Design of an MMX arithmetic unit

Student: Tudor Bartha

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

Introduction

1.1 Context

The aim of this project is to develop an arithmetic unit that implements the MMX (MultiMedia eXtension) instruction set. The architecture of the MMX Processor enables SIMD (Single Instruction Multiple Data) processing, allowing for efficient parallel arithmetic operations on packed data. Operations such as packed integer multiplication, addition and shifting will be showcased in this project.

By creating this ALU I will explain and exemplify how the MMX revolutionized data processing and improved performance in applications like image processing, audio manipulation and other multimedia tasks where rapid parallel computations are essential. The resulting arithmetic unit will provide a robust MMX ALU that can be used for fast integer computations on packed data.

1.2 Objectives

The arithmetic unit will be designed using VHDL based on my previous experience with the CA (Computer Architecture) course, during which I implemented the ALU for a MIPS. A testbench will be provided for testing the unit. I will make use of a debouncer and a memory where 8/16/32/64-bit numbers will be stored as input for the registers of the MMX. Implementation of the adders and multipliers will be discussed during the analysis part of the documentation. The results of the operations performed will be shown on the SSD of a Basys3 FPGA board.

The arithmetic unit must provide at least 6 correctly implemented operations that showcase the great improvement that the MMX instruction set brought due to the SIMD principle. Through simulation and FPGA implementation, the project will showcase how these arithmetic operations can improve performance in real-time multimedia applications.

List of possible operations:

Figure 1.1 MMX Instructions [6]

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
paddb	PADDB	add packed byte integers	
paddd	PADDD	add packed doubleword integers	
paddsb	PADDSB	add packed signed byte integers with signed saturation	
paddsw	PADDSW	add packed signed word integers with signed saturation	
paddusb	PADDUSB	add packed unsigned byte integers with unsigned saturation	
paddusw	PADDUSW	add packed unsigned word integers with unsigned saturation	
paddw	PADDW	add packed word integers	
pmaddwd	PMADDWD	multiply and add packed word integers	
pmulhw	PMULHW	multiply packed signed word integers and store high result	
pmullw	PMULLW	multiply packed signed word integers and store low result	
psubb	PSUBB	subtract packed byte integers	
psubd	PSUBD	subtract packed doubleword integers	
psubsb	PSUBSB	subtract packed signed byte integers with signed saturation	
psubsw	PSUBSW	subtract packed signed word integers with signed saturation	
psubusb	PSUBUSB	subtract packed unsigned byte integers with unsigned saturation	
psubusw	PSUBUSW	subtract packed unsigned word integers with unsigned saturation	
psubw	PSUBW	subtract packed word integers	

Figure 1.2 MMX Instructions [6]

Logical Instructions (MMX)

The logical instructions perform logical operations on quadword operands.

Table 3-24 Logical Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
pand	PAND	bitwise logical AND	
pandn	PANDN	bitwise logical AND NOT	
por	POR	bitwise logical OR	
pxor	PXOR	bitwise logical XOR	

Shift and Rotate Instructions (MMX)

The shift and rotate instructions operate on packed bytes, words, doublewords, or quadwords in 64–bit operands.

Table 3-25 Shift and Rotate Instructions (MMX)

Solaris Mnemonic	Intel/AMD Mnemonic	Description	Notes
pslld	PSLLD	shift packed doublewords left logical	
psllq	PSLLQ	shift packed quadword left logical	
psllw	PSLLW	shift packed words left logical	
psrad	PSRAD	shift packed doublewords right arithmetic	
psraw	PSRAW	shift packed words right arithmetic	
psrld	PSRLD	shift packed doublewords right logical	
psrlq	PSRLQ	shift packed quadword right logical	
psrlw	PSRLW	shift packed words right logical	

After a brief examination of the operations, I consider implementing the following instructions: (I will analyze them more in the analysis part of the project)

PMADDWD - multiply and add packed word integers

PMULHW - multiply packed signed word integers and store high result

PMULLW - multiply packed signed word integers and store low result

PADDB - add packed byte integers

PADDSB - add packed signed byte integers with signed saturation

PSUBSB - subtract packed signed byte integers with signed saturation

PSUBUSB - subtract packed unsigned byte integers with unsigned saturation

PSUBSW - subtract packed signed word integers with signed saturation

PCMPGTW - compare packed signed word integers for greater than

PCMPEQD - compare packed doublewords for equal

PAND - bitwise logical AND

PANDN - bitwise logical AND NOT

POR - bitwise logical OR

PXOR - bitwise logical XOR

PSLLD - shift packed doublewords left logical

PSRAW - shift packed words right arithmetic

To make things as easy as possible for the user of this VHDL application, I consider storing the numbers to be as input for the registers in a ROM memory and storing the instructions to be performed in a ROM memory too or inputting them manually from the FPGA's switches. From a button we signal the execution of an instruction.

1.3 Weekly planning

Week1: choose a project

Week3: inform myself about the MMX technology, make a short introduction, plan some goals for this project, discuss about the instructions

Week5: present the architecture of the MMX arithmetic unit, clarify input methods, show structural implementation, discuss improvements of the architecture

Week7: write VHDL code, create testbenches for simulation, discuss problems that might arise during the VHDL implementation

Week9: refine implementation, solve problems that appeared during week7, test on the FPGA board

Week11: Make slight adjustments to the documentation, if necessary, solve any problems that might have appeared during testing on the FPGA on Week9

Week13: Presentation of the project

Bibliographic research

2.1 WHY MMX?

I had previously heard about MMX and learnt a few basic principles during the ALP course. Because of this, learning more about MMX raised my curiosity and made me want to dive deeper into what are the main reasons this technology emerged. MMX instruction set is an extension to the existing Intel Architecture present on the Pentium processor. During the 1990s, multimedia applications became more and more popular, and developers needed more computational power. Intel came up with the idea of increasing the computational power of their processors. They noticed that most of the operations performed by multimedia applications were integer operations on small sizes: 8 bits or 16 bits. Matrix and vector multiplication and addition were among the most common. 57 new instructions were created. The big challenge was not to make any drastic changes to hardware, so they decided to use the existing FP registers. These registers were on 80 bits and only the lower 64 were actively used by MMX instructions. The remaining bits were all set on 1 to indicate that MMX operations are taking place. This raised another problem: it was not possible to use both mmx instructions and FP operations at the same time. However, Pentium2 processor solved this issue. As I said earlier, the need for processing a large number of integer operations at the same time gave way for the SIMD paradigm. In this way data could be packed into registers and more computations are made at the same time.

2.2 MMX REGISTERS

In this section I will present the registers in more detail. There are 8 MMX data registers: MM0-MM7. As explained earlier, they are actually using the FP registers. They can store the following data types:

- **Packed byte**: this consists of eight 8-bit bytes packed into a single 64-bit register, they are commonly used to compute color intensity of pixels
- **Packed word**: this consists of four 16-bit words packed into a single 64-bit register, they are commonly used in audio processing
- **Packed doubleword**: this consists of two 32-bit doublewords packed into a single 64-bit register, they are commonly used in image processing

- **Quadword**: this consists of a 64-bit value that takes up a whole register, they are commonly used in large data processing

I will also provide a visual representation of the registers and the data types.

Figure 2.1 MMX registers [14]

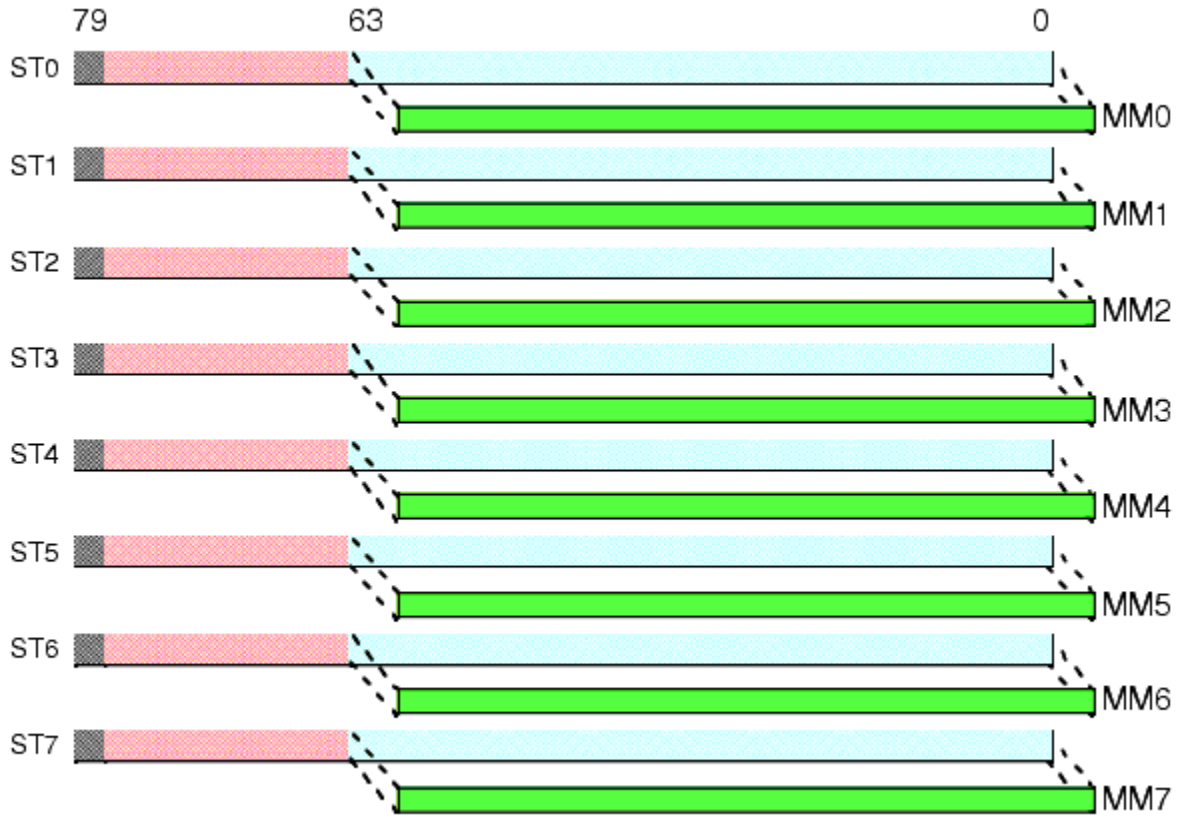
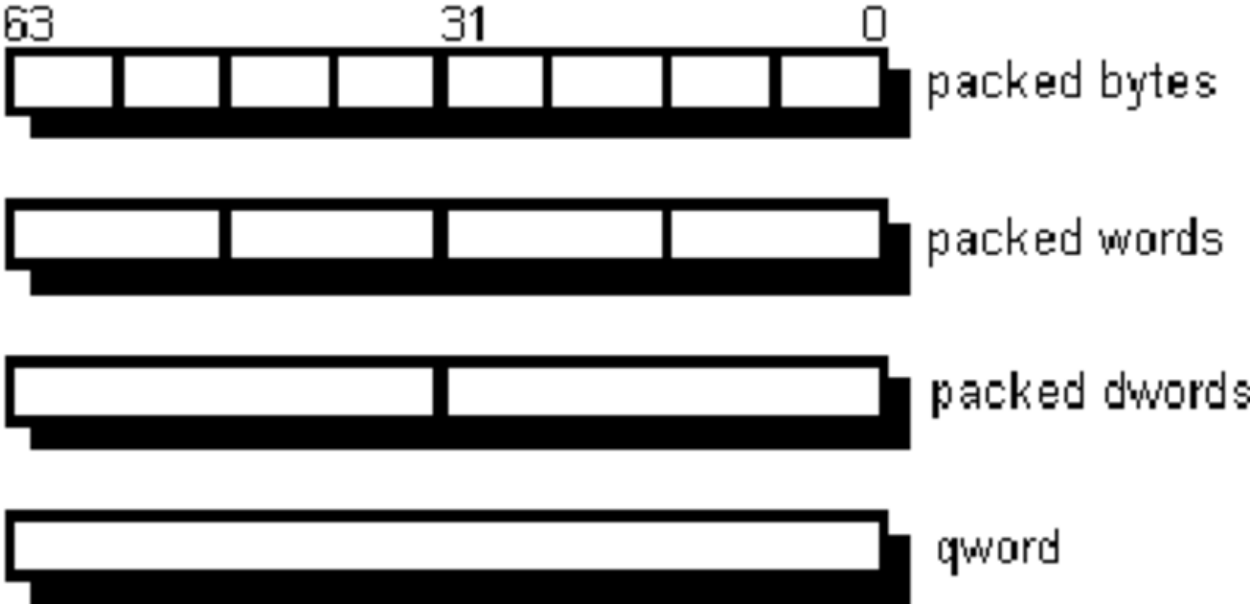


Figure 2.2 Data types [15]



2.3 ARITHMETIC UNIT FEATURES

The arithmetic unit works based on the types and paradigms mentioned above. SIMD enables the unit to efficiently process multiple data in parallel. A unique feature, not mentioned until now, is the signed and unsigned saturating arithmetic used for addition and subtraction. This means that we can't have overflow, when adding for example 8-bit unsigned integers, the range varies from 0-255. If we add 100+200, the result would normally result in overflow, however, due to saturating arithmetic, the result is 255. This is particularly useful when computing color intensity, as multiple filters will overlap and create a very dark or very white color. Wrap-around logic may also be used in the case some applications desire it. In this case, the value of a computation wraps around to the opposite limit if they exceed the representable range. Some operations can also be performed with signed integers. In this case, 2's complement representation is used, where the MSB is the sign bit (0 for positive and 1 for negative).

