Embedded Systems Laboratory Assignments

University of Twente, Embedded Systems Laboratory

Glen te Hofsté (s2204320) Kilian van Berlo (s2613352)

March 12, 2022

1. Introduction

In a car, what do the engine, the meters, the air bags and the audio system have in common? They are all embedded systems in the car. Embedded systems are all over the place, not only in cars but basically everywhere you look, even lights or doorbells can be embedded systems these days. These systems are tunable and configurable via software or firmware which gives a lot of flexibility in order to easier build, maintain and update such a system. What makes an embedded system particularly challenging is that these systems have to interact with the real world, which leads to time performance constraints that have to be taken into account. But not only constraints in the time performance, also constraints in the resources like the available processors, memory or other programmable hardware have to be considered. Since embedded computer system parts often control physical systems, also reliability and safety are of utmost importance to most of these systems.

With all the constraints and requirements an embedded system has to take in to account, it becomes difficult to design these systems. Therefore it is of utmost importance to get familiar with the nature of embedded systems software, the fairly complex workflow, and the points of attention when producing optimal software or firmware. In order to get a grip on this complexity and to structure the design flow properly, a V-model is used (figure 1) in this project. The design flow of a V-model ensures top-down design efforts and bottom-up testing and composing of parts, all in an modern iterative design approach.

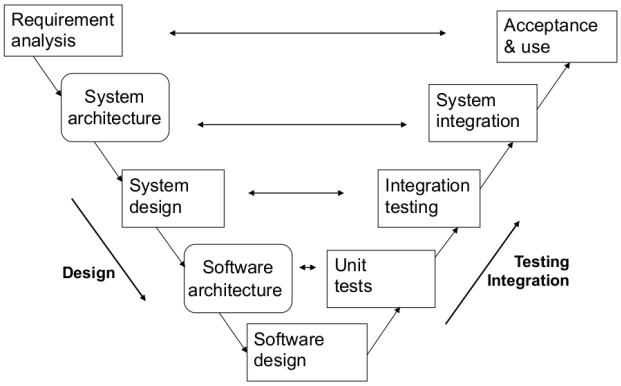

Figure 1: Design Flow V-model

In this project the embedded system design problem dealt with is about visual servoing, where data has to be interpreted from a camera and is then used to steer the motors of that same camera. The report is built in such a way that it resembles the design flow of the V-model. In chapter 2, first the requirement analysis is discussed after which in chapter 3 the system architecture is laid out in more detail. Chapter 4 pinpoints the system's design and its design space exploration followed by chapter 5 covering the implementation and testing of all subsystems. Then in chapter 6 the whole system is being integrated and tested followed by a discussion and a conclusion on the project in chapter 7. Eventually, in chapter 8 the learning points on the methods / Design Space Exploration approach of the course are discussed.

2. Requirements

Important before one starts designing an embedded system is to create a thorough understanding of what the system is required to do and when the system is considered to be accepted and ready for use. In the case of this Embedded Systems Laboratory the specific goal of the system is visual servoing. The way this is shown in this specific test case is with a camera that recognizes a post-it and consecutively follows it. When this eventually works then it can be concluded that the acceptance and use tests are passed and the system has been successfully implemented.

In order to smoothen the design process it is important to, next to having a clear goal, know what kind of hardware, software and tools one has at his/her disposal. All of this is already given in the project description, however they are shortly discussed here for a complete overview of the project. For software it is fairly simple since Linux is used both on the development PC and the Overo Fire. Also only few tools are used, which are mainly Altera Quartus and Modelsim for configware and 20-sim for control law design. As opposed to the software and tools, the hardware components available for the project are plenty. Which components are going to be used is yet to be determined by Design Space Exploration, however it is good to know what is available. Hardware that is available is:

- Altera DE0-NANO with Altera cyclone IV FPGA
- Gumstix Overo Fire with Arm Cortex A8 processor
- Ramstix (Overo FireStorm & FPGA on one board)
- Logitech C250 webcam
- Maxon DC motors
- Motor amplifier (H-Bridge)
- Encoder (angle sensor)

Knowing what the goal is and knowing what is available is enough information to safely continue to the next step, namely the system architecture.

3. System architecture

For this stage all the various components that the system needs to function properly are mapped onto the architecture (figure 2). Keep in mind that this architecture is fairly high-level, meaning it is not yet considered which of the available hardware components are eventually used and on which

hardware component each subsystem of the system is exactly implemented. These more detailed choices are made in the next step, the Design Space Exploration.

