Technical Analysis of the functional safety in TMS

Overview:

Battery systems used in modern energy storage and electric mobility applications operate under strict safety constraints, as temperature-related faults can lead to severe hazards such as thermal runaway, cell degradation, or system failure. Reliable temperature monitoring is therefore a critical requirement in any Battery Management System (BMS). To address this challenge, this project implements a functional safety–oriented battery temperature management system using a redundant processing architecture on an FPGA platform.

The primary objective of the system is to ensure safe and reliable temperature monitoring by detecting sensor faults, communication errors, or processing anomalies before they can lead to unsafe operating conditions. This is achieved through the use of Dual Modular Redundancy (DMR) with an asymmetric master–monitor architecture, a well-established approach in safety-critical embedded systems.

The system is composed of two independent processing nodes implemented as MicroBlaze soft processors. MicroBlaze 0 (MB0) acts as the primary safety controller and is responsible for all safety-critical decisions. MicroBlaze 1 (MB1) operates in the background as a monitoring unit, continuously measuring temperature and supplying redundant data to the primary controller. Both processors acquire temperature readings from independent TMP3 sensors, ensuring physical and logical separation between the sensing paths.

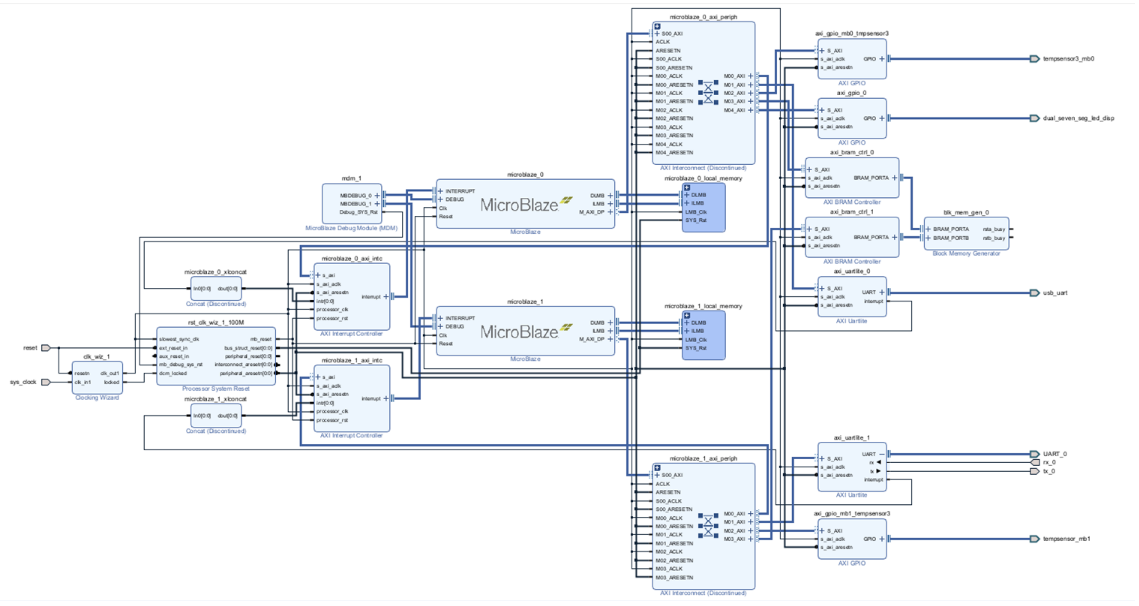
Temperature values measured by MB1 are transmitted to MB0 through shared on-chip memory. MB0 performs a continuous comparison between its own measurements and the values received from MB1. As long as the difference between the two measurements remains within a predefined tolerance, the system is considered to be operating correctly. Under these conditions, MB0 alone is permitted to report temperature values and control system outputs.

If the temperature difference exceeds a safety threshold, the system interprets this condition as a fault, potentially caused by sensor failure, communication errors, or processing malfunction. In response, MB0 immediately transitions the system into a safe shutdown state, halting normal operation and preventing further propagation of potentially invalid data. This behavior reflects a fail-safe design philosophy, where any detected inconsistency results in conservative system behavior rather than continued operation.

This architecture represents an asymmetric Dual Modular Redundancy system, also known as a Master–Monitor or Controller–Checker architecture. Unlike lockstep or voting-based redundancy schemes, only the primary processor is authorized to make decisions and control outputs, while the secondary processor serves exclusively as a supervision and fault-detection mechanism. This approach reduces complexity while maintaining strong fault coverage, making it particularly suitable for functional safety applications such as battery management.

