RMC75E Codebase Documentation

Project: RMC75E TEST BENCH
Author: Satchel Hamilton
Company: Delta Motion
Date: 8/1/2023
Last updated: August 1st, 2023

Contents

Introduction

This document provides an in-depth guide to the inner workings of the RMC75E Motion Controller, a powerful and sophisticated device developed by Delta Motion. Designed to deliver accurate control of hydraulic and electric actuators, the RMC75E has set a benchmark for motion control systems.
As one of the advanced controllers in Delta Motion's portfolio, the RMC75E is defined by its exceptional performance and versatility. The controller handles a variety of motion control tasks, including position control, pressure/force control, velocity control, and more, making it an excellent choice for industrial and high-precision applications.

This document will dive into the source code that drives the RMC75E, written in VHDL (VHSIC Hardware Description Language). VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

The RMC75E codebase is a complex and interconnected assembly of various modules. Each one has a specific role in controlling hardware, processing data, or managing operations. These modules are not only crucial to the operation of the device but also illustrate the complex tasks that the RMC75E can perform. Understanding this codebase is essential for anyone looking to modify the device's operation, add new features, or debug existing ones.

As the author of this documentation, my aim is to provide a detailed and comprehensive overview of the RMC75E's codebase. We will delve into each module, explaining their functionality, interactions, and role within the overall system. In addition, the document will also cover the system overview, hierarchy diagram, code implementation details, and the testing and verification methods used during the codebase's development.
By the end of this document, readers will have gained a thorough understanding of the code that drives the RMC75E, the design principles behind it, and the methods used to ensure its reliability and effectiveness.

This document aims to explore the intricate details of the RMC75E Motion Controller, its VHDL codebase, and the rigorous testing and verification processes that underscore Delta Motion's commitment to excellence in motion control systems.

Next, we will begin with a detailed breakdown of each module, starting with the 'Analog' module and continuing through the rest of the list. Each one of these modules contributes to the robust and effective performance of the RMC75E and understanding them is the key to fully appreciating the intricacy of this powerful yet lightweight motion controller.

Modules

Analog

- **Module Name:** Analog.vhd

- **Description:** The analog component of the RMC75E source code that performs analog data processing. The module has several input and output ports, including control signals, clock signals, data signals, and various read and write signals. It includes internal components such as "StateMachine," "Serial2Parallel," and "DataBuffer" to handle different functionalities of the ADC processing.

- **Inputs:** List all input signals used by the module along with their descriptions.

- **Outputs:** List all output signals produced by the module along with their descriptions.

- **Internal Signals:** If applicable, mention any internal signals used by the module.

- **Processes and State Machines:** State Encoding: The state encoding is defined using the **STATE_TYPE** array. Each state is assigned a binary value using the **std_logic** type. Here are the defined states:

    - constant StartState : STATE_TYPE :="000";
    - constant SampleHoldState : STATE_TYPE :="001";
    - constant ConvertState : STATE_TYPE :="011";
    - constant ConvertWaitState1 : STATE_TYPE :="010";
    - constant ConvertWaitState2 : STATE_TYPE :="110";
    - constant ConvertDoneState : STATE_TYPE :="111";
    - constant IncConvCountState : STATE_TYPE :="101";
    - constant InterConvDelayState : STATE_TYPE :="100";

    - The **State** signal is of type **STATE_TYPE** and represents the current state of the state machine.
    - Delays between Conversions: There are different delay times between conversions based on the **LoopTime** signal. The delays are defined as follows:

    - constant eighth_ms_interconversion_delay : std_logic_vector (10 downto 0) := "00000000101"; -- 5 counts
    - constant quarter_ms_interconversion_delay : std_logic_vector (10 downto 0) := "00001000000"; -- 60 counts
    - constant half_ms_interconversion_delay : std_logic_vector (10 downto 0) := "00010110100"; -- 180 counts
    - constant one_ms_interconversion_delay : std_logic_vector (10 downto 0) := "00110011100"; -- 412 counts
    - constant two_ms_interconversion_delay : std_logic_vector (10 downto 0) := "01101110001"; -- 881 counts
    - constant four_ms_interconversion_delay : std_logic_vector (10 downto 0) := "11100011100"; -- 1820 counts

- These delays are used to spread the data samples over the entire control loop period.
- Process for State Transitions: The **StateMachine_arch** process describes the state transitions and conditions. Let's analyze it step by step:
- The process is sensitive to the **SysClk** signal, which represents the system clock.
- On a rising edge of the clock, the process evaluates the state transitions and updates the corresponding signals.
- If **SysReset** or **SynchedTick** signals are asserted, the state is set to the **StartState** (reset state).
- If **SlowEnable** signal is asserted, the process enters the state machine logic.
- The state machine logic is implemented using a **case** statement based on the current state.
- Each state has different conditions and actions associated with it.
- The state transitions and corresponding assignments are as follows:

    - **StartState**: If **Converting** is asserted, the **ExpA_CS_L** signal is set to '0' (converter chip select active), and the state transitions to **SampleHoldState**.
    - If not converting, the **ExpA_CS_L** signal is set to '1' (converter chip select inactive), and the state transitions to **StartState**.
    - **SampleHoldState**:If **SampleHoldDone** is asserted, the **intSerial2ParallelEN** signal is set to '1' (converter data is being received), and the state transitions to **ConvertState**.
    - If **SampleHoldDone** is not asserted, the **ExpA_CS_L** signal remains '0' (converter chip select still active), and the state remains in **SampleHoldState**.
    - **ConvertState**:
    - If **ConversionDone** is asserted, the **ExpA_CS_L** signal is set to '1' (converter chip select turned off), the **intSerial2ParallelEN** signal is set to '0' (converter data is finished), **intWriteConversion** is set to '1', and the state transitions to **ConvertWaitState1**.
    - If **ConversionDone** is not asserted, the **ExpA_CS_L** signal remains '0' (converter chip select still active), and the **intSerial2ParallelEN** signal remains '1' (converter data is being received), and the state remains in **ConvertState**.
    - **ConvertWaitState1**: The **intWriteConversion** signal is set to '0', and the state transitions to **ConvertWaitState2**.
    - **ConvertWaitState2**: The state transitions to **ConvertDoneState**.

- **ConvertDoneState**: If **EndDelay** is not asserted, the **ConversionCounterEN** signal is set to '1', and the state transitions to **IncConvCountState**.
- If **EndDelay** is asserted, the **Serial2ParallelCLR** signal is set to '1', and the state remains in **ConvertDoneState**.
- **IncConvCountState**:
- The **ConversionCounterEN** signal is set to '0', and the state transitions to **InterConvDelayState**.
- **InterConvDelayState**:
- If **InterConversionDelayTC** is asserted, the **InterConversionDelayEN** signal is set to '0', and the state transitions to **StartState**.
- If **InterConversionDelayTC** is not asserted, the **InterConversionDelayEN** signal is set to '1', and the state remains in **InterConvDelayState**.
- Other states:
- The default case sets the state to **StartState**.

- **Key Components:** The **Analog** module is a wrapper that instantiates three other modules: **StateMachine**, **Serial2Parallel**, **DataBuffer**. Each module serves a specific purpose and contributes to the overall functionality of the design.

    - **StateMachine**: This module represents a state machine that controls the ADC conversion process. It receives inputs such as system reset (**SysReset**), system clock (**SysClk**), enable signal (**SlowEnable**), synchronized tick signal (**SynchedTick**), loop time (**LoopTime**), and ADC chip select (**ExpA_CS_L**). It generates outputs to control the ADC clock (**ExpA_CLK**), enable signal for the shift register (**Serial2ParallelEN**), clear signal for the shift register (**Serial2ParallelCLR**), and end of conversion indicator (**WriteConversion**).
    - **Serial2Parallel**: This module converts serial data to parallel data. It receives inputs such as system clock (**SysClk**), synchronized tick signal (**SynchedTick**), control axis data (**CtrlAxisData**), ADC data (**ExpA_DATA**), enable signal for conversion (**Serial2ParallelEN**), clear signal for conversion (**Serial2ParallelCLR**), and address for parallel data (**S2P_Addr**). It generates parallel data output (**S2P_Data**).
    - **DataBuffer**: This module implements a data buffer to store the converted analog data. It receives inputs such as system reset (**SysReset**), write clock (**H1_CLKWR**), system clock (**SysClk**), enable signal (**SlowEnable**), synchronized tick signal (**SynchedTick**), synchronized 60Hz tick signal (**SynchedTick60**), read control signals (**AnlgPositionRead0**, **AnlgPositionRead1**, **ExpA0ReadCh0**, **ExpA0ReadCh1**, **ExpA1ReadCh0**, **ExpA1ReadCh1**, **ExpA2ReadCh0**, **ExpA2ReadCh1**, **ExpA3ReadCh0**, **ExpA3ReadCh1**), write conversion signal (**WriteConversion**), address for parallel data (**S2P_Addr**), and parallel data input (**S2P_Data**). It generates the analog data output (**DataOut**), ADC chip select signal (**ExpA_CS_L**), and ADC clock signal (**ExpA_CLK**).
    - **Analog**: This module serves as the top-level entity that instantiates the other modules. It provides the input and output ports required for the entire design.

- **Important Calculations:** Mention any critical calculations or assignments performed by the module.
- **Interactions with Other Modules:** Explain how the module interacts with other modules in the system, if relevant.
- **Test bench and simulation:**

Clockcontrol

- **Module Name:** Clockcontrol.vhd

- **Description:** The **ClockControl** module is responsible for controlling the clock signals in the system. It interfaces with the **Clock_Gen** component, which generates the clock signals based on the input clock sources. The module takes various input and output ports, including **H1_PRIMARY**, **H1_CLKWR**, **H1_CLK**, **H1_CLK90**, **SysClk**, **RESET**, **DLL_RST**, **DLL_LOCK**, **SysRESET**, **PowerUp**, **Enable**, and **SlowEnable**.

- Inside the **ClockControl** module, there are internal signals and processes that handle the synchronization and control of clock-related operations. The module ensures proper synchronization and control of the clock signals based on the DLL lock status and system reset conditions. It coordinates the generation and distribution of clock signals to different components within the system.

- **Inputs:** List all input signals used by the module along with their descriptions.

- **Outputs:** List all output signals produced by the module along with their descriptions.

- **Internal Signals:** If applicable, mention any internal signals used by the module.

- **Key Components:** The ClockControl module uses the Clock_Gen component to generate clock signals. It contains a process to synchronize the DLL lock status (DLL_LOCK_Int) to the system clock (SysClk) and a process to synchronize the DLL reset command (DLL_RST_sync) to the system clock. It also includes a process to generate initialization pulses (PowerUpOneShot) on startup or PLL reset. The EnableCount signal is used to generate enable pulses for clocking the logic at different frequencies (Enable and SlowEnable).
- The module has commented-out sections related to a previous implementation of a state machine (StateMachine) that monitored the status of the DLL and performed certain actions based on its state. However, this state machine is no longer utilized in the current design, which uses the Clock_Gen component.

- **Important Calculations:** Mention any critical calculations or assignments performed by the module.

- **Interactions with Other Modules:** The Clockcontrol module manages the clock signal routing for the entire system. It retrieves the conditioned clock signals from the Clock_Gen module, which conditions the raw clock signals generated by the Clock_Gen_Clock_Gen_0_FCCC module.

- **Test bench and simulation:**

Clock_Gen

- **Module Name:** Clock_gen.vhd

- **Description:** The clock_gen module acts as an intermediary between the raw clock signal generation in the clock_gen_fccc module and the clock control module, which routes the final clock signals required by the system. This provides the conditioned and calibrated clock signals derived from the raw clock signals received from the clock_gen_fccc module.

- **Inputs:**

  - CLK1_PAD: The input clock signal to the clock generator.
  - PLL_ARST_N: Active-low reset signal for the PLL (Phase-Locked Loop).
  - PLL_POWERDOWN_N: Active-low signal to enable the PLL.

- **Outputs:**

  - GL0: Clock output 0.
  - GL1: Clock output 1.
  - GL2: Clock output 2.
  - LOCK: Output signal indicating the lock status of the PLL.

- **Internal Signals:**
  - GL0_net_0, GL1_net_0, GL2_net_0, LOCK_net_0: Internal signals to store clock generator outputs.
  - GL0_net_1, GL1_net_1, GL2_net_1, LOCK_net_1: Internal signals used for sequential updates of the clock outputs.
  - GND_net: Internal signal representing a constant ground (logic 0).
  - PADDR_const_net_0: Internal signal storing a constant value for the address.
  - PWDATA_const_net_0: Internal signal storing a constant value for the write data.

- **Key Components:**

  1. The module connects the CLK1_PAD input directly to the "Clock_Gen_Clock_Gen_0_FCCC" component.
  2. The clock generator component takes input signals and generates three clock outputs (GL0, GL1, and GL2) and a LOCK signal that indicates whether the PLL is locked.
  3. Internal signals are used to store the clock generator outputs and update them sequentially to avoid glitches.

