Design and Development of Stream Processor Architecture for GPU Application Using Reconfigurable Computing

Sanket Dessai, Krishna Bhushan Vutukuru
Department of Computer Engineering, M.S.Ramaiah School of Advanced Studies, Bengaluru, India

Article Info

Article history:

Received Aug 21, 2012
Revised Dec 16, 2012
Accepted Jan 3, 2013

Keyword:

Stream Processor
GPU
FPGA
Cluster Array
Host Processor

ABSTRACT

Graphical Processing Units (GPUs) have become an integral part of today's mainstream computing systems. They are also being used as reprogrammable General Purpose GPUs (GP-GPUs) to perform complex scientific computations. Reconfigurability is an attractive approach to embedded systems allowing hardware level modification. Hence, there is a high demand for GPU designs based on reconfigurable hardware. Stream processor consists of clusters of functional units which provide a bandwidth hierarchy, supporting hundreds of arithmetic units. The arithmetic cluster units are designed to exploit instruction level parallelism and subword parallelism within a cluster and data parallelism across the clusters.For decreasing the area and power, a single controller is used to control data flow between clusters and between host processor and GPU. The designed of stream processor unit has been carried out in Verilog on Altera Quartus II and simulated using ModelSim tools. The functionality of the modelled blocks is verified using test inputs in the simulator.The simulated execution time of 8-bit pipelined multiplier is 60 ps and 100 ns for 8-bit pipelined adder while operating at 90 MHz.

Corresponding Author:

Sanket Dessai,
Department of Computer Engineering,
M.S.Ramaiah School of Advanced Studies,
#470-P,Peenya Industrial Area,Peenya 4th Phase,
Bengaluru, Karnataka 560058, India.
Email: sanketdessai@gmail.com

1. INTRODUCTION

As 3D graphics are becoming more vital in our lives, there is a need of fast graphics processors. With the introduction of the GPU, computationally intensive transform and lighting calculations were offloaded from the CPU onto the GPU—allowing for faster graphics processing speeds. This means all scenes increase in detail and complexity without sacrificing performance. For fast-paced games and other signal processing tasks, the computer has to go through this process about tens to hundreds of billions of times per second. Without a graphics card to perform the necessary computations, the workload would be too much for the computer to handle. To achieve these computation rates, current media processors use special purpose architectures customized to one specific application. Field Programmable Gate Arrays (FPGAs) are basically pieces of programmable logic. FPGAs have become more affordable they have found their way into more and more designs.Mapping a components of graphic processor on to FPGA is a challenging task and leds to new arena of research in reconfigurable computing.

Stream processors are signal and image processors, which offer both efficiency and programmability. Stream processors have efficiency comparable to ASICs, while being programmable in a high-level language. Streaming processors have been widely developed for many applications. The streaming processor can also be implemented as a reconfigurable streaming processor that can be reconfigured for several scientific computing applications (Rajagopal, et al. 2004).

Stream processors in GPUs can process vertices, pixels, geometry or physics—they are effectively general purpose floating-point processors. Most signal-processing applications are naturally expressed in this style. Stream processors are highly efficient computing engines that perform calculations on an input stream, while producing an output stream that can be used by other stream processors. Stream processors can be grouped in close proximity, and in large numbers, to provide immense parallel processing power.

Generally, specialized high-speed instruction decodes and execution logic is built into a stream processor, and similar operations are performed on the different elements of a data stream. On-chip memory is normally used to store output of a stream processor, and the memory can be quickly read as input by other stream processors for subsequent processing. SIMD (single instruction/multiple data) instructions can be implemented across groupings of stream processors in an efficient manner, and massively parallel stream processor clusters are well-suited for processing graphics data streams.

2. STREAM PROCESSOR ARCHITECTURE

Streaming processors have been widely developed for many applications. Some of them are implemented for example as an application-specific streaming processor. A few application examples can be found such as streaming processors for fluid dynamic computations based on lattice Boltzmann method and for accelerating ray tracing traversal algorithm in a graphic rendering application. The streaming processor can also be implemented as a reconfigurable streaming processor that can be reconfigured for several scientific computing applications.

A few floating-point-based processors that also support streaming computations have been developed so far. Intel Corporation has designed a 6.2-GFlops Floating-Point Multiply-Accumulator (FPMAC). The FPMAC processors are dedicated for a Teraflops Multicore System, which could run Tera-floating-point operations per second by interconnecting 80 FPMAC cores through a 2D mesh 8×10 on-chip network. Another floating-point-based processor that supports streaming computations is SPE (Synergistic Processing Element) processor from IBM Corporation.

Graphic Processing Units generally used for graphics data processing in multimedia applications can be classified as a streaming processor. The clock frequency of GPUs is frequently slower than premium CPUs. However, since GPUs are commonly interconnected each other to run graphic processing algorithms in parallel, GPUs can even provide better performance. The GPU such as NVIDIA Tesla architecture consists of 8 Texture/Processor Clusters (TPCs), where each TPC unified several streaming processors. The NVIDIA Tesla is a scalable array processor that can be programmed in C or via graphics APIs (Application-Programming Interfaces) to run parallel multithreaded computational models (Glaskowsky 2009).

2.1. BANDWIDTH HIERARCHY

In this model, the three-tiered storage bandwidth hierarchy enables the architecture to provide the instruction and data bandwidth necessary to efficiently operate ALUs in parallel. The hierarchy consists of a streaming memory system, a 128KB stream register file, and direct forwarding of results among arithmetic units via local register files. This hierarchy can be used to exploit the parallelism and locality of streaming media applications (NVIDIA 2008).

2.2. STREAM REGISTER FILE

The SRF is a 128KB memory organized to handle streams. The SRF can hold any number of streams of any length. The only limitation is the actual size of the SRF. Streams are referenced using a stream descriptor, which includes a base address in the SRF, a stream length, and the record size of data elements in the stream.An array of 22 64-word stream buffers is used to allow read or write access to 22 stream clients simultaneously (NVIDIA 2006). The clients are the units which access streams out of the SRF, such as the memory system, network interface, and arithmetic clusters. The internal memory array is 32 words wide, allowing it to fill or drain half of one stream buffer every two cycles, providing a total bandwidth of 25.6GB/s for all 22 streams.

Each stream client may access its dedicated stream buffer every cycle if there is data available to be read or space available to be written. The eight stream buffers serving the clusters are accessed eight words at a time, one word per cluster. The eight stream buffers serving the network interface are accessed two words at a time (NVIDIA 2006). The other six stream buffers are accessed a single word at a time. The peak bandwidth of the stream buffers is therefore 86 words per cycle, allowing peak stream demand to exceed the SRF bandwidth during short transients. Stream buffers are bidirectional, but may only be used in a single direction for the duration of each logical stream transfer.

2.3. MEMORY SYSTEM

As described above, all memory references are made using stream load and store instructions that transfer an entire stream between memory and the SRF. This stream load/store architecture is similar in concept to the scalar load/store architecture of contemporary RISC processors.

2.4. CLUSTER ARRAY

Each input of every functional unit in the cluster is fed by a separate local register file (LRF). These local register files store kernel constants, parameters, and local variables, reducing the required SRF bandwidth. Each cluster has 17 LRFs (4 32-entry and 13 16-entry LRFs) for a total of 272 words per cluster and 2176 words across the eight clusters (NVIDIA 2006). Each local register file has one read port and one write port. The 15 local register files collectively provide 54.4 GB/s of peak data bandwidth per cluster, for a total bandwidth of 435 GB/s within the cluster array.

