

Arithmetic Unit with MMX x86 Instructions

Sîrca Florentina Raluca
Technical University of Cluj-Napoca

Contents

Contents	2
Introduction.....	2
Project proposal	2
Project Plan	2
Bibliographic study.....	3
Analysis	5
Design	7
Implementation	10
Testing and validation	12
Conclusions.....	17
Bibliography	18

Introduction

Project proposal

The goal of this project is to design, implement and test 6 operations on the Arithmetic Unit with MMX x86 Instructions on 64 bits.

Hardware Design: The project will involve the design of the following hardware components:

- A custom arithmetic unit capable of handling MMX x86 instructions.
- Integration with an FPGA platform for testing.

Software Development:

- Developing testbenches for simulation and validation.
- Developing software code snippets to generate MMX instruction streams for testing.

Project Plan

- Week 2: choose the project.
- Week 4: informing myself and looking for information about the project, planning the work for next weeks.
- Week 6: Define the architecture and specifications for the arithmetic unit.
Design the control and data paths.

- Week 8: Write VHDL code for the Data Processing Unit. Develop software to drive the unit and validate functionality. Implement testbenches for simulation.
- Week 10: Conduct functional testing and performance optimization. Refine the design based on testing results.
- Week 12: Create project documentation, including a detailed report and user manual.
- Week 14: presentation of the project.

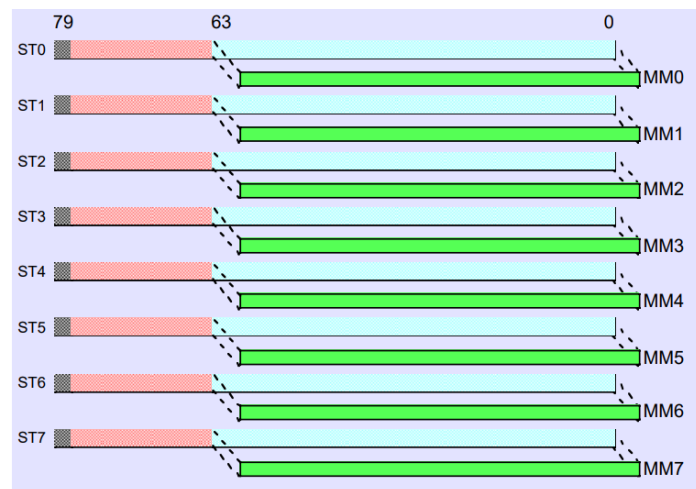
Bibliographic study

Intel was developing an instruction set architecture extension for multimedia applications. It developed 57 instructions that would greatly accelerate the execution of multimedia applications.

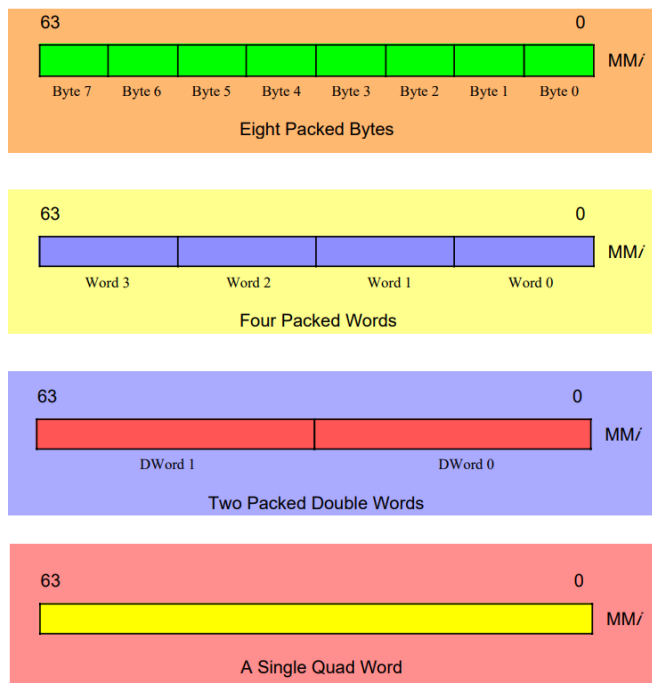
Since the instruction set has been available for quite some time, you can probably use the MMX instructions without worrying about your software failing on many machines.