2.4 MMX INSTRUCTIONS

The instructions introduced by MMX work mainly with the data registers but may also use memory addresses or an immediate. Most operations have 2 operands, however, there are some working with three, such as the shift where you can specify shift amount and the destination and source register. When using memory addressing, MMX instructions directly access data from memory, which allows the processor to perform operations on larger data sets without needing to load everything into registers first.

2.5 MMX APPLICATIONS

MMX technology played a crucial role in the improvement of Dolby Digital audio decoding by providing faster computations. The collaboration between the two entities aimed at making the decoding process feasible on general-purpose processors without the need for additional hardware. Initially, the decoding was performed on 16 bits only and this led to quantization noise, reducing audio quality. However, the MMX instruction set allowed operating on 32-bit data and by doing so, eliminating the errors associated with intermediate result rounding. Moreover, MMX's integer-based operations proved to be more efficient than floating point operations and Dolby Digital decoding became a standard on all modern personal computers.

2.6 MMX ENHANCEMENTS

Some enhancements of the classic 64-bit MMX instruction set are the SSE, SSE2, SSE3 and SSSE3. SSE was introduced with Pentium 3 and extended MMX by adding 8 XMM registers, each of 128 bits. It focused on single-precision floating-point operations, allowing parallel processing of 4 32-bit floats. It improved floating-point operations by implementing the SIMD paradigm. SSE2 was launched with the Pentium4 processor and included double-precision floating-point operations and improved the integer operations. Added the 128-bit packed data types. Also, it enhanced existing operations on floating point data. SSE3 was introduced in Pentium4 with Hyper-Threading and added enhancements for multi-threaded applications. Improved data parallelism with by introducing for handling unaligned data loads and improved synchronization. SSSE3 was included in Intel Core 2 processors. Provided horizontal addition/subtraction operations, absolute value calculations, and byte-level shuffling, which improved efficiency for matrix computations. Added new instructions for accelerating cryptographic computations, specifically useful for encoding and decoding tasks.

Analysis

3.1 Project Proposal

The final MMX ALU will encompass the features described below:

- Addition and subtraction for all data types, signed, unsigned, wraparound, saturation, signed, unsigned
- Multiply and add packed word integers, multiply packed signed word integers and store high result
- Compare packed bytes for equal, compare packed doublewords for equal, compare packed words for equal,
- Bitwise logical and, bitwise logical and not, bitwise logical or, bitwise logical xor
- An instruction memory for showcasing the ALU operations

- A ROM memory from which we load the MMX registers
- A control unit
- A register for storing the result of an operation

3.2 Project Analysis

3.2.1 How it works

I need to design the whole project in a structural manner. In order to do this, I need to search for the best algorithms that are efficient from a resources point of view. Speed is another important factor when analyzing this project. I want to make this project easy to use, so no inputs for the registers should be introduced by the end-user. I thought of how the project should logically work and I made a state diagram of the project:

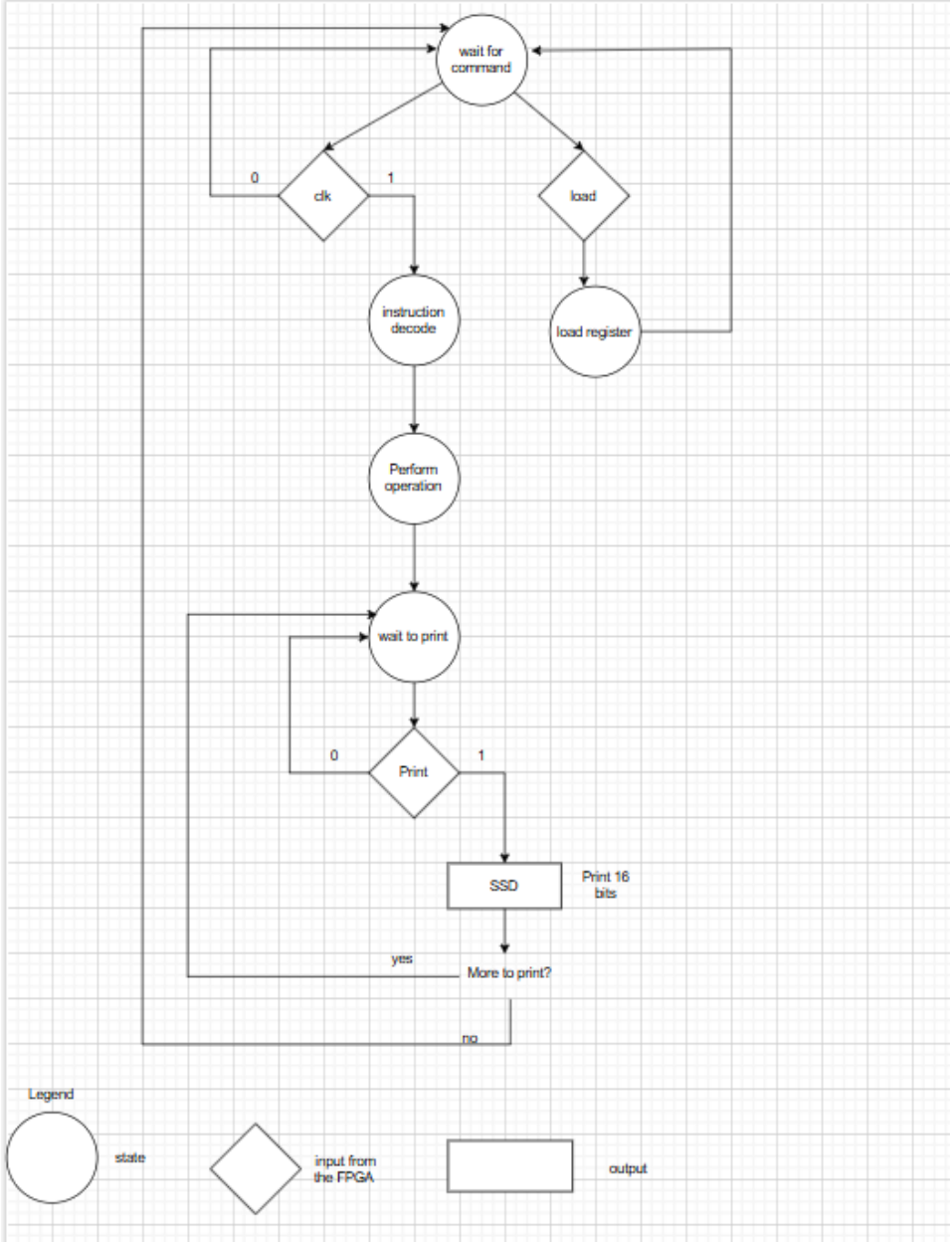


Figure 3.1 showing the state diagram of the top level of the project

3.2.2 Hardware components

1. **Instruction memory**: a ROM memory which will host the operations that are to be showcased by the project
2. **ROM memory for Registers**: Multiple sets of 8 64-bit values can be loaded into the MMX registers by pressing a button

3. **Control unit**: a unit which will decode the input from the instruction memory and send the necessary signals to the main alu
4. **Registers**: 8 64-bit registers
5. **Adders** and other combinational components used for the design of the actual ALU, I will go into further detail at the design step
6. **Result register**: A 64-bit register that stores the result of the operation for output on the SSD
7. MUX, DMUX

3.2.3 Performance optimization

1. **Parallel Processing**: By implementing SIMD (Single Instruction, Multiple Data) capabilities, the arithmetic unit will handle multiple data points simultaneously, making it highly efficient for multimedia tasks.
2. **Saturating Arithmetic**: Signed and unsigned saturation prevents overflow in arithmetic operations, particularly useful for multimedia tasks where values must stay within a specific range (e.g., pixel values in image processing).
3. **Efficient Memory Access**: The use of ROM for initializing registers and direct memory access for instructions minimizes time spent on loading data, allowing for smoother and faster operation.
4. **Optimized VHDL Implementation**: By writing optimized VHDL code, the unit will make efficient use of FPGA resources, which is essential for real-time applications.

3.2.4 Instruction format

I will use a format that encompasses the 2 registers I use and an opcode.

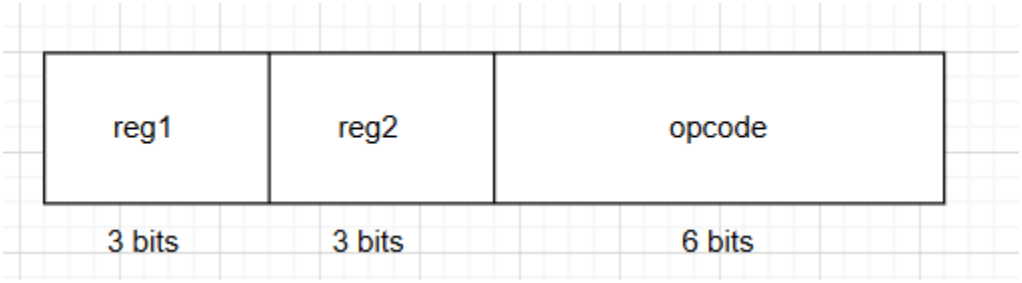


Figure 3.2 showing the encoding method of the instructions

I calculated that I need 6 bits to represent all the operations I will implement.

I have 8 registers, so 3 bits will be enough to encode them.

I will also give an example of an operation to make the process clearer.

In the case of an unsigned 16-bit saturation addition, if I add the contents of MM0 and MM4, the instruction will be: 0001000010. That result will be decomposed by the control unit and some signals will be transmitted to the ALU.

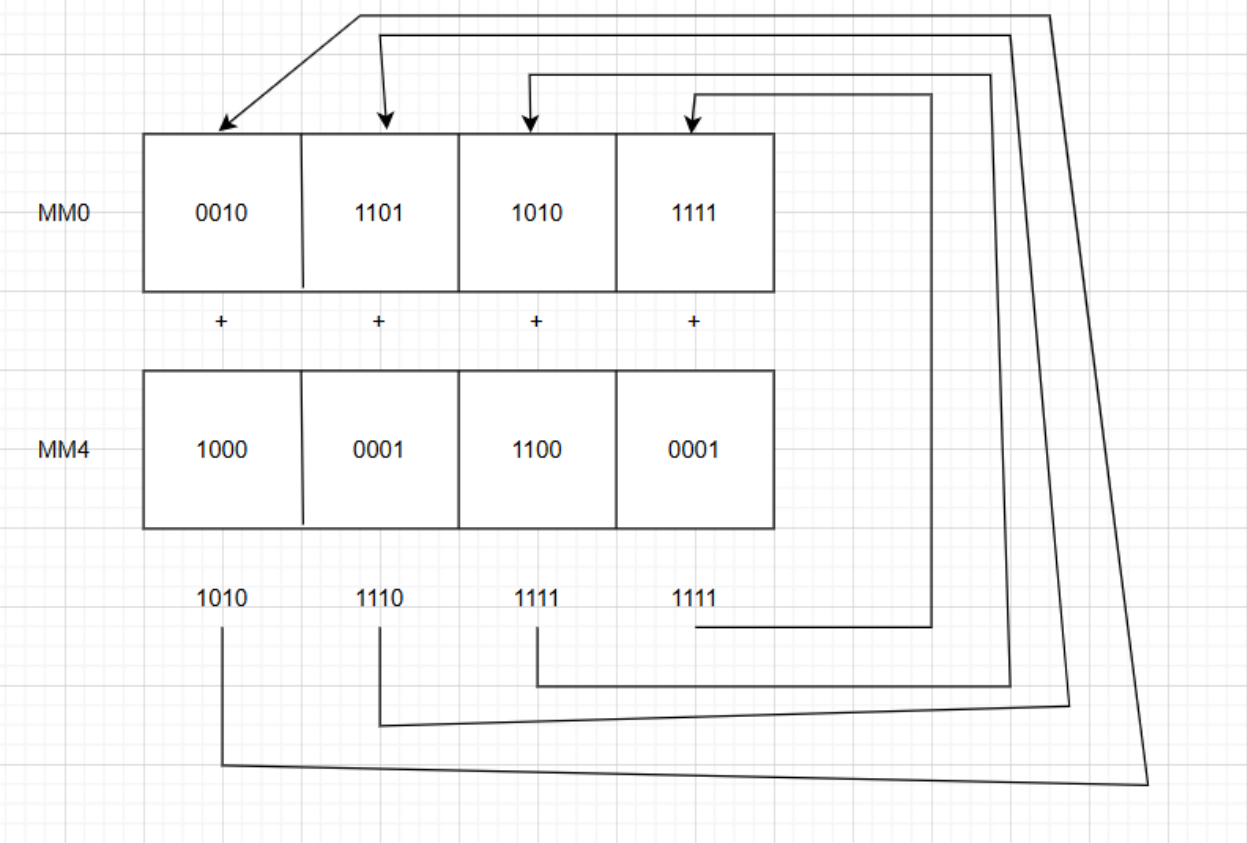


Figure 3.3 example of 16-bit unsigned saturation logic addition

The drawing above exemplifies how the addition is performed using saturation. When overflow occurs, the result is the largest possible number that can be represented on 16 bits. Also, we can see that the result is stored back in the first register.

