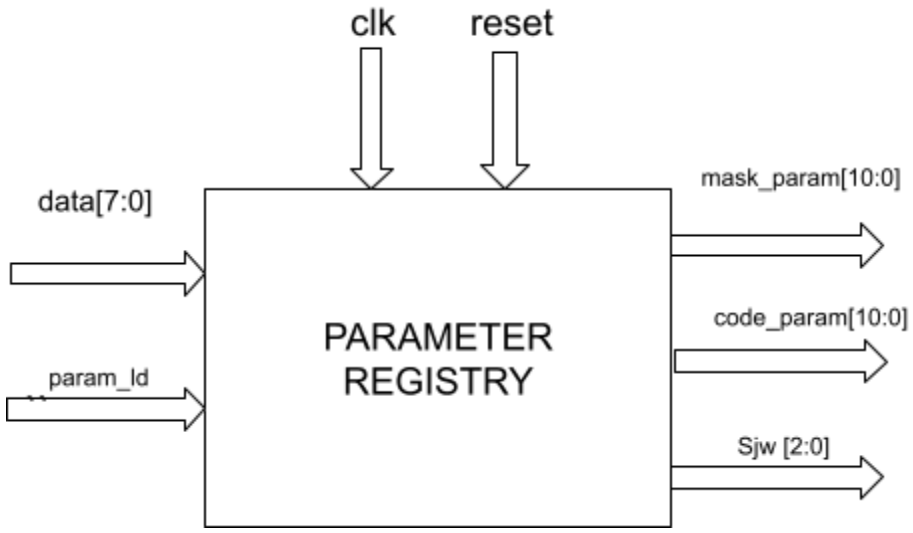
AHP DIGITAL SYSTEM DESIGN -2023

List of projects

1) Parameter Registry block
2) Transfer buffer (Tx buffer)
3) Parallel to serial convertor
4) Single cycle RISC-V processor

1) Parameter Registry block



Registry block implements a finite state machine (FSM) model for managing parameter loading and manipulation based on input data.

Functionality

Input and Output:
The module takes several inputs, including a clock signal (clk), a global reset signal (reset), an 8-bit data input (data_in), and a parameter load signal (param_ld).
It provides outputs such as a 11-bit mask parameter (mask_param), an 11-bit code parameter (code_param), and a 2-bit synchronization jump width (sjw).

Finite State Machine:
The module implements a finite state machine with the following states:
idle
prmtr_0
prmtr_1
prmtr_2
Prmtr_ld_comp

Parameter Loading and Manipulation:
The primary purpose of the module is in handling parameter loading and manipulation. Here's a high-level description of how this process works:

The module starts in the idle state, waiting for a parameter load (param_ld) signal to be asserted.
When the param_ld signal is detected, the module transitions to the prmtr_0 state.
In the prmtr_0 state, the module stores the least significant 8 bits of the incoming data_in as the mask parameter (mask_param).
The module then transitions to the prmtr_1 state, where it further manipulates the mask_param and code_param values based on the incoming data_in.
The prmtr_1 state transitions to the prmtr_2 state, where the code parameter (code_param) is further manipulated using bits from the data_in.
The prmtr_2 state also modifies the synchronization jump width (sjw) using specific bits from the data_in.
Finally, the module transitions to the prmtr_ld_comp state, where it waits for another parameter load signal to return to the prmtr_0 state and repeat the parameter loading process.

Outputs at each state of FSM

idle :
Initial state where all outputs (mask_param, code_param, sjw) are set to zero.
No parameter manipulation occurs in this state.
Serves as the starting point of the FSM and the state where the FSM returns after completing a parameter loading cycle.

prmtr_0 :
Stores the least significant 8 bits of data_in as mask_param the remaining msb bits are concatenated with zeros..
code_param and sjw outputs retain their values from the previous state (idle).

prmtr_1 :

Modifies the 8-bit mask_param using bits [2:0] of
data_in(mask_param=data_reg[2:0]+mask_param[7:0])
Stores bits [7:3] of data_in as the least significant 5 bits of code_param remaining msb bits are concatenated with zeros.

sjw output remains unchanged.

prmtr_2 :
Stores the least significant 6 bits of data_in as the MSB 6 bits of code_param and the remaining 5 bits of code_param remains the same in the LSB bits.
Uses bits [7:6] of data_in to set the sjw output.
mask_param output remains unchanged.

