

Autonomously Moving Car with Ultrasonic Sensor

Project Report for CS4110, December 2023

Knut J Bjuland¹ and Sondre Bitubekk²

Group 2

Department of Science and Industry Systems

Faculty of Technology, Natural Sciences and Maritime Sciences

University of South-Eastern Norway

Kongsberg, Norway

Abstract—This report contains the work we did during the CS4110 course project, in which we are tasked with creating the hardware design for an autonomous car with an attached ultrasonic sensor. The hardware design is created in an editor using VHICL Hardware Description Language (VHDL), and can be synthesized, implemented, and compiled to a bitstream to be added to an FPGA. We have used two types of approaches for the design; an FSMD (Finite-State Machine with Datapath), and an ASIP (Application-Specific Instruction Set Processor). This document will highlight our implementations, findings and challenges.

Index Terms—ASIP, ASMD, Hardware, FSMD, FPGA, VHDL

I. INTRODUCTION

The project assignment requires us to create an autonomous car which has 3 possible movement sets; forward, backward, turning left. In conjunction with an ultrasonic sensor, these movements sets are selected autonomously based on certain requirements. The ultrasonic sensor is triggered every 100 milliseconds, and the trigger is held in high (bit value 1) for 10 microseconds. After a short burst, the sensor board will send ultrasonic waves with a frequency above human hearing, and set its echo signal high. By measuring how long the echo signal is high, we can figure out the distance to an object by calculation using the speed of sound (340m/s), where a shorter duration means a short distance.

The general movement loop of the car is (1) to drive forward until too close to an object, (2) to start going backward until far enough away from the object, and (3) to turn left 90 degrees. The echo sensor is used to measure both 1 and 2, while turning is performed using a timer. To perform this task, we are supplied with a Basys-3 board (containing an FPGA), a HC-SR04 ultrasonic sensor, and a car frame with 2 L298N motor board driving 2 motors and wheels each.

Our goal is to implement the ability for autonomous movement in a model car using an ultrasound sensor. We intend to implement as FSMD in different form as as FSDM implement in hardware, as emulated riscv running on hardware and finally as software only solution. The last time of approach is called an application-specific instruction set processor (ASIP) which adds application-specific instruction to a general cpu like the risc V (1). The ASIP will be more flexible than the traditional FSMD

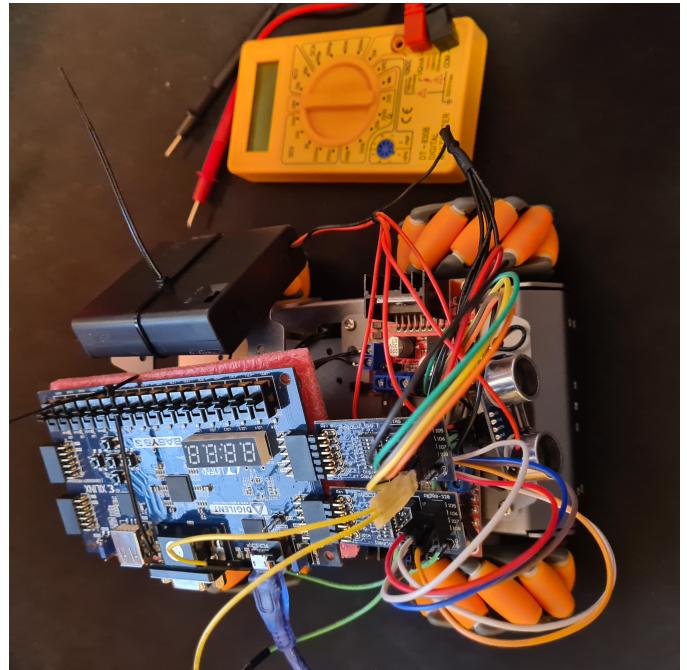


Fig. 1. Picture of the assembled hardware

as it is difficult to repurpose an FSMD to a different task. This is possible for a ASIP which has more general data paths.

Disclaimer: We also had the opportunity to design the logic using a software approach or a High Level Synthesis (HLS) approach, both of which require a different board, Zybo Zynq-7, with a Central Processing Unit (CPU). Both of these approaches require a predefined design of logical components, but HLS incorporates hardware design and requires planning of where to put logic. Hardware and software is generally sort of opposite in terms of efficacy and versatility.

However, due to group size and time constraints, we were only able to focus on the two approaches, and decided upon looking into the hardware-based approaches as these differentiate the most from previous experience.

II. HARDWARE

This section will describe the hardware components that have been used in this project, and consists of the aforementioned parts; Basys-3, ultrasonic sensor, motor boards, motors, and also a level shifter and a battery pack for power. Connecting these components together correctly is essential for the operation of the car, and planning how to connect everything together affects the hardware design to some extent in terms of what is required to be read and what is required to be sent out. Which ports to use, and what data to send and receive can be defined in the top file, and connected to hardware via a constraints file.

Disclaimer: Unfortunately, we are unable to get this part of the project right, and as such, none of our implementations have been verified by observing the actual car move as predicted. Whether this is due to incorrect use of the power source, incorrect use of physical connections, incorrect binding between connections and logical ports, or an issue in the design itself, is not yet known. This has been attempted debugged by using a measuring instrument to verify that the wires are whole and connections go as expected, and by verifying the design in test benches.

A. Overview

- Basys-3 FPGA Board
- HC-SR04 Ultrasonic Sensor
- L298N Motor Board
- LVLSHFT PMOD Level Shifter (3.3v-5v)
- Battery Pack (4xAA)

The Basys-3 board has a programmable FPGA, and both hardware designs are transformed to bitstreams that can be run on it. Since we have multiple hardware components, the PMOD connectors are needed to be used to be able to output enough signals. As such, the constraints file needs to map the digital output signal to the correct ports going out of the PMOD.

Since each motor board has 6 pins each that need to be controlled, and we also need to connect the echo sensor with 2 pins, two PMODs must be used. In this case, we also use a level shifter between the Basys-3 board, and both the ultrasonic sensor and the motor boards, as these run at a slightly higher voltage (3.3v to 5v). Based on the reference manuals for all components, we connect each pin together in a specific order.

B. Connections

Below are the four main steps of connecting everything together to produce the end result depicted in image 1, where the Basys-3 board is connected to each motor board and the sensor via two PMOD level shifters and wires.

```

##Pmod Header JXADC      -- Left Side Motors + Ultrasonic Sensor
##Sch name = XA1_P        -- IO1 on level shifter      -- IN1 on motor board
set_property PACKAGE_PIN J3 [get_ports {dig_out[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {dig_out[0]}]
##Sch name = XA2_P        -- IO2 on level shifter      -- IN2 on motor board
set_property PACKAGE_PIN L3 [get_ports {dig_out[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {dig_out[1]}]
##Sch name = XA3_P        -- IO3 on level shifter      -- IN3 on motor board
set_property PACKAGE_PIN M2 [get_ports {dig_out[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {dig_out[2]}]
##Sch name = XA4_P        -- IO4 on level shifter      -- IN4 on motor board
set_property PACKAGE_PIN N2 [get_ports {dig_out[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {dig_out[3]}]
##Sch name = XA1_N        -- IO5 on level shifter      -- ENA on motor board
set_property PACKAGE_PIN K3 [get_ports {dig_out[8]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {dig_out[8]}]
##Sch name = XA2_N        -- IO6 on level shifter      -- ENB on motor board
set_property PACKAGE_PIN M3 [get_ports {dig_out[8]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {dig_out[8]}]
##Sch name = XA3_N        -- IO7 on level shifter      -- TRIG IN on sensor board
set_property PACKAGE_PIN M1 [get_ports trig]
    set_property IOSTANDARD LVCMOS33 [get_ports trig]
##Sch name = XA4_N        -- IO8 on level shifter      -- ECHO OUT on sensor board
set_property PACKAGE_PIN N1 [get_ports echo]
    set_property IOSTANDARD LVCMOS33 [get_ports echo]

```

Fig. 2. PMOD JXADC definitions excerpt from constraints file

1) *Constraints File:* The constraints file and the input and output values of the top file are connected together. PMOD JA's 8 output pins (It has 12 pins, but 4 of these are for power) are designated as the right side motor control, where the first 4 pins are set as the movement direction, and the next 2 pins are set as the PWM signals for each motor. PMOD JXADC's 8 output pins are designated in the same way, but for the left two motors, and the last 2 pins available are set as trig to and echo from the ultrasonic sensor, respectively.

2) *PMOD Level Shifter:* The two level shifters are attached to the two PMOD connectors. The level shifter has one set of input pins, and two possible sets of output pins, and can pass power. We assume that pin 1 in the J1 section goes out pin IO1 in the J3 section, providing a 1-to-1 relationship. In the middle of the board are 8 switches which control the direction of the signals, and since we send signals out to the motors and to trig on the sensor, all of these are set to the right (away from the Basys-3 board), except for the echo pin.

