

# **“Basics of 16 Bit SIMD Processor using Verilog”**

**Special Assignment Report**

*Submitted in Partial Fulfillment of the  
Requirements for completion of*

**Course on  
2EC601 Computer Architecture**

By

**Sanika Joshi (18BEC046)  
Vimoli Mehta (18BEC056)  
Suyash Malik (18BEC112)  
Dehit Trivedi (18BEC119)**



**Department of Electronics and Communication Engineering,  
Institute of Technology,  
Nirma University,  
Ahmedabad 382 481**

**April 2021**

# CERTIFICATE

This is to certify that the Comprehensive Evaluation Report entitled “Basics of 16 Bit SIMD Processor using Verilog” submitted by Mr. Dehit Trivedi towards the partial fulfillment of for completion of Course on 2EC601 Computer Architecture is the record of work carried out by him/her individually.

**Date:20/4/21**

**Faculty Coordinator: Dhaval Shah and Sachin Gajjar**

## **Undertaking for Originality of the Work**

I, Dehit Trivedi 18BEC119, give undertaking that the Comprehensive Evaluation Report entitled “Basics of 16 Bit SIMD Processor using Verilog” submitted by me, towards the partial fulfillment of for completion of Course on 2EC601 Computer Architecture, is the original work carried out by me and I give assurance that no attempt of plagiarism has been made. I understand that in the event of any similarity found subsequently with any other published work or any report elsewhere; it will result in severe disciplinary action.

Dehit Trivedi

Signature of the Student

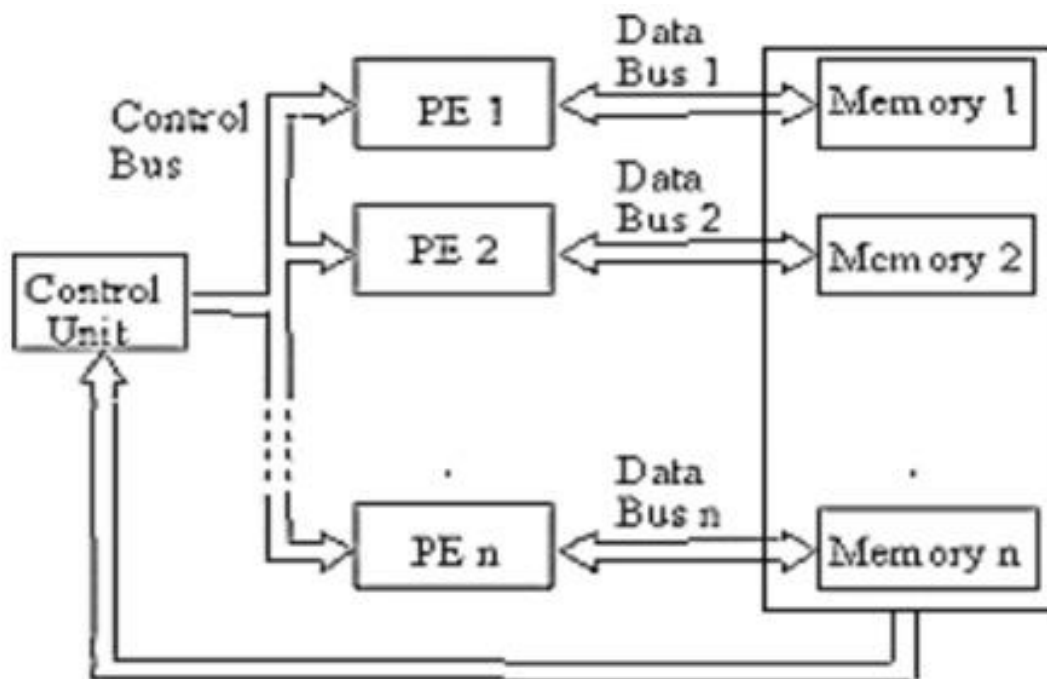
Date: 20/04/21

Place: Ahmedabad

## ABSTRACT

The volume and complexity of data processed by today's personal computers are increasing exponentially, placing incredible demands on the microprocessors. Computing performance that can be achieved by increasing the clock speed of a microprocessor is reaching to physical limits thus making the architectural solutions more prominent. Due to this an important architectural feature is added to recent microprocessors, single instruction multiple data (SIMD), which is a set of instructions that can speed up an application performance by allowing basic operation to be performed on multiple data elements in parallel with fewer instruction. It is considered to take advantage of data level parallelism within an algorithm. These became popular with the demanding increase on data streaming applications such as real-time games , video and image processing where large number of samples or pixels calculated with the same instruction.

The below **figure-1** represents the block diagram of a Basic SIMD processor. Here all the processing elements receive a single instruction from the control unit but this instruction operates on multiple different data sets.



*Figure1. Basic SIMD Processor*

# INDEX

Chapter No.	Title	Page No.
	<b>Abstract</b>	<b>4</b>
	<b>Index</b>	<b>5</b>
	<b>List of Figures</b>	<b>6</b>
	<b>List of Tables</b>	<b>6</b>
	<b>Nomenclature</b>	<b>7</b>
<b>1</b>	<b>Introduction</b>	<b>8-9</b>
	1.1 Introduction/ Prologue/Background	<b>8</b>
	1.1 Importance of the Topic	<b>8</b>
	1.2 Objective of the Report	<b>9</b>
	1.3 Scope of the Report	<b>10</b>
	1.4 Organization of the Rest of the Report	<b>10</b>
<b>2</b>	<b>Literature Review</b>	<b>11</b>
	2.1 Work in the area of the topic	<b>11-12</b>
<b>3</b>	<b>Design of SIMD</b>	<b>13</b>
<b>4</b>	<b>Implementing SIMD in Verilog</b>	<b>16</b>
<b>5</b>	<b>Opcode and instruction</b>	<b>16</b>
<b>6</b>	<b>Pros and Con of SIMD</b>	<b>18</b>
<b>7</b>	<b>SIMD processor performance trend2</b>	<b>19</b>
<b>9</b>	<b>Conclusion and future scope</b>	<b>20</b>
	<b>Reference</b>	
	<b>Appendix</b>	

## **LIST OF FIGURES**

<b>Figure No.</b>	<b>Title</b>	<b>Page No.</b>
<b>1</b>	Basic SIMD Processor	<b>4</b>
<b>2</b>	Simple SIMD Processor Architecture	<b>9</b>
<b>3</b>	The Interface of ALU Components	<b>13</b>
<b>4</b>	Datapath of SIMD Adder	<b>13</b>
<b>5</b>	Example of SIMD Shifter	<b>14</b>
<b>6</b>	1x16x16 multiplication	<b>14</b>
<b>7</b>	4x4x4 multiplication	<b>15</b>
<b>8</b>	2x8x8 multiplication	<b>15</b>
<b>9</b>	RTL and Simulation results of Adder	<b>20</b>
<b>10</b>	RTL and Simulation results of Shifter	<b>21</b>
<b>11</b>	RTL and Simulation results of Multiplier	<b>22</b>
<b>12</b>	RTL and Simulation results of Whole Processor	<b>23</b>

## **LIST OF TABLES**

<b>Table No.</b>	<b>Title</b>	<b>Page No.</b>
<b>1</b>	Implementing SIMD in Verilog	<b>16</b>
<b>2</b>	Opcode and Instructions	<b>19-20</b>
<b>3</b>	R-type instruction format	<b>17</b>
<b>4</b>	I-type instruction format	<b>17</b>
<b>5</b>	J-type instruction format	<b>17</b>

# NOMENCLATURE

## Subscripts

e.g.                      Example

## Abbreviations

DSP	Digital Signal Processing
KNI	Katmai's New Instructions
MDMX	MIPS Digital Media eXtension
VSI	Visual Instruction Set
AI	Artificial Intelligence
MIT	Massachusetts Institute of Technology
CM	Connection Machine
ALU	Arithmetic and Logic Unit
IF	Instruction Fetch
ID	Instruction Decode
EX	Execution Unit
WB	Write Back
MEM	Memory
BRAM	Block Random Access Memory
SIMD	Single Instruction Multiple Data
IC	Instruction Count
CPI	Cycle Per Instruction
CT	Cycle Time



# INTRODUCTION

## 1.1 Prologue

Performance in a computer system is defined by the amount of useful work accomplished by the computer system compared to the time and the resources used. There are several aspects for improving performance of computer systems. Researchers from several areas are striving to achieve higher performance ranging from algorithm, compiler, OS, and hardware designers.

Nowadays, most CPU designs contain at least some vector processing instructions, typically referred to as SIMD in which typically operate on a few vectors' elements per clock cycle in a pipeline. These vector processors run multiple mathematical operations on multiple data elements simultaneously. Thus, they have effects on the performance equation.

