MMX Unit Design

-Structure of computer systems-

Name: Negru Vlad – Andrei

Date:

Computer Science, 3rd Year of study

Lab teacher: Czako Zoltan

Table of content

1. Introduction....................................................................3
2. Objectives.......................................................................5
3. Theoretical approach......................................................6
4. Analysis..........................................................................11
5. Design.............................................................................16
6. Implementation...............................................................18
7. Tutorial............................................................................23
8. Testing.............................................................................25
9. Conclusions.....................................................................30
10. Bibliography............................................................30

1.Introduction

MMX is the abbreviation from MultiMedia eXtension. It is a supplemental instruction set introduced by Intel in 1996. Most of the instructions are “single instruction, multiple data” (SIMD), meaning that single instructions work with multiple data in parallel.

The MMX technology consists of three improvements over the non-MMX Pentium microprocessor:

1. 57 new microprocessor instructions have been added that are designed to handle video, audio, and graphical data more efficiently.
2. A new process, Single Instruction Multiple Data ([SIMD](https://whatis.techtarget.com/definition/Single-Instruction-Multiple-Data-SIMD)), makes it possible for one instruction to perform the same operation on multiple data items.
3. The [memory](https://searchstorage.techtarget.com/definition/memory-card) [cache](https://searchstorage.techtarget.com/definition/cache) on the microprocessor has increased to 32 thousand bytes, meaning fewer accesses to memory that is off the microprocessor.

It operates on single 64-bit quantities, 2 32-bit quantities, 4 16-bit quantities or 8 8-bit quantities, all at once, as we can see in the Figure 1.b). The operations can be logic, arithmetic, data movement, comparisons and data packing.

A major drawback is that we can’t use MMX and FPU (Floating Point Unit) at the same time, as they use the same register space. Hence, the user is provided with 8 general-purpose registers, all 64-bits wide, as seen in Figure 1.a).

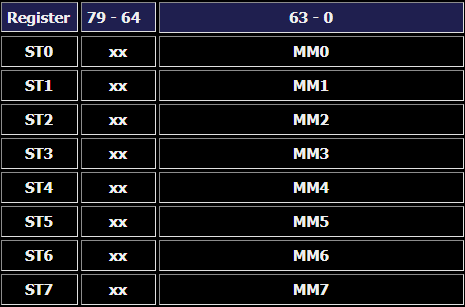


Figure 1.a) General Purpose Registers of the MMX unit.

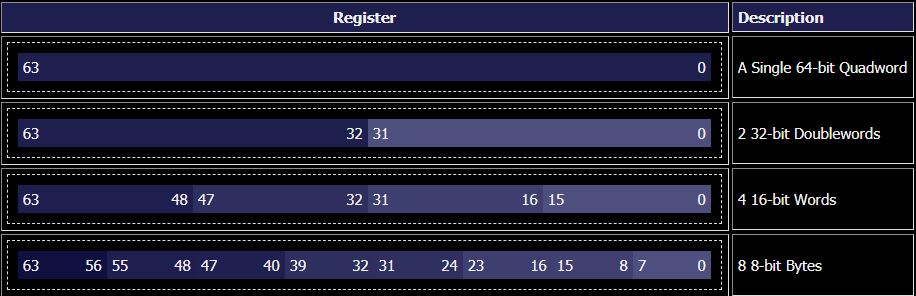


Figure 1.b) Structure of a MMX General Purpose Register

2.Objectives

The main objective is to design an MMX unit, capable of executing six arithmetic operations from the MMX x86 instruction set. This will be implemented on an FPGA board, using VHDL.

The first thing we have to define is the structure of the MMX instructions. They will be SIMD instructions, capable of computing on 32-bit data, 16-bit data and 8-bit data.

Secondly, we must choose the arithmetic instructions we want to implement.

