Arithmetical Logical Unit Design

Lazăr-Iulian Boariu
Maria-Alexia Crișan

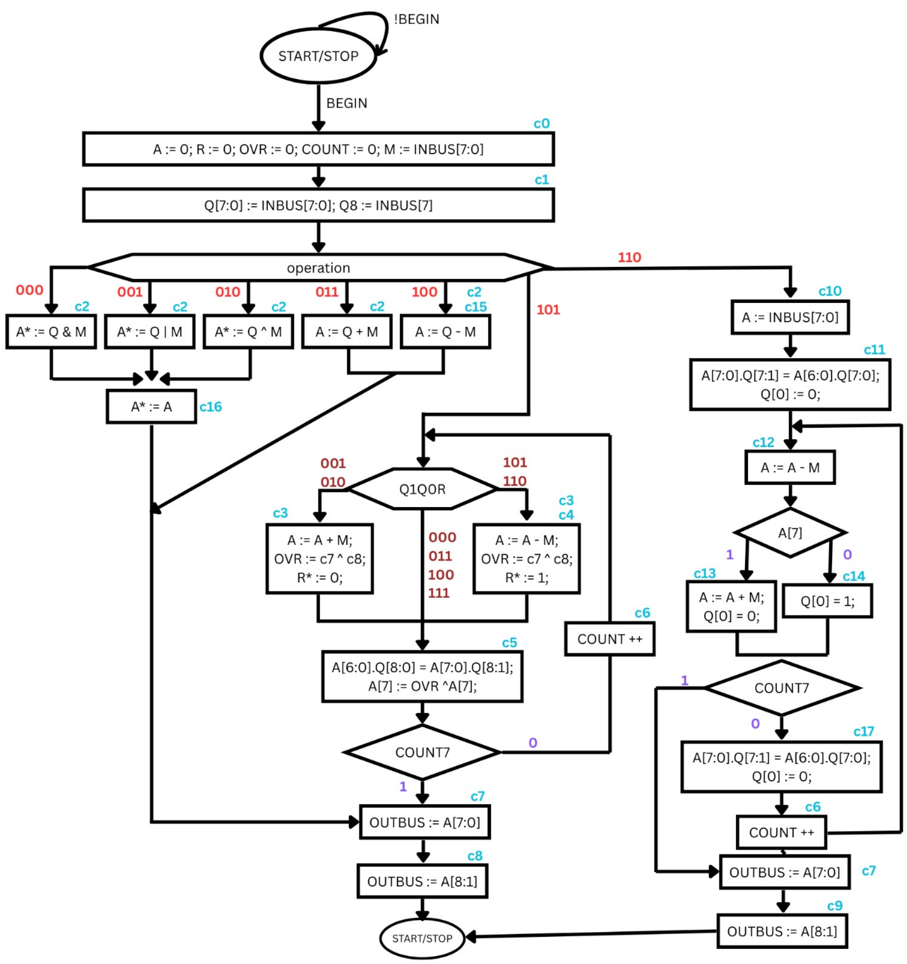
18th of April, 2025

Contents

1 Abstract

This project presents the design and implementation of an Arithmetic Logic Unit (ALU) capable of performing both logical and arithmetic operations on two 8-bit operands. The ALU supports basic logical operations including AND, OR, and XOR, as well as fundamental arithmetic operations such as addition, subtraction, multiplication, and division. The goal of this project is to show how these basic operations can be combined into a simple digital system. The project was developed as part of the Calculatoare Numerice course.

2 Finite State Diagram



The diagram illustrates the overall behavior of the ALU, detailing the steps involved in processing data and selecting the appropriate operation based on the provided operation code.

The design begins with the initialization of registers and variables, including the two 8-bit operands (Q and M), a remainder register (R), overflow flag (OVR), and a counter. The control logic then decodes the operation code to determine which operation to execute—logical (AND, OR, XOR) or arithmetic (addition, subtraction, multiplication, division). Each operation follows a specific path in the flowchart, with conditional branches guiding the transitions between processing stages.

Multiplication is implemented using the Modified Booth algorithm, which improves performance by reducing the number of partial products, and is triggered by specific control signals. Similarly, division is handled using the Restoring Division method, as shown on the right side of the diagram. This method involves conditional subtraction and shifting, iteratively refining the quotient and remainder.

The process continues until the counter reaches its limit, after which the final result is written to the output bus. The flowchart provides a structured overview of how the ALU manages complex operations in a controlled and sequential manner.

3 Algorithms Used

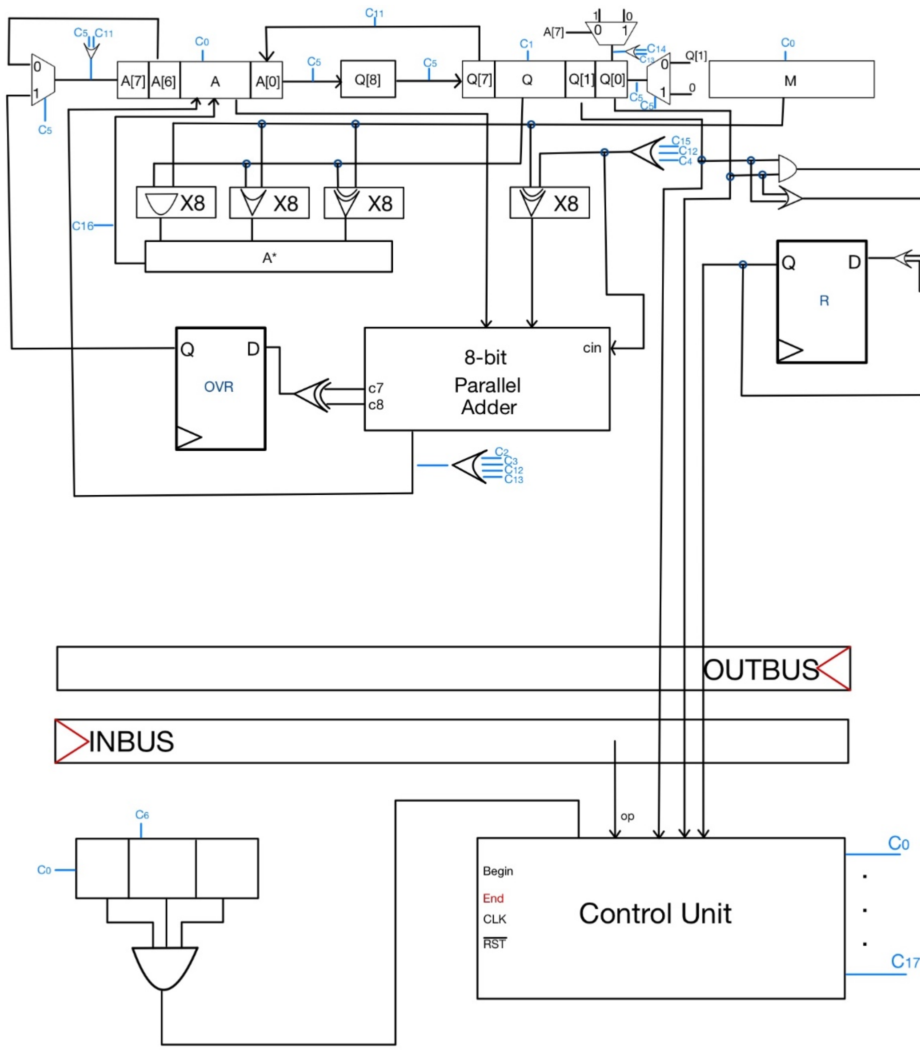
Modified Booth Multiplication

The Modified Booth algorithm is an efficient technique used for signed binary multiplication. It works by encoding the multiplier in groups of three bits, allowing the algorithm to reduce the number of partial products that need to be generated and added. Instead of performing a simple shift-and-add for every bit of the multiplier, Booth encoding identifies patterns and replaces them with fewer arithmetic operations, such as subtracting or addition. This method increases speed and reduces the hardware complexity of the multiplier, making it especially useful for larger word sizes or performance-critical designs.