From the Iron law, we know that performance of a program is calculated by the formula: -

$$\text{Performance} = \text{IC} \cdot \text{CPI} \cdot \text{CT}$$

Where, **IC** is the number of instructions (instruction count),

**CPI** is the number of cycles per instruction

**CT** is the cycle time of the processor.

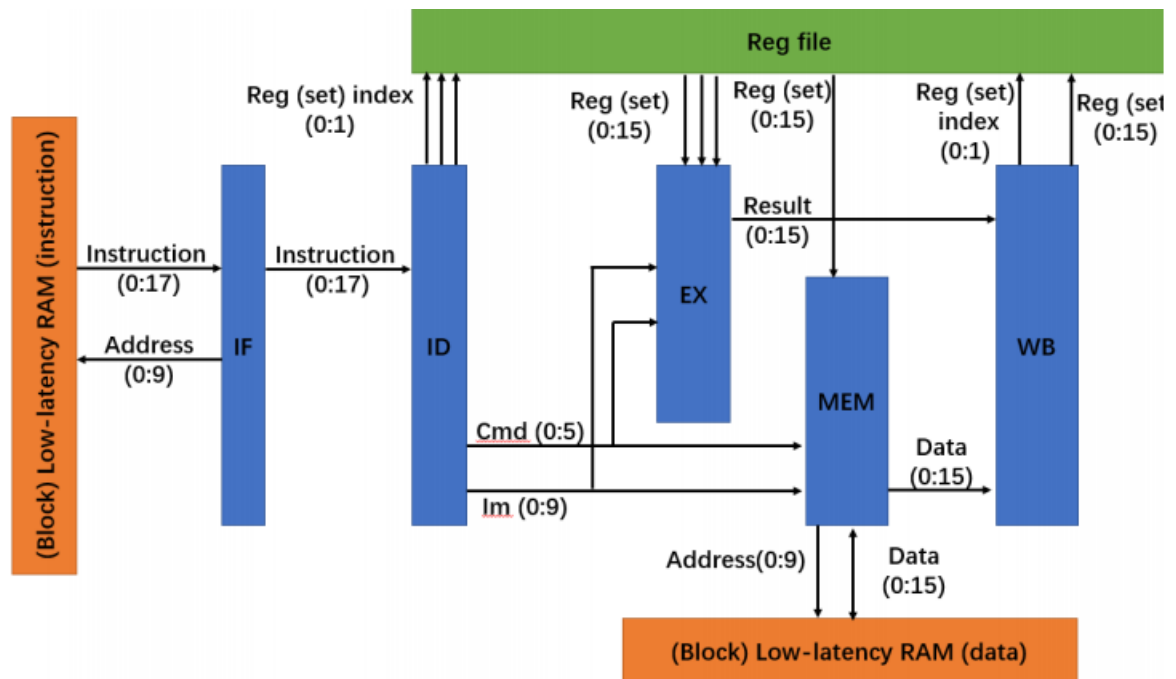
*Note:* The use of SIMD architecture changes IC and CPI values of a program

## 1.2 Introduction of Project

In This project we have implement a simple SIMD processor, core of which is a 16-bit SIMD ALU. 2's compliment calculations are implemented in this ALU. The ALU operation will take two clocks. The first clock cycle will be used to load values into the registers. The second will be for performing the operations. 6-bit opcodes are used to select the functions. The instruction code, including the opcode, will be 18-bit. The ALU will be embedded into a simple processor based on 5-stage, delay of each stage will be 1 cycle, meeting the delay of ALU, as shown in the figure below.

The 5 typical stages are **IF**, **ID**, **EX**, **MEM** and **WB**, without pipeline.

- (1) In the stage **IF**, a 10-bit address will be sent to an instruction **Block-RAM (BRAM)** to fetch 18-bit instruction.
- (2) In the stage **ID**, the instruction will be decoded and some of control registers will be set to control the following stage.
- (3) In the stage **EX**, **ALU** will process data in registers or implement some control commands, e.g. jump.
- (4) In the stage **MEM**, if the instruction is “store” or “load”, data would be read from/ written to data **BRAM**, based on instruction and address.
- (5) Finally, in the stage **WB**, data will be written back to register. The pins of clock, reset, address, data and **BRAM** enable will be exposed on the interface of processor. The architecture of processor is shown in **Figure 2**.



*Figure 2: Simple SIMD Processor Architecture*

### **1.3 Scope of the Report**

With the new enhancements in the processor architectures, the current modern processors started supporting 256-bit vector implementations. Moreover, the interest on SIMD architectures by the computer architecture research community is increasing. Most CPU's produced have SIMD architectures some of them are Intel SSE introduced in Pentium III processor and MMX ARM Neon introduced in Cortex-A8 and Cortex-A9 processors and MIPS MDMX. These architectures include instruction set extensions which allow both sequential and parallel instructions to be executed. Some architectures include separate SIMD coprocessors for handling these instructions. In the near future, the new powerful machines are expected to make new high-Performance multiple data applications available with the enhanced SIMD architectures.

### **1.4 Organization of Rest of the Report**

The rest of the Report is organized as follows:

Chapter II briefly summarizes the history of SIMD architectures and the related work. We describe the ALU (SIMD adder, SIMD multiplier and SIMD shifter) and We also provide a detailed description for our benchmarks in Chapter III. The results of the experiments are presented in Chapter IV. We finally conclude in Chapter V.

# Literature Review

## 2.1 History of SIMD Architectures

The first use of SIMD instructions was in the ILLIAC IV, which was completed in 1966. SIMD was the basis for vector supercomputers in the early 1970s such as the CDC Star-100 and the Texas Instruments ASC. It could operate on a "**vector**" of data with a single instruction. This architecture specially became common when Cray Inc. used it in their supercomputers. However, vector processing used in these machines nowadays are considered different from the SIMD machines.

Next was a Connection Machine (CM) which is a member of a series of massively parallel supercomputers which were alternatives to the traditional von Neumann architecture of computers by Danny Hillis at Massachusetts Institute of Technology (MIT) in the early 1980s. Starting with CM-1, the machines were intended originally for applications in artificial intelligence (AI) and symbolic processing, but later versions found greater success in the field of computational science. This actually started a new era in using SIMD for processing the data in parallel. However, current researches focus on using SIMD instructions in *desktop computers*. A lot of tasks desktop computers do these days like video processing and real-time gaming need to do the same operation on a bunch of data. So, companies tried to use this architecture in desktops. As one of the earliest attempts, Visual Instruction Set, or VIS, is a SIMD instruction set extension for SPARC V9 microprocessors developed by Sun Microsystems. There are five versions of VIS.

## 2.2 Details of Technology Used

MIPS introduced **MDMX** (MIPS Digital Media eXtension) and was developed to accelerate multimedia applications that were becoming more popular and common in the 1990s on RISC and CISC systems.

### 1. Intel's MMX

Intel made SIMD widely-used by introducing **MMX** extensions to the x86 architecture in 1996. MMX was the first implementation of SIMD processing in a mainstream processor. Intel backed MMX with an aggressive marketing campaign. Initially MMX was meant for games and other entertainment software but eventually, mainstreams programs such as Photoshop became optimized for MMX.

Unfortunately, Intel's limited implementation of SIMD in MMX drew much criticism from the computing community before the first Pentium with MMX was even sold. 3D rendering and video decoding, two of SIMD's strongest applications, fell short on MMX enabled Pentiums. Also, due to the difficult MMX programming API, Intel was concerned that developer would not write code for MMX properly. Although with Intel's market power and resources, MMX is widely supported.

With the launch of MMX, Intel was already working on fixing the limitations of MMX with MMX2. MMX2, otherwise known as Katmai's New Instructions (KNI) enhances the Pentium II's MMX instruction set and speeds up *floating-point operations*. All of the applications that MMX was originally intended for will probably actually function well on MMX2.

## **2. AltiVec**

**AltiVec** is Motorola's first implementation of SIMD in its mainstream processor, the PowerPC. It is a clean implementation that is more powerful than MMX2. Motorola is not only targeting *desktop computers* but also *embedded systems* where the chips could be used in devices such as network routers. Unfortunately, only Macintosh computers from Apple Computers use PowerPC chips. This limits the consumer audience that may garner advantages from AltiVec. But Apple computer did play a role in developing AltiVec and plans on using it extensively as soon as the chips become available. Also, Adobe already has made plans on developing an AltiVec enabled version of Photoshop for the MacOS. Due to the fact that the PowerPC has a much smaller market base than the Intel processors, the impact of AltiVec may not have a large consumer impact.

