32-bit Graphics Processing Unit

A Project Report

submitted by

D V H PHANI TEJA

(EE14M053)

in partial fulfilment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

in

Microelectronics & VLSI Design

Under the guidance of

Prof V . Kamakoti

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

JUNE 2016

THESIS CERTIFICATE

This is to certify that the thesis titled 32-bit Graphics Processing Unit, submitted by D V H PHANI TEJA with roll number EE14M053, to the Indian Institute of Technology Madras, for the award of the degree of Master Of Technology in Microelectronics & VLSI Design, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof Dr V. Kamakoti

Dept. of Computer Science
IIT-Madras, 600 036

Date: 17th June 2016 Place: Chennai

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my guide, Prof Dr. V.Kamakoti for his valuable guidance, encouragement and advice. His immense motivation helped me in making firm commitment towards my project work.

My special thanks to Mr. G.S.Madhusudan for his encouragement and motivation throughout the project. His valuable suggestions and constructive feedback were very helpful in moving ahead in my project work.

I would like to thank my co-guide Dr.Nitin Chandrachoodan.

I would lie to thank my faculty advisor Prof. Dr. Deleep R Nair, who had patiently listened, evaluated, guided me throughout the program.

My special thanks to my project team members Neel Gala, Arjun Menon, Rahul, Gopi for their help and support.

ABSTRACT

KEYWORDS: highly parallel architecture, demand of parallel computations, choosable over CPU

The graphics processing unit(GPU) has become an integral part of today's mainstream computing systems. The highly parallel architecture of GPU makes it to feature large number of peak arithmetic operations in parallel, substantially than it's counterpart CPU. The rapid increase in the demand of parallel computations in specific applications has made GPU a powerful computational engine and choosable over CPU for such applications.

This thesis describes the background, hardware, and programming model in designing the basic GPU architecture, summarize the state of the art in tools and techniques that deliver order-of-magnitude performance gains over optimized CPU applications.

Table Of Contents

ACKNOWLEDGEMENTS iii

ABSTRACT iv

LIST OF FIGURES vii

ABBREVIATIONS viii

1 INTRODUCTION 1

2 BACKGROUND 2

2.1 SIMT 2

2.2 GPU Execution 2

2.3 Streaming Multiprocessor (SM) 4

2.3.1 Stream Programming Model 5

2.3.2 Instruction Cache 7

2.3.3 Warp Scheduler 7

2.3.4 Cores 9

2.3.5 D-cache 9

2.3.6 Load/Store unit 9

2.3.7 Register File 10

2.3.8 Special Function Unit 10

3 IMPLEMENTATION DETAILS 11

3.1 Data 11

3.2 Instruction Details 12

3.3 Load Instruction 15
3.4 Arithmetic Instructions 16
3.5 Store Instruction 18

4 LIST OF SIGNALS 20

5 DESIGN FLOW & RESULTS. 23
5.1 Hardware Design Flow 23
5.2 Implementation, Simulation 25

6 CONCLUSION & FUTURE WORK 29
6.1 Conclusion 29
6.2 Future Work 29

LIST OF FIGURES

2.1 SIMT 2

2.2 GPU internal blocks 3

2.3 Streaming Multiprocessor 4

2.4 Warp Scheduler 7

2.5 Instruction Scheduler 8

5.1 Hardware Design Flow 24

5.2 Simulation Result screenshot 1 26

5.3 Simulation Result screenshot 2 27

5.4 Simulation Result screenshot 3 28

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
GPU	Graphics Processing Unit
SM	Streaming Multiprocessor

CHAPTER 1

INTRODUCTION

There are various applications that require a 3D world to be simulated as realistically as possible on a computer screen. These include 3D animations in games, movies and other real world simulations. It takes a lot of computing power to represent a 3D world due to the great amount of information and the complex mathematical operations that must be used to project this 3D world onto a computer screen. In this situation, the processing time and bandwidth are at a premium due to large amounts of both computation and data.

The functional purpose of a GPU then, is to provide a separate dedicated graphics resources, including a graphics processor and memory, to relieve some of the burden off of the main system resources, namely the Central Processing Unit, Main Memory which would otherwise get saturated with graphical operations and I/O requests. The abstract goal of a GPU, however, is to enable a representation of a 3D world as realistically as possible. So these GPUs are designed to provide additional computational power that is customized specifically to perform these 3D tasks.

Furthermore, GPUs have moved away from the traditional fixed-function 3D graphics pipeline toward a flexible general-purpose computational engine. Today, GPUs can implement many parallel algorithms directly using graphics hardware.

CHAPTER 2

BACKGROUND

2.1. SIMT:

Single instruction, multiple thread (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multi threading.

Fig. 2.1. Group of threads executing at once on a single instruction independently

This is achieved by each processor having multiple "threads" (or "work-items"), which execute in lock-step. Access time of all the widespread RAM's is still very low, that inevitably comes with each memory access.

The SIMT execution model has been implemented on several GPUs and is relevant for general-purpose computing on graphics processing units (GPGPU).

2.2. GPU Execution:

A GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or

more thread blocks; and cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. This can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.

A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls.

The GPU internal blocks are as shown below:

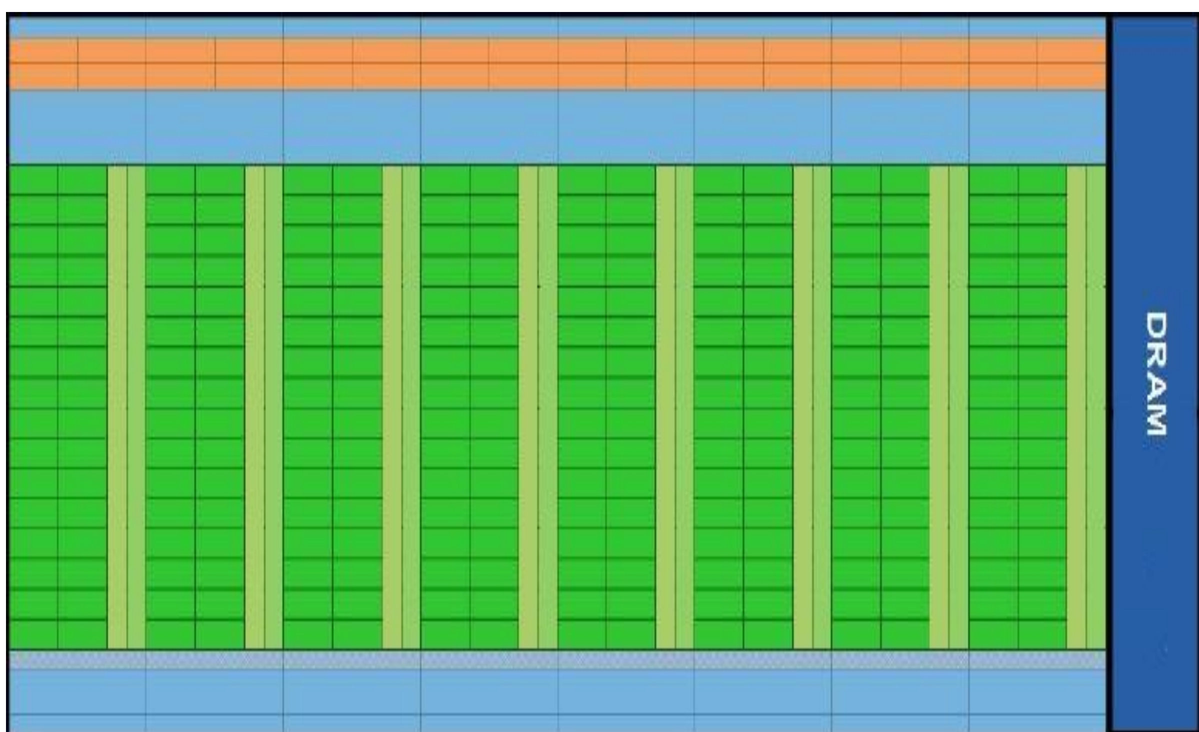

Fig. 2.2. GPU internal blocks with different colours indicating separate units

It has 8 Streaming Multiprocessors (SMs) and a DRAM. Each SM is described as in next sections.

2.3. Streaming Multiprocessor (SM):

This GPU will have 8 Streaming Multiprocessors (SMs). Each SM will have the following
1 Warp Scheduler, 1 Dispatch Unit , 1 I-Cache, 1 D-Cache, 32 ALU cores, 1 Load/Store unit, 1 Special Function Unit, 1 Register file.

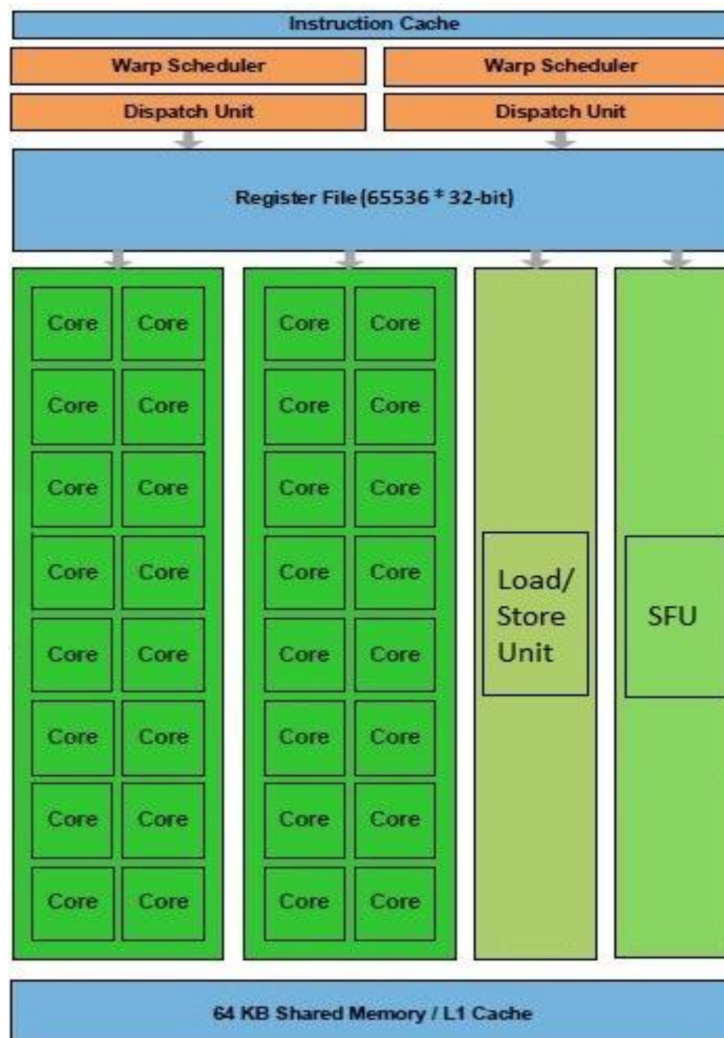

Fig. 2.3. Streaming Multiprocessor block diagram

Stream processors belong to a new class of architectures that are compute intensive, and achieve high performance by reducing dynamic control, and providing a large number of programmable functional units. Stream processors are specifically designed for the stream execution model, in which applications have large amounts of explicit parallel computation, structured and predictable control, and memory accesses that can be performed at a coarse granularity. Hence, their architecture also takes into account the concurrency and communication levels defined by the stream programming model in order to maintain high locality and parallelism levels. Specifically, their memory hierarchy is partitioned into three distinct levels of storage. Each of the memory levels provides an order of magnitude more bandwidth as it gets closer to the functional units. This maintains temporary data close to the functional units while only true global data are stored in the external memory. Because of the efficient bandwidth use, most of the work of a stream processor is done on-chip with only 1% of global data references requiring external memory access. The three levels of memory hierarchy are given below:

Local Register File (LRF): Used for local data communication and fast access of temporary data by the functional units.