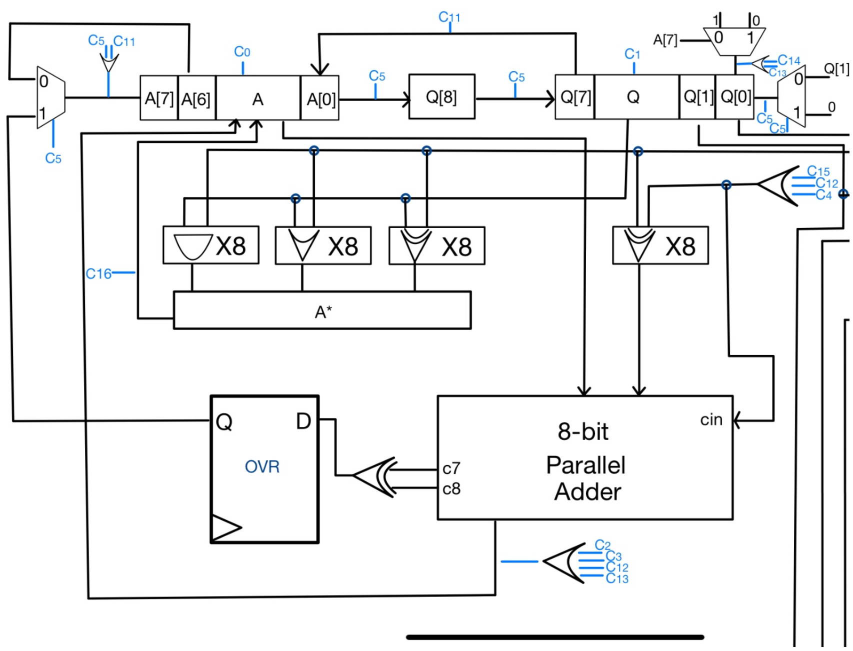
Restoring Division

Restoring division is a classic algorithm used for performing binary division by sequentially subtracting the divisor from the partial remainder. At each step, the dividend is shifted left, and the divisor is subtracted from the current remainder. If the result is positive, the subtraction was valid, and the quotient bit is set to 1. If the result is negative, the original remainder is restored by adding the divisor back, and the quotient bit is set to 0.

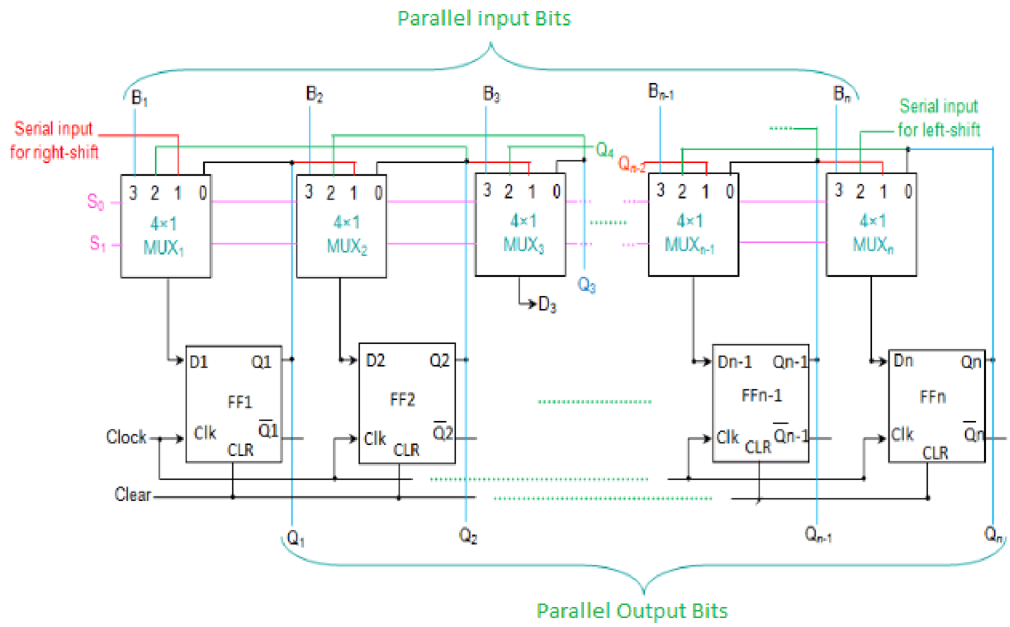
4 ALU Design



When looked at from the inside, the ALU exposes a series of low-level hardware components that work together, under the guidance of the control signals generated by the control unit, to implement each step of the chosen operation accordingly. The main elements at play here are the A, Q and M **Universal Shift Registers**, used for storing both the values of the incoming operands and the partial results and also shift them if needed, the **D-type Flip-Flops** used to store the two intermediate auxilliary states, the overflow flag (OVR), which tells us when the partial result of the Modified Booth algorithm needs to undergo a correction step due to the limitations of modulo-n arithmetic, and the run-of-bits flag, R, also useful in the context of Modified Booth, for taking advantage certain simplifications that help us skip through some of the arithmetic involved in doing multiplication, the **8-bit Paralled Adder** that computes partial sums or differences, a suite of **logic gates** for the simple logical operations that are multiplexed together and get computed all the time, **8-bit Input and Output Busses** used for delivering the data, both from the user to the hardware and vice versa, and the **Control Unit** that takes into consideration the current state of the ALU machine (including a **counter** for keeping the track of repetitive operations) in order to decide which signal to activate so that it helps with the transition into the next state.



There are three registers used to handle loading, dispatching and alteration of data. The Q and M registers store the two operands required for each operation, and in addition to them there's the A register that acts as an accumulator for partial results and also as an extension of the Q register, especially useful for the expansion of the bit domain of the first operand, from 8-bit to 16-bit, when performing division. A and Q hold the final results, at the end of the operation, and, depending on user selection, get concatenated to store the outcome of multiplication, are left separated for simple operations, case in which A stores the output value, or represent the remainder and the quotient respectively, after the end of division. There's and additional D-type flip-flop, Q8, used to expand the bit domain of Q when executing the Modified Booth algorithm. The values stored are in two's complement, a storing techinque for signed numbers that has the major advantage of preserving corectness of arithmetical operations, regardless of sign. These registers have both the capacity to be loaded into in parallel fashion, meaning all 8 bits get stored simultaneously, within the same clock cycle, and to perform both left and right shifting operations, required by the multiplication and division algorithms, meaning neighbouring bits get copied into one another, either from left to right, or right to left, while the leftmost or rightmost bit gets overwritten by an external incoming bit, chosen in function of the current operation and implicitly by the currently active control signal. This fundamental duality gives these hardware elements the name of universal shift registers. Their logic design was mainly inspired by the implementation provided below.