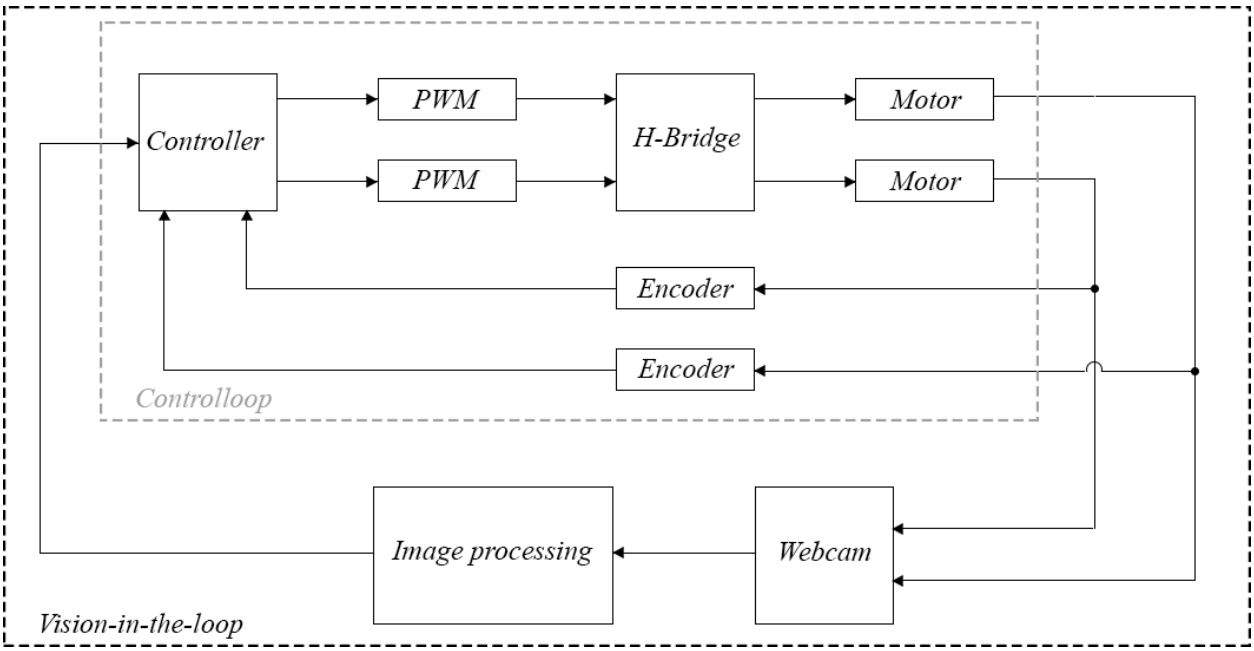

Figure 2: Architecture for the visual servoing system

In the architecture two loops can be identified that help in determining the movements of the camera, namely the control loop and a vision-in-the-loop loop. The control loop gives direct feedback from the angle of the motor whereas the vision-in-the-loop loop provides feedback through the images that the camera processes. The vision-in-the-loop loop is fairly short since it only consists of the camera and the image processing software which processes the image data that the camera collects. This software then calculates the set point desired for the camera and sends this data to the controller, which is part of the control loop.

Now the control loop is more extensive with more components being part of the loop. Components that are part of the control loop are a controller, pulse-width modulators (PWMs), an H-bridge, motors that drive the camera and quadrature encoders. These components are set up as follows: when the controller receives feedback data it processes it to produce control signals for the speed and direction of the motors. These signals are then fed to the PWM modules in which pulse width modulated signals are generated. These modulated signals are then transmitted to the motor via the H-bridge. As the motors rotate the camera (vision-in-the-loop), the change in position is being observed by the quadrature encoders and this change is on its turn being fed back to the controller. Together with the results of the image processing, this data is then used to calculate the next position and determine the new control signals.

4. Design Space Exploration

Design space exploration (DSE) in engineering refers to the exploring of design alternatives prior to implementation. It examines multiple candidates for specific components, focusing on aspects like performance and accuracy while also taking metrics such as complexity and development time into account. These criteria can then be used to decide on which of the alternatives is most suitable for

the design. Given the system architecture in Chapter 3, five subsystems are identified for which DSE has to be performed. These subsystems are the Quadrature Encoder, Pulse Width Modulator, motor controller, image processing and the way of communication. Each of these subsystems will be subjected to the DSE process, mainly focusing on five specific criterion:

- Development time:
  The required amount of time to implement each subsystem;
- Execution time:
  The speed of execution of the implemented subsystem;
- Accuracy:
  How precise and accurate the subsystem functions;
- Complexity:
  The difficulty to implement each solution;
- Resource usage:
  The required amount of system resources for the subsystem.

Through the DSE, the suitability of each subsystem for the three different available platforms gets compared:

- FPGA:
  An FPGA platform, being either on the DE0-NANO or the Ramstrix;
- NIOS:
  A NIOS (II) 32-bit embedded-processor architecture running on an FPGA platform;
- Gumstix:
  A SoC running a Linux operating system.

One exception on the comparison of these platforms is for the communication subsystem since this is not implemented on one platform but rather facilitates the data exchange between platforms. Therefore comparison for communication is between UART (Universal asynchronous receiver-transmitter) and GPMC (General Purpose Memory Controller) communication instead. For communication also the criteria are adjusted a bit since for the communication the accuracy criterion is taken out and overhead is used for evaluation instead.

