

**Design of a**

**MMX arithmetic unit**

Benedec Paula-Andreea

Group: 30433

Discipline: Structure of Computer Systems

Date: 22/10/2019

*Content*

*1. Introduction*

*2.Objectives*

*3.Theoretical approach*

*4. Implementation*

*5. Tutorial*

*6. Testing*

*7. Conclusion*

*8. References*

***1. Introduction***

Intel’s MMX™ technology [1, 2] is an extension to the basic Intel Architecture (IA) designed to improve performance of multimedia and communication algorithms. The technology includes new instructions and data types, which achieve new levels of performance for these algorithms on host processors.

MMX technology exploits the parallelism inherent in many of these algorithms. Many of these algorithms exhibit the property of “fixed” computation on a large data set.[1]

VHDL ,VHSIC-HDL ~ Very High Speed Integrated Circuit Hardware Description Language is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. VHDL can also be used as a general purpose parallel programming language.

VHDL is generally used to write text models that describe a logic circuit. Such a model is processed by a synthesis program, only if it is part of the logic design. A simulation program is used to test the logic design using simulation models to represent the logic circuits that interface to the design. This collection of simulation models is commonly called a testbench.

***2.Objectives***

The purpose is to design a MMX Unit, by choosing six operations that will be implemented from the MMX x86 instruction set in VHDL.

**PADDSW & PADDUSW**: instructions add packed double word integers from the first source operand and second source operand and store the packed integer results in the destination operand with saturation. When an individual result is too large to be represented in 32 bits (overflow), the result is given a set value, the saturation.

**PCMPEQB** instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPEQW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPEQD instruction compares the corresponding signed doubleword integers in the destination and source operands.

**PSLL**: shifts a specified register left a certain number of bits, operating on words, double words or bytes in parallel.

**PMULHW**: multiplies a register with another register or memory locatioin using signed 16-bit words. It then stores the upper 16 bits of each 32-bit result.

PMULLW multiplies just like pmulhw except it stores the lower 16 bits of each 16-bit result.

**PAND**: can bitwise-and (AND) any two MMX registers, an MMX register and memory, or an MMX register and a constant.

**PXOR:** can exclusive-or (XOR) any two MMX registers, an MMX register and memory, or an MMX register and a constant.

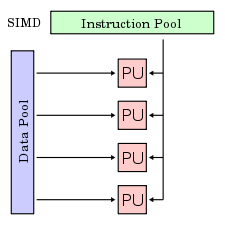
***3. Theoretical approach***

MMX is a supplemental instruction set introduced by Intel in 1996. Most of the new instructions are "single instruction, multiple data" (SIMD), meaning that single instructions work with multiple pieces of data in parallel.

**SIMD (Single Instruction, Multiple Data streams)**

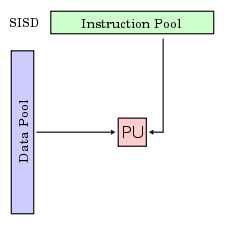
Single Instruction, Multiple Data (SIMD) is an Instruction Set Architecture that have a single control unit (CU) and more than one processing unit (PU) that operates like a von Neumann machine by executing a single instruction stream over PUs, handled through the CU. The CU generates the control signals for all of the PUs and by which executes the same operation on different data streams. The SIMD architecture, in effect, is capable of achieving data level parallelism just like with vector processor.

Some of the examples of the SIMD based systems include IBM's AltiVec and SPE for PowerPC, HP's PA-RISC Multimedia Acceleration eXtensions (MAX), Intel's MMX and iwMMXt, SSE, SSE2, SSE3 and SSSE3, AMD's 3DNow! etc.



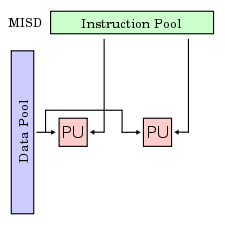
**SISD (Single Instruction, Single Data stream)**

Single Instruction, Single Data (SISD) refers to an Instruction Set Architecture in which a single processor (one CPU) executes exactly one instruction stream at a time and also fetches or stores one item of data at a time to operate on data stored in a single memory unit. Most of the CPU design, based on the von Neumann architecture, from the beginning till recent times are based on the SISD. The SISD model is a typical non-pipelined architecture with the general-purpose registers, as well as dedicated special registers such as the Program Counter (PC), the Instruction Register (IR), Memory Address Registers (MAR) and Memory Data Registers (MDR).



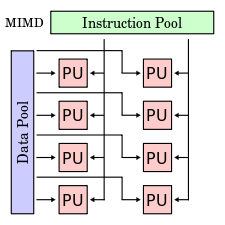
**MISD (Multiple Instruction, Single Data stream)**

Multiple Instruction, Single Data (MISD) is an Instruction Set Architecture for parallel computing where many functional units perform different operations by executing different intructions on the same data set. This type of architecture is common mainly in the fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors.



**MIMD(Multiple Instruction, Multiple Data streams)**

Multiple Instruction stream, Multiple Data stream (MIMD) is an Instruction Set Architecture for parallel computing that is typical of the computers with multiprocessors. Using the MIMD, each processor in a multiprocessor system can execute asynchronously different set of the instructions independently on the different set of data units. The MIMD based computer systems can used the shared memory in a memory pool or work using distrbuted memory across hetrogeneous network computers in a distributed environment. The MIMD architectures is primarily used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modeling, communication switches etc.[2]

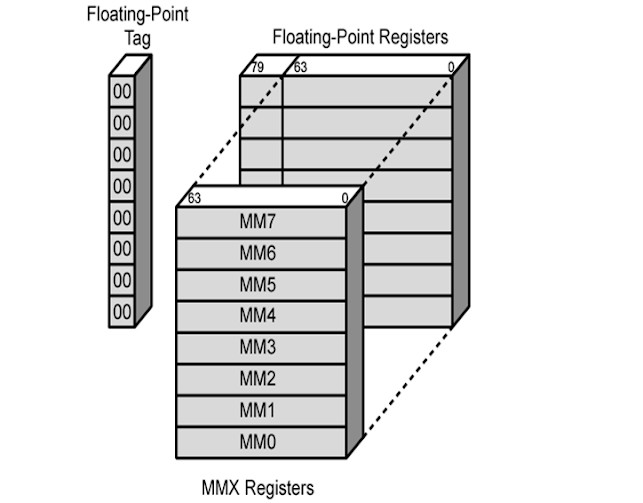


MMX technology defines new register formats for data representation. The key feature of multimedia applications is that the typical data size of operands is small. Most of the data operands’ sizes are either a byte or a word (16 bits). Also, multimedia processing typically involves performing the same computation on a large number of adjacent data elements. These two properties lend themselves to the use of SIMD computation.

Concepts that are fundamental to the SIMD approach and multimedia applications:

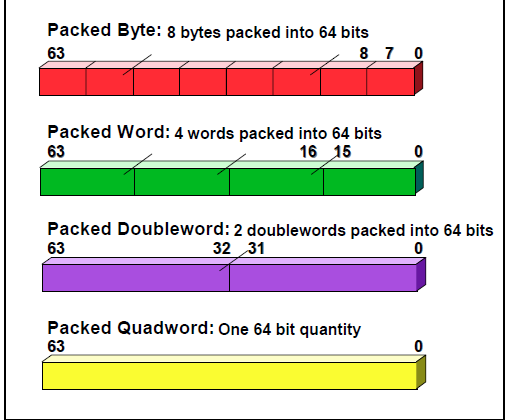
* Packed data format
* Conditional execution
* Saturating arithmetic vs. wrap-around arithmetic
* Fixed-point arithmetic
* Repositioning data elements within packed data format
* Data alignment

MMX defines eight registers, called 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: a single instruction can then be applied to two 32-bit integers, four 16-bit integers, or eight 8-bit integers at once. With the exception of emms, movd, and movq, all MMX instructions start with the letter 'p'. When any action is taken on an MMX register, it is applied to all the elements of the register at the same time. This allows software to operate up to 8 times faster (though in real life this never happens).[1]



The MMX™ technology defines the following new 64-bit *data types*:

* Packed bytes: Eight bytes packed into one 64-bit quantity.
* Packed words: Four (16-bit) words packed into one 64-bit quantity.
* Packed doublewords: Two (32-bit) doublewords packed into one 64-bit quantity.
* Quadword: One 64-bit quantity.



In an 8-bit grayscale picture, 255 is the value for pure white, and 0 is the value for pure black. In a regular register (AX, BX, CX ...) if we add one to white, we get black! This is because the regular registers "roll-over" to the next value. MMX registers get around this by a technique called "Saturation Arithmetic". In saturation arithmetic, the value of the register never rolls over to 0 again. This means that in the MMX world, we have the following equations:

255 + 100 = 255

200 + 100 = 255

0 - 100 = 0;

99 - 100 = 0;

This may seem counter-intuitive at first to people who are used to their registers rolling over, but it makes sense in some situations: if we try to make white brighter, it shouldn't become black.