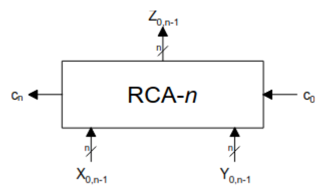
The modules situated right bellow the registers are simply an array of logic gates used to perform bit by bit logic operations between correspoding bits of the operands. The result is then decidedly stored inside the A* temporary register, through an implicit multiplexer, waiting to be stored inside A when the dedicated control signal is activated.

After that there is the Parallel Adder module. It is used for both subtraction and addition. It takes advantage of the following fact:

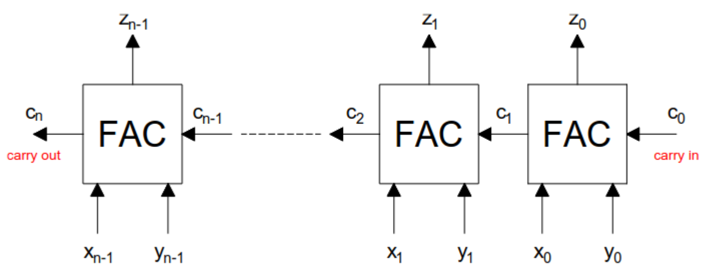
$$a \text{ xor } 0 = a$$

$$a \text{ xor } 1 = \overline{a}$$

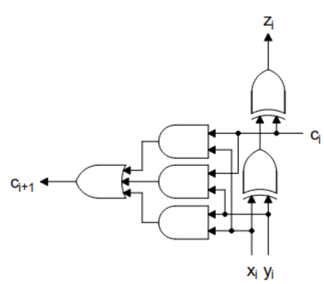
and uses it to negate the second operand and add then 1 to it through the initial carry, which is essentialy two's complementing the operand, equivalent to treating the expression $X - Y$ as $X + (-Y)$. It is then known that the state of the overflow flag can be computed as an exclusive or between the last and next to last carries of the addition operation. The adder chosen for this ALU is a Ripple Carry Adder (RCA):

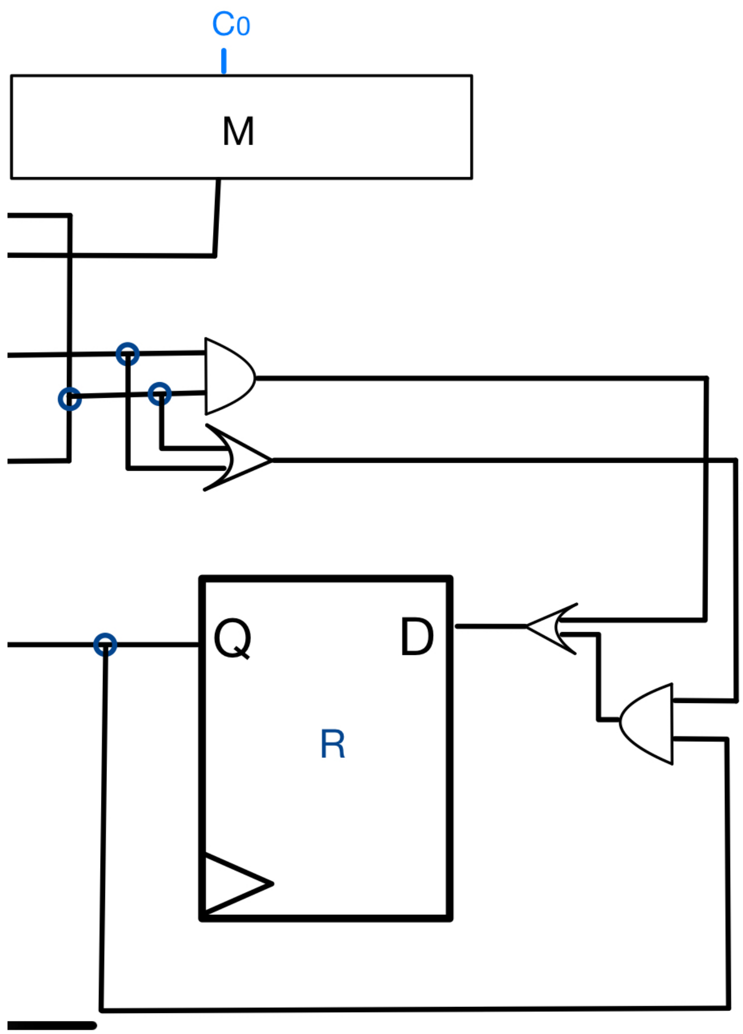


An n-bit RCA is an abstraction of n concatenated Full Adder Cells (FACs), each capable of performing 1-bit addition, including computing the carry.



The standard hardware implementation of a FAC is provided below.