In addition to fault detection, the system provides user feedback through a seven-segment display, indicating system states such cooling conditions, or shutdown events. Serial communication is used for monitoring and debugging purposes, allowing real-time observation of system behavior.

Overall, the implemented system demonstrates how functional safety principles, redundant sensing, and deterministic decision-making can be combined to create a robust and reliable battery temperature management solution suitable for safety-critical environments.



System Design and Development Environment:

1-Initial Hardware Platform: Single MicroBlaze with Temperature Sensing.

This section describes the system design process, technical challenges, and the solutions adopted during the hardware development phase using Vivado. The objective of this stage was to create a reliable hardware platform capable of supporting temperature sensing, communication, and later redundancy mechanisms.

The initial step focused on implementing a single MicroBlaze processor integrated with a temperature sensor and UART communication. Establishing a stable UART interface was essential, as it provided a reliable debugging and monitoring channel through the serial terminal. Once UART communication was successfully validated, attention was directed toward enabling temperature sensing using the Pmod TMP3 sensor, which communicates via the I²C protocol.

In Vivado, the natural approach for enabling I²C communication is to use the AXI IIC IP core, which provides a standardized hardware-based I²C controller. While this solution was successfully integrated at the hardware level, significant challenges emerged during software development in Vitis. Specifically, driver-related issues and compilation errors prevented reliable access to the AXI IIC peripheral from the software layer. These issues made it impossible to build and run stable firmware using the AXI IIC interface within the given development constraints.

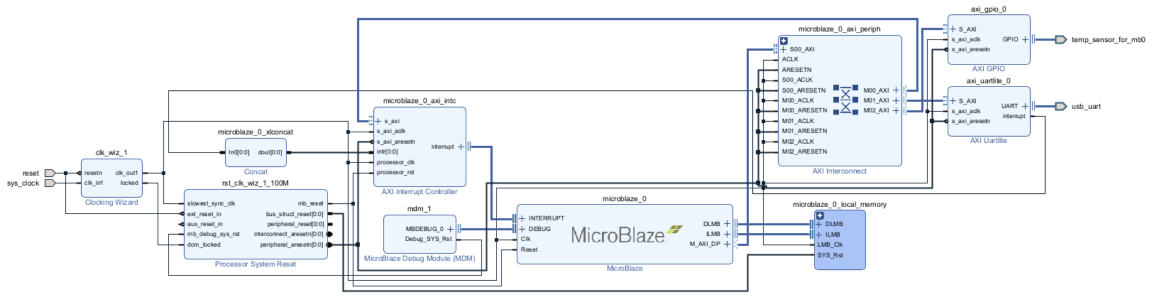
To overcome this limitation, an alternative solution was adopted: implementing a bit-banged I²C interface using AXI GPIO. Instead of relying on a dedicated I²C controller, two GPIO pins were configured dynamically in software to emulate the I²C signals (SCL and SDA). The GPIO direction was controlled at runtime, allowing each pin to switch between input (high-impedance) and output states as required by the I²C protocol. This approach does not explicitly define the pins as fixed inputs or outputs, enabling correct open-drain behavior.

This technique, commonly referred to as bit-banged I²C, proved to be fully compatible with Vitis and allowed successful compilation and execution of the firmware. Using this method, the temperature values from the Pmod TMP3 sensor could be read correctly and transmitted to the serial monitor via UART. This solution established a stable and reliable first processing node, which later became MicroBlaze 0 (MB0) in the Dual Modular Redundancy architecture.

For clocking, the system clock (sys_clock) provided by the FPGA board was used to ensure consistent timing across all components. Additionally, the system reset was configured as active-low, ensuring deterministic startup behavior and compatibility with the board-level reset circuitry.

The temperature sensor for this initial node was physically connected to the JA Pmod port, requiring the creation of a dedicated XDC constraint file. This constraint file defined the pin mappings and I/O standards for the GPIO-based I²C signals, ensuring correct electrical behavior and reliable communication with the sensor hardware.

By resolving the I²C communication challenge through GPIO-based bit banging and validating the UART output, a robust foundation was established for extending the system toward a dual-processor redundant architecture.



2- Extension to a Dual MicroBlaze System.

