Imperial College London

CPU PROJECT REPORT

ELEC40006 – 1st Year Electronics Design Project 2021

Prepared by

Christos Patsalidis, CID: 01866599

Rahimi Mohd Ridzal, CID: 01852253

Lou Alsteens, CID: 01917074

Word Count: 10312 words

Contents

1. Introduction 3
   1.1. CPU Importance 3
   1.2. Project Outline 3
2. Feature Requirements and Intendent Use 3
   2.1. Serial Communication 3
   2.2. Floating-Point Arithmetic 4
   2.3. Dual Core CPU 5
3. UART Implementation 6
   3.1. Transmitter and Receiver 7
   3.2. UART Status 10
   3.2. Advanced Interrupt 14
   3.3. Testing 15
4. Floating-Point Arithmetic Implementation 17
   4.1. Research 17
   4.2. Half Precision Floating-Point 17
   4.3. Half Precision Floating-Point Addition and Subtraction 18
   4.4. Half Precision Floating-Point Multiplication 26
   4.5. Testing 28
5. Dual Core Implementation 32
   5.1. Research 32
   5.2. Design Explanation 33
   5.3. UART Involvement 35
   5.4. Testing 36
6. Final Evaluation 39
   6.1. Benchmark 39
   6.2. Benchmark result 41
7. Project Management 45
   7.1. Meetings and Note Sharing 45
   7.2. Milestones 46
   7.3. Meeting Structure and Gantt Chart 46
8. Conclusions 48
   8.1. Thoughts on the Project 48
   8.2. How to Improve on Current and Future Work 48

References 50

Appendix A 51
Appendix B 52
Appendix C 54
Appendix D 57
Appendix E 62
Appendix F 65

1. Introduction

1.1. CPU Importance

Modern computers are becoming increasingly more demanding with the passage of time. Therefore, a high performing and efficient CPU is required, to achieve a great level of smoothness and speediness when running our programs. The following quote perfectly illustrates how vital a decent CPU is to a computer:

*'The **CPU** is the brain of a computer, containing all the circuitry needed to process input, store data, and output results.*

The CPU is constantly following instructions of computer programs that tell it which data to process and how to process it. Without a CPU, we could not run programs on a computer.'[1]

1.2. Project Outline

A CPU has a specific number of features that benefit the user. (e.g., Arithmetic) In this project we are required to implement three additional features to our MU0-ARM CPU design, which will be explained in detail below. This integration will not only make our CPU more efficient, but it will also solve some of the problems that we face when dealing with pre-existing features.

Our goal is to develop a fully functioning CPU which is positively enhanced by these additional elements, and also strive to create designs which do not unnecessarily waste real life resources.

2. Feature Requirements and Intendent Use

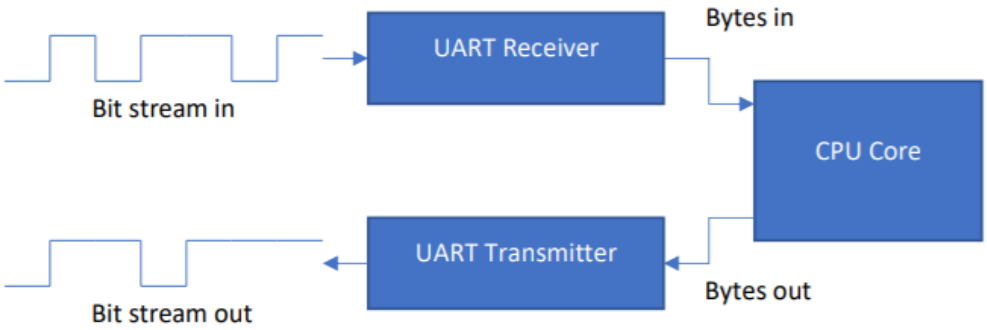
2.1. Serial Communication

Serial communication is the first feature that needs to be implemented. More specifically, we will be designing a UART (universal asynchronous receiver transmitter) to act as a memory-mapped device. This means that our CPU will be able to receive and transmit bytes, which we

[1] Pamela Fox n.d., *"Central Processing Unit (CPU)"*, Khan Academy, accessed 2 June 2021, <https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:computers/xcae6f4a7ff015e7d:computer-components/a/central-processing-unit-cpu>

will be able to store and access through special registers in our CPU. (Not in any RAM locations)

The UART block needs to have two sub-blocks, which will be responsible for receiving or transmitting a 10-bit word.



The word that is to be received/transmitted needs to include 1 start bit, (of logic 0) 1 stop bit (of logic 1) and 8 data bits. (Where the LSB of the byte is the first one to be received/transmitted)

Since the UART is built to receive/transmit data one bit at a time, for this implementation we must take into consideration that the rate of transmission will be 1 bit per 4 clock cycles.

In addition, we need to implement a way for the CPU to be able to recognise if:

1) a byte has been received since the receive register was last read by the CPU.

2) a transmission is in progress.

3) there was a receive overflow. (a byte was received before the previous byte was read by the CPU)

4) there was a transmit overflow. (a byte was written by the CPU before the previous byte had finished transmission)

The advanced portion of this implementation is to add an interrupt function, which is explained in greater detail later.

2.2. Floating-Point Arithmetic

Arithmetic is vital part of a CPU and is therefore greatly advantageous to be able to improve that aspect of our CPU, by implementing **Floating-Point Arithmetic**. It will allow us to represent numbers with more accuracy, and thus be able to have more precise calculation results.

This feature needs to support Addition, Subtraction and Multiplication. Each operation requires a different way of handling, and consequently three different designs should be created.

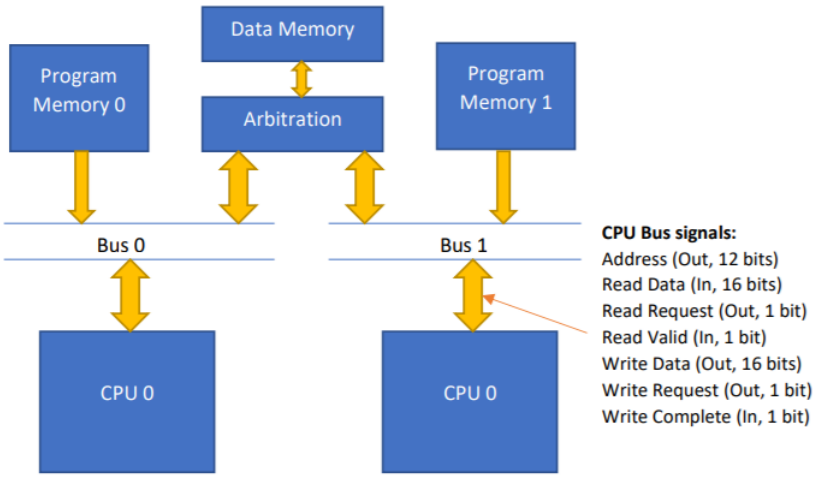
For our implementation, we shall be using the 16-bit IEEE 754 half-precision number format. The 16-bit word which we will be using to represent our numbers, should include 1 sign bit, 11 significant bits and 5 exponent bits.

We are also designing the advanced segment of the Floating-Point Arithmetic feature, which uses 32-bit IEEE 754 single precision number format. In contrast to the previous number representation, for this one we use 1 sign bit, 23 significant bits and 8 exponent bits. Both number formats will be explained in more detail later.

2.3. Dual Core CPU

The **Dual Core CPU** is the third and final feature that we must implement. For this integration, we are required to design a way for two CPU cores to be able to share data, whilst having separate instruction memory.

Problems may arise from sharing data, such as when both CPUs try to access the data memory simultaneously. To counteract this problem, we will be including an arbitration logic in our design, which will grant access to the data memory to a specific CPU and force the other CPU to stall and access the data in the next clock cycle.



With the aid of the various Bus signals shown in the figure above, the arbitration logic will be able to recognise when a read or write operation is taking place/has been completed.

The addition of this feature, will allow us to execute something twice as fast since we can now split the workload into two CPUs, working simultaneously.

'... two cores allow two programs to run simultaneously. It's hard to quantify the many complex calculations a computer must execute to create the average browsing experience; however, single-core processors only create the illusion of multitasking through technologies such as hyperthreading.'[2]

[2] Jacob Andrew n.d., *"The Advantages of a Dual Processor on a Computer"*, Chron, accessed 2 June 2021, <https://smallbusiness.chron.com/advantages-dual-processor-computer-70316.html>

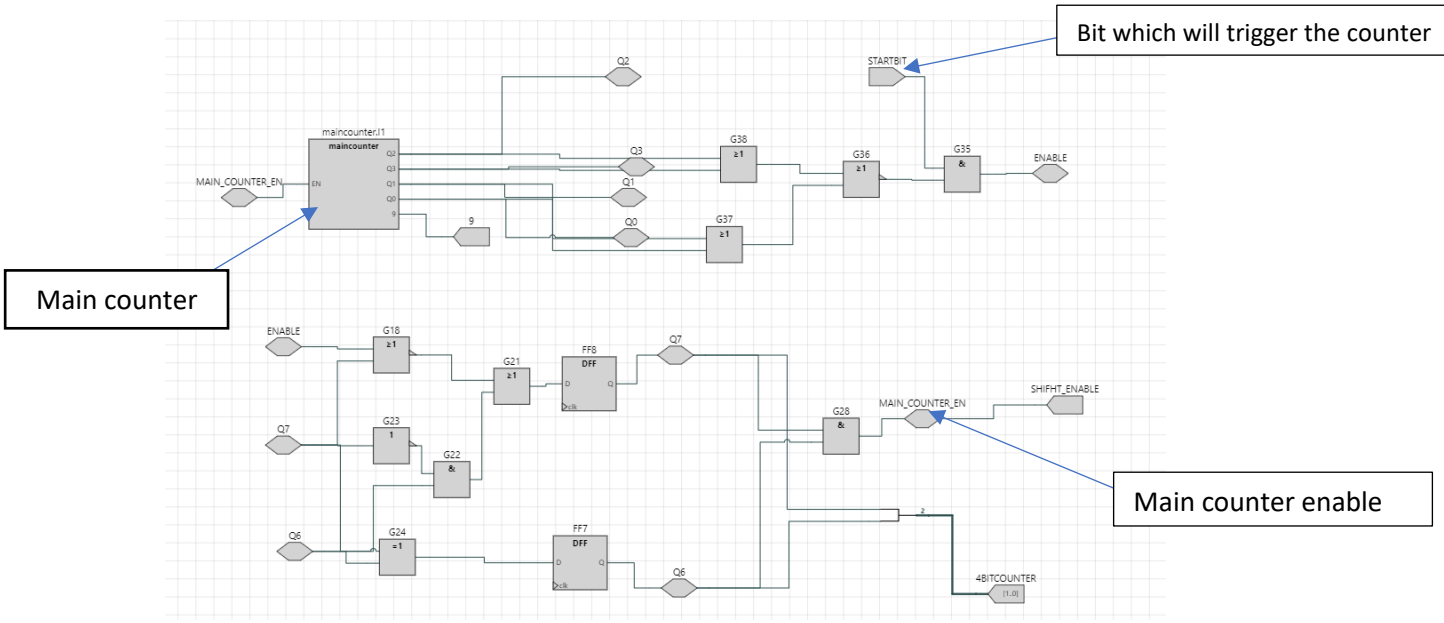
3. UART Implementation

Each byte received and transmitted needs be contained in a packet of 10 bits:

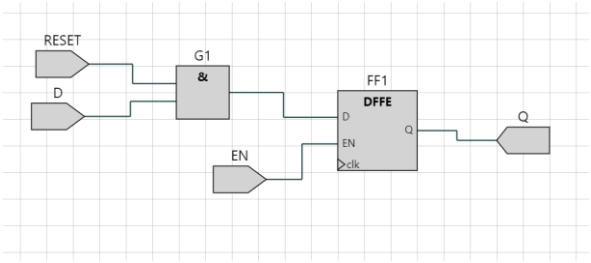
Start bit	Data bit	Data bit	Data bit	Data bit	Data bit	Data bit	Data bit	Data bit	Stop bit

In fact, UART stands for Universal Asynchronous Receiver/Transmitter. This means that we will make use of the start and stop bits, in order to keep track of the state of the UART.

In order to cope with the asynchronous nature of the UART, we had to design a counter which could be a substitute for our state machine. The counter is actually made of two counters, the main counter which can count up to nine and then resets back to zero, and another counter which counts up to four. The second counter is used to force the main counter to change every 4 clock cycles, since the bit rate of the UART is one bit per four clock cycles



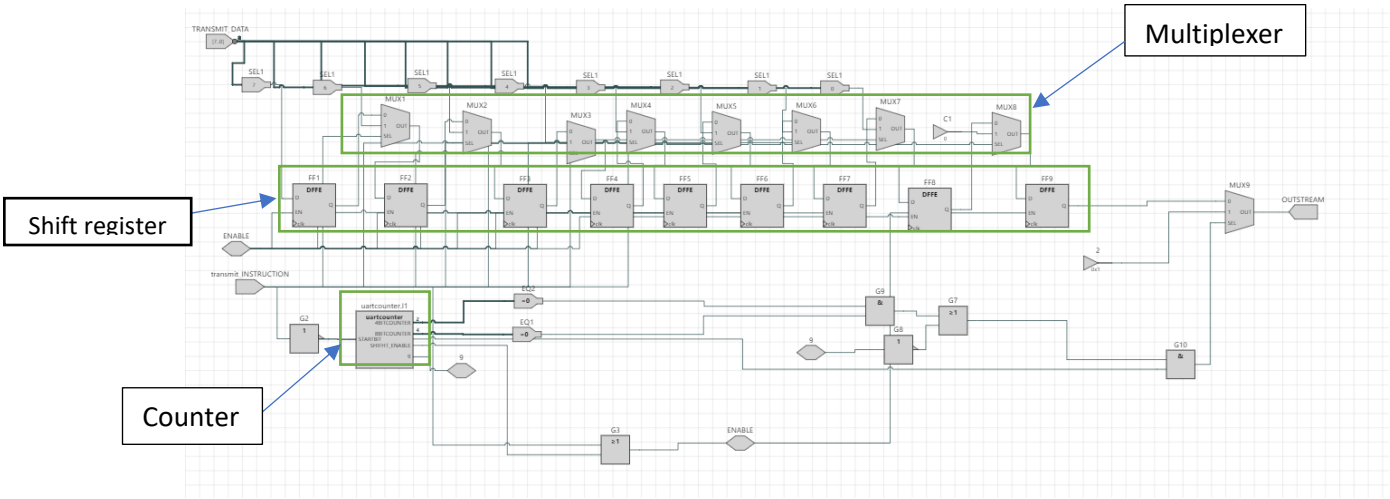
An interesting thing to note, is that we actually designed our own flip flop in order to include a reset:



Even though this feature was not essential, it facilitated the process of designing the UART, as it enabled us to easily change the reset number and efficiently test a variety of options for the counter. (As we were not entirely sure up to which number the counter would count)

3.1. Transmitter and Receiver

Transmitter:



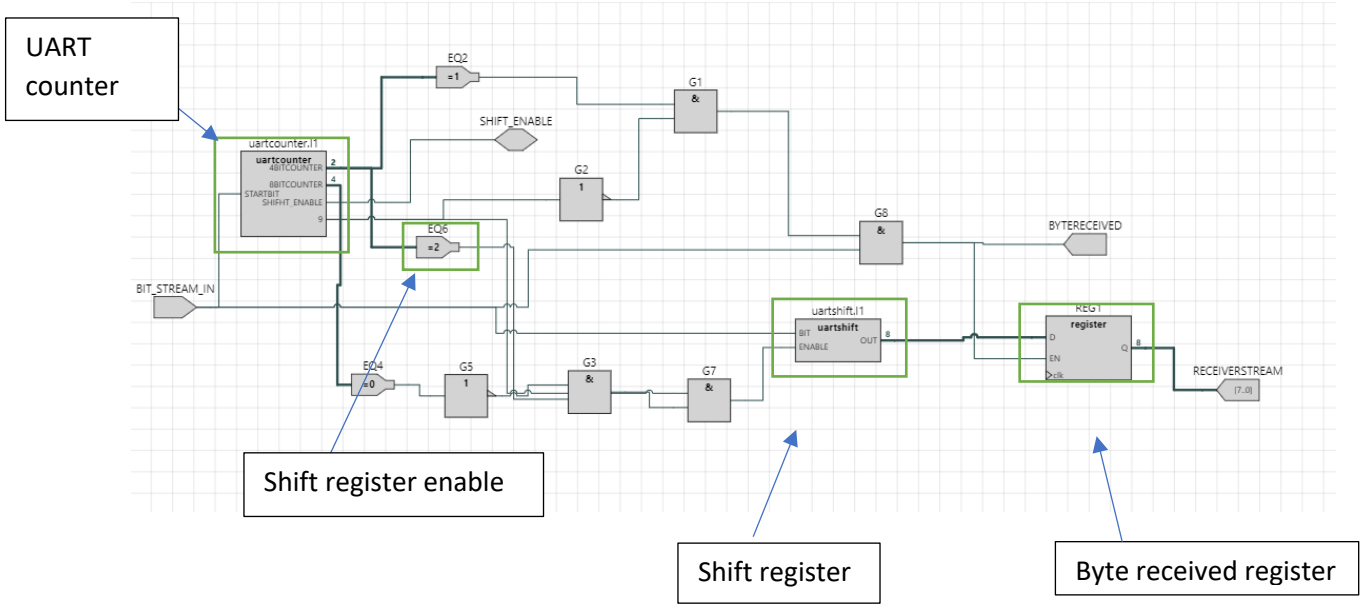
The transmitter is made of 3 main parts:

- **Counter**: The counter is triggered by a transmit instruction coming from the IR block. As the transmission is ongoing, the counter will control every enable input in the block, such as the shift register and the multiplexer output bus.

- **Multiplexer**: There are two ways for the data to change in the shift register. The task of the multiplexer is to choose between the bit which was contained in the previous flipflop or the bit that was fetched by a new transmit instruction.

- **Shift register:** In UART the byte is transmitted bit by bit every 4 cycle. This is why we are using a shift register to shift the bit of the byte as the transmission is going.

Receiver:



The receiver is made out of three main parts:

- **UART counter:** The UART counter is there to control the receiver block. In this case, the block is trigger by a start bit. As the module is receiving, it will enable the uartshift every four cycles.

  We need to choose when the bit will actually be read and stored in the shift register. In fact, the 4 cycles between every bit received, exist to prevent miss-reading a bit. (In the case of some unwanted behaviour during transmission)

- **Shift register:** The shift register's only purpose, is to temporarily store the receiving stream. Like we mentioned previously, it is enabled in the second cycle of the UART counter. During a receive, it only stores the 8 data bits and discards the start and stop bits.

- **Byte receive register:** A byte can only be stored in this register, if the stop bit is valid (Bit is high), otherwise the byte is discarded.

Instruction:

The UART was implemented as a memory map. We decided to follow the suggestion, given by the instruction set, even thought we could have chosen different addresses.

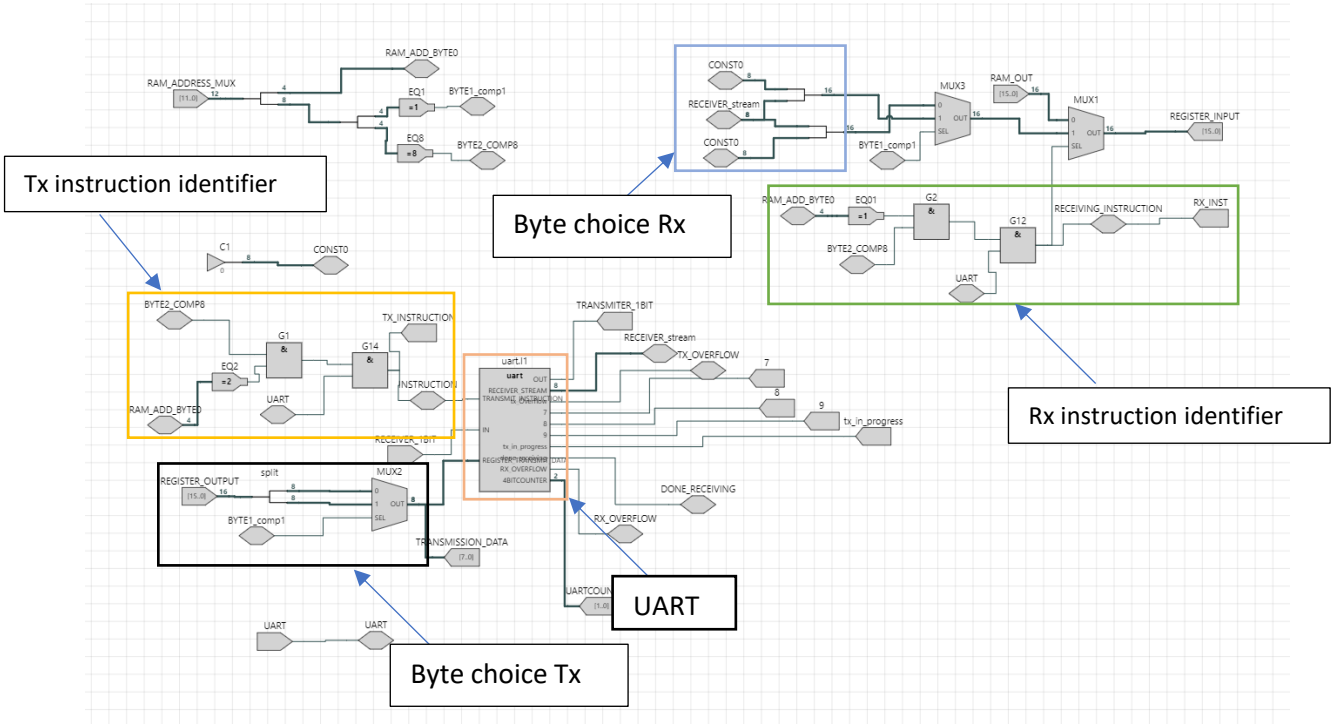
Address	Target
0x000 ... 0x7ff	RAM
0x800	UART Status
0x801	UART Receive
0x802	UART Transmit

Since UART is a memory map, it uses the load at address 0x800 to load status, load at address 0x801 for receive and store at address 0x802 for transmit.