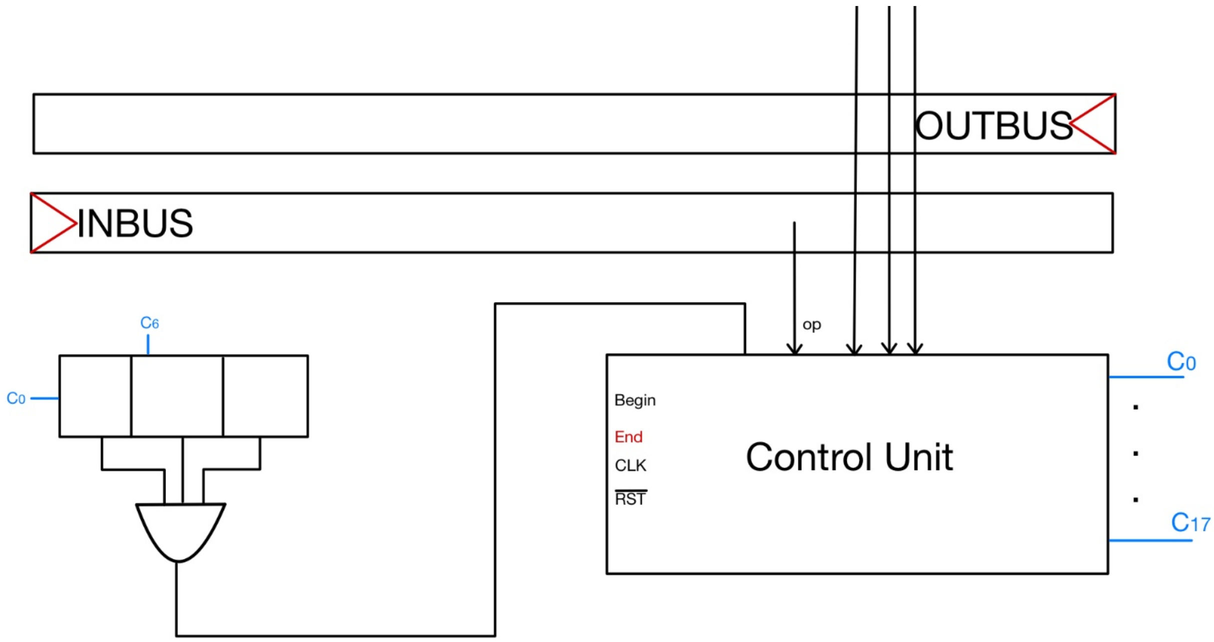




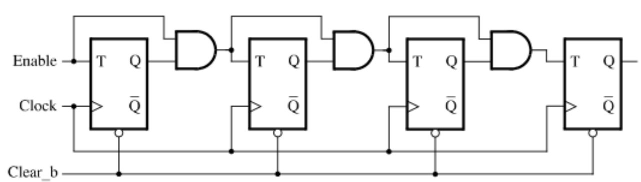
The R flag is determined by the formula

$$R_n = x_1 x_0 + R_{n-1}(x_1 + x_0),$$

which tells us whether we are in the middle, at the beginning, or at the end of a run of 1 bits, case in which we are going to perform the operation decided by the control unit, in accordance with the steps of the Modifiend Booth algorithm.



The counter is there to make sure we perform exactly 8 steps of the looping parts in the Modified Booth and Restoring Division algorithms, one for each bit of the operand. It simply counts from 0 to 7 and then it resets its value back to 0. The incrementation happens when its internal *count up* signal is activated (concretely, the *count up* signal is sent by the control unit, at the end of each repetitive step). It is implemented using T-type flip-flops, that toggle recursively, whenever the previous digits of the counter reach together the maximum value up to that order of magnitude.



The control unit is the single most complex and non-standard, in regards to implementation, component of the ALU system, and therefore is going to be discussed in great detail in its own separate section.

5 Control Unit

The control unit is a key component of the ALU system, responsible for coordinating the execution of operations by generating the necessary control signals at each step. In this project, the control logic is implemented using a sequence counter, which acts as a simple state machine. Each state corresponds to a specific stage in the execution process, such as loading operands, performing the selected operation, or storing the result. By advancing through its states in a predefined order, the sequence counter ensures that all operations are carried out in the correct sequence and with proper timing. This approach simplifies the control logic and makes the overall system easier to manage and debug.

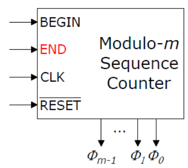
To implement the control logic, a Modulo-5-sequence counter was designed, which cycles through five distinct states corresponding to the stages of the ALU operation. This module serves as the core timing mechanism of the control unit. The system is composed of several interconnected submodules: Decoder-1-out-of-5, a Modulo-5-counter, and an SR flip-flop. Each component plays a specific role in generating and controlling the state transitions.

1. **Modulo-5-counter**
This module generates a repetitive sequence of five binary values (from 0 to 4) that represent the current phase. It increments its output on each clock cycle, looping back to zero after reaching four. The modulo-5 behavior ensures that the control unit progresses through exactly five phases before restarting the sequence.

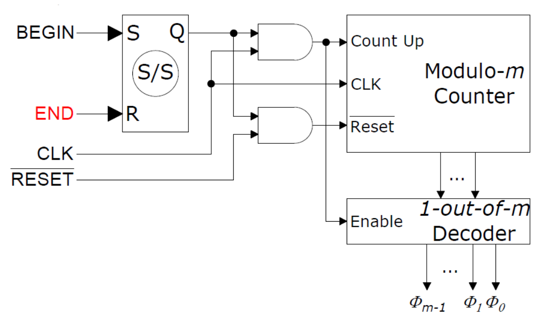
2. **Decoder-1-out-of-5**
The decoder takes the 3-bit output from the modulo-five counter and activates exactly one of its five outputs at a time. This ensures that only one phase signal is active during each step, allowing precise control over which control lines are enabled. Essentially, it translates the binary count into a one-hot encoding format, which is easier to use for controlling sequential logic.

3. **SR Flip-Flop**
The Start/Stop latch controls whether the sequence counter is active or halted. It enables or disables the advancement through the counting sequence based on external conditions or control inputs. When the latch is set,

the modulo-5 counter is allowed to advance on each clock cycle, progressing through the defined control states. When reset, the counter is held in place, effectively pausing the control unit. This mechanism provides a simple and effective way to synchronize the ALU's operation with the system's global control logic.



The picture illustrates the internal structure and signal connections within the Modulo-5-sequence counter used in the control unit. It shows how the Modulo-5-counter, 1-out-of-5 Decoder and the Start/Stop latch are connected to manage state transitions.



Each operation supported by the ALU has a specific set of control signals that are activated during certain steps of the control sequence. These signals coordinate actions such as loading operands, enabling the ALU, performing addition or subtraction, and storing the result. The list below outlines the control signals associated with each operation and indicates during which phase (based on the output of the Modulo-5 sequence counter) each signal is active:

Control Signals by Operation

1. AND / OR / XOR (Logical Operations)

Operation codes: AND(000), AND(001), XOR(010)

Cycle 0:

- Phase 0: c0 -> Load operand M
- Phase 1: c1 -> Load operand Q
- Phase 2: c2 -> Enable logic unit
- Phase 3: c16 -> Store result in A-star based of off op

Cycle 9:

- Phase 0: c7 -> Load result A
- Phase 1: c8 -> Load result Q

———-> Activate END signal

2. ADD / SUB (Arithmetic Operations)

Operation codes: ADD(011), SUB(100)

Cycle 0:

- Phase 0: c0 -> Load operand M
- Phase 1: c1 -> Load operand Q
- Phase 2: c2 -> Triggers addition, the result is stored in A
- Phase 2: c15 -> Triggers subtraction, the result is stored in A

Cycle 9:

- Phase 0: c7 -> Load result A
- Phase 1: c8 -> Load result Q

———> Activate END signal

3. MULTIPLICATION (Modified Booth)

Operation codes: MULTIPLICATION(101)

Cycle 0:

- Phase 0: c0 -> Load operand M and initialize A, COUNT, R, OVR
- Phase 1: c1 -> Load operand Q and Q8

Cycle 1 - 8:

- Phase 0: c3 -> Triggers addition for Q1Q0R in 001, 010, 101, 110, R* = 0, OVR is calculated
- Phase 0: c4 -> Additional signal for subtraction for Q1Q0R in 101, 110, R* = 1, OVR is calculated
- Phase 1: c5 -> Triggers right shift
- Phase 2: c6 -> COUNT is updated

Cycle 9:

- Phase 0: c7 -> Load result A
- Phase 0: c5 -> Triggers final right shift
- Phase 1: c8 -> Load result Q

———> Activate END signal

4. DIVISION (Restoring Division)

Operation codes: DIVISION(110)

Cycle 0:

- Phase 0: c0 -> Load operand M

- Phase 1: c1 -> Load operand Q
- Phase 2: c10 -> Load operand A (first half of the dividend)
- Phase 3: c11 -> Triggers initial left shift