- **Interactions with Other Modules:** Acts as intermediary between the raw clock signal generation from the lower module, providing conditioning and calibration before routing them to the clock control module to be directed to the rest of the system.

- **Test bench and simulation:**

Clock_Gen_Clock_Gen_0_FCCC

- **Module Name:** Clock_Gen_Clock_Gen_0_FCCC.vhd
- **Description:** The innermost component of the clock generation mechanism in the system. It is responsible for generating the raw clock signals that will ultimately be directed towards the other modules. These raw clock signals will be further processed and conditioned by the subsequent "clock_gen" module before being sent to the clock control module for routing and control.
- **Inputs:**
  - PLL_ARST_N: Active-low reset signal for the PLL (Phase-Locked Loop).
  - PLL_POWERDOWN_N: Active-low signal to enable the PLL.
  - CLK1_PAD: The input clock signal to the clock generator.
- **Outputs:**
  - LOCK: Output signal indicating the lock status of the PLL.
  - GL0: Clock output 0.
  - GL1: Clock output 1.
  - GL2: Clock output 2.
- **Internal Signals:**
  1. CLKINT: An internal component that generates a clock output based on its input.
  2. INBUF: An internal component that buffers its input signal.
  3. VCC: An internal component responsible for voltage supply.
  4. GND: An internal component providing a constant ground signal.
  5. CCC: An internal component responsible for generating clock signals with specific configurations.
- **Key Components:**
  1. The module uses internal components to handle buffering, voltage supply, and clock generation functionalities.

2. The CCC component is instantiated, which is responsible for generating clock signals with specific configurations based on its inputs.
3. The clock generator provides the raw clock signals (GL0, GL1, GL2) that will be further processed and conditioned by the "clock_gen" module.

- **Interactions with Other Modules:** This module generates the raw clock signals that are sent to the clock_gen module for conditioning.

- **Test bench and simulation:**

Controlio

- **Module Name:** *controlIO.vhd*

- **Description:** Control Board IO and LED Interface - The LED information is clocked out to 74HCT595 devices and is locked on the rising edge of the clock. ControlIO is a control module responsible for managing IO operations and LED status for two axes (Axis0 and Axis1).

  The primary purpose of the ControlIO module is to control the communication and configuration of LEDs and IO operations for Axis0 and Axis1. It includes a state machine that orchestrates the shifting of data in and out of shift registers, controls the timing and sequencing of operations, and manages the LED status and IO control signals.

- **Inputs:**
  - RESET: Asynchronous reset signal.
  - H1_CLKWR: Clock signal for the module.
  - SysClk: System clock signal.
  - Enable: Enable signal for the module.
  - SynchedTick: Synchronous tick signal.
  - intDATA: Data input for IO operations.
  - Control signals for Axis0 and Axis1: These signals include signals for LED status read/write, IO read/write, and fault signals.

- **Outputs:**
  - controlIoDataOut: Data output for control IO operations.
  - M_IO_OE: Output enable signal for the shift register controlling LEDs on axis modules.
  - M_IO_LOAD: Control signal to select input latch or shift register for IO operations.
  - M_IO_LATCH: Control signal to transfer data from shift register to outputs and latch inputs.
  - M_IO_CLK: Clock signal for input and output data through shift registers.
  - M_IO_DATAOut: Data output for IO operations.
  - M_ENABLE: Enable signals for Axis0 and Axis1.
  - QA0AxisFault: Output signals for Axis0 fault status.
  - QA1AxisFault: Output signals for Axis1 fault status.

- **Internal Signals:**

    - ShiftOutRegister and ShiftInRegister are internal signals used to store the data being shifted in and out of the shift registers.
    - DataBufferOut and DataBufferIn are internal signals used to buffer the data coming in and out from the processor and the shift registers.
    - Count is a 4-bit signal used as a synchronous counter with count enable and asynchronous reset.
    - State is a 3-bit signal representing the current state of the state machine controlling the LED write sequence.
    - PowerUpLatch and StartStateMachine are signals used to control the state machine and manage the power-up sequence.
    - The StateMachine process controls the state transitions and operations of the LED write sequence based on the current state and input signals.
    - The M_LED_CLK process generates the output clock for the shift registers and increments the count for the state machine.
    - The module also includes various assignments and calculations for control signals, LED status, and fault signals based on the input and internal signals.
    - Overall, the ControlIO module provides the necessary functionality to manage the IO operations and LED status for Axis0 and Axis1. It uses a state machine to control the shifting of data in and out of shift registers, and it handles the timing and sequencing of operations to ensure proper communication and configuration of LEDs and IO signals.

- **Interactions with Other Modules:**

- **Test bench and simulation:**

Controloutput

- **Module Name:** *controloutput.vhd*

- **Description:** This module is responsible for controlling the output data to the digital-to-analog converter (DAC). The module takes input data (intDATA) from the processor and converts it to the MAX5541 format to drive the DAC. It uses a state machine to control the timing of data shifting and clocking into the DAC.

- **Inputs:**

    - H1_CLKWR: The write clock signal from the processor, used to latch intDATA into the DataBuffer.
    - SysClk: The system clock signal used for synchronous operations.
    - RESET: A synchronous reset signal to initialize the module.

- SynchedTick: A synchronized tick signal used to synchronize operations with other modules in the system.
- intDATA: The input data (16-bit) from the processor to be sent to the DAC.
- ControlOutputWrite: A control signal that indicates when new data is available for the module to process.
- PowerUp: A signal indicating that the system is powering up.

- **Outputs:**
  - M_OUT_CLK: The output clock signal used for clocking data into the DAC.
  - M_OUT_DATA: The output data signal sent to the DAC.
  - M_OUT_CONTROL: The control signal used to select the DAC (chip select).
- **Internal Signals:**
  - ShiftRegister: A 16-bit shift register used to shift data to the DAC.
  - DataBuffer, DataBufferOut: A 16-bit word buffer that stores the input data (intDATA) coming from the processor. DataBufferOut is used to convert the data from two's complement to MAX5541 format.
  - Count: A 5-bit synchronous counter used for timing control during the shift process.
  - ControlOutputWriteLatched0, ControlOutputWriteLatched1, ControlOutputWriteLatched2: Signals used to detect the falling edge of ControlOutputWrite to start the shift process.
  - ControlOutputOneShot: A one-shot signal that becomes active for a single SysClk cycle after a data write or during power-up.
  - OutputClock: An output clock signal used for clocking data into the DAC.
  - ShiftEnable: A control signal to enable data shifting.
  - ShiftComplete: A signal indicating the completion of the shift process.
  - ShiftDataOutput: The data output of the shift register to be sent to the DAC.
- **Processes and State Machines:**
  1. H1_CLKWR Process: This process handles the latching of the input data (intDATA) into the DataBuffer when ControlOutputWrite is asserted.
  2. DataBufferOut Process: This process converts the data from two's complement to MAX5541 format by inverting the 14-bit data and appending the sign bit.
  3. StateMachine Process: This process controls the state transitions based on the system clock (SysClk), SynchedTick, ControlOutputOneShot, and ShiftDataOutput. It manages the states s0 and s1. In s0 state, the module waits for the falling edge of ControlOutputWrite to transition to the s1 state and start the shift process.

4. Shift Register and Counter Process: This process handles the data shifting operation. It uses the ShiftEnable and OutputClock signals to load data into the ShiftRegister, and it performs the shift operation during the OutputClock's rising edge.

- **Key Components:**
- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

  To simulate the ControlOutput module, a testbench has been created with appropriate stimuli for the inputs (H1_CLKWR, SysClk, RESET, SynchedTick, intDATA, ControlOutputWrite, PowerUp, Enable). The simulation should observe the behavior of the M_OUT_CLK, M_OUT_DATA, and M_OUT_CONTROL signals and verify that the data is correctly shifted into the DAC according to the state machine's control. The testbench should test various scenarios, including the data write process, shift operation, and module functionality during power-up, to ensure the correct functionality of the module.

CPUConfig

- **Module Name*:*** *CPUconfig.vhd*
- **Description:** The CPUConfig module is responsible for handling various configurations related to the CPU and control loop parameters. It takes multiple input signals and processes them to control several output signals, including the CPU drive enable (M_DRV_EN_L) line, the control loop time selection (LoopTime), and the reset signal for internal components (DLL_RST). The module receives a 32-bit data input (intDATA) and a control signal (CPUConfigWrite) to update its internal configuration registers. It also receives reset signals (RESET and SysRESET) and a clock signal (H1_CLKWR and H1_PRIMARY) for synchronization. The module also interfaces with the HALT drive signal (HALT_DRIVE_L) and the Ethernet build signal (ENET_Build).
- **Inputs:**
  - **int_M_DRV_EN**: A signal used to handle the drive enable line (**M_DRV_EN_L**). The actual drive enable line is active low, but the processor writes to this signal as if it's active high.
  - **int_DLL_RST**: A signal used to handle the DLL reset (**DLL_RST**) condition.
  - **intLoopTime**: A 3-bit signal used for control loop time selection.
  - **dll_rst_pre_queue**: A signal used for queueing the DLL reset signal to avoid glitches.
  - **dll_rst_queue**: A 2-bit signal used for queueing the DLL reset to avoid glitches.

- **Outputs:**
    - **cpuConfigDataOut**: The output signal concatenates several values, including the control loop time selection, DLL lock status, and drive enable status, to communicate with external components.
    - **M_DRV_EN_L**: The drive enable output signal, active low, controls the drives.
    - **DLL_RST**: The output signal is used to clear the SysRESET condition.
- **Internal Signals:**
- **Processes:**
    1. The first process block handles the CPU drive enable (**M_DRV_EN_L**) line and the control loop time selection (**intLoopTime**). It sets **int_M_DRV_EN** based on **HALT_DRIVE_L** and updates the **intLoopTime** value when **CPUConfigWrite** is active and there is a rising edge on **H1_CLKWR**. The **intLoopTime** is updated with the three least significant bits of **intDATA**, and **M_DRV_EN_L** is driven with the inverted value of **int_M_DRV_EN**.
    2. The second process block handles the DLL reset (**DLL_RST**). It clears the **int_DLL_RST** signal when **SysRESET** is low and updates it based on **intDATA(2)** when **CPUConfigWrite** is active and there is a rising edge on **H1_PRIMARY**. To prevent glitches, the **dll_rst_pre_queue** is used to queue the DLL reset signal when no writes are occurring (**CPUConfigWrite = '0'**), and **dll_rst_queue** holds the previous and current value to generate the **DLL_RST** signal.
- **Key Components:**
- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

CPULED

- **Module Name**: *CPULED.vhd*
- **Description:** The CPULED module is responsible for driving CPU status LEDs on the RMC75E Rev 3.0 board. It takes input signals, processes them, and generates output signals to control the LEDs. The module receives a 32-bit data input (intDATA) and a control signal (CPULEDWrite) to update the status of the CPU status LEDs. It also receives a reset signal (RESET) and a clock signal (H1_CLKWR) for synchronization. The module includes a 2-bit signal (CPUStatusLED) that represents the state of the two CPU status LEDs.
- **Outputs:**

- **cpuLedDataOut**: The output signal concatenates **CPUStatusLED** with other fixed values to control the CPU status LEDs.

- **Internal Signals:**
  - **CPUStatusLED**: A 2-bit signal representing the state of the CPU LEDs.

- **Processes:**

  Process Block: The process block inside the architecture updates the **CPUStatusLED** signal based on the input conditions. When the **RESET** signal is active, the **CPUStatusLED** is set to "00" to clear the status of the LEDs. When the **CPULEDWrite** signal is active and there is a rising edge on the **H1_CLKWR** signal, the **CPUStatusLED** is updated with the two least significant bits of the **intDATA** input. This action effectively sets the CPU status LEDs based on the incoming data.

  LED Control Logic: The **CPUStatLEDDrive** signal controls the behavior of the CPU status LEDs. When the **CPUStatusLED** signal is "00" (undefined state), the output is set to high impedance (**Z**). This behavior is intentional to allow external circuitry to force the LEDs to turn red when the FPGA is not driving them. For all other states of **CPUStatusLED**, the **CPUStatLEDDrive** signal inverts the **CPUStatusLED**, determining the state of the LEDs.

- **Key Components:**

  The **CPULED** module is a simple implementation for driving CPU status LEDs on the RMC75E Rev 3.0 board. It takes input signals, processes them, and generates output signals to control the LEDs. The module receives a 32-bit data input (**intDATA**) and a control signal (**CPULEDWrite**) to update the status of the CPU status LEDs. It also receives a reset signal (**RESET**) and a clock signal (**H1_CLKWR**) for synchronization. The module includes a 2-bit signal (**CPUStatusLED**) that represents the state of the two CPU status LEDs.

  The **CPULED** module is implemented in the **CPULED_arch** architecture. It consists of signal declarations and a process block that updates the **CPUStatusLED** signal based on the input conditions.

- **Important Calculations:**

- **Interactions with Other Modules:**

- **Test bench and simulation:**

Databuffer

- **Module Name***: DataBuffer.vhd*

- **Description**: This module serves as a buffer for handling data from different input sources and facilitating data distribution to various output channels within the system.

