OPTION 2 :
Central Processing Unit

ELEC40006: 1ST YEAR ELECTRONICS DESIGN PROJECT 2021

Feng Shen Foo (CID : 01882052)

Tsz Hang Wong (CID : 01864937)

Xuan Cai (CID : 01860896)

Imperial College London
MEng Electrical and Electronic Engineering

TABLE OF CONTENTS

1. Introduction

The purpose of this project is to implement an efficient CPU from MU0-ARM CPU by adding extra features. This is an application of our knowledge from DECA labs Spring Term, and to further explore the capabilities and performance of CPU.

With the provided specification, the team was able to implement key features that improves the performance of the CPU. Evaluations were made to show how the key features contribute to form an efficient CPU and help the team come up with improvements on design.

2. Project Outline

Computers have many features to perform different functions effectively and communicate with other devices. In our MU0 ARM CPU, there are some limitations:

1. Data cannot be exchanged between the CPUs and peripheral devices bit by bit
2. It is incapable of doing arithmetic operations that involve values as small as 1μ and values as big as 1M.
3. The number of cycles needed to carry out certain programs is very large and it cannot perform different functions simultaneously.

The task is to integrate three extra features: serial communication, floating point arithmetic and dual core in the MU0-ARM CPU. Different evaluation methods can be used to check its correctness and performance.

3. Design Criteria (Product Design Specifications)

3.1 Specifications

Time-scale

The project must be done and documented by 13th June 2021. Project timeline should be made to meet the deadline.

Standards and Specifications

IEEE 754 Standards are used for floating-point implementation.

Environment

Environment used to build the hardware is Issie (Interactive Schematic Simulator and Integrated Editor). It is a block schematic based digital logic editor and simulator.

3.1.1 Operational Requirements

- The CPU should be able to allow exchange of data between peripheral devices and computer using the UART logic.
- The CPU should be able to perform calculations with the floating-point implementation.
- The CPU should be able to have dual cores to perform different instructions simultaneously.

3.1.2 Functional Requirements

Three main features will be added to the MU0-ARM CPU:

- Serial communication
  To handle information bit-by-bit, receive and transmit data from a peripheral.
- Floating point arithmetic
  To support more accurate processing of data.
- Dual core CPU
  To be able to execute 2 different instructions simultaneously.

The full specification of the features can be found in Appendix I.

3.2.2 Non-functional Requirements

Performance

1. **Speed**
   The speed of the CPU should be optimized to be able to carry out more operations in a shorter amount of time. It can be evaluated by counting the number of clock cycles and determining the maximum clock frequency of the CPU.
2. **Power consumption**
   The power consumption is desirable to be at minimum to allocate smaller batteries and produce smaller chips. It can be evaluated by determining the number of logic gates and clock speed.

Testing and Reliability

The CPU must be tested extensively to ensure the output is consistent. Corner cases should be considered.

4. Overview of the design

Figure 1 shows the overview of the team's CPU design.

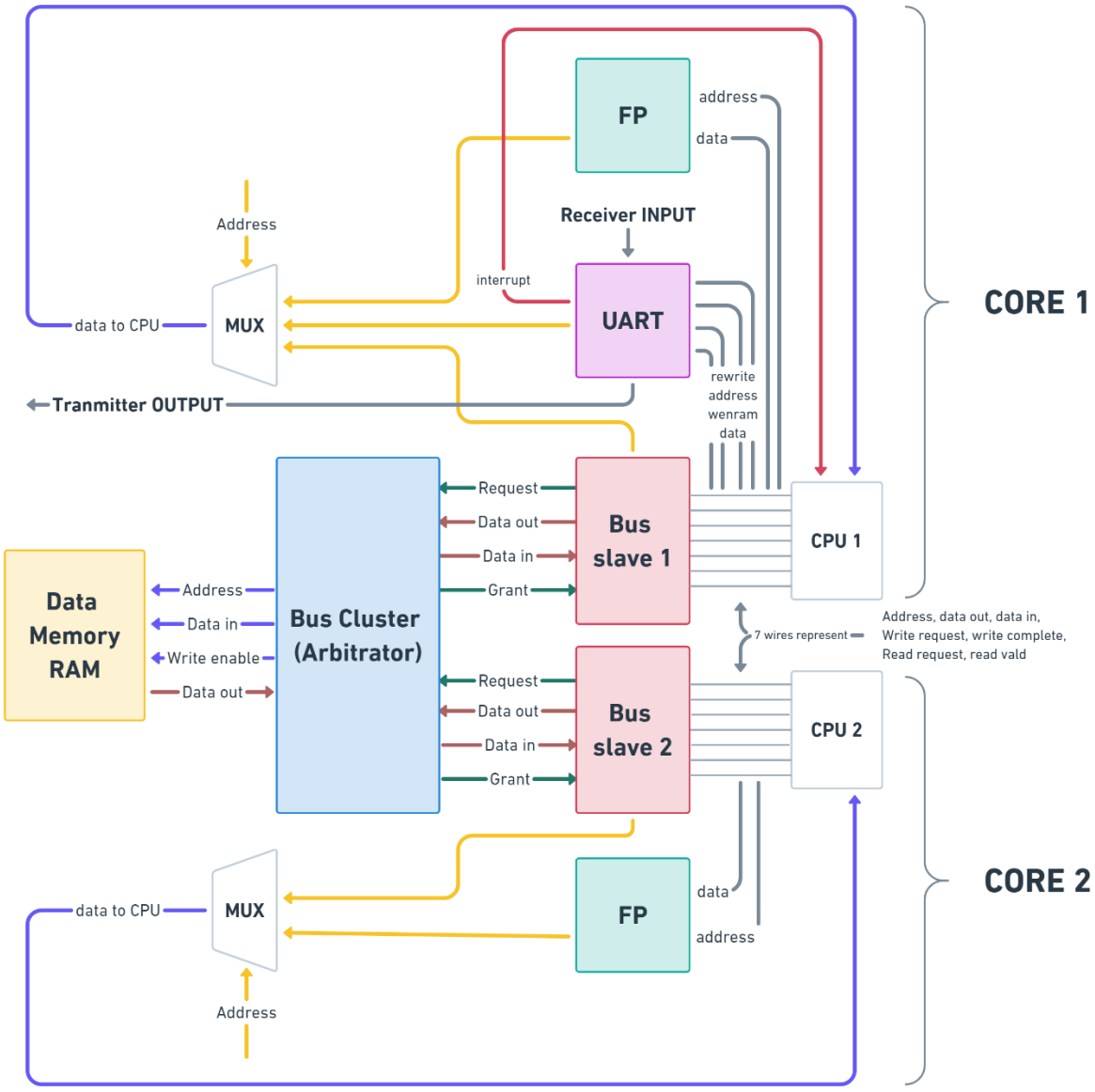


Figure 1 Overview of CPU design

It is a dual-core CPU with floating point arithmetic on both cores and UART serial communication on only one of the cores.

UART logic is often used for serial communication between input or output devices and the CPU. It consists of the UART receiver and transmitter.

Floating point arithmetic is used to perform calculations that involve values with large dynamic range. It can carry out add, subtract and multiply operations.

Bus arbitration logic was also included to tackle events when both cores try to access the same memory. To prevent conflict on data access, stall was implemented when the memory is busy.

The floating point was implemented on both cores because the team thought that floating point arithmetic calculations are very common and often large-scaled in computer algorithms and hence needs to be added on both cores so it can perform simultaneously and increase the speed.

Memory-mapped approach was used to integrate the devices into the CPU. The memory map of the CPU is shown in the table below.

Address	Target
0x000 … 0x7ff	RAM
0x800 0x801 0x802	UART Status UART Transmit UART Receiver
0x803 … 0x8ff	Unused
0x900 0x901 0x902 0x903 0x904	Floating point load first operand Floating point addition Floating point subtraction Floating point multiplication Floating point load result
0x905 … 0xfff	Unused

Figure 2 Memory map table for the CPU

5. Design Process

A systematic approach is required to tackle the problem efficiently. In general, our design process consists of 6 stages. For each part of the project, this process was repeated for a smooth flow of the project and achieve our goals.

1. **Identify the Problem**

   Problem identification is vital to help us have a deeper understanding on the project.

2. **Research and design**

   Desk research was done to develop practical ideas and come out with different solutions that allow comparisons. Drafts and sketches were also made as part of the initial design and brainstorming process.

3. **Discuss and analyze**

   With each member of the group having done the research, discussion was carried out to compare and analyze the different approaches.

4. **Implement**

   Implementation of the design was done based on the outcome of the discussion.

5. **Test and Debug**

   Test cases were formed to ensure our design works as expected without errors. When error occurs, debugging will take place.

6. **Evaluate and improve**

   After all the testing has passed, evaluations have been made to check whether improvements are possible. Drawbacks of the design were pointed out so the team can tackle them and optimize the design.

6. Components of the CPU

Include background materials (research), different approaches, discussion, main idea (flowchart, timing diagrams), initial sketch

Attach testing link (will be in the section after this)

6.2 Serial Communication

Identifying the problem

The CPU should be able to allow exchange of data between peripheral devices and computer using the UART logic. The UART should include logics that:

- Receive the data bit-by-bit, process them and transmit in bytes, then output bit by bit.
- Keep the bit rate at 1 bit per 4 clock cycles.
- Determine overflows in UART Receiver and Transmitter.
- Prompt when byte receive or transmit is complete

Research

Serial communication is the process of sending data one bit at a time, sequentially, over a communication channel or computer bus. In contrast to parallel communication, where several bits are sent as a whole, on a link with several parallel channels.

Serial communication is often used as a link of communication between CPUs and peripheral devices because it only requires one wire to transmit the data. Although the transmission of serial communication is slower than parallel communication, serial communication ensures that the space occupied by devices and the computer is small. [1]

A **universal asynchronous receiver-transmitter (UART)** is a computer hardware device for asynchronous serial communication. It usually consists of a receiver and transmitter, which sends data bits one by one, from LSB to MSB, framed by start and stop bits so that precise timing is handled by the communication channel. [2]

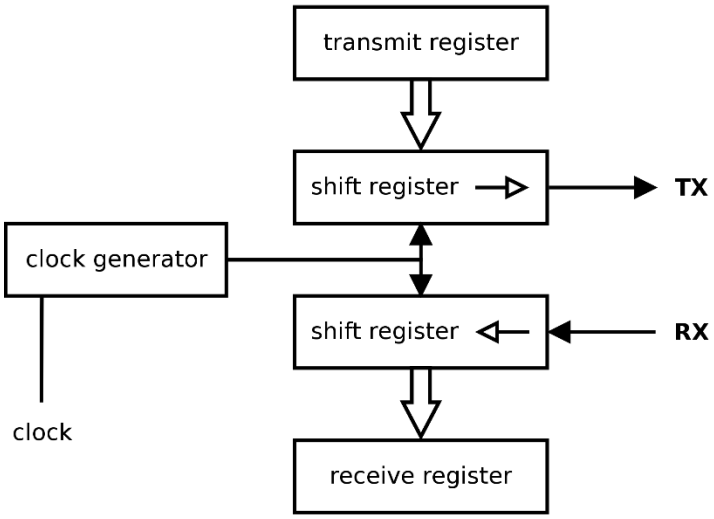


Figure 3 Block diagram for a UART (source from Wikipedia)

1. UART State Machine

The UART State Machine will show which state the Receiver and Transmitter are in respectively. It has 4 states, Idle state, Start state, Data state and End state. When the condition between the states is met, the State Machine will jump to the next state. But the conditions for the Boolean expression of this State Machine can be derived from the corresponding truth table.

Input						Output	
Q1	Q0	A	B	C	D	Q1+	Q0+
0	0	0	X	X	X	0	0
0	0	1	X	X	X	0	1
0	1	X	0	X	X	0	1
0	1	X	1	X	X	1	0
1	0	X	X	0	X	1	0
1	0	X	X	1	X	1	1
1	1	X	X	X	0	1	1
1	1	X	X	X	1	0	0

Figure 4 Truth table for state machine

Boolean expression: $Q0 = \overline{Q1}\,\overline{Q0}\,A + \overline{Q1}\,Q0\,B + Q1\,\overline{Q0}\,C + Q1\,Q0\,D$

$Q0 = \overline{Q1}\,Q0\,B + Q1\,\overline{Q0} + Q1\,Q0\,\overline{D}$

2. UART Receiver

A UART Receiver can receive an input which fits the UART protocol shown below and output the 8 data

bits in terms of a byte to the CPU.

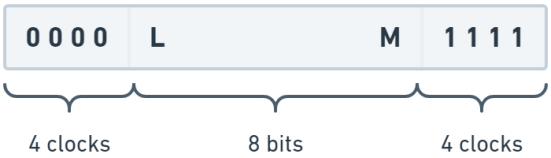


Figure 5 The UART Protocol

Sample rate block

Sample rate block is used in UART Receiver to sample the middle of the input data bits as shown in the figure below. As the edges are ambiguous between the two bits (see diagram below), it is safer to sample the middle of every data bit. Two different blocks were created to produce 2 types of sample rate and meet the condition where the bit rate is 1 bit per 4 clock cycles. They are **Clock 2** and **Clock 4** respectively.

Clock 2	0101
Clock 4	0001

Implemented by a D flip-flop, Clock 2 will remain at 0 and only start when the start bit is detected. During the Start and Data states, Clock 4 is produced by using AND gates on *Clock 2* and signal 0011 (0001 AND 0011 = 0001).

Figure 7 Clock to sample the middle of data bits

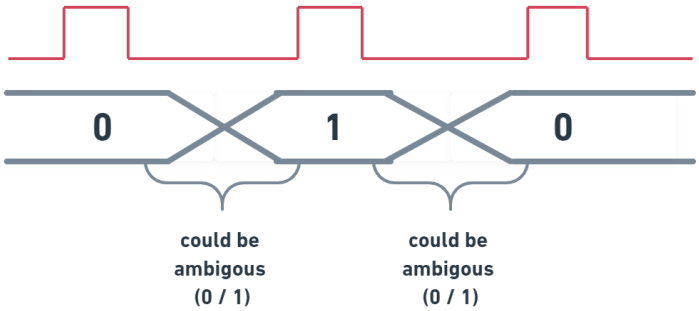


Figure 6 Ambiguous edges between real life digital signal

Conditions for receiver

Diagram below show how the state of the UART will change depending on the conditions.

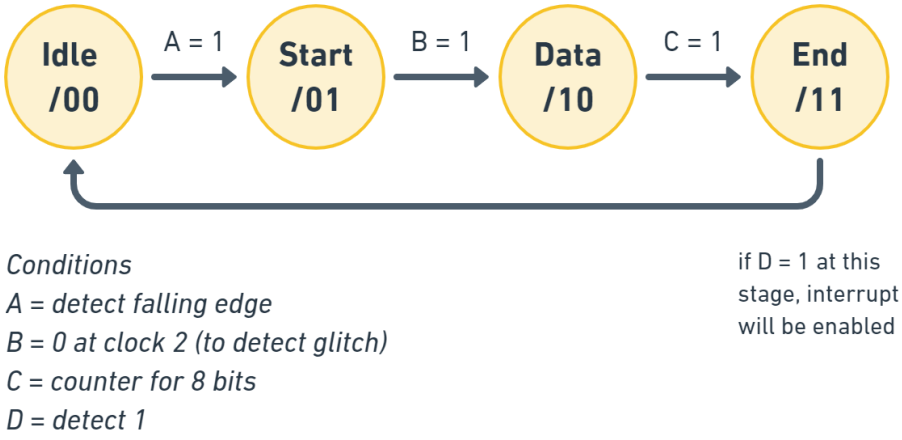


Figure 8 States of the UART Receiver

The section below will explain the conditions with the corresponding blocks that check the condition.

Condition A: The transmissions the receive signal is logic 1, hence the start bit is logic 0. When there is an input, a falling edge will form. If a falling edge is detected, Condition A is met and Receiver is prompted to move from Idle state to Start state.

Detect Falling Edge block

D Flip Flops is used to detect logic 0 when it is at logic 1.

Condition B: Based on Condition A, further insurance is needed because 1 bit will last 4 clock cycles and the second cycle of the 4 clock cycles should still be 0 to avoid glitch.

Zero at Clock 2 block

This block is designed to check if Condition B is met to avoid glitches at the starting bit by checking the start bit's 2nd clock. It is achieved by using the Clock 2 sample rate since it is asserted in the 2nd clock. The Zero at Clock 2 block is asserted to 1 when the second bit in starting state remain at 0, allowing the state to move on to Data state. Otherwise, the state would jump back to Idle, since the detected 0 bit is a glitch.

e.g. start bit 0000: Condition B logic 1.
start bit 0100 (glitch, not a proper starting bit): Condition B logic 0.

Condition C: Once the Receiver enters the Data state, a counter will start to count from 0 to 7 in the rate of counting 1 every 4 clock cycles to take sample from the 8 data bits. The sample is taken from the 2nd

cycle of each bit (at the middle) and then saved into the UART register (see UART register). After the 8th bit is saved, Condition C will be asserted and move on to End state.

Count-to-7 block

This block is used to track the number of bits received by counting from 0 to 7 (8 bits in total). It is designed to be counted at different rate depending on its input. For example, when the input sample rate is Clock 4, then the counter will plus 1 every 4 cycles.

e.g. During Data state, 8 data bits are counted by the 0-to-7 counter at the sample rate of Clock 4, Condition C will be logic 1.In Receiver, sample rate is always Clock 2 during Idle state and Clock4 during other states.

Condition D: During the end state, if the stop bit equals to logic 1, Condition D will be asserted, and an interrupt will be prompted. The usage of interrupt is to inform CPU that data has been received and can be processed.

Detect-ending-bit block

As for Condition D, this block is designed to detect the stop bit (logic 1) by sampling the ending bit at its middle. Its algorithm is similar to the Zero at Clock 2 block. Besides, it is used to trigger the interrupt when the last bit is a valid ending bit (logic 1)

e.g. Condition D and interrupt will be 1, when the second of 4 cycles of the ending bit is

UART register

UART register is used to store the 8 data bits bit-by-bit, using 8 registers with enable. During Data state, each register will be enabled by the Count-to-7 block, which should be counting at the rate of clock 4 (0001). For example, when counter equals to 0, then the register which is designed to store the 1st bit will be enabled and output it. Then, all 8 data bits are merged to form the 8 data bits output.

Overflow

The overflow for UART Receiver depends on two conditions. If these conditions are met, there will be no overflow. (Figure 9)

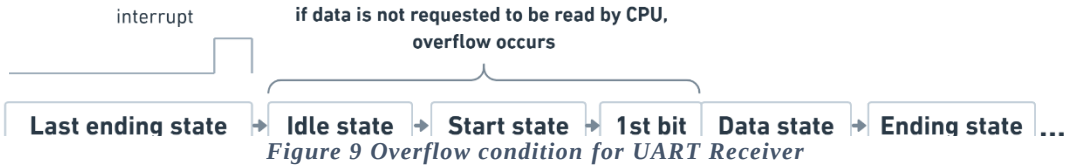


Figure 9 Overflow condition for UART Receiver

- The interrupt of last Ending state is logic 1.
- During the Idle state, Start state and the 1st bit (first four clock cycles) of Data state, the CPU must send a request to read the data from the Receiver before the next data input. Otherwise, the next set of data will be written into the UART register and previous data will be lost.

Two D flip-flops were used to store the last interrupt and the read request from CPU respectively. Thus, if read request is not enabled within the period, overflow signal will be asserted.

3. UART Transmitter

UART Transmitter transmits an input byte and output the 8 data bits bit-by-bit in the form of the UART protocol.

Sample rate block

The conditions to trigger the Sample rate block here are different to the Receiver. When the Condition A is met, it will last for 1 cycle, then the falling edge of it will enable **Clock 2** (0101) and **C0** (0011). Thus, Clock 2 AND with C0 to form **Clock 4** (0001). In comparison, the conditions in Receiver to trigger it are more complicated, as the state will affect it as well. For example, for Receiver, sample rate will be Clock 2 during Idle state and be Clock 4 during Start, Data and Ending states. However, only Clock 4 will be used as the sample rate in Transmitter and Clock 4 would be triggered when Condition A is met.

Conditions for transmitter

Diagram below show how the state of the UART will change depending on the conditions.

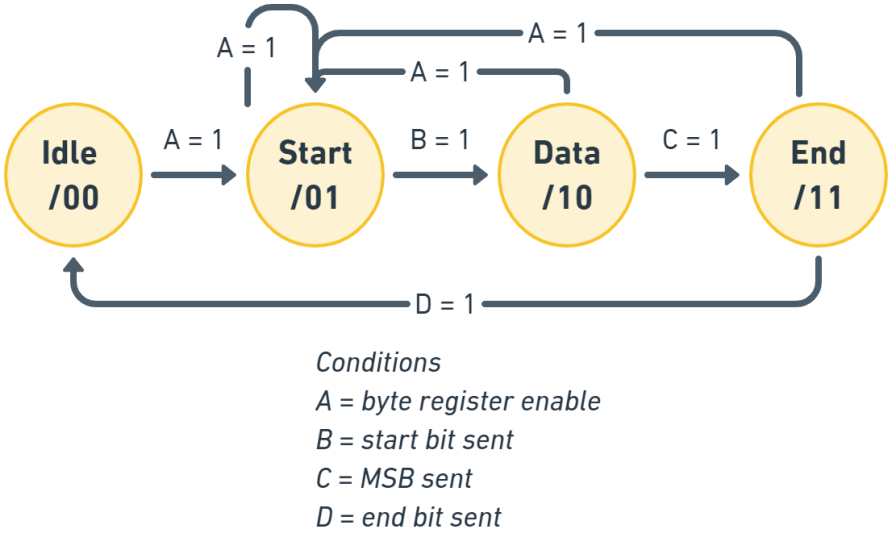


Figure 10 States of UART Transmitter

Condition A: When CPU send a request the UART Transmitter to get the data, Condition A will be asserted.

Condition B: According to the UART protocol, the start bit should be logic 0. When the start bit is sent, Condition B will be asserted.

Condition C: According to the UART protocol, the 8 data bits will be sent from LSB to MSB. When the MSB is sent, it means the Data state of the Transmitter is over and Condition C will be met.

Condition D: According to the UART protocol, the stop bit should be logic 1. When the end bit is sent, Condition D will be asserted.

Series output block

Series output block can receive the one-byte input and output it bit-by-bit during Data state with a 0-to-7 counter at the sample rate of Clock 4(0001). This can be implemented by multiple multiplexers with the counter being the select input. Besides, the output of this block is only one 1 bit. For example, when counter equals to 2, the 3rd bit of the byte will be output. During Start state, the output will always be logic 0, as start bit is 0. During Idle and Ending states, the output will always be logic 1, because between transmissions the transmit signals should be logic 1 and stop bit is logic 1.

Overflow

The overflow for UART Transmitter only depends on one condition.

- If data is requested to be written by CPU during Transmitting state, overflow occurs.

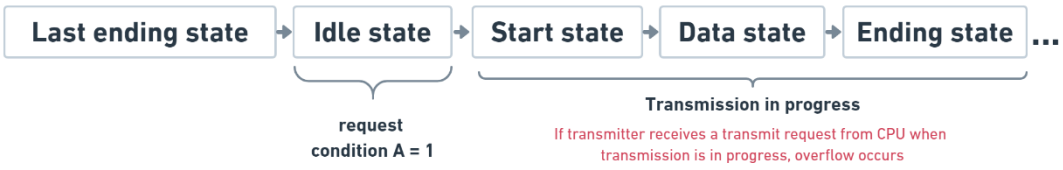


Figure 11 Overflow condition for UART Transmitter

A D flip-flop with enable is used to store the overflow. Once the overflow signal is asserted to high, it will only reset when the rewrite signal is high (refer UART Status).

4. UART Status

Referring to Figure 2, as a memory-mapped device, if the input address for UART equal to 0x800, UART will output its status in the form of the 0Status diagram shown below. Because UART is connected to CPU

as a memory-mapped device, it needs to keep consistency with the data RAM in CPU. Since addresses are available during EXEC1 in CPU, UART status will be updated during EXEC1.

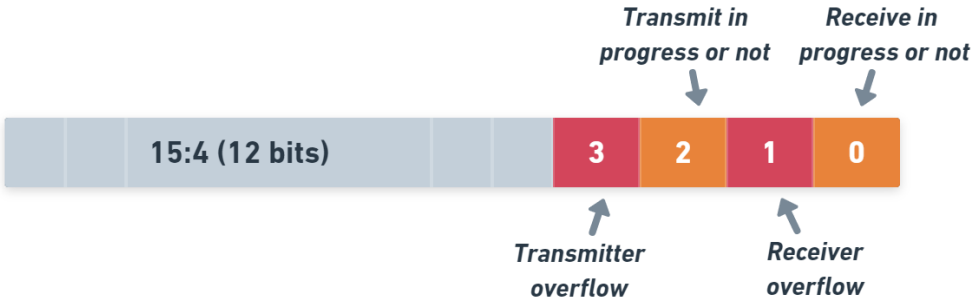


Figure 12 Data bits assignment in UART Status

If overflow occurs in Receiver, data will be lost and new data will be written to the UART register during the present Data state. If overflow occurs in Transmitter, data will be lost and jump straight to next Start state.

Whenever CPU tries to retrieve UART status, overflow will be rewritten because the overflow signal has been acknowledged by the CPU.

5. Context saving

Context saving is implemented to save the data and restore CPU to its original state after an interrupt function. It consists of three parts, Store shadow (1 bit output in CPU), Interrupt at EXEC2 (1 bit output in CPU) and Shadow registers (registers added in specific blocks).

Shadow registers

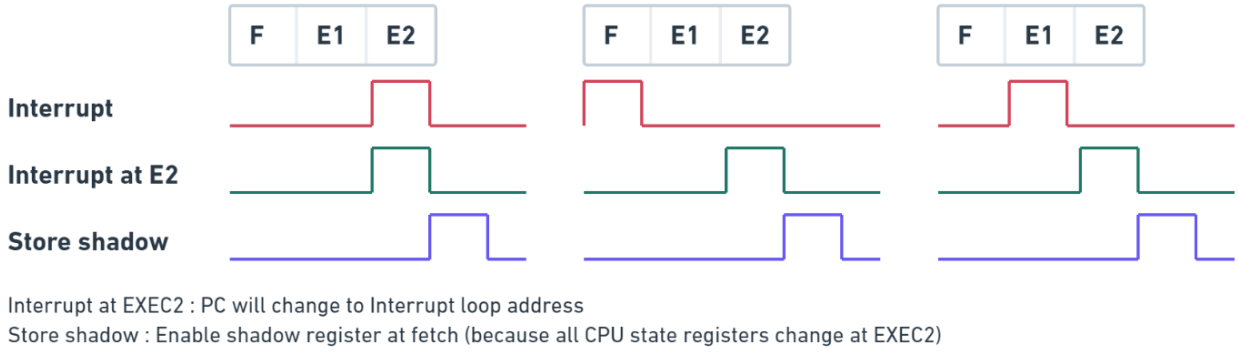
Shadow registers are a set of registers that store the CPU's present data before going into an interrupt function. This is done by connecting the output of the main register (register file's registers, conditional registers, carry-in register, PC register) to a shadow register. Multiplexers were used to select main registers while interrupt function is executing and select the shadow register (where data was stored) when interrupt function ends.

Store shadow

Store shadow is the input used to control the register enable so present data can be stored into shadow registers. It will be enabled during Fetch state (Figure 13) of the MU0 ARM CPU so storage can happen at EXEC1.

Interrupt at EXEC2

PC only change during EXEC2, so the interrupt must be held until EXEC2 to form Interrupt at EXEC2 which is a single clock signal (Figure 13). If PC data changes at the state where the interrupt occurs, it will lose synchronisation with other blocks like Load/Store and ALU.



Interrupt at EXEC2 : PC will change to Interrupt loop address
Store shadow : Enable shadow register at fetch (because all CPU state registers change at EXEC2)

Figure 13 Timing diagram for interrupts at different state

6.3 Floating Point Arithmetic

Identifying the problem

Computers are often required to calculate real-world numbers such as current values that range from 1nA to 1kA. A variable with 50 bits is required to store value up to 10^{15}pA and represent the current values as integers with a resolution of 0.1%. Hence, it should be able to do calculations with the floating-point implementation.

Research

Floating point arithmetic is the one of the most common way to approximate real numbers due to its large dynamic range. It is based on the scientific notation where the radix point can "float", meaning that it is not at a fixed position and will depend on the exponent.

In general, a floating-point number is represented approximately with three main components: the sign, significand, and the exponent.

Figure 14 Floating number representation

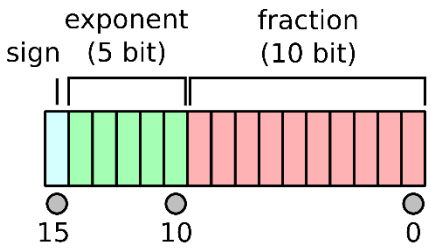
- **Sign** - indicates whether the number is positive or negative.
- **Significand** - a fixed number of significant digits that represents the value of the number.
- **Exponent** - the value of the base power.

Offset-binary representation is used in most IEEE 754 binary floating-point format where a bias is applied to the exponent. Hence, true exponent is obtained by subtracting the offset value.

The number formats used in this project is 16-bit IEEE 754 half-precision and 32-bit IEEE 754 single precision.

16-bit IEEE 754 half-precision (binary16)

According to the Wikipedia, [3]



In binary16,

- **Sign bit**: the most significant bit (15)
- **Exponent:** middle 5 bits (14:10)
- **Fraction:** last 10 bits (9:0)

Figure 15 Format of binary16

- The format is assumed to have an implicit lead bit with value 1 in the fraction unless the exponent field is stored with all zeros.
- Hence, the total precision is **11 bits**.
- Exponent bias: **15**
- Minimum positive normal value: $2^{-14} \approx 6.10 \times 10^{-5}$
- Minimum positive (subnormal) value: $2^{-24} \approx 5.96 \times 10^{-8}$
- The maximum representable value: $(2 - 2^{-10}) \times 2^{15} = 65504$
- Table below shows the exponent encoding:

Exponent	Significand = zero	Significand ≠ zero	Equation
00000_2	zero, −0	subnormal numbers	$(-1)^{signbit} \times 2^{-14} \times 0.significantbits_2$
00001_2, ..., 11110_2	normalized value		$(-1)^{signbit} \times 2^{exponent-15} \times 1.significantbits_2$
11111_2	±infinity	NaN (quiet, signalling)	

Figure 16 Exponent encoding for binary16

32-bit IEEE 754 single-precision (binary32)

According to the Wikipedia, [4]

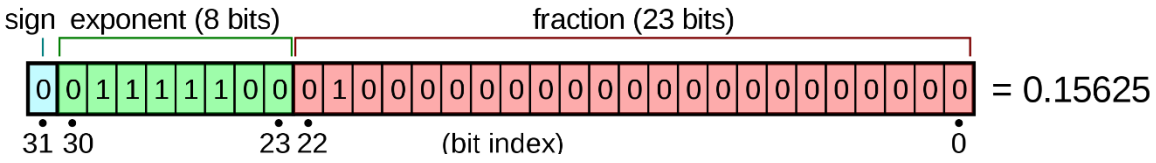


Figure 17 Format for binary32

In binary32,

- **Sign bit:** the most significant bit (31)
- **Exponent:** middle 8 bits (30:23)
- **Fraction:** last 23 bits (22:0)

- The format is assumed to have an implicit lead bit with value 1 in the fraction unless the exponent field is stored with all zeros.
- Hence, the total precision is **24 bits**.
- Exponent bias: **127**
- Minimum positive normal value: $2^{-126} \approx 1.18 \times 10^{-38}$
- Minimum positive (subnormal) value: $2^{-149} \approx 1.4 \times 10^{-45}$
- Table below shows the exponent encoding:

Exponent	fraction = 0	fraction ≠ 0	Equation
00_H	zero	subnormal number	$(-1)^{sign} \times 2^{-126} \times 0.fraction$
01_H, ..., FE_H	normal value		$(-1)^{sign} \times 2^{exponent-127} \times 1.fraction$
FF_H	±infinity	NaN (quiet, signalling)	

Figure 18 Exponent encoding for binary32

There are several exceptions :– NaN, +infinity, -infinity, overflow, underflow

Floating Point Addition / Subtraction

There are a few possible approaches to implement the adder and subtractor. First, it involves only combinational logic. It is easy to implement but may have significant delays. Another method is to use sequential logic. It can achieve higher maximum clock frequency but is complicated as it involves many states and require pipelining. In this design, the combinational logic approach was used.

The implementation of the algorithm used to add and subtract two floating point numbers is explained in the next section by using a worked example. Figure 19 shows the stages in floating-point addition / subtraction.

Operands		**Sign (15)**	**Exponent (14:10)**	**Fraction (9:0)**
A	$+1.727 * 2^7$	0	10110	10111 01000
B	$+1.999 * 2^8$	0	10111	11111 11111

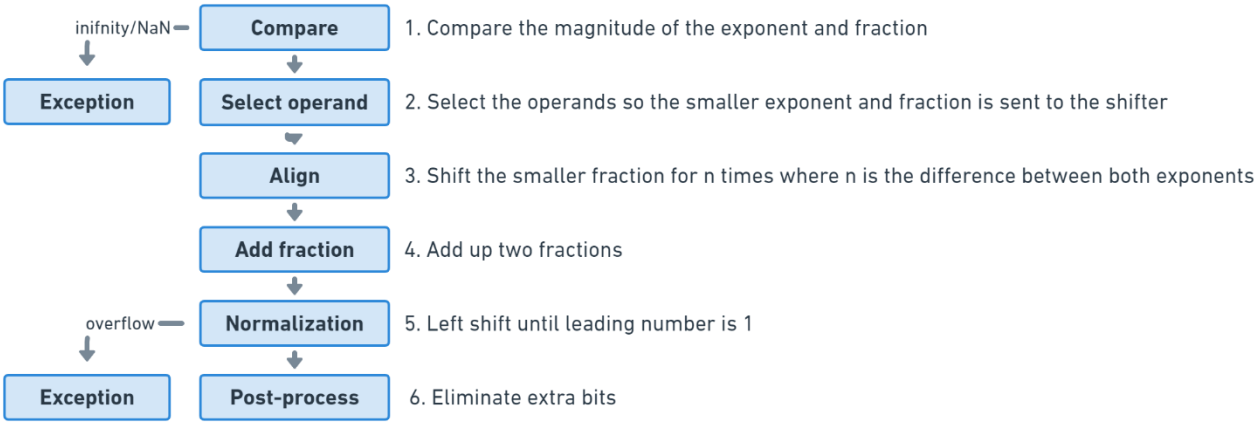


Figure 19 Stages in floating point addition

Computing A + B / A – B:

1. Comparison

In this stage, we filter values that cause exception and compare the operands.

When exponents of A or B has a value of infinity, exception signal is triggered. Otherwise, exponents of operands will be compared. When operand is a subnormal number, implicit bit is set to 0. Otherwise, it is set to 1 by adding an extra bit in front of the most significant bit.

e.g. A: **1**1011 10100 0

B: **1**1111 11111 1

An adder is used to compare and compute the magnitude of the difference between both exponents, which will be useful in the alignment stage.

- Subtraction with 2's complement (Invert and add 1 to operand B)
- Special case: when exponent is 00000_2, it is switched to 00001_2 because in the IEEE 754 format, both numbers represent the same exponent value.
- Carry out is used to determine which exponent is larger. Algorithm reasoning is as follows:
  - **A – B:**

Dec	Binary	Result in binary	Carry out	Comparison
4 – 2	0100 – 0010	0010	1	A > B
4 – 6	0100 – 0110	1010	0	A < B
5 – 1	0101 – 0001	0100	1	A > B
6 – 3	0110 – 0011	0011	1	A > B
6 – 7	0110 – 0111	1111	0	A < B
7 – 7	0111 – 0111	0000	1	A = B

Figure 20 Table above shows the pattern of the carry out while doing 2's complement subtraction

- When A < B, carry out = 0
- When A > B, carry out = 1, result ≠ 0
- When A = B, carry out = 1, result = 0

e.g. exponent A - exponent B = $10110_2 - 10111_2 = 10110_2 + 01001_2 = 11111_2$

To compute the magnitude, the process of 2's complement is inverted to obtain a positive number for the difference between the exponents,

e.g. 11111_2 in binary, -1 and invert to get positive number → 00001_2, ∴ exp. difference = 1

2. Select operand

Selecting operand is required to direct the right fraction and exponent to the right shifter for alignment. There are several conditions to consider while selecting.

