December 16th, 2020

Fall 2020

CS 5387

Final Project – Application Specific Vector Processor

Table of Contents

Introduction:

In the project we requested to build a processor to compute specific arithmetic on three different input vectors of dimension four, ***a, b, d***. Each dimension represents 8 bits 2's complement number. The output of the processor produces a single output ***f***, four dimensions, of 8 bits 2's complement number as well. The calculation of ***f*** should be as described below:

$$(1) \qquad c_i[15:8] = a_i * b_i \quad ; \quad i = 1,2,3,4 \quad ; \quad \vec{c} = [c_1, c_2, c_3, c_4]$$

$$(2) \qquad \vec{e} = \vec{c} + \vec{d}$$

$$(3) \qquad \vec{f} = \vec{e} - (256 * \vec{b})$$

First, we will multiply each value in vector ***a*** with the corresponding value in vector ***b***. Since the multiplication is 8 bits by 8 bits, the product is a 16 bits number. The vector ***c*** will be the 8 most significant bits of the product produced from the operation, as we can see in equation (1) above. At this time, we will add the input vector ***d*** to the value we received for ***c***, to compute the vector ***e,*** as shown in equation (2). since ***c*** is the 8 most significant bits of its original product, ***d*** will be counted as values shifted 8 bits to the left (multiplied by 256). Finally, we will shift the values of ***b*** 8 bits to the left, as we did for ***d***, and subtract this value from ***e***, which already shifted, to receive the output vector ***f,*** as shown in equation (3)***.***

At each clock cycle, the processor will receive one input at a time. First four cycle the processor will receive the values of ***a***, then next four cycles the values of ***b***, and cycles 9 to 12 it will receive the input vector ***d***. The result computation vector ***f*** should come out in order of f_1, f_2, f_3, f_4 respectively.

The design should have a maximum initiation rate of 13 clock cycle and a maximum latency of 15 clock cycles. Since the initiation rate does not equal to the latency, pipelining is required for the implementation. In addition, each component as well as the arithmetic units should be specified at the RTL level.

Resource Scheduling:

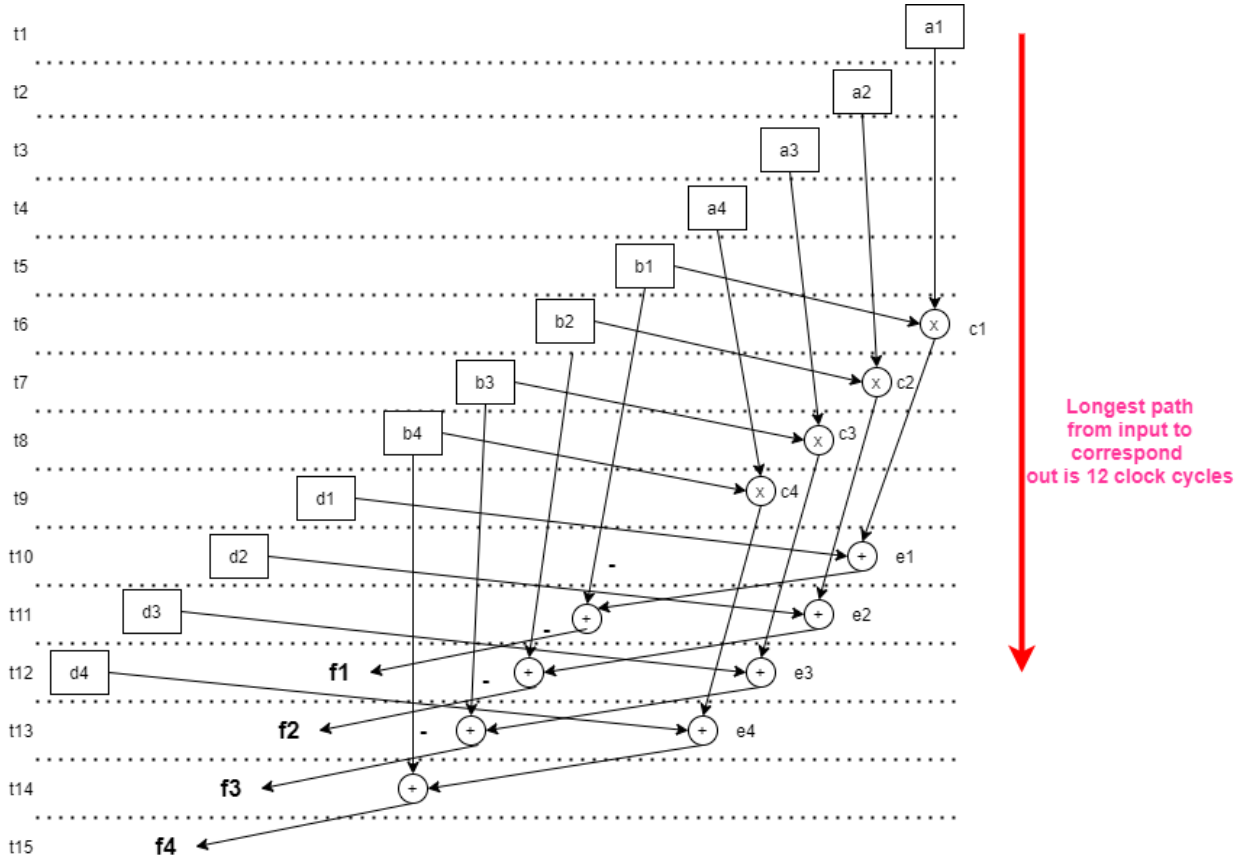


Figure 1 – data flow diagram[1]