3.2.5 Description of signed, unsigned, saturation and wrap-around arithmetic

1. Signed vs Unsigned arithmetic
   In signed arithmetic, the numbers are represented using 2's Complement form, allowing the representation of both positive and negative numbers. The MSB is the sign bit, if it is 1 => negative number, if it is 0 => positive number
   In unsigned arithmetic, there is no sign bit, and the representation is the default binary representation of a number.
2. Saturating arithmetic can be used for both addition and subtraction. It is particularly useful in multimedia applications due to the fact that in the case of an underflow it

clamps the result to the lowest value that can be represented and in the case of overflow to the largest possible value.

3. Wrap-around, or modular arithmetic, allows values to "wrap" when they exceed the maximum or minimum representable range. Instead of clamping to a maximum or minimum, the values cycle back around to the opposite end of the range. For example, in the case of signed arithmetic, if we have numbers on 8 bits and I add 1 to 127 the result will be –128. Subtracting 1 from –128 will output 127.

3.2.6 How I plan to implement

For the addition operations I will choose to use Carry Lookahead Adder due to its speed and efficiency. It was shown to me during the laboratory that this is the best option for implementing addition. I will implement a 4-bit adder and then cascade multiple of them to obtain the necessary number of bits for addition. The MMX instruction set allows for adding at most 2 numbers of 32-bits. Subtraction is addition with one number represented in 2's complement.

I will implement all types of addition and subtraction.

For signed addition and subtraction, I will use a 2's complement representation.

Two's Complement Add/Subtract with Saturation for 8-bit Representation

1. **Two's Complement Basics**:
    a. In two's complement, the most significant bit (MSB) is the sign bit, with 0 indicating positive and 1 indicating negative numbers.
    b. Adding and subtracting follows similar principles as unsigned, but we must account for signed overflow (e.g., adding two positives and getting a negative or adding two negatives and getting a positive).
2. **Carry and Overflow Conditions**:
    a. **Overflow** occurs when the result is too large for 8-bit two's complement representation.
    b. If adding two positive numbers yields a negative (overflow), the result should be clamped to 0111 1111 (127 in decimal).
    c. If adding two negative numbers yields a positive (underflow), the result should be clamped to 1000 0000 (-128 in decimal).
3. **Saturation Logic**:
    a. Use the Carry bit (or overflow condition in two's complement) to determine if the result has exceeded the bounds.
    b. Similar to the unsigned case, a multiplexer can select the correct result based on overflow or underflow conditions.

For signed multiplication I will check if the numbers have different signs or not. Transform the negative numbers represented in 2's complement to normal binary representation and then if the

numbers had different signs, turn the result of the multiplication back to 2's complement, if not, leave the result as it is.

To implement saturation or wraparound logic I will have a signal from the control unit that indicates which of the 2 I must select. In the case of wraparound logic, I allow overflow and underflow to occur. In the case of saturation, the carry-out will signal if I must saturate the value: the maximum possible in the case of addition and lowest possible in the case of subtraction.

Here I will outline the conditions a bit better:

- Unsigned + addition => if cout=1 then overflow
- Unsigned + subtraction => if MSBa = 0 and MSBb=1
  MSBa=0 and MSBresult = 1
  MSBb=1 and MSBresult=1 [18]
- Signed + addition => Sign(a) = Sign(b) & Sign(a) != Sign(result)
- Signed + subtraction =>
  - If Sign(a)=0 & Sign(b)=1 & Sign(result)=1 => overflow
  - If Sign(a)=1 & Sign(b)=0 & Sign(result)=0 => underflow

The comparation operations will be implemented using subtraction. Based on the result of the subtraction between the 2 registers I will decide which is the outcome of the comparation. If the result is 0, then the elements are equal, if the result is greater than 0, then the first register has a number bigger than the second register, the last option being the second register having a number greater than the one in the first register.

Bitwise logical operations have a straightforward implementation, and I will not go into extensive detail. I will give a short example for the bitwise and operation: a0 and b0 = r0, a1 and b1 = r1 ..etc.

Design

Here I will give a detailed overview of the whole system and present in more detail the main components that make the operations: adders, multipliers etc.

I will explain the diagram below.

The Instruction memory is a ROM that will hold multiple instructions that are to be executed by the main ALU. We should note that the ALU on this diagram will also be detailed further in the next section of the Design chapter. I maintain a Program Counter which will fetch the next operation that must be performed by the ALU. This memory holds a few instructions of 12 bits. The Data Memory is also a ROM, and I maintain a few sets of 64-bit data that can be loaded in the MMX registers to perform various tests. I also have an Address counter to hold the address of

the next set of data which must be loaded to the MMX registers. The Data memory will be of 512 bits and might hold a few pairs of 512 bits (512 = 8 registers * 64 bits). The control Unit will output bits_8_16_32 a signal that indicates the data type we operate on, Add_Sub a signal that is used to make the difference between addition and subtraction inside the Alu, logical_op a signal which in case of a logical operation will specify which of the 4 will be performed, sel_unit which select the result that is fed to the output register based on its type, S/U a signal that will tell if the operation requires signed or unsigned numbers, S/W a signal that tells if the operation uses saturation or wraparound and, the codification of the two registers that will be used to perform the operation.

After the operation is performed in the alu, the output will be stored in OUTREG which is a shift register which will shift out 16 bits at a time. Those 16 bits will be showed on the SSD of the basys3. The shift operation is performed when pressing the Print button.

Also, a new instruction will be executed when clk button is pressed. LoadData button will load the data into the 8 MMX registers.

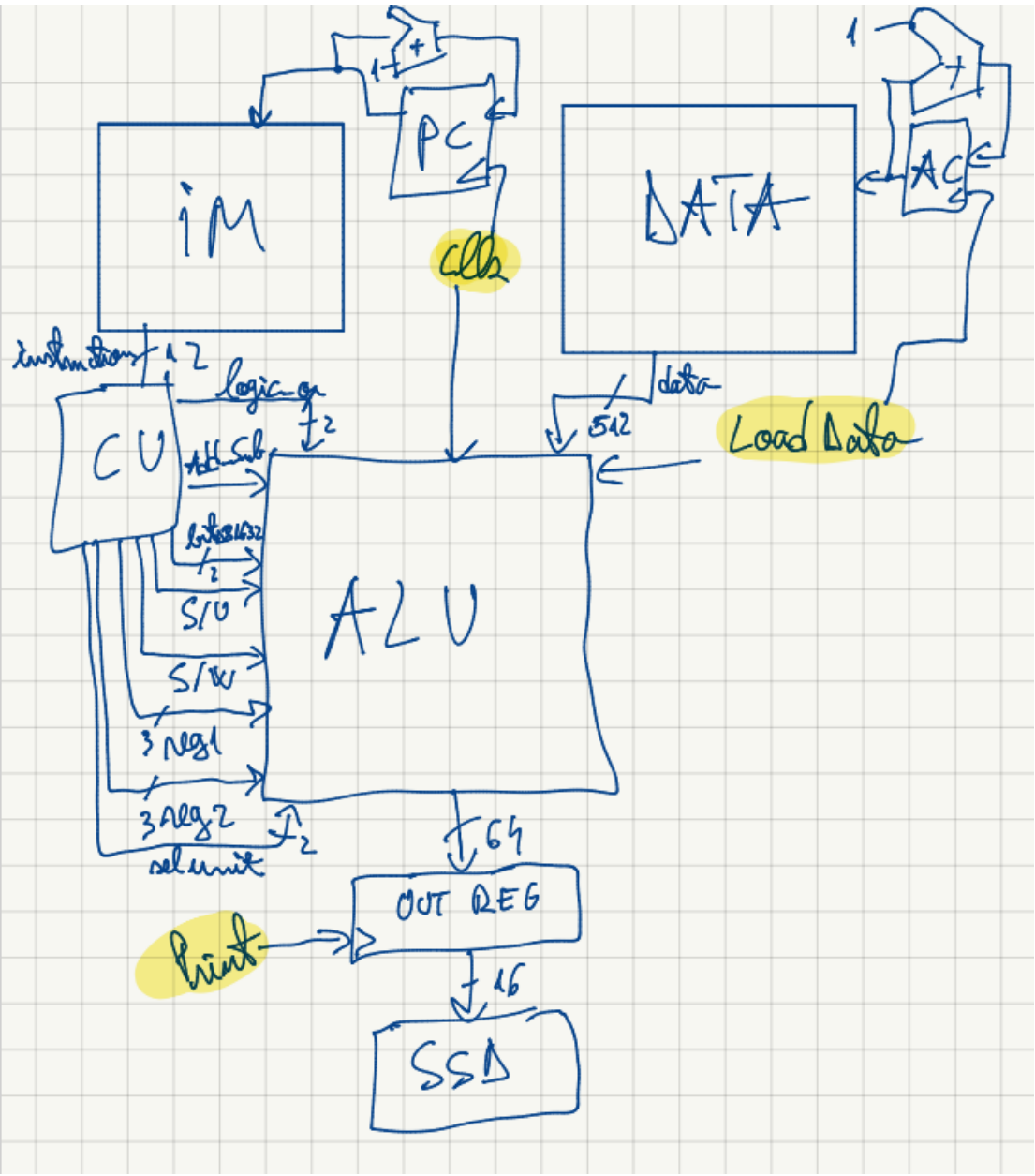


Figure 4.1 top level of the project

4.1 Main ALU

The diagram below shows the ALU from the previous diagram in more detail. The 8 MMX registers are shown here. Aso we can see the LoadData work on positive logic and the process of loading is asynchronous. The clk signal is a button and synchronizes the writing in the registers. The small muxs will select based on the s signals if the registers gets the result from the ALU unit or if the same value will be written again. At a time only one of the 8 s signals will be 1 based on the reg1 signal. The 2 big MUXs select the two registers used for the operatios. The ActualALU will be described below. From ActualALU we can see that the result is transmitted to the OUT.

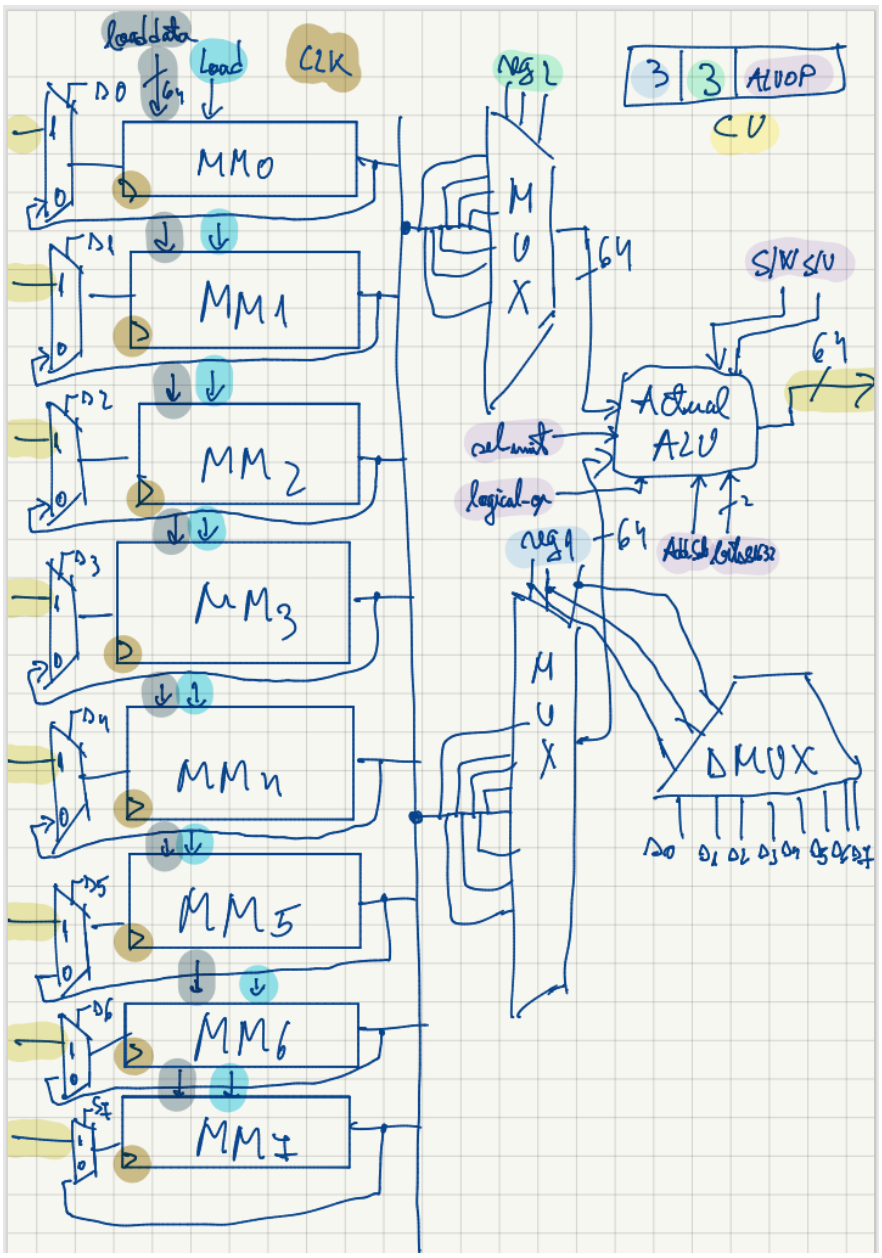


Figure 4.2 the unit that is holding the MMX specific componnets

4.2 Actual ALU

Here I roughly sketched the way the ALU looks from a very abstract point of view.