After successfully implementing and validating the first processing node, the system was extended to a dual MicroBlaze architecture in order to support redundancy and fault detection. The objective of this extension was to implement two independent processing nodes operating on the same FPGA device while maintaining deterministic behavior and reliable comparison of temperature measurements.

The redundancy is implemented using two virtual processors, both instantiated as MicroBlaze soft-core processors. Although both processors execute independently, they are deployed on a single physical FPGA, the Nexys A7-100T. Each MicroBlaze represents a complete embedded system with its own execution context and peripheral set, enabling independent operation while sharing the same hardware platform.

To preserve independence between the two processing nodes, each MicroBlaze was assigned its own local Block RAM (BRAM). This local memory architecture ensures that program execution and data storage remain isolated, preventing unintended interference between the processors. Such isolation is essential in redundant systems, as it limits fault propagation and preserves the integrity of comparison results.

Similarly, each processor was equipped with its own AXI Interrupt Controller (AXI INTC). By handling interrupts locally, each MicroBlaze maintains full control over its peripheral interactions and execution flow. This separation reinforces modularity and ensures that timing or interrupt-related faults in one node do not directly disrupt the operation of the other.

To enhance observability and debugging capabilities, a dedicated UART interface was connected to each MicroBlaze. This decision allows each processor to communicate independently with a serial terminal, making it possible to monitor internal behavior, verify execution correctness, and confirm that both nodes are operating concurrently and independently during runtime.

While replicating the MicroBlaze subsystems was conceptually straightforward, a critical challenge arose in the area of clocking and synchronization. An initial design approach connected each MicroBlaze to its own Clock Wizard IP. Although valid for standalone systems, this configuration introduced multiple unsynchronized clock domains within the same FPGA. As a result, conflicting timing constraints were generated, leading to timing violations and preventing successful bitstream generation.

In real-time and safety-relevant systems, redundancy is only meaningful when all components operate under a common time base. Independent clocks undermine determinism and complicate system-level comparison and decision-making. For this reason, the design was revised to employ a single global clock source, generated by one Clock Wizard and distributed to both MicroBlaze processors and their associated peripherals.

This unified clocking strategy eliminated clock domain conflicts, ensured deterministic timing behavior, and enabled reliable synchronization between the redundant nodes. With this foundation in place, the dual MicroBlaze system could operate cohesively on a single FPGA, providing a robust platform for inter-processor communication, temperature comparison, and fault handling mechanisms implemented in later stages of the design.

3-Comparator Design and Redundancy Strategy.

After connecting both MicroBlaze processors to the same clock source and ensuring system-wide synchronization, MicroBlaze 1 (MB1) was extended with UART and GPIO peripherals to support bit-banged I²C communication. The same procedure previously validated for MicroBlaze 0 (MB0) was reused, allowing MB1 to independently acquire temperature data from its own Pmod TMP3 sensor. With both processing nodes operating correctly and concurrently, the system was ready to implement redundancy verification.

At this stage, the core requirement of the Dual Modular Redundancy (DMR) architecture became the comparison of temperature measurements obtained from the two independent nodes. Redundancy alone is not sufficient; a mechanism is required to evaluate the consistency of the measured data and detect faults or abnormal behavior.

To implement this comparison functionality, two possible design approaches were considered:

- Hardware comparator.
- Software comparator.

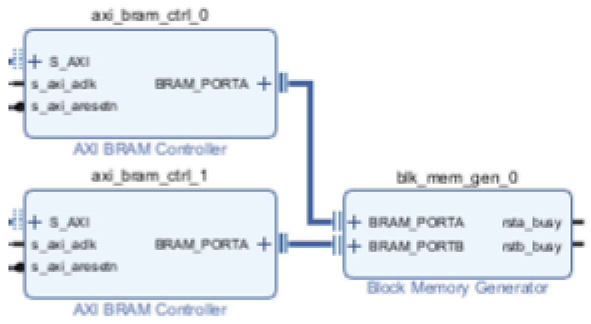
In this design, MB1 operates as a background monitoring node, continuously measuring temperature and writing its results to a shared memory region. MB0 acts as the primary decision-making node, responsible for reading both its own temperature measurement and the value produced by MB1, performing the comparison, and determining the system response.

This asymmetric software-based comparison strategy was selected for several reasons:

- It simplifies the hardware design by avoiding additional custom logic.
- It allows dynamic modification of thresholds and safety policies through software.
- It reflects real-world safety architectures where one controller acts as the system supervisor.
-
- It aligns naturally with the DMR concept, where disagreement detection is the primary goal.