3) *Wiring:* Each motor board and the sensor are connected to the corresponding pin out on the level shifter via wires. To reiterate, the IN1, IN2, IN3, and IN4 pins from each motor board are connected to the first 4 pins on the level shifter, IO1, IO2, IO3, IO4. Then the ENA and ENB pins are connected to the pins IO5 and IO6, and finally, for the left side level shifter, the trig is connected to the IO7 pin and the echo is connected to the IO8 pin.

4) *Power Source:* The cables from the battery pack are connected to the motor boards and the sensor. Each motor board has a VS and a GRND input for power, were a male red wire is attached to the VS connector and a male black wire is attached to the GND connector. Similarly, a red female wire is connected to the sensor's VCC pin, and a female black wire to the GND pin.

III. ULTRASONIC SENSOR

The ultrasonic sensor is designed in the same way in both implementations as most of its logic works separately from the rest aside from a write_limit input to load a value to the threshold register and an above_limit output to signify when the counter has counted above the specified threshold limit.

A. Trig Counter

In our implementations, a trig counter starts automatically and runs continuously without awaiting any signal. During its lifetime it continuously sends a trig signal lasting 10 microseconds and repeats every 100 microseconds. This ensures that an echo is measured throughout the operation of the car, so that distance to objects in front of the sensor is always measured and accounted for.

While it can take a start signal, this signal is set to high by default, as interaction with the trig timer is not deemed necessary. It can be left to start automatically and continuously send a periodic trig signal as long as the board has power.

B. Components

Other components, aside from the trig counter, is an up counter to measure how many rising edges where echo is high, a hold register to hold the value of the up counter, a threshold register for comparison with the hold register, a segment display to display the distance to an object in centimeters, and finally a FSMD with two states, of which mainly decides when to clear the up counter, start counting, and load the value to the hold register, all based on the current value of echo (s0 when echo is low to clear the counter, s1 when echo is high to start the counter).

As with any design, all of these components are tied together in a top file, which is then instantiated in the top file of either the main FSMD or the ASIP, which also requires that this top file has the ports trig, echo, an and seg defined.

C. Segment Display and Distance

The segment display on the Basys-3 board is set to display the calculated distance to objects in centimeters. Distance is equal to time multiplied by speed, but in this case the value must also be divided by 2 since the sound waves goes the same distance twice (to and from).

The time it takes from when the echo is sent until it is received, can be calculated using the up counter's value. It counts the number of rising edges while echo is high, and this value is multiplied by 20 nanoseconds, which is the time between each rising edge when the clock is high for 10ns. The speed of sound is $343\text{m/s} = 34300\text{cm/s} = 34.3\text{cm/ms} = 0.0343\text{cm/us}$. So, given a count of 50,000, we multiply this value by 20ns to get $1,000,000\text{ns} = 1000\text{us}$. Then we can find the distance by applying these values to the equation: $(1,000\text{us} * 0.0343\text{cm/us}) / 2 = 17.15\text{cm}$. This value is rounded to the nearest integer and displayed in the segment display. Generally, the segment display can account for decimals, but the implemented component does not.

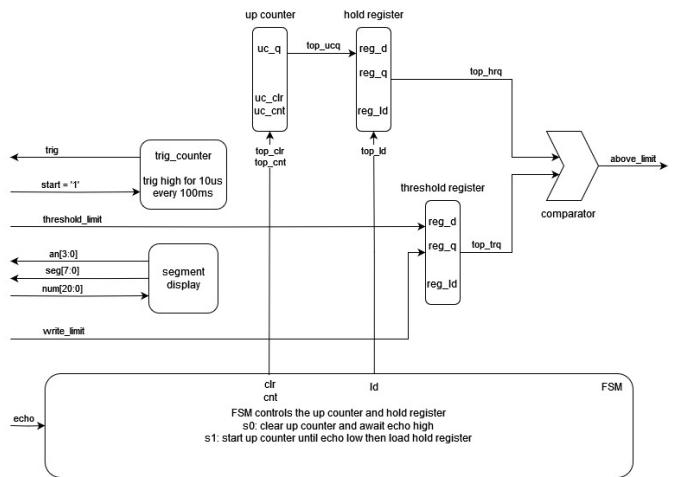


Fig. 3. Diagram of Ultrasonic Sensor design (Mostly provided in course)

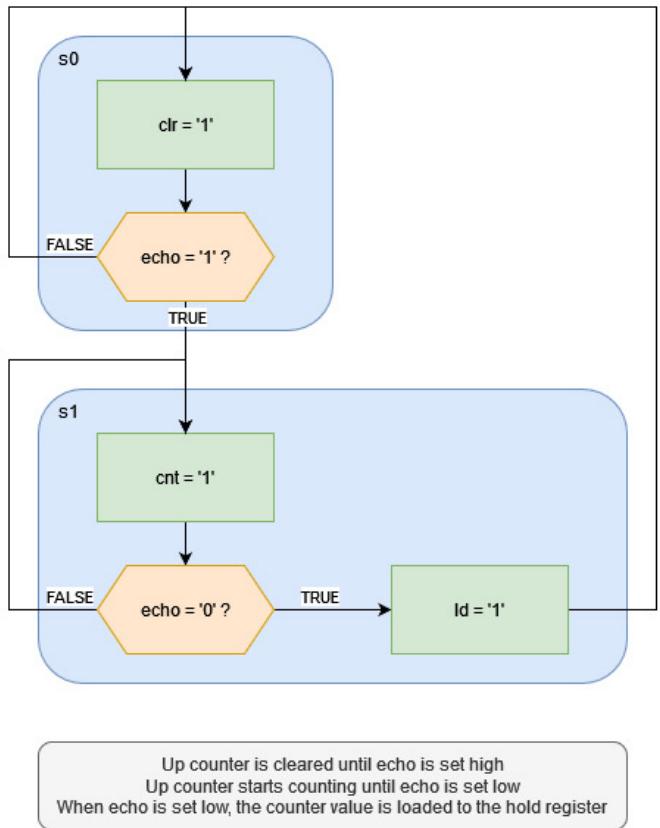


Fig. 4. ASM Chart of Ultrasonic Sensor FSM (Unchanged from the course)

IV. FSMD

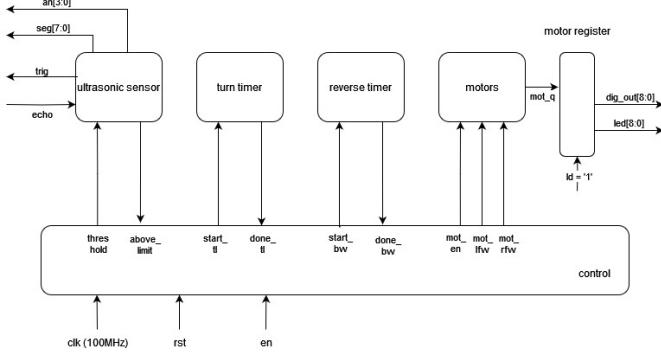


Fig. 5. Diagram of implemented FSMD (Partially based on supplied diagram during meeting)

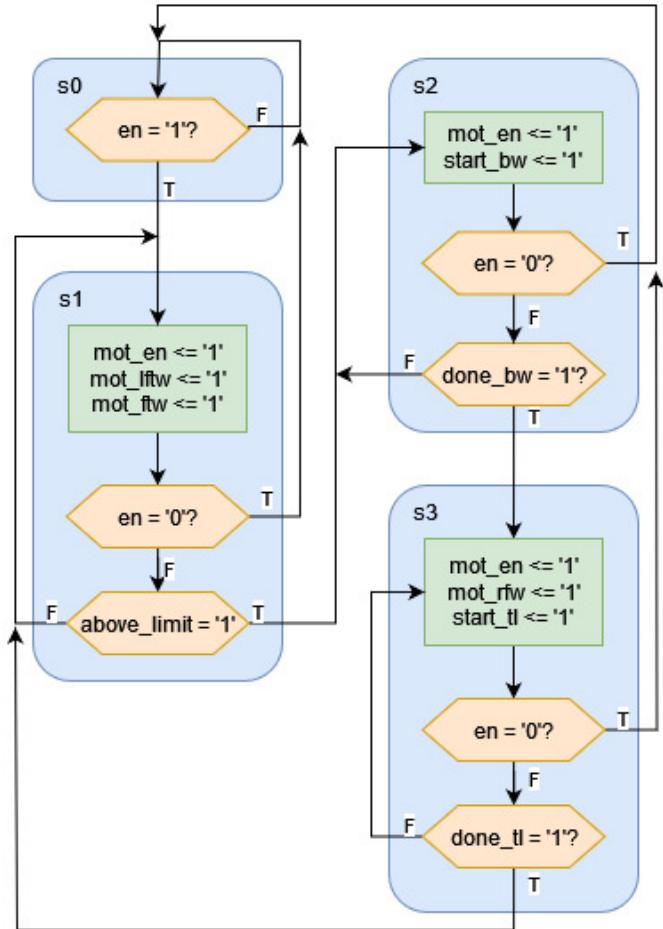


Fig. 6. ASMD chart of implemented FSMD

A Finite-State Machine with Datapath is a good approach to hardware design, where a control logic handles a number of predefined states and the flow between them. Each state is used to set specific output signals to high in order to orchestrate correct behavior among all components at given points during its operation. This allows for coordination and synchronization, and since hardware works concurrently, aside from explicit use of e.g. a process, synchronization of events is ensured.