After the exploration of each design subspace a total rating is calculated based on the points for each criterion, multiplied by the weight that is given to that criterion. The points attributed to each option and the weight of each criterion is given in the tables of each subsystem DSE. Eventually the total points are calculated and a rank from 1 to 3 is given to each possibility. In total, as already mentioned, five of the most important criteria are taken into account to perform the DSE on. Three of these criteria (execution time, accuracy and resource usage) are regarded as equally important and therefore all received a common weight of 1. Next to these three criteria, two other criteria are regarded as being slightly more important. During this project the most important criterion is defined to be the required development time with a weight of 2. This is because of the limited amount of lab sessions over a short period of time, making it is crucial to not remain working on a component for too long. This also takes into account that the debugging, evaluation and testing should be rather easy to do. The other majorly important criterion is the complexity of the subsystem, which receives a weight of 2. There are a lot of ways in which a component can be

implemented but due to the time constraints it is important to keep the implementation simple and easy to work with. Complex components can allow more functionality which can allow re-usage in future designs, however since this is not necessary within the scope of this project the component are mainly required to be simple and stable.

The overall DSE ranking results can be seen in table 1, where one can observe that for the Quadrature Encoder and PWM components the FPGA is the best option. The NIOS is more suitable for the motor controller, while for the image processing the Gumstix turns out to be the winner. The result of the communication DSE is not shown in the table but from the exploration it is concluded that in this case the UART is more beneficial than the GPMC. Each component will now be examined individually to elaborate on the design choices made.

Table 1: Design Space Exploration for the system

System	**Implementation**		
	FPGA	NIOS	Gumstix
Quadrature Encoder	1	2	3
Pulse Width Modulator	1	3	2
Motor Controller	3	1	2
Image Processing	3	2	1

4A. Quadrature encoder

The quadrature encoder is used to determine the position of the motor. It does this by outputting pulses in a specific sequence on which the direction and speed of rotation can be determined. The used encoder (Broadcom HEDL-5540#A11) is a high-quality encoder which generates 500 pulses per rotation. Since the output of this specific encoder is filtered, a minimal amount of noise is expected, making it easy to process the data. The results of the DSE for the quadrature encoder is shown in table 2.

Table 2: Design Space Exploration for the Quadrature Encoder

Criterion	**Weight**	**Implementation**		
		FPGA	NIOS	Gumstix
Development time	2	-	+	+
Execution time	1	++	+/-	-
Accuracy	1	++	+/-	-
Complexity	2	+/-	+	+
Resource usage	1	++	-	-
Total rating		4	3	1
Rank		1	2	3

In table 2 the first criterion dealt with is the development time (including testing). The time for development is quite high in VHDL as compared to the other platforms. Not only is development in C++ easier for us, also debugging VHDL code is difficult and costs tremendous amounts of time

compared to the alternative. On the NIOS and the Gumstix debugging is simpler, given that GDB can be used.

Next up, the execution time, which on the FPGA is way faster. It can be designed to not miss any edges even at high rotation speeds, hereby also resulting in a very high accuracy. Therefore, both execution time and accuracy are highly in favour of the use of the FPGA. On the NIOS and Gumstix to gain the highest possible accuracy the system has to use interrupts, which in case of the encoder will cause an interrupt to be triggered in very rapid succession. This storm of interrupt firings is undesirable on the NIOS given the interference with the main program loop, but especially the Gumstix since it will cause a lot of interference with the OS.

The complexity of the VHDL code is quite high which is less desirable. An internal state machine needs to keep track of the encoder's output state while comparing it with previous states so a reliable counter increment or decrement can be made which is slightly easier to do in C++ code on the NIOS and Gumstix. On the contrary, the resource usage can be low on the FPGA when designed to fit only this specific purpose without the need for additional components. Therefore the FPGA becomes again a lot more interesting as a platform. So interesting even that, partly due to the resource usage, it is determined that the FPGA is the best platform to implement the Quadrature Encoder on.

4B. Pulse Width Modulator

The Pulse Width Modulator (PWM) subsystem is used to control the motor while also supporting functionality to set the motor directions. This component should have a configurable frequency and duty cycle while also being able to set the clockwise or counterclockwise rotation of a motor. To ensure the correct speed of each motor, it is therefore important that the frequency and duty cycle of each PWM are accurate. The DSE of this component can be seen in table 3 and is elaborated on next.

Table 3: Design Space Exploration for the Pulse Width Modulator

Criterion	**Weight**	**Implementation**		
		FPGA	NIOS	Gumstix
Development time	2	+/-	+	+
Execution time	1	++	+/-	++
Accuracy	1	++	+/-	++
Complexity	2	+	-	-
Resource usage	1	++	-	++
Total rating		8	-1	6
Rank		1	3	2

Similar to the Quadrature Encoder component the development time is fairly high on the FPGA due to the debugging difficulty with VHDL. The easiest platform to implement this subsytem is the Gumstix given it has PWM peripherals integrated. Also for NIOS it is possible to achieve a PWM, although this has to be achieved by using timers in that case.

When looking at the execution time there is again a lot of overlap compared to the quadrature encoder. The PWM module can be rapid when it is properly implemented in VHDL. When implemented in VHDL the subsystem can run completely independent of other hardware components besides of the reference clock. This is also somewhat the case for the Gumstix since it also uses

independent hardware, however this does have the downside that it has to be controlled from an OS.

When evaluating the accuracy it can be noted that the PWM output pulse can be accurately timed depending on the precision of the clock input frequency on the FPGA. Since the clock input frequency is 50MHz and PWM frequencies are often in the order of a KHz the clock input frequency is more than sufficient. This same accuracy holds for the Gumstix architecture which houses dedicated PWM hardware.

