Hardware/Software Co-design with movement control

1st Linas Vidziunas
(222001)
Faculty of Technology, Natural Sciences and Maritime Sciences
University of South-Eastern Norway
Kongsberg, Norway
linasvidz@gmail.com

2nd Sindre Kristoffersen Olsen
(231461)
Faculty of Technology, Natural Sciences and Maritime Sciences
University of South-Eastern Norway
Kongsberg, Norway
sinols2000@hotmail.no

3rd Jesper Sylte Drønnen
(267076)
Faculty of Technology, Natural Sciences and Maritime Sciences
University of South-Eastern Norway
Kongsberg, Norway
jesper.sylte.dronnen@gmail.com

***Abstract*— This project involves a comparative report of four design methodologies in hardware/software co-design for autonomous navigation in a car platform. The approaches include a hardware-only solution, an application-specific instruction-set processor solution, a co-design solution using high-level synthesis, and a purely software-based approach. The primary focus is on assessing and comparing the design complexity, cost, performance, and flexibility of each method. This analysis aims to highlight the trade-offs and benefits inherent in different approaches to embedded system design, particularly for a task requiring autonomous movement and obstacle detection.**

***Index Terms*— Hardware/Software Co-Design, Comparative Analysis, Design Complexity, Performance Evaluation, Cost Efficiency.**

I. Introduction

This report delves into a comprehensive project centered on hardware/software co-design in embedded systems, with a specific application to an autonomous car platform equipped with four DC motors and an ultrasonic transducer. The project's primary aim is to explore and compare four distinct design methodologies: a hardware-only approach using a finite state machine with datapath (FSMD), an Application Sepcific Instruction-set Processor (ASIP) solution, a High-Level Synthesis (HLS)-based co-design solution, and a software-centric approach. This comparative report tries to understand the synergies and trade-offs between hardware and software components in autonomous navigation and obstacle detection.

Embedded systems, characterized by their seamless integration of software and hardware, play a pivotal role in a wide range of applications, from consumer electronics to automotive systems. The core contribution of this report is its systematic analysis of different design strategies, offering insights into design complexity, cost efficiency, performance, and flexibility. The outcomes of this project are expected to provide valuable guidance for selecting appropriate design methodologies in developing efficient, cost-effective, and high-performing embedded systems.

To effectively explore and compare these solutions, the group was assigned the task of designing movement control for predefined behaviors in a car platform. The specified requirement entailed that the car should autonomously drive forward until an obstacle is detected at a predetermined distance. Upon obstacle detection, the car is programmed to reverse a certain distance, execute a 90-degree turn to the left, and then continue driving forward. This assignment requirement is intricately woven into the project, ensuring a practical and hands-on approach to evaluating the effectiveness of each design methodology in real-world scenarios.

II. Theoretical Framework

A. Car platform

The car platform comes equipped with two motor driver modules, a daughter board, and four motors, one for each one of the wheels.

The wheels are a type of specialized wheels, known as mecanum wheels which set the car apart from conventional vehicles. Unlike traditional wheels, mecanum wheels are omnidirectional, allowing the vehicle to move in any direction without requiring a turning axle, as typically seen in consumer cars. This remarkable omnidirectionality is achieved through the ingenious use of external rollers attached to the entire circumference of the wheel's rim. These rollers are positioned at a 45-degree angle to the plane of the wheel, enabling the car to glide in all directions. [1]

Each of these wheels is connected to a separate motor, enabling independent control and facilitating the unique movement capabilities offered by macanum wheels.

B. Ultrasonic transducer (HC-SR04)

The ultrasonic distance sensor is an economical device that offers a measurement range of 2cm to 400cm with an accuracy of up to 3mm. [2] The HC-SR04 module comprises an ultrasonic transmitter, a receiver, and an embedded control circuit.

The module features four pins: VCC (Power), Trig (Trigger), Echo (Receive), and GND (Ground). Additionally, it incorporates control circuitry designed to mitigate inconsistencies in data, particularly in scenarios where "bouncy" reading might occur.

The fundamental principle to calculate the distance to an object involves a simple three-step process:

1) **Trigger signal**: Initiate trigger signal with a high-level duration of at least 10 microseconds.
2) **Ultrasonic Pulse Emission**: The module emits eight 40kHz ultrasonic pulses and listens for the return of a pulse signal.
3) **High-Level Time Measurement**: Upon receiving a signal, the module records the duration of the high-level output. This duration corresponds to the time taken for the ultrasonic pulse to travel to the object and back. [2]

Distance can be calculated using the following formulas, each at an increasing level of detail:

$$\text{Distance} = \text{High Level Time} \times \frac{\text{Velocity of Sound}}{\text{Number of Ultrasonic Trips}} \quad (1)$$

$$\text{Distance} = \frac{\text{No. ticks}}{\text{Clock Freq.}} \times \frac{\text{Velocity of Sound}}{\text{Number of Ultrasonic Trips}} \quad (2)$$

$$\text{Distance} = \frac{\text{No. ticks}}{100MHz} \times \frac{340m/s}{2} \quad (3)$$

In practice, we assume the velocity of sound to be 340 m/s, and the ultrasonic trips to be two; one to the object and another for the signal's reflection back to the ultrasonic receiver. It's worth noting that, as per the documentation, using a measurement cycle of over 60 milliseconds is advisable. This prevents the trigger signal from being misinterpreted as the echo signal, ensuring reliable distance measurements. [2]

C. Motor driver (L298N)

The L298N motor driver is a versatile component that offers control over the speed and rotation direction of up to two Direct Current (DC) motors.

Controlling the speed of a DC motor with the L298N is accomplished by adjusting the input voltage. A common technique for this purpose is Pulse Width Modulation (PWM). PWM involves sending a series of ON and OFF pulses to the motor, where the average voltage is directly proportional to the width of these pulses, known as the duty cycle. By varying the duty cycle, the speed of the motor can be precisely controlled.

To change the direction of rotation of a DC motor, the L298N utilizes an H-bridge configuration. An H-bridge consists of four switches arranged in the shape of an "H", with the motor placed at the center. By selectively closing specific pairs of switches, the polarity of the voltage applied to the motor can be reversed, effectively changing its direction of rotation. The L298N motor driver is equipped with two integrated H-bridges, one for each motor, simplifying the control process.

The L298N motor driver features a set of essential pins, each serving a specific function:

- **Power Pins**
  - **VS** - This pin powers the integrated circuit's H-bridges, responsible for driving the motors. Acceptable voltage inputs range from 5 to 12V.
  - **VSS** - The pin powers the logic circuitry within the integrated circuit.
  - **GND** - This is the common ground connection.
- **Output Pins**
  - **OUT1 and OUT2** - These pins are inputs for motor A, with OUT1 serving as the positive connection.
  - **OUT3 and OUT4** - Similarly, these pins are inputs for motor B, with OUT3 as the positive connection.
- **Direction Control Pins**
  - **IN1 and IN2** - These pins control the direction of rotation for motor A.
  - **IN3 and IN4** - Similarly, these pins control the direction of rotation for motor B.
- **Speed Control Pins**
  - **ENA** - Used to turn motor A on or off, with the added capability of speed control through PWM.
  - **ENB** - Similarly, this pin turns motor B on or off and provides speed control through PWM.

[3]–[5]

III. Hardware-Only Solution

A. Implementation

The Hardware-Only solution is implemented as a Finite State Machine with Datapath using the VHSIC Hardware Description Language (VHDL) and the Vivado software suite. A FSMD combines a finite state machine (FSM) responsible for controlling the sequence of operations with a datapath responsible for the actual processing of data. The datapath in our FSMD takes input from the ultrasonic transducer via the PWM Interface (PWI) module and determines whether we are too close to an obstacle or not. The FSMD and algorithmic state machine with datapath (ASMD) of the PWI is modified from the provided examples given with the assignment to include the required trigger signal, and can be seen in appendix B. Based on this signal, the control path will issue the correct sequence of control signals to an output decoder to produce correct output signals for the motors. The control flow can be seen from the ASMD diagram in Figure 2. In this diagram, we have four states that can be roughly summarized as such:

- **S0**: Initial state
- **S1**: Driving forward
- **S2**: Driving in reverse
- **S3**: Turning 90° to the left

We begin in an initial state S0 where the controll_signal is 000 making the car stand still. On the next clock cycle we move to state S1 where the control signal is 001, and the car drives forward. If the ultrasonic sensor detects an obstacle closer than a predefined threshold, the under_the_limit signal triggers a transition of the FSM to state S2. In state S2 the control signal is 010 and the car drives in reverse. It will continue to do so until it is sufficiently far away from the obstacle. When the car has finished reversing, it transitions to state S3, where the control signal is set to 011, prompting the car to turn left. It will turn left until it has turned for a sufficient amount of time to have turned 90°, where it will transition back to state S1. For the purpose of turning for a set period and for the PWI, we employed a timer module, the details of which are outlined in Appendix C. The FSMD diagram in Figure 1 illustrates how the control and data paths are interconnected at the top-level.

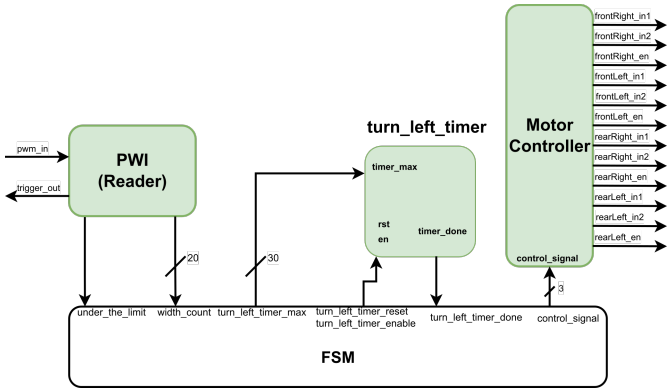

Fig. 1. FSMD chart for the Hardware-Only solution

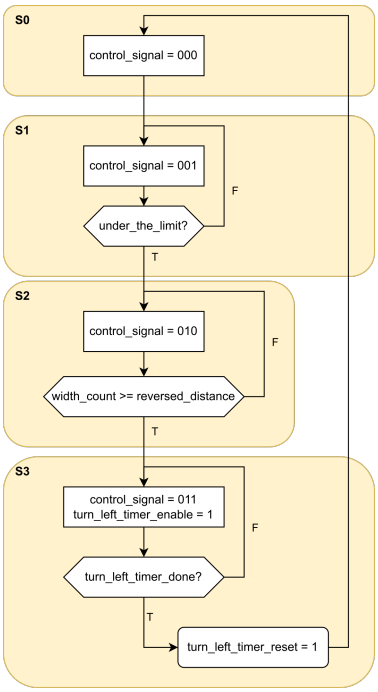

Fig. 2. ASMD chart for the Hardware-Only solution

B. Simulation

We were successful in implementing and simulating the solution. The main part of the testbench used for the simulation tries to mimic the functionality required to handle the sensor input and motor output. It begins with a reset followed by repeating the sequence of