I select the correct final result using select_unit, to select between the 4 units, and I must now describe how each of the elements looks like.

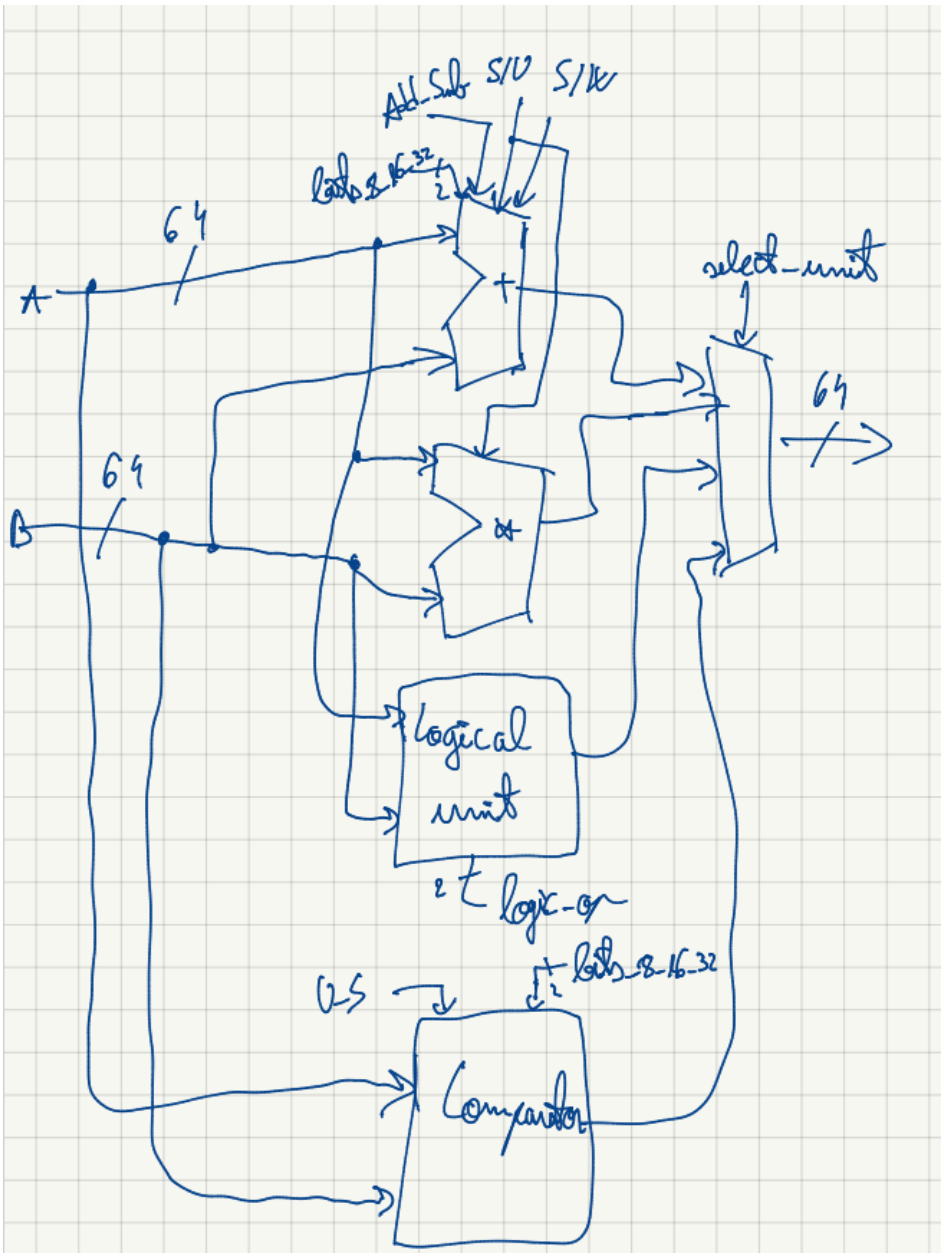


Figure 4.3 ALU unit

4.3 Adder unit

I really wanted to make this as cost efficient as possible, so I will only use 8 8-bit adders. Because of this choice of design, my logic became much more complex, using a lot of MUX.

I have a unit that takes as input ALUOP and then sends a signal made up of 3 bits which tells if the addition/subtraction must be made on 8/16/32 bits. Then I also have a 2'Complement unit

which will decide how the second operator should be converted based on the operation (explained at the analysis step). I divide my 2 64-bit numbers into bunches of 8 each. I use a series of MUXs to decide if I need to propagate the carryout or not, based on the data type I work with. Then I use another series of MUXs which will calculate the logic in the case of 8-bit numbers. My next series of MUXs is used to calculate the logic for 16-bit numbers. The last 2 MUXs calculate the logic for 32-bit numbers. The saturation logic is computed as presented in the class, based on the operation (add/sub) and the carryout bit. All these MUXs have as selector the Add/Sub, S/W signals and caryyout bit.

The final step is to choose between these 3 results based on the signal 8/16/32.

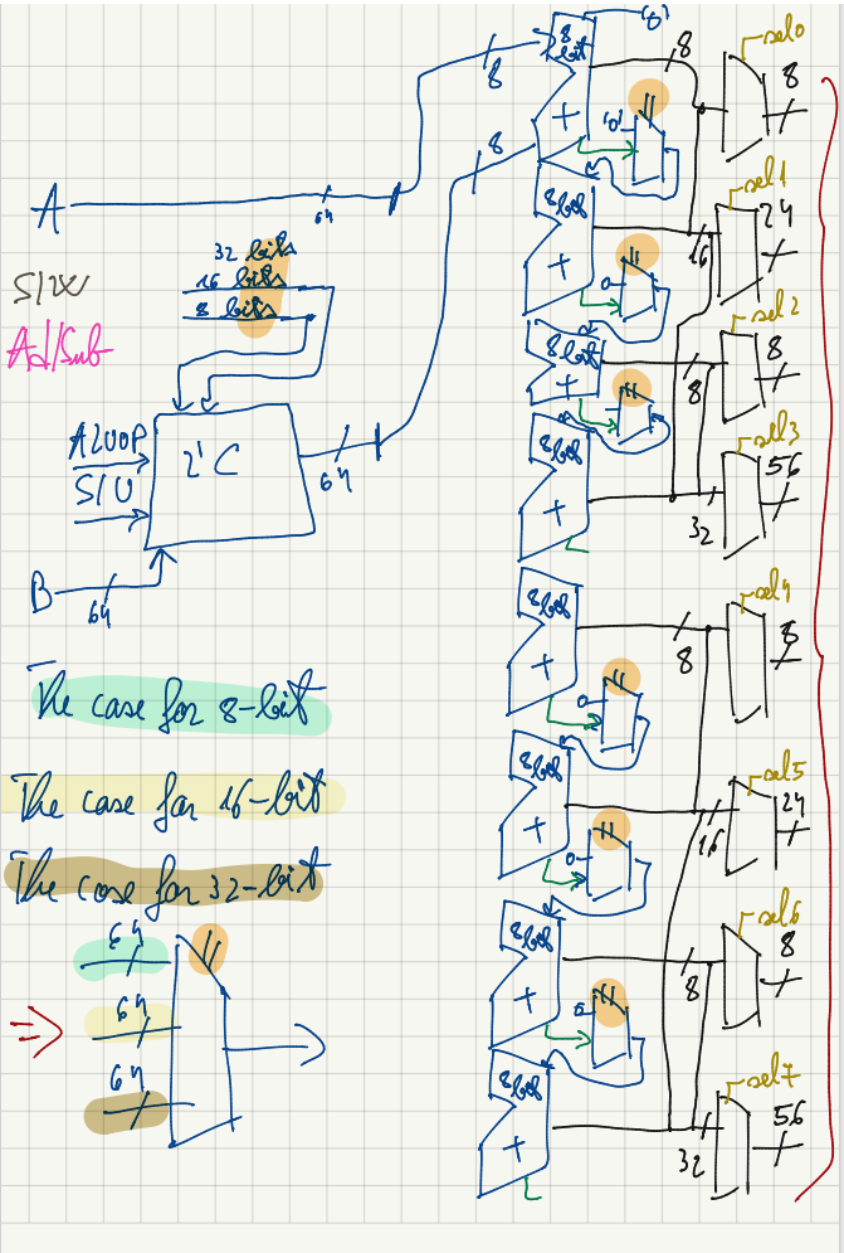


Figure 4.4 addition unit

The 8-bit adders are obtained using 2 concatenated 4-bit Carry Lookahead adders.

I will present here the structure of the adder.

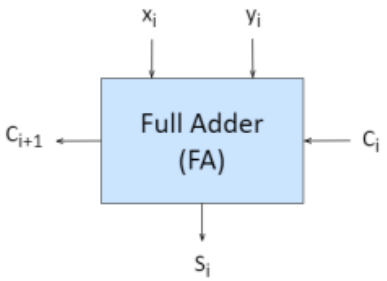


Figure 1: Full adder

x_i	y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1: Full adder truth table

Given the truth table above, the Boolean expressions of the outputs (after reduction) are:

$$S_i = x_i \oplus y_i \oplus C_i$$
$$C_{i+1} = x_i \cdot y_i + (x_i + y_i) \cdot C_i$$

Figure 4.5[16]

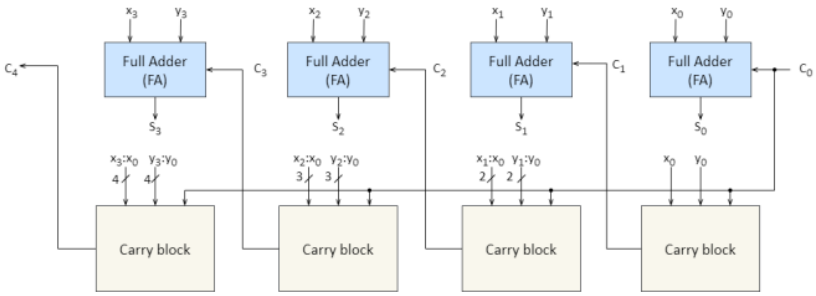


Figure 3: Carry lookahead adder design for adding 4-bit numbers

The main advantage of such a type of adder is that the output carries can be individually computed based on the input carry and the bits in the operand. Thus, the latency is significantly reduced, as the computation of the output carry at level $i+1$ does not require all the previously i carry outputs to be already computed, as it can be expressed only in terms of the inputs. An example for two 4-bit operands, $x(3..0)$ and $y(3..0)$, is given below:

$$C_1 = g_0 + p_0 \cdot C_0 = x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0$$

$$C_2 = g_1 + p_1 \cdot C_1 = x_1 \cdot y_1 + (x_1 + y_1) \cdot C_1 = x_1 \cdot y_1 + (x_1 + y_1) \cdot (x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0)$$

$$\begin{aligned} C_3 = g_2 + p_2 \cdot C_2 = x_2 \cdot y_2 + (x_2 + y_2) \cdot C_2 = x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot C_1) \\ = x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot (x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0)) \end{aligned}$$

$$\begin{aligned} C_4 = g_3 + p_3 \cdot C_3 = x_3 \cdot y_3 + (x_3 + y_3) \cdot C_3 = x_3 \cdot y_3 + (x_3 + y_3) \cdot (x_2 \cdot y_2 + (x_2 + y_2) \cdot C_2) \\ = x_3 \cdot y_3 + (x_3 + y_3) \cdot (x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot C_1)) \\ = x_3 \cdot y_3 + (x_3 + y_3) \cdot (x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot (x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0))) \end{aligned}$$

Figure 4.6 [16]

4.4 Multiplication Unit

My algorithm of choice is matrix multiplication, as I consider it is the best option due to the fact that it uses only adders and shifter. For an asynchronous design it works best, this way after validating the start of a multiplication, the result will be instantaneous, the only delays are due to the logic gates and wires. The diagram below shows the multiplication unit using 16-bit multipliers shift and add, which are described below. The first diagram shows the input signals being divided into 4 segments of 16-bits, as the multiplication operations are on 16-bit data. The make positive unit will transform the numbers from a signed representation to an unsigned representation. Here the algorithm works as presented in the analysis part. There are those sign xor gates which will determine if the result of each multiplication must be converted to 2's Complement representation or not, based on the sign of the operands and the state of the U/S signal. Then, after all the 16-bit operations have been made, I compute the 2 partial results based on the type of operation I perform: I either store the high part of each 32-bit result into the 64-bit

MMX register, or I add two 32-bit partial results and then store them in the MMX register. The selection is made based on the S/U signal.