Exponent A < Exponent B	Select fraction A to right shifter
Exponent A > Exponent B	Select fraction B to right shifter
Exponent A = Exponent B	Compare fraction A and fraction B Select the smaller fraction to right shifter

Besides, the bigger exponent is always selected to the subsequent stages.

e.g. Exponent A < exponent B, hence fraction A (11011 10100 0) will be selected into the right shifter; exponent B is selected to the subsequent stages.

3. Alignment

An additional 2 bits are added (sticky bit and guard bit) at the least-significant end of the fraction in the right shifter, details as follows:

i. Sticky bit: the last shifted bit

ii. Guard bit: when shifted bits consist of 1, bit with logic 1 is added, achieved by using bus compare and NOT gates.

The number of times required to right shift the fraction is determined by the exponent difference.

e.g. ***Step 1. Right shift by 1 (exponent difference = 1):*** **0**1101 11010 00

Step 2. Bus compare the shifted bits (in this case, none are shifted so no sticky and guard are added): **0**1101 11010 00**0**

4. Addition of fraction

The signs of the operand are compared with a XOR gate to determine whether the operation should be addition or subtraction.

Note:

- In this design, the first operand of the floating-point adder is set to be the larger number
- This allows us to set the sign of the output to the sign of the number in the first operand, which largely simplify the circuit involved.
- The algorithm applies to most cases, except one special case when:
    - A < B and operation is subtraction
    - Result of A – B will be negative number so sign must be switched to negative

During addition, when the result of the sum has a carry out of 1, the least significant bit is discarded because the carry out will be added to the most significant end of the fraction. During subtraction, when the result of the subtraction has a carry out of 1, carry out will be discarded.

e.g.

Addition	11111 11111 100 + 01101 11010 000 = 01101 11001 100 carry out 1 Output: 10110 11100 110
Subtraction	11111 11111 100 – 01101 11010 000 = 10010 00101 100 Output: 10010 00101 100, A < B, sign switched to negative (1)

5. Normalization

I. Leading one detector

Every bit is identified from the most significant bit to determine the leading one position, and the number of times needed to left shift to obtain normalization.

e.g. ***Addition:*** **1**0110 11100 110 – there is a leading one at the MSB, shift number = 0

Subtraction: **1**0010 00101 100 – there is a leading one at the MSB, shift number = 0

Parallel leading one detector was also implemented as part of the optimization and improvement stages. Refer leading one detector of floating-point with 32 bits.

II. Left shift normalizer [5]

Fractions and exponents are shifted in this stage to achieve normalization.

Diagram below shows the conditions to be considered:

- Eout represents the output from select operand stage (the bigger exponent)

- Shiftnum represents the number of times needed to left shift

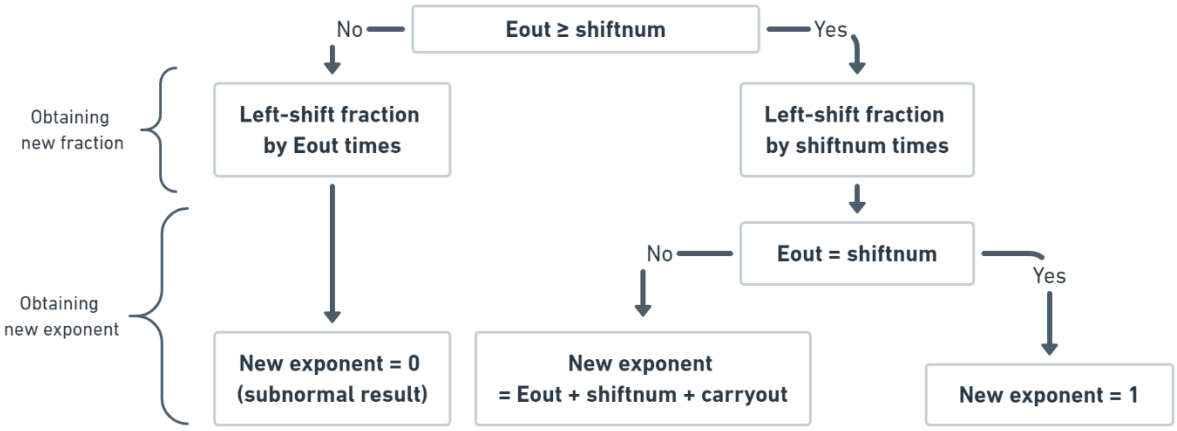


Figure 21 Conditions for left shift normalizer

- Fraction is left-shifted by Eout or shiftnum times depending on whether Eout is larger or equal to shiftnum.
- New exponent is 0 when Eout is smaller than shiftnum which will produce a subnormal result.
- Otherwise, new exponent is obtained by checking if Eout is equal to shiftnum. If yes, new exponent is 1; otherwise, it is equal to the sum of Eout, shiftnum and carryout.

e.g.

	Eout	Shiftnum	Carry out	New exponent	New fraction
Addition	10111	0	1	11000	**1**0110 11100 110
Subtraction	10111	0	0	10111	**1**0010 00101 100

6. Post-process

The most significant bit (implicit bit) and the last 2 least significant digits of the fraction are discarded.

e.g.

Operation		**Sign (15)**	**Exponent (14:10)**	**Fraction (9:0)**
A + B	$+1.432 * 2^9$	0	11000	01101 11001
A – B	$-1.136 * 2^8$	1	10111	00100 01011

Floating-point Multiplication

There are a few possible approaches to implement the multiplication. [6]

- **Array multipliers**

  By using sequential method, the period for each clock cycle can be decreased. Carry save adders can also be used instead of ripple adder to reduce the total time delay.

- **Wallace Tree multiplier**

  This method is complete combinational logic, so it only requires one clock cycle to complete. However, it involves more amount of logic gates that might increase power consumption. The main advantage of this method is that it uses an algorithm that reduces the partial product and hence reduces the number of adder delays.

The implementation of the algorithm used to multiply two floating point numbers is explained in the next section by using a worked example. Figure 22 shows the stages in floating-point multiplication.

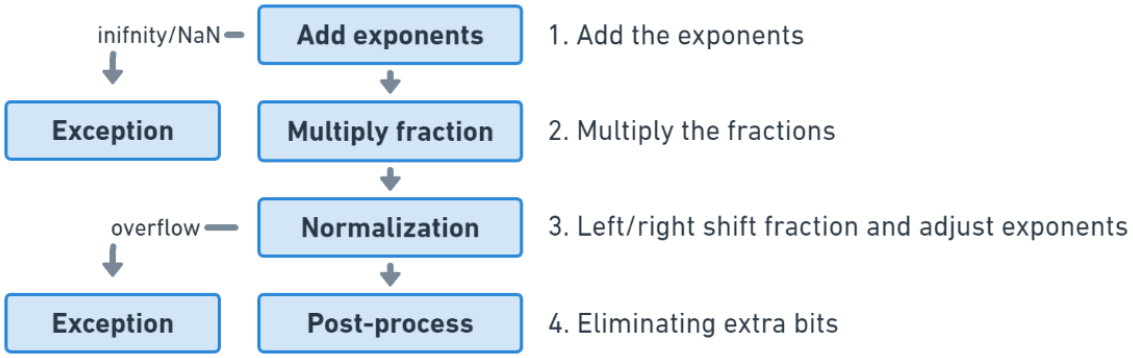


Figure 22 Stages in floating- point multiplication

Operands		**Sign (15)**	**Exponent (14:10)**	**Fraction (9:0)**
A	$-1.539 * 2^{-6}$	1	01001	10001 01000
B	$+1.856 * 2^{-13}$	0	00010	11011 01101

Computing A x B:

1. Addition of exponents

This stage filters the values that cause exception and add the exponents of the operand.

When exponents of A or B has a value of infinity, exception signal is triggered. When operand is a subnormal number, implicit bit is set to 0. Otherwise, it is set to 1 by adding an extra bit in front of the most significant bit.

e.g. A: **1**1000 10100 0

B: **1**1101 10110 1

Both exponents are then added (based on rule of indices)

The carry out and an extra bit 0 is added to the most significant end before the bias (15 in binary16 case) is subtracted from the sum of the exponents to prevent overflow (results lie outside signed range)

Explanation for why subtraction is needed:

- **E** represents **unbiased** exponents; **e** represents **biased** exponents

  e = e1 + e2, whereas e1 = E1 -15, e2 = E2 - 15

  e_{result} = E + 15 = e1 + e2 + 15

  e_{result} = (E1 - 15) + (E2 - 15) + 15

  $\boldsymbol{e_{result}}$ **= E1 + E2 -15**

e.g. exponent A + exponent B = $01001_2 + 00010_2 = 001011_2$

Result – $15_{10} = 0001011_2 - 0001111_2 = 1111100_2\ (-4_{10})$

2. Multiplication of fraction

Two different multiplication methods were implemented, and evaluation was carried out to determine the most suitable approach.

Computing A x B where A is the multiplicand and B is the multiplier:

- Serial multipliers (sequential)

  The algorithm of serial multiplier is illustrated in the flowchart below:

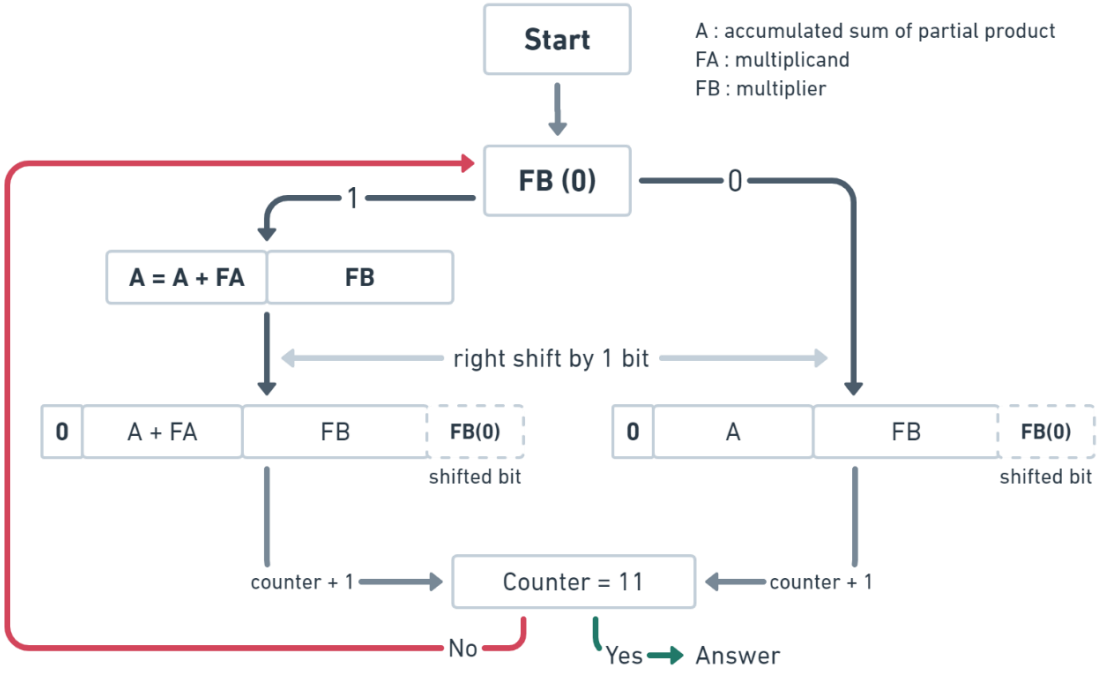


Figure 23 Flowchart for serial multiplier

- When the LSB of the multiplier is 0, the combined bits that consist of accumulated sum of the partial product (which is 0 when start) and multiplier is shifted right by 1 bit.
- When the LSB is 1, multiplicand is connected to the most significant end of the multiplier. It is then also shifted right by 1 bit.
- After the shift process, the counter is updated. Counter is used to count the number of bits shifted to track the multiplication process.
- When below 11, the process of checking the LSB of multiplier is repeated. It will end when all bits of multiplier are shifted.

Delays

The most simple and conventional method of multiplication has a large delay. A N-bit ripple adder's total time delay is equal to the sum of *n* full adder delay. For multiplication, it will take *n* times to do addition of *n* partial products, hence the total time delay would be *n*n* delay of a FA.

This time delay could be largely reduced by replacing it with n-bit carry save adder (CSA).

- CSA store its carry out and sum into two DFFs, hence the propagation time delay of the carry bit could be eliminated.

The algorithm needs to compute n partial sum. CSA is used for the first n - 1 partial summation. For the last partial sum computation, it will reconfigure itself from CSA to n bit ripple adder (n bit carry + n bit sum) to calculate the final sum and the final carry-out bit. [7]

- Therefore, the time delay would be reduced to n – 1 + n = 2n – 1, about 10 times less than that in initial shift method (which is n*n). [8]

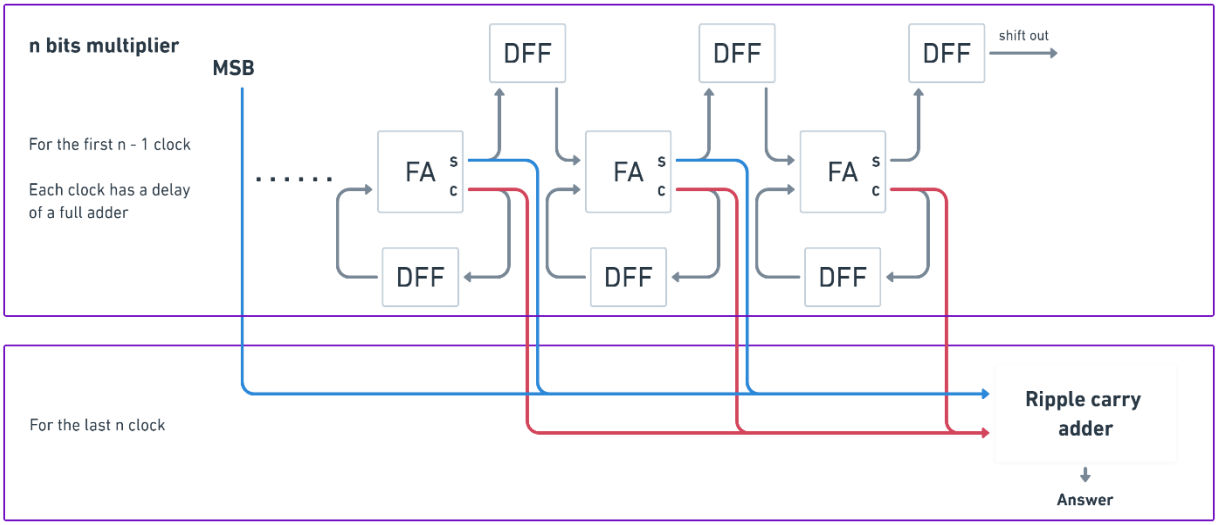


Figure 24 Delays for serial multiplier

- Wallace Tree multiplier (combinational)

  There are three stages to achieve multiplication with the Wallace Tree design.

  Multiply

  Each bit of the multiplier is multiplied with the 16/32 bits multiplicand by using AND gates.

  Reduction of partial products

  In this stage, 3 partial products are combined into 2 by using half adders and full adders. Half adders are used when only 2 bits are involved in the summation whereas full adders are use when 3 bits are involved in the summation because half adders have 2 inputs whereas full adders have 3 inputs.

  The 2 partial products formed each represent sum and carry outs from the 3 partial products. The bit is unchanged if no other bits are involved in the summation.

  e.g. frac. A x frac. B = 10110 11011 01110 00010 00

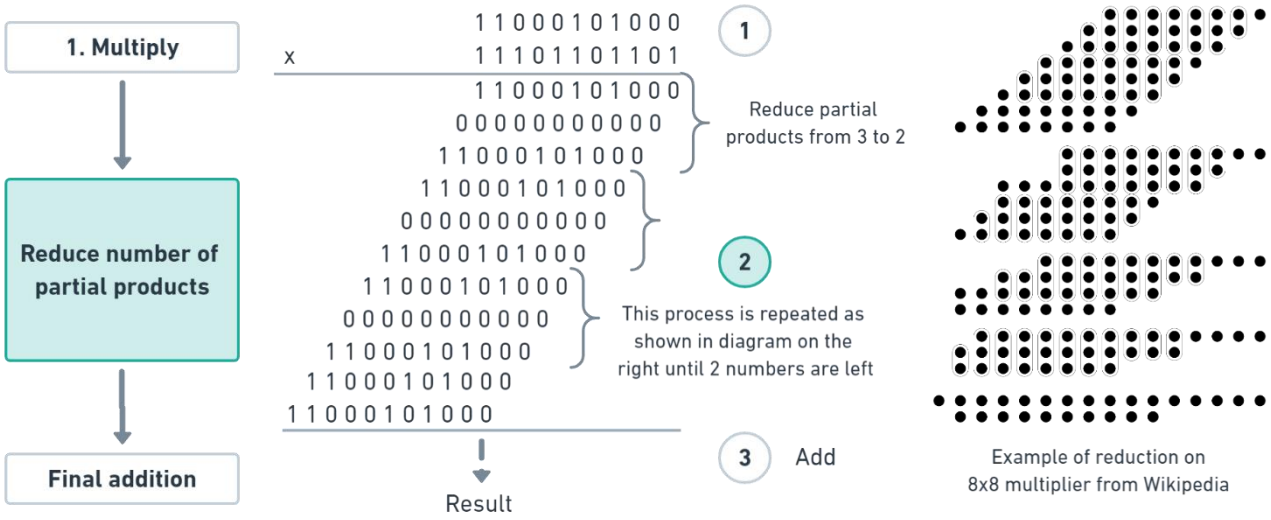


Figure 25 Flowchart of Wallace Tree multiplier implementation

Delays

Delays in Wallace Tree multiplier depend on the number of bits.

e.g. For 11 bits, the maximum number of partial products is 16 and the reduction process:

$11 \rightarrow 8 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \Rightarrow$ Hence, there are 6 adder delays.

<u>Evaluation of both multipliers</u>

Wallace tree multiplier involves a larger amount of logic cells[1] than serial multiplier but smaller delays. Instruction speed is worked out using the equation:

$$\frac{the\ number\ of\ clocks\ cycles\ required\ to\ perform\ the\ multiplication}{maximum\ clock\ frequencu}$$

	Serial multiplier	Wallace Tree multiplier
Max clock frequency	Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 218.10 MHz (PASS at 12.00 MHz)	Max frequency for clock 'clk$SB_IO_IN_$glb_clk': 190.22 MHz (PASS at 12.00 MHz)
Delays	2.2 ns logic, 2.4 ns routing	0.5 ns logic, 3.5 ns routing
Instruction speed	$\frac{11\ clocks}{218.10MHz} = 50.04ns$	$\frac{1\ clock}{190.22MHz} = 5.26ns$

Figure 26 Table of comparison of serial and Wallace Tree multiplier

Since Wallace Tree multiplier has smaller delays and higher instruction speed, it is more suitable to be implemented.

3. Normalization

I. Leading one detector (series)

Every bit is identified from the most significant bit to determine the leading one position, and the number of times needed to left shift to obtain normalization.

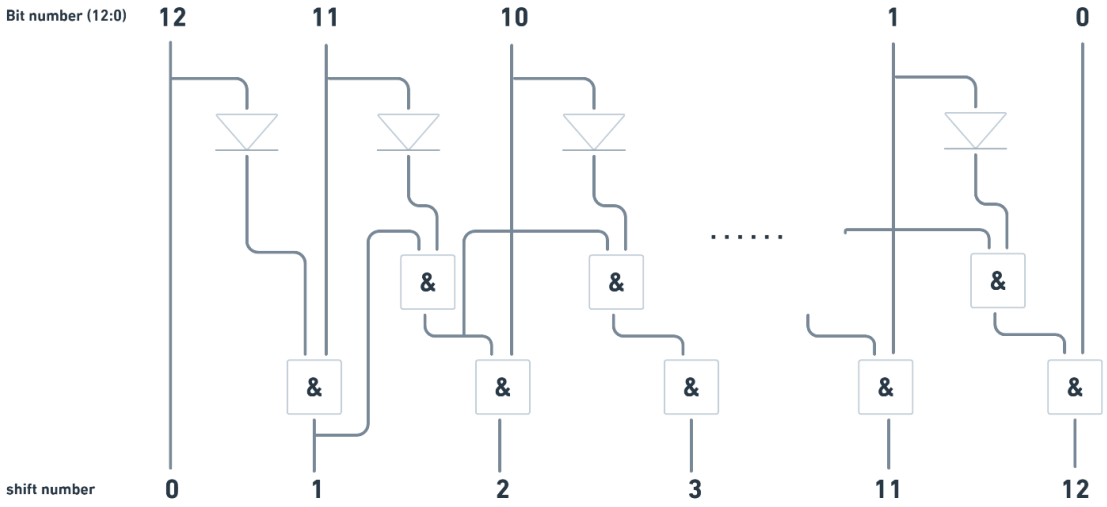


Figure 27 Leading one detector logic

[1] Evaluation from Issie synthesizer did not produce the expected result for the number of logic cell, hence the screenshot of result will be included.

Parallel leading one detector was also implemented as part of the optimization and improvement stages. Refer leading one detector of floating-point with 32 bits.

Before left/right shifting, few conditions are considered, as illustrated in the diagram below:

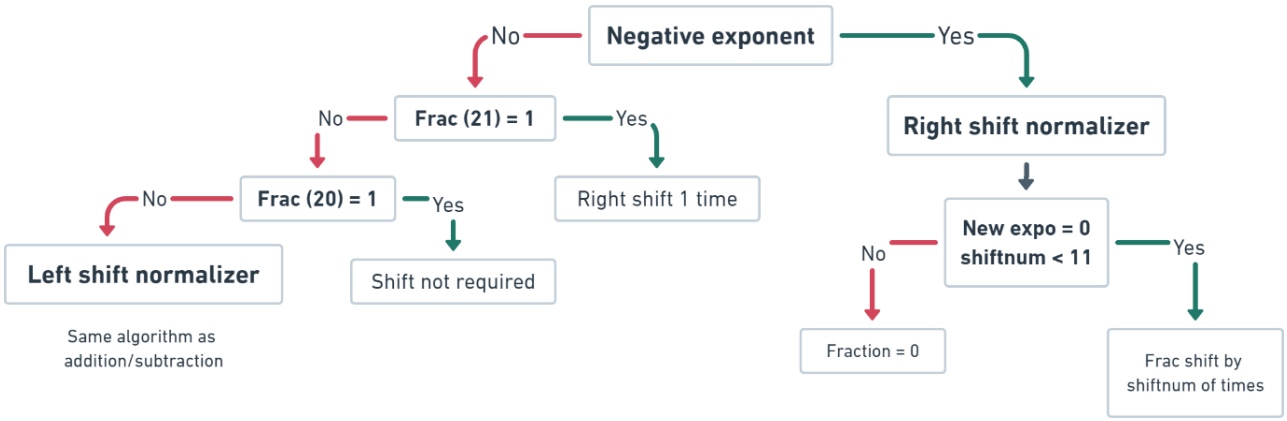


Figure 28 Conditions for normalizer

- **Negative exponent –** determine by checking whether MSB is 1.
- When exponent is negative, ***right shift normalizer block*** is used.
- When exponent is positive, the MSB of fraction is checked to determine whether shifting is required.
- When MSB of fraction is 1, right-shifting by 1 bit and adding one to exponent is required. If exponent exceeds 11111_2, overflow is detected, and exception will occur.
- When bit 20 is 1, shifting is not required.
- Otherwise, the ***left shift normalizer block*** is used.

II. Left shift normalizer

The algorithm for left shifting for multiplication is the same as the algorithm used in addition and subtraction. Refer page

III. Right shift normalizer (for negative exponent)

Fractions and exponents are shifted in this stage to achieve normalization.

- **Note:** In the IEEE 754 format, 00000_2 and 00001_2 represent the same exponent value.

- Hence, the number of times required to right shift the fraction is the sum of the modulus of the negative exponent and 1 which is represented in the diagram below.

Figure 29 Explanation for why addition of 1 is needed

- The modulus of the negative exponent is achieved with 2's complement.
- When the result is larger than 10 (which is larger or equal to the size of the fraction), fraction will output as 0.

e.g. (negative) exponent = 1111100_2, inverted number = 4_{10}, ∴ shiftnum = 5

Right shift by 4 times: 00 00010 11011 01101 11000 01

New exponent = 00000

4. Post-process

The 2 most significant bits (integer bits) and the last 10 least significant digits of the fraction are discarded. The sign of the result can be easily obtained using a XOR gate.

e.g.

Operation		**Sign (15)**	**Exponent (14:10)**	**Fraction (9:0)**
A x B	$-0.0889 * 2^{-14}$	1	00000	00010 11011

Floating Point Arithmetic (Single-precision 32 bits / binary32)

It shares the same flowchart with the 16-bits Arithmetic block.

Some logics are redesigned to accommodate a larger scale (32 bits).

1. Multiplication of fraction

Initial Method for 16 bits:

Wallace Tree multiplier: The number of logic gates used in this method would escalate with increment in the number of bits involved. Hence, this method is discarded for 32-bits considering its large power consumption.

Final Implementation for 32 bits:

Serial multipliers: Refers to 16-bits Carry Save Adder Part. Increased the bandwidth of the wire from 16 to 32-bits.

2. Normalization

Initial Method for 16 bits:

Series Leading one detector:

A 22 series leading-one-detector was used in 16-bits multiplication block and a 11 series leading-one-detector was used in 16-bits addition block. Signal propagation of many gates was involved, which halted the multiplication process and added up to the total delay. The efficiency would be even lower if we insist on using the same method to compute the 32-bits floating point multiplication/addition (a 48-bit series leading-one-detector is needed). Therefore, we redesigned the leading-one-detector from series to parallel.

Final Implementation for 32 bits:

Leading one detector (parallel):

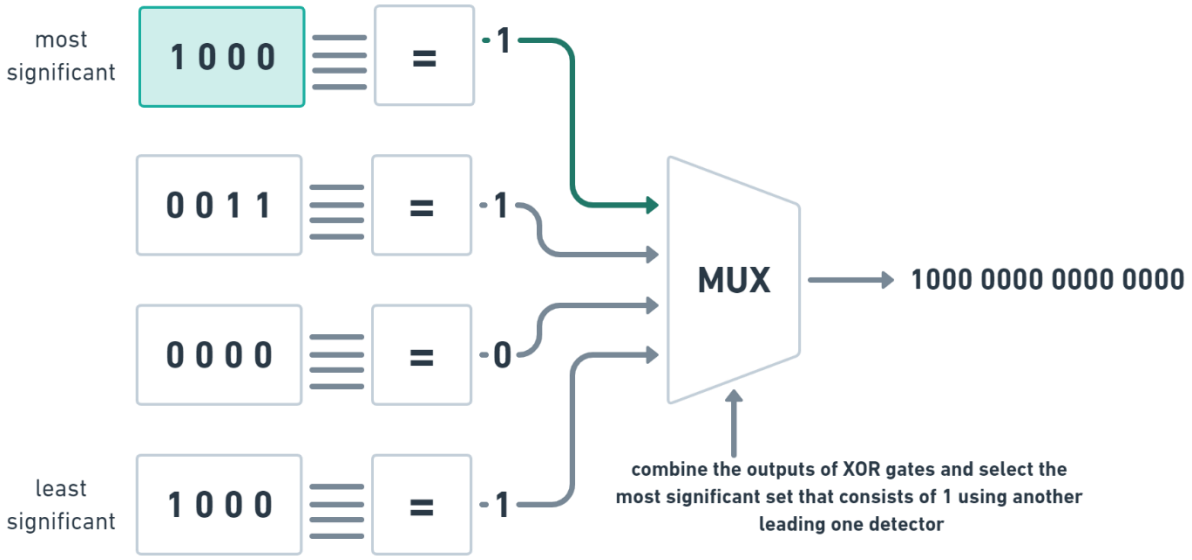


Figure 30 Parallel leading one detector

By first separating a n-bits number into parts of 4 bit. A 4 bits leading-one-detector (LOD) is then used to detect the leading-one in each part and output 4 bits keeping only the leading one.

eg. In the 2nd set 0011, 0011 → 0010

A XOR gate is connected to the output of each LOD to determine whether leading-one is present in the set. Then, all XOR gate outputs are combined. These combined bits will be input into another LOD to determine which part of the n-bit number consists of the leading-one. Finally, this result is used to select the part with the leading-one.

Note: At the multiplexer, each part is pre-programmed with bits 0 to from a 16 bits output

eg. In the 2nd set, output of LOD is 0010, hence at multiplexer, it will become 0000 0010 0000 0000)

This leading-one-detector is particularly useful for multiplication of smaller numbers that requires large number of bit shifting.

6.4 Dual Core

Identifying the Problem

When multiple CPUs share the same data memory, conflicts occur when both CPUs want to access the data memory at the same time. Hence, the CPU should have an arbitration logic to decide which CPU would have the current access and installing a stall function for the other one and thus allow different functions to execute simultaneously.

Research:

A dual core is a CPU with two separate processing units (cores) in the same integrated circuit. [9] Each processor has its own cache which enables it to read and executes program instructions. Two processors are linked with each other, reading and executing instructions at the same time. It could increase the speed up to twice as fast compared to a single core. However, it could not always operate twice as fast because programs are not optimised for multiprocessing.

Dual core implementation

This is done by integrating two MU0 ARM CPUs together with independent instruction memory and shared data memory. To manage the access, Bus CPU blocks, arbitrator and stall are needed.

Design plan for dual core system:

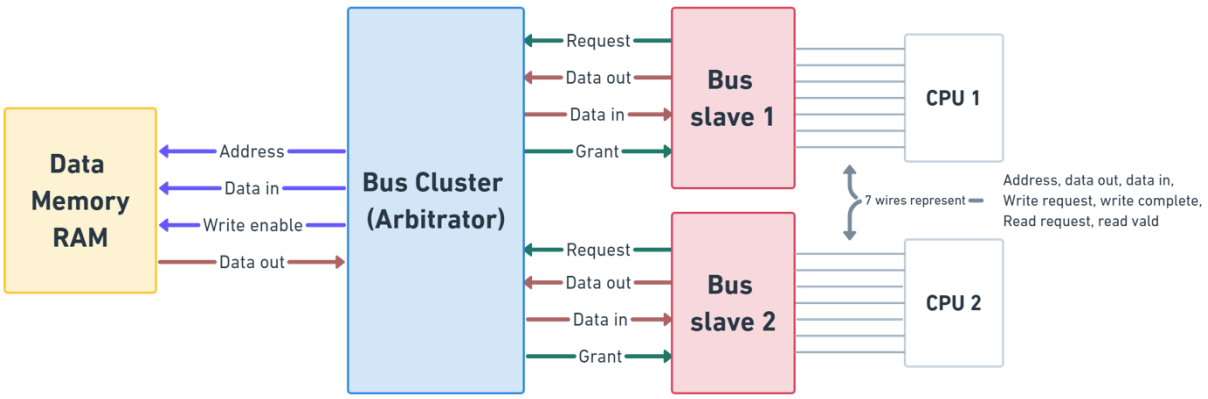


Figure 31 Dual core system top level

1. Bus CPU (Bus slave)

A BUSCPU block is a link between the Arbitrator and the CPU.

Input ports:

From CPU	
Address (12 bits)	Instruction memory address at Fetch state and data memory address at EXEC1 and EXEC2
Write Data (16 bits)	Data to be written to RAM
Read Request (1 bit)	When a Load instruction is executed at EXEC1, the signal will be asserted and data will be available at EXEC2.
Write Request (1 bit)	When a Load instruction is executed at EXEC1, the write enable signal will be asserted so data can be written at EXEC2.
Priority Code	*Refer to Priority section*

From Arbitrator	
Grant control (1 bit)	Signal will be asserted to high when the CPU grants access after a read/write request.
Read Data (16 bits)	Output of data memory

Note: Request and Grant signals are always linked and acts as a communication path between CPU and Bus Controller.

Output Ports:

To CPU	
Read Data (16 bits)	Contents retrieved from data memory
Read Valid/Write Complete (1 bit)	Asserted when access is granted after a read/write request signal

To Arbitrator	
WEN Data (29 bits)	Write Enable Input (1 bit) + Data Address Input (12 bits) + Data Write (16 bits)
Request Control (2 bits)	Priority Code 1 bit + Write Request or Read Request 1 bit

2. Arbitrator (Bus controller/Cluster):

There are 3 possible approaches for bus arbitration system.[10]

Method	**Daisy Chaining**	**Polling / Rotating Priority**	**Fixed priority / Independent Request**

Advantage	• Simple • Scalable	• Simple • Unfixed priority	Quick response
Disadvantage	• Fixed priority • Large delay (the entire system will stop working if one device fails)	• The entire system will stop working if one device fails • Adding bus masters would increase the number of address lines of the circuit.	Many control lines are required (figure 31)

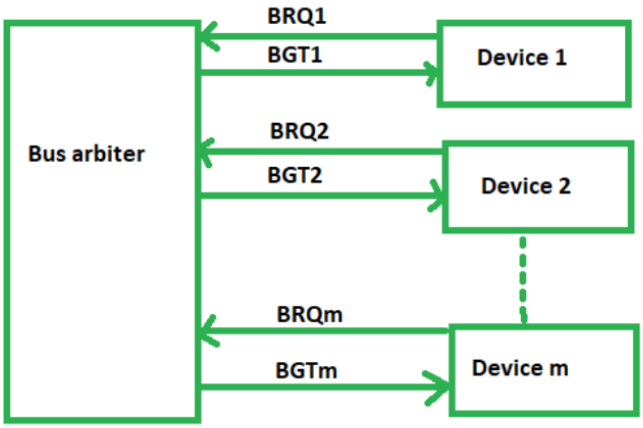


Figure 312 Fixed priority / Independent request method, source: website

Independent Request Method is chosen as our general design direction for our arbitration logic, considering it is the fastest among the three. Additional logic was implemented to improve priority flexibility. (Refers to Priority section)

Grant Request

There are 2 conditions to consider when granting access:

i. Only one CPU requested access to data memory.
   - A grant signal would be given to the CPU that made the request.

ii. Both CPUs requested access data memory at the same time. This is when **priority** is needed to be considered:

Priority code is designed to be a N-bits signal where the number of bits used will depend on the background requirement[2].

[2] Background requirement: If a CPU is designed to focus on specific instructions that are important, these specific instructions in this CPU should have the highest value of priority code.

The value of the priority code number determines the priority level of the instruction. The higher the priority code number, the higher the priority level of the instruction. When there are multiple requests from CPUs, arbitrator will give access to the CPU that has the highest priority code number through the arbitrator bus line. Arbitrator will then grant CPU the access in descending order of the priority code number.

This code is designed to be changed depending on the requirements and could vary in different scenarios.

e.g. *If there are six different main functions that would possibly occur in each CPU, the event that needs to be dealt most urgently shall have the highest priority code.*

In this design, the priority code is a one-bit signal and the shared device will be the data memory (RAM) because:

- Both CPUs will only need to access the data memory when they have one of these two instructions: LDR and STR.
- Only one interrupt function is needed to be executed.
- No other specific background requirements.

Priority code is set to be high when the current executing program is from the UART interrupt loop so data can be transmitted as soon as it is received. The arbitration block will then check the priority code and grant control to the one with a larger priority code (which is 1 in this case).

If their priority codes are the same, then the arbitration block will assign priority alternately between CPU1 and CPU2.