- Wait $10\mu s$ for trigger pulse to finish
- Set pwm_in high for a duration
- Set pwm_in low

By setting the pwm_in high for durations of $600\mu s$, $400\mu s$, and then $700\mu s$ we simulate the circuit initially detecting no obstacle, detecting an obstacle, reversing until the obstacle is sufficiently far away, turning 90° to the left and then continue to drive forward unless an obstacle is detected again. From the resulting waveform diagram in Figure 3 we see the transition from state S0 to S1 on the first rising edge of the clock after the initial reset. When entering S1 the pwm_reader module sets the trigger_out signal to high, and it will stay so for $10\mu s$. The control_signal that dictates the motor output is changed to 001 and the motor outputs are changed according to the predefined actions of the decoder. From Figure 4 we can see

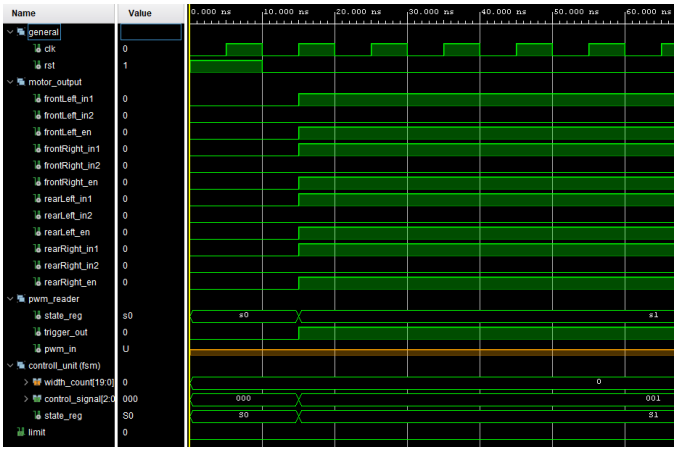

Fig. 3. Wave diagram for S0 to S1 transition

the entire simulation. We can see that we stay in state S1 until the limit is high (detected an obstacle) and we move to S2. We stay in S2 until the distance to the obstacle has become large enough and transition to S3. In state S3 we turn left for a given amount of time (1ms in the simulation). We move back to S1 after S3 and stay in S1 since limit is low.

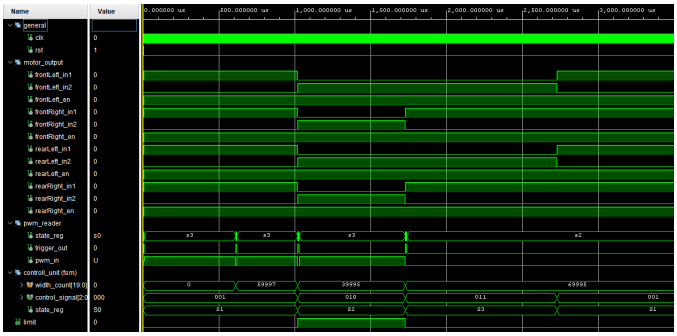

Fig. 4. Wave diagram for the whole simulation

This shows that the implemented functionality corresponds to the functional requirements given in the assignment.

IV. High-Level Synthesis Solution

A. Design methodology

For our decision-making between developing functionality using HLS or directly in the application code, we opted to implement the ultrasonic component in HLS, given its predictive and timing-specific nature. The rest of maneuvering logic was implemented in the application code. This approach demonstrates interactions between HLS and software.

B. Implementation

The processing of the ultrasonic transducer is implemented as an HLS component. This high-level approach allows us to manage the sensor's functionality while abstracting away the lower-level details. The maneuvering logic, on the other hand, is embedded within the application code that runs on the field programmable gate array (FPGA) platform and interacts with the ultrasonic transducer component.

The ultrasonic transducer component in HLS follows a structured FSM design, cycling through three states:

- Trigger: Initiates the ultrasonic sensor's trigger mechanism by applying a high pulse for $10\mu s$.
- Listen: Monitors the echo response and measures its duration.
- Delay: Implements a delay before initiating the next measurement.

The design integrates three essential blocks: two general-purpose input/output (GPIO) blocks, one for the ultrasonic transducer and another for individual control of the motors. As well as the HLS implemented ultrasonic block as depicted in Figure 5.

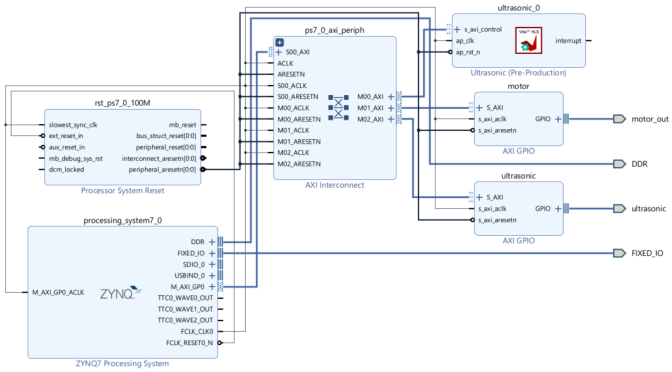

Fig. 5. Circuit design.

The maneuvering logic within the application layer follows three simple states.

- Forward: Driving the vehicle forward until the distance to an object is below the 10cm threshold.
- Reverse: Driving in reverse until the vehicle is sufficiently far away from the object.
- Turn: Initiates a turning maneuver to the left for a predefined duration.

Its important to note that the HLS solution was not fully finished. While the HLS component simulates correctly, as detailed in the subsequent subsection described later, it is not a guarantee that it will correctly interact with the rest of the system. Nonetheless, in Section IX, we still include resource and power consumption data for this solution, anticipating it to be representative of a fully functional system. Additionally, we discuss the challenges associated with HLS solution in Section X.

C. Simulation

The testbench for the HLS component is designed to continuously invoke the component, such that we observe the component through all its states. The state transitions are indicated by the dashed lines in Figure 6. The component begins in the first state, setting the trigger high. The second transition occurs after the $10\mu s$, after fully initializing the trigger mechanism of the ultrasonic transducer. Here, the component has stopped triggering and is monitoring for an echo signal. On the following state transition, the simulation responds with a predefined echo for 58824 clock cycles, corresponding to a 10cm distance measurement. The final state transition happens when the echo goes from high to low, updating the distance measurement based on the echo duration.

```
TEST 58824
   Trigger = 1
   Echo = 0
   Distance = 0.000000mm
   ---
   Trigger = 0
   Echo = 0
   Distance = 0.000000mm
   ---
   Trigger = 0
   Echo = 1
   Distance = 0.000000mm
   ---
   Trigger = 0
   Echo = 0
   Distance = 100.000801mm
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
```

Fig. 6. HLS simulation of the ultrasonic sensor.

V. Software Solution

In the software-only implementation for the car platform project, the control logic is entirely managed through a C program, developed in the Vitis Integrated Development Environment (IDE). This program interprets data from a conceptual ultrasonic sensor to navigate around obstacles, replacing the FSMD's function in hardware solutions. It processes sensor inputs to make real-time decisions, controlling the car's movement, including forward drive, reverse, and 90-degree turns. The software directly commands the DC motors via driver interfaces. This method, focused on flexibility and adaptability, is executed on the Zybo board, providing a distinct approach to movement control compared to hardware-centric solutions.

VI. Implementation

The software for our simulated car control project is implemented in Vitis using the C programming language, built on top of a platform created in Vivado which utilizes the Zynq Processing System and a GPIO block. The implementation revolves around a structured sequence of operations to simulate the actions of a car, such as moving forward, detecting obstacles, reversing, and turning.

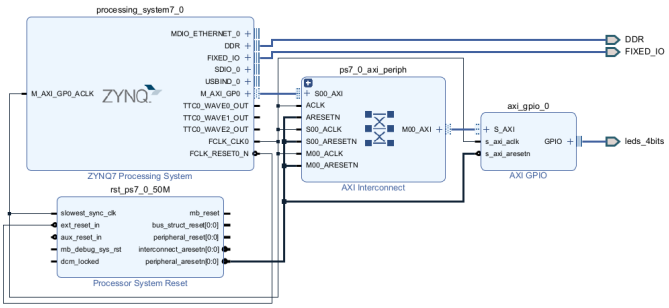

Fig. 7. Circuit design.

At the core of the software is the `main` function. This function is responsible for initializing the GPIO interface, which is crucial for controlling Light Emitting Diode (LED)s. These LEDs are used to visually represent the car's actions in our simulation. Once the initialization is successful, the `main` function invokes `RunTestSequence`, which activates the car's operational loop.

`RunTestSequence` implements a state machine with four states to control the car's actions. The sequence starts with the car in a 'moving forward' state, simulated by the `MoveForward` function. This function engages the LEDs to indicate forward movement. The next state in the sequence is 'object detection'. In this phase, the `DetectObject` function simulates detecting an object. For the purpose of this proof of concept, object detection is not based on sensor input but is instead simulated. The function is designed to mimic detecting an object at regular intervals - every tenth call, specifically - to provide a consistent pattern for testing the sequence of states.

Upon 'detecting' an object, the state machine transitions the car into a 'reversing' state, handled by the `Reverse` function. This function, like `MoveForward`, operates the LEDs to simulate the car moving backward. The final state in the sequence is 'turn left', managed by the `TurnLeft` function. This function completes the loop by simulating a left turn, again using the LEDs for visual representation.

Each of these functions - `MoveForward`, `DetectObject`, `Reverse`, and `TurnLeft` - plays a specific role in the sequence, manipulating the GPIO to control the LEDs. This control mimics the car's actions, creating a visual narrative of the car's movements.

VII. Simulation

The simulation was conducted using onboard LEDs to visually represent the car's simulated movements on the Zybo board. This approach enabled an immediate understanding of the software's control logic effectiveness. Steady, flashing, and sequential LED patterns were used to indicate forward movement, reversing, and turning left, respectively. These patterns provided clear feedback on the state transitions and actions of the simulated car.

The software demonstrated prompt responsiveness, with LEDs reacting instantly to state changes, indicating rapid processing of simulated sensor inputs. Throughout multiple cycles, the consistent behavior of these visual indicators underscored the software's reliability. While based on predefined patterns rather than real sensor data, the simulation effectively showcased the software's potential in real-world navigation scenarios, setting a strong foundation for future enhancements involving physical sensors and actuators.