MMX defines eight processor registers, named MM0 through MM7, and operations that operate on them. Each register is 64 bits wide and can be used to hold either 64-bit integers, or multiple smaller integers in a "packed" format: one instruction can then be applied to two 32-bit integers, four 16-bit integers, or eight 8-bit integers at once. MMX provides only integer operations. MMX instructions are part of SIMD (Single Instruction, Multiple Data) processing.

These are strictly data registers, you cannot use them to hold addresses nor are they suitable for calculations involving addresses. Unlike the x87 registers, which behave like a stack, the MMX registers are each directly addressable (random access). The MMX registers overlay the FPU registers in much the same way that the 16-bit general purpose registers overlay the 32-bit general purpose registers.



The MMX instruction set facilitates the manipulation of four distinct data types: an eight-byte array, a four-word array, a two-element double-word array, and a quadword object. Each MMX register is specifically designed to handle one of these four data types.



Most MMX instructions operate on two operands, a source and a destination operand. A few instructions have three operands with the third operand being a small immediate (constant) value. In this section we'll look at the common MMX instruction operands. I will not use instructions which have three operands in this project.

The MMX instruction set accommodates various data types, which can be either an MMX register or a memory location. Memory locations typically represent quad-word entities, though certain instructions may operate on double-word objects. It's important to note that "quad-word" and "double-word" refer to sequences of eight or four consecutive bytes in memory, not necessarily the data type being processed by the MMX instruction.

MMX instructions generally follow this syntax in High-Level Assembly (HLA):

- `mmxInstr(source, dest);`

Specific forms include:

- `mmxInstr(mmi, mmi); // i=0..7`
- `mmxInstr(mem, mmi); // i=0..7`

Certain instructions may need a small immediate value or constant. Such instructions usually take the following form:

- `mmxInstr(imm8, source, dest);`

Generally, MMX instructions don't allow immediate constants as operands, except for specific cases like shift counts. The source operand in an MMX instruction should be a register or a quad-word variable, not a 64-bit constant. To emulate specifying a constant as the source operand, you must initialize a quad-word variable in the READONLY or STATIC section of your program and specify this variable as the source operand. However, HLA does not support 64-bit constants, so initializing the value can be a challenge. Two approaches can resolve this: breaking the constant into smaller pieces and emitting those pieces in a way HLA can process, or creating custom numeric conversion routines using HLA's compile-time language to emit a 64-bit constant.

The first approach, breaking constants into smaller pieces, is more commonly used. Most MMX instructions work on arrays of bytes, words, or double words rather than 64-bit data operands. Since HLA supports expressions for byte, word, and double word constants, specifying a 64-bit MMX memory operand as a short array of objects is a practical solution. To accommodate MMX instructions expecting a 64-bit operand fetched from memory, declare such objects as qword variables. The challenge with this approach is that the qword type does not allow initializers because HLA cannot handle 64-bit constant expressions. When declared in the STATIC segment, HLA initializes qword variables with zero, which may not be the desired initial value.

Analysis

Instruction Examples:

The data type shown in this section will be the 16-bit word data type, most of these operations also exist for 8-bit or 32-bit packed data types.

The following example shows a packed add word with wrap-around. It performs four additions of the eight, 16-bit elements, with each addition independent of the others and in parallel. `FFFFh + 8000h` would be a 17 bit result. The 17th bit is lost because of wrap-around, so the result is `7FFFh`.

a3	a2	a1	FFFFh
+	+	+	+
b3	b2	b1	8000h
<hr/>			
a3+b3	a2+b2	a1+b1	7FFFh

PADD[W]: Wrap-around Add

The following example is for a packed add word with unsigned saturation. This example uses the same data values from before. The right-most add generates a result that does not fit into 16 bits and in this case saturation occurs. Saturation means that if addition results in overflow, the result is clamped to the largest or the smallest value representable.

