

After designing the monitoring unit, choosing its inputs, and defining the related instruction, it should be included in the model. For this aim, the decoding unit of the processor is modified to make it able to handle the new instruction by commanding the performance monitoring unit and also other stages. The fetch, read and write stages are almost untouched. The output of the monitoring unit is connected to the execution stage. In the case of performing a read operation, the execution stage will forward the data from the PMU to the write stage while the destination address will also be provided by the decoder and forwarded to the write stage.

After modifying the basic model, a simulation-based validation campaign has been performed to verify the correct behavior of the modified processor. Synthesis has also been performed on both the basic and the new model to get a vision about hardware costs of the modification, including the timing, power, and area ones.

The current thesis report is organized as follows: In chapter 1 a brief introduction about GPUs, selected model and testing methods are given, Chapter 2 introduces the proposed test solution and describes details of the designed unit, Chapter 3 reports simulation results and modification costs, Finally, Chapter 4 provides the conclusions of this work.

Acknowledgement

Many thanks to Prof. Matteo Sonza Reorda for proposing this interesting topic and accepting me as his student to work on this subject.

I would also like to thank my tutor, Prof. Sterpone Luca, for his valuable guidance throughout this work.

Special thanks to Dr. Josie Esteban Rodriguez Condia for helping me through this journey by being always available and giving me hints to finish this thesis.

And thanks to my father and mother who let me come far away for studying and set me off on the road to get my master degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter some basic information about parallel processors and specially GPUs is given. Next, a chosen hardware and its programming model is briefly studied. Finally, some testing and fault detection methods and the suggested one are introduced.

1.1 A background on parallel processing

Over the past decade, processing power demand is significantly increased in new applications, the famous Moore's law is no longer valid because of technical problems and due to the energy issues caused by increasing the clock frequency, consequently, the traditional single-core architectures are almost being considered an outdated option nowadays and are not able to meet the needs of fast high-performance applications.
The mentioned issue pushed the computer system designers to a different way of responding to the current demands which are named **parallel processing** [1]. Meaning that a program that can be run in a parallel way using multiple hardware units is able to be far away faster than a single sequential based one, in other words, parallelism means trying to perform the same action on independent data at the same time.
Although this solution provides systems with a significant increase in processing speed, there are many challenges in following this approach which is partitioning the code into balanced sections, scheduling, synchronizing, communication between different sections, high memory load, etc. Even if there are many issues, the achieved speed-up by using parallel processing methods keeps this approach still appropriate and worthy [2].
Computer systems can be classified based on different characteristics such as the number of processors, memory architecture, and how many programs they can execute. This classification base that has been introduced by Michael J. Flynn in 1966 and is known as FLYNN's taxonomy [3], leads us categorizing computer systems into four main groups which are:

- **SISD**: Single Instruction Stream, Single Data Stream. In this type, a single CPU works in a serial way, meaning that it fetches and executes a single instruction, and stores a single data item at a time.

- **SIMD**: Single Instruction Stream, Multiple Data Stream. This structure uses a single control unit which deals with a single instruction at a time, but they benefit from using multiple numbers of processing cores. These multiple cores are being controlled by a single control unit that is in charge of creating control signals, but each processor takes its own data set.

- **MISD**: Multiple Instruction Stream, Single Data Stream. These machines can execute a set of instructions on the same data item.
- **MIMD**: Multiple Instruction Stream, Multiple Data Stream. Being also called as multiprocessors, these type uses multiple processors to deal with multiple instructions and multiple data stream.

The SISD structure is clearly outdated, the MISD one is also not being used in modern systems. In contrast, SIMD and MIMD structures are the basic architectural organization of modern computer architectures. In general SIMD architecture is almost preferred to MIMD due to many advantages including less hardware and memory requirements, less startup time, easier programming and debugging, less complicated control unit, easier synchronization procedure, and finally, fewer cost [1].
The current thesis work is focused on a SIMD system. In this kind of architecture, the processor uses a single control unit which takes a single instruction into account to control multiple execution units in parallel. The control unit creates signals to command all processing units which are taking different data as their input. In other words, different processing units have their own data to work on while all will do the same task at the same time. A general view of this architecture can be seen in the figure 1.1.

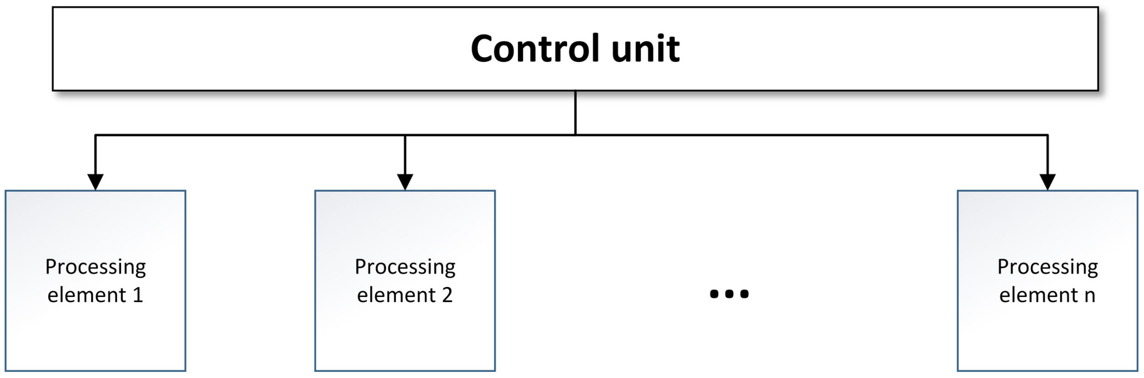

Figure 1.1: General view of the SIMD architecture, adapted from [1]

Clearly, the cost of the control unit is divided over the multiple processing units which is the main advantage of this architecture. The reduced instruction bandwidth and memory space are other advantages gained by SIMD structures.

1.2 GPUs

The main historical aim of introducing the SIMD architectures was to handle the time-consuming graphical operations as fast as possible. As the gaming market size became larger the need for graphical processing power increased. Since the gaming community was seeking different aims compared to the microprocessor community, specific processing solutions which were totally different than normal sequential processors or CPUs have been developed. Nowadays the kinds of processors designed for those purposes are being called by their own terminology and are known as **GPUs** or **Graphical Processing Units**. It should be mentioned that nowadays there is two main terminologies related to the GPUs that are being used by two leading companies of NVIDIA and AMD. Based on the chosen model, in the current work we will stick on Nvidia's terminology in the further discussions.
A GPU is actually an extra proceeding unit that is normally being used besides the main CPU, although GPUs are fast processing units so far they are not being used as a stand-alone unit, it is because the codes dealing with I/O devices cannot be well parallelized to be used efficiently by a GPU, which leads to significant performance degradation. Therefore, they are always being used in a system including a CPU that is more efficient in handling the mentioned operations. Since a CPU and a GPU are designed for different goals some differences can be noticed between them. Compared to a CPU, a GPU is capable to handle higher instruction throughput and memory bandwidth with the same power and area requirements.
A GPU is designed in a way to be able to perform a big number of computations in parallel while in the case of dealing with a single instruction it is slower than a CPU. As the result, the silicon area of a GPU is mostly dedicated to data processing units instead of control and data catch ones [4].
The area dedication difference between a CPU and a GPU is illustrated in the figure 1.2.

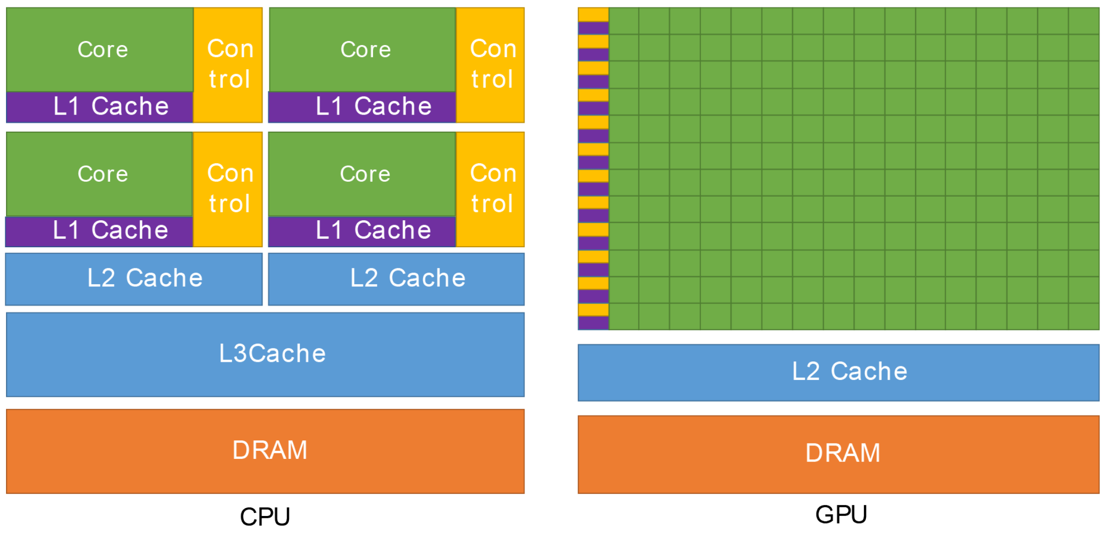

Figure 1.2: Area dedication comparison between a GPU and a CPU, adapted from[4]

Another important difference that can be discussed is about the memory requirements. The systems with an integrated GPU use the same DRAM memory space with the same technology for both processors. This case is more common in low-power applications. But due to different functionalities, to increase the performance, some systems include discrete GPUs. In these structures, each processor has its own DRAM memory space in which different technologies due to different requirements are used. For a CPU a low latency DRAM is needed while for a GPU high throughput (Bandwidth) is more essential [5]. That is because a GPU overcomes the latency of memory by working on large amounts of data in parallel, while a CPU employs complicated memory hierarchies and catch management policies.
A basic memory configuration diagram of a system including a GPU and a CPU is given in the figure 1.3.

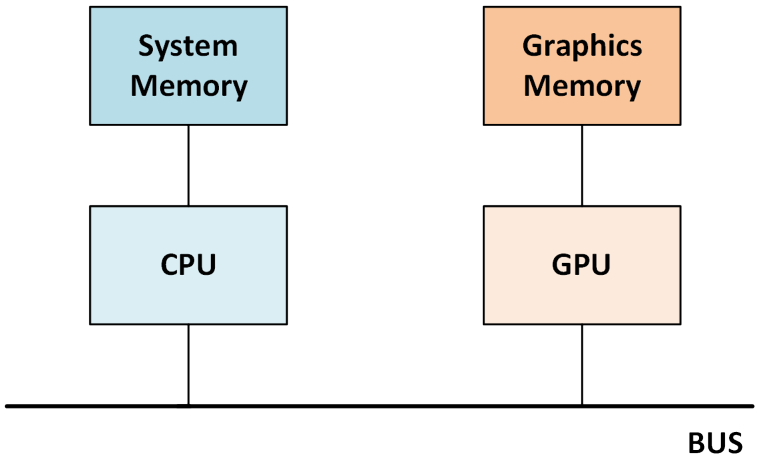

Figure 1.3: Scheme of a system including GPU and CPU, extracted from [5]

While running a code, whenever a parallel computation was needed, the CPU configures the GPU, and provides it with the proper commands, the CPU will also transfer the input data toward the GPU and take the final results at the end of computations. This procedure includes specifying the number of threads and addresses of the related data as well. The part of code related to GPU is known as the kernel. More details about the programming model of a GPU will be given in the section 1.4.
A Modern GPU contains multiple numbers of cores, these cores are called *Streaming Multiprocessors* by NVIDIA and *Compute Units* by AMD. The mentioned cores execute a **single-instruction multiple-thread (SIMT)** program related to the lunched kernels. The described model is a single SIMD processor being used by multiple numbers of threads that can reach thousands. The main aim of implementing SIMT processing architecture and employing many threads is to cover probable catch misses and overcome memory latencies [5]. Managing the execution and operation of the parallel programs is done by some specific scheduling controllers and dispatchers which will be discussed more extensively in the next section. A generic architecture of a modern GPU can be seen in the figure 1.4.