Stream Register File (SRF): Used to store streams and transfer data between the LRFs of major components.

Off-Chip Memory: Stores global data and is used only when necessary.

As with the memory hierarchy the functional units of a stream processor are distributed in ALU Clusters. All clusters consist of the same type and amount of functional units. The ALU clusters execute kernel instructions that they receive from the μController in SIMD fashion. Each of the clusters operates on an element of the input stream, thus, providing data parallelism. The SIMD organization helps provide the necessary data bandwidth to feed all ALU clusters.

2.3.1. Stream Programming Model :

The stream programming model arranges applications into a set of computation kernels that operate on data streams. Expressing an application in terms of the stream programming model exposes the inherent locality and parallelism of that application, which can be efficiently handled by appropriate hardware to speed-up parallel applications. By using the stream programming model to expose parallelism, producer-consumer localities are revealed between kernels, as well as true data localities in applications. These localities can be exploited by keeping data movement locally between kernels that communicate which are more efficient rather than using global communication paths.

- *Streams:* A collection of data records of the same type, ranging from single numbers to complex elements.

- *Kernels:* Operations that are applied on the input stream elements. Kernels can perform simple to complex computations and can have one or more input and output streams.

The advantage of expressing the application in the form of a stream program is that it exposes two types of localities. The first is kernel locality. During a kernel execution, all references are to variables local to the kernel, except the values read from the input stream or written to the output stream. The second is producer-consumer locality, which involves streams moving between kernels. One kernel produces a stream which another kernel in the immediately consumes. By efficiently sequencing such kernels, the stream values are kept local and are consumed soon after they are produced. The stream model ensures that kernel programs will never access the main memory directly. The stream programming model defines communication and concurrency between *streams* and *kernels* at three distinct different levels. In this way take the locality and parallelism of the application are exposed. These restrictions in communication help in the most efficient use of bandwidth.

Communication:

- *Local:* Used for temporary results produced by scalar operations within a kernel.

- *Stream:* For data movement between kernels. All data are expressed in the form of streams.

Concurrency:

- *Instruction Level Parallelism (ILP):* Parallelism exploited between the scalar operations within the kernel.

- *Data Parallelism:* Applying the same computation pattern on different stream elements in parallel.

- *Task Parallelism:* As long as no dependencies are present, multiple computation and communication tasks can be executed in parallel.

2.3.2. Instruction Cache:

The instruction cache contains the instruction words. The address in the I-cache where the kernel starts, is given by the micro-processor to the GPU as an input. Basing on this address, GPU will start the process of fetching kernel's first instruction word from the I-cache.

2.3.3. Warp Scheduler:

The processor knows the information about number of elements it needs to be operated on, in total and block size. So basing on this it calculates the number of warps per block. Suppose, the number of data elements are 50 and the number of core units in SM is 32, then it has 2 warps with each warp as size of the number of core units i.e. 32.

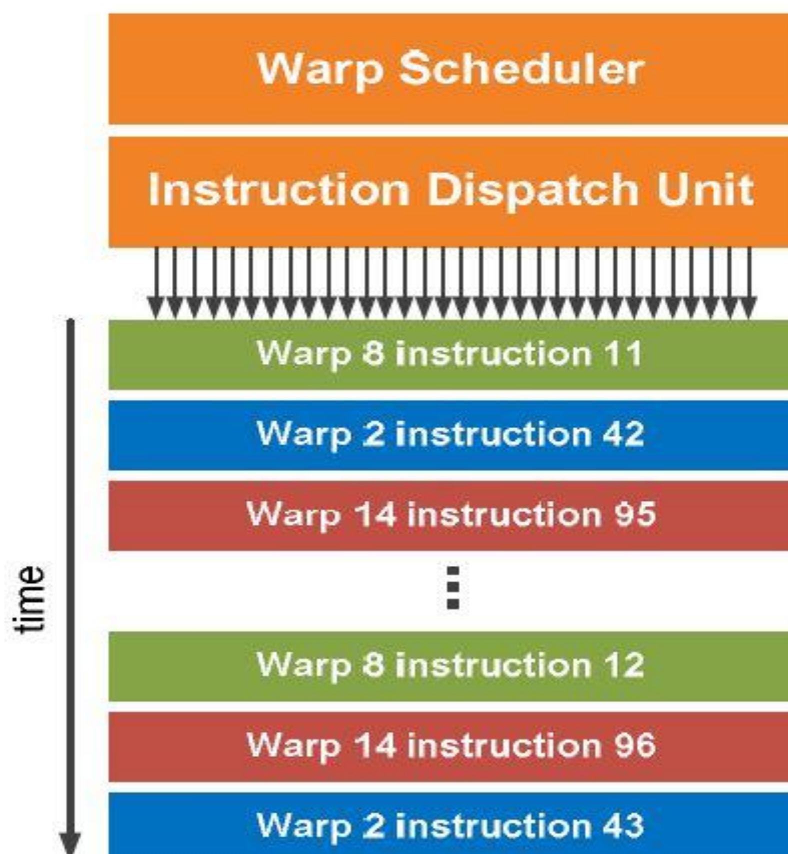

Fig. 2.4. Warp scheduler

The warps for instruction are issued as per priority or due to the processing of background work for previous warps. So, in order to hide latency, the SM will switch the warp if current warp got stalled due to any background task (such as data fetching from memory). In this case, the warp which got stalled has it's own memory such that it can run in background once it is initialized. So, the functioning units which are doing no-operation due to stall can be utilised by throwing another warp onto them. This way hardware will be utilized always to their peak performance.