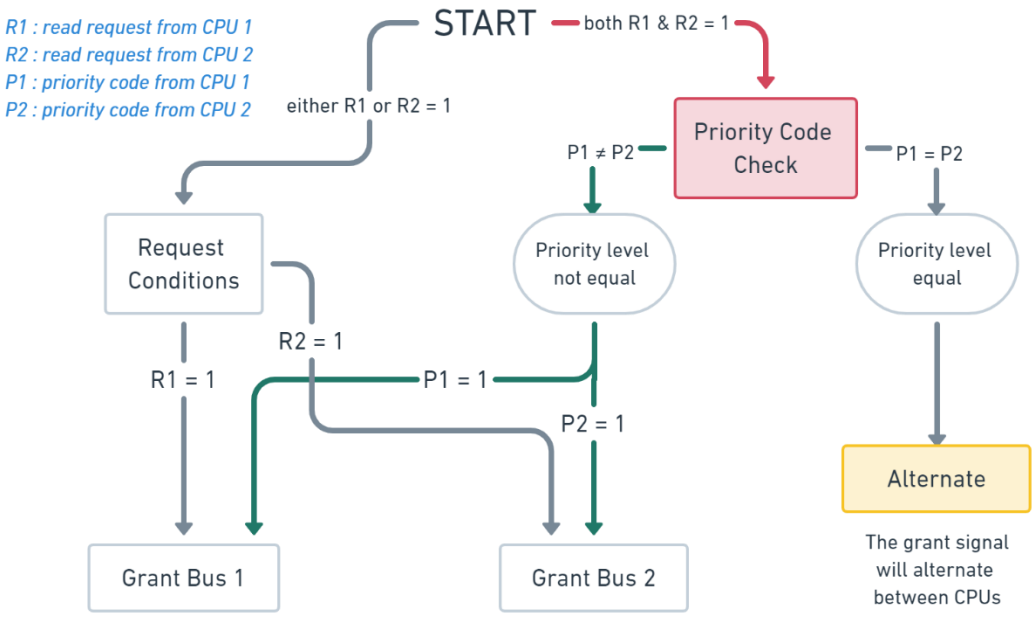


Figure 33 Flow chart for priority code

3. Stall

When two CPUs are access same hardware (eg. data memory), there will be an overlapped operation or overwrite, thus one of the CPUs needs to be stalled.

There are four parts in the CPU that needs to be changed to implement a stall.

1. Stall at any time:

   To pause a CPU, the first step would be pausing the state machine by adding a stall input to it. When the stall is high, the state machine would remain at its current state. A stall will hold the state for two clock cycles.

Clock	1	2	3	4	5
State	F	E1	E2	E2	F
Stall			S		

2. Stall at E1:

   Delay the stall signal for one clock cycle and close the register to hold IR instruction during a stall. Multiplexer is used to retrieve the data from register.

Clock	**1**	**2**	**3**	**4**	**5**
State	F	E1	E1	E2	F
Stall		S	Hold stall for another cycle		
IR		Open Register	Close Register	Close Register	Reopen register

3. Stall at E1:

   Delay the stall signal for one clock cycle and close the register to hold Rm during a stall.

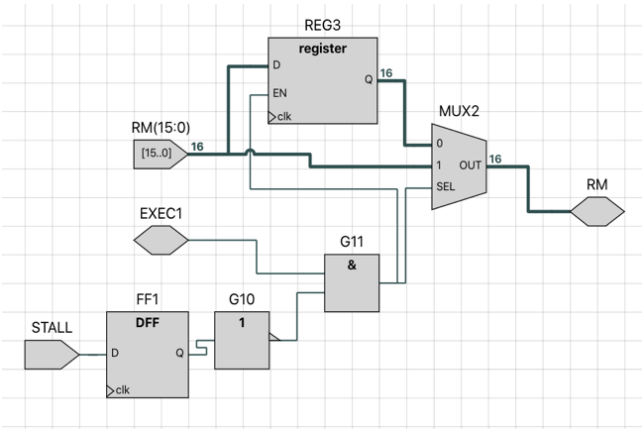


Figure 32 Register to hold Rm and multiplexer to retrieve data

Clock	1	2	3	4	5
State	F	E1	E1	E2	F
Stall		S	Hold stall for another cycle		
LS.Rm		Open register	Close register	Close Register	Reopen register

4. Stall at E2:

   Close the register output in PC block to hold PC address during a stall.

Clock	**1**	**2**	**3**	**4**	**5**
State	F	E1	E2	E2	F
Stall			S		
PC	PC register close	PC register close	PC register close	PC register open	
PC.Q	N	N	N	PC	PC

7. Integration with the CPU

Memory-mapped is used to integrate the UART and floating point arithmetic with the CPU.

7.1 UART

Integration of UART with the CPU was mostly done when interrupt function was implemented (Refer to context saving from UART section)

7.2 Floating point arithmetic

Diagram below represents the logics and memory addresses used:

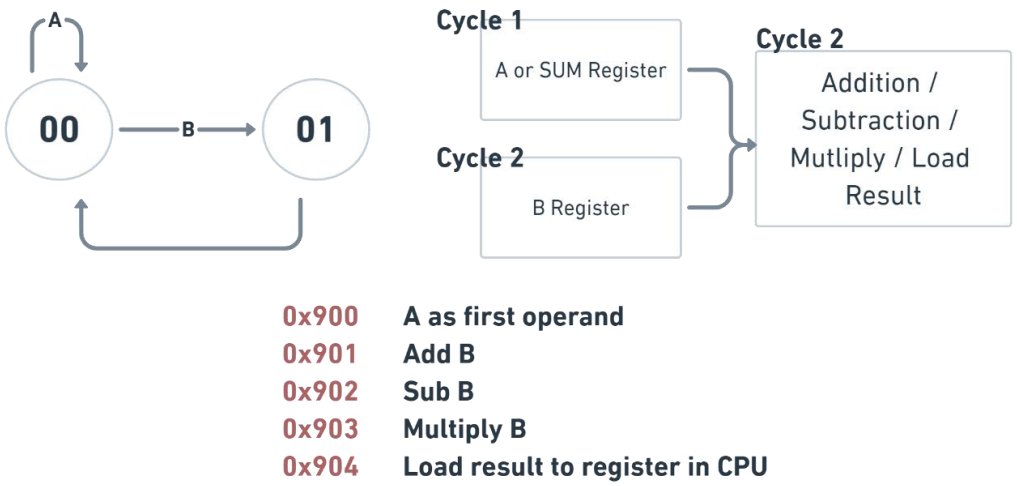


Figure 335 Memory-mapped floating-point arithmetic

There are 2 cycles in total to carry out a floating-point arithmetic operation. In cycle 1, A or SUM Register is loaded whereas in cycle 2, B Register is loaded. The arithmetic operation is determined by the memory addresses as specified above.

When the operations are finished, memory 0x904 will be accessed to load the result to one of the 4 registers in CPU.

Note:

Stalls are implemented because floating-point unit is integrated with the CPU by memory-map (hence data memory will be accessed when there are LDR/STR instructions). There will be conflict when two CPUs are requesting control of the Data Memory / Floating-point unit / UART at the same time. The address to trigger the FP unit would then be possibly extended for one more clock cycle during a stall, so the arithmetic operation would be triggered twice, leading to a computation error.

8. Testing

The testing and debugging process was done systematically where few practices are repeated when an error is found. This includes:

- Starting from block level testing before proceeding to top level testing
- Using step simulation when circuit logic is completely combinational
- Adding relevant outputs to track lower-level design
- Using step-up counters as input of ROM to simulate inputs
- Using waveform simulator to ease debugging process because more wires will be able to be detected without having to add wire labels or outputs.
    - Added ROM and constant to inputs.
    - Able to check each input and output of the blocks.

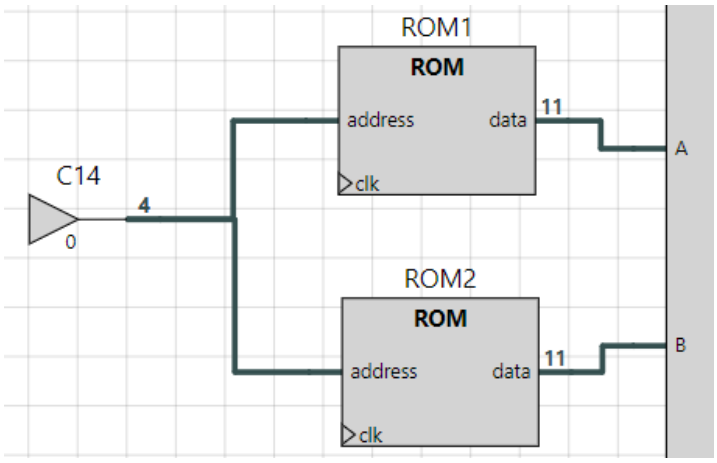


Figure 346 Example of using waveform simulator for combinational logic

8.1 UART

Transmitting the sum of all the bytes receive by UART receiver so far by triggering the interrupt function and checking if contest saving works properly.

UART Receiver:

Byte 1																
Clock	0	1	2	3	4	5	6	7-10	14	18	22	26	30	34	35-38	39-42
Input data	0	1	1	0	0	0	0	1	0	1	0	1	1	0	1	1
Note	N	Idle		Start bit				First bit							Last bit	End bit

Byte 2											
Clock	48	52	56	60	64	68	72	76	80	81-84	85-89
Input data	1	0	0	0	1	1	1	0	0	0	1
Note	Idle	Start bit	First bit							Last bit	End bit

CPU Background Code:

Memory address	Assembler	IR Register Shift Immediate constant LDR /STR	Note
0x0	MOV R0 #0x01	0xC250	R0 = 0x0001
0x1	LDR R1 mem[R0-1] WB	0x0784	R0 = 0x0000 R1=0xC250
0x2	LDR R3 mem[R0] WB R0+2	0x0F48	R0 = 0x0002 R3=0xC250
0x3	ADDS R1=R1+Op2 Op2 = R1 LSR#1	0x8517	R1=2378 C=1
0x4	SBC R1=R1-Op2+C-1 Op2=R3 LSR#1	0xB417	R1=0xC250
0x5	MOV R3 #0x1010	0xCE0A	R3=0x000A
0x6	JEQ #0x008	0x6008	Jumpto8 Error
0x7 & 0x8	STA mem[R0] = R0 LDR R22=mem[R0]	0x0000 0x0A00	mem[0x0002]=0x0002 R22 = 0x002

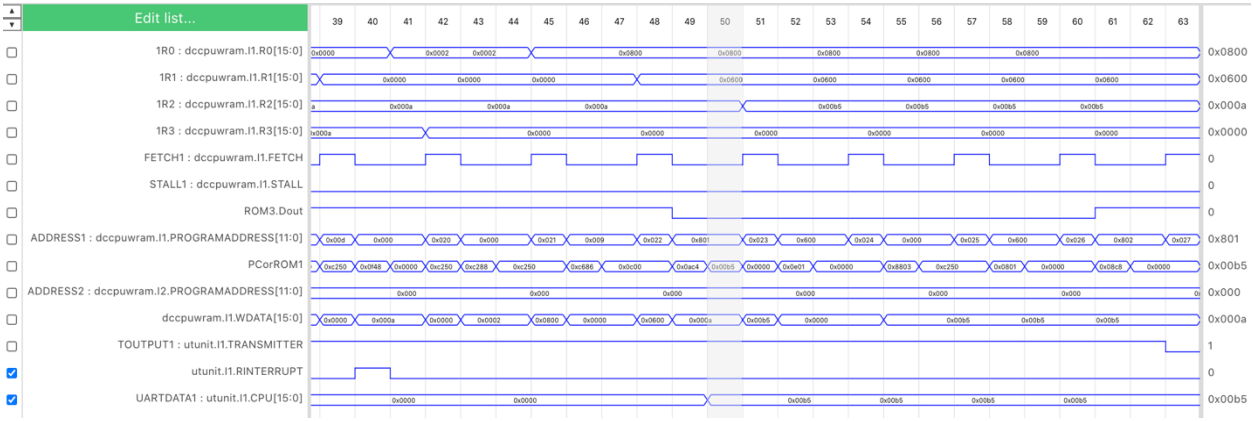
0x9	STA mem[R0] = R3	0x0C00	mem[0x0002]=0x000A
0xA	LDR R2 = mem[R0]	0x0A00	R2=0x000A
0xB	MOV R0 #0x01	0xC250	R0 = 0x0001
0xC	LDR R1 mem[R0-1] WB	0x0784	R0 = 0x0000 R1=0xC250
0xD	LDR R3 mem[R0] WB R0+2	0x0F48	R0 = 0x0002 R3=0xC250
0xE	ADDS R1=R1+Op2 Op2 = R1 LSR#1	0x8517	R1=2378 C=1
0xF	SBC R1=R1-Op2+C-1 Op2=R3 LSR#1	0xB417	R1=0xC250
0x10	MOV R3 #0x1010	0xCE0A	R3=0x000A
0x11	JEQ #0x008	0x6008	Jumpto8 Error
0x12 & 0x13	STA mem[R0] = R0 LDR R22=mem[R0]	0x0000 0x0A00	mem[0x0002]=0x0002 R22 = 0x002
0x14	STA mem[R0] = R3	0x0C00	mem[0x0002]=0x000A
0x15	LDR R2 = mem[R0]	0x0A00	R2=0x000A
0x16	STP	0x7000	

Background code is designed to have instruction that change Register file's register, Jump Condition Register. It repeated for twice before it stopped to ensure enough time is given to UART receiver for receiving the data.

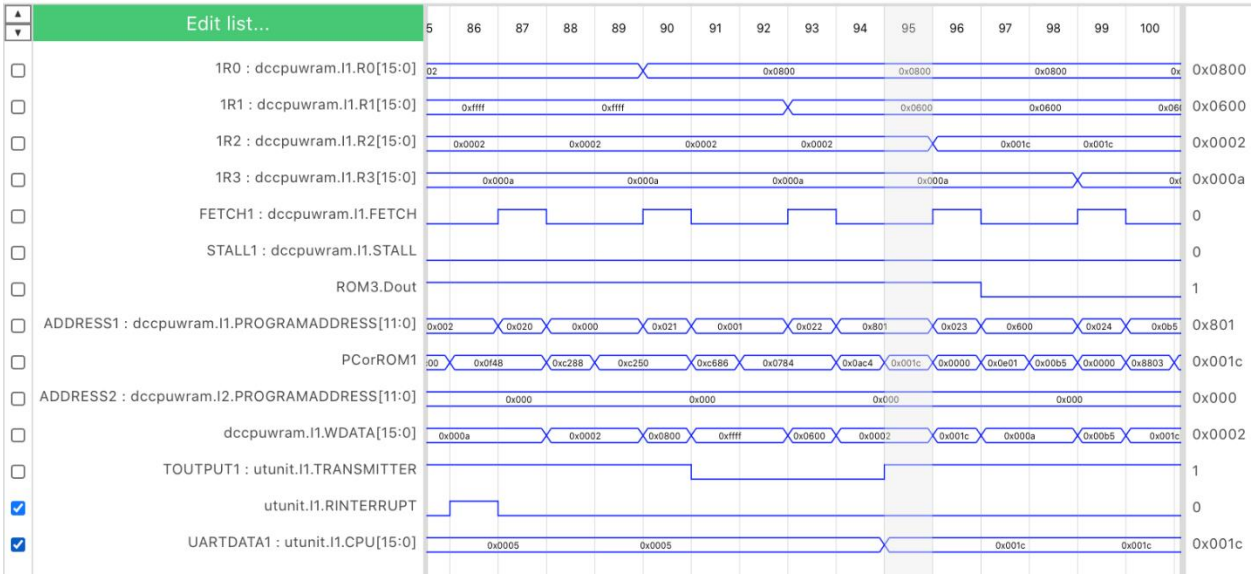
Expected Output:

UART Receiver (CPU's read request high)	1011 0101(0x00b5)	0001 1100(0x001C)
Contest saving	CPU return to its initial state after interrupt is finished	
UART Transmitter	1011 0101+ 0 ~0 1010 1101 1	1011 0101 + 0001 1100 ~0 1000 1011 1

UART Receiver:

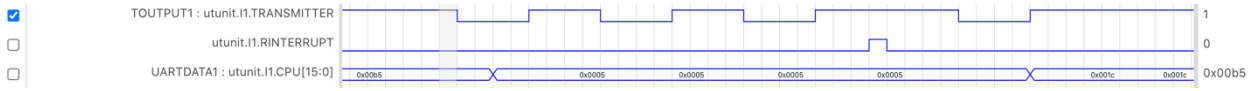


Receiver Output at 50(LDR R3 #801):0x00b5

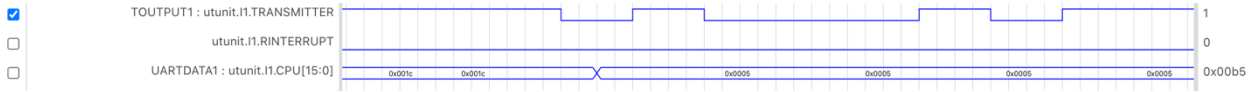


Receiver Output at 95(LDR R3 #801):0x001c

UART Transmitter:

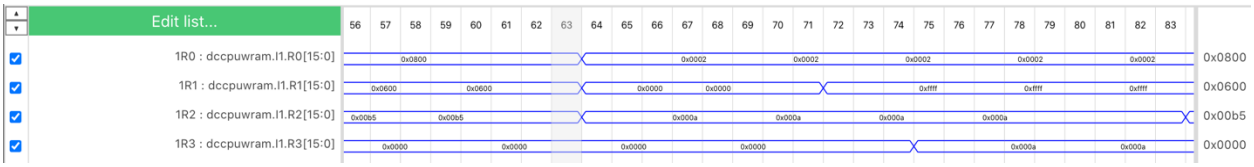


Transmitter Output: 0 1010 1101 1

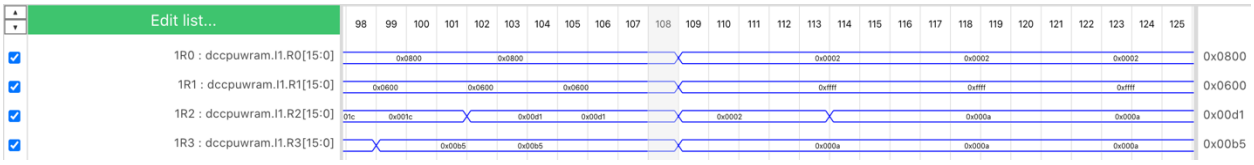


Transmitter Output: 0 1000 1011 1

Contest Saving:



The first Interrupt finished at 63



The second Interrupt finished at 108

8.2 Floating point arithmetic (16 bits and 32 bits)

Addition / Subtraction

Possible Inputs

Non-exceptions	Exceptions
Integer - AB combination (00, 01, 10, 11) but always the bigger number minus the smaller one	**NaN -** Both Exponents are equal to 15 and both of their integer parts equal to 1
Sign - AB combination (00, 01, 10, 11)	**Infinity -** Test for infinity, exponent A or B or both equal to 11111
Exponents - Equal - Not equal - One of the Exponent equal to -14 One of the exponent equal to 15	**Underflow -** Both words are 0x0

Figure 35 Test table for FP Addition / Subtraction

Multiplication / Division

Possible Inputs

Non-exceptions	Exceptions
Integer combination - **1,1** o Result from fraction = 01. ⋯ o Result from fraction = 10. ⋯ (right shift) o Result from fraction = 11. ⋯ (left shift) - **1,0** o With shift o Without shift	**Overflow -** Both exponents are large enough to form an infinity number
Negative exponent[3] - Integer combination 1,1 and 0,0	**Underflow -** Both integer = 0
Largest result - Integer combination 1,1 and 0,0 **Smallest result** - Integer combination 1,1 and 0,0	

Figure 38 Test table for FP Multiplication

Full testing table and screenshots of test can be found in appendix II.

Sin x calculation using Floating-Point Implementation

Calculating sin x with Maclaurin series up to 4th term because $\frac{1}{9!}$ exceed the lower limit of 16 bits floating-point implementation.

Method of testing:

[3] Negative exponent – refer floating-point multiplication normalization process.

8 RAM addresses are used for storing factorials and product terms x, $\frac{1}{3!}x^3, \frac{1}{5!}x^5, \frac{1}{7!}x^7$. The result is also programmed to be stored in the RAM after the calculation.

Sin x	**Decimal**	**Notes**
Expected result	0.84147098	Calculated using calculator
Obtained result	0.84082031	Convert from $0\ 01110\ 10101\ 11010_2$/ 0x3ABA

Comparison of sin x with fixed-point implementation

Assembly code for sin x with floating-point implementation

Memory Addresses	**Assembler**	**IR** **Register Shift** **Immediate constant** **LDR /STR**	**Note**
0x0	MOV R0 #0x0700	0xC287	R0 := 0x0700
0x1	MOV R1 #0x0900	0xC689	R1 := 0x0900
0x2	LDR R2 [R0] #1	0x0B44	R2 := x, EA R0 := 0x0701
0x3	LDR R3 R0	0x0E00	R3 := 1/3!
0x4	STR R3 R1	0x0CC1	mem[R1] = mem[0x0900] := 1/3!
0x5	STR R2 [R1, #3]	0x08CD	Product = 1/3! * x
0x6	STR R2 [R1, #3]	0x08CD	Product = 1/3! * x^2
0x7	STR R2 [R1, #3]	0x08CD	Product = 1/3! * x^3
0x8	LDR R3 [R1, #4]	0x0ED1	R3 := 1/3! * x^3
0x9	STR R3 [R0, #1]!	0x0DC4	mem[0x0702] := 1/3! * x^3, EA R0 := 0x702
0xA	LDR R3 [R0, #1]!	0x0FC4	R3 := 1/5!, EA R0 := 0x703
0xB	STR R3 R1	0x0CC1	mem[R1] = mem[0x0900]: = 1/5!
0xC	STR R2 [R1, #3]	0x08CD	Product = 1/5! * x
0xD	STR R2 [R1, #3]	0x08CD	Product = 1/5! * x^2
0xE	STR R2 [R1, #3]	0x08CD	Product = 1/5! * x^3
0xF	STR R2 [R1, #3]	0x08CD	Product = 1/5! * x^4
0x10	STR R2 [R1, #3]	0x08CD	Product = 1/5! * x^5
0x11	LDR R3 [R1, #4]	0x0ED1	R3 := 1/5! * x^5
0x12	STR R3 [R0, #1]!	0x0DC4	mem[0x0704] := 1/5! * x^5, EA R0 := 0x704
0x13	LDR R3 [R0, #1]!	0x0FC4	R3 := 1/6!, EA R0 := 0x705
0x14	STR R3 R1	0x0CC1	mem[R1] = mem[0x0900] := 1/7!
0x15	STR R2 [R1, #3]	0x08CD	Product = 1/7! * x
0x16	STR R2 [R1, #3]	0x08CD	Product = 1/7! * x^2
0x17	STR R2 [R1, #3]	0x08CD	Product = 1/7! * x^3
0x18	STR R2 [R1, #3]	0x08CD	Product = 1/7! * x^4
0x19	STR R2 [R1, #3]	0x08CD	Product = 1/7! * x^5
0x1A	STR R2 [R1, #3]	0x08CD	Product = 1/7! * x^6
0x1B	STR R2 [R1, #3]	0x08CD	Product = 1/7! * x^7
0x1C	LDR R3 [R1, #4]	0x0ED1	R3 := 1/7! * x^7
0x1D	STR R3 [R0, #1]!	0x0DC4	mem[0x0706] := 1/7! * x^7, EA R0 := 0x706
0x1E	MOV R0 #0x0700	0xC287	R0 := 0x0700
0x1F	STR R2 R1	0x08C1	mem[R1] = mem[0x0900] := x
0x20	LDR R2 [R0, #2]!	0x0BC8	R2 := 1/3! * x^3, EA R0 := 0x702

0x21	STR R2 [R1, #2]	0x08C9	Result = x - 1/3! * x^3
0x22	LDR R2 [R0, #2]!	0x0BC8	R2 := 1/5! * x^5, EA R0 := 0x704
0x23	STR R2 [R1, #1]	0x08C5	Result = x - 1/3! * x^3 + 1/5! * x^5
0x24	LDR R2 [R0, #2]!	0x0BC8	R2 := 1/7! * x^7, EA R0 := 0x706
0x25	STR R2 [R1, #2]	0x08C9	Result = x - 1/3! * x^3 + 1/5! * x^5 - 1/7! * x^7
0x26	LDR R3 [R1, #4]	0x0ED1	R3 := Result
0x27	STR R3 [R0, #1]!	0x0DC4	mem[0x0707] := Result, EA R0 := 0x707
0x28	STP		

Full screenshots of waveform simulation can be found in [appendix III.]

The tables below show the comparison of actual and calculated *sin x* values. A graph (figure 38) was also plotted based on the table results.

Floating point implementation

	binary16	**Decimal**	**binary16**	**Decimal**	**binary16**	**Decimal**
x	1 01101 00000 00000	-0.25	0 01111 10000 00000	1.5	0 10000 01000 00000	2.5
1/3! * x^3	1 00110 01010 10110	-0.002605	0 01110 00100 00000	0.5625	0 10000 01001 10100	2.602
1/5! * x^5	1 00000 00001 00010	-0.000002	0 01011 00000 01011	0.0632	0 01110 10100 00001	0.813
1/7! * x^7	1 00000 00000 01010	-0.0000006	0 00110 10111 10000	0.003387	0 01011 11101 11110	0.121
Ideal sin x value		-0.24740396		0.99749498		0.59847214
Calculated value		-0.2473964		0.997313		0.59

Fixed point implementation

	binary16	**Decimal**	**binary16**	**Decimal**	**binary16**	**Decimal**
x	1 01101 00000 00000	-0.25	0 01111 10000 00000	1.5	0 10000 01000 00000	2.5
1/3! * x^3	1 00110 01010 10110	-0.002605	0 01110 00100 00000	0.5625	0 10000 01001 10100	2.602
1/5! * x^5	0	0	0	0	0	0
1/7! * x^7	0	0	0	0	0	0
Ideal sin x value		-0.24740396		0.99749498		0.5984721
Calculated value		0.24740600		0.94140625		0.5

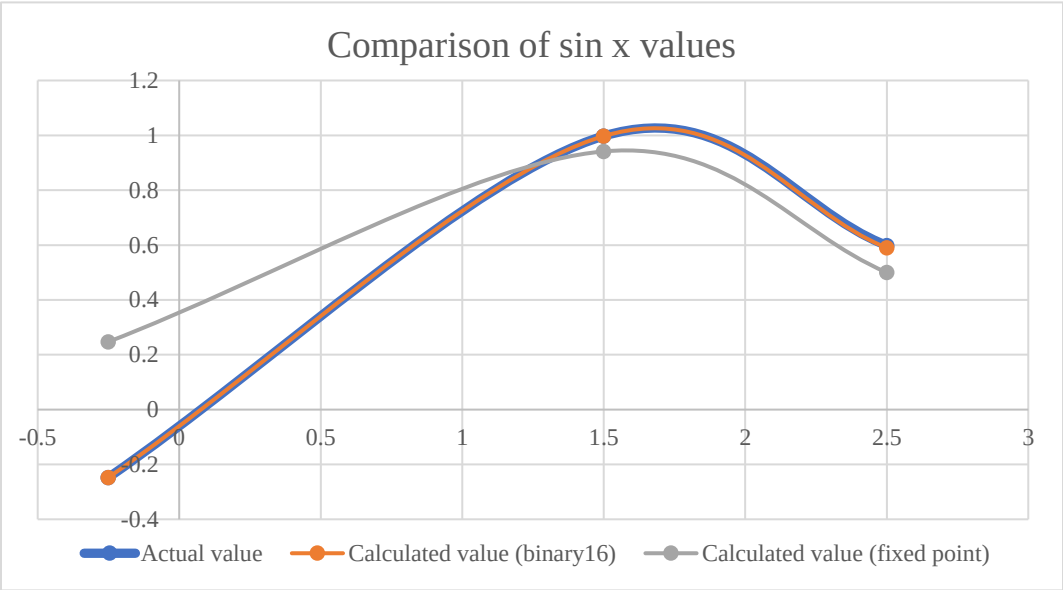


Figure 39 Graph comparing the result of sin x by using fixed point and floating-point implementation

From the graph above, it is observed that the calculated sin x using the binary16 format is much more accurate than using the fixed-point implementation.

In addition to that, further research is done to discover its speed and delay. The graph below shows the comparison of the delays for fixed-point and floating-point multipliers. [11]

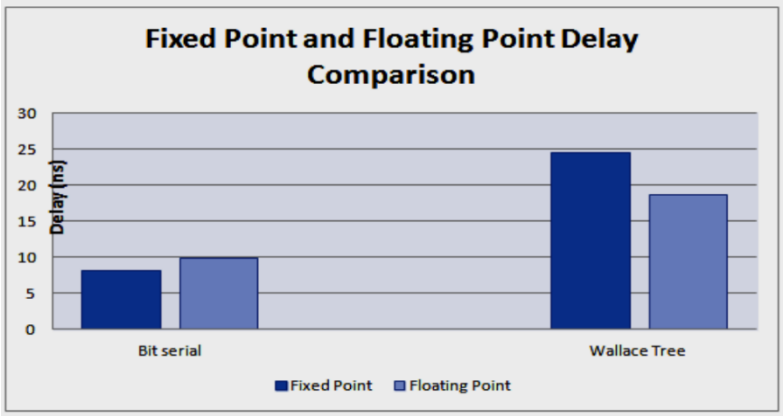


Figure 36 Fixed-point and floating-point delay comparison graph

Floating-point multipliers that use the Wallace Tree design tend to have smaller delays.

Conclusion on comparison with fixed-point implementation

The position of decimal point in fixed-point implementation depends on how large the number is. In some cases, all the bits can represent fraction, which can prevent wasting bits in floating-point implementation.

However, when the value is large, integer bits are required and when multiplication is done with fixed-point implementation, the accuracy will decrease. (Figure 40)

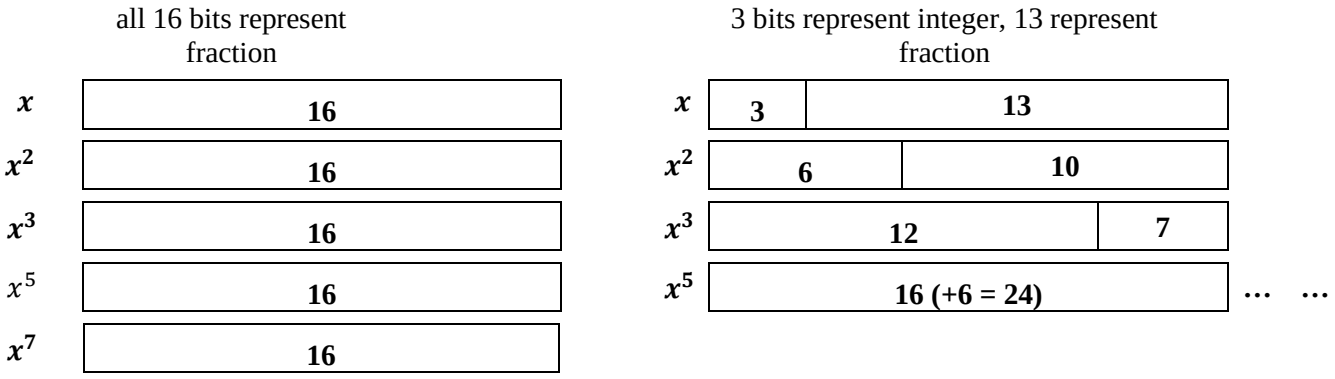


Figure 40 Fixed-point multiplication (Right side of the diagram shows how fixed-point multiplication has low accuracy)

8.3 Dual Core

Single core was tested by having a stop instruction on the other core.

Full screenshots of waveform simulation can be found in appendix-III.

Test Objective: **Compute the mean of a 2^n array.**

2n array	Sign	Fraction	Exponent	Hex	Value
0x1	0	0.00 0000 0001	00000	0x0001	2^-24
0x2	0	0.00 0000 0010	00000	0x0002	2^-23
0x3	0	0.00 0000 0100	00000	0x0004	2^-22
0x4	0	0.00 0000 1000	00000	0x0008	2^-21
0x5	0	0.00 0001 0000	00000	0x0010	2^-20
0x6	0	0.00 0010 0000	00000	0x0020	2^-19
…	…	…	…	…	…
0x017	0	1.00 0000 0000	01111	0x3800	2^0 =
….	…	…	…	…	…
0x026	0	1.00 0000 0000	11101	0x7400	2^14
0x027	0	1.00 0000 0000	11110	0x7800	2^15
…..	…..	……	……	……	…..
0x700	0	0.00 0001 1001	00000	0x019	0.025=1/40

The array we chose have 40 elements, n {-24 to 15}. This array is chosen because it covers all the range of the 16-bit floating point unit we designed. Number beyond that range could no longer be expressed with binary.

0x700 is used to store constant 1/40.

Mean = Total Sum* 1/40

Tests with 20, 30 and 40 arrays are carried out with both single core and dual core.

Comparing the speed of single core and dual core:

Array Size	Single-Core	Dual-Core	Clock Cycle Reduction	Speed Increment%
40	254	147	107	42.1%
30	209	124	85	40.7%
20	137	87	50	36.5%

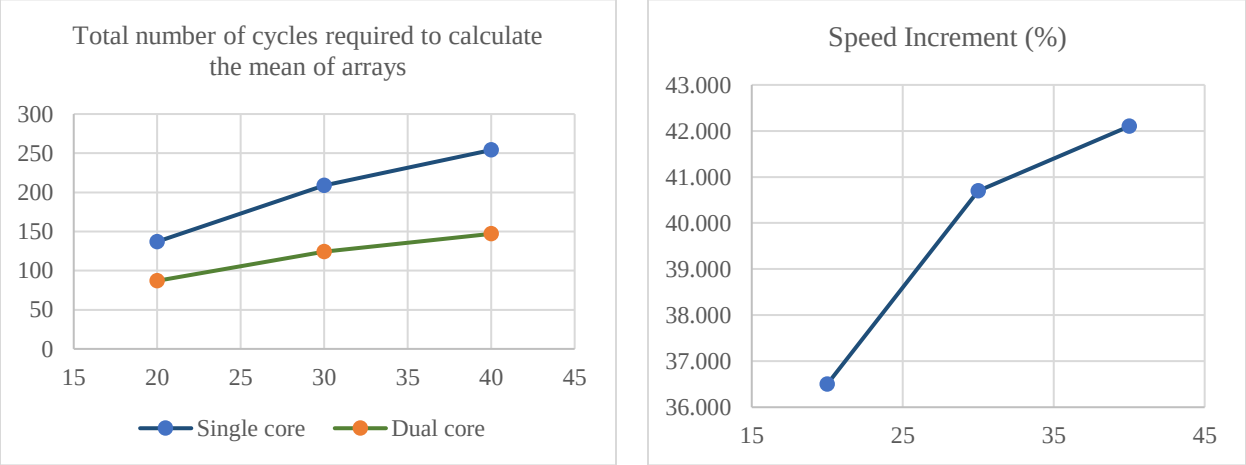


Figure 37 Tables to evaluate the speed when different size of arrays are used

The speed of the dual-core could be increased by up to twice as fast as single core's. The larger the number of elements we have in the array, the closer the speed increasement approaches 50%.

9. Evaluation and Optimization

Using the FPGA compilation tool, the results below were obtained.

Block name	**Max clock frequency**	**Number of cells**
MU0-ARM CPU	61.42 MHz	723

Block name	**Max clock frequency**	**Number of cells**
UART Receiver	173.88MHz	59
UART Transmitter	179.47MHz	38
Floating Point Adder/Subtractor	41.83MHz	185
Floating Point Multiplier (Wallace Tree design)	181.29MHz	39
Floating Point Unit	39.75MHz	292
Dual core arbitrator	626.57 MHz	42

Dual core Bus CPU	626.57 MHz	48
Dual core Final	25.98MHz	1742

Figure 38 Table of evaluation of CPU design

From the results above, the overall design has a maximum clock frequency of 25.98MHz. Since power consumption is proportional to clock frequency and logic cells, it can be concluded that the power consumed is at a medium level since clock frequency is relatively low whereas number of logic cells is high.

The clock frequency is limited by the floating-point unit which consists of the floating-point adder / subtractor because its value are below 50MHz while the others surpass 150MHz.