The MMX registers cannot easily be used for 64 bit arithmetic. Let's say that we have 4 bytes loaded in an MMX register: 10, 25, 128, 255. We have them arranged as such: MM0: | 10 | 25 | 128 | 255 |

By adding 10 to MM0 we get: MM0: | 10+10 | 25+10 | 128+10 | 255+10 | = | 20 | 35 | 138 | 255 |

Remember that our arithmetic "saturates" in the last box, so the value doesn't go over 255. Using MMX, we are essentially performing 4 additions in the time it takes to perform 1 addition using the regular registers, using 4 times fewer instructions.

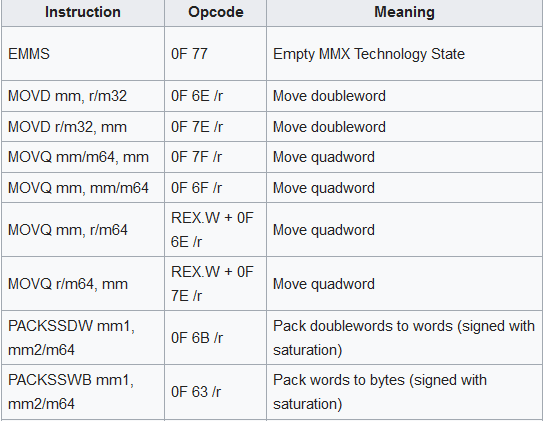
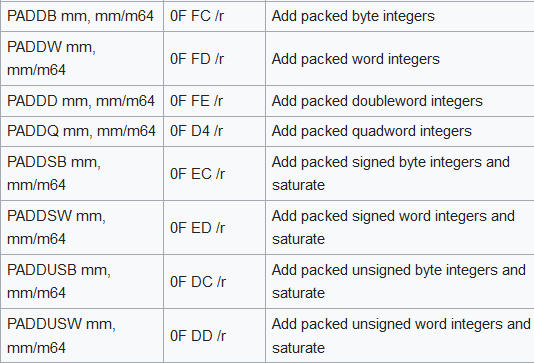
MMX has a few problems, though: instructions run slightly slower than the regular arithmetic instructions, the Floating Point Unit (FPU) can't be used when the MMX registers are in use, and MMX registers use saturation arithmetic. MMX registers are addressed directly, and do not need to be accessed by pushing and popping in the same way as the FPU registers. MMX registers correspond to the same numbered FPU registers on the FPU stack.

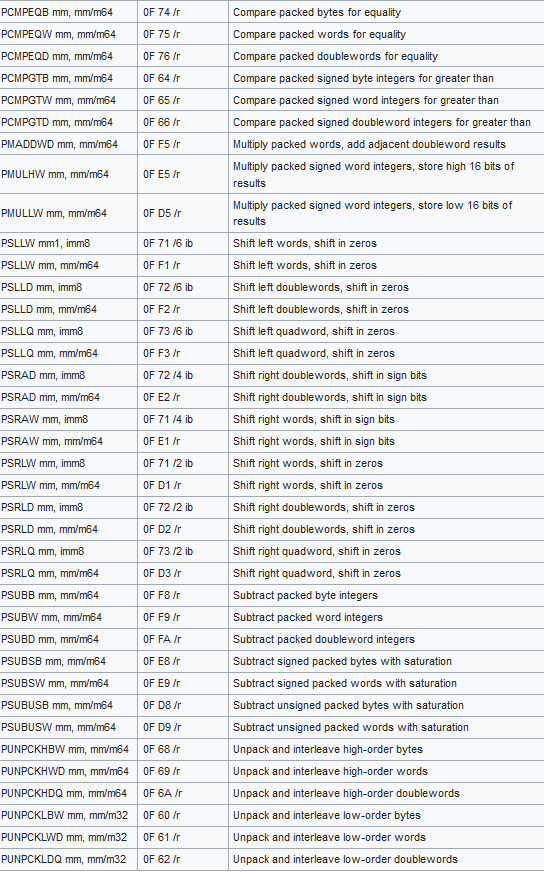
Usually when you initiate an assembly block in your code that contains MMX instructions, the CPU automatically will disallow floating point instructions. To re-allow FPU operations you must end all MMX code with emms.[3]

*MMX Instruction Set:*

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

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. [4]

******

***1. PACKED ADDITION AND SUBTRACTION***

The PADDSB, PADDSW, and PADDWD (packed add) and PSUBB, PSUBW, and PSUBD (packed subtract) instructions add or subtract the signed or unsigned data elements of the source operand to or from the destination operand in wrap- around mode. These instructions support packed byte, packed word, and packed doubleword data types.

The PADDSB and PADDSW (packed add with saturation) and PSUBSB and PSUBSW (packed subtract with saturation) instructions add or subtract the signed data elements of the source operand to or from the signed data elements of the destination operand and saturate the result to the limits of the signed data-type range. These instructions support packed byte and packed word data types.

The PADDUSB and PADDUSW (packed add unsigned with saturation) and PSUBUSB and PSUBUSW (packed subtract unsigned with saturation) instructions add or subtract the unsigned data elements of the source operand to or from the unsigned data elements of the destination operand and saturate the result to the limits of the unsigned data-type range. These instructions support packed byte and packed word data types. [3]

***2. PACKED MULTIPLICATION***

Packed multiplication instructions perform four multiplications on pairs of signed 16-bit oper- ands, producing 32-bit intermediate results. Users may choose the low-order or high-order parts of each 32-bit result.

The PMULHW (packed multiply high) and PMULLW (packed multiply low) instructions multiply the signed words of the source and destination operands and write the high-order or low-order 16 bits of each of the results to the destination operand. [3]

***3. PACKED MULTIPLY ADD***

The PMADDWD (packed multiply and add) instruction calculates the products of the signed words of the source and destination operands. The four intermediate 32-bit doubleword products are summed in pairs to produce two 32-bit doubleword results.[3]

***4. Comparison Instructions***

The PCMPEQB, PCMPEQW, and PCMPEQD (packed compare for equal) and PCMPGTB, PCMPGTW, and PCMPGTD (packed compare for greater than) instructions compare the corre- sponding data elements in the source and destination operands for equality or value greater than, respectively. These instructions generate a mask of ones or zeros which are written to the desti- nation operand. Logical operations can use the mask to select elements This can be used to implement a packed conditional move operation without a branch or a set of branch instructions. These instructions support packed byte, packed word and packed doubleword data types.[3]

***5.Conversion Instructions***

The conversion instructions convert the data elements within a packed data type. The PACKSSWB and PACKSSDW (packed with signed saturation) instruction converts signed words into signed bytes or signed doublewords into signed words, in signed saturation mode. The PACKUSWB (packed with unsigned saturation) instruction converts signed words into unsigned bytes, in unsigned saturation mode. The PUNPCKHBW, PUNPCKHWD, and PUNPCKHDQ (unpack high packed data) and PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ (unpack low packed data) instructions convert bytes to words, words to doublewords, or doublewords to quadwords.[3]

***6.Logical Instructions***

The PAND (bitwise logical AND), PANDN (bitwise logical AND NOT), POR (bitwise logical OR), and PXOR (bitwise logical exclusive OR) instructions perform bitwise logical operations on 64-bit quantities.[3]

***7.Shift Instruction***

The logical shift left, logical shift right and arithmetic shift right instructions shift each element by a specified number of bits. The logical left and right shifts also enable a 64-bit quantity (quad-word) to be shifted as one block, assisting in data type conversions and alignment operations.

The PSLLW and PSLLD (packed shift left logical) and PSRLW and PSRLD (packed shift right logical) instructions perform a logical left or right shift, and fill the empty high or low order bit positions with zeros. These instructions support packed word, packed doubleword, and quad-word data types.

The PSRAW and PSRAD (packed shift right arithmetic) instruction performs an arithmetic right shift, copying the sign bit into empty bit positions on the upper end of the operand. This instruction supports packed word and packed doubleword data types. [3]

***8.EMMS (Empty MMXState) Instruction***

The EMMS instruction empties the MMX state. This instruction must be used to clear the MMX state (empty the floating-point tag word) at the end of an MMX routine before calling other routines that can execute floating-point instructions.[3]

***9. MOVD/Q***

This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the

source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.[3]

***4.Implementation***

**PADDSW & PADDUSW:**

The addition, signed or unsigned, is made by cascading 1-bit full adders into ripple carry adders. The saturation consists in setting the right value in case of overflow.

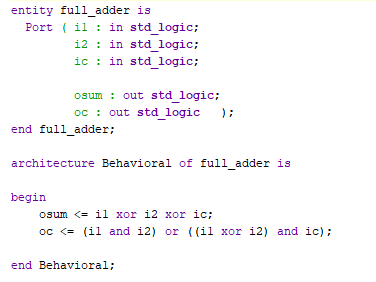


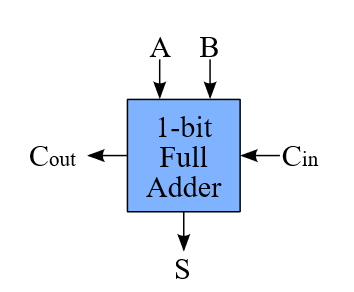


For PADDSW & PADDUSW operations, the implementation consists of a cascaded 1-bit adder into 8-bit adders. In order to operate on word data types, pairs of two adders were cascaded into 16-bit adders.