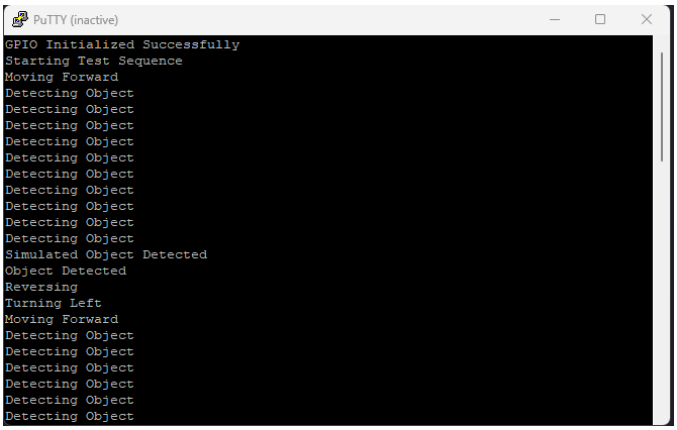

Fig. 8. Software simulation of control logic

VIII. Application Specific Instruction-set Processor Solution

A. Design methodology

Our ASIP solution is strategically engineered for efficient management of movement control and obstacle detection tasks. This involves a fine-tuned balance between hardware and software integration, pivotal for system performance and adaptability. We embed high-speed, low-latency tasks such as real-time ultrasonic echo processing in hardware, capitalizing on their predictable nature. In contrast, complex and flexible functions like movement control are managed by software, allowing for ease of updates without hardware changes.

B. Implementation architecture design

After considering various interaction methods between software and hardware, we initially adopted the use of operation codes for direct interaction with hardware components. This initial approach however, led to a blurring of the typical instruction categorization, which traditionally falls into three distinct groups: data transfer, control flow, and arithmetic/logical operations [6, p. 13–18]. This approach is detailed in Appendix A. We eventually transitioned to a mapped memory interface, utilizing the pre-existing data memory register. This refined approach adheres to the traditional instruction categorization by utilizing a dedicated memory region, which can be accessed and modified by the standard store and load operations available in the base ASIP.

The ultrasonic transducer module in our ASIP is designed as a distinct FSMD. In this context, a 'module' refers to a group of hardware components collectively performing a specific function. This module's data path includes several mod-m counters, a register, and combinational logic. The module utilizes four counters, each with a specific function to ensure accurate distance measurement. The counters are described in the order as follows in Figure 9, oriented in the correct direction, from left to right.

1) **trigger_delay_ctr**: Manages the intervals between consecutive measurement cycles.
2) **trigger_pulse_ctr**: Ensures the trigger signal is maintained high for the necessary duration, ensuring the ultrasonic pulse is emitted correctly.
3) **echo_delay_ctr**: Introduces a delay between the trigger signal and the subsequent listening for an echo. This counter acts as a threshold preventing false readings from 'bouncy' echo signals.
4) **echo_ctr**: Measures the duration the echo signal remains high, which correlates to the distance from an object.

Upon the echo signal's transition form high to low, the echo duration is transferred to a register. The combinational logic block, at the end of the data-path in Figure 9, then interprets this duration (measured in ticks), converting it to a distance measurement expressed in millimeters.

The states of the ultrasonic transducer, as depicted in Figure 10, are outlined below:

- **S0**: Trigger signal initiation.

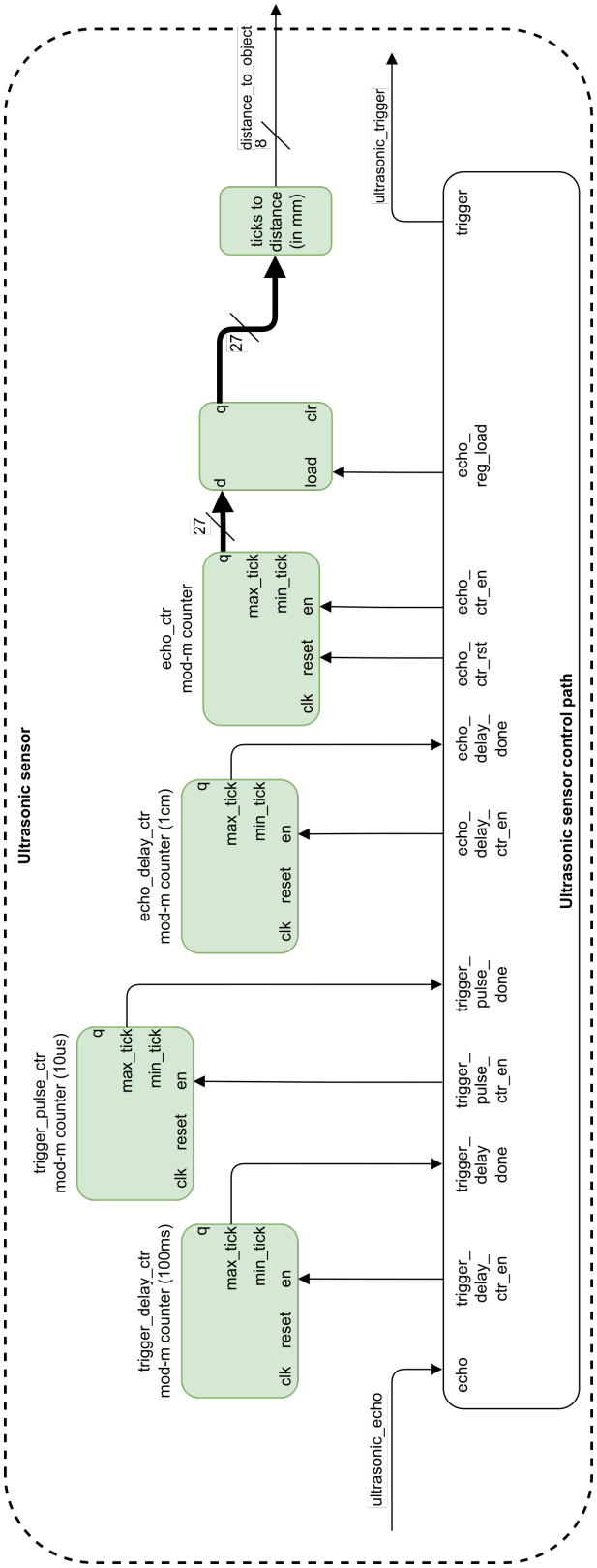

Fig. 9. FSMD diagram for the ultrasonic transducer module implemented in the ASIP solution.

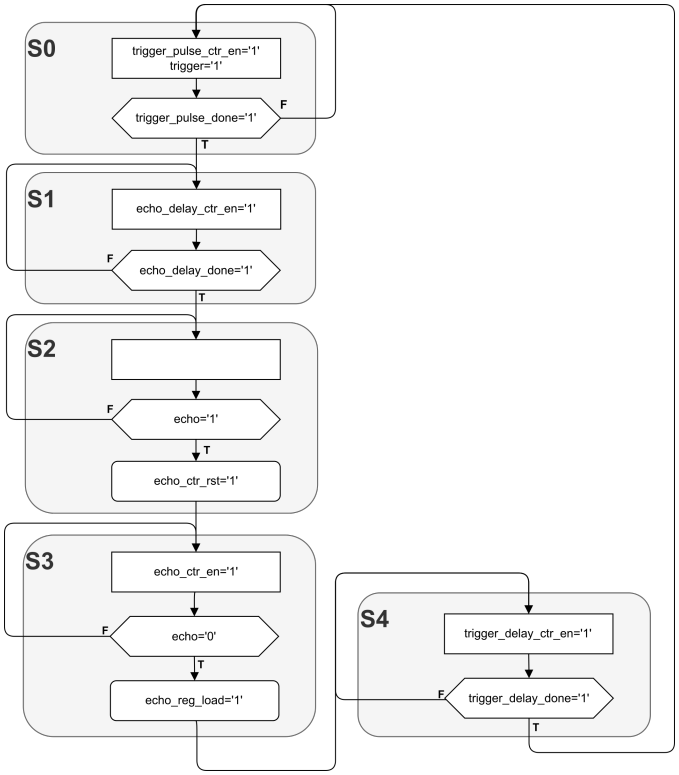

Fig. 10. ASMD diagram for the ultrasonic transducer module implemented in the ASIP solution.

- **S1**: Delay to filter short-range echoes.
- **S2**: Echo detection and counter resetting.
- **S3**: Counting duration of high echo signal.
- **S4**: Delay before the next measurement cycle.

In developing our motor control strategy, we evaluated two methods: one using a shared memory value distributed across all motors, and another using four separate addresses, one for each motor. We selected the latter approach, assigning separate addresses to each motor. Although this method requires four clock cycles to configure all the motors, it simplifies control by avoiding shared values and allows for a wider range of PWM settings for each motor.

The key component for motor control with PWM is a counter operating at a 100Hz frequency. Values from the mapped memory addresses, corresponding to each motor are used to determine the motor's rotation direction and speed. The rotation direction is determined by the most significant bit (MSB) of the value. The magnitude of these values, converted to unsigned integers using two's complement for negative values, sets the PWM duty cycle. This duty cycle, when compared against the 100Hz counter, decides whether the motor is on or off.

Another key component of the ASIP solution is the hardware counter, used for precise left-turning maneuvers. This counter's design enables it to output time in increments of 10 milliseconds. This specific resolution choice, dictated by our use of eight-bit data registers, allows for a maximum count of 2.56 seconds. Opting for a finer 1ms resolution would limit us to only 0.256 seconds, which falls short of our needs. While a software-based approach could serve as an accurate counter, we chose a hardware implementation for its simplicity and directness. A software solution would require multiple eight-bit registers and intricate operation tracing to ensure accurate time measurements.

The hardware counter module comprises two counters: a precision counter that increments up to 10ms before emitting a max tick signal, and a duration counter that increments with each max tick. This duration counter is directly connected to the data memory, from which its value can be loaded to the central processing unit (CPU) using a standard load instruction.

To simplify the CPU's decision-making, we introduced a conditional branching instruction. This instruction is a 'branch if greater or equals unsigned' (BGEU), and is specifically used to take actions based on the distance from an object compared to a threshold value.

For an exhaustive understanding of our ASIP implementation, complete with in-depth FSMD and ASMD diagrams, the complete set of available instructions, and the specific machine code employed for navigating the car through predefined maneuver patterns, refer to Appendix A.