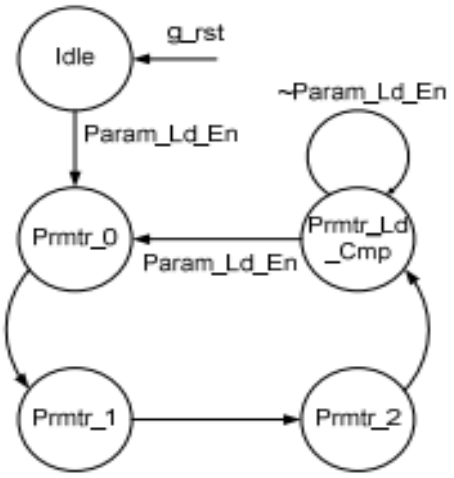
prmtr_ld_comp :
Maintains the outputs (mask_param, code_param, sjw) from the previous state (prmtr_2).
FSM waits in this state until the parameter load signal (param_ld) becomes active again.
Upon receiving the signal, transitions back to the prmtr_0 state to initiate a new parameter loading cycle.
default:

Initialization and Reset:
Upon a global reset (reset), the module sets its internal registers to initial values, and output parameters (mask_param, code_param, sjw) are reset to zero.
The FSM state is also reset to the idle state on global reset.

(Suggested coding style to prevent metastabiltiy):

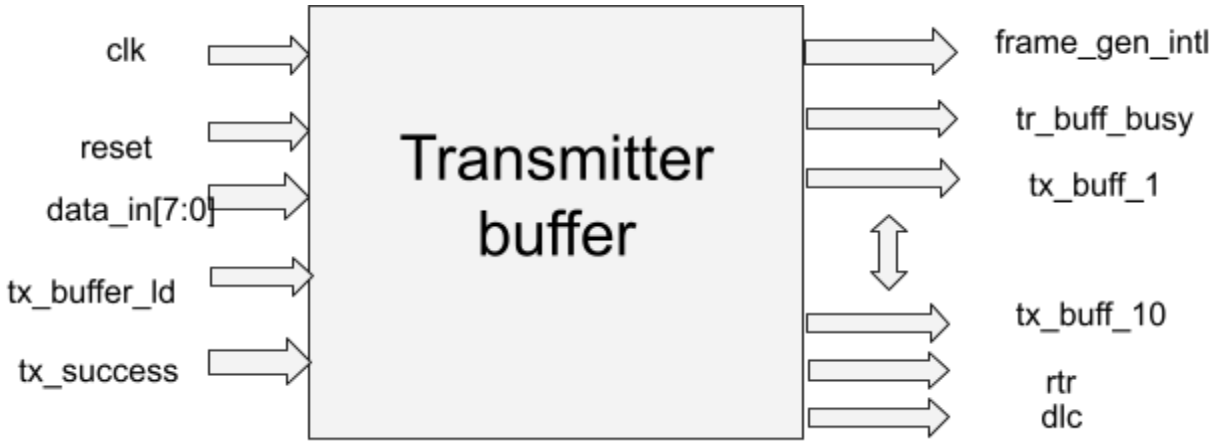
Synchronization of Data Input:
The incoming data (data_in) is synchronized to the clock signal (clk) using two flip-flops (eg:-data_reg_in and data_reg).

Summary :this design module handle the loading and manipulation of parameters using a finite state machine approach. It's a part of a larger system or hardware design that involves configuring and controlling various parameters based on external input.



FSM -PARAMETER REGISTRY

2) Transmitter buffer



The tx_buff digital block is a fundamental module that plays a vital role in managing the transmission of data within digital communication systems. It is designed to facilitate the efficient and controlled transfer of data from a source (such as a host controller) to a destination (potentially a communication

interface). This module ensures that data is organized, synchronized, and transmitted reliably.

Functionality

Inputs and Outputs:

clk: The clock signal that drives the operation of the module.
reset: The global reset signal, which initializes the module when active.
data_in: An 8-bit input representing the data to be transmitted.
tx_buff_ld: A control signal to trigger the loading of data into the transmit buffers.
tx_success: A signal indicating the successful completion of a transmission.
frame_gen_intl: An output indicating whether the buffer is full (1) or empty (0).
tx_buff_busy: An output indicating whether the transmit buffer is currently loaded.
tx_buff_1 to tx_buff_10: Ten 8-bit output signals representing individual transmit buffers.
rtr: An output signal indicating the remote transfer request bit.
dlc: A 4-bit output signal representing the data length code.