With regards to the complexity it is easy to pinpoint a clear winner. Implementing the PWM module is moderately complex on the FPGA given the amount of clock pulses needs to be very strictly monitored. The NIOS allows PWM by the usage of timer interrupts which is tricky to use in combination with an output pin. Whereas the Gumstix requires its internal hardware to be configured which requires deeper knowledge on its internal architecture. For resource usage nearly the same story as with the quadrature encoder holds since, the resource usage is low on the FPGA and higher on the NIOS. With regards to the Gumstix it is a bit different however, since now also the Gumstix makes use dedicated hardware without the need for additional components. Based on the DSE of this subsystem both FPGA and Gumstix turn out to be interesting possibilities. However, with the Gumstix implementation being considerably more complex it is determined that the FPGA is the best platform to implement the PWM.

4C. Motor Controller

The motor controller is designed as a 20-sim model which is exported as C++ code. The motor controller ensures that the DC motor does not overshoot or undershoot from its position when moving to a specified target and moves the motor to a direction based on its current position. The calculations which are performed which determine the output current for the motor are executed periodically which makes it important that this period is correct in order to improve the motor's accuracy. Also, the more calculations that can be performed within the total execution from start to finish determine the accuracy integration which is performed. The DSE of this component can be seen in table 4.

Table 4: Design Space Exploration for the 20-sim controller

Criterion	**Weight**	**Implementation**		
		FPGA	NIOS	Gumstix
Development time	2	- -	+	+/-
Execution time	1	++	+/-	+/-
Accuracy	1	++	+	- -
Complexity	2	- -	+	++
Resource usage	1	++	- -	+
Total rating		-2	3	3
Rank		3	1	2

Again starting with the development time. The required time to implement a 20-sim motor controller on an FPGA would be high since lots of code is already provided in C or C++, making it very undesirable for this project. It is, however, fairly easy to implement on the Gumstix since it is fast with a lot of resources. The NIOS solution on its turn requires an additional floating point unit to allow the quick execution of floating point operations. This logically requires a bit more

development time. For a Gumstix implementation it has to be noted that it is more difficult to achieve the connection to the Quadrature Encoder and PWM module, requiring a communication interface between the NIOS which facilitates these components. Taking this into account makes the Gumstix score lower on development time than might be expected.

When taking into account the execution time, it is clear that this will be the fastest on the target which has dedicated hardware to perform the calculations. Since this can be done rapidly in VHDL and on the Gumstix due to its FPU pipeline it makes sense that these two options receive the best score. An important note is that the execution time on the Gumstix is dependent on the amount of resources allocated to it by the operating system, resulting in different execution times over each period. This inconsistency is undesired as the 20-sim model requires its calculations to be well timed, resulting in lower score for the Gumstix. Looking at the NIOS it is known that its speed is bottle-necked by the speed of the additional FPU unit, which is slower than the Gumstix due to the limited resources dedicated to the pipeline.

The accuracy is dependent on the reliable timing of the periodic calculations and the floating point precision. The FPGA can be timed with great accuracy while being able to perform double precision floating point calculations by using additional hardware. The NIOS can also be accurately timed by using timer interrupts. These interrupts can be called frequently, while also being able to quickly retrieve the current position from the Quadrature encoder hardware. The FPU in the NIOS is limited to only supporting single-precision floating point operations which is a limitation of the platform. The Gumstix does allow double precision floating point operations but is not able to performed accurately timed operations due to the fact that the code is ran on an operating system. The current position of the motor also has to be retrieved by using a communication interface to the Quadrature Encoder, introducing additional delays which decreases accuracy even more.

When looking at the complexity of the implementation, it is clear that this is the lowest on the NIOS, since it already has access to both the Quadrature Encoder and PWM components while allowing simple timer interrupts to be used. The addition of the FPU unit does require a bit more work to implement due to the a more difficult NIOS core design. Fitting the large library into the limited amount of memory available for the NIOS is somewhat challenging. Although after this is correctly done, all floating point operations are automatically performed by using the FPU pipeline. The Gumstix implementation is also quite easy due to it not being constraint by resources. The FPGA is by far the most complex and is therefore undesirable.

The last criterion, resource usage, seems to be high on the NIOS. The addition of the FPU to the NIOS core requires a lot of hardware and memory. On the other hand, the Gumstix has a lot of resources available, meaning that a large code-base is not a problem. The FPGA will likely use the lowest amount of resources when the motor controller is fully implemented in hardware.

Based on the DSE of the motor controller subsystem it is determined that the NIOS and Gumstix are both equally suitable for implementing the motor controller. Through this analysis and by looking at the other subsystems it is decided to rank the NIOS as the number 1 option given the better communication possibilities it presents. The reasoning behind this is that through utilizing the NIOS the speed of the control loop can be kept high by the direct access to the hardware components and the usage of only single floating point variables.

4D. Image processing

Image processing is required to track an object in front of the webcam, which on its turn is attached to the motors. The motors can turn the camera over the x and y axes, allowing it to follow an object within its field of view. For this DSE a combination of Gstreamer and OpenCV is examined, where Gstreamer handles the video stream coming from the webcam and OpenCV runs the object

recognition algorithm. The results of this DSE are shown in table 5.