C. Simulation

The simulation of our integrated ASIP is designed to closely mimic real-world scenarios, with a focus on accurately replicating the behavior of ultrasonic sensors. This simulation responds to trigger pulses from the unit under test by emitting echo pulses of pre-determined lengths. To simulate real-time travel of the pulses, the simulation initially holds a low signal echo signal for the predefined duration before switching to a high echo signal, mimicking the journey of sound waves to an object and back.

As depicted in Figure 12, the simulation results confirm correct state changes when the ultrasonic echo reception completes. The signal transitions illustrate how the echo duration counter's value is captured into a register. This value is then processed by a combinational block, converting it into the distance to the object in millimeters, as indicated by the distance_to_object signal.

In Figure 11, we see the simulation of our PWM regulated motor controller. The simulation demonstrates that the enable signal remains high for approximately 50% of the PWM period, confirming the correct functionality of the PWM implementation.

The simulation results from Figure 13 showcase the sequential execution of instructions within the CPU. The ram signals displayed indicate the current values stored in the registers. Specifically, register 1 holds the distance, while register 2 contains the distance threshold. This particular snapshot is captured at the point in the logic that decides whether to move forward or backward. The operation at program counter (PC) 7 is a conditional branch instruction, evaluating if the threshold in register 2 is greater or equal to the last measured distance

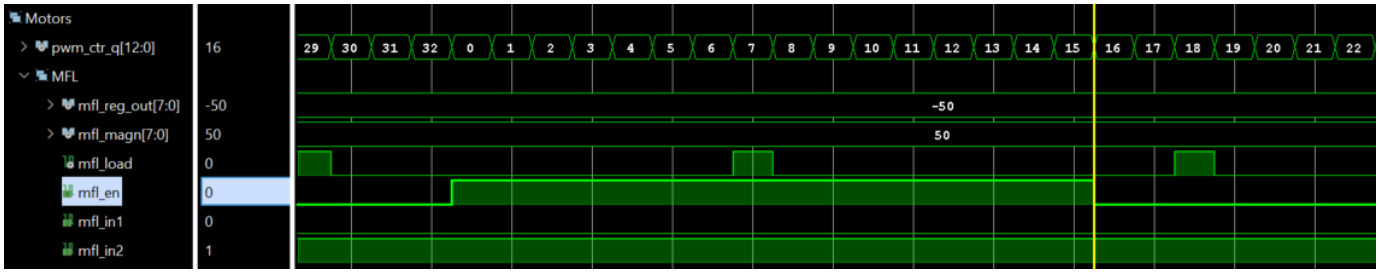

Fig. 11. Simulation results demonstrate PWM output with a 50% duty-cycle.

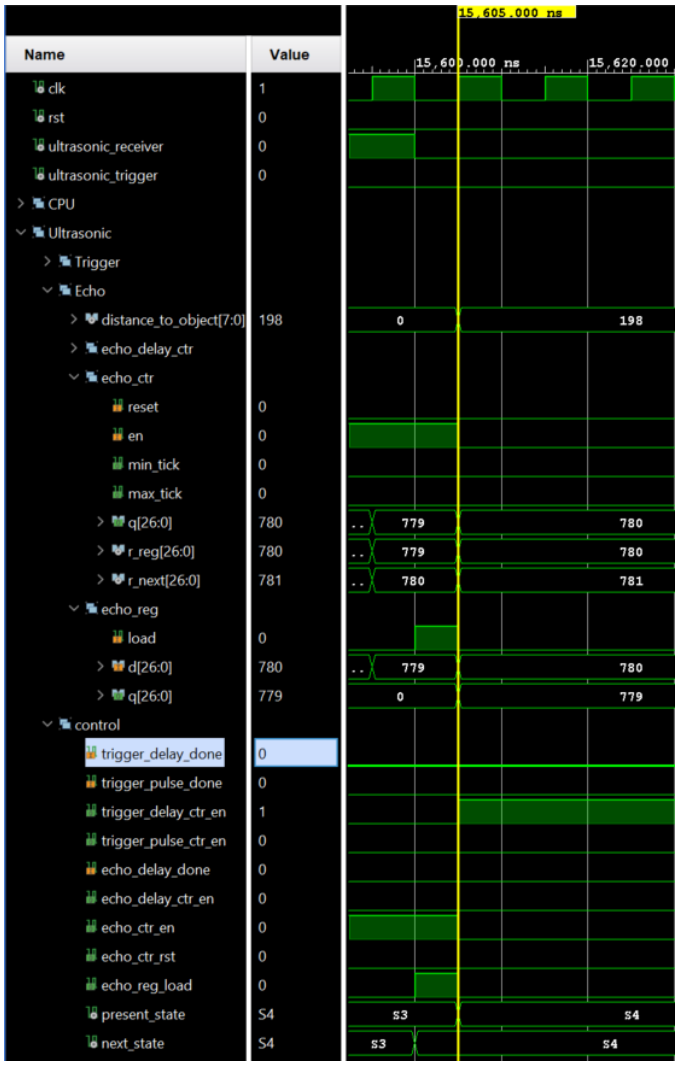

Fig. 12. The simulation demonstrates successful conversion of echo signal duration into distance measurement in millimeters.

in register 1. In this case, it is not, leading the system to progress to PC 8. Here, the operation loads a signed value of 100 into register 4. The next instruction at PC 9 bypasses the reverse logic, jumping to PC 14, where the value in register 4 is stored to the mapped memory address of the front left motor. Similarly, PC 15 and 16 stores the values of register 4 to the front right and rear right motors, respectively.

Fig. 13. Simulation results of CPU operations.

IX. Results

The FPGA resources required for the FSMD, ASIP and HLS solutions are presented in table I. This shows the number of units and the percent utilization where possible. During the project several changes has been made to both the FSMD and ASIP solution that resulted in this resource usage. Earlier implementation used more resources. This means that how you write your VHDL code can have a significant impact on the hardware it generates and thus the resources it requires. It's worth noting that, in contrast to FSMD and ASIP solutions, the maneuvering logic within the HLS solution was implemented as software, and hence does not impact the FPGA resource consumption. We also notice that the resource usage of the software only solution is less than that of the HLS, but more than that of the FSMD and ASIP solutions. This is due to the fact that the software-only solution only uses the integrated LEDs, while the HLS solution is implemented as an external peripheral. We assume that an effective HLS implementation would use less resources, as this can be more tailored towards specific needs.

TABLE I
FPGA resource utilization (Post implementation)

Resource	FSMD	ASIP	HLS	SW
I/O	17 (16%)	16 (16%)	14 (14%)	4 (4%)
LUT	137 (1%)	2744 (13%)	1784 (10%)	432 (2%)
FF	171 (1%)	2206 (5%)	1909 (5%)	575 (2%)
BUFG	1 (3%)	1 (3%)	1 (3%)	1 (3%)
Cells	3	58	3	5
Nets	42	676	150	146

For the above solutions, we also looked at the power consumption summaries, these can be seen in table II. From this we can see that the ASIP solution consumes more power than the FSMD solution and that the increase comes from the dynamic power load, meaning that the power consumption comes from extra i/o, logic, signals, and clocks. It was expected that the ASIP solution would use more power since it's a larger circuit capable of doing more than just what is required to control our car platform. Both the HLS and software-only solutions use more than 10 times more power than the FSMD and ASIP solutions. It is worth noting that the FSMD and ASIP solutions run on the Basys 3 board, while the software-only and HLS solutions used the Zybo board.

TABLE II
FPGA power load (PL)

Metric	FSMD	ASIP	HLS	SW
Total	87mW	123mW	1707mW	1687mW
Static	72mW	72mW	124mW	124mW
Dynamic	15mW	51mW	1583mW	1563mW

X. Discussion and Conclusion

After successfully implementing the three out of the four above solutions we found that there is no real difference in the perceived performance and functionality of the solutions from a user's perspective, but from a technical perspective, they differ enough that it matters which solution you choose. Whether you go for an all-hardware, all-software or a co-design solution depends on your needs. In general, we have found that going from software to hardware provides better efficiency and performance while sacrificing flexibility and adding complexity, as summarized by Patrick R. Schaumont in *A Practical Introduction to Hardware/Software Codesign* [7, p. 15–16]. The time to market (TTM) of our software solution is considerably shorter than our hardware-only solution, it is also somewhat easier to add new functionality. The co-design solution was meant to merge the performance benefits of dedicated hardware with the flexibility of software. It would allow for better performance than a software-only solution, with just a marginal increase in complexity. This means the TTM of the solution does not increase as drastically as for the other solutions. However, we faced challenges in integrating the ultrasonic component with the top-level application, particularly concerning input/output operations and interpreting the documentation. We observed a notable difference in VHDL implementation compared to HLS. In VHDL, crafting inefficient code proved more challenging. This is attributed to VHDL's requirement for specifying hardware and its functionalities at a significantly lower level of abstraction, whereas HLS emphasizes functionality, automatically generating the necessary hardware. This observation reinforces our conviction that while hardware-centric approaches are more efficient, they demand a greater investment of time.

The ASIP solution presents unique challenges. Compared to standard hardware solutions, it is resource-heavy, and exhibits a degree of inflexibility and complexity, while underperforming against typical consumer-grade CPUs. According to Dake Liu and Jian Wang, an ASIP solution is suitable when:

1) The assembly instruction set will not be used for general-purpose computing
2) The performance or power consumption requirements cannot be achieved by a CPU
3) The volume is high or the design cost is not a sensitive parameter
4) It is possible to use the ASIP for multiple products

[8, p. 671–673]. Based on these points, especially points 2) and 4), the task it handles may not be complex enough to justify its resource usage compared to alternatives. Additionally, its intricate design introduces several dimensions to design complexity, encompassing decisions about hardware versus software functionality, the core CPU design, and necessary operations.

However, it's not all drawbacks for the ASIP. True to its name, the ASIP's strength lies in its application-specific nature. Flexibility, in this context, is not absolute but relative. While it might appear less adaptable in a broad sense, the ASIP provides enhanced flexibility for specific modifications within its designed application. For instance, in an autonomous car platform, if a minor change like altering turn directions is needed, the ASIP can easily incorporate this adjustment,

more so than typical hardware solutions. It's important to note, though, that this flexibility dwindles when modifications venture beyond the ASIP's application-specific framework.

All solutions provide the same functionality, with varying degrees of flexibility, resource usage, and power consumption. The best solution considering the car platform is battery-powered would be the FSMD solution due to the low power consumption. With regards to flexibility the software only or HLS solution might be a better option, but the ASIP solution might also be viable considering only the application-specific domain of the car platform. Yet another thing to consider is the expertise within the organization or group, as using a software-only or HLS solution might give a shorter TTM if there is little VHDL competence. To conclude we believe that no best solution suits all needs, and rather that the solution must be chosen based on the requirements and available resources.