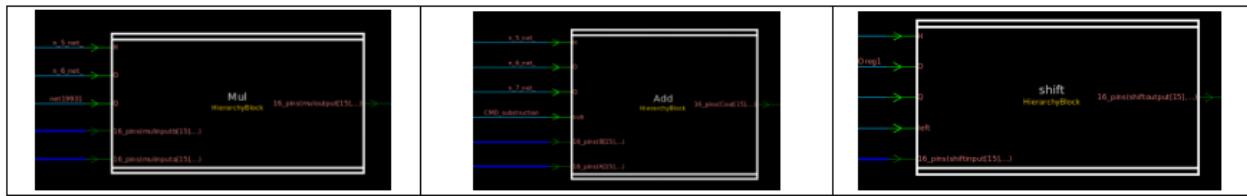
### **2.3 Current Trends:**

Due to Motorola's introduction of AltiVec system in its PowerPC's which also was used in IBM's POWER systems caused the Intels respond which was *introduction of SSE*.

Today, SIMD is used for various applications like image processing, 3D rendering, video and sound applications, speech recognition, networking and DSP functions. This report focuses on the design of a 16 -SIMD processor using Verilog language.

### 3. Design of SIMD ALU

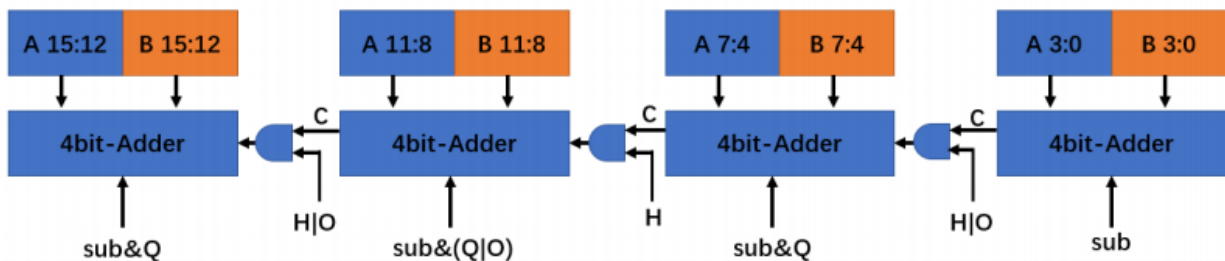
The ALU consists of three SIMD basic computation units: SIMD adder, SIMD multiplier and SIMD shifter. These three components can be reused for the computation of data with different width (4-bit, 8-bit or 16-bit) and can handle all the instructions (listed in Section 4) in the design specification. Each of them is controlled by three input port, H (for 16-bit), O (for 8-bit) and Q (for 4-bit), indicating the kind of input data, so that the unit can handle the data properly, as shown below in **figure 3**.



**Figure 3:** The Interface of ALU Components

#### 1) SIMD adder:

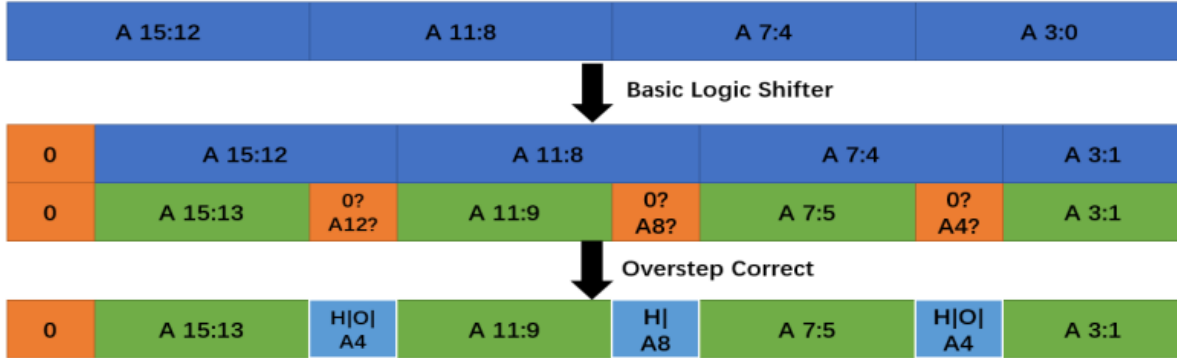
The SIMD adder is implemented based on 4 4-bit adders. To reuse the adder for data with different width, the input signals, H, O and Q, control the forwarding of the carries between the adders. Moreover, the adder can also support subtraction, by flipping the second operand and add one to it when the signal sub is set. The data path of the SIMD adder is shown in **Figure 4**.



**Figure 4:** Datapath of SIMD Adder

## 2) SIMD shifter:

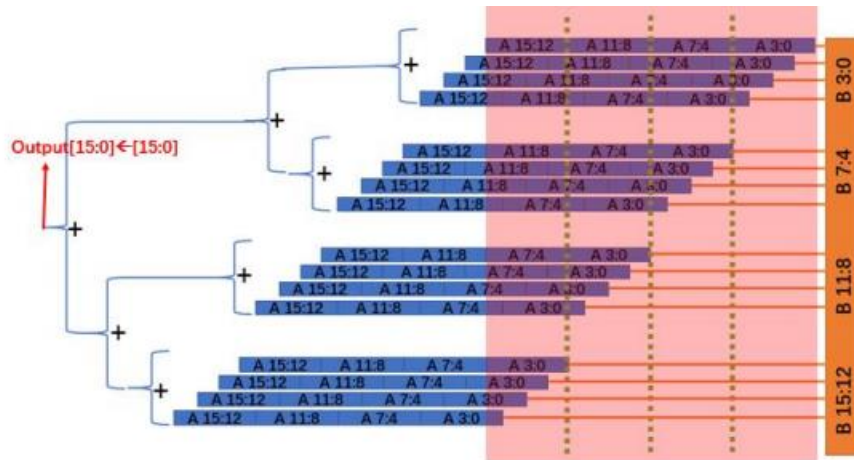
The SIMD shifter can also support data with different width. It is based on two 16-bit shifters and necessary overstep-correct logic. To illustrate the implementation of the shifter, the example based on logic-left-shift is shown in **Figure 5**. The shifter will determine whether the MSB of 4-bit block should be set to 0 or just inherit the value from the LSB from 4-bit block in front of it, according to the input signal, H and O.



*Figure 5: Example of SIMD Shifter*

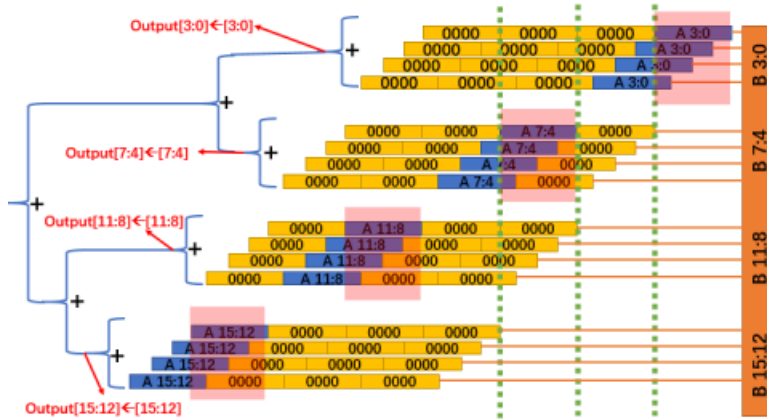
## 3) SIMD multiplier:

The SIMD multiplier is implemented with adders and shifters. To make it support multiplication with different data width, the input signal, H, O and Q, are used to control the input operands of the adders and select corresponding outputs for the target data width. The 1x16x16 multiplication is implemented as a typical multiplier, as **Figure 6**, where adder tree is utilized. The least significant 16 bits of the sum is the output of multiplication.

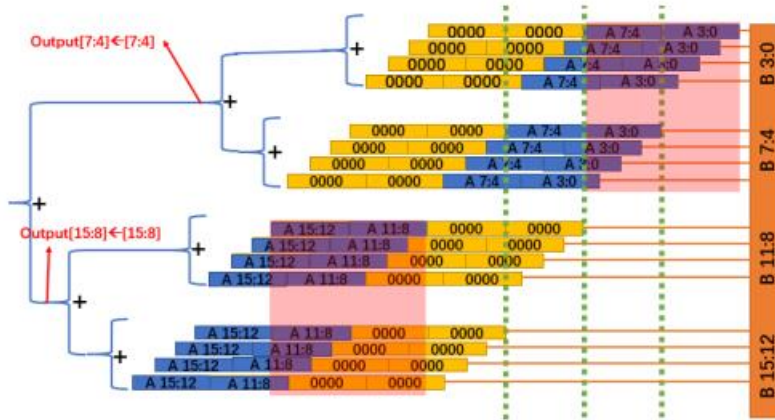


*Figure 6: 1x16x16 multiplication*

To reuse the structure of 1x16x16 multiplication implemented as above, for the input signal, H, O and Q, are used to select the inputs of the adders, to control the adders to only sum up the range of bits. The 4x4x4 multiplication is implemented as shown **Figure 7**, where for each adder, just a part of input bit is valid while other bits will be set to 0. Similarly, the 2x8x8 multiplication is shown in **Figure 8**.