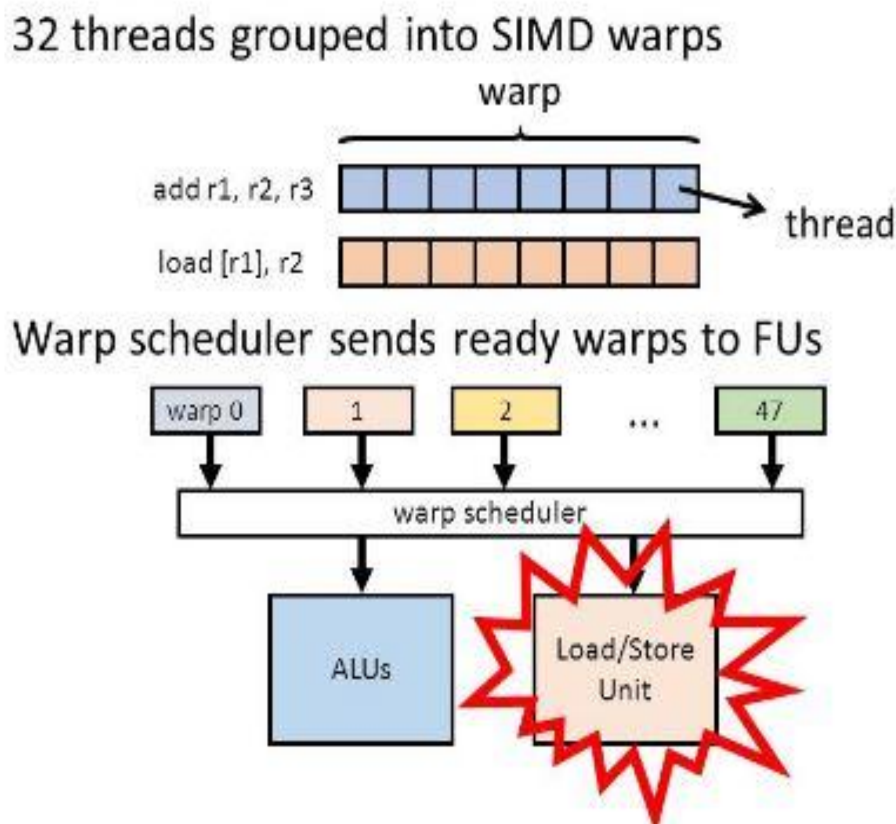

Fig. 2.5. Instructions getting scheduled to different blocks independently

Warp scheduler sends the warps which are ready to their respective functional units. For this to happen, the corresponding warps should have both data and functional units to be available.

2.3.4. Cores :

There are 32 ALU clusters in each Streaming Multiprocessor (SM) and each is comprised of arithmetic logic. The clusters operate in a 32 wide SIMD manner. That is, the same instruction is executed on all 32 clusters on different elements i.e., when an instruction is fetched and passed on, to do some arithmetic logic, the clusters will do the same arithmetic logic on the corresponding registers of all the respective clusters which were loaded with data in their registers.

Each SM has 32 cores. There are 8 SMs. So in total there are 256 cores on the chip with each core having it's own register file (per warp).

2.3.5. D-Cache :

The Data cache contains the data words. The first element address of the data in the D-cache is given in the Load instruction word and this word is decoded to get the address which is then sent to the load/store unit(L/S). L/S unit fetches the data from the D-cache. All the data elements are stored in the consecutive memory locations in the cache.

2.3.6. Load/Store Units:

There is 1 load/store unit per Streaming Multiprocessor (SM). This unit will get the address by decoding the incoming Load or Store instructions from the I-cache. This decoded input is then used to produce the addresses of the next data elements to be fetched for the cores in while on load instruction, or to produce the addresses of the next locations to store the data results of the cores in the consecutive locations of the D-cache while on store instruction. All the data elements are placed in consecutive memory locations. So the L/S unit needs a starting address for the fetching or storing the data.

Load/Store units will generate the addresses required for fetching or storing data from the memory. Once it gets the starting address it will increase the address value by one unit for every clock cycle and send it to the memory address lines for fetching/storing data.

2.3.7. Register File:

There are 32 cores in each Streaming Multiprocessor (SM). The block size for each SM is 256. So there can be eight warps per block per SM. When a warp is executed, it involves execution of 32 data pairs on their respective 32 cores at an instant. So for each core per thread, there should be one set of 32 registers. Likewise in a warp execution there is a need for 32 sets of register files each with 32 registers in number.

So as per above analysis, per warp there are 1024 registers which are required for the execution of a warp. Likewise there can be eight warps per block i.e., each core needs eight sets of 32 set registers. So there should be 8192 registers per SM. Now there are 8 SMs. So in total 65536 register with each register as 32-bit size are required for execution of SMs.

The register file in a GPU is usually very large in size when compared to a CPU. This is because of the extreme data parallelism and execution in the GPU with a relatively large number of core in a GPU than in the CPU. This extreme data parallelism also requires more data elements to flow in at any instant which serves the purpose of having relatively large register file.

2.3.8. Special Function Units :

There is one special function unit in each of the SMs. This special function unit does different operation than the regular core units. These take multiple clock cycles to complete the special operation relative to an arithmetic operation.

The special function may include square root finder or cosine or sine function etc kind of operations. This unit is separated from the core unit block such that these operations can be done in parallel to the operations which will be happening in core unit such that both arithmetic and special operations are done in parallel on different data elements.

CHAPTER 3

IMPLEMENTATION DETAILS

3.1. Data:

Block: A group of data elements which are corresponding to a single Streaming Multiprocessor (SM).

Each block will have a block ID.

Warp: The Streaming Multiprocessor (SM) executes threads in groups of 32 threads called a warp.

Each warp will have a warp ID.

The processor gives the information about address of the first instruction in the kernel(in I-Cache) and the number of data elements to be processed in total. This information is passed into the graphics processor as input word.

This word which graphics processor received is decoded into address of the first kernel instruction and information about number of data elements (say ‘n’) to be processed. The number ‘n’ is then used to find the number of blocks of data and warps of data in each block. Each block is then launched onto each streaming multiprocessor.

Each element will also be having a thread ID. These number’s block ID, warp ID and thread ID are used to locate any data element uniquely.

After receiving the word from the processor, it is decoded into 2 parts.

- ➔ Lower 16 bits gives the address of the instruction in the I-cache.
- ➔ Upper 16 bits give the information about the total number of data elements (n).

