Princess Sumaya University for Technology

King Abdullah II Faculty of Engineering

Computer Engineering Department



Computer Architecture 2

Final Report

Pipelined Hardware Elliptic Curve Point Multiplication

*Authors: Supervised by:*

Bader Unsal 20200996 Dr. Awos Kan`an.

Omar Abu Khalaf 20200321

**Abstract:**

One of the core components of elliptic curve cryptography, which is currently a frequently used encryption protocol because of its effectiveness and security, is point multiplication. A point on the elliptic curve is multiplied by a scalar value to produce another point on the curve in the process of point multiplication. It can be applied in a variety of ways, including the traditional double-and-add algorithm.

**Introduction:**

We have designed pipelined hardware architecture that performs ECC (Elliptic Curve Cryptography) Point Multiplication. This will enhance the efficiency of computing multiplications on the curve. We have divided the whole process over 4 stages, where each stage will perform specific operations and passes the output result to the next stage immediately. This overlapping of operations will lead to an increased throughput and reduced latency

**Proposed Design:**

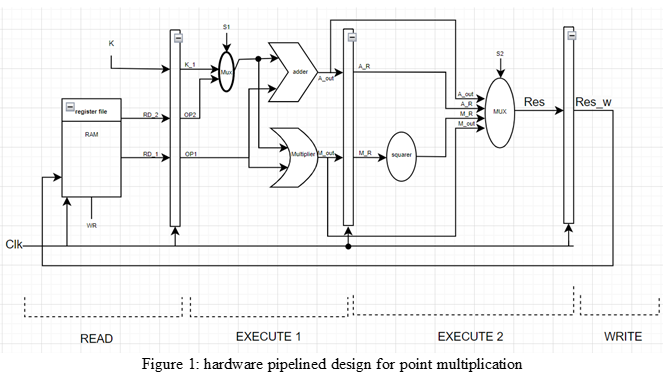
Stage 1: The first stage will consist of a RAM which will be filled with point coordinates on the elliptic curve. Each coordinate (X and Y) is located in consecutive memory locations. In the first stage 2 points will be read and stored in the buffer in order to use them in the 2nd stage.

Stage 2: In the 2nd stage, we have a 2x1 mux, a multiplier module, and an adder module. The multipler and adder work in parallel. The mux chooses between a constant or 1 of the points we got in stage 1. Thus the multiplier and adder perform their respective operations either between 2 points or 1 point and the constant k, depending on the output of the mux. The output of both modules are then stored in the buffer between stage 2 and 3. They can also be forwarded depending on the need.

Stage 3: This stage consists of a squarer module and a 4x1 mux. The output of the multiplier in stage 2 will enter the squarer module and go towards the mux. Control signal s2 will then decide between the 4 signals as the output will be stored in the buffer between stage 3 and 4.

Stage 4: This is the write back stage where the final result will get stored in order to get used in the next stage of Elliptic Curve Cryptography.

**Design Block Diagram:**



**Design Analysis:**

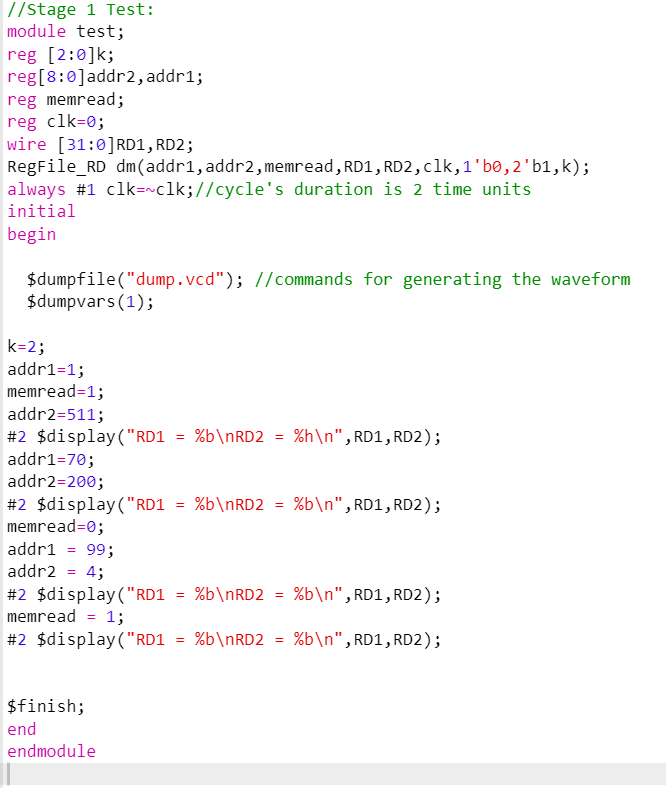
Point Multiplication is made from addition and multiplication operations. So, we needed a memory location that is filled with points. These points will be multiplied and added and squared depending on the control signals and then the final output will be stored again.