ALUs Estimation:

In the diagram above we see the data flow of the processor. We can see that the longest path for all four outputs equals to 12 clock cycles. The number of additions in the processor is 8 and the number of multiplications is 4. Using the equations to calculate the minimum number of adders and multipliers we will receive:

$$\#\ adders = \frac{\#\ additions}{\#\ clocks\ in\ longest\ path} = \frac{8}{12} = 1$$

$$\#\ multipliers = \frac{\#\ multipications}{\#\ clocks\ in\ longest\ path} = \frac{4}{12} = 1$$

Using the minimum number of adders and multipliers, I will try to satisfy the requirements of maximum clock cycles for latency and initiation rate.

time	input	multiplier	adder	output
1	A1			
2	A2			
3	A3			
4	A4			
5	B1			
6	B2	C1 = a1* b1		
7	B3	C2 = a2* b2		
8	B4	C3 = a3* b3		
9	D1	C4 = a4* b4		
10	D2		e1= c1+d1	
11	D3		e2= c2+ d2	
12	D4		e3= c3+ d3	
13			e4=c4+ d4	
14			F1 = e1 + (-b1)	
15			F2 = e2 + (-b2)	F1

Table 1 – first try to satisfy requirements with minimum number of adders and multipliers

As we can see in *table 1,* we could **not** satisfy the requirement of latency of 15 clock cycles using just one adder and one multiplier. We received just one dimension of the four required for the vector output ***f.***

From the data flow diagram in *figure 1*, we can see that another multiplier will not help us reaching our goal since the uses of it are in different cycles. However, in cycles t11, t12, and t13, we see that there should be two additions in parallel – means if we will add another adder, it will reduce our latency.

time	input	multiplier	Adder1	Adder2	output
1	A1				
2	A2				
3	A3				
4	A4				
5	B1				
6	B2	C1 = a1* b1			
7	B3	C2 = a2* b2			
8	B4	C3 = a3* b3			
9	D1	C4 = a4* b4			
10	D2		e1= c1+d1		
11	D3		e2= c2+ d2	F1 = e1 + (-b1)	
12	D4		e3= c3+ d3	F2 = e2 + (-b2)	F1
13	A1		e4=c4+ d4	F3 = e3 + (-b3)	F2
14	A2			F4 = e3 + (-b4)	F3
15	A3				F4

Table 2 - second try to satisfy requirements with another adder

In *table 2*, where we added another adder, we could satisfy the requirement for latency of 15 clock cycle since we produce all four dimensions of the output vector in the 15th clock cycle.

In order to satisfy the requirement of initial rate of 13 clock cycle, we just need to continue loading the next inputs starting in time t13, as shown in red above. Since the first five cycles in the process do not require any arithmetic operation, there is no need for additional adder or multiplier, and this requirement can be satisfied as well just with one multiplier and two adders.

Register Estimation:

Once we figured out how many arithmetic units (ALU) we will use in our design, we can discover how many registers we be needed so store all required values. In order to do so, we will go through each cycle to discover its unique requirements for registers.

Looking at *table 2* we see that the first five cycles we are just loading inputs which will be used later in the process; therefore, we need to store each one of them and allocate 5 registers (registers 1-5). In the sixth cycle, t6, where we use the first multiplication, we will need space for two additional values: the new input, b2, and the product of the multiplication. Since we do not need a1 anymore after we calculate c1, we can store c1 in the register that stored a1 previously (register 1). However, for the new input we will need a new register (register 6). We will do the same for cycles t7 through t9, which require total of 9 registers.

Once we get to t10, we face the first addition. In this addition we will use c1 (register 1) and d1 (register 9), which we loaded previous cycle. Since we do not need the value in either one of these registers, we will choose to store the sum, e1, in register 1. The new incoming input, d2, cannot be loaded to any of the already exist registers, due to future use or current use, and we will allocate a new register (register 10).

Heading to t11, we can load the new input, d3, into the register 9, which stored d1 previously. The sum from the first adder, e2, we can load to register 2, which hold c2 previously. The sum of the second adder and the output, f1, do not need to be registered and will be assigned to the output of the processor immediately. Going through t12 to t14, we can use the same technique as before and assign the sum of the first adder to the register of c, will help us to do not add any additional register and finish our process with total of 10 registers. Since we do not need the value in register 1 (e1) after t11, it will be free to reload the new input value of a1 in the next group of inputs starting at t13.

The conclusion of the register assignment is illustrated below:

Register 1: a1 → c1 → e1

Register 2: a2 → c2 → e2

Register 3: a3 → c3 → e3

Register 4: a4 → c4 → e4

Register 5: b1

Register 6: b2

Register 7: b3

Register 8: b4

Register 9: d1 → d3

Register 10: d2 → d4

State Assignment:

As we saw before, our latency is 15 clock cycles, and the initiation rate is 13 clock cycles. The initiation rate refers to that we have total of 12 different state, and we start the process once over again every 13th cycle. Therefore, we will need to have state assignment for total of 12 states.