*Full Adder:*

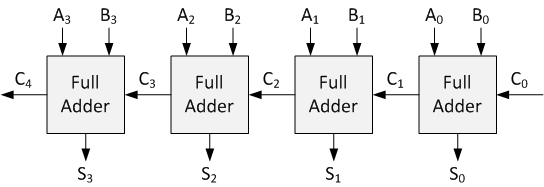
A single full-adder has two one-bit inputs, a carry-in input, a sum output, and a carry-out output. Many of them can be used together to create a ripple carry adder which can be used to add large numbers together. A single full-adder is shown in the picture below.

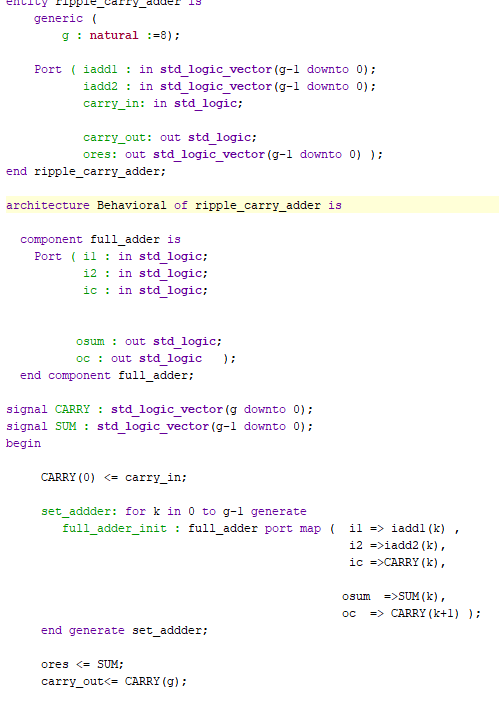




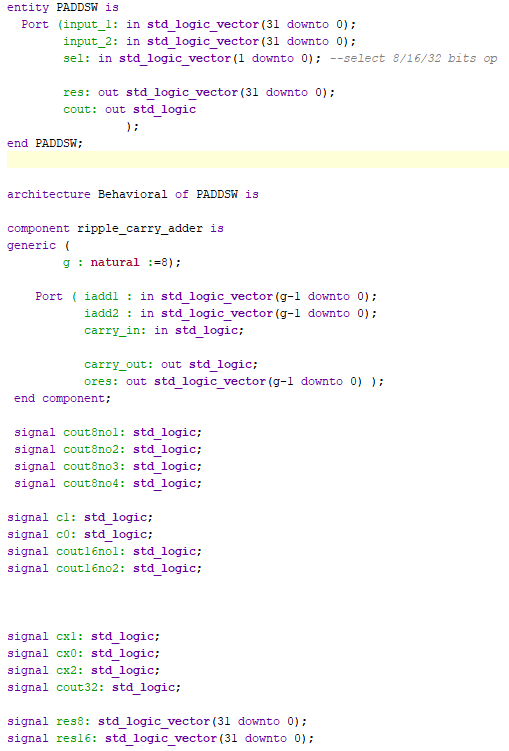
*Ripple Carry Adder*:

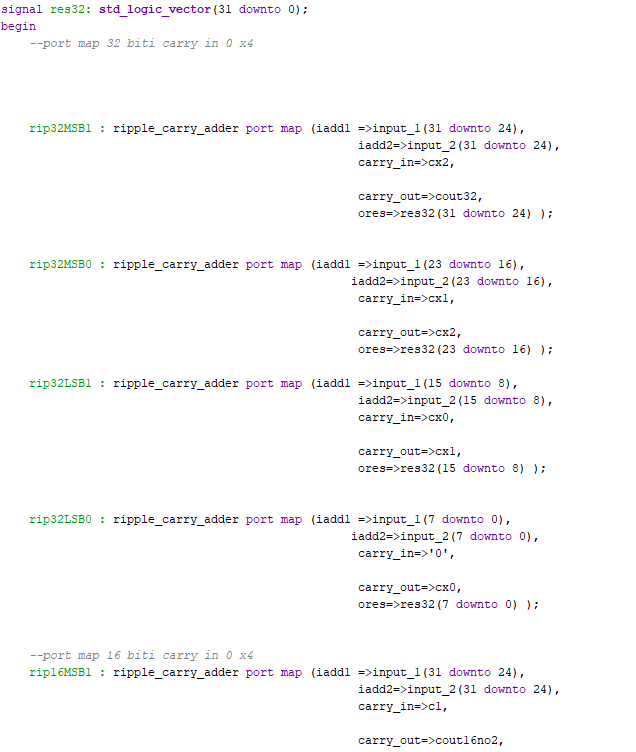
A Ripple Carry Adder is made of a number of full-adders cascaded together. It is used to add together two binary numbers using only simple logic gates. The figure below shows 4 full-adders connected together to produce a 4-bit ripple carry adder.

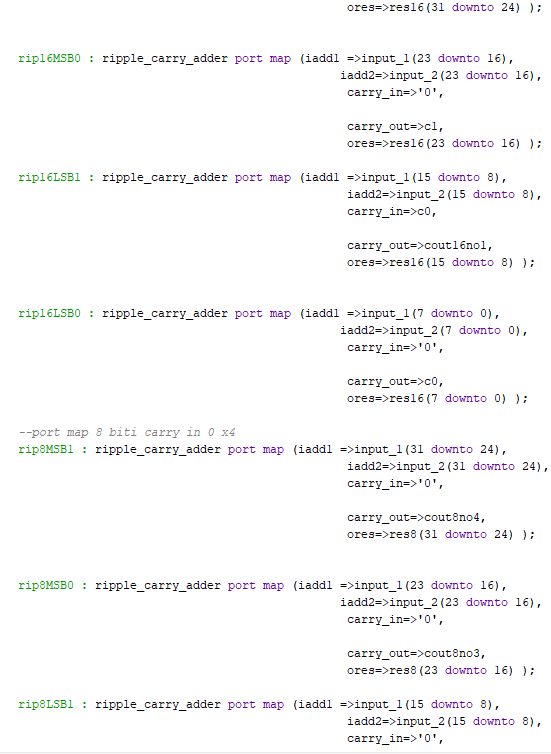


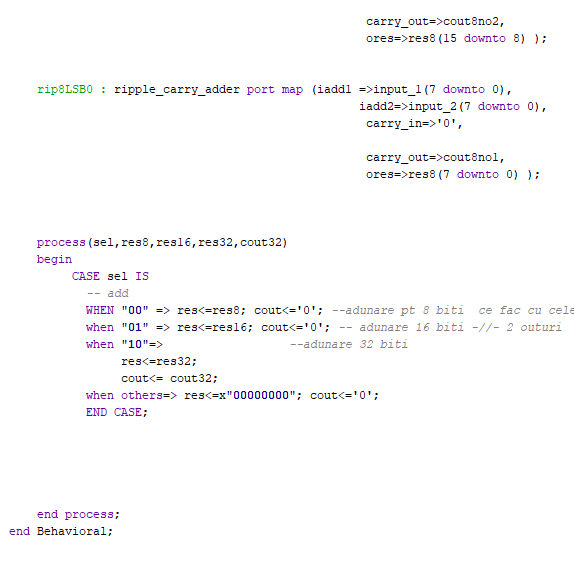


PADDSW: There a will have the 2 inputs of 32 bits, which will be added, the selection, if the result will be on 32 bits, 2 x 16 bits or 4 x 8 bits. The result is on 32 bits, and we also have a carry out.







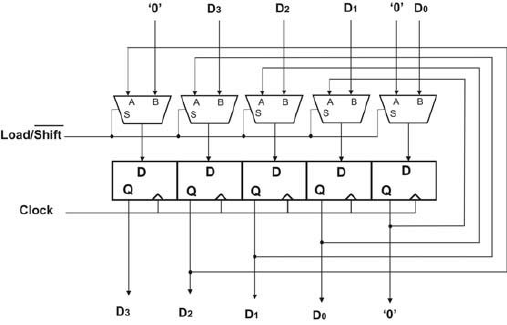


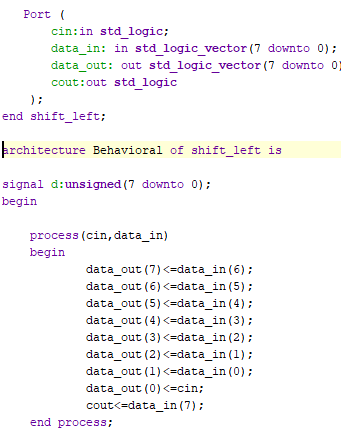
For each type of result I have 4 port maps, 12 port maps. I break the inputs and the result in 4 vectors of 8 bits 31-24 for MSB1 port maps, 23-16 for the MSB0 port maps, 15-8 for the LSB1 and 7-0 for the LSB0. For the rip8 components, the inputs are simply added, the carry in being ‘0’, in this way we have 4 result of 8 bits. For the rip16 the carry out is being transmitted from rip16LSB0 to rip16LSB1, c0, and from rip16MSB0 to rip16MSB1, c1. In this way we have 2 results of 16 bits. For the rip32, the carry out is sent as a carry in from one component to the other. The sel signal decides which result will be stored in the res.