One of our main concerns, was for the transmitter not to be triggered for no reason. To avoid any risks, we decided to use a specific opcode (in this case 0x3) which would be similar to opcode 0x0 (used for load and store), but specific to the UART. Having implemented this feature, we never had any issues with random receiving or transmitting occurring.

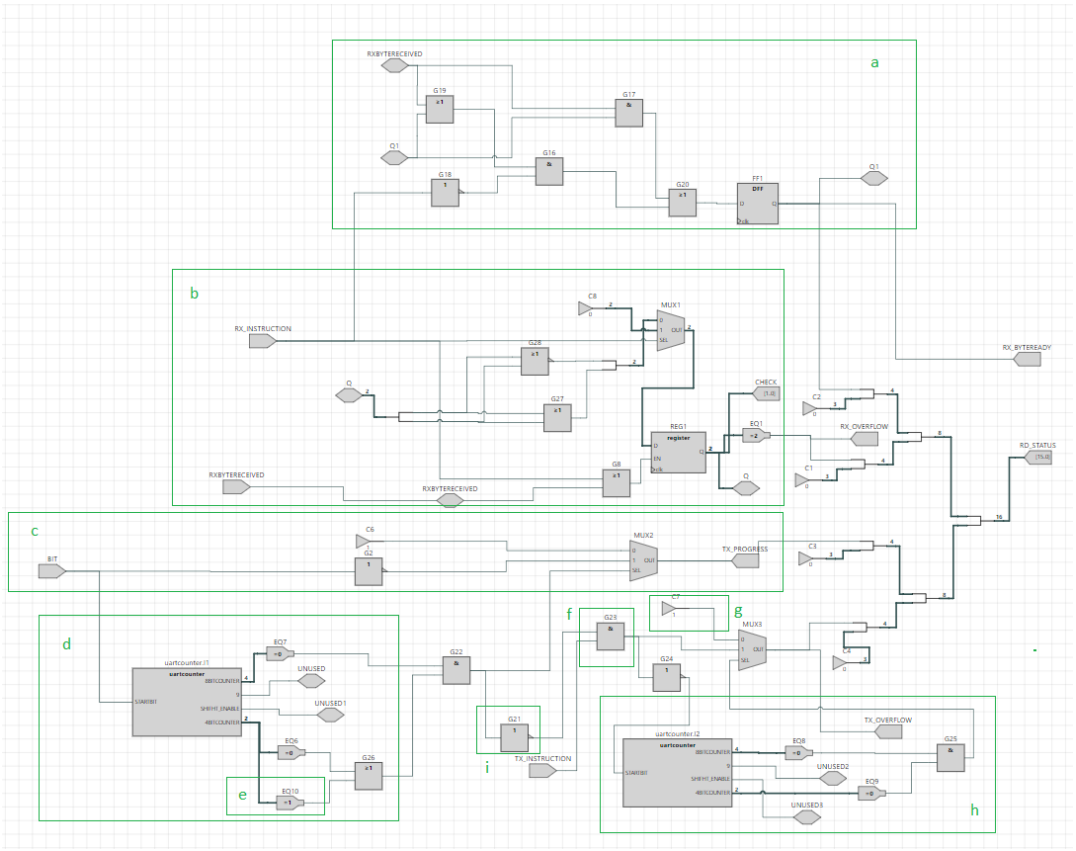
Finally, we also implemented a feature which allowed the coder to choose which byte of a 16-bit word he would like to transmit:

Address	Target
0x801	Receive in byte 0
0x811	Receive in byte 1
0x802	Transmit byte 0
0x812	Transmit byte 1



Complete design of the current stage

3.2. UART Status



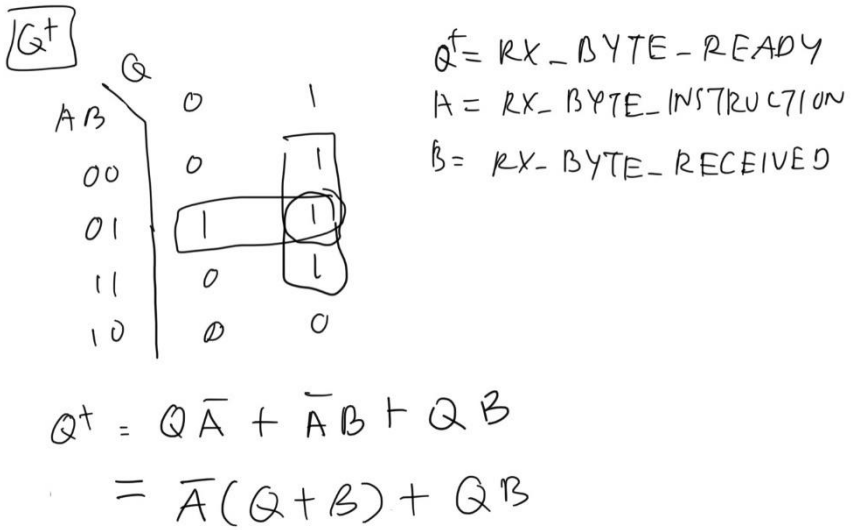
- Provides information to the CPU about the current status of the UART.
- The status consists of four signals:

a: **RX_BYTE_READY** is HIGH when a complete byte has been received by UART receiver. When RX_BYTE_READY is HIGH, we can load the value into a register using LOAD instructions (as specified in the UART receiver explanation). Once it has been received by one of the registers in the register file, the RX_BYTE_READY signal will become LOW. It will turn HIGH again if a new complete byte is received.

RX_INSTRUCTION	RX_BYTE_RECEIVED	Q	Q+ = RX_BYTE_READY
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Truth Table

Logic:



---

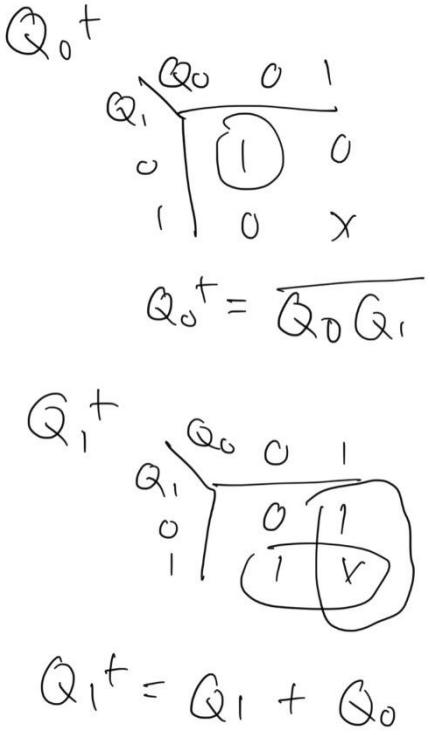
b: **RX_OVERFLOW** is HIGH when two consecutive complete bytes are received by the UART receiver, without the first one being loaded into any register. This signal indicates that the earlier complete byte has been wasted, since a new complete byte has overwritten the UART receiver. There is no way we could retrieve the first complete byte. This signal will become LOW once the new complete byte has been loaded by the CPU. In the advanced part, we will implement a function that can avoid such waste to occur.

Logic:

- Build a counter with enable and reset that counts up to 2.
- RX_INSTRUCTION OR RX_BYTE_RECEIVED as enable.
- Use a multiplexer with RX_INSTRUCTION as select input to become the reset. Constant 0 at input1 of MUX.
- Bus compare (=2) at output to indicate two consecutive RX_BYTE_RECEIVED. This output is RX_OVERFLOW.

Q1	Q0	Q1+	Q0+
0	0	0	1
0	1	1	0
1	0	1	0
1	1	X	X

Counter truth table and logic



TX_IN_PROGRESS is HIGH when the UART transmitter is transmitting a complete byte (immediately goes HIGH when the "start bit" starts being transmitted). Transmission of a complete byte takes 40 cycles, because each bit takes 4 cycles to be transmitted. Once the "last bit" has been transmitted, TX_IN_PROGRESS goes back to LOW.

Logic:

- UART counter in d starts count when the BIT becomes 0 (indicating start bit).
- When a start bit is received, MUX2 at c is used to make TX_IN_PROGRESS goes HIGH when counter = 0 (because counter will only change to 1 in the next cycle).
- And since, it takes 40 cycles to complete a transmission, make the TX_IN_PROGRESS stay HIGH until the counter = 0 again using NOT gate and Bus Compare.

TX_OVERFLOW is HIGH when a new transmitting instruction is executed while TX_IN_PROGRESS is HIGH. This scenario will cause the new complete byte that will be transmitted overwrites the old complete byte that is being transmitted. The earlier transmission will stop immediately, and the new transmission will proceed. Once the new transmission is complete, the TX_OVERFLOW will become LOW. In the advanced part, we are going to implement a function to avoid such things from occurring.

Logic:

- TX_OVERFLOW indicates when there is a new transmission started before the previous transmission ended. By ANDing TX_IN_PROGRESS and TX_INSTRUCTION, TX_OVERFLOW can be obtained. However, for normal transmission, TX_INSTRUCTION = 1 at cycle 0 and 1 of each transmission. To differentiate this case, add a bus compare (= 1) at e (this will not affect the functionality of TX_IN_PROGRESS because start bit is 4 cycles long).

  The output of i is high when the counter is not at cycle 0 or 1. At f, AND this with TX_INSTRUCTION to obtain TX_OVERFLOW. NOT(TX_OVERFLOW) to activate the UART counter at h. MUX3 selects input0 for the next 39 cycles so TX_OVERFLOW stays high for 40 cycles (at cycle 0, input1 is TX_OVERFLOW).

How does the UART status give information to the CPU? It stores the 4 signals as 16 bits in STATUS REGISTER. This value can be loaded using LOAD instruction, specifically LOAD RD =MEM[0x800]. The format of the 4 signals as 16 bit is as follows:

15:13	**12**	**11:9**	**8**	**7:5**	**4**	**3:1**	**0**
000	TX_OVERFLOW	000	TX_IN_PROGRESS	000	RX_OVERFLOW	000	RX_BYTE_READY

Possible cases:

TX_OVERFLOW	**TX_IN_PROGRESS**	**RX_OVERFLOW**	**RX_BYTE_READY**	**STATUS REGISTER (HEX)**
0	0	0	0	0x0000
0	0	0	1	0x0001
0	0	1	1	0x0011
0	1	0	0	0x0100
0	1	0	1	0x0101
0	1	1	1	0x0111
1	1	0	0	0x1100
1	1	0	1	0x1101
1	1	1	1	0x1111

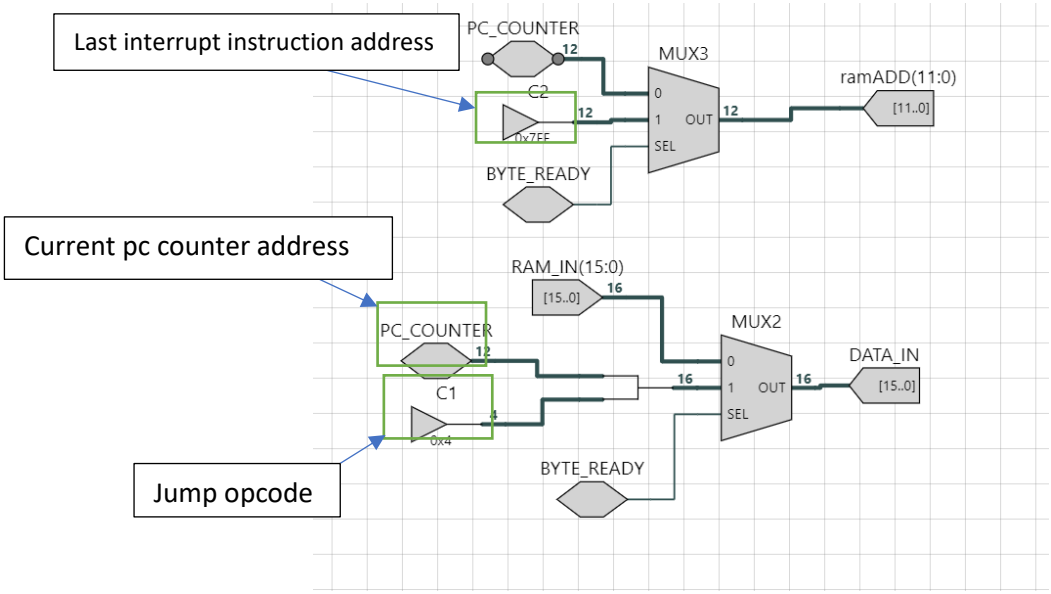
3.2. Advanced Interrupt

The implementation of the interrupt was divided into two parts:

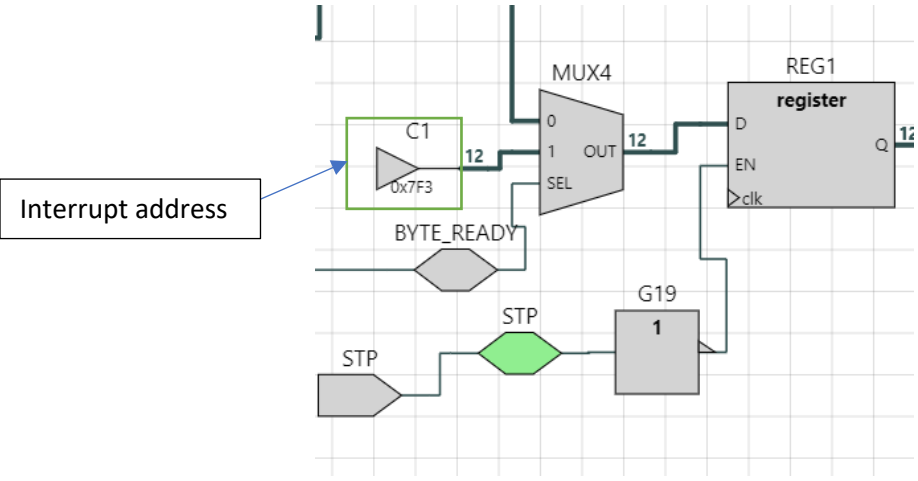
- hardware
- software

We had not seen any decent implementation of the interrupt, without any additional logic. So, our interrupt is made in a way to run as soon as a byte is ready. This is why we needed some change in our design.

- Control path: As soon as we have a byte ready, we store a jump instruction containing the current RAM address, which will be used at the end of the interrupt.



- pc counter: We also have to fetch the address from where the code of the interrupt started.

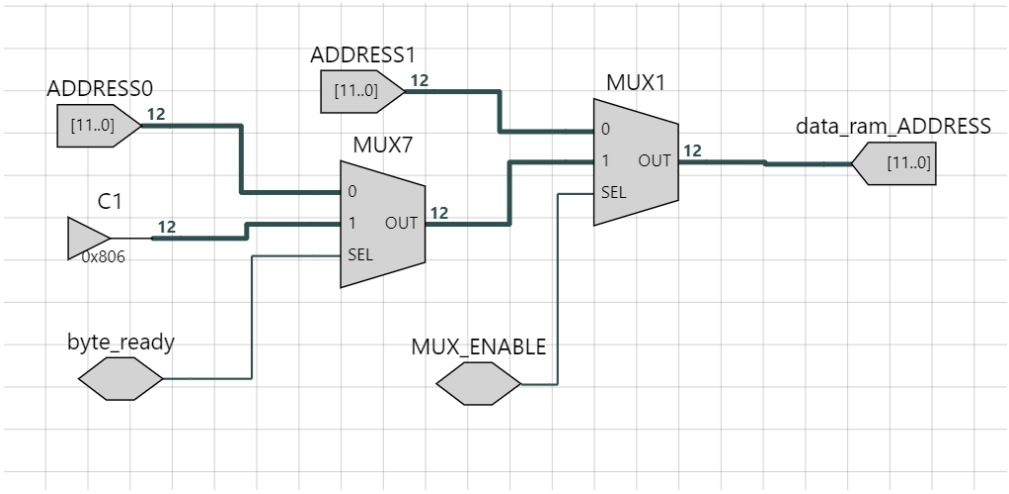


Next, we had to implement the code:

The main thing to consider, is that we will need to restore the register, after the interrupt, exactly as it was at the beginning. Therefore, the first we needed to do was to store the register value in address:

Register	Address store
R0	0x803
R3	0x804
Sum of received byte	0x805
R2	0x806

The main issue, was that the load and store instruction require a register which contains the address namely Rm. However, if we store, for instance, the address 0x800 into one of the register Rm, we would then potentially loose important data, as we would not be able to store it in a specific address beforehand. This is why in the Dual Core feature, we decided to add extra logic in order to store the register data in the RAM location 0x806, before loading the address, so it could be recovered after the interrupt.



3.3. Testing

The following test was created, in order to test whether the Receiver and Transmitter are working correctly. Further testing was done in the Benchmark Evaluation, which also included the interrupt function.

These test results were obtained by using the code which can be found in Appendix E. (As well as the waveforms)

Note: Stimulus block: 0xAB with 0 start bit and 1 end bit 3 times. (RX_DONE_RECEIVING triggered 3 times)

Result	Cycle	Waveform
R1 = 0x0004	3	1
R1 = 0x0804	6	1
R3:= 0X0000	9	1
R3:= 0X0001	51	2
R0:=0X00AB	54	2`
R3:= 0X0000	57	2
Transmission begins with 0 as start bit (from cycle 58 to cycle 97)	58	2
R3:= 0X0100	63	3
R3:= 0X0001	102	4
R3:= 0X0011	147	5
R0:=0XAB00	150	6
R3:= 0X0000	153	6
Transmission begins with 0 as start bit (affected by next transmission)	155	6
R3:= 0X0100	159	6
Transmission begins with 0 as start bit (cause an overflow)	161	6
R3:= 0X1100	165	6
Program stops	168	6
TX_OVERFLOW ends 40 cycles (a complete transmission time length) after being triggered	199	7

Conclusion:

- All results meet the initial expectations.
- UART receiver received the byte and was able to load it into the register.
- UART transmitter was able to transmit the byte. (when there was no overflow).
- A byte was lost due to RX_OVERFLOW. The last transmission did not work, as intended, due to TX_OVERFLOW. These problems will be solved using an interrupt function.

4. Floating-Point Arithmetic Implementation

4.1. Research

For the research portion of the implementation, we focused on finding out more about the IEEE 754 floating point number format, and how addition/subtraction and multiplication can be implemented with this new format. (As well as finding the differences between the 16- and 32-bit representation)

'The IEEE has standardized the representation for binary floating-point numbers as IEEE 754 and this standard is generally used in modern computers.[3] Precision denotes the accuracy of a given floating-point number. There are several formats with increasing levels of accuracy. In this part of the project, we are going to implement the 16 (Half) and 32 (Single) bits FP precisions. The 32 and 64-bit implementations are the most widely used in modern microprocessors.'[4]

In this section of the report, we will not be talking about the advanced portion, since the implementation is the same. The only thing that changes, is the number of bits that are being used. The 32-bit IEEE 754 single precision is however being used for some of the testing and is being showcased in the video as well. For more information on this type of number representation, you can visit Appendix A.

4.2. Half Precision Floating-Point

Two inputs of the floating-point arithmetic block: RD and OP2

Output of the floating-point arithmetic block: ANS

The inputs and the outputs of the floating-point arithmetic block are encoded according to the "binary16" interchange format [5] , illustrated in the table below.

.

Sign (S)	Exponent (E)	Significand (F)
1 bit	5 bits	10 bits

IEEE "binary16" floating point format

'Sign: Sign is a single bit that determines the sign of the number, '0' for positive and '1' for negative.

Exponent: Exponent has five bits to indicate the exponent value with the bias.

Significand: Significand has eleven bits which includes a hidden bit. If significand is represented as 'F', the representation for a Normal form (learn more about Normal and Subnormal form in Appendix A) number is '1.F', where 1 is the hidden bit.'[6]

[3] "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008, vol., no., pp.1-70, 29 Aug. 2008, doi:10.1109/IEEESTD.2008.4610935.

[4] Kannan, Balaji, "THE DESIGN OF AN IC HALF PRECISION FLOATING POINT ARITHMETIC LOGIC UNIT" (2009). All Theses. 689., p 4 - 6, accessed 8 June 2021, <https://tigerprints.clemson.edu/all_theses/689>

[5] "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008, vol., no., pp.1-70, 29 Aug. 2008, doi:10.1109/IEEESTD.2008.4610935.

[6] Kannan, Balaji, "THE DESIGN OF AN IC HALF PRECISION FLOATING POINT ARITHMETIC LOGIC UNIT" (2009). All Theses. 689., p 4 - 6, accessed 8 June 2021, <https://tigerprints.clemson.edu/all_theses/689>

4.3. Half Precision Floating-Point Addition and Subtraction

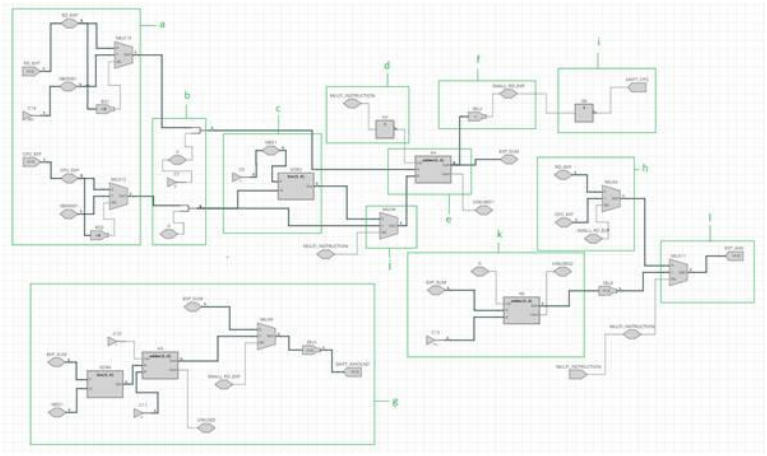
The algorithm used in this block follows the standard implementation of floating-point addition. A useful source that describes this algorithm specifically is available in [7].