**Test Runs:**

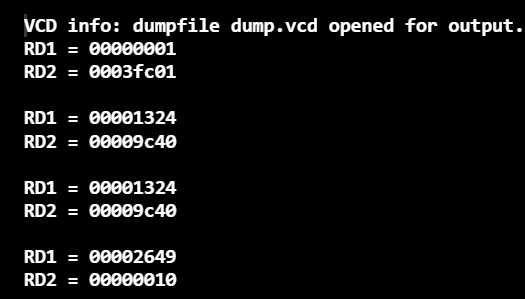
**Stage 1:**

The RAM of size 512X32 has been filled with different coordinates and in this stage, according to our design, a constant scale (k) should be sent as an input along with the addresses addr1,addr2 to perform the arithmetic operations needed in stage 2. The data stored in these addresses contains two points (x,y) that will be given as an output as shown in the test cases below to generate a third point at the elliptic curve graph. The ROM file only outputs the data when memread is equal to 1.

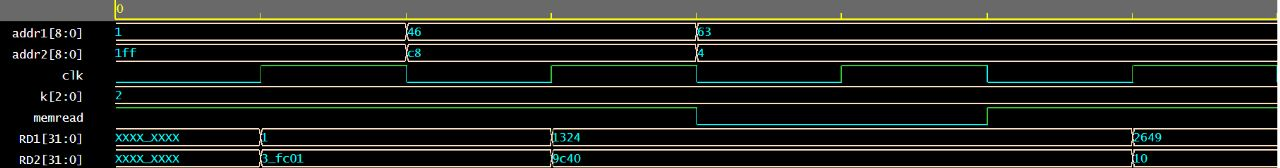
**Testbench:**



**Output:**



**Waveform:**



As shown in the waveform, the output registers RD1, RD2 in the first half cycle have undefined values before reaching the clock’s positive edge. This validates the dependency of the module on the system clock. The values in the testbench where printed in hexadecimal form to mach with the waveform values.

After that, a new value is randomly assigned to the output registers on the next two positive edges according to the function written within the loop in the attached code (RAM[i]=i\*i). At the third cycle, memread flag is cleared low for testing the module’s behavior, therefore; the values of RD1 and RD2 weren’t affected unless memread is set high.

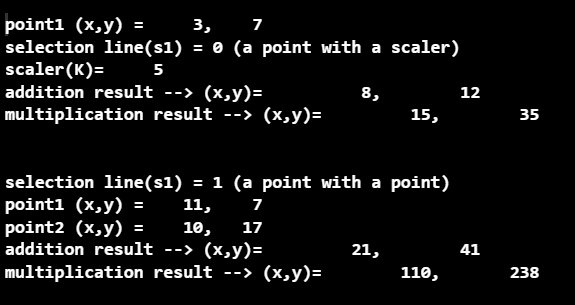
**Stage 2:**

In this stage the mux, implemented as a case statement, either chooses between a constant k or a second point coordinates and performs both addition and multiplication with the first point coordinate as seen in the test case below. The 2 inputs x and y are stored in one 64-bit reg where the lower 32 bits are the X coordinate, and the higher 32 bits are the Y coordinates. This makes it easier to perform both the addition and multiplication operations. The output shows all possible cases where either the constant k is chosen or if the second point is chosen depending on the mux selection line.

