POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Degree Thesis

Reconfigurable solutions to increase GPGPUs reliability

Advisor
Prof. Sonza Reorda Matteo
Co - Advisors
Prof. Sterpone Luca
Dr. Rodriguez Condia Josie Esteban
Dr. Du Boyang

Candidates
Pierpaolo Narducci

March 2020

Summary

Nowadays, General Purpose Graphic Processing Units (GPGPUs) are effective solutions for high-demanding data processing applications. They are used in a lot of application fields like multimedia and gaming, but they started to be used in safety critical applications such as automotive, where reliability plays a significant role.

The aim of this thesis is to propose different implementations to increase fault mitigation for the reference FlexGrip model, which represents a simplified version of the NVIDIA GPU architecture. These solutions are not extensively included in GPGPUs, due to the limited reliability requirements of the applications they were originally intended for.

Three solutions have been developed and are presented in the thesis document:

- A Dynamically configurable self-repairing (BISR) mechanism aimed at reducing the impact of permanent faults in the Scalar Processor (SP) cores in GPGPUs. This first solution mechanism is based on spare SP modules that can be used to replace a possible faulty SPs when a fault affecting it is detected. In this architecture there are some cold stand-by modules (Spare SPs, or SSPs) in parallel with the existing SPs. Two switching units, based on meta-crossbar structures, targeting the data-path input and output interconnections in the SPs are used. An instruction specifically created allows to control the faulty SP and substitute it with a SSP.
This method is flexible because it does not require any change in the application software. Experimental results show that the solution introduces a moderate area overhead.
This strategy seems particularly suitable for long-term missions since it allows mitigating the effects of fault accumulation in the SP cores.

- A Dynamic duplication with comparison (DDWC) mechanism intended to harden the Scalar Processor units in GPGPUs.
The second solution mechanism targets the detection of permanent faults that may arise inside the SPs. The architecture has one additional SP unit to compute the same operations of a selected SP. A reconfiguration instruction is used to dynamically select the target SP to be monitored.

Experimental results show that the proposed mechanism introduces a limited area overhead while it provides a significant increase in the in-field fault detection capabilities of the GPGPU.

Thanks to its flexibility, low hardware overhead, and moderate performance degradation, this strategy could be effectively employed to increase the reliability of GPGPUs when they are adopted in safety-critical applications.

- The combined solution merging dynamic self-repairing configuration (BISR) and dynamic duplication with comparison (DDWC) aims to obtain a robust fault mitigation solution.
This latter solution mechanism is based on the possibility to use both the BISR and DDWC mechanism at the same time. In this architecture it is possible to define the number, from 1 up to 8, of cold stand-by modules that can support both BISR and DDWC mechanism, in parallel with the existing SPs.

Two switching units, based on meta-crossbar structures, targeting the data-path for input and output interconnections in the SPs. Based on the configuration instruction it is possible to control the switching units and to implement the mechanism. By comparison with the two previous architectures this is the most complete and optimized one.
Experimental results show that the solution introduces a moderate area overhead (up to 14.87% in the worst scenario) and a moderate performance degradation.
This strategy covers the mitigation effects of fault accumulation in the SPs cores and increase the reliability of GPGPUs.

Acknowledgements

Ringrazio il Professor Matteo Sonza Reorda per la disponibilità dimostrata nei miei confronti e per avermi offerto di poter lavorare ad un lavoro innovativo ed interessante.

Questi mesi mi hanno permesso di acquisire nuove competenze mettendomi alla prova con l'architettura di nuove tecniche innovative non ancora implementate del tutto nel mondo delle schede video.

Grazie a Esteban per avermi supportato durante questo percorso, rendendosi sempre disponibile nel momento del bisogno. Il suo aiuto è stato prezioso ai fini del raggiungimento dell'obiettivo.

Un ringraziamento anche al professor Strepone per la collaborazione nella stesura degli articoli, che verranno pubblicati.

Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction to FlexGrip

FlexGrip (FLEXible GraphIcs Processor) is an open-source GPGPU, which has been optimized for FPGA implementation. The first model, developed by the University of Massachusetts, was based on the NVIDA G80 microarchitecture and optimized for Xlinx FPGA. It is also compatible with the CUDA programming environment under SM_1.0 compatibility.[1]

1.1 FlexGrip General Architecture

GPGPUs have a many-core device architecture and possess substantial parallel processing capabilities. They have a multicore architecture, based on an array of streaming multiprocessor (SM), each one having a certain number of parallel scalar processors (SP), which enable the device to execute more threads in parallel.

A thread block represents a collection of operations which can be performed in parallel by each scalar processor simultaneously. The block scheduler is responsible for schedule thread block in a round-robin fashion. CUDA instructions define the kernel instructions and parameters (thread blocks, grid dimensions, etc.), data, control and status that are stored and used by FlexGrip.

The number of thread blocks scheduled at the same time is determined by the number of scalar processors in a streaming multiprocessor and the number of SMs. After scheduling the thread blocks, the block scheduler signals the warp unit to initiate scheduling the warps, which are contained within the respective thread blocks. The maximum number of thread blocks that can be scheduled to a SM is restricted by the available shared memory and SM registers.

1.1.1 Streaming Multiprocessor

The internal architecture of FlexGrip is based on the SIMT (Single-Instruction Multiple-Thread) paradigm and exploits a custom SM core with a five stages

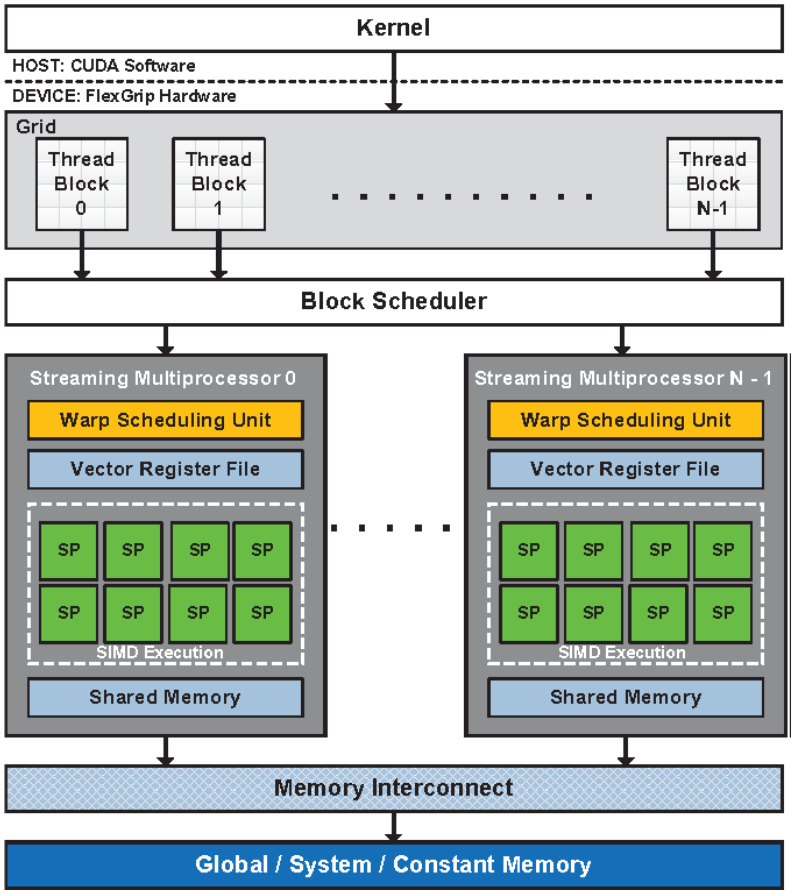

Figure 1.1. Overview of a GPGPU architecture

pipeline (Fetch, Decode, Read, Execution/Control-flow and Write-back), as shown in Figure 1.2.

In the SIMT paradigm, one warp instruction is fetched, decoded and distributed to be processed on an independent SP within the SM. The Read and Write-back stages load and store data operands from/to Register Files, shared, global or constant memories. The constant memory is a read-only memory, which is initialized by the host.

All pipeline stages output have a stall signal that is fed to the preceding stage. This signal indicates that the stage is busy and not ready to accept new data.

Warp Unit

This special-purpose parallel processor executes the same instruction (warp instruction) for a set of threads. A warp is defined as a group of 32 threads, as shown in Figure 1.3. Each warp includes a program counter (PC), a thread mask and

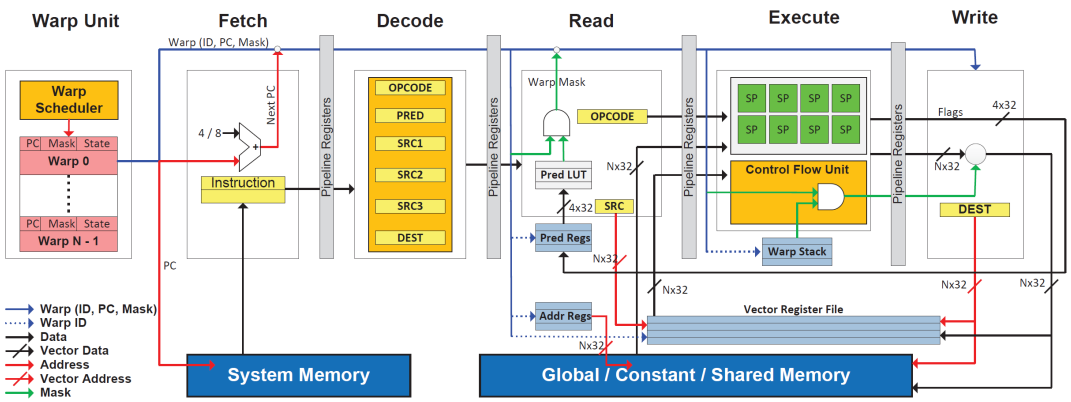

Figure 1.2. Block diagram details of the FlexGrip Streaming Multiprocessor

a state. The PC is used for each warp so they can follow their own conditional path, the mask is used to prevent threads execution under wrong conditions and the warp state indicates the status of the warp: Ready, Active, Waiting or Finished. This latter mechanism is also called warp scheduler controller (WSC) for thread management.

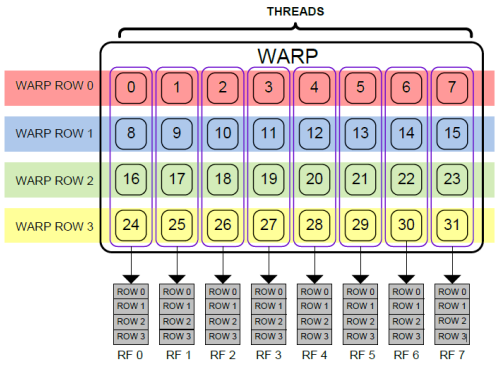

Figure 1.3. Register le configuration with 8 cores

Fetch and Decode Stage

This stage is responsible for fetching four or eight bytes CUDA binary instructions from the system memory. The instruction bus of the system memory is 32-bit wide and is possible to have 32- or 64-bit length instruction. To complete the stage, the fetch stage take 1 or 2 cycles. After fetching the instruction, the PC value is incremented (by 4/8 bytes) to point to the next instruction.