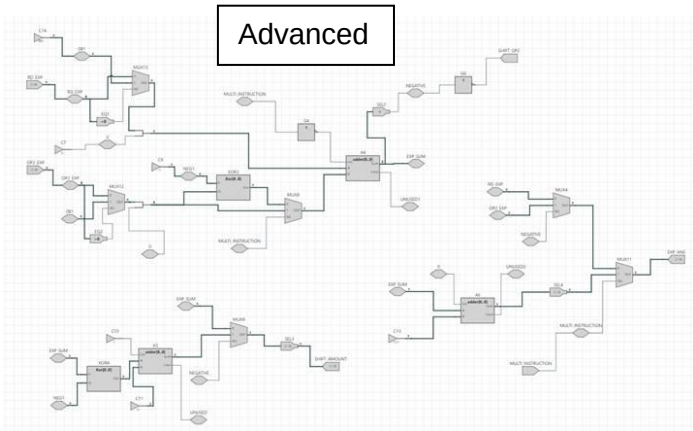
Note: MULTI_INSTRUCTION is always 0 during addition and subtraction instruction. SUB_INSTRUCTION is 1 when subtraction and 0 when addition.

First step: Input comparison and manipulation

The goal of this step is to match the smaller exponent with the larger exponent by manipulating the significand of the input with the lower exponent, without changing the number it represents. Exponent, pre-shift, right shift and final significand block are required in order to complete this step.

Exponent Block





a: A multiplexer and a bus compare that detects 0 are used to identify Subnormal case. Since the actual E value of Subnormal case is equals to the Normal case when E = 1, we select E = 1 instead of 0 because we are only interested in the magnitude difference of the "actual E value" between the two input exponents.

b: The exponent for each input is zero extended by 1 bit at the MSB using a merge wire to give them two's complement representation. This bit determines the sign of the exponent (0 for positive, 1 for negative).

Sign	Exponent
1 bit	5 bits

Exponent bits in two's complement

c: By XORing OP2's exponent with -1 and adding it by 1 (at CIN of the adder at d), we obtain its negative form.

e: We add RD's exponent with negative OP2's exponent to obtain the difference between them (may be positive or negative).

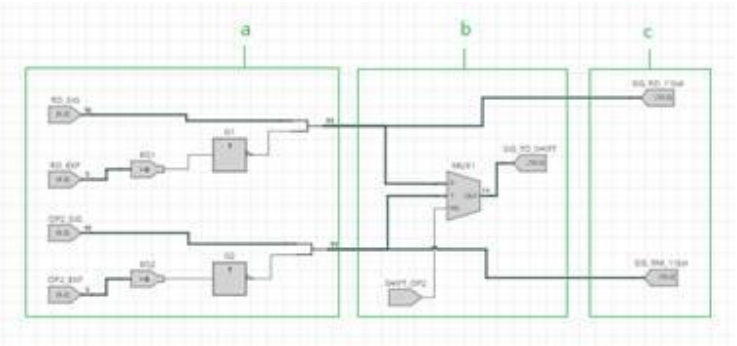
[7] *CS 301 Class Account Mon Sep 13 11:15:41 ADT 1999,* "Addition and Subtraction", UAF Computer Science, accessed 8 June 2021, <https://www.cs.uaf.edu/2000/fall/cs301/notes/notes/node51.html>

f: A bus select that checks the MSB of the difference, is used to determine its sign. The output of the bus select is called SMALL_RD_EXP. SMALL_RD_EXP = 1 means that the difference is negative and RD's exponent is smaller than OP2's. For this case, at g, a multiplexer with a select input SMALL_RD_EXP is tasked to select the inverted plus one form (from negative to positive) of the difference to obtain its magnitude. Otherwise, the sum remains. The output of the difference is labelled SHIFT_AMOUNT. SHIFT_AMOUNT indicates how many times does the smaller input significand need to be shifted to the right.

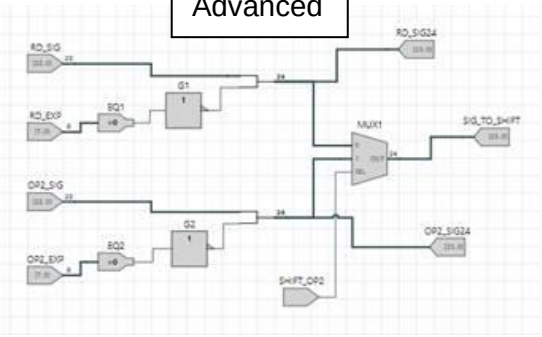
h: A multiplexer is used, to select the larger exponent as the pre-answer's exponent, labelled EXP_ANS. This exponent is not final because there are several cases that require manipulation of exponent in the upcoming steps.

i: We NOT the SMALL_RD_EXP to give SHIFT_OP2 output, which indicates when the OP2's significand needs to be shifted.

Pre-shift Block



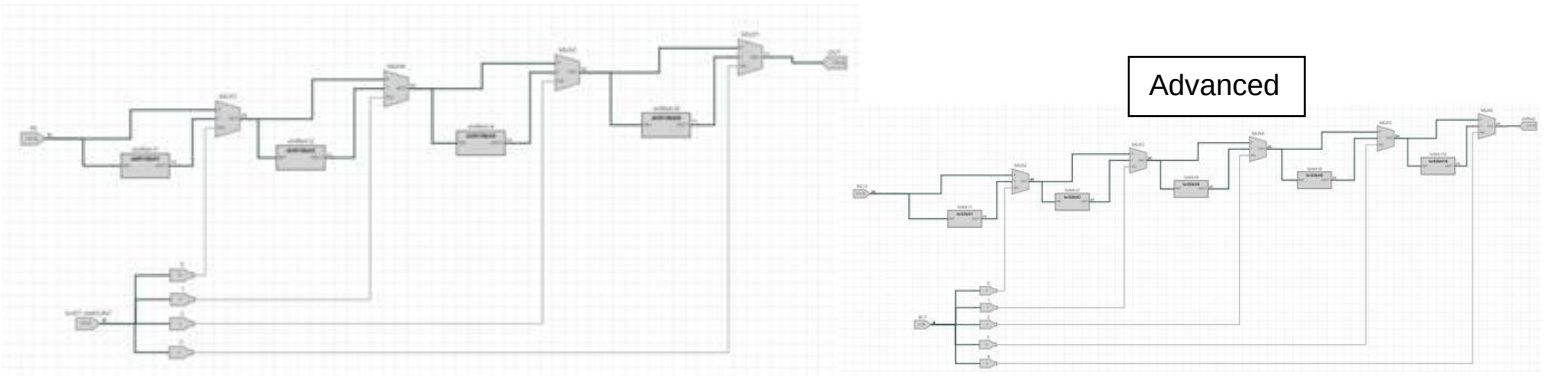
Advanced



a: The hidden bit is added to the two inputs significand using a merge wire and a bus compare that detects 0. The hidden bit is always 1 except for the Subnormal case (E = 0).

b: A multiplexer with SHIFT_OP2 as the Select input is used to choose the significand that needs to be shifted, and the output is labelled as SIG_TO_SHIFT. SIG_TO_SHIFT will be the input for the next block, the Right Shift Block. The outputs in c, are the significands of RD and OP2 (labelled RM) which includes the corresponding hidden bit.

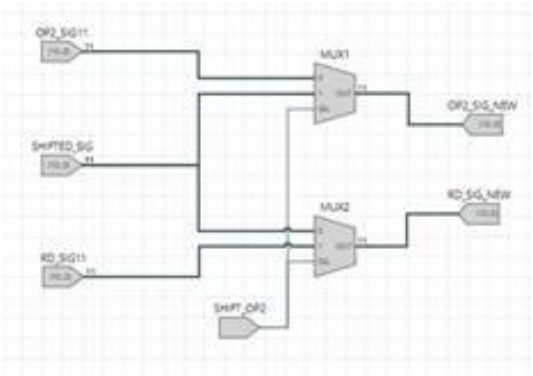
Right Shift Block



This block shifts the significand (SIG_TO_SHIFT from Pre-shift Block) to the right N times, where N is equals to the SHIFT_AMOUNT (from Exponent Block). The output OUT is the shifted significand.

(This idea was inspired by our previous work on the OP2 Block.)

Final Significand Block

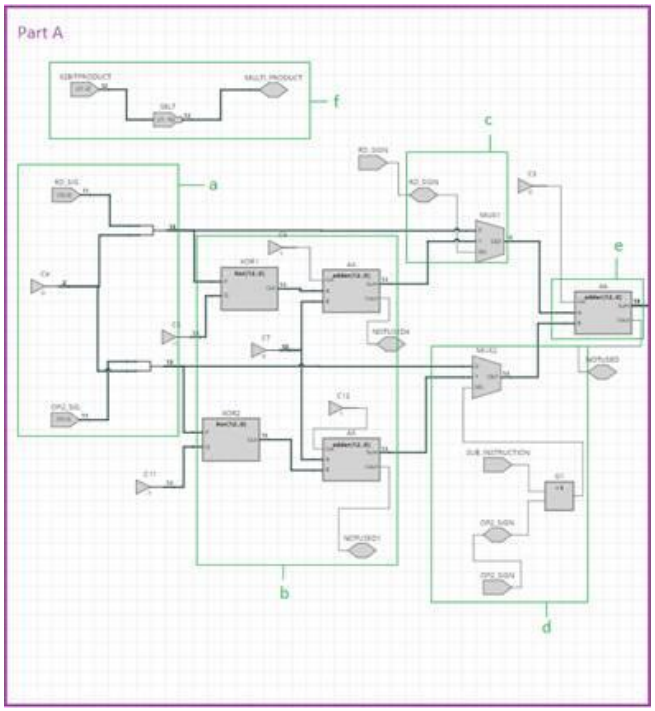


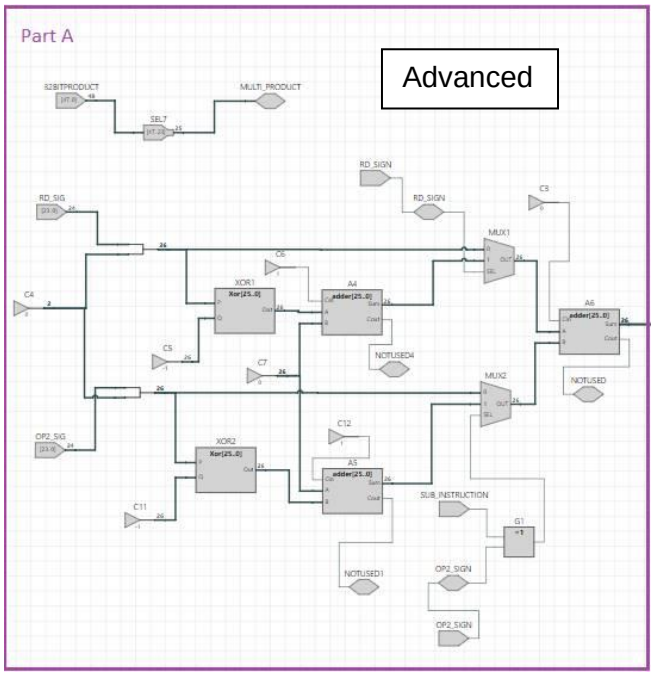
This block decides the significand of RD and OP2 at the final stage before entering Floating Arithmetic Block. If SHIFT_OP2 (from Exponent Block) is 1, MUX1 will select the shifted significand (OUT from Shift Block) as the new OP2's significand while MUX2 will select the original RD's significand as the new RD's significand.

Second step: Significand addition

The goal of this step is to perform either addition or subtraction by manipulating the significands based on their corresponding sign bit and the arithmetic instruction itself. For example, when we need to perform a subtraction operation, SUB_INSTRUCTION = 1 (refer Note). Part A of the Floating Arithmetic Block is required to complete this step.

Floating Arithmetic Block (Part A)





a: Both significands are zero-extended by 2 bits. The first bit is the overflow bit (to check overflow for Normal case) and the other bit gives two's complement representation to the significand.
b: Both significands are inverted into its negative form using XOR gates and adders. Part b gives the flexibility for the multiplexer in c and d to choose RD and OP in either positive or negative form.

c: RD_SIGN acts as the Select input of the multiplexer, when it is 1, it will choose the negative form and vice versa.

d: The Select input for the multiplexer is not only OP2_SIGN because we need to include the possibility of having a SUB instruction. A subtraction while OP2 is negative cancels out the minus (addition). Hence, the select input of the multiplexer should be OP2_SIGN XOR SUB_INSTRUCTION.

e: The adder completes the addition (or subtraction).

Two's complement sign bit	Overflow bit	Significand including hidden bit
1 bit (MSB)	1 bit	11 bits

Significand in two's complement form

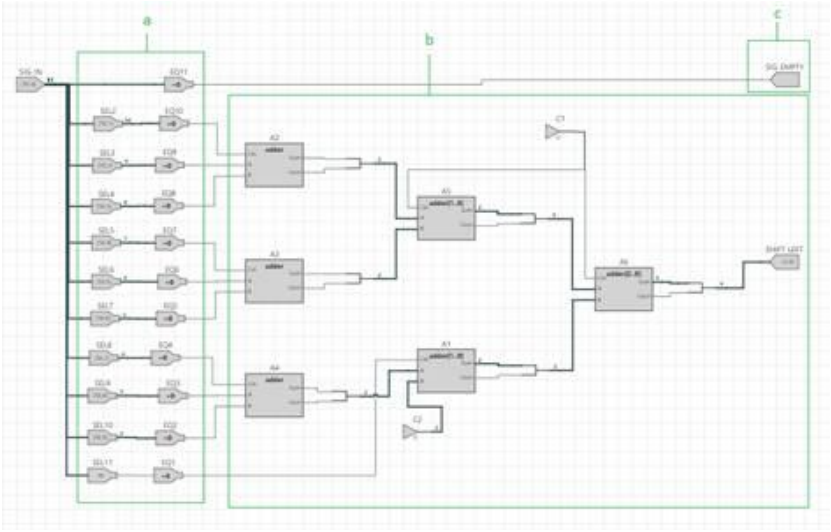
OP2_SIGN	SUB_INSTRUCTION	XOR output (MUX2 select input)
0	0	0 (positive)
0	1	1 (negative)
1	0	1 (negative)
1	1	0 (positive)

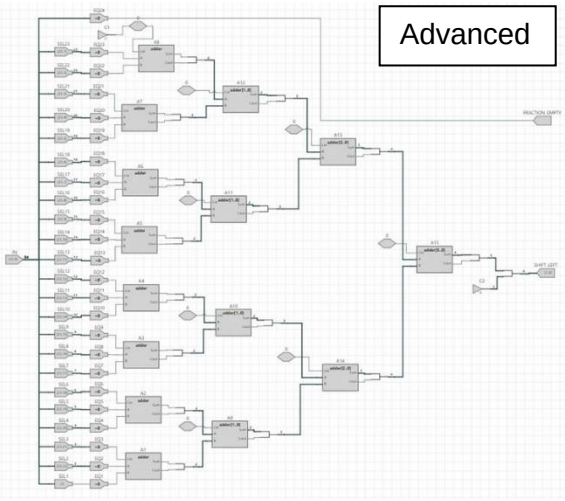
OP2_SIGN XOR SUB_INSTRUCTION truth table

Third step: Normalization

The goal of this step is to have an answer that obeys the binary16 format (such as having 1 as the hidden bit for Normal case). There are two general cases that require us to handle it with care. There are called 'significand underflow' and 'significand overflow'. We will use the shorthand and just call it Underflow and Overflow instead. Part B of the Floating Arithmetic Block, MS Zero Identifier Block and Left Shift Block are required to complete this step.

MS Zero Identifier Block





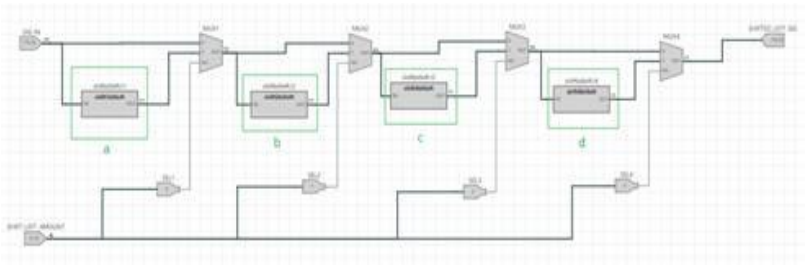
a: Bus compares (= 0) are used to detect how many leading zeroes does the significand have from SIG[11] to SIG[0]. It will stop searching for the next zero once "1" is present in between SIG[11] and SIG[0].

b: The adder sums up the number of leading zeroes present in the significand. The sum output is labelled as SHIFT_LEFT. SHIFT_LEFT indicates the number of times the significand needs to be shifted to the left due to underflow. If the whole significand is filled with zeroes, the output SIG_EMPTY at c will be high (1). SIG_EMPTY indicates the significand is empty (only filled with zeroes).

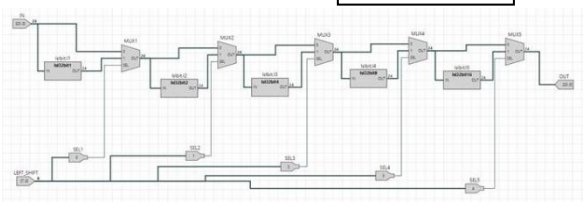
SIG10	SIG9	SIG8	SIG7	SIG6	SIG5	SIG4	SIG3	SIG2	SIG1	SIG0	SHIFT_LEFT (DECIMAL)	SIG_EMPTY
1	X	X	X	X	X	X	X	X	X	X	0	0
0	1	X	X	X	X	X	X	X	X	X	1	0
0	0	1	X	X	X	X	X	X	X	X	2	0
0	0	0	1	X	X	X	X	X	X	X	3	0
0	0	0	0	1	X	X	X	X	X	X	4	0
0	0	0	0	0	1	X	X	X	X	X	5	0
0	0	0	0	0	0	1	X	X	X	X	6	0
0	0	0	0	0	0	0	1	X	X	X	7	0
0	0	0	0	0	0	0	0	1	X	X	8	0
0	0	0	0	0	0	0	0	0	1	X	9	0
0	0	0	0	0	0	0	0	0	0	1	10	0
0	0	0	0	0	0	0	0	0	0	0	10	1

Truth table of MS Zero Identifier Block

Left Shift Block

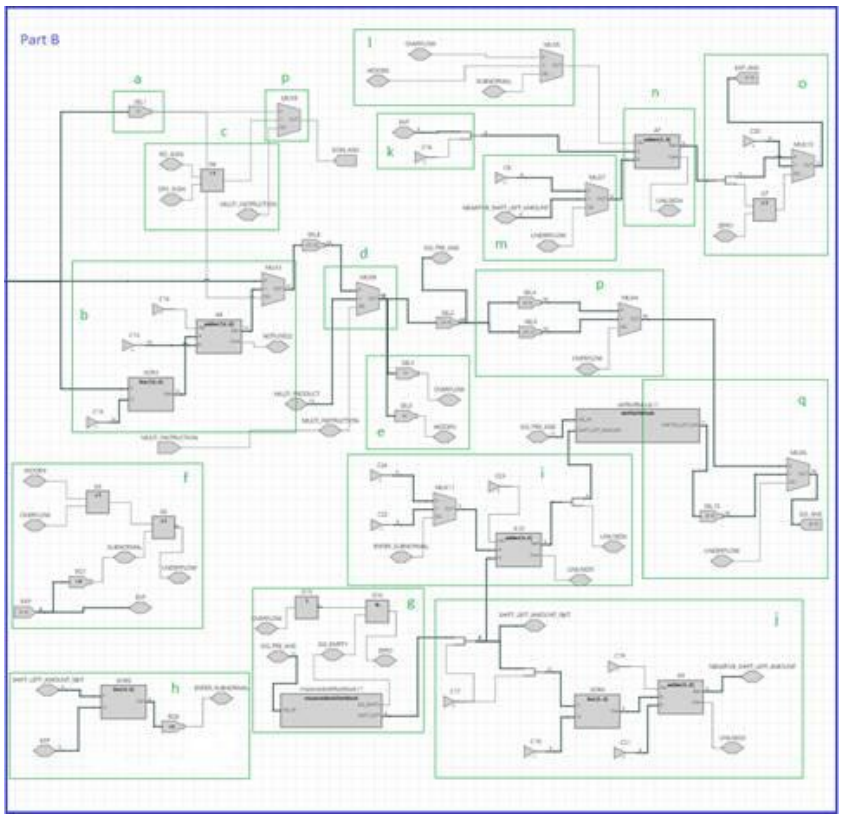


Advanced

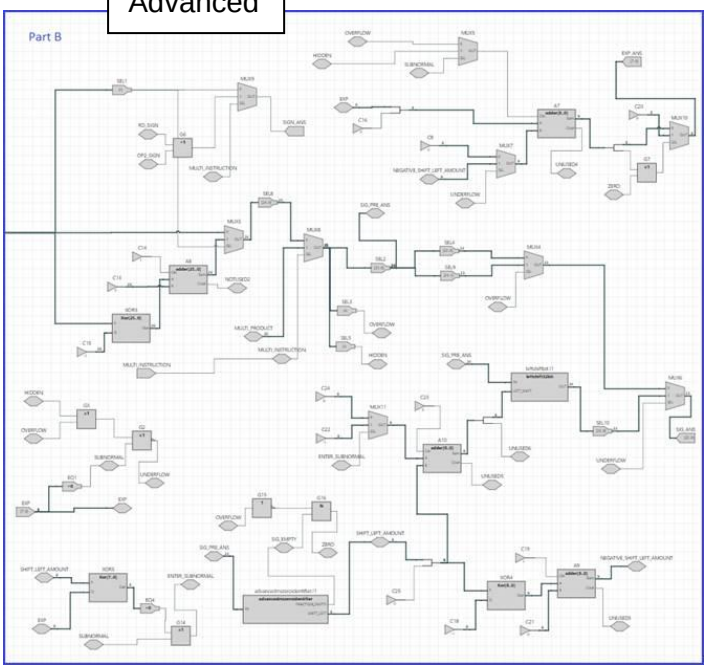


This block shifts 11 bits Sig_IN to the left N times, where N is equal to the SHIFT_LEFT_AMOUNT input. Every time the significand is shifted to the left, a zero will be inserted at the LSB. Sub-block a will shift by one bit, sub-block b shifts by two bits, sub-block c shifts by for bits, and sub-block d shifts by 8 bits.