*Figure 7: 4x4x4 multiplication*



*Figure 8: 2x8x8 multiplication*



### 3.2 SIMD Implementation in Verilog

<b>Input</b>	18-bit instruction, 16-bit i/p data, clk and reset
<b>Output</b>	16-bit o/p data, 10-bit data address, 10-bit instruction address
<b>Important variable</b>	3-bit current state, 10-bit program counter, and next program counter.
<b>Register file</b>	4 sets of 16-bit H, 3 sets of 16-bit O, 3 sets of 16-bit Q, 10-bit immediate register
<b>Index for register file</b>	2-bit R0, R1, R2, and R3
<b>Control signals</b>	Hreg1, Hreg2, Hreg3, Him, Oreg1, Oreg, Oreg3, Oim, Qreg1, Qreg2, Qreg3, Qim.

### 3.3 Register Instruction Formats of Processor

- A 16-bit register is used to store two data values each of 08 bits.
- Used for 8 bit registers (3 sets)- {O1,O2},{O3,O4},{O5,O6}



- A 16-bit register is used to store four data values each of 04 bits.
- Used for 4 bit registers(3sets)-{Q1,Q2,Q3,Q4},{Q5,Q6,Q7,Q8},{Q9,Q10,Q11,Q12}



### 1. R-type Instruction Format

The instruction is considered of total 18 bits.

Opcode of [5 bits]	Index Register 0 [2 bits]	Index Register 1 [2 Bits]	Index Register 2 [2 bits]	Index Register 3 [2 bits]
Opcode [17:12]	R0[11:10]	R1[5:4]	R2[3:2]	R3 [1:0]

Example :    add16bit H3, H2  
                 000000\_00000000\_11\_10  
                 Opcode \*\_H3\_H2

### 2. I-type Instruction Format

Opcode of [5 bits]	Index Register 0 [2 bits]	Immediate Operand
Opcode [17:12]	R0[11:10]	Imm register [9:0]

Example : load16bit H0, immediate  
                 100110\_00\_0000000000  
                 Opcode\_H0\_immediate

### 3. J-type instruction format

Opcode of [5 bits]	Index Register 0 [2 bits]	Immediate Operand
Opcode [17:12]	R0[11:10]	Imm register [9:0]

**Example** : loop jump ,im

100100\_00\_0000010000

Opcode\_\*\_immediate

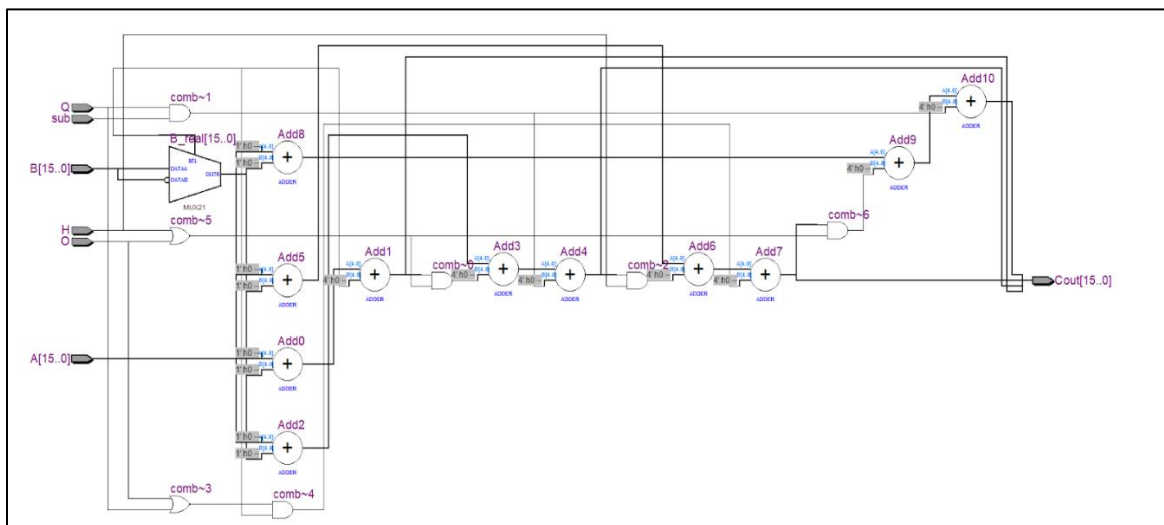
### 3.4 Opcode and Instruction

Opcode	Instruction
000000	Add 16 bit
000001	Add 8 bit
000010	Add 4 bit
000011	Add 16 immediate
000100	Add 8 bit immediate
000101	Add 4 bit immediate
000110	Sub 16 bit
000111	Sub 8 bit
001000	Sub 4 bit
001001	Sub 16 bit immediate
001010	Sub 8 bit immediate
001011	Sub 4 bit immediate
001100	Mul 16 bit
001101	Mul 8 bit
001110	Mul 4 bit
001111	Mul 16 bit immediate
010000	Mul 8 bit immediate
010001	Mul 4 bit immediate
010101	Lsl 16 bit
010110	Lsl 8 bit
010111	Lsl 4 bit
011000	Lsr 16 bit
011001	Lsr 8 bit

011010	Lsr 4 bit
100110	Load 16 bit
100111	Load 8 bit
101000	Load 4 bit
101001	Store 16 bit
101010	Store 8 bit
101011	Store 4 bit
100100	Loop Jump to value in immediate

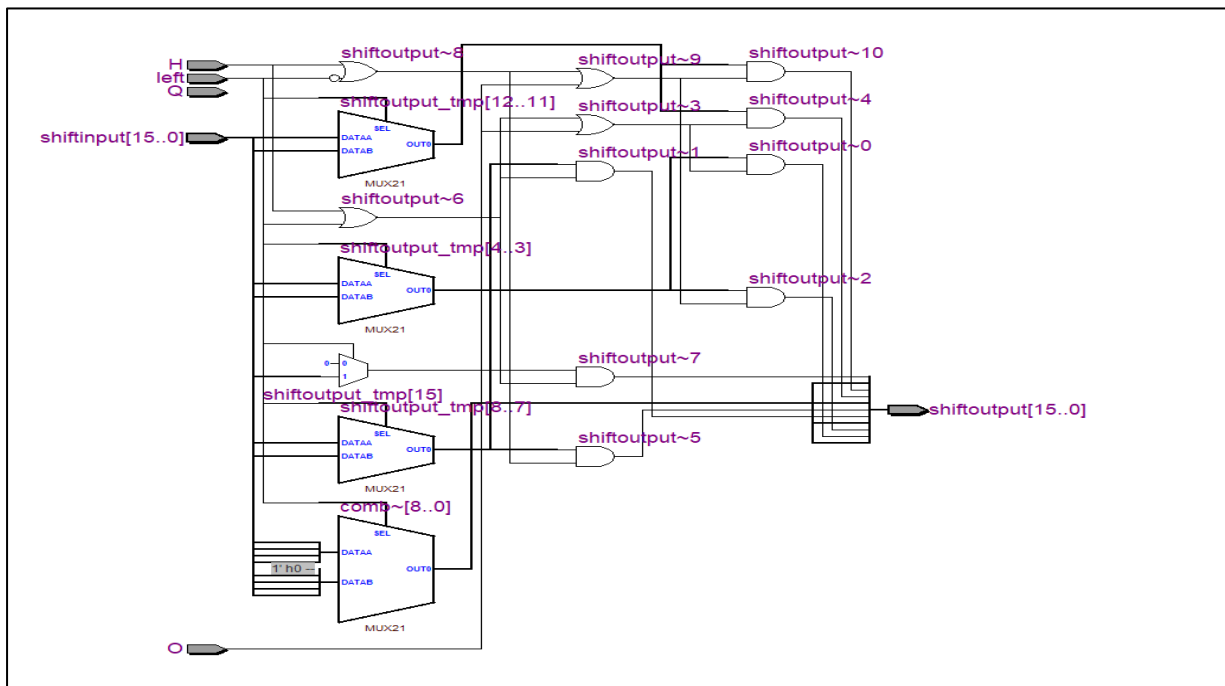
## 4. Simulation and Results

### 1. Adder



/SIMDadd/Cout	0000000000000011	0000000000000011
/SIMDadd/sub	St0	
/SIMDadd/Q	St0	
/SIMDadd/O	St0	
/SIMDadd/H	St1	
/SIMDadd/Cout	0000000000000011	0000000000000011
/SIMDadd/C3	00000	00000
/SIMDadd/C2	00000	00000
/SIMDadd/C1	00000	00000
/SIMDadd/C0	00011	00011
/SIMDadd/B_real	0000000000000010	0000000000000010
/SIMDadd/B	0000000000000010	0000000000000010
/SIMDadd/A	0000000000000001	0000000000000001