So upper 16 bits are processed to extract the block, warp and thread IDs.

Lower 16 bits are then passed into the I-Cache block. When the clock event happens, the instruction word is fetched from the I-cache block and passed back to the main module SM.

The instruction word length is 20 bits in size. The instructions are as shown below:

3.2. Instruction Details:

ADD →

X	DEST. REG	SOURCE REG 2	SOURCE REG 1	OPCODE

OPCODE BITS → 0th -3rd bits

SOURCE REG 1 is 5 bit data → 4th -8th bits

SOURCE REG 2 is 5 bit data → 9th -13th bits

DESTINATION REG 1 is 5 bit data → 14th -18th bits

19th bit is a don't care bit.

SUB →

X	DEST. REG	SOURCE REG 2	SOURCE REG 1	OPCODE

OPCODE BITS → 0th -3rd bits

SOURCE REG 1 is 5 bit data → 4th -8th bits

SOURCE REG 2 is 5 bit data → 9th -13th bits

DESTINATION REG 1 is 5 bit data → 14th -18th bits

19th bit is a don't care bit.

BITWISE AND →

X	DEST. REG	SOURCE REG 2	SOURCE REG 1	OPCODE

OPCODE BITS → 0th -3rd bits

SOURCE REG 1 is 5 bit data → 4th -8th bits

SOURCE REG 2 is 5 bit data → 9th -13th bits

DESTINATION REG 1 is 5 bit data → 14th -18th bits

19th bit is a don't care bit.

BITWISE OR →

X	DEST. REG	SOURCE REG 2	SOURCE REG 1	OPCODE

OPCODE BITS → 0th -3rd bits

SOURCE REG 1 is 5 bit data → 4th -8th bits

SOURCE REG 2 is 5 bit data → 9th -13th bits

DESTINATION REG 1 is 5 bit data → 14th -18th bits

19th bit is a don't care bit.

REGISTER INCREMENT →

X	REG	XXXX	OPCODE

OPCODE BITS → 0th -3rd bits

4th -13th bits are don't care

REGISTER is 5 bit data → 14th -18th bits

19th bit is a don't care

LOAD →

X	DEST. REG	SOURCE ADDRESS	OPCODE

OPCODE BITS → 0th -3rd bits

SOURCE ADDRESS is 10 bit data → 4th -13th bits

DESTINATION REG is 5 bit data → 14th -18th bits

19th bit is a don't care

STORE →

X	SOURCE REG	DESTINATION ADDRESS	OPCODE

OPCODE BITS → 0th -3rd bits

DESTINATION ADDRESS is 10 bit data → 4th -13th bits

SOURCE REG is 5 bit data → 14th -18th bits

19th bit is a don't care

Now the fetched instructions are passed into the SM’s main module where they are decoded to do the corresponding operation.

3.3. Load Instruction:

The opcode for the load instruction will be checked and the load operation process will start. The instruction word is then passed into the loadunit.v module where the instruction word is decoded. The instruction word syntax is as below:

X	DEST. REG	SOURCE ADDRESS	OPCODE

OPCODE BITS → 0th -3rd bits

SOURCE ADDRESS is 10 bit data → 4th -13th bits

DESTINATION REG is 5 bit data → 14th -18th bits

19th bit is a don’t care

In the loadstoreunit.v module, the source address is extracted from the instruction word and passed to load_address signal. This load_address value is copied into dM_address signal for first time when the instruction is called.

There is a count register which counts the number of clock pulses This loadstoreunit.v module will generate 32 different addresses one per every clock cycle and pass them onto dataMemory.v module . All these addresses are generated from the initial incoming address and the count register used to count clock pulses. So, for every clock pulse, a different address is generated and passed on. This dM_address signal is output of loadstoreunit.v module. This signal is taken as input by dataMemory.v module.

When the instruction is fetched into the main module, it is also passed into the controlLogic.v module. This module generates all the control signals required for the load operation to get completed. Memtoreg and regwrite are two important control signals which

allow enable signals for memory to register transfer if data andwriting the data into the specified register respectively. The PCwrite control signal is activated only when L/S unit finishes loading all the 32 elements data into their registers.

The dataMemory.v module will capture the incoming address on its lines. The DM_Read signal is active which makes the dataMemory.v retrieve the data from the given address on it's input lines of DM_Address signal. Data will be placed onto temp_dM_read line. This is read by dM_Read signal and placed in the data incoming signal of the register file and the data is written into the register specified in the instruction word. So, at the end of load instruction, from the given syntax, the data will be read and placed in the destination register specified.

For the time being the caches and other memories are implemented using register arrays only. So, the cache modules are implemented using register arrays of size which are sufficient enough to hold the 256 array A elements, 256 array B elements and 256 result array elements. So around 1024 elements can fit into the caches. So, the address field required is only around 10 bits.

3.4. Arithmetic Instructions:

The general syntax for arithmetic instructions is as shown below:

X	DEST. REG	SOURCE REG 2	SOURCE REG 1	OPCODE

OPCODE BITS → 0th -3rd bits

SOURCE REG 1 is 5 bit data → 4th -8th bits

SOURCE REG 2 is 5 bit data → 9th -13th bits

DESTINATION REG 1 is 5 bit data → 14th -18th bits

19th bit is a don't care bit.

Instruction fetched from the I-cache is passed into the topmodule.v where the

instruction word is decoded. The opcode for the arithmetic instruction will be checked and the operation process will start.

The registers information is retrieved from the respective places of the instruction word. The registers which are supposed to be added(say add is the arithmetic operation to be done for instance) and the contents are to be placed into the destination register for all the cores, this process is done in a single clock cycle.

The controlsignals which are needed for the arithmetic operations to carry out will be generated from the controlLogic.v module.

The function which needs to be done in the arithmeticLogic.v module is passed into it. Basing on the switch-case statement, corresponding operation is done and the results are placed in the result signal which is output of the arithmeticLogic.v module. This result is connected to the input data signal of the register files. So when the operation is completed, the results will get stored into the corresponding registers whose destination value is already fetched from the instruction word.