The decode is a complex module that decodes the binary instruction to generate several instructions depending on the opcode, predicate data, source and destination operands fields. There are 27 different instructions already implemented and works correctly and also verified [2]

Read and Write Back Stage

In the Read stage, source operands are read from the vector register file or shared/-global/constant memory blocks depending on the decoded inputs. All instructions can include an optional predicate flag that controls conditional execution of the instruction (predicate instructions). The predicate register file is used to store these predicate flags. The warp mask is updated by combining the current mask with the predicated instruction.

The Write stage, instead, stores intermediate data in the vector register file, memory addresses in the address register file and predicate flags in the predicate register file. Output results are stored in the global memory.

Execute Stage

The Execution stage is the pivotal of the whole architecture, because it contains the scalar processors with all the logic and arithmetic modules necessary required to perform in parallel the operations supported by the Instruction Set Architecture (ISA). So, the SPs are the real core of the system and without them any operations will be possible inside one GPGPU.

The original design of the NVIDIA G80 architecture can have 8, 16 and 32 SPs, also in FlexGrip the execution block have a hierarchical number of Scalar processors, configurable in different ways as 8-16-32. It is worth nothing that the number of SP in parallel increase the execution process, so for instance having 32 SP in parallel is faster than having just 8, notable using one same testbench program in both the cases.

It should be noticed that, results of the output of each SP could be affected by errors coming from the logic or arithmetic blocks. These latter blocks are the most important because without them it is not possible to perform any logic-arithmetic operations. So, a fault affecting one SP-cores may cause output errors to the next pipeline stage.

The input and output data channels are independent for each SP. In contrast, the control-path connections are shared among all SPs because it deals with branches, synchronization point settings, block barrier points and kernel return instructions.

As output, the SP produces the result (DST) signals and the changes of the predicate flags.

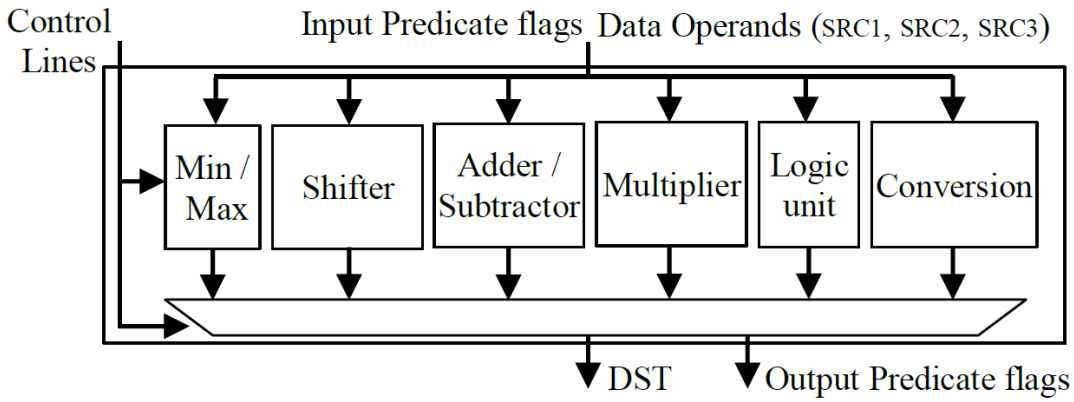

Figure 1.4. A general scheme of the internal architecture of the integer SP core

The SPs process signed and unsigned integer operands and include hardware modules for addition/subtraction (ADD/SUB), multiplication (ML), integer conversion (ICON), comparison (COMP), shifting (SHF) and logic unit (LU) with basic logical operations (AND, OR, XOR and NOT), as shown in Figure 1.4

1.2 Instruction Format Support

The supported assembly instructions supported (SASS) by FlexGrip, executable under the CUDA programming environment under SM_1.0 micro-architectural compatibility, are presented in the following tables. [3]

Table 1.1, 1.2 and 1.3 introduce the arithmetic and logic instructions, the data handling and memory instructions and the control-flow instructions, respectively. In Table 1.3, COMP_TYPE refers to comparison type and it depends on the predicate flag generated by an arithmetic or logic operation. In Table 1.2, COND parameters refer to predicate conditions. g[] and c[0x1][] correspond to shared memory and constant memory locations respectively.

Mnemonic	Description	The revised format in the improved version
MVC	*Load from constant memory*	MVC RX, c [0x1] []
GLD	*Load from global memory*	GLD.U32|U16|S16|U8|S8 RZ, global14[]
GST	*Store to global memory*	GST.U32|U16|S16|U8|S8 global14[], RX
MOV	*Move register to register/load from shared memory*	MOV RZ, RX / g[] MOV.U16 RZ(L|H), RX(L|H) / g[].(U16|U8)
MOV32		MOV32 RZ, RX / g[] MOV32.U16 RZ(L|H), RX(L|H)
MVI	*Move immediate to destination*	MVI RX, Imm
R2G	*Store to shared Memory*	R2G.U32.U32 R2G.U16.U16 R2G.U16.U8
R2A	*Move general purpose register to address register*	R2A AX, RX
A2R	*Move address register to general purpose register*	A2R RX, AX

Table 1.1. Supported data handling and memory SASS instructions

Mnemonic	Description	The revised format in the improved version
BRA	*Branch*	BRA CX.COND Imm BRA Imm
BAR	*Barrier Synchronization*	BAR.ARV.WAIT b0, 0xFFF
RET	*Return from kernel*	RET RET CX.COND
SSY	*Set Synchronization point*	SSY Imm
NOP	*No Operation*	NOP NOP.S

Table 1.2. Supported control-flow SASS instructions

Mnemonic	Description	The revised format in the improved version
I2I	*Integer to Integer conversion*	I2I.U32.U16/S16 RZ, RX(L|H) / g{[}{]}.U16 I2I.U32.S32 RZ, |RX| / -RX I2I.U32.U16.BEXT RZ, RX(L|H) / g{[}{]}.U8 I2I.S32.S16.BEXT RZ, RX(L|H) / g{[}{]}.S8
IMUL	*Integer Multiplication*	IMUL.U16.U16 RZ, RX(L|H) / g{[}{]}.U16, RY(L|H) IMUL.S16.S16 RZ, RX(L|H) / g{[}{]}.S16, RY(L|H)
IMUL32		IMUL32.U16.U16 RZ, RX(L|H) / g{[}{]}.U16, RY(L|H)
IMUL32I		IMUL32I.U16.U16 RZ, RX(L|H), Imm IMUL32I.S16.S16 RZ, RX(L|H), Imm
SHL	*Shift Left*	SHL RZ, RX, RY / Imm SHL RZ, g {[}{]}, Imm SHL.U16 RZ(L|H), RX(L|H), Imm
SHR	*Shift Right*	SHR.S32 RZ, RX, RY / Imm SHR.S32 RZ, g {[}{]}, Imm SHR.U16 / S16 RZ(L|H), RX(L|H), Imm SHR RZ, g{[}{]}, Imm SHR RZ, RX, RY / Imm
IADD	*Integer Add*	IADD RZ, RX / -RX, RY IADD RZ, g{[}{]}, RX / -RX IADD RZ, RX, c{[}0x1{]}{[}{]}
IADD32		IADD32 RZ, RX, RY / -RY IADD32 RZ, g {[}0x..{]}, RX / -RX IADD32.U16 RZ(L|H), RX(L|H), RY(L|H) / -RY(L|H)
IADD32I		IADD32I RZ, RX / -RX, Imm IADD32I RZ, g{[}{]}, Imm
IMAD	*Integer Multiply and Add*	IMAD.U16/ S16 RZ, RX(L|H), RY(L|H), RW IMAD.U16/ S16 RZ, RX(L|H), c{[}0x1{]}{[}{]}, RY IMAD. RZ, RX(L|H), c{[}0x1{]}{[}{]}, RY
IMAD32		IMAD32.U16 RZ, RXL|H, RYL|H, RZ
IMAD32I		IMAD32I.U16/ S16 RZ, RX(L|H), Imm, RZ
LOP	*Bitwise logical Operation*	LOP.AND/OR/XOR/PASS_B RZ, RX/ g{[}{]}, RY LOP.AND/OR/XOR/PASS_B RZ, RX, c{[}0x1{]} {[}{]} LOP.U16.AND/OR/XOR/PASS_B RZ(L|H), RX(L|H), RY(L|H)
ISET	*Integer Comparison*	ISET RZ, RX, RY / c{[}0x1{]}{[}{]}, COMP_TYPE ISET RZ, g{[}{]}, RX, COMP_TYPE ISET.S32 RZ, RX, RY / c{[}0x1{]}{[}{]}, COMP_TYPE ISET.S32 RZ, g{[}{]}, RX, COMP_TYPE

Table 1.3. Supported arithmetic and logic SASS instructions

Part II

Implementation

Chapter 2

Built-in Self Repair (BISR)

The first strategy presented is the Built-in self-repair (BISR) mechanism that aiming to solve problems related to permanent faults effects, during the in-field operation, inside the Streaming Multiprocessor of the GPGPU. The advantage of this solution is the possibility to identify the presence of a permanent fault using technique as Design for Testability (DfT) or Software-Based Self-Test (SBST).

2.1 State of the art

This BISR strategy is based on the Reconfigurable logic blocks (RLB), where there are N identical functional units plus one additional backup (spare) block used to replace the faulty functional unit.[4]

The architecture explored, shown in Figure 2.1, required switching element blocks at inputs and outputs for re-configuration of the units. Some extra logic is needed to control the switches. Multiplexer and De-multiplexer may be used for the switching blocks.

In the past, BISR has been successfully applied to increase the reliability of digital design in memory blocks of processor-based systems or targeting data-path units or functional units, using hardware, software and hybrid approaches. Hardware solutions include Duplication with Comparison (DWC), Double and Triple Modular Redundancy (DMR, TMR), ECC and the hardening of selective logic gates. [5]

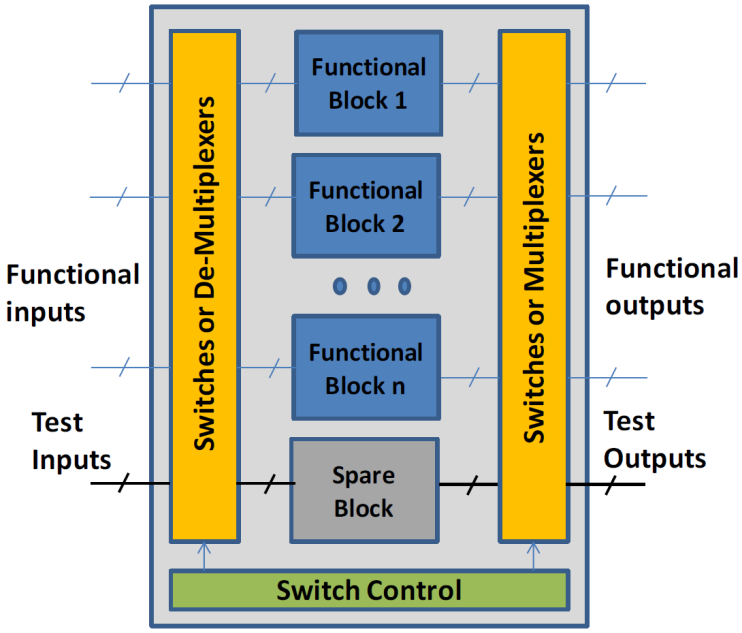

Figure 2.1. Structure of a re-configurable logic block

2.2 Working on FlexGrip

The inclusion of additional fault tolerance and self-repair modules is needed to implement the BISR technique. Particularly, the design of interconnections between the existing SPs blocks and the SSPs spares, both for the input and for the output part.