Table 5: Design Space Exploration for the image processing

Criterion	**Weight**	**Implementation**		
		FPGA	NIOS	Gumstix
Development time	2	- -	- -	++
Execution time	1	++	- -	+
Accuracy	1	++	- -	+
Complexity	2	- -	- -	++
Resource usage	1	+	- -	-
Total rating		-3	-14	9
Rank		3	2	1

Given the time constraints with this project and since image processing is clearly faster to develop in C(++) than in VHDL, it is reasonable to assume that developing this on an FPGA does not make sense. Image processing is known to be very fast but also very difficult on an FPGA, meaning that the development time on this platform would be far too long. The Gumstix runs a Linux based operating system, on which Gstreamer and OpenCV run natively making this the best option looking at development time.

Despite the huge amount of development time, the execution time on the FPGA would actually be the fastest option, if a developer would be able to implement it on this target. Gstreamer and OpenCV have been designed to run on target such as the Gumstix, meaning that good performance can also be expected from this target. In this case the accuracy overlaps a bit with execution time, it will namely be the highest on the fastest target, so with the lowest execution time. The most accurate would thus be the FPGA followed by the Gumstix.

The complexity of implementing this on an FPGA is huge, as already stated before, while it is also very difficult to pull off on the NIOS core due to the limited amount of resources. The Gumstix is therefore by far the easiest as it only needs to be tuned for the camera configuration.

Something again in favour of the FPGA would be its resource usage given it does not use a lot if it is directly implemented on the hardware. Of course relative to other options. The NIOS would likely not have enough to facilitate the large libraries of Gstreamer and OpenCV and also on the Gumstix it will use a lot of resources. However, this is acceptable if it is the primary job of the Gumstix to only run the image processing part. Even when other things have to be run on the Gumstix it still comes out as the preferred approach, meaning that this platform is by far the best option for this application according to the results of the DSE.

4E. Communication

When multiple platforms are used together they need to have some way to communicate, based on previous DSE results which selected the NIOS as the ideal platform for the motor controller the DSE to aid with the selection of a communication interface can be performed. Since it is determined that the controlling is done by a NIOS core which runs on an FPGA it has to be determined how the NIOS has to receive direction commands. The 20-sim model requires a position it should move towards in radians. The controller should carry out the commands as received over a communication interface while providing feedback about the actual current position of the motor. The commands

on where the controller should move the motor to will be received from the Gumstix. The DE0-NANO + Gumstix hardware configuration is different in this regard compared to the Ramstix configuration. The DE0-NANO + Gumstix setup only supports a single (serial) UART interface while the Ramstix board has support for a bus which can send 16 bits up and 16 bits down in parallel. Both communication interfaces are compared in table 6. Note that accuracy is not really applicable here, so instead an overhead criterion is used.

Table 6: Design Space Exploration for the communication

Criterion	**Weight**	**Implementation**	
		UART	GPMC
Development time	2	++	-
Execution time	1	-	++
Overhead	1	+/-	+/-
Complexity	2	++	- -
Resource usage	1	+/-	++
Total rating		7	-2
Rank		1	2

The development time required to implement a UART interface compared to GPMC is fairly low. A UART peripheral can be easily added to the NIOS, allowing interrupts and data buffers to be used. It is true that for the GPMC interface there already is example code available which decreases development time, but it still requires a new component to make it work with the NIOS.

The execution time of the UART takes longer than on the GPMC. When a word of 16-bits is send over the GPMC bus this can be transmitted in one go, while the UART has to send each bit sequentially. The execution time does, however, not matter that much, since the commands which are sent to the motor controller are soft real-time.

The overhead of both systems is different. For the GPMC you are always required to send 16 bits even though 8 bits might be enough. This makes the GPMC very inflexible. The UART bus only requires two wires for RX and TX but does require more time to transmit the same amount of data, requiring more time overhead. With regards to complexity, however, there is a clear winner: the UART is by far the best. This is because it is available as a component for the NIOS it is very easy to add to the system. On the other hand, UART uses more resources when used on the NIOS compared to the GPMC interface. This is partly caused by the buffers and interrupts which are available in the NIOS component.

Based on the DSE it is determined that the UART communication interface is the best platform to add to the rest of the defined architecture. This mostly has to do with it low development time, acceptable execution time and low complexity. The NIOS had a sufficient amount of resources available to facilitate this component making it an easy and effective option.

5. Subsystems implementation and testing

Based on results of the Design Space Exploration a demonstrator is created of the seemingly most promising choice. Before the entire system can be realised, first all subsystems are implemented and tested separately. In this section all subsystems are discussed.