Additional storage is provided by a 256-word scratch-pad register file. It can be indexed with a base address specified in the instruction word and an offset specified in a local register. The scratch-pad allows for coefficient storage, short arrays, small lookup tables, and some local register spilling.The intercluster communication unit allows data to be transferred between clusters over the intercluster network using arbitrary communication patterns.

The adders and multipliers are fully pipelined and perform single precision floating point arithmetic, 32-bit integer arithmetic and 16-bit or 8-bit parallel subword integer operations. The divide/square root unit is not pipelined and operates only on single precision floating point and 32-bit integers (NVIDIA 2006). The divider can support two simultaneous operations, with latencies ranging from 16-23 cycles depending on the operation and data type. The 48 total arithmetic units, six units replicated across eight clusters, provide a peak computation rate of over 16GOPS for both single precision floating point and 32-bit integer arithmetic (NVIDIA 2006).

2.5. NETWORK INTERFACE

The network interface connects the SRF to four bidirectional links (400MB/s per link in each direction) that can be configured in an arbitrary topology to interconnect processors. A *send* instruction executed on the source processor reads a stream from the SRF and directs it onto one of the links and through the network as specified by a routing header. At the destination processor, a *receive* instruction directs the arriving stream into the SRF. The send and receive instructions both specify a tag to allow a single node to discriminate between multiple arriving messages.

Using the stream model, it is easy to partition an application over multiple processors using the network. To partition an application across two processors, the application is adapted by dividing the stream-level code across the two processors, inserting a *send* instruction at one end, and inserting a *receive* instruction at the other.

2.6. STREAM CONTROLLER

Host processor issues stream-level instructions to stream processor with encoded dependency information. The stream controller buffers these instructions in a scoreboard and issues them when their resource requirements and dependency constraints are satisfied.

2.7. HOST INTERFACE

The host interface allows a stream processor to be mapped into the host processor's address space, so the host processor can read and write stream memory. The host processor also executes programs that issue the appropriate stream-level instructions to the stream processor. These instructions are written to special memory mapped locations in the host interface.

3. REQUIREMENT ANALYSIS

The requirement analysis for the stream processor architecture modelling includes on FPGA devices and its peripherals. Since a stream processor and texture filtering unit needs to be verified, the output can be monitored on a terminal for the multiprocessing tasks. In order to debug the processor and monitor the processor state a ‘JTAG Debugger’ must be present in the system, which should be controlled by a processor debugging module and be able to switch between multiple processors.

3.1. DESIGN OF STREAM PROCESSOR

The design is based on Harvard architecture, which defines physically separated memories for program instructions and data. This implies that the widths of data busses may differ per memory type. This is especially useful for VLIW architectures, because to issue very long instruction words from instruction memory.

The stream controller is designed with following four stages fetch, decode, execute and write back. General-purpose Register (GR) file with 64 32-bit registers and a Branch Register (BR) file with eight 1-bit registers are used within the each arithmetic cluster.

The fetch unit of the stream controller fetches a VLIW instruction from the attached host processor, and passes it on the internal decode unit. In this stage, the instruction is being split. Also, the register contents used as operands are fetched from the register files. The actual operations take place in the execute unit of stream controller, parallel microcontroller and SRF unit. Arithmetic, multiplication, and division/square root operations in arithmetic clusters are performed in the execute stage. This stage is designed parametric, so that the number of arithmetic clusters could be adapted. Stream processor should have exactly one microcontroller and SRF unit, so these units are designed outside the parametric stream controller. All jump and branch operations are handled by the microcontroller, and all data memory load/store operations are handled by the SRF unit. To ensure that all results to the GR, BR, SRF and the internal Program Counter (PC) are written at the same time per instruction, all write activities are performed in the write back unit of stream controller. The functionality of the blocks is explained below:

3.2. STREAM REGISTER FILE (SRF)

SRF effectively isolates the arithmetic clusters from the memory system, making streaming processor load/store architecture for streams. Two arithmetic clusters and a stream controller are connected to the SRF. The arithmetic clusters consume data streams from the SRF, process them, and return their output data streams to the SRF. The two clusters operate simultaneously on interleaved data records transferred from the SRF, allowing two elements to be processed in parallel. Each arithmetic cluster contains three adders, two multipliers, and one divide/square root unit. These units are controlled by statically scheduled VLIW instructions issued by the controller.

The flow of data in the SRF is explained in the flow chart shown in Figure 1. The data flows from SRF to C (Mux_3) only when there is a positive edge clock pulse and if the write signal is high which is issued by the B (stream controller). The address register itself stores the address of the next instruction; the present instruction address is stored in SRF that is sent by A (Address_Register).

3.3. STREAM CONTROLLER

Stream Controller unit is implemented in the streaming processor to control the processor configuration and the flow of streaming computations. The processor configuration and the flow of the streaming computations are controlled by fetching and executing instruction vectors stored in the instruction memory of host processor implemented on the stream controller. Each instruction vector contains micro codes to enable the arithmetic and memory units and to control the streaming computation flows. The controller has three phases of operation: fetch, decode, and execute. Fetching retrieves an instruction from memory, decoding decodes the instruction, manipulates data paths, and loads registers; execution generates the results of the instruction. The fetch phase will require two clock cycles – one to load the address register and one to retrieve the addressed word from memory. The decode phase is accomplished in one cycle. The execution phase may require zero, one, or two more cycles, depending on the instruction.

First, the functional units are declared according to the partition of the controller as shown in Figure 2. Then their ports and variables are declared and checked for syntax. Then the individual units are described, debugged, and verified. The last step is to integrate the design and verify that it has correct functionality.
The instruction mnemonics and their actions mentioned in the Figure 2, 3, 4, and 5 are listed below:

NOP	No operation is performed;
ADD	Adds the contents of the source and destination registers parallel and stores the result into the destination register
MUL	Multiplies the contents of the source and destination registers parallel and stores the result into the destination register
RD	Fetches a memory word from the location specified by the second byte and loads the result into the destination register. The source register bits are don't cares

WR	Writes the contents of the source register to the word in memory specified by the address held in the second byte. The destination register bits are don't-cares
BR	Branches the activity flow by loading the program counter with the word at the location (address) specified by the second byte of the instruction. The source and destination bits are don't-cares
BRZ	Branches the activity flow by loading the program counter with the word at the location (address) specified by the second byte of the instruction if the zero flag register is asserted.