**PSLL**:

As seen in the image below, the shift operation is made using D flip-flops, transferring information from one flip-flop to another. In addition to this, multiplexers will select if the shift will be to the right or to the left.

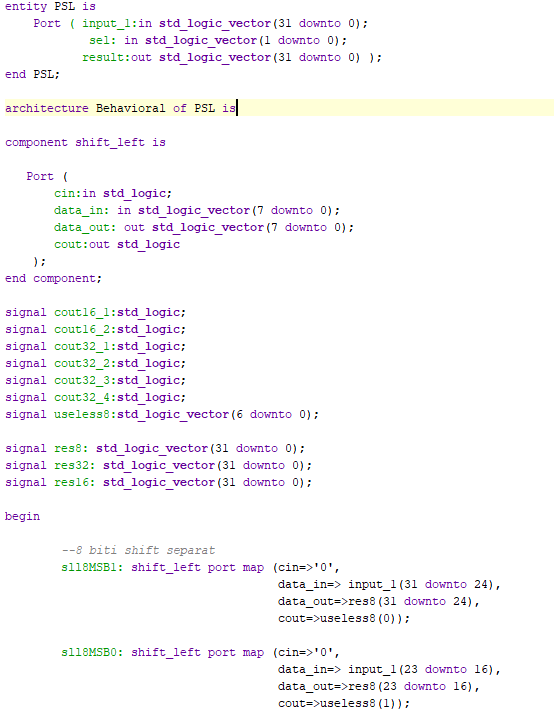
The PSLL operation does a left shift by one position by copying the first 7 bits of the input numbers into the first 7 bits of the result. The LSB of the result is set to 0.

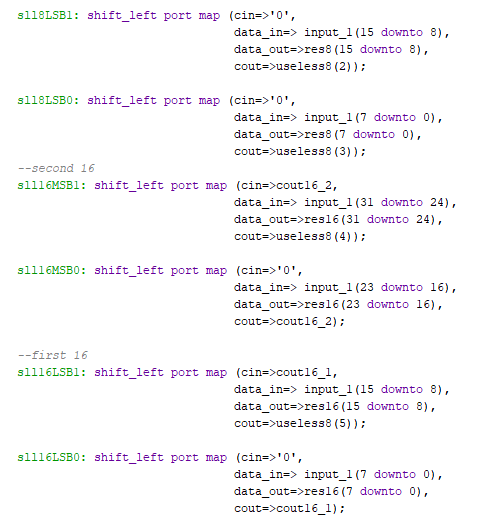




We have the carry in for the shift, cin, the input of 32 bits, data\_in, The 32 bits output, data\_out and the carry out, cout.

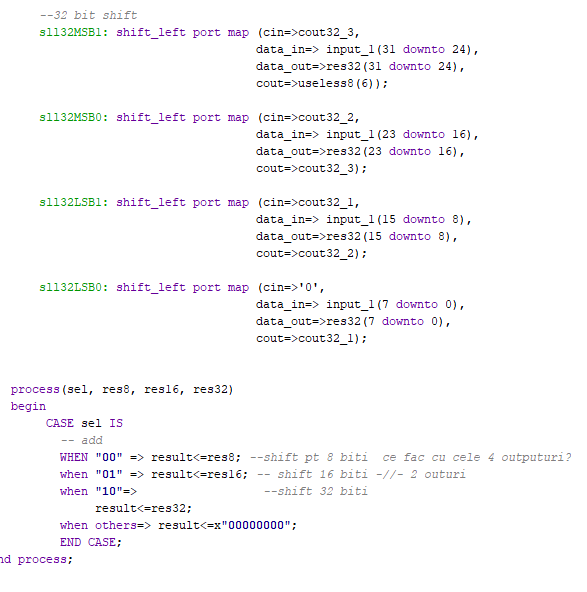
In the PSL component, the code is similar to the PADDSW. We have 12 portmaps, 4 for 8 bits, 4 for 16 bits and 4 for the 32 bits.





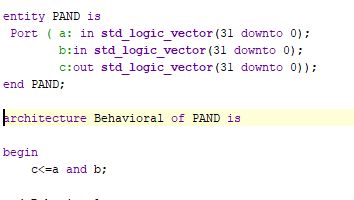
We have the 32 bits input which has to be shifted, the sel which will select of the results will be the output, and the 32 bits output, result. For the 8 bit packed result, the port maps do not send the carry out as a carry in, while for the 16bits packed, from LSB0 is sent the cout16\_1 as carry in for the LSB1 and from MSB0 is sent the cout16\_2 as carry in for the MSB1. For the 32 bits, the carry out is sent as carry in from one component to the other.

The sel signal selects the type of packed data for the result.



**PAND**:

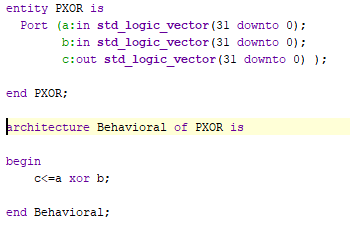
Performs a bitwise logical AND operation on the first operand and second operand and stores the answer in the result operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0. a, b are the 32 bits inputs, c is the 32 bits output.



**PXOR:**

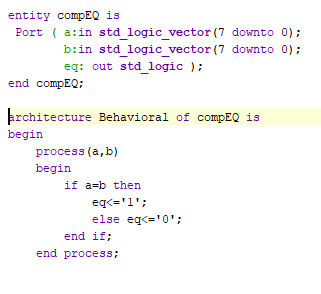
Performs a bitwise logical exclusive-OR (XOR) operation between the two operands and stores the result in the out operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

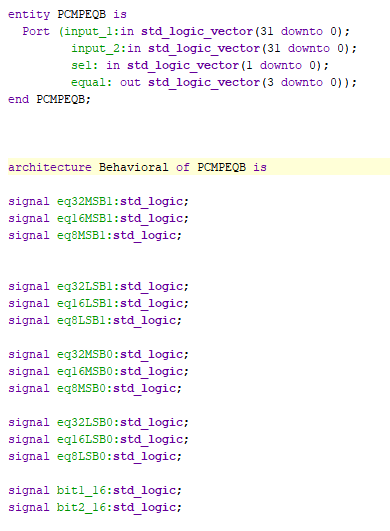
a, b are the 32 bits inputs, c is the 32 bits output.

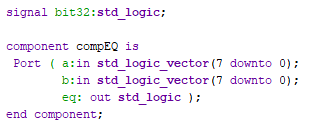


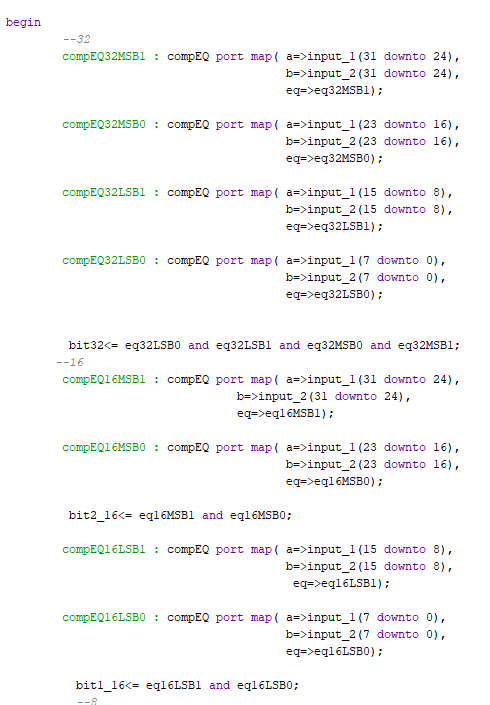
**PCMPEQB:**

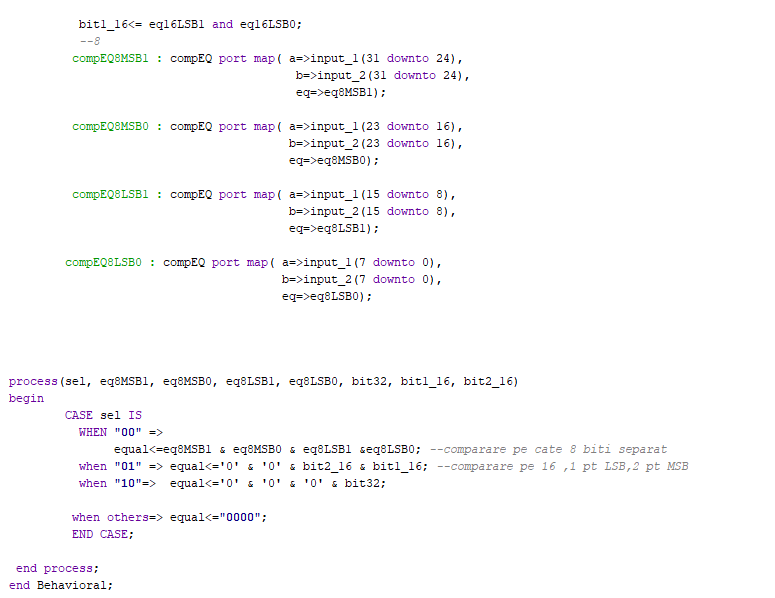
Performs compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements are equal, the result is set to 1 ; otherwise, it is set to 0.