The state machine is structured as a process with a sensitivity list. The sensitivity list decides whether to execute the contents of the process, and at the very least it must include the state reg type for it to change between states. Any input signal that is read in the process and requires an action as soon as it changes value must be added to this list. Input signals are generally used for deciding whether to change output signals and/or to change state. By default, all output signals are defaulted to low in the start of the process, and set to high depending on the current state and input signals.

A. Implementation

Our implementation consists of 5 components plus the ultrasonic component composition (See figure 5 for diagram), which are the control path, a motors file which contains the movement options for each motor which are set by the control path, a motor register to hold the output value from the motors, and two timers for timing reversing and turning. The timers can be specified with a limit for number of rising edges to count, where for instance 1 second is equal to 1,000,000,000ns divided by 20ns (Period between each edge), which equals 500,000,000 counts. The control path supplies a start signal and awaits until the done signal is set high depending on the current state.

1) Control Path: The control path evaluates input signals such as above_limit and the two timer done signals, and will output the signals mot_en, mot_lfw and mot_rtw, as well as the two timer start signals. An ASMD chart overview of the logic in this component is available in 6. The first signal, mot_en, activates the motors by setting ENA and ENB on the motor boards to high, while the following two, mot_lfw and mot_rtw, assesses whether to move the motors forwards or backwards. These three signals are sent to the motors component to create the combined 9-bit signal sent to the motor register and further out to the motors via the Basys-3's PMOD connector.

2) Timers: The timers are started in the appropriate states and state change awaits until the timer is done. As visible in the chart, an enable signal overrides the state transition, and can be bound to for example a switch on the Basys-3 board. This can be useful if there is a desire not to let the motors start at the moment the FPGA is programmed with the bitstream. The idea here is if trying to use the car with the battery pack, it might be necessary to program it first via

the USB connection with the short wire, disconnect it and place the car on the floor before starting movement.

3) *Sensor:* The ultrasonic sensor works sort of separately with little input, but will continuously send sound waves to measure distance and calculate whether above_limit is true. While this implementation uses a backward timer, it might be a better option to approach this as in the ASIP below, where instead the threshold limit is heightened and reversing is set to continue until above_limit is the opposite value. This reuses this functionality in a good way, but requires some fine tuning to correctly switch the threshold value and correctly asses the counter value. As describe in the previous section, distance is displayed in the segment display of the Basys-3 board. This is quite helpful for ensuring that it operates as expected, but as described in the hardware section, we were not able to get the assembled hardware to work.

B. Behavior

However to fully implement our solution we need to draw a ASMD which enable to separate the different mode in autonomous as different state in a FMSD. The first amsd will deal with how we send out the echo pulse. The system will switch between waiting for an echo and measuring the echo. The echo will be sent out continuously where there is 10 us between each pulse. As can be seen on figure 8 the car will move forward until it. When the car is moving the software will send a trigger to emit a ultrasound pulse. This pulse can only be send every 10 us and we will also wait for the pulse to return. In order to measure the distance we will count the time from the pulse is sent to the time it is received.

If this distance to obstacle it is to short, the car will stop and drive backwards before turning left and driving forward. If the distance it not to short the car will drive forward but when the distance it to short the car will stop and then reverse. Until the timer has reached the threshold it will drive backward and then turn left for a short amount of time before again driving forward. We used the VHSIC hardware description language in order to implement the riscv and FMSD in hardware. And we used an example file provided by Jose Ferriera, which we modified in order to implement either FMSD or riscv emulation in hardware.

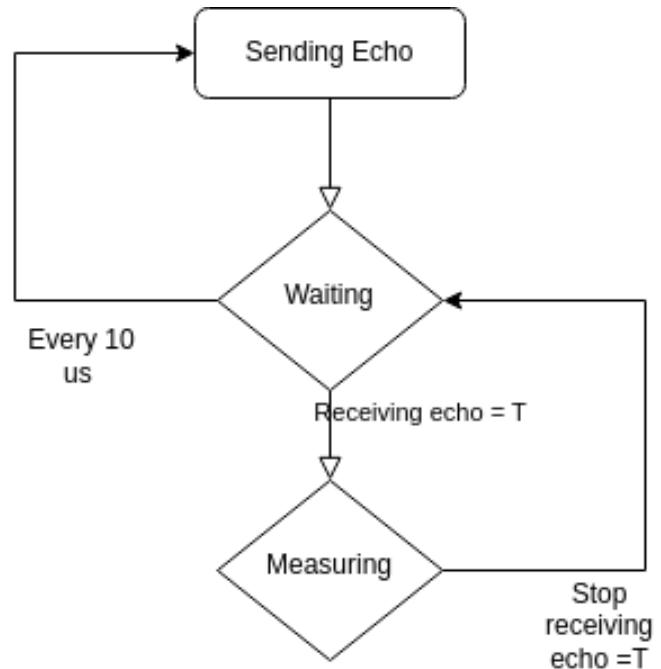


Fig. 7. Echo

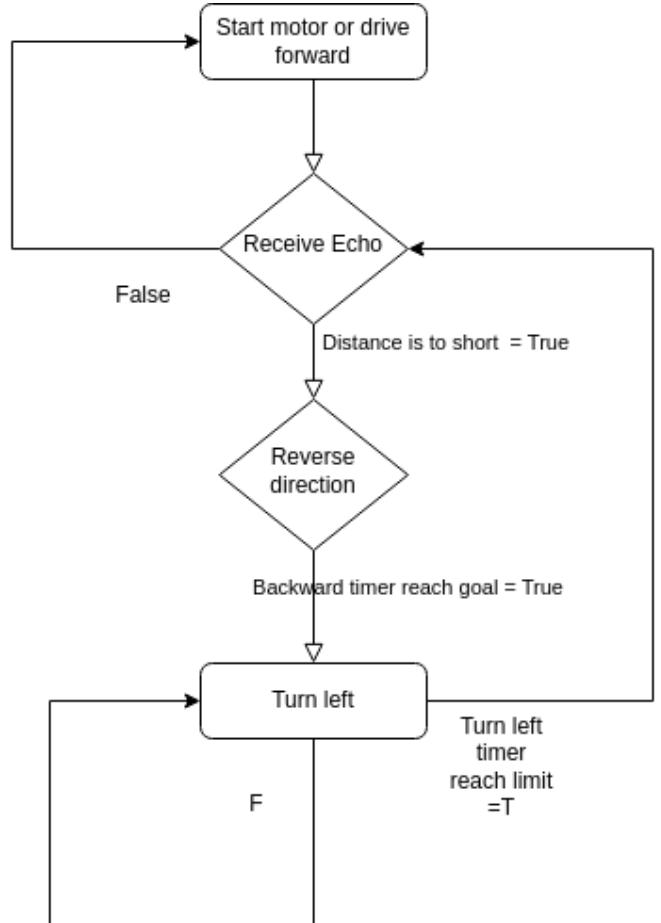


Fig. 8. ASMD for motor movement

V. ASIP

The Application-Specific Instruction Set Processor consists of many of the same components as the FSMD, but instead of relying on a state machine to send out signals, it instead uses a set of instructions to execute logic. As such, this architecture is programmable, where primarily only the instruction memory (imem) needs to be changed in order to change its behavior. To support this possibility, it has a number of other components than just the imem, such as a program counter, register and memory, and values in the register can either be used for the next instruction or be sent to a different part of the architecture.

The instructions consist of 24 bits designated as, from the left, immediate (8 bits), rs2 (3 bits), rs1 (3 bits), rd (3 bits) and opcode (7 bits). The immediate value is used for either loading a value into a register, or how many instructions to jump over forwards or backwards. RS2, RS1, and RD all represent the registers used for a wide range of operations. Finally, the opcode designates what specifically the instruction should do, and is read in the Arithmetic Logic Unit (ALU) and the control. As an instruction is read in the control, muxes, or switches, for various components can be enabled / loaded to control whether a signal passes through, and thus control the flow in the architecture to perform specific tasks.

An important instruction in the case of our implementation is the jump instruction, which can be used to check whether the threshold has been met or a timer is done, and then use this information to decide whether to jump to the immediate value or get the next instruction.

A. Implementation

Our implementation is based on the provided design from the course, which includes a default architecture for the ASIP and a separate architecture for the ultrasonic sensor as a Pulse Width Modulation (PWM) reader. These two designs are combined by adding the top file of the PWM to the top file of the ASIP, and defining all the required signals and ports, such as the trig and echo signals required for the sensor. A diagram of this updated design is available in figure 9, with the corresponding ASMD chart available in figure 10.

