TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Electrical and Computer Engineering Institute of Circuits and Systems

Circuit and System Design: CORDIC

Yixin Yuan

Matriculation number: 4975933

15th March 2024

Supervisor

Dr.-Ing. Sebastian Höppner

Supervising professor

Prof. Dr.-Ing. habil. Christian Mayr

Statement of authorship

I hereby certify that I have authored this document entitled *Circuit and System Design: CORDIC* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 15th March 2024

Yixin Yuan

Contents

1. Introduction

In the evolving landscape of digital signal processing, the implementation of efficient algorithms within application-specific integrated circuits (ASICs) has become a pivotal area of research and development. This paper presents the design and realization of an ASIC tailored for the CORDIC (Coordinate Rotation Digital Computer) algorithm, undertaken as a part of the "Circuit and System Design" module at TU Dresden. The CORDIC algorithm stands out due to its versatility in computing a wide array of mathematical functions, including trigonometric, logarithmic, and exponential functions, thereby serving as a fundamental building block in numerous digital signal processing (DSP) applications.

The design journey embarks within the CADENCE DESIGN FRAMEWORK 2 environment, a choice driven by its comprehensive suite of tools catering to intricate ASIC design processes. The selection of the CORDIC algorithm is motivated by its unique architecture that adeptly balances computational efficiency with hardware simplicity, making it an ideal candidate for ASIC implementation. Unlike conventional numerical algorithms that may not cater to the periodic nature of input and output data typical in DSP applications, the CORDIC algorithm inherently accommodates this periodicity, making it exceptionally suited for real-time signal processing tasks.

In Chapter 2, I will introduce what the CORDIC algorithm is, illustrate how the algorithm iterates with examples, explain the data formats used by the algorithm, its precision, and memory usage. In addition, I will discuss the limitations of the CORDIC algorithm.

In Chapter 3, I will delve into the process of algorithm design, including the data flow graph, the architecture of the data path, register transfer sequences, FSM (Finite State Machine) state graph, and state encoding.

In Chapter 4, I will explain how the Datapath is implemented and integrate the three circuit components—FSM, Control logic, and Datapath. And I will test the performance of both the FSM and the overall circuit.

This report primarily references the "Anleitung für das Praktikum Schaltkreis- und Systementwurf, Rechnergestützter Schaltkreisentwurf [1]" and the theoretical descriptions from Wikipedia [2]. All source code is placed in the appendix.

2. Description of the algorithm

2.1. Algorithm

The CORDIC (Coordinate Rotation Digital Computer) algorithm is a simple yet powerful technique to compute trigonometric functions, hyperbolic functions, multiplication, division, and square roots, among others, using only shift and add operations. Here's a step-by-step explanation of the CORDIC algorithm, focusing on its application to calculate the angle of a point in Cartesian coordinates:

1. **Initialization:** Initialize the vector (X_0, Y_0) that you want to rotate, where X_0 and Y_0 are the Cartesian coordinates of the point. Also, initialize the angle accumulator Z_0 to 0. The goal is to rotate this vector to the x-axis and determine the angle of rotation, which corresponds to the angle of the original point.

2. **Pre-calculation of Arctangent Values:** Pre-calculate and store $arctan(2^{-i})$ for i ranging from 0 to a predetermined number of iterations. These values are used in each iteration to adjust the rotation angle. The arctangent values are scaled appropriately to match the desired angle representation.

3. **Iterative Rotation:**

   1) Determine the direction of rotation (d_i) based on the sign of Y_i. If Y_i is positive, rotate clockwise; otherwise, rotate counterclockwise.

   2) Calculate the new values of X_{i+1}, Y_{i+1}, and Z_{i+1} using the following formulas:

$$\begin{aligned} X_{i+1} &= X_i - d_i \cdot Y_i \cdot 2^{-i} \\ Y_{i+1} &= Y_i + d_i \cdot X_i \cdot 2^{-i} \\ Z_{i+1} &= Z_i - d_i \cdot arctan(2^{-i}) \end{aligned} \tag{2.1}$$

4. **Output:** The final value of Z represents the calculated angle. It can be converted to degrees or radians as needed and then used for further processing or displayed as the result.

2.2. Example

In the demonstration of the CORDIC algorithm, we consider an initial vector represented by the point (10,18) in Cartesian coordinates. The objective is to iteratively rotate this vector towards

the x-axis and compute the corresponding angle. The process unfolds over several iterations, with the first four detailed as follows:

Initial Configuration The vector is initialized with $X_0 = 10$ and $Y_0 = 18$, and the angle accumulator $Z_0 = 0$. The algorithm proceeds to rotate this vector by a series of predefined angles, aiming to align it with the x-axis.

Iteration 1 The rotation direction for the first iteration, denoted d_0, is determined by the sign of Y_0. Given $Y_0 > 0$, d_0 is set to -1, The vector components are updated as follows:

$$\begin{aligned} X_1 &= X_0 - d_0 \cdot Y_0 \cdot 2^{-0} = 28, \\ Y_1 &= Y_0 + d_0 \cdot X_0 \cdot 2^{-0} = 8, \\ Z_1 &= Z_0 - d_0 \cdot \arctan(2^{-0}) = 45°. \end{aligned}$$

Iteration 2 Similarly, for the second iteration with $Y_1 > 0$, $d_1 = -1$. The updates are:

$$\begin{aligned} X_2 &= X_1 - d_1 \cdot Y_1 \cdot 2^{-1} = 32, \\ Y_2 &= Y_1 + d_1 \cdot X_1 \cdot 2^{-1} = -6, \\ Z_2 &= Z_1 - d_1 \cdot \arctan(2^{-1}) = 71.57°. \end{aligned}$$

Iteration 3 For the third iteration, $Y_2 < 0$ leads to $d_2 = +1$, yielding:

$$\begin{aligned} X_3 &= X_2 - d_2 \cdot Y_2 \cdot 2^{-2} = 34.5, \\ Y_3 &= Y_2 + d_2 \cdot X_2 \cdot 2^{-2} = 2, \\ Z_3 &= Z_2 - d_2 \cdot \arctan(2^{-2}) = 57.53°. \end{aligned}$$

Iteration 4 Continuing the pattern, the fourth iteration with $Y_3 > 0$ sets $d_3 = -1$, resulting in:

$$\begin{aligned} X_4 &= X_3 - d_3 \cdot Y_3 \cdot 2^{-3} = 35, \\ Y_4 &= Y_3 + d_3 \cdot X_3 \cdot 2^{-3} = -2.3125, \\ Z_4 &= Z_3 - d_3 \cdot \arctan(2^{-3}) = 64.65°. \end{aligned}$$

Through this iterative process, the CORDIC algorithm effectively rotates the initial vector towards the x-axis, with the accumulated angle Z_i representing the total rotation applied to align the vector with the x-axis. The precision of the CORDIC algorithm generally increases with the number of iterations. Subsequent iterations follow a similar process, and through multiple iterations, an angle value with finite error can be obtained.

2.3. Arithmetic

Fixed-point arithmetic (showed in Table 2.1) is a method of representing numbers that has a fixed number of digits before and after the decimal point. Unlike floating-point arithmetic,

which can represent a vast range of numbers by using a floating exponent, fixed-point numbers allocate a specific portion of the number to the integer part and a specific portion to the fractional part. This method simplifies the hardware required for calculations, making it more efficient and less costly, especially in systems where the range and precision of numbers can be predetermined

To represent negative numbers within this system, two's complement notation is typically used. In this notation, the most significant bit (MSB) indicates the sign of the number; a '1' in the MSB signifies a negative number, which must be interpreted by converting it into its positive equivalent through two's complement conversion.

Table 2.1.: Number representation

Decimal	binary	Hexadecimal
1	0000.0000.0000.0001,0000.0000.0000.0000	0001,0000
2	0000.0000.0000.0010,0000.0000.0000.0000	0002,0000
-2	1111.1111.1111.1110,0000.0000.0000.0000	FFFE,0000
4	0000.0000.0000.0100,0000.0000.0000.0000	0004,0000
0,5	0000.0000.0000.0000,1000.0000.0000.0000	0000,8000
-0,5	1111.1111.1111.1111,1000.0000.0000.0000	FFFF,8000
0,25	0000.0000.0000.0000,0100.0000.0000.0000	0000,4000

2.4. Memory for input and output

The circuit interfaces with memory via a **32-bit EAB** (External Address Bus) for transmitting addresses, a **32-bit EDB_IN** (External Data Bus In) for inputting data into the memory, and a **32-bit EDB_OUT** (External Data Bus Out) for outputting the final computational results. Memory input and output operations are controlled by the **ram_rd_en** (RAM Read Enable) and **ram_wr_en** (RAM Write Enable) signals.

The memory subsystem comprises a total of 10 units, with **mem[0]-mem[6]** dedicated to storing the arctan value table (Table 2.3). **mem[7]** is allocated for storing the fixed-point representation of 1, designated as 00010000 in hexadecimal format. The coordinates y and x are stored in **mem[8]** and **mem[9]**, respectively. The final calculation result will be written in **mem[32]**

2.5. Value range and accuracy

In the design of CORDIC circuits, the primary consideration is the **accuracy** of the results. The CORDIC algorithm is iterative, with each iteration bringing the results closer to the target angle. However, without sufficient accuracy, the computed results would be rendered meaningless. Analysis has shown that to achieve an accuracy **within 1 degree** for the final angle calculation, at least **7 iterations** are required, as each additional iteration halves the error margin.

To **further reduce** the error by an order of magnitude, to achieve an accuracy of 0.1 degrees, at least **4 more iterations** would be necessary. However, this significantly increases the computational time cost, and the marginal benefit of increased accuracy is very low. Therefore, the algorithm was ultimately set to **7 iterations**. This decision balances the need for accuracy with considerations of time cost and marginal benefits. The CORDIC algorithm I designed computes the angle values corresponding to points within the first quadrant. After 7 iterations, the angles rotated in each step are as shown in Table 2.2. Theoretically, if the last rotation angle of the CORDIC algorithm is **0.9 degrees**, the precision should generally be **within 1 degree**, as

each iteration approximates the target angle more closely, and the final rotation angle of the last iteration determines the ultimate precision of the algorithm.

This restructured explanation clarifies why **7 iterations** were chosen for the CORDIC algorithm: it represents a compromise between the requirements for accuracy and the costs of computation.

Table 2.2.: $arctan(1/2^i)$ Table

i	0	1	2	3	4	5	6
degree	45.0°	26.57°	14.04°	7.13°	3.58°	1.79°	0.90°
radian	$\frac{\pi}{4}$	$\frac{\pi}{6.83}$	$\frac{\pi}{12.85}$	$\frac{\pi}{25.24}$	$\frac{\pi}{50.27}$	$\frac{\pi}{100.40}$	$\frac{\pi}{200.56}$

To facilitate processing in digital circuits, the angles are normalized using the formula

$$(2^{19}/\pi) * arctan(1/(2^n))$$

After converting these numbers to hexadecimal, the new table is showed in Table 2.3. These are the values stored in mem[0] to mem[6].

Table 2.3.: Normalized hexadecimal table

i	0	1	2	3	4	5	6
degree	20000	12E40	09FB3	05111	028B0	0145D	00A2F

2.6. Program Flowchart

The algorithm's flowchart is illustrated in the figure 2.1, with the program flow divided into two phases: 'Load' and 'Compute'. The *arctan* angle results are stored in mem[0]-mem[6] and need to be initially loaded into registers. The variable 'i' emulates the function of an External Address Bus (EAB) for addressing, whereas 'E' is predefined for iterative calculations of rotation angles. 'C' stores the fixed-point representation of the number 1 (i.e., the hexadecimal 00010000), with 'Y' and 'X' holding the vertical and horizontal coordinates, respectively. 'j' is used for register iteration. In the computation phase, 'C' is right-shifted and multiplied by 'A' and 'B' to avoid division operations, thereby reducing the circuit's required time. The final result is stored in mem[32] and the state is returned to IDLE1 (More detailed explanation is in section 3.5), saving the time needed to load the *arctan* angle values for multiple calculations.