5A. Quadrature encoder

A quadrature encoder is an incremental encoder with two out-of-phase outputs. Due to these two outputs the encoder is able to detect the angular displacement. For example, if the motor is at its initial state (00, see figure 3) and there is a clockwise rotation, then the quadrature encoder will first enter state 01, then 11, followed by 10 after eventually returning to output 00. It will follow this loop until the movement is no longer in a clockwise direction. Whenever the motor rotates counterclockwise the state transitions are reversed, going from 00 to 10, 11 and 01 consecutively, eventually entering state 00 again. The state transitions are visualised in the state transition diagram of the quadrature encoder in figure 4. These state transitions are then detected and processed by a decoder. When the decoder detects a change in state, it either increases or decreases (depending on direction) a counter by one, as also shown in figure 4. This is done in order to be able to keep track of the angular displacement of the motors. Since the quadrature encoder is implemented on the FPGA the code is written in VHDL (Appendix A).

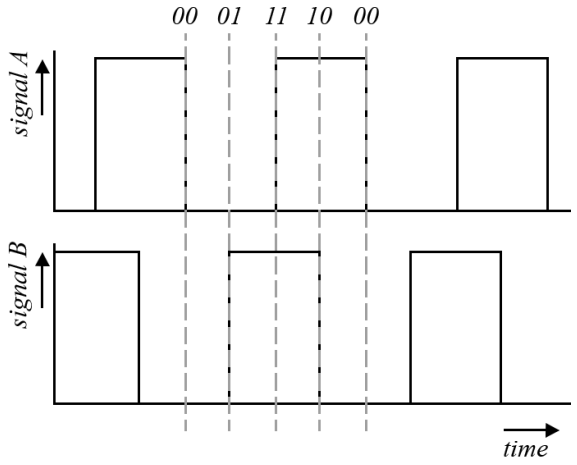

Figure 3: Different states of the quadrature encoder

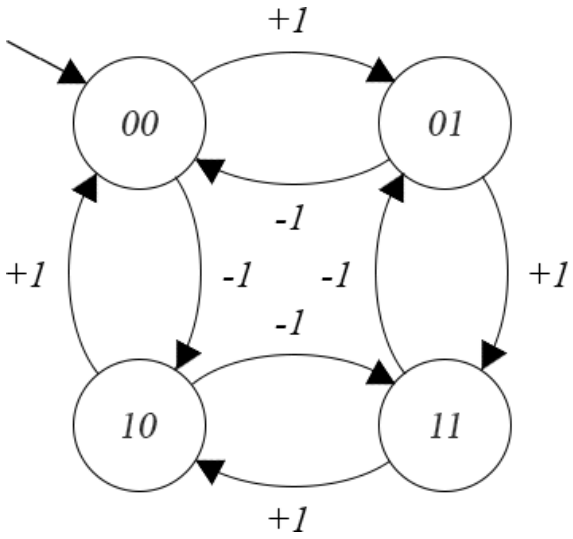

Figure 4: State transition diagram of the quadrature encoder

In order to test the quadrature encoder, a VHDL testbench is created to test the Device Under Test (DUT) in Modelsim. The exact test setup is shown in figure 5 where a testbench is used as the test environment to connect a testset with the quadrature encoder. The structural description in VHDL of the above test environment can be found in the zip folder under the folder quadrature_encoder. The results obtained via simulation in Modelsim can be seen in figure 6, where one can observe that the quadrature encoder indeed works as described above.

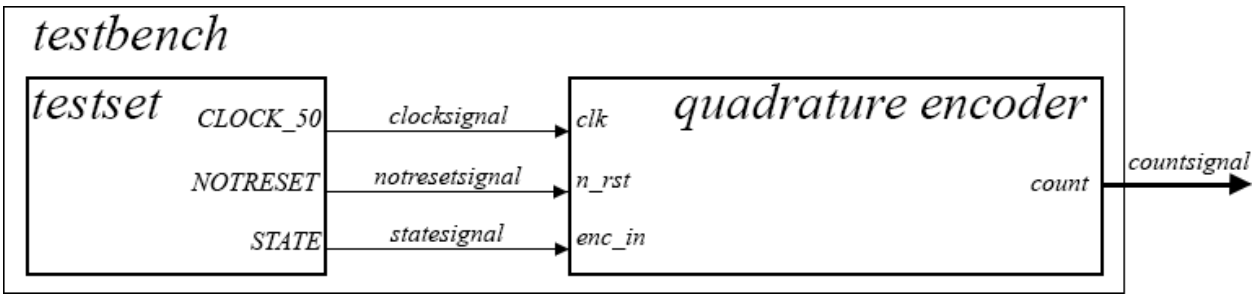

Figure 5: Test environment for the quadrature encoder

Figure 6: Simulation results of the quadrature encoder

Everything on the tests looked good and when testing the real quadrature encoder also no strange results showed up. One thing considered beforehand as a possible problem was a probable bounce in the signal that the quadrature encoder processes, making the output counter value erroneous at times. Despite some tests on the setup there was no sign of bounce affecting the performance of the encoder (example in figure 7), making the implementation a bit easier. In the zip folder however, a debouncer IP is still shown. This is because when testing it at home with an improvised setup it did show a lot of bounce. Since it is developed and given it has a very small hardware footprint and does not form any kind of negative impact on the signal it is decided to keep the IP in the documentation, as an extra possibility.

Based on the tests, the counter values for the tilt turned out to be ranging between -233 and 160 (393 angular displacements in total) for a total of 65.47 degrees and the counter values for the pan turned out to be ranging between -2480 and 2043 (4523 angular displacements in total) for a total of 331.28 degrees.