* MOVD, which is move doubleword. Through this operation we move from the register file/immediate a 32-bit value into the MMX memory.
* POR – bitwise or
* PSUBB, PSUBW, PSUBD – subtract
* PSRLW, PSRLB, PSRLD – shift left
* PMADDWD – multiply packed bytes, add adjacent words results
* PCMPGTB, PCMPGTW, PCMPGTD – compare for greater than

Another problem we must resolve is the way we input data on our FPGA board and how it will be displayed for the user.

3.Theoretical approach

The definition of MMX technology was guided by a clear set of priorities. Priority number one was to substantially improve the performance of multimedia, communications and emerging Internet applications. Moreover, any application that has execution constructs that fit the SIMD architecture paradigm (Single Instruction, Multiple Data) can enjoy substantial performance speed-ups from the technology. Applications were broken down to reveal that, in most cases, they were built out of a few key compute-intensive routines where the application spends most of its execution time:

* Small, native data types
* Regular and recurring memory access patterns
* Localized, recurring operations performed on the data
* Compute-intensive

The key principle that allows compatibility to be maintained is that MMX technology is defined to map inside the existing IA floating-point architecture and registers.

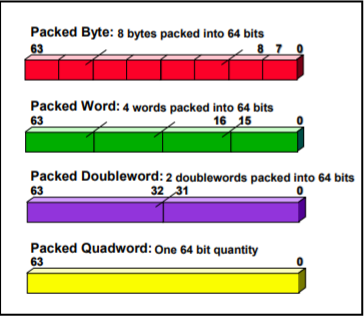
An efficient solution for media applications necessitates addressing some concepts that are fundamental to the SIMD approach and multimedia applications:

* Packed data format
* Conditional execution
* Saturating arithmetic vs wrap-around arithmetic
* Fixed-point arithmetic
* Repositioning data elements within packed data format
* Data alignment

We should next discuss about each of these concepts that define our MMX unit structure and capabilities.

3.1 Packed Data Format

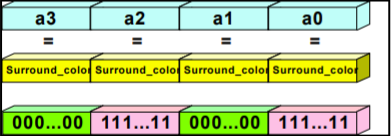
MMX technology defines new register formats for data representation. The key feature is that the typical data size of operands is small. Most of the sizes are either byte or word. Also, multimedia processing typically involves performing the same computation on a large number of adjacent elements. These two properties lend themselves to the use of SIMD computation.



3.2 Conditional Execution

One question can rise using the MMX is “What happens when a computation is only done if the operand value passes some conditional check?” There are different approaches possible and some are simpler than others. Using a branch approach does not work well.

Compare operations in MMX technology result in a bit mask corresponding to the length of the operands. For example, a compare operation operating on packed byte operands produce byte-wide masks. These masks then can be used in conjunction with logical operations to achieve conditional assignment.



3.3 Saturating arithmetic

In media applications, typically the desired behavior is to provide not wrap-around value but the maximum value as the result. MMX technology provides an option to the application program, which determines whether a wrap-around result or maximum result is provided in case of an overflow. Also, there is no flag for overflow signaling.

3.4 Fixed point arithmetic

Floating point units are hardware intensive. In fixed-point computation, from the point of view of the processor architecture, computations are done on integer values as fraction values. Some number of leading bits are interpreted as an integer, while the remaining bits of the value are interpreted as a fraction. It is application’s job to do the shifts to scale the number.