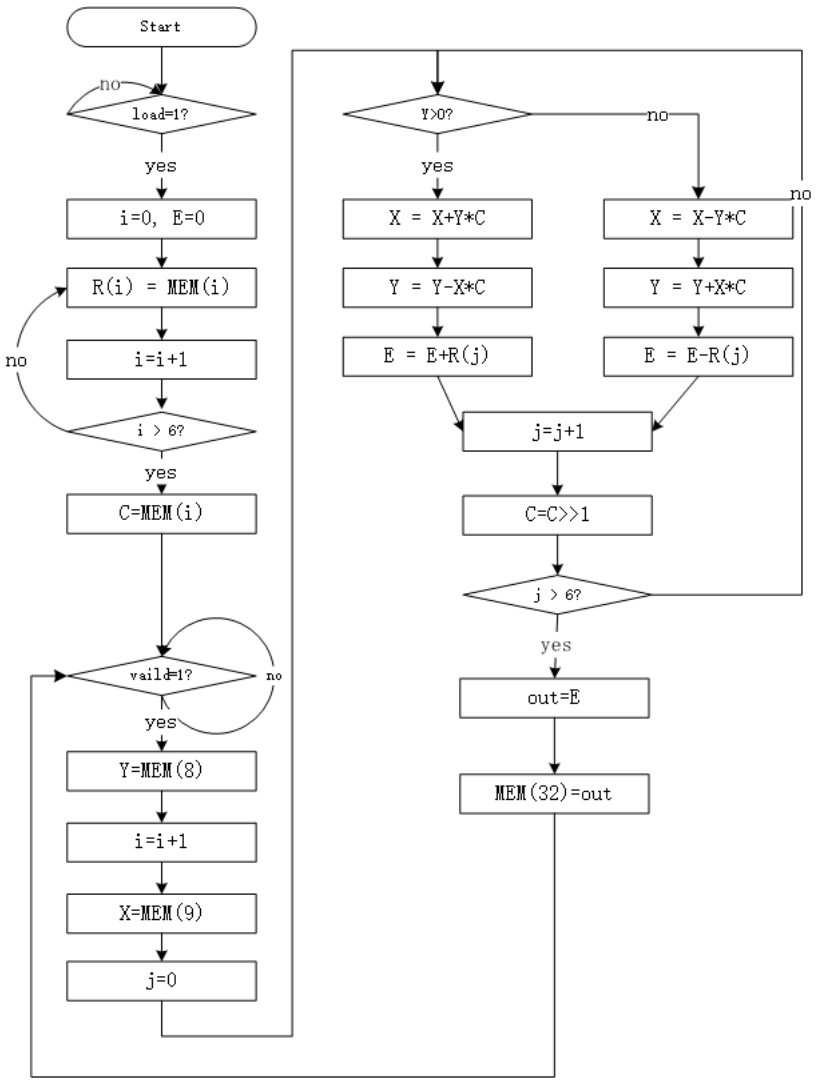

Figure 2.1.: Caption

2.7. limitations of the algorithm

The CORDIC (Coordinate Rotation Digital Computer) algorithm, widely used for trigonometric, hyperbolic, exponential, and logarithmic function computations, comes with certain limitations and its accuracy can be affected by several factors:

1. Convergence Speed: The convergence speed of the CORDIC algorithm depends on the number of iterations. To achieve higher precision, more iterations are required, which increases computational time. This trade-off can be a significant limitation in time-sensitive applications.

2. Finite Precision: CORDIC operates with finite precision. The use of fixed-point arithmetic limits the achievable accuracy, as rounding errors accumulate with each iteration. The precision is directly influenced by the bit-width of the computation.

3. Quantization Error: In digital implementations, quantization error is introduced when continuous values are represented in discrete form. This error affects the final precision and is more pronounced in fixed-point implementations.

4. Angular Range Limitation: The basic CORDIC algorithm is limited to certain angular ranges. Extending the range requires additional logic, such as range reduction and symmetries, which can complicate the algorithm and introduce potential sources of error.

3. Theoretical preparation

3.1. Scheduling

The total work is devided into two parts: loading and calculating. The design objective is to minimize computational time while ensuring accuracy, and to reduce the usage of hardware and buses as much as possible through reuse. Consequently, two MAC modules and two ALUs have been adopted. To further reduce hardware consumption, reusing a single MAC module and ALU is possible, though this would incur a time cost. In the design process, division by 2 operations are replaced with shifters and multipliers, and the arctan values need to be loaded from memory only once. This allows for multiple uses in subsequent calculations, thereby saving significant load time during continuous computations. The computational tasks of the MAC and ALU are pipelined, making adjustments straightforward even if changes in precision or time requirements arise later on.

3.2. Data flow graph

Due to the frequent occurrence of calculations such as $A = A + B * C$ and $B = B + A * C$, the Multiply-Accumulate (MAC) (Figure 3.1) module proves to be an invaluable tool in such scenarios. The MAC module is a fundamental component in digital signal processing and related computational tasks, efficiently combining multiplication and accumulation operations. Given that both the Arithmetic Logic Unit (ALU) and the Multiplication (MUL) module require one clock cycle each to process data, the MAC operation inherently necessitates two clock cycles for data processing. This is because the MAC module performs the multiplication of two numbers followed by the addition of the product to an accumulator in a sequential manner, aligning with the inherent processing requirements of the ALU and MUL modules.

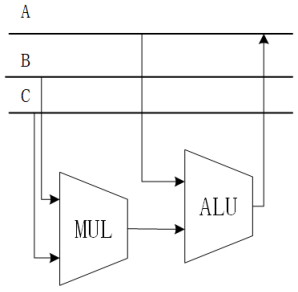

Figure 3.1.: Multiply-Accumulate

The data flow graph for the loading and computation segments are presented in Figure 3.2. Within Figure 3.2, the section preceding IDLE1 predominantly illustrates the iterative loading of *arctan* values from Memory. The part following IDLE1 commences with the loading of the Y and X coordinates into registers R10 and R11, respectively. As the calculations pertain to coordinates within the first quadrant, the initial operation executed by ALU1 is invariably an addition, thereby negating the necessity for a flag signal. R7 introduces a fixed-point representation of 1 into the MUL component of the MAC, which is subsequently right-shifted through a shifter and fed into the MUL component of the MAC in the ensuing operational cycle. Owing to the repetitive nature of the iterative process, the ALU and MUL components of the MAC have been merged and simplified in the diagram. In reality, each MAC cycle encompasses two operational cycles, delineated by green lines within the diagram. The flag signal, crucial for determining whether the ALU performs an addition or subtraction, is derived from the outcome of the previous computation cycle's MAC_ALU. The process of acquiring the flag from the MAC_ALU module is omitted in Figure 3.2.

3.3. Architecture of the data path

During the design of the data path (as shown in Figure 3.3), I used two MAC modules, one shifter, and two ALUs, along with registers DIN, R0-R11, and three constant modules: const0, const1, and const32. The ALUs primarily perform the accumulation of EAB, with ALU1 calculating the CORDIC angle values. The operation sign (addition or subtraction) for ALU1 depends on the result of the MAC operation. The MAC computes A+BC and forwards the result to the A bus for subsequent accumulation and flag status determination. MAC1 computes B+AC and sends the result to the B bus, also for further accumulation and multiplication operations. In the actual implementation, five MUX modules were also used to control the signals transmitted to the buses, avoiding bus contention (as shown in Figure 4.1). However, for clarity and simplicity, these were omitted in Figure 3.3.

3.4. Register transfer sequences

As illustrated in Figure 3.4, the register transfer sequences encompass two principal components: data loading from memory and computation, along with the corresponding control logic. The control logic is further elucidated in the appendix, where it is implemented in code. This section provides a detailed explanation of the data processing and control mechanisms at each step.

3.5. FSM state graph and state encoding

The FSM (Finite State Machine) has a total of 44 valid states, with the state transition diagram shown in Figure 3.5. During the load phase, the loading of the arctan table is initiated by the load signal. The computation phase is activated by the valid signal, and the key value for determining the branches during the computation phase is the flag signal. The flag signal controls the state transitions, thereby governing the addition and subtraction operations of the MAC and ALU.

To encode these 44 states, a minimum of 6 bits is required due to the binary numbering system. In addition to this, there are three signal states: load, valid, and flag. The complete state transition table is shown in Figure 3.6. For unused states, IDLE is defaulted as the next state.