XI. Usage of ChatGPT

As a part of this year's edition of the course, we were given access to a ChatGPT Plus subscription, to test how it can be used for learning. To share our experiences using it for this project assignment we have included a section on ChatGPT usage.

A. Enhancing report writing with ChatGPT

ChatGPT has significantly accelerated our report writing. Both versions 3.5 and 4 have proven to be effective writing assistants, enhancing not just text creation but also the quality of our reports.

A major strength of ChatGPT in our work is its ability to refine text, improving flow and structural coherence. It provides valuable suggestions, making our reports more logical and reader-friendly, and ensuring ideas are conveyed smoothly.

Additionally, ChatGPT has enriched our language use, offering varied phrases and word choices that enhance our report with creativity and a broader vocabulary. This has made our reports more engaging and compelling.

In summary, ChatGPT was an essential tool in our report writing process. It enhances content creation speed, quality, coherence, and creativity. Through collaborative editing, ChatGPT helped us produce informative and engaging reports.

B. Limitations within code generation

While ChatGPT is quite effective at generating code for more conventional programming scenarios, its application in VHDL and HLS presents a mixed bag of results.

In VHDL, we often observed instances where ChatGPT's code modification led to unintended behaviors or suboptimal efficiency. Common issues included incomplete updates post-modifications and an inclination to overcomplicate designs, such as incorporating unnecessary states. Furthermore, it misplaced combinational logic within sequential blocks or vice versa, leading to non-compiling code.

In the context of HLS, ChatGPT showed a tendency to shift from basic C to C++ with continued interaction. While C++ offers more advanced features, this automatic transition does not align with the requirements for this project, since we write simpler C implementations.

Based on our experiences, we advise caution in relying on ChatGPT for extensive code generation, particularly for specialized FPGA and circuit design tasks. Its current capabilities seem to better align with providing explanations and clarifications on code-related queries. ChatGPT excels in breaking down complex concepts, offering clear, concise explanations, making it a valuable tool for understanding code, rather than generating it from scratch.

As such, we recommend leveraging ChatGPT as a supplementary resource for gaining insights and clarifications on specific coding challenges, rather than as a primary tool for code generation in FPGA and HLS contexts.

References

[1] "Wikipedia - mecanum wheel," (visited on 12/01/2023). [Online]. Available: https://en.wikipedia.org/wiki/Mecanumwheel

[2] "Ultrasonic ranging module hc-sr04," (visited on 12/01/2023). [Online]. Available: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf?_gl=1*1m9t5pf*_ga*MTE5ODQ4MjIyMC4xNzAwMzQzOTYy*_ga_T369JS7J9N*MTcwMTQzNTAzNi4zLjAuMTcwMTQzNTAzNi42MC4wLjA.

[3] "L298n dual motor controller module 2a datasheet," (visited on 12/01/2023). [Online]. Available: https://cdn.bodanius.com/media/1/9d61596_img.pdf

[4] "Interface l298n dc motor driver module with arduino," (visited on 12/01/2023). [Online]. Available: https://lastminuteengineers.com/l298n-dc-stepper-driver-arduino-tutorial/

[5] "Tutorial - l298n dual motor controller module 2a and arduino," (visited on 12/01/2023). [Online]. Available: https://cdn.bodanius.com/media/1/d6d1595_img.pdf

[6] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[7] P. R. Schaumont, *A practical introduction to hardware/software codesign*. Springer Science & Business Media, 2012.

[8] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, *Handbook of signal processing systems*. Springer, 2013.

APPENDIX A
EXTENDED ASIP SOLUTION DESCRIPTION

This appendix focuses on graphical representations of the modifications made to the base ASIP architecture. It expands on the alternate implementation referred to in Section VIII. While the reasons behind various design choices have been previously addressed, this appendix focuses on visual and descriptive elements of these implementations.

As previously mentioned, we have two slightly different implementations for ASIP the solution: the primary solution uses mapped memory addresses to control for interacting with the hardware components; the second uses distinct operation codes for interaction with hardware.

Both of the implementations share the same ultrasonic transducer and hardware counter modules. However, they differ in how the CPU interfaces with the hardware components. The PWM configuration also varies slightly; in the memory-mapped solution, the data memory doubles as a register to store instruction values, whereas the distinct operation code approach necessitates additional registers for storing values passed to the hardware components.

Figure 14 depicts the FSMD diagram for the mapped memory implementation, while Figure 18 and Figure 19 depicts the FSMD diagram for the implementation using distinct operation codes. Contrasting the memory-mapped implementation, this implementation controls the four PWM modules for precise motor control via specialized instructions. Similarly to the memory-mapped implementation, we had to decide whether to use several instructions, each for one motor, or one instruction that divides the immediate field across all of them. We opted for the individual operation codes for each motor, prioritizing simplicity at the cost of efficiency. In this setup, the ultrasonic sensor's distance readings and the timer's operations are also managed through distinct instructions. Aside from these differences, the instruction set is the same as in the mapped memory solution.

The ASMD diagram depicted in 16 shows the modifications made to the decode and execution control path for the implementation using distinct operation codes. Note that the "BGEU" instruction is also implemented in the mapped memory implementation.

Due to the differences between these two implementations, the machine code is also slightly different. Figure 15 shows the used machine code to execute the predefined maneuver patterns for the mapped memory address implementation, while 17 shows the machine code for the distinct instruction implementation.

Tables III and IV show a notable difference in resource consumption between the two implementations. Pinpointing pointing the exact cause requires further analysis. Intriguingly, the distinct operation code approach, which utilizes more registers due to not using the data memory block as a register, was expected to be more resource-intensive. Contrary to this expectation, its resource usage of certain resources is lower.

TABLE III
POST-IMPLEMENTATION RESOURCE CONSUMPTION COMPARISON FOR THE TWO IMPLEMENTATIONS OF THE ASIP.

Resource	Mapped memory impl.		Operation code impl.	
LUT	2744	(13.19%)	2118	(10.18%)
LUTRAM	12	(0.13%)	44	(0.46%)
FF	2206	(5.30%)	196	(0.47%)
DSP	5	(5.56%)	4	(4.44%)
I/O	16	(15.09%)	16	(15.09%)
BUFG	1	(3.13%)	1	(3.13%)
Cells	58		74	
Nets	676		729	

TABLE IV
POWER CONSUMPTION COMPARISON

Power type	Mapped memory impl.		Distinct operation impl.	
Dynamic	51mW	(42%)	44mW	(38%)
Static	72mW	(58%)	72mW	(62%)
Total	123mW		116mW	

A plausible explanation for this disparity may lie in the usage of data memory. In the distinct operation code implementation, the data memory is not used at all. It's conceivable that the synthesis process recognizes this and accordingly optimizes the data memory component out of the design. While in the mapped memory implementation, the optimizer includes the data memory block as it plays a crucial role in the system's functionality.