Register file and core units are generated using genvar statement. This will help duplicating the modules multiple times instead of writing the module instances again. In this process the corresponding signal lines between register files and core, load-store signals are also linked internally.

The CPU takes multiple clock cycles for loading the data and then adding them into the third register. But GPU will takes multiple clock cycles for loading the data but once the data is loaded, all the cores will do the arithmetic operation at the same time saving a lot of clock cycles time.

The selection of data between the result from loading the instruction and core operation results is done by signal memtoreg. This ensures the data which needs to be written into register files.

Similar to load instruction, the Store instruction does the work of storing the results from core units registers into address in the memory.

3.5. Store Instruction:

The opcode for the store instruction will be checked and the store operation process will start. The instruction word is then passed into the loadstoreunit.v module where the instruction word is decoded. The instruction word syntax is as below:

X	SOURCE REG	DESTINATION ADDRESS	OPCODE

OPCODE BITS → 0th -3rd bits

DESTINATION ADDRESS is 10 bit data → 4th -13th bits

SOURCE REG is 5 bit data → 14th -18th bits

19th bit is a don't care

In the loadstoreunit.v module, the source address is extracted from the instruction word and passed to load_address signal. This load_address value is copied into dM_address signal for first time when the instruction is called.

There is a count register which counts the number of clock pulses This loadstoreunit.v module will generate 32 different addresses one per every clock cycle and pass them onto dataMemory.v module . All these addresses are generated from the initial incoming address and the count register used to count clock pulses. So, for every clock pulse, a different address is generated and passed on. This dM_address signal is output of loadstoreunit.v module. This signal is taken as input by dataMemory.v module.

The dataMemory.v module will capture the incoming address on its lines. The DM_MW i.e., signal which corresponds to writing of data into the memory is active which makes the dataMemory.v to read the data present in it's input lines of DM_write signal and write it into the address specified by the input address lines of dM_address signal. The content of resister which needs to be stored in memory is loaded into DM_Write signal and then write process happens.

So, at the end of store instruction, from the given syntax, the data will be written

into the destination address specified in the instruction word for all the particular registers from the register files.

The operation flow for other instructions also will be in the same manner as described above.

CHAPTER 4

LIST OF SIGNALS

data_IN: This signal is 32 bits in length. Upper 16 bits gives the number of elements which are to be processed. Lower 16 bits gives the address location of the first instruction of the kernel in the I-cache.

i_address: This stores the decoded address data i.e., lower 16 bits of data_IN signal.

elements_num: This signal stores the number elements to be processed i.e., upper 16 bits of the data_IN signal.

blocks_num: It stores the value of number of blocks of data.

warps_num: It stores the value of number of warps of data on each streaming multiprocessor.

Clk: It is the clock for the processor.

instruction_mem: It is wire type signal which has instruction word data and is the output of I-Cache module.

Instruction: It acts as register in the code which stores the fetched instruction.

Selr1, Selr2: Each register file has two read ports. These selr1, selr2 registers content gives the information about the registers which are to be read from the register file..

Selrin: The data in it gives the information about the selection of register whose content should get updated by the data in the input of register file.

r1, r2: These are two dimensional registers. Each of them are an array of registers in which each array has registers of 32 bit in length. r1, r2 array size is equal to warps_num data. So, all the r1 array's registers point to same register in all the different streaming processor's register files. Similarly r2.

rin: This is a two dimensional register. It is an array of registers which are 32 bit in length

each. This holds the data which needed to be written into the register file. The selection of register is done by selrin, and the data needs to be written is in the rin register.

result_1: This holds the information of the result of arithmetic operation done by the core.v module and passed onto the topmodule.v

func: This gives information about the type of operation that needs to be carried out on the data elements.

PCWrite: This says the program counter when it has to be incremented to load the address of the next instruction from the cache.

Jump: It says if the jump operation is required or not for that instruction to get executed.

MemtoReg: It says if there is a need for memory to register transfer of data. This signal can be used as an enable signal and selection signal to select between dM_read and result_1 signals.

MemWrite: This says if there is a need for memory writing operation in that instruction's execution.

ALUControl: This give the information about the type of arithmetic operation need to be performed by the cores on the registers selected by selr1, selr2.

RegWrite: This says if the register write operation needs to be carried out in the process.

Done: This signal halts the processor after the execution of a kernel.

Some of these control signals can be used as enable signals for performing some operations in the processor's code.

dM_address: This is local signal to the topmodule.v which holds the address information of the memory location from which reading or to which writing needs to take place.

dM_write: This is local signal to the topmodule.v which holds the data that needs to be written into the memory location specified by dM_address signal

MemWrite: This signal gives the information if the data needs to be written or to be read from the memory. It is write enable signal.

temp_dM_read: This is local signal to topmodule.v which holds the read data from the memory, temporarily, before placing into one of the input lines of the register files data lines.

dM_read: It is two dimensional register which stores the value that we have got from the data memory by load operation and needed to be written into register file into the register specified by the selrin.

cnt: This signal holds the information of the number of clock pulses. Depending on this cnt signal, temp_dM_read data is placed into one of the dM_read array's registers that will later be written into register file's registers.

enable: It is the signal that enables the loadstore unit module to function.

core_ID: This genvar variable that is used to generate the hardware modules for core and register files.

Control: It has information about the read/write signals of the register file.

CHAPTER 5

DESIGN FLOW

5.1. Hardware Design Flow:

The hardware or the VLSI design flow as depicted in figure 5.1 gives the major steps taking the design towards physical realization.

The design of any product starts with an idea. The idea is born out of a client requirement. This idea is put down as a higher level behavioural model of the final product using the high level languages like Verilog. The behavioural model is then compiled into a RTL using a suitable compiler. RTL are generally the description of the circuit at the module level where input output interfaces, clock and other signals are visible. Any design can be described in RTL using the Huffman's model.