Execution time can be calculated with the equation

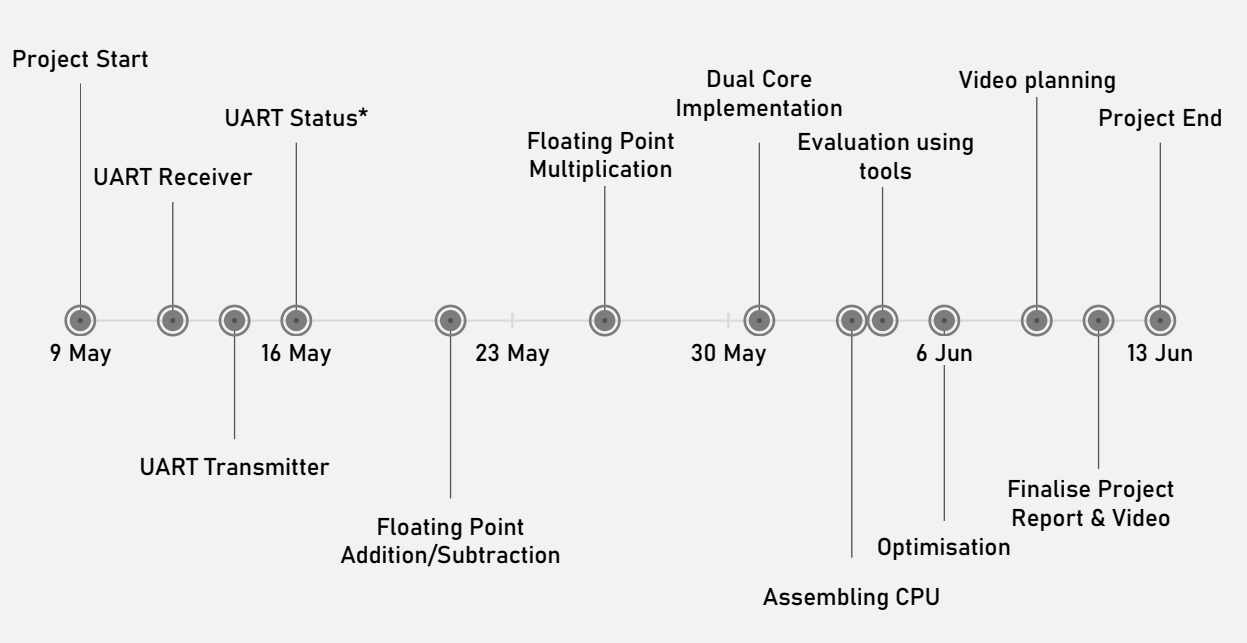
$$\frac{the\ number\ of\ clocks\ cycles\ required}{maximum\ clock\ frequencu}$$

∴ Each instruction takes 38.49ns to execute.

10. Project Management

10.2 Timeline

Planned timeline



DATE	PLANNED	DATE	ACTUAL
9-May	Project Start	9-May	Project Start

12-May	UART Receiver	12-May	UART Receiver
14-May	UART Transmitter	13-May	UART Transmitter
16-May	UART Status	14-May	UART Status
21-May	Floating Point Add/Sub	16-May	UART Interrupt
26-May	Floating Point Multiplication	23-May	Floating Point Add/Sub
31-May	Dual Core Implementation	26-May	Floating Point Multiplication
3-Jun	Assembling CPU	31-May	Dual Core Implementation
4-Jun	Evaluation using tools	5-Jun	Assembling CPU & top-level testing
6-Jun	Optimization	6-Jun	Evaluation using tools
9-Jun	Video planning	8-Jun	Report Writing
11-Jun	Finalize Project Report & Video	10-Jun	Video Planning & Recording
13-Jun	Project End	12-Jun	Finalise and Proofread Report & Video
		13-Jun	Project End

10.3 Meeting details & Consultation

The team had meetings 5 times a week during working hours. Discussions are very frequent especially when the team planned to implement the same part at the same time and often come up with different designs. Team members presented their approach on Microsoft Team and decide the best design as a team. Sometimes, tasks that are easier are delegated to different members. On the other hand, team members work together when the task is relatively complicated.

The team went for 4 consultations in total to get clarification on the project specifications and advice on implementation. Notes were taken down by one of the members.

10.4 Maintenance

Microsoft Teams channel was created for organising meetings and documenting our progress and data. Files such as testing data, research papers, backup files are posted regularly to keep track of our progress and avoid losing important documents.

11. Future Work

These extra features may be implemented in the future:

1. **Interrupt in an interrupt**

The similar priority idea can be ulitised to handle cases of a new interrupt occurred in an interrupt. Different value of priority code could be attached to addresses storing the interrupt loop's instructions. The CPU could decide whether to leave the current executing loop to the new interrupt loop by comparing their address priority code.

If Executing loop's priority code > New Interrupt loop's priority code: Stay at the current loop. Perform the new interrupt loop only after the current loop had finished.

If Executing loop's priority code < New Interrupt loop's priority code: Leave the current loop and execute the new interrupt loop immediately.

2. **Program CPU to check UART status before receiving and transmitting to detect errors**

3. **Combine 16 bits into 32 bits**

A 32-bits data could be form by combining two CPU together, which could be processed by the 32-bits floating points arithmetic block. Hence, arithmetic base on 32-bits could now be implemented into the CPU, which generally has higher accuracy and capacity.

12. Conclusion

In summary, the aim of designing a CPU to solve commonly occurring computing problems was successfully achieved. Factors including speed, power consumption, were carefully considered during the design process of the CPU.

The UART block and the floating-point arithmetic block was implemented and integrated to the main CPU by using memory-mapped approach, which allows the CPU to perform floating-point data processing and data transmission in parallel while running its main CPU instructions. This method made it easier to add new components on the CPU.

The logic between CPUs and Arbitrator is well structed and simplified, only four essential ports are used which made it easier to add cores or shared components on it in the future.

Each of the teammate was required to come up with his/her design idea from the top-level to the basic gate-level design as the project progress, this might slow down the team progress to some extends, but as a trade-off, a much thoughtful and comprehensive design could be achieved.

Overall, the project was successful, the team worked efficiently and was able to gain a deeper understanding in CPU and interpersonal skills through this project.

Reference

[1] ‘Serial communication’, *Wikipedia*. Jun. 04, 2021. Accessed: Jun. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Serial_communication&oldid=1026894179

[2] ‘Universal asynchronous receiver-transmitter’, *Wikipedia*. May 08, 2021. Accessed: Jun. 13, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Universal_asynchronous_receiver-transmitter&oldid=1022070276

[3] ‘Half-precision floating-point format’, *Wikipedia*. Jan. 21, 2021. Accessed: May 29, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Half-precision_floating-point_format&oldid=1001831829

[4] ‘Single-precision floating-point format’, *Wikipedia*. May 24, 2021. Accessed: Jun. 11, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Single-precision_floating-point_format&oldid=1024960263

[5] R. Dhobale and S. Chaturvedi, ‘Implementation of 32 Bit Binary Floating Point Adder Using IEEE 754 Single Precision Format’, p. 4.

[6] V. Kaushik and H. Saini, ‘A Review on Comparative Performance Analysis of Different Digital Multipliers’, p. 16.

[7] ‘Carry-save multiplier algorithm’, *Mathematics Stack Exchange*. https://math.stackexchange.com/questions/1187959/carry-save-multiplier-algorithm (accessed Jun. 13, 2021).

[8] Gnanasekaran, ‘A Fast Serial-Parallel Binary Multiplier’, *IEEE Trans. Comput.*, vol. C–34, no. 8, pp. 741–744, Aug. 1985, doi: 10.1109/TC.1985.1676620.

[9] ‘Dual-Core Definition’. https://techterms.com/definition/dualcore (accessed Jun. 13, 2021).

[10] ‘BUS Arbitration in Computer Organization - GeeksforGeeks’. https://www.geeksforgeeks.org/bus-arbitration-in-computer-organization/ (accessed Jun. 13, 2021).

[11] A. Ahmed and A. Shahzad, ‘Fixed Point and Floating-Point High-Speed Hardware Multipliers- A comparison of Bit Serial and Wallace Tree Multipliers Using Booth Recoding’, *Int. J. Eng.*, vol. 15, no. 01, p. 6, 2015.

Appendix-I

Specifications of the functional requirements (Part of PDS)

1. Serial communication
   - UART (Universal Asynchronous Receiver Transmitter) Block
     1. UART protocol shall have 10 bits in total starting with logic 0 and ending with logic 1. There will be a bit (logic 1) between receive and transmit signals.
     2. The bit rate: 1 bit per 4 clock cycles.
     3. Status of UART block will determine if[4]:
        i. A byte has been received since the receive register was last read by the CPU.
        ii. A transmission is in progress.
        iii. There was a receive overflow (a byte was received before the previous byte was read by the CPU)
        iv. There was a transmit overflow (a byte was written by the CPU before the previous byte had finished transmission)
     4. UART block must be able to receive bytes and transmit processed[5] bytes to the CPU.
     5. *Advanced*: Interrupt function shall be integrated to prompt the CPU after a byte receive or transmit is complete.

2. Floating point arithmetic
   - Floating point representation
     1. Including arithmetic operations: add, subtract and multiply.
     2. Number format: 16-bit IEEE 754 half-precision
        i. (Advanced) Use 32-bit IEEE 754 single precision.
     3. Rounding behavior and exceptions are undefined.

   Test[6]: Compare the accuracy of the floating-point representation with fixed-point representation.

3. Dual core CPU
   - Bus arbitration logic
     1. Require stall when both cores try to access data memory.
     2. Include extra signals: read and write requests, read valid and write complete.
     3. Mean of an array of 2^n numbers shall be calculated with the dual core CPU.

   Test: Find the acceleration of the calculation when compared to a single core.

[4] Source from EEE Project Initial Specification
[5] Accumulated sum of all bytes
[6] Test to prove that the features can improve the performance of the CPU

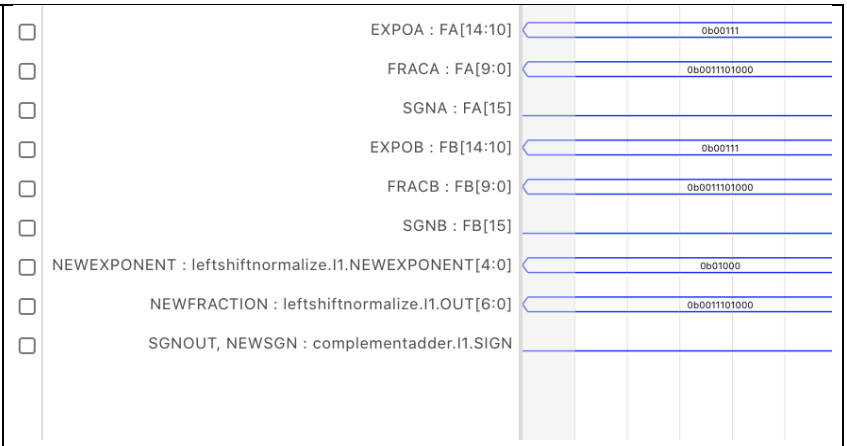
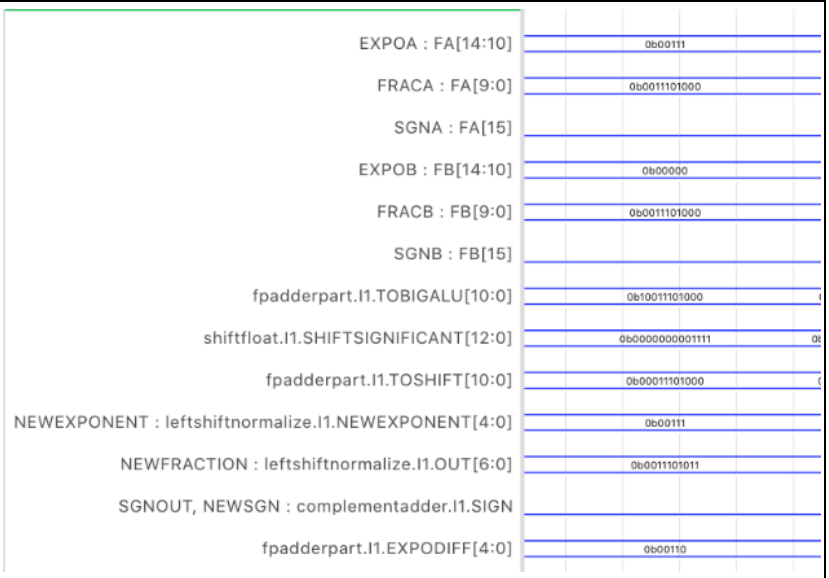
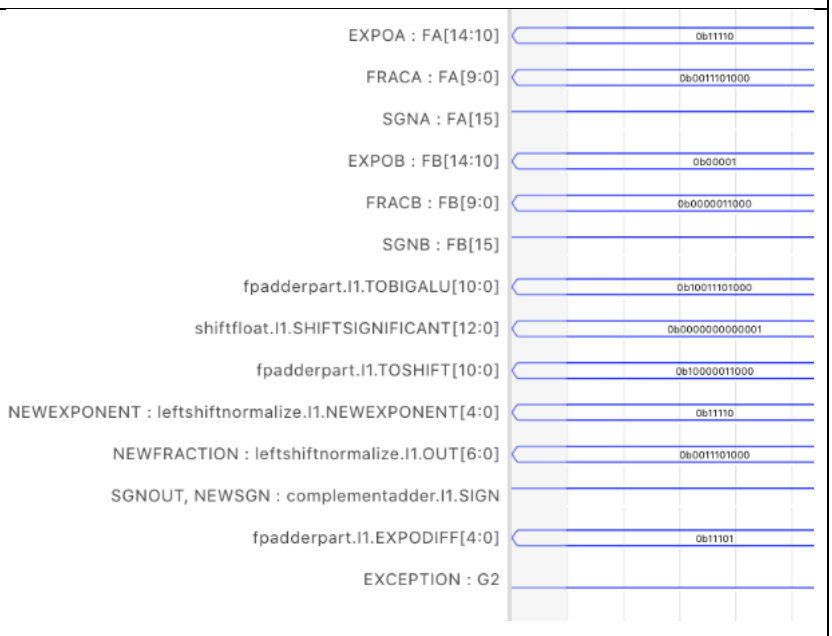
Appendix-II

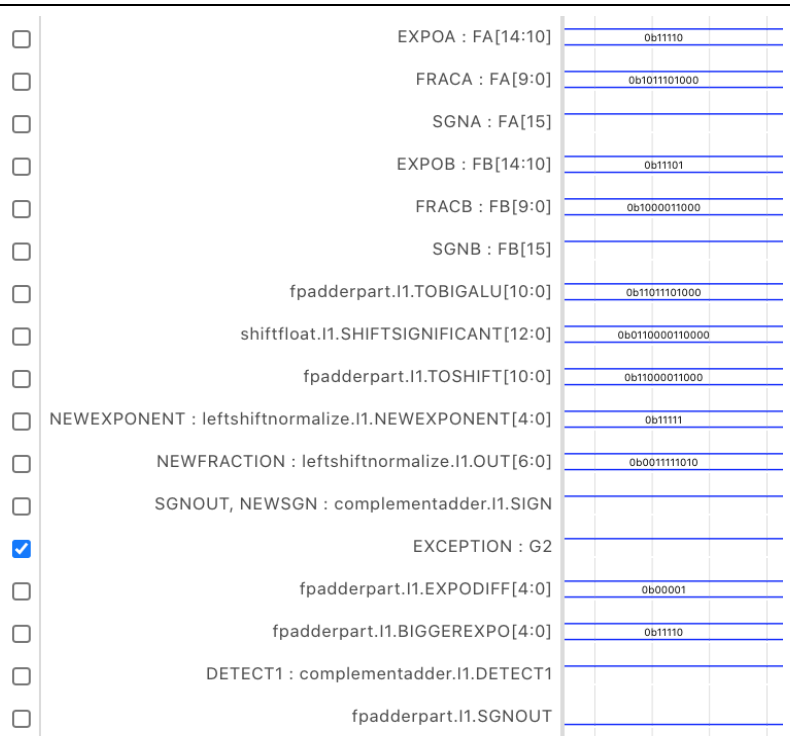
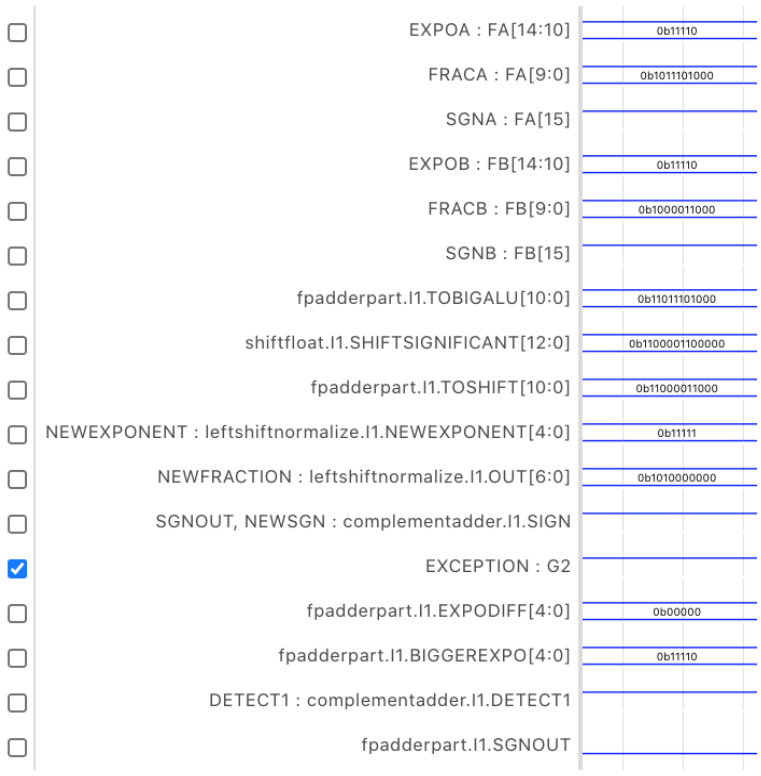
- Exhaustive test tables with simulation screenshots

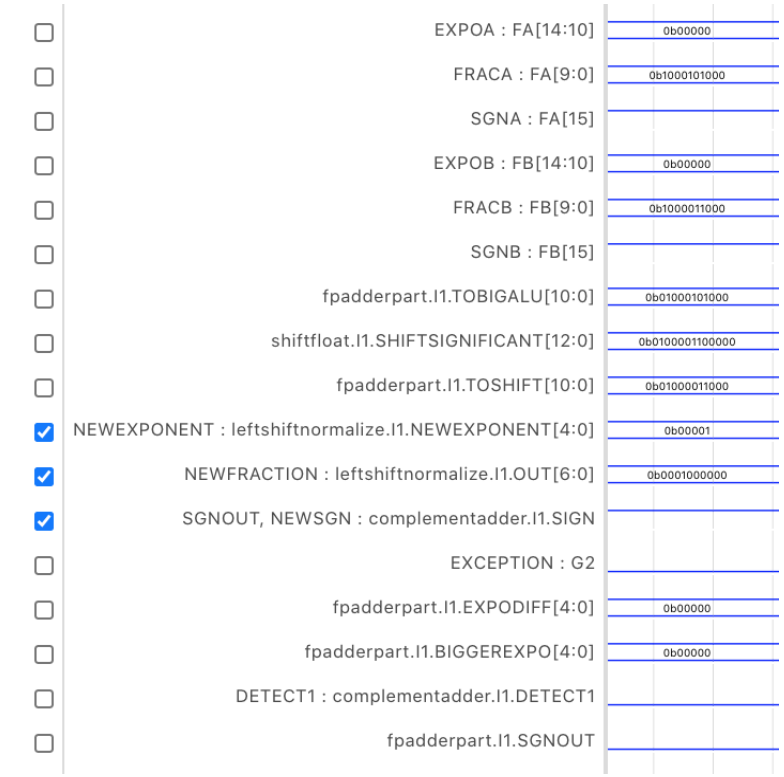
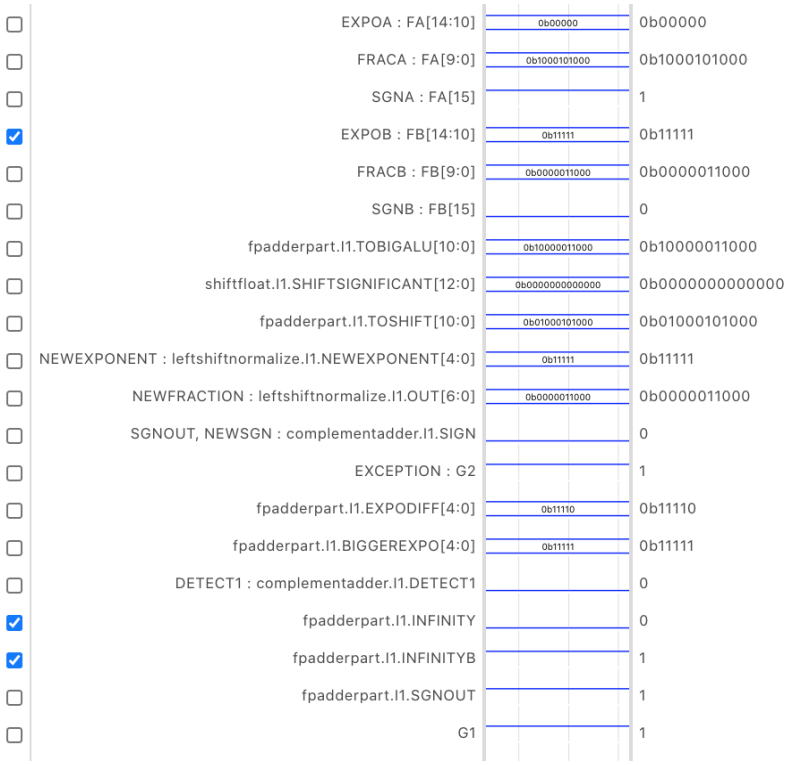
Binary16 Addition Tests

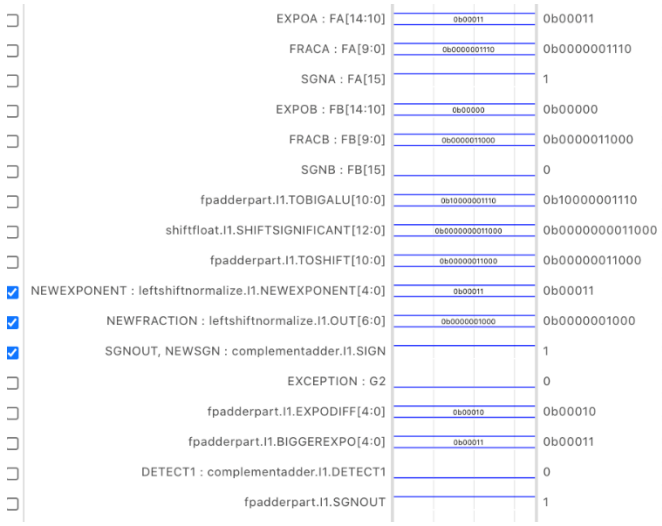
Exhaustive Test tables from figure 36

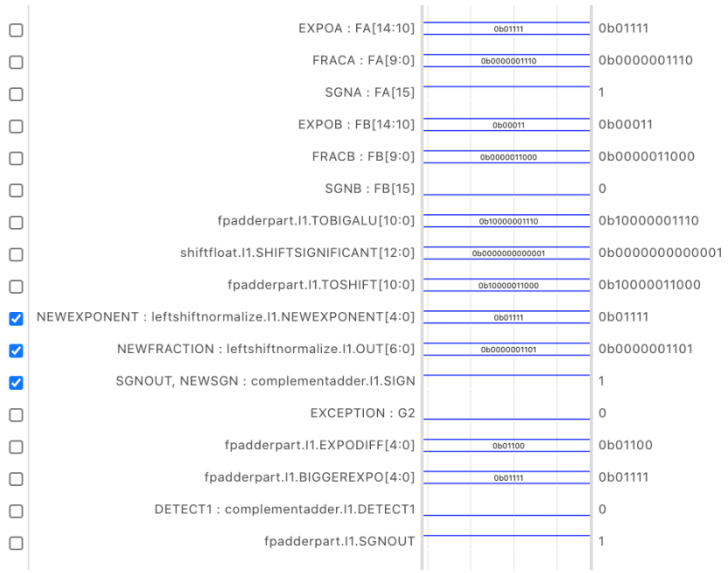
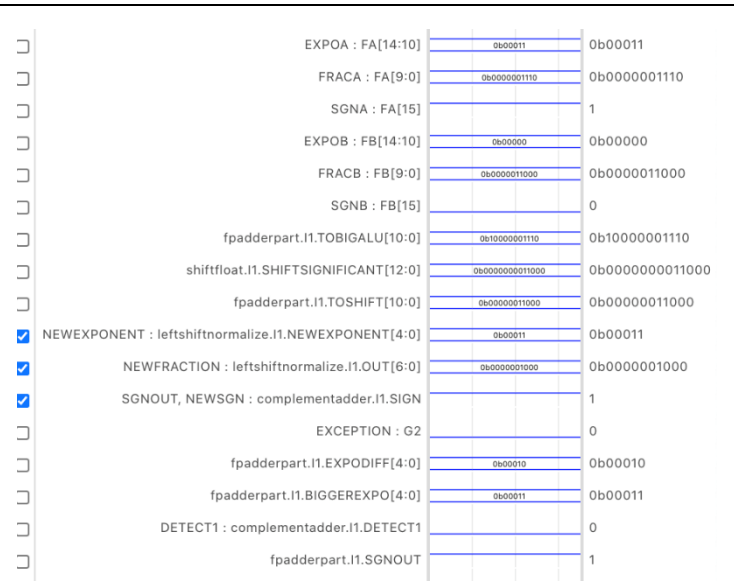
		sign	integer	significand	exponent	overflow
Addition 1. Exponents are equal	Input1	0	1	0011101000	00111	0
	Input2	0	1	0011101000	00111	0
	output	0	1	0011101000	01000	0
Addition 2. One of the exponent equals to -14	Input1	0	0	0011101000	00000	0
	Input2	0	1	0011101000	00111	0
	output	0	1	0011101011	00111	0
Addition 3. One of the exponent equal to 15, sum is normal	Input1	1	1	0011101000	11110	0
	Input2	1	1	0011101000	00001	0
	output	1	1	0011101000	11110	0
Addition 4. One of the exponent equal to 15, sum is overflow	Input1	1	1	1011101000	11110	0
	Input2	1	1	1000011000	11101	0
	output	1	1	0011111010(doesn't matter)	11111	1
Addition 4. Both exponents equal to 15 and both integer parts equal to1, sum is overflow	Input1	1	1	1011101000	11110	0
	Input2	1	1	1000011000	11110	0
	output	1	NaN	1010000000(doesn't matter)	11111	1
		sign	**integer**	**significand**	**exponent**	**overflow**
Addition 5. Both exponents equal to -14 integer parts equal to 0 (test for smallest)	Input1	1	0	10 0010 1000	00000	0
	Input2	1	0	10 0001 1000	00000	0
	output	1	1	00 0100 0000	00001	1
Addition 6. Test for infinity	Input1	1	1	10 0010 1000	00000	0
	Input2	0	1	00 0001 1000	11111	0
	output	0	NaN	00 0001 1000	11111	1
Addition 7. Test for integer #1&0(1)	Input1	1	1	00 0000 1110	00011	0
	Input2	0	0	00 0001 1000	00000	0
	output	1	1	00 0000 1000	00011	0
Addition 7. Test for integer #1&1	Input1	1	1	00 0000 1110	01111	0
	Input2	0	1	00 0001 1000	00011	0
	output	1	1	00 0000 1110	01111	0
Addition 7. Test for integer #0&1	Input1	1	0	00 0000 1110	00000	0
	Input2	0	1	00 0001 1000	00011	0
	output	0	1	00 0001 0100	00011	0
Addition 7. Test for integer #1&0(2)	Input1	1	1	00 0000 1110	00011	0
	Input2	0	0	00 0001 1000	00000	0
	output	1	1	00 0000 1000	00011	0

1	Exponents are equal	
2	One of the Exponent equal to -14	
3	One of the exponent equal to 15, sum is normal	

4	Overflow test: One of the exponent equal to 15	
4	Overflow test: Both Exponents are equal to 15 and both integer parts equal to 1	
5	Test for the smallest number, both exponents equal to -14 integer parts equal to 0	

		
6	Test for infinity	
7	Test integer (00,01,10,11)	#0&0

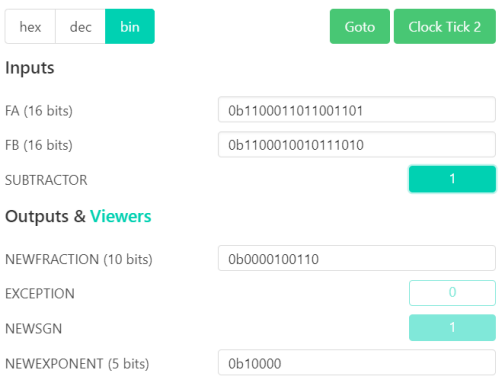
		 EXPOA : FA[14:10] 0b00000 0b00000 FRACA : FA[9:0] 0b0000001110 0b0000001110 SGNA : FA[15] 1 EXPOB : FB[14:10] 0b00011 0b00011 FRACB : FB[9:0] 0b0000011000 0b0000011000 SGNB : FB[15] 0 fpadderpart.I1.TOBIGALU[10:0] 0b10000011000 0b10000011000 shiftfloat.I1.SHIFTSIGNIFICANT[12:0] 0b0000000001110 0b0000000001110 fpadderpart.I1.TOSHIFT[10:0] 0b00000001110 0b00000001110 NEWEXPONENT : leftshiftnormalize.I1.NEWEXPONENT[4:0] 0b00011 0b00011 NEWFRACTION : leftshiftnormalize.I1.OUT[6:0] 0b0000010100 0b0000010100 SGNOUT, NEWSGN : complementadder.I1.SIGN 0 EXCEPTION : G2 0 fpadderpart.I1.EXPODIFF[4:0] 0b00010 0b00010 fpadderpart.I1.BIGGEREXPO[4:0] 0b00011 0b00011 DETECT1 : complementadder.I1.DETECT1 0 fpadderpart.I1.SGNOUT 1
		#0&1  EXPOA : FA[14:10] 0b00000 0b00000 FRACA : FA[9:0] 0b0000001110 0b0000001110 SGNA : FA[15] 1 EXPOB : FB[14:10] 0b00000 0b00000 FRACB : FB[9:0] 0b0000011000 0b0000011000 SGNB : FB[15] 0 fpadderpart.I1.TOBIGALU[10:0] 0b00000011000 0b00000011000 shiftfloat.I1.SHIFTSIGNIFICANT[12:0] 0b0000000111000 0b0000000111000 fpadderpart.I1.TOSHIFT[10:0] 0b00000001110 0b00000001110 NEWEXPONENT : leftshiftnormalize.I1.NEWEXPONENT[4:0] 0b00000 0b00000 NEWFRACTION : leftshiftnormalize.I1.OUT[6:0] 0b0000001010 0b0000001010 SGNOUT, NEWSGN : complementadder.I1.SIGN 0 EXCEPTION : G2 0 fpadderpart.I1.EXPODIFF[4:0] 0b00000 0b00000 fpadderpart.I1.BIGGEREXPO[4:0] 0b00000 0b00000 DETECT1 : complementadder.I1.DETECT1 0 fpadderpart.I1.SGNOUT 1
		#1&0  EXPOA : FA[14:10] 0b00011 0b00011 FRACA : FA[9:0] 0b0000001110 0b0000001110 SGNA : FA[15] 1 EXPOB : FB[14:10] 0b00000 0b00000 FRACB : FB[9:0] 0b0000011000 0b0000011000 SGNB : FB[15] 0 fpadderpart.I1.TOBIGALU[10:0] 0b10000001110 0b10000001110 shiftfloat.I1.SHIFTSIGNIFICANT[12:0] 0b0000000011000 0b0000000011000 fpadderpart.I1.TOSHIFT[10:0] 0b00000011000 0b00000011000 NEWEXPONENT : leftshiftnormalize.I1.NEWEXPONENT[4:0] 0b00011 0b00011 NEWFRACTION : leftshiftnormalize.I1.OUT[6:0] 0b0000001000 0b0000001000 SGNOUT, NEWSGN : complementadder.I1.SIGN 1 EXCEPTION : G2 0 fpadderpart.I1.EXPODIFF[4:0] 0b00010 0b00010 fpadderpart.I1.BIGGEREXPO[4:0] 0b00011 0b00011 DETECT1 : complementadder.I1.DETECT1 0 fpadderpart.I1.SGNOUT 1

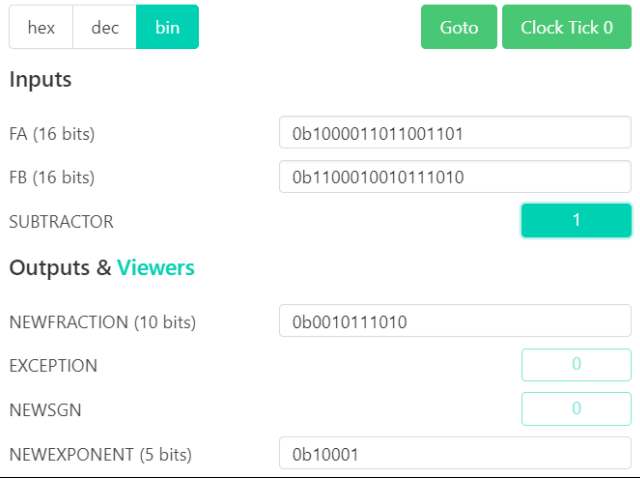
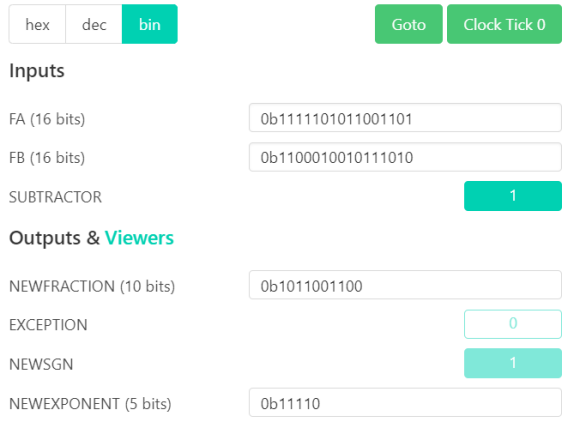
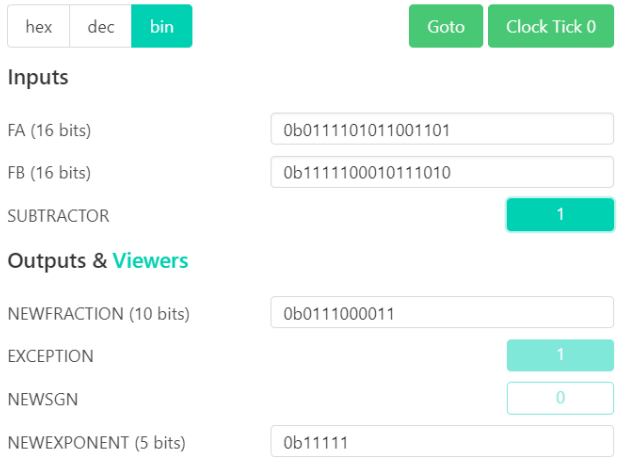
		#1&1 
8	Different sign	