Floating Point Arithmetic Block (Part B)



Advanced



a: A bus select is used to check the MSB (sign) of the sum from Step 2. The output determines the SIGN_ANS.

b: The multiplexer will select the inverted sum if the sum is negative because in binary16, we are using sign-magnitude form instead of two's complement. The multiplexer at d selects input 0.

e: Two bus selects are used to determine the overflow and the hidden bit.

f: A combinational logic is used to determine when SIG_PRE_ANS (significand including the overflow bit) is underflow.

Overflow bit	Hidden bit	Significand	Type	Underflow
0	0	XXXXXXXXXX	Normal (Subnormal = 0)	1
0	0	XXXXXXXXXX	Subnormal	0
0	1	XXXXXXXXXX	Normal (Subnormal = 0)	0
0	1	XXXXXXXXXX	Subnormal	0

Underflow logic

g: The MS Zero Identifier sub-block determines the number of leading zeroes in SIG_PRE_ANS for in the event of underflow. For example, 0b1111111111 – 0b11111111110 = 0b00000000001. The MS Zero Identifier sub-block determines how many times the SIG_PRE_ANS needs to be shifted to the left until the "1" reaches the hidden bit (for Normal case).

h: A XOR gate and a bus compare (= 0) acts as a comparator to determine whether SHIFT_LEFT_AMOUNT is equal to the EXP or not. If this is true, the last shift of the SIG_PRE_ANS is not needed since the exponent will change from 1 to 0, entering the Subnormal form (signal labelled ENTER_SUBNORMAL).

EXP	SHIFT_LEFT_AMOUNT	SIG_PRE_ANS	SHIFTED LEFT
5	5	0b (0).0000100000	None
5 – 1 = 4	4	0b (0).0001000000	Once
4 – 1 = 3	3	0b (0).0010000000	Twice
3 – 1 = 2	2	0b (0).0100000000	3 times
2 – 1 = 1	1	0b (0).1000000000	4 times
1 – 1 = 0	1	(If the significand is shifted one more time, the hidden bit will be 1 when E = 0, which is incorrect)	4 times

Example when EXP is equals to SHIFT_LEFT_AMOUNT

i: SHIFT_LEFT_AMOUNT will be subtracted by 1 if the ENTER_SUBNORMAL = 1 before it goes to the Left Shift sub-block to complete the left shifting on SIG_PRE_ANS.

Each time the SIG_PRE_ANS is shifted to the left, the EXP is subtracted by 1.

j: SHIFT_LEFT_AMOUNT is inverted to its negative form.

k: The EXP is given two's complement representation.

l: EXP will be added by 1 when Overflow occurs (the multiplexer differentiates Normal and Subnormal Overflow).

n: EXP – LEFT_SHIFT_AMOUNT (from m) when Underflow.

o: The OR gate identifies ANS = 0 and out of range case (when EXP – LEFT_SHIFT_AMOUNT < 0). EXP_ANS is obtained.

p: A multiplexer with OVERFLOW as select input will select the right shifted SIG_PRE_ANS in the event of overflow.

q: A multiplexer with UNDERFLOW as select input will select the left shifted SIG_PRE_ANS in the event of underflow. If both cases are not true, SIG_PRE_ANS remains. SIG_ANS is obtained.

4.4. Half Precision Floating-Point Multiplication

Note: MULTI_INSTRUCTION is always 1 during multiplication operation

The algorithm used in this block follows the implementation of floating-point multiplication as described in [8].

First step: Exponent addition.

The goal of this step is to add up the two input exponents while maintain the binary16 format. Exponent block is needed to complete this step.

For example, when multiplying 2^14 and 2^4, the products exponent equals to 14 + 4 = 18. However, the actual E value in binary16 format equals to E – 15. Based on the previous example, if the actual E value is 14 and 4, the two input Es will equal to 29 and 19, respectively.

If we directly add 29 and 19, we get 48. By comparing it with the answer's actual E value, 18, the value is now biased by 30 instead of 15. This is because when we sum the two Es, the bias also doubled. So, after adding the two input Es, we have to subtract 15 at the end to maintain the 15 biased E.

The following are in reference to the Floating-Point Arithmetic Block (Part B):

j: The multiplexer will select the initial OP2's significand instead of its inverted form.

d: CIN will be 0. The adder at e will sum the two input exponents. The output is labelled EXP_SUM. As the EXP_SUM now is biased by 30, we have to subtract it by 15 at k using an adder an a constant -15. The bus Select between k and l will discard the MSB (two's complement representation sign bit from a) of the exponent.

l: When performing multiplication instruction (MULTI_INSTRUCTION =1), the multiplexer will select the output of the bus select mentioned earlier. Pre-answer's exponent labelled EXP_ANS is obtained (similar to addition, this is not final because it may change in Normalization step).

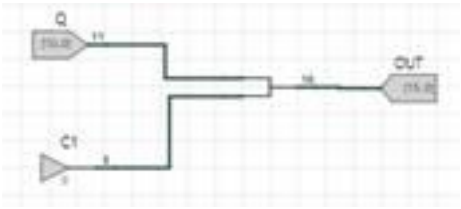
Second step: Significand multiplication and manipulation.

The goal of this step is to multiply the two input significands (including the hidden bit) and keep the most significant 12 bits of the product to be processed in the next step. Pre-shift, Zero-extend, Multiplication (from Datapath) and Floating Arithmetic Block (Part A) are needed to complete this step.

Multiplying two 11-bit numbers, will produce at most a 22-bit product. The most significant 12 bits of the product will be normalized in the Floating Arithmetic Block (Part B) .

[8] rajkumarupadhyay515, Last Updated: 04 May 2020, "Multiplying Floating Point Numbers", Geeks-for-Geeks, accessed 9 June 2021, <https://www.geeksforgeeks.org/multiplying-floating-point-numbers/>

Zero Extend Block



This block zero extends the 11-bit input by 5 bits at the MSB. The output is 16 bits with at least 5 leading zeroes. In Pre-shift block, the outputs at C, RD's and OP2's significands will be zero extended by 5 bits using this block to make it into 16 bits.

This is because we want to use the Multiplication Block that is 16-bit long in the Datapath, to multiply these significands. This will not be a problem because the most significant 10 bits of the product will be all zeroes, so we can discard it afterwards.

Multiplication Block

The two 16-bit inputs are multiplied together, producing a 32-bit output. The outcome will then be the input of the Floating Arithmetic Block (Part A), named 32BITPRODUCT.

f: A bus select that select bit 21:10 is used, to discard the 10 leading zeroes and the 10 least significant bits.

31:22 (Discarded)	21 (Kept)	20 (Kept)	19:10 (Kept)	9:0 (Discarded)
0000000000 (product of the 5 zero extended bits)	X (Overflow bit)	X (Hidden bit)	XX XXXX XXXX	Out of range

Third step: Normalization

The goal of this step is similar with the normalization step of addition and subtraction. There is one extra aim for this step, which is to obtain the sign of the product because the logic that provides that is built here, Part B of the Floating Arithmetic Block.

C: As S = 0 represent positive and S = 1 represent negative, XOR RD_SIGN with OP2_SIGN to obtain the product sign.

RD_SIGN	OP2_SIGN	SIGN_ANS (for multiplication)
0	0	0
0	1	1
1	0	1
1	1	0

RD_SIGN XOR OP2_SIGN truth table

d: The multiplexer will select the 12-bits output from step 2. This 12-bits will be processed the same way as the third step of floating arithmetic addition and subtraction.

4.5. Testing

MU0ARM Level Testing (For Block Level Testing, refer to Appendix B)

Algorithm used for testing: Maclaurin series of sin(x) up to n = 7

$\sin x = x - x^3(1/3!) + x^5(1/5!) - x^7(1/7!)$

Test 1

Test 1 will only test the ADD and SUB instruction of the floating point arithmetic block.

When **X = 1**,

Sin(1) = 1 - 1/3! + 1/5! - 1/7!

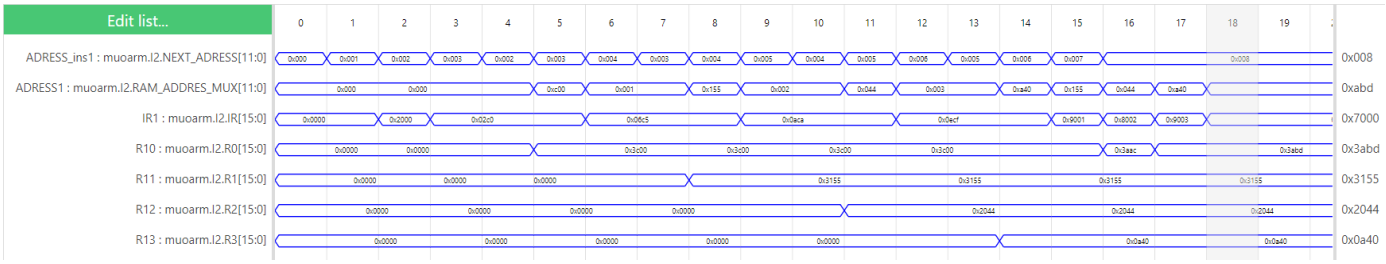
This table shows us the closest value in binary16 to represent the numbers needed to calculate the Maclaurin series of sin(1).

Decimal	Binary16
1	0X3C00
0.1666 =~ 1/3!	0X3155
8.3313x10^-3 =~ 1/5!	0X2044
1.9073x10^-4 =~ 1/7!	0X0A40

The code which can be found in Appendix C, was used to calculate sin(1) using Maclaurin Series. The table below, illustrates the results that we obtained from simulating the code:

Result (binary16)	Decimal equivalent	Cycle
R0 = 0x3c00	1	5
R1 = 0x3155	0.1666	8
R2 = 0x2044	8.3313x10^-3	11
R3 = 0x0a40	1.9073x10^-4	14
R0 = 0x3aac	0.83398	16
R0 = 0x3abd	0.84229	17
R0 = 0x3abd	0.84229 = 0.842 (3dp)	18

Waveform:



In conclusion:

- The answer is true for up to 3 decimal places. (Sin(1) = 0.84147...)
- The corresponding value of registers during the process were true for up to 2 or 3 decimal places.
- The floating point addition and subtraction work.

Test 2

Test 2 will test all three floating point instructions. (ADD, SUB and MULTI)

When X = π/2,

$\text{Sin}(\pi/2) = \pi/2 - (\pi/2)^3(1/3!) + (\pi/2)^5(1/5!) - (\pi/2)^7(1/7!)$

This table shows us the closest value in binary16 to represent the numbers needed to calculate the Maclaurin series of sin(π/2).

Decimal	Binary16
1.5703125 =~ π/2	0X3E48
0.1666 =~ 1/3!	0X3155
8.3313x10^-3 =~ 1/5!	0X2044
1.9073x10^-4 =~ 1/7!	0X0A40

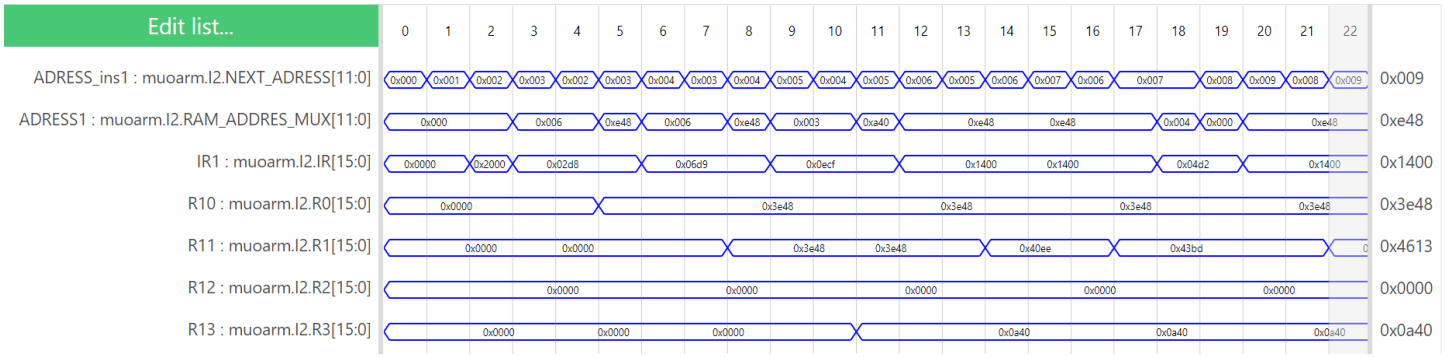
The code which can be found in Appendix C, was used to calculate sin(π/2) using Maclaurin Series. The table below, illustrates the results that we obtained from simulating the code:

Note:

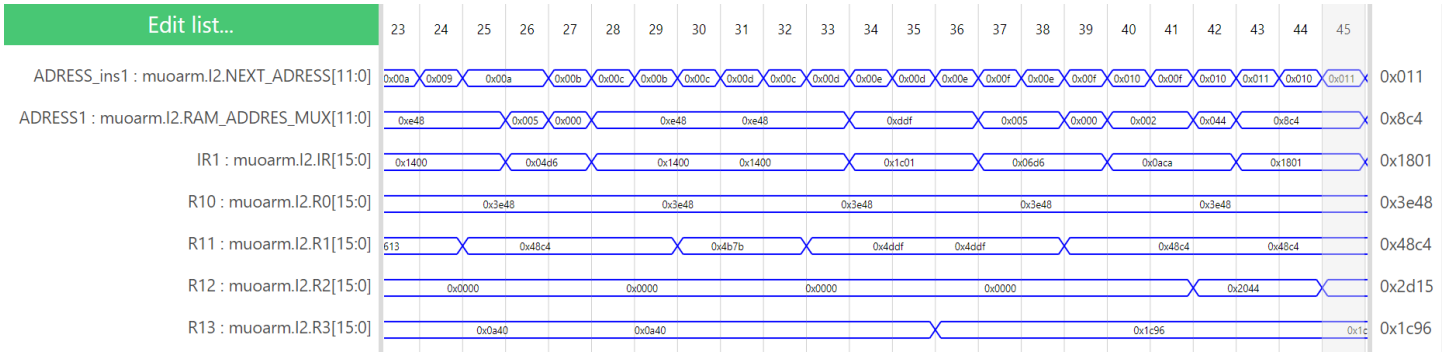
- This code can also calculate Maclaurin Series for other values of x. Choose your desired x value and put it in memory location 0x6 in DATA RAM.
- See Details column to see how the Maclaurin Series of sin X is built up in the code

Result (binary16)	Decimal equivalent	Cycle	Waveform
R0 = 0X3E48	1.5703125	5	1
R1 = 0X3E48	1.5703125	8	1
R3 = 0X0A40	1.9073x10^-4	11	1
R1 = 0X40EE	2.46484375	14	1
R1 = 0X43BD	3.869140625	17	1
R1 = 0X4613	6.07421875	22	1
R1 = 0X48C4	9.53125	25	2
R1 = 0X4B75	14.9140625	30	2
R1 = 0X4DDF	23.484375	33	2
R3 = 0X1C96	0.004478455	36	2
R1 = 0X48C4	9.53125	39	2
R2 = 0X2044	8.3313x10^-3	42	2
R2 = 0X2D15	0.07940673828	45	2
R0 = 0X0000	0	47	3
R1 = 0X43BD	3.869140625	49	3
R0 = 0X3155	0.1666	52	3
R1 = 0X3928	0.64453125	55	3
R0 = 0X0000	0	57	3
R0 = 0X3E48	1.5703125	59	3
R0 = 0X3B68	0.92578125	61	3
R0 = 0X3C05	1.004882813	62	3
R0 = 0X3C01	1.000976563 = 1.00 (3 decimal digits)	63	3

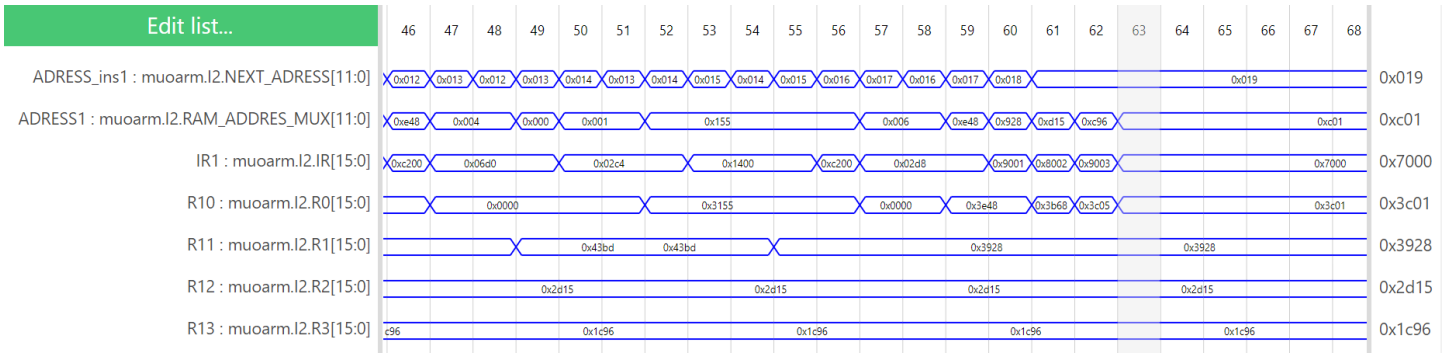
Waveform 1



Waveform 2



Waveform 3



In conclusion:

- The answer is true for up to 3 decimal digits.
- The corresponding values of the registers during the process are true for up to 2 or 3 decimal digits.
- The floating point addition, subtraction and multiplication works.
- The half precision floating point arithmetic block implementation is a success.

5. Dual Core Implementation

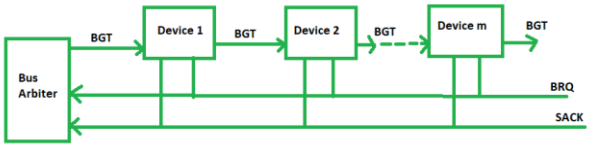
5.1. Research

The part of our design which needed the most attention, was the arbitration block. Therefore, our initial research was centred mostly around how to design the arbiter. This led us to finding the following three design options: (All of the following designs were inspired by the article "BUS Arbitration in Computer Organization" on the Geek-for-geeks website[9])

- **Daisy channel bus**

  The way this system works, is by transmitting a grant signal, going through each device until it finds one that requests access to the Bus. Then the arbiter will grant access, and any other devices down the line that request access also, are made to wait their turn. (Prioritising)

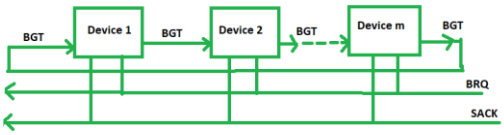
The Daisy channel design was the easiest one we could implement, since it is simple, and it allows us to extend it by increasing the number of connected devices.



Daisy chained bus arbitration

- **Polling priority method**

  For this method, a unique address is generated for each device connected to the system. When a device wants to gain access to the Bus, it will recognise its address and thus send out a signal that the Bus is busy. This will grant access to the device, and any other remaining devices will not be granted access as long as the Bus is busy.



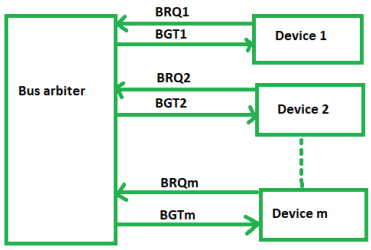
Rotating priority bus arbitration

The polling priority arbiter does not give priority to one core in particular. However, the delay could be quite significant since the arbiter communicates with one core at a time. In terms of reliability, the polling method does not have the same problem as the Daisy channel had, because if one device malfunctions, then the system will continue to work properly.

[9] SUDIPTADANDAPAT, Last Updated: 01 Apr 2021, "BUS Arbitration in Computer Organization", Geek-for-geeks, accessed 8 June 2021, <https://www.geeksforgeeks.org/bus-arbitration-in-computer-organization/>

- **Fixed priority and independent Request method**

  In this case, every device is connected to the arbiter, whilst each of them having a separate request/grant signal. It also allows for dynamic prioritisation, since you can assign which device has greater priority.



Fixed priority bus arbitration method

This last design is the swiftest one of the three since each core can communicate directly with the arbiter. However, if our project needed us to connect more devices together, then the costs of this method would probably outweigh its benefits.

Whilst having a clear idea on what our constraints for the project were, we understood that we only needed to implement an arbiter which controls two cores only. Therefore, we didn't have the concern of scalability when choosing our design method.

We immediately eliminated the **daisy channel bus** as a potential design since its main advantage was scalability. In addition, the design introduced some propagation delay, and this did not fit with the overall spirit of our CPU as we tend to prioritize reliability and speed over simplicity.