3.5 Repositioning of data elements within packed data format

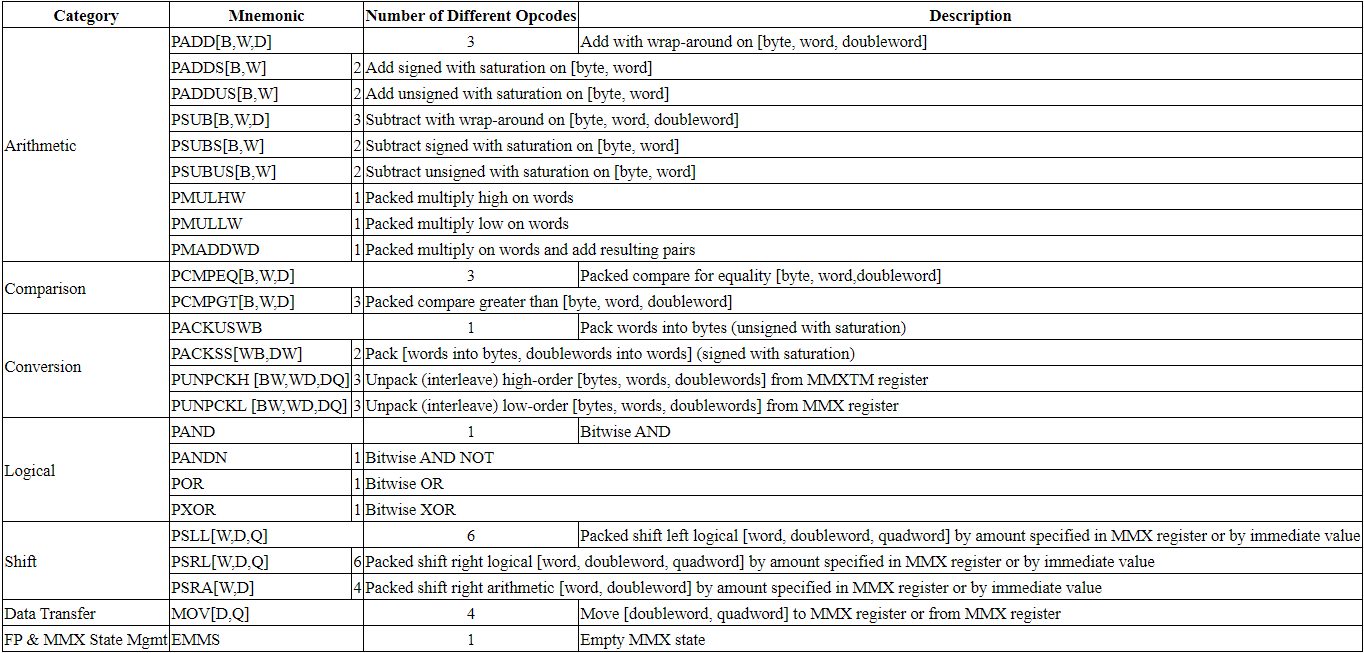
There are cases where elements of packed data may be required to be repositioned within the packed data, or the elements of two packed data operands may need to be merged.

3.6 Data Alignment

Use of packed data also presents data alignment issues. In some cases, the data may be aligned on its natural boundary and not on the size of the packed data operand. MMX technology includes logical shift-left, shift right, and or operations to assemble the desired byte from the aligned data that encompasses the desired bytes.

3.7 MMX Instructions

MMX instructions operate on the mm registers, which are 64 bits wide. They are also shared with the FPU registers.



In the next chapter we will make an analysis of the operations that we will implement in our MMX unit. The operations are enumerated in the Objective section.

4.Analysis

4.1 Data transfer instructions.

MOVD is the operation used to move a doubleword from the register file to the MMX memory, or from the MMX memory to the register file. The assembly structure of the command is:

MOVD mm, r/m32 or MOVD r/m32, mm

4.2 Logical instructions

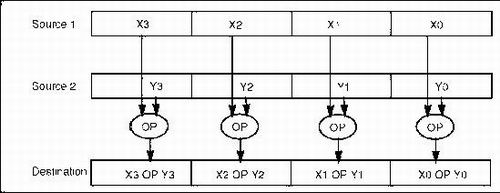
PAND, PANDN, PXOR are logical operations that don’t consider the packed data type. The operation will pe applied on each bit of the representation.