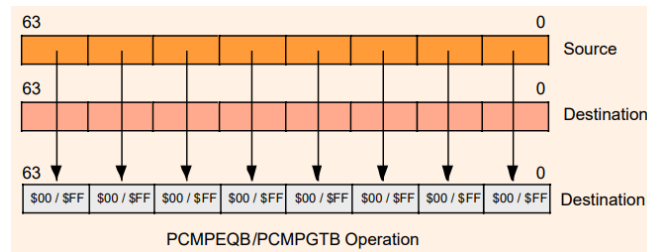
a3	a2	a1	FFFFh
+	+	+	+
b3	b2	b1	8000h
a3+b3	a2+b2	a1+b1	FFFFh

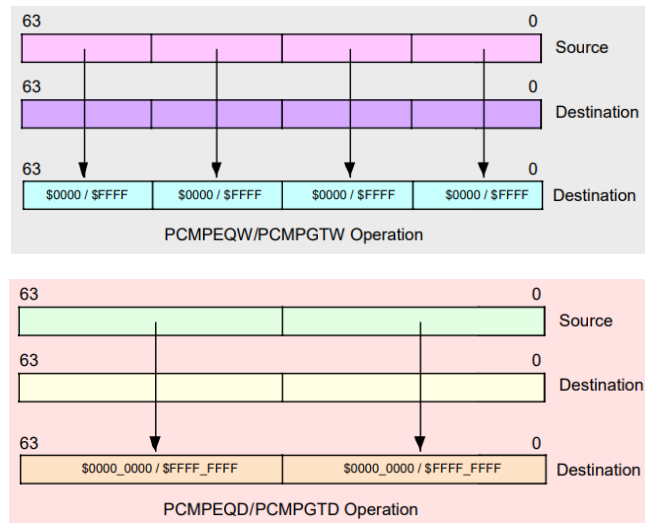
PADDUS[W]: Saturating Arithmetic

For a packed add word with signed saturation, the result is clamped to the largest or the smallest value representable. Overflow occurs when the result of an arithmetic operation exceeds the maximum positive or minimum negative value that can be represented in the given number of bits. Overflow occurs when the result is greater than +127 or less than -128.

When overflow occurs, signed saturation clamps the result to the maximum positive or minimum negative value that can be represented. If the result goes beyond these bounds, it's set to the nearest limit.

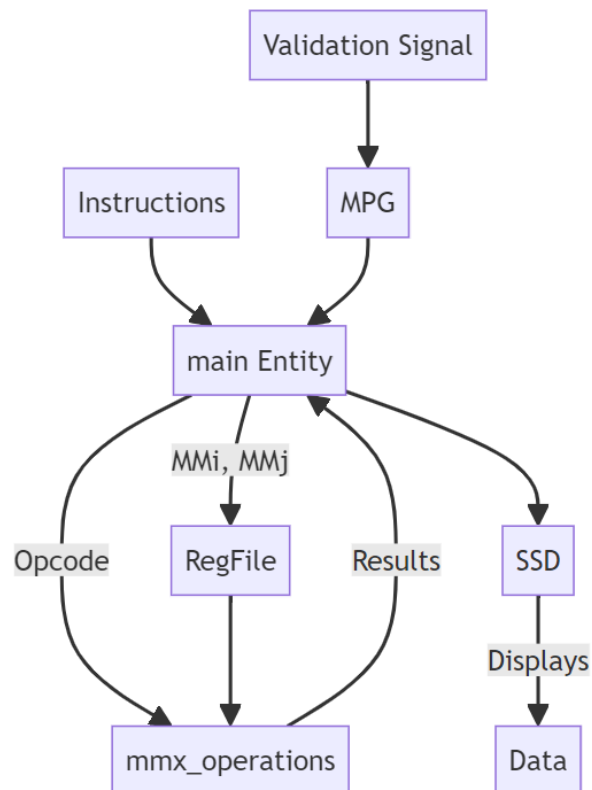
The following examples are for a Packed compare for equalities on Bytes, on Words and on DoubleWords. The packed comparison instructions compare the destination (second) operand to the source (first) operand to test for equality or greater than. These instructions compare eight pairs of bytes (PCMPEQB, PCMPGTB), four pairs of words (PCMPEQW, PCMPGTW), or two pairs of double words (PCMPEQD, PCMPGTD). These instructions return their true or false values in the destination operand.





Design

This simplified block diagram represents the structure of a VHDL code for a Arithmetic Unit with MMX.