Since the beginning of our project, we aimed to make our processor as fast as we could, even if that meant generating more complex designs. (e.g., implementing pipeline) This is why we chose to go along with the **fixed priority and independent request method** instead of the **poling priority method** since it has the fastest respond time.

5.2. Design Explanation

The Dual Core is made of:

- **Arbiter:**

The following are the signals we needed to implement together with the Dual Core:

CPU Bus signals:
Address (Out, 12 bits)
Read Data (In, 16 bits)
Read Request (Out, 1 bit)
Read Valid (In, 1 bit)
Write Data (Out, 16 bits)
Write Request (Out, 1 bit)
Write Complete (In, 1 bit)

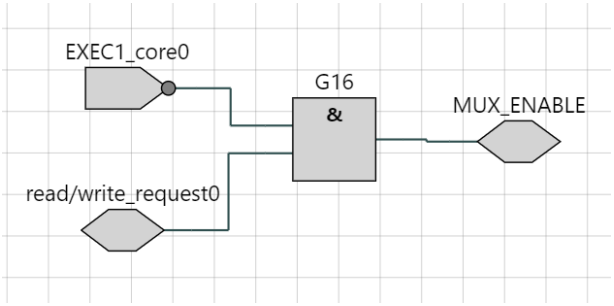
We decided that it would be more efficient to replace the write request and read request signals, with only one signal which we named **'ls'**. Similarly, the read valid and write complete

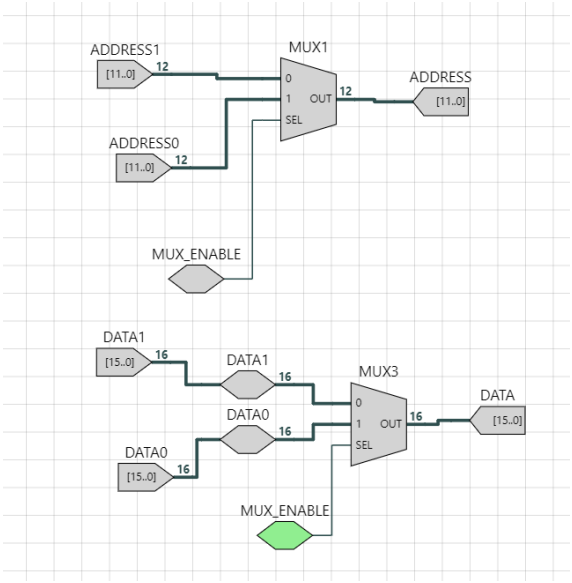
signals have also been replaced by the signal called **'en_stall'**. By default, the instruction will be complete or valid, but in the event of a conflict between the two cores, then the **en_stall** will go high. (When both cores want to read or write at the same time)

- **Priority:**

In our design, we decided that the MU0ARM0(1) will always get priority over the MU0ARM1(2). The main concern for us, was that there was a possibility where the MU0ARM1 would never be able to access the RAM. This is why we designed the arbiter in such a way, so that the MU0ARM1 would only have to wait for up to 1 clock cycle if a conflict between the cores would occur.

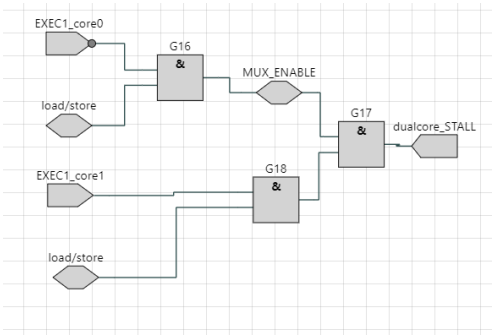
During a load and store instruction, we only access the RAM during EXEC1. This is because the CPU stalls during those instructions, therefore it is not possible for a core to access the RAM two clock cycles in a row, thus providing enough time for the other core to access the RAM. To implement this, we used a multiplexer which is shown below:



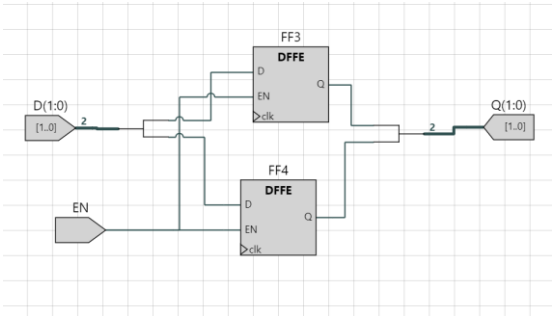


As we mentioned before, conflicts could occur at any time when both cores try to load or store at the same time. It was therefore important to force MU0ARM1 to stall. This implemented by having to slightly improve our state machine.

Here, we used the inputs EXEC1 for each of the cores, and also for whether there is a load/store instruction. The output dualcore_stall will be high only when both CPUs are at EXEC1, and when they both have a load/store instruction. This will indicate if there is any conflict between the cores.



The dualcore_stall is then directly connected to the enable of the state machine of MU0ARM1 (to be more precise, it's going through a not gate) and the load/store signal **'ls'** is low for only one cycle if a conflict occurred.



5.3. UART Involvement

For the Dual Core, we decided that both cores must be able to use a single UART, though only MU0ARM0 will be able to transmit data. After testing a design where both cores where able to receive and transmit data, we concluded that 2 read and 1 transmit is the best solution.

We came to that conclusion, because reading information from the UART is a relatively simple process in comparison to the RAM, in terms of time and priority. This is due to the fact that the UART has only two different memory locations to read from (byte received and status, with two reading port accessible to each core), thus we do not need to implement any logic for priority.

Transmitting, however, is a more complex process. Our initial thoughts were to implement two transmit, since it seemed reasonable enough to believe that if both cores have the ability to transmit, then the overall Dual Core CPU would be faster. But we then considered that the process of transmitting is quite long. (40 cycles per byte transmitted) After much reflection, we figured out that trying to increase speed on a process which occurs only every 40 cycles, was not the best the strategy.

The way we saw the transmitter being used, is by checking the status and then deciding whether to transmit or not. In the event where both cores need to transmit at the exact same time, then the arbiter would need to neglect one byte. There are ways to prevent this from happening, but we thought it was best to not add more complexity to our design and increase power consumption for such a meaningless and time-consuming instruction.

5.4. Testing

For testing, we compared the speed at which a Single and a Dual Core CPU can compute the Mean of Array of sized 2^n . (n = 4)

No	HEX (binary16)	DEC (decimal)
1	0x493D	10.4765625
2	0x4800	8
3	0x4540	5.25
4	0x44AE	4.6796875
5	0x42C2	3.37890625
6	0xC600	-6
7	0x3c00	1
8	0x3800	0.5
9	0x36C2	0.4223632813
10	0x3400	0.25
11	0x3155	0.1666259766
12	0xB1B5	-0.1783447266
13	0xB940	-0.65625
14	0xC024	-2.0703125
15	0xC1B7	-2.857421875
16	0xC200	-3
	MEAN =	19.36181641/16 = 1.210113525

Table of the array used and its mean.

For both Dual and Single Core CPUs, we implemented a code, which can be found in Appendix D, to test the number of cycles it takes to calculate the desired result that was found in the above table.

The following tables, showcase the results we obtained from the code:

Dual Core:

MU0ARM 0:

Result	Cycle	Waveform
R0 = 0x493D	5	1
R1 = 0x4800	8	1
R0 = 0x4C9E	10	1
R1 = 0x4540	12	1
R0 = 0x4DEE	14	1

R1 = 0x44AE	16	1
R0 = 0x4F19	18	1
R1 = 0x42C2	20	1
R0 = 0x4FF1	22	1
R1 = 0xC600	24	1
R0 = 0x4E71	26	1
R1 = 0x3C00	28	1
R0 = 0x4EB1	30	2
R1 = 0x3800	32	2
R0 = 0x4ED1	34	2

MU0ARM 1:

Result(binary16)	**Cycle**	**Waveform**
R0 = 0x36C2	6	1
R1 = 0x3400	9	1
R0 = 0x3961	11	1
R1 = 0x3155	13	1
R0 = 0x3AB6	15	1
R1 = 0xB1B5	17	1
R0 = 0x3949	19	1
R1 = 0xB940	21	1
R0 = 0x1C80	23	1
R1 = 0xC024	25	1
R0 = 0xc022	27	1
R1 = 0xC1B7	29	2
R0 = 0xC4EC	31	2
R1 = 0xC200	33	2
R0 = 0xC7EC	35	2
R3 = 0x2C00	37	2
R1 = 0x4ED1	41	2
R0 = 0x4CD6	43	2
R0 = 0x3CD6	45	2

Note: The waveforms mentioned above and below, can be found in Appendix D.

Single Core:

RESULT (BINARY16)	CYCLE	WAVEFORM
R0 = 0x493D	5	1
R1 = 0x4800	8	1
R0 = 0x4C9E	10	1
R1 = 0x4540	12	1
R0 = 0x4DEE	14	1
R1 = 0x44AE	16	1
R0 = 0x4F19	18	1
R1 = 0x42C2	20	1
R0 = 0x4FF1	22	1
R1 = 0xC600	24	1
R0 = 0x4E71	26	2
R1 = 0x3C00	28	2
R0 = 0x4EB1	30	2
R1 = 0x3800	32	2
R0 = 0x4ED1	34	2
R1 = 0x36C2	36	2
R0 = 0x4EEC	38	2
R1 = 0x3400	40	2
R0 = 0X4EFC	42	2
R1 = 0x3155	45	2
R0 = 0X4F06	47	2
R1 = 0xB1B5	50	2
R0 = 0X4EFB	52	3
R1 = 0xB940	54	3
R0 = 0X4ED1	56	3
R1 = 0xC024	58	3
R0 = 0X4E4D	60	3
R1 = 0xC1B7	62	3
R0 = 0X4D97	64	3
R1 = 0xC200	66	3
R0 = 0X4CD7	68	3
R1 = 0x2C00	70	3
R0 = 0x3CD7 (73 cycles)	73	3

Conclusions:

- DUAL CORE needs less cycles to complete the MEAN of an ARRAY code, compared to SINGLE CORE (45 cycles compared to 73 cycles).
- DUAL CORE has higher computing speed.

6. Final Evaluation

6.1. Benchmark

Test strategy and optimization:

Notes before testing:

Before testing, we need to take into account the size of our code. In fact, the code needs to be long enough as to allow us to clearly see the real effect of pipeline and the real time improvement over a standard, non-pipelined processor.

The pipeline can be slowed down by instructions such as load, store, or multiplication. In a short code, the "weight" of those instructions will give us a false sense on how we improved when using the pipelined CPU. This mainly happens at the beginning of the code, if the number of instructions is not sufficiently large. As we increase the code size, the time improvement should converge to 1 third in comparison to standard a CPU.

How we tested:

The benchmark test is key for the final stages of our CPU, because it allows us first to evaluate our design in its entirety. We can also demonstrate the effectiveness of all the functionality that we've implemented, in order to facilitate the use of MU0-ARM. (The code for the Benchmark can be found in Appendix F)

To make this test as meaningful as possible, we distinguish three different characteristics to test:

- multi-tasking
- calculus
- speed

Calculus:

We have coded several math operations that the CPU should successfully operate:

- **Maclaurin series of sin(x)**:

  For the calculation of the sin(x) Maclaurin series, we generated a random number, as result we will calculate sin (0.96) ≈ 0.819.

- **Maclaurin series of $\frac{1}{1-x}$:**

  In this case x = 0.65, which again, was picked randomly. This Maclaurin series is a really good test as it also involved multiplication floating pointing:

  $$\frac{1}{1-0.65} = 0.65 + 0.65^2 + 0.65^3 + 0.65^4 + \cdots = 2.8$$

  In our test we will perform the series up to the fourth power, thus the result will not be very accurate (=2.52)

- **Mean value of a 2^n series:**

  This calculation might not be the most difficult, however having chosen a 2^4 series, it does actually give us information on the behaviour of the CPU over a relatively long calculation. Although, the downside of this test is that it will not take full advantage of the pipeline, since it involves a good amount of load instructions.

No	HEX (binary16)	DEC (decimal)
1	0x493D	10.4765625
2	0x4800, move	8
3	0x4540	5.25
4	0x44AE	4.6796875
5	0x42C2	3.37890625
6	0xC600, move	-6
7	0x3c00, move	1
8	0x3800, move	0.5
9	0x36C2	0.4223632813
10	0x3400, move	0.25
11	0x3155	0.1666259766
12	0xB1B5	-0.1783447266
13	0xB940	-0.65625
14	0xC024	-2.0703125
15	0xC1B7	-2.857421875
16	0xC200, move	-3
	MEAN =	19.36181641/16 = 1.210113525

Determinant of a square matrix:

We consider this test to be the most complex one, as it involves both, multiplication and addition calculations with a 32-bit number. The following matrix is a random matrix chosen from our math notes:

$$\det\begin{pmatrix} 1 & 4 & 1 \\ 2 & 1 & 3 \\ 3 & 2 & 2 \end{pmatrix} = 17$$

- Multitasking:

Dual Core and UART interrupt, are both implementations aiming to enable our CPU to operate different task at the same time. Therefore, it is really important to see how the CPU can perform whilst trying to handle different tasks simultaneously. In addition, a continuous flow of inputs will be received by UART and then transmitted back, all done automatically by the UART interrupt

- speed:

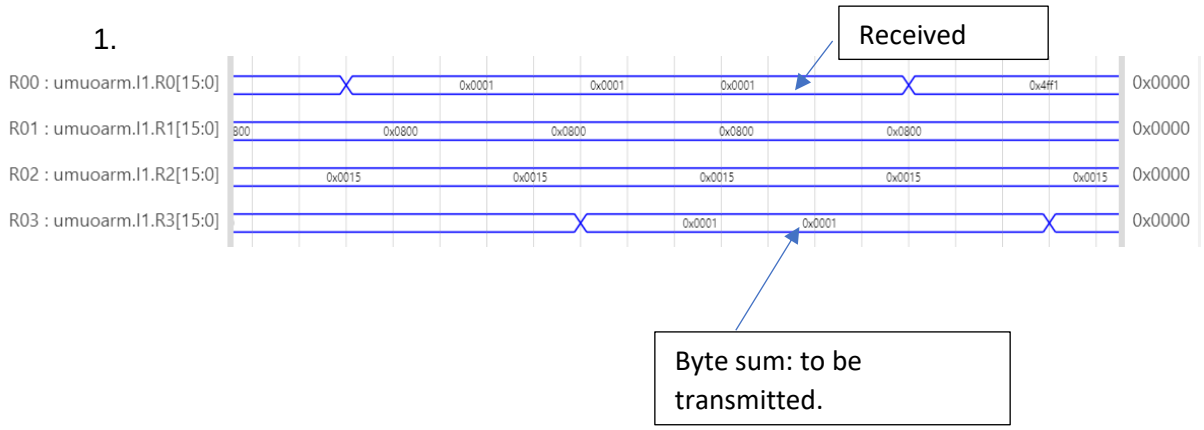
Pipelining and Dual Core are there to improve the speed of the CPU, in comparison to a normal CPU. Since speed is something relative, it is only interesting to test it if we can compare it with a non-pipeline dual core CPU.

6.2. Benchmark result

At the end of the benchmark, the dual core pipeline CPU had performed:

- 2 receiving and transmitting:
  - Expected result:
    - Receiving: 1.0x1 2.0x2
    - Transmitting: 1.0x1 2. 0x3
  - Result:
    - Receiving:

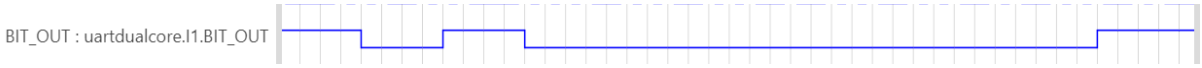
1.



2.

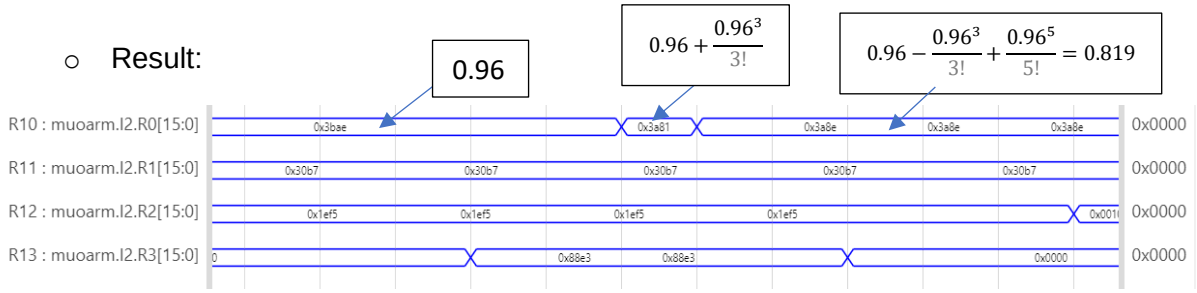
- Transmitting

1.

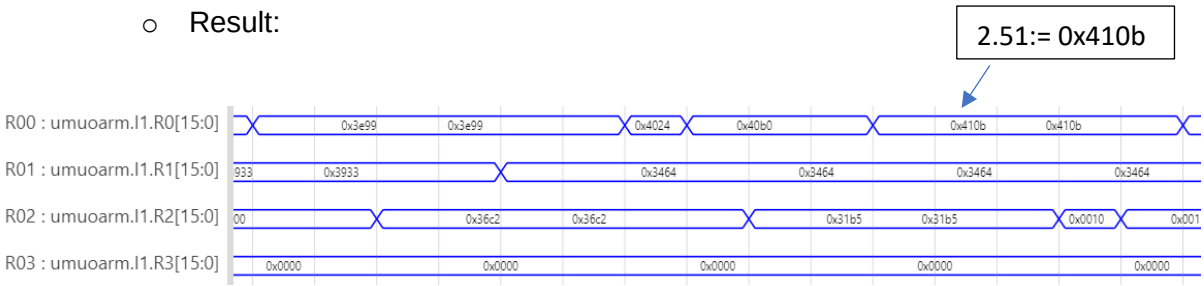


2.

- Sin(x) Maclaurin series up to the seventh power:
  - Expected result:
    - 0.819 := 3A8D
  - Result:

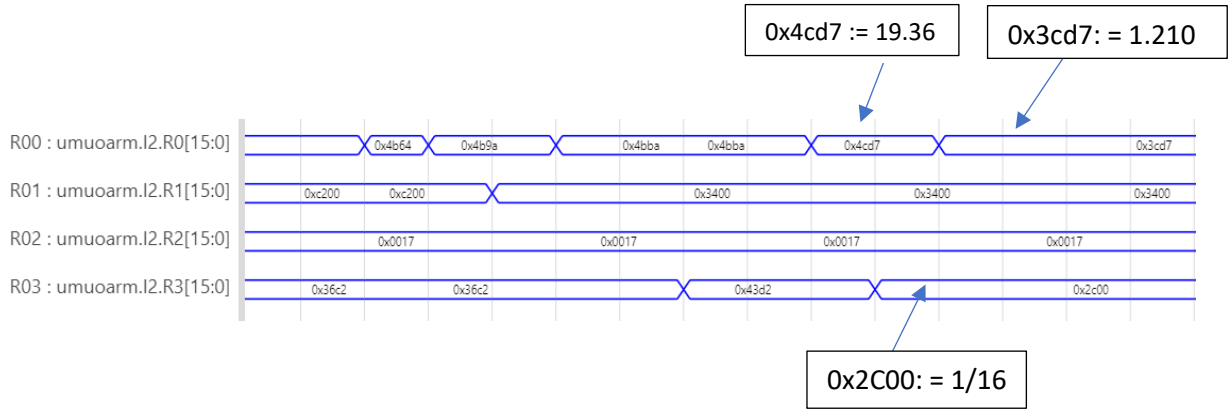


- $\frac{1}{1-x}$ Maclaurin series to the fourth power:
  - Expected result: 2.51: = 0x410b
  - Result:



- mean values of a 16-number series:
  - Expected result:

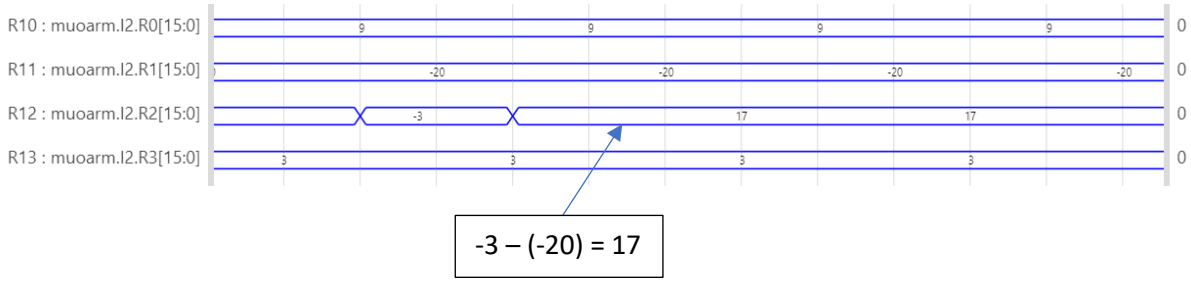
Mean = 19.36181641/16 = 1.210113525 = 0x3CD6



- Determinant of a 3 by 3 square matrix:
  - Expected result:

$$\det\begin{pmatrix} 1 & 4 & 1 \\ 2 & 1 & 3 \\ 3 & 2 & 2 \end{pmatrix} = 17$$

  - Result:



Notes after Benchmark process:

Number of instructions	123
Number of cycle both core combine	206
Number of cycle core 0	117
Number of cycle core 1	90
Number of cycles for non-pipelined dual core CPU	369
Number of cycles for pipelined single core CPU	229

As expected, the pipelining was quite slowed-down due to the massive use of load, multiplication and store instructions. As a result, the cycle optimization was more around ½ than 1/3, so an improvement of the immediate value capability could level up the speed massively.