Figure 2.2 shows the original configuration of n SP in a Streaming Multiprocessor, inside the Execution Unit of the GPGPU. It should be noticed that each SP has three input data operands SRC1, SRC2, SRC3 (32 bits length) and predicate flags (4 bit-size) forming a data channel. SRC1, SRC2, and SRC3 are selected depending on the instruction type, while some control signals select and configure the SP.

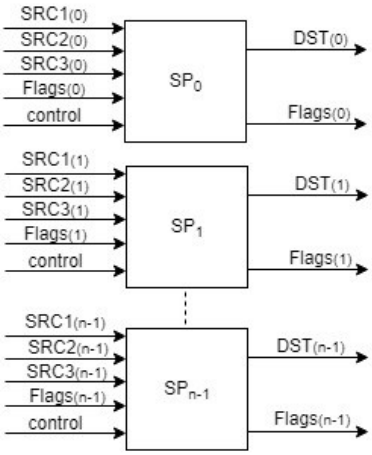

Figure 2.2. Original configuration of n-Scalar Processor

The proposed method to mitigate errors in data-path units, in the GPGPU model, consider the addition of a Crossbar-type structure. This structure allows the configurability of the Scalar Processors with one of the m additional passive Spare Cores (SSP). The number n of spare cores should be set up by the designer. It is worth noting that the number of Spares should not be higher because it affects the area and consumption overheads, but also because this technique aims to switch the Spare Core just in the case of a fault.

The switching structures was designed parametrically, in the worst-case using the maximum reachable data and wires configuration with 32 cores, thus the structure can easily be modified to add up to 32 Spare cores. Nevertheless, as commented above the implementation is restricted to 8 additional SSP cores.

2.2.1 Input Crossbar Architecture

The purpose of the input crossbar is to deliver the correct input data based on which scalar processor is active and substitute one of the original SP that pretend to be switched off.

In order to have this scenario, there are different blocks that simulate this behavior. It should be noticed that the crossbar will be use just during the switch on or the reset condition, otherwise some data dependencies problem make appear.

The input crossbar behaviour's is to take all the input data-bus and take them to the SPs and SSPs. This operation is done by means of the 11 bits instructions, that will be introduced later in the text.

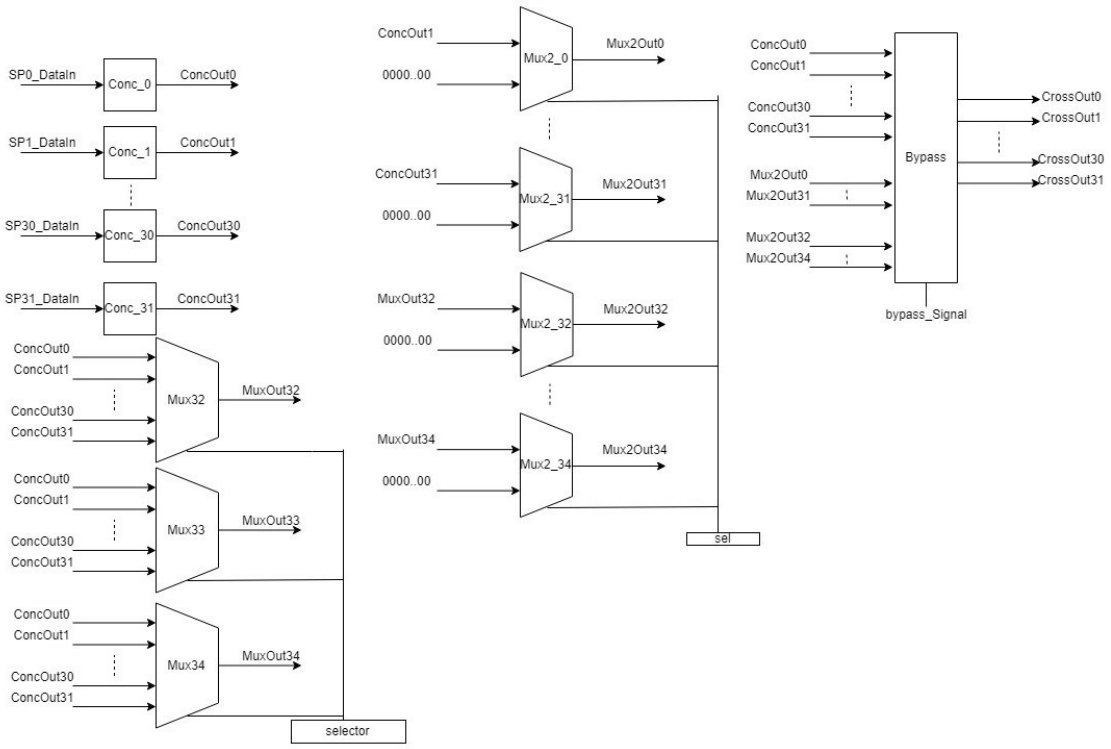

Figure 2.3. Input Crossbar architecture

Entering more into design specification, will be report what each block does:

- Concatenation: this block is used to concatenate for each SP all the input data sr1, sr2, sr3 and carry_in flag in a string of 100 bits length;

- Mux32: there are 3 of these blocks, that have in input all the 32 possible output of the concatenation block. This block has also an enable, that based on the selected Spare processor channel enables the mux, otherwise the mux is switched off. More than that, there is the selector, the same of the next block, that says which is the selected spare cores.

- Mux2_i: for each possible output of previous block, the original one (ConcOut_i) or for the spares one (MuxOut_3x), based on which is the selected spare core enables just the correct mux, otherwise the output of the original SP goes to ground;

- Bypass: this block is used whenever you want to use all the original configuration (bypass=0) or the SSPs one;

2.2.2 Output Crossbar Architecture v1

Because this structure connects the original SP plus the m spares, the output crossbar was designed as the input one, but with some changes: the output data for each of the n-1+m cores is concatenated, then goes into one multiplexer that has (1+m spare) inputs and based on the controller, depending on the selected channel and the selected spare, the correct output is selected. As the crossbar in, it has the bypass block, according to the same bypass_signal. In this way the crossbar output has n-1+m spare inputs and n-1 outputs, as the original design.

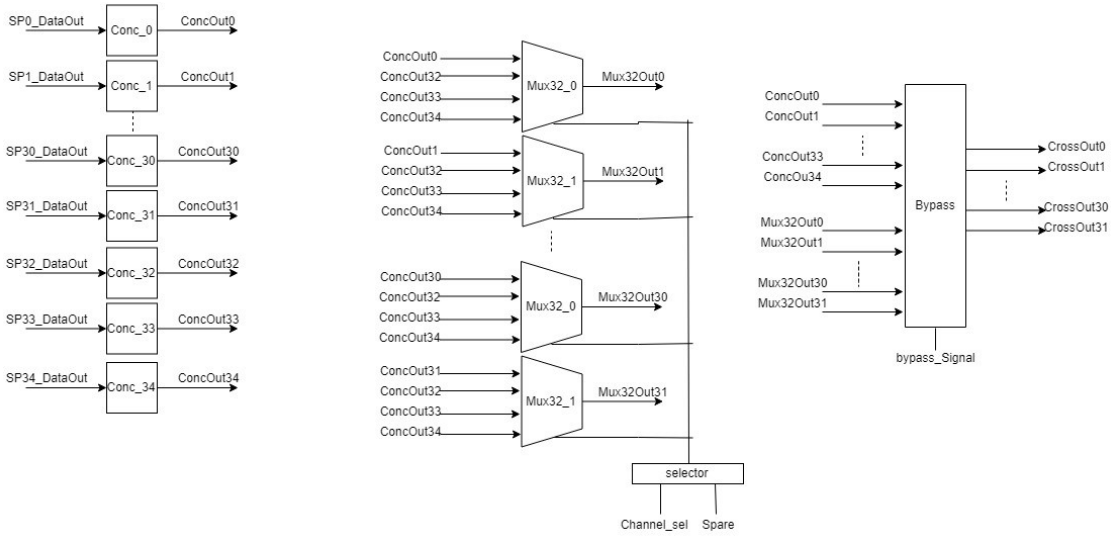

Figure 2.4. Output Crossbar architecture

Entering more into the design specification will be report what each block does:

- Concatenation: this block is used to concatenate, for each SP, all the output data: carry out, overflow, sign, zero and result;

- Mux32: there are 32 of these blocks, that have in input one of the n-1 possible output of the concatenation block of the original SP and the n outputs of the m spare cores of the concatenation block.

  This block has one selector, that has in input the selected channel and the spare core wanting to be substitute.

- Bypass: this block is used whenever you want to use the original configuration (bypass=0) or the the spares one (bypass=1);

2.2.3 Output Crossbar Architecture v2

Since the whole designed proposed in this paper was synthesized for other purpose, during this long phase, new version for the crossbar output was implemented and verified.

This later configuration is a bit different compare to the version presented above. The new one is smaller in term of number of cells and total cells area. Particularly, there is just one mux8 to select the proper spare output signal and n mux2 to select where the output of the SP plus SSP should go.

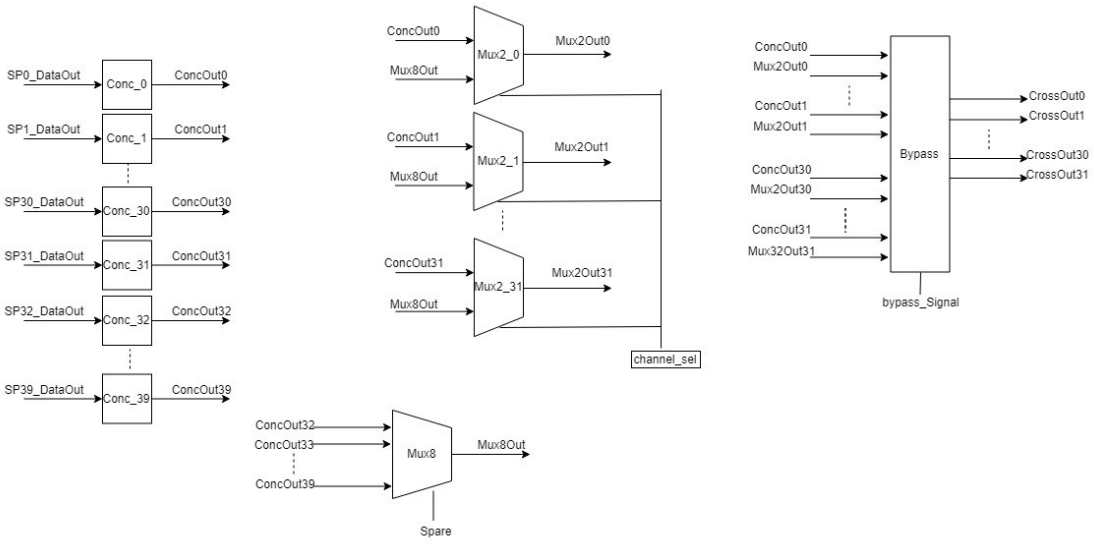

Figure 2.5. Output Crossbar architecture v2

Entering more into the design specification will be report what each block does:

- Concatenation: this block is used to concatenate, for each SP, all the output data: carry out, overflow, sign, zero and result;

- Mux8: there is just one of this, that has in input all the n outputs of the m spare cores. This block has one selector, based on the selected spare core wanting to be switched on;

- Mux2_i: there are n-1 of this block, each one has for the first input the output of the i concatenation block and as second input the output of the mux8. The selector of each mux is based on the channel selected in the crossbar input;

- Bypass: this block is used to use the original configuration (bypass=0) or the spares one (bypass=1);

2.2.4 Complete Architecture

The BISR architecture is implemented in the Execution/Control Stage of the GPGPU, it has N Scalar Processor in parallel with M Spare Core (SSP). The SSPs are cold stand-by modules, that reducing the power consumption during inactivity.