Buffer Management and State Machine:

The core of this module is a state machine that manages the sequential loading and transmission of data.
The state machine progresses through various states, such as idle, buffer loading states (buf0 to buf9), and a completion state (buf_ld_comp).
The state variable dictates the current state of the module's operation.

Output Control:

The module maintains control signals and data for each buffer, such as tx_buff_busy, which indicates whether a buffer is loaded.
Depending on the state, the module populates individual transmit buffers and updates control signals.
State Transitions:

The state transitions occur based on different conditions, like the availability of data, buffer loading, and transmission success.
The state machine guides the flow of the module through its various stages, ensuring proper data handling.
Remote Transfer Request (RTR) and Data Length Code (DLC):

The module extracts the Remote Transfer Request bit and Data Length Code from the incoming data, if applicable.
These control signals and codes are used in various communication protocols.

Refer the below FSM diagram for state machine coding

Outputs of each FSM :

Reset (reset is active):

During a global reset, all the outputs and internal signals are reset to initial values:
frame_gen_intl is set to 0 (buffer is not full).
tx_buff_busy is set to 0 (buffer is not loaded).
tx_buff_1 to tx_buff_10 are all set to 8-bit zero values.
rtr is set to 0 (no remote transfer request).
dlc is set to 0 (data length code is reset).
This ensures that all outputs are in a known state when the system is reset.

State: Idle (state is idle):
When in the idle state, the module waits for data to be loaded into the transmit buffer.
Outputs are reset to initial values to prepare for new data:
frame_gen_intl, tx_buff_busy, and rtr are all set to 0.
tx_buff_1 to tx_buff_10 are all set to 8-bit zero values.
dlc is set to 0.
This state ensures that the module is ready to load new data.

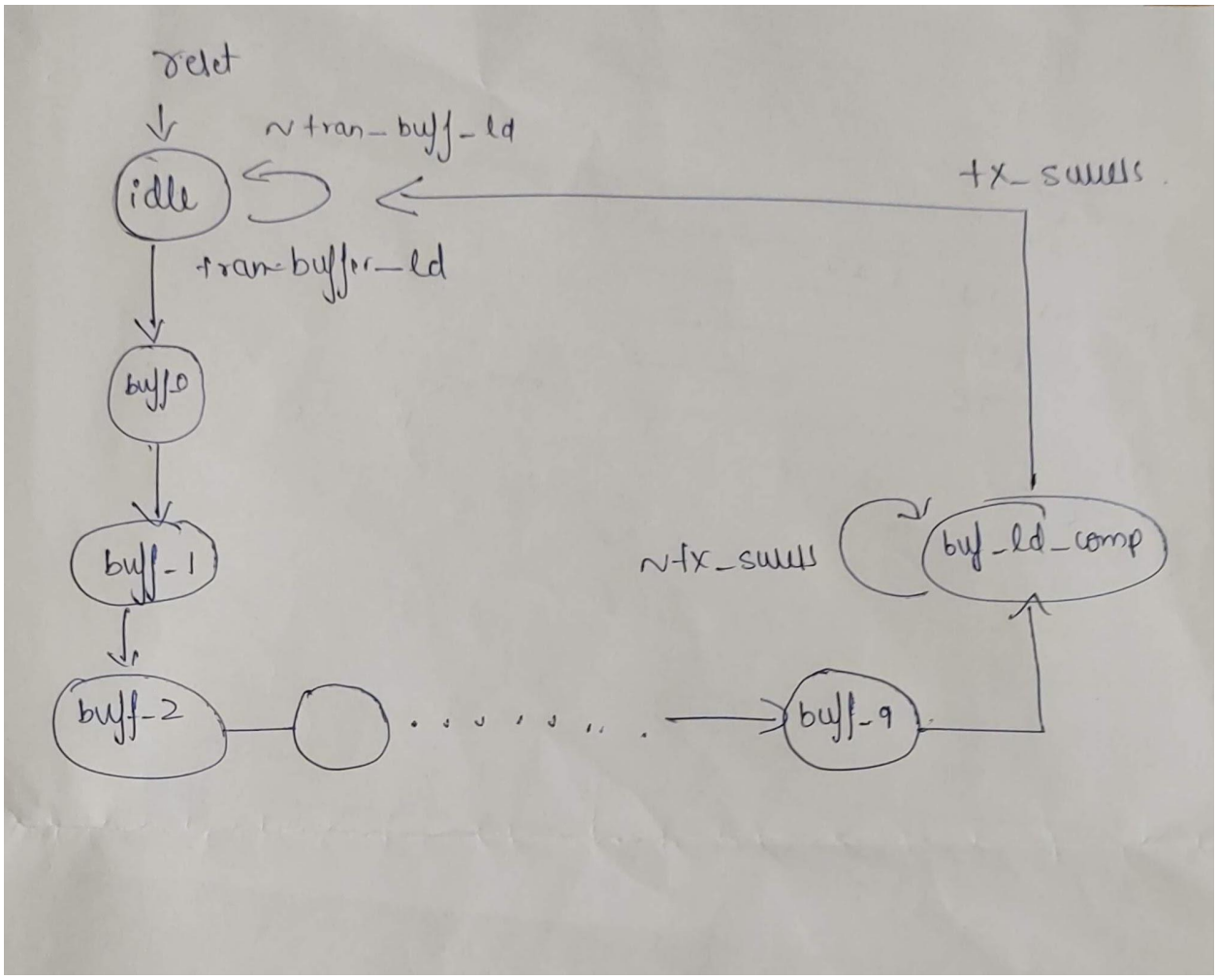
State: Buffer 0 (state is buf0):
Data from data_reg (synchronized input data) is loaded into tx_buff_1.