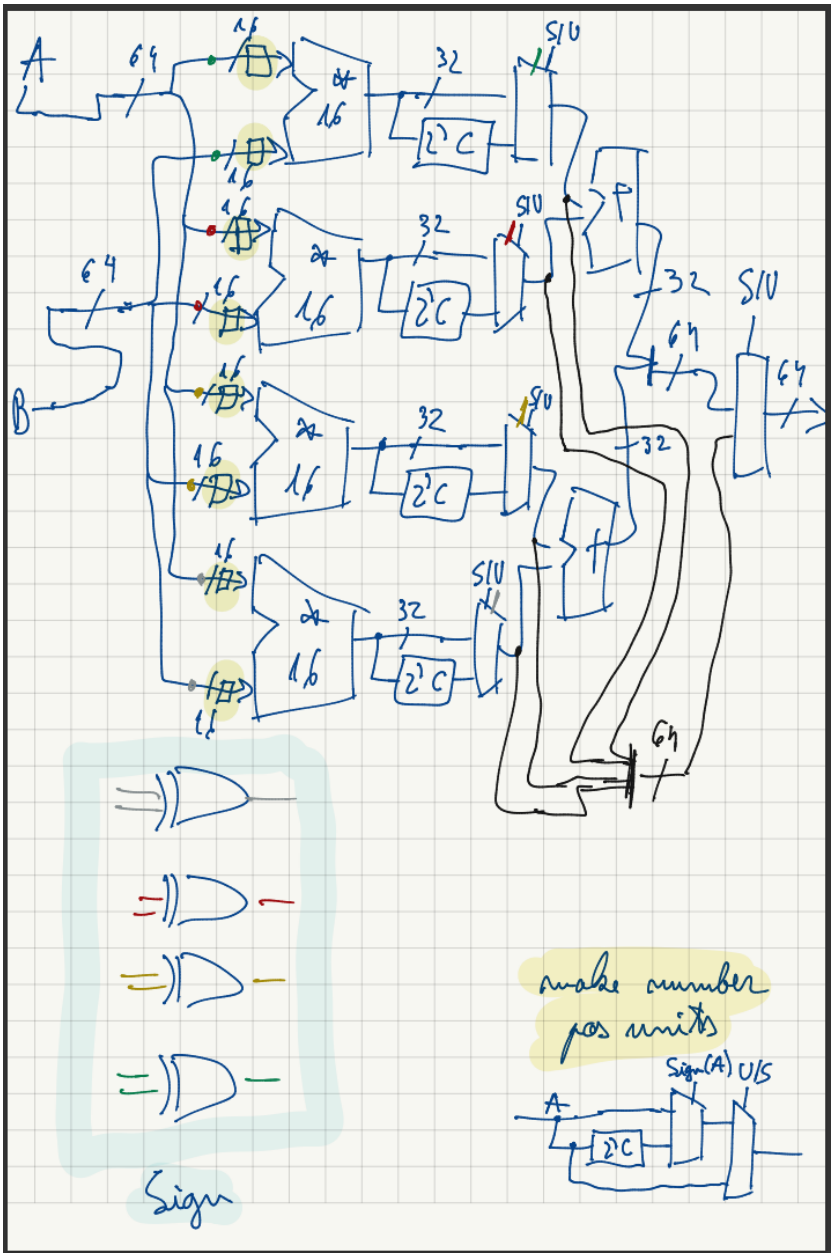


Figure 4.7 multiplication unit

The next diagram is a snippet from the course where multiplication is explained.

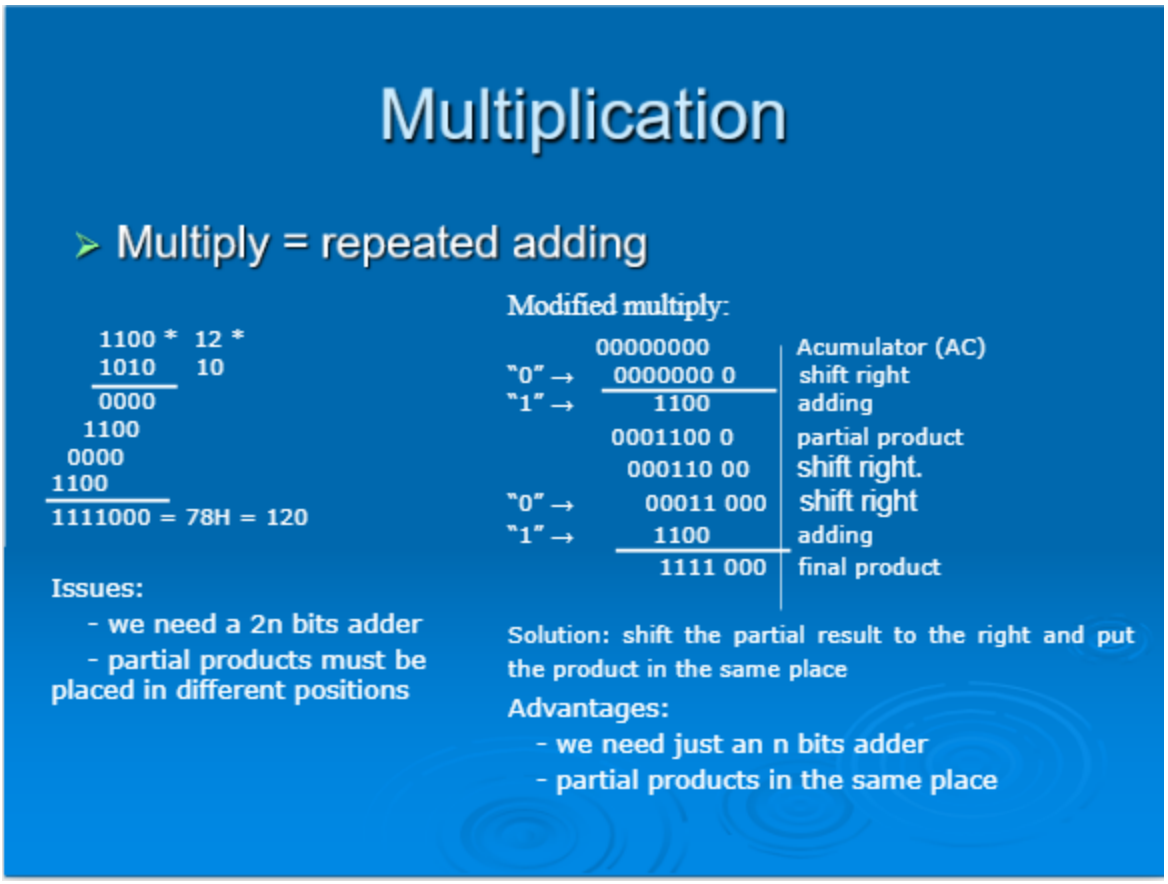


Figure 4.8 [10]

The third diagram shows how I compute the final 32-bit result. I must shift every product based on the order of priority. This is a bit hard to explain, but basically, I want my products to be aligned as in the natural way of making a decimal multiplication.

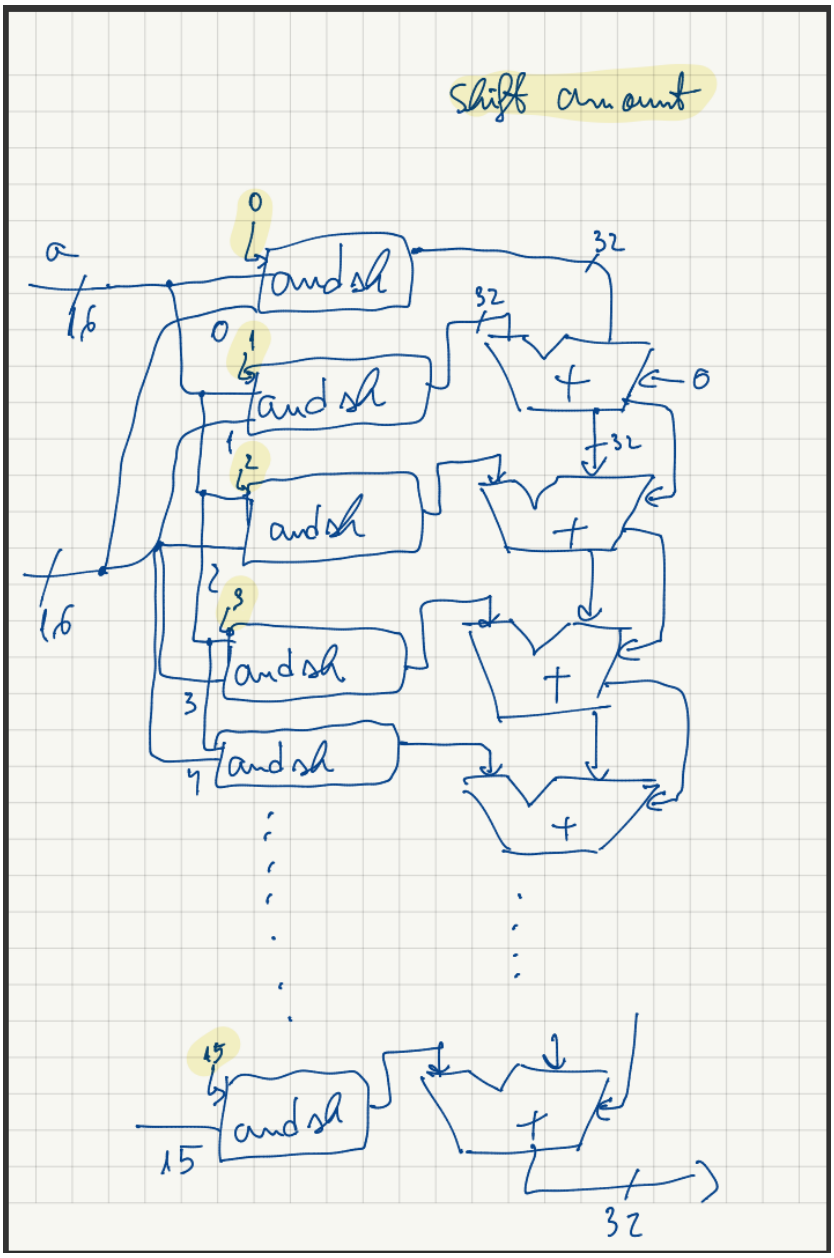


Figure 4.9 internal structure of a 16-bit MUL

Next up, I will present the structure of an andsh unit, first I make and “and” using a mux between the number a and the i-th bit of the second operand, b. If the selection is ‘1’ I leave the number, else I replace it with 0,

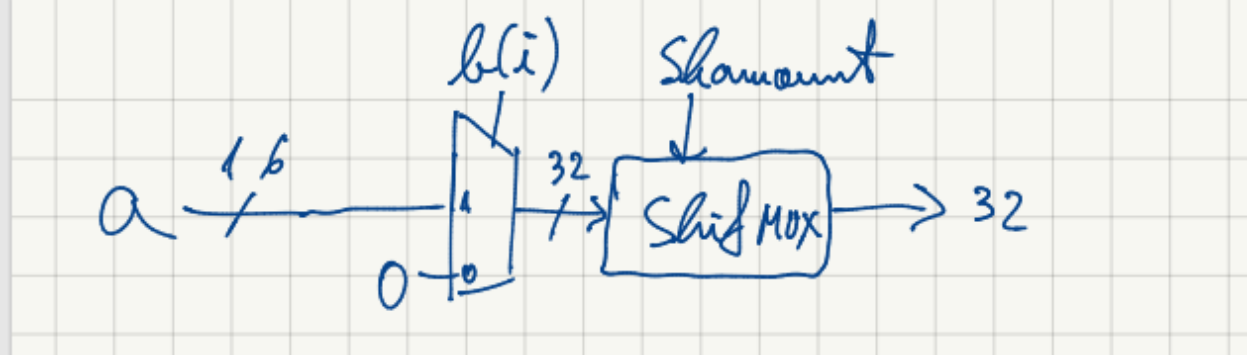


Figure 4.10 structure of the andsh uit

4.5 Comparison Unit

The comparison unit has an initial design very similar to the adder unit. Inside this unit, however, I will only perform subtractions on signed or unsigned values. I have a 2'Complement unit for the second number. If I have unsigned values, then the second number will be converted. However, if we have signed values, if the second number is already negative, then I will convert it to positive representation and then perform the addition.