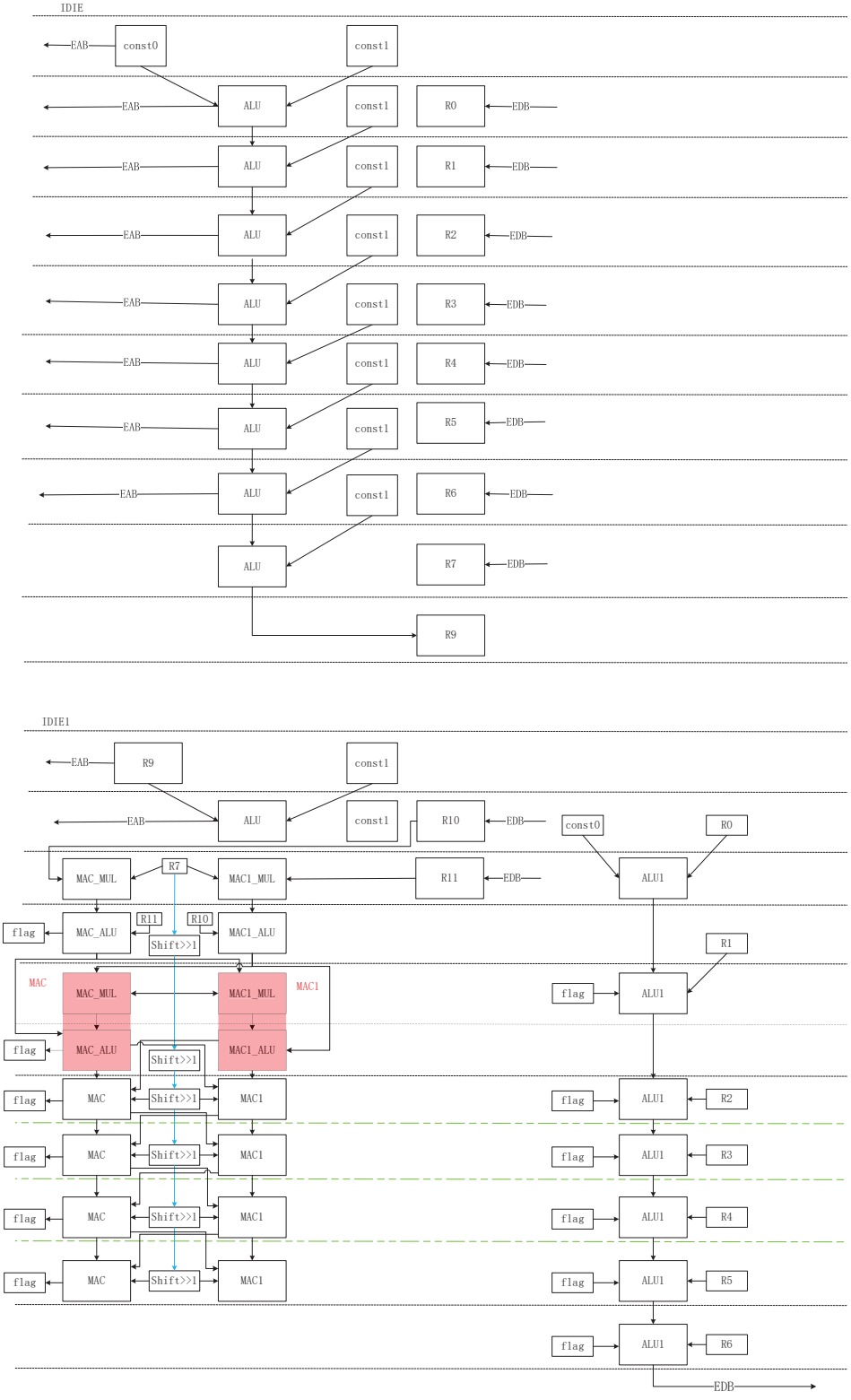

Figure 3.2.: Data flow graph

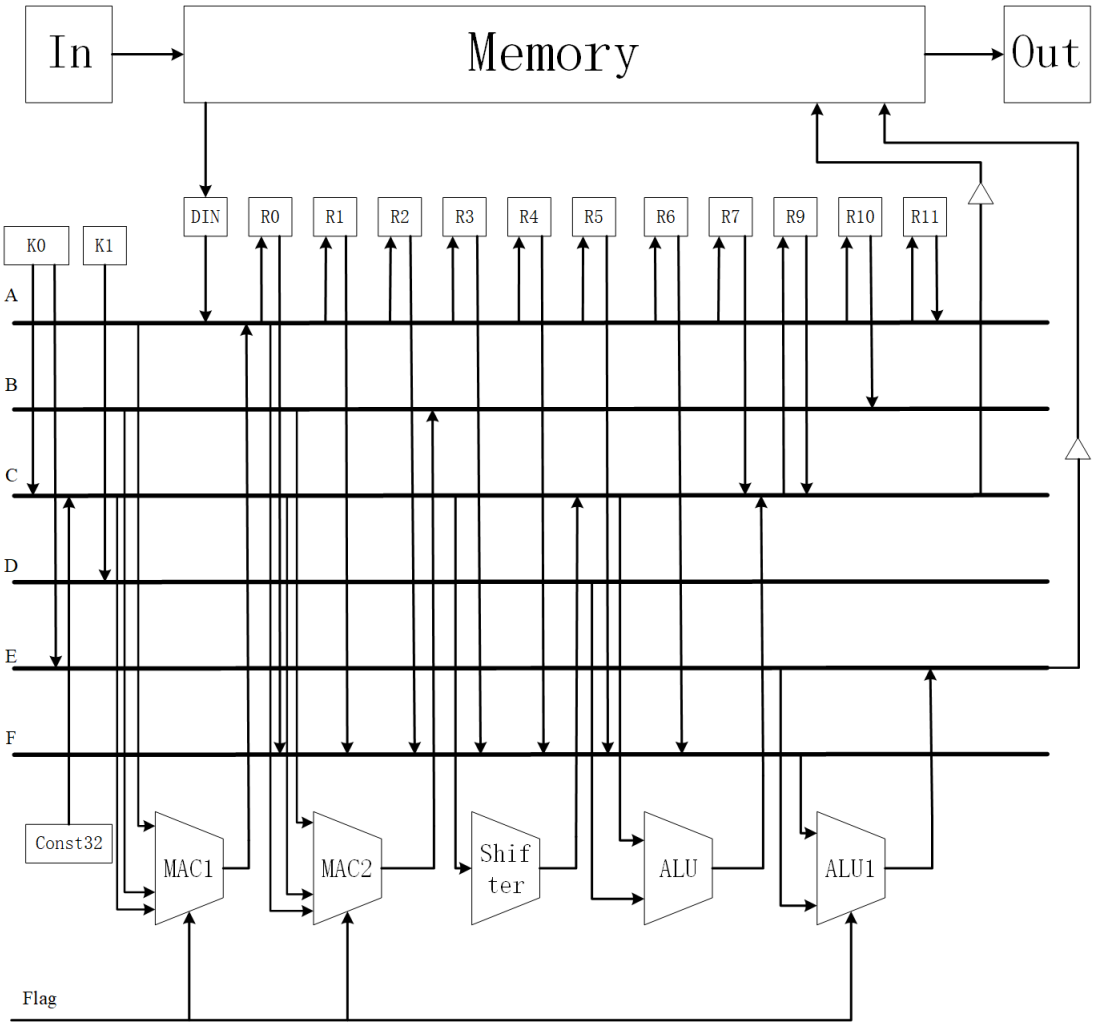

Figure 3.3.: Architecture of data path

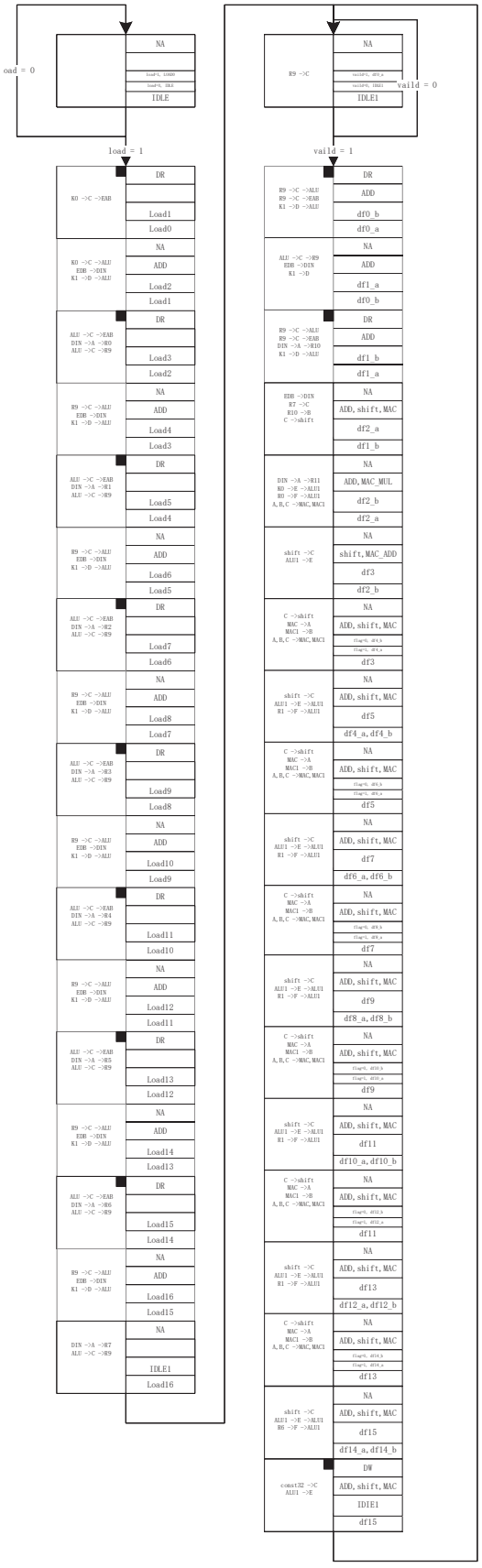

Figure 3.4.: Register transfer sequences

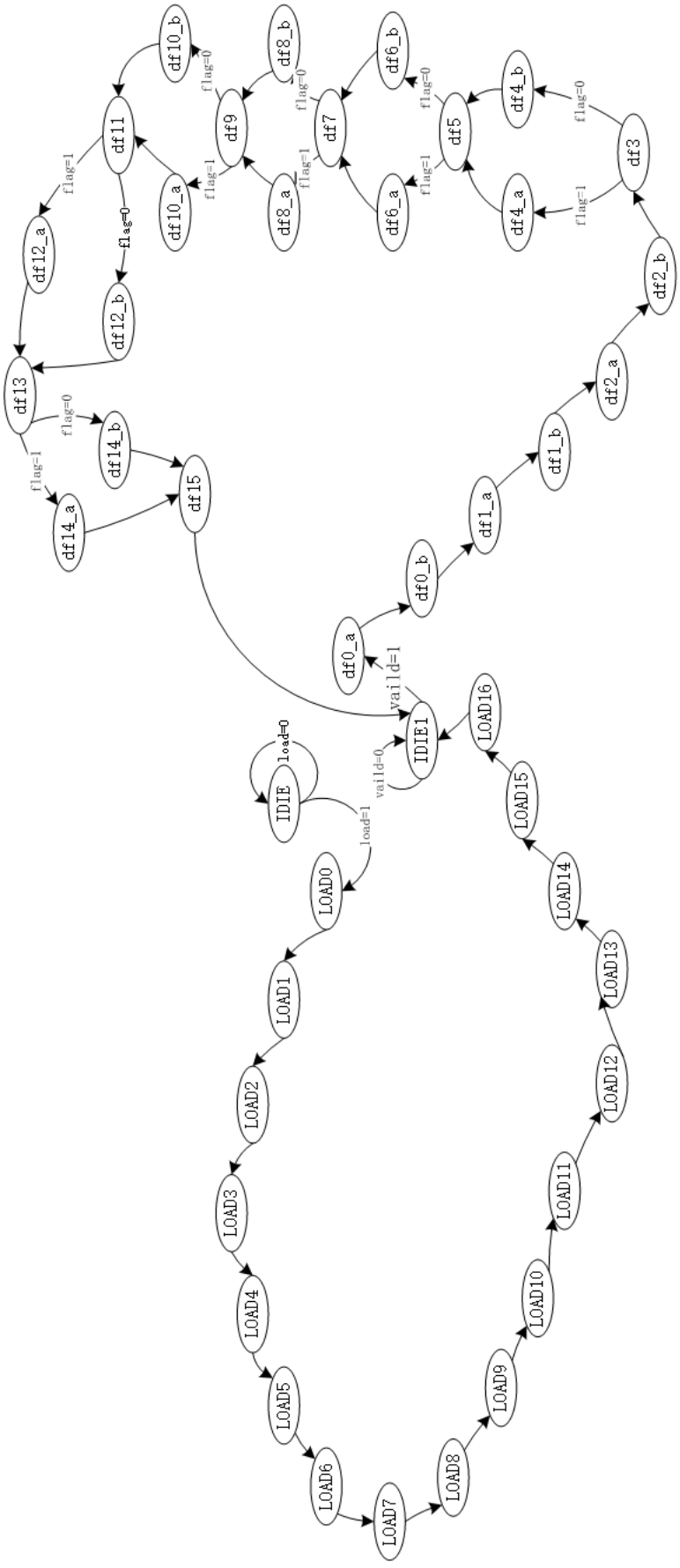

Figure 3.5.: FSM state graph

state	x5	x4	x3	x2	x1	x0	valid	load	flag	next state	y5	y4	y3	y2	y1	y0
IDLE	0	0	0	0	0	0	x	0	x	IDLE	0	0	0	0	0	0
IDLE	0	0	0	0	0	0	x	1	x	LOAD0	0	0	0	0	0	1
LOAD0	0	0	0	0	0	1	x	x	x	LOAD1	0	0	0	0	1	0
LOAD1	0	0	0	0	1	0	x	x	x	LOAD2	0	0	0	0	1	1
LOAD2	0	0	0	0	1	1	x	x	x	LOAD3	0	0	0	1	0	0
LOAD3	0	0	0	1	0	0	x	x	x	LOAD4	0	0	0	1	0	1
LOAD4	0	0	0	1	0	1	x	x	x	LOAD5	0	0	0	1	1	0
LOAD5	0	0	0	1	1	0	x	x	x	LOAD6	0	0	0	1	1	1
LOAD6	0	0	0	1	1	1	x	x	x	LOAD7	0	0	1	0	0	0
LOAD7	0	0	1	0	0	0	x	x	x	LOAD8	0	0	1	0	0	1
LOAD8	0	0	1	0	0	1	x	x	x	LOAD9	0	0	1	0	1	0
LOAD9	0	0	1	0	1	0	x	x	x	LOAD10	0	0	1	0	1	1
LOAD10	0	0	1	0	1	1	x	x	x	LOAD11	0	0	1	1	0	0
LOAD11	0	0	1	1	0	0	x	x	x	LOAD12	0	0	1	1	0	1
LOAD12	0	0	1	1	0	1	x	x	x	LOAD13	0	0	1	1	1	0
LOAD13	0	0	1	1	1	0	x	x	x	LOAD14	0	0	1	1	1	1
LOAD14	0	0	1	1	1	1	x	x	x	LOAD15	0	1	0	0	0	0
LOAD15	0	1	0	0	0	0	x	x	x	LOAD16	0	1	0	0	0	1
LOAD16	0	1	0	0	0	1	x	x	x	IDLE1	0	1	0	0	1	0
IDLE1	0	1	0	0	1	0	0	x	x	IDLE1	0	1	0	0	1	0
IDLE1	0	1	0	0	1	0	1	x	x	df0_a	0	1	0	0	1	1
df0_a	0	1	0	0	1	1	x	x	x	df0_b	0	1	0	1	0	0
df0_b	0	1	0	1	0	0	x	x	x	df1_a	0	1	0	1	0	1
df1_a	0	1	0	1	0	1	x	x	x	df1_b	0	1	0	1	1	0
df1_b	0	1	0	1	1	0	x	x	x	df2_a	0	1	0	1	1	1
df2_a	0	1	0	1	1	1	x	x	x	df2_b	1	0	1	0	1	1
df2_b	1	0	1	0	1	1	x	x	x	df3	0	1	1	0	0	0
df3	0	1	1	0	0	0	x	x	0	df4_b	0	1	1	0	1	0
df3	0	1	1	0	0	0	x	x	1	df4_a	0	1	1	0	0	1
df4_a/df4_b	0	1	1	0	1/0	0/1	x	x	x	df5	0	1	1	0	1	1
df5	0	1	1	0	1	1	x	x	0	df6_b	0	1	1	1	0	1
df5	0	1	1	0	1	1	x	x	1	df6_a	0	1	1	1	0	0
df6_a/df6_b	0	1	1	1	0	0/1	x	x	x	df7	0	1	1	1	1	0
df7	0	1	1	1	1	0	x	x	0	df8_b	1	0	0	0	0	0
df7	0	1	1	1	1	0	x	x	1	df8_a	0	1	1	1	1	1
df8_a/df8_b	0/1	1/0	1/0	1/0	1/0	1/0	x	x	x	df9	1	0	0	0	0	1
df9	1	0	0	0	0	1	x	x	0	df10_b	1	0	0	0	1	1
df9	1	0	0	0	0	1	x	x	1	df10_a	1	0	0	0	1	0
df10_a/df10_b	1	0	0	0	1	0/1	x	x	x	df11	1	0	0	1	0	0
df11	1	0	0	1	0	0	x	x	0	df12_b	1	0	0	1	1	0
df11	1	0	0	1	0	0	x	x	1	df12_a	1	0	0	1	0	1
df12_a/df12_b	1	0	0	1	0/1	1/0	x	x	x	df13	1	0	0	1	1	1
df13	1	0	0	1	1	1	x	x	0	df14_b	1	0	1	0	0	1
df13	1	0	0	1	1	1	x	x	1	df14_a	1	0	1	0	0	0
df14_a/df14_b	1	0	1	0	0	0/1	x	x	x	df15	1	0	1	0	1	0
df15	1	0	1	0	1	0	x	x	x	IDIE1	0	1	0	0	1	0
unused	1	0	1	1	x	x	x	x	x	IDLE0	0	0	0	0	0	0
unused	1	1	x	x	x	x	x	x	x	IDLE1	0	0	0	0	0	0