We are using a timer for timing the left turn. One instruction starts the timer, and another will check whether it is done. Once the timer is done, this instruction will cause a jump all the way back to the forward loop. In the sensor architecture, we are using two new components; a counter for periodically setting the trig signal high, and a display for updating the segment display of the board based with the calculated distance.

The trig counter sets trig high for a duration of 10 microseconds, and does this every 100 milliseconds continuously, as this action can start as soon as the board has power and is programmed, and can run uninterrupted throughout the program's lifetime. The display takes the calculated distance value, where the time passed for the echo to be received is equal to counting each rising edge of the clock while the echo signal is high.

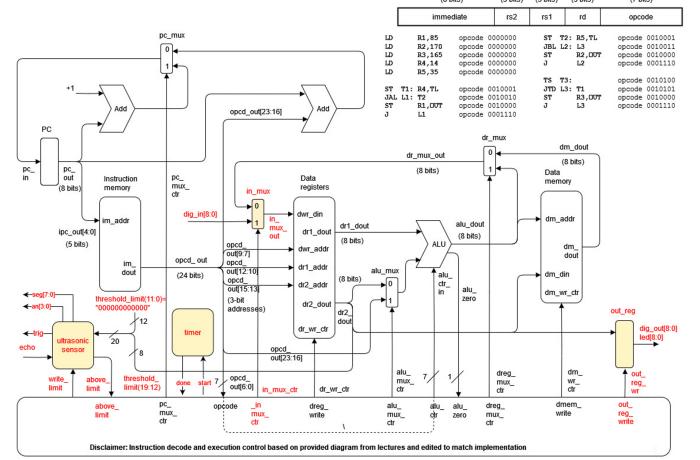


Fig. 9. Diagram of updated ASIP (Mostly provided in course)

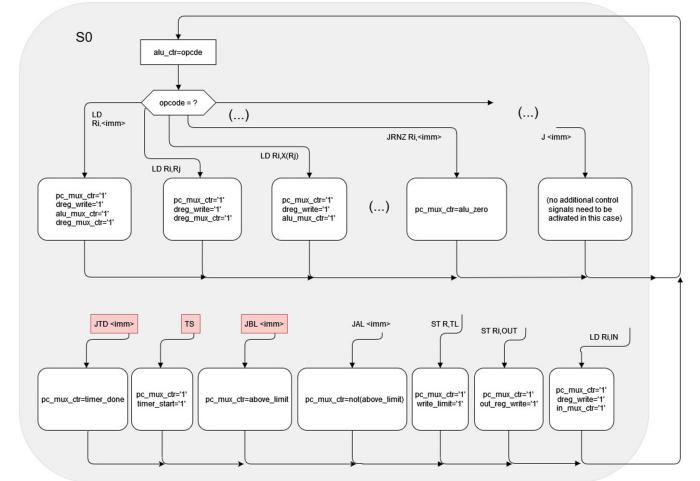


Fig. 10. ASMD Chart of updated ASIP (Mostly provided in course)

B. Instruction Set

Instructions are the most defining part of the ASIP as they describe the logical flow in the design, and allows for flexibility in functionality without changing the core design. Finding the right instructions is difficult, but a good approach is to think in terms of pseudocode and what the car should do on a high level. Since we are not using all the registers or performing arithmetic operations, there is no need to use the memory to store any data.

Table I displays the 17 instructions that are defined and used in the architecture, where the remaining 15 (32 total) are unused. We have added two additional instructions, TS and JBL. TS will start the turn left timer, and JBL, jump below limit, is equal to JAL, but works opposite. The threshold registers below are appended to a 12 bit binary of 0's to create a larger number consisting of 20 bits, as the echo counter can count up to a significantly high value.

addr	instruction	pointer	type	action
00	x"550080"		LD	R1,85
01	x"AA0100"		LD	R2,170
02	x"A50180"		LD	R3,165
03	x"0E0200"		LD	R4,14
04	x"230280"		LD	R5,35
05	x"008011"	T1	ST	R4,TL
06	x"030012"	L1	JAL	T2
07	x"002010"		ST	R1,OUT
08	x"FE000E"		J	L1
09	x"00A011"	T2	ST	R5,TL
10	x"030013"	L2	JBL	T3
11	x"004010"		ST	R2,OUT
12	x"FE000E"		J	L2
13	x"000014"	T3	TS	
14	x"F70015"	L3	JTD	T1
15	x"006010"		ST	R3,OUT
16	x"FE000E"		J	L3

TABLE I
ASIP INSTRUCTIONS

1) Load Registers:

- 0 - Load reg 1 with forward movement (01010101 - 85)
- 1 - Load reg 2 with backward movement (10101010 - 170)
- 2 - Load reg 3 with turn left movement (10100101 - 165)
- 3 - Load reg 4 with forward threshold (00001110 - 14)
- 4 - Load reg 5 with backward threshold (00100011 - 35)

2) Forward Threshold and Loop:

- 5 - Store reg 4 to threshold register
- 6 - Jump to instruction 9 if above_limit is high
- 7 - Store reg 1 to output register
- 8 - Jump to instruction 6 to restart loop

3) Backward Threshold and Loop:

- 9 - Store register 5 to threshold register
- 10 - Jump to instruction 13 if below_limit is high
- 11 - Store reg 2 to output register
- 12 - Jump to instruction 10 to restart loop

4) Turn Left Timer and Loop:

- 13 - Timer start
- 14 - Jump to instruction 5 if timer_done is high
- 15 - Store reg 3 to output register
- 16 - Jump to instruction 14 to restart loop

5) Limitations: As this instruction set consists of 3 loops and does not change between loops before a check is met, the program is sort of locked. While there are no other instructions than these, if the program were to perform other tasks, such as a calculation, while also ensuring to control the car's movement, it would require a few changes. For example, since turning left uses a hardware timer, this timer could have locked the output until done, while the instruction could have jumped back to the forward loop.

VI. CHATGPT

We used Chat gpt as a computer-based tool and asked chat gpt for advice. Furthermore, we send our code to chapt gtp to check for errors and for suggestions. With this in mind we also used chatpgpt as a search engine to search for how to use VHDL and we used it to figure out how to connect our port through a signal in the top file to another in port.

We also asked chatgpt for an example on how to make a counter as we could not use wait for in non-simulation vhdl file.

A. Sondre's Experiences

I am quite happy with being able to use ChatGPT for this project, and a large reason for this is the fact that there aren't as many good sources for knowledge as I feel there are for software programming. I often pasted snippets of code, and sometimes even the entire contents of the VHD file, and asked it to clarify some uncertainties and challenges, and also assess if I am somewhat in the right. I must emphasize that I believe it was sometimes incorrect in some of its statements, but this could be due to limited knowledge of the other parts of the project. While this discrepancy is no revelation, it is important to be very aware of possible limitations and errors especially for myself as I am entirely new to this field of computer engineering.

Worth noting is also the fact that while the ASIP uses an instruction set similar to RISC-V, the instructions are custom and thus ChatGPT is less able to provide help. However, it can help understand the flow and describe it back to me, evaluate whether my instruction hexadecimal values are correct, and provide some points to what to improve or change.

Overall, I feel that it is able to provide good examples that can be used as a basis. If anything is unclear to me, or I am curious about possibilities, I find that it provides good insight and helps me learn about embedded systems and hardware design. Especially being able to ask about possibilities is something I find very useful, as I am generally curious about trying out new approaches and figuring out the boundaries of what can be achieved. Since it can form an opinion about specific questions, it can provide a better answer than any forum post can, as these are typically more general.

ChatGPT has also been helpful when writing this project report, especially when it comes to using LaTeX and Overleaf. Formatting and structuring text in LaTeX can be quite challenging at first, and I often find myself prompting about how to achieve specific results. For instance, how to create a table, how to use and reference appendices, and how to place figures. Generally, it is able to provide code snippets for achieving whatever you want, and it can also help incorporate the text you wish to apply in a fully generated snippet of code. Other useful aspects is to provide synonyms for common binding words, word suggestions for something you are only able to describe, and provide general guidelines for writing different types of papers.

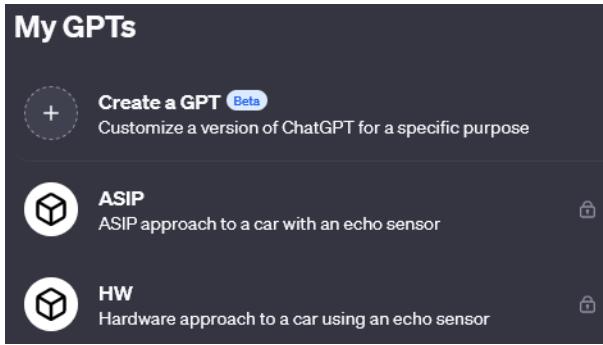


Fig. 11. Image of two created GPTs

1) Personal GPT Helper: I attempted to create my own GPT helpers by supplying the project files, asking these to analyze the files and provide some feedback on structure, possible issues, and suggestions, but I did not find this to be very helpful. It wouldn't always read the entirety of the files, and it did not seem to be able to string everything together. So, that specific experience did not quite impress me, but I might have approached it incorrectly as it is my first time using ChatGPT in this way. What I was hoping for was for it to provide some insight into how to change the design to achieve specific behavior in the FPGA, but I found it better to ask more simpler and specific questions.