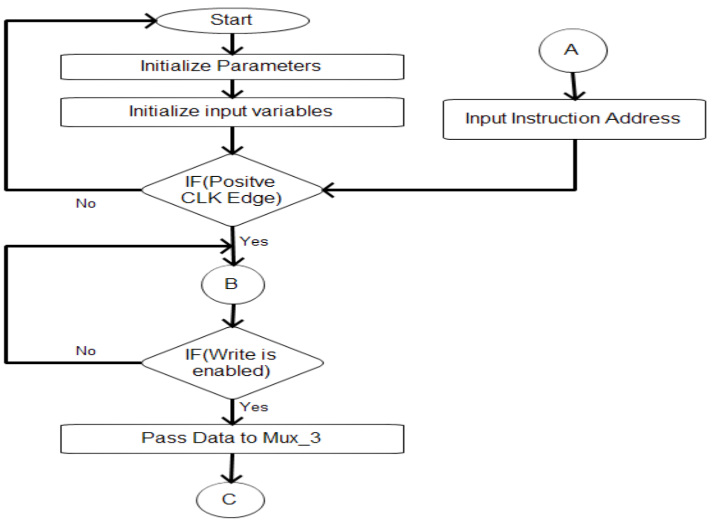


Figure 1. Flow Chart of the SRF

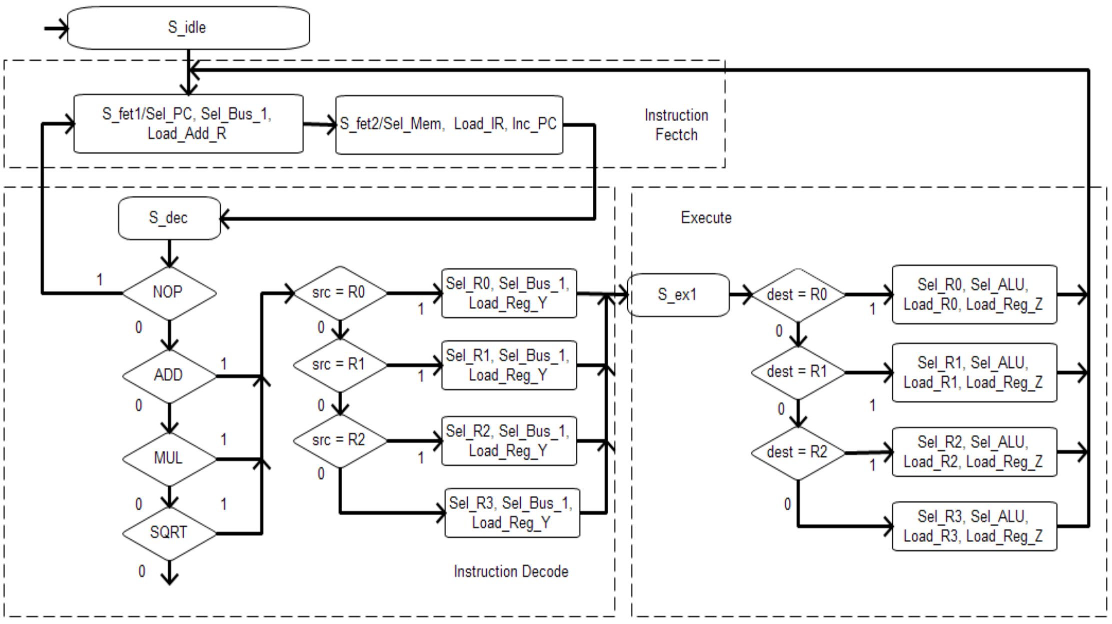

S_idle
S_fet1/Sel_PC, Sel_Bus_1, Load_Add_R
S_fet2/Sel_Mem, Load_IR, Inc_PC
Instruction Fectch
S_dec
Execute
NOP
ADD
MUL
SQRT
src = R0
src = R1
src = R2
Sel_R0, Sel_Bus_1, Load_Reg_Y
Sel_R1, Sel_Bus_1, Load_Reg_Y
Sel_R2, Sel_Bus_1, Load_Reg_Y
Sel_R3, Sel_Bus_1, Load_Reg_Y
S_ex1
dest = R0
dest = R1
dest = R2
Sel_R0, Sel_ALU, Load_R0, Load_Reg_Z
Sel_R1, Sel_ALU, Load_R1, Load_Reg_Z
Sel_R2, Sel_ALU, Load_R2, Load_Reg_Z
Sel_R3, Sel_ALU, Load_R3, Load_Reg_Z
Instruction Decode
1
0

Figure 2. Flow Chart for the Controller of a Processor that Implements the Instruction Set NOP, ADD, MUL, SQRT

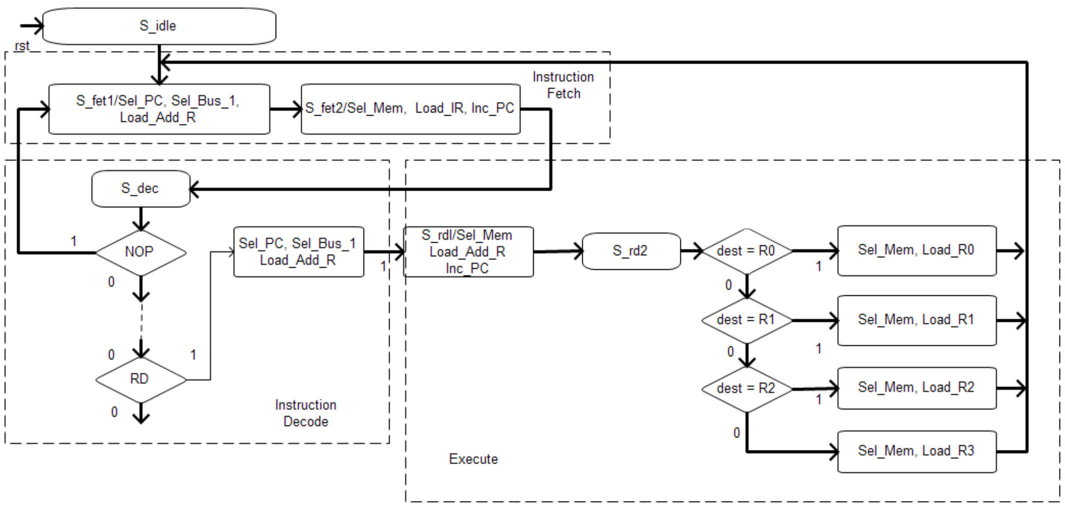


Figure 3. Flow Chart for the Controller of a Processor that Implements the Instruction Set RD

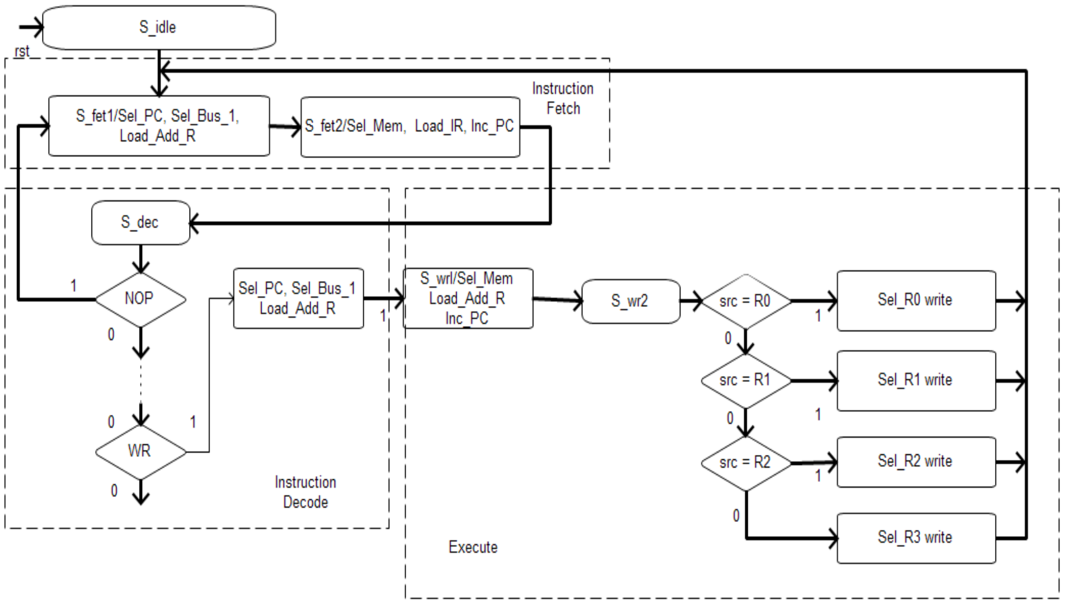


Figure 4. Flow Chart for the Controller of a Processor that Implements the Instruction Set WR