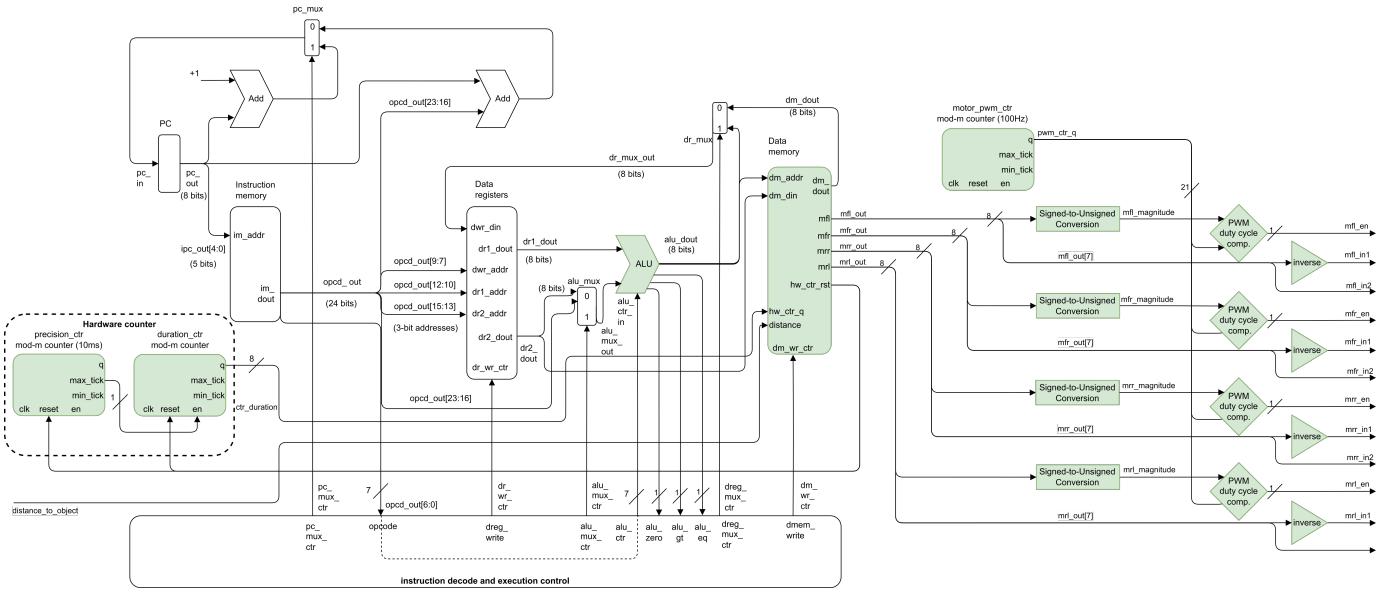

Fig. 14. FSMD diagram of the ASIP solution using mapped memory addresses, where the ultrasonic module's output, distance_to_object, is directly mapped to the data memory block.

```
constant instr_opcodes: rom_type:=(
    -- Init
    "11111000000000000000000", -- addr 00: MVi R0, 248              Move start of the application specific mapped memory address into R0
    "00000001000000001000000010", -- addr 01: LD R1, R0(1)          Load distance to object into R1
    "11111111000001000000001100", -- addr 02: JRZ R1, -1        `(dec)   Jump back if the distance measurement is 0

    -- Setup
    "01100100000000100000000",  -- addr 03: MVi  R2, 100        (dec)   Setting distance threshold into R2
    "00000000000000011000000",  -- addr 04: MVi  R3, 0          (dec)   Indicating previous driving state, forward

    -- Forward/bakward logic
    "00000001000000001000000010", -- addr 05: LD R1, R0(1)              Move distance measurement into R2
    "00000100000011000001101",  -- addr 06: JRNZ R3, 4          (dec)   Jump to backward if state is not zero
    "00000011010001000100000",  -- addr 07: BGEU R2, R1, 3      (dec)   Drive backwards if threshold is greater or equals to current distance (R1 <= R2)

    -- Forward
    "01100100000001000000000",  -- addr 08: MVi  R4, 100        (dec)   Move signed value into R3 (aka. motors to move forward full speed)
    "00000101000000000001110",  -- addr 09: J 5                 (dec)   Jump to Drive all

    -- Backward
    "00000001000000011000000",  -- addr 10: MVi  R3, 1          (dec)   Backing up status
    "10010110000000100000000",  -- addr 11: MVi  R2, 150        (dec)   Update the distance threshold, such that the car backs up 5cm
    "00000111001010000100000",  -- addr 12: BGEU R1, R2, 7      (dec)   Turn to the left if distance is greater or equals the new threshold (R2 <= R1)
    "11001110000001000000000",  -- addr 13: MVi  R4, -50        (dec)   Move -50 to R3 (aka. motors to move backwards at half speed)

    -- Drive all wheels
    "00000100100000000000011",  -- addr 14: ST  R4, R0(4)               Drive front left motor according to R4
    "00000101100000000000011",  -- addr 15: ST  R4, R0(5)               Drive front right motor according to R4
    "00000110100000000000011",  -- addr 16: ST  R4, R0(6)               Drive rear right motor according to R4
    "00000111100000000000011",  -- addr 17: ST  R4, R0(7)               Drive rear left motor according to R4
    "11110011000000000001110",  -- addr 18: J -13               (dec)   Jump back to MVD

    -- Turn left
    "11001110000000101000000",  -- addr 19: MVi  R5, -50        (dec)
    "00110010000000100000000",  -- addr 20: MVi  R4, 50         (dec)
    "00000100101000000000011",  -- addr 21: ST  R5, R0(4)               Drive front left motor according to R5
    "00000101100000000000011",  -- addr 21: ST  R4, R0(5)               Drive front right motor according to R4
    "00000110100000000000011",  -- addr 22: ST  R4, R0(6)               Drive rear right motor according to R4
    "00000111101000000000011",  -- addr 24: ST  R5, R0(7)               Drive rear left motor according to R5

    -- Wait for left turn to finish
    "00000011100000000000011",  -- addr 25: ST R4, R0(3)
    "11111110000000101000000",  -- addr 26: MVi  R5, 254        (dec)   Move left turn counter threshold into R5
    "00000010000000110000010",  -- addr 27: LD R6, R0(2)        (dec)   Load counter value into R6
    "11111111101110000100000",  -- addr 28: BGEU R5, R6, -1     (dec)   Wait until counter is greater than or equal to threshold in R5
    "11100110000000000001110",  -- addr 29: J -26               (dec)   Jump to Setup

    "11111111111111111111111",  -- addr 30: (void)
    "11111111111111111111111"   -- addr 31: (void)
);
```

Fig. 15. Machine code executing predefined maneuver patterns in the ASIP solution utilizing mapped memory addresses for hardware interaction.

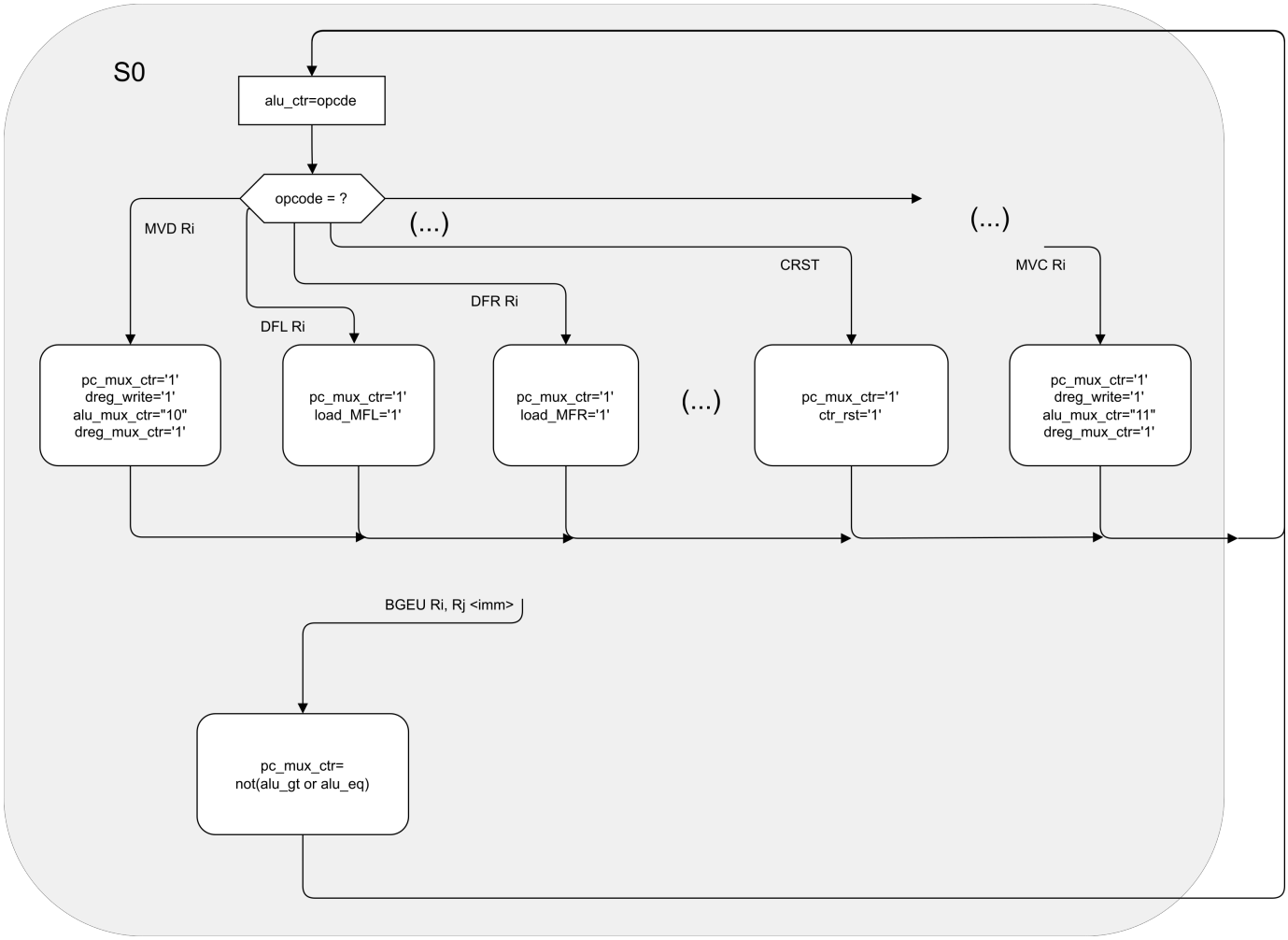

Fig. 16. ASMD diagram illustrating the additional operations added to the core ASIP's control path. Note that the "BGEU" operation is also present in the mapped memory implementation, however the others are not.

```
constant instr_opcodes: rom_type:=(
    -- Init
    "11111000000000000000000", -- addr 00: MVi R0, 248                Move start of the application specific mapped memory address into R0
    "00000001000000010000010", -- addr 01: LD R1, R0(1)               Load distance to object into R1
    "11111111000001000000100", -- addr 02: JRZ R1, -1          `(dec) Jump back if the distance measurement is 0

    -- Setup
    "01100100000000100000000",  -- addr 03: MVi  R2, 100       (dec)  Setting distance threshold into R2
    "00000000000000110000000",  -- addr 04: MVi  R3, 0         (dec)  Indicating previous driving state, forward

    -- Forward/bakward logic
    "00000001000000010000010",  -- addr 05: LD R1, R0(1)              Move distance measurement into R2
    "00000100000011000001101",  -- addr 06: JRNZ R3, 4         (dec)  Jump to backward if state is not zero
    "00000011010001000100000",  -- addr 07: BGEU R2, R1, 3     (dec)  Drive backwards if threshold is greater or equals to current distance (R1 <= R2)

    -- Forward
    "01100100000001000000000",  -- addr 08: MVi  R4, 100       (dec)  Move signed value into R3 (aka. motors to move forward full speed)
    "00000101000000000001110",  -- addr 09: J 5                (dec)  Jump to Drive all

    -- Backward
    "00000001000000110000000",  -- addr 10: MVi  R3, 1         (dec)  Backing up status
    "10010110000000100000000",  -- addr 11: MVi  R2, 150       (dec)  Update the distance threshold, such that the car backs up 5cm
    "00000111001010000100000",  -- addr 12: BGEU R1, R2, 7     (dec)  Turn to the left if distance is greater or equals the new threshold (R2 <= R1)
    "11001110000000100000000",  -- addr 13: MVi  R4, -50       (dec)  Move -50 to R3 (aka. motors to move backwards at half speed)

    -- Drive all wheels
    "00000100100000000000011",  -- addr 14: ST  R4, R0(4)             Drive front left motor according to R4
    "00000101100000000000011",  -- addr 15: ST  R4, R0(5)             Drive front right motor according to R4
    "00000110100000000000011",  -- addr 16: ST  R4, R0(6)             Drive rear right motor according to R4
    "00000111100000000000011",  -- addr 17: ST  R4, R0(7)             Drive rear left motor according to R4
    "11110011000000000001110",  -- addr 18: J -13              (dec)  Jump back to MVD

    -- Turn left
    "11001110000000101000000",  -- addr 19: MVi  R5, -50       (dec)
    "00110010000000100000000",  -- addr 20: MVi  R4, 50        (dec)
    "00000100101000000000011",  -- addr 21: ST  R5, R0(4)             Drive front left motor according to R5
    "00000101100000000000011",  -- addr 21: ST  R4, R0(5)             Drive front right motor according to R4
    "00000110100000000000011",  -- addr 22: ST  R4, R0(6)             Drive rear right motor according to R4
    "00000111101000000000011",  -- addr 24: ST  R5, R0(7)             Drive rear left motor according to R5

    -- Wait for left turn to finish
    "00000011100000000000011",  -- addr 25: ST R4, R0(3)
    "11111110000000101000000",  -- addr 26: MVi  R5, 254       (dec)  Move left turn counter threshold into R5
    "00000010000000110000010",  -- addr 27: LD R6, R0(2)       (dec)  Load counter value into R6
    "11111111101110000100000",  -- addr 28: BGEU R5, R6, -1    (dec)  Wait until counter is greater than or equal to threshold in R5
    "11100110000000000001110",  -- addr 29: J -26              (dec)  Jump to Setup

    "11111111111111111111111", -- addr 30: (void)
    "11111111111111111111111"  -- addr 31: (void)
);
```

Fig. 17. Machine code to execute the predefined maneuver patterns.