In order to do so, I will use the Gray code method for changing just one bit at the time when moving from state to state. For 12 states, we will need presenting each state using 4 bits:

0000 0001 0011 0010 0110 0111 ~~0101 0100 1100 1101~~ 1111 1110 1010 1011 1001 1000

Since we use just 12 states and 4 bits can represent 16 states, we will remove the middle four states and we will not lose the advantage of changing just one bit at a time.

Concluding the section of the resource scheduling, we can see the final design of the states, registers and ALUs in *table 3* below:

time	state	input	multiplier	Adder1	Adder2	out
1	0000	Load a1 to reg1				
2	0001	Load a2 to reg2				
3	0011	Load a3 to reg3				
4	0010	Load a4 to reg4				
5	0110	Load b1 to reg5				
6	0111	Load b2 to reg6,	C1 = reg1(a1)*reg5(b1)			
7	1111	c1 to reg1 , Load b3 to reg7,	C2 = reg2(a2)*reg6(b2)			
8	1110	c2 to reg2, Load b4 to reg8,	C3 = reg3(a3)*reg7(b3)			
9	1010	c3 to reg3, Load d1 to reg9,	C4 = reg4(a4)*reg8(b4)			
10	1011	c4 to reg4, Load d2 to reg10		e1=reg1(c1)+reg9(d1)		
11	1001	e1 to reg1 Load d3 to reg9		e2=reg2(c2)+reg10(d2)	f1=reg1(e1)-reg5(b1)	
12	1000	e2 to reg2 Load d4 to reg10		e3=reg3(c3)+reg9(d3)	f2=reg2(e2)-reg6(b2)	F1
13	0000	e3 to reg 3 load next a1 to reg1		e4=reg4(c4)+reg10(d4)	f3=reg3(e3)-reg7(b3)	F2
14	0001	e4 to reg 4 load next a2 to reg2			f4=reg4(e4)-reg8(b4)	F3
15	0011	load next a2 to reg3				F4

Table 3 – final design of the resource scheduling

Implementation:

For designing the processer, I needed two different type of ALUs: 8 bits adder and a multiplier. Also, I needed a register to store the data, datapath to connect between ALUs and registers and a controller to control which register will be used in each state. All will be cover by a top layer which will receive as input just the clock, reset, and the next input. Below is a description of each part of my implementation:

Adder:

The adder for 2'complement is a regular adder. My adder designed from a basic structure of full adder. each full adder will receive as input 1 corresponding bit from each input to the adder as well as a carry in bit. The output will be the sum and the carry out. The truth table for the full adder is shown below:

A	B	Cin	Sum	Cout
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

Figure 2 – truth table full adder[2]

Creating K maps from the results from the table above, we can calculate the outputs using the following logic gates:

$$sum = A \oplus B \oplus Cin \tag{4}$$

$$Cout = AB \mid ACin \mid BCin \tag{5}$$

The sum is simply an EOR of all inputs and the carry out is the majority circuit.

Building on the top of this basic structure, we implemented the 8 bits adder using the following structure:

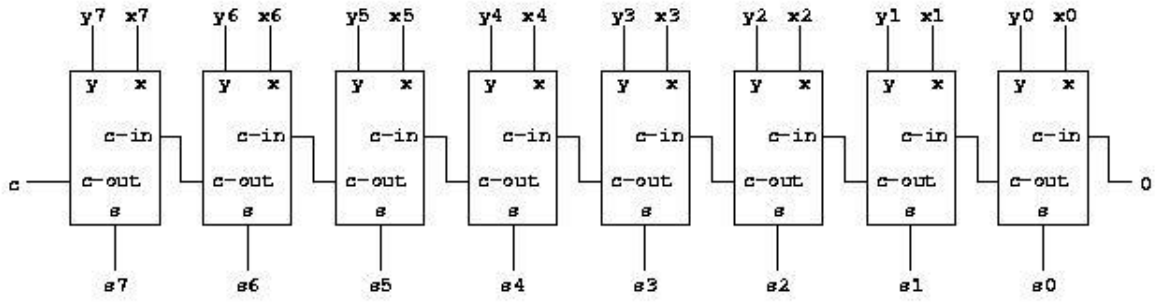


Figure 3 – 8 bit adder structure[3]

The implementation for adder can be found in the file ***adder.v.*** this file contains the two modules: full adder (fa) and the top layer of the adder (adder). the fa module receives the inputs A, B, CIN and produces the outputs S and COUT as described above. The adder module receives the inputs in1, in2 of 8 bits as well as sub of 1 bit. The sub variable is to determine if it is a subtraction instead of addition (for the second adder in *table 3)*. Subtraction in 2'comlemet is just addition with the inverted number plus one of the second input, therefore, if it is a subtraction, this bit will set to one and in2 will be inverted when passed to this module and one will be added to it. Inside of the top module, I created 8 full adder which connect to each other like show in *figure 3.* The adder module's output is an 8 bits number called out. For this project, we do not care about the carry out of the last full adder.

Testing of the adder will be shown below in the testing section.