- **Inputs:**

  - H1_CLKWR: Input clock signal.
  - SysClk: System clock signal.
  - SynchedTick: Synchronized tick signal.
  - SynchedTick60: Synchronized tick signal with a 60 Hz period.
  - AnlgPositionRead0, AnlgPositionRead1: Analog position data inputs.
  - ExpA0ReadCh0, ExpA0ReadCh1, ExpA1ReadCh0, ExpA1ReadCh1, ExpA2ReadCh0, ExpA2ReadCh1, ExpA3ReadCh0, ExpA3ReadCh1: Expansion module data inputs.
  - WriteConversion: Control signal for write operation.
  - S2P_Addr: Output, 4-bit address for selecting the output channel.
  - S2P_Data: Input, 16-bit parallel data to be written into the buffer.

- **Outputs:**
  - DataOut: Output, 16-bit parallel data read from the buffer.
- **Internal Signals:**
  - BankSelect: Signal to toggle between two data buffers.
  - WriteEnable: Control signal for write operation.
  - WriteBank0, WriteBank1: Control signals for writing to specific banks.
  - DetectRead: Signal to detect read operation.
  - DecrementReadPointer: Control signal for decrementing read pointer.
  - ReadPointer, WritePointer: 3-bit read and write pointers for buffer access.
  - ReadEnableEncode: 10-bit signal encoding read enable for different channels.
  - ModuleSelect: 4-bit signal for selecting the module.

- **Processes and State Machines:**

- **Key Components:**

  1. Data received at S2P_Data input is written to two separate RAM buffers based on the state of BankSelect.
  2. The module handles switching between the buffers on every rising edge of the SynchedTick signal.

3. Data is read from the buffer based on the address specified in S2P_Addr.
4. The read address is determined by ModuleSelect and ReadPointer, or WritePointer and S2P_Addr, based on BankSelect state.
5. The output DataOut provides the selected 16-bit parallel data based on S2P_Addr.

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

Decode

- **Module Name:** *Decode.vhd*
- **Description:** The module "Decode" is responsible for generating WRITE control lines and READ control lines based on the input address and control signals. It is a decoder module for interfacing with various peripheral devices.
- **Key Features:**

1. This module takes various inputs, including the address (ADDR), read (RD_L), write (WR_L), and chip select (CS_L) control signals, as well as several other control signals for different peripheral devices.
2. The module supports multiple peripheral devices, such as FPGA, analog interface, quadrature decoder, etc. Each peripheral has its read and write control signals.
3. The module provides various output control signals, one for each peripheral, to enable/disable read or write operations to the respective peripherals.
4. Some of the peripherals supported include FPGA (with read and write enable), CPU configuration (read and write), CPU LED (read and write), Watch Dog Timer (read and write), PROFIBUS address, Serial Memory Interface (read and write), etc.
5. The module seems to handle different peripheral addressing, allowing communication with multiple instances of a peripheral type (e.g., multiple axes).
6. The decoding process is carried out through conditional assignments of signals based on the input address and control signals, allowing the appropriate peripheral control signals to be activated when the correct conditions are met.

- **Key Functionalities:**

- Handling control signals for the FPGA and CPU board.
- Decoding CPU board configuration reads and writes.
- Decoding CPU board LED reads and writes.
- Handling control signals for Axis 0 and Axis 1 modules.
- Decoding signals related to the MDT and Analog modules.
- Decoding signals related to the Quadrature (QUAD) module.
- Decoding expansion card control signals, including analog and digital I/O.

- **Important Calculations:**

- **Interactions with Other Modules:**

- **Test bench and simulation:**

DIO8

- **Module Name***: DIO8.vhd*

- **Description:** The DIO8 module is an 8-bit digital input/output (DIO) interface designed to work with multiple expansion slots. It provides access to four DIO8 modules, each containing eight bidirectional digital I/O channels. The module operates on a clocked architecture and includes a state machine that controls the read and write sequences to the DIO8 modules. It provides a versatile and configurable interface for handling digital input and output operations. It is well-suited for applications that require multiple DIO8 modules to be accessed through different expansion slots. The state machine-based design ensures efficient handling of read and write operations, making it suitable for real-time applications with strict timing requirements.

- **Architecture:**

  The architecture **DIO8_arch** implements the behavior of the **DIO8** entity. Let's go through the code to understand its functionality:

  1. State Encoding: The code defines a custom type called **STATE_TYPE** which represents different states of the state machine used in the code.
  2. Signals and Registers: Various signals and registers are defined to hold and manipulate the data during the operation of the module. These include **State** (representing the current state of the state machine), **ExpSlot** (selects the expansion slot location), **InputShiftRegister** (holds the input data for shifting), **OutputShiftRegister** (holds the output data for shifting), and several others.
  3. Multiplexer: There is a multiplexer section that maps the appropriate signals to the respective expansion slot based on the value of **ExpSlot**. It controls the data flow between the different DIO8 modules in the expansion slot stack-up.
  4. Data Output: The **d8DataOut** signal represents the data output lines. Depending on the values of **ExpDIO8DinRead** and **ExpDIO8ConfigRead**, it combines the input data registers and output data registers to generate the output data.

5. Register Write: The process labeled **RESET, H1_CLKWR** handles the write operation to the output registers (**D8OutputReg0**, **D8OutputReg1**, **D8OutputReg2**, **D8OutputReg3**) when the **H1_CLKWR** signal rises. The values to be written are taken from **intDATA** based on the values of **ExpDIO8ConfigWrite**.
6. Register Mapping: The process labeled **ExpSlot, D8OutputReg0, D8OutputReg1, D8OutputReg2, D8OutputReg3** maps the appropriate output register based on the **ExpSlot** value to the **IntDout** signal. It ensures that the correct register is selected for reading.
7. Register Read: The process labeled **rising_edge(SysClk)** handles the read operation from the input registers (**D8InputReg0**, **D8InputReg1**, **D8InputReg2**, **D8InputReg3**) when **IntWrite** is active and the corresponding **ExpSlot** value matches. It loads the data from the input shift register (**InputShiftRegister**) into the appropriate input register.
8. Data Output Assignment: The **d8DataOut** signal is assigned the value of the **OutputShiftRegister(15)** for the purpose of data output.
9. State Machine: The process labeled **RESET, SysClk** implements the state machine functionality. It defines the behavior of the state transitions based on the current state (**State**) and various control signals.
    - The state machine starts in the **IdleState** and waits for a rising edge of **SysClk**.
    - The state machine progresses through different states such as **LoadState1**, **LoadState2**, **LoadState3**, **s3_LoadShiftState**, **s4_ShiftIOState**, **s5_SRAMWriteState**, **s6_SRAMReadState**, **s7_LoadShiftState**, **s8_ShiftOutState**, and **s9_EndState**.
    - The state transitions are controlled by conditions based on control signals such as **SynchedTick**, **SlowEnable**, **ExpDIO8ConfigWrite**, and **ExpSlot**.
    - The state machine manages the loading, shifting, writing, and reading of data registers and controls the flow of data within the module.

- **Key Components:**

1. Input and Output Interfaces: The module contains separate input and output interfaces. The input interface receives data from the external devices through the Exp0D8_DataIn, Exp1D8_DataIn, Exp2D8_DataIn, and Exp3D8_DataIn lines. The output interface sends data to external devices through the d8DataOut line.
2. Configuration Control: The module supports configuration control through the ExpDIO8ConfigRead and ExpDIO8ConfigWrite input signals. These signals allow the CPU to read and write configuration data to the dual-port memory in the DIO8 module.
3. Data Storage: The module features dual-port memory for storing data from external devices and the CPU. The registers, D8OutputReg0, D8OutputReg1, D8OutputReg2, and D8OutputReg3, store output data, while D8InputReg0, D8InputReg1, D8InputReg2, and D8InputReg3 hold input data.
4. State Machine: The core of the module is a state machine that sequences through the read and write operations. The state machine controls various aspects of the module, such as

loading data into registers, shifting data in and out, and managing the read and write states. The state machine transitions through different states such as IdleState, LoadState1, LoadState2, LoadState3, s3_LoadShiftState, s4_ShiftIOState, s5_SRAMWriteState, s6_SRAMReadState, s7_LoadShiftState, s8_ShiftOutState, and s9_EndState.

5. Clocking: The module uses multiple clocks, including the system clock (SysClk), a synchronized tick (SynchedTick), and an internal clock (intExpD8_Clk).
6. Expansion Slot Selection: The module supports four expansion slots (Exp0, Exp1, Exp2, Exp3). The ExpSlot signal is used to select the desired expansion slot for read and write operations.
7. Slow Enable: The SlowEnable signal enables a 3.75MHz clock output for serial communication to control the read and write sequences to the DIO8 modules.
8. Shift Register: The module utilizes shift registers for data loading and shifting operations.

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

Discovercontrol

- **Module Name***: DiscoverID.vhd*
- **Description:** The "DiscoverID" module is a key component of the RMC75E Rev 3.0 board design. It serves the purpose of identifying and configuring different types of modules connected to the FPGA. The module reads the Control Module ID word during system power-up, which is then used by the CPU and FPGA to determine the transducer interface logic configuration. The design employs state machines and conditional logic to handle the identification and configuration process. The module interacts with other modules like "DiscoverControlID" and "DiscoverExpansionID" to achieve its intended functionality.

- The "DiscoverID" module's primary role is to identify the types of modules connected to the FPGA and configure the interface logic, contributing to the proper functioning of the RMC75E Rev 3.0 board.

It interacts with other modules such as "DiscoverControlID" and "DiscoverExpansionID" through component declarations. The main operations of the "DiscoverID" module are described below in more detail:

1. Input and Output Ports:
    - The module has various input and output ports, including RESET, SysClk, SlowEnable, FPGAIDRead, ControlCardIDRead, etc. These ports enable communication with other parts of the system.

2. Constants and Identifiers:
    - The module uses constants to represent different FPGA IDs, versions, and module types, such as AP2, A2, DIO8, Q1, MDT1, MDT2, QUAD1, QUAD2, ANLG1, ANLG2, and ENET.
    - FPGA_ID, FPGA_Major_Rev, and FPGA_Minor_Rev are used to store the identification information of the FPGA being used.
3. State Machines and Module Identification:
    - The module employs state machines and conditional checks to identify the connected modules based on the Control Module ID word.
    - By examining specific bits of the Control Module ID, the module determines the type of module connected (e.g., AP2, A2, DIO8, Q1, MDT, QUAD, ANLG, or ENET).
4. Control Signal Configuration:
    - Depending on the module type detected, the module sets appropriate control signals like intExp0Mux, intExp1Mux, intExp2Mux, and intExp3Mux.
    - The module also sets ExpOldAP2 and Exp0Mux, Exp1Mux, Exp2Mux, and Exp3Mux signals to configure the transducer interface logic for connected modules.
5. Discovery Completion and LED Control:
    - The "DiscoveryComplete" signal is used to control the multiplexer for the Expansion Module LED signals. After the discovery process is complete, the connected modules' LED control is activated.
6. Retention and Reset Signals:
    - The module uses various reset signals to control the state machines and ensure proper initialization and behavior during power-up and system reset.
    - Configuration data related to the connected modules is retained and used until the next system power cycle.

- **Interactions with Other Modules:**

- **Test bench and simulation:**

DiscoverControlID

- **Module Name:** *DiscoverControlID.vhd*

- **Description:** This module is responsible for controlling the discovery of the control ID for the motion controller. The module handles the process of capturing the control ID using a state machine and shifting the ID into the ControlID signal.

- **Inputs:**

    - RESET: A synchronous reset signal to initialize the state machine and shift register.
    - SysClk: The system clock signal used for synchronous operations.
    - SlowEnable: A control signal that enables the generation of a 5MHz clock output for the serial communication state machine.
    - M_Card_ID_DATA: The data input signal containing the control ID data from an external module.

- **Outputs:**
    - M_Card_ID_DATA: The data input signal containing the control ID data from an external module.
    - ControlID: A 17-bit bidirectional signal that holds the captured control ID data.
    - M_Card_ID_CLK: The output clock signal for the state machine.
    - M_Card_ID_LATCH: The output signal to control the latch signal for capturing the control ID data.
    - M_Card_ID_LOAD: The output signal to control the load signal for capturing the control ID data.

- **Processes and State Machines:**
    1. StateMachine process: This process controls the state transitions based on the system clock (SysClk) and the SlowEnable signal. It drives the control signals M_Card_ID_LOAD, M_Card_ID_LATCH, and ShiftEnable. The state machine goes through different states to perform the capture operation. It starts with s0_LatchState, proceeds to s1_DelayState1, s2_LoadState, s3_DelayState2, s4_ClockState, and finally reaches s5_StopState.
    2. Process for Shift Register and Counters: This process handles the data capture and shifting process. It captures the M_Card_ID_DATA and performs a shift operation on the ControlID signal based on the OutputClock signal generated by the state machine. It also handles the Count signal, which serves as a 4-bit synchronous counter with count enable and asynchronous reset. The process uses TerminalCountValue (X"F") to determine the terminal count value, and when the count reaches this terminal count and SlowEnable is '1', the ShiftComplete signal is set to '1'. ShiftComplete is used to indicate the completion of the shift operation.