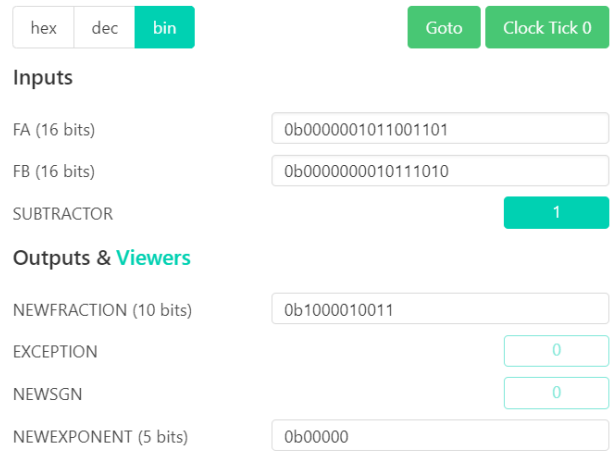
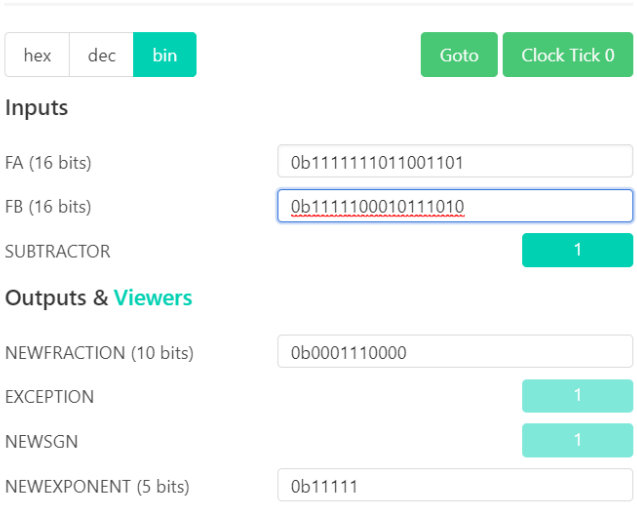
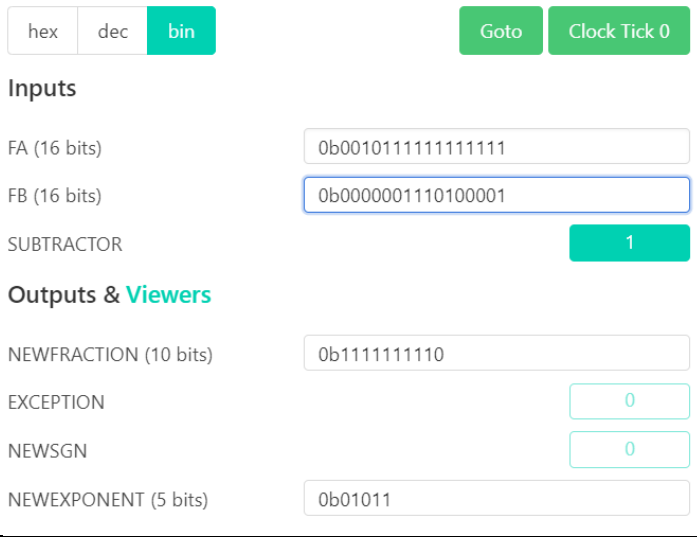
Binary16 Subtraction Tests

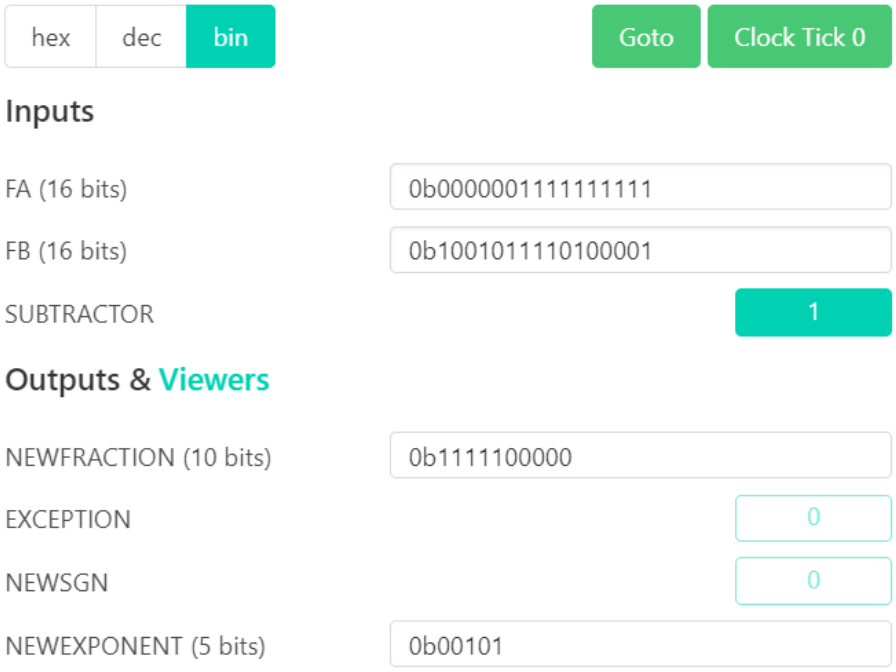
Exhaustive Test tables from figure 36

		sign	integer	significand	exponent	overflow
Subtraction 1. Exponents are equal	Input1	1	1	10 1100 1101	10001	0
	Input2	1	1	00 1011 1010	10001	0
	output	1	1	00 0010 0110	10000	0
Subtraction 2. One of the exponent equals to -14	Input1	1	1	10 1100 1101	00001	0
	Input2	1	1	00 1011 1010	10001	0
	output	0	1	00 1011 1010	10001	0
Subtraction 3. One of the exponent equal to 15, sum is normal	Input1	1	1	10 1100 1101	11110	0
	Input2	1	1	00 1011 1010	10001	0
	output	1	1	10 1100 1101	11110	0
Subtraction 4.Overflow test: Both of the exponent equal to 15 and integer parts are 1	Input1	0	1	10 1100 1101	11110	0
	Input2	1	1	00 1011 1010	11110	0
	output	0	NaN	00 0000 0000	11111	1
Subtraction 5. Both of the exponents equal to -14 integer parts equal to 0 (test for smallest)	Input1	0	0	10 1100 1101	00000	0
	Input2	0	0	00 1011 1010	00000	0
	output	0	0	10 0001 0011	00000	0
Subtraction 6.Test for infinity		**sign**	**integer**	**significand**	**exponent**	**overflow**
	Input1	1	0	11 1111 1111	11111	0
	Input2	1	0	11 1111 1111	11111	0
	output	1	0	00 0000 0000	11111	1
Subtraction 7. One of the exponent equals to -14 #1&0	Input1	0	1	11 1111 1111	01011	0
	Input2	0	0	11 1010 0001	00000	0
	output	0	1	11 1111 1110	01011	0
Subtraction 7. One of the exponent equals to -14 #0&1	Input1	0	0	11 1111 1111	00000	0
	Input2	1	1	11 1010 0001	00101	0
	output	0	1	11 1110 0001	00101	0

1	Exponents are equal	

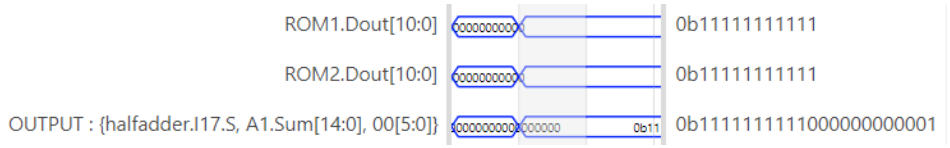
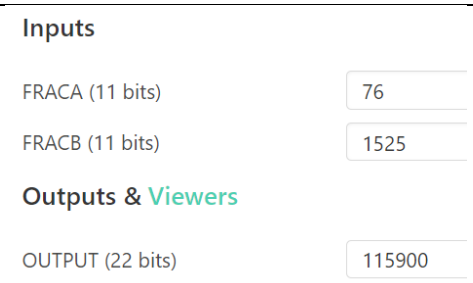
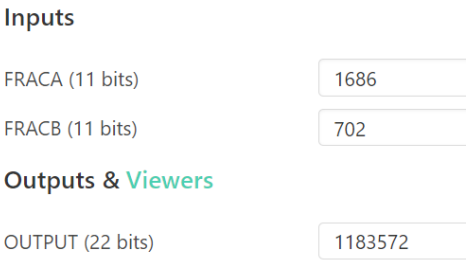
2	One of the Exponent equal to -14	
3	One of the exponent equal to 15	
4	Overflow test: Both Exponents are equal to 15 and both of their integer parts equal to 1	

5	Test for smallest number, both of the exponents equal to -14, with their integer parts equal to 0	
6	Test for infinity, exponent A or B or both equal to 11111	
7	Test integer, AB combination (00, 01, 10, 11) but always the bigger number minus the smaller one Integer 00 and 11 are tested above both are correct	**10** 

		01  hex dec bin Goto Clock Tick 0 Inputs FA (16 bits) 0b0000001111111111 FB (16 bits) 0b1001011110100001 SUBTRACTOR 1 Outputs & Viewers NEWFRACTION (10 bits) 0b1111100000 EXCEPTION 0 NEWSGN 0 NEWEXPONENT (5 bits) 0b00101

Binary16 Multiplication Tests

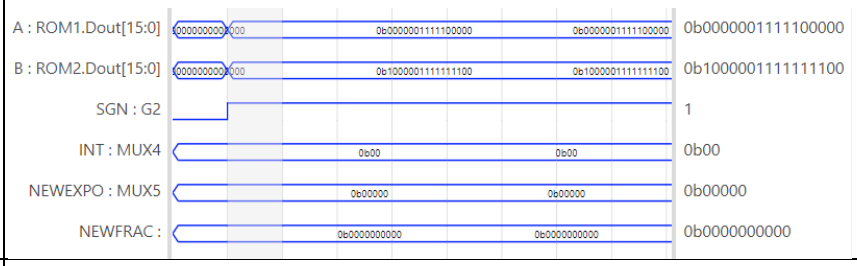
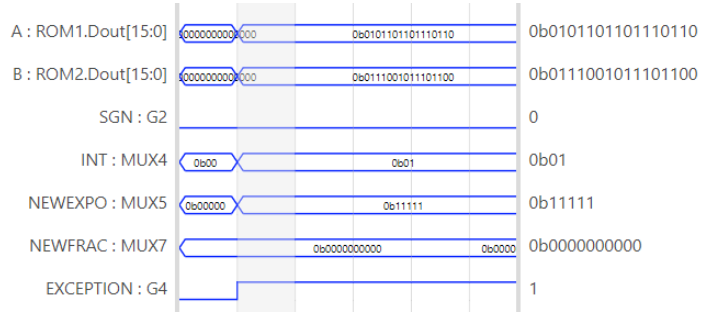
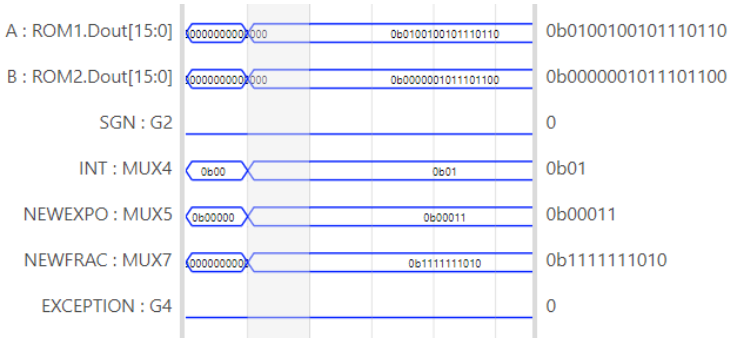
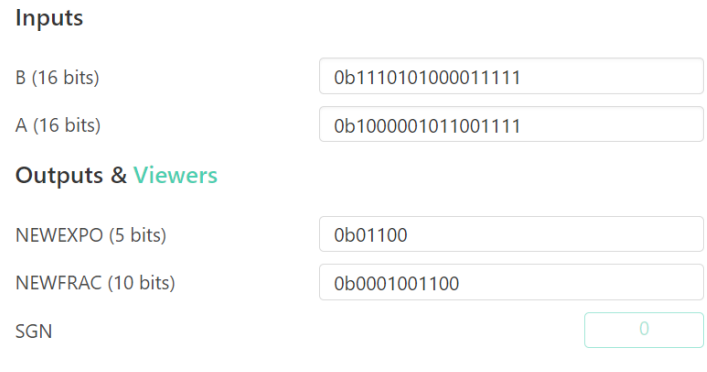
Block level testing (floating point multiplier)

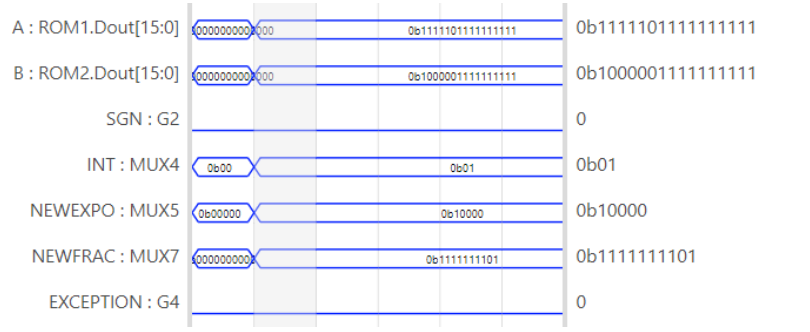
Expected results	
0b11111111111* 0b11111111111	
76*1525 = 115900	
1686*702 = 1183572	

Exhaustive Test tables from figure 37

		sign	**integer**	**significand**	**exponent**	**overflow**
1. Underflow	Input1	0	0	11 1111 1111	00000	0
	Input2	1	0	11 1111 1111	00000	0
	output	1	0	00 0000 0000	00000	0
2. Overflow	Input1	0	1	11 0111 0110	10110	0
	Input2	0	1	10 1110 1100	11100	0
	output	0	NaN	00 0000 0000	11111	1
3.1 Integer combination: 1 and 0 with shift	Input1	0	1	01 0111 0110	10010	0
	Input2	0	0	10 1110 1100	00000	0
	output	0	1	11 1111 1010	00011	0
3.2 Integer combination: 1 and 0 with shift	Input1	1	1	10 0001 1111	11010	0
	Input2	1	0	10 1100 1111	00000	0
	output	0	1	00 0100 1100	01100	0
4. Negative exponent	Input1	1	1	11 0011 0111	01010	0
	Input2	1	0	10 1100 1111	00000	0
	output	0	0	00 0010 0101	00000	0
5.smallest input without overflow	Input1	1	0	11 1111 1111	00000	0
	Input2	1	1	11 1111 1111	00100	0
	output	0	0	00 0000 0001	00000	0
	Input1	1	1	11 1111 1111	11110	0

6. Largest input without overflow	Input2	1	0	11 1111 1111	00000	0
	output	0	1	11 11111101	10000	0

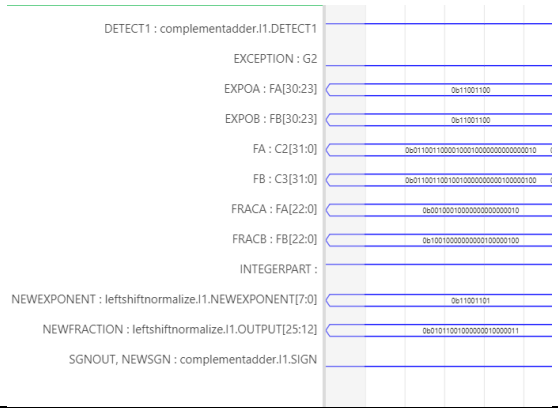
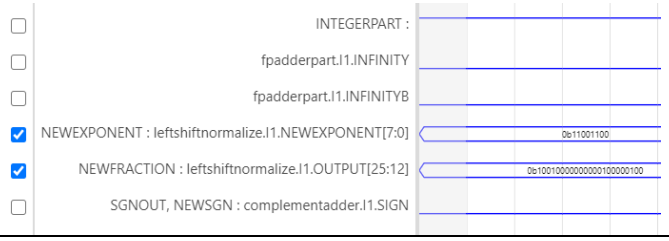
1	Underflow	
2	Overflow	
3.1	Integer combination: 1 and 0 with shift	
3.2	Integer combination: 1 and 0 without shift	

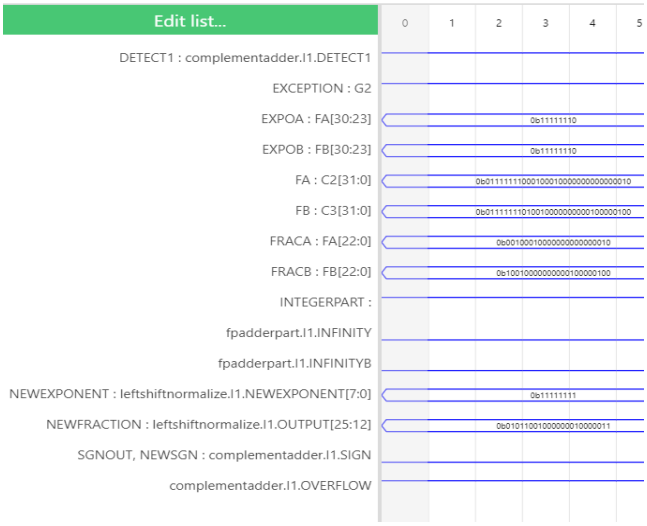
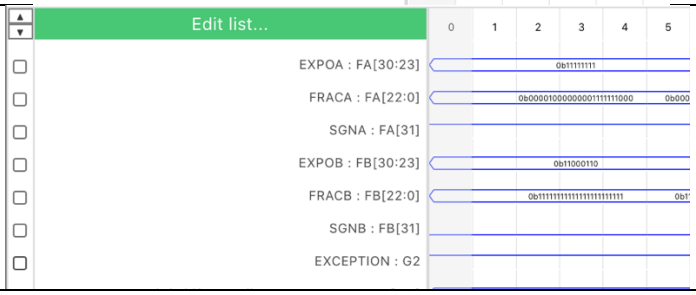
4	Negative exponent	
5	Smallest input without overflow	
6	Largest input without overflow	

Binary32 Addition tests

Exhaustive Test tables from figure 36

		sign	integer	significand	exponent	overflow
Addition 1. Exponents are equal	Input1	0	1	001 0001 0000 0000 0000 0010	1100 1100	0
	Input2	0	1	100 1000 0000 0000 1000 0100	1100 1100	0
	output	0	1	010 1100 1000 0000 1000 0011	1100 1101	0
Addition 2. One of the Exponent equals to -126	Input1	0	1	001 0001 0000 0000 0000 0010	0000 0001	0
	Input2	0	1	100 1000 0000 0000 1000 0100	1100 1100	0
	output	0	1	100 1000 0000 0000 1000 0100	1100 1100	0
Addition 3. One of the exponent equals to 127	Input1	1	1	001 0001 0000 0000 0000 0010	1111 1110	0
	Input2	0	1	100 1000 0000 0000 1000 0100	1100 1100	0
	output	1	1	001 0001 0000 0000 0000 0010	1111 1110	0
Addition 4. Overflow test: Both of the exponent equal to 127 and integer parts are 1	Input1	0	1	001 0001 0000 0000 0000 0010	1111 1110	0
	Input2	0	1	100 1000 0000 0000 1000 0100	1111 1110	0
	output	0	NaN	000 0000 0000 0000 0000 0000	1111 1111	1
Addition 5. Both exponents equal to -126 integer parts equal to 0 (test for smallest)	Input1	0	0	001 0001 0000 0000 0000 0010	0000 0000	0
	Input2	0	0	100 1000 0000 0000 1000 0100	0000 0000	0
	output	0	0	101 1100 1000 0000 1000 0110	0000 0000	0
Addition 6. Test for infinity	Input1	1	NaN	111 1111 1111 1111 1111 1111	1111 1111	0
	Input2	1	NaN	111 1111 1111 1111 1111 1111	1111 1111	0
	output	1	NaN	111 1111 1111 1111 1111 1111	1111 1111	1
Addition 7. Test for integer combination #0&1	Input1	1	0	111 0111 1110 0111 0001 1100	0000 0000	0
	Input2	1	1	001 1100 1111 1111 0000 1101	1100 1100	0
	output	1	1	001 1100 1111 1111 0000 1101	1100 1100	0
Addition 7. Test for integer combination #1&0	Input1	1	1	111 0111 1110 0111 0001 1100	1100 0000	0
	Input2	1	0	001 1100 1111 1111 0000 1101	0000 0000	0
	output	1	1	111 0111 1110 0111 0001 1100	1100 0000	0

1	Exponents are equal	
2	One of the Exponent equal to -126	
3	One of the exponent equal to 127	

4	Overflow test: Both Exponents are equal to 127 and both of their integer parts equal to 1	
5	Test for smallest number, both of the exponents equal to -126, with their integer parts equal to 0	
6	Test for infinity, exponent A or B or both equal to 11111111	

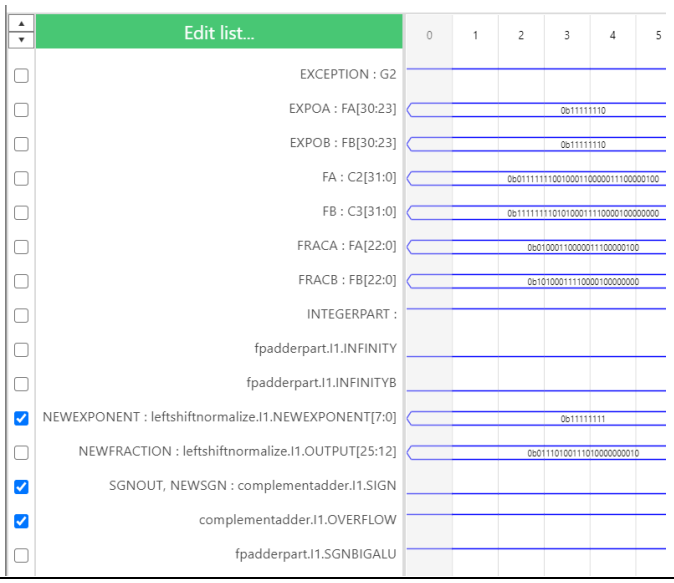
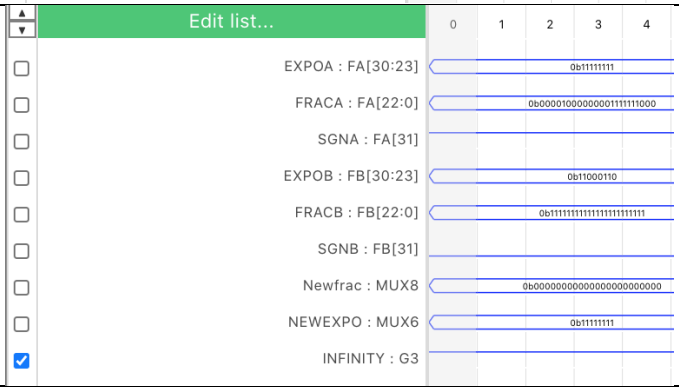
7	Test integer, AB combination (00, 01, 10, 11) (00,11 are tested in the above tests)	01 condition 
		10 condition 

Binary32 Subtraction tests

Exhaustive Test tables from figure 36

		sign	integer	significand	exponent	overflow
Subtraction 1. Exponents are equal	Input1	0	1	000 0011 0000 0111 0000 0100	1100 1100	0
	Input2	0	1	100 0001 1110 0001 0000 0000	1100 1100	0
	output	1	1	111 1011 0110 0111 1111 0000	1100 1010	0
Subtraction 2. One of the exponent equals to -126	Input1	0	1	110 0011 0000 0111 0000 0100	0000 0001	0
	Input2	0	1	100 0001 1110 0001 0000 0000	1100 1100	0
	output	1	1	100 0001 1110 0001 0000 0000	1100 1100	0
Subtraction 3. One of the exponent equals to 127	Input1	0	1	010 0011 0000 0111 0000 0100	0000 0001	0
	Input2	1	1	101 0001 1110 0001 0000 0000	1111 1100	0
	output	0	1	101 0001 1110 0001 0000 0000	1111 1110	0
Subtraction 4. Overflow test: Both exponents equal to 127 and integer parts are 1	Input1	0	1	010 0011 0000 0111 0000 0100	1111 1110	0
	Input2	1	1	101 0001 1110 0001 0000 0000	1111 1110	0
	output	0	NaN	000 0000 0000 0000 0000 0000	1111 1111	1
Subtraction 5. Both exponents equal to -126 integer parts equal to 0 (test for smallest)	Input1	0	0	010 0011 0000 0111 0000 0100	0000 0000	0
	Input2	0	0	101 0001 1110 0001 0000 0000	0000 0000	0
	output	1	0	010 1110 1101 1001 1111 1100	0000 0000	0
Subtraction 6. Test for infinity	Input1	1	NaN	111 1111 1111 1111 1111 1111	1111 1111	0
	Input2	1	NaN	111 1111 1111 1111 1111 1111	1111 1111	0
	output	1	NaN	111 1111 1111 1111 1111 1111	1111 1111	1
Subtraction 7. Test for integer combination #0&1	Input1	0	0	010 0011 0000 0111 0000 0100	0000 0000	0
	Input2	0	1	101 0001 1110 0001 0000 0000	1100 0000	0
	output	1	1	101 0001 1110 0001 0000 0000	1100 0000	0
Subtraction 7. Test for integer combination #1&0	Input1	0	1	010 0011 0000 0111 0000 0100	1100 1000	0
	Input2	0	0	101 0001 1110 0001 0000 0000	0000 0000	0
	output	0	1	010 0011 0000 0111 0000 0100	1100 1000	0

1	Exponents are equal	
2	One of the Exponent equal to -126	
3	One of the exponent equal to 127	

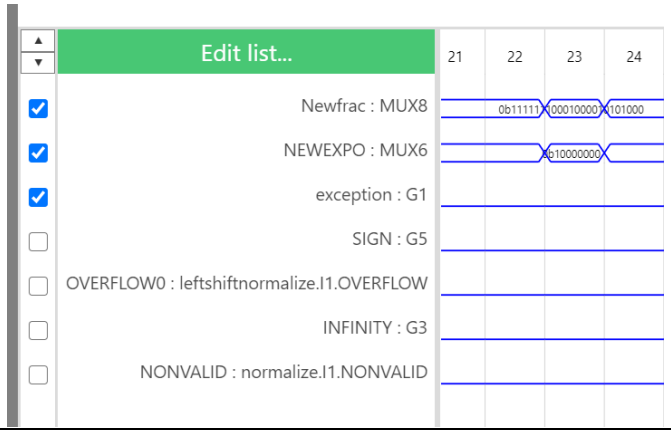
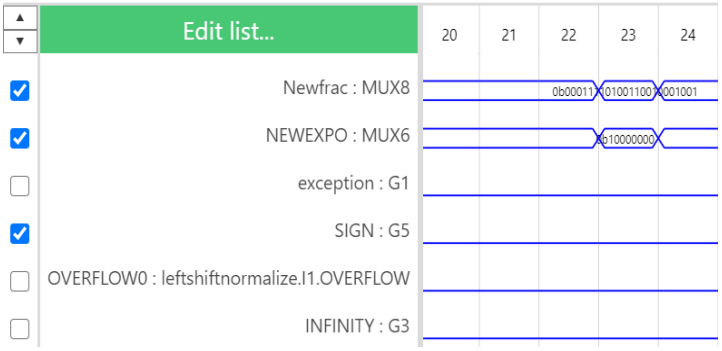
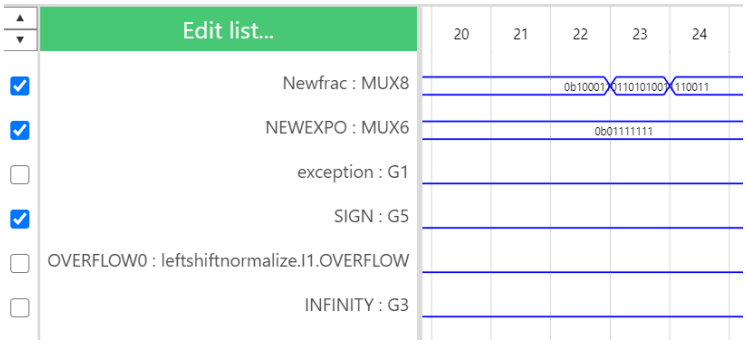
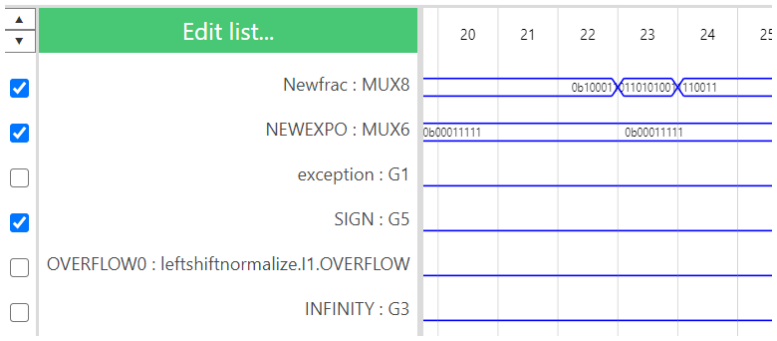
4	Overflow test: Both Exponents are equal to 127 and both of their integer parts equal to 1	
5	Test for smallest number, both of the exponents equal to -126, with their integer parts equal to 0	
6	Test for infinity, exponent A or B or both equal to 11111111	
7	Test integer, AB combination (00, 01, 10, 11) (00,11 are tested in the above tests)	01 condition

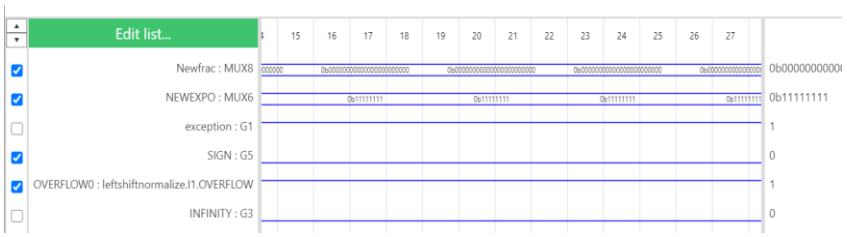
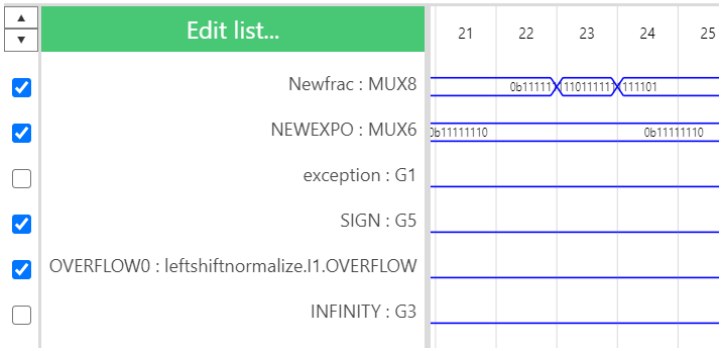
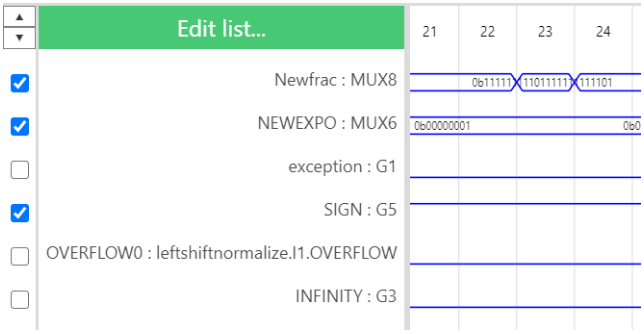
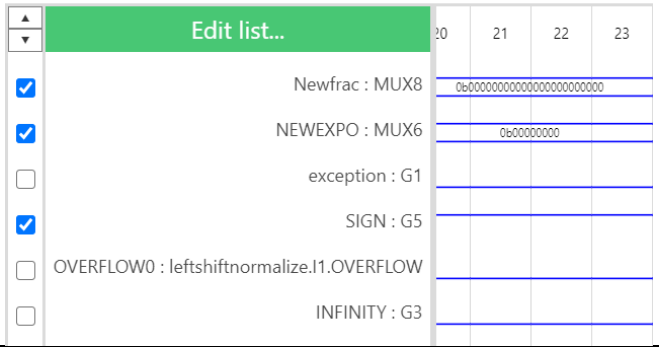
		
		10 condition 

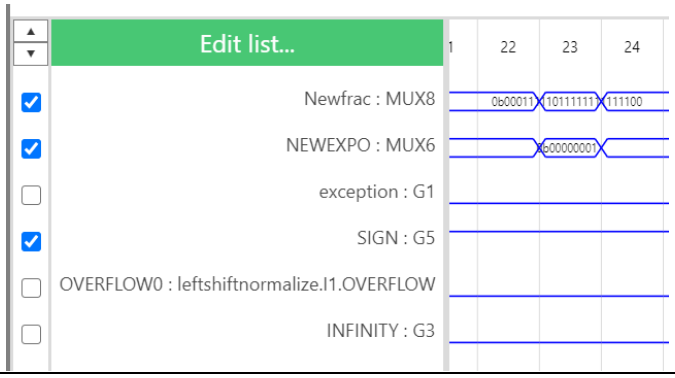
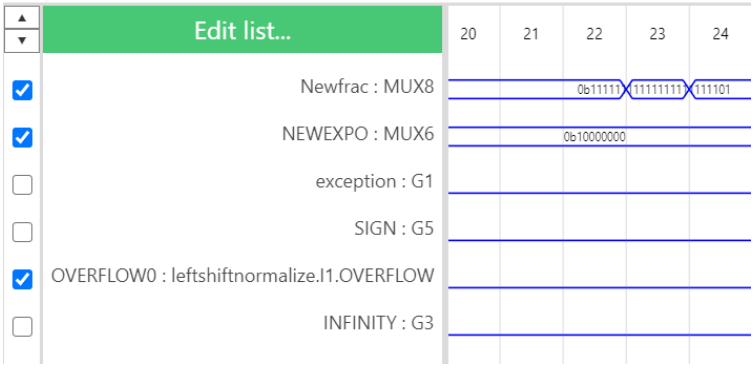
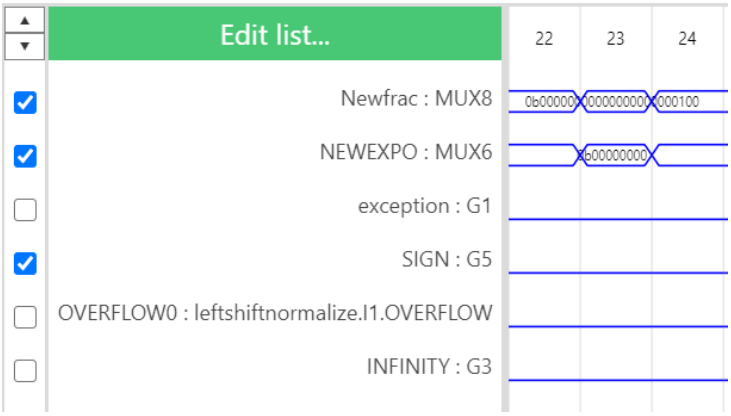
Binary32 Multiplication tests

Exhaustive Test tables from figure 37

		sign	integer	significand	exponent	overflow
1. Integer #1&1 Result is 3	Input1	0	1	111 1111 0100 1110 0000 1000	0111 1111	0
	Input2	0	1	111 1111 1100 0010 0111 0110	0111 1111	0
	output	0	1	111 1111 0001 0000 1010 1001	1000 0000	0
2. Integer #1&1 Result is 2	Input1	0	1	011 1111 0100 1110 0000 1000	0111 1111	0
	Input2	0	1	011 1111 1100 0010 0111 0110	0111 1111	0
	output	0	1	000 1111 0100 1100 1000 1001	1000 0000	1
3. Integer #1&1 Result is 1	Input1	0	1	001 1111 0100 1110 0000 1000	0111 1111	0
	Input2	0	1	001 1111 1100 0010 0111 0110	0111 1111	0
	output	0	1	100 0110 1101 0100 1111 0011	0111 1111	0
4. Integer #1&1 With Negative exponent	Input1	0	1	001 1111 0100 1110 0000 1000	0011 1111	0
	Input2	0	1	001 1111 1100 0010 0111 0110	0101 1111	0
	output	0	1	100 0110 1101 0100 1111 0011	0001 1111	0
5. Integer #1&1 Overflow test	Input1	0	1	001 1111 0100 1110 0000 1000	1011 1111	0
	Input2	0	1	001 1111 1100 0010 0111 0110	1101 1111	0
	output	0	NaN	000 0000 0000 0000 0000 0000	1111 1111	1
6. Integer #1&1 Smallest input without overflow	Input1	1	1	011 0001 1111 1111 1111 1111	0100 0000	0
	Input2	0	1	011 0111 1111 1111 1111 1111	0100 0000	0
	output	1	1	111 1111 1101 1111 1111 1111	0000 0001	0
7. Integer #1&1 Largest input without overflow	Input1	0	1	011 0001 1111 1111 1111 1111	1011 1111	0
	Input2	0	1	011 0111 1111 1111 1111 1111	1011 1110	0
	output	0	1	111 1111 1101 1111 1111 1111	1111 1110	0
8. Integer #0&0 Underflow	Input1	0	0	011 0001 1111 1111 1111 1111	0000 0000	0
	Input2	1	0	011 0111 1111 1111 1111 1111	0000 0000	0
	output	0	0	000 0000 0000 0000 0000 0000	0000 0000	0
9. Integer #0&1	Input1	0	0	011 0001 1111 1111 1111 1111	0000 0000	0
	Input2	1	1	011 0111 1111 1111 1111 1111	1111 1110	0
	output	1	1	000 1111 1101 1111 1111 1100	0111 1111	0
10. Integer #0&1 With negative exponent	Input1	0	0	011 0001 1111 1111 1111 1111	0000 0000	0
	Input2	1	1	011 0111 1111 1111 1111 1111	1000 0000	0
	output	1	1	000 1111 1101 1111 1111 1100	0000 0001	0
11. Integer #0&1 Smallest input without overflow	Input1	0	0	111 1111 1111 1111 1111 1111	0000 0000	0
	Input2	0	1	111 1111 1111 1111 1111 1111	1111 1110	0
	output	0	1	111 1111 1111 1111 1111 1101	1000 0000	0
12. Integer #0&1 Largest input without overflow	Input1	0	0	000 0000 0000 0000 0000 0001	0000 0000	0
	Input2	0	1	000 0000 0000 0000 0000 0001	1000 0000	0
	output	0	0	000 0000 0000 0000 0000 0010	0000 0000	0