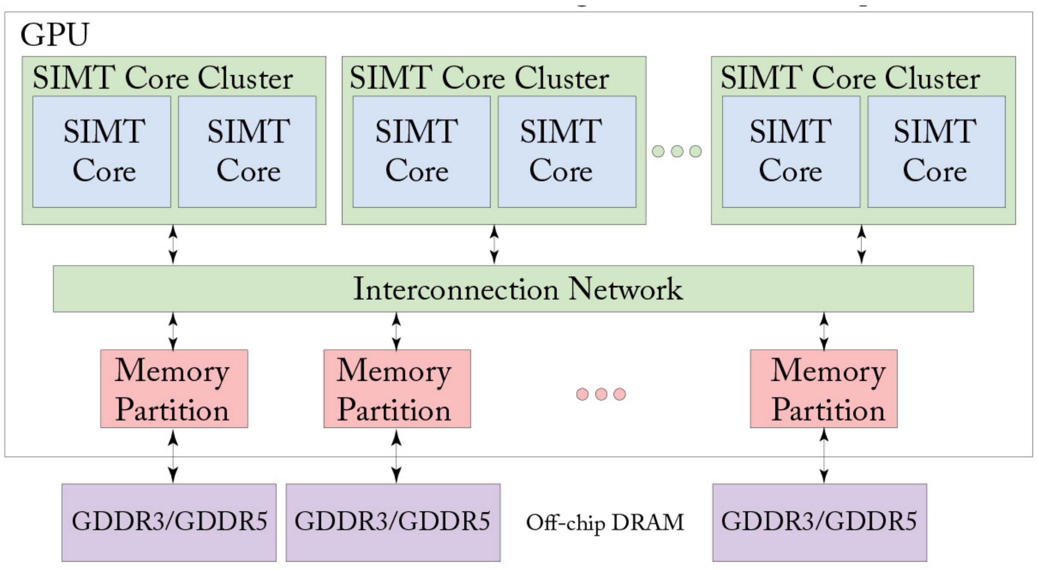

Figure 1.4: Generic architecture of a modern GPU, taken from [5]

1.3 FlexGripPlus GPGPU

A **General-Purpose Graphics Processing Unit (GPGPU)** is basically a GPU that is being used for performing fast applications beyond graphical ones such as DSP [6], bioinformatics [7], machine learning [8], data analysis [9], etc.
The aim of introducing this idea was to accelerate running applications that were traditionally handled by CPUs and improving the overall performance of a CPU-included system. These devices are easy to be programmed using high-level languages such as CUDA, and their capability to automatically handle complicated multithreaded applications helps the programmers to mainly work on increasing the parallelism and efficiency of their codes.
Like GPUs a GPGPU implies multi-core architecture with the key feature of parallel computing for achieving high processing speeds. Such a multicore device contains an array of streaming multiprocessors (SMs) which are containing a certain number of scalar processors (SPs). Figure 1.5 shows an overall architecture of a GPGPU.

The given architecture is divided into two parts which are the **HOST** or CPU running the kernel containing the GPGPU related instructions from the main code and the **DEVICE** which is the GPGPU. Whenever a kernel is being launched, it includes a number of threads, the overall collection of these threads is called as **Grid**. Furthermore, each group of threads is managed in a way to be scheduled toward an available SM and executed in parallel, these groups of threads located inside a grid are known as **Thread Blocks**. The illustrated GPGPU structure in the figure 1.5 can be described based on its hardware blocks as:

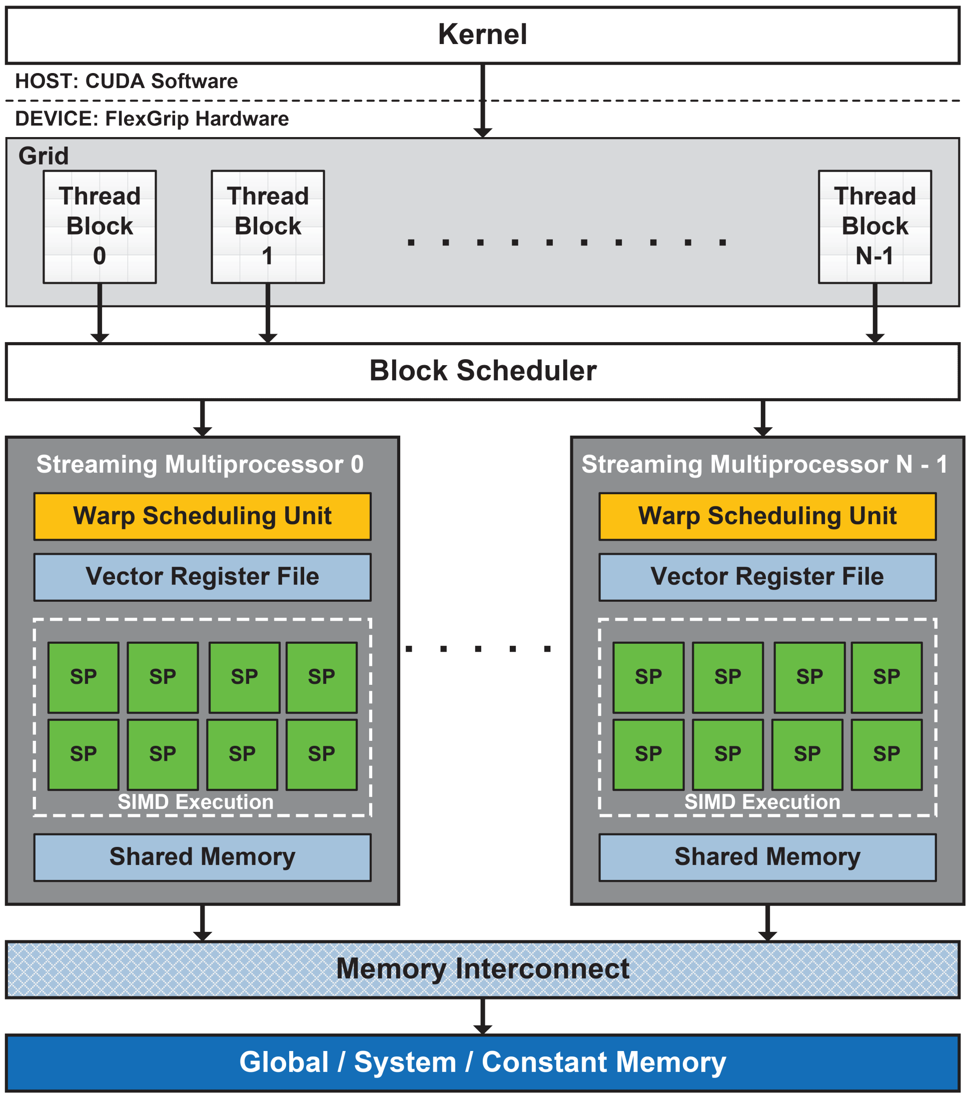

Figure 1.5: Overall architecture of a GPGPU, extracted from [10]

- **Block Scheduler**: The part of GPGPU in charge of scheduling the thread blocks. The number of Blocks scheduling at the same time is limited by the available processing resources which is the number SMs and the SPs inside them.

- **Streaming Multiprocessor**: A pipelined streaming multiprocessor containing SPs. The word streaming refers to SIMD architecture. The pipeline stages are Fetch, Decode, Read, Execute and Write.

- **Warp Scheduling Unit**: A smaller set of simultaneous operations within a block is named WARP in NVIDIA's terminology. Like blocks the warps have to be scheduled. This procedure is done by one of the most critical units of the GPGPU known as the Warp scheduling unit.

- **Vector Register Files**: A pool of registers which is partitioned to be used specifically by each SP with the aim of removing data hazards. Each SP has its own section in the register pool for storing the intermediate results.

- **Scalar Processors**: The processing unit in charge of performing desired operations on input data such as addition, subtraction, multiplication, boolean logic operations, etc.
- **Shared Memory**: A memory to be used for performing communications between the SPs inside the same SM.
- **Global Memory**: A memory performing communications between the SMs.
- **System Memory**: This memory is in charge of storing the kernel instructions.
- **Constant Memory**: The read-only memory initialized by the host and readable by all threads. This memory acts as a fast-accessible cache for the SMs.

In this thesis work, a GPGPU hardware model named as **FlexGripPlus or FLEXible GRaphIc Processor Plus** is selected to be studied [11], [12]. The mentioned model is a debugged and extended version of a basic model called as FlexGrip. FlixGrip is an open-source model introduced for academic usages in 2010 by researchers at the University of Massachusetts Amherst and is based on NVIDIA G80 microarchitecture [10], later extensions are done at politecnico di torino which was about making it able to execute more instructions, removing dependencies of the model on specific technology libraries and making it fully compatible with CUDA programming environment. The RTL code of FlexGrip is developed using VHDL and is optimized to be implemented on the XILINX FPGAs. Along with Speedups of up to 30 compared to the MicroBlaze microprocessor and the ability to run the CUDA binary code, its main advantage is that the designed FPGA model does not need to be recompiled (and resynthesized) by changing the application. Since FlexGrip modification does not affect its general architecture, we will introduce the model based on the basic model. The block diagram of the FlexGrip streaming multiprocessor is given in the figure 1.6.

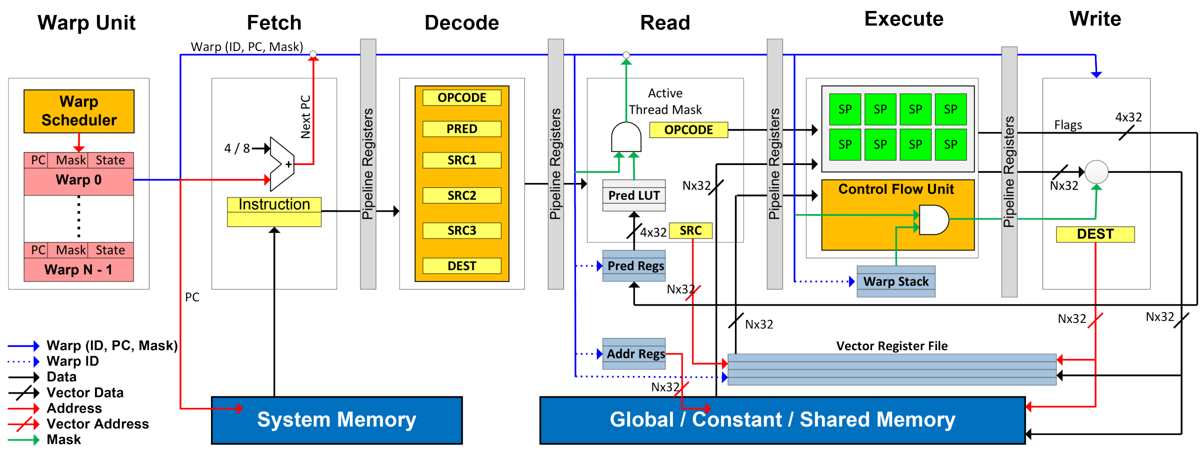

Figure 1.6: FlexGrip streaming multiprocessor block diagram, extracted from [10]

The designed model implies a pipelined 5 stage streaming multiprocessor. As it can be seen there is no central control unit in this architecture and the pipeline is being managed by a handshake communication between stages [13]. This idea is implemented using two signals which are:

- **Done**: If the done output signal of a unit were equal to 1 it means that its outputs are valid for the next stage.
- **Stall**: This signal is used to inform the previous stage if the next stage is busy or free to get new data inputs.

The basic structure of the mentioned communication idea is shown in the figure 1.7.