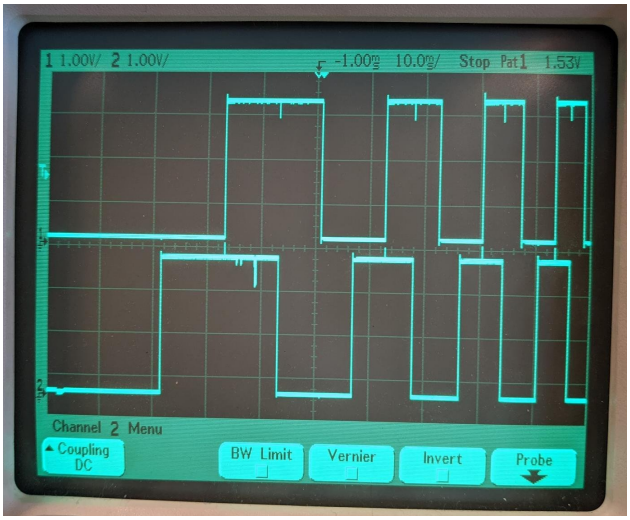

Figure 7: No substantial bounce detected on the signal

5B. PWM

The Pulse Width Modulator is a module that drives a motor with the desired direction and speed. Given the period for each cycle, the length of the duty cycle and the desired direction, the pulse width modulator outputs the direction and a pwm_out signal driving the activation of the motor. While the output direction is directly copied from the input direction that is given, the pwm_out signal is not directly copied from an input. For pwm_out it holds that the value is high for a certain percentage of each period. This percentage is defined based on the duration of the duty cycle as a percentage of the given period. The code written for the pulse width modulator is in VHDL since it is implemented on the FPGA and can be found in Appendix B.

To test the pulse width modulator, the same approach as for the quadrature encoder is applied. This means that a VHDL testbench is created to test the Device Under Test (DUT) in Modelsim. The exact test setup is shown in figure 8 where a testbench is used as the test environment to connect a testset with the pulse width modulator. The structural description in VHDL of the above test environment can be found in the zip folder under the folder pwm_module. The results obtained via simulation in Modelsim can be seen in figure 9, where one can observe that the pulse width modulator indeed works as described above.

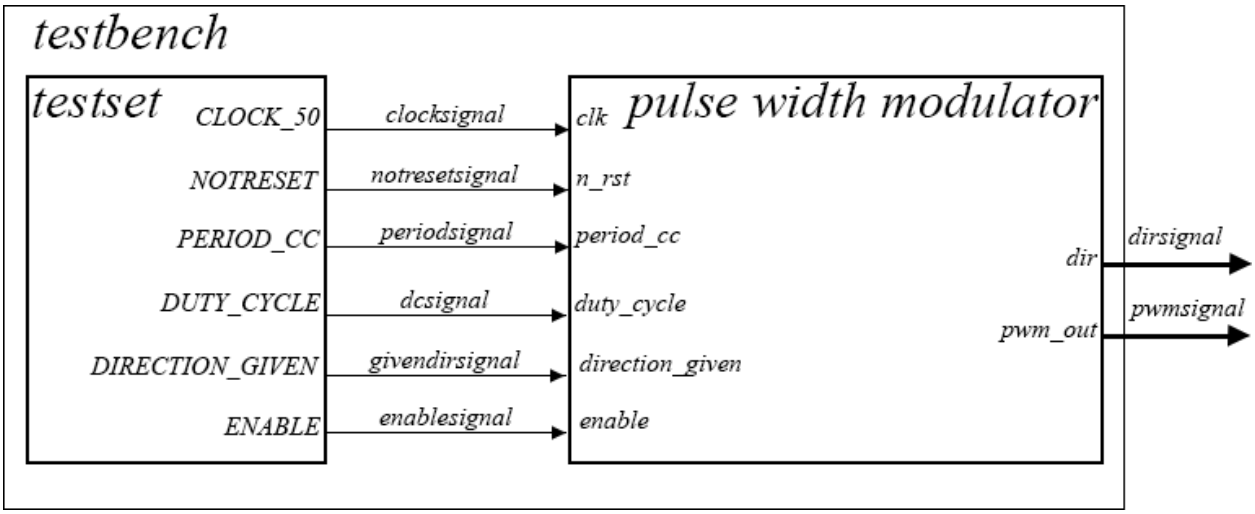

Figure 8: Test environment for the pulse width modulator

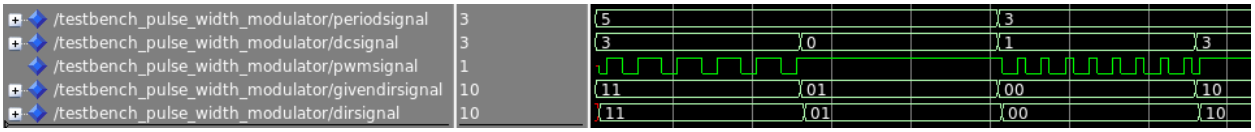

Figure 9: Simulation results of the pulse width modulator

5C. Motor controller

The motor controller components based on the 20-sim model are used to smoothly move the DC motors to their desired position. The block diagram of the 20-sim model can be seen in figure 10 where one can also see that both motors have their own model which can be configured and simulated. The parts of interest in this block diagram are the "PositionControllerPan" (figure 11) and the "PositionControllerTilt" (figure 12).