## 2. Shifter



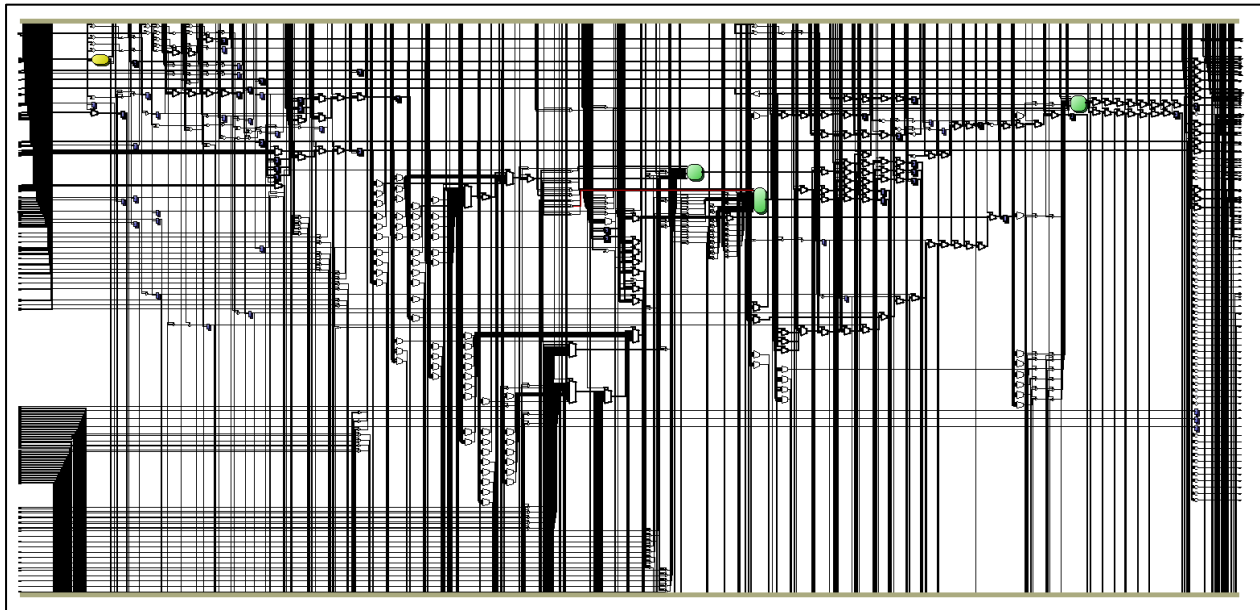
/SIMDshifter/shifto...	00000000000000100	00000000000000100
/SIMDshifter/shifto...	00000000000000100	00000000000000100
/SIMDshifter/shiftn...	0000000000000010	0000000000000010
/SIMDshifter/right_...	000000000000001	000000000000001
/SIMDshifter/left_shift	0000000000000010	0000000000000010
/SIMDshifter/left	St1	
/SIMDshifter/Q	St0	
/SIMDshifter/O	St0	
/SIMDshifter/H	St1	

### 3. Multiplier



+  /SIMDmultiply/sel1	1111111111111111	1111111111111111	
+  /SIMDmultiply/sel0	1111111111111111	1111111111111111	
+  /SIMDmultiply/muloutput	0000000000000010	0000000000000010	
+  /SIMDmultiply/mulinputb	0000000000000010	0000000000000010	
+  /SIMDmultiply/mulinputa	0000000000000001	0000000000000001	
+  /SIMDmultiply/a9	0000000000000000	0000000000000000	
+  /SIMDmultiply/a8	0000000000000000	0000000000000000	
+  /SIMDmultiply/a7	0000000000000000	0000000000000000	

## 4. RTL of whole Processor



## 5. Pros and Cons of SIMD

### Pros:

- Require Lower Instruction Bandwidth: Reduced by fewer fetches and decodes
- Easier Addressing of Main Memory: Load/Store units access memory with known pattern
- Elimination of Memory Wastage
  - Unlike cache access, every data element that is requested by the processor is actually used – no cache misses
  - Latency only occurs once per vector during pipelined loading
- Simplification of Control Hazards: Loop-related control hazards from the loop are eliminated
- Scalable Platform: Increase performance by using more hardware resources
- Reduced Code Size: Short, single instruction can describe N operations

### Cons:

- Larger registers and functional units use more chip area and power
- Difficult to parallelize some algorithms (Amdahl's Law)
- Parallelization requires explicit instructions from the programmer

## **6. Conclusion**

With the new enhancements in the processor architectures, the current modern processors started supporting 256-bit vector implementations. Moreover, the interest on SIMD architectures by the computer architecture research community is increasing. Most CPU's produced have SIMD architectures some of them are Intel SSE introduced in Pentium III processor and MMX ARM Neon introduced in Cortex-A8 and Cortex-A9 processors and MIPS MDMX. In the near future, the new powerful machines are expected to make new high-Performance multiple data applications available with the enhanced SIMD architectures.

### **6.2 Future Scope**

There is no doubt that future processors will differ significantly from the current designs and will reshape the way of thinking about programming. SIMD is one of the most important advancements of modern CPUs. This project deals with SIMD and how you to realize a 16-bit SIMD processor using Verilog. SIMD architectures have numerous applications in the field of audio, video and signal processing systems. It also states that SIMD is used to increase the performance of the processor and details of work related to the topic and various tool and technologies used.



## 7. References

- [1] Solmaz, G., Rahmatizadeh, R. and Ahmadian, M., A study on SIMD architecture.
- [2] Li, S.Y., Cheuk, G.C., Lee, K.H. and Leong, P.H.W., 2003, April. FPGA-based SIMD processor. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.* (pp. 267-268). IEEE.
- [3]<https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html>
- [4]<http://ftp.cvut.cz/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html>

# Appendix

## Verilog code for SIMD processor

```
//Computer Archietecture : Speacial Assignment  
//Vimoli Mehta, Sanika Joshi, Suyash mallik, Dehit Trivedi
```

```
module SIMD_test(  
    input clk,  
    input rst,  
    input [17:0] instruction_in,  
    input [15:0] data_in,  
    output [15:0] data_out,  
    output [9:0] instruction_address,  
    output [9:0] data_address,  
    output data_R,  
    output data_W,  
    output done  
);  
  
wire [5:0] opcode = instruction_in[17:12];  
  
parameter [2:0] STATE_IDLE = 0;  
parameter [2:0] STATE_IF = 1;  
parameter [2:0] STATE_ID = 2;  
parameter [2:0] STATE_EX = 3;  
parameter [2:0] STATE_MEM = 4;  
parameter [2:0] STATE_WB = 5;  
parameter [2:0] STATE_HALT = 6;  
  
reg [2:0] current_state;  
reg [9:0] PC,next_PC;  
reg [9:0] current_data_address;  
reg rdata_en;  
reg wdata_en;  
reg [15:0] data_out_reg;  
  
assign data_out = data_out_reg;  
assign data_R = rdata_en;  
assign data_W = wdata_en;  
assign data_address = current_data_address;  
  
//data register  
reg [15:0] H[0:3];  
reg [15:0] Oset[0:2];  
reg [15:0] Qset[0:2];  
reg [9:0] LC;  
reg [9:0] im_reg;  
  
//control register  
reg CMD_addition;  
reg CMD_multiplication;  
reg CMD_substruction;  
reg CMD_mul_accumulation;  
reg CMD_logic_shift_right;
```

```

reg CMD_logic_shift_left;
reg CMD_and;
reg CMD_or;
reg CMD_not;
reg CMD_load;
reg CMD_store;
reg CMD_set;
reg CMD_loopjump;
reg CMD_setloop;

//cmd type
reg Hreg1,Hreg2,Hreg3,Him,Oreg1,Oreg2,Oreg3,Oim,Qreg1,Qreg2,Qreg3,Qim;

//result register
reg [15:0] result_reg_add;
reg [15:0] result_reg_sub;
reg [15:0] result_reg_mul;
reg [15:0] result_reg_mac;
reg [15:0] result_reg_Lshift;
reg [15:0] result_reg_Rshift;
reg [15:0] result_reg_and;
reg [15:0] result_reg_or;
reg [15:0] result_reg_not;
reg [15:0] result_reg_load;
reg [15:0] result_reg_store;
reg [15:0] result_reg_set;
reg [1:0] R0,R1,R2,R3;

wire [15:0] comp_input_A =
Hreg1?(H[R0]):((Hreg2|Hreg3)?(H[R2]):(Oreg1?(Oset[R0]):((Oreg2|Oreg3)?(Oset[R2]
):(Qreg1?(Qset[R0]):((Qset[R2])))))));
wire [15:0] comp_input_B =
Hreg1?(im_reg):((Hreg2|Hreg3)?(H[R3]):(Oreg1?({im_reg[7:0],im_reg[7:0]}):((Ore
g2|Oreg3)?(Oset[R3]):(Qreg1?({im_reg[3:0],im_reg[3:0],im_reg[3:0],im_reg[3:0]}
):((Qset[R3])))))));

wire [15:0] Add_output_Cout;
wire [15:0] Mul_output_Cout;
wire [15:0] MAC_output_Cout;

SIMDadd Add(
    .A(comp_input_A),
    .B(comp_input_B),
    .H(Hreg1|Hreg2|Hreg3),
    .O(Oreg1|Oreg2|Oreg3),
    .Q(Qreg1|Qreg2|Qreg3),
    .sub(CMD_substruction),
    .Cout(Add_output_Cout)
);

wire [15:0] shiftinput = Hreg1?(H[R3]):(Oreg1?(Oset[R3]):(Qset[R3]));
wire [15:0] shiftoutput;

SIMDshifter shift(
    .shiftinput(shiftinput),