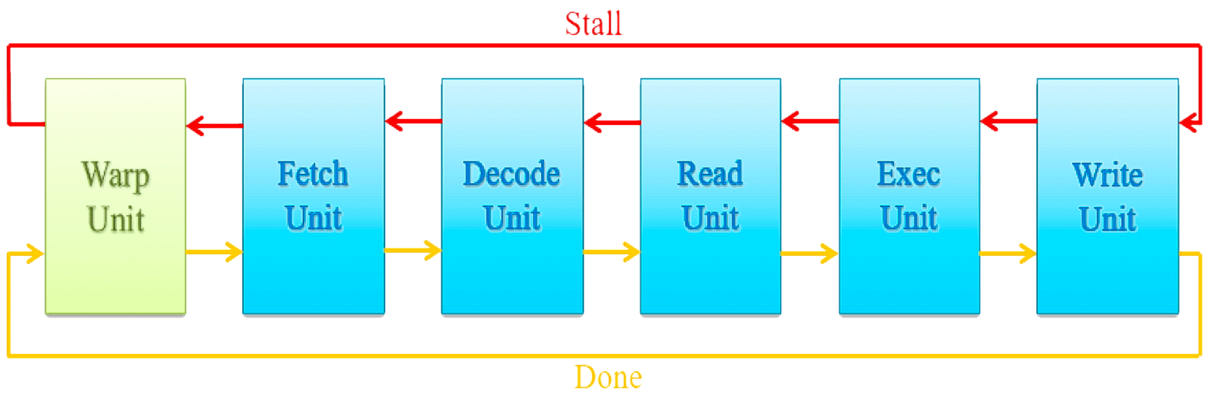

Figure 1.7: General scheme of communication between pipeline stages, taken from [13]

The pipeline stages will be discussed in the following.

1.3.1 Fetch and decode

In this stage, each binary instruction code is being fetched from the memory and fed into the decoding pipeline registers. The fetched codes are then being decoded by the decoder and executed in SIMD style by the threads. The decoder unit will send proper commands toward other units based on the opcode, source and destination type and address, and so on. Addressing the memory to give the right instruction bit is based on instruction length which can be 4 or 8 bytes.
The designed unit in the current thesis work takes its inputs including command bits and addresses of the desired subunits from this stage. More detail about this unit and related instruction bits are given in the next chapter.

1.3.2 Read

As it can be seen from the figure 1.6 in the read stage, based on the decoded instruction, the source operands are taken from the vector register file or global/shared memory. The vector register file is being used to store intermediate results and is managed in a way to be divided between threads, the threads within a warp are categorized into warp rows, each tread running within an SP has a dedicated portion in the register bank. The total number of registers is a hardware specification and should be chosen based on the maximum requirement, once fixed it can be configured to be divided between threads, in some applications even fewer registers compared to available ones will be needed. Figure 1.8 shows the mentioned configuration.

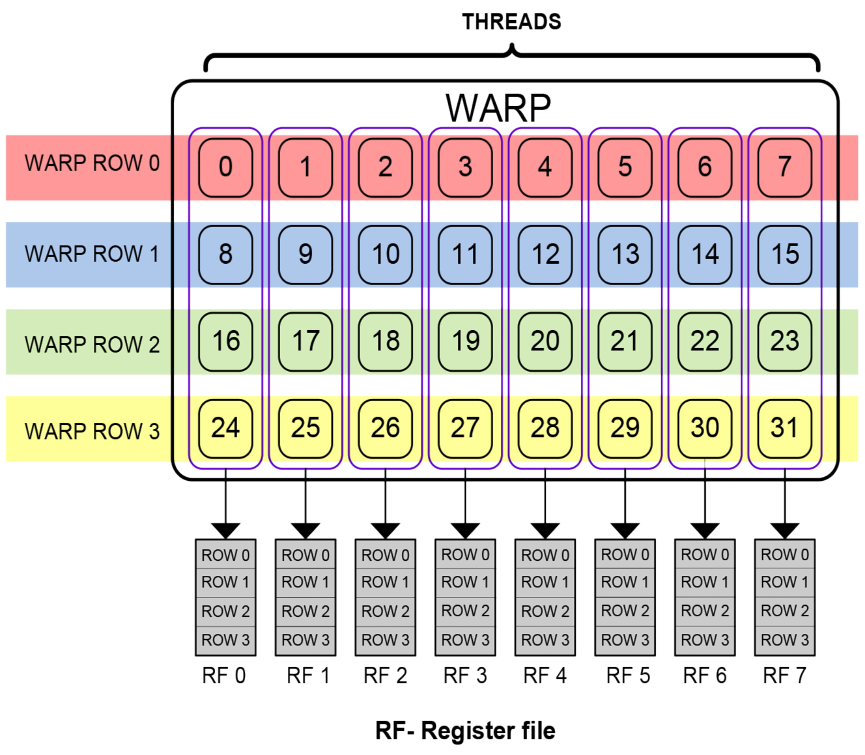

Figure 1.8: Register file configuration of the FlexGrip GPGPU, adapted from [14]

The read pipeline stage in this work will be kept almost untouched, in the case of dealing with test instructions, since no read operation from the mentioned sources is needed, this stage will simply skip the instruction.

1.3.3 Execute

Execution is the fourth and most important stage of Flexgrip's pipelined architecture. The scalar processors and the flow control unit are in this section. Each SP is being used by a single thread and will provide the GPGPU with the ability to perform arithmetic operations such as multiplication, addition, subtraction, data type conversion, and boolean logic and shifting operations. The control flow unit is in charge of managing the synchronization of the threads and also the branches.The basic scheme of an SP core is given in the figure 1.9.
The output of the designed performance monitoring unit is given to this stage and is treated similarly to arithmetic subunits at this stage. Whenever a read of the contents

was desired, this stage will provide the write stage with the data given by the mentioned unit. Since this stage is one of the most important stages some of the input signals of the performance monitoring unit are taken from this stage.

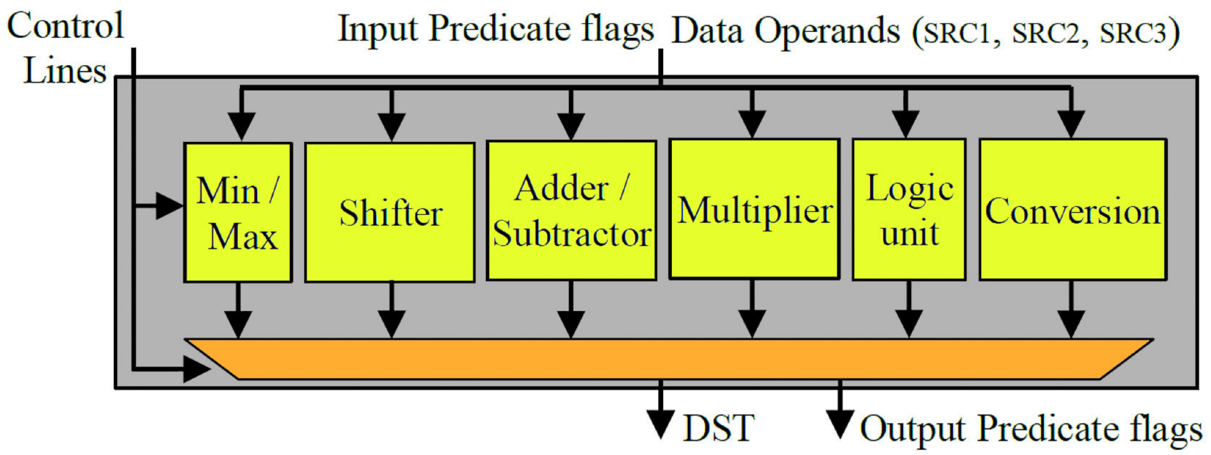

Figure 1.9: The basic scheme of an SP core, taken from [15]

1.3.4 Write

This stage is the last one in which the intermediate results of operations are stored in the register file which is our desired end in this work. Other operations done in this stage are reported as storing addresses in the address registers, predicate flags in the predicate registers, and final results in the global memory.
The final aim of this work is to make this stage able to store the monitoring results given from the execution stage to the destination addresses provided by the decode stage. It should be considered that the number of monitored signals may be more than the number of available registers to each thread, leading to possible data hazards. In such cases, the user may increase the number of registers in configuring the GPGPU or simply read the contents and store them in the global memory to prevent any possible hazard.

1.3.5 Warp unit

As already mentioned, a set of threads managed to be executed in parallel is named WARP in NVIDIA's terminology. After assigning a thread block into an SM by the block scheduler, the warp scheduler starts mapping warps into SPs, as result, each thread is mapped to a specific SP to be executed in parallel with others but in a data-dependent way. In other words, the warp unit is an initial stage in charge of the generation and coordination of the warps toward the other pipeline stages. This procedure includes dispatching the warps and scheduling them (in a round-robin fashion) in order to manage the execution of the threads. The importance of this unit can be noticed more when the resources are not enough for simultaneous execution of threads or when there are divergences due to thread-dependent branches.
Each warp has a program counter (PC), a thread mask, and a state. Any warp keeps

its own PC and can follow its own conditional path. To prevent the execution of the threads not matching the predefined conditions and managing the divergences (in the case of branches) the idea of masking is introduced. Since the architecture is a SIMT one, whenever a thread within a warp diverges from others due to a data-dependent branch, all the threads are being considered to take the branch but those which are not truly taking the branch are disabled by the masking procedure. This idea is illustrated in figure 1.10 in which two groups of threads are following different paths and are being masked in the case they should not be active. This procedure continues until all the threads reach a re-convergence point.

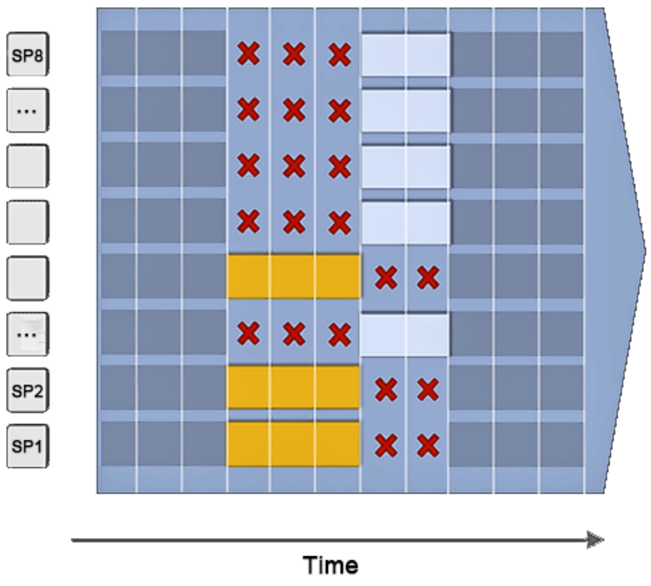

Figure 1.10: An example showing thread divergence management within a warp, taken from [16]

Another specification of a warp is the state which can be READY, ACTIVE, WAITING, and FINISHED. A warp is in the ready state when it is idle and ready to be scheduled towards SPs, active while it is being executed, waiting when it is stopped, and waits for the other warps to reach a synchronization point and finally finished after passing the mentioned states.
Based on a previous research [17], the warp scheduler located inside this unit can be considered as the most critical unit inside a GPGPU. For this reason, the warp scheduler subunit is one of the sources of input signals taken into the designed performance monitoring unit in the current work.

1.4 GPGPU programming model

The GPU programming model introduced by NVIDIA is called **CUDA** (Compute Unified Device Architecture), which is actually an extension to C++ language. The normal thread-based programming is not being followed in this model and the programmer does

not need to be aware of graphic or parallel programming concepts. The programmer should just manage the GPU-related part of the code by a loop, and then, as already mentioned the execution will be done by the hardware using multiple parallel threads. This part of code which is being executed by the device (GPU) and is being defined as a function is known as **kernel**.
The software flow of GPGPU programming is given in the figure 1.11. At the first stage, the kernel will be compiled by NVidia Cuda Compiler or NVCC and the result will be a code like assembly codes named as parallel thread execution or PTX code. Finally, the PTX code is passed to the CUDA driver API and the result will be a binary CUDA code with the .cubin suffix.
It should be mentioned that although a PTX code is apparent to the user, it does not directly correspond to the actual instructions to be mapped to the GPU. There is an extra stage between converting PTX to cubin file which is creating another file named SASS. The SASS file represents the assembly code to be executed by the GPU. The main problem is that, due to marketing policies, this extra conversion stage is not apparent to the user and to be able to reach a SASS code we need to disassemble the CUDA binary using a tool named cuodjdump.