Using shared AXI BRAM as a mailbox, MB1 periodically updates its temperature value and sets a validity flag, while MB0 reads these values, computes the difference, and evaluates it against predefined safety thresholds. Only MB0 is permitted to take action based on the comparison results, such as triggering cooling indications or initiating a shutdown state. This ensures that control authority is centralized, preventing conflicting outputs and maintaining deterministic system behavior.

By adopting a software-based comparator with asymmetric responsibility, the system achieves a practical and robust redundancy verification mechanism while maintaining flexibility and simplicity. This approach establishes the foundation for subsequent fault-handling logic and safe-state enforcement within the overall functional safety framework.



This approach was achieved by using a shared memory architecture.

To enable reliable communication and data exchange between the two MicroBlaze processors, a shared memory architecture was introduced using Block RAM (BRAM). This shared memory acts as a deterministic and low-latency communication medium, allowing both processing nodes to exchange temperature values and status information without relying on complex communication protocols.

The implementation consists of a single Block Memory Generator connected to two independent AXI BRAM Controllers, one for each MicroBlaze processor. Each AXI BRAM Controller provides an AXI slave interface that allows its corresponding MicroBlaze to access the same physical BRAM memory space. This configuration enables true shared-memory operation while maintaining clean separation between the two processor subsystems.

From a system perspective:

- MicroBlaze 0 writes its locally measured temperature values into the shared BRAM and reads the temperature values produced by MicroBlaze 1.
- MicroBlaze 1 continuously writes its measured temperature values into the shared BRAM and asserts a validity flag once data becomes available.
- Both processors access the same memory locations, ensuring deterministic data visibility and minimal latency.

This approach avoids the need for UART, SPI, or other serialized communication methods between processors, which would introduce unnecessary latency and synchronization complexity. Shared BRAM provides cycle-accurate access and is well-suited for safety-critical comparison logic.

4-Software Development Environment (Vitis)

This section describes the software architecture, design decisions, and implementation details developed using the Xilinx Vitis environment. The software layer is responsible for temperature acquisition, inter-processor communication, redundancy management, decision-making logic, and system state indication. Emphasis is placed on reliability, determinism, and functional safety rather than raw performance.

4.1- Full code breakdown for MB0 code.

4.1.1- GPIO Address Abstraction and Register-Level Control for Bit-Banged I²C

To ensure software robustness and maintain compatibility across hardware platform regenerations, the GPIO base address associated with the Pmod TMP3 temperature sensor is resolved dynamically at compile time rather than being hard-coded. In Xilinx-based systems, all hardware peripherals instantiated in Vivado are described within the xparameters.h file, where base addresses may appear under either user-defined instance names or canonical identifiers generated automatically by the toolchain.

To accommodate both naming conventions, conditional compilation directives are used to select the correct GPIO base address during the build process. If neither expected definition is present, compilation is intentionally halted with an explicit error message. This behavior ensures that the required GPIO peripheral has been correctly exported in the hardware platform and prevents software from silently binding to an invalid or unintended memory region.

```
#ifdef XPAR_AXI_GPIO_MB0_TMPSENSOR3_BASEADDR
  #define TMP3_GPIO_BASE   XPAR_AXI_GPIO_MB0_TMPSENSOR3_BASEADDR
#elif defined(XPAR_XGPIO_1_BASEADDR)
  #define TMP3_GPIO_BASE   XPAR_XGPIO_1_BASEADDR
#else
  #error "Cannot find TMP3 GPIO base address in xparameters.h"
#endif
```

This approach guarantees deterministic binding between software and hardware and prevents silent failures caused by mismatched or missing peripheral definitions, which is particularly important in safety-critical embedded systems.

4.1.2- Memory-Mapped GPIO Register Definition

Once the GPIO base address is resolved, direct memory-mapped access is used to control the GPIO peripheral. Two registers are of primary importance:

- DATA Register: Represents the logical state of the GPIO pins. When configured as outputs, this register determines the driven output value. When configured as inputs, it reflects the sampled logic level present on the pins.
- TRI (Tri-State) Register: Controls the direction of each GPIO pin individually. A TRI bit value of 0 configures the corresponding pin as an output, while a value of 1 places the pin in a high-impedance input state.

```
#define TMP3_DATA (TMP3_GPIO_BASE + 0x0)
#define TMP3_TRI  (TMP3_GPIO_BASE + 0x4)
```

This register-level separation between data and direction enables fine-grained runtime control of GPIO behavior, which is essential for protocol emulation.