- **Key Components:**
- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

DiscoverExpansionID

- **Module Name***: DiscoverExpansionID.vhd*
- **Description**: This module is responsible for discovering the ID information from the expansion modules after a reset.

- **Inputs:**
    - RESET: A synchronous reset signal to initialize the state machine and shift register.
    - SysClk: The system clock signal used for synchronous operations.
    - SlowEnable: A control signal that enables the generation of a 3.75MHz clock output for the serial communication state machine.
    - Exp_ID_DATA: The data input signal containing the ID information from an external module.

- **Outputs:**
    - ExpansionID0: A 17-bit bidirectional signal that holds the ID information from Expansion Module 0.
    - ExpansionID1: A 17-bit bidirectional signal that holds the ID information from Expansion Module 1.
    - ExpansionID2: A 17-bit bidirectional signal that holds the ID information from Expansion Module 2.
    - ExpansionID3: A 17-bit bidirectional signal that holds the ID information from Expansion Module 3.
    - Exp_ID_CLK: The output clock signal for the state machine.
    - Exp_ID_LATCH: The output signal to control the latch signal for capturing the ID data.
    - Exp_ID_LOAD: The output signal to control the load signal for capturing the ID data.

- **Key Components:**
    1. Clock Generation:
        - The module receives the system clock **SysClk**, which is used to synchronize internal operations.
        - The **SlowEnable** signal is used to enable a 3.75MHz clock output for the serial communication state machine.
    2. State Encoding:
        - The module uses a 3-bit state encoding (**STATE_TYPE**) to represent different states in the state machine.
        - Constants like **s0_LatchState**, **s1_DelayState1**, etc., define the different states and their values.
    3. State Machine:

- The module employs a synchronous state machine (**StateMachine**) to manage the process of querying ID information from expansion modules.
- After reset (**RESET** signal asserted), the state machine initializes in **s0_LatchState**.

4. Control Signals Initialization:
    - On reset, certain control signals (**Exp_ID_LOAD**, **Exp_ID_LATCH**, and **ShiftEnable**) are initialized to specific values.
    - **Exp_ID_LOAD** is set to '1', **Exp_ID_LATCH** is set to '0', and **ShiftEnable** is set to '0'.
5. State Transitions:
    - The state machine transitions through different states based on the current state and other conditions.
    - The **SlowEnable** signal is used to enable state transitions and control the operation of the state machine.
6. Clocking Mechanism:
    - The module generates an **Exp_ID_CLK** signal, which is derived from the system clock (**SysClk**) and is used for clocking data during the ID querying process.
7. 6-bit Synchronous Counter:
    - The module uses a 6-bit synchronous counter (**Count**) to keep track of clock cycles during the ID querying process.
    - The counter is reset to "000000" when in the **s0_LatchState** or when **ShiftEnable** is '0'.
    - The counter increments on each positive edge of **SysClk** when **ShiftEnable** and **OutputClock** are '1'.
8. Shift Register:
    - The module employs a 64-bit shift register to shift in data received during the ID querying process.
    - The shift register is updated when **ShiftEnable** is '1', **OutputClock** is '0', and **SlowEnable** is '1'.
    - The data from **Exp_ID_DATA** is serially loaded into the shift register, shifting the existing data to the right.
9. Shift Complete Detection:
    - The module uses a **TerminalCount** value (64 + 1) and a counter (**Count**) to detect the end of the shift operation (**ShiftComplete**).
    - When the counter reaches the **TerminalCount**, **ShiftComplete** is set to '1', indicating the end of the shift operation.
10. Data Valid Flag:

- The most significant bit (bit 16) of each expansion ID (**ExpansionID0**, **ExpansionID1**, **ExpansionID2**, **ExpansionID3**) acts as a data valid flag.
- It is set to '1' when the state machine is in **s5_StopState**, indicating that the ID data is valid and can be read.

Overall, the **DiscoverExpansionID** module employs a state machine to control the process of querying ID information from expansion modules. It utilizes a shift register and a counter to synchronize and manage data reception during the querying process. The module also provides a clock output (**Exp_ID_CLK**) and sets a data valid flag to indicate when the ID data is ready to be accessed.

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

ExpModuleLED

- **Module Name:** *ExpModuleLED.vhd*
- **Description:** The ExpModuleLED provides the interface for writing LED values to expansion boards in the RMC75E.
- **Inputs:**
  1. **Reset**: Asynchronous reset signal to initialize the module.
  2. **H1_CLKWR**, **SysClk**, **SynchedTick**: Clock signals used in the design.
  3. **SlowEnable**: A control signal to enable slow clock operation.
  4. **intDATA**: Input data to the module.
  5. Various other input control signals for different LED writes and reads, as well as EEPROM access and discovery completion flags.
- **Outputs:**
  1. **expLedDataOut**: Output for LED data to be displayed on the expansion boards.
  2. Various control signals for LED data (e.g., **ExpLEDOE**, **ExpLEDLatch**, **ExpLEDClk**, **ExpLEDData**, **ExpLEDSelect**, etc.).
  3. **ExpOldAP2**: Input signal representing the state of the old AP2 module.
- **Internal Signals:**

Several internal signals (**ShiftRegister0**, **ShiftRegister1**, **ShiftRegister2**, **ShiftRegister3**, **Count**, **OutputClock**, **ShiftEnable**, **StartStateMachine**, **Exp0LED**, **Exp1LED**, **Exp2LED**, **Exp3LED**, **ClearExpLEDLatch**, **intExpLEDSelect**, **intExp0LED**, **intExp1LED**, **intExp2LED**, **intExp3LED**, **intExpLEDOE**, **EnableDelay**, **State**) are used for controlling LED writes and shifts.

- **Key Components:**
  - The interface is designed to work with an 8-bit shift register, where each bit corresponds to an LED.
  - The **ShiftRegister0**, **ShiftRegister1**, **ShiftRegister2**, and **ShiftRegister3** signals are used to hold the LED data that will be shifted out to the LED drivers.
  - The state machine controls the write sequence to the LED driver and is driven by the **SysClk** and **SynchedTick** signals.
  - The **StartStateMachine** signal is used to initiate the state machine's operation when LED writes are requested and other conditions are met.
  - The state machine goes through four states: **IdleState**, **ShiftState**, **ClearState**, and **EndState**. It controls the LED data shifting and latching process.
  - The **ExpLEDSelect** signal selects which LED data should be written to the shift registers based on the LED write requests (**Exp0LEDWrite**, **Exp1LEDWrite**, etc.).
  - The LED data is latched when the state machine enters the **EndState** state, and the **ExpLEDOE** signal is used to control the enable pin for the LED drivers.
  - The state machine is reset when the **Reset** signal is asserted.
- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

ExpansionSigRoute

- **Module Name:** *expsigroute.vhd*
- **Description:** The ExpSigRoute module is a signal routing module that takes multiple inputs and routes them to various output signals based on the values of control signals like ExpMux, ExpSerialSelect, and ExpLEDSelect. It acts as a multiplexer and logically connects the inputs to different parts of the RMC75E modular motion controller based on the configuration settings. The module is responsible for routing various signals within the expansion modules. It acts as a

central signal routing unit that connects different logic modules in the system based on the configuration set by "ExpMux" and other control signals.

- **Inputs:**
  - ExpMux: A 2-bit signal that determines which module is currently selected for signal routing.
  - ExpSerialSelect: A control signal that selects whether the serial interface is active or not.
  - ExpLEDSelect: A control signal that selects whether the LED module is active or not.
  - ExpLEDData: The data input signal containing LED data.
  - ExpData: A 6-bit inout signal used for data routing between different modules.
  - ExpA_CS_L: The chip select signal for the analog input logic module.
  - ExpA_CLK: The clock signal for the analog input logic module.
  - ExpD8_Clk: The clock signal for the DIO input logic module.
  - ExpD8_DataOut: The data output signal from the DIO input logic module.
  - ExpD8_OE: The output enable signal for the DIO input logic module.
  - ExpD8_Load: The load signal for the DIO input logic module.
- **Outputs:**
  - ExpQ1_A, ExpQ1_B, ExpQ1_Reg, ExpQ1_FaultA, ExpQ1_FaultB: These are output signals that connect to the Q1 logic module based on the ExpMux value.
  - SerialMemoryDataIn: The data input signal for the serial memory module.
  - ExpA_DATA: The data output signal for the analog input logic module.
  - ExpData(0): The output signal for the clk connection to different modules.
  - ExpData(1): The output signal for the data connection to different modules.
  - ExpData(2): The output signal for the latch connection to different modules.
  - ExpData(3): The output signal for the output enable connection to different modules.
  - ExpData(4): The output signal for the chip select connection to different modules.
  - ExpData(5): The output signal for the load connection to different modules.
- **Architecture:** The architecture of the ExpSigRoute module, named ExpSigRoute_arch, consists of a process block that performs the signal routing based on the configuration signals (ExpMux, ExpSerialSelect, and ExpLEDSelect).

The process block logically routes the signals based on the current values of ExpMux, ExpSerialSelect, and ExpLEDSelect, directing them to different parts of the RMC75E modular motion controller. The connections are made as follows:

- For the Q1 logic module, the ExpData(0:4) signals are routed based on the value of ExpMux.
- For the Analog input logic module (Ax), ExpA_DATA(0) and ExpA_DATA(1) are selected based on ExpMux, while SerialMemoryDataIn is also selected if ExpSerialSelect is '1'.
- For the DIO input logic module (D8), ExpD8_DataIn is selected based on ExpMux, while ExpData(0:5) are used for clk, data, latch, output enable, load, and data connections respectively.
- For the LED module, ExpData(1) and ExpData(5) are used for data and data connection respectively if ExpLEDSelect is '1'.

Input Signal Routing:
The module receives signals from various expansion modules, including analog inputs (from an external memory), DIO (Digital Input/Output) signals, and other control signals.

Signal Multiplexing (ExpMux):
The "ExpMux" input determines which logic module's signals need to be routed to the output "ExpData."
The module can direct signals from three different logic modules: Q1 (unknown), Ax (Analog Input), and D8 (DIO input).

Output Signal Routing:
The module routes the appropriate signals from the selected logic module to the "ExpData" output.

The "ExpData" output is a 6-bit signal that carries various information, including clock signals, data signals, latch signals, enable signals, etc., depending on the selected logic module.

- **Important Calculations:**

- **Interactions with Other Modules:**

Handling Invalid Modules:
If there is no valid module connected or selected (indicated by "ExpMux"), the output signals are tied to specific values (e.g., '1', '0', or 'Z' for high, low, or high-impedance state, respectively), ensuring proper behavior even when no valid module is present.

Control Signals:
The module also handles control signals such as "ExpSerialSelect" and "ExpLEDSelect," which determine whether the serial memory or LED module is active for routing.
The "ExpSigRoute" module acts as a flexible and efficient signal router, enabling different expansion modules to interact and share data within the system. Its flexibility allows for easy

integration of additional modules in the future without significant changes to the overall system architecture.

- **Test bench and simulation:**

LatencyCounter

- **Module Name:** *latencycounter.vhd*

- **Description:** The LatencyCounter module is responsible for receiving clock and control signals and providing output data based on its internal latency calculations. The module accepts three inputs: a 60MHz clock signal (H1_CLK), a control signal to synchronize the operation (SynchedTick), and a signal to read the latency count (LatencyCounterRead). The output of this module is a 32-bit data output (latencyDataOut) representing the latency calculation.

- **Design Details:** The LatencyCounter module is designed as a synchronous process, triggered by the falling edge of the H1_CLK signal. It uses two internal signals, "LatencyCounter" and "latencyDataOut," both of which are 32-bit vectors.

- **Functionality:**
  1. On each falling edge of H1_CLK, the process checks the SynchedTick signal.
     - If SynchedTick is '1', indicating the start of a new latency measurement, the LatencyCounter is reset to zero.
     - If SynchedTick is '0' (low), and LatencyCounterRead is '0' (low) as well, the LatencyCounter is incremented by one.
  2. The latencyDataOut output always reflects the current value of the LatencyCounter register, representing the calculated latency in clock cycles.

MDSSIRoute

- **Module Name:** *MDTSSIRoute.vhd*

- **Description:** The MDTSSIRoute module describes the interface and behavior of a routing module for the Magnetostrictive Displacement Transducer (MDT) and Synchronous Serial Interface (SSI) clocks in the RMC75E modular motion controller. It takes input signals related to the SSI select, internal MDT clock, SSI clock, and MDT interrupt, and provides output signals for the internal clocks of Axis0 and Axis1. It is designed as a synchronous process, controlled by conditional assignments. It uses the SSISelect signal to select the appropriate clock source for each axis.