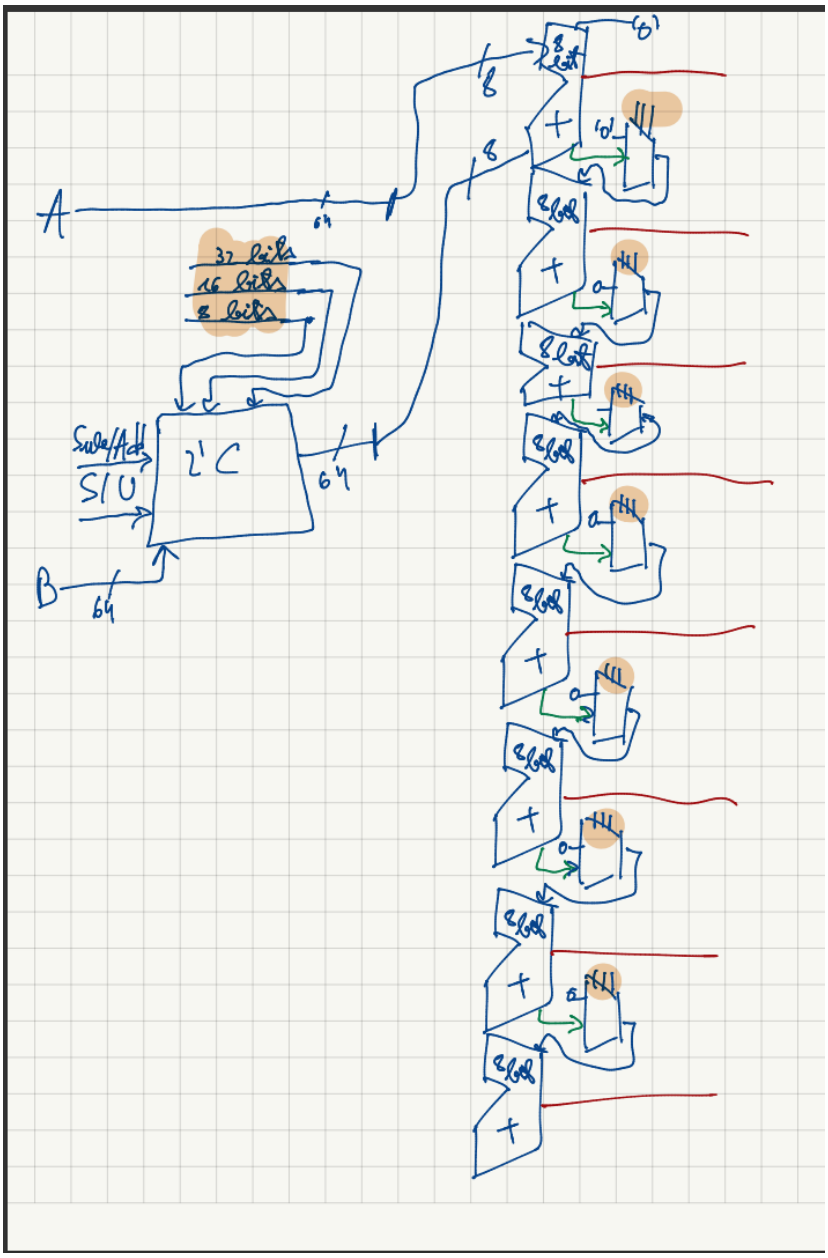


Figure 4.11 [19]

After the 8-bit partial results are computed I have a series of MUXs represented below which perform the comparison for all types of data 8/16/32 bits. Then we concatenate each result 8x8 bits => 64, 4x16 bits => 64, 2x32 bits => 64. The final MUX will choose between the three representations based on the ALUOP.

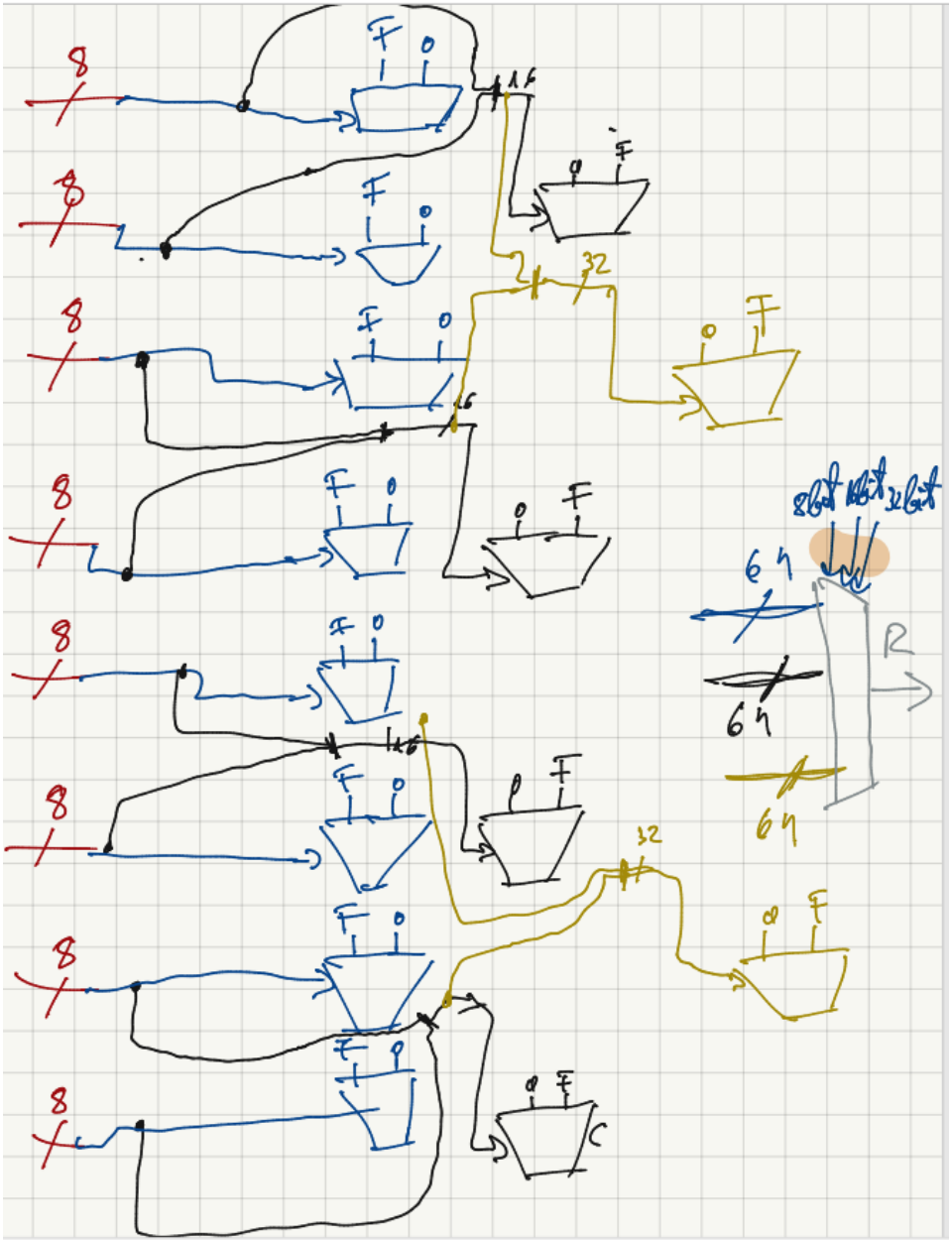


Figure 4.12 [19]

4.6 Control Unit

This control unit indicates which signals I need to operate the ALU.

M=multiplication 1 bit

Reg1, reg2 = MMX registers address 3 bits each

S/W = saturated/wraparound 1 bit

S/U = signed unsigned 1 bit

8/16/32 signal indicates the type of data I operate with. 3 bits

Add/Sub indicates which kind of operation I perform 1 bit

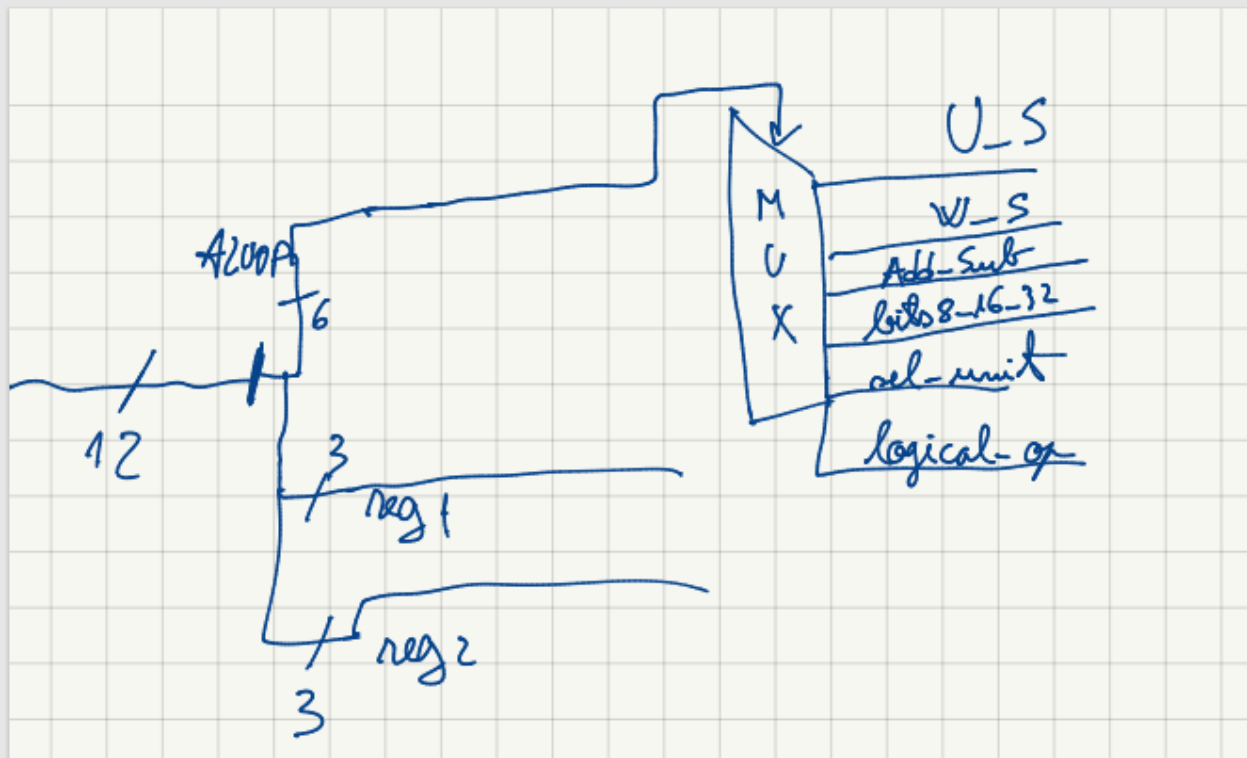


Figure 4.13 [19]

4.7 2'Complement complex unit

This 2's Complement unit is the one used in both adder/subtraction unit and comparison unit. It takes all possible data types and chooses between the representations. I realized after further analysis that I only need the Sub/Add signal in order to determine whether to use the representation that went through the 2's complement unit or to keep the original representation.

I also realized that I need a new signal called Add/Sub which will tell me which form of the number I must select. Add/Sub will tell me what kind of operation I perform; addition or subtraction.