Mu0arm has only four registers, consequently, store instructions are generally needed in a single core CPU to compensate for the lack of space in the registers. This is equivalent to the way the two cores talk to each other, and this is why the cycle gain between single core and dual core was around ½.

In the end, the benchmark was an excellent way to showcase the arithmetic potential of our dual core, by resolving challenging arithmetic problems. We still notice some difficulty to process 32-bit arithmetic calculations, such as integer multiplication, even though the different features implemented in the instruction are of great help.

Verilog test

Clock cycle:

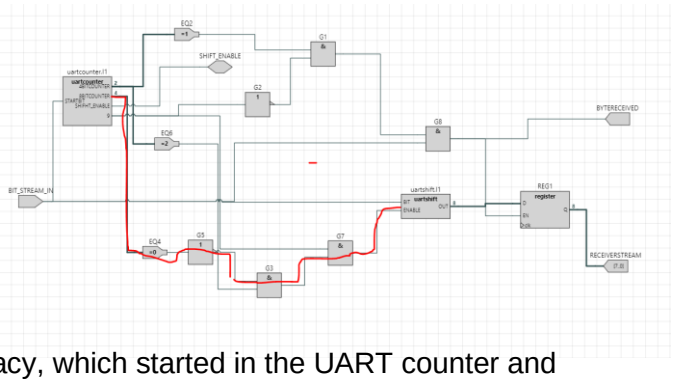
In our case, speed can be seen as the number of cycles which a program takes to be executed. After running the Verilog several times, we found an average maximum frequency of:

```
Info: Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 203.33 MHz (PASS at 12.00 MHz)
```

This is equivalent to: 4.9*10^-9 sec per cycle.

The way we proceed with the decryption, was by running the Verilog on a slightly different version of our design to see which path and module were coming out. Then we tried our best to understand where the delay came from and why. We were able to see some significant delay, coming from the UART receiver.

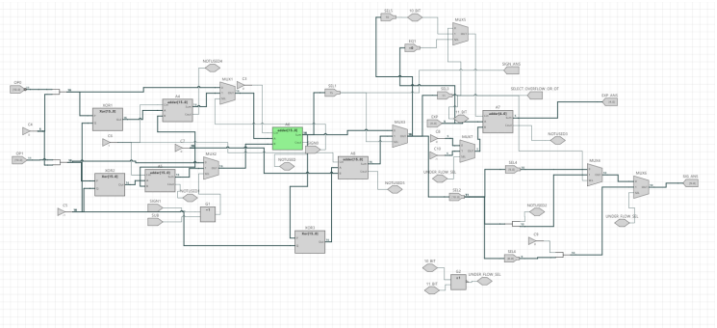
```
Info: Critical path report for clock 'clk$SB_IO_IN_$glb_clk' (posedge -> posedge):
Info: curr total
Info:  0.5  0.5  Source $abc$112019$auto$blifparse.cc:492:parse_blif$112033_LC.O
Info:  0.6  1.1    Net v$8BITCOUNTER_13202_out0[0] budget 27.183001 ns (2,1) -> (2,1)
Info:                Sink $abc$112019$auto$blifparse.cc:492:parse_blif$112037_LC.I0
Info:                Defined in:
Info:                  ./dualcore.v:1570
Info:  0.4  1.6  Source $abc$112019$auto$blifparse.cc:492:parse_blif$112037_LC.O
Info:  1.5  3.1    Net $abc$112019$v$EQ4_2125_out0_new_ budget 27.266001 ns (2,1) -> (4,2)
Info:                Sink $abc$112019$auto$blifparse.cc:492:parse_blif$112036_LC.I1
Info:                Defined in:
Info:                  ./dualcore.v:4198
Info:  0.4  3.5  Source $abc$112019$auto$blifparse.cc:492:parse_blif$112036_LC.O
Info:  1.7  5.1    Net v$ENABLE_1737_out0 budget 27.264999 ns (4,2) -> (2,3)
Info:                Sink $auto$simplemap.cc:420:simplemap_dff$111697_DFFLC.CEN
Info:                Defined in:
Info:                  ./dualcore.v:7424
Info:  0.1  5.2  Setup $auto$simplemap.cc:420:simplemap_dff$111697_DFFLC.CEN
Info: 1.5 ns logic, 3.8 ns routing
```



Luckily, we were able to trace the path with accuracy, which started in the UART counter and continued in the UART shift. Initially, we expected a more complex combinational logic to slow down our CPU.

One thing that we can take away from this, is to be careful with the use of bus compare which can add a lot of delay. We also need to be careful about adding too many gates in main sheet, as we do not pay attention to logic already implemented in sub sheet.

```
Info: Critical path report for clock 'clk$SB_IO_IN_$glb_clk' (posedge -> posedge):
Info: curr total
Info:  0.5  0.5  Source $auto$alumacc.cc:474:replace_alu$91449.slice[4].adder_LC.O
Info:  0.6  1.1    Net I497.a[4] budget 20.350000 ns (6,14) -> (7,14)
Info:                Sink $abc$92127$auto$blifparse.cc:492:parse_blif$92135_LC.I0
Info:                Defined in:
Info:                  ./main.v:344
Info:  0.4  1.6  Source $abc$92127$auto$blifparse.cc:492:parse_blif$92135_LC.O
Info:  1.0  2.5    Net $abc$92127$auto$simplemap.cc:168:logic_reduce$91699[1]_new_inv_ budget 20.350000 ns (7,14) -> (7,15)
Info:                Sink $abc$92127$auto$blifparse.cc:492:parse_blif$92132_LC.I3
Info:  0.3  2.8  Source $abc$92127$auto$blifparse.cc:492:parse_blif$92132_LC.O
Info:  0.6  3.4    Net v$G1_3000_out0 budget 20.349001 ns (7,15) -> (6,15)
Info:                Sink $abc$92127$auto$blifparse.cc:492:parse_blif$92141_LC.I1
Info:                Defined in:
Info:                  ./main.v:5061
Info:  0.4  3.8  Source $abc$92127$auto$blifparse.cc:492:parse_blif$92141_LC.O
Info:  1.9  5.8    Net I10490.we budget 20.349001 ns (6,15) -> (10,15)
Info:                Sink $auto$simplemap.cc:420:simplemap_dff$91781_DFFLC.CEN
Info:                Defined in:
Info:                  ./main.v:14003
Info:  0.1  5.9  Setup $auto$simplemap.cc:420:simplemap_dff$91781_DFFLC.CEN
Info: 1.8 ns logic, 4.1 ns routing
```



The tricky thing with optimization, is that sometimes it is just not possible. This is exactly what is happening with the floating arithmetic block, where we tried everything to reduce the logic, but could not get around it.

In that case, we believe the only solution is to divide the instruction into more steps. At the moment, our MU0-ARM CPU is using 3 cycles per instruction. We could add one more cycle, which could help us by dividing the combinational path into more steps, thus reducing the clock cycle.