  The MDTSSIRoute module provides the functionality to select the appropriate clock source for each axis based on the SSISelect signal. It enables the routing of clock signals between the MDT internal clock and the SSI clock, allowing flexible control over the timing and synchronization of operations within the modular motion controller.

- **Inputs:**

- SSISelect: A 2-bit signal that selects the clock source for each axis. '00' selects the MDT internal clock, and '01' selects the SSI clock.
- M_AX0_SSI_CLK: Input SSI clock signal for Axis0.
- M_AX1_SSI_CLK: Input SSI clock signal for Axis1.
- M_AX0_MDT_INT: Input MDT interrupt signal for Axis0.
- M_AX1_MDT_INT: Input MDT interrupt signal for Axis1.

- **Outputs:**

- Functionality:
  On each clock cycle, the MDTSSIRoute module checks the value of the SSISelect signal and performs the following assignments:

  M_AX0_INT_CLK is assigned the value of M_AX0_MDT_INT when SSISelect(0) is '0', indicating that the MDT internal clock is selected for Axis0. Otherwise, it is assigned the value of M_AX0_SSI_CLK, indicating that the SSI clock is selected.
  M_AX1_INT_CLK follows a similar assignment logic as M_AX0_INT_CLK but for Axis1.

MDTTopSimp

- **Module Name*:*** *MDTTopSimp.vhd*

- **Description**: This module is the MDT interface in the RMC75E modular motion controller, providing an interface and control for communication with PWM, Start/Stop, and SSI type transducers. The module handles the counting sequence, detects pulses, and calculates position data based on the received signals.

- Inputs:
  - `SysReset`: Main system reset signal.
  - `H1_CLK`: 60MHz clock for the MDT interface.
  - `H1_CLKWR`: CPU clock for reads and writes.
  - `H1_CLK90`: 60MHz clock with a 90-degree phase shift for MDT counters.
  - `SynchedTick60`: Synchronized tick signal at 60MHz.
  - `intDATA`: Input data signal (32 bits).
  - `PositionRead`: Signal indicating a position read operation.
  - `StatusRead`: Signal indicating a status read operation.
  - `ParamWrite`: Signal indicating a parameter write operation.
  - `M_RET_DATA`: Input data signal for M_RET_DATA.
  - `SSISelect`: Signal indicating the selection of SSI transducer.

- Outputs:
  - `mdtSimpDataOut`: Output data signal from the MDT (32 bits).
  - `M_INT_CLK`: Output internal clock signal.

- `SSI_DATA`: Output SSI data signal.

- Internal Signals and Processes:
  - `DelayDone`: Marks the end of the 1.6us Interrogation pulse.
  - `RetPulseDelayDone`: Marks the setting of the NoTransducer flag.
  - `PWMMagnetFault`: Indicates a PWM magnet fault condition.
  - `PWMMagnetFaultLatch`: Latches the PWMMagnetFault or itself and not ClearCounter at each rising edge of H1_CLK.
  - `RisingEdgeA`: Process that shifts the value of RisingA array at every rising edge of H1_CLK.
  - `RisingAPosEdgeFound` and `RisingANegEdgeFound`: Indicate positive or negative edge in RisingA.
  - `RisingACountEnable` and `RisingACountDisable`: Enable and disable the RisingACountEnableLatch based on certain conditions.
  - `CountRA`: 18-bit synchronous counter that counts at every rising edge of H1_CLK.
  - `LeadingCountDecode` and `TrailingCountDecode`: Determine whether to add or subtract counts based on the value of Edge and other signals.
  - `LeadingCount` and `TrailingCount`: Latch the counts at the start and end of the count cycle.
  - `RisingEdgeB` and `FallingEdgeB`: Similar to RisingEdgeA and FallingEdgeA, triggered on H1_CLK90.
  - `XfrToH1_CLK`: Process that combines bits from RisingB, FallingA, and FallingB to form Edge at every rising edge of H1_CLK.
  - `MDTPosition`: 20-bit register that receives the result of CountRA and the Leading and Trailing counts.

Quad

- **Module Name:** *Quad.vhd*

- **Description:** The Quad module serves as a wrapper for multiple QuadXface components, enabling the processing of quadrature signals and generating output data for multiple quadrature interfaces and axes in the RMC75E motion controller.

- **Inputs:**

  - H1_CLKWR: Clock signal.
  - SysClk: System clock signal.
  - SynchedTick: Synchronized tick signal.
  - intDATA: 32-bit input data signal.

  - Exp0QuadCountRead, Exp1QuadCountRead, Exp2QuadCountRead, Exp3QuadCountRead: Signals to initiate the count read operation for each respective quadrature interface. Exp0QuadLEDStatusRead, Exp1QuadLEDStatusRead, Exp2QuadLEDStatusRead, Exp3QuadLEDStatusRead: Signals to read the LED status for each respective quadrature interface.

- Exp0QuadLEDStatusWrite, Exp1QuadLEDStatusWrite, Exp2QuadLEDStatusWrite, Exp3QuadLEDStatusWrite: Signals to write the LED status for each respective quadrature interface.

- Exp0QuadInputRead, Exp1QuadInputRead, Exp2QuadInputRead, Exp3QuadInputRead: Signals to read the input for each respective quadrature interface.

- Exp0QuadHomeRead, Exp1QuadHomeRead, Exp2QuadHomeRead, Exp3QuadHomeRead: Signals to read the home position for each respective quadrature interface.

- Exp0QuadLatch0Read, Exp0QuadLatch1Read, Exp1QuadLatch0Read, Exp1QuadLatch1Read, Exp2QuadLatch0Read, Exp2QuadLatch1Read, Exp3QuadLatch0Read, Exp3QuadLatch1Read: Signals to read the latch values for each respective quadrature interface.

- Exp0Quad_A, Exp0Quad_B, Exp1Quad_A, Exp1Quad_B, Exp2Quad_A, Exp2Quad_B, Exp3Quad_A, Exp3Quad_B: Signals representing the quadrature input signals for each respective quadrature interface.

- Exp0Quad_Reg, Exp1Quad_Reg, Exp2Quad_Reg, Exp3Quad_Reg:
Signals representing the quadrature home position for each respective quadrature interface.
Exp0Quad_FaultA, Exp0Quad_FaultB, Exp1Quad_FaultA, Exp1Quad_FaultB, Exp2Quad_FaultA, Exp2Quad_FaultB, Exp3Quad_FaultA, Exp3Quad_FaultB: Signals representing the fault status for each respective quadrature interface.

- QA0CountRead, QA1CountRead: Signals to initiate the count read operation for each respective axis.

- QA0LEDStatusRead, QA1LEDStatusRead: Signals to read the LED status for each respective axis.

- QA0LEDStatusWrite, QA1LEDStatusWrite: Signals to write the LED status for each respective axis.

- QA0InputRead, QA1InputRead: Signals to read the input for each respective axis.
QA0HomeRead, QA1HomeRead: Signals to read the home position for each respective axis.
QA0Latch0Read, QA0Latch1Read, QA1Latch0Read, QA1Latch1Read:
Signals to read the latch values for each respective axis.

- QA0_SigA, QA0_SigB, QA0_SigZ, QA0_Home, QA0_RegX_PosLmt, QA1_SigA, QA1_SigB, QA1_SigZ, QA1_Home, QA1_RegX_PosLmt, QA1_RegY_NegLmt:
Signals representing various control and status signals for each respective axis.

- QA0AxisFault, QA1AxisFault: 3-bit signals representing the fault status for each respective axis.

- **Outputs:**

- Exp0QuadDataOut, Exp1QuadDataOut, Exp2QuadDataOut, Exp3QuadDataOut: 32-bit output data signals for each respective quadrature interface. QuadA0DataOut, QuadA1DataOut: 32-bit output data signals for each respective axis.

- **Internal Signals:**

  Exp0_LineFault, Exp1_LineFault, Exp2_LineFault, Exp3_LineFault:
  3-bit internal signals representing the line fault status for each respective quadrature interface.
  The architecture section of the module contains multiple instances of a component called "QuadXface."

  Each instance represents a quadrature interface or axis and is responsible for processing the respective signals and generating the output data.

  The QuadXface component has the following ports:

  H1_CLKWR: Clock signal.
  SysClk: System clock signal.
  SynchedTick: Synchronized tick signal.
  intDATA: 32-bit input data signal.
  QuadDataOut: 32-bit output data signal.
  CountRead: Signal to initiate the count read operation.
  LEDStatusRead: Signal to read the LED status.
  LEDStatusWrite: Signal to write the LED status.
  InputRead: Signal to read the input.
  HomeRead: Signal to read the home position.
  Latch0Read: Signal to read the first latch value.
  Latch1Read: Signal to read the second latch value.
  Home: Signal representing the home position.
  RegistrationX: Signal representing the X-axis registration.
  RegistrationY: Signal representing the Y-axis registration.
  LineFault: 3-bit signal representing the line fault status.
  A, B: Signals representing the quadrature input signals.
  Index: Signal representing the quadrature index signal.

  The QuadXface components are instantiated for each quadrature interface and axis, and their ports are connected to the corresponding signals from the Quad entity.The instantiation maps the signals to the appropriate inputs and outputs of the QuadXface components.

- **Interactions with Other Modules:**

- **Test bench and simulation:**

QuadXface

- **Module Name:** *QuadXface.vhd*

- **Description:** The QuadXface module is a quadrature counter module designed to work with incremental encoders or similar devices. It provides the ability to count pulses generated by A and B inputs and detect the direction of rotation.The module supports four different count registers: QuadCount, HomeReg, Latch0Reg, and Latch1Reg.

  The QuadXface module also includes several features related to homing and capturing counts:
  - Homing: It supports different homing triggers like rising edge, falling edge, index edge, etc.
  - Capture: It can capture the QuadCount value on specific events (e.g., rising/falling edges of A or B, index edge).

  The module is configurable and supports multiple settings:
  - Homing Trigger Type: Defines the type of trigger for homing (rising, falling, index edge, etc.).
  - Edge Mode: Determines whether to detect the index pulse on rising or falling A/B input.
  - Learn Mode: Enables capturing the home position for homing calibration.
  - Latch Inputs: Allows selecting which axis (X or Y) to latch during Latch0 and Latch1 events.
  - Home and Index Polarity: Configurable polarity for home and index inputs.
  - Accumulator Overflow Detection: Detects overflow of the internal count accumulator.

The entity QuadXface defines the interface of the module. Here is a description of each port:

H1_CLKWR: Clock signal.
SysClk: System clock signal.
SynchedTick: Synchronized tick signal, typically used for timed actions.
intDATA: Input data as a 32-bit vector.
QuadDataOut: Output data as a 32-bit vector.
CountRead, LEDStatusRead, LEDStatusWrite, InputRead, HomeRead, Latch0Read, Latch1Read: Control signals used to select certain operations within the architecture.
Home, RegistrationX, RegistrationY, LineFault, A, B, Index: Additional inputs to the module.

Architecture

The architecture QuadXface_arch defines the logic of the module. There are numerous signals and constants declared, including PosDir and NegDir which are constants set to '1' and '0' respectively, and are likely to be used to denote positive and negative directions.
Multiple 16-bit signals are declared to hold values for latches and registers (Latch0Reg, Latch1Reg, HomeReg, QuadLatch, QuadCount). Additional 16-bit high signals are sign extended to 32-bit signals (QuadSignExt, HomeSignExt, Latch0SignExt, Latch1SignExt).
Various 1-bit std_logic signals are declared, most of them possibly used as flags or control signals for various operations within the module. For example, Increment and Decrement might be used to control up/down counting, and IllegalTransition may be used to indicate an invalid state change.
The architecture also defines an array of signals like QA, QB, QZ, QH, QL0, QL1, that are each 4-bit and 3-bit std_logic vectors respectively. They could be used to store intermediate states for processing.

Several assignments are made within the architecture. For instance, the high 16 bits of the 32-bit QuadSignExt, HomeSignExt, Latch0SignExt, Latch1SignExt signals are assigned from the 16th bit of their corresponding 16-bit signals (QuadLatch, HomeReg, Latch0Reg, Latch1Reg). These operations perform sign extension to allow 16-bit values to be represented in a 32-bit vector.

The QuadDataOut output is conditional on several signals such as CountRead, LEDStatusRead, InputRead, HomeRead, Latch0Read, Latch1Read. The multiplexer-like structure implemented through the when-else keywords allows different sources of data to be selected for output based on the condition being tested.

The first process in the architecture is clocked on the H1_CLKWR signal and seems to be used to configure several aspects of the module's operation based on the LEDStatusWrite signal and intDATA input vector.

QuadCount: This process increments or decrements the counts based on the A/B sequence. The logic is as follows:

If SynchedTick is True, QuadCount is reset to 0.
If LatchedInc is True, QuadCount is incremented by 1.
If LatchedDec is True, QuadCount is decremented by 1.
If none of the conditions are True, PostCount is set to 0.