2) Examples of Prompts and Replies: This subsection covers 7 different prompts and replies that I found useful during project development and project report writing. A few of the prompts answer specific technical questions, some provide code examples, and one helped me format a table to display the ASIP instructions.

Appendix A displays some code examples for how to implement a package to contain a procedure, which is similar to a function in software. This was not a functionality I was aware of existed, but something I have found useful in order abstract and reuse logic, which is quite typical in software. Keeping logical components clean, concise, and limited to specific tasks seems quite similar to the thought process in software. The provided code examples gave me insight into how to structure the package and process, and how to reference it in other files.

Appendix B helped answer a question I had regarding concurrency and sequence of code lines. In software, lines of code are normally executed sequentially. Because of this, I was a little curious whether assigning an output signal based on changing internal signals would happen instantly (such as in the comparator), and it does. Signals in hardware are either on or off, and can be stopped with muxes in order to direct the signal at the right time. As soon as the mux is set to high, the signal will continue in its path.

Appendix C helped answer a question I had about whether there is a need to handle state for the ASIP, which I found was not really the case since there is essentially just one state

in which all instructions are executed. Personally, I did not find it necessary to handle state here as I could not think of a situation where the instructions should differentiate based on state. At that point, I would assume it would evolve towards a FSMD hardware implementation.

Appendix D provided a code example of a timer, which can be provided a count limit and outputs a done signal. This piece of code is quite simple and reusable, and it simply counts up until reaching the designated limit and set done low. Once the limit is reached, the counter is reset and done is set high. This timer is not that unlike a modulus m counter, and both will continuously loop through. Adding a start signal could help control the timer further.

Appendix E explains how many centimeters a count of 4096 equals. I asked this question specifically, as this is the lowest value possible if the ASIP outputs 1 to the threshold limit. The output value is appended to 12 0-bits, where $2^{12} = 4096$. The purpose was to get a quick calculation of how to find the distance in centimeters, provided the known variables of period between rising edges and speed of sound.

Appendix F provides a piece of code for the test benches to mimic the ultrasonic sensor, by slowly getting closer to an object (decreasing the time when echo is high), then slowly reversing backwards again (increasing the time echo is high). This provided a good starting point, but being able to simulate the exact logic requires more tweaking to the test bench.

Appendix G is an example of how I used ChatGPT to help me with this report. Here I specifically got it to generate a table based on the instructions defined in the ASIP. To create the columns and apply the data, it also read the comments from the code. This is a prime example of how ChatGPT can help make tedious and error prone activities easier, more reliable, and more effectively use your time.

VII. SOURCE CODE

We shall now look at the top file for our implementation of FMSD which we implemented in hardware using only VHDL. The file below is the top file that is used to connect different parts of our program. We have three counter, a file for display and a for control of the echo and car movements. This file will have a FSMD. The counter was used to set the time the car drove backwards, went left and to ensure that the trig only sent out a ultrasound pulse each 10 us. The source code for the FSMD is provided at this site github.com/knutj/sensorcar/tree/main/hw

The next thing we will look at is the ASIP implementation where we emulate a simple riscv with io on fpga chips. This implementation is a mixture of hardware programming using VHDL and software program using assembly for riscv. We will loop through the riscv instruction that is listed below which fetch echo starts a timer and issues command if the timer is to short which reverse the car. Most of the logic is implemented in VHDL and we added just what was necessary to interface with the ultrasound sensor, a counter and way to send signal to the car, and the source code for this is provided here github.com/knutj/sensorcar/tree/main/asip

VIII. RESULTS

We will first take a look at the simulation of the FSMD state machine as seen in figure 12

As we can see on this figure there is echo signal has a fixed amount of time between each pulse as we use counter in order not to send a pulse before the ultra sound sensor is capable of sending a new echo pulse. IN order to measure the distance we measure the time from the ultra-sound is sent to when receive it. Furthermore, we also see that the motor signal indicates the motor stops when it gets too close to a wall, as tested in our test bench script. Due to time constraints, we were unable to test our script in the cars.

The difference between ASIP with FSMD we were unable to get the car to stop in our simulation, but we managed to start and saw an echo pulse as seen in figure 13

A. General Observation

While we were unable to get the hardware to work, we can instead evaluate our results based on the waveforms produced by the test benches. Simulation the hardware design is in any case very important prior to testing it out on actual hardware. The useful information a waveform can bring will help illustrate how the logic functions given certain inputs and variables, and can give important insight into how the design will work in reality. We are able to see how and when the bits switches between low and high, and what binary values this results in. Particularly interesting is to see whether the control paths work as intended given specific inputs, and in regards to the ASIP, it is particularly interesting to observe the flow of the logic based on how it steps through the instruction addresses. Since several of the instructions are jump instructions, it becomes quite apparent why a jump occurs, and how these help create both loops and change in behavior when needed.

B. FSMD Waveform

The FSMD waveform (Figure 12) does not illustrate how the motor changes direction because the given threshold is too high for the simulated echo duration, but we can observe that the counter and threshold functions as expected, and that the segment display value is changed when the counter updates. It is also clear what motor inputs are outputted based on the dig_out binary of 9 bits, where the most significant bit is coupled to both the ENA and ENB inputs of the motors, and the following pairs of bits assures forward movement with 0 and 1 (01). As mentioned, we were unable to get the car to move, but based on the waveform, it should have.

The segment display should also have been updated with a value for distance, based on this waveform. However, this is a simulation, and it is clear the culprit is that the hardware connections are not working as intended. The issue lies either in the constraints file or in the actual hardware with regards to boards, connections and wires. Forward movement is the default state, so the motors would normally have engaged. The display is depending the up counter, and if the trig signal is not sent out or the echo signal input is not set high, it will never

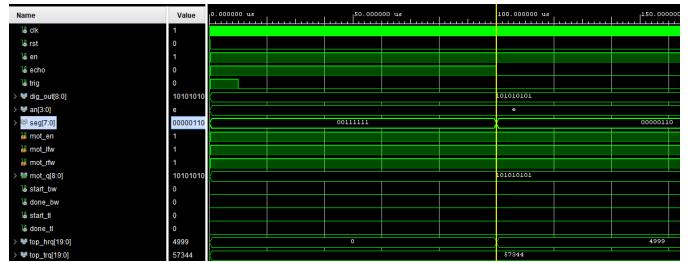


Fig. 12. FSMD simulation waveform

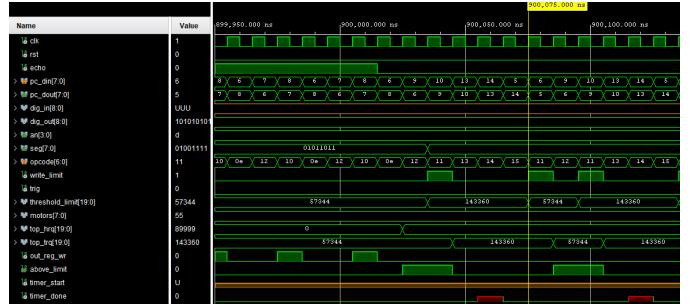


Fig. 13. ASIP simulation waveform

start counting, subsequently not updating the display with the distance in centimeters.

C. ASIP Waveform

The ASIP waveform (Figure 13) illustrates how the instruction flow changes based on various input values, and as the threshold value has been altered in this case, the above_limit is periodically set to high. At first, we can observe the forward movement loop for address 6 to 8, which is as desired. As soon as above_limit is high, it jumps from address 6 to 9.

Address 9 first loads the backward threshold value to the threshold register by setting write_limit high. It then moves to address 10, in which there is a check for below_limit, and here it is observed a clear error. The idea here was to jump to the turn left loop if the echo duration was above the new threshold limit, to indicate that it is Y centimeters away from the object as a higher echo duration equals a further distance. Obviously there is a logical break here, and instead it performs the jump to the turn left timer start and loop at address 13 right away without outputting the backward movement.

Address 13 starts the timer, but the waveform displays an error for the timer_done, while timer_start is not even set to high, even though it should be set high via this instruction in the control path. Although timer_done is low, address 14 still performs the jump back to the forward threshold assignment at address 5. As above_limit is again set to high, the forward movement loop jumps to the backward threshold assignment before outputting forward movement. These observations indicate that there are still some logical issues to work out with both how to evaluate threshold in the cases of both forward and backward movement, and to properly handle the turning timer.