Figure 3.6.: FSM State coding table

4. Implementation and Simulation

4.1. Datapath circuit diagram

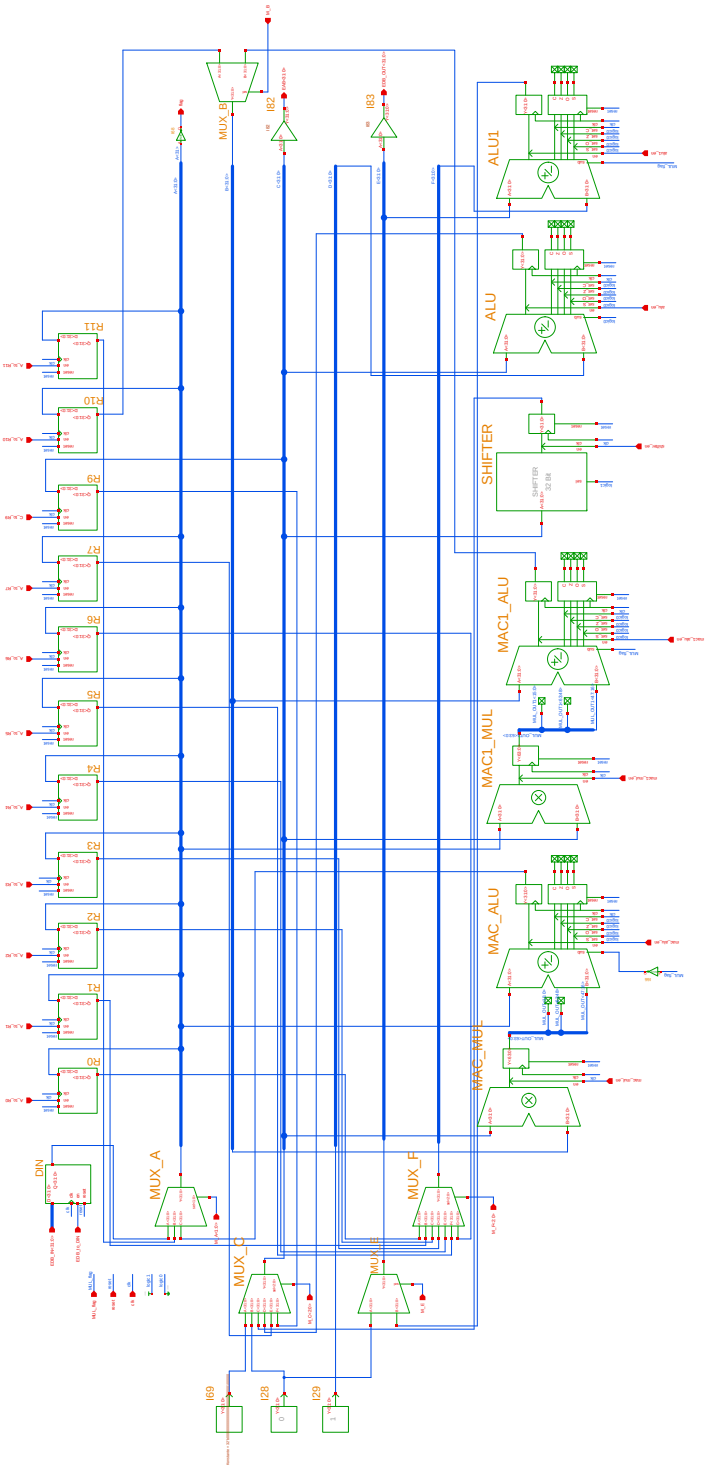

Figure 4.1.: Datapath in CANDENCE

4.2. The overall circuit

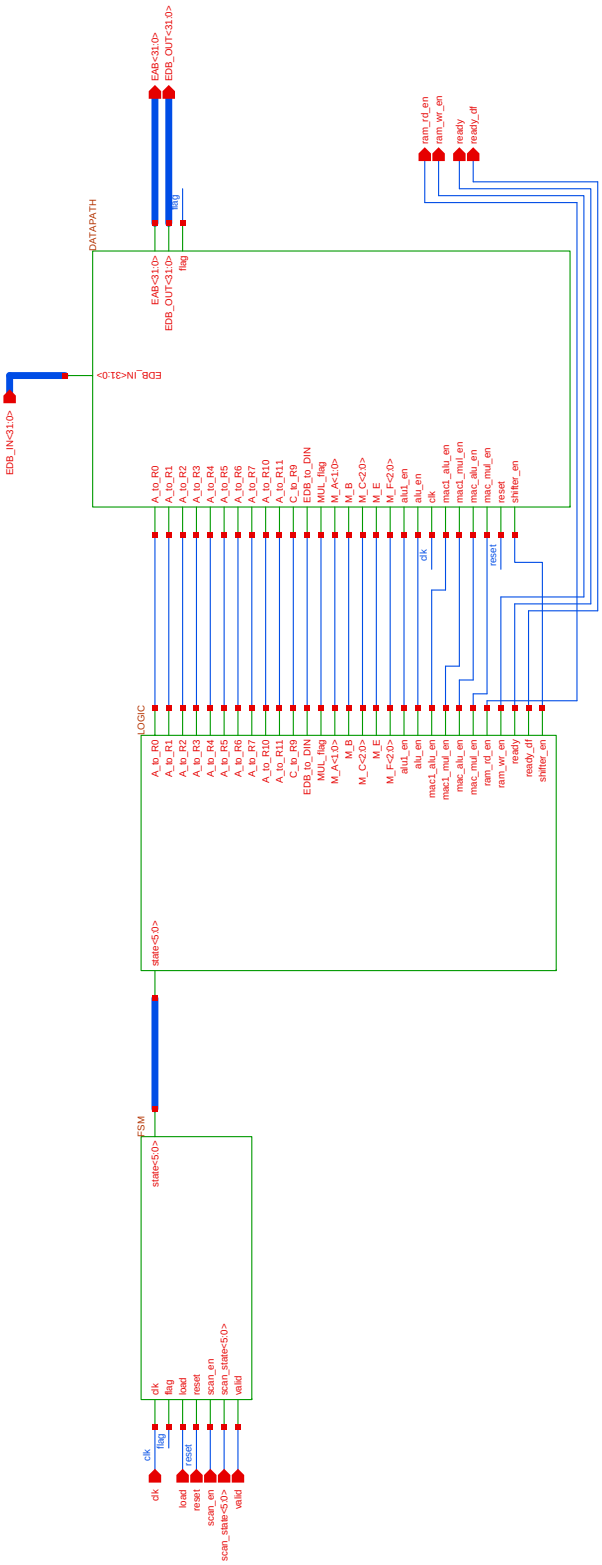

Figure 4.2.: The overall circuit in CANDENCE

4.3. Test of the overall circuit

From Figure 4.3, we can see that as the control signals are issued, mem[0] to mem[7] are written into registers R0 to R7. The next address of EAB is stored in R9, which is used for data reading during the computation phase.

In the subsequent Figure 4.4, the Y and X coordinates are loaded into R10 and R11, respectively. Meanwhile, the fixed-point number 1 is prepared on bus C for multiplication in the MAC MUL unit. In the test bench, the coordinates (6666, 3333) are used to test the performance of the CORDIC algorithm. The values 6666 and 3333, when converted into fixed-point hexadecimal, become 1A0A0000 and 0D050000, respectively.

In Figure 4.5, it is observed that the multiplier within the MAC has completed its operation. The timing for the addition or subtraction operations within the ALU is controlled by the ALU_en signal. The results from the MAC are transmitted to the A and B buses to participate in the calculations of the next operational cycle.

The computational process in Figure 4.6 is similar to that in Figure 4.5, ultimately yielding the final result shown in Figure 4.7, which is then written into mem[32].

The final result obtained is hexadecimal 12A86 (decimal 76422). Based on the normalization formula given in section 2.5, the angle can be calculated as

$$Degree = \frac{76422 * \pi}{2^{19}} \approx 26.24^{\circ}$$

Comparing with the results obtained from a calculator, it can be seen that the error in the angle has been controlled within 1 degree.

$$arctan(\frac{3333}{6666}) = arctan(\frac{1}{2}) \approx 26.57^{\circ}$$

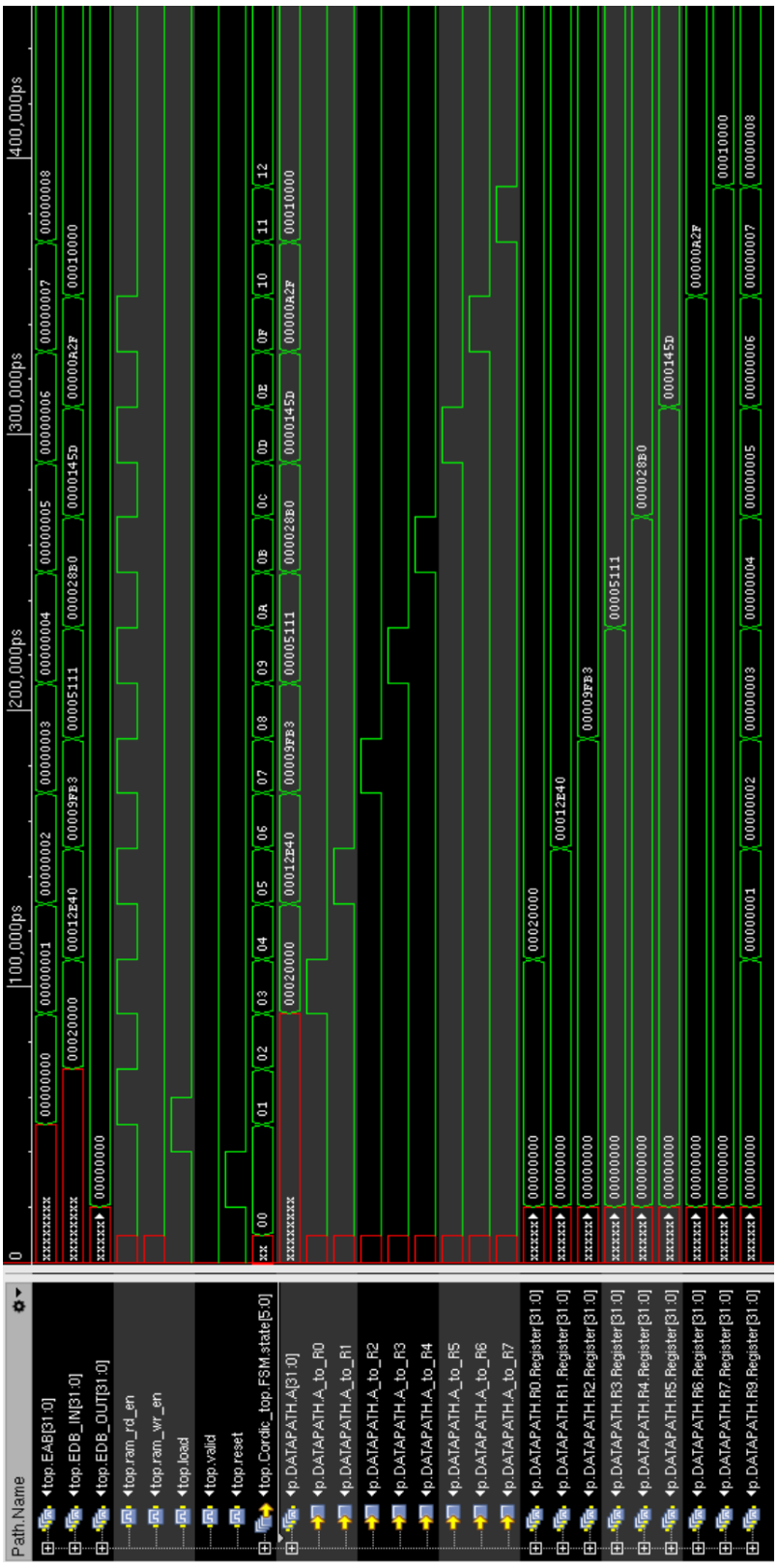

Figure 4.3.: Simulation results of the overall circuit(part1)