4.1.3-Open-Drain Emulation and I²C Signal Mapping

The I²C protocol mandates open-drain signaling, where devices actively drive lines low but never drive them high. Logic-high levels are achieved by releasing the line and relying on external pull-up resistors. This electrical behavior was emulated entirely in software by dynamically modifying the GPIO direction using the TRI register.

Two GPIO bits were assigned to represent the I²C clock and data lines:

```
#define SCL_MASK 0x1
#define SDA_MASK 0x2
#define TEMP_I2C_ADDR 0x48
```

By setting the corresponding TRI bit to 0, the software drives the line low. By setting the TRI bit to 1, the line is released into a high-impedance state, allowing the external pull-up to assert a logic-high level. This mechanism faithfully reproduces the open-drain behavior required by the I²C specification.

All I²C primitives including start condition generation, stop condition generation, byte transmission, acknowledgment detection, and byte reception were implemented using this mechanism, with timing constraints enforced through calibrated software delay loops.

```
static inline uint32_t tmp3_tri_read(void)
{
    return Xil_In32(TMP3_TRI);
}

static inline void tmp3_tri_write(uint32_t v)
{
    Xil_Out32(TMP3_TRI, v);
}

static inline uint32_t tmp3_data_read(void)
{
    return Xil_In32(TMP3_DATA);
}

static inline void tmp3_data_write(uint32_t v)
{
    Xil_Out32(TMP3_DATA, v);
}
```

The functions above provide direct, low-level access to the memory-mapped GPIO registers associated with the TMP3 I²C interface. The TRI register dynamically controls the direction of each GPIO pin, allowing it to operate either as a driven output or as a high-impedance input. The DATA register reflects the logical voltage level present on the physical line when the pin is released, or drives the line low when output mode is selected.

Access to these registers is performed using Xil_In32() and Xil_Out32(), which guarantee deterministic 32-bit memory-mapped I/O operations on the MicroBlaze architecture. This mechanism is essential for accurately emulating the open-drain signaling required by the I²C protocol.

```c
static void set_dir(uint32_t mask, int input)
{
    uint32_t tri = tmp3_tri_read();

    if (input)
        tri |= mask;
    else
        tri &= ~mask;

    tmp3_tri_write(tri);
}
```

The set_dir() function dynamically controls the electrical ownership of one or more GPIO lines. The mask parameter specifies which GPIO bit(s) are affected, while the input parameter determines the direction of operation.

When input = 1, the corresponding TRI bit is set, placing the GPIO pin in a high-impedance state. In this configuration, the FPGA releases control of the line, allowing an external device specifically the TMP3 temperature sensor in this system to drive the signal.

When input = 0, the TRI bit is cleared, enabling the FPGA to actively drive the line low.

This behavior is essential for emulating open-drain signaling, where devices are permitted to assert a logic-low level but must never actively drive a logic-high level. Logic-high states are instead achieved by releasing the line and relying on external pull-up resistors, in accordance with the I²C protocol specification.

Up to this point, the code solely performs GPIO configuration and open-drain setup. It defines the memory-mapped register addresses associated with the AXI GPIO peripheral and configures pin direction control via the TRI register to selectively drive the I²C lines low or release them into a high-impedance state. Releasing the SDA line merely permits it to be pulled high by the external pull-up resistor or driven low by the sensor; no sensor data is read at this stage. Sensor communication begins only after the software explicitly generates valid I²C timing, including START conditions and SCL clock pulses, and samples the SDA line through reads of the DATA register

```c
static void scl_write(int level)
{
    set_dir(SCL_MASK, level ? 1 : 0);
}

static void sda_write(int level)
{
    set_dir(SDA_MASK, level ? 1 : 0);
}
```

The scl_write() and sda_write() functions implement a semantic abstraction layer that maps logical I²C signal levels to GPIO direction control. A logical high (level = 1) is realized by configuring the corresponding GPIO pin as an input, thereby releasing the line into a high-impedance state and allowing the external pull-up resistor to assert a logic-high voltage. A logical low (level = 0) is realized by configuring the GPIO pin as an output, which actively pulls the line low. This abstraction allows higher-level I²C routines to operate in terms of logical clock and data states without direct manipulation of the underlying GPIO TRI register, while preserving correct open-drain electrical behavior.