Once the RTL is arrived at, the next step in the design flow is the logic synthesis. Using commercial EDA tools, the designer converts the RTL into a netlist which is nothing but a list of gates and wires whose input output are specified. The EDA tools gives a lot of options like types of gates to be used, constraints for the design with respect to the power, area and timing, thus a highly optimized netlist is achieved after logic synthesis.

On getting the netlist, more EDA tools are used to do place and route of gates and wires or floor planning as it is popularly called. The result of place and route is the mask that could be handed over to the foundry for carrying out the fabrication of the chip. Two most important part of the design flow are the testing and verification. Testing is done to ensure final chip does not suffer from manufacturing defects and verification is done at each stage of

the flow to ensure the design meets the requirements as were originally projected.

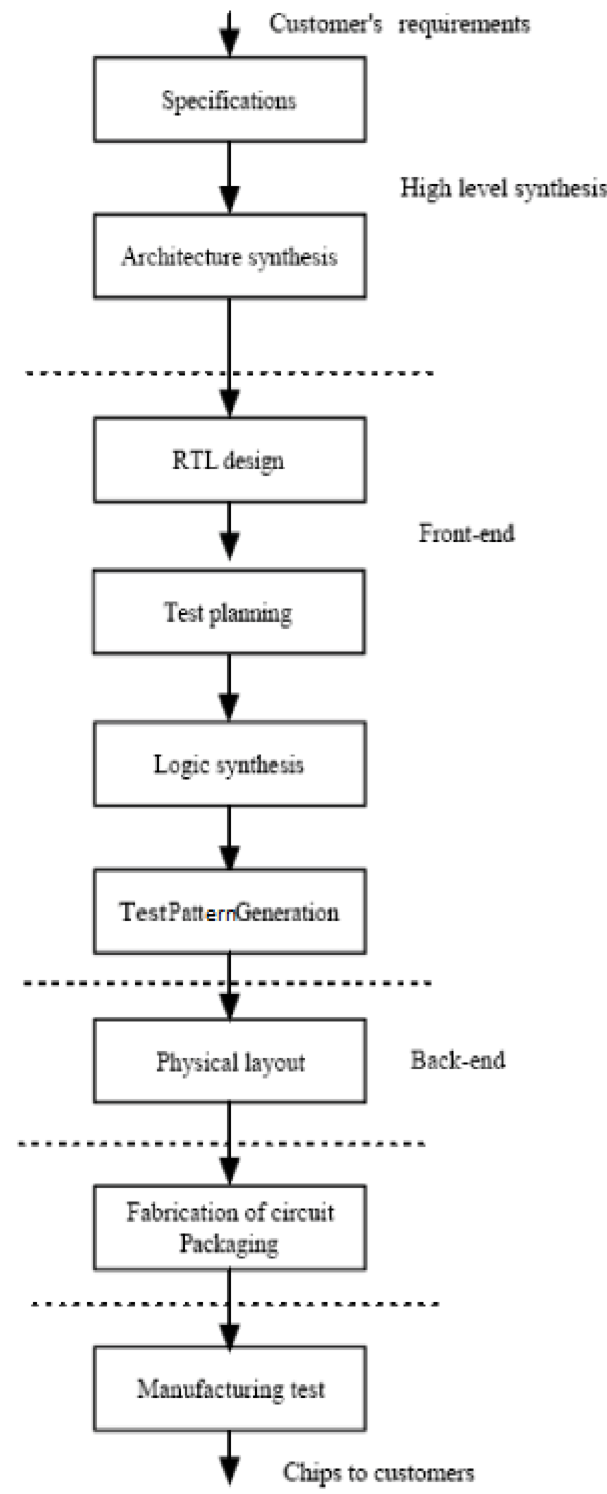

Fig 5.1. Hardware Design Flow

In our case however, we limit the scope to design, implementation of the design in verilog, logic synthesis and post-synthesis simulations to verify performance.

5.2. Implementation, Simulation :

Since the design process has been dealt with in earlier chapters adequately. We look at implementation in the verilog. Like previously mentioned we have adopted the top down approach for coding the design.

On completion of the verilog coding, the project is simulated with in-built modelsim simulator. Once the simulation is done and all the errors are rectified, further the design is needs to complete the logic synthesis as discussed in the design flow diagram. We not only receive an optimized netlist after logic synthesis but also reports for power, area and timing which are required for analysis.

The synthesis tool accepts the verilog files of the design and runs the synthesis algorithm for logic minimization. The synthesis culminates with generation of synthesized design schematic and detailed synthesis report with hardware units used in the final design.

The generated netlist is further used for post-synthesis simulations for arriving at power utilization by the design.

The 32-bit graphics processor architecture is simulated. Attached are the screen shots of simulation results.

32 bit registers addition done by 32 core units at the same time and result is printed. The addition registers are r1, r2 with values as 1, 2 in all of them. The addition result is stored in register r3 whose value will be 3 due to (1+2=3 i.e., r3 = r1+r2) addition.

ISim (P.49d)

File Edit View Simulation Window Layout Help

1.00us Re-launch

Console

```
0Addition result in core number 0 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 1 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 2 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 3 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 4 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 5 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 6 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 7 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 8 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 9 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 10 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 11 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 12 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 13 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 14 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 15 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 16 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 17 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 18 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 19 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 20 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 21 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 22 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 23 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 24 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 25 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 26 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 27 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 28 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 29 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 30 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Addition result in core number 31 is 3 with r1 as 1 and r2 as 2. This is new core with 4 bits in case statement
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 0
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 1
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 2
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 3
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 4
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 5
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 6
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 7
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 8
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 9
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 10
```