```

```

        .H(Hreg1),
        .O(Oreg1),
        .Q(Qreg1),
        .left(CMD_logic_shift_left),
        .shiftoutput(shiftoutput)
    );

SIMDmultiply Mul(
    .mulinputa(comp_input_A),
    .mulinputb(comp_input_B),
    .H(Hreg1|Hreg2|Hreg3),
    .O(Oreg1|Oreg2|Oreg3),
    .Q(Qreg1|Qreg2|Qreg3),
    .muloutput(Mul_output_Cout)
);

//some signal

assign instruction_address = PC;
assign done = current_state == STATE_HALT;

always @(posedge clk) //state machine
begin
    if (rst)
    begin
        current_state <= STATE_IDLE;
        PC <= 0;
    end
    else
    begin
        if (opcode == 63) current_state <= STATE_HALT;
        else
        if (current_state == STATE_IDLE)
        begin
            current_state <= STATE_IF;
        end
        else if (current_state == STATE_IF)
        begin
            $display("===== another instruction =====");
            $display("H00: %b",H[0]);
            $display("H01: %b",H[1]);
            $display("H10: %b",H[2]);
            $display("H11: %b",H[3]);
            $display("Oset00: %b",Oset[0]);
            $display("Oset01: %b",Oset[1]);
            $display("Oset10: %b",Oset[2]);
            $display("Qset00: %b",Qset[0]);
            $display("Qset01: %b",Qset[1]);
            $display("Qset10: %b",Qset[2]);
            $display("        -- execute --        ");
            current_state <= STATE_ID;
        end
        else if (current_state == STATE_ID)
        begin
            current_state <= STATE_EX;
        end
    end
end

```

```

        else if (current_state == STATE_EX)
        begin
            current_state <= STATE_MEM;
        end
        else if (current_state == STATE_MEM)
        begin
            current_state <= STATE_WB;
        end
        else if (current_state == STATE_WB)
        begin
            current_state <= STATE_IF;
            PC <= next_PC;
        end
    end
end

always @(posedge clk)//STATE_ID
begin
    if (rst || current_state == STATE_IDLE || current_state == STATE_IF)
    begin
        CMD_addition <= 0;
        CMD_multiplication <= 0;
        CMD_substruction <= 0;
        CMD_mul_accumulation <= 0;
        CMD_logic_shift_right <= 0;
        CMD_logic_shift_left <= 0;
        CMD_and <= 0;
        CMD_or <= 0;
        CMD_not <= 0;
        CMD_load <= 0;
        CMD_store <= 0;
        CMD_set <= 0;
        CMD_loopjump <= 0;
        CMD_setloop <= 0;
        CMD_halt <= 0;
        Hreg1<=0;
        Hreg2<=0;
        Hreg3<=0;
        Him<=0;
        Oreg1<=0;
        Oreg2<=0;
        Oreg3<=0;
        Oim<=0;
        Qreg1<=0;
        Qreg2<=0;
        Qreg3<=0;
        Qim<=0;
        im_reg <= 10'b0000000000;
        R0 <= 0;
        R1 <= 0;
        R2 <= 0;
        R3 <= 0;
    end
    else
    begin
        if (current_state == STATE_ID)
        begin

```

```

//cmd
CMD_addition <= (opcode<=5);
CMD_substruction <= (opcode>=6)&&(opcode<=11);
CMD_multiplication <= (opcode>=12)&&(opcode<=17);

CMD_logic_shift_left <= (opcode>=21)&&(opcode<=23);
CMD_logic_shift_right <= (opcode>=24)&&(opcode<=26);
CMD_and <= (opcode>=27)&&(opcode<=29);
CMD_or <= (opcode>=30)&&(opcode<=32);
CMD_not <= (opcode>=33)&&(opcode<=35);
CMD_loopjump <= opcode==36;
CMD_setloop <= opcode==37;
CMD_load <= (opcode>=38)&&(opcode<=40);
CMD_store <= (opcode>=41)&&(opcode<=43);
CMD_set <= (opcode>=44)&&(opcode<=46);

Hreg1<=(opcode==3) || (opcode==9) || (opcode==15) || (opcode==21) || (opcode==24) || (opcode==33) || (opcode==38) || (opcode==41) || (opcode==44);

Hreg2<=(opcode==0) || (opcode==6) || (opcode==12) || (opcode==27) || (opcode==30);
Hreg3<=(opcode==18);

Him<=(opcode==3) || (opcode==9) || (opcode==15) || (opcode==38) || (opcode==41) || (opcode==44);

Oreg1<=(opcode==4) || (opcode==10) || (opcode==16) || (opcode==22) || (opcode==25) || (opcode==34) || (opcode==39) || (opcode==42) || (opcode==45);

Oreg2<=(opcode==1) || (opcode==7) || (opcode==13) || (opcode==28) || (opcode==31);
Oreg3<=(opcode==19);

Oim<=(opcode==4) || (opcode==10) || (opcode==16) || (opcode==39) || (opcode==42) || (opcode==45);

Qreg1<=(opcode==5) || (opcode==11) || (opcode==17) || (opcode==23) || (opcode==26) || (opcode==35) || (opcode==40) || (opcode==43) || (opcode==46);

Qreg2<=(opcode==2) || (opcode==8) || (opcode==14) || (opcode==29) || (opcode==32);
Qreg3<=(opcode==20);

Qim<=(opcode==5) || (opcode==11) || (opcode==17) || (opcode==40) || (opcode==43) || (opcode==46);

im_reg <= instruction_in[9:0];
R0 <= instruction_in[11:10];
R1 <= instruction_in[5:4];
R2 <= instruction_in[3:2];
R3 <= instruction_in[1:0];
$display("PC: %0d : instruction = %b", PC,instruction_in);
end
end
end

always @(posedge clk)//STATE_EX

```

```

begin
    if (rst || current_state == STATE_IDLE || current_state == STATE_IF)
    begin
        result_reg_add <= 0;
        result_reg_sub <= 0;
        result_reg_mul <= 0;
        result_reg_mac <= 0;
        result_reg_Lshift <= 0;
        result_reg_Rshift <= 0;
        result_reg_and <= 0;
        result_reg_or <= 0;
        result_reg_not <= 0;
        result_reg_set <= 0;
        current_data_address <= 0;
        rdata_en <= 0;
        wdata_en <= 0;
        if (rst)
        begin
            next_PC <= 0;
        end
    end
end

else if (current_state == STATE_EX)
begin
    if (CMD_addition) // do addition
    begin
        result_reg_add <= Add_output_Cout;
        if (Hreg2)
        begin
            $display("add16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
        end
        else if (Oreg2)
        begin
            $display("add8bit R%d=%b R%d=%b",
R2,Oset[R2],R3,Oset[R3]);
        end
        else if (Qreg2)
        begin
            $display("add4bit R%d=%b R%d=%b",
R2,Qset[R2],R3,Qset[R3]);
        end
        else if (Him)
        begin
            $display("add16bit R%d=%b im=%b", R0,H[R0],im_reg);
        end
        else if (Oim)
        begin
            $display("add8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
        end
        else if (Qim)
        begin
            $display("add4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
        end
    end

    else if (CMD_substruction) // do substruction