IX. DISCUSSION

We should have gotten similar results when we compared the ASIP autonomous car vs FMSD autonomous car. There are several points that need to be examined before we decide the amount of software and the amount of hardware we should use in our solution. In order to come to a conclusion we have to measure the performance of both the FMSD or the ASIP. The ASIP uses a mixture of hardware and software in the form of emulated simple risc V cpu that has added the ability to interface with the ultra-sound sensor and with output to the motor. We did not add an instruction to our ASIP implementation, but according to an application-specificor ASIP it is possible to add application specific instruction to make it more suitable for the designed task (1).

The FMSD main strength compared to the ASIP implementation is that it use less space on the chip if we were to make a custom chips. In addition every thing is implemented in hardware and it will therefore run faster and with less power consumption. However it also less general than applications specify instruction set. The ASIP is less specialized than a pure finite state machine implemented in hardware as it also contain a custom made CPU along with application specify data-path and instruction (1). ASIP is easier to repurposed to new task than a FSMD as it more general. However, as this solution has more flexibility but comes at the cost of increased power and slow ere execution of the software. However as we have not test this it is difficult for us to predict the performance of both solution and how much power the would draw from the battery. When we ran our simulation, we used a computer that did not use a battery.

However, we did test a very simple solution where a slow moving car avoids obstacles by backing away and driving forward. There are several devices like self moving vacuum cleaner and lawnmower that used similar methods.

A. Improvements

We would have improved the implementation by use an automate method to select how much of the design that should be software and how much of solution shoAnotherhardware. Another way to improve the solution would be to optimize iusage,sor power usage,such that we could run the car asong as possible of a battery charge. Another way to make an improvement would be to use RTL analysis in Vivado in order to get overview over the connection between the different software parts. If we could do the project again this part would have been used much more and we would use drawing to select where instant a timer should be placed.

1) Test Bench: Our test benches are quite generic and would require some more fine tuning specifically due to the fact that the time to perform actions in real time are much higher than what is feasible to measure in a simulation without having to wait for a long time. Trying to use higher values for simulation length takes significantly longer time each time is it multiplied with 10. As such, timers and limits need to be adjusted to a value that can fit within the

time span selected. The caveat here is that you change the parameters of the design which may affect how it works, but if all variables are adjusted in correct relation to each other, it should still be able to mimic reality. After all, it is a simulation, an approximation, to help envision that both bits are switched correctly and values are set and applied correctly.

2) Implementation: Since the test benches are not quite correctly set up, and we are unable to engage the motors or sensor, there are likely to be a few issues in the implementation. The first logical issue that comes to mind is in the ASIP, where the jump checks for above limit, below limit, and timer done might not work entirely as expected.

Whether there is a misunderstanding here or not, we are not sure. As we understand it, the up counter will count a lower and lower value as you get closer to an object, because the time it takes for the ultrasonic wave to travel to and from an object is lowered. Therefore, we would assume the first jump, from forwards to backwards, would occur when the counter counts below the limit, not above.

At this point, the intention is to set the threshold higher and now jump once the counter is above the limit, which would signify that the car has moved a number of centimeters away from the object. Now the timer can start counting, and a jump back to forwards movement can happen once the timer is done.

B. Challenges

One challenge was to figure out how to use Assembly to send out a trigger, receive the echo and then start the car, and it was also challenging to figure out how to set up the wire to connect all part that was used in the car.

1) Group Size Decrease: A significant challenge to our project work was an abrupt change in group size, from 4 to 2, about half way through the process. Although we had already assigned each part of the project and had started with the hardware implementations, it would still have been quite helpful with another set of members to discuss problems with. That being said, the project took a little time to get started, and the time together could have been spent more effectively. It might not have changed too much, but sometimes just a small change of perspective, a small suggestion, or just having to explain a problem multiple times, can help in finding the solution to a problem. Naturally, it helped us tremendously that we could focus on only two approaches to the project, and as such, in terms of amount of work, it was possible to complete the project.

2) Sondre's Challenges: I have come across multiple challenges, ranging from understanding the hardware design logic, the full meaning of the ASIP, how to connect the hardware together, and having issues with both Vivado and Vitis.

While I understand how to connect all the hardware design parts together and how to structure it, I have had issues with figuring out why the design does not always perform as expected. The main reason to this is that while the test

bench can be somewhat simple, it still might require some fine tuning to be able to correctly simulate the design. As such, certain variables needs to be defined in the generic part of the top file and not as constants within the design, so that they can be altered in the test bench. This is mainly due to the fact that simulating above 10 milliseconds starts to take some time. For instance, turning the car takes 100 milliseconds, and it is desired to see the turning action multiple times in the waveform, the simulation would have to be a few seconds long, taking significant time to generate.

I spent more time than necessary on a specific issue. I wanted to represent the digital output via the LEDs to try to verify it as working, but what I learned is that the dig_out signal cannot be used for too many connections at a time. Instead, I had to define a separate LED std_logic_vector and assign it the same value as dig_out. This has limited significance to the project as whole, but it took some hours away to figure out.

I found it a little challenging to get a good grasp of the ASIP. While I understood it entirely as presented in class, I just could not get my head around about how to use the sensor's above_limit to cause a jump in the instruction set. It became quite obvious when explained to me, as the control file already exhibits the same logical use of alu_zero for two of the other jump instructions. I used ChatGPT to help me find a solution to this, but it would generally only propose to use a sort of hardware interrupt. For example, the above_limit could interrupt the instruction cycle in the program counter by forcing a jump here. If limited by number of instructions or wanting to keep the program less tied up, this could have been a valid solution.

Unfortunately, I was not able to figure out what I did wrong when connecting the hardware together. I expected I could at least get the ultrasonic sensor to function by only attaching this to the PMOD level shifter with power from the battery pack. However, I am very unsure whether this is acceptable. If the sensor board requires a 5v power supply, the battery pack might be supplying a little too much. The battery pack has 4xAA batteries which each output a 1.5v, possibly resulting in 6v combined in sequence. Otherwise, I can only assume I have done it correctly based on the description provided in the hardware section, and still do not get any results from running the FSMD or ASIP. I find this quite disappointing. Good test benches help simulate desired behavior, but it is something else to observe it physically. Sometimes things simply do not work out as anticipated.

Vivado and Vitis have had a few issues for me. Vivado keeps crashing now and then, so I must always make sure to save each file I work in as soon as I have made any significant change. Vitis will not start from the desktop shortcut, and requires that I open it via Vivado in the tools dropdown in the toolbar. I was not able to come across any suitable alternatives to these two, but once up and running, they are of course powerful tools tailored for this sort of usage.

X. CONCLUSION

This has been a very challenging project for both of us, as neither has any previous experience with embedded systems, FPGAs or hardware design. We have experience from software development within primarily Python and C#, respectively. While some core concepts of logical thinking is transferable, hardware design is still quite a different way of thinking and planning. This is especially true due to the fact that hardware is concurrent by default, while software is not. In software, you can create variables and loops with just a few lines, while the same would require multiple hardware components in the FSMD or specific instructions in correct order in the ASIP.

Since all software code is compiled down to at least assembly instructions similar to RISC-V, or even machine code, there is a great benefit to having observed how the ASIP works with its RISC-V architecture. This has provided a good insight into how an ALU or CPU executes instructions to perform high level software logic. While we did not get to spend much time on the software aspect of the project, it at least highlighted the power of C and C++ in terms of accessing memory directly in order to send signals to the pins. The programming languages mentioned at the start of this section typically do not require direct memory access.

Learning is achieved in the face of adversity, and is often characterized with a slow pace forward and multiple mistakes along the way. While the software and HLS approaches could have been somewhat easier for us, it felt more interesting and challenging to go for the two hardware approaches to get familiar with hardware design.

Overall, this has been a very rewarding learning experience with plenty of challenges along the way, and mistakes made. There is clearly still a lot to learn, but we have obtained a good basis for further study.

REFERENCES

- [1] Wikipedia, “Application-specific instruction set processor — Wikipedia, the free encyclopedia,” <http://en.wikipedia.org/w/index.php?title=Application-specific%20instruction%20set%20processor&oldid=1169497096>, 2023, [Online; accessed 08-December-2023].

APPENDIX
APPENDIX A

ChatGPT Prompt: How do I define a procedure in its own file?

Defining a VHDL procedure in its own file is typically done by creating a VHDL package. A package in VHDL is a collection of reusable declarations that can be shared across multiple design units (like architectures and entities). This is especially useful for procedures that you want to use in several parts of your project.

1) *Create a VHDL Package:* First, you'll create a VHDL package file ('.vhd' or '.vhd') where you'll define your procedure.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3
4 PACKAGE my_procedures_pkg IS
5
6     -- Procedure Declaration
7     PROCEDURE my_procedure(
8         SIGNAL some_signal : OUT STD_LOGIC_VECTOR;
9         constant_value : IN INTEGER
10    );
11
12 END PACKAGE my_procedures_pkg;
```

2) *Define the Procedure in the Package Body:* You then create a package body where the actual implementation of the procedure is written.