Multiplier:

For the multiplier I used the Booth algorithm. The code for it shown in the file ***multiplier1.v***. Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation in efficient way, i.e., a smaller number of additions and subtractions required in general. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^{k+1} to 2^m.[4]

As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following cases:

1. The multiplicand (M) is subtracted from the partial product (A) when the least significant bit of the multiplier (Q) is 0 and the previous one (q0) is 1.
2. The multiplicand is added to the partial product when the least significant bit of the multiplier is 1 and the previous one is 0.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

After each step, the concatenation of the partial product and the multiplier shift one bit to the right, filing out the MSB with the previous MSB. This process repeats n times, where n in the number of bits of the inputs. In our case it is n = 8.

We can see the ASM diagram of the algorithm in the *figure 4* below:

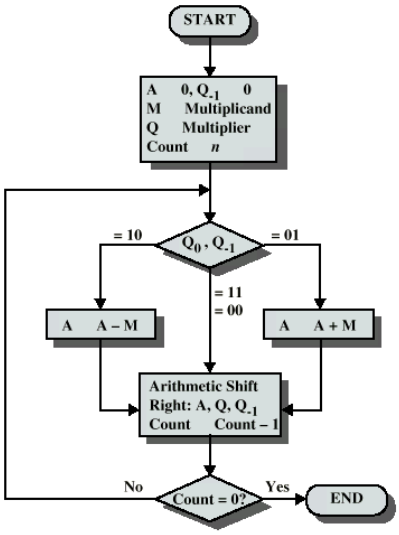


Figure 4 – Booth algorithm – ASM diagram[4]

In my implementation once again, I had the basic block structure of the Booth algorithm (named Booth) as well as a top layer named multiplier. As shown in the diagram above, the Booth module receive A_in – the partial product, M – the multiplicand, Q_in – the multiplier, and q0 (shown as Q_{-1} in the diagram above) the previous LSB of Q as inputs, and produce the new partial product – A_out, the new multiplicand – Q_out, and the new LSB q0_out as outputs.

The top layer of the multiplier received just two inputs of 8 bits in1 and in2, and produced the output out of 8 bits as well, to represent the 8 MSB of the real product. Inside of the module, instead of having a counter, I used 8 modules of Booth, B1 to B8, which each receive the previous Booth outputs as show in the diagram below:

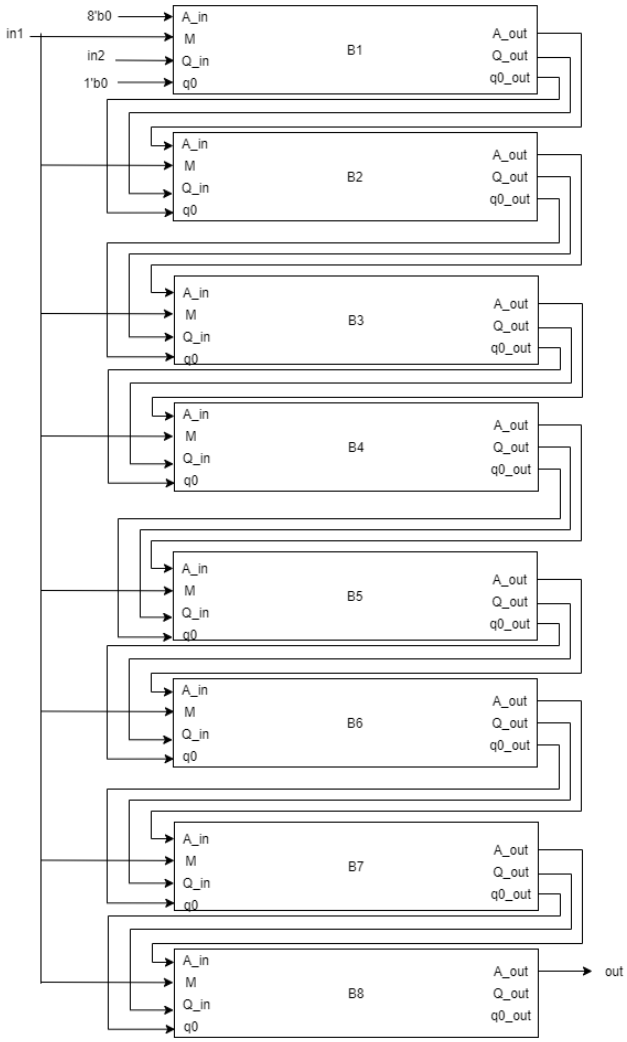


Figure 5 – Booth multiplier – circuit diagram[1]

Testing of the multiplier will be shown below in the testing section.

Register:

The implementation for the register I took from lab 5, where I also tested it. The code shown in the file ***register.v.*** the inputs for the register are: 8 bits number din, clk, reset, and ld (load). The only output is the data that it contains. The module contains only one always block active on the positive edge of the clock or the asynchronous reset. When the reset is set the data assign to zero. When the clk is set, first it checks if the ld flag set as well and just then set the data to din, otherwise it will do nothing.