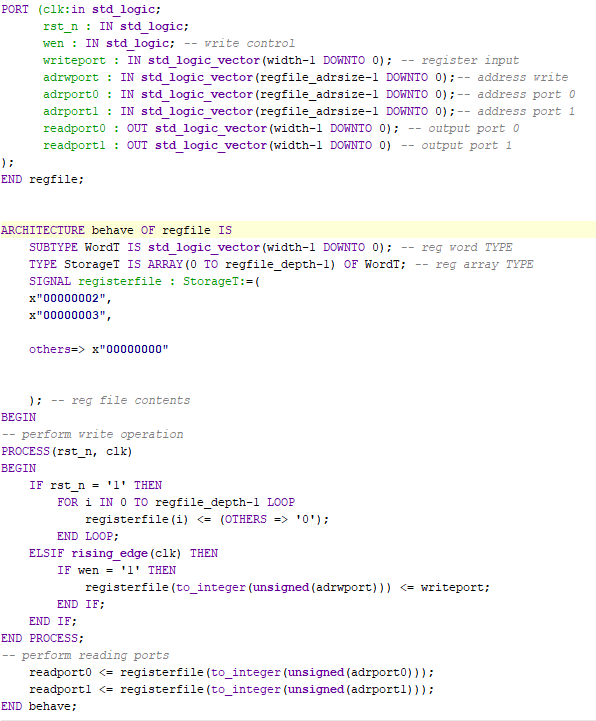
We have the 32 bits input, the sel, and the 4 bits output, equal. We compare for equality, and if the 8 bits data from the component are equal, we have 1 as output. Depending on the result we will have to do an and between the 4 outputs from the components, for the 32 bit data, or 2 and for the 16 bit data.

**PMULHUW** :

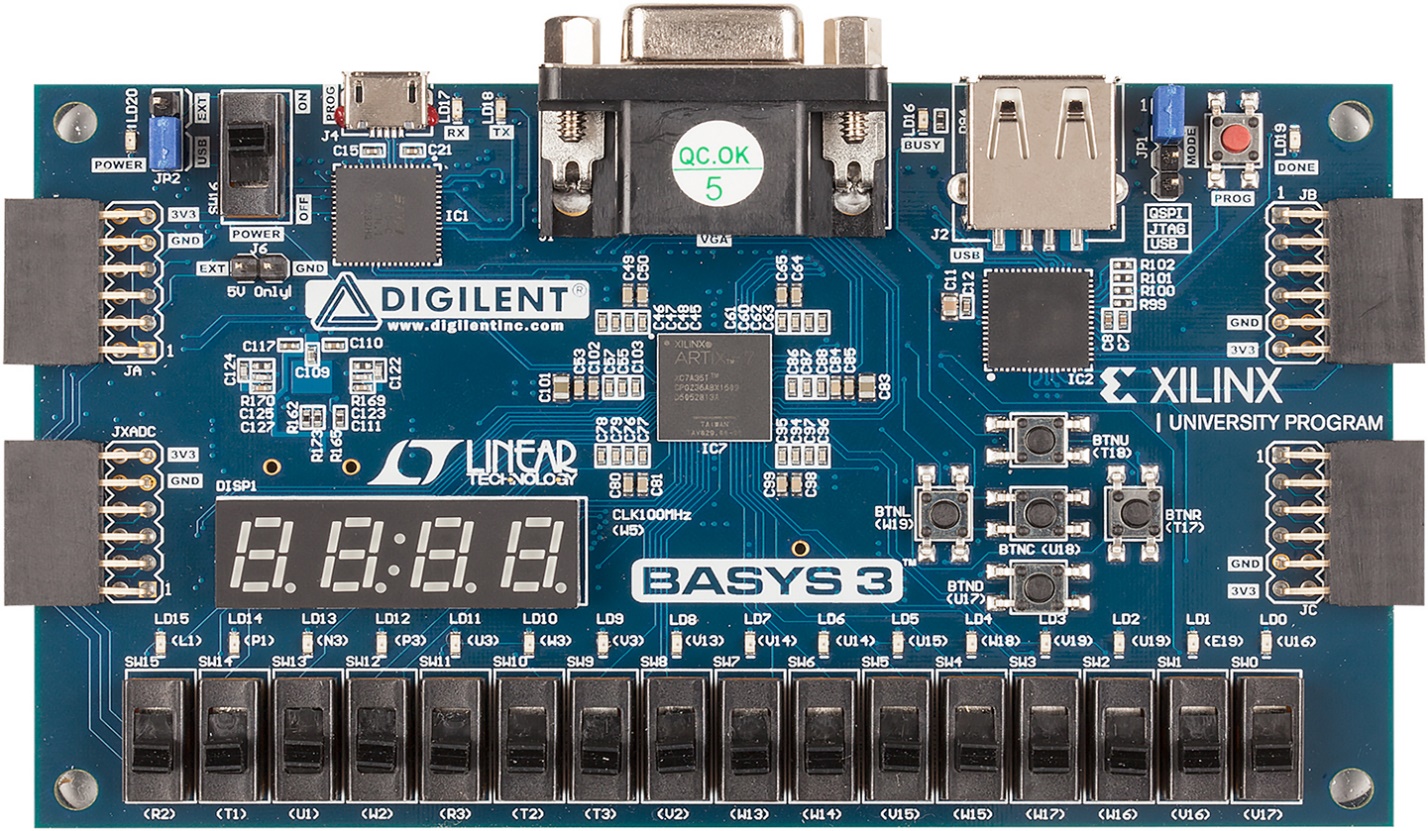
Multiply Packed Unsigned Integers and Store High Result. We convert the two operands to unsigned so we can multiply them and following this we convert the result of the multiplication to std\_logic\_vector. The inputs are of 16 bits and the output is of 32 bits.

**regFile** :

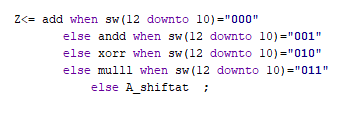
In regfile we make the MMX registers, 8 in number, 32 bits each. I stored the values 2 and 3 at address “000” and “001”.



***5. Tutorial***



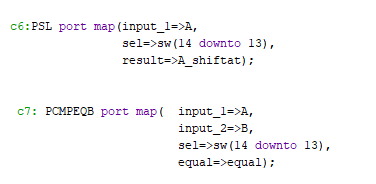
The switches 12 to 10 select the result to be displayed.



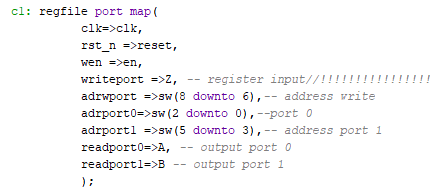
The switches 14 to 13 select the kind of packed type: 8bts, 16 bits or 32 bits.



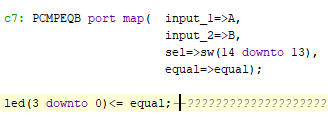
 The led 15 is for the cout



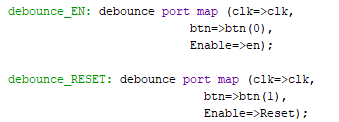
The switches from 2 to 0, 5 to 3 give the addresses of the operands in the MMX registers. The switches 8 to 6 give the address where the result of the operation will be stored in the MMX registers. To the address “000” it is stored 2 and to the address “001” is stored 3. This are the values on which I will perform the operations.



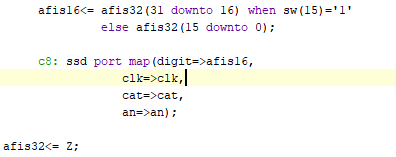
The leds 3 to 0 show the equality of the operands.