QuadLatch: This process transfers the counts to the latch when the control loop ticks comes by. MaxPosNum and MaxNegNum signals check for overflow of the transition counter. They are set to '1' when QuadCount equals X"7FFF" and X"8000", respectively.

intAccumOverflow logic handles overflow detection, and resets if SynchedTick is True.
Direction tracking is performed. The direction is '1' for positive direction (PosDir) and '0' for negative direction (NegDir) and is updated on the falling edge of the SysClk if there is an increment or decrement.

HomeRisingArmed, HomeFallingArmed, HomeIndexArmed, HomeIndexHomeArmed, and HomeIndexNotHomeArmed are set based on the HomeTriggerType and HomeArm.
RisingHome and FallingHome signals capture rising and falling edge transitions for the home input.
RisingHomeEvent and FallingHomeEvent are set when the respective edge transitions occur.
IndexEdgeEvent captures edge transitions of the Index pulse based on the direction of the axis.
EdgeDetectInput is set when IndexEdgeEvent is '1' and any of HomeIndexArmed, HomeIndexHomeArmed, or HomeIndexNotHomeArmed is '1'.

IndexEdgeDetected latches the detection of the edge of the Index Pulse if the index is used as part of the active homing routine.

intEdgeMode is updated based on the QB signal during the learning mode.
EdgeMode and intLearnModeDone are updated on the rising edge of H1_CLKWR and SysClk respectively.

IndexEvent, IndexHomeEvent, and IndexNotHomeEvent signals determine edge events and the home event for the index.

CaptureHomeCounts is set when a RisingHomeEvent, FallingHomeEvent, IndexEvent, IndexHomeEvent, or IndexNotHomeEvent occurs and the respective event is armed.

HomeReg captures the current counts when an Event occurs. It also manages the status bits for home latch.

Latch0Input, RisingLatch0, FallingLatch0, RisingLatch0Event, FallingLatch0Event, CaptureLatch0Counts, and Latch0Reg are similar to Home related signals but for Latch0.

Latch1Input, RisingLatch1, FallingLatch1, RisingLatch1Event, FallingLatch1Event, CaptureLatch1Counts, and Latch1Reg are similar to Home related signals but for Latch1.

ABreak, BBreak, ZBreak, AccumOverflow, IllegalTransition are updated during the SysClk and includes 3 for context:

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

Ram128x16bits

- **Module Name***: Ram128x16bits.vhd*
- **Description:** The RAM128x16 module creates a 128x16-bit random access memory (RAM) capable of storing and retrieving data based on provided address and control signals. It serves as a reliable and efficient means of storing and retrieving data within the RMC75E modular motion controller, making it an integral component for various motion control operations.

Inputs:

- clk: The clock signal used for synchronous operations.
- we: The write enable signal that controls write operations.
- a: The address input signal used to specify the memory location to access.
- d: The data input signal containing the 16-bit data to be written into the specified memory location.

Outputs:

- o: The data output signal that provides the 16-bit data read from the specified memory location.

Architecture: The architecture of the RAM128x16 module, named RAM128x16_arch, utilizes an array type named ram_type. The ram_type represents an array of 128 elements, where each element is a 16-

bit std_logic_vector, serving as the main storage for the RAM module. The RAM128x16 module includes the following components:

- RAM: A signal of type ram_type, representing the actual memory storage. It is an array with 128 elements, each capable of storing a 16-bit value. The initial value of the RAM signal is set to all '1's.
- read_a: A signal of type std_logic_vector(6 downto 0), used to hold the current address input for read operations.

The behavior of the RAM128x16 module is defined within a process block sensitive to the clk signal. The process handles both write and read operations based on the rising edge of the clock signal.

During a rising edge of the clock, the module checks if we (write enable) is asserted. If so, the module writes the 16-bit data d into the memory location specified by the address a. The address a is converted to an integer index for RAM access.

The read_a signal is continuously updated with the current address a to ensure the correct data is read from the RAM during subsequent clock cycles.

Finally, the output signal o is assigned the value stored in the RAM at the memory location specified by read_a, providing the requested 16-bit data output.

Initialization Note: The RAM signal should have a value initialization to avoid undefined behavior in the analog module. The RAM128x16 module sets the initial value of the RAM signal to all '1's. This initialization ensures that the RAM module is not left undefined when initialized, allowing it to operate correctly within the modular motion controller.

- **Interactions with Other Modules:**
- **Test bench and simulation:**

Rtdexpidled

- **Module Name***: Rtdexpidled.vhd*
- **Description:** The RtdExpIDLED module serves as a multiplexer and control logic for selecting and controlling the clock, latch, and load signals for an external LED module based on the state of the DiscoveryComplete signal. It reroutes the control of the CLK, LATCH, and LOAD lines for the Expansion modules. Initially, these control lines are used to capture the ID of the modules and then switch to control the LED colors of the modules after the discovery process is complete.

Inputs:

- DiscoveryComplete: A control signal indicating the completion of the device discovery process.
- Exp_ID_CLK: The clock signal for the internal ID module.
- Exp_ID_LATCH: The latch signal for the internal ID module.

- Exp_ID_LOAD: The load signal for the internal ID module.
- ExpLEDOE: The output enable signal from the external LED module.
- ExpLEDLatch: The latch signal from the external LED module.
- ExpLEDClk: The clock signal from the external LED module.

Outputs:

- Exp_Mxd_ID_CLK: The selected clock signal for the ID module.
- Exp_Mxd_ID_LATCH: The selected latch signal for the ID module.
- Exp_Mxd_ID_LOAD: The selected load signal for the ID module.

Architecture: The architecture of the RtdExpIDLED module, named RtdExpIDLED_arch, includes a process block that controls the selection of clock, latch, and load signals based on the state of the DiscoveryComplete signal and the signals from the external LED module.
During the process, if the DiscoveryComplete signal is low (0), indicating that the device discovery process is not yet complete, the internal ID signals (Exp_ID_CLK, Exp_ID_LATCH, Exp_ID_LOAD) are selected and assigned to the corresponding output signals (Exp_Mxd_ID_CLK, Exp_Mxd_ID_LATCH, Exp_Mxd_ID_LOAD).

However, if the DiscoveryComplete signal is high (1), indicating that the device discovery process is complete, the signals from the external LED module (ExpLEDClk, ExpLEDLatch, ExpLEDOE) are selected and assigned to the corresponding output signals (Exp_Mxd_ID_CLK, Exp_Mxd_ID_LATCH, Exp_Mxd_ID_LOAD).

The RtdExpIDLED module provides the necessary logic to control the selection of clock, latch, and load signals based on the state of the DiscoveryComplete signal. This functionality allows seamless integration of the external LED module into the RMC75E modular motion controller, enabling the control of LED colors once the device discovery process is complete.

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

Serial2parallel

- **Module Name***: serial2parallel.vhd*
- **Description:** The "Serial2Parallel" module acts as a serial-to-parallel converter and data distributor within the RMC75E modular motion controller. It takes serial input data and converts it into parallel format, distributing the data to various output channels based on control signals.

- **Inputs:**
  - SysClk: System clock signal.
  - SynchedTick: Synchronized tick signal.
  - CtrlAxisData: Two-bit control axis data input.
  - ExpA_DATA: Eight-bit data input for expansion module A.
  - Serial2ParallelEN: Enable signal for the module.
  - Serial2ParallelCLR: Clear signal for internal data registers.
  - S2P_Addr: Four-bit address input for selecting the output channel.

- **Outputs:**
  - S2P_Data: Sixteen-bit parallel data output.

- **Internal Signals:**
  - Internal registers hold 16-bit data for each output channel.

  - **Functionality:**

1. On the rising edge of SysClk, when SynchedTick or Serial2ParallelCLR is high, all internal data registers are cleared.
2. When Serial2ParallelEN is high, data conversion and distribution are enabled. Input data is shifted into internal registers for each output channel.
3. S2P_Data output is determined by S2P_Addr, providing the selected output channel's data.

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

Serial_mem

Module Name*: serial_mem.vhd (SerialMemoryInterface)*

Description: The `SerialMemoryInterface` module is designed as a serial memory interface for the RMC75E modular motion controller built by Delta Motion. The module provides communication with an external serial EEPROM, specifically the AT24C01A device.

The **SerialMemoryInterface** component converts data from parallel to serial format before sending it to the EEPROM (Electrically Erasable Programmable Read-Only Memory). This conversion is necessary because many EEPROM devices communicate using a serial protocol, which means they expect data to be sent one bit at a time.

Here's how the data conversion process works:

1. **Data Parallelization**: The external system provides data in parallel format through the **intDATA** signal, which is a 32-bit wide vector. This data needs to be written to the EEPROM.
2. **Serialization**: The **SerialMemoryInterface** converts this parallel data into a serial stream. It does so by sequentially sending each bit of the data to the EEPROM on the **intSerialMemoryDataOut** signal. The **intSerialMemoryClock** signal controls the timing of the data transmission.
3. **Start and Stop Bits**: Before sending the actual data, the interface generates start and stop bits. A start bit (logic '0') indicates the beginning of the data transmission, while a stop bit (logic '1') marks the end of the data.
4. **Shift Register**: The component uses a shift register (**SerialDataOutput**) to hold the data temporarily while serializing it. The data is shifted out bit by bit during the communication process.
5. **ACK Handling**: After sending the data, the component waits for an acknowledgment signal (**ACK**) from the EEPROM. This ACK signal confirms that the EEPROM has received the data correctly.

For a write operation, the state machine follows a specific sequence (Start, Device Address, Memory Address, Data) to write the data into the EEPROM. ACK handling is essential to ensure successful data transmission.

The component also supports reading data from the EEPROM using similar serialization and parallelization processes. However, in the read operation, the ACK handling is not required as data is being read from the EEPROM rather than written to it.

Overall, the **SerialMemoryInterface** is responsible for managing the communication protocol and data format conversion between the external parallel data bus and the EEPROM's serial interface.

- Inputs:
  - `SysReset`: System Reset or PLL not locked signal.
  - `H1_CLK`: 60MHz clock signal.
  - `SysClk`: 30MHz system clock signal.
  - `SlowEnable`: Signal to enable slow operations.
  - `intDATA`: Input data to be written to the serial memory.
  - `SerialMemXfaceWrite`: Signal to initiate a write operation.
  - `SerialMemoryDataIn`: Input data from the serial memory.

- Outputs:
  - `serialMemDataOut`: Output data from the serial memory.
  - `SerialMemDataOut`: Output data from the serial memory (connected to the `serialMemDataOut` internally).
  - `SerialMemoryDataControl`: Control signal for the serial memory data.
  - `SerialMemoryClk`: Clock signal for the serial memory.

- `Exp0SerialSelect`, `Exp1SerialSelect`, `Exp2SerialSelect`, `Exp3SerialSelect`: Selection signals for expansion slots.
- `EEPROMAccessFlag`: Flag indicating access to the EEPROM.
- `M_SPROM_CLK`: Clock signal for the serial EEPROM module.
- `M_SPROM_DATA`: Bidirectional data signal for the serial EEPROM module.

The module operates based on a state machine that controls serial communication with the AT24C01A EEPROM. The state machine progresses through different states to perform operations such as writing to and reading from the EEPROM.

The main states of the state machine are as follows:
1. IdleState: The initial state where the module waits for a write or read operation to be triggered.
2. SignalStartState: Sends a start condition to initiate the communication.
3. DeviceAddrState: Loads the device address into the shift register for transmission.
4. CheckDeviceAddrACKState: Checks the acknowledgment (ACK) received after sending the device address.
5. MemAddrState: Loads the memory address into the shift register for transmission.
6. CheckMemoryAddrACKState: Checks the ACK received after sending the memory address.
7. OperationTypeState: Determines the operation type (write or read) based on the WriteFlag signal.
8. WriteState: Transmits the data to be written to the EEPROM.
9. WriteAckState: Checks the ACK received after writing data.
10. ReadState: Reads data from the EEPROM by shifting it out bit by bit.
11. ReadNoACKState: Sends a NACK (no acknowledgment) after reading data.
12. StopState: Sends a stop condition to end the communication.
13. ClearState: Clears internal flags and prepares for the next operation.

The module generates the necessary control signals and manages the data transfer between the motion controller and the serial memory. It also handles error conditions and keeps track of operation faults.

The module has the following input ports:
- `SysReset`: System reset signal or PLL not locked indication.
- `H1_CLK`: 60MHz clock signal.
- `SysClk`: 30MHz system clock signal.
- `SlowEnable`: Signal to enable slow operation mode.
- `intDATA`: 32-bit input data from the motion controller.
- `SerialMemXfaceWrite`: Signal indicating a write operation to the serial memory device.
- `SerialMemoryDataIn`: Input data from the serial memory device.
- `M_SPROM_DATA`: Bidirectional data signal for communication with the Serial EEPROM.

And the following output ports:
- `serialMemDataOut`: 32-bit data output for the motion controller.
- `SerialMemoryDataOut`: Output data signal to the serial memory device.
- `SerialMemoryDataControl`: Output control signal for the serial memory device.
- `SerialMemoryClk`: Output clock signal for the serial memory device.
- `Exp0SerialSelect`, `Exp1SerialSelect`, `Exp2SerialSelect`, `Exp3SerialSelect`: Output select signals for expansion modules.
- `EEPROMAccessFlag`: Output signal indicating access to the Serial EEPROM.
- `M_SPROM_CLK`: Output clock signal for the Serial EEPROM.