```

```

begin
    result_reg_sub <= Add_output_Cout;
    if (Hreg2)
    begin
        $display("sub16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
    end
    else if (Oreg2)
    begin
        $display("sub8bit R%d=%b R%d=%b",
R2,Oset[R2],R3,Oset[R3]);
    end
    else if (Qreg2)
    begin
        $display("sub4bit R%d=%b R%d=%b",
R2,Qset[R2],R3,Qset[R3]);
    end
    else if (Him)
    begin
        $display("sub16bit R%d=%b im=%b", R0,H[R0],im_reg);
    end
    else if (Oim)
    begin
        $display("sub8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
    end
    else if (Qim)
    begin
        $display("sub4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
    end
end

else if (CMD_multiplication) // do multiplication
begin
    result_reg_mul<=Mul_output_Cout;
    if (Hreg2)
    begin
        $display("mul16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
    end
    else if (Oreg2)
    begin
        $display("mul8bit R%d=%b R%d=%b",
R2,Oset[R2],R3,Oset[R3]);
    end
    else if (Qreg2)
    begin
        $display("mul4bit R%d=%b R%d=%b",
R2,Qset[R2],R3,Qset[R3]);
    end
    else if (Him)
    begin
        $display("mul16bit R%d=%b im=%b", R0,H[R0],im_reg);
    end
    else if (Oim)
    begin
        $display("mul8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
    end
    else if (Qim)
    begin

```



```

        $display("mul4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
    end
end

else if (CMD_mul_accumulation) // do mac
begin
    result_reg_mac <= Add_output_Cout;
    if (Hreg3)
    begin
        $display("MAC16bit R%d=%b R%d=%b R%d=%b",
R1,H[R1],R2,H[R2],R3,H[R3]);
    end
    else if (Oreg3)
    begin
        $display("MAC8bit R%d=%b R%d=%b R%d=%b",
R1,Oset[R1],R2,Oset[R2],R3,Oset[R3]);
    end
    else if (Qreg3)
    begin
        $display("MAC4bit R%d=%b R%d=%b R%d=%b",
R1,Qset[R1],R2,Qset[R2],R3,Qset[R3]);
    end
    end
end

else if (CMD_logic_shift_right) // do shift right
begin
    result_reg_Rshift <= shiftoutput;
    if (Hreg1)
    begin
        $display("Rshift16bit R%d=%b", R3,H[R3]);
    end
    else if (Oreg1)
    begin
        $display("Rshift8bit R%d=%b", R3,Oset[R3]);
    end
    else if (Qreg1)
    begin
        $display("Rshift4bit R%d=%b", R3,Qset[R3]);
    end
    end
end

else if (CMD_logic_shift_left) // do shift left
begin
    result_reg_Lshift <= shiftoutput;
    if (Hreg1)
    begin
        $display("Lshift16bit R%d=%b", R3,H[R3]);
    end
    else if (Oreg1)
    begin
        $display("Lshift8bit R%d=%b", R3,Oset[R3]);
    end
    else if (Qreg1)
    begin
        $display("Lshift4bit R%d=%b", R3,Qset[R3]);
    end
    end
end
end

```

```

else if (CMD_and) // do and
begin
    if (Hreg2)
    begin
        result_reg_and <= H[R2] & H[R3];
        $display("and16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
    end
    else if (Oreg2)
    begin
        result_reg_and <= Oset[R2] & Oset[R3];
        $display("and8bit R%d=%b R%d=%b",
R2,Oset[R2],R3,Oset[R3]);
    end
    else if (Qreg2)
    begin
        result_reg_and <= Qset[R2] & Qset[R3];
        $display("and4bit R%d=%b R%d=%b",
R2,Qset[R2],R3,Qset[R3]);
    end
end

else if (CMD_or) // do or
begin
    if (Hreg2)
    begin
        result_reg_or <= H[R2] | H[R3];
        $display("or16bit R%d=%b R%d=%b", R2,H[R2],R3,H[R3]);
    end
    else if (Oreg2)
    begin
        result_reg_or <= Oset[R2] | Oset[R3];
        $display("or8bit R%d=%b R%d=%b", R2,Oset[R2],R3,Oset[R3]);
    end
    else if (Qreg2)
    begin
        result_reg_or <= Qset[R2] | Qset[R3];
        $display("or4bit R%d=%b R%d=%b", R2,Qset[R2],R3,Qset[R3]);
    end
end

else if (CMD_not) // do not
begin
    if (Hreg1)
    begin
        result_reg_not <= ~H[R3];
        $display("not16bit R%d=%b", R3,H[R3]);
    end
    else if (Oreg1)
    begin
        result_reg_not <= ~Oset[R3];
        $display("not8bit R%d=%b", R3,Oset[R3]);
    end
    else if (Qreg1)
    begin
        result_reg_not <= ~Qset[R3];

```

```

        $display("not4bit R%d=%b", R3,Qset[R3]);
    end
end

else if (CMD_set) // do set
begin
    if (Hreg1)
    begin
        result_reg_set <= im_reg;
        $display("set16bit R%d im=%b", R0,im_reg);
    end
    else if (Oreg1)
    begin
        result_reg_set[7:0] <= im_reg;
        result_reg_set[15:8] <= im_reg;
        $display("set8bit R%d im=%b", R0,im_reg);
    end
    else if (Qreg1)
    begin
        result_reg_set[3:0] <= im_reg;
        result_reg_set[7:4] <= im_reg;
        result_reg_set[11:8] <= im_reg;
        result_reg_set[15:12] <= im_reg;
        $display("set4bit R%d im=%b", R0,im_reg);
    end
end

else if (CMD_load) // do load
begin
    rdata_en <= 1;
    current_data_address <= im_reg;
    if (Hreg1)
    begin
        $display("load16bit R%d im=%b", R0,im_reg);
    end
    else if (Oreg1)
    begin
        $display("load8bit R%d im=%b", R0,im_reg);
    end
    else if (Qreg1)
    begin
        $display("load4bit R%d im=%b", R0,im_reg);
    end
end

else if (CMD_store) // do store
begin
    wdata_en <= 1;
    rdata_en <= 1;
    current_data_address <= im_reg;

    if (Hreg1)
    begin
        data_out_reg <= H[R0];
        $display("store16bit R%d=%b im=%b", R0,H[R0],im_reg);
    end
end

```

```

        else if (Oreg1)
        begin
            data_out_reg <= Oset[R0];
            $display("store8bit R%d=%b im=%b", R0,Oset[R0],im_reg);
        end
        else if (Qreg1)
        begin
            data_out_reg <= Qset[R0];
            $display("store4bit R%d=%b im=%b", R0,Qset[R0],im_reg);
        end
    end

    if (CMD_loopjump)
    begin
        $display("loopjump LC=%d im=%d", LC,im_reg);
        if (LC != 0)
        begin
            next_PC <= im_reg;
            LC <= LC - 1;
        end
        else
        begin
            next_PC <= next_PC + 1;
        end
    end
    else
        next_PC <= next_PC + 1;

    if (CMD_setloop)
    begin
        $display("setloop im=%d", im_reg);
        LC <= im_reg;
    end
end
end

always @(posedge clk)//STATE_WB
begin
    if (rst || current_state == STATE_IDLE || current_state == STATE_IF)
    begin
        end

    else if (current_state == STATE_WB)
    begin

        if (CMD_addition) // do addition
        begin
            if (Hreg2)
            begin
                H[R2] <= result_reg_add;
            end
            else if (Oreg2)
            begin
                Oset[R2] <= result_reg_add;
            end
            else if (Qreg2)
            begin

```

```

        Qset[R2] <= result_reg_add;
    end
    else if (Him)
    begin
        H[R0] <= result_reg_add;
    end
    else if (Oim)
    begin
        Oset[R0] <= result_reg_add;
    end
    else if (Qim)
    begin
        Qset[R0] <= result_reg_add;
    end
end

else if (CMD_substruction) // do substruction
begin
    if (Hreg2)
    begin
        H[R2] <= result_reg_sub;
    end
    else if (Oreg2)
    begin
        Oset[R2] <= result_reg_sub;
    end
    else if (Qreg2)
    begin
        Qset[R2] <= result_reg_sub;
    end
    else if (Him)
    begin
        H[R0] <= result_reg_sub;
    end
    else if (Oim)
    begin
        Oset[R0] <= result_reg_sub;
    end
    else if (Qim)
    begin
        Qset[R0] <= result_reg_sub;
    end
end
end

else if (CMD_multiplication) // do multiplication
begin
    if (Hreg2)
    begin
        H[R2] <= result_reg_mul;
    end
    else if (Oreg2)
    begin
        Oset[R2] <= result_reg_mul;
    end
    else if (Qreg2)
    begin