Power consumption:

```
Info: Device utilisation:
Info:            ICESTORM_LC:    57/ 7680     0%
Info:           ICESTORM_RAM:     2/   32     6%
Info:                  SB_IO:     9/  256     3%
Info:                  SB_GB:     2/    8    25%
Info:           ICESTORM_PLL:     0/    2     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

From what we can see from the given statistic from the Verilog test, 57 logic cells are being used at the same time.

If we consider the average number of CMOS in a logic is four, (the power consumption of one CMOS is 3e-09W) then the power consumption of our design is $4*57*3e-09 = 6.84 * 10^{-7}$ W (Disclaimer: Since we were not experienced in finding the power consumption, we were not completely sure if the calculated value is correct)

7. Project Management

For any project to be successful, it needs to have a proper work schedule. It is a necessity, before any work is done, to decide amongst your team members how and when to labour on specific tasks. This technique is of great assistance, since it breaks down a mountain of workload into small individual tasks, making the project seem a little bit less intimidating. Moreover, a good schedule allows for great time management, as it forces you not to fall behind on the work that needs to be done and sets you up on track to deliver your project on time. These reasons, along with many others, are what lead our team to agree on a plan which suited all of us.

7.1. Meetings and Note Sharing

All our team's project meetings were held on Microsoft Teams. The screen-sharing feature was a simple solution for trouble-free communication whilst working remotely. In addition,

every team member was accustomed to working on this application, which allowed for a smooth experience when collaborating.

We had also decided on the use of OneNote for sharing notes and test result recordings. The application seemed like the easiest way of quickly sharing information between members. Lastly, as stated above, all our team members were already familiar with OneNote, which resulted saving us some valuable time.

7.2. Milestones

Before we begun planning our timeline, we needed to have some clear goals for the amount of time it would take us to complete a certain big task. After an effective discussion, considering the feasibility of achieving our milestones in time, we concluded that it should take us no more than one week to implement each feature, and a few more days to complete the advanced portion.

For the report and the video, it seemed reasonable to give ourselves 12 days before the deadline. This was done in order to ensure that we expressed all our thoughts and work in the best way possible.

7.3. Meeting Structure and Gantt Chart

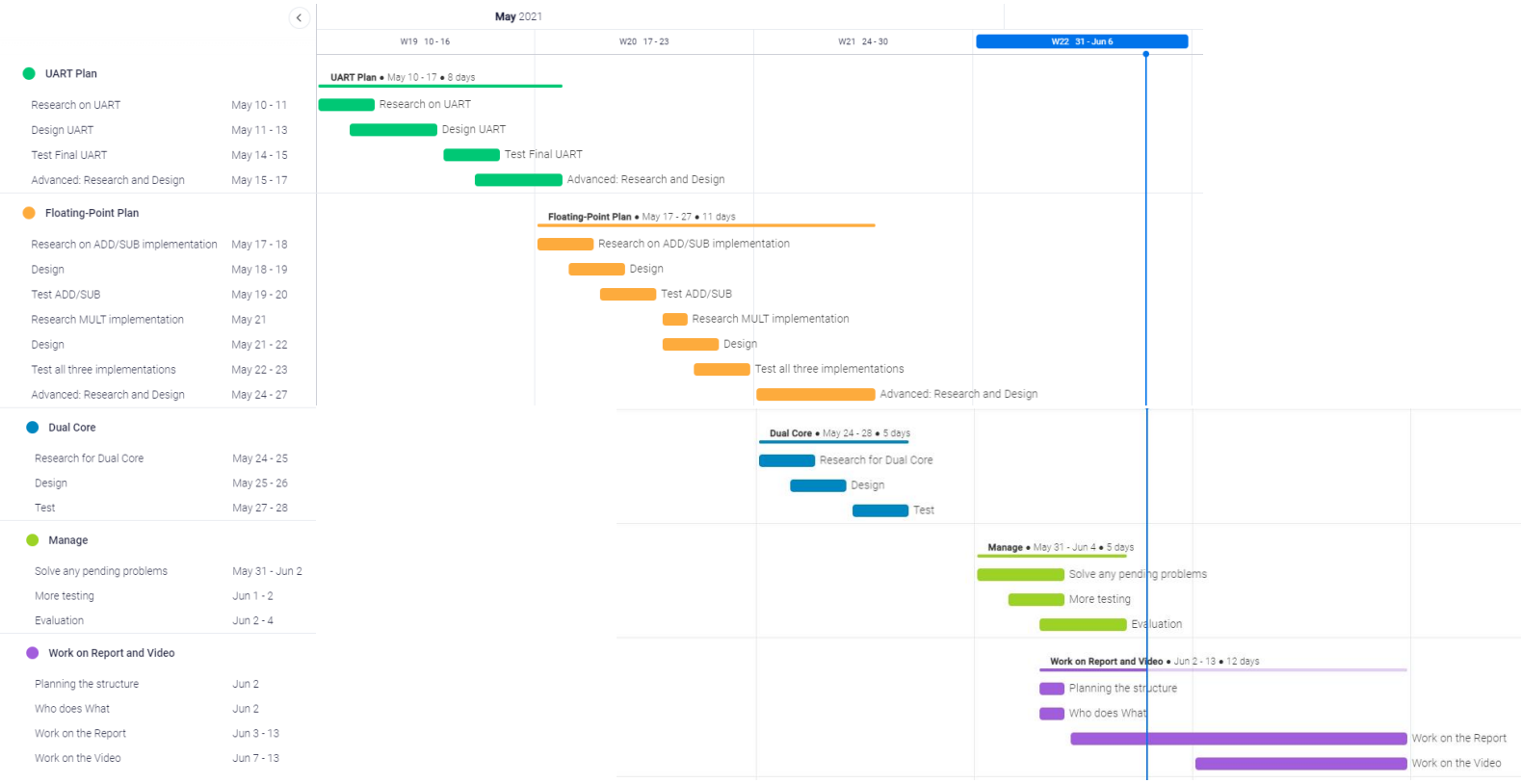
It was crucial for our team to settle on a strict schedule, which would not only give us the best chances at finishing on time, but also test our commitment and drive to provide a well-polished and thought-out design.

We used the Monday.com website to construct our schedule ahead of time. This website would provide us with the tools and freedom to constantly track and update our schedule in case something went wrong. Furthermore, we had the ability to convert our timeline into a Gantt chart, which would help us visualise every little detail we included more easily.

'The underlying concept of a Gantt chart is to map out which tasks can be done in parallel and which need to be done sequentially. If we combine this with the project resources we can explore the trade-off between the scope (doing more or less work), cost (using more or less resources) and the time scales for the project.'[10]

[10] Paul Naybour 18 Sep 2014, "Using a Gantt Chart to manage a project schedule", APM - the chartered body for the project profession, accessed 4 June 2021, <https://www.apm.org.uk/blog/using-a-gantt-chart-to-manage-a-project-schedule/>

Gantt Chart



In our schedule, we included the days which we would focus on Research, Designing and Testing. What is not shown in this diagram, is that we created additional sub-sections to remind us of when we had available time to improve our design. Additionally, we would update our timeline to include what each group member was working on, at a specific period of time.

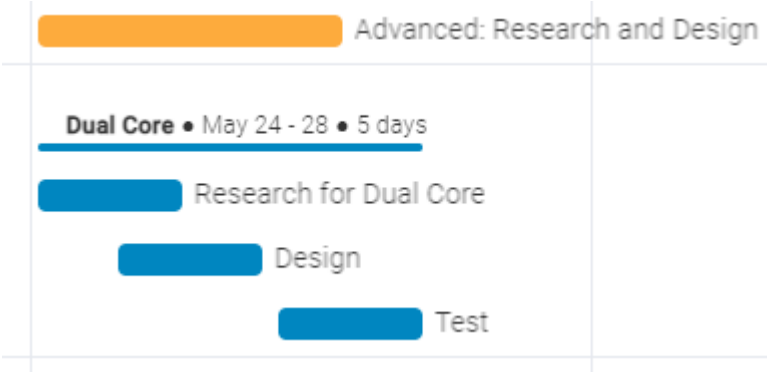
As we can see from the Gantt Chart, there was work needed to be done almost every day. This is why we planned to have team meetings for every day indicated on the chart. This was decided, in order to have a good idea on what each of the team members was working on, and to also be able to provide assistance to one another, if needed.

At the very beginning of our meetings, we would take some time off discussing what each of the group members should focus their time on. We did not want to assign any roles beforehand, since each member had various strengths which applied to different parts of the project. For example, one member could have been more familiar with the idea of the UART, so he would be responsible for its research. However, another member might have been more suitable for the research of the Floating-Point implementation, so assigning the role of research to one member felt like a mismanage of our strengths and potential.

Since we were working almost daily on our project, we decided not to work past 6 hours in a day. This was mainly to reduce the risk of becoming overburdened and exhausted, as this would have a negative impact on future work and our mental state.

As we draw our attention back to the Gantt Chart, it is noticeable that there was some overlapping of tasks. For those cases, we selected the member who was most comfortable with a task, to continue on his own as the other two group members tackled the other task.

This process helped us save time, and also showed us how much we trusted one another to complete their part of the work.



8. Conclusions

8.1. Thoughts on the Project

Despite any limitations that we might have faced as a team (e.g., remote collaboration etc.), we believe that the project we were tasked to complete was an overall success. Even though at times the project seemed challenging and exhausting, we were able to finish what we started, by meeting all the technical and non-technical requirements.

All three of the new features that were implemented, have been tested and proven that they work well together, as well as individually. The project, however, demanded much more than just the implementation. (e.g., Considering the speed of the CPU, creating efficient designs which function correctly etc.) Through those extra tasks that we were given to complete, we were taught a very important lesson on how real-life engineering projects are being carried out every day, and how to approach them.

8.2. How to Improve on Current and Future Work

Although our work on the project gave us great results, there were somethings which we could improve upon for future projects. More specifically, we found ourselves getting lost in the sea of files that we had created in ISSIE. This problem can be avoided by better organising our files and creating more folders for different versions of our designs. In addition, we could have been using a software which updated our ISSIE files automatically for every group member, in case a change had been made. This would eliminate the time wasted on trying to figure out which member had the latest version of a certain file.

Furthermore, instead of taking time off almost every group meeting, discussing what each member would work on that day, we could have planned that ahead of time. We also found that for any group project, it is important to be quite familiar with your group members. By already being accustomed to each other's strengths and weaknesses, you can create a much more efficient plan, suited for a better performing team.

It is true that for projects with strict deadlines, such as this one, we often limit ourselves and deny the project from reaching its full potential. If there was more time available, we would firstly improve on it by carrying out more tests to spot even the most minute details missed.

From the beginning of this project, we always had in mind to use the least number of gates and come up with the best ideas to make our designs work well together with the rest of the CPU. This is why it would have been important to spend more time on trying to enhance our designs to the best of our abilities.

References

[1] P. Fox, “Khan Academy,” "Central Processing Unit (CPU)", n.d.. [Online]. Available: https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:computers/xcae6f4a7ff015e7d:computer-components/a/central-processing-unit-cpu. [Accessed 2 June 2021].

[2] J. Andrew, “The Advantages of a Dual Processor on a Computer", Chron, n.d. [Online]. Available: https://smallbusiness.chron.com/advantages-dual-processor-computer-70316.html. [Accessed 2 June 2021].

[3] *"IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008, vol., no., pp.1-70, 29 Aug. 2008, doi:10.1109/IEEESTD.2008.4610935..*

[4] B. Kannan, “THE DESIGN OF AN IC HALF PRECISION FLOATING POINT ARITHMETIC LOGIC UNIT" (2009). All Theses. 689., p 4-6, [Online]. Available: https://tigerprints.clemson.edu/all_theses/689. [Accessed 8 June 2021].

[5] C. 3. C. Account, “Addition and Subtraction", UAF Computer Science, 13 September 1999. [Online]. Available: https://www.cs.uaf.edu/2000/fall/cs301/notes/notes/node51.html. [Accessed 8 June 2021].

[6] rajkumarupadhyay515, “Multiplying Floating Point Numbers", Geeks-for-Geeks, 4 May 2020. [Online]. Available: https://www.geeksforgeeks.org/multiplying-floating-point-numbers/. [Accessed 9 June 2021].

[7] SUDIPTADANDAPAT, “BUS Arbitration in Computer Organization", Geek-for-Geeks, 1 April 2021. [Online]. Available: https://www.geeksforgeeks.org/bus-arbitration-in-computer-organization/. [Accessed 8 June 2021].

[8] P. Naybour, “Using a Gantt Chart to manage a project schedule", APM - the chartered body for the project profession, 18 September 2014. [Online]. Available: https://www.apm.org.uk/blog/using-a-gantt-chart-to-manage-a-project-schedule/. [Accessed 4 June 2021].

Appendix A

Normal and Subnormal form:

There are two forms of numbers that can be represented in binary16 format, Normal and Subnormal form[11]. The following table illustrates how to differentiate when a number in binary16 format is in Normal or Subnormal form. The equation to translate a binary16 number into the decimal format for each form is also included in the last column.

Exponent	Significand = 0	Significand ≠ 0	Binary16 to decimal equation
0b00000	0, -0	Subnormal	(-1)^S x 2(-14) x 0.F
0b00001 – 0b11110	Normal	Normal	(-1)^S x 2(E – 15) x 1.F

Type of number that can be represented in binary16 and how to convert binary16 to decimal

Normal form range:

- The bias for this format = 15
- True E value = E – Bias
- Minimum value of E is 1 and the maximum value of E is 30
- Minimum true E value is -14 and the maximum true E value is 15.
- Largest magnitude for Normal number: 0b0 11110 1111111111 = 2^15 x (1 + 1023/1024) = 65504
- Smallest magnitude for Normal number: 0b0 00001 0000000000 = 2^-14 = 6.1035 x 10^-5

Subnormal form range:

- 0b0 00000 0000000000 represents zero
- True E value is always -14
- Largest magnitude for Subnormal number: 0b0 00000 1111111111 = 2^-14 x (1023/1024) = 6.0976 x 10^-5
- Smallest magnitude for Subnormal number: 0b0 00000 0000000001 = 2^-14 x 2^-10 = 5.9605 x 10^-8

32-bit IEEE 754 single precision:

The inputs and outputs of the floating-point arithmetic block are encoded according to the "binary32" interchange format.[12]

Sign (S)	Exponent (E)	Fraction (F)
1 bit	8 bits	23 bits

IEEE "binary32" floating point format

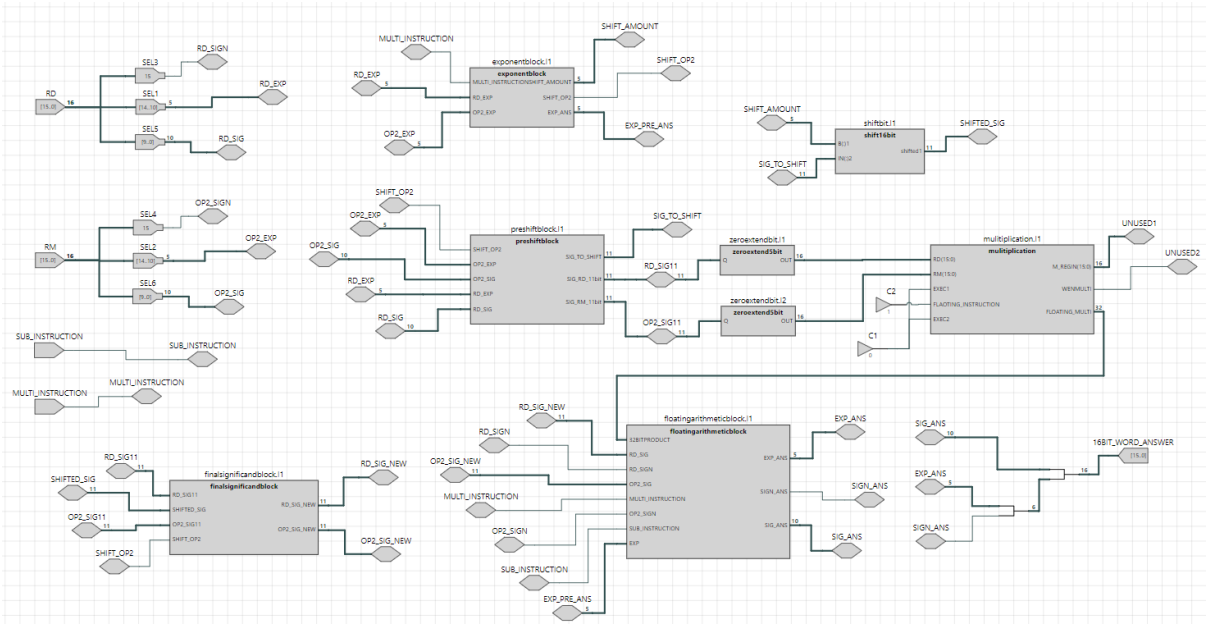
[11] "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008, vol., no., pp.1-70, 29 Aug. 2008, doi:10.1109/IEEESTD.2008.4610935.

[12] "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008, vol., no., pp.1-70, 29 Aug. 2008, doi:10.1109/IEEESTD.2008.4610935.

Appendix B

Block level testing

Multiplication block is added solely for block level testing purposes. In actual MU0ARM CPU, the multiplication block would be in the Datapath.



Summary table that includes all the test results, followed by Some screenshots for reference on how the data was taken:

Notes:

- Block Inputs: RD(Hex) and OP2(Hex)
- Block Output: Add(Hex) for addition, Sub(Hex) for subtraction and Multi(Hex) for multiplication
- Actual Add, Sub and Multi are the expected answers calculated using a calculator
- “Out of range” indicates the actual answer is outside the range of decimal numbers that can be represented using binary16 (see "Half-precision Limitation in Appendix A")

Test	Type	EXP diff	EXP total	Rd(Hex)	Rd (Dec)	Op2(Hex)	Op2 (Dec)	Add (Hex)	Add (Dec)	Actual Add	Sub (Hex)	Sub (Dec)	Actual Sub	Multi (Hex)	Multi (Dec)	Actual Multi
1	Both Normal, Rd > Op2	4	18	0x69d7	2990	0x5af8	223	0x6a46	3212	3213	0x6968	2768	2767	Out of range	Out of range	6.67 x 10^5
2	Both Normal, Rd < Op2	5	-7	0x2773	0.0291	0x3a7d	0.8110	0x3ab8	0.8398	0.8401	0xba42	-0.7822	-0.7819	0x260a	0.02359	0.02360
3	Both Normal, Rd > Op2	9	-3	0x4b6e	14.859	0x24e7	0.01915	0x4b70	14.875	14.8785	0x4b6c	14.84375	14.8402	0x348d	0.28442	0.28455
4	Both Normal, Rd < Op2	2	-14	0x9c59	-4.246x10^-3	0x2437	0.01646	0x2242	0.012222	0.012218	0xa54d	-0.020706	-0.020710	0x8494	-6.986x10^-5	-6.990x10^-5
5	Both Normal, Rd = Op2	0		0xC600	-6	0xC600	-6	0xca00	-12	-12	0x0000	0	0	0x5080	36	36
6	Both Normal, Rd < Op2	11	-13	0xba10	-0.7578	0x8c21	-2.5201x10^-4	0xba10	-0.7578	-0.75805	0xba10	-0.7578	-0.75755	0x0a42	1.9097x10^-4	1.9097x10^-4
7	Subnormal Op2, Rd > Op2	1	-28	0x0400	6.1035x10^-5	0x03ff	6.0976x10^-5	0x07ff	1.220x10^-4	1.220x10^-4	0x0001	5.9605x10^-8	5.9605x10^-8	Out of range	Out of range	3.7217x10^-9
8	Both Normal, Rd < Op2 (Rd and Op2 values are close)	1		0x33FE	0.2497558594	0x3400	0.25	0x37ff	0.49975585594	0.499755859 4	0x8c00	-2.4414x10^-4	-2.4414x10^-4	0x2bfe	0.0624389648	0.0624389648
9	Both Subnormal, Rd > Op2	0	-28	0x0231	3.34382x10^-5	0x01ce	2.75373x10^-5	0x03ff	6.09756x10^-5	6.09755x10^-5	0x0063	5.901x10^-6	5.901x10^-6	Out of range	Out of range	9.208x10^-10

	Test 1	Test 5	Test 8
ADD	Inputs RM (16 bits) 0x5af8 MULTI_INSTRUCTI... 0 RD (16 bits) 0x69d7 SUB_INSTRUCTION 0 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x6A46	Inputs RM (16 bits) 0xc600 MULTI_INSTRUCTI... 0 RD (16 bits) 0xc600 SUB_INSTRUCTION 0 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0xCA00	Inputs RM (16 bits) 0x3400 MULTI_INSTRUCTI... 0 RD (16 bits) 0x33fe SUB_INSTRUCTION 0 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x37FF
SUB	Inputs RM (16 bits) 0x5af8 MULTI_INSTRUCTI... 0 RD (16 bits) 0x69d7 SUB_INSTRUCTION 1 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x6968	Inputs RM (16 bits) 0xc600 MULTI_INSTRUCTI... 0 RD (16 bits) 0xc600 SUB_INSTRUCTION 1 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x0000	Inputs RM (16 bits) 0x3400 MULTI_INSTRUCTI... 0 RD (16 bits) 0x33fe SUB_INSTRUCTION 1 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x8C00