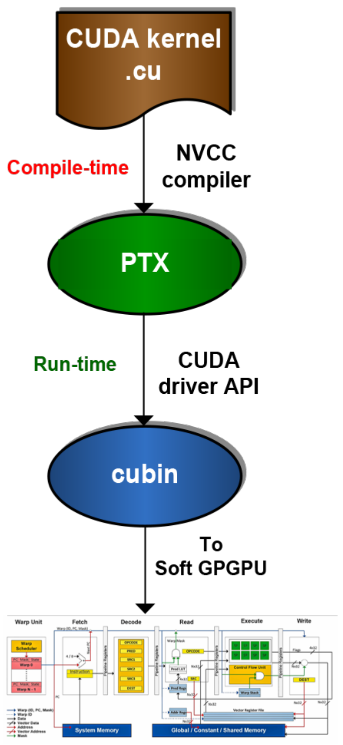

Figure 1.11: The software flow being seen by the user, adapted from [14]

In the current work, PTX defined keywords will be used for introducing the instructions and also the binary format of other predefined instructions will be taken into account.

1.5 Testing and fault detection

The massive complexity of GPGPU devices makes them prone to different types of faults, this may cause them to fail. Considering the increasing popularity of using GPGPUs in critical applications, any fault can result in catastrophic consequences. Since relatability and safety are mandatory requirements in many applications, providing the devices with an easy to use testing feature has been considered as an essential requirement and is studied in many works [11], [17]–[20]. Some of the testing methods applicable on GPGPUs will be discussed in the following.

1.5.1 BIST

A system including a BIST or built-in self-test feature is a system in which a self-test functionality is provided using specific units in charge of generating test patterns and analyzing the results gathered from the circuit under the test.
This idea is introduced with the aim of reducing the test costs by removing the need for using external test devices, reducing test time, accessing the internal units, and performing at-speed tests. Although the idea of including the test circuitry inside the chip has some disadvantages from the area and power consumption point of view, the mentioned advantages can cover these costs [21].
A simple BIST architecture is shown in the figure 1.12. As it can be seen this architecture includes, a unit to generate test patterns named TPG, a unit in charge of analyzing the results and declare the result as pass or fail, a test controller to change the normal operation mode into the test one by controlling the inputs toward the CUT and configuring it and other sub-blocks to follow the desired test procedure.

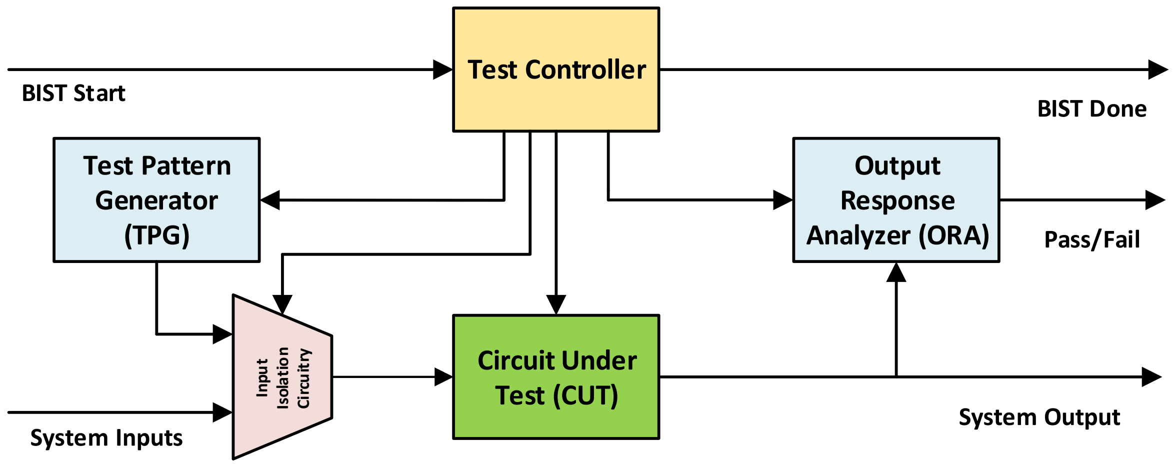

Figure 1.12: A simple BIST architecture, adapted from [22]

BIST method is studied and implemented to be used in a variety of products including microprocessors [23], [24] and can be considered as an option to be used for testing

GPGPUs as well but it should be mentioned that this option can be quite complex and dependent on the circuit characteristics.

1.5.2 Scan-Chain

Another possible option for providing a digital system with test functionality is using scan-based testing. Scan is of the most popular design for testability technique [25]. A basic scheme showing the implementation of this idea is given in the figure 1.13.

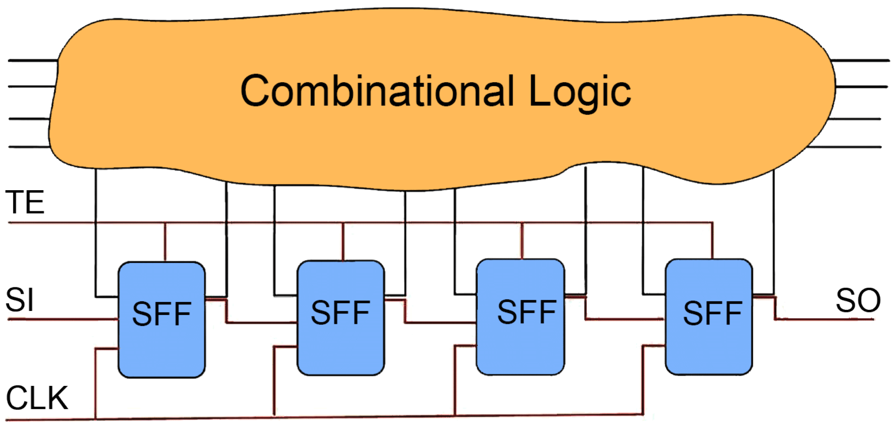

Figure 1.13: A simple scan chain scheme, adapted from [25]

The idea of the scan chain is based on replacing normal flip-flops (FFs) with scan flip-flops (SFFs). As shown in the figure 1.13 these flip-flops are connected together to create a chain. When the test enable signal (TE) is high these flip-flops create a shift register. In the test mode, the test pattern is inserted into the circuit using SI input and the gained results are shifted out through the SO output. The test program then compares the gathered results with the expected one to verify the correct behavior of the system.
This method is easy to implement, results in low pin overhead, and provides the ability to test sequential circuits, but it requires changing normal FFs into FFSs while it degrades the performance and increases the area. Although applying the scan method on GPUs is studied in some resources [26], [27], considering the mentioned disadvantages, clearly, applying this method on a GPGPU requires significant changes in the circuitry and can be quite complicated.

1.5.3 SBST

Software-Based Self-Testing or SBST is one of the most common test strategies for performing functional tests. This idea is basically introduced in 1980 [28] with the aim of providing different microprocessor architectures with an efficient testing method without degrading the performance and imposing high test expenses.
The SBST process is done by using processors native instruction set which causes elimination of the need for external test hardware. A typical SBST procedure is composed of

three main steps which are: downloading the test code into the memory, at-speed execution of the test code, and finally uploading the results into a memory to perform further analysis [29]. This method is easy to apply and its application on the FlixGrip GPGPU is already studied [18].

1.5.4 Performance monitoring unit

As a system became more complex (with the new technologies) its testing procedure (especially after being mounted on a board) became more difficult and complicated. The idea of using a hardware-based *Performance Monitoring Unit* (PMU) is considered an option to deal with this issue. Using a PMU to record different parameters in a processor is an effective method being vastly studied in many resources [30]–[35]. Nowadays, counter-based PMUs are present in all major processors and provide the user with different information such as the number of catch misses, memory latency, clock cycles, number of instructions executed, etc.
This option is easy to implement, flexible in target event selection and results in a very limited area and power overhead. Since a PMU can record a variety of events, by selecting specific events and targeting critical units, it is possible to perform tests and detect the probable hardware faults too. Such a unit can use counters to monitor the activities of some specific internal signals of the design and store the results. The recorded results then can be compared with the expected ones to identify the faults. Acting on the PMU unit can be done easily by including simple instructions in the code.
The described idea is the base of the current work and its implementation in the FlixGripPlus GPGPU is discussed in the next chapter.

Chapter 2

Proposed monitoring structure

In this chapter, the procedure of extending the FlexGripPlus GPGPU with the aim of providing it with an in-field test feature is discussed. To achieve this feature, considering the project requirements a counter-based performance monitoring unit is designed, signals with the highest index of toggle activities are chosen, the new required instruction to act on the designed unit is defined and finally, the hardware model is modified and the designed unit is included in it.

2.1 Performance monitoring unit design

As mentioned already, the main aim of this thesis work is to make the GPGPU model able to perform some software-based in-field tests by including a performance monitoring unit in charge of recording activities on 16 signals taken from different parts, the limitation of 16 is due to the limited number of bits dedicated to addressing the counter subunits in the instruction bit field, which is 4 (16 addresses). Obviously, each subunit will be able to track and record activities on one of the 16 specific vector or single-bit signals that can be freely chosen from the different parts of the whole processor. Since counter subunits are designed to monitor activities on a single-bit signal in the case of choosing vectors, a proper conversion should be performed on the vector input. To keep the unity of the design same procedure should be applied on single-bit signals as well. The unit takes four sets of inputs and has a 32-bit output that can be driven by one of the monitoring subunits (counters). These inputs and output are defined as:

- **ADDRESS**: The 4-bit input for addressing one of the 16 subunits.
- **COMMAND**: The 2-bit command signal coming from the main decoder unit of the processor for controlling the pre-chosen subunit.
- **INPUT ARRAY**: The set of 16 vectors with flexible length to be converted into 16 single-bit signals and recorded.
- **CLK**: The clock signal going through the counters and converter.
- **RESET**: The overall system reset signal to simultaneously reset the whole unit.

- **MONITOR-OUT**: The 32-bit output representing the stored values of the chosen subunit.

The ADDRESS, COMMAND, and RESET inputs of the unit are driven by the DECODER stage of the pipelined processor while the INPUT ARRAY is flexible and based on the monitoring plan can be taken from any part of the processor. Although each monitoring subunit can be reset separately by the decoder, a global reset signal is also included to reset the whole unit contents. The output of the unit will then be fed to the execution stage. The execution stage will then pass the results to the write stage to store the recorded content to pre-addressed general-purpose registers to let the user access it for further aims. This procedure's management is fully software-based, and the user can decide about it with simple codes.
Considering the project requirements, the mentioned functionality can be achieved by considering 5 blocks inside the performance monitoring unit which are:

- **DECODER**: This block is in charge of doing two main jobs. first, it will decode a 4-bit input address into a 16-bit signal. Secondly, decoding the given address, it will provide the addressed counter subunit with the desired commands taken from the processor's decoder. Obviously, the command bits going through the subunits which are not addressed will be kept at '00' which has no effect on them. In the case of having a global reset, the address decoding stage will be skipped, and all of the subunits will be rested.

- **CONVERTER**: Since the model is quite complex the granularity level of registers may be chosen instead of single-bit signals. In this case, a proper conversion should be done to match the raw inputs (vector or single-bit) with the counter's input. To keep the unity of design all the inputs will be taken to the converter and the mentioned action will be done on all the inputs. More detail about this block will be given in the further sections.

- **COUNTERS**: Set of 16 counting units which are including a synchronous counter with clear and enable bits plus a finite state machine to control them. This block will be discussed in more detail in the next section.