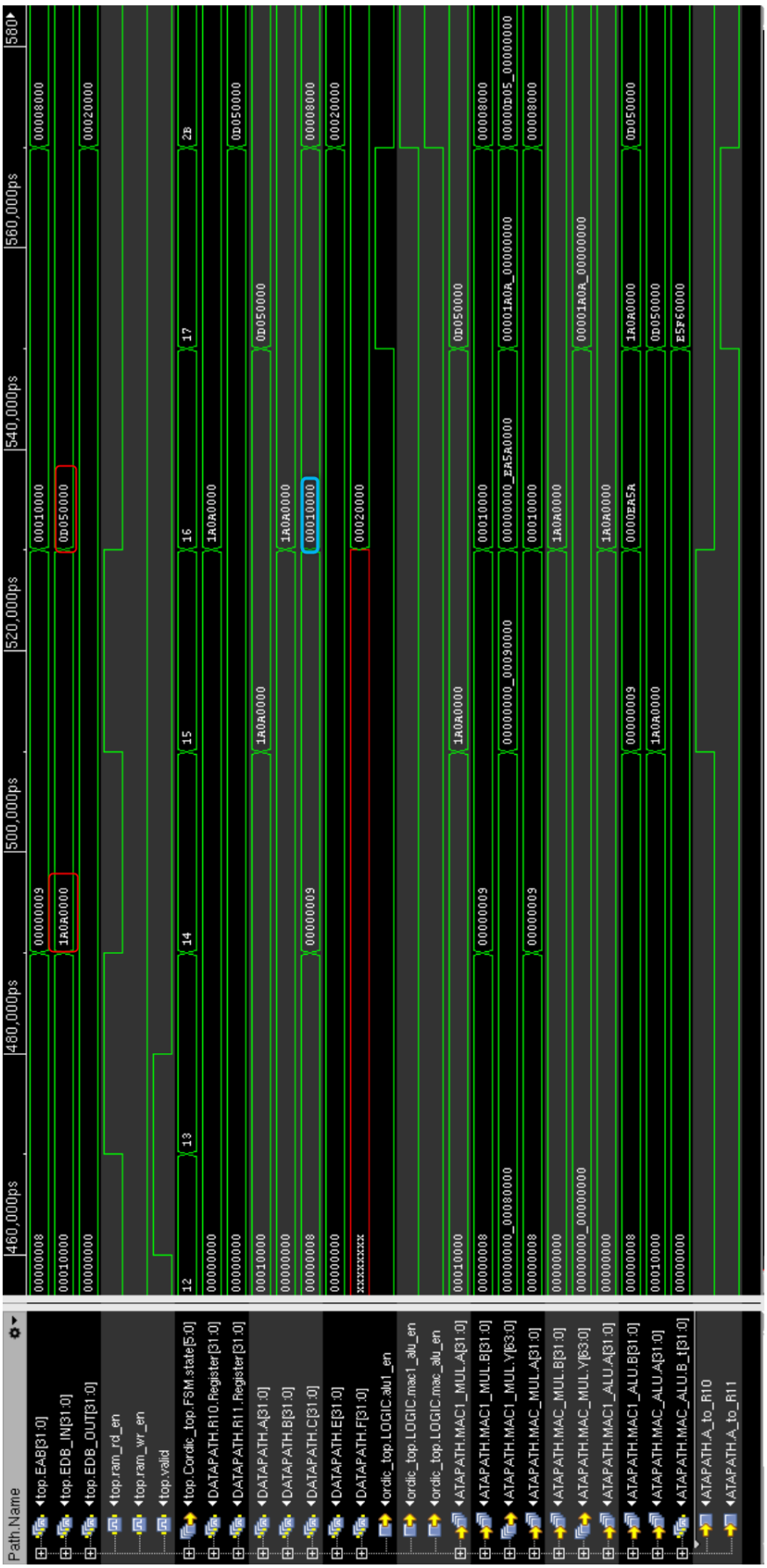

Figure 4.4.: Simulation results of the overall circuit(part2)

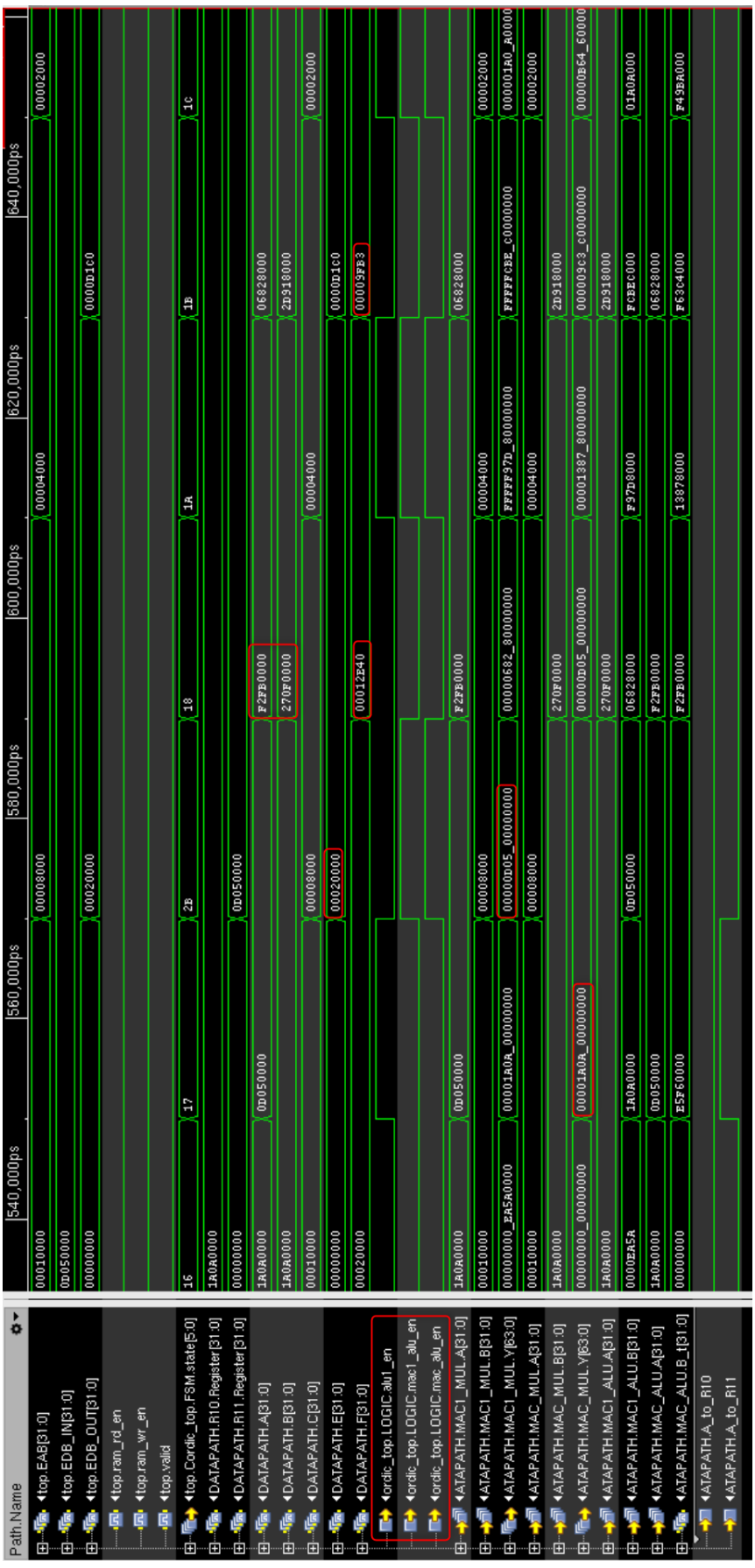

Figure 4.5.: Simulation results of the overall circuit(part3)

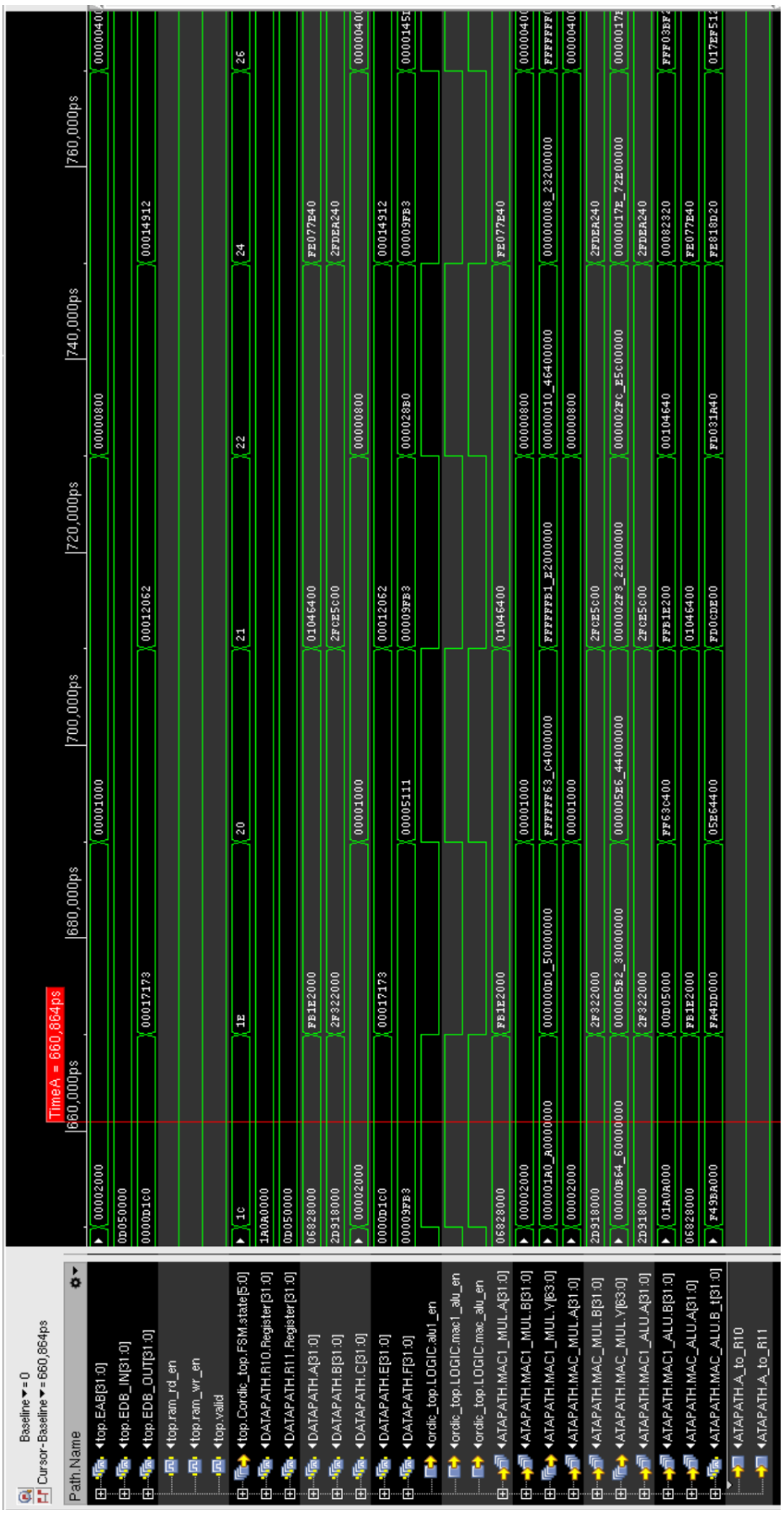

Figure 4.6.: Simulation results of the overall circuit(part4)