State: Buffer 1 (state is buf1):
tx_buff_1 retains its value from the previous state.
Data from data_reg is loaded into tx_buff_2.
The remote transfer request (rtr) is updated with bit 4 of the data.
The data length code (dlc) is updated with bits 3 to 0 of the data.

State: Buffer 2 to Buffer 9 (state is buf2 to buf9):
Similar to previous buffer states, the current tx_buff_x retains its value.
Data from data_reg is loaded into the next buffer (tx_buff_x+1).
This sequence continues until buffer 9.

State: Buffer Load Complete (state is buf_ld_comp):
In this state, the last buffer (tx_buff_10) retains its value.
tx_buff_busy is set to 1, indicating that the buffer is now loaded.
frame_gen_intl is set to 0, indicating that the buffer is not full.
Default State:

If the state machine encounters an undefined state, all outputs are reset to initial values as in the global reset condition:
frame_gen_intl, tx_buff_busy, and rtr are set to 0.
tx_buff_1 to tx_buff_10 are set to 8-bit zero values.
dlc is set to 0.
Summary:
The module starts in the idle state, ready to load data into the transmit buffer.
As data is loaded, it progresses through the buffer states (buf0 to buf9), updating the transmit buffers accordingly.
When all buffers are loaded, the system enters the buf_ld_comp state, indicating that the buffer is fully loaded and ready for transmission.
If a transmission is successful (tx_success signal), the system returns to the idle state, resetting outputs for the next cycle.



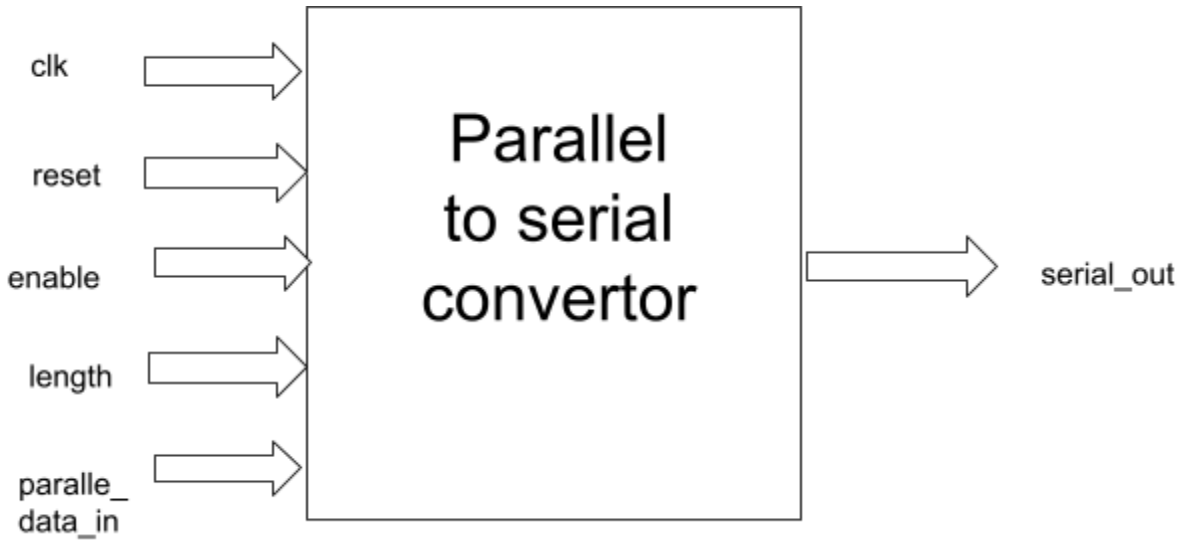
FSM DIAGRAM

(Suggested coding style to prevent metastability):

Synchronization of Data Input:
The incoming data:
Data_in is synchronized to the clock signal (clk) using two flip-flops (eg:-data_reg_in and data_reg).