- **OUTPUT SELECTION MUX**: A simple 16 to 1 mux that selects the addressed subunits to drive the monitoring unit's output.

The overall structure of the performance monitoring unit is shown in figure 2.1.

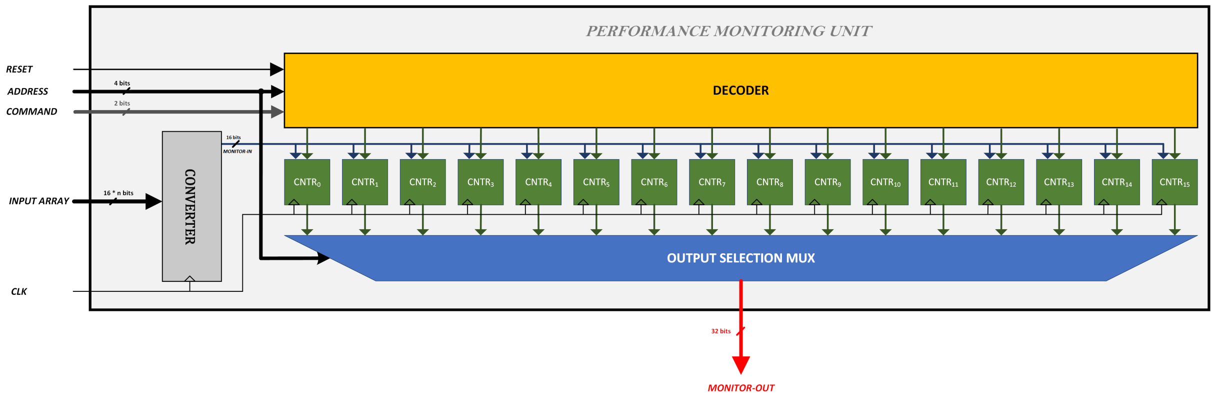

Figure 2.1: Overall structure of performance monitoring unit

2.1.1 Counter subunit

Based on the project requirements, asynchronous counters with enable and clear capability are chosen for recording the activities. Each counting subunit has a 32-bit asynchronous counter whose clear and enable bits are being controlled by a finite state machine-based controller. This structure is shown in the figure 2.2.

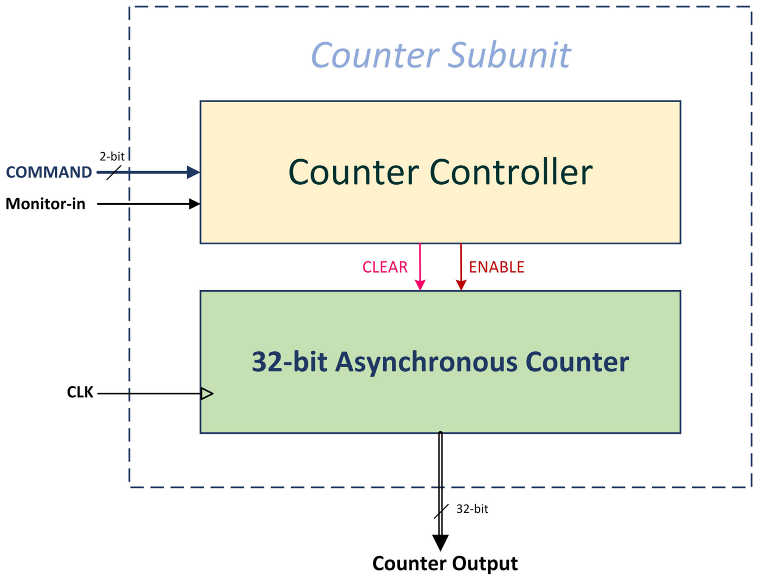

Figure 2.2: Overall scheme of the counter subunit

The FSM-based controller has three states: **RESET**, **COUNTING** and **IDLE**. Changing the states is done by the decoder unit which controls the monitor by a 2-bit command signal. These states can be described as:

- **RESET**: The initial state of all counters in which the counters are not enabled and their contents are all reset to 0. The command bits ”10” will bring the FSM to this state. The clear and enable bits will be 1 and 0. Although the reset of each counter subunit can be done separately when an overall processor reset is performed, the decoder will bring all of the counters simultaneously into this state.

- **COUNTING**: Counters in this state will become active and record the activities of their input by counting the number of clock cycles in which their inputs are at logic one. The command bits ”01” are considered for reaching this state. The clear bit in this state will be 0 while the enable bit will follow the monitor-in signal.
- **IDLE**: In this state, the counters will stop counting while keeping their contents to be read and delivered to the execution stage. This state can be reached using the command bits ”11”. In the IDLE state, the clear and enable bits are both being kept at 0. Reading the counters content will be performed at this state.

It should be mentioned that the command bit ”00” has no effect on the FSM; these bits are the default command bits given to the subunits while there is not any specific command.
In order to manage these states, the decoder stage design is updated with the commands (PMURST, PMUIDL, PMUCNT, PMUNOP) and subunit addresses are extracted by the processor decoder, the procedure of opcode selection and other instruction bits will be discussed in further sections.
Although the asynchronous counter is simpler the delay due to cascaded clocking can degrade the performance. This delay can be removed by using the synchronous counter in which clocking of all the flip flops is done at the same time [36].
The designed counter has clear and enable ability. The clear signal which is active-low will clear the content of the counter when it becomes equal to 0, while the enable signal will let the counter count the low to high edges on the clock signal when it is equal to 1, otherwise the counter will keep it is content constant. these signals as already mentioned are being controlled by an FSM controller. This configuration is illustrated in the figure 2.3.

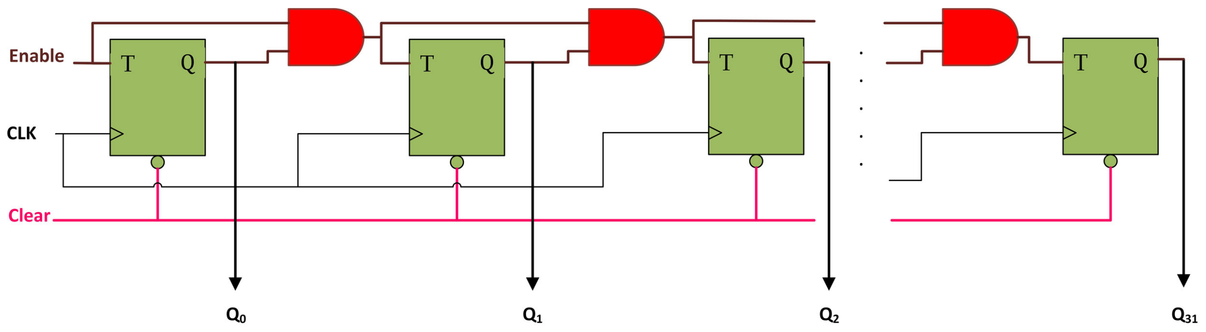

Figure 2.3: Synchronous counter with enable and clear capability

2.1.2 Converter

To deal with less complexity, the granularity level of registers instead of single bits can be targeted. To match the input vector with the single-bit input of the counters a kind

of conversion should be performed. This aim is gained by checking changes of the chosen signals at each clock cycle. The checking is done simply using an array of D Flip-flops constructing a register and a comparator. At each clock cycle, the comparator compares the current input and the previous one which is delayed by registers. Any changes will cause the output of the converter to become logic 1, while consistency will be declared by a 0 at the output.
In this way counter subunits are able to simply track the toggle activities of chosen inputs; in the case of dealing with the single bits in all inputs this conversion stage can be skipped and counters will just monitor the clock cycles in which inputs are at the high level. An input-output waveform sample of the converter is shown in the figure 2.4.

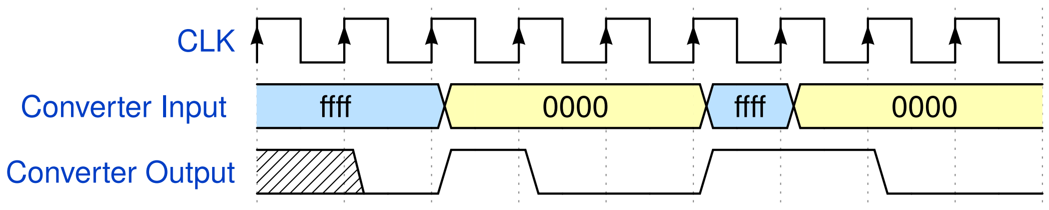

Figure 2.4: A sample of converter's waveforms

The described structure handling conversion is illustrated in figure 2.5.

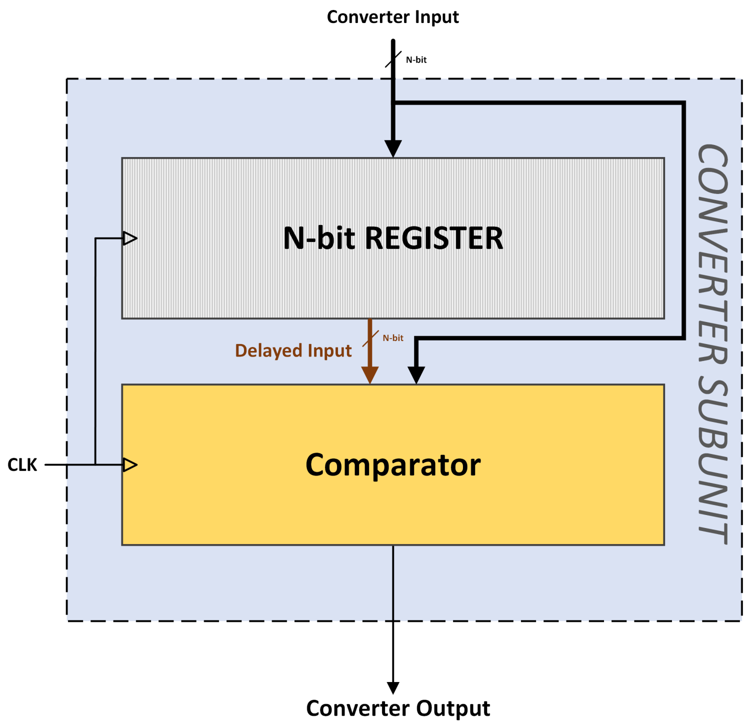

Figure 2.5: Overall scheme of the converter subunit

To reach 16 proper input bits, the conversion should be done on all the inputs. It should be mentioned that maintaining the monitoring policy the described conversion procedure should be applied to the single-bit inputs as well. The converter subunits are managed in an array shown in figure 2.6.