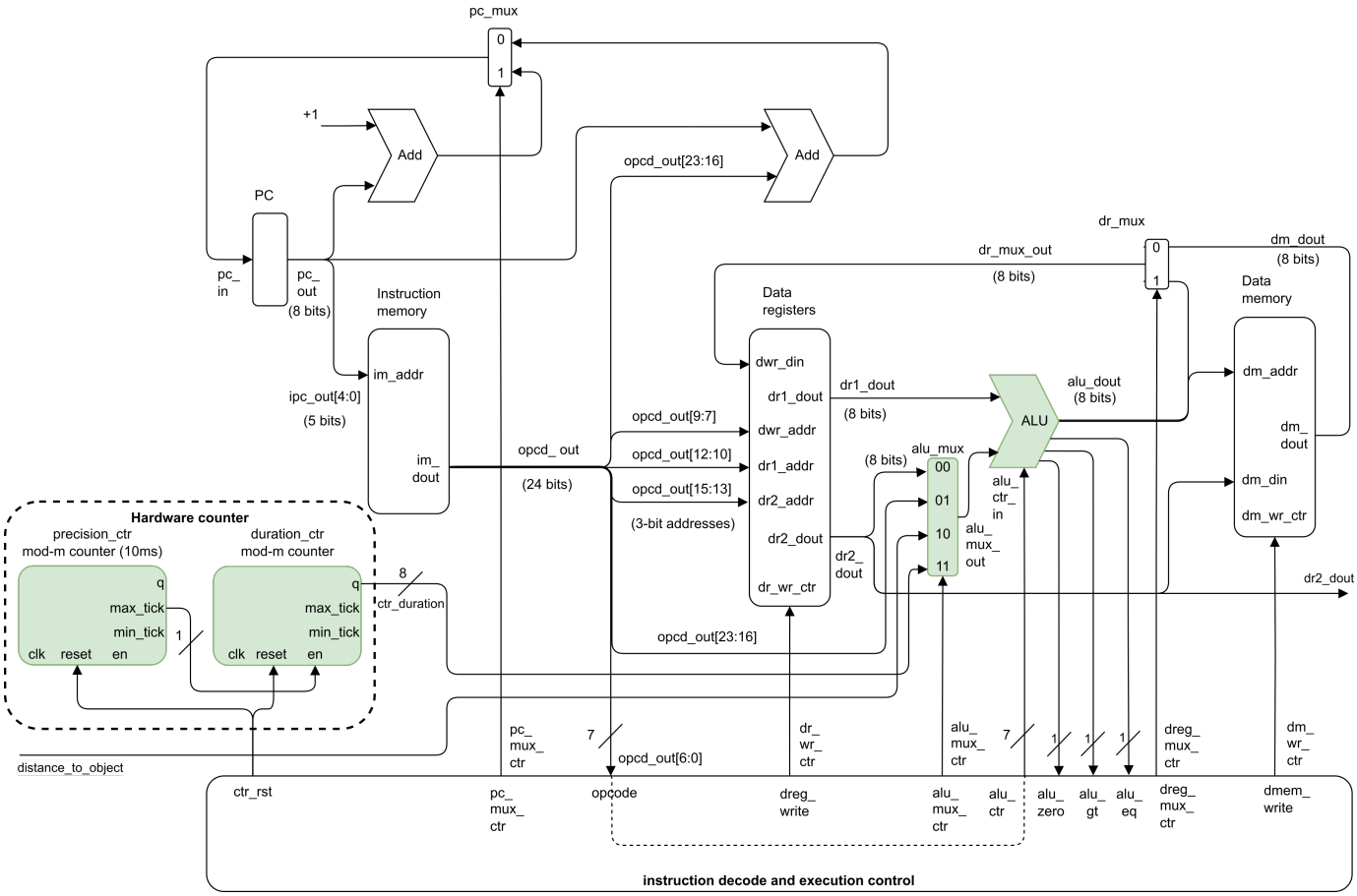

Fig. 18. FSMD diagram for the core CPU with the hardware counter.

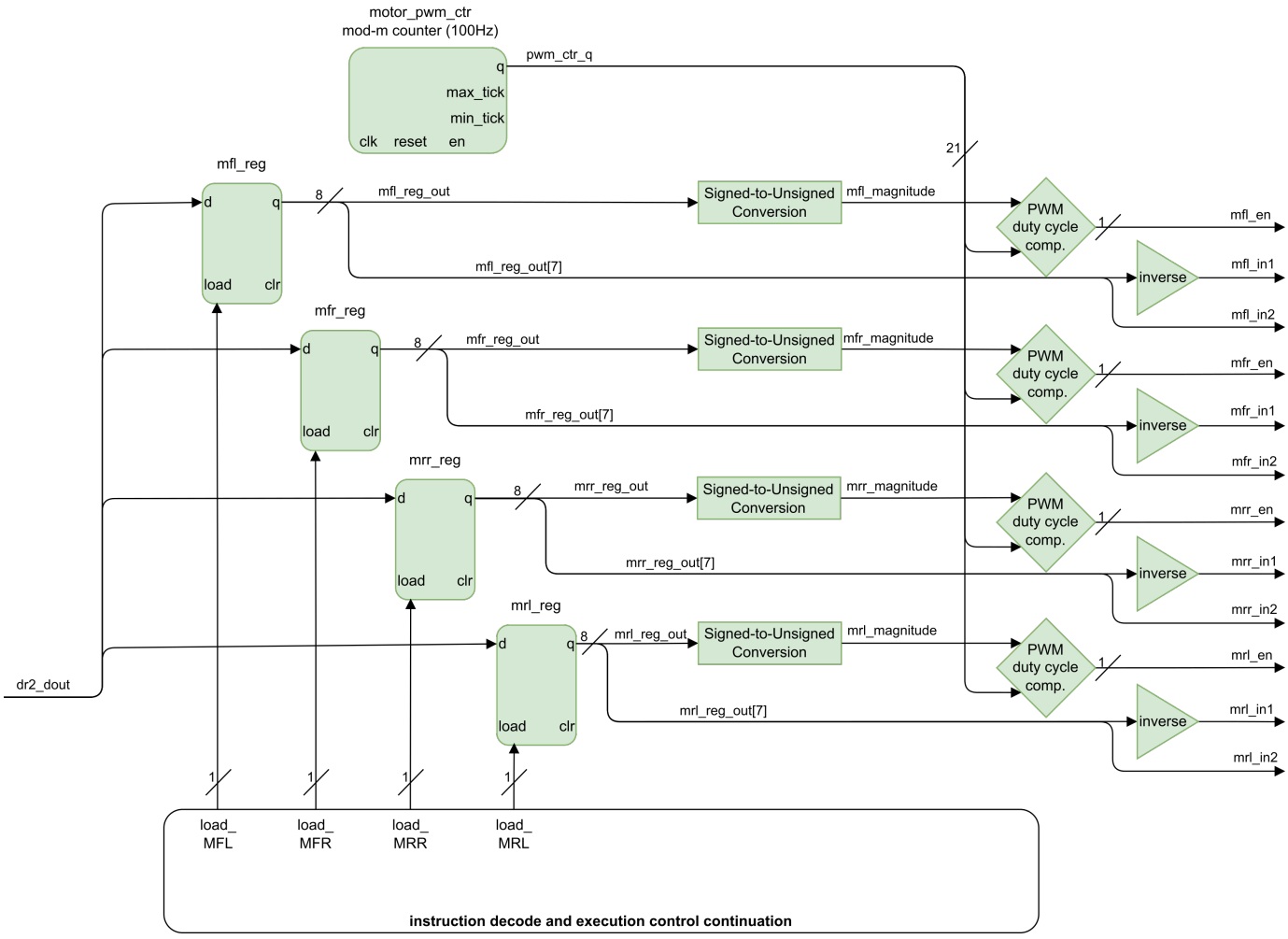

Fig. 19. FSMD diagram of the PWM controlled motors. Note that dr2_dout connects to the CPU's data register component. The outputs on the right side are directly connected to the motor controller on the car platform.

APPENDIX B
PWM READER WITH TRIGGER

PWM Reader with trigger (FSMD)

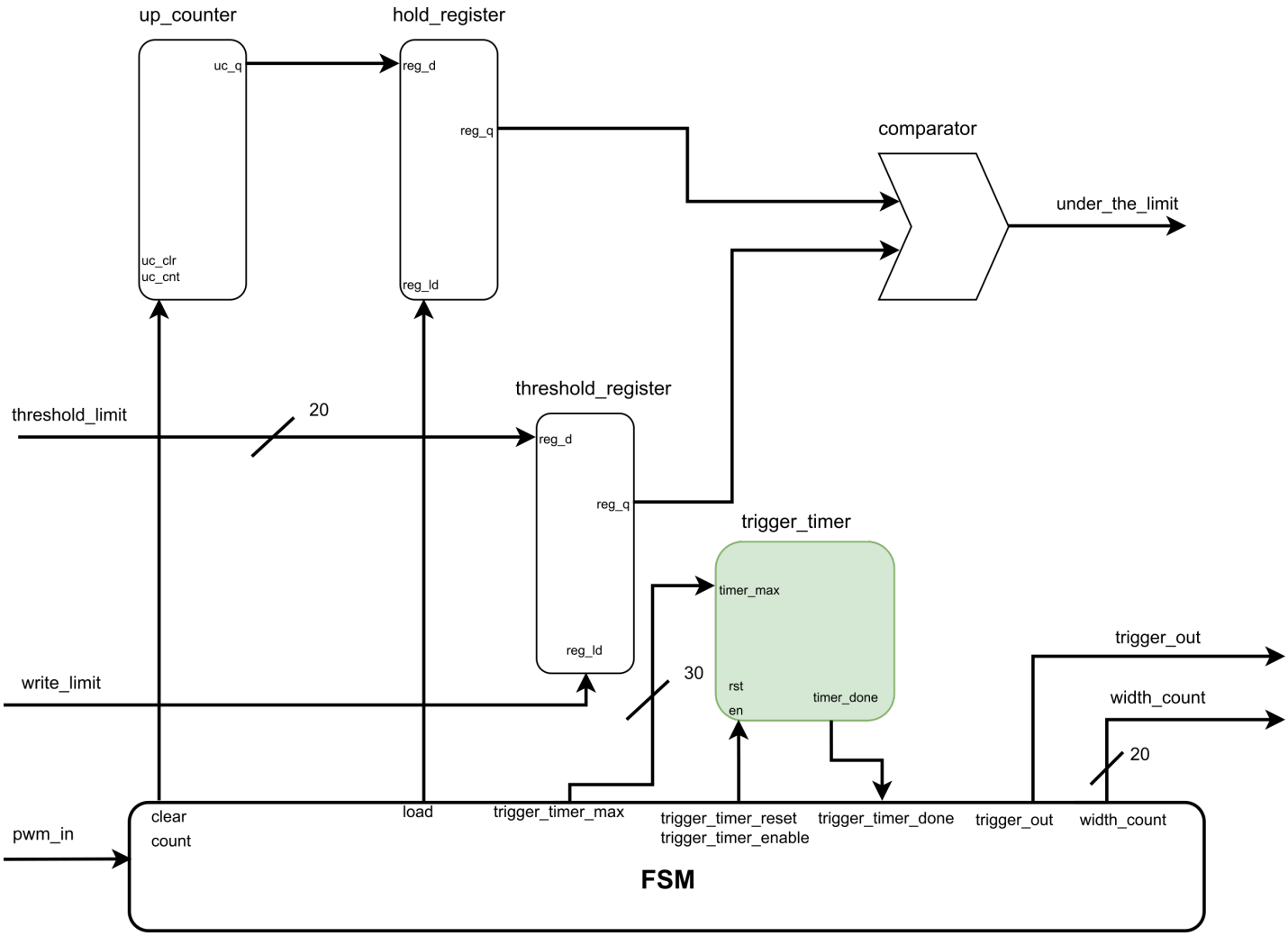

Fig. 20. FSMD diagram for the PWM Reader module, modified to include a trigger signal