Datapath:

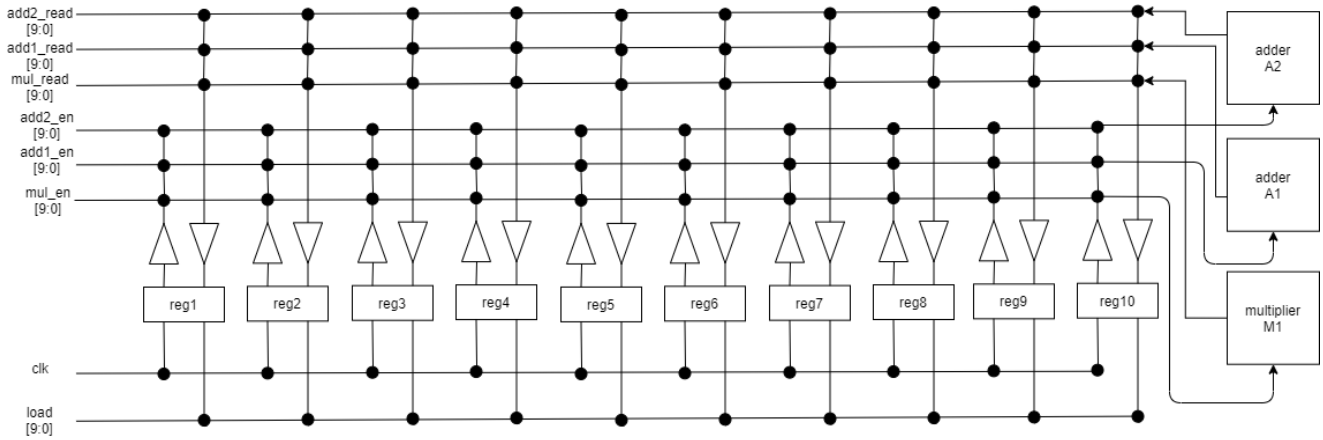


Figure 6 – Datapath diagram[1]

My implementation for the datapath is shown in the file ***datapath.v.*** as described before in the resource scheduling section and as shown in the diagram in *figure 6*, the final implementation contains one multiplier (M1), two adders (A1,A2), and 10 registers (reg 1 through 10).

The datapath module receive the inputs clk, reset, and next_in from the user. The other inputs will be passed by the controller (describe below in the controller section). The inputs are: state (4 bits) and the 10 bits long enable buses: load, mul_en, add1_en, add2_en, mul_read, add1_read, and add2_read, as shown in the diagram above.

The module produces the only output f, which is 8 bits long, as described in the introduction. In my implementation, for testing purposes only, I also set the registers values, r1 through r10 as outputs to be shown in the testing board. As shown in the comments in the file, it is very easy to change it and set those just as internal reg type variables inside the module instead of outputs (please see lines 29 and 46 in datapath.v).

Inside the datapath module I called the modules of the multiplier (M1) and twice to the adder module, to represent adder1 (A1) and adder2 (A2). The variables I send to the multiplier are *mul1* and *mul2*, and receive as an output the variable *prod*. Each variable will be assigned to the corresponding register by the enable busses from the controller. For the adders, the first one is doing just addition, therefore the sub flag will be set to 0. The second adder is doing only subtraction, the sub flag set to 1 and the second

input is inverted. The same as the multiplier, the inputs and the output of each adder set by the enable buses.

Each register will be created by calling the register module. Below is an example of creating the first register:

```
register reg1((load[0]) ? next_in : (mul_read[0]) ? prod : (add1_read[0]) ? sum1 : (add2_read[0])
? sum2 : 8'bz,

        clk,reset,

        load[0] || mul_read[0] || add1_read[0] || add2_read[0],

        r1);
```

The first argument we send to the register is the data we want to store in the register. In order to make sure I store the correct data I check the corresponding load or read values from each enable bus. For this register, the location of the bit is 0, for reg 2 is location 1 and so on.

The second and third argument are the clk and the reset which send to all register, and the last input is ld flag, which will be set to 1 just if it need to load or read from one of the ALUs so it will be able to store the new data.

In addition, I create a variable flag (named ***not_first_flag***) to be set to true if it pass state number 4, in order to do the operations needed in state 1 trough 3 just after the first time the program in those states i.e. calculating e4, f3, and f4 where should occur just after the whole process, at t13, and not at the beginning.

In the always block inside the module, I am using **data forwarding** – since a result of operation can be stored in a register just in the next clock cycle, if I was taking the data from the register in the next cycle it would not be available yet, but just after two cycles e.g. t11, when we store the sum of the first adder in reg1 and use it in the next cycle to calculate f1. Therefore, since the data exists in the circuit but not yet in the register, I will set the next input to the output of the relevant ALU instead of to the register specified in the table above. In the code it will be shown in s11 (datapath.v - line 193), where I set add21 (the first input to second adder) to sum1 which is the output from the first adder (e1) and not the register where we stored it (reg1).

In each state we do not need to specify which register to load the new input since the way I created the modules of each register will do it automatically outside of the always block. However, I set the relevant

inputs of the adders and the multiplier to the relevant registers according to the description in *table 3* e.g. at state 8 we set the input to the multiplier, mul1 and mul2, to reg 2 and reg6 respectively.

Controller:

The code for the controller in the file ***controller.v.*** The controller inputs are just the reset and the clk. It produces the state and all the enable buses as outputs as shown in *figure 6.*

The states assign to parameters for readability. Like in the datapath, there is a need for a flag (not_first_flag) to make sure we passed the first 3 cycle to define additional functionality to state s1 – s3. The module contains two always block. The first block is the state control. It will be trigger by the clk or by the asynchronous reset. In each state, the next state will be assigned. The default is the first state s1 and setting the flag to 0. The second block is the bus enable control. In each state it will assign the relevant values to the enable buses according to the description in *table 3*. For example: in s7 where we need to load b3 to reg 7, multiply reg2 and reg6, and read the previous product to reg1, we set the busses as follows:

```
load = 10'b0001000000;
mul_en = 10'b0000100010;
mul_read = 10'b0000000001;
```

we see that for the load the 7th bit is set, in the mul enable the second and the sixth bits are set, and in the read from the mul we set the first bit to send the product to reg1. The other busses are set to 0 to prevent unwanted activities.