The controller module is added to control SPs and SSPs in the system with two switching unit based on meta-crossbar structure, that are used to control the data-path signals, as shows in Figure 2.6.

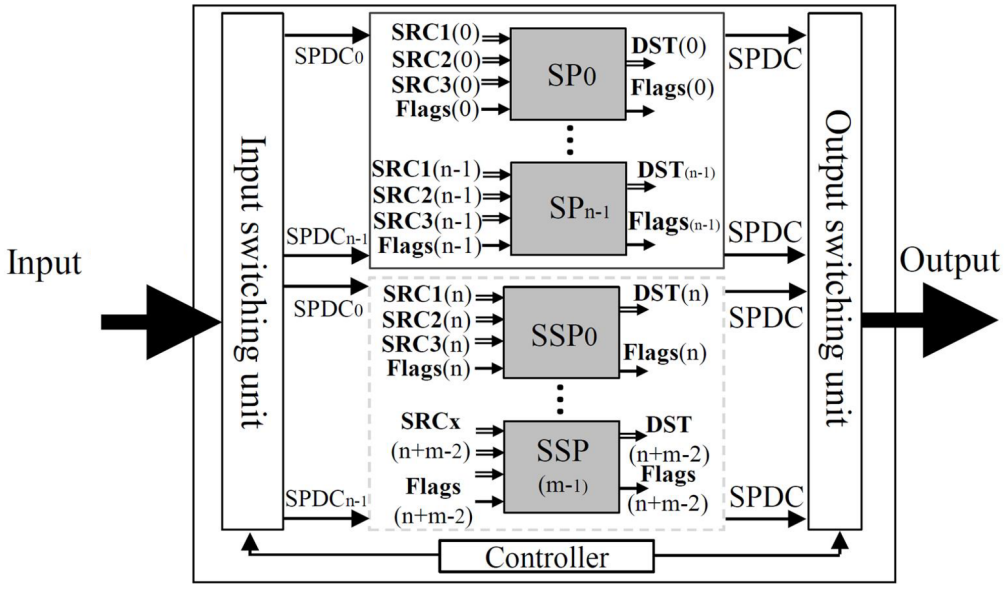

Figure 2.6. Complete Architecture with Input and Output crossbar

The Figure above has an adapted, not equal, architecture described in other works. Theory behind the crossbar structure is the same, but the implementation is slightly different. [6]

One of the specification of the proposed solution is to have the same controller for both input and output switching units. By this way is possible to select correctly the channel of the selected SP, that pretend to be affected by a fault, and selecting the spare cores that have to substitute in the configuration.

2.3 BISR instruction

The dimension of the control signals was crucial for the creation of the instruction used for switching between the SP and the spare SP.

Particularly, the instruction could be 32 or 64 bits long, it was chosen based on the dimension of some signals bits as the bypass, the channel_sel and the spare_sel.

A 32-bits long instruction was implemented, knowing that bypass signal is just one bit long, channel_sel and spare_sel signals are both 5 bits long, 4 bits for the opcode and 2 more bits to say the instruction width and if the instruction is normal or flow control.

Opcode	Not Used	Bypass	Channel_sel	Spare_sel	Not Used	flow(1)	32bit long(0)
4bit	9bit	1bit	5bit	5bit	6bit	1bit	1bit
31-28	27-19	18	17-13	12-8	7-2	1	0

Table 2.1. BISR Instruction Structure

The first problem was to find the unique opcode, not used yet, for the instruction. The method employed to solve was to insert 16 new random instructions, each one with all the 16 different possible opcodes, and observing some signal of the logic simulation. If the simulation stall, using one special opcode, means that the opcode was not used yet, so it was discovered that there are 9 possible free opcodes, but 4 are used yet with different control signal. Finally, the 5 possible opcodes are: 0111-1001-1011-1100-1110, the selected one is 1001.

Once the structure from Table 2.1 was defined, the instruction was implemented in the decode stage of the GPGPU description (pipeline_decode.vhd). Is noticed that the purpose of the used bits:

- bits 31-28 are used for the opcode, this is univocally for my instruction;
- bit 18 is used for declaring if I use the normal configuration (0) or the spare one (1);
- bits 17-13 used for selecting the proper channel of the pretended break SP;
- bits 12-8 used for selecting the new spare core that will be use;
- bit 1 is used for declaring that is a flow type instruction;
- bit 0 is used for declaring that is a 32-bit long instruction;
- bits 27-19 and 7-2 not used.

2.4 Experimental Results

2.4.1 Hardware Overhead

This section is concerning the analyze the effect of BISR architecture implemented on the FlexGrip model in its 3 configuration (8, 16 and 32) for the SP. For each case, the number of introduced Spare SP is ranged from 0 to 7. The analyses were performed comparing the original design of FlexGrip and the implemented one the estimation of area and cells area overheads, using the ultra-compiler configuration of Design Vision Tool by Synopsis .

Version	SP Cores	SSPs	Total Cell in design	Area Overhead (%)	Total SP cells in the design	SP/SSP core cells(%)
Original	8	0	229,515	-	52,984	23.08
	16	0	280,132	-	105,968	37.82
	32	0	386,100	-	211,936	54.89
Fault Tolerance	8	0	231,343	0.8	52,984	22.90
		1	237,279	3.4	59,607	25.12
		2	243,063	5.9	66,230	27.24
		4	254,692	11.0	79,476	31.20
		6	266,182	16.0	92,722	34.83
		7	271,757	18.4	99,345	36.55
	16	0	283,160	1.1	105,968	37.42
		1	290,034	3.5	112,591	38.81
		2	296,164	5.7	119,214	40.25
		4	309,318	10.4	132,460	42.82
		6	321,529	14.8	145,706	45.31
		7	335,139	19.6	152,329	45.45
	32	0	392,476	1.7	211,936	53.99
		1	400,902	3.8	218,559	54.51
		2	410,280	6.3	225,182	54.88
		4	425,172	10.1	238,428	54.07
		6	440,576	14.1	251,674	57.12
		7	460,372	19.2	258,297	56.10

Table 2.2. Hardware overhead of the BISR strategy for multiple configuration of GPGPU

The modules taking into account for implementing the BISR strategy are the Decode, Read and Execution stages. The RT level of these modules are modified and then the GPGPU model was synthesised at gate level.

Table 2.2 reports, for each configuration, the required number of cells and the

percent of area overhead. The cost to implement the instruction, introduced by BISR strategy, the switching modules and the switching controller. The 0 SSPs configuration is used to estimate the cost in the BISR structure.

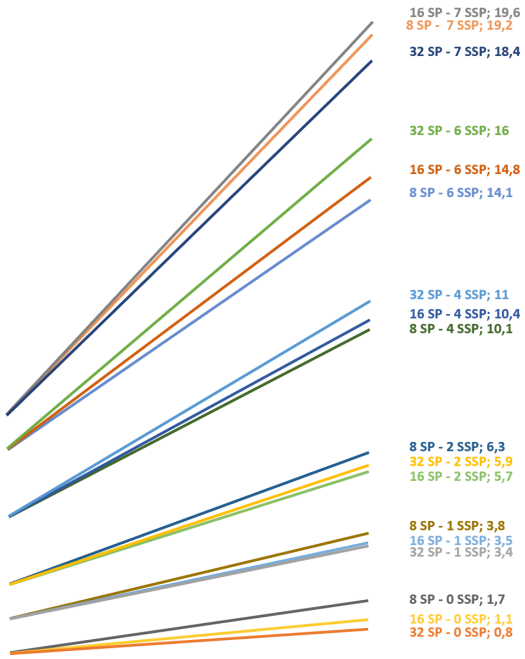

Figure 2.7. BISR Overhead (%) for all the configurations

The hardware overhead for this configuration represents a small percentage of the whole hardware. For all the 3 configuration the hardware cost is in the range from 0.8% to 1.7%, while the cost of the SSPs linearly grows with their number. In fact, the cost for one SSP core is 6623 cells, that introduces an area overhead greater then 3% in al Scalar processor configurations. The optimum choice of both parameters depends on the design requirements. In any case, it is worth noting that the hardware overhead remains below 20% for all the considered GPGPU configurations.

The last two columns of the Table report some figures allowing the evaluation of the relative size of the SPs with respect to the total size of the FlexGrip model. From the results, it is shown that the percent area of the whole SM that can be protected resorting to the BISR strategy ranges from about 25%, in the 8 SPs configuration, to about 55% with 32 SPs. It is worth noting that the adopted BISR mechanism was aimed to mitigate faults in the SP cores, only. Other solutions can be used to mitigate faults in other modules.

In Figure 2.7 is show the percentage of area overhead comparison between each configuration varying the number of SSPs.

2.4.2 Performance and Power overhead

Since the BISR mechanism requires the introduction of the switching modules to increase the flexibility of the interconnections, is clear that it impacts on the GPGPU overall performance. Using the synthesis tool experimental analysis is performed. The impact on the performance of the implemented BISR strategy has been evaluated by analysing the changes in the critical path delay for all configurations.

Results showed that the performance degradation reach up to 20% for a large number of SSPs (6 or 7). This is mainly caused by the logic control of the two switching modules. More in detail, for one SSP, the timing degradation is up to 15% and up to 16% for 2 SSPs. Moreover, any specific optimization was done to reduce performance overhead.

The power overhead can be neglected for this BISR strategy due to the inactivity of the SSPs, act as cold stand-by modules. Clearly in a real implementation only static power by leakage current is consumed during operation and it depends on the transistor technology. Thus, the final power overhead of the BISR strategy is negligible in comparison with the dynamic power consumption produced in the GPGPU.

2.5 Reliability Advantages

This strategy is performed on the device power-on or reset, so the fault detection and location phases, as well as the reconfiguration one, can be executed without any strict time and memory constraints

The aim of the BISR technique is to allow that the GPGPU continues is working even if one or more fault arose inside the SPs. It is noticed that this strategy is independent on which fault affect the SP core.

We want to determine which is the probability of proper execution using BISR method (R_{BISR}). Knowing that a GPGPU is composed of n SPs and m SSPs, the execution of the system is correct if all thread instructions are operated without failures in the available execution units of an SM, after a time t. Thus, because all the units SPs and SSPs are identical and operate independently among them, so at time t the probability of correct operation in the SPs ($P_{SP(t)}$) and the number of SSPs is equal. The probability of proper operation in the GPGPU can be described as the probability of GPGPU failure, when at most (m+1) SPs or SSPs fail. In this way, the probability of proper execution, using the BISR mechanism (R_{BISR}), follows a cumulative distribution function:

$R_{BISR} = \sum_{i=0}^{m+1}(\frac{n+m}{i})[P_{SP(t)}]^{n+m-i}[1-P_{SP(t)}]^{i}$

Moreover, there is a direct relationship between the number of SSPs (m) and R_{BISR}. As previously told, the number (m) of SSPs can not be higher than the number (n) of SPs.