Components

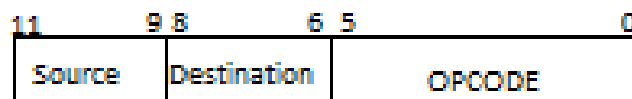
- **MPG:** generates a monostable pulse based on a clock signal (clk) and a reset signal (rst). The process within the architecture ensures that when a rising edge is detected on the clock signal, the pulse is activated and held high for the defined duration, after which it resets to a low state. This monostable pulse generator can produce a controlled pulse output for the input signal “ok”.
- **Register file:** Contains the 8 64-bit registers MM0-MM7. The register file will read the source and destination, these are two 3-bit data inputs representing the register number, then will produce two 64-bit data outputs, the two numbers that will be used in ALU.
- **MMX Data Processing Unit:** Reads two 64-bit data inputs and 1 instruction input from MMX Control, executes the appropriate instruction, and outputs the result. The result will be shown on the 7 Segment-Display and will be rewritten in the memory.

Signals:

- **Inputs:** 12-bit instruction: 6-bit for Opcode, 3-bit for source and 3-bit for destination; the signal ‘ok’ serves as an input that is activated or pressed after the insertion of the instruction; the ‘count’ signal that changes the 16 bits of the result to be displayed on the SSD.
- **Outputs:** The output is presented on the Seven-Segment Display (SSD). In this particular implementation, the SSD is partitioned into four distinct segments, each responsible for displaying a specific portion of a 64-bit number. This division allows the efficient representation of the entire 64-bit numerical value by distributing it across these four segments, with each segment contributing to the display of a 16-bit subset of the overall number. By organizing the information in this manner, the SSD can visually convey the complete 64-bit value by combining the outputs from the individual segments, providing a clear and concise representation of the numerical data to the observer; other output is instrLed which represents the operation that is done by the unit; the reset button will reset all registers to 0.

MMX Instruction Set

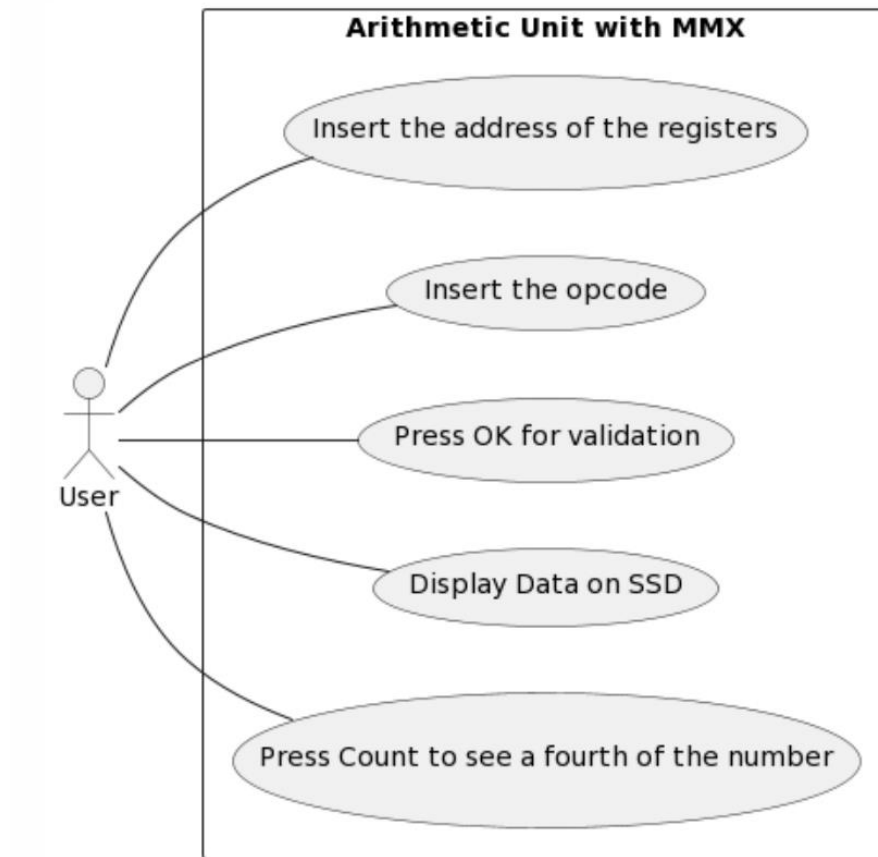
Instruction Format:



The project incorporates a set of 15 instructions derived from Intel's MMX extensions. These instructions were specifically designed for this project, featuring unique opcodes that set them apart from standard instructions.