4.3 Arithmetic instructions

4.3.1 Packed Addition and subtraction

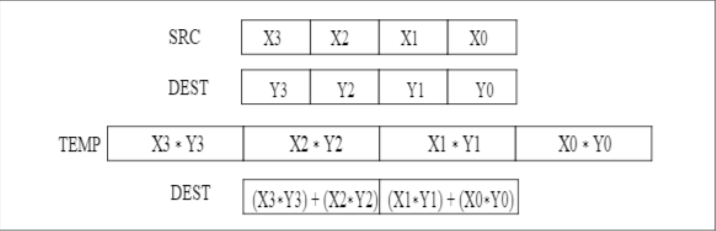
The PADDB, PADDW, PADDD are operations to add byte, word or double word integers. PSUBB, PSUBW, PSUBD are intended for subtraction of packed integers. Both work with signed numerical values. Their operands must be inside the MMX memory. We also have addition and subtraction for unsigned values, the only difference in the command is a U before the data type (B- byte, W- word, D- doubleword).

We also can have saturation add or subtract; term introduced in the chapter 3.3. Instructions PADDSB, PADDSW (packed add with saturation) and PSUBSB, PSUBSW (packed subtract with saturation) add or subtract the signed data elements and saturate the result to the limits of the signed data-type range. These instructions support packed byte and packed word data types.



4.3.2 Packed multiply and add

The PMADDWD (packed multiply and add) instruction multiply the individual signed words of the destination operand by the corresponding signed words of the source operand, producing temporary signed double-word results. The adjacent double-word results are then summed up and stored in a destination register.



4.4 Comparison operations

PCMPEQ (packed compare for equal) and PCMPGT (packed compare for greater) compare the source and destination data elements for equality or value greater than, respectively. These instructions generate a mask of ones and zeroes which are written to the destination operand. Logical operations can use the mask to select elements in order to make branch instructions. These instructions support packed byte, packed word and packed double-word data types.

4.5 Shift operations

The logical shift left, logical shift right and arithmetic shift right instructions shift each element by a specified number of bits.

5.Design

5.1 PSUBB, PSUBW, PSUBD

In order to compute the subtraction on byte, word and doubleword, we have to be able to compute the subtraction of 2 simple bits, having borrow-in and borrow-out. (similar to a full adder, but we make subtractions). If we have a 1bit subtractor, we can pair eight of these units into 1-byte subtractor. We will use 4 of these units for our operations and for different types of subtractions (byte, word, doubleword) we will decide weather the barrow is transmitted between units. For example, if we have PSUBB, we set all barrows to 0, if we use PSUBD we have to transmit all the barrow bits from one unit to the other.

5.2 POR

This is a simple operation, since it will be the same for byte, word or doubleword data types. All we have to do is include an unit that makes the or operation between 2 bits. If we use 32 of these units, we get the needed operation done.

5.3 PSRLW, PSRLB, PSRLD

Here, we have to shift left the bits of each byte, word or doubleword. In order to do this, we may have a 1 bit shifter, which displays as the result the carry-in (provided from the 1 bit shifter to the right) and propagates as carry-out the bit for the shifter to the left. This way, we can construct 1-byte shifters. We use 4 of these shifters and we manage to select the data type we want to operate on by manipulating the carry-out bits. (as in the case of the subtraction)

5.4 PCMPGTB, PCMPGTW, PCMPGTD

For making comparisons for greater than, we first need to see how we want to store the result. We will have a mask of 4 bits, where if one bit is one, it means that one operand is greater then the other. We can do this via subtraction, and we can see if the most significant bit is 0. If so, the result of the subtraction is positive, so the first operand is greater than the other. Also, we have to check that all the bits of the result are different from 0.

5.5 MOVD

We will use the MOVD instruction in order to move an immediate value coming from the user input into the MMX register. The only operation we have to make in our ALU is a passB, where B will be the immediate transmitted by the user.