MULTI	Inputs RM (16 bits) 0x5af8 MULTI_INSTRUCTI... 1 RD (16 bits) 0x69d7 SUB_INSTRUCTION 0 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x0916	Inputs RM (16 bits) 0xc600 MULTI_INSTRUCTI... 1 RD (16 bits) 0xc600 SUB_INSTRUCTION 0 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x5080	Inputs RM (16 bits) 0x33fe MULTI_INSTRUCTI... 1 RD (16 bits) 0x3400 SUB_INSTRUCTION 0 Outputs & Viewers 16BIT_WORD_ANSW... (16 bits) 0x2BFE

Appendix C

Test 1 Code

RAM Table:

Address	Hex	Mnemonic	Operation	Expected Result(Hex [binary16 format])	Expected result (Dec [decimal format])	Details
0x0	0x2000	FLOAT	Floating mode (ON)	-	-	-
0x1	0x02C0	LOAD R0 = R0	R0 = mem[0x0]	R0 = 0x3c00	1	= 1
0x2	0x06C5	LOAD R1 = R1 + 1	R1 = mem[0x1]	R1 = 0x3155	0.1666	= 1/3!
0x3	0x0ACA	LOAD R2 = R2 + 2	R2 = mem[0x2]	R2 = 0x2044	8.3313x10^-3	= 1/5!
0x4	0x0ECF	LOAD R3 = R3 + 3	R3 = mem[0x3]	R3 = 0x0a40	1.9073x10^-4	= 1/7!
0x5	0x9001	SUB R0 R1	R0 = R0 - R1	-	1 - 0.1666 = 0.8334	= 1 - 1/3!
0x6	0x8002	ADD R0 R2	R0 = R0 + R2	-	0.8334 + 8.3313x10^-3 = 0.8417	= 1 - 1/3! + 1/5!
0x7	0x9003	SUB R0 R3	R0 = R0 - R3	-	0.8417 - 1.9073x10^-4 = 0.8415 = 0.842 (3dp)	= 1 - 1/3! + 1/5! - 1/7! = sin 1
0x8	0x7000	STP	Program STOPS	-	-	-

DATA RAM:

Address	Hex
0x0	0x3c00
0x1	0x3155
0x2	0x2044
0x3	0x0a40

Test 2 Code

RAM Table:

Address	Hex	Mnemonic	Operation	Expected result (Hex [binary16 format])	Expected result (Dec[decimal format])	Details
0x0	0x2000	FLOAT	Floating mode (ON)			
0x1	0x02D8	LOAD R0 = R0 + 6	R0 = mem[0x6]	R0 = 0x3E48	1.5703125	= X
0x2	0x06D9	LOAD R1 = R1 + 6	R1 = mem[0x6]	R1 = 0x3E48	1.5703125	= X
0x3	0x0ECF	LOAD R3 = R3 + 3	R3 = mem[0x3]	R3 = 0x0A40	1.9073x10^-4	= 1/7!
0x4	0x1400	MULTI R1 R0	R1 = R1 x R0	R1 =	1.5703125 x 1.5703125 = 2.4658813	= X^2
0x5	0x1400	MULTI R1 R0	R1 = R1 x R0	R1 =	2.4658813 x 1.5703125 = 3.8722042	= X^3
0x6	0x04d2	STORE R1 [R2 + 4]	Mem[0x4] = R1	-	-	-
0x7	0x1400	MULTI R1 R0	R1 = R1 x R0	R1 =	3.8722042 x 1.5703125 = 6.0805707	= X^4
0x8	0x1400	MULTI R1 R0	R1 = R1 x R0	R1 =	6.0805707 x 1.5703125 = 9.5483962	= X^5
0x9	0x04d6	STORE R1 [R2 + 5]	Mem[0x5] = R1	-	-	-
0xa	0x1400	MULTI R1 R0	R1 = R1 x R0	R1 =	9.5483962 x 1.5703125 = 14.9939659	= X^6
0xb	0x1400	MULTI R1 R0	R1 = R1 x R0	R1 =	14.9939659 x 1.5703125 = 23.5452121	= X^7
0xc	0x1c01	MULTI R3 R1	R3 = R3 x R1	R3 =	23.5452121 x 1.907x10^-4 = 0.0044901	= X^7(1/7!)
0xd	0x6d6	LOAD R1 = R2 + 6	R1 = mem[0x5]	R1 =	9.5483962	= X^5
0xe	0xaca	LOAD R2 = R2 + 2	R2 = mem[0x2]	R2 =	8.3313x10^-3	= 1/5!
0xf	0x1801	MULTI R2 R1	R2 = R2 x R1	R2 =	9.5483962 x 8.3313x10^-3 = 0.0795506	= X^5(1/5!)
0x10	0xc200	MOV R0 = 0	R0 = 0x0000	R0 = 0x0000	0	-
0x11	0x6d0	LOAD R1 = R0 + 4	R1 = mem[0x4]	R1 =	3.8722042	= X^3
0x12	0x2c4	LOAD R0 = R0 + 1	R0 = mem[0x1]	R0 = 0x3155	0.1666	= 1/3!
0x13	0x1400	MULTI R1 R0	R1 = R1 x R0	R1 =	3.8722042 x 0.1666 = 0.6451092	= X^3(1/3!)
0x14	0xc200	MOV R0 = 0	R0 = 0x0000	R0 = 0x0000	0	-
0x15	0x02D8	LOAD R0 = R0 + 6	R0 = mem[0x6]	R0 = 0x3e48	1.5703125	= X
0x16	0x9001	SUB R0 R1	R0 = R0 - R1	R0 =	1.5703125 - 0.6451092 = 0.9252033	= X - X^3(1/3!)

0x17	0x8002	ADD R0 R2	R0 = R0 + R2	R0 =	0.9252033 + 0.0795506 = 1.0047539	= X - X^3(1/3!) + X^5(1/5!)
0x18	0x9003	SUB R0 R3	R0 = R0 - R3	R0 =	1.0047539 - 0.0044901 = 1.0002638 = 1.00 (3 decimal digits)	= X - X^3(1/3!) + X^5(1/5!) - X^7(1/7!) = sin X
0x19	0x7000	STP	Program STOPS	-	-	-

DATA RAM:

Address	Hex
0x0	0x0
0x1	0x3155
0x2	0x2044
0x3	0x0a40
0x4	0x0
0x5	0x0
0x6	0x3e48

Appendix D

Dual Core:

Shared DATA RAM Table:

Address	HEX (binary16)	DEC (decimal)
0	0x493D	10.4765625
1	0x4800	8
2	0x4540	5.25
3	0x44AE	4.6796875
4	0x42C2	3.37890625
5	0xC600	-6
6	0x3c00	1
7	0x3800	0.5
8	0	0
9	0x36C2	0.4223632813
a	0x3400	0.25
b	0x3155	0.1666259766
c	0xB1B5	-0.1783447266
d	0xB940	-0.65625
e	0xC024	-2.0703125
f	0xC1B7	-2.857421875
10	0xC200	-3
11	0x2C00	1/16

MU0ARM 0 Instruction Memory Table:

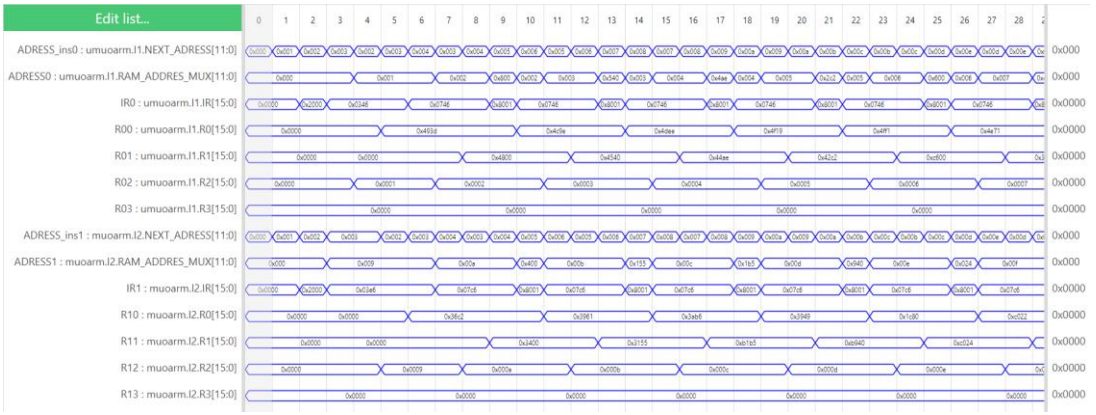
Address	Hex	Mnemonic	Expected Result (binary16)	Expected result (decimal)
0x0	0x2000	FLOAT	Floating Mode (ON)	
0x1	0x0346	LOAD R0 = R2 (+1) = mem[0x0]	R0 = 0x493D	10.4765625
0x2	0x0746	LOAD R1 = R2 (+1) = mem[0x1]	R1 =0x4800	8
0x3	0x8001	ADD R0 = R0 + R1	R0 =	10.4765625 + 8 = 18.4765625
0x4	0x0746	LOAD R1 = R2 (+1) = mem[0x2]	R1 = 0x4540	5.25
0x5	0x8001	ADD R0 = R0 + R1	R0 =	18.4765625 + 5.25 = 23.7265625
0x6	0x0746	LOAD R1 = R2 (+1) = mem[0x3]	R1 = 0x44AE	4.6796875
0x7	0x8001	ADD R0 = R0 + R1	R0 =	23.7265625 + 4.6796875 = 28.40625
0x8	0x0746	LOAD R1 = R2 (+1) = mem[0x4]	R1 = 0x42C2	3.37890625
0x9	0x8001	ADD R0 = R0 + R1	R0 = 0xC600	28.40625 + 3.37890625 = 31.78515625
0xA	0x0746	LOAD R1 = R2 (+1) = mem[0x5]	R1 =	-6

0xB	0x8001	ADD R0 = R0 + R1	R0 =	31.78515625 - 6 = 25.78515625
0xC	0x0746	LOAD R1 = R2 (+1) = mem[0x6]	R1 =	1
0xD	0x8001	ADD R0 = R0 + R1	R0 =	25.78515625 + 1 = 26.78515625
0xE	0x0746	LOAD R1 = R2 (+1) = mem[0x7]	R1 =	0.5
0xF	0x8001	ADD R0 = R0 + R1	R0 =	26.78515625 + 0.5 = 27.28515625
0x10	0x0142	STORE R0 = MEM[R2] = mem[0x8]	Mem[0x8] =	27.28515625
0x11	0x7000	STP		

MU0ARM 1 Instruction Memory Table:

Address	Hex	Mnemonic	Expected Result (binary16)	Expected result (decimal)
0x0	0x2000	FLOAT	Floating Mode (ON)	
0x1	0x03E6	LOAD R0 = R2 + 9 = mem[0x9]	R0 = 0x36C2	0.4223632813
0x2	0x07C6	LOAD R1 = R2 + 1 = mem[0xA]	R1 = 0x3400	0.25
0x3	0x8001	ADD R0 = R0 + R1	R0 =	0.4223632813 + 0.25 = 0.6723632813
0x4	0x07C6	LOAD R1 = R2 + 1 = mem[0xB]	R1 = 0x3155	0.1666259766
0x5	0x8001	ADD R0 = R0 + R1	R0 =	0.6723632813 + 0.1666259766 = 0.8389892579
0x6	0x07C6	LOAD R1 = R2 + 1 = mem[0xC]	R1 = 0xB1B5	-0.1783447266
0x7	0x8001	ADD R0 = R0 + R1	R0 =	0.8389892579 - 0.1783447266 = 0.6606445313
0x8	0x07C6	LOAD R1 = R2 + 1 = mem[0xD]	R1 = 0xB940	-0.65625
0x9	0x8001	ADD R0 = R0 + R1	R0 =	0.6606445313 - 0.65625 = 0.0043945313
0xA	0x07C6	LOAD R1 = R2 + 1 = mem[0xE]	R1 = 0xC024	-2.0703125
0xB	0x8001	ADD R0 = R0 + R1	R0 =	0.0043945313 - 2.0703125 = -2.0659179687
0xC	0x07C6	LOAD R1 = R2 + 1 = mem[0xF]	R1 = 0xC1B7	-2.857421875
0xD	0x8001	ADD R0 = R0 + R1	R0 =	-2.0659179687 - 2.857421875 = -4.9233398437
0xE	0x07C6	LOAD R1 = R2 + 1 = mem[0x10]	R1 = 0xC200	-3
0xF	0x8001	ADD R0 = R0 + R1	R0 =	-4.9233398437 - 3 = -7.9233398437
0x10	0x0FC6	LOAD R3 = R2 + 1 = mem[0x11]	R3 = 0x2C00	1/16
0x11	0x07A6	LOAD R1 = MEM[R2 - 9] = mem[0x8]	R1 =	27.28515625
0x12	0x8001	ADD R0 = R0 + R1	R0 =	-7.9233398437 + 27.28515625 = 19.3618164063
0x13	0x1003	MULTI R0 = R0 x R3	R0 =	19.3618164063 x 1/16 = 1.2101135254 = 1.2101135254 1.21 (3 decimal digits)
0x14	0x7000	STP		

Waveform 1



Waveform 2

Single Core:

DATA RAM Table:

Address	HEX (binary16)	DEC (decimal)
0	0x493D	10.4765625
1	0x4800	8
2	0x4540	5.25
3	0x44AE	4.6796875
4	0x42C2	3.37890625
5	0xC600	-6
6	0x3c00	1
7	0x3800	0.5
8	0x36C2	0.4223632813
9	0x3400	0.25
A	0x3155	0.1666259766
B	0xB1B5	-0.1783447266
C	0xB940	-0.65625
D	0xC024	-2.0703125

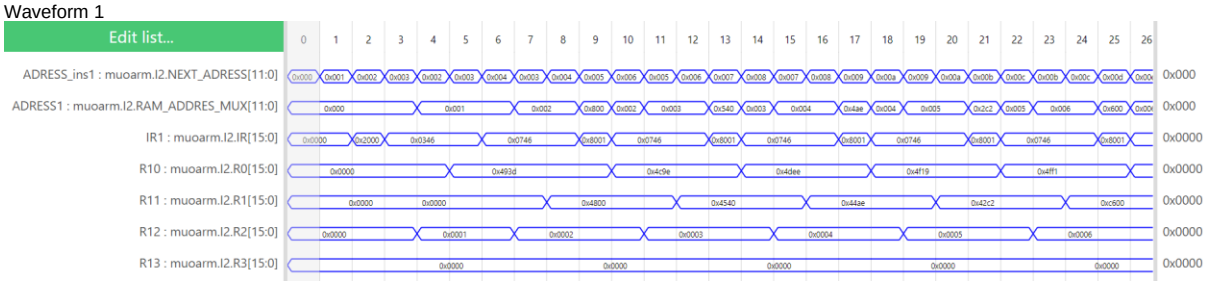
E	0xC1B7	-2.857421875
F	0xC200	-3
10	0x2c00	1/16

Instruction Memory Table:

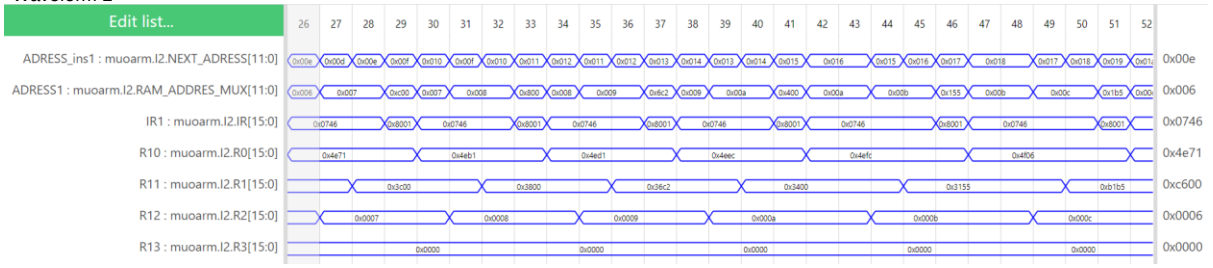
Address	Hex	Mnemonic	Expected Result(binary16)	Expected result (decimal)
0x0	0x2000	FLOAT	Float (ON)	
0x1	0x0346	LOAD R0 = R2 (+1) = mem[0x0]	R0 = 0x493D	10.4765625
0x2	0x0746	LOAD R1 = R2 (+1) = mem[0x1]	R1 = 0x4800	8
0x3	0x8001	ADD R0 = R0 + R1		10.4765625 + 8 = 18.4765625
0x4	0x0746	LOAD R1 = R2 (+1) = mem[0x2]	R1 = 0x4540	5.25
0x5	0x8001	ADD R0 = R0 + R1		18.4765625 + 5.25 = 23.7265625
0x6	0x0746	LOAD R1 = R2 (+1) = mem[0x3]	R1 = 0x44AE	4.6796875
0x7	0x8001	ADD R0 = R0 + R1		23.7265625 + 4.6796875 = 28.40625
0x8	0x0746	LOAD R1 = R2 (+1) = mem[0x4]	R1 = 0x42C2	3.37890625
0x9	0x8001	ADD R0 = R0 + R1		28.40625 + 3.37890625 = 31.78515625
0xA	0x0746	LOAD R1 = R2 (+1) = mem[0x5]	R1 = 0xC600	-6
0xB	0x8001	ADD R0 = R0 + R1		31.78515625 - 6 = 25.78515625
0xC	0x0746	LOAD R1 = R2 (+1) = mem[0x6]	R1 = 0x3C00	1
0xD	0x8001	ADD R0 = R0 + R1		25.78515625 + 1 = 26.78515625
0xE	0x0746	LOAD R1 = R2 (+1) = mem[0x7]	R1 = 0x3800	0.5
0xF	0x8001	ADD R0 = R0 + R1		26.78515625 + 0.5 = 27.28515625
0x10	0x0746	LOAD R1 = R2 (+1) = mem[0x8]	R1 = 0x36C2	0.4223632813
0x11	0x8001	ADD R0 = R0 + R1		27.28515625 + 0.4223632813 = 27.7075195313
0x12	0x0746	LOAD R1 = R2 (+1) = mem[0x9]	R1 = 0x3400	0.25
0x13	0x8001	ADD R0 = R0 + R1		27.7075195313 + 0.25 = 27.9575195313
0x14	0x0746	LOAD R1 = R2 (+1) = mem[0xA]	R1 = 0x3155	0.1666259766
0x15	0x8001	ADD R0 = R0 + R1		27.9575195313 + 0.1666259766 = 28.1241455079
0x16	0x0746	LOAD R1 = R2 (+1) = mem[0xB]	R1 = 0xB1B5	-0.1783447266

0x17	0x8001	ADD R0 = R0 + R1		28.1241455079 - 0.1783447266 = 27.9458007813
0x18	0x0746	LOAD R1 = R2 (+1) = mem[0xC]	R1 = 0xB940	-0.65625
0x19	0x8001	ADD R0 = R0 + R1		27.9458007813 - 0.65625 = 27.2895507813
0x1A	0x0746	LOAD R1 = R2 (+1) = mem[0xD]	R1 = 0xC024	-2.0703125
0x1B	0x8001	ADD R0 = R0 + R1		27.2895507813 - 2.0703125 = 25.2192382813
0x1C	0x0746	LOAD R1 = R2 (+1) = mem[0xE]	R1 = 0xC1B7	-2.857421875
0x1D	0x8001	ADD R0 = R0 + R1		25.2192382813 - 2.857421875 = 22.3618164063
0x1E	0x0746	LOAD R1 = R2 (+1) = mem[0xF]	R1 = 0xC200	-3
0x1F	0x8001	ADD R0 = R0 + R1		22.3618164063 - 3 = 19.3618164063
0x20	0x0746	LOAD R1 = R2 (+1) = mem[0x10]	R1 = 0x2C00	1/16
0x21	0x1001	MULTI R0 = R0 x R1	R0 = 0x3CD7 (73 cycles)	19.3618164063 x 1/16 = 1.2101135254
0x22	0x7000	STP		

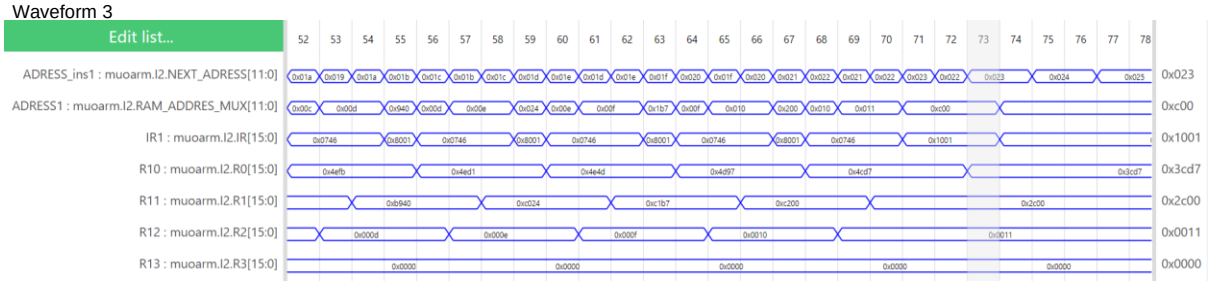
Waveform 1



Waveform 2



Waveform 3



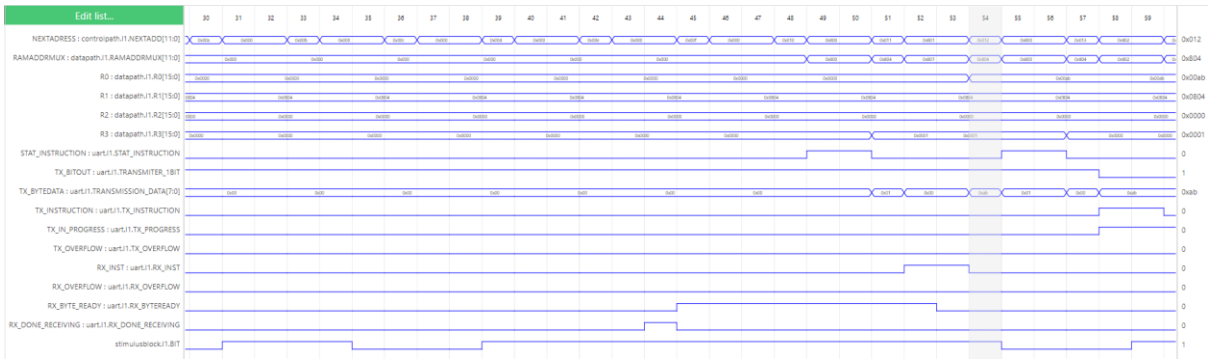
Appendix E

Test Code

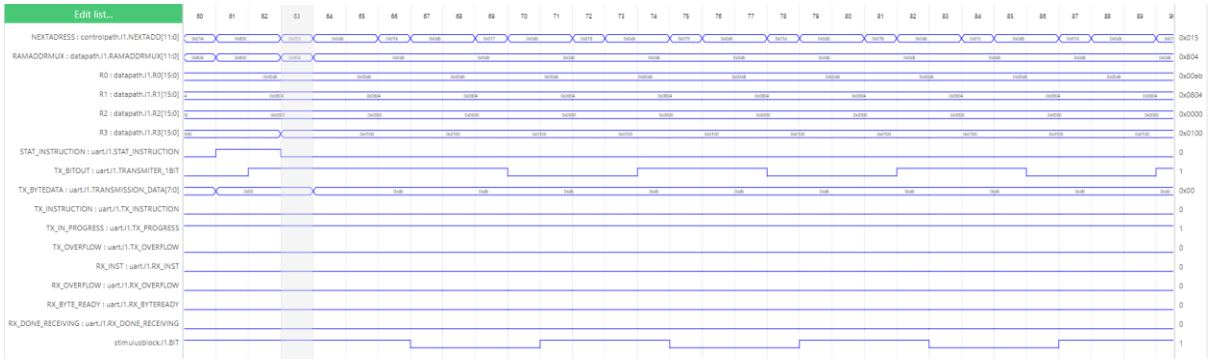
Address	Hex	Mnemonic	Expected Result	Detail
0x000	0XC604	MOV R1 #0x004	R1 = 0x0004	
0X001	0X847D	ADD R1[ROR7 R1]	R1 = 0x0804	
0X002	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0000)	All signals are LOW
0X003 - 0X00F	0			Waiting for a complete BYTE to be received
0X010	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0001)	RX_BYTE_READY = 1
0X011	0X028D	LDR R0[R1 - 3] / RX R0[7:0]	R0:=0X00(RX_BYTE)	BYTE is loaded into R0[7:0]
0X012	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0000)	All signals are LOW (the BYTE has already been used)
0X013	0X0089	STR R0[R1 -2] / TX R0[7:0]	TX_BYTE:=R0[7:0]	Content of R0[7:0] starts being transmitted (begin with 0 as start bit, and end with 1 as stop bit)
0X014	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0100)	TX_IN_PROGRESS = 1
0X015 - 0X020	0			
0X021	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0001)	RX_BYTE_READY = 1 (the transmission has completed at this point and a new BYTE has been received)
0X022 - 0X02F	0			
0X030	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0011)	RX_BYTE_READY = 1 and RX_OVERFLOW = 1 (another new BYTE has been received without the first one being loaded to any register. The first BYTE is lost)
0X031	0X02F5	LDR R0[R1 + 13] / RX R0[15:8]	R0:=0X(RX_BYTE)00	BYTE is loaded into R0[15:8]
0X032	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0000)	All signals are LOW
0X033	0X00F9	STR R0[R1 + 14] TX R0[15:8]	TX_BYTE:=R0[15:8]	Content of R0[15:8] starts being transmitted
0X034	0X0E91	LDR R3[R1 - 4] / STAT R3	R3:= UART_STATUS (0X0100)	TX_IN_PROGRESS = 1
0X035	0X00F9	STR R0[R1 -2] / TX R0[7:0]	TX_BYTE:=R0[7:0]	Content of R0[15:8] starts being transmitted
0X036	0X0089	LDR R3[R1 - 4]	R3:= UART_STATUS (0X1100)	TX_IN_PROGRESS = 1 and TX_OVERFLOW = 1 (a new transmission begins before the previous transmission ends)
0X037	0X7000	STP		

Waveform 1

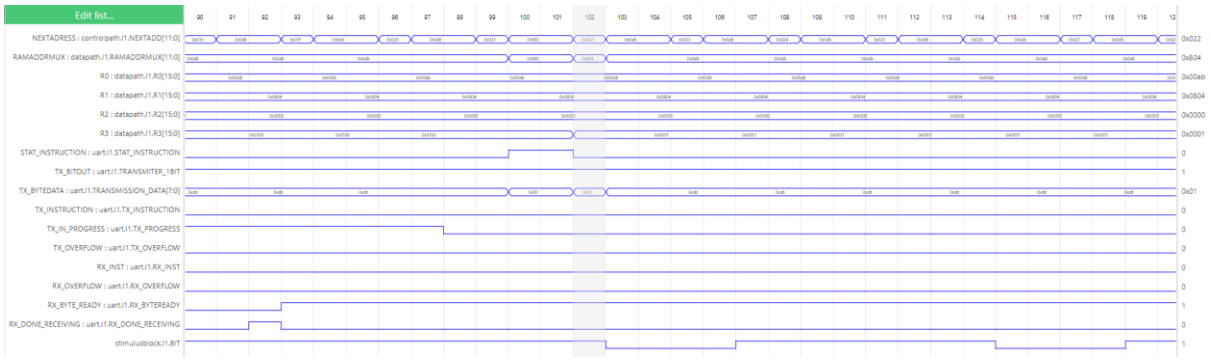
Waveform 2



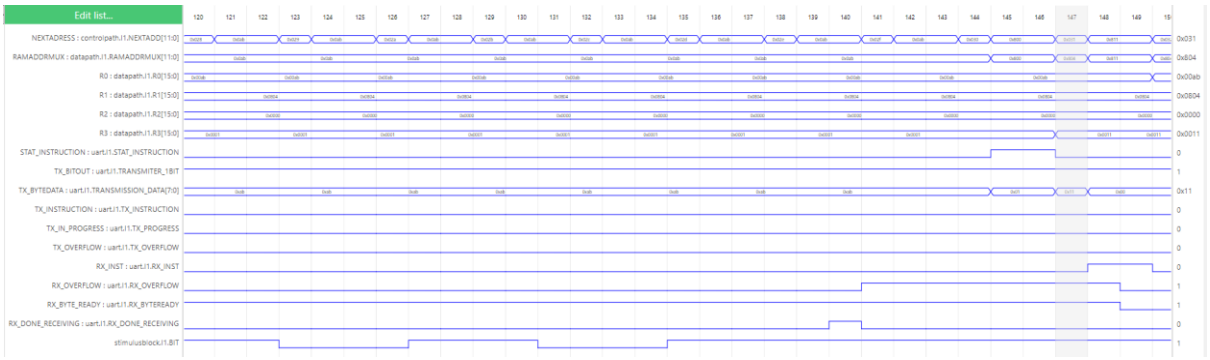
Waveform 3



Waveform 4

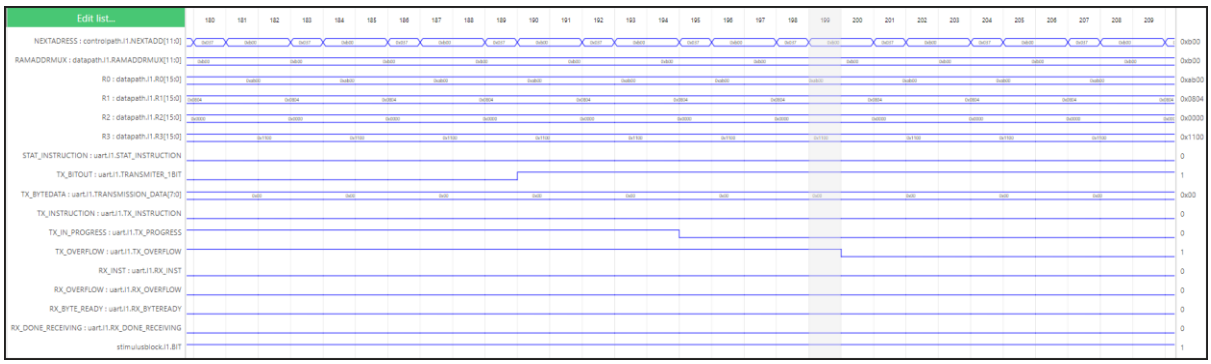


Waveform 5



Waveform 6

Waveform 7



Appendix F

Code for the Benchmark:

Address	Assembly	Description	Code
Muo_arm0			
$\frac{1}{1-x}$			
0x000	LDR R0, [R1, 0x7]	R0 := mem[0x7]= 1	0x02DD
0x001	Floating mode	Floating mode	0x2000
0x002	LDR R1, [R3, #0xA]	R1 := O.65	0x06EB
0x003	ADD R0, R1	R0 := 1 + 0.65	0x8001
0x004	LDR R2, [R3, #0xA]	R2 = 0.65^2	0x0AEB
0x005	Multi R1, R2, R1	R1 = 0.65^3	0x1406
0x006	ADD R0, R2	R0 := R0 + R2	0x8002
0x007	ADD R0, R1	R0 = R0 + R2 = 2.07	0x8001
0x008	Multi R2,R2, R2	R2 = 0.65^4	0x180a
0x009	ADD R0, R2	R0 := R0 + R2	0x8002
0x00A	Str R0, [R3, 0xA]	Mem[0xA] := R0	0x00EB
Mean sequence			
0xb	Move R2, [0x10]	R2 := 0x10	0xca10
0xc	LDR R0,[R2,#0x]!	R0 := mem[0x10]	0x0346
0xd	LDR R1,[R2],#0x1	R1 := mem[0x11] R2 := R2 + 0x1	0x0746
0xe	LDR R3,[R2],#0x1	R3 := mem[0x12] R2 := R2 + 0x1	0x0f46
0xf	Add R0, R1	ADD R0 := R0 + R1	0x8001
0x10	Add R0, R3	R0 := R0+ R3	0x8003
0x11	LDR R1,[R2],#0x1	R3 := mem[0x13] R2 + 1 = 3	0x0746
0x12	LDR R3,[R2],#0x1	R1 := mem[0x14]	0x0f46
0x13	ADD R0, R1	R0 := R0 + R1	0x8001
0x014	Add R0, R3	R0 := R0 + R3	0x8003
0x015	LDR R3,[R2],#0x1	R3 := mem[0x15]	0x0f46

		R2 := R2 + 0x1	
0x16	LDR R1,[R2],#0x1	R1 := mem[0x16] R2 := R2 + 0x1	0x0746
0x17	Add R0, R1	Add R0, R1	0x8001
0x18	Add R0, R3	Add R0, R3	0x8003
0x19	Move R1 := 0x3400	R1 := 0x3400	0xc66d
0x1a	Add R0, R1	R0 := R0 + R1	0x8001
0x1b	LDR R3 := mem[0x18]	LDR R3 := mem[0x18]	0x0ece
	STP		0x7000
Muoarm1			
$\frac{1}{1-x}$			
0x000	LDR R1, [R3, #0xA]	R1 := 0,65 =	0x06EB
0x001	Floating mode	Floating mode	0x2000
0x002	Multi R1, R1, R1	R1 := 0.65^2 =	0x1405
0x003	STR R1, [R3, #0xA]		0x04EB
Sin(x)			
0x004	LDR R0,[R3,0xB]	R0 := 0x3bae	0x02ef
0x005	LDR R1,[R3,0xC]	R1 := 0x30b7	0x06f3
0x006	LDR R2,[R3,0xD]	0x1ef7	0x0AF7
0x007	LDR R3,[R3,0xE]	0x88e3	0x0efb
0x008	SUB R0, R1	R0 := R0 – R1 = 0x3A81	0x9001
0x009	ADD R0, R2	R0 := R0 + R2 = 0x3A8e	0x8002
0x00a	ADD R0, R3	R0 := R0 + R3 = 0x3a8e	0x8003
0x00b	MOVE R3, [0x0]	R3 := 0x0000	0xcc00
0x00C	Store r0,[R3, 0xb]	Mem[0xb] := R0	0x00EF
Mean sequence			
0xd	Move R0, [0x4800]	R0 := 0x4800	0xc272
0xe	Move R1, [0x3800]	R1 := 0x3800	0xC66e
0xf	Add R0, R1	R0 := R1 + R0	0x8001
0x10	Mov R1, [0x3C00]	R1 := 0x3C00	0xC66F
0x11	Add R0, R1	R0 := R0 + R1	0x8001
0x12	Mov r2 [0x10]	R2 := 0x18	0xCA10

0x13	LDR R1,[R2],#1	R1 := mem[0x1a] R2 + 1 = 0x1a	0x07E6
0x014	LDR R3,[R2]#1	R1 := mem[0x1a] R2 + 1 = 0x1a	0x0fc6
0x015	Add R0, R1	R0 = R0 + R1	0x8001
0x16	Add R0, R3	R0 := R0 +R3	0x8003
0x17	LDR R3, [R2]#1	R1 := mem[0x2] R2 + 1 = 3	0x07c6
0x18	LDR R3, [R2]#1	R1 := mem[0x2] R2 + 1 = 3	0x0fc6
0x19	ADD R0, R1	R0 = R0 + R1	0x8001
0x1a	Add R0, R3	R0 := R0 +R3	0x8003
0x1b	LDR R3, [R2]#1	R1 := mem[0x2] R2 + 1 = 3	0x07c6
0x1c	ADD R0, R1	R0 = R0 + R1	0x8001
0x1d	Str R0, [R2, 0x3]		0x0096
Determinant			
0x01E	Integer mode	Integer mode	0x2000
0x01f	Mov R1, 0x3	R1 := 0x3	0xC603
0x020	Mov R2, 0x2	R2 := 0x2	0xCA02
0x021	MULTI, R2, R2, R3	R0 := R1*R2 = 4	0x180E
0x22	SUB R3, R1	R3 := 0x1	0x9C01
0x23	MULTI R1,[0x2], R0	R0 := 0x6	0x1602
0x24	Mov R2, 0x2	R2 := 0x2	0xCA02
0x25	SUB R2, R0	R2 := R2 – R0 = -4	0x9800
0x26	ADD R3, R2	R3 := R2 + R2 = -3	0x8C02
0x27	Str R3, [R1,0x4]	Mem[0x6] := R3 = -3	0x0cD1
0x28	Mov R1, 0x2	R1 := 0x2	0xC603
0x29	Mov R2, 0x3	R2 := 0x3	0xCA02
0x2a	Multi R1,R1,R0	R0 := R1 * R1 = 0x9	0x1401
0x2b	Multi R2,R2,R3	R3 := R2 * R2 = 0x4	0x180e
0x2c	SUB R3,R0	R3 := R3 – R0 = -5	0x9c00
0x2d	Multi R3,[0x4], R1	R1 := R3 * 4 = -20	0x1e04
0x2e	LDR R2, [R2,0x4]	R2 := mem[0x6] = -3	0x0AD2
0x2f	SUB R2, R1	R2 := -3 – (-20)= 17	0x9801
0x30	STP		0x7000