Syntax	Description	Opcode
PADDB src, dest	Add with wrap-around on Bytes in registers src and dest	010000
PADDW src, dest	Add with wrap-around on Words in registers src and dest	010001
PADDD src, dest	Add with wrap-around on DoubleWords in registers src and dest	010010
PADDSB src, dest	Add signed with saturation on Bytes in registers src and dest	010101
PADDSW src, dest	Add signed with saturation on Words in registers src and dest	010101
PADDUSB src, dest	Add unsigned with saturation on Bytes in registers src and dest	011010
PADDUSW src, dest	Add unsigned with saturation on Words in registers src and dest	011011
PCMPEQB nr1, nr2	Packed compare for equalities on Bytes	110000
PCMPEQW nr1, nr2	Packed compare for equalities on Words	110001
PCMPEQD nr1, nr2	Packed compare for equalities on DoubleWords	110010
PAND src, dest	Bitwise AND	000000
PANDN src, dest	Bitwise AND NOT	000001
POR src, dest	Bitwise OR	000010
PXOR src, dest	Bitwise XOR	000011

Implementation



Main Entity

Entity Name: main

Description: This is the top-level entity of the design. It interfaces with external components through the following ports:

clock: Input clock signal.

reset: System reset signal.

ok: Input signal to trigger operations.

instr: 12-bit instruction input.

alu_output: 64-bit output from the ALU.

Two main processes: one that takes care of rewriting the destination registers at the next clock signal; and one that helps to display the entire number using a 2-bit counter, it also resets itself if the number to be displayed is changed to write the number from left to right

Components

MMX Operations Component

Name: mmx_operations

Description: Responsible for executing MMX instructions. It takes two 64-bit numbers and an opcode as input and outputs a 64-bit result along with a finish signal.

Ports:

no1, no2: Input operands.

opcode: Instruction opcode.

nr_out: Output result.

finish: Signal indicating operation completion.

Register File Component

Name: RegFile

Description: Manages the register operations including read and write functionalities.

Ports:

clk, rst: Clock and reset signals.

write_en, read_en: Enable signals for write and read operations.

write_addr, read_addr1, read_addr2: Address signals for register access.

data_in: Input data for write operation.

data_out1, data_out2: Output data from read operations.

Monostable Pulse Generator (MPG) Component

Name: MPG

Description: Generates a monostable pulse based on input and clock signals.

Ports:

en: Enable signal for the pulse.

input: Input trigger.

clock: Clock signal.

SSD Component

Entity Name: SSD

Description: This entity represents a Seven-Segment Display (SSD) driver. It is designed to display hexadecimal digits on a 4-digit 7-segment display.

Ports:

clk: Input clock signal.

digits: 16-bit input representing four hexadecimal digits.

an: Output control signal for the anode connections of the 7-segment display.

cat: Output control signal for the cathode segments of the 7-segment display.

Signals: Various signals are used to interconnect components and manage data flow and control within the design.

Testing and validation

- **PADDB:** Add with wrap-around on Bytes

The test:

reg1+	REG1[63:0]	ffffffffffffffff
reg2 =	REG2[63:0]	2222222222222222
result.	output[63:0]	2121212121212121

Explication: each reg has 8 bytes: It performs 8 additions of the 8-bit elements(byte). The 9th bit is lost because of wrap-around, so the result is 21h.

1111 1111 (FF h) +

0010 0010 (22 h) =

(1-carry out) 0010 0001 (21 h)

- **PADDW:** Add with wrap-around on Words

The test:

reg1+	REG1[63:0]	fffffffffffffff
reg2 =	REG2[63:0]	2222222222222222
result.	output[63:0]	2221222122212221

Explication: each reg has 4 words: It performs 4 additions of the 16-bit elements(word). The 17th bit is lost because of wrap-around, so the result is 2221h.

1111 1111 1111 1111 (FFFF h) +
 0010 0010 0010 0010 (2222 h) =
 (1-carry out) 0010 0010 0010 0001 (2221 h)

- **PADDD:** Add with wrap-around on Double Words

The test:

reg1+	> REG1[63:0]	fffffffffffffff
reg2 =	> REG2[63:0]	2222222222222222
result.	> output[63:0]	2222222122222221

Explication: each reg has 2 double-words: It performs 2 additions of the 32-bit elements(double word). The 33rd bit is lost because of wrap-around, so the result is 22222221h.