5.6 PMADDWD

We will use combinational multiplication in order to compute this operation. When we have the 4x16 bit multiplication results, we have to add them 2 by 2, having in the end 2x16bit sums.



5.7 IO implementation

The user will have to be able to introduce on the FPGA board the commands to be executed by the MMX unit and the immediate for immediate operations. For the immediate, we will introduce the values from the switches. Because we have 16 switches and 32-bit data, we need 2 buttons in order to save the upper half of the immediate and the lower half of the immediate. Also, we will display data on the seven-segment display, which is limited to 16-bit data. Using the same principle, we will use a button in order to output the lower part or the higher part of the double-word on the display.

On the switches we will select the operation, the data type (byte, word or doubleword) and the addresses of MMX registers (source A, source B and destination).

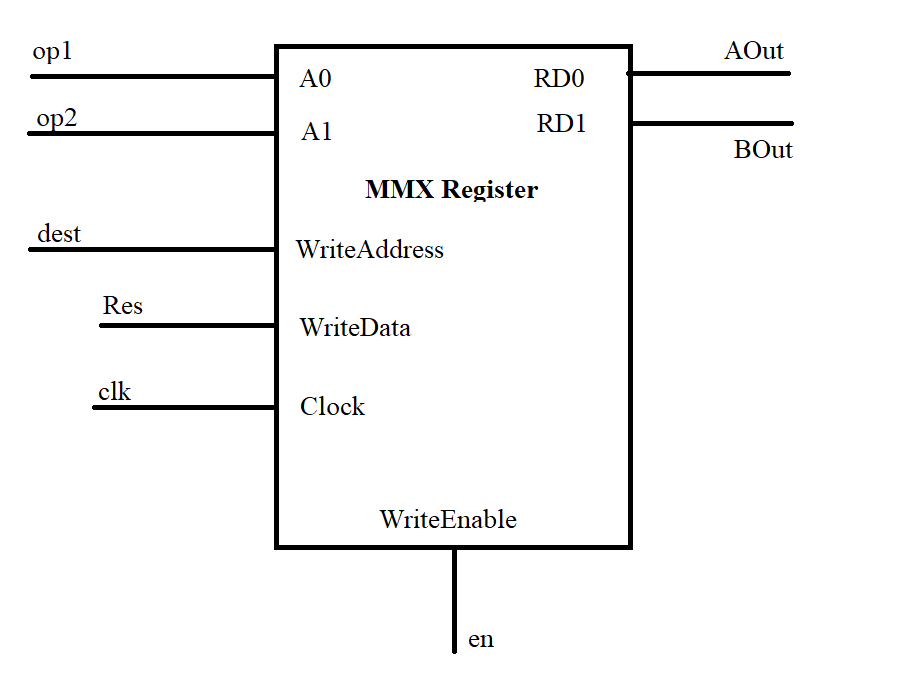
6.Implementation

The major components of our MMX unit are:

* The MMX registers
* The Arithmetic-Logic unit
* The Control Unit

6.1 MMX registers

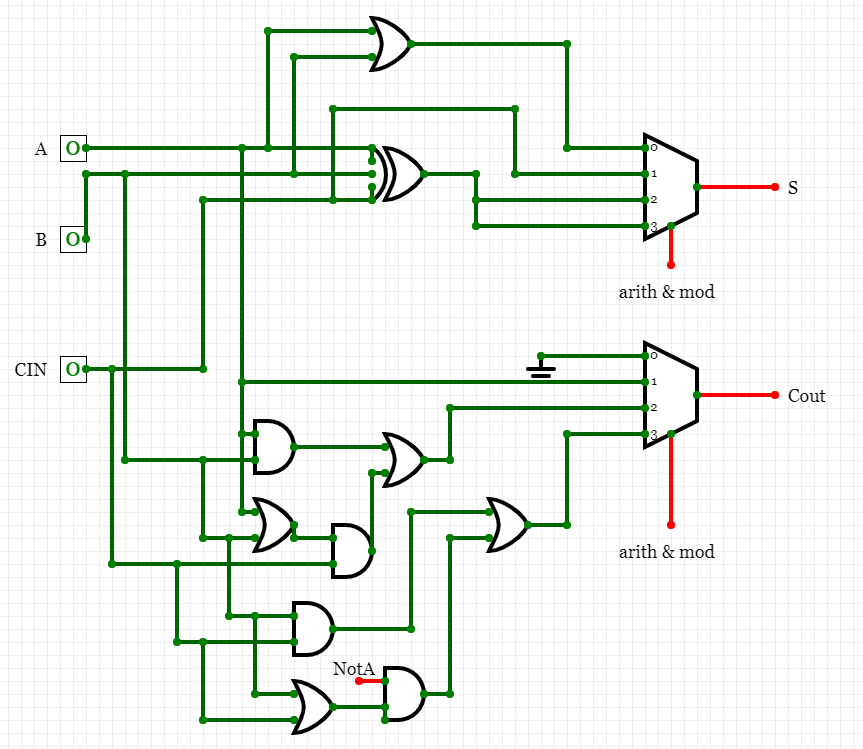
Here we store the 8 doublewords with which the MMX works. We will have an asynchronous reading from 2 addresses, A0 and A1, and a synchronous writing in the destination. The data we will write will come from the ALU and the write control signal will come from the control unit (there are operations in which we don’t have to write the result in the register, like compare greater than). The two addresses for reading the register will come from the user input, as well as the address for writing data.



6.2 ALU

The Arithmetic-Logic Unit will do the operations mentioned above on 32-bit bit data. The A operand will always come from the register, but the B operand can be provided from the register or from the user (immediate value).

We will have a 1-bit structure that will be able to make the following operations: or, shift, subtraction, addition. The operation will be selected by 2 bits, mode and arith. These will be set in the control unit accordingly. It will use carry-in and carry-out in order to be scalable to 1-byte structure. Now, we may put 4 of these units together and work with the carries between them in order to make operations on byte, word or doubleword. The type of operations will be selected by 2 bits that come from the user input.



We have the following combinations for the 1-bit unit:

* Arith and mode 00 => we have the OR instruction. The S is the result of A or B, and the COUT is set to 0.
* Arith and mode 01 => we have shift operation. Here the result will be the carry in bit coming from the right unit, and the carry out will be the A bit, to propagate the value to the left unit.
* Arith and mode 10 => we have addition.
* Arith and mode 11 => we have subtraction.

The ALU also provides a 4bit mask which holds the results of the greater than operation. It will also provide the operation of passB, for the times we need to propagate a value unchanged. This will be selected by a control bit which will be set in the control unit.

6.3 Control Unit

The purpose of this unit is to set the control bits according to the command bits given by the user. For every type of command, we will set the control bits and propagate them to the other units. With mode and arith we will select the type of operation done in the 1-bit cell of ALU. With passB we indicate if we want to have the passB operation (useful for mov command). ExtImm selects if for the second ALU source, we want the Immediate or the B result coming from the MMX register.