1nteger #1&11	Result is 3	
2	Result is 2	
3	Result is 1	
4	With Negative exponent	

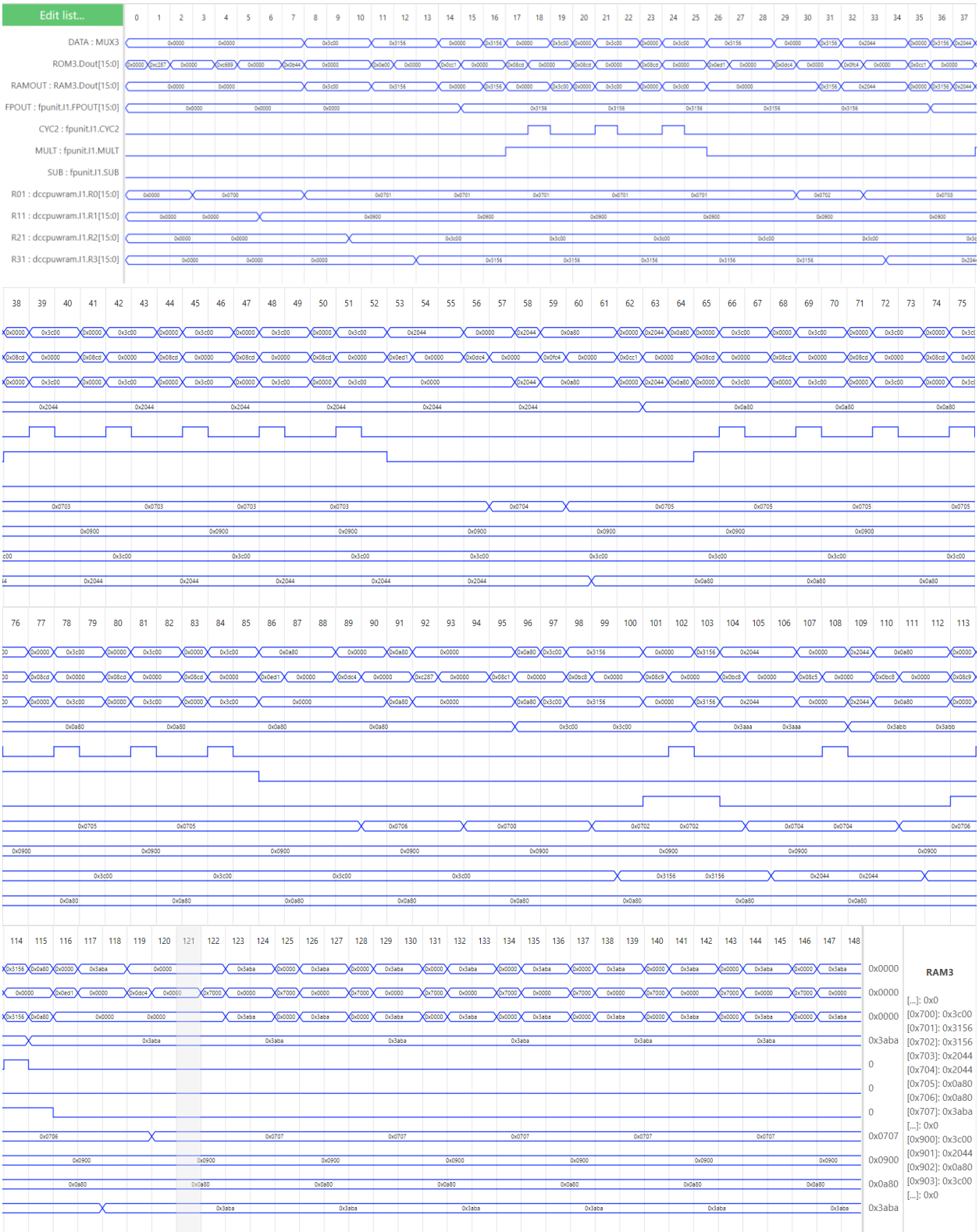
5.	Overflow	
6	Largest result without overflow	
7	Smallest result without overflow	
Integer parts #0&08	Underflow	
Integer parts #1&09		

10	With negative exponent	
11	Largest result without overflow	
12	Smallest result without overflow	

Appendix-III

- Waveform simulation screenshots

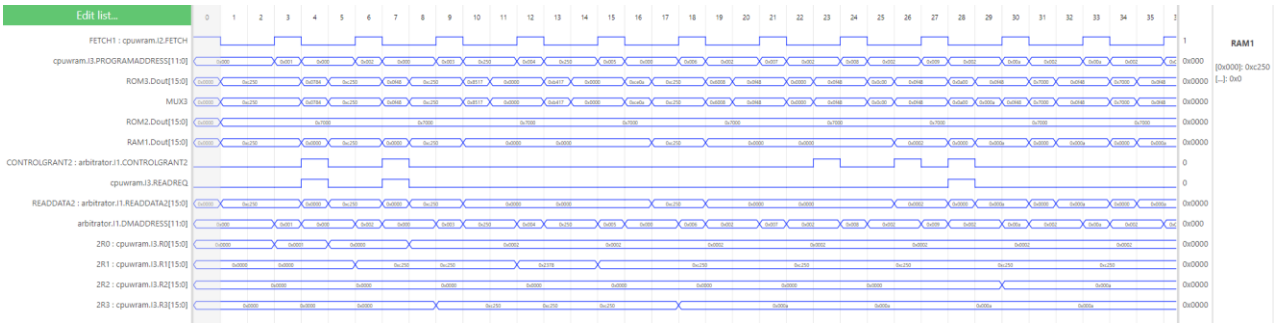
1. ***Sin x* test**



2. Single core without UART and floating-point arithmetic

Assembly Code and waveform results

Memory Address	Assembler	IR	Note
0x1	MOV R0 #0x01	0xC250	R0 = 0x0001
0x2	LDR R1 mem[R0-1] WB	0x0784	R0 = 0x0000 R1=0xC250
0x3	LDR R3 mem[R0] WB R0+2	0x0F48	R0 = 0x0002 R3=0xC250
0x4	ADDS R1=R1+Op2 Op2 = R1 LSR#1	0x8517	R1=2378 C=1
0x5	SBC R1=R1-Op2+C-1 Op2=R3 LSR#1	0xB417	R1=0xC250
0x6	MOV R3 #0x1010	0xCE0A	R3=0x000A
0x7	JEQ #0x008	0x6008	Jumpto8 Error
0x8 & 0x9	STA mem[R0] = R0 LDR R22=mem[R0]	0x0000 0x0A00	mem[0x0002]=0x0002 R22 = 0x002
0xA	STA mem[R0] = R3	0x0C00	mem[0x0002]=0x000A
0xB	LDR R2 = mem[R0]	0x0A00	R2=0x000A
0xC	STOP	0x7000	Stop here

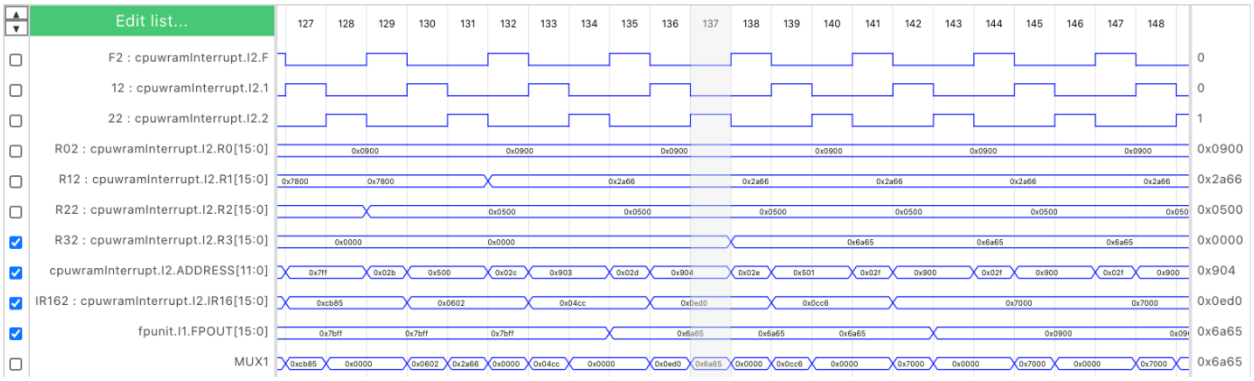


3. Mean of 20 arrays

Single core

Memory Address(single Core 1)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	MOV R2 #0x0014	8A14	R2 = 0x014
0x2	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x3	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x4	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x5	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x6	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17
0x7	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x8	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18
0x9	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xA	LDR R1 R2,#1	0746	R1 = mem[#0x0x18] R2 = 0x19
0xB	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1

0xC	LDR R1 R2,#1	0746	R1 = mem[#0x019] R2 = 0x1a
0xD	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xE	…	0746	….
0xF	…	04C4	….
0x2A	MOV R2 0x0500	CB85	R1 = 0x0500
0x2B	LDR R1 R2	0602	R1 = 0x2A66
0x2C	STR R1 [R0,#3]	04CC	mem[0x901]~Boperand mutiply:= R1
0x2D	LDR R3 [R0,#4]	0ED0	
0x2E	STR R3 [R2,#1]	0CC6	

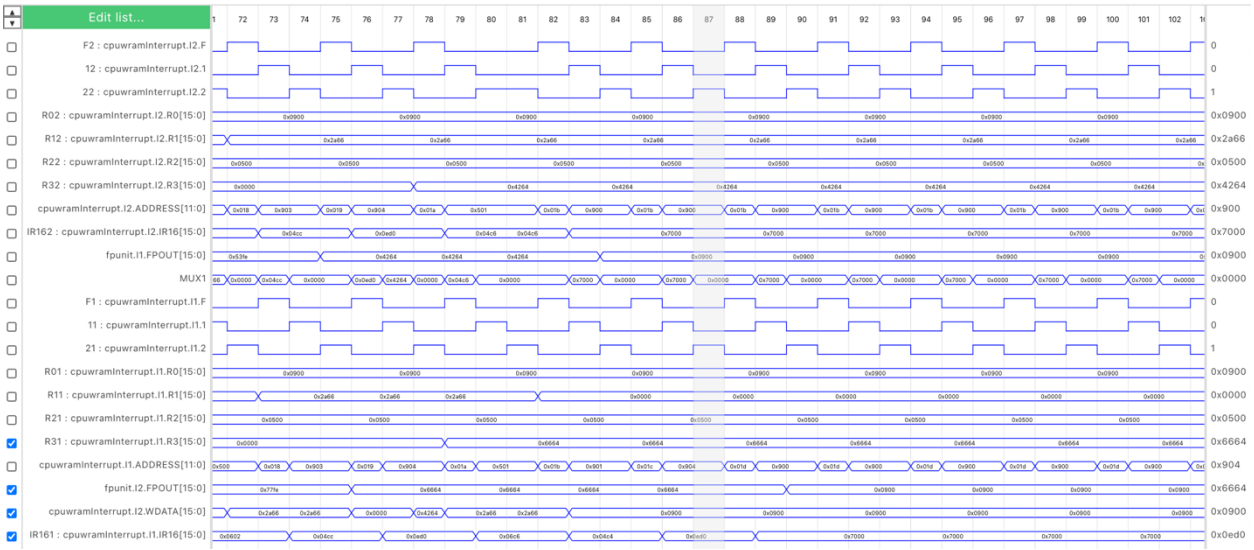


The mean of the array is stored in R3 at 137

Dual core

Memory Address (Dual Core 1)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	MOV R2 #0x0014	8A14	R2 = 0x014
0x2	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x3	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x4	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x5	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x6	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17
0x7	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x8	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18
0x9	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xA	LDR R1 R2,#1	0746	R1 = mem[#0x0x18] R2 = 0x19
0xB	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xC	LDR R1 R2,#1	0746	R1 = mem[#0x019] R2 = 0x1a
0xD	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xE	…	0746	….
0xF	…	04C4	….
0x16	MOV R2 0x0500	CB85	R1 = 0x0500
0x17	LDR R1 R2	0602	R1 = 0x2A66
0x18	STR R1 [R0,#3]	04CC	mem[0x901]~Boperand mutiply:= R1
0x19	LDR R3 [R0,#4]	0ED0	
0x1A	STR R3 [R2,#1]	0CC6	

Memory Address (Dual Core 2)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	MOV R2 #0x001D	8A1D	R2 = 0x01D
0x2	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x3	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x4	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x5	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x6	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17
0x7	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x8	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18
0x9	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xA	LDR R1 R2,#1		R1 = mem[#0x0x18] R2 = 0x19
0xB	STR R1 [R0,#1]		mem[0x901]~Boperand := R1
0xC	LDR R1 R2,#1		R1 = mem[#0x019] R2 = 0x1a
0xD	STR R1 [R0,#1]		mem[0x901]~Boperand := R1
0xE	…	….	….
0xF	…	….	….
0x16	MOV R2 0x0500	CB85	R1 = 0x0500
0x17	LDR R1 R2	0602	R1 = 0x2A66
0x18	STR R1 [R0,#3]	04CC	mem[0x903]~Boperand mutiply:= R1
0x19	LDR R3 [R0,#4]	0ED0	not necessary
0x1A	LDR R1 [R2,#1]	06C6	R1 := mem[0x701]
0x1B	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand add:= R1
0x1C	LDR R3 [R0,#4]	0ED0	



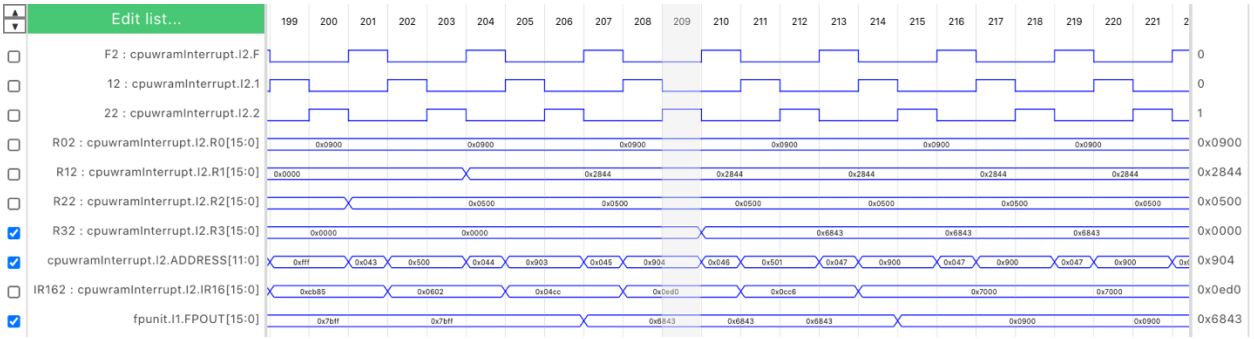
The mean of the last 20 elements is stored into R3 at clock 87.

4. **Mean of 30 arrays**

Single core

Single-Core test set:

Memory Address (single Core 1)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	MOV R2 #0x000F	8A0A	R2 = 0x000A
0x2	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x3	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x4	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x5	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x6	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17
0x7	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x8	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18
0x9	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xA	LDR R1 R2,#1	0746	R1 = mem[#0x0x18] R2 = 0x19
0xB	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xC	LDR R1 R2,#1	0746	R1 = mem[#0x019] R2 = 0x1a
0xD	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xE	…	0746	….
0xF	…	04C4	….
0x42	MOV R2 0x0500	CB85	R1 = 0x0500
0x43	LDR R1 R2	0602	R1 = 0x2844
0x44	STR R1 [R0,#3]	04CC	mem[0x901]~Boperand mutiply:= R1
0x45	LDR R3 [R0,#4]	0ED0	
0x46	STR R3 [R2,#1]	0CC6	



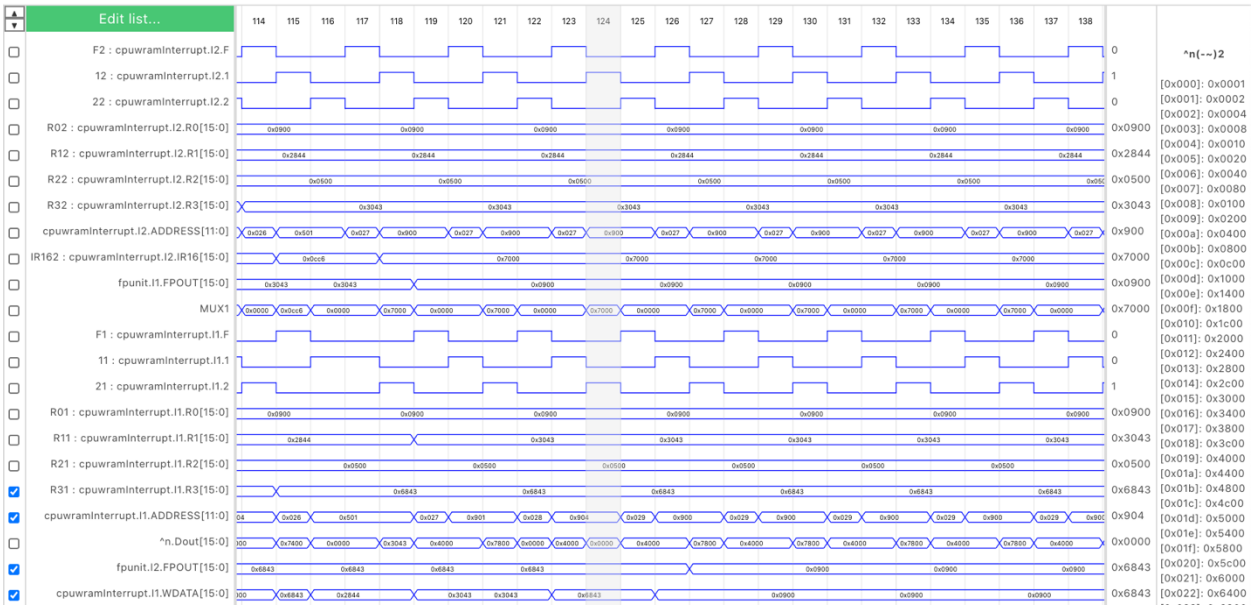
The mean stored to R3 at 209.

Dual core

Memory Address (Dual Core 1)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	MOV R2 #0x000F	8A0A	R2 = 0x000A
0x2	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x3	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x4	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x5	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x6	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17
0x7	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x8	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18

0x9	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xA	LDR R1 R2,#1	0746	R1 = mem[#0x0x18] R2 = 0x19
0xB	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xC	LDR R1 R2,#1	0746	R1 = mem[#0x019] R2 = 0x1a
0xD	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xE	…		….
0xF	…		….
0x22	MOV R2 0x0500	CB85	R1 = 0x0500
0x23	LDR R1 R2	0602	R1 = 0x2844
0x24	STR R1 [R0,#3]	04CC	mem[0x901]~Boperand mutiply:= R1
0x25	LDR R3 [R0,#4]	0ED0	
0x26	STR R3 [R2,#1]	0CC6	

Memory Address (Dual Core 2)	**Assembler**	**IR**	**Note**
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	MOV R2 #0x001D	8A19	R2 = 0x0019
0x2	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x3	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x4	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x5	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x6	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17
0x7	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x8	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18
0x9	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xA	LDR R1 R2,#1	0746	R1 = mem[#0x0x18] R2 = 0x19
0xB	STR R1 [R0,#1]	04CA	mem[0x901]~Boperand := R1
0xC	LDR R1 R2,#1	0746	R1 = mem[#0x019] R2 = 0x1a
0xD	STR R1 [R0,#1]	04CA	mem[0x901]~Boperand := R1
0xE	…	….	….
0xF	…	….	….
0x22	MOV R2 0x0500	CB85	R1 = 0x0500
0x23	LDR R1 R2	0602	R1 = 0x2844
0x24	STR R1 [R0,#3]	04CC	mem[0x903]~Boperand mutiply:= R1
0x25	LDR R3 [R0,#4]	0ED0	not necessary
0x26	LDR R1 [R2,#1]	06C6	R1 := mem[0x701]
0x27	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand add:= R1
0x28	LDR R3 [R0,#4]	0ED0	



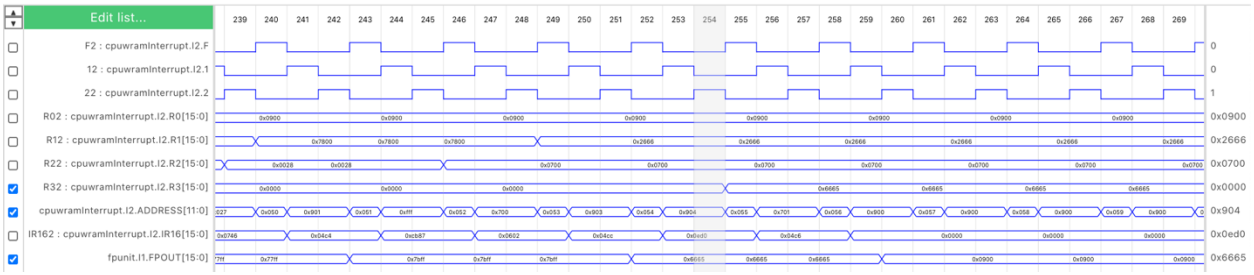
Mean of the last 30 elements is stored to R3 at clock 124.

5. **Mean of 40 arrays (Assembly code and waveform results)**

Single-Core

Memory Address (Single Core 1)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	LDR R1 R2,#1	0746	R1 = mem[#0x0] R2 = 0x1
0x2	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x3	LDR R1 R2,#1	0746	R1 = mem[#0x1] R2 = 0x2
0x4	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x5	LDR R1 R2,#1	0746	R1 = mem[#0x2] R2 = 0x3
0x6	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x7	LDR R1 R2,#1	0746	R1 = mem[#0x3] R2 = 0x4
0x8	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x9	LDR R1 R2,#1	0746	R1 = mem[#0x4] R2 = 0x5
0xA	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xB	LDR R1 R2,#1	0746	R1 = mem[#0x6] R2 = 0x7
0xC	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xD	…	…	…
0xE	…	…	…
0xF	…	…	…
0x1			
0x29	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x2A	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x2B	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x2C	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x2D	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17

0x2E	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x2F	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18
0x30	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x31	LDR R1 R2,#1	0746	R1 = mem[#0x0x18] R2 = 0x19
0x32	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x33	LDR R1 R2,#1	0746	R1 = mem[#0x019] R2 = 0x1a
0x34	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x35	…	….	….
0x36	…	….	….
0x51	MOV R2 0x0700	CB87	R1 = 0x0700
0x52	LDR R1 R2	0602	R1 = 0x0019
0x53	STR R1 [R0,#3]	04CC	mem[0x901]~Boperand mutiply:= R1
0x54	LDR R3 [R0,#4]	0ED0	
0x55	STR R3 [R2,#1]	04C6	



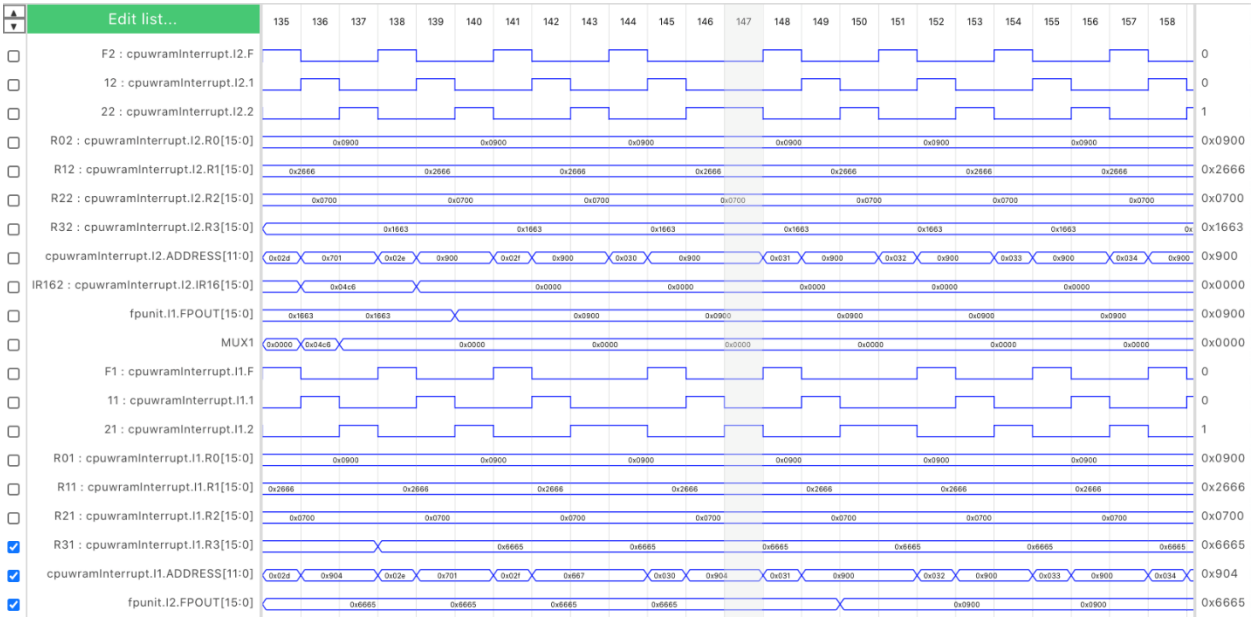
The mean of the array is stored into R3 at clock 254.

Dual-Core

Memory Address (Core 1)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	LDR R1 R2,#1	0746	R1 = mem[#0x0] R2 = 0x1
0x2	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x3	LDR R1 R2,#1	0746	R1 = mem[#0x1] R2 = 0x2
0x4	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x5	LDR R1 R2,#1	0746	R1 = mem[#0x2] R2 = 0x3
0x6	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x7	LDR R1 R2,#1	0746	R1 = mem[#0x3] R2 = 0x4
0x8	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x9	LDR R1 R2,#1	0746	R1 = mem[#0x4] R2 = 0x5
0xA	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xB	LDR R1 R2,#1	0746	R1 = mem[#0x6] R2 = 0x7
0xC	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xD	…	…	…
0xE	…	…	…
0xF	…	…	…
0x19	MOV R2 0x0700	CB87	R1 = 0x0700
0x1A	LDR R1 R2	0602	R1 = 0x0019

0x1B	STR R1 [R0,#3]	04CC	mem[0x901]~Boperand mutiply:= R1
0x1C	LDR R3 [R0,#4]	0ED0	
0x1D	STR R3 [R2,#1]	0CC6	

Memory Address (Core 2)	Assembler	IR	Note
0x0	MOV R0 #0x0900	C289	R0 = 0x0900
0x1	MOV R2 #0x0014	8A14	R2 = 0x014
0x2	LDR R1 R2,#1	0746	R1 = mem[#0x014] R2 = 0x015
0x3	STR R1 R0	0400	mem[0x900]~Aoperand := R1
0x4	LDR R1 R2,#1	0746	R1 = mem[#0x015] R2 = 0x016
0x5	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x6	LDR R1 R2,#1	0746	R1 = mem[#0x016] R2 = 0x17
0x7	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0x8	LDR R1 R2,#1	0746	R1 = mem[#0x0x17] R2 = 0x18
0x9	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xA	LDR R1 R2,#1	0746	R1 = mem[#0x0x18] R2 = 0x19
0xB	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xC	LDR R1 R2,#1	0746	R1 = mem[#0x019] R2 = 0x1a
0xD	STR R1 [R0,#1]	04C4	mem[0x901]~Boperand := R1
0xE	…	….	….
0xF	…	….	….
0x2A	MOV R2 0x0700	CB87	R1 = 0x0700
0x2B	LDR R1 R2	0602	R1 = 0x0019
0x2C	STR R1 [R0,#3]	04CC	mem[0x903]~Boperand mutiply:= R1
0x2D	LDR R3 [R0,#4]	0ED0	R3 = (sum of the last 20 number/ 20)
0x2E	LDR R1 [R2,#1]	06C6	R1 := mem[0x701], load mean from core 1
0x2F	STR R1 [R0,#1]	04C4	mem[0x901]~B operand add:= R1
0x30	LDR R3 [R0,#4]	0ED0	R3 = mean of 2n array

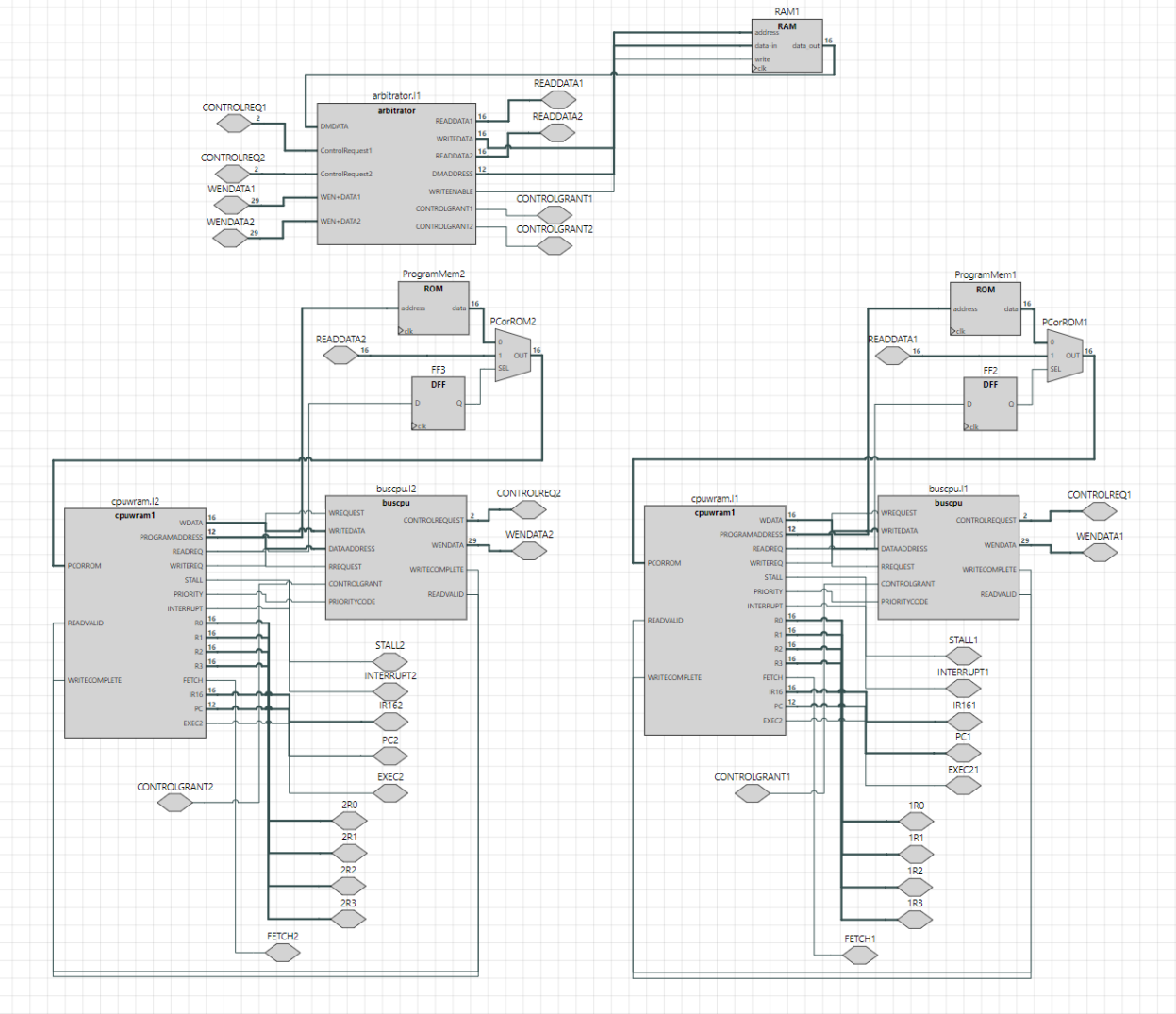


The mean of the array is stored into R3 at clock 147.