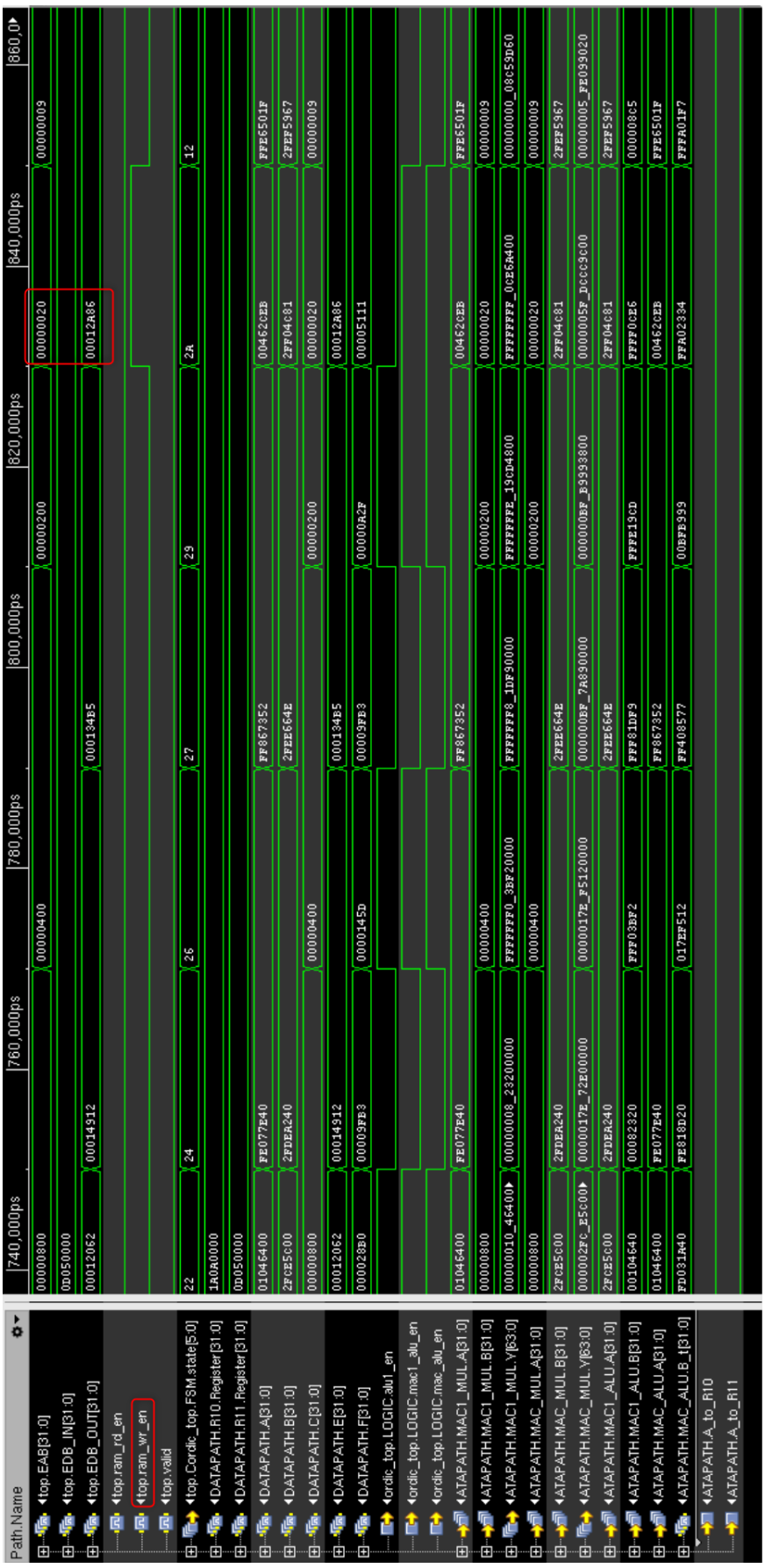

Figure 4.7.: Simulation results of the overall circuit(part5)

4.4. Circuit and layout synthesis of the entire circuit

In this section, the synthesis and layout of the entire circuit will be completed. Although it was not possible to perform simulation after introducing delays, I will honestly describe the work I have accomplished and analyze the potential causes of errors.

4.4.1. Analysis and evaluation of constraints

According to the information given in the report (Table 4.1), there are three constraint violations, maybe there are some ways to improve it, but I didn't succeed.

- Max Leakage Power Violation: Optimizing the design for low-power by revising the logic and possibly using more power-efficient architectures or components.
- Max Transition Violation: Reducing the load on critical nets by minimizing fan-out or using buffer/inverter chains to strengthen the drive capability. Reviewing and optimizing the placement and routing to shorten interconnect lengths, thus reducing capacitive load.
- Max Area Violation: Optimizing the floorplan and placement to ensure a more compact layout, potentially through more aggressive place-and-route settings.

Constraint	Cost
multiport_net	0.00 (MET)
max_transition	4.04 (VIOLATED)
max_capacitance	0.00 (MET)
max_delay/setup	0.00 (MET)
sequential_clock_pulse_width	0.00 (MET)
critical_range	0.00 (MET)
max_leakage_power	10529303.00 (VIOLATED)
max_area	891893.38 (VIOLATED)

Table 4.1.: Constraint Analysis

4.4.2. kritischen Timingpfaden

It can be observed that using a clock with CYCLE=80 meets the requirements(Table 4.2), but the slack time is very short, only 0.06ns. The majority of the delay primarily originates from the Data path section.

Path	Data Arrival Time	Data Required Time	Slack
FSM to DATAPATH	81.24ns	81.30ns	0.06ns
scan_en to FSM	11.65ns	81.30ns	69.65ns
FSM to EAB[31]	19.53ns	71.30ns	51.77ns

Table 4.2.: Summary of Timing Report

4.4.3. power and area

The total power consumption is 2.6544 mW, and the total area is 891893. In the final layout, it was placed on a board with a weight of 1000 and a height of 900, as shown in Figure 4.8.

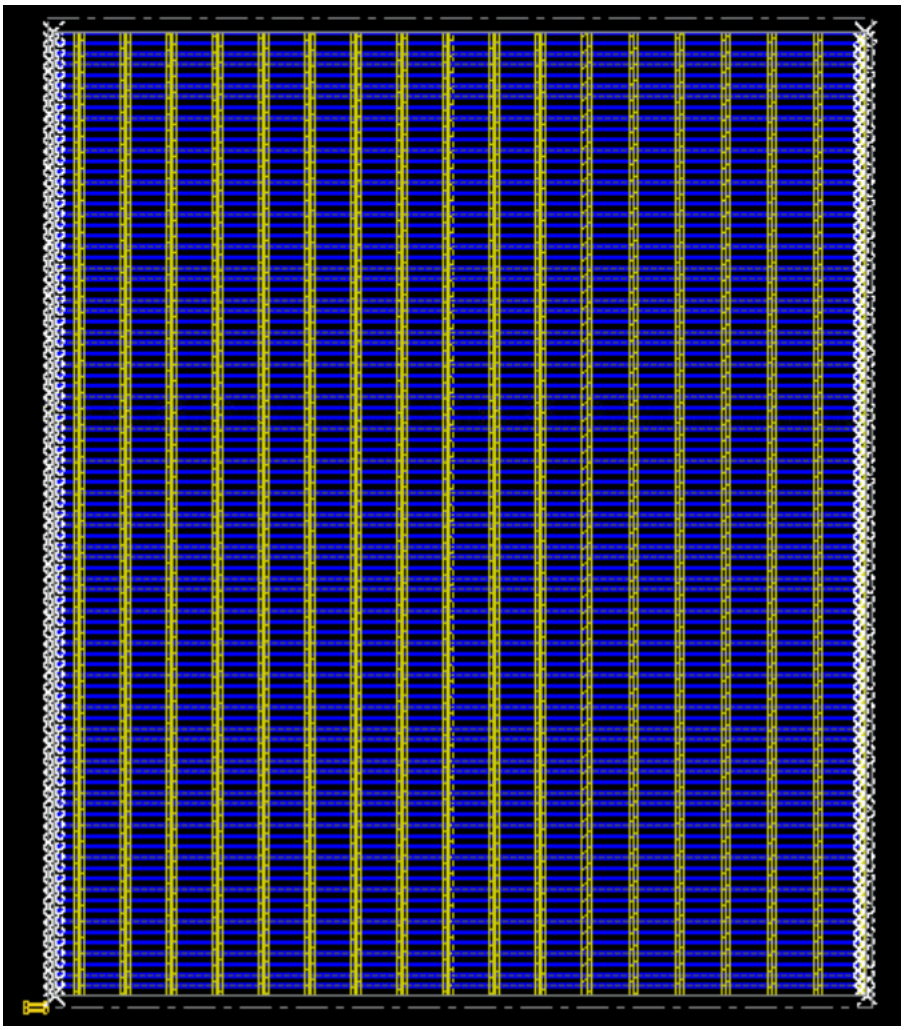

Figure 4.8.: floor-plan of Layoutsynthese

4.4.4. simulation with delay

When I introduced delays in the simulation, a noticeable latency became evident. However, the circuit ceased to function correctly upon receiving the reset signal. I reviewed the compile.log file from the synthesis process and did not identify any issues related to latches. The cause might be related to the violation of constraints mentioned in section 4.4.1.

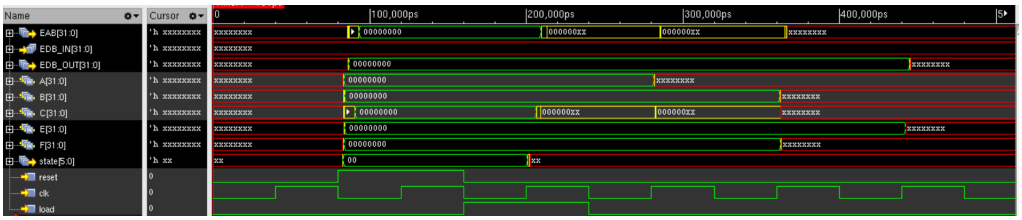

Figure 4.9.: simulation with delay

5. Conclusion

In this report, we introduced a circuit designed for computing CORDIC, comprised of three main components: FSM, control logic, and datapath. The circuit's performance was validated through simulation, achieving the desired accuracy after seven iterations of the circuit. A notable limitation encountered was the inability to complete simulations with added delays during the synthesis and layout process, highlighting the complexity of practical circuit design. This experience underscored the importance of considering the performance of fundamental components (analog elements) in digital circuit design, rather than assessing circuit feasibility from a singular perspective.

Another area identified for potential improvement involves expanding the applicability of the CORDIC algorithm to cover the 2nd, 3rd, and 4th quadrants, which would necessitate additional processing logic. Moreover, designing a CORDIC with variable precision, where precision requirements are input variables, could offer a better balance between performance and time trade-offs.

This design endeavor has facilitated a systematic learning of the Verilog language. The website HDLbits was instrumental in enhancing my coding skills in this domain, and my supervisor provided patient responses to all my inquiries, further enriching my learning experience.

List of Figures

List of Tables

Bibliography

[1] Prof.Mayr. Anleitung für das praktikum schaltkreis- und systementwurf, rechnergestützter schaltkreisentwurf. Technische Universität Dresden, Lehrstuhl Hochparallele VLSI-System und Neuromikroelektronik, Institut für Grundlagen der Elektrotechnik und Elektronik der Fakultät Elektrotechnik. Available online: https://tu-dresden.de/ing/elektrotechnik/iee/hpsn/studium/materialien/sse_praktikum.

[2] CORDIC - Wikipedia. https://en.wikipedia.org/wiki/CORDIC, 2024. Accessed: [03.13.2024].