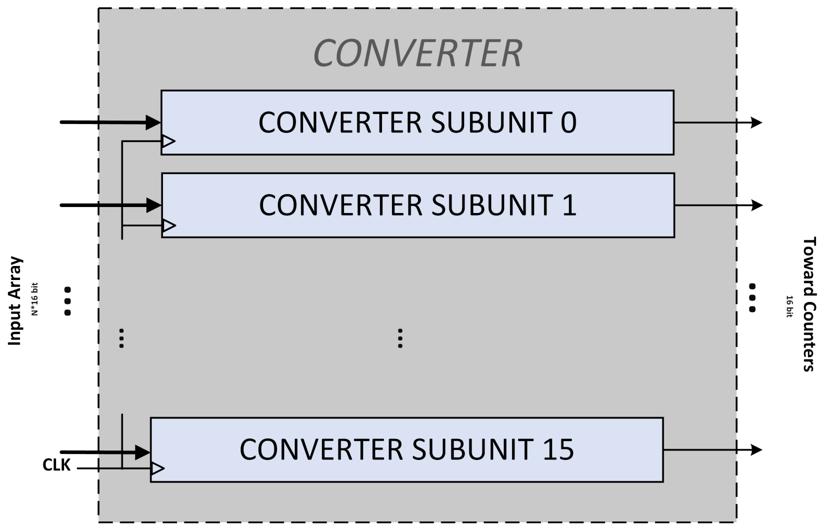

Figure 2.6: Converter unit scheme

2.2 Input signal selection

The input signal selection of the designed performance monitoring unit is done based on their toggle activities. This aim is achieved using the toggle coverage feature of Modelsim to trace the signals that change more than others in the model. For this aim different available benchmark codes are executed. The provided benchmarks are: edge detection, FFT, floating-point matrix multiplication, sorting, reduction, scalar vector production, transpose, vector addition (integer and floating), M3 (matrix multiplication using a tile optimization approach) and NN (nearest neighbour). After running the mentioned benchmarks, the most frequently changing signals with the highest rate of repeating are chosen. This criterion is chosen as a basic criterion to test the PMU, and more accurate studies should be performed on the model to choose inputs more accurately.
Since the model is quite complicated tracking signals from all parts is not feasible and they should be taken from the most critical parts. In this work, the focus is on the warp scheduler controller and execution unit which are being considered as the most critical parts of the GPGPU. To get a more comprehensive result, the selection is done by simulating various benchmark applications which are already available.
Clearly, after finding proper signals, considering the flexibility of the monitoring unit in terms of the bit length, the VHDL design of the converter unit should be modified to match the length of the inputs. This modification is quite simple and is limited to changing converter subunit inputs and PMU inputs. Since some signals are internal and not accessible in the streaming multiprocessor top entity, some test ports should be added to the desired units.

2.3 Adding management support to the performance monitor unit

To include the monitoring system in the processor the first stage after designing the unit and connecting the input signals, is extending the model to be able to execute the new related instruction. A single instruction is enough to deal with the monitoring unit; this instruction will be used for commanding the monitoring subunits, accessing the contents of subunits, and pass them to the desired general-purpose registers.

2.3.1 PMEVENT instruction

Since the identification of the operative code of the instruction to follow the CUDA standards is complex, and programming tools maintain this operative code partially hidden, the only possible option to implement the PMU related instruction is to choose an available and free opcode. The PTX instruction introduced by Nvidia [37] which triggers the performance monitor is *pmevent*. The same keyword will be used to define our instruction, a specific free opcode will be chosen for the mentioned instruction and further bits will be managed in a way to make the decoder able to command each of the pre-described counter subunits.
The mentioned instruction can be implemented with 32 or 64 bits. The actual required number of bits for extending the model are four bits for the opcode, four bits for addressing the subunits, two bits for controlling the addressed subunit, seven bits for addressing the desired general-purpose register as the destination to store the results, one bit for distinguishing if the instruction is 64 or 32 bits long, and finally one bit to know if the instruction is normal or flow control one. Since the total number of needed bits is equal to 19, this instruction can be implemented using the 32-bit long format. Although it is possible to control the monitoring unit with different instructions, in order to save other opcodes for further applications and reducing the complexity, only one instruction (opcode) with different options (command bits) is considered for the mentioned functionalities.
Based on the previous work done with the aim of extending the design there are 5 free opcodes available on the basic version of processor [15]. These bits are extracted by checking the simulation after adding 16 new random opcodes, any stall during simulation indicates that the added opcode is not included in the basic model and is free to be used. The available opcodes are reported as 1110-1100-1011-1001-0111. Although all of these opcodes are free in the basic version, to prevent possible errors, the taken opcodes by the previous work which are 1001 and 1011 will be skipped and **0111** will be selected as an instruction opcode in the current work.
The suggested format of instruction bits format, considering the format of predefined ones is given in the table 2.1.

Opcode	Not Used	Address	Command	Not Used	GPR Address	Normal(0)	32bit Long(0)
4bit	11bit	4bit	2bit	2bit	7bit	1bit	1bit
31-28	27-17	16-13	12-11	10-9	8-2	1	0

Table 2.1: PMEVENT instruction format

The mnemonic of the defined instruction is:

PMEVENT (destination counter), (destination register), (command)

The destination register address bits, in the case of performing activation and reset, should not have any effect and will be skipped by the decoder. Some SASS format examples of the described instruction are given in the following:
PMEVENT CNT0, CNT (70000800)
PMEVENT CNT0, RST (70001000)
PMEVENT CNT0, R5, IDL (70001814)
The table 2.2 summarises the suggested format and gives some examples.

Bit(s)	Mnemonics	Commentary
0	instr_is_long	0=32 bit long (Default), 1=64 bit long
1	instr_is_flow	0=Normal ins (Default), 1=System ins (Flow Control)

(8-2) 7	destiny_register	Register Address:
		R0=0000000,
		R1=0000001,
		R5=0000101, ...

(10-9) 2	not used	

(12-11) 2	command	Activate=11,
		Idle=01,
		Reset=10,
		No Effect=00

(16-13) 4	destination_counter	Destination Counter Address:
		CNT0=0000,
		CNT8=1000,
		CNT11=1011 ...

(27-17) 11	not used	
(31-28) 4	instruction_opcode	PMEVENT=0x7

Table 2.2: PMEVENT instruction samples

2.3.2 Hardware modifications

After designing the performance monitoring unit, choosing input signals, and defining the new instruction, it can be included in the hardware model. To do this, the PMU is located as a component in the streaming multiprocessor. The Decoder stage is modified by adding the new opcode and defining new ports and signals related to the PMU including the address of the counters, command bits, and reset. The input signals are taken from the target units in the top entity, in the case of dealing with internal signals, new test ports are defined to make them accessible in the top entity. The output of the PMU is treated similarly to execution subunits. in the case of having PMU read operation, the Execution stages output will be driven by the PMU which is already in the idle state to keep the output stable. Since no read operation is required the Read stage is almost untouched. The Write stage is also kept untouched. To ease the further changes, all the modifications in the basic model are addressed by proper comments. The GPGPU structure shown in the figure 1.6 after including the PMU inside it is shown in the figure 2.7.

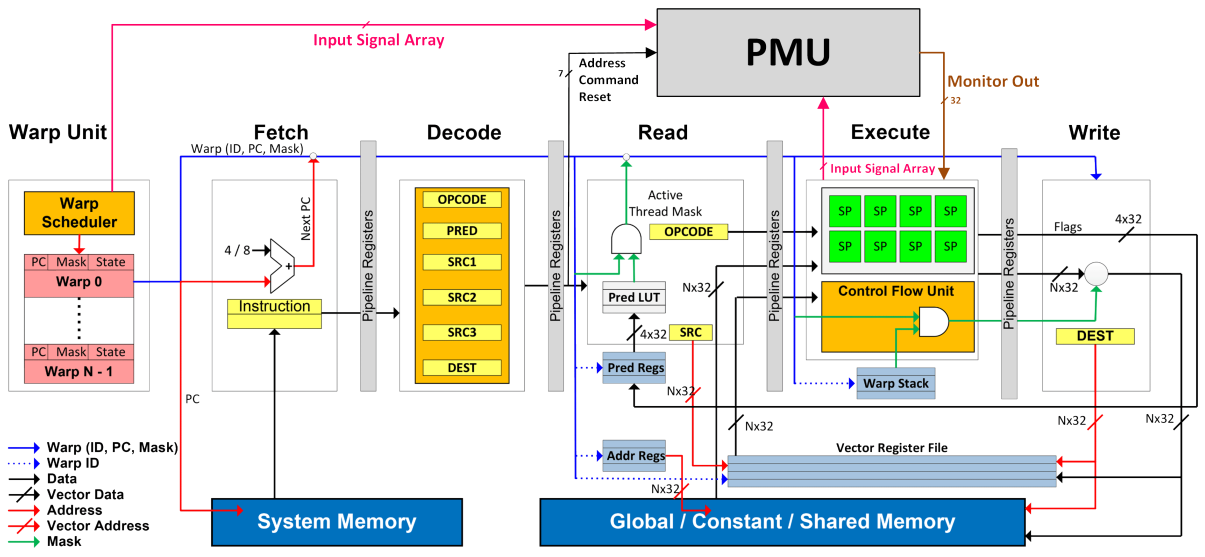

Figure 2.7: The overall structure of the GPGPU after including the PMU

Chapter 3

Experimental results

In this chapter, the simulation results of the designed PMU and modified GPGPU are presented. After the simulation, a synthesis is done to evaluate the hardware characteristics of the PMU. Same work is done on both untouched GPGPU and modified one to get a vision about changes in timing, power, and area characteristics.

3.1 Performance monitoring unit testing

After defining the requirements of the design, and describing the model in a hardware description language (VHDL), the verification of the PMU before including it in the model is accomplished. This aim is achieved by using a test bench in charge of providing commands, addresses, and input signals to the design and checking its behavior. It should be mentioned that the simulation is accomplished using the Modelsim tool used Figure 3.1 shows the simulation results of the PMU.

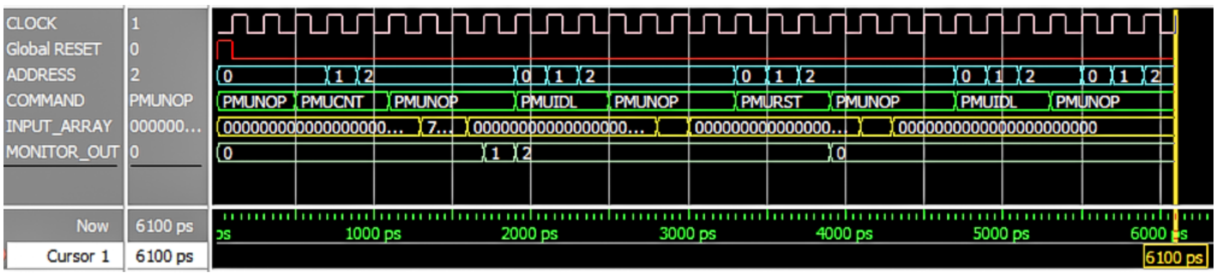

Figure 3.1: PMU simulation results

The of the shown signals are given in section 2.1 and the related commands are introduced in subsection 2.1.1. During this simulation after an overall reset (shown in red), the counters with the addresses of 0, 1, and 2 (shown in cyan) are activated by the command PMUCNT, later they count two toggling activities on their input (illustrated by yellow), the mentioned value by running the command PMUIDL is given at output which is the MONITOR_OUT signal, since the counters are in the idle stage they do not record later toggling activities, at the next stage the local reset command which is PMURST is sent to the addressed counters, and again PMUIDL command is issued, observing the value of 0 at output verifies the correct behavior of the system. The command PMUNOP has no effect on the system and is defined to cover all other unrelated instructions.

3.2 Performance monitoring unit synthesis results

After verification of the design, synthesis and mapping are performed. In this step, the synthesis tool converts the RTL description to the gate level netlist (synthesis). The generated netlist is then provided with timing, area, and power characteristics using the cells present in the library (mapping). This operation is done using the Synopsis Design Vision software. The chosen library is 15nm FinFET-based Open Cell Library (OCL) [38]. Since the modern GPUs are implemented at 22 and 7 nm tech, this library is more representative of real ones.
The first characteristic to be discussed is static timing analysis. The tool reports the result of this analysis using three parameters which are slack, data arrival time, and data required time. Data arrival time represents the time required for the data to travel through the data path. Data Required Time is a parameter showing the time taken for the clock signal to go through the clock path. Finally, slack is the difference between the required and arrival time. These parameters are being checked in the case of the longest path (critical path) to guarantee a safe margin for the worse case. A positive slack will be reported with the keyword MET, while in the case of a negative one the keyword VIOLATED will be seen in the report. The timing report summary of the PMU is given in the table 3.1.

Parameter	Value (ps)
data required time	1991.95
data arrival time	-205.25
slack (MET)	1,786.70

Table 3.1: Timing characteristics of the PMU

The resulted values are based on the target clock frequency of 500MHz (clock period of 2000 ps) which is the working frequency of the chosen GPGPU. The achieved slack proves that the PMU is much faster than other units and will not contribute to the critical path.
The next parameter to be analyzed is the area of the design. The values being reported by the tool are library dependent. The area characteristics of the designed unit are given in the table 3.2.