- `M_SPROM_DATA`: Bidirectional data signal for communication with the Serial EEPROM.

At the lowest and most granular level, the **SerialMemoryInterface** module operates using a clocked state machine to control the data transfer between the processor (CPU) and the Serial EEPROM device. Let's delve into the granular details of how the module works:

1. Clock Generation:
    - The module generates two clock signals: **SerialMemoryClk** and **intSerialMemoryClock**.
    - The **SerialMemoryClk** is used for clocking data in and out of the Serial EEPROM device.
    - The **intSerialMemoryClock** is an internally generated clock used for timing within the module.
    - The **SerialMemoryClockEnable** signal controls the generation of the **intSerialMemoryClock** and ensures the clock period does not exceed 100 kHz.
2. State Machine:
    - The module employs a state machine (**StateMachine**) to manage the communication process with the Serial EEPROM device.
    - The state machine sequentially moves through different states to carry out read and write operations.
3. Data Transfer:
    - The **SerialDataOutput** and **SerialDataInput** signals are used to buffer data being transmitted to and received from the Serial EEPROM device.
    - The **SerialDataOutput** signal contains the data to be sent to the EEPROM during write operations, while **SerialDataInput** holds the data received during read operations.
4. Communication Protocol:
    - The module follows a specific communication protocol to interact with the Serial EEPROM device.
    - Communication begins with a "Start" condition, followed by sending the device address, memory address, and data during write operations.
    - For read operations, the module sends the device address and memory address, followed by a repeated "Start" condition to initiate read data retrieval.
5. Data Shift Register:
    - The module utilizes a shift register to serialize and deserialize data for communication with the Serial EEPROM device.
    - Various control signals (**ShiftEnable**, **LoadDeviceAddr**, **LoadMemAddr**, and **LoadWriteData**) control the loading and shifting of data in and out of the shift register.
6. Error Handling:
    - The module includes logic to handle cases where the Serial EEPROM device does not acknowledge communication attempts (NO ACK response).
    - The **OperationFaultFlag** and **OperationFaultCount** signals are used to detect and track communication failures.
7. Control Signals:
    - The module uses several control signals (**WriteFlag**, **ReadFlag**, **ACK**, **StartStopBit**, etc.) to coordinate the various stages of data transfer and communication with the Serial EEPROM device.
8. Tri-state Buffering:
    - To enable bidirectional communication with the Serial EEPROM device, the module uses tri-state buffers for **M_SPROM_DATA**, allowing it to drive the data line when needed and relinquish control when not required.
9. Interface Selection:

- The module uses various control signals (**ControlSerialSelect**, **Exp0SerialSelect**, etc.) to determine which external module interface to interact with during EEPROM access.

Overall, the **SerialMemoryInterface** module orchestrates the precise timing and control signals required for reliable data transfer with a Serial EEPROM device. It uses the state machine to guide the communication protocol and ensures the correct sequence of operations for both read and write functionalities.

The module is implemented using an architecture called `SerialMemoryInterface_arch`. It contains several constants, signals, and processes that control the state machine and data flow between the motion controller and the serial memory device.

The architecture defines various constant values used for device addresses, clock terminal counts, and selection addresses for expansion modules. These constants are converted to the appropriate vector types using the `To_StdLogicVector` function.

The module uses a state machine (`StateMachine`) to control serial communication with the Serial EEPROM. The state machine has several states, including IdleState, SignalStartState, DeviceAddrState, CheckDeviceAddrACKState, MemAddrState, CheckMemoryAddrACKState, OperationTypeState, WriteState, WriteAckState, ReadState, ReadNoACKState, StopState, and ClearState.

The architecture includes processes that handle clock generation, write and read flags, module and memory addresses, data buffering, ACK (acknowledgment) signal handling, and state transitions. The module also generates control signals (`intSerialMemoryDataControl`, `intSerialMemoryClock`, `intSerialMemoryDataOut`) and manages the selection signals for the expansion modules (`Exp0SerialSelect`, `Exp1SerialSelect`, `Exp2SerialSelect`, `Exp3SerialSelect`).

Additionally, there are processes that handle the shifting of data bits, capturing input data, counting shifted bits, handling operation faults, and generating the final output signals (`serialMemDataOut`, `intEEPROMAccessFlag`, `intOperationFaultFlagInput`, `OperationFaultFlag`).

Overall, this module provides an interface for the RMC75E motion controller to communicate with a Serial EEPROM, enabling read and write operations and managing various control signals and data transfers.

The **M_SPROM_DATA** signal is part of the **SerialMemoryInterface** entity, which is the interface to communicate with a serial EEPROM (Electrically Erasable Programmable Read-Only Memory) device via the I2C protocol, specifically the AT24C01A.

The purpose of the **M_SPROM_DATA** signal is to provide bidirectional data communication with the serial EEPROM module when the interface is selected for communication. This signal is declared as **inout**, which means it can be used as both an input and an output. It allows data to be sent to the serial EEPROM for write operations and also receives data from the serial EEPROM during read operations.

The specific data transmitted on this line depends on the operation being performed (read or write) and the address and data being sent. The data format and protocol are controlled by the state machine (**StateMachine**) in the architecture. The state machine manages the entire communication sequence, including address transmission, read/write operations, and handling acknowledgment signals.

Here's a summary of the main states and operations of the state machine in the architecture:

1. SignalStartState: This state is responsible for initiating communication with the serial EEPROM by sending a Start condition on the bus.
2. DeviceAddrState: In this state, the device address of the serial EEPROM is loaded into the shift register to prepare for transmission.
3. CheckDeviceAddrACKState: After sending the device address, the state machine checks for acknowledgment (ACK) from the serial EEPROM. If ACK is received, it proceeds to the next state; otherwise, it may retry or signal an operation fault.
4. MemAddrState: In this state, the memory address for read/write operations is loaded into the shift register to prepare for transmission.
5. CheckMemoryAddrACKState: Similar to the device address check, this state checks for acknowledgment (ACK) for the memory address sent.
6. OperationTypeState: This state determines the type of operation to be performed (read or write).
7. WriteState: In write operations, this state is responsible for transmitting the data to be written to the serial EEPROM.
8. WriteAckState: After sending the data, this state checks for acknowledgment (ACK) from the serial EEPROM. If ACK is received, it proceeds to the StopState; otherwise, it may retry or signal an operation fault.
9. ReadState: In read operations, this state enables the counter to clock, indicating the number of bits to be read.
10. ReadNoACKState: In read operations, this state outputs a '1' during the ACK phase.
11. StopState: The state machine signals a Stop condition to end the communication sequence.
12. ClearState: This state clears the FLAG_CLR signal, indicating the end of the operation.

The **M_SPROM_DATA** signal will carry different data during each of these states to perform the required read or write operations with the serial EEPROM.

Here we can see a bit-by-bit breakdown of the data contained in the M_SPROM_DATA signal:

Bit 31: Operation Fault Flag - Indicates if there was an operation fault during EEPROM communication.

Bit 30: Serial Data Input Bit 7 - Represents the most significant bit of the data input during read operations.

Bit 29: Serial Data Input Bit 6 - Represents the 6th bit of the data input during read operations.

...

Bit 23: Serial Data Input Bit 0 - Represents the least significant bit of the data input during read operations.

Bit 22: Operation Fault Flag Input - Input to check if there was an operation fault during EEPROM communication.

Bit 21: Inc Operation Fault Count - Signal to increment the operation fault counter during communication.

Bit 20: Second Pass Read - Used by the Read operation to loop through the state machine multiple times.

Bit 19: Start/Stop Bit - Output to signal a Start or Stop condition during communication.

Bit 18: Shift Enable - Signal to enable the shifting of data during communication.

Bit 17: Serial Memory Data Control - Signal to control the direction of data transmission (input/output).

Bit 16: EEPROM Access Flag - Output to indicate control over the data line to the EEPROM.

Bit 15: Expansion Module Select Address Bit 2 - Used to select the appropriate expansion module.

Bit 14: Expansion Module Select Address Bit 1 - Used to select the appropriate expansion module.

Bit 13: Expansion Module Select Address Bit 0 - Used to select the appropriate expansion module.

Bit 12: Control Module Select - Signal to select the control module.

Bit 11: Memory Address Bit 5 - Represents the 6th bit of the memory address for read/write operations.

...

Bit 6: Memory Address Bit 0 - Represents the least significant bit of the memory address for read/write operations.

Bit 5: Data Buffer Bit 7 - Represents the most significant bit of the data buffer for write operations.

...

Bit 0: Data Buffer Bit 0 - Represents the least significant bit of the data buffer for write operations

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

SSITop

- **Module Name***: SSITop.vhd*
- **Description:** SSI Interface will provide a signal interface to Synchronous Serial Interface type linear and rotary transducers. The module communicates with the transducers through an SSI interface and processes the received data for further use. The module handles various aspects of SSI communication, such as clock generation, data transfer, and wire break detection. Below is a concise technical description of the module:

- **Inputs & Outputs:**

    - H1_CLKWR: A 60MHz system clock.
    - SysClk: A 30MHz system clock.
    - Enable: A 7.5MHz system enable signal, active every 4th 30MHz clock.
    - SlowEnable: A 3.75MHz system enable signal, active every 8th 30MHz clock.
    - SynchedTick: A control loop tick signal, valid on the rising edge of the 30MHz clock.
    - intDATA: A 32-bit data input representing configuration and control parameters.
    - ssiDataOut: A 32-bit data output representing the processed data or status for external access.
    - PositionRead: A signal indicating that the SSI data needs to be read.
    - StatusRead: A signal indicating that the status data needs to be read.
    - ParamWrite1, ParamWrite2: Signals to enable writing to specific parameters.
    - SSI_CLK: An output signal representing the SSI clock.
    - SSI_DATA: An input signal representing the SSI data line.
    - SSISelect: An output signal to select between different transducer types (Binary Analog or Gray Analog).

- **Internal Signals:**

    - The module uses various internal signals to process and manage the SSI communication, including counters, latches, and control signals. It implements the following functionalities:
    -
    - Configuring and managing the transducer selection, clock rate, and data length based on the input parameters (intDATA).
    - Generating the SSI clock (SSI_CLK) for data transfer.
    - Handling the data transfer process, including shifting in incoming data, checking and latching the data line status, and detecting wire breaks.
    - Controlling the SSI read cycle, enabling the clock, and shifting in external data into a holding register.
    - Managing clock toggling at the requested clock rate (ClockRate).
    - Performing data validation and clearing the DataValid status bit.
    - Latching the contents of the shift register and clearing it for the next input cycle.

- **Processes and State Machines:**

- **Key Components:**

    - Overall, the "SSITop" module facilitates communication with SSI type transducers, processes the received data, and provides the necessary control and management signals for proper data transfer and wire break detection.

- **Important Calculations:**

- **Interactions with Other Modules:**

- **Test bench and simulation:**

Statemachine

- **Module Name*:*** *statemachine.vhd*

- **Description:** This module implements a state machine to control the sampling of data from an ADC. It performs sample/hold operations and conversions based on a specified loop time. The ExpA_CLK signal exhibits a pattern where it is active every 8th system clock with a frequency of 3.75 MHz. The exact polarity of the clock (positive or negative) depends on the SlowEnable signal. The behavior and timing of the module ensure proper synchronization and sequencing of the ADC operations within the system.

- **Inputs & Outputs:**

  - The module takes several input signals, including:
  - SysReset: A system reset signal or an indication that the PLL (Phase-Locked Loop) is not locked.
  - SysClk: A 30 MHz system clock.
  - SlowEnable: An enable signal that is active every 8th system clock (3.75 MHz).
  - SynchedTick: A loop tick signal synchronized to the system clock.
  - LoopTime: A 3-bit vector specifying the loop time duration (from 1/8 ms to 4 ms).
  - It provides the following output signals:
  - ExpA_CS_L: A chip select signal for the ADC (Analog-to-Digital Converter).
  - ExpA_CLK: A clock signal for the ADC.
  - Serial2ParallelEN: An enable signal for a shift register used for ADC output.
  - Serial2ParallelCLR: A clear signal for the shift register used for ADC output.
  - WriteConversion: An end-of-conversion indicator.
  - The module operates based on a state machine with various states defined by constants. Here are the state encodings:
  - StartState: "000"
  - SampleHoldState: "001"
  - ConvertState: "011"
  - ConvertWaitState1: "010"
  - ConvertWaitState2: "110"
  - ConvertDoneState: "111"
  - IncConvCountState: "101"
  - InterConvDelayState: "100"

- **Internal Signals:**
  - The module uses delay times to spread the data samples over the control loop period. The delay values are based on the specified loop time. Here are the defined delay constants for different loop times:
  - eighth_ms_interconversion_delay: 5 counts
  - quarter_ms_interconversion_delay: 60 counts
  - half_ms_interconversion_delay: 180 counts
  - one_ms_interconversion_delay: 412 counts
  - two_ms_interconversion_delay: 881 counts

- four_ms_interconversion_delay: 1820 counts
- The module includes internal signals and registers to keep track of various conditions and timing. Here are some of the significant signals and registers:
- State: A 3-bit signal representing the current state of the state machine.
- EndDelay: A flag indicating whether the module needs to introduce a delay between conversions.
- SampleHoldDone: A flag indicating the completion of sample/hold operation.
- ConversionDone: A flag indicating the completion of the conversion process.
- ConversionCounterEN: A signal enabling the conversion counter.
- ExpA_CLK_EN: A signal enabling the ADC clock.
- intExpA_CLK: A 2-bit vector representing the internal ADC clock.
- ConversionCounter: A 4-bit vector counting the number of conversions within a loop time.
- CycleCounter: A 5-bit vector used to track the logic sequence during individual conversions.
- InterConversionDelayCNTR: An 11-bit vector used to insert a delay between conversions.
- InterConversionDelayEN: A signal enabling the interconversion delay.
- preInterConversionDelayTC: A signal indicating whether the terminal count for interconversion delay is reached.
- intConverting: An internal signal indicating whether the module is currently converting.
- Converting: A synchronized version of intConverting.
- intWriteConversion and intWriteConversion2: Internal signals used to trigger data buffer writes.
- intSerial2ParallelEN: An internal signal indicating whether the converter data is being received.
- WriteEN: A signal indicating whether a write conversion is in progress.
- asyncResetCycleCounter and ResetCycleCounter: Signals used to reset the cycle counter.