A. Source code

A.1. Cordic C program

```
1
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <math.h>
5
6  #define PI 3.141592653589793
7  #define PHASE_2PI 1073741824  //2^14
8
9  int atan_list[32];
10
11 int cordic(int,int);
12
13
14 int main()
15 {
16     int i;
17     int re,imag;
18     int phase;
19
20     for(i=0;i<32;i++)
21     {
22         atan_list[i] = PHASE_2PI*atan
               (1.0/(1<<i))/2/PI;
23     }
24
25     re = -123456;
26     imag = 123456;
27
28     phase = cordic(imag,re);
29         printf("phase %d, %f degree\n",
               phase,phase*360.0/PHASE_2PI);
30
31     return 0;
32 }
33
34 int cordic(int Qps,int Ips){
35     int i;
36     int tmpq,tmpi;
37     int reg_x_sign,reg_y_sign;
38     int reg_x_shift,reg_y_shift;
39     int sign,sigma;
40     int tmp_phase = 0;
41
42     tmpi = (Ips>=0)?Ips:-Ips;
43     tmpq = (Ips>=0)?Qps:-Qps;
44
45
46
47     for (i=0;i<20;i++)
48     {
49         sigma = (tmpq>=0);
50         if(sigma){
51             tmp_phase += atan_list[i];
52             reg_x_sign = - tmpi;
53             reg_y_sign = - tmpq;
54
55         }
56         else {
57             tmp_phase -= atan_list[i];
58             reg_x_sign = tmpi;
59             reg_y_sign = tmpq;
60         }
61         reg_x_shift = (reg_x_sign>>i);
62         reg_y_shift = (reg_y_sign>>i);
63
64         tmpi -= reg_y_shift;
65         tmpq += reg_x_shift;
66     }
67
68     return tmp_phase;
69 }
```

A.2. Cordic FSM

```
1
2  //Verilog HDL for "fc_example", "
       Cordic_fsm" "functional"
```

```
3
4
5  module Cordic_fsm (
6    clk,
7    reset,
8    load,
9    valid,
10   flag,
11   state,
12   scan_en,
13   scan_state
14 );
15   parameter IDLE = 6'b000000;
16   parameter LOAD0 = 6'b000001;
17   parameter LOAD1 = 6'b000010;
18   parameter LOAD2 = 6'b000011;
19   parameter LOAD3 = 6'b000100;
20   parameter LOAD4 = 6'b000101;
21   parameter LOAD5 = 6'b000110;
22   parameter LOAD6 = 6'b000111;
23   parameter LOAD7 = 6'b001000;
24   parameter LOAD8 = 6'b001001;
25   parameter LOAD9 = 6'b001010;
26   parameter LOAD10 = 6'b001011;
27   parameter LOAD11 = 6'b001100;
28   parameter LOAD12 = 6'b001101;
29   parameter LOAD13 = 6'b001110;
30   parameter LOAD14 = 6'b001111;
31   parameter LOAD15 = 6'b010000;
32   parameter LOAD16 = 6'b010001;
33   parameter IDLE1 = 6'b010010;
34   parameter df0_a = 6'b010011;
35   parameter df0_b = 6'b010100;
36   parameter df1_a = 6'b010101;
37   parameter df1_b = 6'b010110;
38   parameter df2_a = 6'b010111;
39   parameter df2_b = 6'b101011; //added
40   parameter df3 = 6'b011000;
41   parameter df4_a = 6'b011001;
42   parameter df4_b = 6'b011010;
43   parameter df5 = 6'b011011;
44   parameter df6_a = 6'b011100;
45   parameter df6_b = 6'b011101;
46   parameter df7 = 6'b011110;
47   parameter df8_a = 6'b011111;
48   parameter df8_b = 6'b100000;
49   parameter df9 = 6'b100001;
50   parameter df10_a = 6'b100010;
51   parameter df10_b = 6'b100011;
52   parameter df11 = 6'b100100;
53   parameter df12_a = 6'b100101;
54   parameter df12_b = 6'b100110;
55   parameter df13 = 6'b100111;
56   parameter df14_a = 6'b101000;
57   parameter df14_b = 6'b101001;
58   parameter df15 = 6'b101010;
59
60
61
62
63
64
65
66   input clk;
67   input reset;
68   input flag;
69   input load;
70   input valid;
71   input scan_en;
72   input [5:0] scan_state;
73   output [5:0] state;
74   reg [5:0] state;
75
76
77   reg [5:0] next_state;
78
79   always@(posedge clk or posedge reset)
80   begin
81     if(reset)
82         state <= IDLE;
83     else if(scan_en)
84       state <= scan_state;
85     else
86       state <= next_state;
87   end
88
89   always@(state or load or valid or flag
       )
90   begin
91     case(state)
92       IDLE:if(load)
93           begin
94             next_state = LOAD0;
95           end
96           else
97           begin
98             next_state = IDLE;
99           end
100      LOAD0: next_state = LOAD1;
101      LOAD1: next_state = LOAD2;
102      LOAD2: next_state = LOAD3;
103      LOAD3: next_state = LOAD4;
104      LOAD4: next_state = LOAD5;
105      LOAD5: next_state = LOAD6;
106      LOAD6: next_state = LOAD7;
107      LOAD7: next_state = LOAD8;
108      LOAD8: next_state = LOAD9;
109      LOAD9: next_state = LOAD10;
110      LOAD10: next_state = LOAD11;
111      LOAD11: next_state = LOAD12;
112      LOAD12: next_state = LOAD13;
113      LOAD13: next_state = LOAD14;
114      LOAD14: next_state = LOAD15;
115      LOAD15: next_state = LOAD16;
116      LOAD16: next_state = IDLE1;
117      IDLE1:if(valid)
```

```
                begin
                  next_state = df0_a;
                end
                else
                begin
                  next_state = IDLE1;
                end
        df0_a:next_state = df0_b;
        df0_b:next_state = df1_a;
        df1_a:next_state = df1_b;
        df1_b:next_state = df2_a;
        df2_a:next_state = df2_b;
          df2_b:next_state = df3;
        df3:next_state = flag?df4_a:df4_b;
            //R1
        df4_a,df4_b:next_state = df5;
        df5:next_state = flag?df6_a:df6_b;
            //R2
          df6_a,df6_b:next_state = df7;
          df7:next_state = flag?df8_a:
              df8_b; //R3
          df8_a,df8_b:next_state = df9;
          df9:next_state = flag?df10_a:
              df10_b; //R4
          df10_a,df10_b:next_state = df11;
          df11:next_state = flag?df12_a:
              df12_b; //R5
          df12_a,df12_b:next_state = df13;
          df13:next_state = flag?df14_a:
              df14_b; //R6
          df14_a,df14_b:next_state = df15;

          df15:next_state = IDLE1;
        default:next_state = IDLE;
        endcase
    end
endmodule
```

A.3. FSM testbench

A.4. Cordic memory

```
//Verilog HDL for "fc_example", "
    Cordic_memory" "functional"


module Cordic_memory ( DIN, DOUT, ADR,
    clk, wr_en, rd_en );

    input         clk;
    input         wr_en;
    input         rd_en;
    input  [31:0] DIN;
    input  [31:0] ADR;
    output [31:0] DOUT;
// RAM
    reg    [31:0] ram [0:1023];
    reg    [31:0] DOUT;

   wire [31:0] ram_add0 = ram[0];
   wire [31:0] ram_add1 = ram[1];
   wire [31:0] ram_add2 = ram[2];
   wire [31:0] ram_add3 = ram[3];
   wire [31:0] ram_add4 = ram[4];
   wire [31:0] ram_add5 = ram[5];
   wire [31:0] ram_add6 = ram[6];
   wire [31:0] ram_add7 = ram[7];
   wire [31:0] ram_add8 = ram[8];
   wire [31:0] ram_add9 = ram[9];


    always @ (posedge clk)
      begin
        if (wr_en)
          ram[ADR[9:0]] <= DIN;
        else
          if (rd_en)
            DOUT <= ram[ADR[9:0]];

      end // always

      initial $readmemh("../
          mem_data_cordic.txt",ram);

endmodule
```