Parameter	Value
Combinational area	$574.685\,199\,\mu m^2$
Buf/Inv area	$45.416\,450\,\mu m^2$
Noncombinational area	$696.778\,728\,\mu m^2$
Total cell area	$1{,}271.463\,927\,\mu m^2$
Number of cells	2,878

Table 3.2: Area characteristics of the PMU

The next important parameters to be discussed are the power-related ones. The used synthesis tool reports power characteristics including dynamic (internal and switching) and leakage ones. The dynamic power represents the power dissipated while there is a switching event, while the leakage power is due to non-zero reverse leakage and sub-threshold currents even if the transistor is not performing any switching activity [39]. Dynamic power is reported in two parts, the switching power in which the power is being drawn to charge output capacitance and the internal power in which power dissipates due to changes of input without affecting the output. The extracted power characteristics of the PMU can be seen in table 3.3.

Parameter	Value
Switch power	$1.66 \times 10^{-2}\,\mathrm{mW}$
Int power	$0.515\,\mathrm{mW}$
Leak power	$4.57 \times 10^{7}\,\mathrm{pW}$
Total power	$0.577\,\mathrm{mW}$

Table 3.3: Power characteristics of the PMU

After verifying the correct behavior of the designed PMU and performing synthesis, we can do the same steps on the modified GPGPU (PMU included) to fully verify the design and accomplish characteristics by doing synthesis. The mentioned procedure is discussed in the following.

3.3 Testing of the modified GPGPU

To test the chosen model, a file of "TP_instructions.vhd" is already included in the design files, which contains the decoded SASS instruction bits. The described three modes of PMEVENT instruction are added to one of the provided applications, the modified version of this application can be seen in appendix A. To prevent any disruption with the main code, the activation of the counters is issued at the beginning, and reading the contents is done at the end and before the global reset.
As the first step of the simulation, all of the 16 counters are activated using the PMUCNT command, then the main application code (which is an integer addition) is included, and at the end, some counters are commanded to become idle by the command PMUIDL or reset with the command PMURST. Since in the mentioned application only 8 registers are available for each thread, reading contents of 5 counters (number 0, 1, 2, 4, and 12) is done and one of them (number 10) is commanded to perform a local reset just to test this functionality. At the end of the code, a global reset signal is triggered which results a reset of all counters inside the PMU.
Figure 3.2 shows the content of registers related to a single thread and PMU counters at the end of running the given modified application. The accomplished result verifies

correct behavior of the extended model.

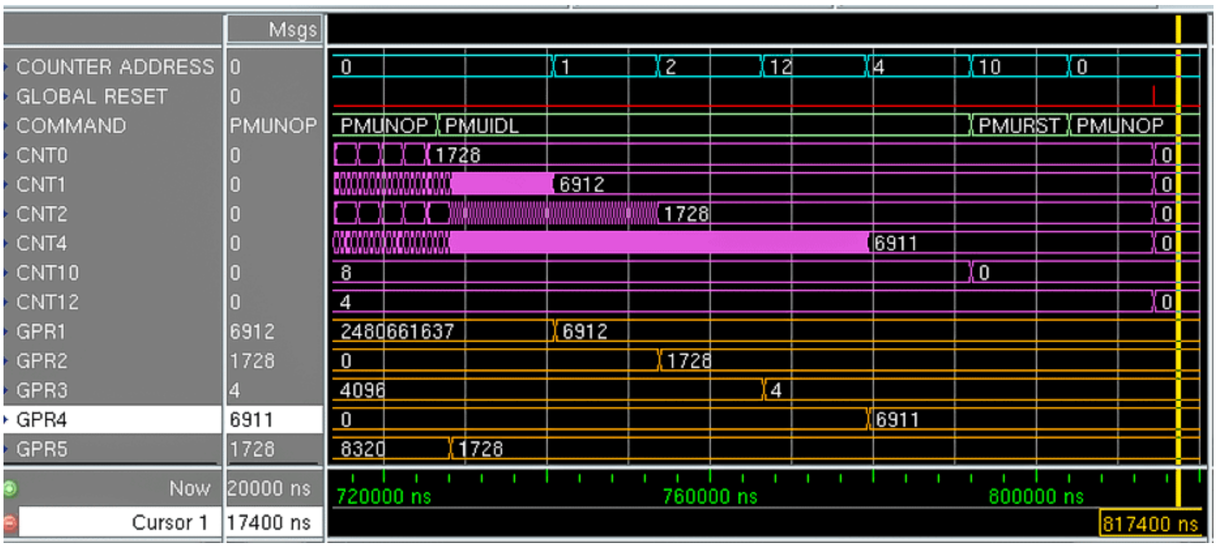

Figure 3.2: GPGPU model simulation result

3.4 Synthesis results of the modified GPGPU

After verifying the correct behavior of modified GPGPU, we can synthesize it as the next stage of design. In this stage, similar to the PMU synthesis, we can achieve an idea about some important characteristics like power consumption, occupied area, and the most important one which is timing and critical-path related data, and compare the generated reports with the basic version characteristics.
Table 3.4 shows the achieved timing characteristics of both basic and modified GPGPUs.

	Basic version	Modified version
Data required time (ps)	1,992.06	1,992.06
Data arrival time (ps)	-1,640.04	-1,640.16
Slack (MET)	352.02	351.90

Table 3.4: Timing characteristics of the models

The gathered results are both showing the same values which are related to a part of the processor not related to the PMU. By checking the details of the report, it can be seen that the added performance monitoring unit does not contribute to the critical path of the design and is safe from this point of view.
The next characteristic to be analysed is the area. The area occupation characteristics are reported in combinational, Buf/Inv, and sequential related area, these characteristics are reported in the table 3.5:

	Basic version	Modified version	Increment
Combinational area	170,800.694 922 µm^2	171,398.088 341 µm^2	0.34%
Buf/Inv area	25,876.464 509 µm^2	25,948.570 496 µm^2	0.28%
Noncombinational area	221,25.624 611 µm^2	221,644.349 966 µm^2	0.28%
Total cell area	391,826.319 533 µm^2	393,042.438 306 µm^2	0.31%
Number of cells	773,721	776,577	0.36%

Table 3.5: Area characteristics of the models

Clearly, the area is increased but the overall increment rate is limited to only 0.31% of the basic design and can be almost ignored.
In addition to the mentioned characteristics, we can get a report about the power characteristics of the designs which are given in the table 3.6:

	Basic Version	Modified Version	Increment
Switch Power	3.435 mW	3.437 mW	0.06%
Int Power	165.036 mW	165.473 mW	0.26%
Leak Power	1.26×10^{10} pW	1.26×10^{10} pW	0.00%
Total Power	181.076 mW	181.547 mW	0.26%

Table 3.6: Power characteristics of the models

Comparing the characteristics, it is apparent that the power contribution of the PMU to the overall value is very low and can be ignored.
To better address the modification costs, the decoding stage (as the most changed one) is synthesized separately. The following tables are representing the changes of the mentioned pipeline stage.

	Basic version	Modified version
Data required time (ps)	1,991.77	1,990.44
Data arrival time (ps)	-304.04	-216.89
Slack (MET)	1,687.73	1,773.55

Table 3.7: Timing characteristics of the decoder

The changes in timing characteristics are due to different mapping on the two versions by the tool. Since the decoder does not contribute to the critical path this change has no effect on the overall performance.

	Basic version	Modified version	Increment
Combinational area	$301.006\,852\,\mu m^2$	$310.640\,644\,\mu m^2$	3.2%
Buf/Inv area	$63.799\,298\,\mu m^2$	$64.978\,946\,\mu m^2$	1.85%
Noncombinational area	$458.784\,752\,\mu m^2$	$465.174\,512\,\mu m^2$	1.39%
Total cell area	$759.791\,604\,\mu m^2$	$775.815\,156\,\mu m^2$	2.11%
Number of cells	1,604	1,640	2.24%

Table 3.8: Area characteristics of the decoder

	Basic Version	Modified Version	Increment
Switch Power	3.87×10^{-2} mW	4.04×10^{-2} mW	4.39%
Int Power	0.376 mW	0.380 mW	1.06%
Leak Power	2.62×10^{7} pW	2.67×10^{7} pW	1.91%
Total Power	0.441 mW	0.447 mW	1.36%

Table 3.9: Power characteristics of the decoder

The gathered results from analyzing the decoder show that the change rate is low and the main changes to the design are due to the PMU unit itself and not the changes of the decoder.
As an overall result, the timing, power, and area of the modified version are almost the same as the basic one and the costs of the added unit are almost negligible.

Chapter 4

Conclusion

In this thesis work, the impacts of adding an **in-field test** feature to a **GPGPU** (General-Purpose Graphics Processing Unit) model are evaluated. The chosen GPGPU is an academic model designed by researchers at the University of Massachusetts Amherst.
Considering the complexity of GPUs, detecting the probable hardware defects during the usage of the processor (in-field test) is a critical requirement. To obtain this aim a flexible and easy to implement option which is extending the model by including a monitoring unit inside it is presented. The basic model after this work is able to monitor toggle activities of sixteen (vector or single-bit) signals and store the results into general-purpose registers. This operation is being managed by a single assembly instruction. Monitoring of the signals is done by including a **Performance Monitoring Unit (PMU)** into the design. This unit consists of four main sub-blocks which are:

1. A decoder to deliver commands to the desired counting subunit.

2. A converter taking a group of signals from different parts of the model as input and convert their toggles activities into logic 1 to be monitored.

3. An array of 16 monitoring subunits including 32bit synchronous **counter**s plus a finite state machine controller to command them.

4. A 32 bit sixteen to one multiplexer in charge of driving the output with the content of the desired subunit.

The input signals of the PMU are taken from two critical parts of the processor which are warp scheduler and execution stage. For the aim of the current work, the signals with more toggling activities are chosen.
After designing the mentioned unit, including the design into the model is done. The first step of this modification is to define an instruction matching with the basic GPGPU model and the designed PMU, which is mainly about choosing an empty opcode and also deciding about the usage of bit fields. Following this step, the DECODER of the processor is modified to make it able to recognize the new opcode and command the PMU and other pipeline stages to act in the desired way. The reading procedure of PMU contents is done similarly to ALU subunits and its output is connected to the EXECUTION stage of the pipeline, the contents of the addressed counting subunit are then forwarded to the WRITE stage. Similarly, the address of the destination register is also extracted by the

DECODER and forwarded to this stage. To ease the modifications, the READ stage of the pipeline is almost kept untouched.
After finishing the VHDL design, functionality verification of the developed PMU and modified GPGPU is done, The extended GPGPU is tested by using a modified version of a pre-provided benchmark code, in which the new test instructions are included in the given file and a simulation using the **ModelSim** software is done, checking the signals in the design and also the final stored values, and comparing them with the expected ones proves the correct behavior of the modified model.
Following the testing phase, the PMU and the new GPGPU model are synthesized using **Synopsys Design Vision** software. During this phase, after analyzing the blocks and elaborating the designs, area, timing, and power reports are extracted. The extracted results of the new GPGPU are then compared with the results of the basic untouched model to get a vision of the area and power costs.
The final result proves that the included PMU has no contribution in the critical path and the resulted power and area costs are almost negligible. Since the major issues in the design of modern complex embedded systems are reliability and optimization for the mentioned characteristics, the studied method can be considered as a proper option to be implemented in these systems.
While the results achieved by this thesis work prove that a counter-based performance monitoring unit is a good option to be used in complex systems, future research may move toward checking the effectiveness of using the designed unit for fault analysis. The suggested instruction format has 13 free bits which can be used to develop a more complex PMU system. As a suggestion future work may extend the number of input signals and use the free bits as selection bits (of some MUXs) for targeting more signals to be monitored by the PMU. Finding a more accurate criterion for choosing the input signals can be considered as an important topic for future work, too.