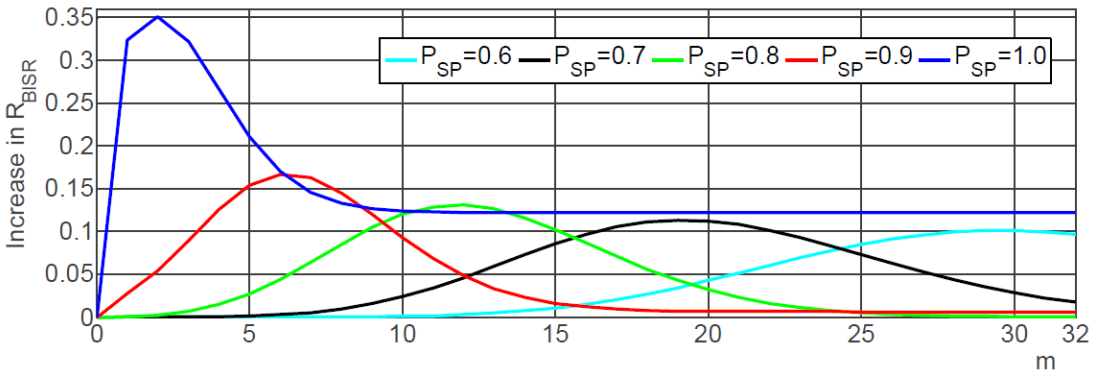

Figure 2.8. Improvement in the reliability of the BISR structure for multiple probabilities of correct execution under multiple configurations of the SSPs (m)

Figure 2.8 told us that the Probability of correct execution R_{BISR} is dependant on the values of m and the Probability of correct operation P_{SP}.In fact, the maximum reliability peak can be use to select the number of SSPs considering the target (R_{BISR}).

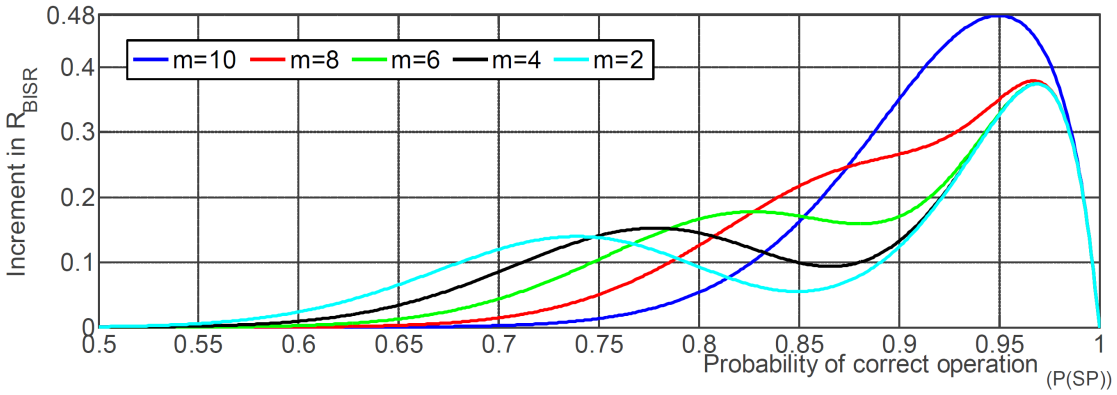

Figure 2.9. Improvement in the reliability of the system R_{BISR} with respect to the probability of correct execution for various values of SSPs (m)

The graph in Figure 2.9 describes gained benefits in terms of reliability for multiple BISR configurations. Increasing the number of m has a positive impact in the reliability of target structure. Both the previous Figures 2.8 and 2.9 can be used to reach a reliability target selecting the best trade-off among the parameters. [7]

Chapter 3

Dynamic duplication with comparison (DDWC)

The second strategy presented is called Dynamic Duplication with Comparison (DDWC), is a technique targeting the detection of permanent faults in the Execution Unit of FlexGrip. The idea is based on the adaptation of classical DWC mechanism to protect the SP cores in a GPGPU. This solution introduces small or null performance overhead and minimal modifications in the existing structures.

3.1 State of the art

The DDWC technique is based on the Redundant multithreading (RMT), that is a transient error detection technique, that uses redundant copies of a thread and compares results. To enable the comparison, both the original and redundant thread must synchronize. [8]

This technique can be implemented on different level inside one GPGPU as Software-based or hardware solutions. In the first one the synchronization is implemented using the shared memory, modifying the kernel description and duplicating the operation in the program. With this solution we increase the reliability of the GPGPU, but the analysis of some trade off is crucial. This solution may use spatial and timing redundancy, but performance degradation directly depends on the behaviour of the application. [9]

Moreover, with hardware solutions the spatial redundancy and the the addition of special modules is exploited. By this way is possible to increase the reliability by allocating operations to redundant and independent modules. Although the fault detection is done by a comparison between the original thread and the duplicated one. This kind of mechanisms do not introduce high performance degradation, but the area overhead is directly depending on the redundant modules included the input and output interconnections.

To sum up, the software mechanisms are developed, targeting the detection or correction of faults. In contrast, the hardware structures are more complex to develop and implement. Furthermore, the original design may require structural modifications.

3.2 Implementing DDWC in FlexGrip

The DDWC mechanism is based on the addition of one spare SSP module in parallel with the existing SPs. This cold stand-by module performs the same operations as one of the original configuration and also monitoring the coherence of results.

Starting from the original configuration of n-Scalar processor, shown in Figure 2.2, two selector switches are implemented, one for the input and the other one for the output.

DDWC employs the concept of sphere of redundancy, as show in Figure 3.1, which is based on replicating a target module to increase the fault detection or to mitigate the fault effect in a system.

A custom instruction was developed to control both the Switching units, that identify the monitored modules.

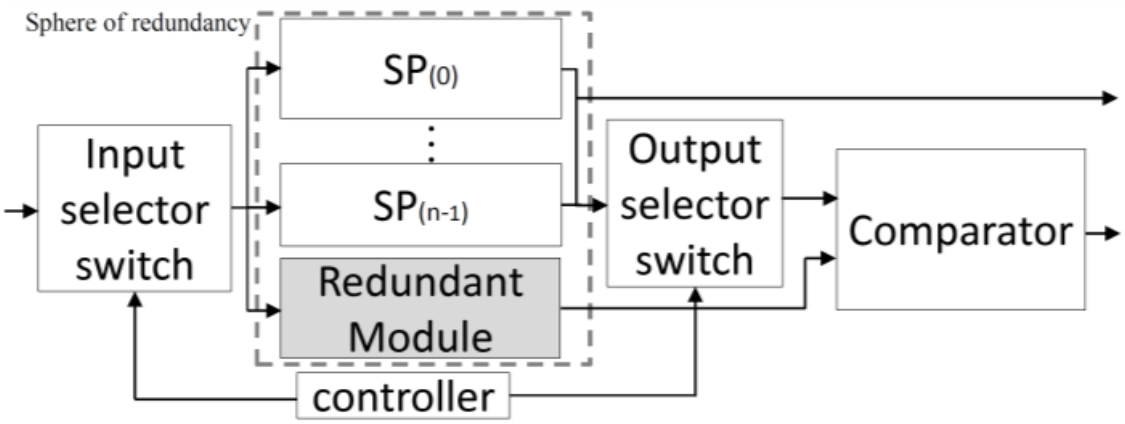

Figure 3.1. general scheme of the concept DDWC redundancy applied to the SPs

The DDWC mechanism is completely trasparent to the programmer, so this solution can be used for the in-field test just modifying the application code and insert the new instruction.

This implemented DDWC structure is intended to be used during the in-field operation of the GPGPU. So the redundant core should swap among the SPs of the Streaming Multiprocessor.

3.2.1 Input selector Switch

The purpose of this block is to duplicate the thread interconnection of one SPs to the redundant Spare. In order to have this scenario, the input selector switch is composed by one multiplexer that, based on the selected channel, connects the interconnections of one SPs to the redundant block. The other Interconnections are completely direct from the previous stage block to the input of each SPs.

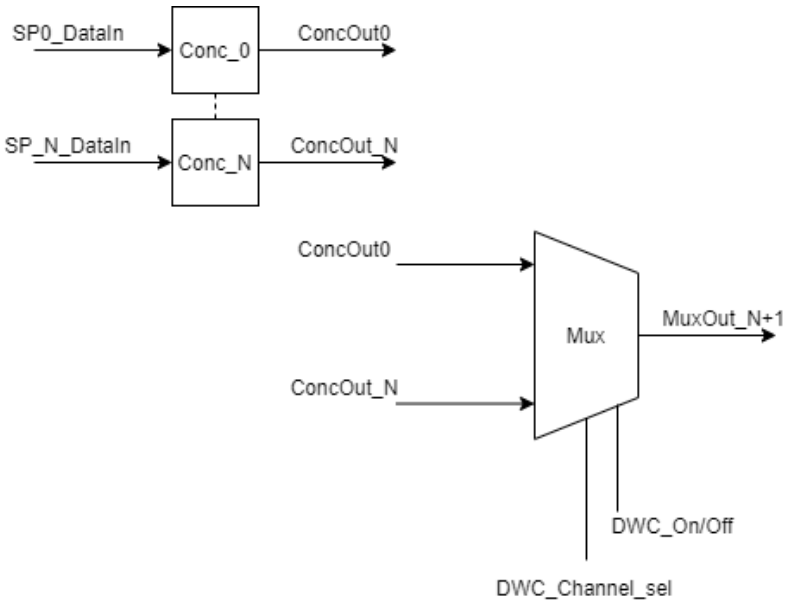

Figure 3.2. Input Switch architecture

Entering more into the design specification will be report what each block does:

- Concatenation: this block is used to concatenate for each SP all the input data sr1, sr2, sr3 and carry_in in one 100 bits string, useful instead of carrying 4 different signals in design part. The output of this block goes directly to the SPs and also to the input of the Mux;

- Mux: this could be 8, 16 or 32 depending on the SP configuration, the output of this mux is directly connected to the redundant block. This block has also an enable controller, managed by the ad-hoc instruction, that switch off the multiplexer and the SSP.

3.2.2 Output selector Switch

The aim of this block is to monitoring the behaviour of the selected SP with the redundant SSP. In order to do that there is a xor comparator with two input, the

first one is for the multiplexer, which inputs are the output of each SPs, while the second one is the output of the SSP module. The output of the comparator signal is crucial because it is connected directly with the host, that in case of a difference between the threads switch off the application. Thus, with the method explain in chapter 3.5 is possible to monitoring the system.