3.4. ARITHMETIC CLUSTER

Two arithmetic clusters are controlled by the controller in a SIMD fashion. The arithmetic clusters operate simultaneously on interleaved data records transferred from the SRF, allowing eight data records to be processed in parallel. Each arithmetic cluster contains two adders, two multipliers, and one divide/square root unit. These units are controlled by statically scheduled VLIW instructions issued by the controller. All of the arithmetic units support both 16-bit single precision floating point and 16-bit integer operations. In addition, the adders and multipliers support 16-bit and 8-bit parallel-subword operations for a subset of the integer operations. The adders and multipliers are fully pipelined, allowing a new operation to issue every cycle. The divide/square root unit has two SRT cores, so no more than two divide or square root operations can be in flight at any given time.

Figure 6 explains pipelined adder. At positive edge clock pulse if the select/reset signal is high the LSB’s registers will get cleared or else if any data is available then the LSB’s of the available two data’s will get added and the result is stored in LSB’s register.

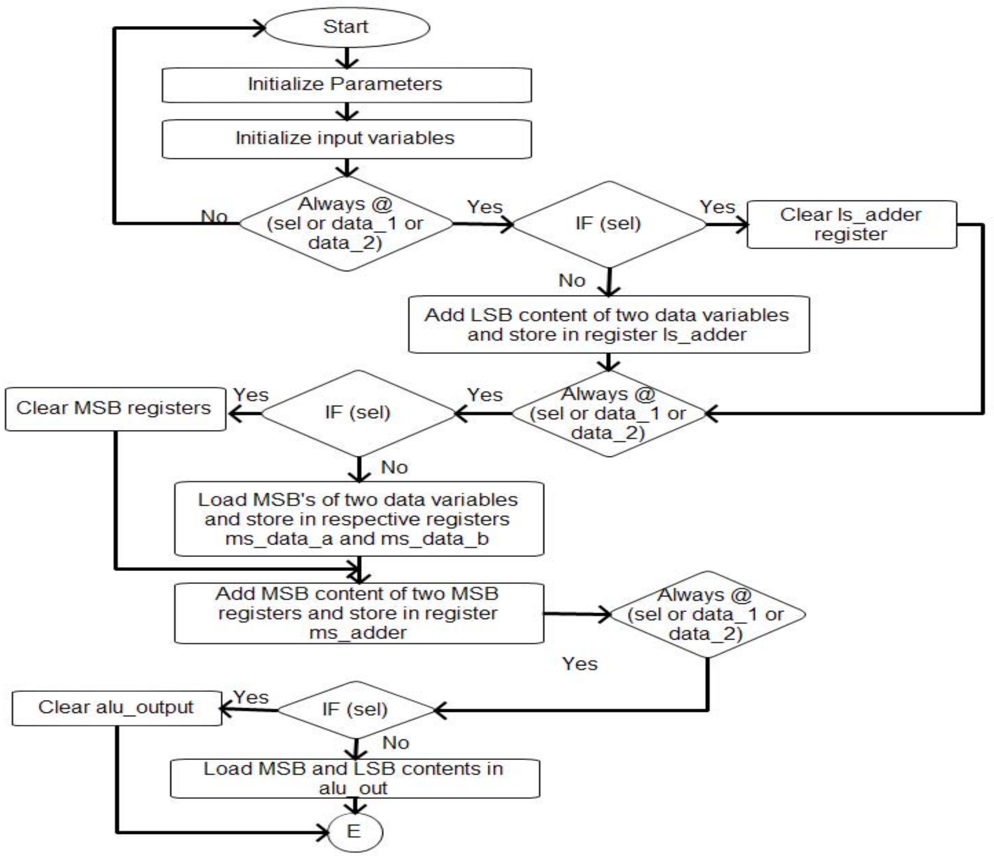

Start
Initialize Parameters
Initialize input variables
Always @ (sel or data_1 or data_2)
No
Yes
IF (sel)
Yes
Clear ls_adder register
No
Add LSB content of two data variables and store in register ls_adder
Always @ (sel or data_1 or data_2)
Yes
IF (sel)
Yes
Clear MSB registers
No
Load MSB's of two data variables and store in respective registers ms_data_a and ms_data_b
Add MSB content of two MSB registers and store in register ms_adder
Always @ (sel or data_1 or data_2)
Yes
IF (sel)
Yes
Clear alu_output
No
Load MSB and LSB contents in alu_out
E

Figure 5. Flow Chart of Pipelined Adder

Similarly for MSB's, first the MSB data will be stored in respective registers and the contents of the MSB registers will get added and stored in another register. Finally, the LSB and MSB get concatenate and result is sent to E (SRF).

3.5. HOST PROCESSOR

An external host processor executes all scalar code and transfers stream instructions to stream processor, which acts as a coprocessor. The host processor is able to execute small serial sections of code that exist in any real-world application. Stream processor is tailored to take advantage of the characteristics of media processing applications. Code that is not actually operating on media data, but is rather coordinating the control flow of the overall application, is much better suited to a conventional serial processor than a SIMD stream processor. Therefore, each processor is able to execute the code for which it was designed — the host processor executes small sections of control intensive serial code and stream processor executes large data parallel stream programs.

4. VERIFICATION, TESTING AND VALIDATION

The developed test cases are validated for the basic functionalities. Testing involves operation of a system or application under controlled conditions and evaluating the results. Test cases have been performed on individual sections of the stream processor and texture filtering unit.

4.1. TEST VECTOR FOR STREAM CONTROLLER

The main function of the stream controller is to issue control signals to the executable unit and to the stream register file. When an instruction of length 8-bit is issued as shown:

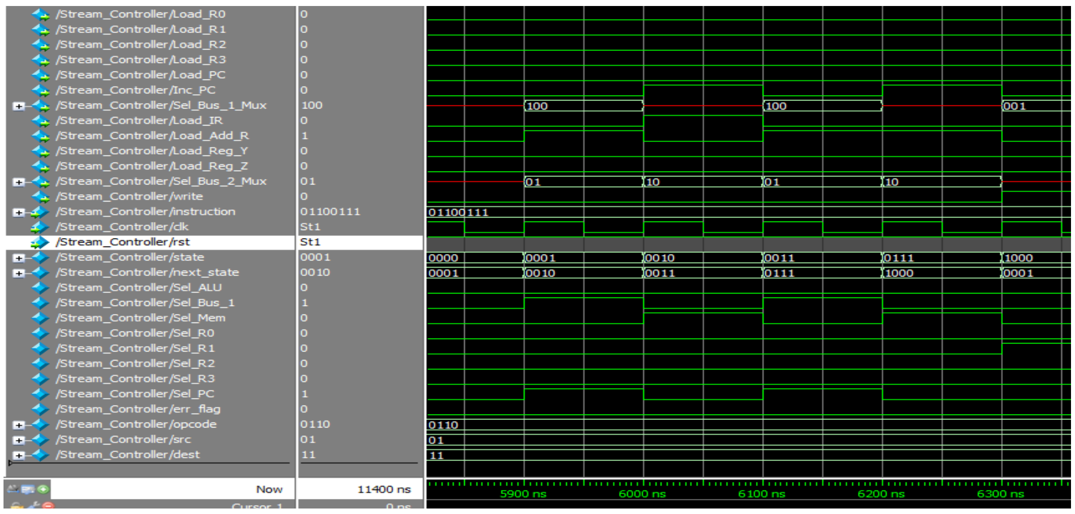

/Stream_Controller/Load_R0 0
/Stream_Controller/Load_R1 0
/Stream_Controller/Load_R2 0
/Stream_Controller/Load_R3 0
/Stream_Controller/Load_PC 0
/Stream_Controller/Inc_PC 0
/Stream_Controller/Sel_Bus_1_Mux 100
/Stream_Controller/Load_IR 0
/Stream_Controller/Load_Add_R 1
/Stream_Controller/Load_Reg_Y 0
/Stream_Controller/Load_Reg_Z 0
/Stream_Controller/Sel_Bus_2_Mux 01
/Stream_Controller/write 0
/Stream_Controller/instruction 01100111
/Stream_Controller/clk St1
/Stream_Controller/rst St1
/Stream_Controller/state 0001
/Stream_Controller/next_state 0010
/Stream_Controller/Sel_ALU 0
/Stream_Controller/Sel_Bus_1 1
/Stream_Controller/Sel_Mem 0
/Stream_Controller/Sel_R0 0
/Stream_Controller/Sel_R1 0
/Stream_Controller/Sel_R2 0
/Stream_Controller/Sel_R3 0
/Stream_Controller/Sel_PC 1
/Stream_Controller/err_flag 0
/Stream_Controller/opcode 0110
/Stream_Controller/src 01
/Stream_Controller/dest 11
100
100
001
01
10
01
10
01100111
0000
0001
0010
0011
0111
1000
0001
0010
0011
0111
1000
0001
0110
01
11
Now 11400 ns
Cursor 1
5900 ns
6000 ns
6100 ns
6200 ns
6300 ns

Figure 6. Test Case Results of Stream Controller

Figure 6, it is observed that the first two bits of the instruction are for destination address and third and fourth bits are for source address. The next four bits are opcode.The program counter holds the address of the next instruction to be executed. When the external reset is asserted, the program counter is loaded with 0, indicating that the bottom of memory holds the next instruction that will be fetched. Under the action of the clock, for single cycle instructions, the instruction at the address in the program counter is loaded into the instruction register and the program counter is incremented. An instruction decoder determines the resulting action on the data paths and the ALU.

4.2. TEST VECTOR FOR PIPELINE ADDER

WR The test case result of the pipelined adder is shown in Figure 7. The time taken to perform the compilation by using pipelined adder is 100 ns. In this model first the LSBs of the two numbers will be added and next the MSBs. The internal carry from the LSBs will be carried to the MSBs.

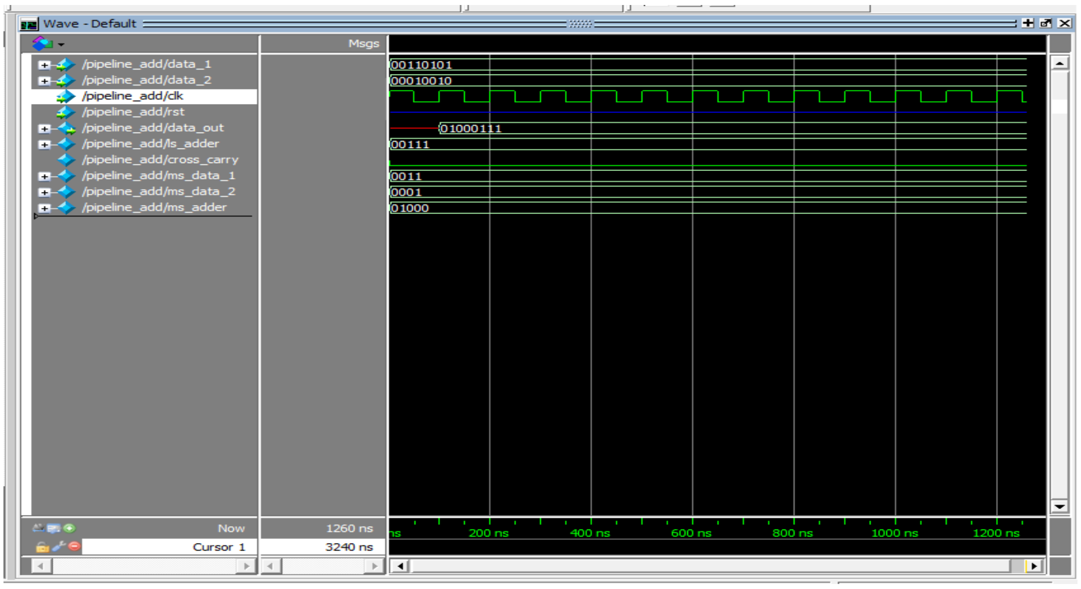


Figure 7. Test Case Results for Pipelined Adder

5. RESULTS AND DISCUSSION

Stream processor architecture IP is designed by using verilog programming language, Altera Quartus II, ModelSim, and Active HDL synthesis and simulation tools. The generated architecture of the stream processor is show in Figure 8.

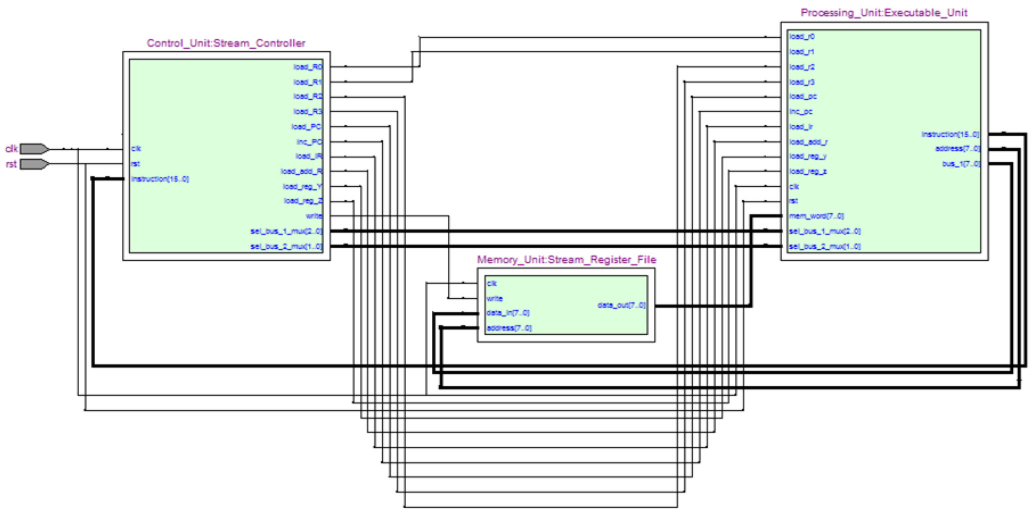


Figure 8. Generated Block Diagram of the Stream Processor

Stream processor architecture IP is designed by using verilog programming language, Altera Quartus II, ModelSim, and Active HDL synthesis and simulation tools. The generated architecture of the stream processor is show in Figure 8. It has been observed from the Figure 8 that the generated stream processor architecture has three main sections, stream controller, stream register file, and executable unit. Each generated section's internal data/instruction flow is explained as follows:

5.1. STREAM CONTROLLER UNIT

The timing of all activity in the processor is determined by the controller. The controller must steer data to the proper destination, according to the instruction being executed. Thus, the design of the controller is strongly dependent on the specification of the machine's arithmetic cluster and data path resources and the clocking scheme available. In program-directed operation, instructions are fetched synchronously from memory, decoded, and executed to:

a) operate on data within the arithmetic cluster unit
b) change the contents of storage registers
c) change the contents of the program counter (PC), instruction register (IR) and the address register (ADD_R)
d) change the contents of memory
e) retrieve data and instructions from memory
f) control the movement of data on the system busses

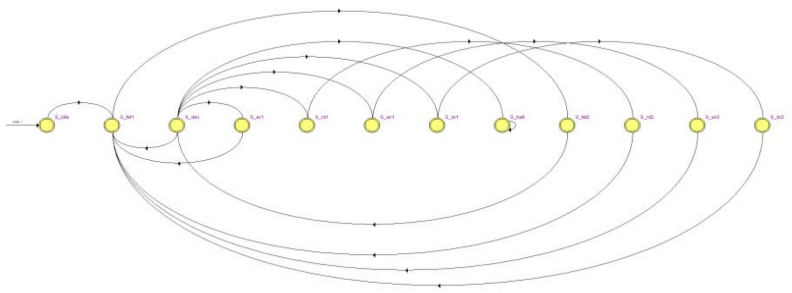


Figure 9. Generated State Machine Diagram of Controller

State machine diagram of the stream controller is shown in Figure 9, explains about how the instruction fetch, decode, and execute is done. In a serial processor three clock cycles are required to perform instruction fetch, decode, and execute, whereas in pipelined processor during first clock cycle instruction fetch of first instruction is done, during second clock cycle fist instructions decode and second instructions fetch is done and finally during third clock cycle first instructions execution, second instructions decode and third instructions fetch will done.

The signals produced by the stream controller, shown in Figure 10, are identified as follows:

Control Signal	Action
Load_Add_Reg	Loads the address register
Load_PC	Loads BUS_2 to the program counter
Load_IR	Loads BUS_2 to the instruction register
Inc_PC	Increments the program counter
Sel_Bus_1_Mux	Selects among the Program_Counter, R0, R1, R2, and R3 to drive Bus_1
Sel_Bus_2_Mux	Selects among Alu_out, Bus_1, and memory to drive Bus_2
Load_R0	Loads general-purpose register R0
Load_R1	Loads general-purpose register R1
Load_R2	Loads general-purpose register R2
Load_R3	Loads general-purpose register R3
Load_Reg_Y	Loads Bus_2 to the register Reg_Y
Load_Reg_Z	Stores output of ALU in register Reg_Z
Write	Loads Bus_1 into the SRAM memory at the location specified by the address

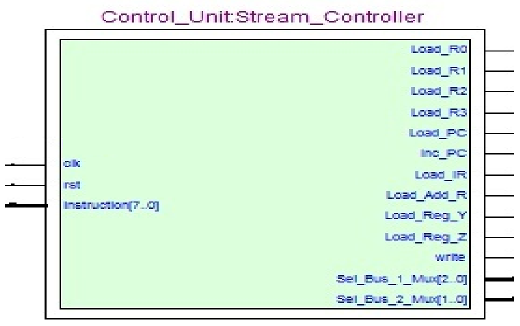


Figure 10. Generated Block Diagram of the Stream Controller

5.2. EXECUTABLE UNIT

The internal sections of the executable unit are shown in Figure 11. The executable unit consists of the two arithmetic cluster units, dedicated general purpose registers for arithmetic units, program counter, instruction register, address register, dedicated register for store output data.

5.3. ARITHMETIC CLUSTER UNIT

Stream processor consists of two pipelined arithmetic cluster units. Each arithmetic cluster unit is a combination of three pipelined adder, two pipelined multiplier units. The generated pipelined adder is shown in Figure 12, the input to the clusters will be from the dedicated registers of the arithmetic clusters, to which data will be from stream memory unit. After processing within arithmetic clusters the output data will be fed back to the dedicated registers of arithmetic clusters and from there to other stream processor.

The simulation result of the pipelined adder is shown in Figure 13. The simulation is of the model is done by using ModelSim simulation tool. The time taken to perform the compilation by using pipelined adder is 20 ns. The generated pipelined multiplier is shown in Figure 14. The input to the clusters will be from the dedicated registers of the arithmetic clusters, to which data will be from stream memory unit.

The simulation result of the pipelined multiplier is shown in Figure 15. The simulation is of the model is done by using ModelSim simulation tool. The time taken to perform the compilation by using pipelined multiplier is 60ps.