PWM Reader with trigger (ASMD)

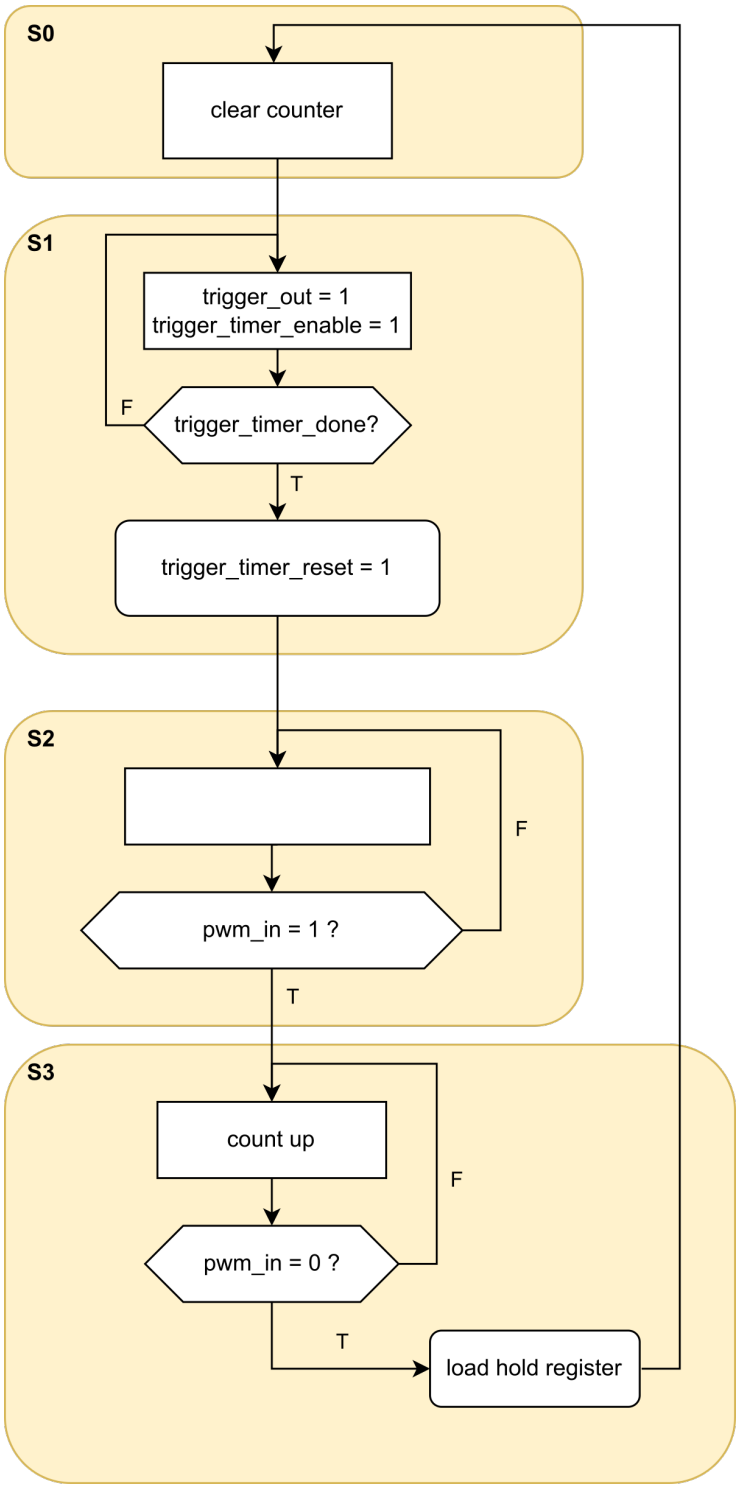

Fig. 21. ASMD diagram for the PWM Reader module, modified to include a trigger signal

APPENDIX C
TIMER COMPONENT FOR FSMD SOLUTION

Timer component (FSMD)

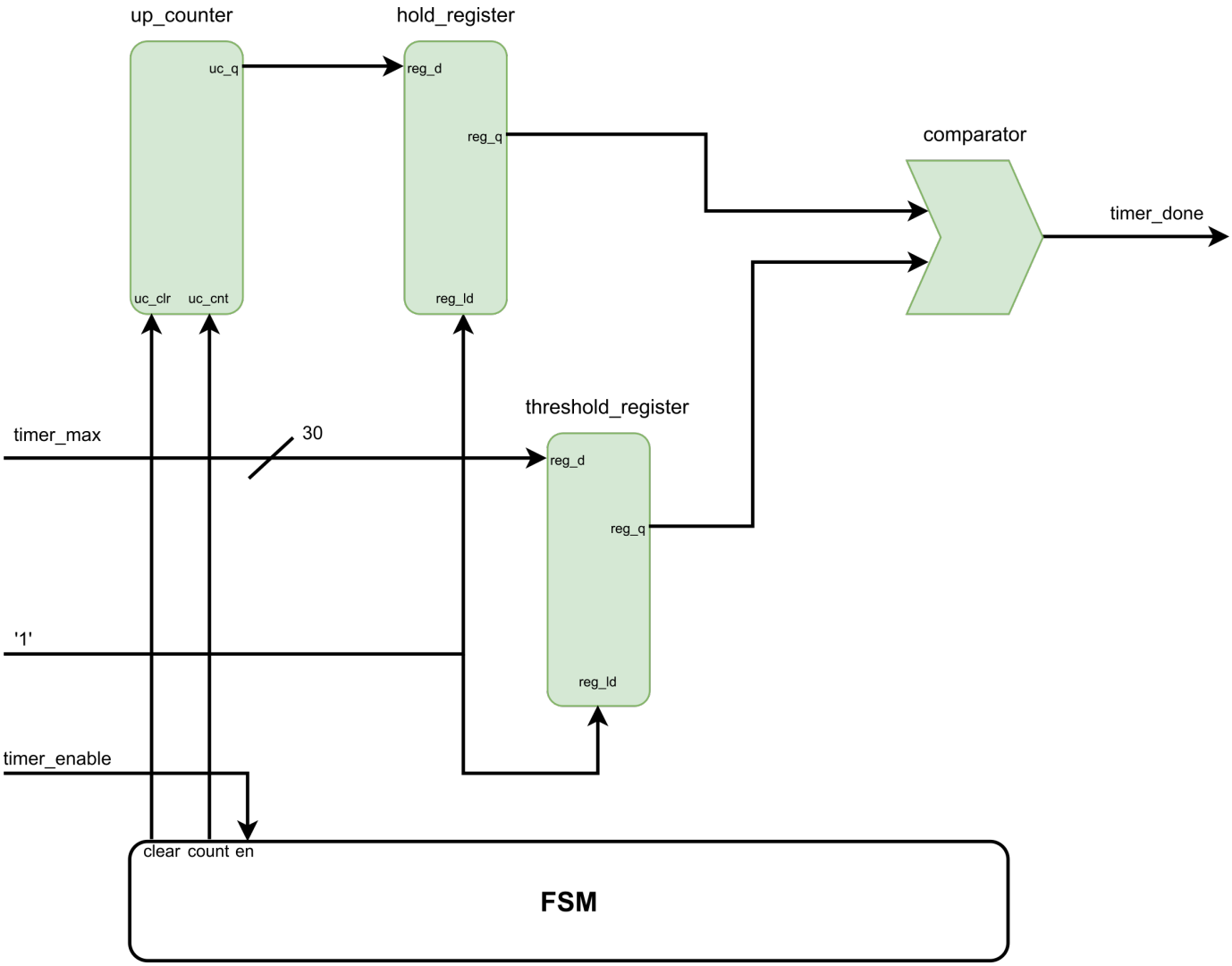

Fig. 22. FSMD diagram for the PWM Reader module, modified to include a trigger signal

Timer component (ASMD)

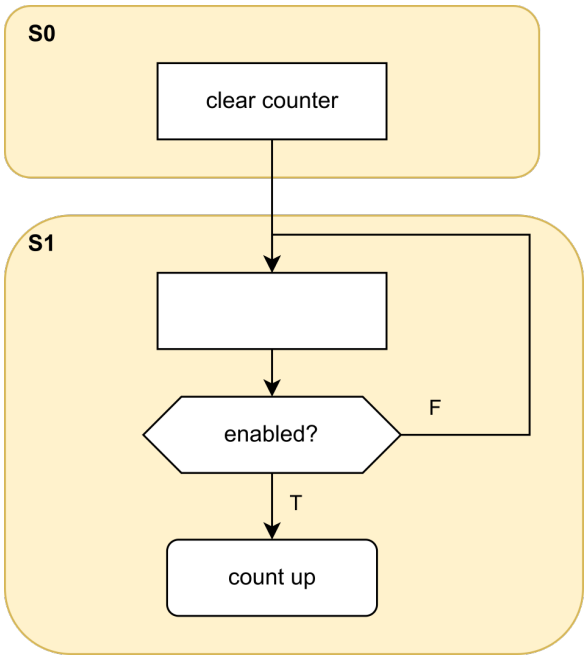

Fig. 23. ASMD diagram for the PWM Reader module, modified to include a trigger signal