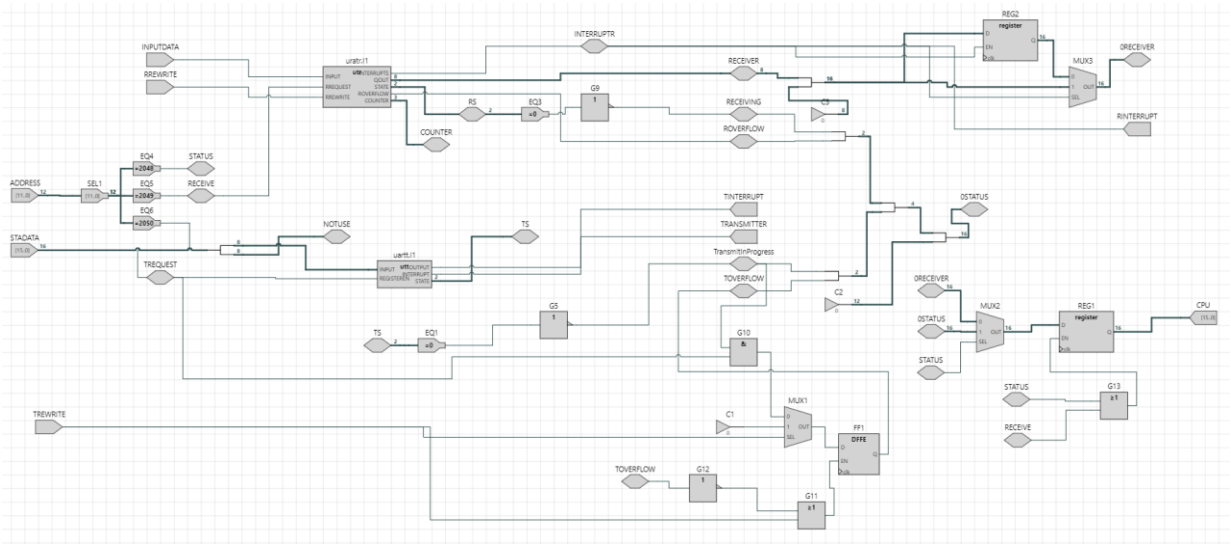
Appendix-V

- Screenshots of Issie block designs

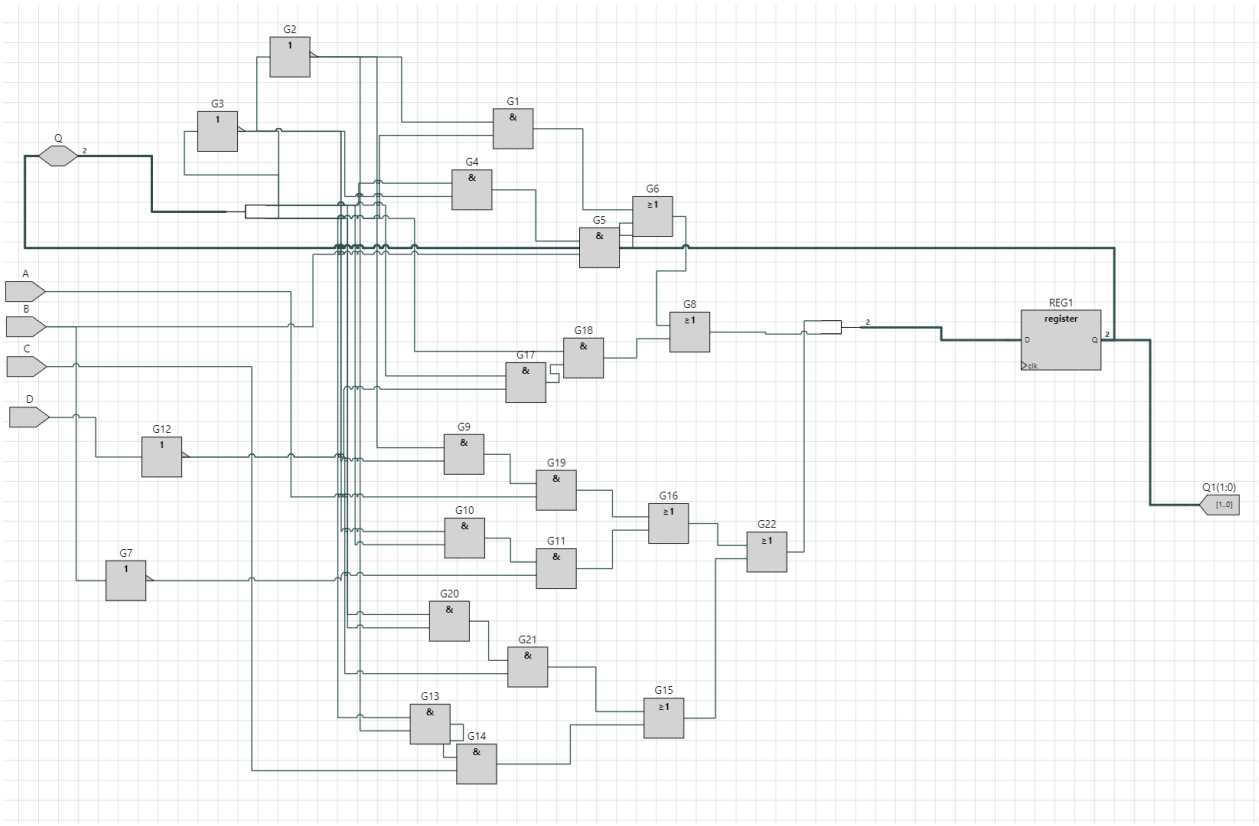
Dual Core



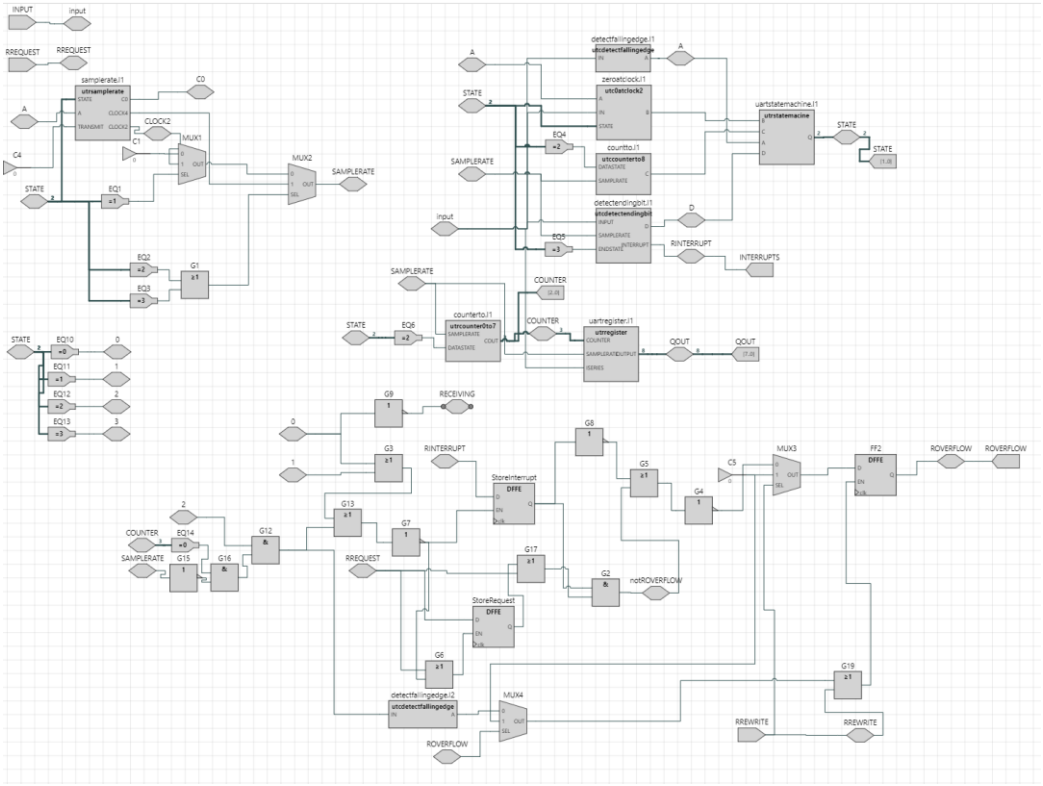
UART



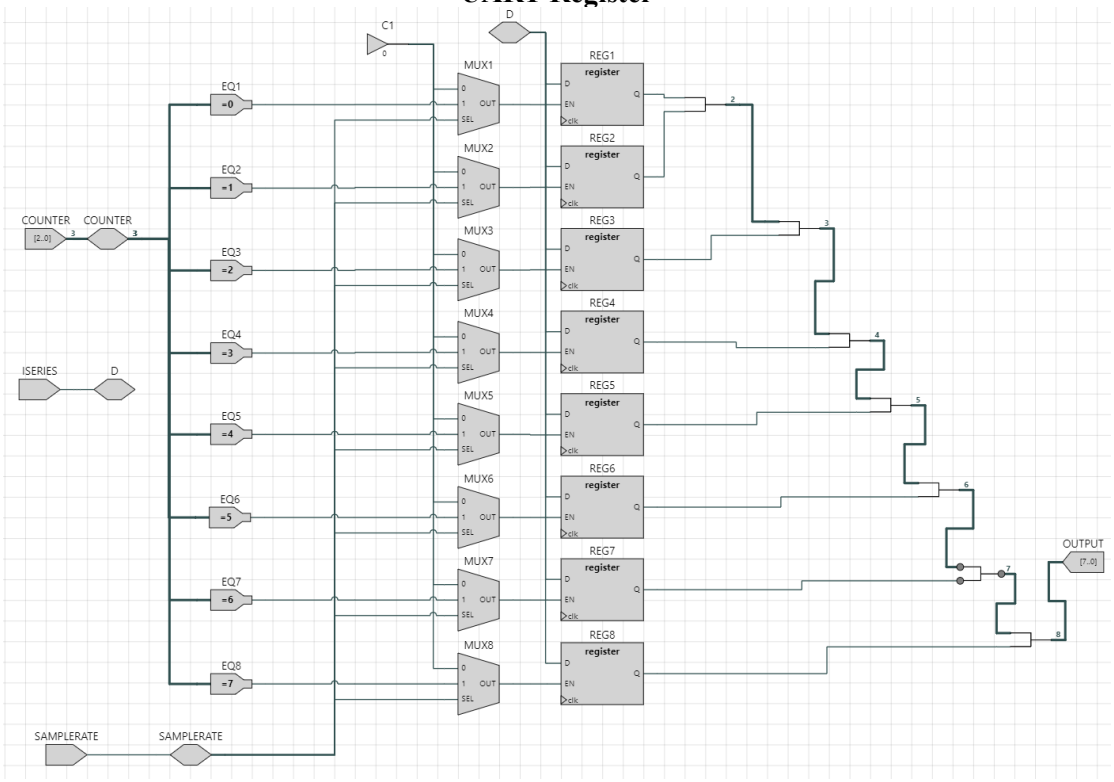
UART State Machine



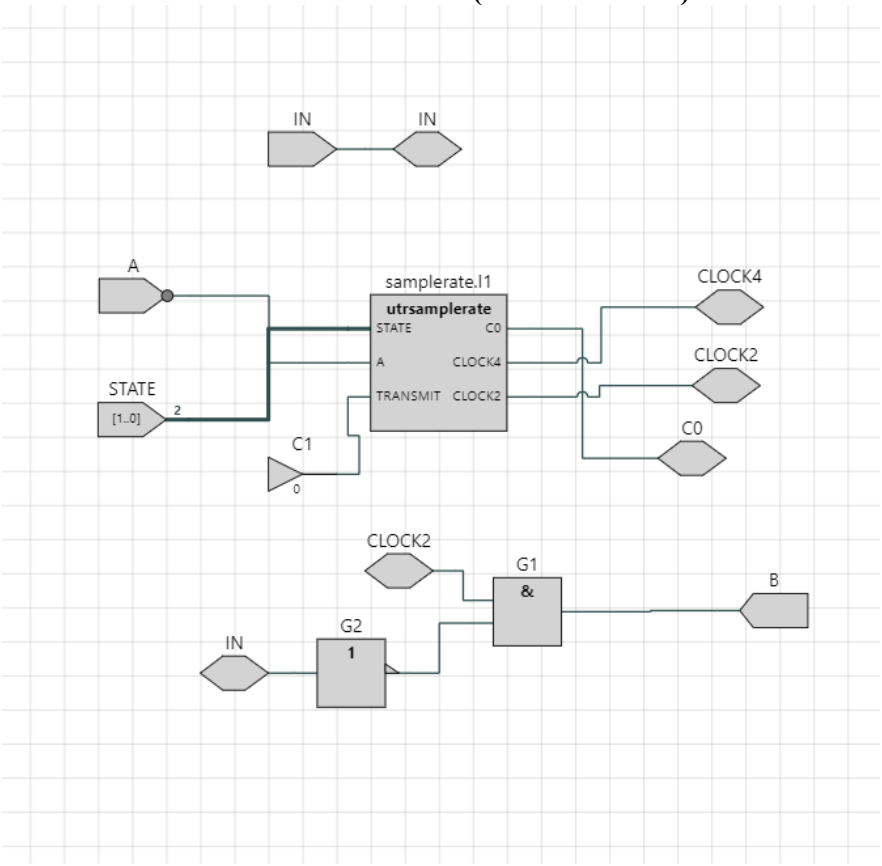
UART Receiver



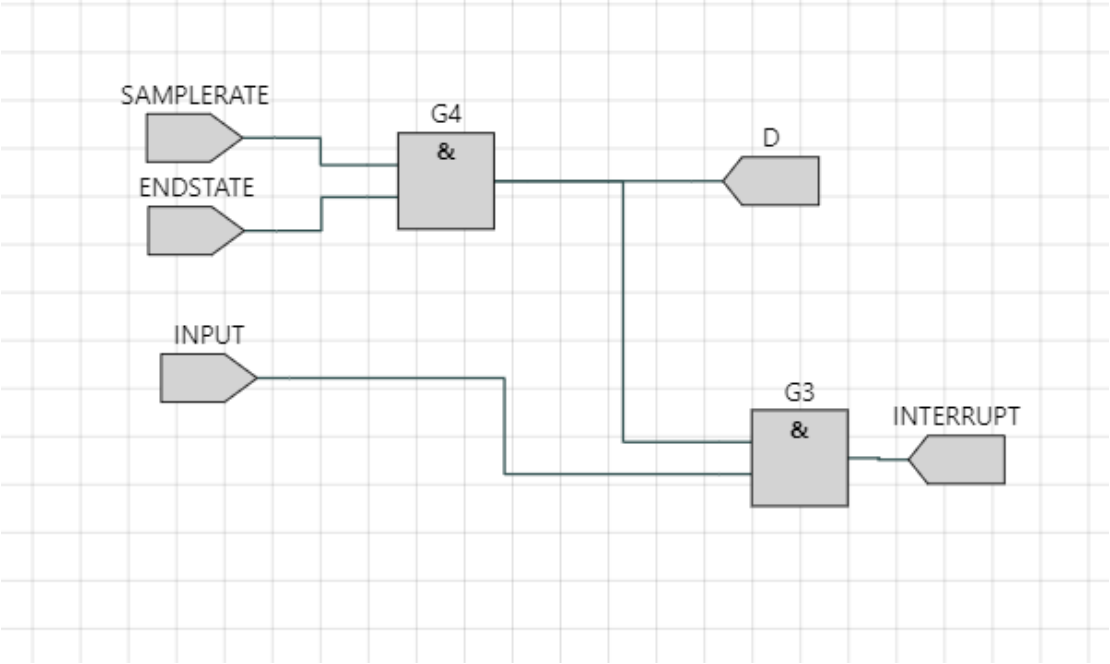
UART Register



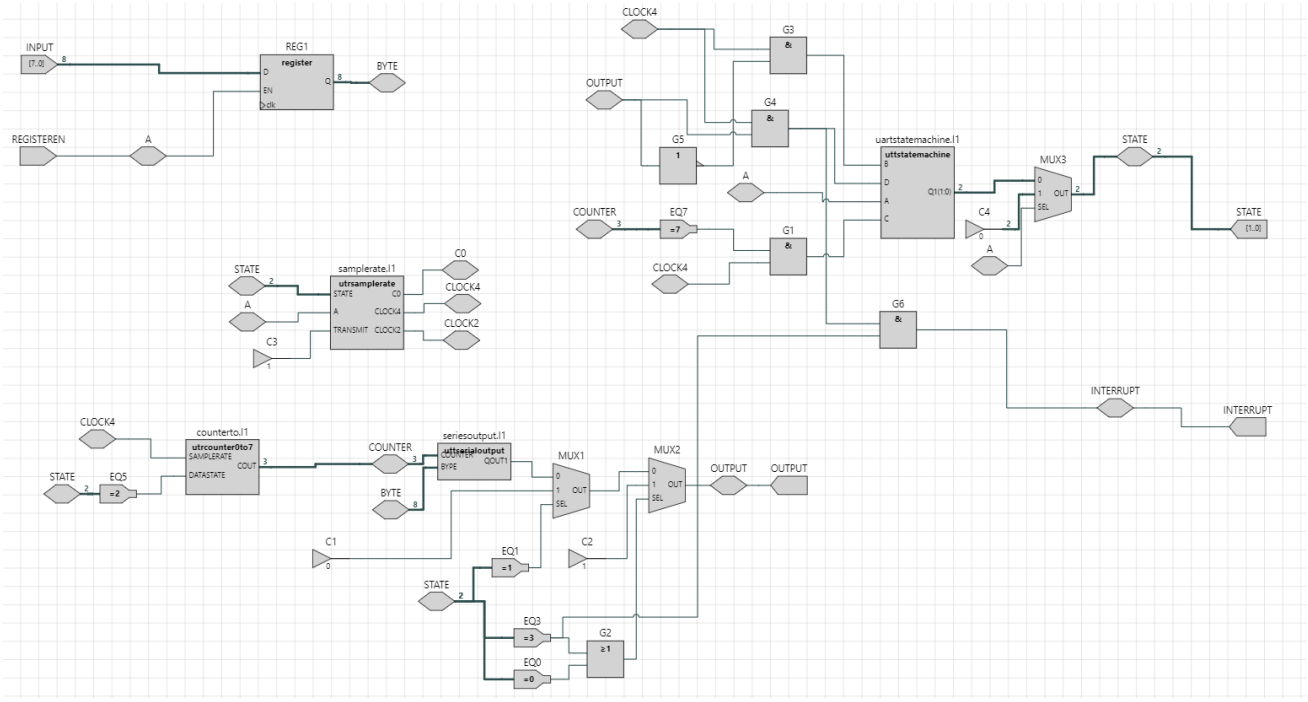
Zero at Clock 2 Block (UART Receiver)



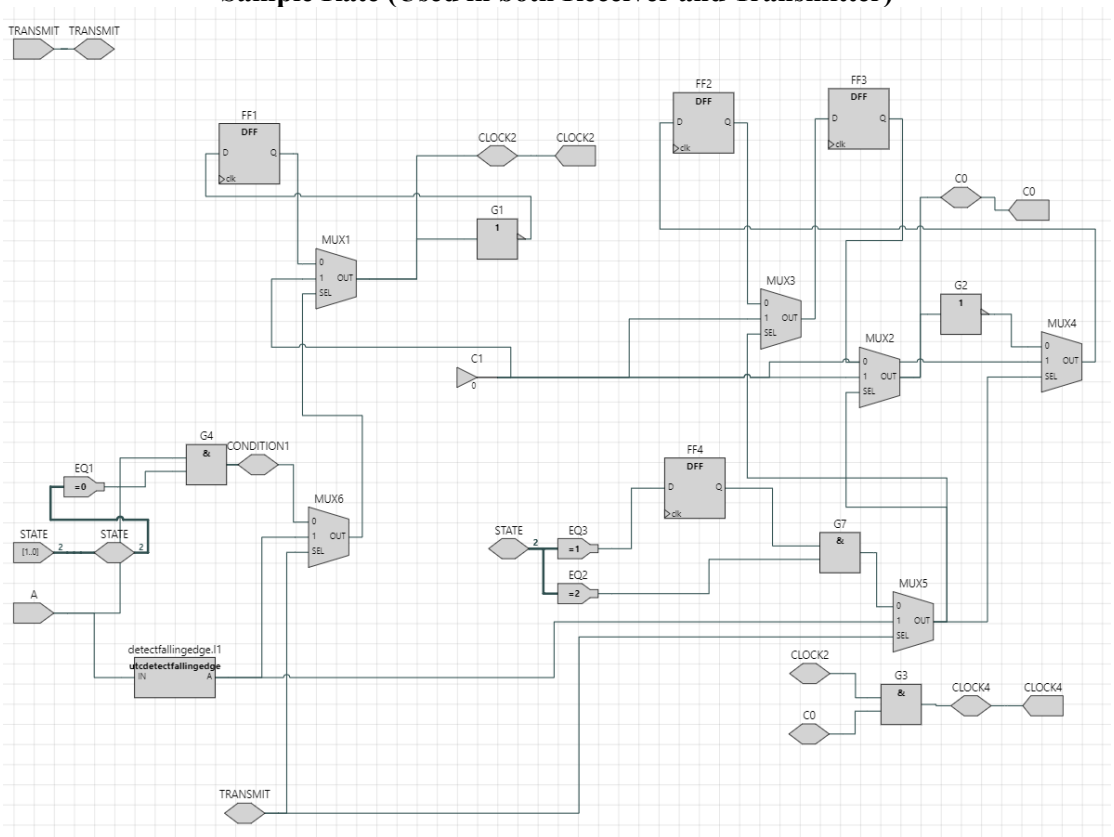
Detect ending bit Block (UART Receiver)



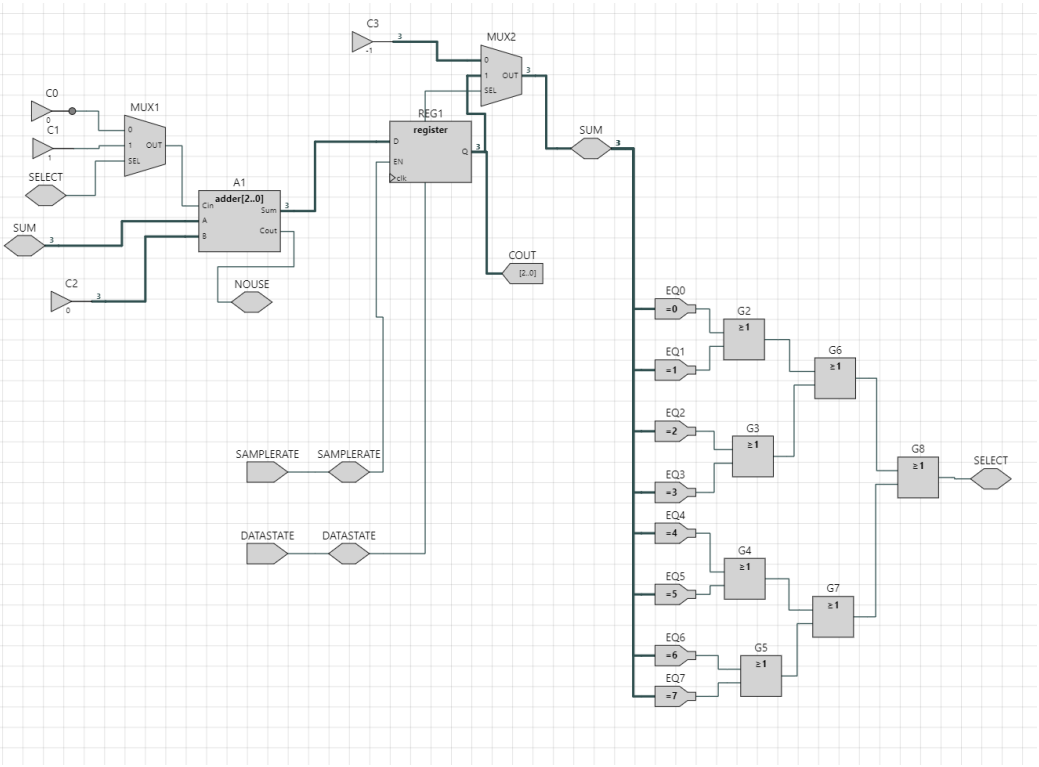
UART Transmitter



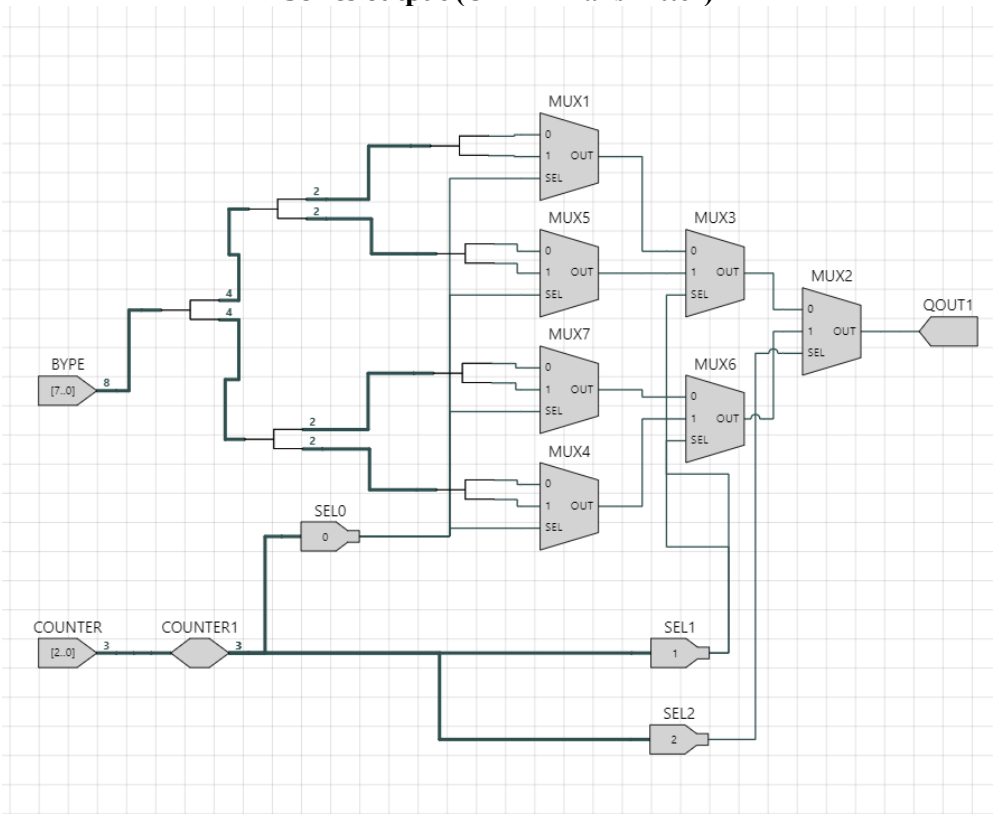
Sample Rate (Used in both Receiver and Transmitter)



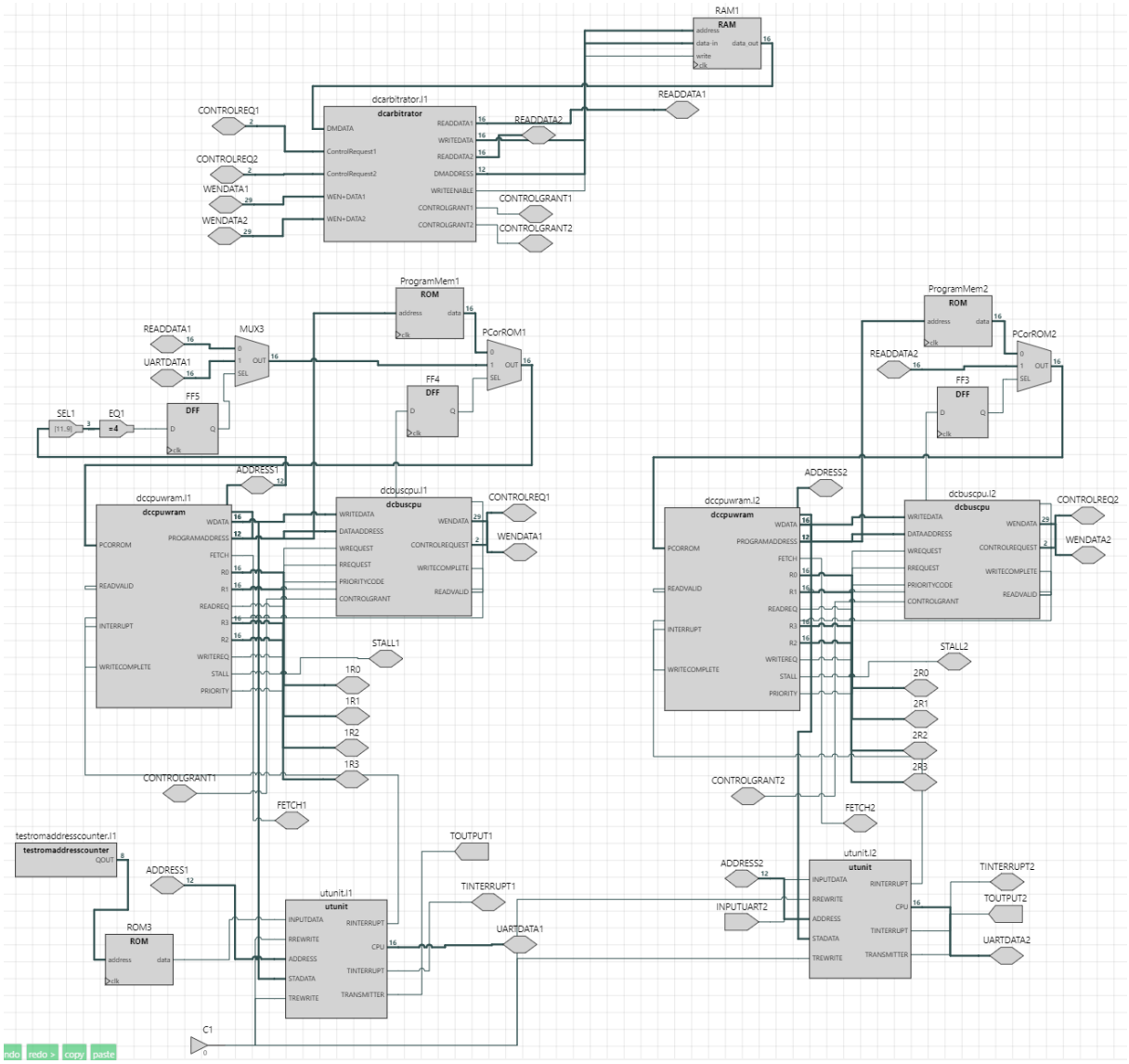
Count-to-7 Block



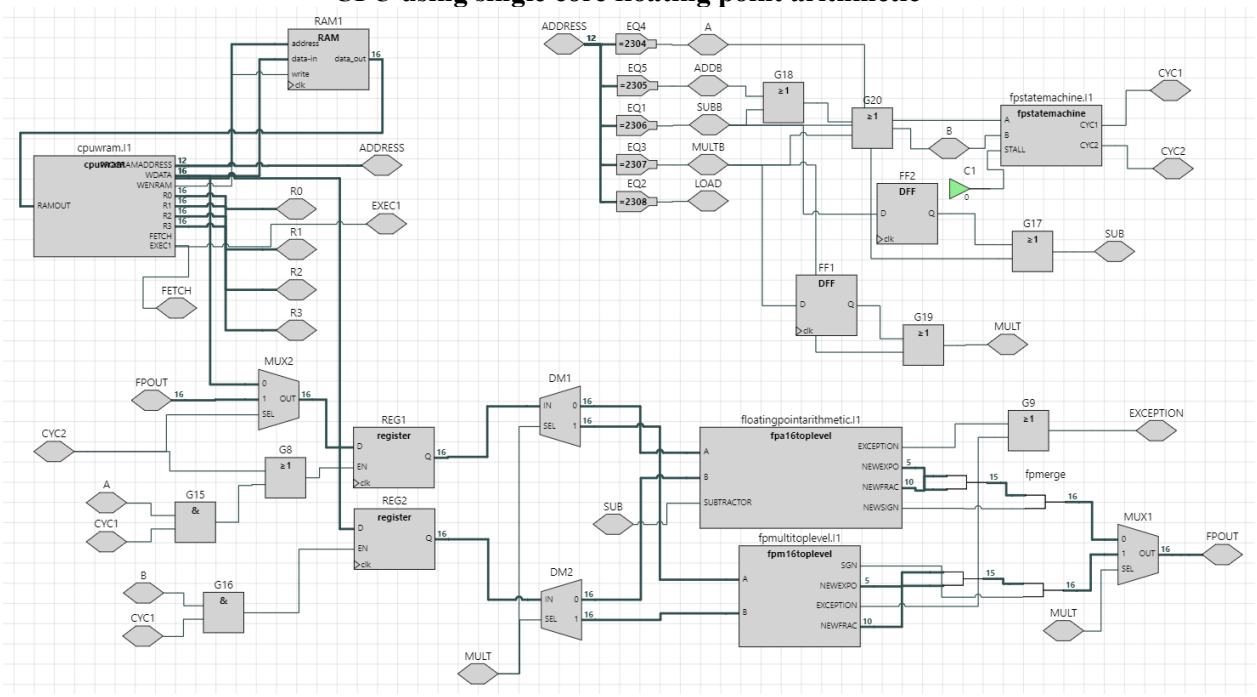
Series output (UART Transmitter)



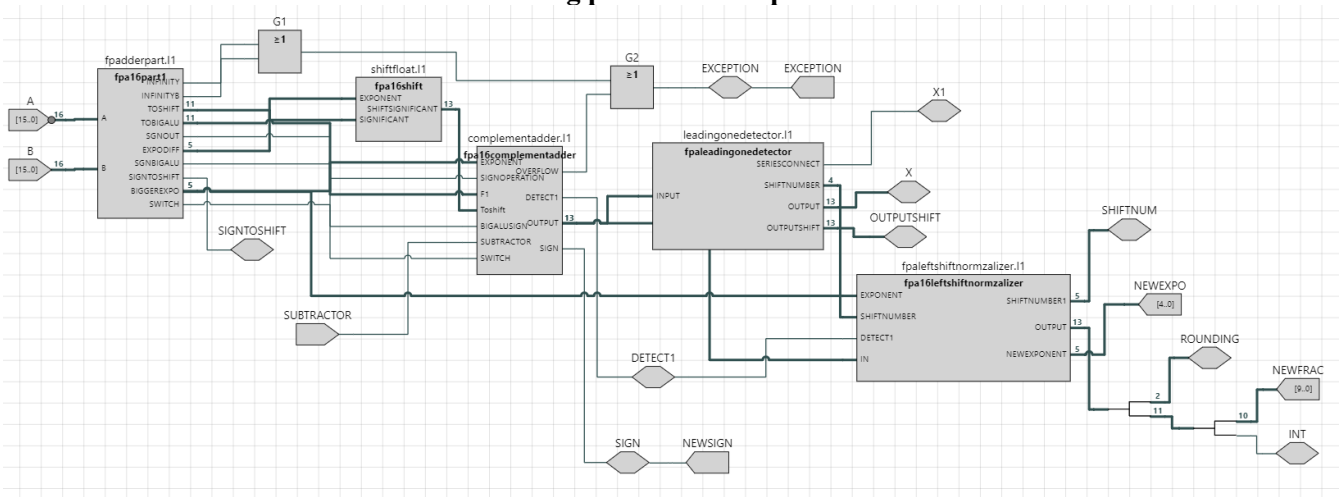
Dual Core UART



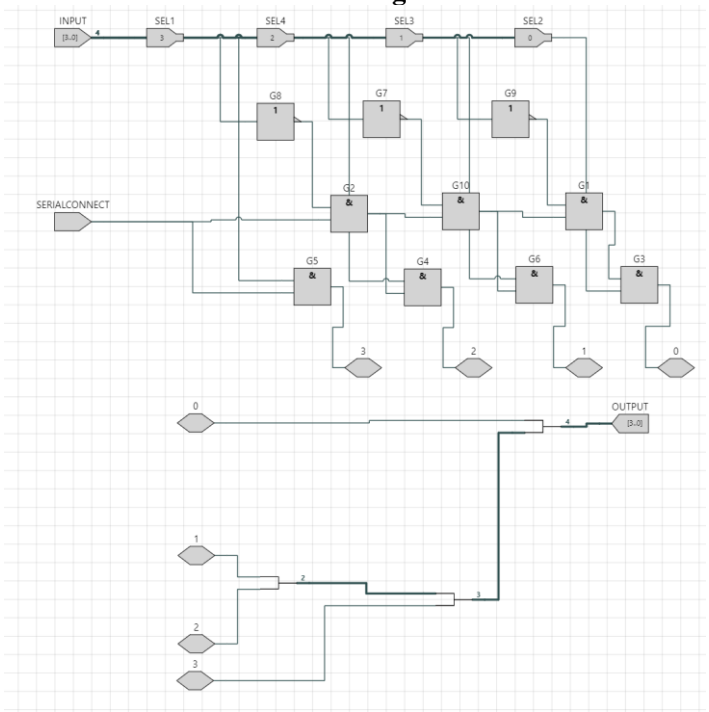
CPU using single core floating point arithmetic