<u>4.1.4- I²C Signaling and Data Transfer Routines</u>

```c
static void i2c_init(void)
{
    tmp3_data_write(0x0);
    set_dir(SCL_MASK | SDA_MASK, 1);
}

static void i2c_start(void)
{
    sda_write(1);
    scl_write(1);
    i2c_delay();

    sda_write(0);
    i2c_delay();

    scl_write(0);
}

static void i2c_stop(void)
{
    sda_write(0);
    i2c_delay();

    scl_write(1);
    i2c_delay();

    sda_write(1);
    i2c_delay();
}
```

i2c_init():

This routine establishes a deterministic initial bus configuration suitable for open-drain I²C emulation. The DATA register is set such that any subsequent output-driving action forces a logic-low level. Both SCL and SDA are then released by configuring the corresponding GPIO direction bits as inputs (high-impedance), allowing external pull-up resistors to assert the idle-high bus state. No I²C transaction occurs at this stage; the function only prepares the electrical conditions required for correct signaling.

i2c_start():

This routine generates a compliant I²C START condition. The bus is first brought to the idle state with both SDA and SCL high (lines released). The START condition is defined by an SDA transition from high to low while SCL remains high, which is the protocol-defined indication that a transaction is beginning. After the SDA falling edge, SCL is driven low to transition into the clocked data phase of the transaction.

i2c_stop():

This routine generates a compliant I²C STOP condition to terminate an active transaction and return the bus to idle. SDA is held low while SCL is released high, then SDA is released high while SCL remains high. The SDA rising edge while SCL is high constitutes the STOP condition per the I²C specification, signaling bus release and end-of-message to all devices on the bus.

```c
static int i2c_write_byte(uint8_t data)
{
    for (int i = 7; i >= 0; --i) {
        sda_write((data >> i) & 1);
        i2c_delay();

        scl_write(1);
        i2c_delay();

        scl_write(0);
    }

    sda_write(1);
    i2c_delay();

    scl_write(1);
    i2c_delay();

    int ack = !sda_read();

    scl_write(0);
    return ack;
}
```

i2c_write_byte():

This routine transmits an 8-bit value on the I²C bus in MSB-first order. For each bit, SDA is set to the required logical state via open-drain behavior (drive low for ‘0’, release for ‘1’), followed by an SCL high pulse during which the slave samples the data bit. After all eight bits are transmitted, the master releases SDA and generates a ninth clock pulse to sample the slave acknowledgment. The slave indicates ACK by pulling SDA low during this clock; the routine samples this state via sda_read() and returns the acknowledgment result to the caller, enabling error detection and transaction control.

```c
static uint8_t i2c_read_byte(int send_ack)
{
    uint8_t data = 0;
    sda_write(1);

    for (int i = 7; i >= 0; --i) {
        scl_write(1);
        i2c_delay();

        if (sda_read())
            data |= (1 << i);

        scl_write(0);
        i2c_delay();
    }

    sda_write(send_ack ? 0 : 1);
    i2c_delay();

    scl_write(1);
    i2c_delay();

    scl_write(0);
    sda_write(1);

    return data;
}
```

i2c_read_byte():

This routine performs byte reception over the I²C bus by placing the SDA line in a released (high-impedance) state and allowing the slave device to drive the data bits. The master generates eight SCL clock pulses, and during each clock high phase the SDA line is sampled to reconstruct the received byte in MSB-first order. After all eight bits have been captured, the master transmits a protocol-defined response bit: an ACK by actively pulling SDA low if additional data is requested, or a NACK by leaving SDA released if the transfer is complete. This acknowledgment is clocked by an additional SCL pulse. The function then releases SDA and returns the assembled data byte, completing the read operation in compliance with I²C timing and open-drain signaling requirements.