```

```

        Qset[R2] <= result_reg_mul;
    end
    else if (Him)
    begin
        H[R0] <= result_reg_mul;
    end
    else if (Oim)
    begin
        Oset[R0] <= result_reg_mul;
    end
    else if (Qim)
    begin
        Qset[R0] <= result_reg_mul;
    end
end

else if (CMD_mul_accumulation) // do mac
begin
    if (Hreg3)
    begin
        H[R1] <= result_reg_mac;
    end
    else if (Oreg3)
    begin
        Oset[R1] <= result_reg_mac;
    end
    else if (Qreg3)
    begin
        Qset[R1] <= result_reg_mac;
    end
end

else if (CMD_logic_shift_right) // do shift right
begin
    if (Hreg1)
    begin
        H[R3] <= result_reg_Rshift;
    end
    else if (Oreg1)
    begin
        Oset[R3] <= result_reg_Rshift;
    end
    else if (Qreg1)
    begin
        Qset[R3] <= result_reg_Rshift;
    end
end

else if (CMD_logic_shift_left) // do shift left
begin
    if (Hreg1)
    begin
        H[R3] <= result_reg_Lshift;
    end
    else if (Oreg1)
    begin
        Oset[R3] <= result_reg_Lshift;
    end
end

```

```

        end
        else if (Qreg1)
        begin
            Qset[R3] <= result_reg_Lshift;
        end
    end

    else if (CMD_and) // do and
    begin
        if (Hreg2)
        begin
            H[R2] <= result_reg_and;
        end
        else if (Oreg2)
        begin
            Oset[R2] <= result_reg_and;
        end
        else if (Qreg2)
        begin
            Qset[R2] <= result_reg_and;
        end
    end

    else if (CMD_or) // do or
    begin
        if (Hreg2)
        begin
            H[R2] <= result_reg_or;
        end
        else if (Oreg2)
        begin
            Oset[R2] <= result_reg_or;
        end
        else if (Qreg2)
        begin
            Qset[R2] <= result_reg_or;
        end
    end

    else if (CMD_not) // do not
    begin
        if (Hreg1)
        begin
            H[R3] <= result_reg_not;
        end
        else if (Oreg1)
        begin
            Oset[R3] <= result_reg_not;
        end
        else if (Qreg1)
        begin
            Qset[R3] <= result_reg_not;
        end
    end

    else if (CMD_set) // do set

```

```

begin
    if (Hreg1)
    begin
        H[R0] <= result_reg_set;
    end
    else if (Oreg1)
    begin
        Oset[R0] <= result_reg_set;
    end
    else if (Qreg1)
    begin
        Qset[R0] <= result_reg_set;
    end
end
else if (CMD_load)
begin
    if (Hreg1)
    begin
        H[R0] <= data_in;
    end
    else if (Oreg1)
    begin
        Oset[R0] <= data_in;
    end
    else if (Qreg1)
    begin
        Qset[R0] <= data_in;
    end
end
end

end

endmodule

```

## Verilog code for SIMD Adder

```

module SIMDadd(
    input [15:0] A,
    input [15:0] B,
    input H,
    input O,
    input Q,
    input sub,
    output [15:0] Cout
);
    wire [15:0] B_real = sub?(~B):B;
    wire [4:0] C0 = A[3:0] + B_real[3:0] + sub;
    wire [4:0] C1 = A[7:4] + B_real[7:4] + (C0[4] & (O|H)) + (Q&sub);
    wire [4:0] C2 = A[11:8] + B_real[11:8] + (C1[4] & H) + ((Q|O) & sub);
    wire [4:0] C3 = A[15:12] + B_real[15:12] + (C2[4] & (O|H)) + (Q&sub);

    assign Cout = {C3[3:0], C2[3:0], C1[3:0], C0[3:0]};
endmodule

```



## Verilog code for SIMD shifter

```
module SIMDshifter(
    input [15:0] shiftinput,
    input H,
    input O,
    input Q,
    input left,
    output [15:0] shiftoutput
);

    wire [14:0] left_shift = shiftinput[14:0];
    wire [14:0] right_shift = shiftinput[15:1];
    wire [15:0] shiftoutput_tmp = left?{left_shift,1'b0}:{1'b0,right_shift};
    assign shiftoutput[3:0] = {(left|H|O)&shiftoutput_tmp[3],
shiftoutput_tmp[2:0]};
    assign shiftoutput[7:4] = {(left|H)&shiftoutput_tmp[7],
shiftoutput_tmp[6:5], (!left|H|O)&shiftoutput_tmp[4]};
    assign shiftoutput[11:8] = {(left|H|O)&shiftoutput_tmp[11],
shiftoutput_tmp[10:9], (!left|H)&shiftoutput_tmp[8]};
    assign shiftoutput[15:12] = {(left|H)&shiftoutput_tmp[15],
shiftoutput_tmp[14:13], (!left|H|O)&shiftoutput_tmp[12]};

endmodule
```

## Verilog code for SIMD multiplier

```
module SIMDMultiply(
    input [15:0] mulinputa,
    input [15:0] mulinputb,
    input H,
    input O,
    input Q,
    output [15:0] muloutput
);

    wire [15:0] sel0 = H?16'hFFFF:(O?16'h00FF:16'h000F);
    wire [15:0] sel1 = H?16'hFFFF:(O?16'h00FF:16'h00F0);
    wire [15:0] sel2 = H?16'hFFFF:(O?16'hFF00:16'h0F00);
    wire [15:0] sel3 = H?16'hFFFF:(O?16'hFF00:16'hF000);

    wire [15:0] a0 = (mulinputb[0]?mulinputa:16'h0000)&sel0;
    wire [15:0] a1 = (mulinputb[1]?mulinputa:16'h0000)&sel0;
    wire [15:0] a2 = (mulinputb[2]?mulinputa:16'h0000)&sel0;
    wire [15:0] a3 = (mulinputb[3]?mulinputa:16'h0000)&sel0;
    wire [15:0] a4 = (mulinputb[4]?mulinputa:16'h0000)&sel1;
    wire [15:0] a5 = (mulinputb[5]?mulinputa:16'h0000)&sel1;
    wire [15:0] a6 = (mulinputb[6]?mulinputa:16'h0000)&sel1;
    wire [15:0] a7 = (mulinputb[7]?mulinputa:16'h0000)&sel1;
    wire [15:0] a8 = (mulinputb[8]?mulinputa:16'h0000)&sel2;
```

```

wire [15:0] a9 = (mulinputb[9]?mulinputa:16'h0000)&sel2;
wire [15:0] a10 = (mulinputb[10]?mulinputa:16'h0000)&sel2;
wire [15:0] a11 = (mulinputb[11]?mulinputa:16'h0000)&sel2;
wire [15:0] a12 = (mulinputb[12]?mulinputa:16'h0000)&sel3;
wire [15:0] a13 = (mulinputb[13]?mulinputa:16'h0000)&sel3;
wire [15:0] a14 = (mulinputb[14]?mulinputa:16'h0000)&sel3;
wire [15:0] a15 = (mulinputb[15]?mulinputa:16'h0000)&sel3;

wire [15:0] tmp0,tmp1,tmp2,tmp3;
wire [15:0] tmp00,tmp11;
wire [15:0] tmp000;

assign tmp0  = a0    + (a1<<1)    + (a2<<2)    + (a3<<3);
assign tmp1  = a4    + (a5<<1)    + (a6<<2)    + (a7<<3);
assign tmp2  = a8    + (a9<<1)    + (a10<<2)   + (a11<<3);
assign tmp3  = a12   + (a13<<1)   + (a14<<2)   + (a15<<3);

assign tmp00 = tmp0 + (tmp1<<4);
assign tmp11 = tmp2 + (tmp3<<4);

assign tmp000 = tmp00 + (tmp11<<8);

wire [3:0] tmp1h,tmp1o,tmp1q;
wire [3:0] tmp2h,tmp2o,tmp2q;
wire [3:0] tmp3h,tmp3o,tmp3q;

assign muloutput[3:0] = tmp0[3:0];

assign tmp1h = tmp000[7:4];
assign tmp2h = tmp000[11:8];
assign tmp3h = tmp000[15:12];

assign tmp1o = tmp00[7:4];
assign tmp2o = tmp11[11:8];
assign tmp3o = tmp11[15:12];

assign tmp1q = tmp1[7:4];
assign tmp2q = tmp2[11:8];
assign tmp3q = tmp3[15:12];

assign muloutput[7:4]  = H?tmp1h:(O?tmp1o:tmp1q);
assign muloutput[11:8] = H?tmp2h:(O?tmp2o:tmp2q);
assign muloutput[15:12] = H?tmp3h:(O?tmp3o:tmp3q);

```

endmodule

\*\*\*