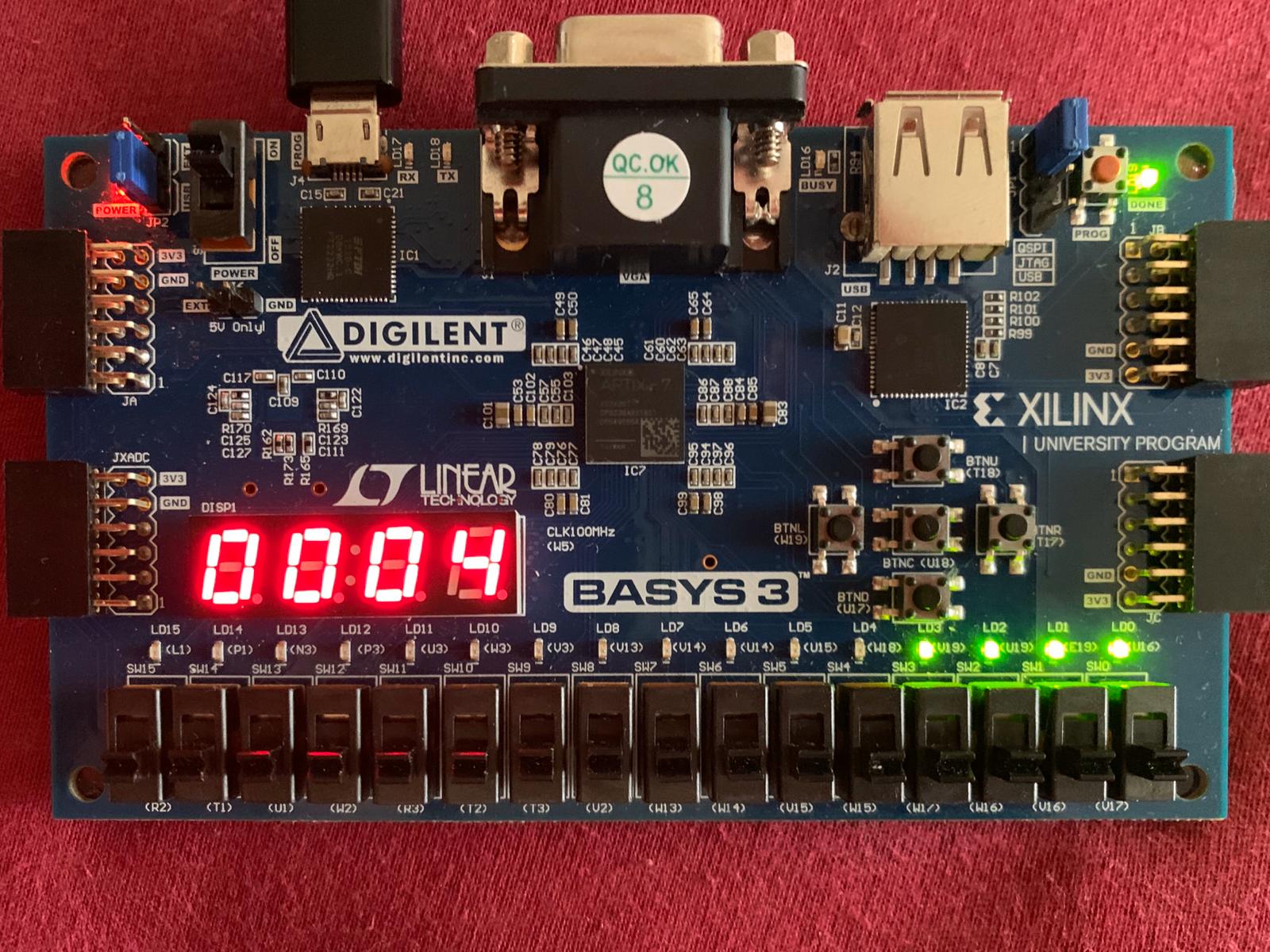
We have an Enable button and a Reset one, the one from the middle is the En and the one that is up is the Reset.



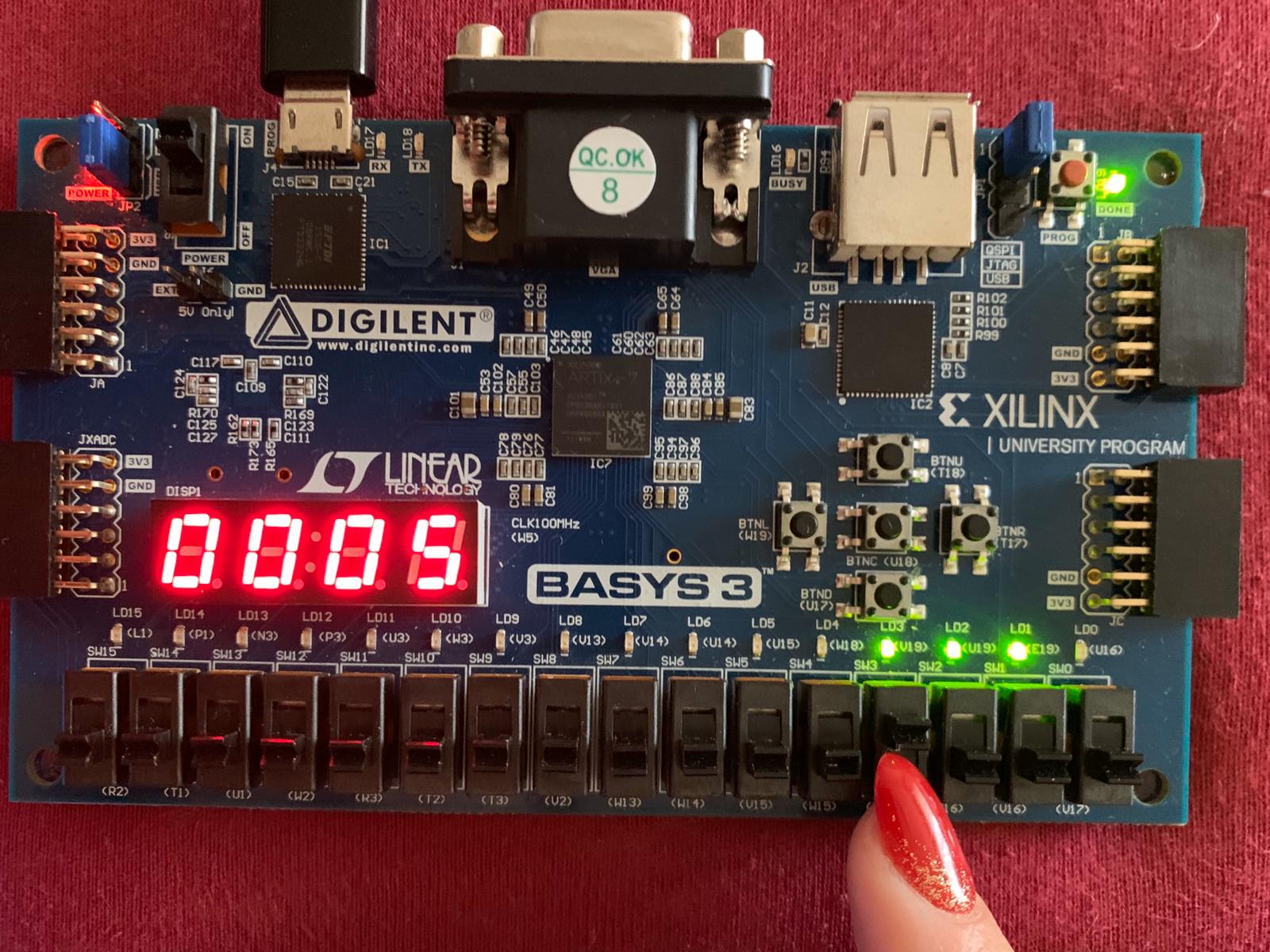
The switch 15 selects the lower or the higher part of the 32 bit result that will be displayed.