Cycle 1 - 8:

- Phase 0: c12 -> Triggers subtraction
- Phase 1: c13 -> Triggers addition when A[7] = 1, updates Q[0] = 0
- Phase 1: c14 -> Updates Q[0] = 1
- Phase 2: c6 -> COUNT is updated
- Phase 3: c17 -> Triggers left shift

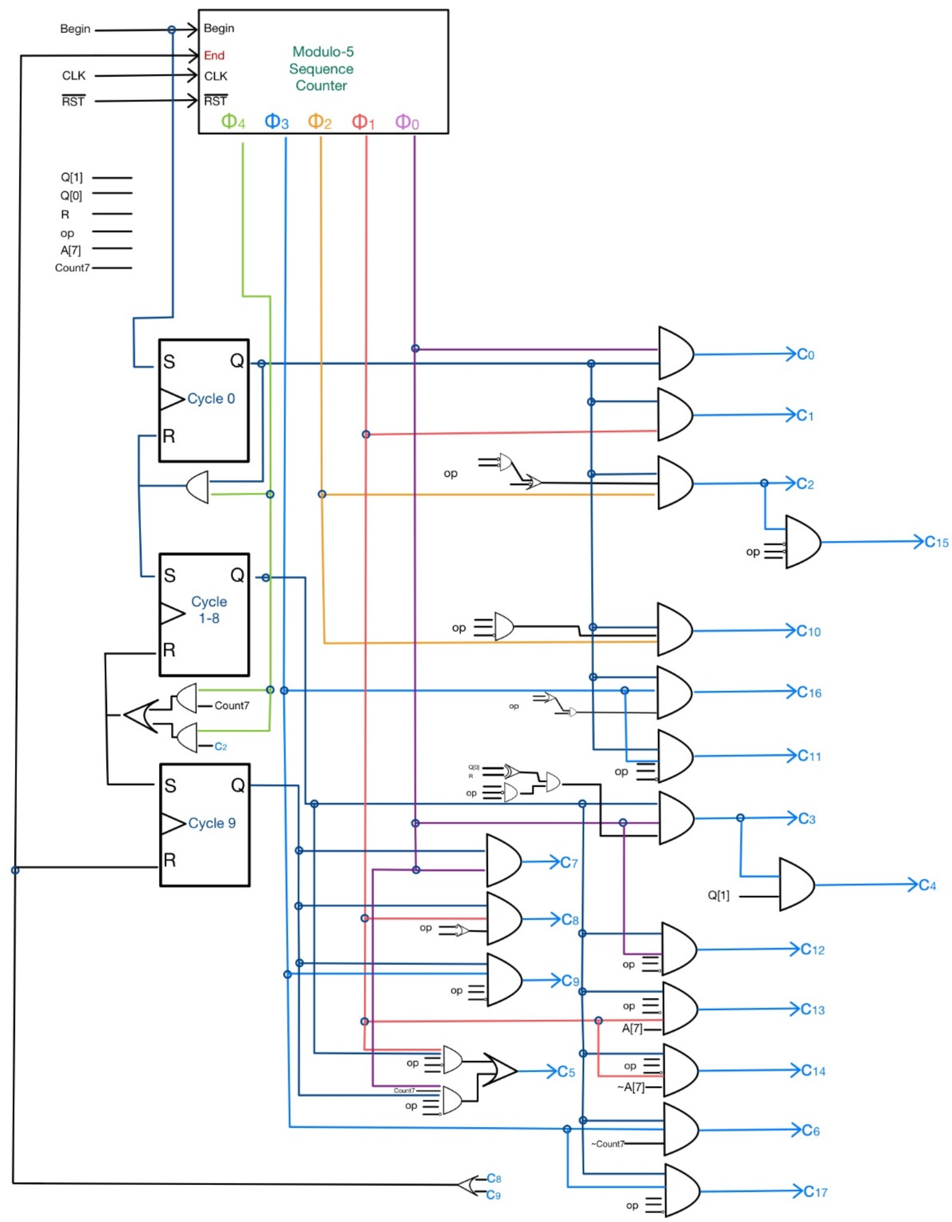
Cycle 9:

- Phase 0: c7 -> Load result A
- Phase 0: c9 -> Load result Q

———> Activate END signal

Observation: Phase 4 is consistently used as the transition point between cycles.

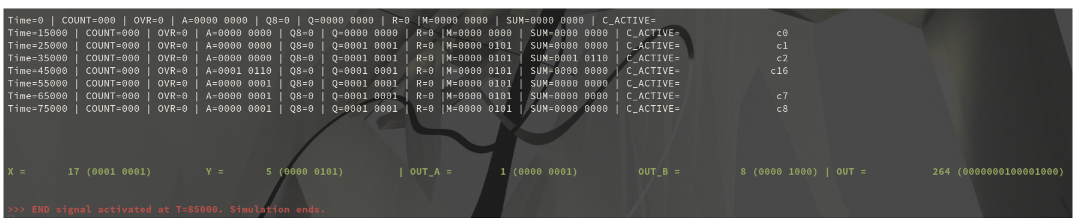
The control unit is built using a Modulo-5-Sequence-Counter and a set of three SR flip-flops, which together manage the execution flow of each operation. The sequence counter generates the control phases, cycling through 5 phases. At the same time, the SR flip-flops hold the information about which cycle is currently being executed. Depending on the combination of the current phase (given by the counter) and the active cycle (stored in the SR-FFs), the system activates specific control signals. This structure ensures that, for each operation type, only the required control lines are triggered at the correct moment during execution.


Begin
CLK
RST
Modulo-5 Sequence Counter
Begin
End
CLK
RST
Φ4 Φ3 Φ2 Φ1 Φ0
Q[1]
Q[0]
R
op
A[7]
Count7
Cycle 0
Cycle 1-8
Cycle 9
C0
C1
C2
C15
C10
C16
C11
C3
C4
C7
C8
C9
C12
C13
C14
C5
C6
C17
~A[7]
~Count7

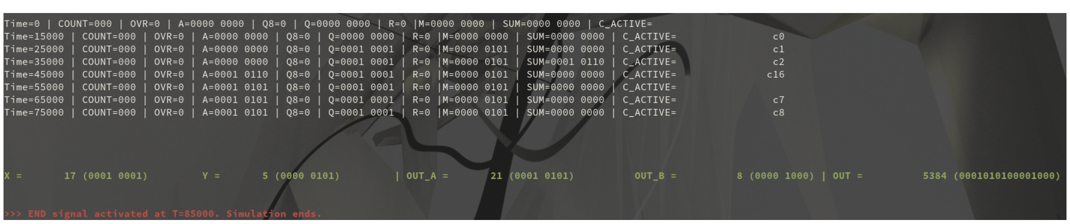
6 Results

To illustrate the functionality of the ALU, each supported operation is presented below along with its corresponding operation code (opcode) and a simulation result. These examples provide a visual and practical understanding of how the ALU processes 8-bit operands to perform logical and arithmetic computations.