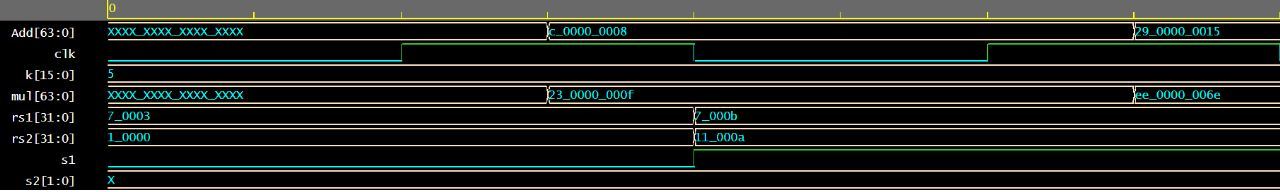
**Testbench:**



**Output:**



**Waveform:**



We have sent dummy values for the selection line (s2) and the address sent to the execute module in stage 2 only for testing its behavior, however; these parameters will have a distinct effect when implementing the fully pipelined hardware.

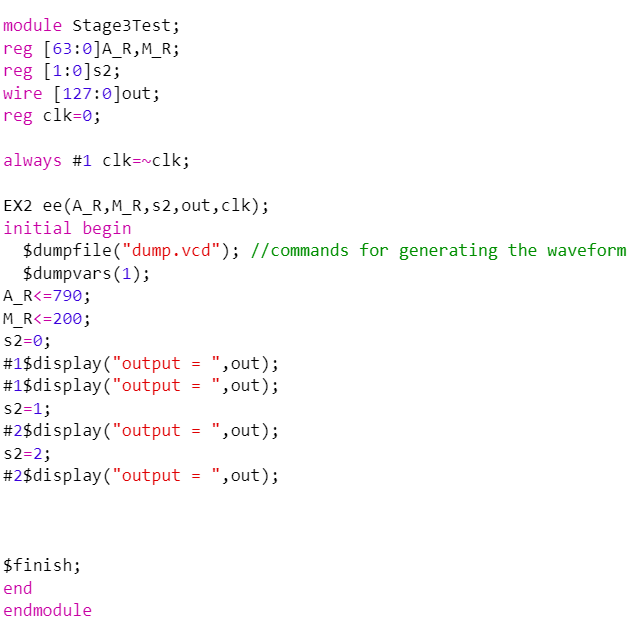
First operation done in the module was splitting the 32 bit-sized operands rs1 and rs2 into four 16 bit-sized coordinates x1 and y1 were derived from rs1 and similarly x2 and y2 from rs2. Then the addition and point multiplication where performed according to the selection line (s1) which will choose between OP2 or K to perform operations with OP1. S1 was cleared to zero in the first cycle to ensure the constant k will be chosen with OP1, and set to high to choose OP2 for the same manner.

Due to the dependency between the two previous operations, one time unit delay is needed before each case in the case statement of the module, therefore; this stage needs 4 time units as the clock cycle’s duration. This delay is clear in the waveform such that the addition and multiplication result are ready after the clock’s positive edge for a short time.

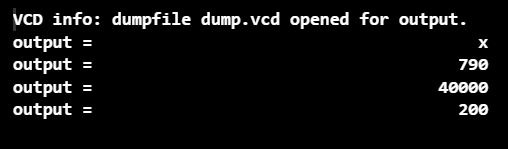
**Stage 3:**

This is where the output of the multiplier in stage 2 gets squared and the squared value enters a 4x1 mux along with the adder’s output and the multiplier’s output. Then the mux chooses between them and stores the generated coordinates in the buffer to be used in stage 4. The testbench and the output below show the 3 cases depending on the selection line (s2)