A.5. Cordic controllogic

```

//Verilog HDL for "fc_example", "
    Cordic_controllogic_mux" "functional"


module Cordic_controllogic (

   state,

   // output
   ready,
   ready_df,
   ram_rd_en,
   ram_wr_en,
   EDB_to_DIN,
   A_to_R0,
   A_to_R1,
   A_to_R2,
   A_to_R3,
   A_to_R4,
   A_to_R5,
```

```
    A_to_R6,
    A_to_R7,
    A_to_R10,
    A_to_R11,
    MUL_flag,
    C_to_R9,


    alu1_en,
    alu_en,
    mac1_alu_en,
    mac1_mul_en,
    mac_alu_en,
    mac_mul_en,
    shifter_en,

    M_A,
    M_B,
    M_C,
    M_E,
    M_F
);
    parameter IDLE = 6'b000000;
    parameter LOAD0 = 6'b000001;
    parameter LOAD1 = 6'b000010;
    parameter LOAD2 = 6'b000011;
    parameter LOAD3 = 6'b000100;
    parameter LOAD4 = 6'b000101;
    parameter LOAD5 = 6'b000110;
    parameter LOAD6 = 6'b000111;
    parameter LOAD7 = 6'b001000;
    parameter LOAD8 = 6'b001001;
    parameter LOAD9 = 6'b001010;
    parameter LOAD10 = 6'b001011;
    parameter LOAD11 = 6'b001100;
    parameter LOAD12 = 6'b001101;
    parameter LOAD13 = 6'b001110;
    parameter LOAD14 = 6'b001111;
    parameter LOAD15 = 6'b010000;
    parameter LOAD16 = 6'b010001;
    parameter IDLE1 = 6'b010010;
    parameter df0_a = 6'b010011;
    parameter df0_b = 6'b010100;
    parameter df1_a = 6'b010101;
    parameter df1_b = 6'b010110;
    parameter df2_a = 6'b010111;
    parameter df2_b = 6'b101011; //added
    parameter df3 = 6'b011000;
    parameter df4_a = 6'b011001;
    parameter df4_b = 6'b011010;
    parameter df5 = 6'b011011;
    parameter df6_a = 6'b011100;
    parameter df6_b = 6'b011101;
    parameter df7 = 6'b011110;
    parameter df8_a = 6'b011111;
    parameter df8_b = 6'b100000;
    parameter df9 = 6'b100001;
    parameter df10_a = 6'b100010;
    parameter df10_b = 6'b100011;
    parameter df11 = 6'b100100;
    parameter df12_a = 6'b100101;
    parameter df12_b = 6'b100110;
    parameter df13 = 6'b100111;
    parameter df14_a = 6'b101000;
    parameter df14_b = 6'b101001;
    parameter df15 = 6'b101010;


    input [5:0] state;

    output ready;
    reg ready;

    output ready_df;
    reg ready_df;

    output ram_rd_en;
    reg ram_rd_en;

    output ram_wr_en;
    reg ram_wr_en;


    output EDB_to_DIN;
    reg EDB_to_DIN;

    output A_to_R0;
    reg A_to_R0;

    output A_to_R1;
    reg A_to_R1;

    output A_to_R2;
    reg A_to_R2;

    output A_to_R3;
    reg A_to_R3;

    output A_to_R4;
    reg A_to_R4;

    output A_to_R5;
    reg A_to_R5;

    output A_to_R6;
    reg A_to_R6;

    output A_to_R7;
    reg A_to_R7;

    output A_to_R10;
    reg A_to_R10;

    output A_to_R11;
    reg A_to_R11;
```

```
137
138     output C_to_R9;
139     reg C_to_R9;
140
141     output MUL_flag;
142     reg MUL_flag;
143
144
145     output [1:0] M_A;
146     reg [1:0] M_A;
147
148     output [2:0] M_C;
149     reg [2:0] M_C;
150
151     output [2:0] M_F;
152     reg [2:0] M_F;
153
154     output M_B;
155     reg M_B;
156
157     output M_E;
158     reg M_E;
159
160     output alu1_en;
161     output alu_en;
162     output mac1_alu_en;
163     output mac1_mul_en;
164     output mac_alu_en;
165     output mac_mul_en;
166     output shifter_en;
167
168     reg alu1_en;
169     reg alu_en;
170     reg mac1_alu_en;
171     reg mac1_mul_en;
172     reg mac_alu_en;
173     reg mac_mul_en;
174     reg shifter_en;
175
176
177     always@(state)
178     begin
179         ready = 0;
180         ready_df = 0;
181         ram_rd_en = 0;
182         ram_wr_en = 0;
183
184         EDB_to_DIN = 0;
185         A_to_R0 = 0;
186         A_to_R1 = 0;
187         A_to_R2 = 0;
188         A_to_R3 = 0;
189         A_to_R4 = 0;
190         A_to_R5 = 0;
191         A_to_R6 = 0;
192         A_to_R7 = 0;
193         A_to_R10 = 0;
194         A_to_R11 = 0;
195         C_to_R9 = 0;
196         MUL_flag = 1'b0;
197         alu1_en = 0; //E+F
198         alu_en = 1;  //C+D
199         mac1_alu_en = 0; //B+A*C
200         mac1_mul_en = 1;
201         mac_alu_en = 0; //A+B*C
202         mac_mul_en = 1;
203         shifter_en = 0; //C -> C
204         M_A = 2'b00;
205         M_B = 1'b0;
206         M_C = 3'b001;
207         M_E = 1'b0;
208         M_F = 3'b000;
209
210
211
212         case(state)
213         IDLE:begin
214           ready = 1;
215         end
216         LOAD0:begin//1
217           M_C = 3'b001;
218
219           ram_rd_en = 1'b1;
220         end
221         LOAD1:begin
222           EDB_to_DIN = 1'b1; //DIN = EDB
223           M_C = 3'b001;
224
225
226
227         end
228         LOAD2:begin
229           M_C = 3'b011; // C = ALU
230           M_A = 2'b00; // A = DIN
231           A_to_R0 = 1'b1; // R0 = A
232           C_to_R9 = 1'b1; // R9 <= C
233
234           ram_rd_en = 1'b1;
235         end
236         LOAD3:begin
237           EDB_to_DIN = 1'b1; //DIN = EDB
238
239
240           M_C = 3'b101;
241
242         end
243         LOAD4:begin
244           M_A = 2'b00; // A = DIN
245           A_to_R1 = 1'b1;
246           M_C = 3'b011;
247           C_to_R9 = 1'b1;
248
249           ram_rd_en = 1'b1;
250         end
251         LOAD5:begin
252           EDB_to_DIN = 1'b1;
```

```
253
254           M_C = 3'b101;
255
256         end
257         LOAD6:begin
258           M_C = 3'b011;
259           M_A = 2'b00; // A = DIN
260           A_to_R2 = 1'b1;
261           C_to_R9 = 1'b1;
262
263           ram_rd_en = 1'b1;
264         end
265         LOAD7:begin
266           EDB_to_DIN = 1'b1;
267
268           M_C = 3'b101;
269
270         end
271         LOAD8:begin
272           M_C = 3'b011;
273           M_A = 2'b00; // A = DIN
274           A_to_R3 = 1'b1;
275           C_to_R9 = 1'b1;
276
277           ram_rd_en = 1'b1;
278         end
279         LOAD9:begin
280           EDB_to_DIN = 1'b1;
281
282           M_C = 3'b101;
283
284         end
285         LOAD10:begin
286           M_C = 3'b011;
287           M_A = 2'b00; // A = DIN
288           A_to_R4 = 1'b1;
289           C_to_R9 = 1'b1;
290
291           ram_rd_en = 1'b1;
292         end
293         LOAD11:begin
294
295           EDB_to_DIN = 1'b1;
296
297           M_C = 3'b101;
298
299         end
300         LOAD12:begin
301           M_A = 2'b00; // A = DIN
302           A_to_R5 = 1'b1;
303           M_C = 3'b011;
304           C_to_R9 = 1'b1;
305
306           ram_rd_en = 1'b1;
307         end
308         LOAD13:begin
309           EDB_to_DIN = 1'b1;
310
311           M_C = 3'b101;
312
313         end
314         LOAD14:begin
315           M_C = 3'b011;
316
317           M_A = 2'b00; // A = DIN
318           A_to_R6 = 1'b1;
319           C_to_R9 = 1'b1;
320
321           ram_rd_en = 1'b1;
322         end
323         LOAD15:begin
324           EDB_to_DIN = 1'b1;
325
326           M_C = 3'b101;
327
328         end
329         LOAD16:begin
330           M_A = 2'b00; // A = DIN
331
332
333           A_to_R7 = 1'b1;
334           M_C = 3'b011;
335           C_to_R9 = 1'b1;
336         end
337         IDLE1:begin
338           ready_df = 1'b1;
339           M_C = 3'b101;
340         end
341         df0_a:begin
342           M_C = 3'b101;
343
344           ram_rd_en = 1'b1;
345
346
347         end
348
349
350         df0_b:begin
351           EDB_to_DIN = 1'b1;
352           M_C = 3'b011;
353           C_to_R9 = 1'b1;
354
355
356         end
357         df1_a:begin  //15
358           M_A = 2'b00; // A = DIN
359           M_C = 3'b101;
360           A_to_R10 = 1'b1;
361
362           ram_rd_en = 1'b1;
363
364         end
365         df1_b:begin  //16
366           EDB_to_DIN = 1'b1;
367           M_A = 2'b00; // A = DIN
368           M_B = 1'b0; //R10 x R11 y
```

```
369           M_C = 3'b100;  //R7->C
370           M_F = 3'b000; //R0->F
371           M_E = 1'b0; //K0->E
372           shifter_en = 1'b1;
373
374         end
375         df2_a:begin //17
376           M_A = 2'b00; // A = DIN
377           A_to_R11 = 1'b1;
378           alu1_en = 1'b1;
379           M_B = 1'b0; //R10 x R11 y
380           M_C = 3'b100;  //R7->C
381           M_E = 1'b1; //ALU1->E
382           M_F = 3'b000; //R0->F
383         end
384
385         df2_b:begin
386
387           mac1_alu_en = 1'b1;
388           mac_alu_en = 1'b1;
389           M_A = 2'b00; // A = DIN
390           M_B = 1'b0; //R10 x R11 y
391           M_C = 3'b010; //shift->C
392           M_E = 1'b1; //ALU1->E
393           M_F = 3'b000; //R0->F
394
395
396         end
397
398         df3:begin //18
399           M_A = 2'b10;
400           M_B = 1'b1;
401           M_C = 3'b010; //shift->C
402           M_E = 1'b1; //ALU1->E
403           shifter_en = 1'b1;
404           M_F = 3'b001;
405         end
406
407         df4_a:begin
408           MUL_flag = 1'b0;
409           mac1_alu_en = 1'b1;
410           mac_alu_en = 1'b1;
411           alu1_en = 1'b1;
412           M_A = 2'b10;
413           M_B = 1'b1;
414           M_C = 3'b010; //shift->C
415           M_E = 1'b1; //ALU1->E
416           M_F = 3'b001;
417         end
418
419         df4_b:begin
420           MUL_flag = 1'b1;
421           mac1_alu_en = 1'b1;
422           mac_alu_en = 1'b1;
423           alu1_en = 1'b1;
424           M_A = 2'b10;
425           M_B = 1'b1;
426           M_C = 3'b010; //shift->C
427           M_E = 1'b1; //ALU1->E
428           M_F = 3'b001;
429         end
430
431         df5:begin //19
432           M_A = 2'b10;
433           M_B = 1'b1;
434           M_C = 3'b010; //shift->C
435           M_E = 1'b1; //ALU1->E
436           shifter_en = 1'b1;
437           M_F = 3'b010;
438         end
439
440         df6_a:begin
441           MUL_flag = 1'b0;
442           mac1_alu_en = 1'b1;
443           mac_alu_en = 1'b1;
444           alu1_en = 1'b1;
445           M_A = 2'b10;
446           M_B = 1'b1;
447           M_C = 3'b010; //shift->C
448           M_E = 1'b1; //ALU1->E
449           M_F = 3'b010; //R1->F
450         end
451
452         df6_b:begin
453           MUL_flag = 1'b1;
454           mac1_alu_en = 1'b1;
455           mac_alu_en = 1'b1;
456           alu1_en = 1'b1;
457           M_A = 2'b10;
458           M_B = 1'b1;
459           M_C = 3'b010; //shift->C
460           M_E = 1'b1; //ALU1->E
461           M_F = 3'b010;
462         end
463
464
465         df7:begin //19
466           M_A = 2'b10;
467           M_B = 1'b1;
468           shifter_en = 1'b1;
469           M_C = 3'b010; //shift->C
470           M_E = 1'b1; //ALU1->E
471           M_F = 3'b010;
472
473         end
474         df8_a:begin
475           M_A = 2'b10;
476           M_B = 1'b1;
477           MUL_flag = 1'b0;
478           mac1_alu_en = 1'b1;
479           mac_alu_en = 1'b1;
480           alu1_en = 1'b1;
481           M_C = 3'b010; //shift->C
482           M_E = 1'b1; //ALU1->E
483           M_F = 3'b011; //R1->F
484         end
```

```
485
486         df8_b:begin
487           M_A = 2'b10;
488           M_B = 1'b1;
489           MUL_flag = 1'b1;
490           mac1_alu_en = 1'b1;
491           mac_alu_en = 1'b1;
492           alu1_en = 1'b1;
493           M_C = 3'b010; //shift->C
494           M_E = 1'b1; //ALU1->E
495           M_F = 3'b011; //R1->F
496         end
497         df9:begin
498           M_A = 2'b10;
499           M_B = 1'b1;
500           M_C = 3'b010; //shift->C
501           M_E = 1'b1; //ALU1->E
502           shifter_en = 1'b1;
503           M_F = 3'b010;
504         end
505         df10_a:begin
506           M_A = 2'b10;
507           M_B = 1'b1;
508           MUL_flag = 1'b0;
509           mac1_alu_en = 1'b1;
510           mac_alu_en = 1'b1;
511           alu1_en = 1'b1;
512           M_C = 3'b010; //shift->C
513           M_E = 1'b1; //ALU1->E
514           M_F = 3'b100; //R1->F
515         end
516
517         df10_b:begin
518           M_A = 2'b10;
519           M_B = 1'b1;
520           MUL_flag = 1'b1;
521           mac1_alu_en = 1'b1;
522           mac_alu_en = 1'b1;
523           alu1_en = 1'b1;
524           M_C = 3'b010; //shift->C
525           M_E = 1'b1; //ALU1->E
526           M_F = 3'b100; //R1->F
527         end
528
529         df11:begin //19
530           M_A = 2'b10;
531           M_B = 1'b1;
532           shifter_en = 1'b1;
533           M_C = 3'b010; //shift->C
534           M_E = 1'b1; //ALU1->E
535           M_F = 3'b010;
536
537         end
538         df12_a:begin
539           M_A = 2'b10;
540           M_B = 1'b1;
541           MUL_flag = 1'b0;
542           mac1_alu_en = 1'b1;
543           mac_alu_en = 1'b1;
544           alu1_en = 1'b1;
545           M_C = 3'b010; //shift->C
546           M_E = 1'b1; //ALU1->E
547           M_F = 3'b101;
548         end
549
550         df12_b:begin
551           M_A = 2'b10;
552           M_B = 1'b1;
553           MUL_flag = 1'b1;
554           mac1_alu_en = 1'b1;
555           mac_alu_en = 1'b1;
556           alu1_en = 1'b1;
557           M_C = 3'b010; //shift->C
558           M_E = 1'b1; //ALU1->E
559           M_F = 3'b101;
560         end
561
562         df13:begin
563           M_A = 2'b10;
564           M_B = 1'b1;
565           shifter_en = 1'b1;
566           M_C = 3'b010; //shift->C
567           M_E = 1'b1; //ALU1->E
568           M_F = 3'b010;
569
570         end
571         df14_a:begin
572           M_A = 2'b10;
573           M_B = 1'b1;
574           MUL_flag = 1'b0;
575           mac1_alu_en = 1'b1;
576           mac_alu_en = 1'b1;
577           alu1_en = 1'b1;
578           M_C = 3'b010; //shift->C
579           M_E = 1'b1; //ALU1->E
580           M_F = 3'b110; //R1->F
581         end
582
583         df14_b:begin
584           M_A = 2'b10;
585           M_B = 1'b1;
586           MUL_flag = 1'b1;
587           mac1_alu_en = 1'b1;
588           mac_alu_en = 1'b1;
589           alu1_en = 1'b1;
590           M_C = 3'b010; //shift->C
591           M_E = 1'b1; //ALU1->E
592           M_F = 3'b110; //R1->F
593         end
594
595
596
597         df15:begin
598           M_A = 2'b10;
599           M_B = 1'b1;
600           mac1_alu_en = 1'b1;
```

```
601         mac_alu_en = 1'b1;
602         M_E = 1'b1; //ALU1->E
603         M_F = 3'b110; //R1->F
604         ram_wr_en = 1'b1;
605         M_C = 3'b000;
606       end
607
608       endcase
609   end
610 endmodule
```

A.6. Cordic top testbench

```
1  //Verilog HDL for "fc_example", "
       Cordic_top_tb" "functional"
2
3
4  module fc_example_tb;
5
6
7  parameter CYCLE = 20;
8  reg clk;
9  reg reset;
10 reg start;
11
12
13
14 initial
15 begin
16   clk <= 1'b0;
17   reset <= 1'b0;
18   #CYCLE
19   reset <= 1'b1;
20   #CYCLE
21   reset <= 1'b0;
22 end
23
24 always#(CYCLE/2)
25   clk <= ~clk;
26
27
28
29
30 wire [31:0] EAB;
31 wire [31:0] EDB_IN;
32 wire [31:0] EDB_OUT;
33 wire ram_wr_en;
34 wire ram_rd_en;
35 wire ready;
36 wire ready_df;
37 reg load;
38 reg valid;
39 wire scan_en;
40 wire [5:0] scan_state;
41
42 //RAM
43 Cordic_memory MEM(
44    .ADR(EAB),
45    .DIN(EDB_OUT),
46    .DOUT(EDB_IN),
47    .clk(clk),
48    .wr_en(ram_wr_en),
49    .rd_en(ram_rd_en)
50 );
51
52
53 fc_example Cordic_top(
54   .clk(clk),
55   .reset(reset),
56   .EAB(EAB),
57   .EDB_IN(EDB_IN),
58   .EDB_OUT(EDB_OUT),
59   .ram_wr_en(ram_wr_en),
60   .ram_rd_en(ram_rd_en),
61   .ready(ready),
62   .ready_df(ready_df),
63   .load(load),
64   .valid(valid),
65   .scan_en(scan_en),
66   .scan_state(scan_state)
67 );
68
69
70 initial
71 begin
72   load = 0;
73   valid = 0;
74   #(10*CYCLE)
75   load = 1;
76   #(CYCLE)
77   load = 0;
78   #(20*CYCLE)
79   valid = 1;
80   #(CYCLE)
81   valid = 0;
82 end
83
84 endmodule
```