Top:

The top module code is in the file ***top.v.*** its inputs are just clk, reset and the next_in. the output are f, and as said before, for testing purposes, I also set the register data as outputs so those will be shown in the testing board.

Inside the module, I create the wires for the enable buses as well as the state. Also, I called the controller module and send the relevant inputs (as described above), and I called the datapath module as well.

Testing and Results:

For each module in my implementation I generate a test bench file to test it separately. At the end I wrote a test bench to the top module which include all the other files.

Adder:

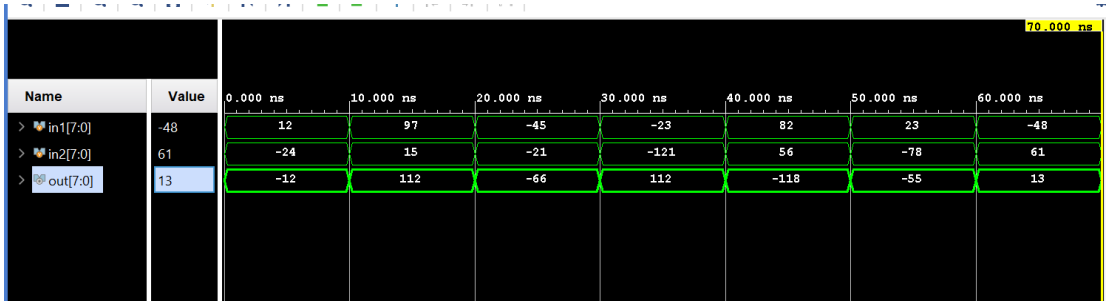


Figure 7 – adder testing results in decimal

The test bench for the adder is in the file ***adder_tb.v.*** In the testing board above, I tested the adder with random inputs. I combined negative and positive numbers. The results shown in decimal for user readability. In the fourth and the fifth test, I generate sum to be larger than 127, which is the maximum of 8 bit 2'complement representation, and smaller than -128, which is the smallest values for this representation. In both on those tested I received expected results.

Multiplier:

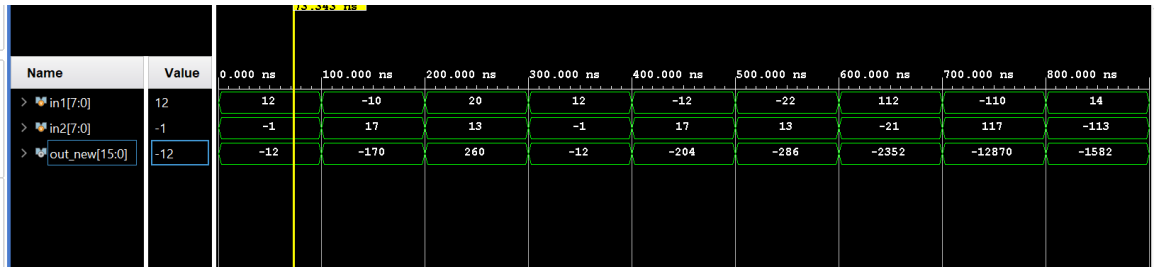


Figure 8 – multiplier testing result in decimal

The test bench or the multiplier is in the file ***new_mul_tb.v.*** In *figure 8* I show my result from testing the multiplier. Just for the testing I set the output to be 16 bits long. The results above shown in decimal. As we can see, all the testing cases passes, and the multiplier works as expected. When I use it in the program, as said before, the output of the multiplier id just the 8 MSB.

Datapath:

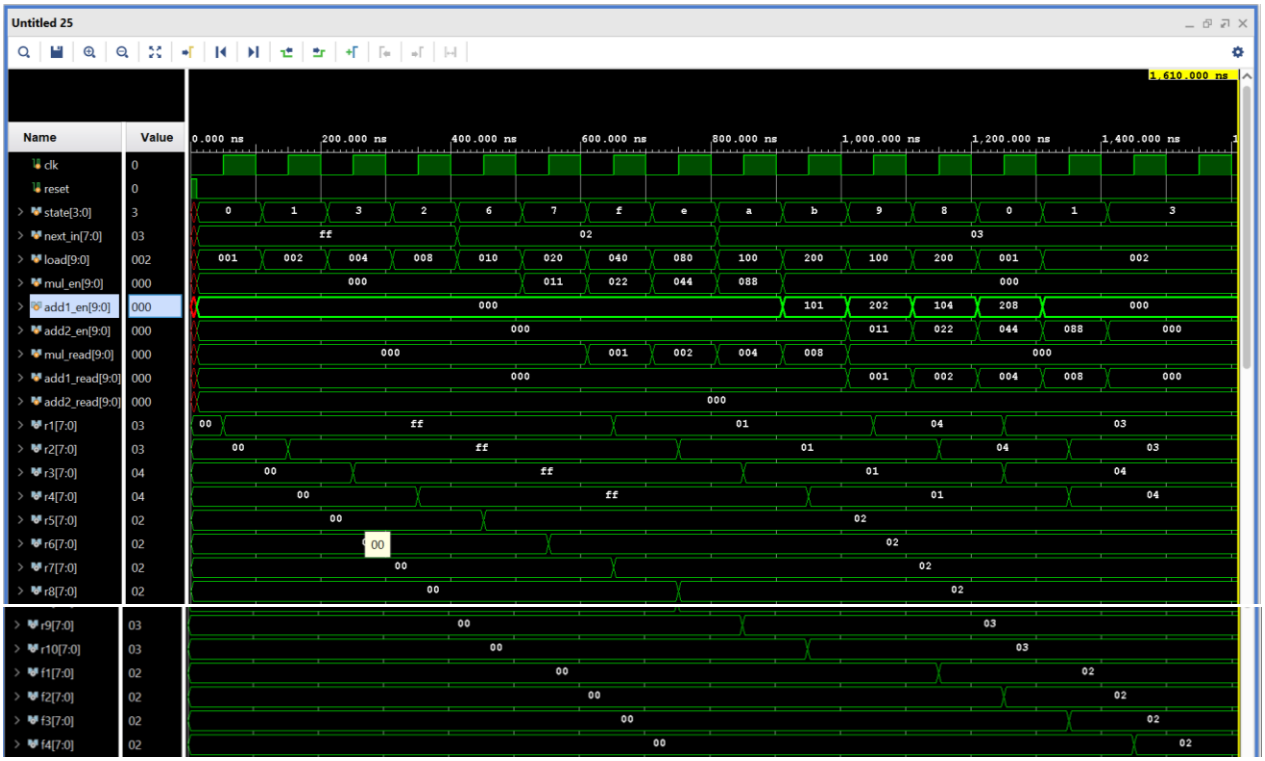


Figure 9 – datapath testing results

The test bench for the datapath shown in the file ***datapath_tb.v.*** In the test, I just wanted to make sure that the datapath function correctly when the user gives the relevant state and enable buses. For the test, I set all a's to 0xff, all b's to 0x02 and all d's to 0x03. The results shown above are correct after hand calculation. In the test bench file, I send to the module the next_in input as well as the current state and its relevant enable buses.

Controller & Top:

After the datapath test passes, I connected the controller to the datapath using the top layer and used the same test case and expected the same results. At this time, in the test bench, I sent to the module the clk, reset and the next_in inputs. The results from the first test shown in *figure 10* below:

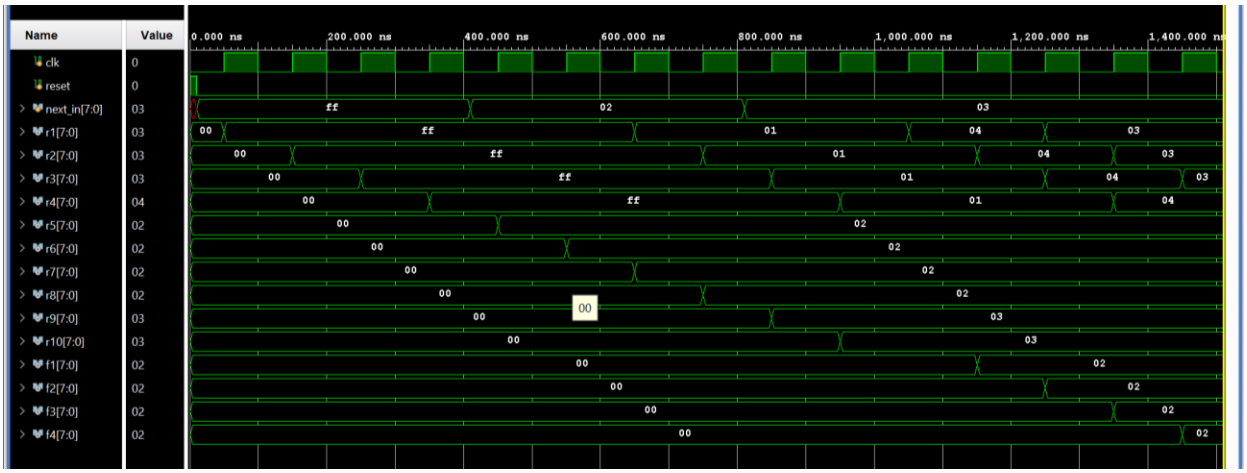


Figure 10 – first test for the top module – same result as in the datapath test

Once this test passed, I changed the output f to be represented just by one output (called f) instead by four different outputs shown previously.

For testing the module in a better way, I created a python scrip to generate random test cases for the module, for more than just 15 clock cycles as shown before. The python scrip also will generate the expected results. The code for the scrip can be found in ***testing_top.py***.