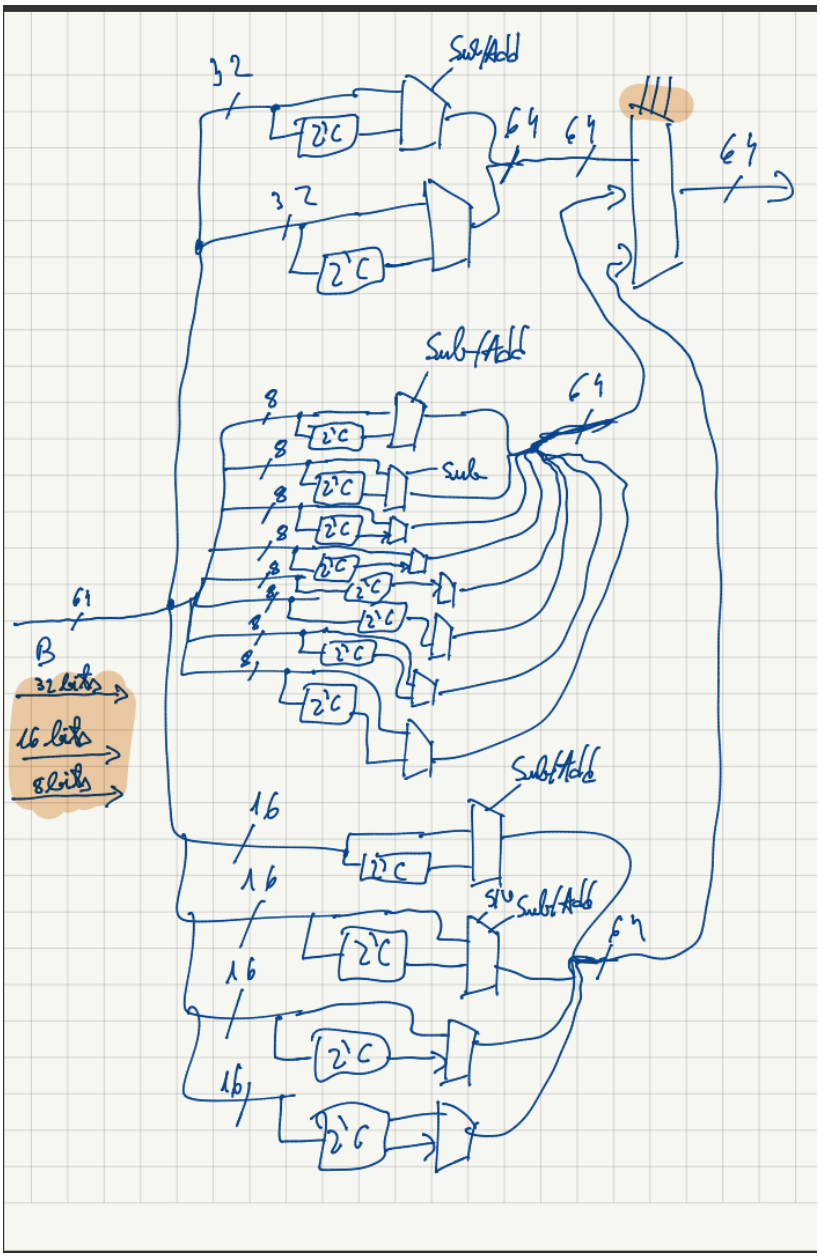


Figure 4.13 internal structure of the comparison unit

4.8 Logical unit

This unit will perform the logical instructions, I have and,or,nand and xor units. I will choose the output of the function based on the logic_op selector.

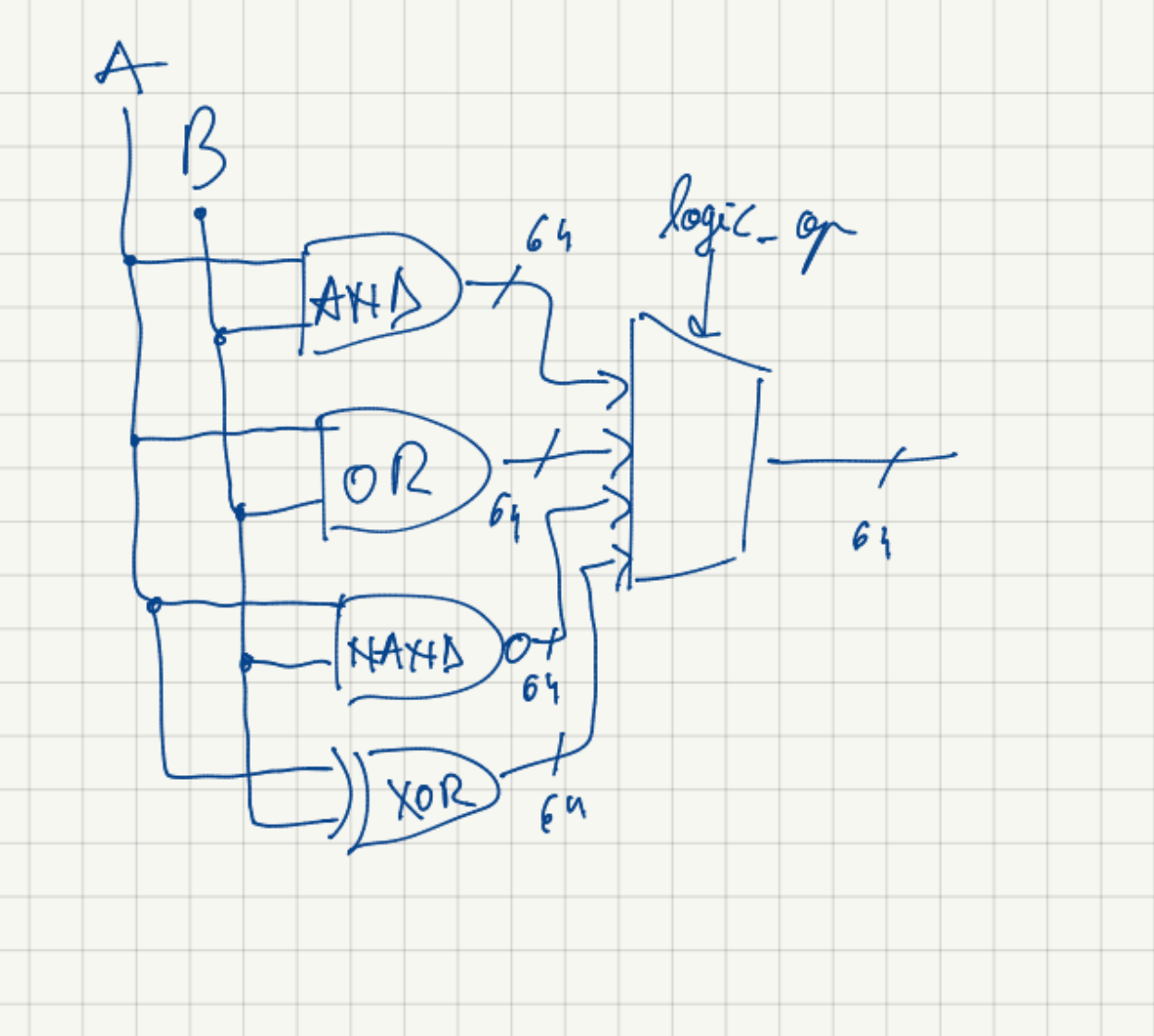


Figure 4.14 internal structure of the logical unit

4.9 Encoding instructions

Encoding Table for MMX Instructions

Addition Operations (Packed Addition)

Instruction	Opcode	Description
paddb	000000	Add packed unsigned 8-bit (wrap-around)
paddw	000001	Add packed unsigned 16-bit (wrap-around)
paddd	000010	Add packed unsigned 32-bit (wrap-around)
paddsb	110010	Add packed signed 8-bit (saturating)

paddsw	000011	Add packed signed 16-bit (saturating)
paddusb	000100	Add packed unsigned 8-bit (saturating)
paddusw	000101	Add packed unsigned 16-bit (saturating)
paddsd	010111	Add packed signed 32-bit (saturating)
paddud	011000	Add packed unsigned 32-bit (saturating)
paddrb	011001	Add packed signed 8-bit (wrap-around)
paddrw	011010	Add packed signed 16-bit (wrap-around)
paddrd	011011	Add packed signed 32-bit (wrap-around)

Multiply Operations

Instruction	Opcode	Description
paddw	000111	Multiply and add packed unsigned 16-bit integers
pmulhw	001000	Multiply packed signed 16-bit integers, store high

Subtraction Operations (Packed Subtraction)

Instruction	Opcode	Description
psubb	001001	Subtract packed unsigned 8-bit (wrap-around)
psubd	001010	Subtract packed unsigned 32-bit (wrap-around)
psubsb	001011	Subtract packed signed 8-bit (saturating)
psubsw	001100	Subtract packed signed 16-bit (saturating)
psubusb	001101	Subtract packed unsigned 8-bit (saturating)
psubusw	001110	Subtract packed unsigned 16-bit (saturating)
psubw	001111	Subtract packed unsigned 16-bit (wrap-around)
psubsd	011100	Subtract packed signed 32-bit (saturating)
psubud	011101	Subtract packed unsigned 32-bit (saturating)
psubrb	011110	Subtract packed signed 8-bit (wrap-around)
psubrw	011111	Subtract packed signed 16-bit (wrap-around)
psubrd	100000	Subtract packed signed 32-bit (wrap-around)

Comparison Operations

Instruction	Opcode	Description
pcmpeqb	010000	Compare packed unsigned 8-bit for equality
pcmpeqd	010001	Compare packed unsigned 32-bit for equality
pcmpeqw	010010	Compare packed unsigned 16-bit for equality
pcmpeqsb	100001	Compare packed signed 8-bit for equality

pcmpeqsw	100010	Compare packed signed 16-bit for equality
pcmpeqsd	100011	Compare packed signed 32-bit for equality

Bitwise Logical Operations

Instruction	Opcode	Description
pand	010011	Bitwise logical AND
pandn	010100	Bitwise logical AND NOT
por	010101	Bitwise logical OR
pxor	010110	Bitwise logical XOR

Control Signal Table

Addition Operations (sel unit = 00)

Instruction	Opcode	S/W	S/U	8/16/32	Add/Sub	sel unit	logic op	Notes
paddb	000000	0	0	10	0	00	-	Unsigned 8-bit wrap-around add
paddw	000001	0	0	01	0	00	-	Unsigned 16-bit wrap-around add
paddd	000010	0	0	00	0	00	-	Unsigned 32-bit wrap-around add
paddsb	110010	1	1	10	0	00	-	Signed 8-bit saturating add
paddsw	000011	1	1	01	0	00	-	Signed 16-bit saturating add
paddusb	000100	1	0	10	0	00	-	Unsigned 8-bit saturating add
paddusw	000101	1	0	01	0	00	-	Unsigned 16-bit saturating add
paddsd	010111	1	1	00	0	00	-	Signed 32-bit saturating add
paddud	011000	1	0	00	0	00	-	Unsigned 32-bit saturating add
paddrb	011001	0	1	10	0	00	-	Signed 8-bit wrap-around add

paddrw	011010	0	1	01	0	00	-	Signed 16-bit wrap-around add
paddrd	011011	0	1	00	0	00	-	Signed 32-bit wrap-around add

Multiply Operations (sel unit = 01)

Instruction	Opcode	S/W	S/U	8/16/32	Add/Sub	sel unit	logic op	Notes
paddw	000111	0	0	01	0	01	-	Interpreted as unsigned multiply-add (word)
pmulhw	001000	0	1	01	0	01	-	Interpreted as signed multiply and store high

Subtraction Operations (sel unit = 00)

Instruction	Opcode	S/W	S/U	8/16/32	Add/Sub	sel unit	logic op	Notes
psubb	001001	0	0	10	1	00	-	Unsigned 8-bit wrap-around sub
psubd	001010	0	0	00	1	00	-	Unsigned 32-bit wrap-around sub
psubsb	001011	1	1	10	1	00	-	Signed 8-bit saturating sub
psubsw	001100	1	1	01	1	00	-	Signed 16-bit saturating sub
psubusb	001101	1	0	10	1	00	-	Unsigned 8-bit saturating sub
psubusw	001110	1	0	01	1	00	-	Unsigned 16-bit saturating sub
psubw	001111	0	0	01	1	00	-	Unsigned 16-bit wrap-around sub
psubsd	011100	1	1	00	1	00	-	Signed 32-bit saturating sub
psubud	011101	1	0	00	1	00	-	Unsigned 32-bit saturating sub

psubrb	011110	0	1	10	1	00	-	Signed 8-bit wrap-around sub
psubrw	011111	0	1	01	1	00	-	Signed 16-bit wrap-around sub
psubrd	100000	0	1	00	1	00	-	Signed 32-bit wrap-around sub

Comparison Operations (sel unit = 10)

Comparisons just compare equality. No add/sub, no saturations. S/U=1 for signed variants, 0 for unsigned. logic op is not applicable. They have defined widths.

Instruction	Opcode	S/W	S/U	8/16/32	Add/Sub	sel unit	logic op	Notes
pcmpeqb	010000	-	0	10	-	10	-	Compare 8-bit unsigned eq
pcmpeqd	010001	-	0	00	-	10	-	Compare 32-bit unsigned eq
pcmpeqw	010010	-	0	01	-	10	-	Compare 16-bit unsigned eq
pcmpeqsb	100001	-	1	10	-	10	-	Compare signed 8-bit eq
pcmpeqsw	100010	-	1	01	-	10	-	Compare signed 16-bit eq
pcmpeqsd	100011	-	1	00	-	10	-	Compare signed 32-bit eq

Logical Operations (sel unit = 11)

For logical ops, we only set logic op accordingly:

Instruction	Opcode	S/W	S/U	8/16/32	Add/Sub	sel unit	logic op	Notes
pand	010011	-	-	-	-	11	00	AND

pandn	010100	-	-	-	-	11	10	AND NOT (NAND)
por	010101	-	-	-	-	11	01	OR
pxor	010110	-	-	-	-	11	11	XOR

S/W = 0 => wrap around

S/W = 1 => saturation

S/U = 0 => unsigned

S/U = 1 => signed

Add/Sub =0 => addition

Add/Sub = 0 => subtraction

Implementation

In this part I will not go into detail and describe the way in which I wrote the VHDL code.