1111 1111 1111 1111 1111 1111 1111 1111 (FFFFFFFF h) +
 0010 0010 0010 0010 0010 0010 0010 0010 (22222222 h) =
 (1-carry out) 0010 0010 0010 0010 0010 0010 0010 0001 (22222221 h)

- **PADDSB**

The test:

reg1+	> REG1[63:0]	d6d6d6d6d6d6d6d6	1101 0110 (D6 h) +
reg2 =	> REG2[63:0]	0404040404040404	0000 0100 (04 h) =
result.	> output[63:0]	dadadadadadadada	1101 1010 (DA h)




Explication: each reg has 8 bytes: It performs 8 additions of the 8-bit elements(byte). The 8th bit is the sign, so the result is DA h.

>  REG1[63:0]	7c7c7c7c7c7c7c7c
>  REG2[63:0]	0404040404040404
>  output[63:0]	7f7f7f7f7f7f7f7f




Here, I added two positive numbers that would normally be 80 h, but because the 8th bit represents the sign, the maximum value of the positive number is 7F h.

- PADDSW

The test:

reg1+	>  REG1[63:0]	ffc9ffe9ffc9ffe9	1111 1111 1100 1001 (FFC9 h) +
reg2 =	>  REG2[63:0]	0016001600160016	0000 0000 0001 0110 (0016 h) =
result.	>  output[63:0]	ffdfffffffdfffff	1111 1111 1101 1111 (FFDF h)







Explication: each reg has 4 words: It performs 4 additions of the 16-bit elements(word). The 16th bit is the sign, so the result is FFDFh.

>  REG1[63:0]	cad3cad3cad3cad3
>  REG2[63:0]	b6ebb6ebb6ebb6eb
>  output[63:0]	8000800080008000

Here, I add 2 negative numbers that would normally be on 17 bits, but because the 16th bit represents the sign, the minimum value of the negative number is 8000 h.

- PADDUSB: Add unsigned with saturation on Bytes

The test:

reg1+	>  REG1[63:0]	fffffffffffffff	>  REG1[63:0]	5555555555555555
reg2 =	>  REG2[63:0]	4444444444444444	>  REG2[63:0]	4444444444444444
result.	>  output[63:0]	fffffffffffffff	>  output[63:0]	9999999999999999

Explication: each reg is 8 bytes: performs 8 additions of the 8-bit elements (bytes). If the 9th bit is 1, that means the result is greater than FF, so the final result is FF. Otherwise, the result is the sum of byte1 and byte2.

1111 1111 (FF h) +

0100 0100 (44 h) =

(1-carry out) 1111 1111 (FF h)

- PADDUSW

The test:

reg1+	> REG1[63:0]	ffffffffffffffff	> REG1[63:0]	5555555555555555
reg2 =	> REG2[63:0]	4444444444444444	> REG2[63:0]	4444444444444444
result.	> output[63:0]	ffffffffffffffff	> output[63:0]	9999999999999999

Explication: each reg is 4 words: performs 4 additions of the 16-bit elements (words). If the 17th bit is 1, that means the result is greater than FFFF, so the final result is FFFF. Otherwise, the result is the sum of byte1 and byte2.

1111 1111 1111 1111 (FFFF h) +
0100 0100 0100 0100 (4444 h) =
(1-carry out) 1111 1111 1111 1111 (FFFF h)

- PCMPEQB

The test:

reg1 comp	> REG1[63:0]	ff0ffffff0f0000
reg2 =	> REG2[63:0]	fffffffffffffff
result	> output[63:0]	ff0ffffff000000

Explication: I compared byte by byte. This byte comparison allows you to determine whether two 64-bit numbers are equal or not by looking at their individual byte values. If the bytes are equal, then the resulting byte is FFh, otherwise it is 00h.

- PCMPEQW

The test:

reg1 comp	> REG1[63:0]	fff0ffffff0f0000
reg2 =	> REG2[63:0]	fffffffffffffff
result	> output[63:0]	0000ffff00000000

Explication: I compared word by word. This word comparison allows you to determine whether two 64-bit numbers are equal or not by looking at their individual word values. If the words are equal, then the resulting word is FFFFh, otherwise it is 0000h.