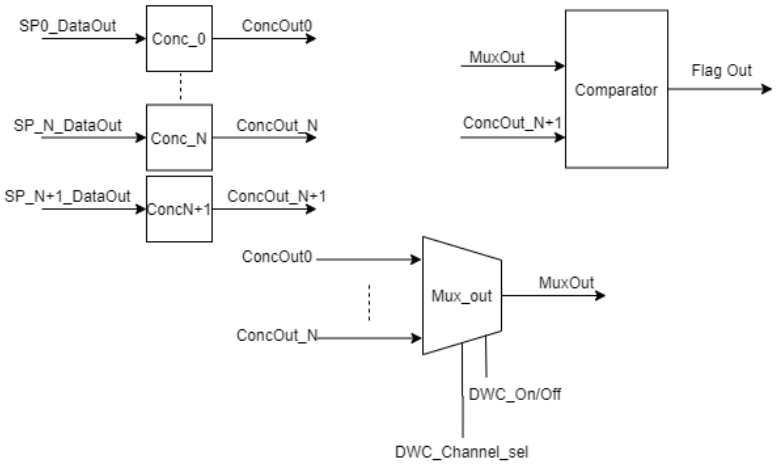

Figure 3.3. Output Switch architecture

Entering more into the design specification will be report what each block does:

- Concatenation: this block is used to concatenate, for each SP, all the output data: carry out, overflow, sign, zero and result; The output of this block goes to MuxOut and also to the next Module (Write Back)
- MuxOut: this could be 8, 16 or 32 depending on the SP configuration there is just one of this, that has in input all the N outputs of the N spare cores. This block has one selector that has in input the selected spare core that is wanting to be monitored;
- Comparator: this block has as first input the Output of the MuxOut, while on the second one the output of the Spare SSP. The Flag Out output signal is the one monitored that perform the DDWC

3.3 DDWC Instruction

Strong as the work done on the BISR instruction, the selected opcode for the DDWC instruction is chosen between the 4 remaining Opcodes, so it is chosen the 1011. Also in this case, the instruction is 32-bits long.

The important bits of this instructions are the DWC_On and DWC_Off, the Channel selected,the instruction width and if the instruction is normal or flow control.

The particular of this instructions is that are present two bits for switching on and off the instructions, when the DWC_Off bit is equal to '1' all the implemented architecture where switched off, otherwise there is the DDWC.

Opcode	Not Used	DWC_On	DWC_Off	Channel_sel	Not Used	flow(1)	32bit long(0)
4bit	8bit	1bit	1bit	5bit	11bit	1bit	1bit
31-28	27-20	19	18	17-13	12-2	1	0

Table 3.1. DWC Instruction Structure

Once this structure, from Table 3.1 was defined, the instruction was implemented in the decoder stage of the FlexGrip model. Is noticed that the purpose of the used bits:

- bits 31-28 are used for the opcode, this is univocally for my instruction;

- bit 19 is used for activate the DWC instruction.

- bit 18 is used for deactivating the DWC instruction;

- bits 17-13 used for selecting the proper channel to be monitored;

- bit 1 is used for declaring that is a flow type instruction;

- bit 0 is used for declaring that is a 32-bit long instruction;

- bits 27-20 and 12-2 not used.

3.4 Experimental Results

3.4.1 Hardware overhead

With Design Vision, the original and the implemented DDWC architecture versions have been synthesized. The obtaining data are present in Table 3.2, results in term of size of cells and relative total area overhead is presented, these results do not include cell costs of memories and file registers in FlexGrip.

Modules	SP Cores	Number of cells		Total Area Overhead (%)
		FlexGrip	DDWC	
Decode	8/16/32	1,229	1,266	3.04
Read	32	142,397	142,545	0.10
Execution	8	60,309	65,959	9.37
	16	113,293	118,739	4.81
	32	219,261	226,822	3.45
All	8	229,515	235,964	2.81
	16	280,132	286,360	2.22
	32	386,100	394,516	2.18

Table 3.2. Hardware overhead of the DDWC strategy

Results show that the hardware overhead is relatively low. In the Decode stage module, this overhead represents only 3%, while in the Read stage is irrelevant with just 0.1%. Instead, in the Execution module, the overhead is inversely proportional to the number of SPs. However, this overhead seems to be moderate (3.5-10%).

The total cost of hardware overhead for all SPs configurations is lower than the 3%. With these results is possible to say that DDWC is an effective solution to increase fault detection strategy without including high area overhead for the GPGPU. With these results the initial intention of limiting hardware overhead are respect.

3.4.2 Performance and Power overhead

As for the previous case Chapter 2.4.2, also in this case, after the logic simulation and logic synthesis using design vision, the tool gives information regarding the critical path delay.

As is possible to see from Table 3.3, that present performance overheads, the additional structure in the Decode module added a percentage of performance degradation of 2.90%.

The insertion of bypass in the Read module does not introduce any overhead. In contrast, the performance degradation in the Execution module seems to be directly affected by the number of SP cores.

When there is the configuration with the lowest number of SP cores, the critical delay path is increased by 7.52%. The configuration of 16 SP cores seems to present the maximum percentage of performance overhead with 14.79%. In contrast, the overhead drops for the 32 SP cores with 13.56%.

It also possible to notice that the total delay overhead is higher in the 32 SPs configuration, but it has the lowest performance degradation. This kind of behaviour is associated to the added SSP in parallel with the existing SPs. However,

the most representative timing effect is related to the input and output selector switch and the comparator.

Modules	SP Cores	Time Delay in the critical path (pS)		Performance degradation (%)
		FlexGrip	DDWC	
Decode	8/16/32	1.72	1.77	1.16
Read	32	3.65	3.65	0
Execution	8	6.51	7	7.52
	16	6.69	7.68	14.79
	32	7.52	8.54	13.56
All	8	11.88	12.42	4.54
	16	12.06	13.1	8.62
	32	12.89	13.96	8.3

Table 3.3. Performance overhead of the DDWC strategy

The power overhead can be neglected for this DDWC strategy due to the activity just in case of monitoring the correctness of one or more SPs, one each time. When it is not used DDWC, the implement hardware is seen as cold stand-by modules, affecting by static power only. The In a real implementation leakage current is consumed during operation and it depends on the transistor technology.

To sum up, under the inactive mode both the selector switch and the comparator are unconnected to avoid unnecessary switching activity, so a reduction of dynamic power during inactivity periods has been implemented.

3.5 Fault Detection

To evaluate the power of the DDWC, different benchmarks was employed. Firstly, some faults were injected in the RTL version of the FlexGrip model, with and without the implemented mechanism. Then single stuck-at fault has been considered. When the output flag of the comparator was used as an observability mechanism to detect faults. Experiments showed that for some case the detection capabilities increased up to 40%.

3.5.1 Estimation of fault detection

The estimation of fault detection is an important topic because, as in the case of DDWC, the mechanism found a fault just in the target inspection. Thus, during the in-field execution several parameters may affect the detection capabilities. The most important parameters are the switching, the detection time and the test patterns.

This latter element is a sequence of values applied to the inputs of target elements, SPs in this case, able to distinguish between the correctness or the faulty by excite a fault and propagate it to output.

Considering the FlexGrip model architecture, the streaming multiprocessor is not able to identify any permanent faults inside its n SPs. So, the detection capabilities is equal to zero. Instead using the DDWC the detection increase, and it could be estimated knowing the relation between the require time to detect a fault and the fault observability.

The fault observability ($Obs_{SP(p)}$) in one Scalar Processor can be defines as the ratio between the number of propagate test pattern and the number of all the test patterns (P), so the propagate (N_p) and not propagate (N_{np}) test pattern.

$$Obs_{SP(p)} = \frac{N_p}{N_p + N_{np}}$$

It could be clear that a fault (tt):

- arise in the system at time t=0;

- is excited by a pattern after a time t1;

- is propagated to the output after a time t2;

- is detected after a time t3

The time for fault detection (ETFD) in a continuous fault detection structure case is

$$ETFD_{DWC(t,p)} = \frac{N_p + N_{np}}{N_p} tt$$

The DDWC architecture strictly depends on:

- the time interval employed to switch among SP cores t4;

- the time required to execute the configuration instruction t5, proportional to the number of instructions between two consecutive configuration instructions.

so, it is possible to express the time for fault detection (ETFD) in the n SPs using the DDWC mechanisms as:

$$ETFD_{GPGPU(tt,p)} = \frac{(N_p + N_{np})(tt + \sum_{i=1}^{n}(t_4 + t_5))}{N_p}$$

This expression can be used to find an optimal trade-off between switching frequency for the DDWC mechanism among the SPs and the performance degradation in the application by the insertion of DDWC instructions.

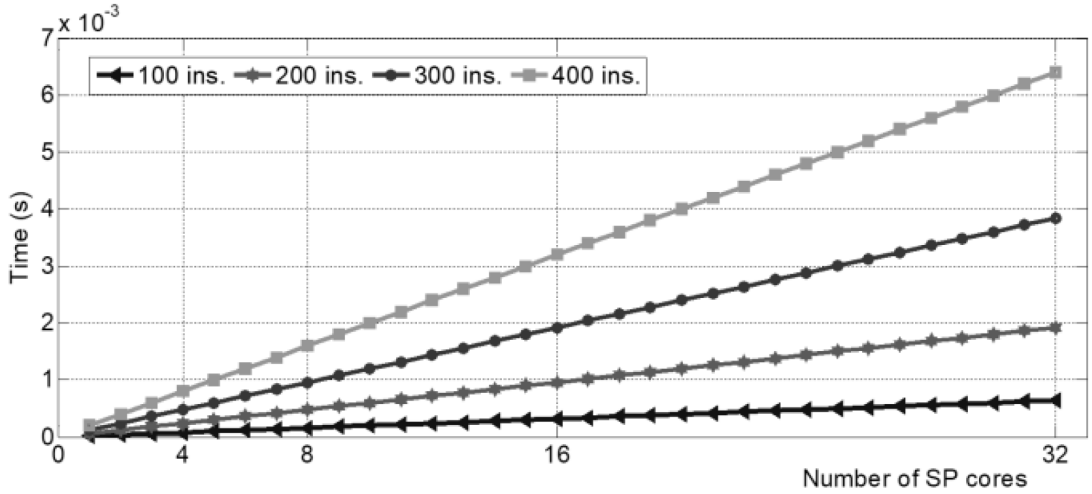

Figure 3.4. ETFD value for multiple frequencies of the DDWC_i instruction

Figure 3.4 shows the worst scenario for fault detection in an SP. Assumed that a fault can be propagated by one test pattern, the switch between the SP and the SSP is performed every 100, 200, 300 and 400 instructions in the program code. Using a reference clock period of 10ns, 20 clock cycles for instruction execution and fixed rate of 3 clock cycles for fault detection, the graph was evaluated. [10]

Chapter 4

BISR and DDWC

The third and last solution presented is the combined solution merging dynamic self-repairing configuration (BISR) and dynamic duplication with comparison (DDWC). The purpose of this architecture is to obtain a robust fault mitigation of fault accumulation in the SP cores and to increase the FlexGrip reliability.

4.1 Architecture

The BISR and DDWC mechanism is based on both the previous solutions, so on the possibility to use both BISR and DDWC at the same time.

In this architecture there is the possibility to define the number,from 1 up to 8, of cold stand-by modules (SSPs) in parallel with the existing SPs.

Even in this case the integrity of data-path is essential because a possible fault in the interconnections can not be discovered without the implemented architecture. So, the presence of two switching units structures, targeting the data-path for input and output interconnections in the SPs and one comparator, that perform the monitoring, is essential for the detection of faults.