After testing several times, with different random inputs, in *figure 11* below shown one of the testing cases, which also can be found in the file ***top_tb.v.***

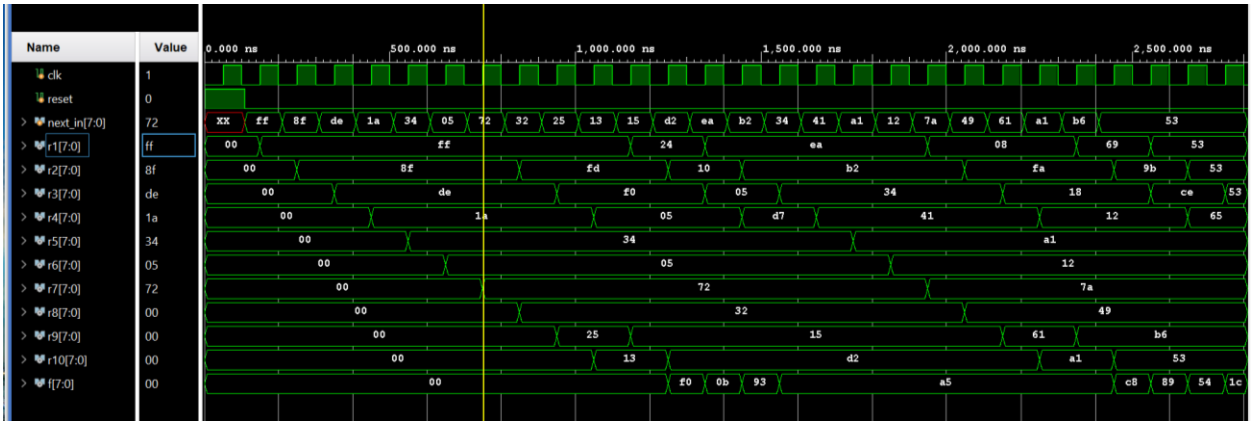


Figure 11 – random test for top module generated by the python script

The expected results from the scrip shown in *figure 12* below:

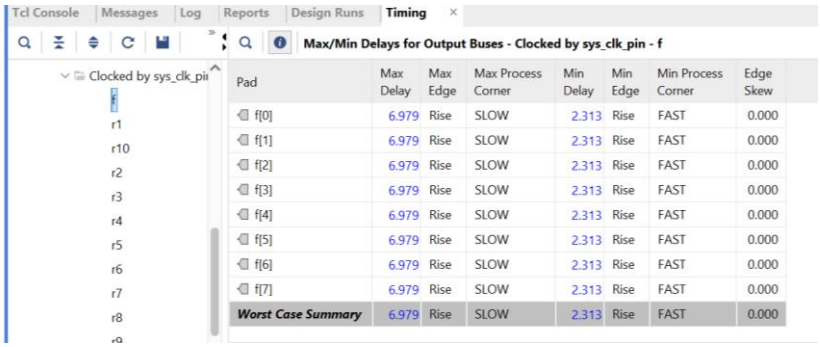
```
---------- peiod 1 ----------
c1 =   0xff e1 =   0x24 f1 =   0xf0
-------------------------------
c2 =   0xfd e2 =   0x10 f2 =    0xb
-------------------------------
c3 =   0xf0 e3 =    0x5 f3 =   0x93
-------------------------------
c4 =    0x5 e4 =   0xd7 f4 =   0xa5
-------------------------------


---------- peiod 2 ----------
c1 =    0x8 e1 =   0x69 f1 =   0xc8
-------------------------------
c2 =   0xfa e2 =   0x9b f2 =   0x89
-------------------------------
c3 =   0x18 e3 =   0xce f3 =   0x54
-------------------------------
c4 =   0x12 e4 =   0x65 f4 =   0x1c
-------------------------------
```

Figure 12 – expected result for the test shown in figure 11

As we can see, the test results we received in *figure 11* are correct according to the results calculated by the python script, as shown in *figure 12.*

Performance:

After testing, I synthesize the project and checked for the max delay using the Vivado timing tool:



Pad	Max Delay	Max Edge	Max Process Corner	Min Delay	Min Edge	Min Process Corner	Edge Skew
f[0]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
f[1]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
f[2]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
f[3]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
f[4]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
f[5]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
f[6]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
f[7]	6.979	Rise	SLOW	2.313	Rise	FAST	0.000
Worst Case Summary	6.979	Rise	SLOW	2.313	Rise	FAST	0.000

Figure 13 – max delay of the output

From the figure above we can see the maximum delay of the output is 6.979ns. calculating for the maximum frequency of our design:

(6) $$freq_{max} = \frac{1}{6.979ns} = 143MHz$$

We receive max frequency of **143 MHz.**

Conclusion:

In this project we created a processer to calculate specific output vector from given vector inputs. First, we calculated the minimum resource, then we add more resource to satisfy requirements. We allocate the minimum number of registers needed and we set the table to indicate what operations happen in each state, just then we start to implement the code of it. the code was written in the RTL level and was tested first, each element by its own, then the whole unit. Eventually, we generate more generic test using the python scrip.

From this project first, I learn the steps needed when creating any kind of processor – resource estimation and scheduling, satisfy requirements, testing, and maximize performance. Also, I got more familiar with the Verilog environment and became more comfortable thinking in a way of "hardware" programing rather then software programming I used to before. In addition, I learn about the Booth algorithm which I was not familiar before.

References:

[1] Using the tool: https://app.diagrams.net/

[2] https://www.researchgate.net/figure/Full-Adder-truth-table_tbl2_277928299

[3] https://dept-info.labri.fr/~strandh/Teaching/AMP/Common/Strandh-Tutorial/circuits-for-binary-arithmetic.html

[4] https://cnx.org/contents/ZxCI0OEL@1/Booth-s-Algorithm