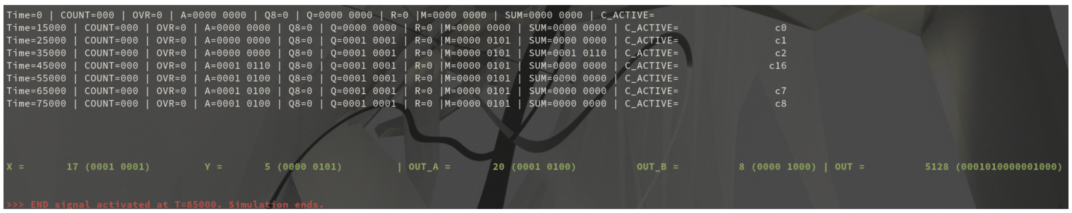
1. **AND Operation Opcode**: 000 The AND operation performs a bitwise conjunction between the two 8-bit operands. Each bit in the result is set to 1 only if the corresponding bits in both inputs are 1. The simulation output confirms this behavior, showing the correct result where only common bits are preserved. For example, if **X = 17 (0001 0001)** and **Y = 5 (0000 0101)**, the result of the operation is OUT_A = **1 (0000 0001)**, which is highlighted in green in the simulation image.



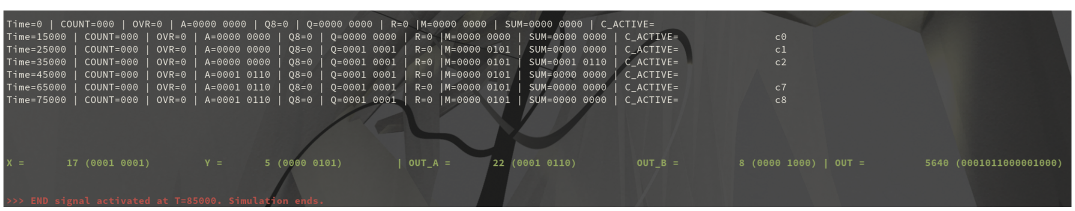
2. **OR Operation Opcode**: 001 The OR operation performs a bitwise disjunction between the operands. In the output, a bit is 1 if it is set in either of the input operands. The simulation result highlights how the operation combines the inputs, setting bits wherever at least one input has a 1. For example, if **X = 17 (0001 0001)** and **Y = 5 (0000 0101)**, the result of the operation is OUT_A = **21 (0001 0101)**, which is highlighted in green in the simulation image.



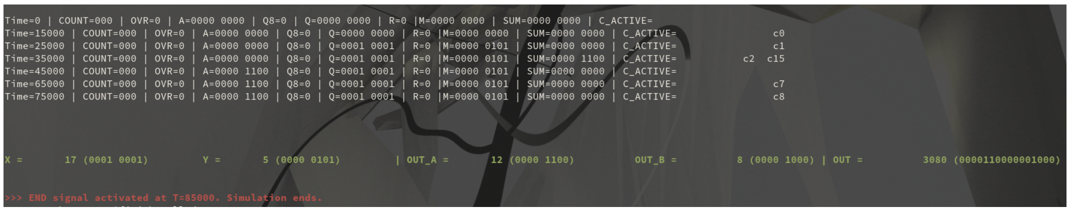
3. **XOR Operation Opcode**: 010 This operation executes a bitwise exclusive OR. A bit in the result is 1 if the corresponding bits in the operands are different. The output correctly reflects this rule, demonstrating how the XOR operation highlights differences between the inputs. For example, if **X = 17 (0001 0001)** and **Y = 5 (0000 0101)**, the result of the operation is OUT_A = **20 (0001 0100)**, which is highlighted in green in the simulation image.



4. **ADD Operation Opcode**: 011 The ADD operation computes the sum of the two 8-bit operands, with carry propagation across bits. The addition is performed using a Ripple Carry Adder (RCA), a simple yet effective design where each bit's carry is passed to the next stage sequentially. For example, if **X = 17 (0001 0001)** and **Y = 5 (0000 0101)**, the result of the operation is OUT_A = **22 (0001 0110)**, which is highlighted in green in the simulation image.



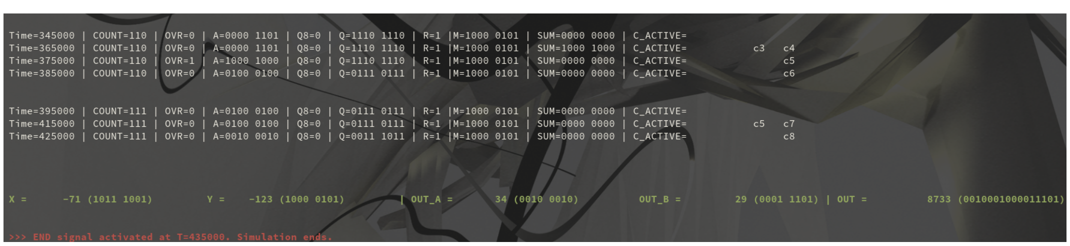
5. **SUB Operation Opcode**: 100 The SUB operation subtracts the second operand from the first using two's complement arithmetic. This operation also uses a Ripple Carry Adder (RCA), similar to the ADD operation. However, before entering the adder, the second operand is passed through an XOR gate to invert its bits, and the carry-in (Cin) of the adder is set to 1. For example, if **X = 17 (0001 0001)** and **Y = 5 (0000 0101)**, the result of the operation is OUT_A = **12 (0000 1100)**, which is highlighted in green in the simulation image.



```
Time=0 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0000 0000 | R=0 |M=0000 0000 | SUM=0000 0000 | C_ACTIVE=
Time=15000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0000 0000 | R=0 |M=0000 0000 | SUM=0000 0000 | C_ACTIVE=        c0
Time=25000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0001 0001 | R=0 |M=0000 0101 | SUM=0000 0000 | C_ACTIVE=        c1
Time=35000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0001 0001 | R=0 |M=0000 0101 | SUM=0000 1100 | C_ACTIVE=    c2  c15
Time=45000 | COUNT=000 | OVR=0 | A=0000 1100 | Q8=0 | Q=0001 0001 | R=0 |M=0000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=65000 | COUNT=000 | OVR=0 | A=0000 1100 | Q8=0 | Q=0001 0001 | R=0 |M=0000 0101 | SUM=0000 0000 | C_ACTIVE=        c7
Time=75000 | COUNT=000 | OVR=0 | A=0000 1100 | Q8=0 | Q=0001 0001 | R=0 |M=0000 0101 | SUM=0000 0000 | C_ACTIVE=        c8


X =       17 (0001 0001)         Y =       5 (0000 0101)         | OUT_A =       12 (0000 1100)         OUT_B =         8 (0000 1000) | OUT =         3080 (0000110000001000)

>>> END signal activated at T=85000. Simulation ends.
```

6. **MUL Operation (Modified Booth) Opcode**: 101 This operation performs multiplication using the Modified Booth algorithm, efficiently handling signed numbers. The output from the simulation confirms the algorithm's correct implementation, showing a 16-bit result as the product of two 8-bit signed numbers. For example, if **X = -71 (1011 1001)** and **Y = -123 (1000 0101)**, the result of the operation is **OUT = 8733 (0010 0010 0001 1101)**, which is highlighted in green in the simulation image.