- PCMPEQD

The test:

```

reg1 comp
reg2 =
result

```

> REG1[63:0]	ffffffff0f0000
> REG2[63:0]	ffffffffffffff
> output[63:0]	ffffffff00000000

Explication: I compared double-word by double-word. This double-word comparison allows you to determine whether two 64-bit numbers are equal or not by looking at their individual double-word values. If the double-words are equal, then the resulting double-word is FFFFFFFFh, otherwise it is 00000000h.

- PAND

The test:

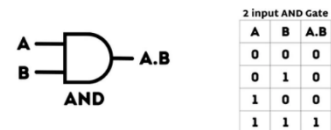
```

reg1 and
reg2 =
result.

```

> REG1[63:0]	ffffffffffffff	ffffffffffffff...	0000000000000000
> REG2[63:0]	0000000000000000	ffffffffffffff...	0000000000000000
> output[63:0]	ffffffffffffff	ffffffffffffff...	0000000000000000

Explication: each reg is divided into 64 bits and I used AND gate for each bit.



- PANDN

The test:

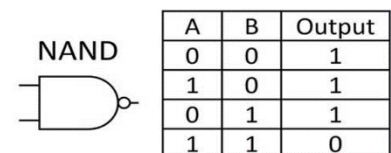
```

reg1 nand
reg2 =
result.

```

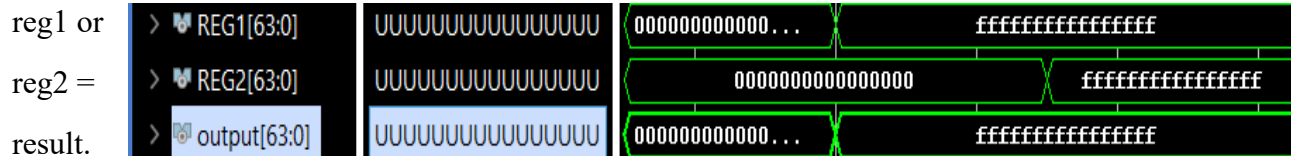
> REG1[63:0]	ffffffffffffff	ffffffffffffff...	0000000000000000
> REG2[63:0]	0000000000000000	ffffffffffffff...	000000000000...
> output[63:0]	ffffffffffffff	000000000000...	ffffffffffffff...

Explication: each reg is divided into 64 bits and I used NAND gate for each bit.



- POR

The test:



Explication: each reg is divided into 64 bits and I used OR gate for each bit.



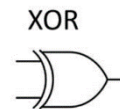
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

- PXOR

The test:



Explication: each reg is divided into 64 bits and I used XOR gate for each bit.



A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

Conclusions

The project successfully designed, implemented, and tested an Arithmetic Unit capable of executing MMX x86 Instructions on a 64-bit architecture. The hardware and software development phases focused on creating a custom arithmetic unit integrated with an FPGA platform, along with the development of testbenches and software for simulation, validation, and generating MMX instruction streams.

The project followed a structured plan, starting from an initial concept and progressing through stages of architectural definition, VHDL coding, and functional testing. The design phase involved the creation of several key components, including the MMX Data Processing Unit, Register File, MPG (Monostable Pulse Generator), and SSD (Seven-Segment Display) driver. These components were crucial in handling the MMX instruction set, which included operations like addition with wrap-around and saturation, packed comparison for equalities, and logical operations like AND, OR, and XOR.

The MMX instructions, carefully chosen and adapted for this project, provided efficient ways to process packed data types such as bytes, words and double-words. The arithmetic unit

was able to perform operations on these data types in parallel, showcasing the power of SIMD (Single Instruction, Multiple Data) processing inherent in MMX technology.

The project met its objectives by successfully creating an Arithmetic Unit that efficiently executes MMX x86 instructions. The unit's capability to handle a variety of arithmetic and logical operations on packed data types, along with its effective output display system, demonstrates its potential for practical applications in multimedia and other computation-intensive fields.

Bibliography

- https://www.csie.ntu.edu.tw/~cyy/courses/assembly/docs/ch11_MMX.pdf
- <https://www.intel.com/content/dam/develop/external/us/en/documents/mmx-manual-tech-overview-140701.pdf>
- <https://www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/TheMMXInstructionSet.html>