It is important to note that I²C communication is established between the MicroBlaze processor and the external temperature sensor; the AXI GPIO peripheral does not participate in the I²C protocol itself, but merely provides the physical SDA and SCL interfaces through which I²C signaling is generated and sampled

<u>**4.1.5-Temperature Read and Conversion**</u>

```
static int tcn75_read_raw(int16_t *raw_out)
{
    uint8_t addr = TEMP_I2C_ADDR;
    uint8_t msb, lsb;

    i2c_init();

    i2c_start();
    if (!i2c_write_byte((addr << 1) | 0)) { i2c_stop(); return -1; }
    if (!i2c_write_byte(0x00))             { i2c_stop(); return -2; }
    i2c_stop();

    i2c_start();
    if (!i2c_write_byte((addr << 1) | 1)) { i2c_stop(); return -3; }

    msb = i2c_read_byte(1);
    lsb = i2c_read_byte(0);
    i2c_stop();

    *raw_out = (int16_t)((msb << 8) | lsb);
    return 0;
}
```

```
static int32_t tcn75_raw_to_centi(int16_t raw)
{
    int32_t code = (int32_t)raw;
    code >>= 4;
    return (code * 25) / 4;
}

static void tcn75_raw_to_temp_c(int16_t raw, int *degC, int *centi_abs)
{
    int32_t t = tcn75_raw_to_centi(raw);
    *degC = (int)(t / 100);
    int c = (int)(t % 100);
    if (c < 0) c = -c;
    *centi_abs = c;
}
```

tcn75_read_raw(): I²C Transaction Sequencing and Raw Data Acquisition

The tcn75_read_raw() routine implements a complete two-phase I²C register read from a TCN75/TMP75-class temperature sensor. The function first initializes the bus to a known open-drain idle state via i2c_init(). It then performs a write transaction to set the sensor's internal register pointer, followed by a read transaction to retrieve the selected register contents.

In the first transaction, the function generates a START condition and transmits the 7-bit slave address combined with a write direction bit ((addr << 1) | 0). Successful transmission is verified via the slave ACK; failure results in an early STOP and a negative error code. The function then transmits 0x00, selecting the temperature register (register pointer), again validating ACK and terminating on failure. A STOP condition completes the pointer-write phase, ensuring the sensor's internal address pointer is latched.

In the second transaction, the function generates a repeated START (implemented here as STOP followed by START) and transmits the slave address combined with a read direction bit ((addr << 1) | 1). Upon acknowledgment, the sensor becomes the bus driver for SDA and transmits the register contents. The master receives two bytes: the most-significant byte is read first and acknowledged (i2c_read_byte(1)) to request continuation; the least-significant byte is then read and not acknowledged (i2c_read_byte(0)) to indicate end of reception. A STOP condition terminates the read phase. The two bytes are combined into a signed 16-bit raw measurement using (msb << 8) | lsb, preserving the sensor's fixed-point encoding and sign representation.

tcn75_raw_to_centi(): Fixed-Point Interpretation and Scaling

The tcn75_raw_to_centi() routine converts the sensor's raw 16-bit register format into centi-degrees Celsius (0.01 °C units). The raw value is first promoted to a 32-bit signed integer to prevent overflow during arithmetic. The right shift by four bits (code >>= 4) discards the least significant fractional/control bits and aligns the fixed-point temperature field to an integer code. The conversion (code * 25) / 4 applies a scaling factor of 6.25 centi-degrees per code step, consistent with a 1/16 °C resolution (0.0625 °C) representation. The use of integer arithmetic ensures deterministic execution and avoids floating-point dependencies.

tcn75_raw_to_temp_c(): Human-Readable Formatting of the Converted Result

The tcn75_raw_to_temp_c() routine formats the centi-degree value into a whole-degree component and a two-digit fractional component suitable for display or logging. The converted temperature t is split into integer degrees via division by 100 and a fractional remainder via modulo 100. Because the remainder can be negative for negative temperatures, the fractional component is converted to an absolute value to ensure consistent formatting (e.g., −5.25 °C is represented as degC = −5, centi_abs = 25). This separation supports fixed-point output without floating-point formatting while maintaining correct sign handling.

For MB1 the same code is written for the initialization and establishing communication with the Temp sensor. The only difference is the main loop as MB1 write the temp value in the BRAM to be read by MB0 and do the difference computation.