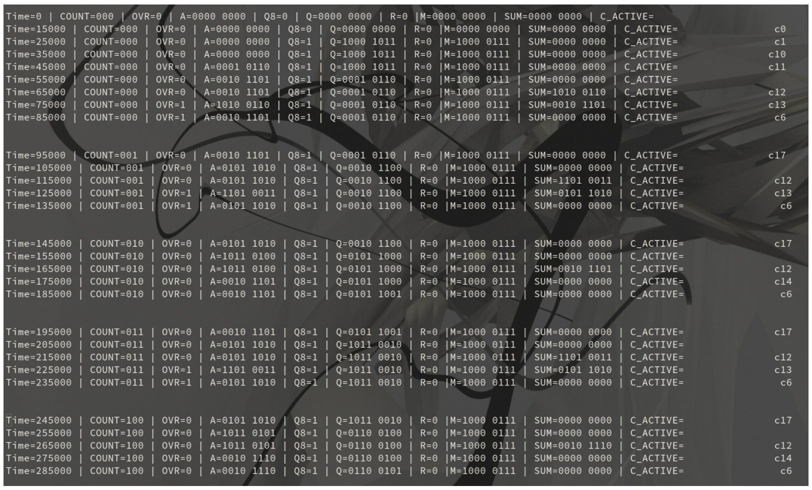
```
Time=0 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0000 0000 | R=0 |M=0000 0000 | SUM=0000 0000 | C_ACTIVE=
Time=15000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0000 0000 | R=0 |M=0000 0000 | SUM=0000 0000 | C_ACTIVE=        c0
Time=25000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=1 | Q=1011 1001 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c1
Time=35000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=1 | Q=1011 1001 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=65000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=1 | Q=1011 1001 | R=0 |M=1000 0101 | SUM=1000 0101 | C_ACTIVE=        c3
Time=75000 | COUNT=000 | OVR=0 | A=1000 0101 | Q8=1 | Q=1011 1001 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c5
Time=85000 | COUNT=000 | OVR=0 | A=1100 0010 | Q8=1 | Q=1101 1100 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c6

Time=95000 | COUNT=001 | OVR=0 | A=1100 0010 | Q8=1 | Q=1101 1100 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=125000 | COUNT=001 | OVR=0 | A=1100 0010 | Q8=1 | Q=1101 1100 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c5
Time=135000 | COUNT=001 | OVR=0 | A=1110 0001 | Q8=0 | Q=1110 1110 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c6

Time=145000 | COUNT=010 | OVR=0 | A=1110 0001 | Q8=0 | Q=1110 1110 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=175000 | COUNT=010 | OVR=0 | A=1110 0001 | Q8=0 | Q=1110 1110 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c5
Time=185000 | COUNT=010 | OVR=0 | A=1111 0000 | Q8=1 | Q=0111 0111 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c6

Time=195000 | COUNT=011 | OVR=0 | A=1111 0000 | Q8=1 | Q=0111 0111 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=215000 | COUNT=011 | OVR=0 | A=1111 0000 | Q8=1 | Q=0111 0111 | R=0 |M=1000 0101 | SUM=0110 1011 | C_ACTIVE=    c3  c4
Time=225000 | COUNT=011 | OVR=0 | A=0110 1011 | Q8=1 | Q=0111 0111 | R=0 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c5
Time=235000 | COUNT=011 | OVR=0 | A=0011 0101 | Q8=1 | Q=1011 1011 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c6

Time=245000 | COUNT=100 | OVR=0 | A=0011 0101 | Q8=1 | Q=1011 1011 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=275000 | COUNT=100 | OVR=0 | A=0011 0101 | Q8=1 | Q=1011 1011 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c5
Time=285000 | COUNT=100 | OVR=0 | A=0001 1010 | Q8=1 | Q=1101 1101 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c6

Time=295000 | COUNT=101 | OVR=0 | A=0001 1010 | Q8=1 | Q=1101 1101 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=325000 | COUNT=101 | OVR=0 | A=0001 1010 | Q8=1 | Q=1101 1101 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c5
Time=335000 | COUNT=101 | OVR=0 | A=0000 1101 | Q8=0 | Q=1110 1110 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c6
```



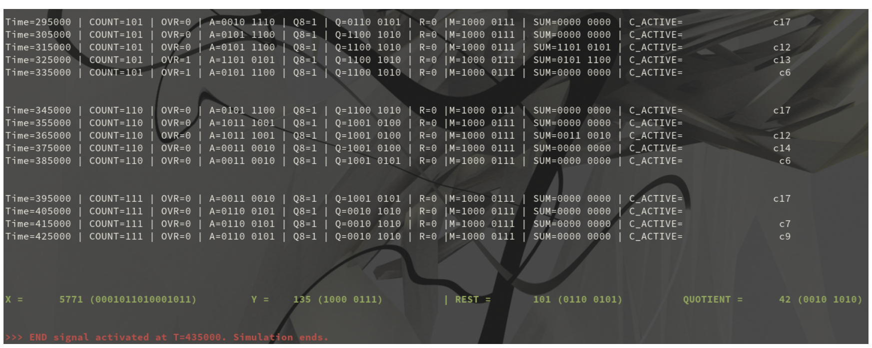
```
Time=345000 | COUNT=110 | OVR=0 | A=0000 1101 | Q8=0 | Q=1110 1110 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=365000 | COUNT=110 | OVR=0 | A=0000 1101 | Q8=0 | Q=1110 1110 | R=1 |M=1000 0101 | SUM=1000 1000 | C_ACTIVE=    c3  c4
Time=375000 | COUNT=110 | OVR=1 | A=1000 1000 | Q8=0 | Q=1110 1110 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c5
Time=385000 | COUNT=110 | OVR=0 | A=0100 0100 | Q8=0 | Q=0111 0111 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c6

Time=395000 | COUNT=111 | OVR=0 | A=0100 0100 | Q8=0 | Q=0111 0111 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=
Time=415000 | COUNT=111 | OVR=0 | A=0100 0100 | Q8=0 | Q=0111 0111 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=    c5  c7
Time=425000 | COUNT=111 | OVR=0 | A=0010 0010 | Q8=0 | Q=0011 1011 | R=1 |M=1000 0101 | SUM=0000 0000 | C_ACTIVE=        c8


X =      -71 (1011 1001)         Y =    -123 (1000 0101)         | OUT_A =       34 (0010 0010)         OUT_B =        29 (0001 1101) | OUT =         8733 (0010001000011101)

>>> END signal activated at T=435000. Simulation ends.
```