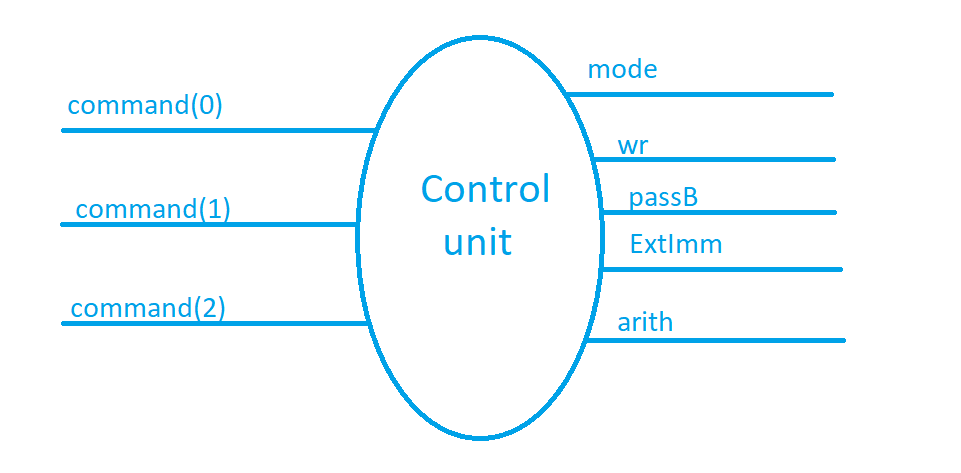
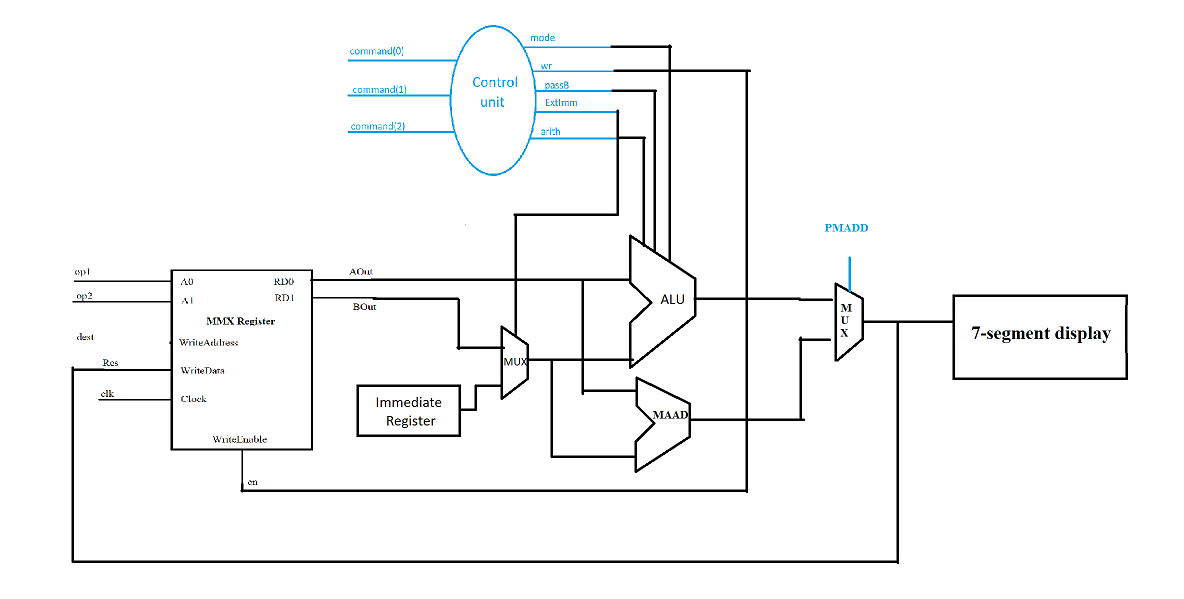
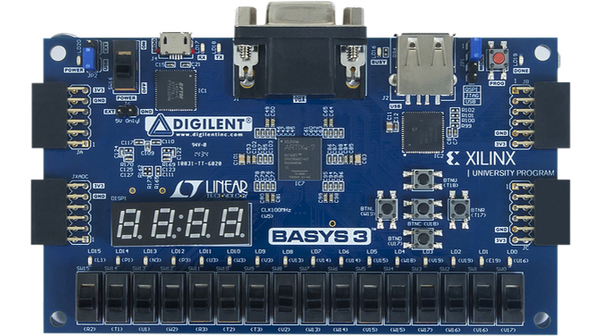


Figure 1. Control unit

The final diagram of the system: 