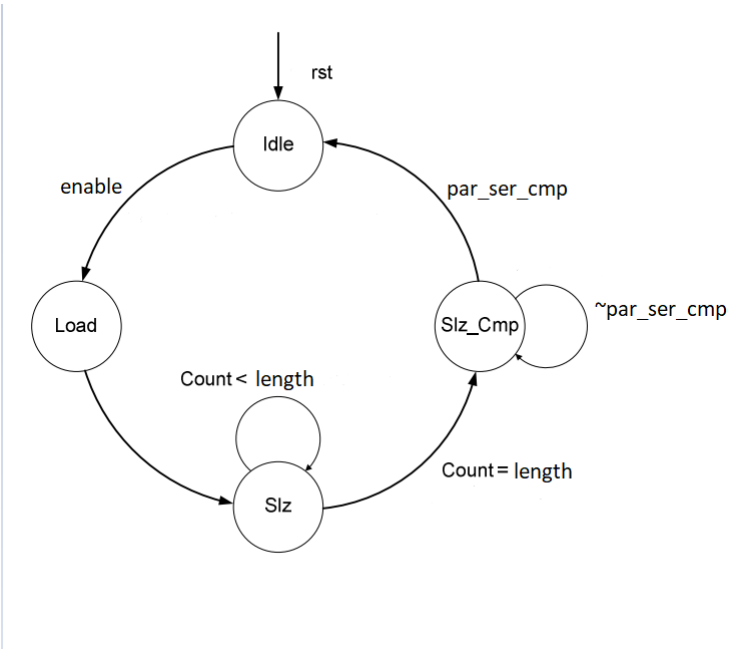
tx_buff_ld signal which enables the loading of data into the transmit buffers can be synchronized to the clock signal (clk),using two flip-flops (eg:-tx_buff_ld_en_in and tx_buff_ld_en) can manage the buffer loading process.

3) Parallel to Serial Convertor



- Serialization is the process of converting parallel data into a serial stream of bits, one after another.
- The functionality is to transmit the bits of **‘parallel_data_in’** in serial order, starting from the most significant bit (MSB) and shifting them out one by one to **‘serial_out’.**
- The process of serialization typically involves a loop that iteratively shifts out bits from the **‘parallel_input_data’**. This is implemented in the form of an FSM.
- The FSM involves 4-states : **idle, load, serialize, serialize_comp;**
- The signals and their roles:
    - clk - System Clock;
    - reset - System Reset;
    - enable - initializes the parallel to serial conversion functionality within the converter block;
    - length - Conveys the length of the input parallel data to be serialized;
    - parallel_data_in - Data input meant to be serialized;
    - serial_out - Serialized output;