7. **DIV Operation (Restoring Division) Opcode**: 110 The DIV operation carries out division using the restoring division method. The simulation

displays the correct quotient and remainder, split across separate registers or outputs. For example, if **X = 5771 (0001 0110 1000 1011)** and **Y = 135 (1000 0111)**, the result of the operation is **REMAINDER = 101 (0110 0101)** and **QUOTIENT = 42 (0010 1010)**, which is highlighted in green in the simulation image.



```
Time=0 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0000 0000 | R=0 |M=0000 0000 | SUM=0000 0000 | C_ACTIVE=
Time=15000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=0 | Q=0000 0000 | R=0 |M=0000 0000 | SUM=0000 0000 | C_ACTIVE=        c0
Time=25000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=1 | Q=1000 1011 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c1
Time=35000 | COUNT=000 | OVR=0 | A=0000 0000 | Q8=1 | Q=1000 1011 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c10
Time=45000 | COUNT=000 | OVR=0 | A=0001 0110 | Q8=1 | Q=1000 1011 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c11
Time=55000 | COUNT=000 | OVR=0 | A=0010 1101 | Q8=1 | Q=0001 0110 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=65000 | COUNT=000 | OVR=0 | A=0010 1101 | Q8=1 | Q=0001 0110 | R=0 |M=1000 0111 | SUM=1010 0110 | C_ACTIVE=        c12
Time=75000 | COUNT=000 | OVR=1 | A=1010 0110 | Q8=1 | Q=0001 0110 | R=0 |M=1000 0111 | SUM=0010 1101 | C_ACTIVE=        c13
Time=85000 | COUNT=000 | OVR=1 | A=0010 1101 | Q8=1 | Q=0001 0110 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c6


Time=95000 | COUNT=001 | OVR=0 | A=0010 1101 | Q8=1 | Q=0001 0110 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c17
Time=105000 | COUNT=001 | OVR=0 | A=0101 1010 | Q8=1 | Q=0010 1100 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=115000 | COUNT=001 | OVR=0 | A=0101 1010 | Q8=1 | Q=0010 1100 | R=0 |M=1000 0111 | SUM=1101 0011 | C_ACTIVE=        c12
Time=125000 | COUNT=001 | OVR=1 | A=1101 0011 | Q8=1 | Q=0010 1100 | R=0 |M=1000 0111 | SUM=0101 1010 | C_ACTIVE=        c13
Time=135000 | COUNT=001 | OVR=1 | A=0101 1010 | Q8=1 | Q=0010 1100 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c6


Time=145000 | COUNT=010 | OVR=0 | A=0101 1010 | Q8=1 | Q=0010 1100 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c17
Time=155000 | COUNT=010 | OVR=0 | A=1011 0100 | Q8=1 | Q=0101 1000 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=165000 | COUNT=010 | OVR=0 | A=1011 0100 | Q8=1 | Q=0101 1000 | R=0 |M=1000 0111 | SUM=0010 1101 | C_ACTIVE=        c12
Time=175000 | COUNT=010 | OVR=0 | A=0010 1101 | Q8=1 | Q=0101 1000 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c14
Time=185000 | COUNT=010 | OVR=0 | A=0010 1101 | Q8=1 | Q=0101 1001 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c6


Time=195000 | COUNT=011 | OVR=0 | A=0010 1101 | Q8=1 | Q=0101 1001 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c17
Time=205000 | COUNT=011 | OVR=0 | A=0101 1010 | Q8=1 | Q=1011 0010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=215000 | COUNT=011 | OVR=0 | A=0101 1010 | Q8=1 | Q=1011 0010 | R=0 |M=1000 0111 | SUM=1101 0011 | C_ACTIVE=        c12
Time=225000 | COUNT=011 | OVR=1 | A=1101 0011 | Q8=1 | Q=1011 0010 | R=0 |M=1000 0111 | SUM=0101 1010 | C_ACTIVE=        c13
Time=235000 | COUNT=011 | OVR=1 | A=0101 1010 | Q8=1 | Q=1011 0010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c6


Time=245000 | COUNT=100 | OVR=0 | A=0101 1010 | Q8=1 | Q=1011 0010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c17
Time=255000 | COUNT=100 | OVR=0 | A=1011 0101 | Q8=1 | Q=0110 0100 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=265000 | COUNT=100 | OVR=0 | A=1011 0101 | Q8=1 | Q=0110 0100 | R=0 |M=1000 0111 | SUM=0010 1110 | C_ACTIVE=        c12
Time=275000 | COUNT=100 | OVR=0 | A=0010 1110 | Q8=1 | Q=0110 0100 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c14
Time=285000 | COUNT=100 | OVR=0 | A=0010 1110 | Q8=1 | Q=0110 0101 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c6
```



```
Time=295000 | COUNT=101 | OVR=0 | A=0010 1110 | Q8=1 | Q=0110 0101 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c17
Time=305000 | COUNT=101 | OVR=0 | A=0101 1100 | Q8=1 | Q=1100 1010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=315000 | COUNT=101 | OVR=0 | A=0101 1100 | Q8=1 | Q=1100 1010 | R=0 |M=1000 0111 | SUM=1101 0101 | C_ACTIVE=        c12
Time=325000 | COUNT=101 | OVR=1 | A=1101 0101 | Q8=1 | Q=1100 1010 | R=0 |M=1000 0111 | SUM=0101 1100 | C_ACTIVE=        c13
Time=335000 | COUNT=101 | OVR=1 | A=0101 1100 | Q8=1 | Q=1100 1010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c6


Time=345000 | COUNT=110 | OVR=0 | A=0101 1100 | Q8=1 | Q=1100 1010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c17
Time=355000 | COUNT=110 | OVR=0 | A=1011 1001 | Q8=1 | Q=1001 0100 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=365000 | COUNT=110 | OVR=0 | A=1011 1001 | Q8=1 | Q=1001 0100 | R=0 |M=1000 0111 | SUM=0011 0010 | C_ACTIVE=        c12
Time=375000 | COUNT=110 | OVR=0 | A=0011 0010 | Q8=1 | Q=1001 0100 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c14
Time=385000 | COUNT=110 | OVR=0 | A=0011 0010 | Q8=1 | Q=1001 0101 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c6


Time=395000 | COUNT=111 | OVR=0 | A=0011 0010 | Q8=1 | Q=1001 0101 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c17
Time=405000 | COUNT=111 | OVR=0 | A=0110 0101 | Q8=1 | Q=0010 1010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=
Time=415000 | COUNT=111 | OVR=0 | A=0110 0101 | Q8=1 | Q=0010 1010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c7
Time=425000 | COUNT=111 | OVR=0 | A=0110 0101 | Q8=1 | Q=0010 1010 | R=0 |M=1000 0111 | SUM=0000 0000 | C_ACTIVE=        c9



X =      5771 (0001011010001011)        Y =    135 (1000 0111)         | REST =       101 (0110 0101)          QUOTIENT =      42 (0010 1010)

>>> END signal activated at T=435000. Simulation ends.
```

7 Bibliography

Tools Used:

EDAPlayground – utilized for writing, testing, and simulating the code in a web-based environment.

Icarus Verilog (iverilog) – used as the underlying simulation engine for compiling and running Verilog code on a local machine.

Canva – used to create clear and visually structured diagrams that support the explanation of the design flow.

ChatGPT – assisted in drafting and refining the written content to ensure clarity and coherence.

Bibliography:

Oprițoiu Flavius, (2025) Computer Architecture, Polytechnic University of Timișoara
https://www.geeksforgeeks.org/universal-shift-register-in-digital-logic/