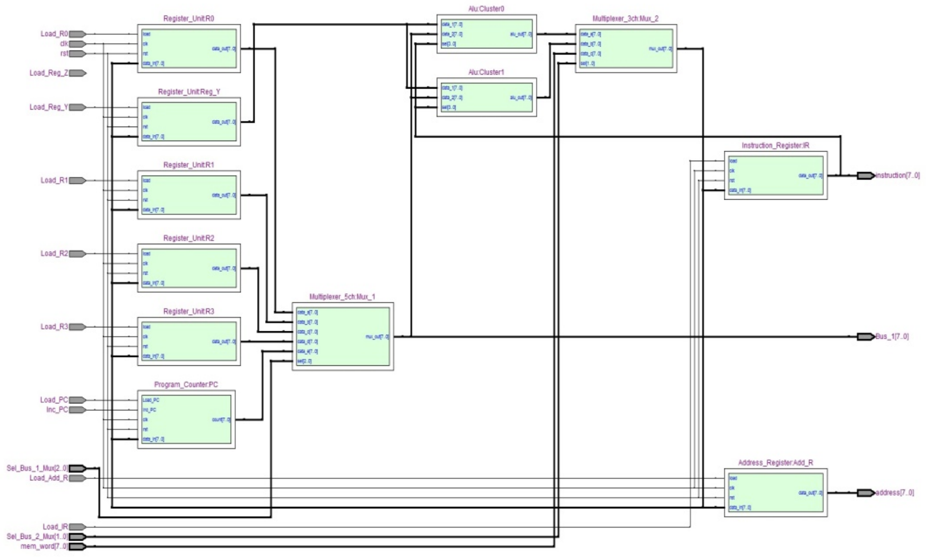


Figure 11. Generated Internal Blocks of the Executable Unit

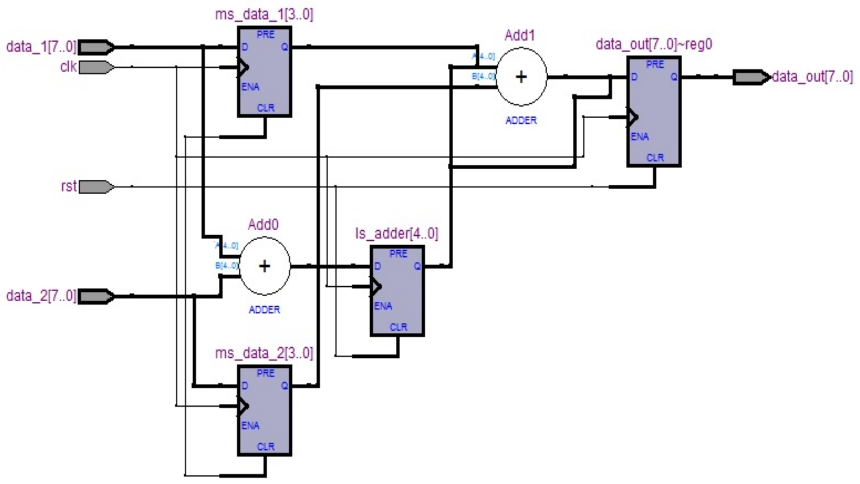


Figure 12. Generated Pipelined Adder Unit

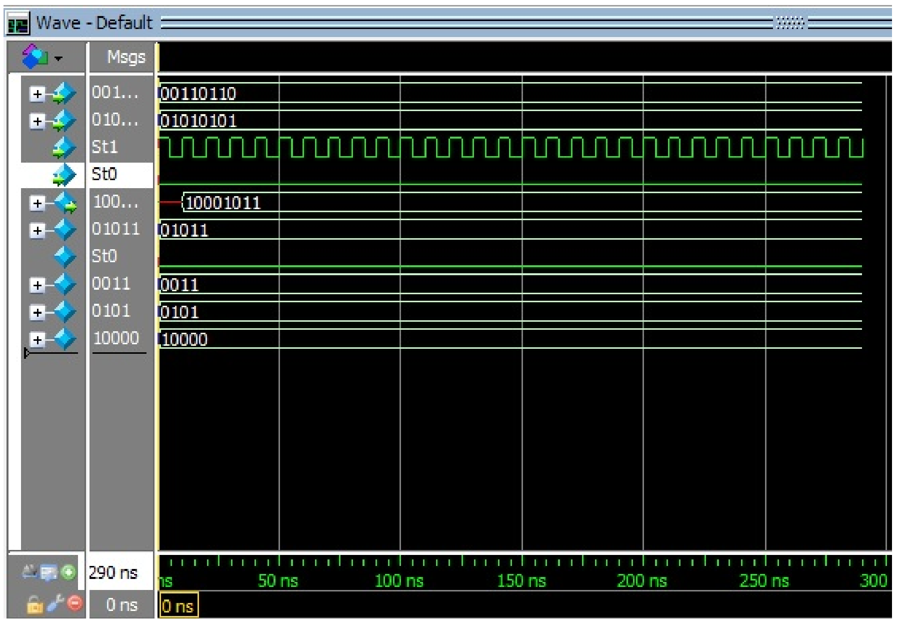


Figure 13. Simulation Results of Pipelined Adder

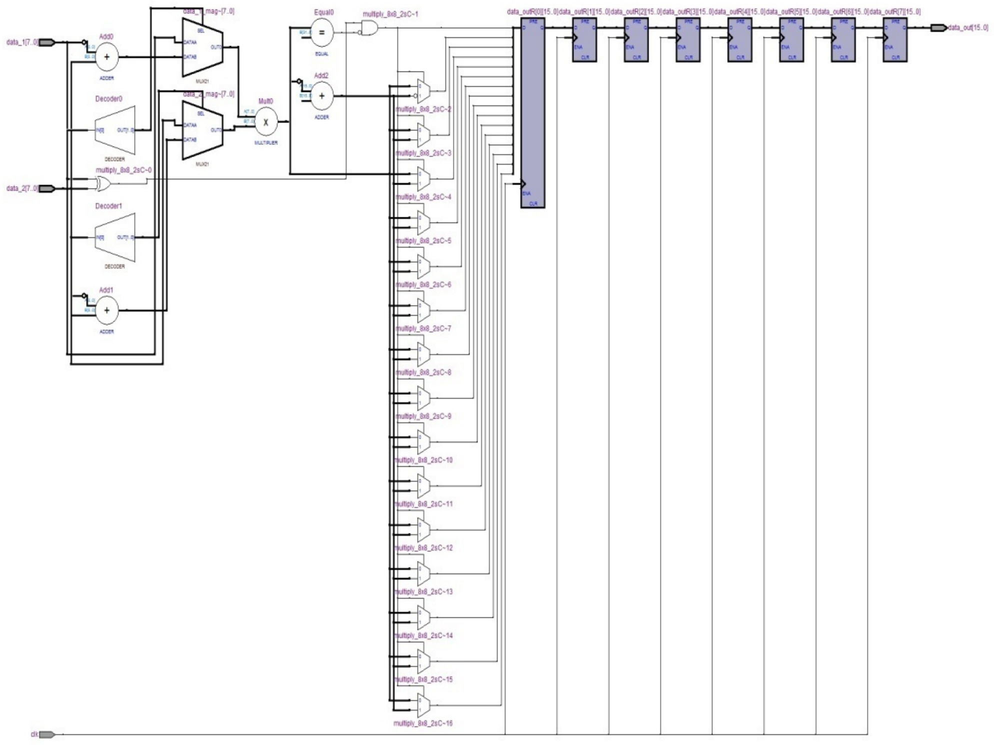
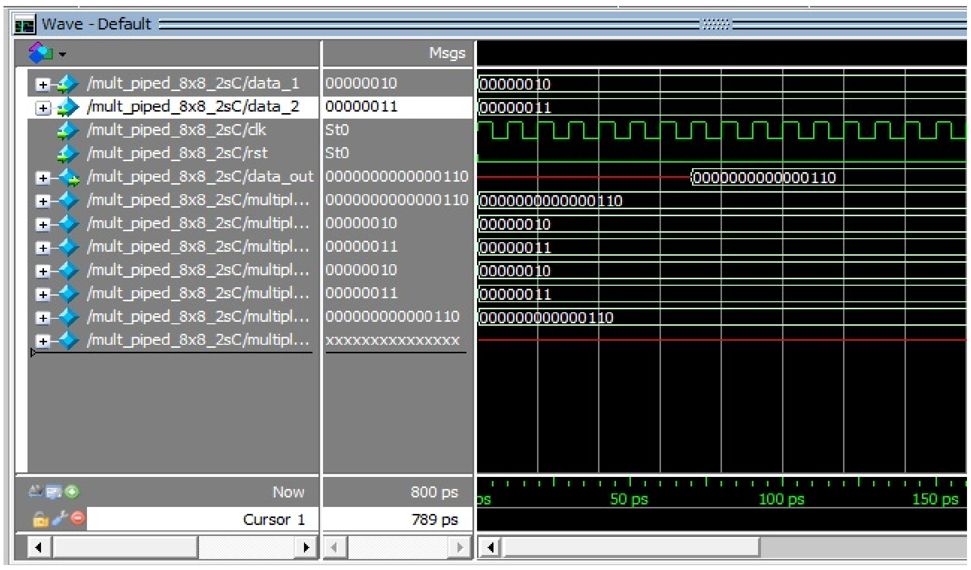


Figure 14. Generated Pipelined Multiplier



```
Wave - Default
                                   Msgs
/mult_piped_8x8_2sC/data_1         00000010           00000010
/mult_piped_8x8_2sC/data_2         00000011           00000011
/mult_piped_8x8_2sC/clk            St0
/mult_piped_8x8_2sC/rst            St0
/mult_piped_8x8_2sC/data_out       0000000000000110   0000000000000110
/mult_piped_8x8_2sC/multipl...     0000000000000110   0000000000000110
/mult_piped_8x8_2sC/multipl...     00000010           00000010
/mult_piped_8x8_2sC/multipl...     00000011           00000011
/mult_piped_8x8_2sC/multipl...     00000010           00000010
/mult_piped_8x8_2sC/multipl...     00000011           00000011
/mult_piped_8x8_2sC/multipl...     0000000000000110   0000000000000110
/mult_piped_8x8_2sC/multipl...     xxxxxxxxxxxxxxxx

Now          800 ps     ps   50 ps   100 ps   150 ps
Cursor 1     789 ps
```

Figure 15. Simulation Result of the Pipelined Multiplier

6. CONCLUSION

Based on design specifications, stream processor unit has been developed.The design and development has been carried out using Verilog with the help of Altera Quartus II tool. A single controller has been used over traditional two controllers for controlling the data flow between host processor and stream processor and also between arithmetic clusters.Hence reducing area and power requirements. Writing the contents of the source register to the word in memory specified by the address held in the second byte. The destination register bits are don't-cares. The functionality of the modelled blocks is verified using test inputs in the simulator.The simulated execution time of 8-bit pipelined multiplier is 60ps and 100ns for 8-bit pipelined adder while operating at 90MHz.