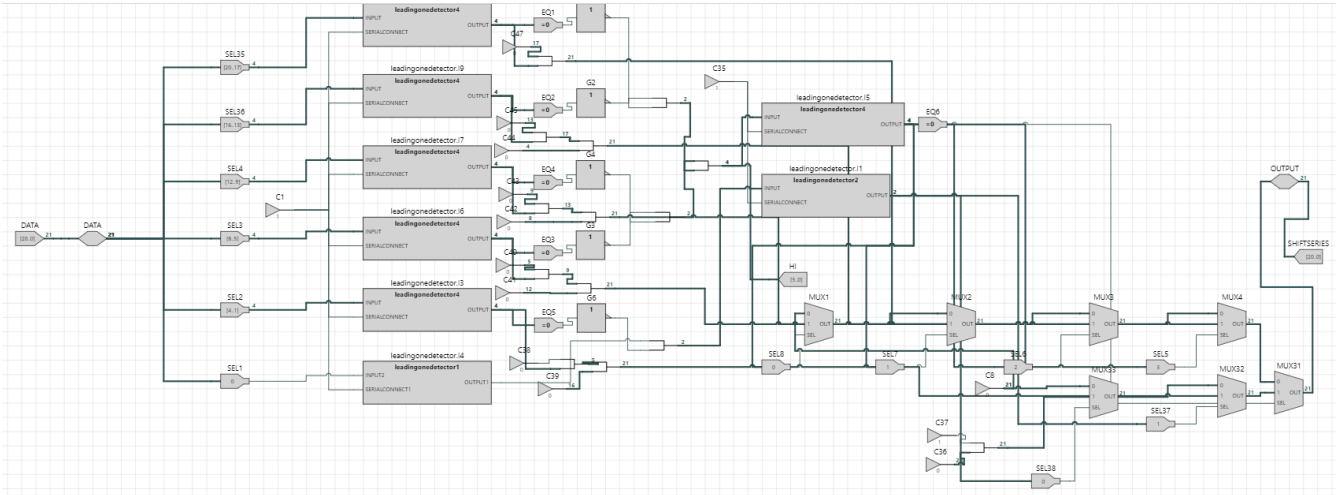
16-bit floating point Adder top level



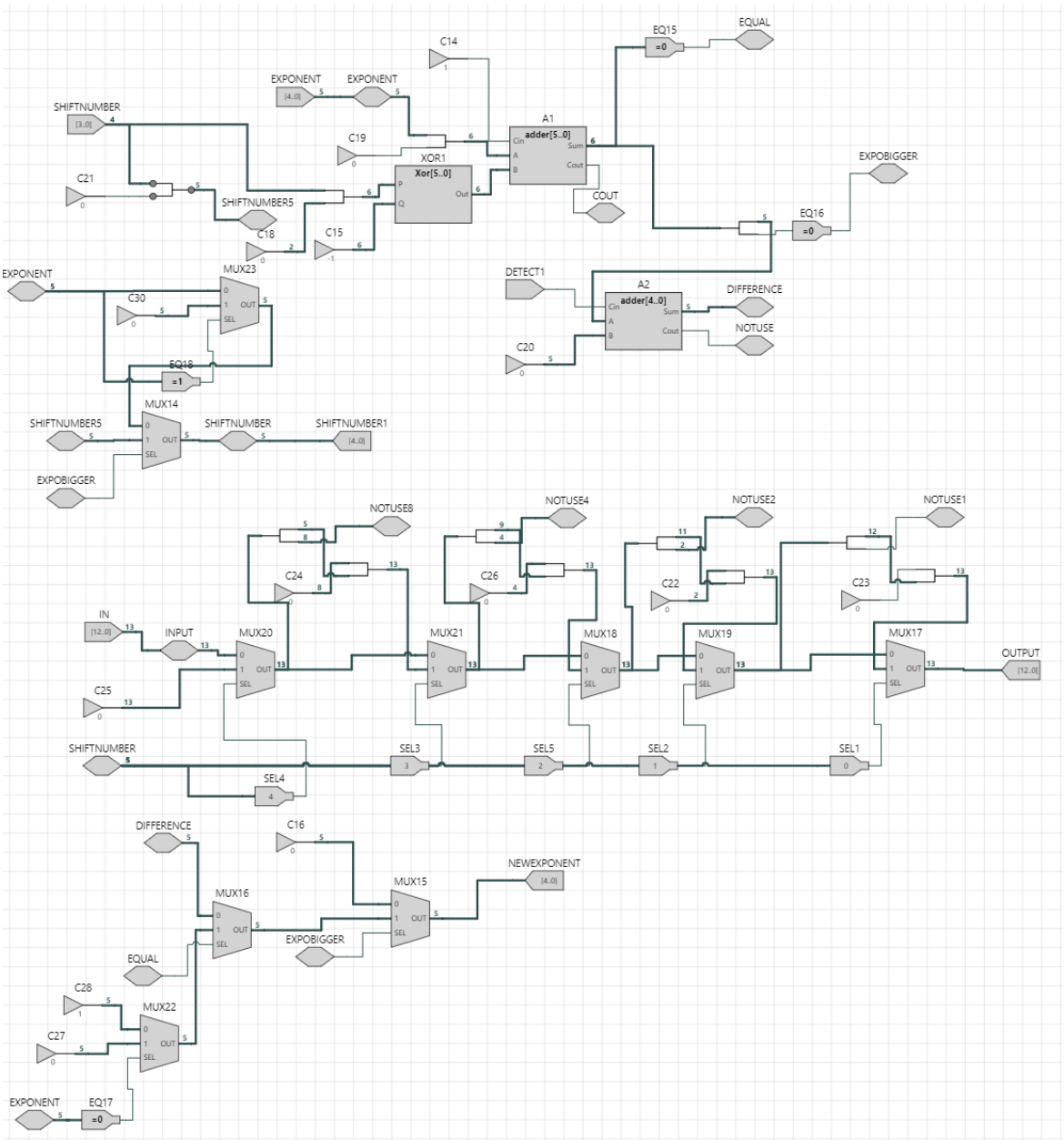
4 bit in series leading one detector



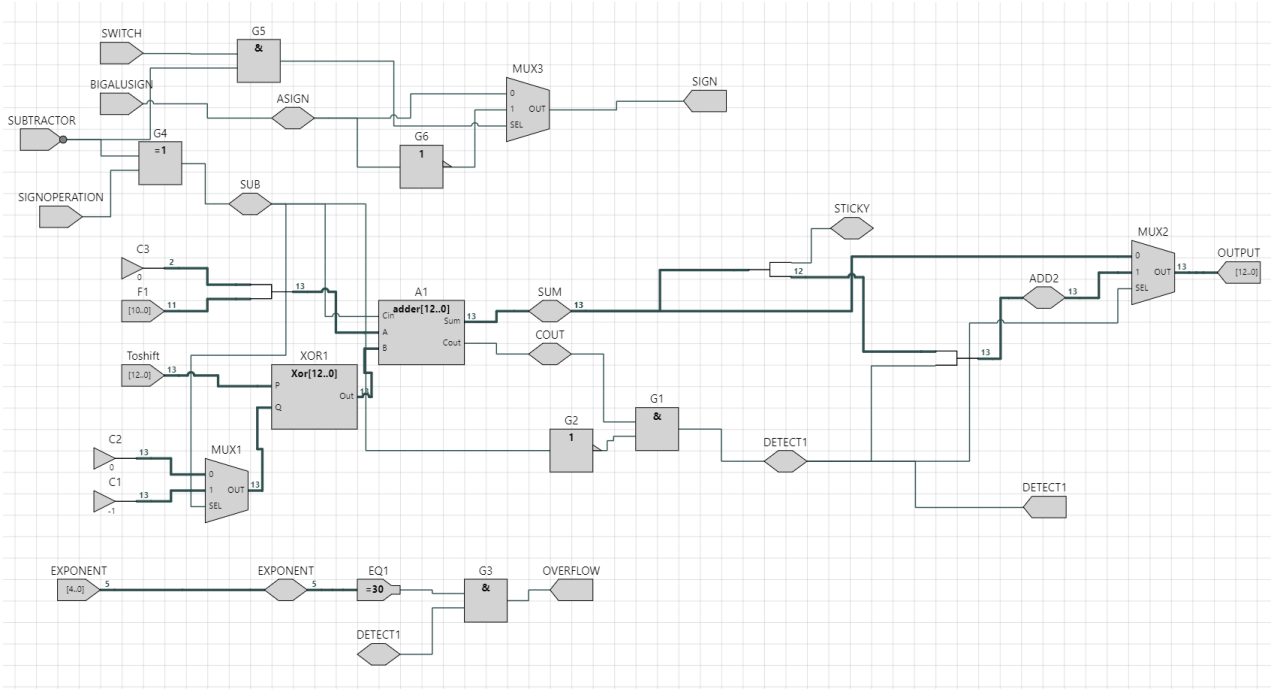
21 bits parallel leading one detector top level



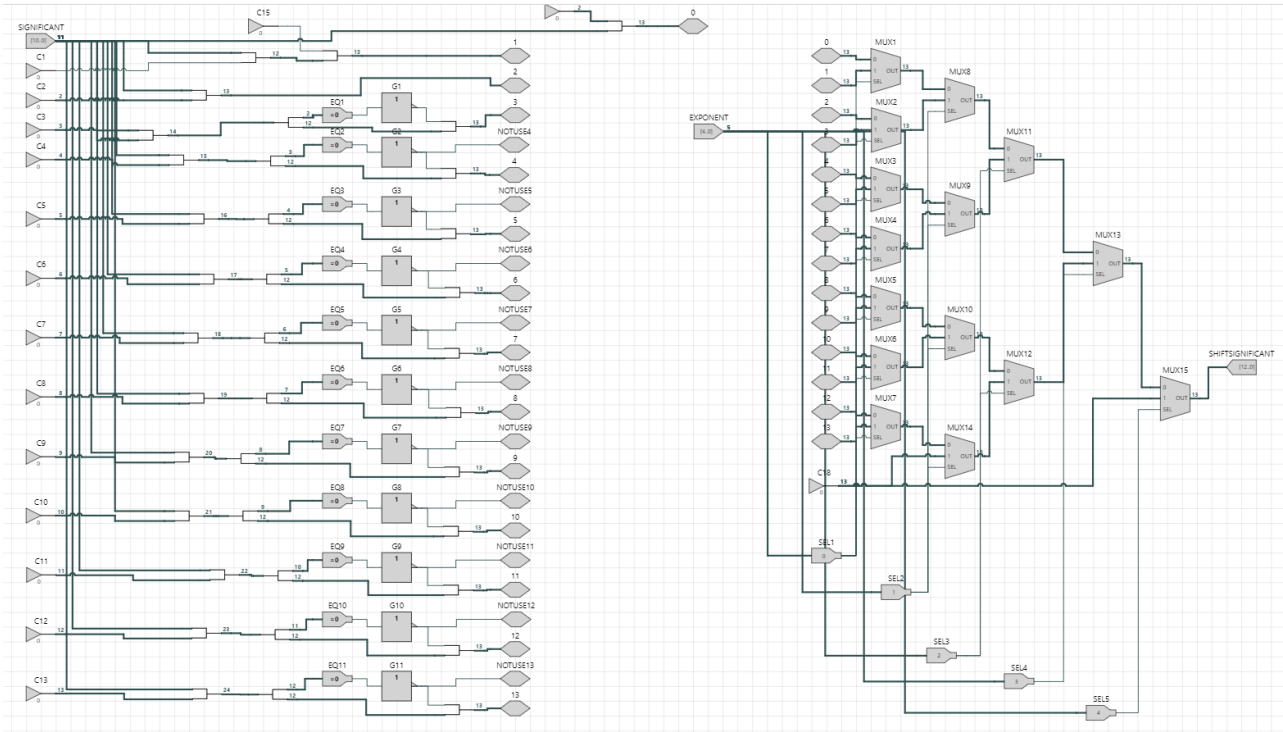
Floating point Adder left shift normalizer



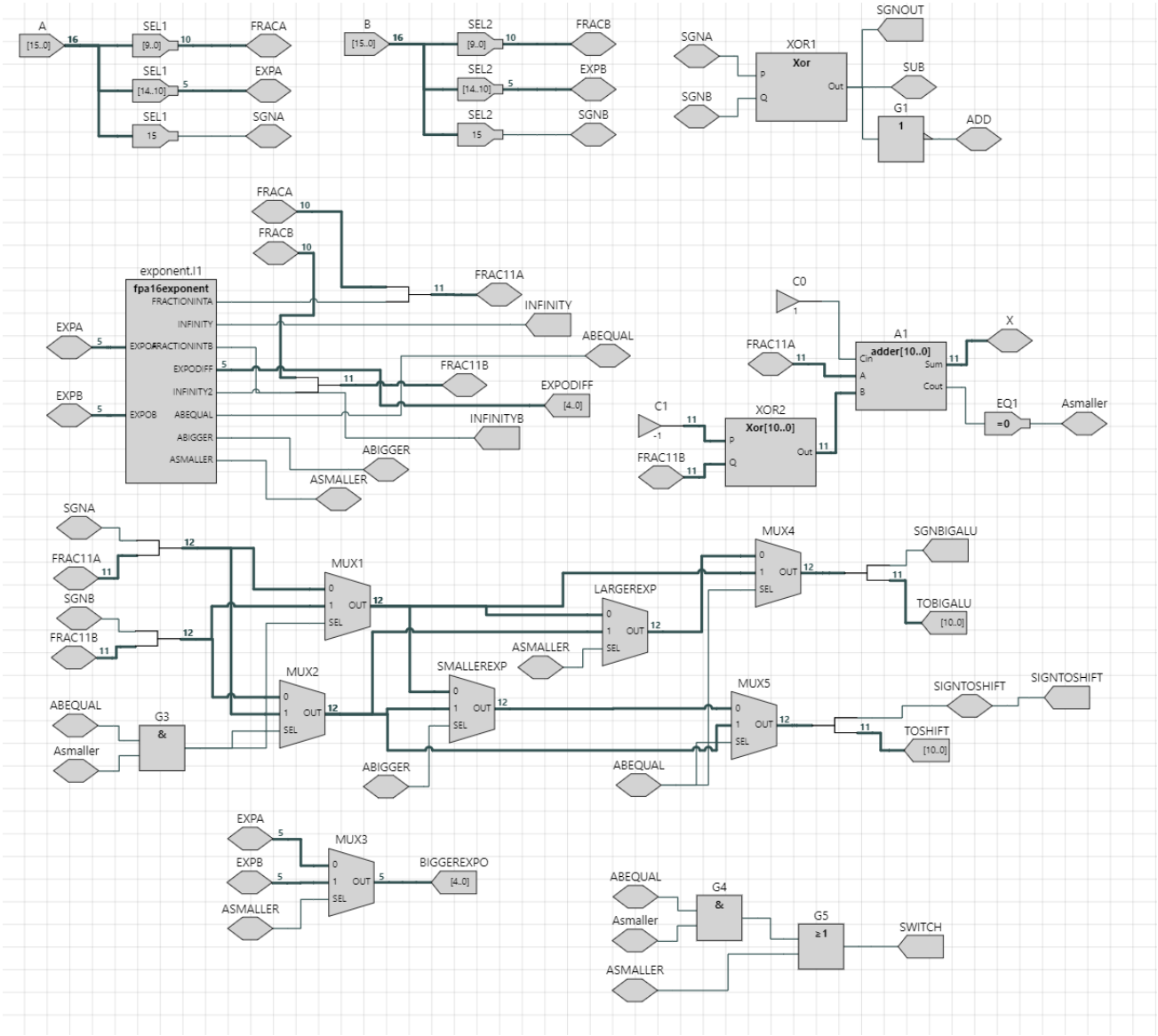
Floating point 16 bit complement Adder



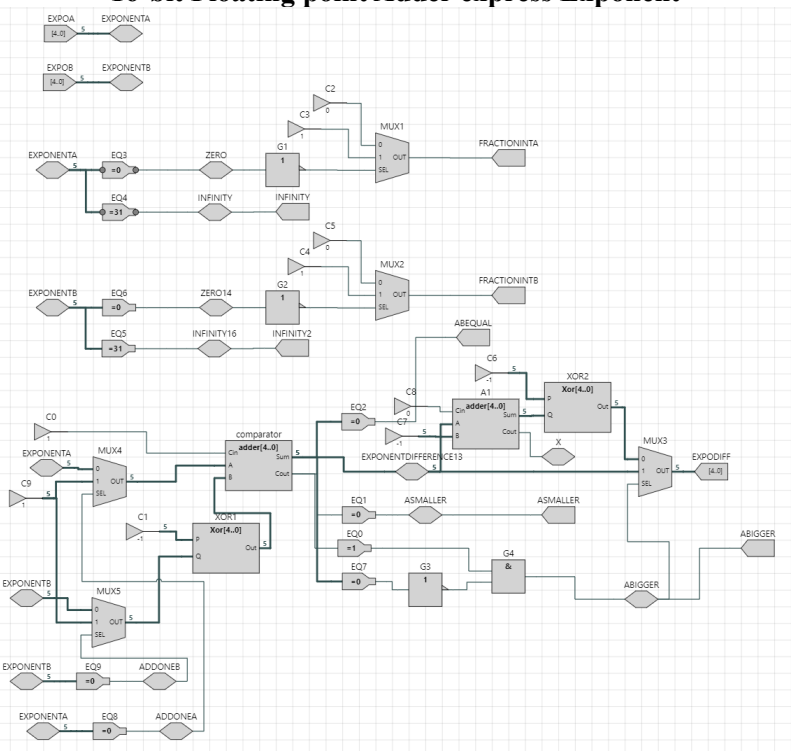
13-bit shift unit in 16-bit floating point Adder



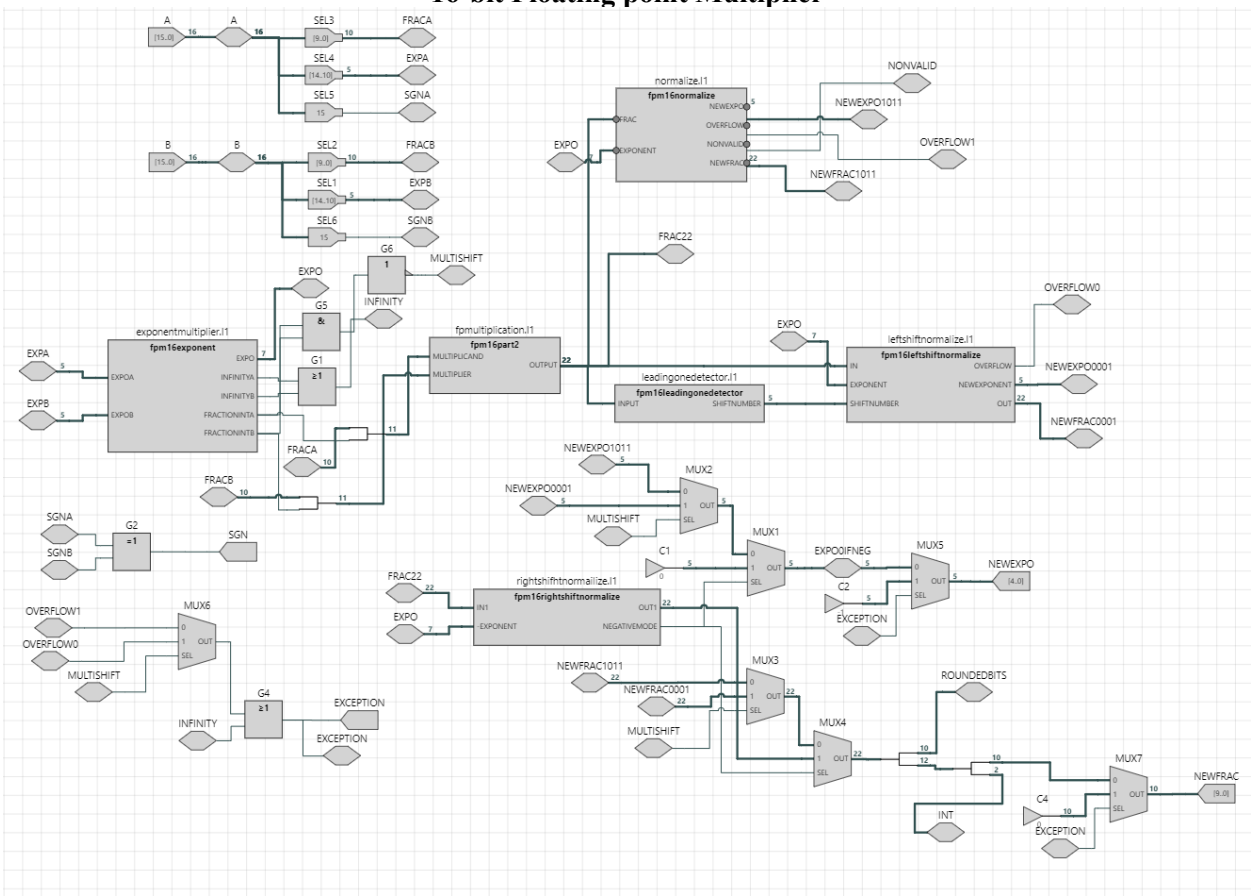
Floating point Adder part 1



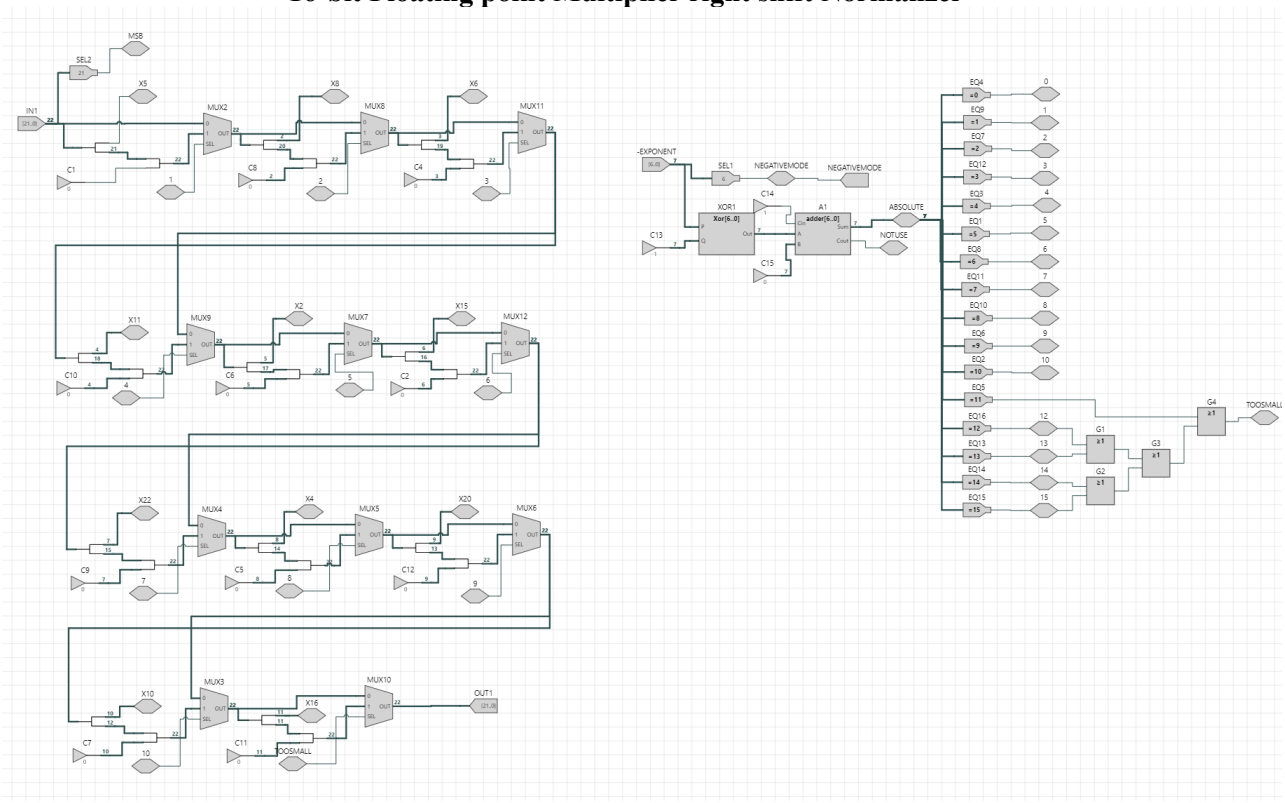
16-bit Floating point Adder express Exponent



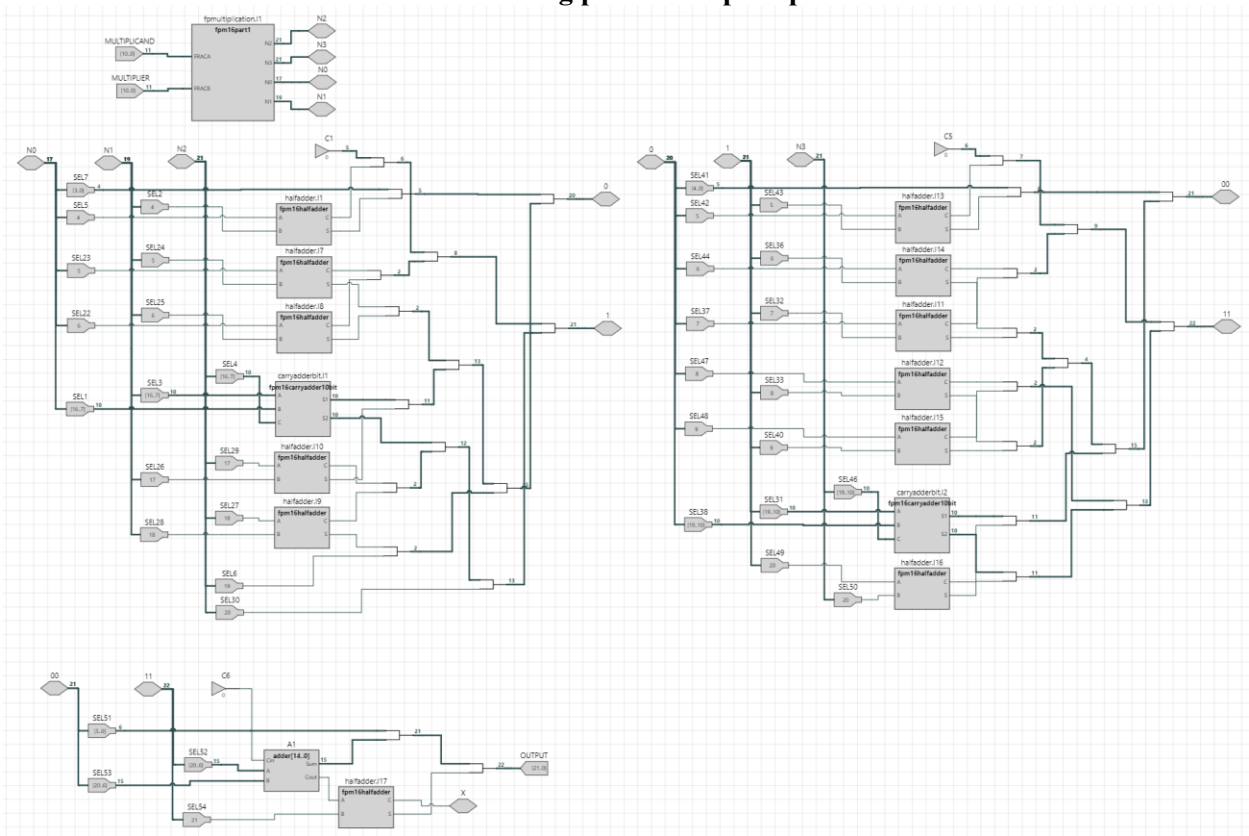
16-bit Floating point Multiplier



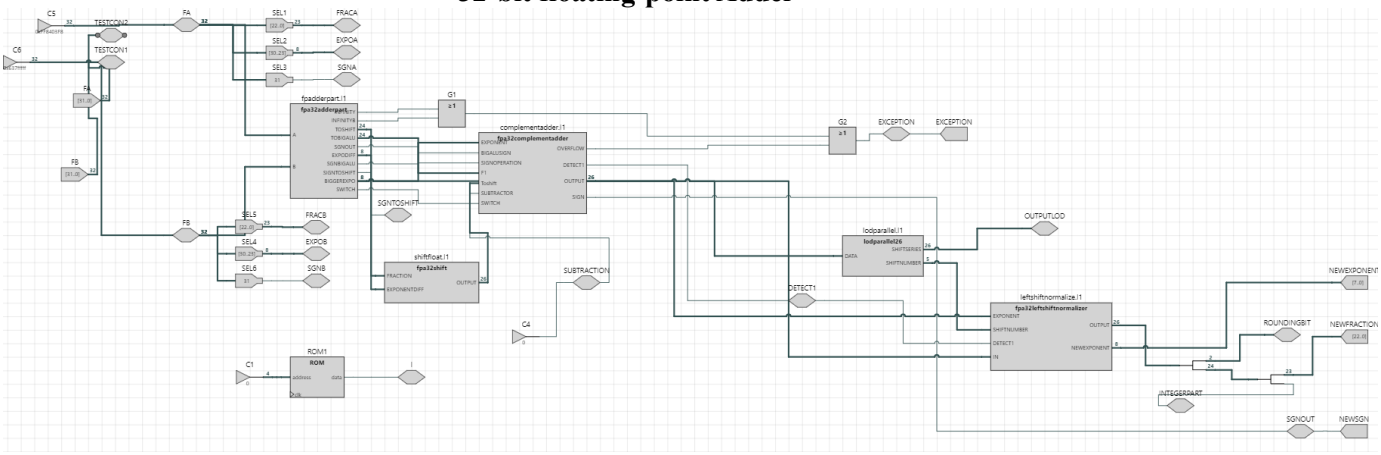
16-bit Floating point Multiplier right shift Normalizer



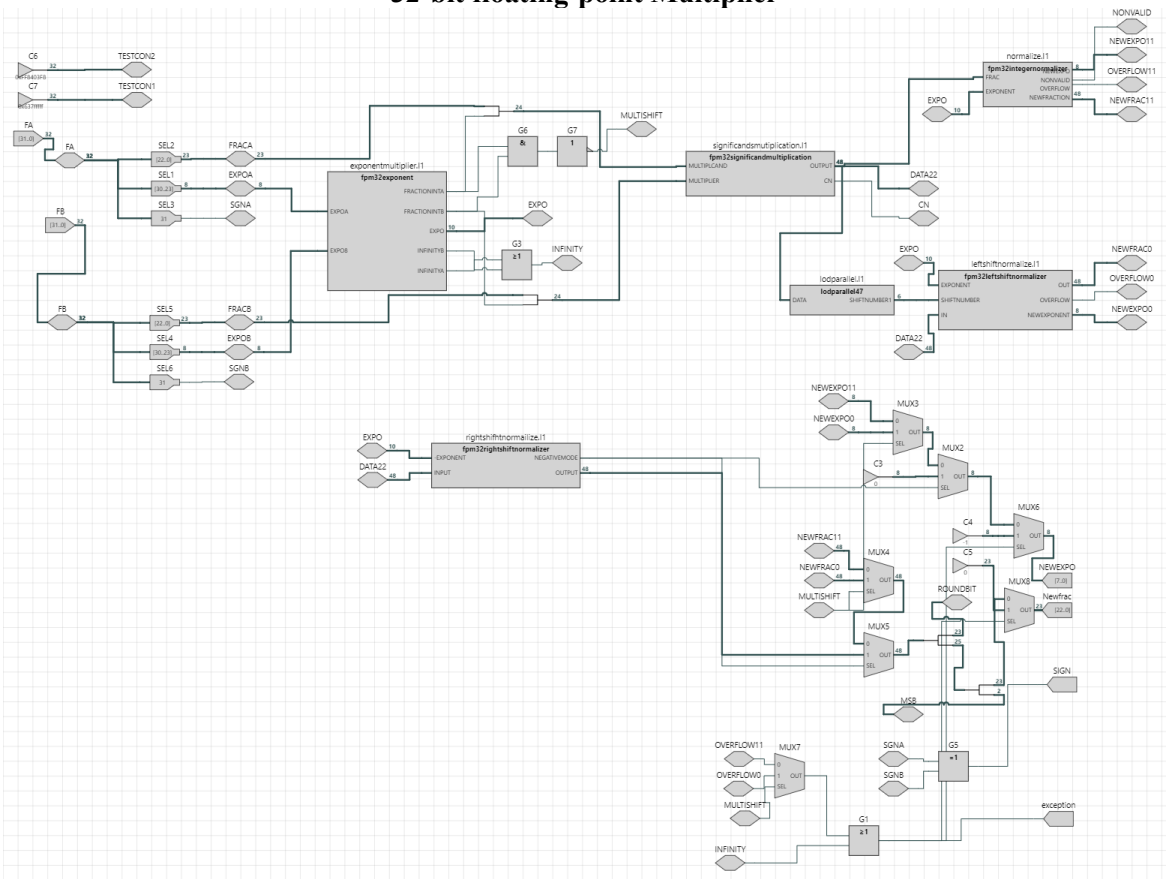
16-bit Floating point Multiplier part 2



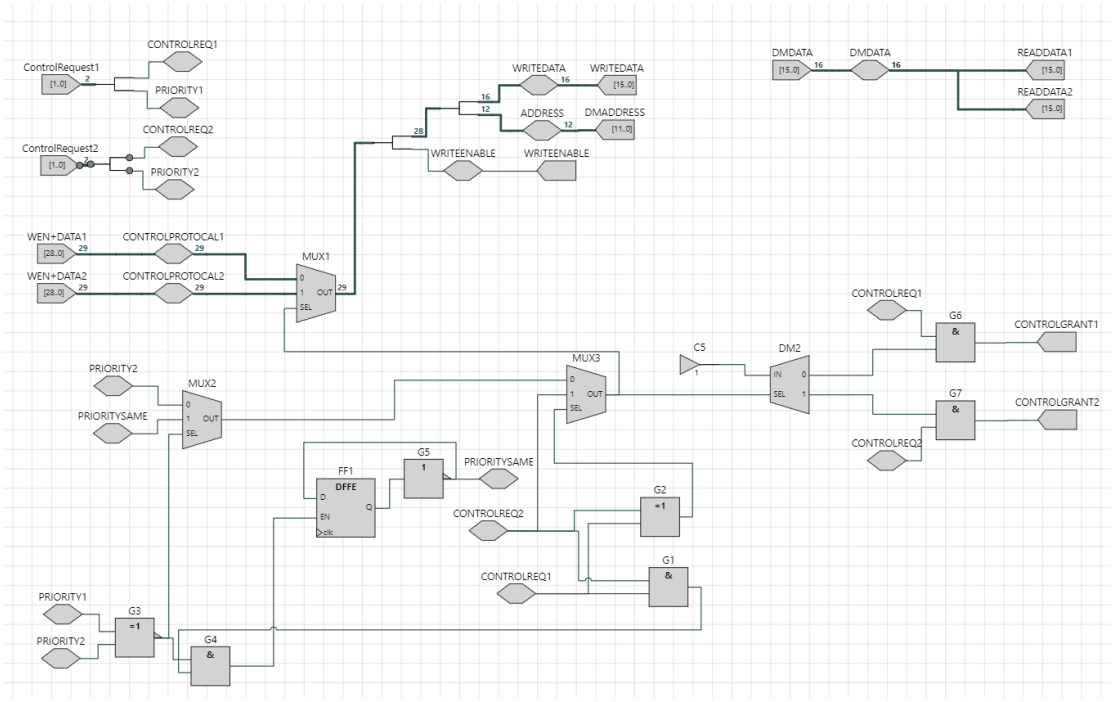
32-bit floating-point Adder



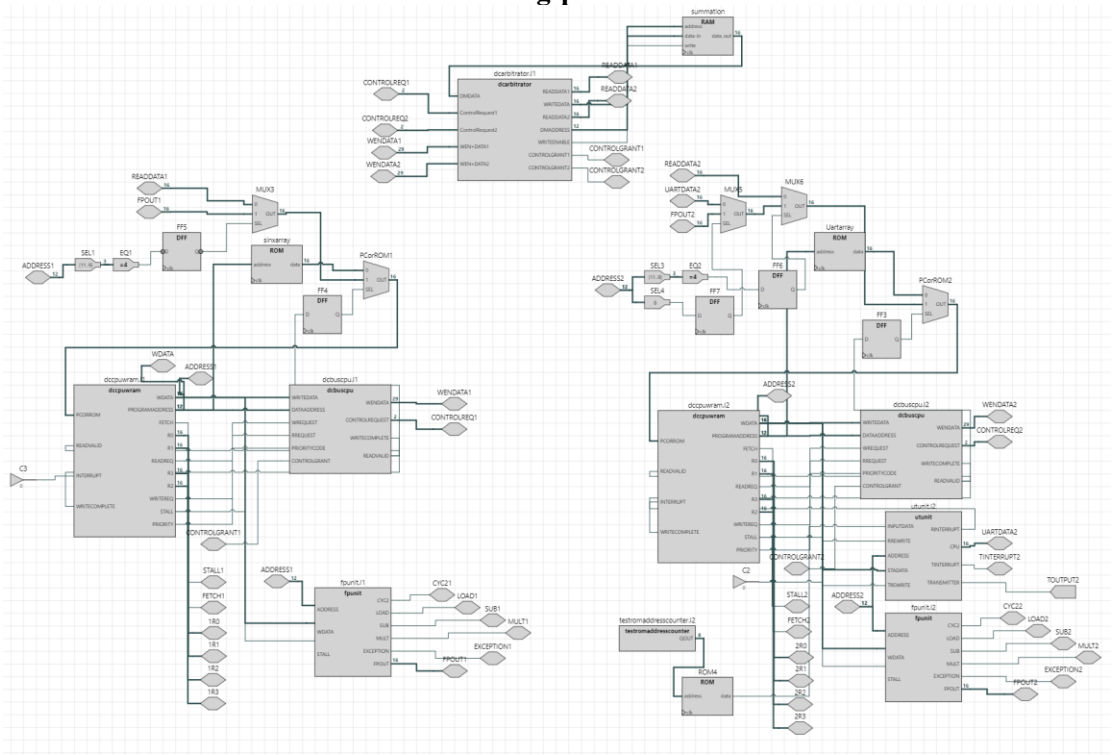
32-bit floating-point Multiplier



Dual Core Bus Arbitration



Final of floating-point arithmetic



Dual Core with floating-point