7. Tutorial



Write

Show Imm

Upper imm

Lower imm

Data type

command

dest

Op2

Op1

We control the operations the mmx unit will perform via switches. The switches in the green rectangle (sw9-sw7) control the destination address. Based on them, the result of the operation will be stored in a specific address in the mmx memory. The switches in the red and orange rectangles define the address in the mmx unit of the first and the second operand of the command. The next switches in the blue rectangle define the operations as follows:

* 000 => substitution
* 001 => compare greater or equal
* 010 => logical or
* 011 => left shift
* 100 => move
* 101 => multiply and add

The last two switches indicate if the operation will be done in byte, word or doubleword.

* 00 => byte
* 01 => word
* 10 => doubleword

The move instruction is a bit different since we have to put the immediate value stored in the immediate register into the mmx memory, at the addressed specified by the first three switches. So, before, we have to load the immediate register. In order to do this, we first have to set a value from the 16 switches at our disposal (in this case, we can see that most of the switches will have two roles). Afterwards, we push one button to load the upper half of the immediate register, and another to load the lower part. After we can make the move operation. The button in the yellow circle is responsible for loading the lower part, and the button in the green circle is responsible for loading the upper part.

The unit result and the immediate will be shown on the 7-segment display. In order for us to see the whole 32bit number, we need one button to change between the lower part and the upper part of the number. This job is done by the button in the red triangle. In order to see the immediate or the value of the result, we use the button in the white circle, to toggle the printing of the immediate.

The button in the brown circle has the job of enabling writing in the mmx memory. Only when it is pressed, the result will be stored.

The LEDs in the brown rectangle are the flags coming from the compare greater or equal operation.

8.Testing

For the testing part, first we will test if all the operations will show the appropriate result for the MMX registers addresses 7 and 6. The operands will be x”00000011” and x”00000022”. The tested operations will be:

* Substitution on byte level

A circuit board

Description automatically generated

The command switches are set on “000” which results on a substitution on a byte level.

A circuit board

Description automatically generated

Here we have the word substitution.

A circuit board

Description automatically generated

On doubleword substitution, the first 16 bits are set as in the word substitution and the last 16 bits become “FFFF”.

* Shift left

It is applied on the first operand, and the result becomes “0022”

A circuit board

Description automatically generated

* Logical OR

A circuit board

Description automatically generated

We applie OR on the operands and the result does not depend on the type of data we choose

* PMAD

Pack Multiply and ADD: we change the operand, operand 1 becoming “12345678” and operand 2 “00000F23”, the result displayed is the higher part “0000” , and the lower part displayed is “1572”

A circuit board

Description automatically generatedA circuit board

Description automatically generated

* MOVD

First we have to update the immediate, we set the switches to the value we want “3333” and update the lower and higher Immediate by pressing the buttons.

A circuit board

Description automatically generated

Then we select the destination (address 1) and the command switches and we write the value of the Immediate in the MMX memory by pressing the Enable button.

A circuit board

Description automatically generated

Now, for showing that the value was stored we compute a substitution with 0.

A circuit board

Description automatically generated

9. Conclusions

Using Single Instruction, Multiple Data type operations is an efficient way to parallelize computations of a processor. The fact that data is organized in packs can be either an advantage or a disadvantage. SIMD is particularly applicable to common tasks such as adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions to improve the performance of multimedia use.

10. Bibliography

|  |
| --- |
| [1] M. Mittal, A.Paleg and U. Weiser, “Carnegie Mellon School of Computer Science,” 1997. [Online]. Available: https://www.cs.cmu.edu/~barbic/cs-740/archite.pdf. [Accessed 2019]. |

[2] “SIMD”, [Online] Available: https://en.wikipedia.org/wiki/SIMD.

|  |  |  |
| --- | --- | --- |
| [3] Intel, Intel Architecture Software Developer’s Manual, Vol. 1, 1999. | | |
|  |  |