REFERENCES

[1] Altera Corporation. *Introduction to Simulation of Verilog Design Using ModelSim Graphical Waveform*. February 2012. ftp://ftp.altera.com/up/pub/Altera_Material/9.1/Tutorials/Verilog/ModelSim_GUI_Introduction.pdf

[2] Arvo J. Graphics Gems II. *USA: Academic Press Inc*.

[3] S Banerjia, E Ozer, TM Conte. *Unified Assign and Schedule:A New Approach to Scheduling for Clustered Registered File Microarchitectures*. Proceeding of IEEE International Symposium on Microarchitecture. 2009; 308-315.

[4] QK Chen, JK Zhang. *A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA*. Proceedings of IEEE International Conference on Information Science and Engineering. 2009; 86-89.

[5] J Chittarmuru, J Euh, W Burleson. *A Low-Power Content-Adaptive Texture Mapping Architecture for Real-Time 3D Graphics*. University of Massachusetts Amherst, http://www.citeseerx.ist.psu.edu. 2012.

[6] Doulos Inc. OVM Golden Reference Guide,Version 2.0. *Hamspire:Doulos*. 2008.

[7] Doulos Inc. The Verilog Golden Reference Guide, Version 1.0. *Hamspire: Doulos*. 1996.

[8] JD Foley, AV Dam, SK Feiner, JK Hughes. Computer Graphics: Principles and Practice in C, 2nd edition. *USA: Addison-Wesley*. 1990.

[9] KS Fu. Syntactic Pattern Recognition and Application. *New Jersey: Prentice-Hall*. 1982.

[10] PN Glaskowsky. NVIDIA’s Fermi: The First Complete GPU Comuting Architecture, <http://www.nvidia.com/content/pdf/fermi_white_papers/P.Glaskowsky_Nvidia’s_fermi_the_first_complete_GPU_architecture.pdf. 2011.

[11] A Greß, G Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. *Proceeding of IEEE Conference on Graphics Hardware*. 2006.

[12] M Gschwind, V Salapura. A VHDL Design Methodology for FPGA. *Proceeding of FPL`95 of the 5th International Workshop on Field-Programmable Logic and Application*. 1995; 208-217.

[13] A Gupta, ZS Hakura. The Design and Analysis of a Cache Architecture for Texture Mapping. http://www.cs.cmu.edu/afs/ cs/academics/class/15869-f11/www/readings/hakura97_texcaching.pdf. 2011.

[14] PS Heckbert. Survey of Texture Mapping. *IEEE Transcation of Computer Graphics and Applications* 6. 1986: 56-67.

[15] Intel’s Next Generation Integrated Graphics Architecture. *Intel Graphics Media Accelerator X3000 and 3000,2006* http://www.intel.com/products/chipsets/gma3000/gma3000.pdf

[16] E Kilgariff, R Fernando. The GeForce 6 Series GPU Architecture. http://www.nvidia.com/page/geforce6.html. 2011

[17] CH Kim, LS Kim. *Adaptive Selection of an Index in a Texture Cache.* In Proceedings of IEEE International Conference on Computer Design:VLSI in Computers and Processors. 2004; 295-300.
[18] Lattice Semiconductor Corporation. FPGA Design Guide. http://www.lattticesemi.com/lit/docs/manuals/fpga_design_guide.pdf. 2012.
[19] J Mc Donald. Insider Guide: FPGAs, Tools and Boards. http://www.eg3.com/report-fpga. 2011.
[20] NVIDIA. GeForce 8800 GPU Architecture Overview. *White paper TB-02787-001_v0.9*. http://www.nvidia.com/object/IO_37_100.html. 2011
[21] NVIDIA. GeForce GTX 200 GPU Architecture Overview ”,*White paper TB-04044-001_v01*, http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf. object/IO_37_100.html. 2011.
[22] S Rajagopal, JR Cavallaro, S Rixner. *Design Space Exploration for Real-Time Embedded Stream Processors*. Proceedings of IEEE Computer Society. 2004
[23] F Remond. The Work Flow of a Block-Based Design Team,Integrated System Design. http://www.eedesign.com/editorial/2000/designtools0012.html. 2011.
[24] J Sanders, E Kandrot, J Dongarra. CUDA by Example an Introduction to General-Purpose GPU Programming. *Boston: Addison-Wesley*. 2011.
[25] HC Shin, JA Lee, LS Kim. A Cost-Effective VLSI Architecture for Anisotropic Texture Filtering in Limited Memory Bandwidth. *IEEE Transactions on VLSI Systems*. 2006; 14: 254-257.
[26] Stream Processors Inc. Stream Processing: Enabling the New Generation of Easy to Use, High-performance DSPs. *White Paper, USA*. 2008
[27] F Tomita, S Tsuji. Computer Analysis of Visual Textures. *Boston: Kluwer Academic Publishers*. 1990.
[28] H Voorhees, T Poggio. *Detecting Textons and texture Boundaries in natural Images.* Proceedings of the First International Conference on Computer Vision, London. 1987; 250-258.
[29] WT Wang, YC Chen, CP Chung. *A Run-Time Reconfigurable Fabric for 3D texture Filtering.* Proceedings of IEEE International Conference on Application Specific Systems, Architectures and Processors. 2007; 180-185.
[30] CM Witternbrink, E Kilgariff, A Prabhu. FERMI GF100 GPU Architecture. *IEEE MICRO.* 2011; 31: 50-59.
[31] Xilinx. System Generator for DSP User Guide Release 10.1. http://www.xilinx.com/support/sw_manuals/sysgen_user.pdf. 2011.
[32] SW Zucker. Toward a model of Texture. *Computer Graphics and Image Processing.* 1976; 5: 190-202.

BIOGRAPHIES OF AUTHORS

Sanket Dessai received his BSc, MSc degree in Physics and MSc [Engg] in Real-Time Embedded Systems from Goa University, India in 2001, 2004 and Coventry University, UK in 2007. Since 2007 he has been a Assistant Professor at M.S. Ramaiah School of Advanced Studies (MSRSAS), Bengaluru in collaboration with Coventry University. His research interests include the field of System on Chip Design, Embedded Systems, MEMS/NEMS Engineering, Nanophysics and Nanotechnology, Solid State Physics and Engineering and Photonics

Krishna Bhushan Vutukuru received his BSc, MSc degree in Electronics and MSc [Engg] in Real-Time Embedded Systems from Osmania University, Acharya Nagarjuna University, India in 2005, 2007 and Coventry University, UK in 2012. His research interests include the field of System on Chip Design, Digital Design and FPGA and Embedded Systems.