Open an existing file | Sim Time: 101,000 ps | Ln 37 Col 1 | Verilog

Fig 5.2: Simulation result screenshot 1

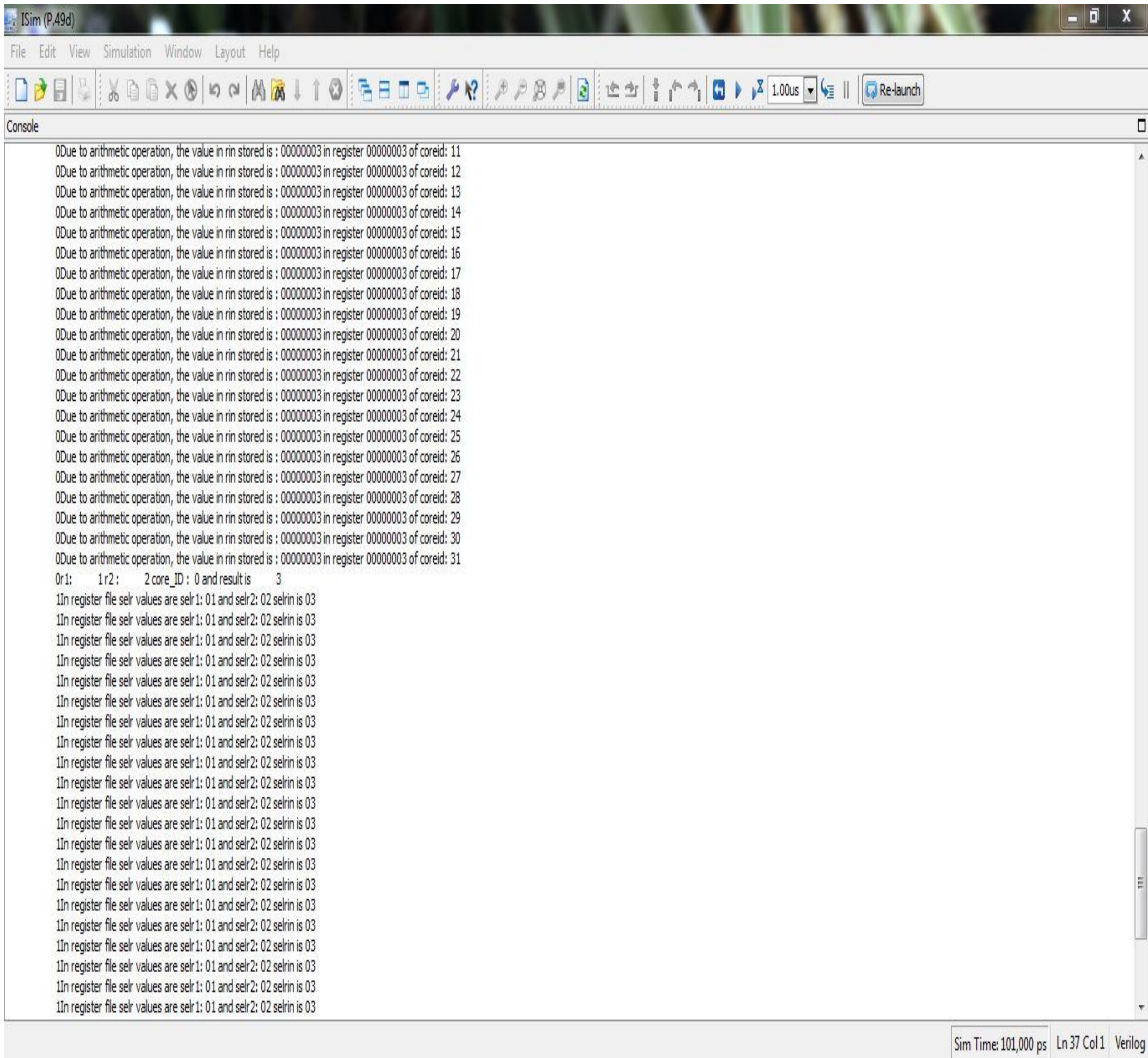

Fig 5.3: Simulation result screenshot 2

ISim (P.49d)

File Edit View Simulation Window Layout Help

1.00us Re-launch

Console

```
0Due to arithmetic operation, the value in rin stored is : 00000003 in register 00000003 of coreid: 31
0r1:      1 r2 :      2 core_ID :  0 and result is       3
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
1In register file selr values are selr1: 01 and selr2: 02 selrin is 03
Address of the 1st instruction is:    0 ; number of elements are:    64 ; number of blocks of data are: 1 ; number of warps are:  2
1 instrution_mem value in top module is 0c414 and MemtoReg value is 0 ////////// results[core_ID] for 0, 1 are       3 and       3 and rin[core_ID] for 0, 1 are       3 and       3
1instruction memory value is 0c414
1selr values are selr1: 01 and selr2: 02 and function is  4 (it should be 4)and ALU control is  4
2Im in data memory with address value as   65
2memtoreg value is 0
dM_address value in load unit is   66 and load address is   65
Stopped at time : 101 ns : File "C:/Users/PHANI/Documents/Xilinx projects/Main project/Main project.srcs/sim_1/new/top_module_tb.v" Line 37
ISim>
```

Sim Time: 101,000 ps | Ln 37 Col 1 | Verilog

Fig 5.4: Simulation result screenshot 3

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1. Conclusion:

The 32-bit graphics processor architecture is simulated for a set of streaming multiprocessors and other components. The parallelism can be further increased by adding more number of streaming multiprocessors, or by adding cores to the existing SMs.

It was verified by passing a series of load, add(arithmetic) and store instructions. The register files and corresponding memory locations are checked for correctness of actual results.

6.2. Future Work:

The graphics processor architecture is successfully implemented in verilog. More arithmetic operations or more other operations can also be included in the code while changing the case statement in the code and giving it a new opcode other than ones already in use.

Register arrays have been used in place of actual caches. So, the implementation of register arrays with actual caches can also be done.

Code can be further optimized to reduce power, area, timing by using other methods of design, than the ones used, which may optimize the above said parameters. Also the

interface of GPU with a CPU can be done and check for the GPGPU operation.

Bibliography:

[1] https://en.wikipedia.org/wiki/Graphics_processing_unit

[2] https://en.wikipedia.org/wiki/General_purpose_computing_on_graphics_processing_units

[3] https://en.wikipedia.org/wiki/Nvidia

[4]http://www.nvidia.in/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[5] http://www.asic-world.com/