- **Key Components:**

- **Important Calculations:**

- **Interactions with Other Modules:**

- **Test bench and simulation:**

Ticksync

- **Module Name***: ticksync.vhd*

- **Description:** The TickSync module synchronizes the incoming control loop clock tick with the internal clock, making it synchronous and usable by the internal logic. It provides the SyncedTick pulse, which is output immediately after syncing up to the incoming tick pulse.

Design Details: The TickSync module is designed to synchronize the incoming LOOPTICK signal with the SysClk signal. It also generates a 60 MHz H1_CLK signal to simulate the incoming clock. The module includes the following ports:

Inputs:

- SysReset: Asynchronous reset signal to initialize loop ticks low.
- SysClk: The system clock.
- H1_CLK: Input clock with a frequency of 60 MHz.
- LOOPTICK: The incoming control loop clock tick.

Outputs:

- SynchedTick: Synchronized tick output.
- SynchedTick60: Synchronized 60 MHz tick output.

Functionality: Upon reset, the SysReset signal initializes LOOPTICK and SynchedTick to low. The TickSync module then synchronizes LOOPTICK to SysClk by using rising_edge detection on SysClk. The synchronized tick output, SynchedTick, is set to high during the rising edge of SysClk when LOOPTICK is high. The module generates a 60 MHz clock, H1_CLK, using rising_edge detection on SysClk, which will serve as an input clock for testing purposes. Additionally, the SynchedTick60 output is set to high during the rising edge of H1_CLK when LOOPTICK is high.

- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

Top

- **Module Name***: top.vhd*
- **Description:** The uppermost module that ties together all other components in the system, essentially acting as a giant wrapper module.
- **Inputs:**
- **Outputs:**
- **Internal Signals:**
- **Processes and State Machines:**
- **Key Components:**
- **Important Calculations:**

- **Interactions with Other Modules:**
- **Test bench and simulation:**

WatchDogTimer

- **Module Name:** *WatchDogTimer.vhd*
- **Description:** The "WDT" entity represents a WatchDogTimer unit responsible for counting down from a user-defined value. When the counter reaches 0x1, the WDT expiration value is set. The module has various inputs and outputs used to control its functionality.
- **Inputs:**
  - RESET: Asynchronous reset input.
  - SysRESET: System reset input.
  - H1_CLKWR: Clock input for synchronization.
  - SysClk: System clock input.
  - intDATA: Input data from the CPU.
  - FPGAAccess: Flag indicating if the CPU is reading or writing to the FPGA.
  - WDTConfigWrite: Input to write configuration data to the WDT.
  - FPGAProgrammedWrite: Input used to verify if the FPGA is programmed.
  - SlowEnable: Enable signal to operate the WDT in slow mode.
  - HALT_DRIVE_L: Input from Drive Enable hardware watchdog timer.
  - WD_TICKLE: Input from CPU to tickle the watchdog timer.
- **Outputs:**
    - wdtDataOut: Output data from the WDT.
    - FPGAProgDOut: Output data for verifying FPGA existence and programming.
    - WD_RST_L: Output to trigger module reset.
    - WDT_RST_L: Output indicating the WDT reset status.
- **Internal Signals:**
  - Various internal signals, such as WDTCounter, WDTDelay, FirstKey, AccessKey, and others, are used for internal control and synchronization.

- **Functionality:**
  1. The WDT unit counts down from a user-defined value specified in the WDTDelay signal.
  2. The WDTExpiration flag is set when the counter reaches 0x1.
  3. The WDTConfigWrite input allows the CPU to write configuration data to the WDT, such as WDTDelay, FPGAResetStatus, DriveHaltStatus, and FPGA_RstReq.
  4. The FirstKey and AccessKey signals are used to enable access to specific configuration settings in the WDT based on specific writes from the CPU.
  5. The WDTKick and LoadWDTCount signals are used to restart the watchdog counter based on specific conditions, including CPU tickling or special sequences of writes to the WDT configuration register.
  6. The FPGAProgrammedWrite input is used for verifying the FPGA existence and programming. The WDT provides output data in FPGAProgDOut for this verification.
  7. The SlowEnable signal enables the WDT to operate in slow mode.
  8. The WD_RST_L signal is generated to trigger a module reset based on the WDT expiration.
  9. The PowerUp process detects power-up conditions by monitoring PUReg.

- **Key Components:**
- **Important Calculations:**
- **Interactions with Other Modules:**
- **Test bench and simulation:**

System Overview

This section provides a brief overview of some of the major modules.

1. Top: The uppermost module that instantiates all other modules in the system. It provides the interface to all the axis modules and expansion modules on the RMC75E, as well as control of the LEDs on the CPU module. The main components of the Top module include:
- Bus interface to the MPC5200 processor
- Sensor interface logic

2. Decode: Responsible for generating WRITE control lines and READ control lines based on the input address and control signals. It serves as a decoder module for interfacing with various peripheral devices.

Key features of the module:

- Takes inputs including address (ADDR), read (RD_L), write (WR_L), and chip select (CS_L) control signals, along with other control signals for different peripheral devices.
- Supports multiple peripheral devices like FPGA, analog interface, quadrature decoder, etc., with respective read and write control signals.
- Provides output control signals for each peripheral to enable/disable read or write operations.
- Supports various peripherals including FPGA (read/write), CPU configuration (read/write), CPU LED (read/write), Watch Dog Timer (read/write), PROFIBUS address, Serial Memory Interface (read/write), etc.
- Handles different peripheral addressing to communicate with multiple instances of a peripheral type.
- Decoding involves conditional assignments of signals based on the input address and control signals to activate the correct peripheral control signals.

3. Serial_mem: The serial memory interface for the RMC75E modular motion controller. It facilitates communication with an external serial EEPROM, specifically the AT24C01A device. The SerialMemoryInterface component converts data from parallel to serial format before sending it to the EEPROM (Electrically Erasable Programmable Read-Only Memory). This conversion is necessary because AT24C01A EEPROM devices communicate using the I2C serial protocol, which sends data one bit at a time.

4. Quad: Acts as a wrapper for multiple QuadXface components, enabling processing of quadrature signals and generating output data for multiple quadrature interfaces and axes in the RMC75E motion controller. QuadXface components are instantiated for each quadrature interface and axis, with their ports connected to corresponding signals from the Quad entity. Instantiation maps signals to inputs and outputs of QuadXface components.

5. QuadXface:

A quadrature counter module designed for use with incremental encoders or similar devices. It counts pulses from A and B inputs and detects rotation direction. The module supports four count registers: QuadCount, HomeReg, Latch0Reg, and Latch1Reg. Features include homing and capturing counts:

- Homing supports triggers like rising edge, falling edge, index edge, etc.
- Capture can capture QuadCount on specific events (rising/falling edges of A or B, index edge).
- Configurable settings: Homing trigger type, edge mode, learn mode, latch inputs, home and index polarity, accumulator overflow detection.

6. MDTTOPSimp:

Provides a signal interface to SPWM, Start/Stop, and SSI type magnetostrictive displacement transducers. Clock rate is 60MHz for all three clocks, SysClk0-90 are 90 degrees out of phase. Each clock can drive a counter when the return pulse signals a start. Counters are added at the end of the cycle.

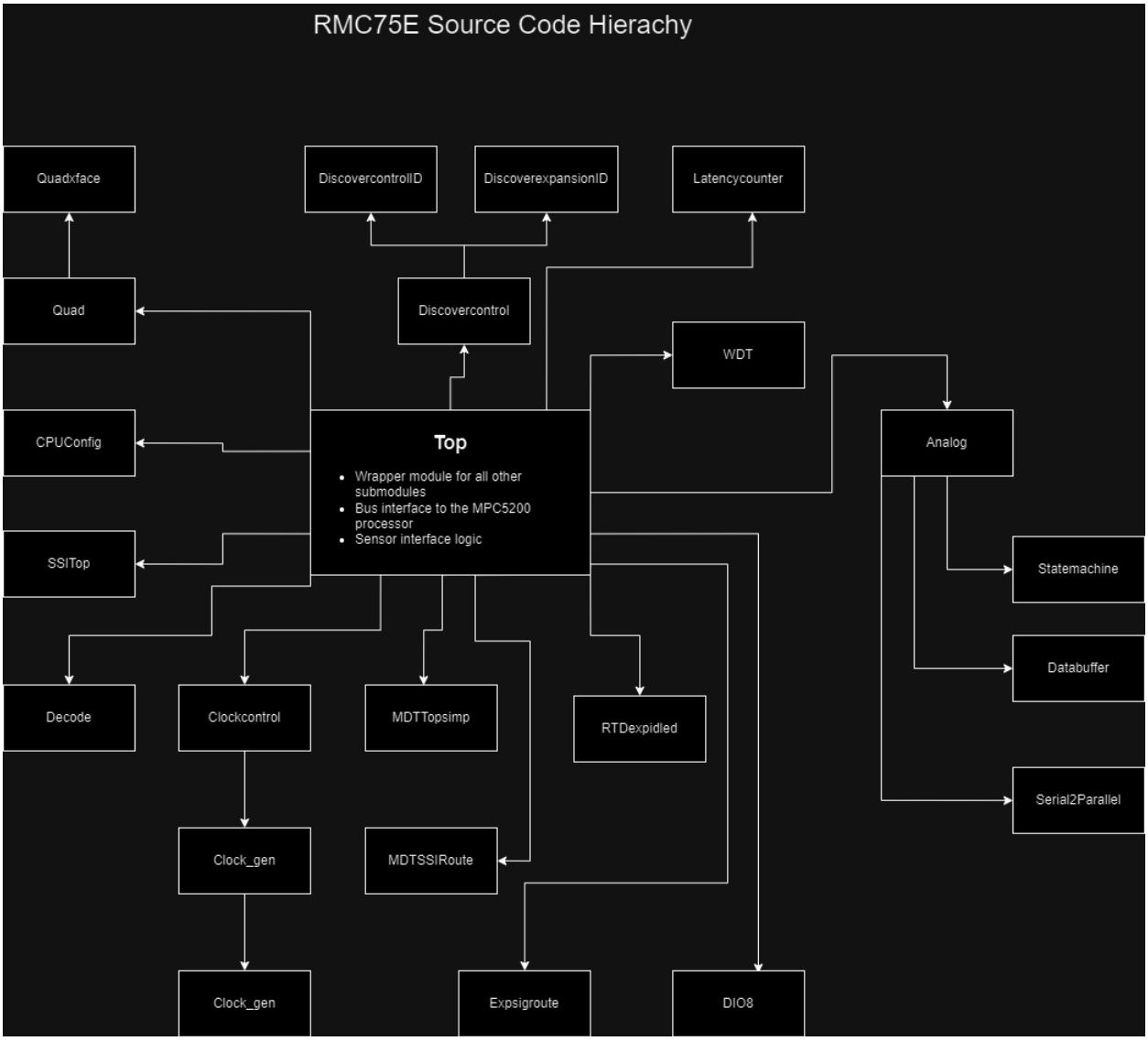
7. ControlIO:

Provides Control Board IO and LED Interface. LED information is clocked out to 74HCT595 devices and locks on the rising clock edge. Responsible for managing IO operations and LED status for Axis0 and Axis1.

8. DIO8:

An 8-bit digital input/output (DIO) interface for multiple expansion slots. Provides access to four DIO8 modules, each with eight bidirectional digital I/O channels. Operates on a clocked architecture with a state machine controlling read and write sequences to DIO8 modules.

Hierarchy Diagram



Testing and Verification

Appendices

References