**Testbench:**



**Output:**



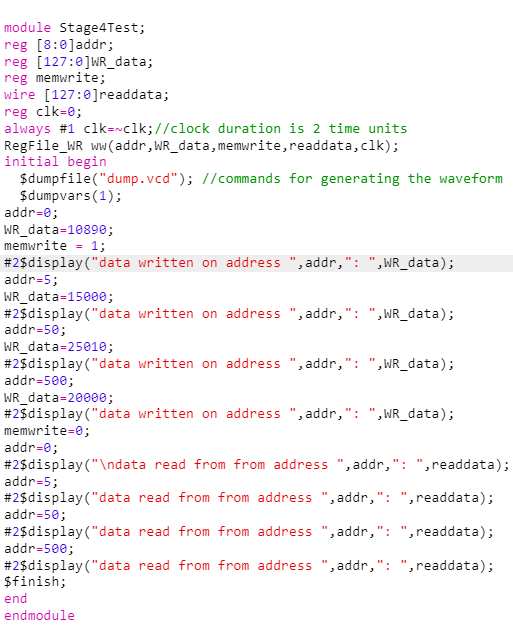
**Waveform:**



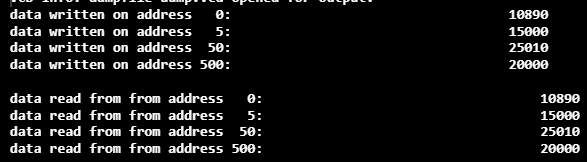
**Stage 4:**

this is the write-back stage, where all our hardware implementation comes to a result by transmitting the point register to the RAM file on a specific address to be used in further stages of encryption. The RAM module should receive data and an address to write on with memwrite signal as a high signal. In our implementation we have tested four addresses to write random data in them then send memwrite as a low signal to read the data ensuring it was stored in the right addresses.

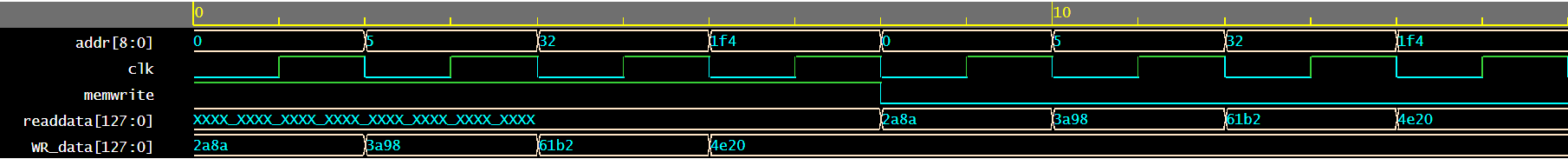
**Testbench:**



**Output:**



**Waveform:**



Eight cycles are shown in the waveform (one for each test instruction), notice that the memwrite signal was driven high and the readdata wire was undefined because the RAM file was updating its data for half of the duration.

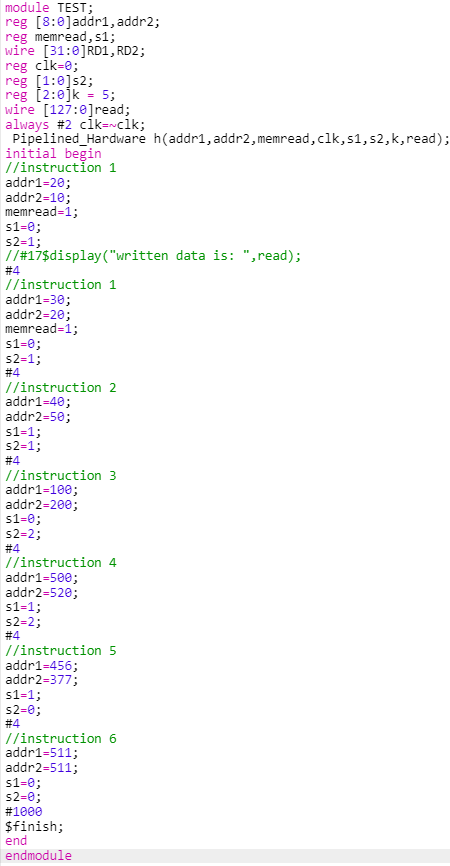
The write register file was triggered with the clock’s negative edge to avoid the contradiction between this stage and the first one so that the system can’t read and write simultaneously.