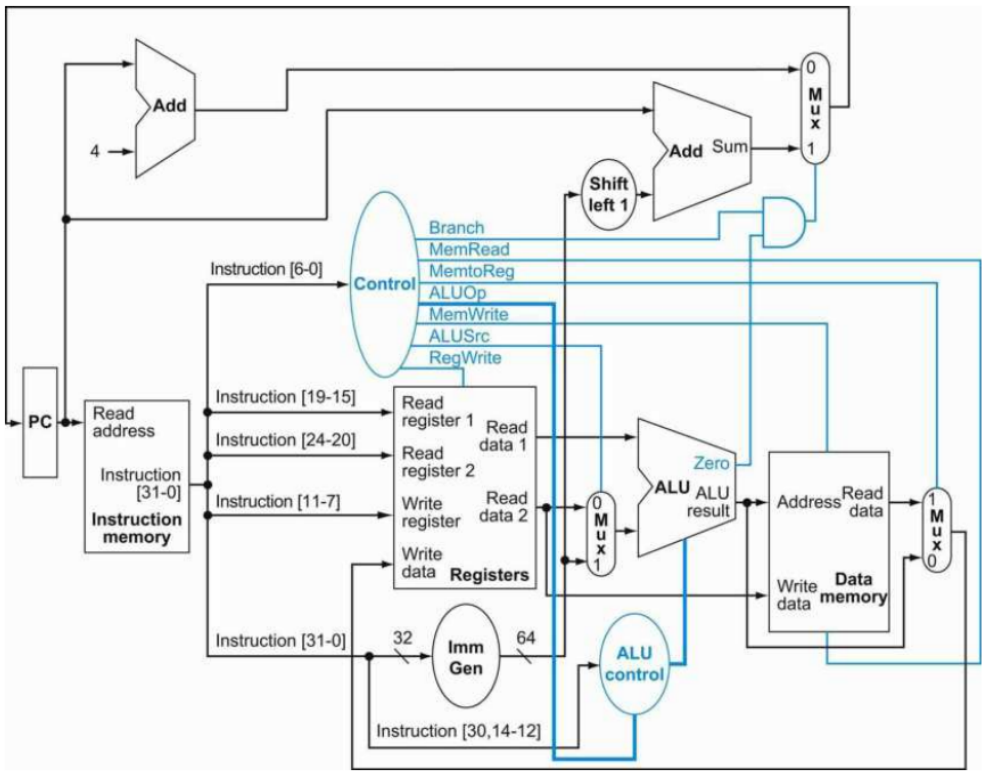
- par_ser_cmp - the signal is high when the serialization of a parallel data is completed.(intermediate signal for fsm)



Defining the FSM-states internally:

- Idle: Waits for the enable signal; When the enable signal is on, the state transitions from idle state to load state.
- Load: The input parallel data is loaded into a ***register**.
- Slz: The process of serialization takes place i.e., the MSB of the data in the ***register** is popped out and the data is left shifted. This process continues until the **‘count’** is less than the length of the input data - **‘length’.**
- Slz_Cmp: Once the serialization process is complete, the signal - ‘par_ser_cmp’ is asserted high.

4) Single cycle RISC-V processor



Phase 1: Instruction Fetch and PC Management

In this phase, you'll focus on building the components responsible for fetching instructions from memory and managing the Program Counter.

Instruction Memory (IMEM): Implement the IMEM module to store and retrieve program instructions based on the PC.

Program Counter (PC): Create the PC module to manage the memory address of the next instruction to be fetched.

PC Adder and Branch Logic (Part 1): Develop the PC adder and branch logic to handle unconditional and conditional branches' target addresses.

Phase 2: Instruction Decoding and Control Unit

This phase involves decoding fetched instructions and generating control signals for the subsequent phases.

Instruction Decoder: Design the decoder module to interpret the fetched instruction, identifying its type and operands.

Control Unit: Develop the control unit to generate control signals based on the decoded instruction, enabling subsequent stages to process the instruction correctly.

Sign-Extend Unit: Implement the sign-extend unit to extend immediate values for arithmetic operations.

Integration (Part 1): Integrate the IMEM, PC, PC Adder, branch logic, instruction decoder, and control unit modules. Test the integrated modules' interaction to ensure proper instruction fetch, decoding, and control signal generation.

Phase 3: Execution and Memory Access

In this phase, the focus shifts to executing arithmetic, logic, and memory access operations.

ALU (Arithmetic Logic Unit): Develop the ALU module to perform arithmetic and logical operations as specified by the instruction.

Register File: Implement the register file to read from and write to registers based on control signals.

Data Memory (DMEM): Design the data memory module to allow for data loading and storing operations.

PC Adder and Branch Logic (Part 2): Complete the PC adder and branch logic to handle conditional branches' calculation of branch offsets.

Phase 4: Write-Back and Integration

In this final phase, you'll complete the processor by adding the write-back stage and integrating all the components together.

Write-Back Unit: Create the write-back unit to update destination registers with the result of executed operations.

Clock Generator: Implement the clock generator to provide the clock signal for synchronization.

Reset Logic: Develop the reset logic to initialize the processor to a known state during startup.

Integration (Part 2): Integrate all the modules developed in the previous phases to create a functional single-cycle RISC-V processor.

Testing and Debugging: Thoroughly test the processor with a variety of RISC-V assembly programs to ensure correct execution of instructions and proper interaction between all components.

Documentation: Document the design, implementation, and testing processes for each module and the integrated processor.

Refer :-Computer organization and design- RISC-V edition by David A patterson for detailed study.