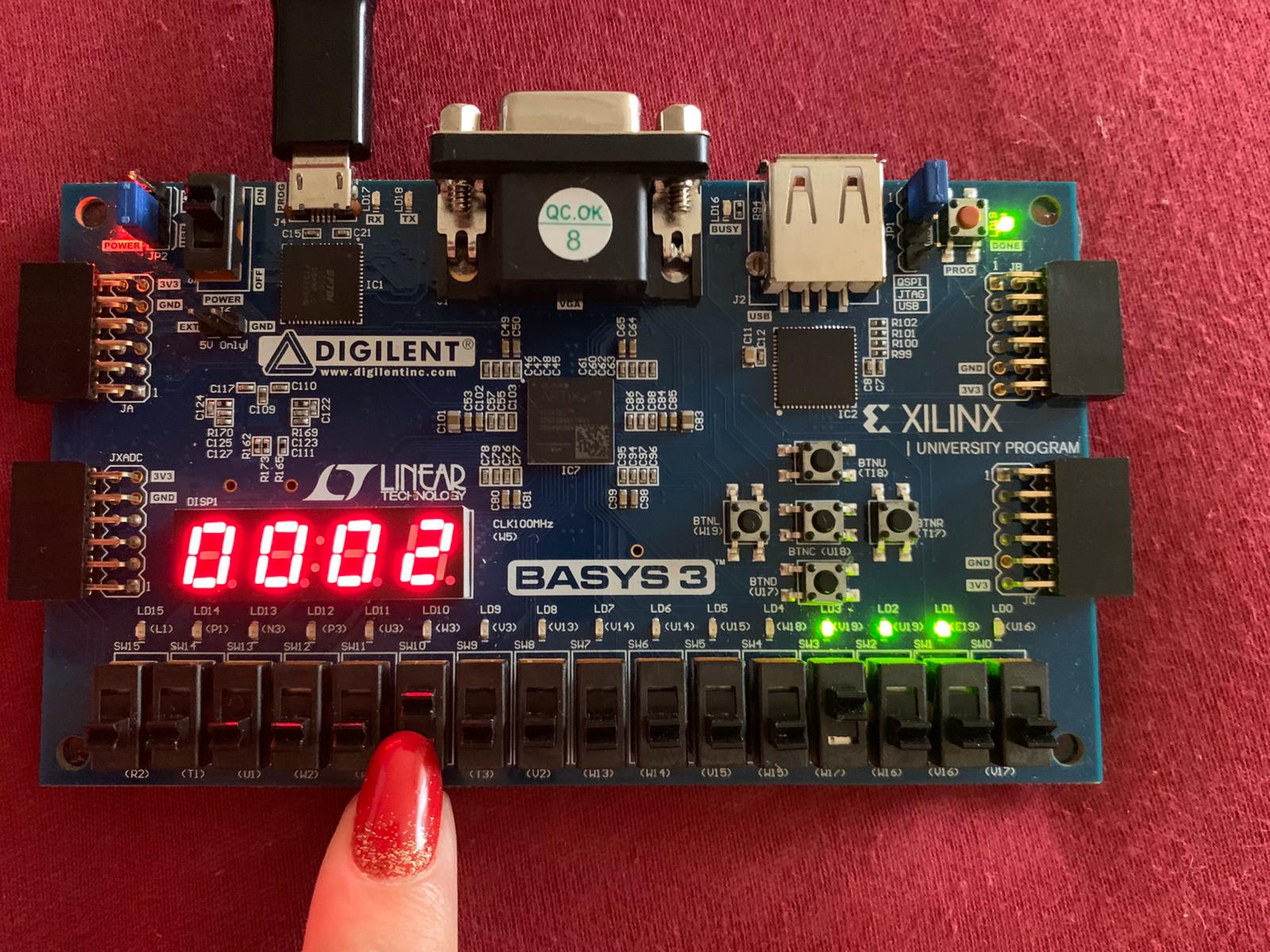
***6. Testing***



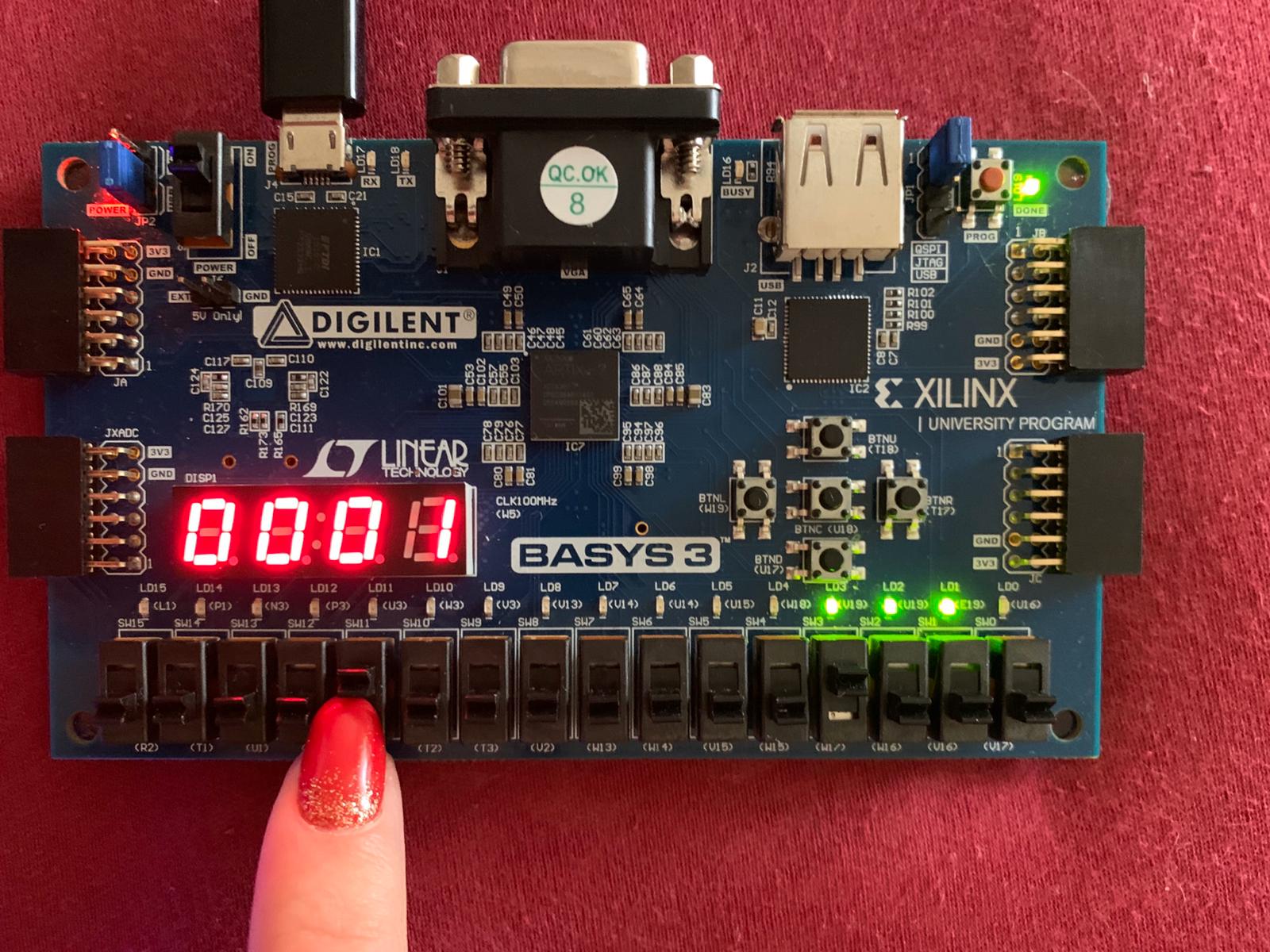
Just as you plug the board to the laptop this is what is displayed. As all the switches are ‘0’ that means that the address for the registers is “000” then the operands will have the values 2 and 2. The switches for operations are “000” so it means that add will be performed so on the display we will hava a 4. The leds 3 down to 0 are on because 2 is equal with 2.



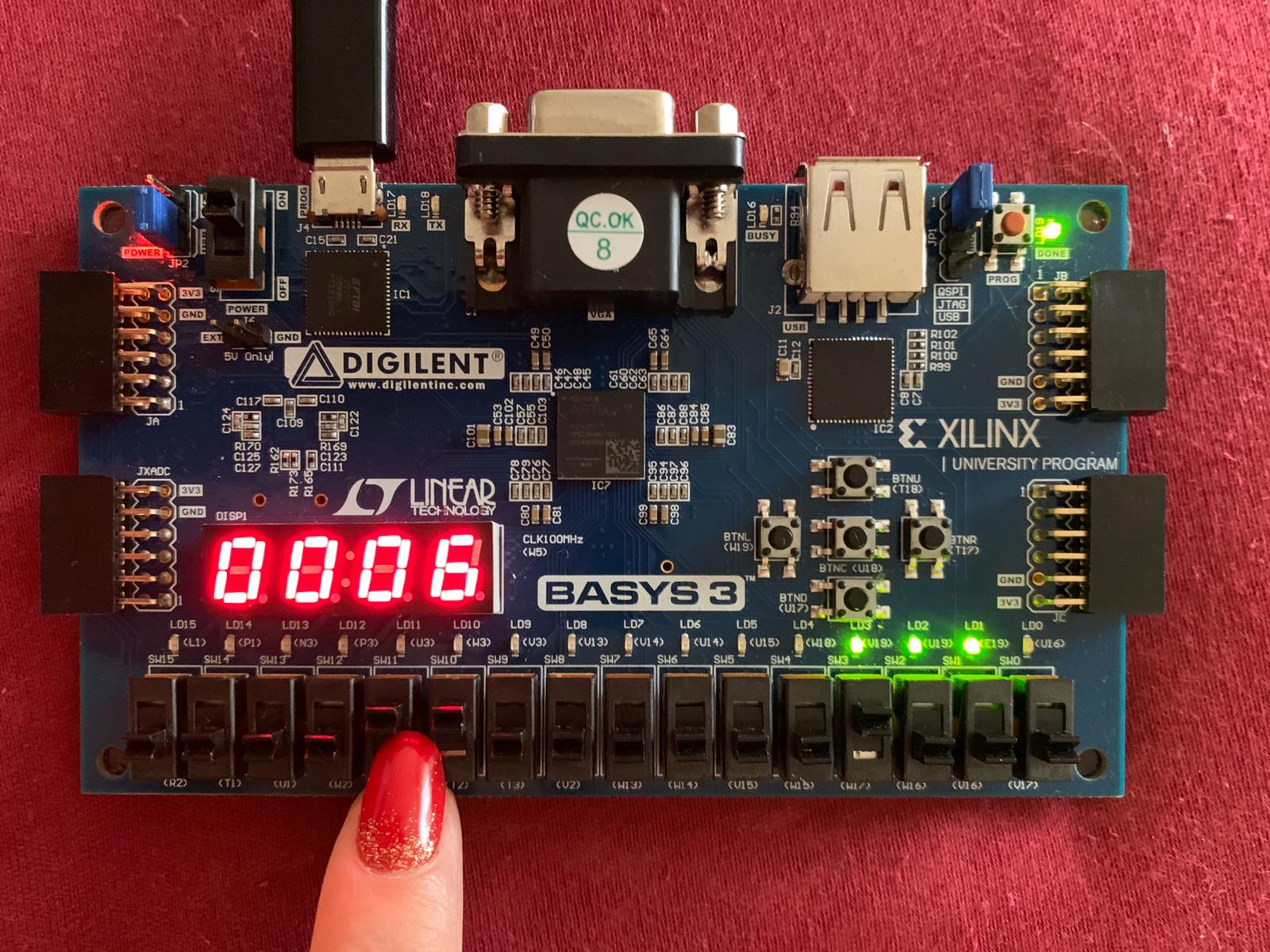
I set the switch 4 to 1 meaning that the second address is “001” so the value of the second operand will be 3. The switches for operations are still “000” so it means that ADD will be performed so on the display we will hava a 5.