The architecture implementation is performed in different part of the original design:

- In the Decode module: the two instructions, one for the BISR and one for the DDWC.
- In the Read module: the bypass signal for the next stage
- In the Execution module: the two switching units, some dedicated registers to store signals for BISR and DDWC instructions and for the flag of the comparator.

4.1.1 Master input switch

The aim of this block is to connect the correct data of the existing SPs to the SSPs based on the working instruction. In case just the BISR is active, the interconnection of the selected SPs is connected to the selected SSPs, for the DDWC, instead, the interconnections of the selected SPs is connected to the SSPs. When both the Instructions are switched on, the different interconnections of the two selected SPs are connected to two different SSPs. Thus, to perform this mechanism the presence of some multiplexers is essential:

- muxBISR: this multiplexer has for input all the interconnections for the SPs. The output is a string of bits containing SRC1, SRC2, SRC3 and Flags data. The control signal are the state of the BISR instruction (On/Off) and the selected channel from the BISR instruction. To be noticed that the dimension of this element is related to the configuration of the system, so for a 32 SP configuration is a 32to1 multiplexer, same for the 16 and 8 cases.

- muxDDWC: same as the muxBISR, but using the DDWC instruction bits.

- mux2: one for each SP, the first input is original data, while the second one is a string of all zeroes. This latter input is used to shut down the SPs and to save power reducing the switching of not used SPs, the faulty one. The output of these multiplexers is directly connected to the input of each SP. The control signal of this Structure will be described in chapter 4.1.3.

- mux3: one for each SSP, this special multiplexer output is directly connected to each SSPs, while it as three input: the output of muxBISR as first, the output of muxDDWC as second and the all zeroes string as third. The control signal of this Structure will be described in chapter 4.1.3.

Figure 4.1 show the complete architecture of the Master input switch. In the figure is possible to identify the input data channels SPC_{in} signals. The SPCs are composed of 32 bit-size input data operands (SRC1, SRC2, and SRC3) and the predicate flags (4 bit-size) coming from the previous pipeline stage.

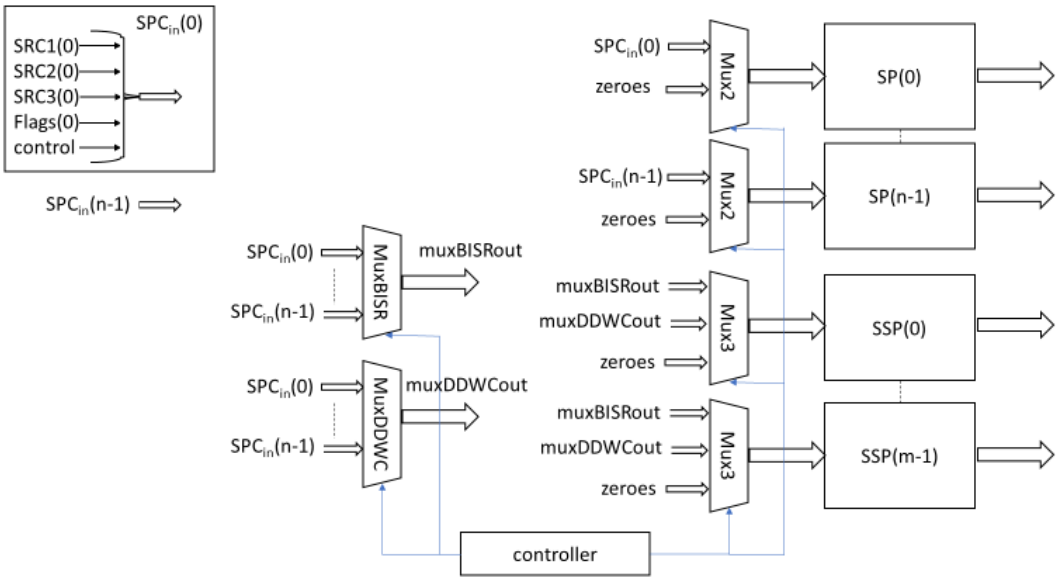

Figure 4.1. Master Input Switch

4.1.2 Master output switch

The aim of this block is to connect the output of SPs and SSPs with the next stage. Also in this architecture the presence of multiplexers is essential:

- muxToComparator: this multiplexer has for input all the Output signal for all the existing SPs and SSPs. The output of this block is used as element to be compared to performing the monitoring of the system.

- mux8BISR: this multiplexer has for input all the SSPs output $SSPC_{out}$, for the existing SSPs. The output of this element is connected to the Mux2 multiplexers. This purpose is to have the right interconnections when performing BISR. The controller of this multiplexer are the BISR state signal (On/Off) and the selected channel with the BISR instruction.

- mux8DDWC: the input signals are the same as the mux8BISR, but the output is the reference used to perform the comparison for monitoring the selected SPs with the DDWC instruction bits.

- mux2: one for each SP, the first input is the output data from each SP, while the second one is the output of the mux8BISR multiplexer. The output of these multiplexers is directly connected to the next stage. The control signal of this Structure will be described in chapter 4.1.3.

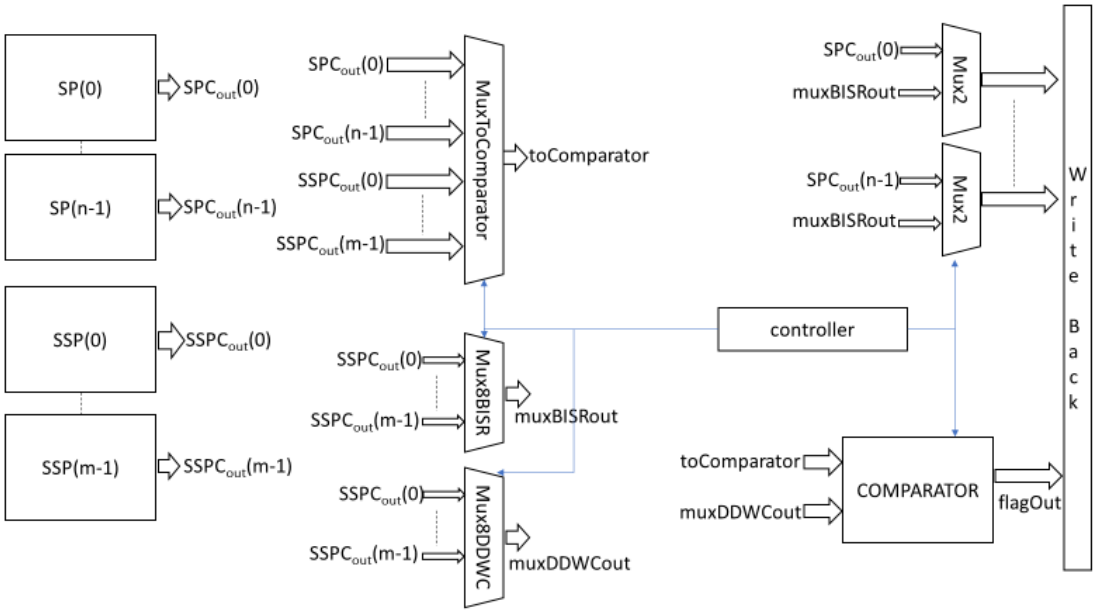

Figure 4.2. Master Output Switch

4.1.3 Switch Controller

The two switching units have one controller essential to perform both the BISR and the DDWC mechanisms.

The control bits of the controllers are coming from the instructions and store in dedicated registers inside the execution module. These bits are fed at the same time in both the switching units, so the functionality of the SPs plus SSPs continued.

Particularly the bits for the controller are the state of both the instructions (BISR on/off and DDWC on/off), the selecting SPs for each BISR and DDWC instructions, the selected SSP where the BISR or the DDWC are performed.

Input Controller

The controller output goes to all the multiplexer.

- MuxBISR: for this block the bits used are the BISR state and the selecting SP use for the BISR mechanism. Thus, the state activate the multiplexer, while the selecting SP is used to select the proper input channel.
- MuxDDWC: same as muxBISR, but using the DDWC instruction bits.
- mux2: the control bit of this multiplexer could be 0 or 1. If it is 0 the original input data goes to SPs, instead when 1 the zeroes goes to the SPs.

- no BISR and no DDWC, all muxs are fed by 0.

- BISR case, all the muxs are fed by 0, but the selected SP of the BISR instruction is fed by 1.

- DDWC case, all muxs are fed by 0.

- BISR and DDWC case, is the same case as BISR.

- mux3: the control bits of this multiplexer could be 00 for BISR, 01 for DDWC or 10 for zeroes. These bits are fed in the proper way depending on the selected SSPs of the instructions.

  - no BISR and no DDWC, all the muxs are fed with 10.

  - BISR case, all muxs are fed with 10, but the mux of the selected SSP is fed with 00.

  - DDWC case, same as BISR case, but the mux of the selected SSP is fed with 01.

  - BISR and DDWC case, it is no possible to use the same SSP for both the instructions, so depending on the selected SSP of the BISR and DDWC instructions the muxs are fed with 00 or 01 when activate. In case for some reason the instructions have the same selected SSP, all the muxs are fed with 10, so all zeroes enter the SSPs.

Output Controller

The controller output goes to all the multiplexer and to the comparator.

- comparator: The DDWC state bits goes to this block, so is possible to save power everytime the DDWC mechanism is not used.

- mux8BISR: for this block the bits used are the BISR state and the selecting SSP use for the BISR mechanism. Thus, the state activate the multiplexer, while the selecting SSP is used to select the proper spare channel.

- mux8DDWC: same as the mux8BISR, but using DDWC instruction bits.

- MuxToComparator: the control bits for this block is used to select the proper channel to be compared in the comparator, so the BISR and DDWC state, selected SP and the BISR selected SSPs is used to select it.

- mux2: the control bit of this multiplexer could be 0 or 1. If it is 0 the output data of each SPs goes to the next stage, instead when 1, the output of the BISR SSPs goes to the next stage.

  - no BISR and no DDWC, all muxs are fed by 0.

- BISR case, all the muxs are fed by 0, but the selected SP of the BISR instruction is fed by 1.

- DDWC case, all muxs are fed by 0.

- BISR and DDWC case, same as BISR case.

4.2 Instructions

With this architecture is possible to use at the same time both the BISR and DDWC mechanism. The difference between the two instructions in chapters 2.3 and 3.3 and the two used in this architecture are the purpose of the used bits.

For the BISR instruction, as is possible to see from Table 4.1 there are some differences compared to the Table 2.1. In this architecture it is possible to have from 1 up to 8 SSPs, while in the original design it could be possible to have 32 of them. Some motivation of the wrongness was debate in above chapter and taking into account for this implementation.

When we perform this method is possible to change the selected SP and SSP with one direct instruction in the application program each time, BISR Rx, Ry where Rx is the selected SP and Ry the selected SSP. For instance

BISR 5,0x20; with this operation the selected SP is the 5 and the selected SSP is the 32, so the first SSP.

Instead using the immediate is possible to have process with loop and in the program we change the variable contain the selected SP channel or the selected SSP. So, is possible to use the existing hardware of the FlexGrip to use the value store in the global memory.