Appendix A

Verification and validation program for the PMU in the GPGPU

The "TP_instructions.vhd" file includes the assembly instruction (.SASS) bits in hex which are supported by the FlexGrip-Plus model. [12]. The following code represents the code executed to achieve the results given in the figure 3.2.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity TP_instructions is
  port(
    instruction_pointer_in  : in  integer;
    num_instructions_out    : out integer;
    instruction_out         : out std_logic_vector(31 downto 0)
  );
end TP_instructions;

architecture arch of TP_instructions is
  constant TP_INSTRUCTIONS : integer := 42;

begin
  num_instructions_out <= TP_INSTRUCTIONS;

  process(instruction_pointer_in)
  begin
    case instruction_pointer_in is
-- PMU ACTIVATION
when 0  => instruction_out <= x"70000800";    -- PMEVENT CNT0, CNT;
when 1  => instruction_out <= x"70002800";    -- PMEVENT CNT1, CNT;
when 2  => instruction_out <= x"70004800";    -- PMEVENT CNT2, CNT;
when 3  => instruction_out <= x"70006800";    -- PMEVENT CNT3, CNT;
when 4  => instruction_out <= x"70008800";    -- PMEVENT CNT4, CNT;
when 5  => instruction_out <= x"7000a800";    -- PMEVENT CNT5, CNT;
when 6  => instruction_out <= x"7000c800";    -- PMEVENT CNT6, CNT;
when 7  => instruction_out <= x"7000e800";    -- PMEVENT CNT7, CNT;
when 8  => instruction_out <= x"70010800";    -- PMEVENT CNT8, CNT;
when 9  => instruction_out <= x"70012800";    -- PMEVENT CNT9, CNT;
when 10 => instruction_out <= x"70014800";    -- PMEVENT CNT10, CNT;
when 11 => instruction_out <= x"70016800";    -- PMEVENT CNT11, CNT;
when 12 => instruction_out <= x"70018800";    -- PMEVENT CNT12, CNT;
when 13 => instruction_out <= x"7001a800";    -- PMEVENT CNT13, CNT;
when 14 => instruction_out <= x"7001c800";    -- PMEVENT CNT14, CNT;
when 15 => instruction_out <= x"7001e800";    -- PMEVENT CNT15, CNT;
----------------
when 16 => instruction_out <= x"10004205";    -- MOV.U16 R0H, g [0x1].U16;
when 17 => instruction_out <= x"0023c780";
when 18 => instruction_out <= x"a0000005";    -- I2I.U32.U16 R1, R0L;
when 19 => instruction_out <= x"04000780";
when 20 => instruction_out <= x"60014c01";   -- IMAD.U16 R0, g [0x6].U16, R0H, R1;
when 21 => instruction_out <= x"00204780";
when 22 => instruction_out <= x"30020009";   -- SHL R2, R0, 0x2;
when 23 => instruction_out <= x"c4100780";
when 24 => instruction_out <= x"2102e800";   -- IADD32 R0, g [0x4], R2;
when 25 => instruction_out <= x"2102ea0c";   -- IADD32 R3, g [0x5], R2;
when 26 => instruction_out <= x"d00e0005";   -- GLD.U32 R1, global14[R0];
when 27 => instruction_out <= x"80c00780";
when 28 => instruction_out <= x"d00e0601";   -- GLD.U32 R0, global14[R3];
when 29 => instruction_out <= x"80c00780";
when 30 => instruction_out <= x"20008204";   -- IADD32 R1, R1, R0;
when 31 => instruction_out <= x"2102ec00";   -- IADD32 R0, g [0x6], R2;
when 32 => instruction_out <= x"d00e0005";   -- GST.U32 global14[R0], R1;
when 33 => instruction_out <= x"a0c00781";
-- PMU READ and WRITE
when 34 => instruction_out <= x"70001814";   -- PMEVENT CNT0, R5, IDL;
when 35 => instruction_out <= x"70003804";   -- PMEVENT CNT1, R1, IDL;
when 36 => instruction_out <= x"70005808";   -- PMEVENT CNT2, R2, IDL;
when 37 => instruction_out <= x"7001980c";   -- PMEVENT CNT12, R3, IDL;
when 38 => instruction_out <= x"70009810";   -- PMEVENT CNT4, R4, IDL;
when 39 => instruction_out <= x"70015000";   -- PMEVENT CNT10, RST;
------------
when 40 => instruction_out <= x"30000003";   -- RET
when 41 => instruction_out <= x"00000780";
      when others => null;
    end case;
  end process;

end arch;
```

Bibliography

[1] S. H. Roosta, "Data parallel programming," in *Parallel Processing and Parallel Algorithms*, Springer, 2000, pp. 477–499.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, 2016.

[3] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[4] NVIDIA Corporation, *CUDA C++ programming guide*, Version 11.1, 2020. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[5] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, "General-purpose graphics processor architectures," *Synthesis Lectures on Computer Architecture*, vol. 13, no. 2, pp. 1–140, 2018.

[6] D. Hallmans, K. Sandström, M. Lindgren, and T. Nolte, "Gpgpu for industrial control systems," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, 2013, pp. 1–4. DOI: `10.1109/ETFA.2013.6648166`.

[7] M. C. Díaz, F. A. González, and R. Ramos-Pollan, "Accelerating common machine learning algorithms through gpgpu symbolic computing," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 387–391. DOI: `10.1109/ColumbianCC.2015.7333450`.

[8] A. Alic, S. Visan, and R. Potolea, "Towards fast bioinformatics algorithms: Benchmarking the gpgpu," in *2011 10th International Symposium on Parallel and Distributed Computing*, 2011, pp. 258–261. DOI: `10.1109/ISPDC.2011.45`.

[9] B. He, H. P. Huynh, and R. G. S. Mong, "Gpgpu for real-time data analytics," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 945–946. DOI: `10.1109/ICPADS.2012.156`.

[10] K. Andryc, M. Merchant, and R. Tessier, "Flexgrip: A soft gpgpu for fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230–237. DOI: `10.1109/FPT.2013.6718358`.

[11] J. E. R. Condia, B. Du, M. Sonza Reorda, and L. Sterpone, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, p. 113 660, 2020, ISSN: 0026-2714. DOI: `https://doi.org/10.1016/j.microrel.2020.113660`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0026271419307978`.

[12] J. E. R. Condia, *Flexgripplus: An open-source gpgpu model for reliability evaluation and simulation*, `https://github.com/Jerc007/Open-GPGPU-FlexGrip-`, 2013.

[13] S. Costa, "Study and development of a VHDL infrastructure for signals probing of GPGPU architectures," M.S. thesis, Politecnico di torino, Italy, 2020.

[14] M. Merchant, "Testing and validation of a prototype gpgpu design for fpgas," M.S. thesis, University of Massachusetts Amherst, 2013.

[15] P. Narducci, "Reconfigurable solutions to increase GPGPUs reliability," M.S. thesis, Politecnico di torino, Italy, 2020.

[16] Brian Railing, Nathan Beckmann, *Lecture notes in parallel computer architecture and programming*, 2021. [Online]. Available: `http://www.cs.cmu.edu/~418/`.

[17] B. Du, J. E. R. Condia, M. Sonza Reorda, and L. Sterpone, "About the functional test of the gpgpu scheduler," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, 2018, pp. 85–90. DOI: `10.1109/IOLTS.2018.8474174`.

[18] S. Di Carlo, J. E. R. Condia, and M. Sonza Reorda, "An on-line testing technique for the scheduler memory of a gpgpu," *IEEE Access*, vol. 8, pp. 16 893–16 912, 2020.

[19] J. E. R. Condia and M. Sonza Reorda, "Testing permanent faults in pipeline registers of gpgpus: A multi-kernel approach," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, 2019, pp. 97–102.

[20] J. E. R. Condia, F. A. Da Silva, S. Hamdioui, C. Sauer, and M. Sonza Reorda, "Untestable faults identification in gpgpus for safety-critical applications," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2019, pp. 570–573. DOI: `10.1109/ICECS46596.2019.8964677`.

[21] E. J. McCluskey, "Built-in self-test techniques," *IEEE Design Test of Computers*, vol. 2, no. 2, pp. 21–28, 1985. DOI: `10.1109/MDT.1985.294856`.

[22] C. E. Stroud, *A designer's guide to built-in self-test*. Springer Science & Business Media, 2006, vol. 19.

[23] P. Parvathala, K. Maneparambil, and W. Lindsay, "Frits - a microprocessor functional bist method," in *Proceedings. International Test Conference*, 2002, pp. 590–598. DOI: `10.1109/TEST.2002.1041810`.

[24] P. P. Gelsinger, "Design and test of the 80386," *IEEE Design Test of Computers*, vol. 4, no. 3, pp. 42–50, 1987. DOI: `10.1109/MDT.1987.295165`.

[25] S. Bhunia and M. Tehranipoor, *Hardware security: a hands-on learning approach*. Morgan Kaufmann, 2018.

[26] P. Kumar Datla Jagannadha, M. Yilmaz, M. Sonawane, S. Chadalavada, S. Sarangi, B. Bhaskaran, and A. Abdollahian, "Advanced test methodology for complex socs," in *2016 IEEE International Test Conference (ITC)*, 2016, pp. 1–10. DOI: `10.1109/TEST.2016.7805857`.

[27] A. Sanghani, B. Yang, K. Natarajan, and C. Liu, “Design and implementation of a time-division multiplexing scan architecture using serializer and deserializer in gpu chips,” in *29th VLSI Test Symposium*, 2011, pp. 219–224. DOI: 10.1109/VTS.2011.5783724.

[28] Thatte and Abraham, “Test generation for microprocessors,” *IEEE Transactions on Computers*, vol. C-29, no. 6, pp. 429–441, 1980. DOI: 10.1109/TC.1980.1675602.

[29] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, “Microprocessor software-based self-testing,” *IEEE Design Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010. DOI: 10.1109/MDT.2010.5.

[30] C. Roth, F. Levine, and E. Welbon, “Performance monitoring on the powerpc 604 microprocessor,” in *Proceedings of ICCD ’95 International Conference on Computer Design. VLSI in Computers and Processors*, 1995, pp. 212–215. DOI: 10.1109/ICCD.1995.528812.

[31] M. Lei, T. .-Y. Yin, Y. .-C. Zhou, and J. Han, “Highly reconfigurable performance monitoring unit on risc-v,” in *2020 IEEE 15th International Conference on Solid-State Integrated Circuit Technology (ICSICT)*, 2020, pp. 1–3. DOI: 10.1109/ICSICT49897.2020.9278263.

[32] Jihong Kim and Yongmin Kim, “Performance monitoring and tuning for a single-chip multiprocessor digital signal processor,” in *Proceedings of 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, ICA/sup 3/PP ’96*, 1996, pp. 76–83. DOI: 10.1109/ICAPP.1996.562860.

[33] C. Woralert, J. Bruska, C. Liu, and L. Yan, “High frequency performance monitoring via architectural event measurement,” in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 114–122. DOI: 10.1109/IISWC50251.2020.00020.

[34] S. Eranian, “What can performance counters do for memory subsystem analysis?” In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’08)*, 2008, pp. 26–30.

[35] W. Lindsay, E. Sanchez, M. Sonza Reorda, and G. Squillero, “Automatic test programs generation driven by internal performance counters,” in *Fifth International Workshop on Microprocessor Test and Verification (MTV’04)*, 2004, pp. 8–13. DOI: 10.1109/MTV.2004.5.

[36] Z. G. Vranesic and S. Brown, *Fundamentals of digital logic with VHDL design*, 3rd. McGraw Hill, 2009, ISBN: 978–0–07–352953–0.

[37] N. C. Team *et al.*, “Nvidia compute ptx: Parallel thread execution,” *ISA version*, vol. 1.4, 2009.

[38] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, 2015, pp. 171–178.

[39] Y. Leblebici and S.-M. Kang, *CMOS digital integrated circuits: analysis and design*. McGraw-Hill, 1996.