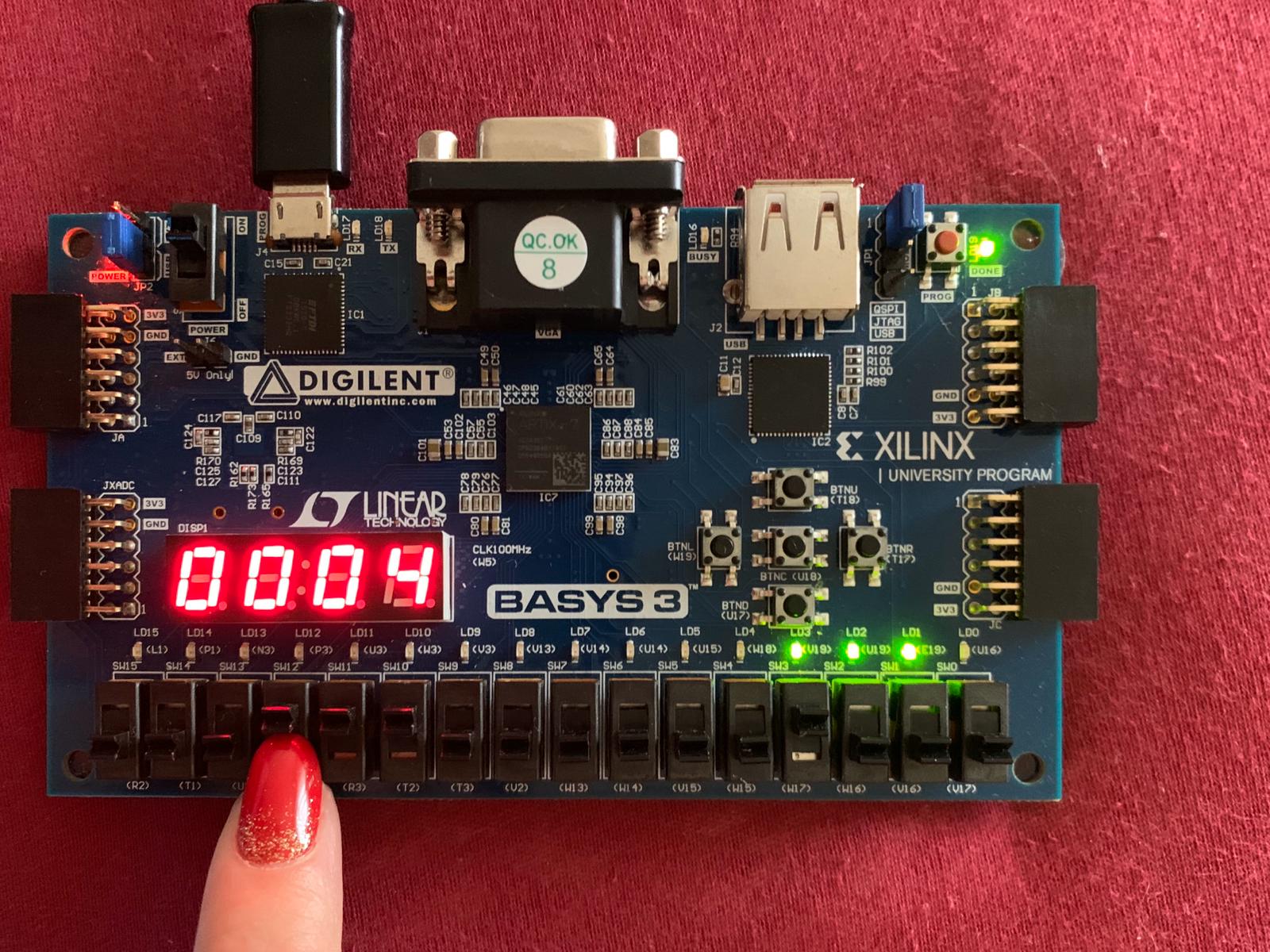
Here the switch 10 is set to 1 => “001” this means the it will do AND, displaying 2.

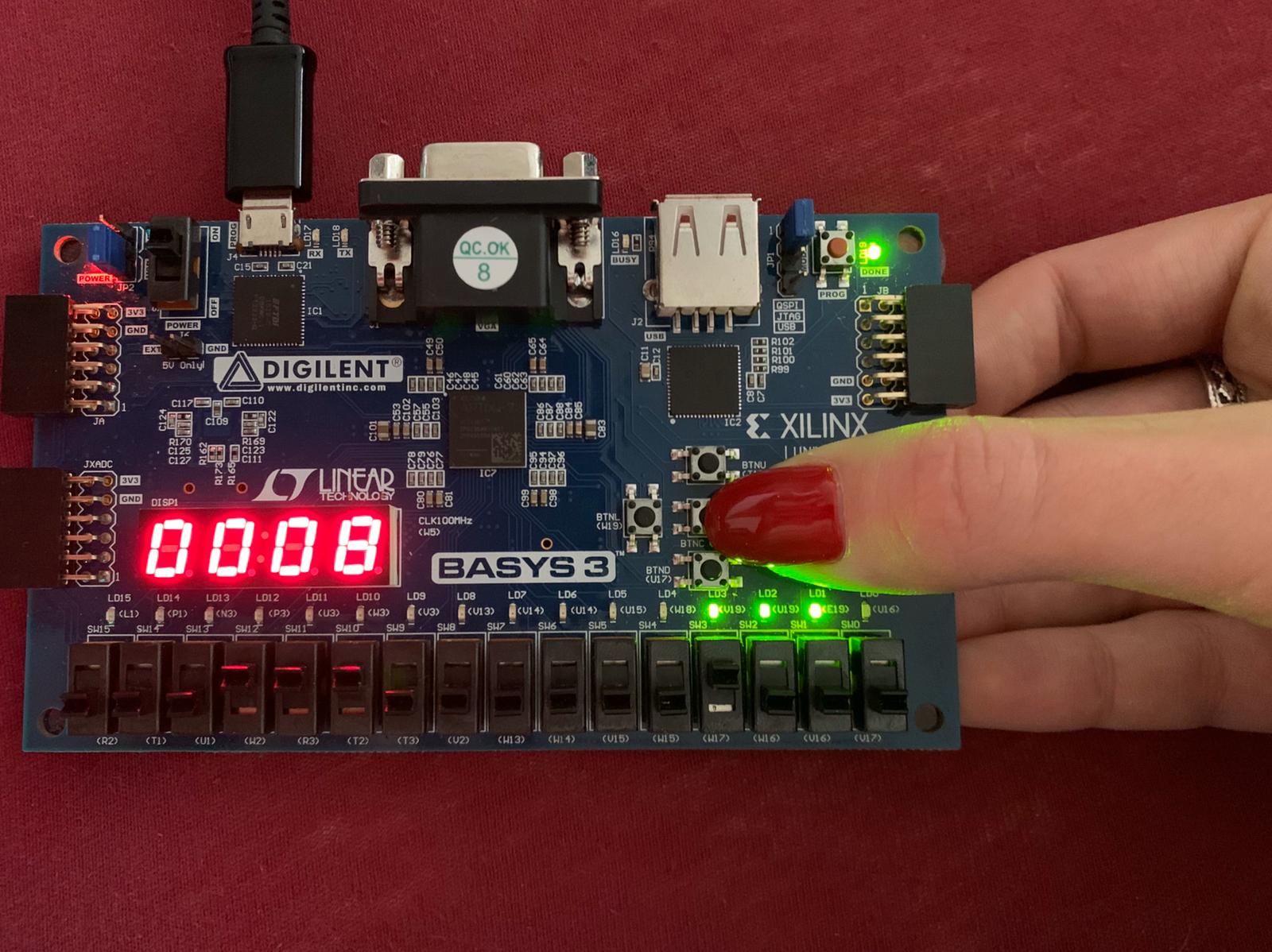


Here the switch 11 is set to 1 => “010” this means the it will do XOR, displaying 1.



Switches 10 and 11 are set to ‘1’ => “011” so there will be done a multiplication resulting in a 6 on the display.



Here switches 10,11 and 12 are all set to 1=> “111” so there will be performed a shift left on the operand with the value 2 so on the display there will be a 4.

By pressing the enable button the value of the result will be stored in the register and the switches 10,11 and 12 are still set to one so on the display is the value 4 shifted => 8.

After this I set the switches for the operation to “000” => ADD and the display will have 7 (4 + 3).

***7. Conclusion***

MMX is a processor supplementary capability that is supported on recent IA-32 processors by Intel and other vendors. Single Instruction, Multiple Data (SIMD) is an efficient Instruction Set Architecture that have a single control unit (CU) and more than one processing unit (PU). Data being arranged in packs is a distinctive feature of the MMX.

***8. References***

[1] M. Mittal, A. Peleg 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] https://www.coursehero.com/file/45514595/Parallel-Processing-Notedocx/ [3] Intel, Intel Architecture Software Developer’s Manual, Vol. 1, 1999.

[4] “x86 instruction listings,” [Online]. Available: https://en.wikipedia.org/wiki/X86\_instruction\_listings#MMX\_instructions..