Opcode	Not Used	Bypass	BISR_SP	BISR_SSP	immediate	Not Used	flow(1)	32bit long(0)
4bit	9bit	1bit	5bit	3bit	1bit	7bit	1bit	1bit
31-28	27-19	18	17-13	12-10	9	8-2	1	0

Table 4.1. BISR Instruction

The DDWC instruction has the same configuration as the BISR except for the opcode that is 1001 for BISR and 1011 for DDWC. So, what is is written above for the BISR instruction is valid also for the DDWC, what it changes is the meaning of the bits for each method and in the two switch controller.

Opcode	Not Used	Bypass	DDWC_SP	DDWC_SSP	immediate	Not Used	flow(1)	32bit long(0)
4bit	9bit	1bit	5bit	3bit	1bit	7bit	1bit	1bit
31-28	27-19	18	17-13	12-10	9	8-2	1	0

Table 4.2. DWC Instruction

4.3 Experimental results

This final part of this chapter is dedicated to the experimental result obtaining after the logic synthesis using Design Vision Tool.

4.3.1 Hardware overhead

Table 4.3 show the Area of the original design of FlexGrip. In the first column is possible to identify the different configuration for the SPs, in the middle column there is the total cell area of the design, without the memories, finally in the third column there is the cell size Of the FlexGrip Decode, Read and Execution modules.

SPs	Total Cell Area	Total cells in design
8	294170	123521
16	545908	229024
32	1065003	446799

Table 4.3. Original hardware for different configuration

Table 4.4 shows the area overhead of the FlexGrip plus BISR and DDWC implemented version.

It is possible to identify which is the cost to implement the instructions, the switching modules, with related controllers. It is shows for each configuration when there are 0 SSPs and it is just few percentage (0.16 - 0.42) in terms of area overhead, as well the total number of cells in the design is quite similar, less than 1%. To be noticed that for the 32 SP configuration there are less cells, so as mentioned this demonstrate that is the most optimize design between the proposed.

The hardware overhead for this configuration represents, depending on the configuration a small percentage of the whole hardware. For the 32 SP configuration the range of area overhead is from 2.33% up to 6.09%, this tell that the area overhead, most introduced by the spare SSPs modules, is limited compare to the high number of SPs. A similar situation is presented for the 16 SP configuration. Instead for the 8 SP case, the cost of the SSPs linearly grows with their number. In fact, the cost for one SSP core is 5229 cells, that introduces an area overhead greater then 14% in al Scalar processor configurations.

Observing the last two columns, instead, is possible to notice that the total number of cells that can be protected with this strategy ranges from about few percentages (3.92%) with one SSPs in the 16 SP conf. up to 26.25% in the worst case of 8 SP and 8 SSPs.

The optimum choice of both parameters depends on the design requirements. In any case, it is worth noting that the hardware overhead remains below 15% for

all the considered GPGPU configurations.

Comparing this data with Table 2.2, of BISR mechanism, it should be noticed that all the overheads are reduced, so this configuration perform same method using fewer area, but also it could perform the DDWC mechanism.

SPs	**SSPs**	**Total Cell Area**	**Area Overhead (%)**	**Total Cells in design**	**SP/SSP cells (%)**
	0	295171	0.34	124437	0.74
	1	302256	2.75	129666	4.97
	2	308066	4.72	134678	9.03
	3	312504	6.23	137627	11.42
8	4	316789	7.69	139886	13.25
	5	322181	9.52	144616	17.08
	6	327309	11.27	148205	19.98
	7	331839	12.80	150438	21.79
	8	337917	14.87	155941	26.25
	0	548191	0.42	231171	0.94
	1	556426	1.93	238000	3.92
	2	556439	1.93	239175	4.43
	3	568352	4.11	245571	7.23
16	4	573822	5.11	249843	9.09
	5	578750	6.02	253768	10.80
	6	583456	6.88	257213	12.31
	7	588662	7.83	260996	13.96
	8	595368	9.06	265496	15.92
	0	1066661	0.16	446027	-0.17
	1	1089812	2.33	466099	4.32
	2	1093656	2.69	467914	4.73
	3	1099485	3.24	472459	5.74
32	4	1106327	3.88	477684	6.91
	5	1113275	4.53	481885	7.85
	6	1118950	5.07	485506	8.66
	7	1123588	5.50	486158	8.81
	8	1129828	6.09	491657	10.04

Table 4.4. Hardware overhead for the fault tolerance implemented version

4.3.2 Power Performance

The power overhead of the BISR plus DDWC mechanism theoretically is bigger than the single BISR or DDWC methods, but also in this case the SSPs act as cold stand-by modules if not used. The presence of comparator should increase the switching activity.

Clearly in a real implementation only static power by leakage current is consumed during operation and it depends on the transistor technology. Thus, the final power overhead of the BISR strategy is negligible in comparison with the dynamic power consumption produced in the GPGPU.

4.3.3 Performance overhead

Table 4.5 shows the performance overheads in all the three synthesized modules: Decode, Read and Execution.

The additional structure in the Decode module increase the performance degradation of almost 14-15% in all the configurations.

The insertion of the bypass in the Read module does not introduce any delay or performance degradation.

In contrast, in the execution module, the performance degradation seems to varying for each SSP insertion. This problem is related to the logic synthesizer that every time performs the synthesis on the "fastest" critical path for him.

Performing the average on the performance degradation for each configuration, we discover that the the average performance degradation is approximately 5.14%.

Modules	SPs	SSPs	Time Delay critical path(ps)		Performance
			FlexGrip	Fault Tolerance	Degradation (%)
Decode	8	-	0.93	1.06	13.98
	16	-		1.06	13.98
	32	-		1.07	15.05
Read	8	-	3.7	3.62	-2.16
	16	-	3.89	4.07	4.63
	32	-	3.03	3.03	0.00
Execution	8	0	7.4	7.36	-0.54
		1		7.58	2.43
		2		7.52	1.62
		3		7.81	5.54
		4		8.07	9.05
		5		7.96	7.57
		6		8.07	9.05
		7		8.01	8.24
		8		8.1	9.46
	16	0	7.81	7.28	-6.79
		1		7.93	1.54
		2		7.47	-4.35
		3		8.72	11.65
		4		8.44	8.07
		5		8.88	13.70
		6		8.29	6.15
		7		8.5	8.83
		8		8.1	3.71
	32	0	7.85	8.25	5.10
		1		8.18	4.20
		2		8.02	2.17
		3		8.04	2.42
		4		8.25	5.10
		5		8.37	6.62
		6		8.02	2.17
		7		8.68	10.57
		8		8.3	5.73

Table 4.5. Performance overhead for the fault tolerance implemented version

Chapter 5

Conclusions

The object of this work was to implemented different solutions to increase fault mitigation and reliability for the reference GPGPU FlexGrip model.

To achieve this results three solutions have been developed. All the proposed mechanisms targets the permanent faults that may arise in the SPs.

The first solution, the Built-In Self Repair (BISR), is based on the addition on some cold stand-by spare modules (SSPs) in parallel with the existing Spare Cores (SPs). The importance of the data-path interconnections is crucial for this mechanism. The insertions of two switching units, one for the input and the other one for the output, controlled by a new Instruction specifically created aim to substitute the faulty SP with one SSPs.

This strategy is performed on the device power-on or reset, so the fault detection and location phases, as well as the reconfiguration one, can be executed without any strict time and memory constraints.

Experimental results show that with this solution the hardware cost is in the range from 0.8% to 1.7%, while the cost of the SSPs linearly grows with their number. In fact the cost for one SSP core is 6623 cells, that introduces an area overhead greater than 3% in all Scalar processor configurations. The optimum choice of both parameters depends on the design requirements. In any case, it is worth noting that the hardware overhead remains below 20% for all the considered GPGPU configurations.

Also, considering performance compare to the original solution, results show that the performance degradation reach up to 20% for a large number of SSPs (6 or 7). This is mainly caused by the logic control of the two switching modules.

The second solution, the dynamic duplication with comparison (DDWC), is based on the addition of one spare module (SSP) and a comparator. The aim of this mechanism is to harden the Scalar Processor units and also in this case the integrity of data-path is crucial. The insertion of two selector switch, one for

the input and one for the output of the SPs, is controlled by a new instruction specifically created.

This implemented DDWC structure is intended to be used during the in-field operation of the GPGPU. So the redundant core should swap among the SPs of the Streaming Multiprocessor.

Experimental results show that the area overhead is relatively low, below the 3% for all SPs configurations. These results support the initial intention of limiting the impact in the hardware overhead by the DDWC structures.

The performance overhead was done looking at the critical path delay. For a low number of SP cores, the critical delay path is increased by 7.52%. The configuration of 16 SP present the maximum percentage of performance overhead with 14.79%, while, in contrast, the overhead drops for the 32 SP cores (13.56%). However, the most representative timing effects are due to the added modules in the path. These modules are the input and output switches and the comparator.

The third solution aims to obtain a robust fault mitigation mechanism, based on the merging of the previous two solutions. Thus, is possible to use both the BISR and the DDWC at the same time. In this architecture it is possible to define the number of SSPs, from 1 to 8, in parallel with the existing SPs.

Two master switching units is used to connect the data-path for input and output interconnections in the SPs. An instruction allows to control the faulty SP and substitute with a SSP and monitoring the selected SPs/SSPs, with the presence of a cold stand-by comparator.

Also in this case the second aim was to verify the area and performance overheads. So experimental results show that in the worst scenario, so with 8 SSP, for all the configurations, there is a moderate area overhead 14.8% in the for the 8 SP case, while it drops for the other two configuration, both in the worst scenario, 9% for 16 SP and 6% for 32SP.

Instead in the performance degradation, considered the critical path delay, we obtained that for the configuration with 8 SP the maximum critical delay is increased by 9.54%, 13.7% for the 16 SP configuration, while in contrast just 6.62% for the 32 SP case.

To sum up, different approaches have been implemented and with the experimental data is possible to say that paying a limited area overhead and few performance degradation is possible to increase the FlexGrip GPGPU reliability.

Bibliography

[1] K. Andryc, M. Merchant, R. Tessier, *FlexGrip: A Soft GPGPU for FPGAs.*

[2] G. Roascio, *"Analysis and extension of an open-source VHDL model of a General-Purpose GPU*, M.S. Thesis, Politecnico di Torino, Torino, Oct 2018.

[3] J. E. Rodriguez Condia, M. Sonza Reorda, *An extended GPGPU model to support detailed reliability analysis.*

[4] T. Koal, D. Scheit, H. T. Vierhaus, *A Concept for Logic Self Repair.*

[5] T. Koal, M. Ulbricht, H. T. Vierhaus, *Combining Fault Tolerance and Self Repair in a Virtual TMR Scheme.*

[6] T. Koal, H. T. Vierhaus, *Logic Self Repair Based on Regular Building Blocks.*

[7] J. E. Rodriguez Condia, P. Narducci, M. Sonza Reorda, L. Sterpone, *A dynamic reconfiguration mechanism to increase the reliability of GPGPUs.*

[8] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, R. Gupta, *Compiler Techniques to Reduce the Synchronization Overhead of GPU Redundant Multithreading.*

[9] J. W. Sheaffer, D. P. Luebke, K. Skadron, *A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors.*

[10] J. E. Rodriguez Condia, P. Narducci, M. Sonza Reorda, L. Sterpone, *A dynamic hardware redundancy mechanism for the in-field fault detection of cores in GPGPUs.*