5.1 Addition_Subtraction_Unit

5.1.1. 8-bit Adders

The first step in implementing these adders was to create a 1-bit full-adder based on the schema described in the laboratory. The diagram and the dataflow formula can be seen in the design chapter. After having the 1-bit full adders I created the carry lookahead units. Now I could implement the 4-bit adder described in the laboratory. Once I have the 4-bit adders, all I have to do is to concatenate two 4-bit adders and obtain an 8-bit adder.

5.1.2 2's Complement complex unit

To create this unit, I used multiple muxs and 2's complement units which work on 8,16 and 32 bits. I created 32-bit,16-bit and 8-bit carry lookahead adders for the 2's complement. After converting to 2's complement I use a mux with the Add_Sub control signal that tells me if I need the number converted, or I should stick to the initial representation. I use a mux which will select which packed data I currently work with, based on the bits_8_16_32 signal.

5.1.3 Main Add/Sub unit

The code implements exactly the diagram proposed at the design step.

I have come to this implementation after consultations during the project hour meetings and I reduced the amount of mux's of the initial design. The mux's that use the same control signals and were concatenated are now a single larger mux. The idea is that I have mux's that only choose the arithmetic logic (wrap around/ saturation) for 8-bit numbers. Then I have mux's which choose the arithmetic logic in the case of both 8 and 16-bit numbers, so the result will be 24 bits, the high 16 -> for the 16-bit logic and the low 8 for the 8-bit logic. Lastly I have muxs that choose the arithmetic for both 8,16 and 32 bits. These muxs have an output of 56 bits, the logic is the same as for the 24-bit output.

I opted to use as little components as possible, so I used only 8-bit adders, concatenating two groups of 4. At last, I use the bits_8_16_32 signal to select between the 3 results, which contain the results for 8,16- and 32-bit packed data operations.

5.2 Multiplication unit

5.2.1 16-bit multiplier

After trying to use Wallace Tree multiplier and deciding that it's too difficult to implement, I choose a simpler, but efficient method, matrix multiplication. The code is implemented in a structural manner, so it obeys the diagram presented in the design part. The Shift-and units will take every 16-bit result, make a bitwise and between the operands and shift the result with a predefined shift amount. I do this for every possible combination of 16-bit of the two 64-bit operands. From these Shift-and units I obtain 32-bit partial results. I basically have a matrix. I add all the lines of the matrix and obtain the final correct 32-bit result.

5.2.2 Multiplication unit

For the multiplication unit I make my 2 operands positive using a make pos unit that chooses between the initial representation of the number or the 2's Complement representation. Then I divide the two numbers into 16-bit parts and input them into 16-bit multipliers. After I have the 32-bit result I will once again choose if I must convert that result to 2's complement representation, based on whether I have signed numbers, and the numbers have different signs.

Then based on whether I have signed or unsigned representation, I can choose if I must store the high part of each of the 4 results, or add them and then store them, this depends on the U_S selection signal.

5.3 Comparison Unit

I implemented the equality operations, so I do this by subtracting the numbers and checking if the result is 0 or not. If the result is 0 then I will put 1 in the result of the comparison unit. I can check for equality on different sized data, 8,16,32 bit data. The unit is described in a structural manner, just like the others. The internal architecture is very much alike the add_sub unit. The difference is the 8-bit results from the adders will now act as selection for mux's that will decide whether the 2 numbers are equal or not. All used muxs are written using the with select statement from VHDL.

5.4 Logical Unit

The logical unit has a simple design, it is comprised of a AND unit, of a OR_UNIT, NAND_UNIT and XOR_UNIT. These units are created using the generate instruction from VHDL, in order to avoid redundancy. Each unit will output a 64-bit result, which will be selected using a mux that has the selection signal logic_op coming from the control unit.

5.5 Main ALU Unit

This unit is the actual Arithmetic unit my project aims to create. The design is again structural and closely follows the indications from the design chapter. I take the 4 units described earlier, instantiate each of them and then select the output of the unit based on the select_unit signal, a 2 bit signal that comes from the control unit and will indicate what type of operation I perform.

5.6 ALU with MMX Registers

This is described using both behavioral and structural methods. I instantiate my main alu, but however, for the 8 MMX registers I used a behavioral style, since I need a clock signal. I have a modular design as I used separate components also for the muxs and register process for a better understanding of the top level.

Testing and Validation

As requested, here I have the table with all 36 test cases which will showcase the correct functioning of the MMX unit.

6. Table with tests for the top level

Before execution of an instruction, data must be loaded into the registers. I have populated the ROM Data memory with multiple sets of random numbers. The first data set is loaded before any instruction is performed, then we load a new set of data after the first 3 instructions. I did this just to show that the load functionality works correctly. Also, I perform another 2 loads, for the sake of having random tests. One is made before the first compare instruction. The other is made before the logical operations.

Fig 6,Top module test table

opcode reg1 reg2	NR1_reg1	NR2_reg2	expected result	operation
000000 001 010	_01020304050 60708	_99AABBCCDDE EFF00	9aacbed0e2f4 0608	Unsigned 8-bit wrap-around add reg1 and reg2
000001 000 001	_08070605040 30298	9aacbed0e2f406 08	a2b3c4d5e6f7 08a0	Unsigned 16-bit wrap-around add reg0 and reg1
000010 111 101	8a47b3c1d2e9 f001	abcdef01234567 89	3615a2c2f62f5 78a	Unsigned 32-bit wrap-around add reg7 and reg5
110010 011 100	778899aabbcc ddee	_2233445566778 899	7fbbddff21438 087	Signed 8-bit saturating add reg3 and reg4
000011 110 101	87654321cdef ab90	aabbccddeeff112 2	80000ffebceeb cb2	Signed 16-bit saturating add reg6 and reg5
000100 001 000	1234567890ab cdef	fedcba09876543 21	ffffff81ffffffff	Unsigned 8-bit saturating add reg1 and reg0
000101 011 100	7fbbddff21438 087	_2233445566778 899	a1eeffff87bafff f	Unsigned 16-bit saturating add reg3 and reg4
010111 010 111	ff0011223344 5566	ba98fedc123456 78	b9990ffe4578a bde	Signed 32-bit saturating add reg2 and reg7
011000 101 110	aabbccddeeff1 122	80000ffebceebcb 2	ffffffffffffffff	Unsigned 32-bit saturating add reg5 and reg6
011001 000 111	fedcba098765 4321	ba98fedc123456 78	b874b8e59999 9999	Signed 8-bit wrap-around add reg0 and reg7
011010 001 100	ffffff81ffffffff	_2233445566778 899	223243d66676 8898	Signed 16-bit wrap-around add reg1 and reg4
011011 010 100	b9990ffe4578a bde	_2233445566778 899	dbcc5453abf0 3477	Signed 32-bit wrap-around add reg2 and reg4
000111 100 001	_22334455667 78899	223243d666768 898	16acd20471e5 07b2	Interpreted as unsigned multiply-add (word) reg4 and reg1
001000 000 001	b874b8e59999 9999	223243d666768 898	f671ed28d703 2fc3	Interpreted as signed multiply and store high reg0 and reg1

001001 110 101	80000ffebceeb cb2	ffffffffffffffff	810110ffbdefb db3	Unsigned 8-bit wrap-around sub reg6 and reg5
001010 011 101	a1eeffff87bafff f	ffffffffffffffff	a1ef000087bb 0000	Unsigned 32-bit wrap-around sub reg3 and reg5
001011 010 011	dbcc5453abf0 3477	a1ef000087bb00 00	3add54532435 3477	Signed 8-bit saturating sub reg2 and reg3
001100 100 111	16acd20471e5 07b2	ba98fedc123456 78	5c14d3285fb1 b13a	Signed 16-bit saturating sub reg4 and reg7
001101 001 011	223243d66676 8898	a1ef000087bb00 00	000043d60000 8898	Unsigned 8-bit saturating sub reg1 and reg3
001110 000 010	f671ed28d703 2fc3	3add545324353 477	bb9498d5b2ce 0000	Unsigned 16-bit saturating sub reg0 and reg2
001111 111 110	ba98fedc1234 5678	810110ffbdefbdb 3	3997eddd5445 98c5	Unsigned 16-bit wrap-around sub reg7 and reg6
011100 101 110	ffffffffffffffff	810110ffbdefbdb 3	7efeef0042104 24c	Signed 32-bit saturating sub reg5 and reg6
011101 001 010	000043d60000 8898	3add545324353 477	_00000000000 00000	Unsigned 32-bit saturating sub reg1 and reg2
011110 000 100	bb9498d5b2ce 0000	5c14d3285fb1b1 3a	5f80c5ad531d 4fc6	Signed 8-bit wrap-around sub reg0 and reg4
011111 001 101	_00000000000 00000	7efeef00421042 4c	81021100bdf0 bdb4	Signed 16-bit wrap-around sub reg1 and reg5
100000 011 000	a1ef000087bb 0000	5f80c5ad531d4f c6	426e3a53349d b03a	Signed 32-bit wrap-around sub reg3 and reg0
010000 001 010	f10223fa67789 000	f11223fa677890 00	ff00ffffffffffff	Compare 8-bit unsigned eq reg1 and reg2
010001 100 101	bbaa99887766 5544	33221100ffeedd cc	_00000000000 00000	Compare 32-bit unsigned eq reg4 and reg5
010010 110 111	7869221100ffe ecc	78695a4b3c2d1e 01	ffff000000000 000	Compare 16-bit unsigned eq reg6 and reg7
100001 001 010	ff00ffffffffffff	f11223fa677890 00	_00000000000 00000	Compare signed 8-bit eq reg1 and reg2
100010 000 010	78695a4b3c2d 9000	f11223fa677890 00	_00000000000 0ffff	Compare signed 16-bit eq reg0 and reg2
100011 011 111	78695a4b3c2d 1e0f	78695a4b3c2d1e 01	ffffffff0000000 0	Compare signed 32-bit eq reg3 and reg7
010011 010 100	fedcba098765 43210	8899aabbccddee ff	8898aa984454 2210	AND reg2 and reg4

010100 100 101	8899aabbccdd eeff	_0011223344556 77	ffeeddccbbaa9 988	NAND reg4 and reg5
010101 001 110	_11112222333 3444	56781234dcba98 76	57793236ffbb dc76	OR reg1 and reg6
010110 010 011	8898aa984454 2210	0123456789abcd ef	89bbefffcdffeff f	XOR reg2 and reg3

Conclusions

Designing and implementing the MMX ALU has been a challenging yet rewarding endeavor. Throughout the process, addressing the intricacies of arithmetic and logical operations within the MMX instruction set architecture required careful analysis, design, and optimization. Each step revealed opportunities to enhance performance and ensure the accuracy of operations tailored to multimedia and SIMD workloads.

The successful implementation of the MMX ALU demonstrates its capability to handle complex vectorized computations efficiently. With its core functionality now established, the design is versatile and can be further extended to include additional features or optimized for specific applications. Integrated into an FPGA platform, such as Digilent's Basys3, this ALU serves as the foundation for a high-performance computing system designed for modern computational challenges. Its potential for scalability and adaptability opens doors to numerous future possibilities.

Bibliography

[1] INTEL_MMX.pdf The Story of Intel MMX Technology, Albert Yu 1997

[2] press on mmx.ppt ALP courses, Emil Cebuc 2023

[3] Documentation_example Hazard detection and avoidance unit, Radu-Augustin Vele 2023

[4] Wikipedia-MMX(Instruction Set) Wikipedia MMX(instruction set) 2024

[5] GeeksForGeeks What is MMX(MultiMedia Extension), hemangshmi8o 08 may 2024

[6] Oracle Oracle x86 Assembly Language Reference Manual, 1995

[7] Wikibooks x86 Assembly/MMX, 2024
[8] Pentium Pentium(original) 2024

[9] Intel_announcement Intel Introduces The Pentium Processor With MMX Technology, Intel Corporation 08 january1997

[10] SCS courses , SCS courses, Sebestyen-Pal Gheorghe, 2024

[11] MMX Technology Architecture Overview, Milind Mittal, Alex Peleg, Uri Weiser 1997

[12] MMX Microarchitecture of Pentium Processors With MMX Technology and Pentium II Microprocessors, Michael Kagan, Simcha Gochman, Doron Orenstien, Derrick Lin 1997

[13] Implementation of a High Quality Dolby Digital Decoder Using MMX Technology, James C. Abel, Michael A. Julier 1997

[14] Incorporating Intel MMX technology into a Java JIT compiler, Milind Girkar, Mohammad R. Haghighat, Aart J. C. Bik

[15] Art of Assembly Language 2nd edition, Randall Hyde 2010

[16] SCS laboratory works, Sebestyen-Pal Gheorghe 2024

[17] Comparative Analysis of different Algorithm for Design of High-Speed Multiplier Accumulator Unit (MAC), Ravi Shankar Mishra, March 2016

[18] Stack overflow, Overflow and underflow in unsigned integers, 2020, Kaz