4.1.6 System Control Logic and Temperature Comparison

```
int main(void)
{
    Xil_Out32(SEVSEG_TRI, 0x00000000);
    seg_write(SEG_BLANK);

    const uint8_t MSG_COOL[4] = { SEG_C, SEG_O, SEG_O, SEG_L };
    const uint8_t MSG_STOP[4] = { SEG_S, SEG_T, SEG_O, SEG_P };

    while (1) {

        int16_t raw;
        int err = tcn75_read_raw(&raw);
        if (err != 0) {
            xil_printf("MB0: I2C error: %d\r\n", err);
            seg_write(SEG_BLANK);
            delay_cycles(5000000);
            continue;
        }

        int deg0, cent0;
        tcn75_raw_to_temp_c(raw, &deg0, &cent0);

        int32_t t0_centi = tcn75_raw_to_centi(raw);
        *temp_mb0 = t0_centi;

        if (*mb1_valid == 0) {
            xil_printf("MB0: MB0=%d.%02d C (waiting for MB1)\r\n", deg0, cent0);
            seg_write(SEG_BLANK);
            delay_cycles(8000000);
            continue;
        }
        int32_t t1_centi = *temp_mb1;

        int32_t diff = t0_centi - t1_centi;
        if (diff < 0) diff = -diff;

        int32_t deg1  = t1_centi / 100;
        int32_t cent1 = abs(t1_centi % 100);

        if (diff >= DIFF_FAULT_CENTI) {
            *status = 1;
            xil_printf("SHUTDOWN\r\n");
            while (1) {
                seg_show_sequence(MSG_STOP, 4);
            }
        }

        if ((t0_centi > COOLING_TEMP_CENTI) && (t1_centi > COOLING_TEMP_CENTI)) {
            *status = 0;
            xil_printf("MB0: COOL  MB0=%d.%02d C, MB1=%ld.%02ld C\r\n",
                       deg0, cent0, (long)deg1, (long)cent1);
            seg_show_sequence(MSG_COOL, 4);
        } else {
            *status = 0;
            xil_printf("MB0: OK    MB0=%d.%02d C, MB1=%ld.%02ld C, |Δ|=%ld.%02ld C\r\n",
                       deg0, cent0, (long)deg1, (long)cent1,
                       (long)(diff/100), (long)abs(diff%100));
            seg_write(SEG_BLANK);
            delay_cycles(8000000);
        }
    }
}
```

The main() function implements the top-level control loop executed by MicroBlaze 0 (MB0). This routine integrates temperature acquisition, inter-processor data exchange via shared memory, system-state evaluation, and user feedback through console output and a seven-segment display. Unlike previous sections, which focus on low-level I²C signaling and sensor interfacing, this function operates at the application logic level.

At initialization, the seven-segment display GPIO is configured as an output by writing to its TRI register, and the display is cleared. Two fixed character sequences corresponding to the “COOL” and “STOP” states are defined for later use. No I²C communication occurs during this initialization phase.

The application then enters an infinite loop, within which MB0 continuously monitors the local temperature sensor and coordinates with MB1. At the beginning of each iteration, MB0 invokes tcn75_read_raw() to perform a complete I²C transaction and retrieve the raw temperature value from the sensor. If an error is detected during this transaction, the error condition is reported via the UART, the display is cleared, and execution is delayed before retrying. This ensures robustness against transient I²C failures.

Upon successful acquisition, the raw sensor value is converted into both a human-readable representation (integer degrees and fractional component) and a fixed-point centi-degree representation. The centi-degree value is written into shared BRAM (*temp_mb0), making MB0’s measurement available to MB1. At this stage, MB0 acts as a data producer in the inter-processor communication scheme.

Before performing any comparison, MB0 checks the shared validity flag (*mb1_valid) to determine whether MB1 has published a valid temperature measurement. If MB1 is not yet ready, MB0 reports its own temperature, blanks the display, delays execution, and continues looping. This prevents undefined behavior caused by stale or uninitialized shared data.

Once MB1’s data is marked valid, MB0 reads MB1’s temperature value from shared memory and computes the absolute temperature difference between the two processors’ measurements. This comparison is performed entirely in fixed-point centi-degree units to maintain determinism and avoid floating-point arithmetic.

Two threshold-based decisions govern system behavior. First, if the absolute temperature difference exceeds a predefined fault threshold (DIFF_FAULT_CENTI), MB0 asserts a shared status flag and enters a terminal shutdown state. In this state, the system continuously displays the “STOP” message and no further processing occurs, enforcing a latched fault response. Second, if both temperatures exceed a defined cooling threshold (COOLING_TEMP_CENTI) without violating the difference constraint, the system enters a cooling state, indicated by the “COOL” display message.

If neither condition is met, the system remains in a normal operating state. MB0 reports both temperature readings and their absolute difference via UART output, clears the display, and delays execution before repeating the loop. Throughout this process, MB0 serves as the coordinating processor, combining sensor input, shared-memory data from MB1, and application-level decision logic to determine system state and user feedback.