**The fully pipelined hardware:**

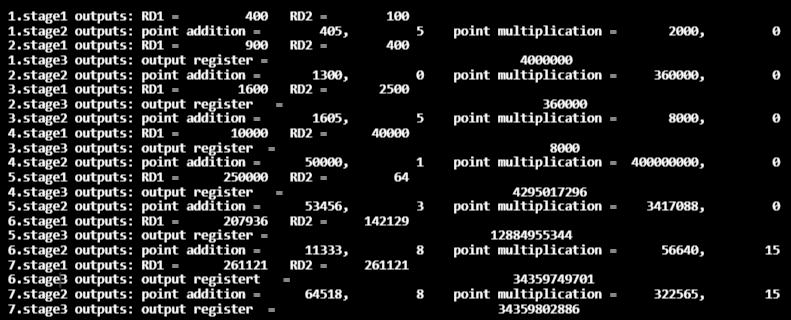
To conclude, we have implemented a hardware module that adds up every component and stage discussed earlier. The inputs of the pipelined hardware module are the same as the first stage inputs, and it outputs a 128 sized register whose lower 64 bits represent the new X coordinates, and the upper represent the new Y coordinates.

The clock cycle’s duration is implemented as the longest stage duration (stage 2) which equals four time units. To test the performance and functionality, seven instructions where given and resulted in the output.

**TestBench:**



**Output:**

****

**Correctness of the design:**

by taking the first instruction as an example, two addresses where sent with the values 20 and 10 respectively to the register file to read the data stored in these addresses according to the function RAM[i]= i\*i, which will result to sending the values 400 and 100 to the buffer leading to stage 2.

In the first execute stage, the selection line sent (s1=0) will allow the mux to choose the constant K which is initialized with a value of 5 to perform arithmetic operations with OP1 which has the value 400, implented in binary as 110010000 with the sign extension to reach the size of 32 bits.

The adder will add the constant k with both the lower and upper 16 bits of OP1 resulting addition of the lower bits 400 + 5 = 405 and the higher bits addition result is 5 + 0 = 5 the implemented module named wiring will sign extend the result to 64 bits to the point addition register A\_R (lower 32 bits = 405, higher 32 bits = 5). The total value in A\_R is 10100000000000000000000000110010101 which is represented as 21474836885 in decimal

Similarly, the multiplier will operate depending on the operands k and OP1 for both the lower and upper 16 bits of OP1 and will take the results 5\*400 = 2000 and 5\*0 = 0 and extend them to the 64-bit register M\_R using the wiring module. The value in M\_R is 2000

Stage 3 contains a multiplexer that will choose the squared value of M\_R and pass it to the output register of size 128 bits according to the selection line s2 which is set to1. The value of the the output register is 2000\*2000 = 4000000 which is the same as the output printed in the screen of stage 3 for the first instruction.

**Comparison with a non-pipelined architecture**

time needed for a single cycle implementation:

tn = tstage1 + tstage2 + tstage3 +tstage4 = 2+ 4 + 2 + 2 = 10

speedup (S):

S= n.tn/(k+(n-1)tp) ; where k=4 (number of pipelined stages)

If we assume n=10 instructions;

S=10\*(10)/((4+9)\*4)

S= 1.92

so, the pipelined hardware is faster 1.92 times than the single cycle implementaion.

Ideal case:

If we assume n=1,000,000 since 1,000,000 >> 1

S=n.tn/n.tp

S=k.tp/tp

S=4

The pipelined hardware is 4 times faster than the single cycle implementaion.

**Conclusion**:

Overall, the project aimed to create a hardware implementation capable of performing arithmetic operations on coordinates in an elliptic curve graph. The four stages were designed to handle different aspects of the operations, including initialization, selection, computation, and data storage, leading to a complete system for Elliptic Curve Cryptography

**References**:

1. Rashid, Muhammad, et al. “An Efficient Elliptic-Curve Point Multiplication Architecture for High-Speed Cryptographic Applications.” *Electronics*, vol. 9, no. 12, 12 Dec. 2020, p. 2126, <https://doi.org/10.3390/electronics9122126>. Accessed 21 May 2021.
2. Arielle Verri Lucca, et al. “A Review of Techniques for Implementing Elliptic Curve Point Multiplication on Hardware.” *Journal of Sensor and Actuator Networks*, vol. 10, no. 1, 31 Dec. 2020, pp. 3–3, <https://doi.org/10.3390/jsan10010003>. Accessed 29 Apr. 2023.
3. Rashid, Muhammad, et al. "A 4-stage pipelined architecture for point multiplication of binary huff curves." Journal of Circuits, Systems and Computers 29.11 (2020): 2050179.

codes link: <https://github.com/omar-abk43/Arch-2.git>