```
1 PACKAGE BODY my_procedures_pkg IS
2
3     PROCEDURE my_procedure(
4         SIGNAL some_signal : OUT STD_LOGIC_VECTOR;
5         constant_value : IN INTEGER
6     ) IS
7     BEGIN
8         -- Procedure implementation
9         -- Example: set some_signal based on constant_value
10        some_signal <= std_logic_vector(to_unsigned(constant_value, some_signal'LENGTH));
11    END PROCEDURE my_procedure;
12
13 END PACKAGE BODY my_procedures_pkg;
```

3) *Use the Procedure in Your Design:* To use this procedure in your design, you'll need to include the package in the VHDL file where you want to use it.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 USE work.my_procedures_pkg.ALL; -- Include the package
4
5 ENTITY my_design IS
6     -- Entity declaration
7 END ENTITY my_design;
8
9 ARCHITECTURE behavior OF my_design IS
10    SIGNAL my_signal : STD_LOGIC_VECTOR(7 DOWNTO 0);
11 BEGIN
12
13     -- Process or architecture body
14     PROCESS
15     BEGIN
16         -- Call the procedure from the package
17         my_procedure(my_signal, 255);
18         WAIT;
19     END PROCESS;
20
21 END ARCHITECTURE behavior;
```

4) *Conclusion:* By defining your procedure in a VHDL package, you create a modular, reusable piece of code that can be included and used in multiple parts of your VHDL project. This approach is highly beneficial for maintaining large projects and for sharing common functionality across multiple design units. Remember to compile both the package declaration and the package body in your project for them to be accessible in your design.

APPENDIX B

ChatGPT Prompt: In the piece of code I am looking at, the assignment below is done between the two original assignments. Does that matter? Are each line read one by one as in software development? mc_q <= std_logic_vector(r_reg);

A. Understanding VHDL Signal Assignments

In VHDL, the order of signal assignments in a process or architecture body can matter, but not in the same way as in traditional sequential programming languages like C or Python. VHDL is a hardware description language, describing hardware behavior in parallel, rather than specifying a sequence of operations to be executed one after the other.

B. Concurrent vs. Sequential Execution

- **Concurrent Execution:** In VHDL, outside of a process block (i.e., in the architecture body), assignments are concurrent, reflecting the parallel nature of hardware.
- **Sequential Execution in Processes:** Inside a process block, statements are executed sequentially. However, signal assignments within a process are scheduled to take effect at the end of the process block, at the next simulation cycle.
- **Scheduled Updates:** The order of assignments in a process can matter if the same signal is assigned multiple times.

C. Your Specific Case

For the code snippet you are referring to:

```
1 mc_q <= std_logic_vector(r_reg);
2 -- Other assignments...
3 r_next <= r_reg + 1 when r_reg < MOD_M else (others => '0');
4 max_tick <= '0' when r_reg < MOD_M else '1';
```

D. Behavior Analysis

- **Outside a Process:** If these assignments are outside a process block, they are concurrent and the order doesn't matter.
- **Inside a Process:** If these are inside a process block, the order can matter.

E. Conclusion

In VHDL, the concept of execution order is different from software programming. It's about scheduling updates to signals, reflecting the parallel operation of hardware.

APPENDIX C

ChatGPT Prompt: If this asip architecture has a pwm for reading a sensor and sends out a signal in 9 bits to control 4 motors, would a state machine be necessary?

A. Necessity of a State Machine in ASIP Architecture

The necessity of a state machine in your ASIP architecture, which includes PWM for reading a sensor and sending out 9-bit signals to control 4 motors, depends on the specifics of how these operations are carried out. Let's consider the scenarios where a state machine might be necessary:

B. Scenarios Where a State Machine Might Be Necessary

- 1) **Sequential Operations:** If controlling the motors or reading the sensor involves a sequence of steps that need to occur in a specific order, a state machine can manage these sequences. For example, if the motors need to be activated in a particular sequence or the sensor data needs to be processed in stages.
- 2) **Timing-Dependent Control:** If the operations have timing requirements, like waiting for a certain duration between reading the sensor and activating the motors, or generating PWM signals at specific intervals, a state machine can be useful to manage these timing dependencies.
- 3) **Complex Control Logic:** If the decision to send signals to the motors is based on complex logic involving the interpretation of sensor data, a state machine can help in organizing this logic into manageable states.
- 4) **Error Handling or Safety Checks:** If there are safety checks or error-handling routines that need to be executed in response to sensor data or during motor control, a state machine can help in structuring these routines.

C. Scenarios Where a State Machine Might Not Be Necessary

- 1) **Direct Control:** If the sensor reading and motor control are straightforward operations that do not require sequential steps or complex decision-making, you might not need a state machine. For instance, if the sensor reading directly translates to motor control signals without intermediate steps.
- 2) **Combinational Logic:** If the logic for reading the sensor and controlling the motors can be implemented as combinational logic (where the output is a direct function of the input without needing to remember previous states), a state machine may be redundant.

D. Conclusion

In summary, whether a state machine is necessary for your ASIP architecture depends on the complexity and nature of the operations involving the PWM sensor and motor control. If these operations involve sequences, timing dependencies, complex decision-making, or safety checks, a state machine could be very beneficial. However, if the operations are more direct and can be managed with combinational logic, you might not need a state machine.

In practical terms, the decision often comes down to the specific requirements of your application and the best way to manage the complexity and reliability of the system.

APPENDIX D

ChatGPT Prompt: Could you show an example of VHDL timer? I want it to count to an input limit value

A. VHDL Timer Implementation

Certainly! A VHDL timer that counts up to a specified limit value can be implemented using a simple counter. The counter increments on each clock cycle and resets either when it reaches the limit value or when a reset signal is asserted. Below is an example VHDL code for such a timer:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity timer is
6     port(
7         clk    : in  std_logic;                      -- Clock input
8         rst    : in  std_logic;                      -- Reset input (active high)
9         limit : in  std_logic_vector(31 downto 0); -- Limit value input
10        done   : out std_logic;                     -- Output signal, goes high when limit is reached
11    );
12 end entity timer;
13
14 architecture Behavioral of timer is
15     signal counter : unsigned(31 downto 0) := (others => '0');
16 begin
17     process(clk, rst)
18     begin
19         if rst = '1' then
20             counter <= (others => '0'); -- Reset the counter
21             done <= '0';
22         elsif rising_edge(clk) then
23             if counter = unsigned(limit) then
24                 counter <= (others => '0'); -- Reset the counter when limit is reached
25                 done <= '1'; -- Indicate that the limit is reached
26             else
27                 counter <= counter + 1;
28                 done <= '0';
29             end if;
30         end if;
31     end process;
32 end Behavioral;
```

B. Explanation

- **Entity Declaration:** The *timer* entity has four ports:

- *clk*: The clock input.
- *rst*: The reset input (active high).
- *limit*: The limit value up to which the counter will count. It's a 32-bit vector, allowing a wide range of values.
- *done*: An output signal that goes high when the limit is reached.

- **Architecture:** Inside the architecture, a 32-bit unsigned signal *counter* is used to count the clock cycles.

- **Process Block:** The process block is sensitive to *clk* and *rst*. On each rising edge of the clock, the counter is incremented until it equals the *limit* value. When the limit is reached, the *counter* is reset to 0, and the *done* signal is set high. If the *rst* signal is asserted, the counter is reset to 0.

This is a basic timer implementation. Depending on your specific requirements, you might want to adjust the width of the *limit* and *counter* signals, add enable control, or modify the reset behavior.

APPENDIX E

ChatGPT Prompt: If the echo sensor counter counts to 4096, how far is that in centimeters?

A. Distance Calculation for Echo Sensor

To determine the distance in centimeters corresponding to a count of 4096 from an echo sensor, we need to consider the speed of sound and the specifics of your timing mechanism. The speed of sound in air is approximately 343 meters per second (m/s), or 34300 centimeters per second (cm/s), at room temperature.

B. Calculation Steps

1) Determine the Time for 4096 Counts:

- If your clock period is 20 ns (as mentioned in a previous discussion), then the total time for 4096 counts is $4096 \times 20 \text{ ns}$.
- Time T in seconds = $4096 \times 20 \times 10^{-9} \text{ s}$.

2) Calculate the Total Distance Traveled by the Sound:

- Sound travels to the object and back in this time, so the actual distance to the object is half the total distance traveled by the sound.
- Total Distance D = Speed of Sound $S \times$ Time T .
- Since the sound travels to the object and back, the distance to the object is $D/2$.

3) Speed of Sound:

- $S = 34300 \text{ cm/s}$.

4) Distance Calculation:

- $T = 4096 \times 20 \times 10^{-9} \text{ seconds}$.
- $D = S \times T \text{ cm}$.
- Distance to Object = $D/2 \text{ cm}$.

C. Performing the Calculation

Substitute the values into the formulas:

- 1) Time $T = 4096 \times 20 \times 10^{-9} \text{ s}$.
- 2) Total Distance $D = 34300 \times T \text{ cm}$.
- 3) Distance to Object = $D/2 \text{ cm}$.

Let's calculate it.

The distance to the object, corresponding to a count of 4096 in your echo sensor setup, is approximately 1.40 centimeters. This means that for this count, the ultrasonic sound wave traveled a round trip distance of about 1.40 cm from the sensor to the object and back.

APPENDIX F

ChatGPT Prompt: Given this test bench, could you create an updated stim_process which will send echo signals with decreasing duration and then back to increasing, to simulate getting closer to an object then reversing?

A. Test Bench stim_process for Echo Sensor Simulation

Certainly! To create a test bench stim_process that simulates an echo sensor getting closer to an object and then moving away, we'll have the duration of the echo signals decrease and then increase. This will be reflected in the length of time the echo signal remains high.

B. Test Bench stim_process Structure

- 1) **Initial Setup:** Initialize necessary signals.
- 2) **Decreasing Echo Duration:** Start with a long echo duration and gradually decrease it over several cycles.
- 3) **Increasing Echo Duration:** After reaching the minimum duration, gradually increase the echo duration.

Let's write a basic VHDL stim_process to simulate this behavior:

```
1  stim_process: process
2      variable echo_duration: integer range 0 to MAX_ECHO_DURATION;
3  begin
4      -- Initialize
5      echo <= '0';
6      wait for 10 us;  -- Wait some time before starting the simulation
7
8      -- Decreasing Echo Duration (Object getting closer)
9      echo_duration := MAX_ECHO_DURATION;  -- Start with maximum duration
10     while echo_duration > MIN_ECHO_DURATION loop
11         echo <= '1';
12         wait for echo_duration * 1 us;  -- Echo signal high
13         echo <= '0';
14         wait for 100 ms - (echo_duration * 1 us);  -- Echo signal low
15         echo_duration := echo_duration - DECREMENT_VALUE;
16     end loop;
17
18     -- Increasing Echo Duration (Object moving away)
19     while echo_duration < MAX_ECHO_DURATION loop
20         echo <= '1';
21         wait for echo_duration * 1 us;
22         echo <= '0';
23         wait for 100 ms - (echo_duration * 1 us);
24         echo_duration := echo_duration + INCREMENT_VALUE;
25     end loop;
26
27     -- End simulation (optional)
28     wait;
29 end process stim_process;
```

C. Explanation

- **Parameters:** MAX_ECHO_DURATION, MIN_ECHO_DURATION, DECREMENT_VALUE, and INCREMENT_VALUE should be defined based on your test requirements.
- **Echo Duration:** The echo duration starts at its maximum, decreases to its minimum, and then increases back to its maximum.
- **Cycle Timing:** The `wait for 100 ms - (echo_duration * 1 us);` line ensures that the total time for each cycle (echo high and low) is approximately 100 ms, simulating the periodic nature of the sensor's operation.
- **Adjustments:** Adjust the INCREMENT_VALUE and DECREMENT_VALUE to control how quickly the echo duration changes.

Remember to tailor the values and the timing to fit the specifics of your system and the resolution of your simulation. This test bench will help you simulate an object approaching and then moving away from the ultrasonic sensor.

APPENDIX G

ChatGPT Prompt: From left to right, please provide these columns: addr (address number), value (the hexdecimal, pointer for L1, T1, etc), operation (for LD, ST), spec (for R1,85 etc), desc (for the last part of the comment e.g. "R1: Forward")

The rest of the prompt is omitted, but contains the defined instruction set in imem for the ASIP

Creating a LaTeX Table

To create a LaTeX table with the provided information, I'll structure it into six columns as specified: ‘addr‘, ‘value‘, ‘pointer‘, ‘operation‘, ‘spec‘, and ‘desc‘. Each row will correspond to the lines of code provided. Here's the LaTeX code for your table:

```

1 \begin{table}[h!]
2 \centering
3 \begin{tabular}{|c|c|c|c|c|l|}
4 \hline
5 \textbf{addr} & \textbf{value} & \textbf{pointer} & \textbf{operation} & \textbf{spec} & \textbf{desc}\\
6 \hline
7 00 & x"550080" & & LD & R1,85 & R1: Forward\\
8 \hline
9 01 & x"AA0100" & & LD & R2,170 & R2: Backward\\
10 02 & x"A50180" & & LD & R3,165 & R3: Turn left\\
11 03 & x"0E0200" & & LD & R4,14 & R4: MIN threshold for forward\\
12 04 & x"230280" & & LD & R5,35 & R5: MAX threshold for Backward\\
13 05 & x"008011" & T1 & ST & R4,TL & T1: Store MIN threshold in threshold register\\
14 06 & x"030012" & L1 & JAL & T2 & L1: Jump to T2 if above limit\\
15 \hline
16 07 & x"002010" & & ST & R1,OUT & Send R1 out via dr2\_dout -> dig\_out\\
17 \hline
18 08 & x"FE000E" & & J & L1 & & Jump to L1 to restart loop\\
19 \hline
20 09 & x"00A011" & T2 & ST & R5,TL & T2: Store MAX threshold in threshold register\\
21 10 & x"030013" & L2 & JBL & L3 & L2: Jump to T3 if below limit\\
22 \hline
23 11 & x"004010" & & ST & R2,OUT & Send R2 out via dr2\_dout -> dig\_out\\
24 \hline
25 12 & x"FE000E" & & J & L2 & & Jump to L2 to restart loop\\
26 13 & x"000014" & T3 & TS & & T3: Timer start\\
27 \hline
28 14 & x"F70015" & L3 & JTD & T1 & & L3: Jump to T1 if timer is done\\
29 \hline
30 15 & x"006010" & & ST & R3,OUT & Send R3 out via dr2\_dout -> dig\_out\\
31 \hline
32 16 & x"FE000E" & & J & L3 & & Jump to L3 to restart loop\\
33 \hline
34 \end{tabular}
35 \caption{Your table title here}
36 \label{your-label-here}
37 \end{table}

```

APPENDIX H

A.

Appendix B

Appendix

Some sample question that we asked Chat GPT

B.

How can I use for loop??

ChatGPT

In VHDL, a for loop is typically used within a process or a generate statement. It's important to remember that in hardware description, loops are used to generate repetitive hardware structures, not to create software-like iterative behavior at runtime. Here's how you can use a for loop in VHDL:

C.

Please explain alu.vhd?

ChatGPT The VHDL code snippet you provided is a modified version of Listing 4.5, which describes an Arithmetic Logic Unit (ALU) entity. The ALU is a fundamental component in CPUs and other digital systems, responsible for performing arithmetic and logical operations. Let's break down the key aspects of this code:

Library and Use Declarations: The code uses the IEEE std_logic_1164 package for digital logic representations and numeric_std for arithmetic operations.

Entity Declaration: The entity alu is defined with two generic parameters: $OPCODE_W$, $IDTH$ and $ALUDAT_Aw$, $IDTH$, which determine

Architecture Declaration: The arch architecture of the alu entity contains signals for internal calculations ($valin_h[i], valin_l[0], valout$), all of type `unsigned`.

Operational Logic:

ALU Output: The final result of the operation (valout) is converted back to `std::logic_vector` and assigned to `alu_dout`.

Functionality: The ALU can perform a variety of operations depending on the opcode. This includes basic arithmetic operations like addition and subtraction, as well as logical operations like bitwise AND and OR. The design is versatile and supports a range of instructions, making it suitable for use in a simple microprocessor or microcontroller.

This type of ALU is commonly found in educational or experimental CPU designs. Its modular and parameterized design allows it to be easily integrated into larger systems and adjusted for different data widths. The inclusion of both immediate and register-based operations gives it a good degree of flexibility for various computational tasks.

I used a custom based ChatGPT with this link VHDL Visinoary

D.

Please comment pc.vhd?

ChatGPT Only part of the answers in is provide

```

1  -- Import standard logic definitions from the IEEE library
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5
6  -- Define a new entity named 'pc'
7  entity pc is
8      -- Generic parameter to define the data width of the counter
9      generic(
10          PCDATA_WIDTH: integer := 8 -- Default width is 8 bits
11      );
12      -- Define the input and output ports for the entity
13      port(
14          clk: in std_logic;                      -- Clock input signal
15          rst: in std_logic;                     -- Reset input signal
16          reg_d: in std_logic_vector(PCDATA_WIDTH-1 downto 0); -- Data input (vector)
17          reg_q: out std_logic_vector(PCDATA_WIDTH-1 downto 0) -- Data output (vector)
18      );
19  end pc;
20
21  -- Architecture definition of the 'pc' entity

```

```
22 architecture arch of pc is
23 begin
24     -- Define a sequential process triggered by changes in 'clk' and 'rst'
25     process(clk, rst)
26 begin
27     -- If reset is high ('1'), then clear the counter
28     if (rst = '1') then
29         reg_q <= (others => '0'); -- Set all bits of the counter to zero
30     -- If a rising edge on the clock signal is detected
31     elsif rising_edge(clk) then
32         reg_q <= reg_d; -- Transfer data from input 'reg_d' to output 'reg_q'
33     end if;
34 end process;
35 end arch;
```