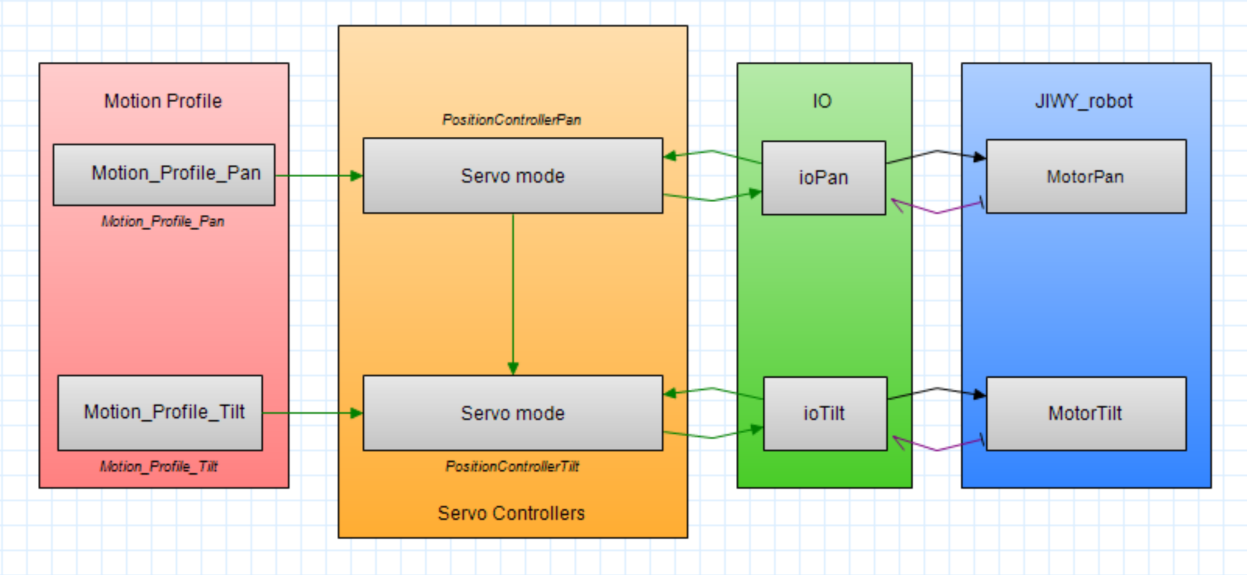

Figure 10: 20-sim model block diagram

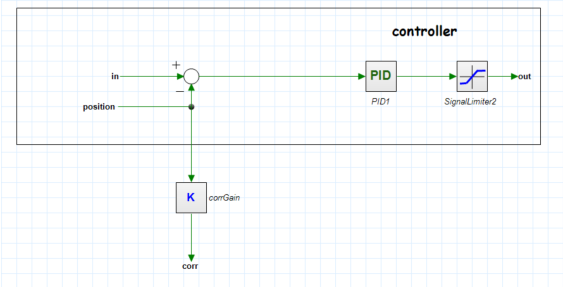

Figure 11: 20-sim pan controller

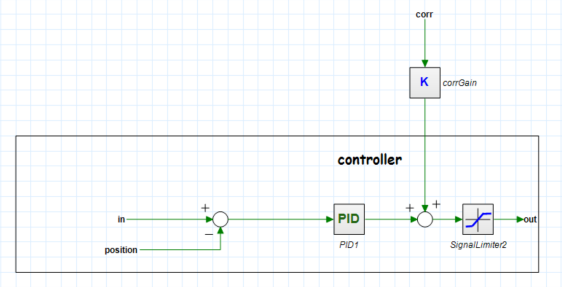

Figure 12: 20-sim tilt controller

The "in" signal of both models is the position where the motor should move towards (in radians). The "position" signal of both models is the actual current position of the motor as determined by the quadrature encoder. The "corr" signal for both models is left unused. In the pan controller the "corr" signal is an output which is connected to the "corr" signal of the tilt controller as an input. There was insufficient time to properly test with these signals, so they were not investigated any further during the lab sessions. The "out" signal is used to control the current of the motor. It's value is between -0.99 to 0.99 with a negative number meaning a counterclockwise rotation while a positive number is a clockwise rotation. Zero speed means that the motor should be standing still.

Several conversions have to made to allow the motor controllers to interface with the hardware. The quadrature encoder simply outputs signed integer which has to be converted into a motor position in radians. Therefore, both motors need to have an origin position, which will be used as a reference point and which will be said to be 0 radians. From that point onwards, every quadrature encoder step over the axis can be converted to a position in radians with respect to the reference point. The output from the motor controller has to be converted to represent the speed and direction of the motor. The speed can be configured by adjusting the duty cycle of the PWM signal. This duty cycle will be between 0 to 99% of the period. The duty cycle is thus the absolute number of the "out" signal from the motor controllers. When the "out" signal is negative, the motor is configured to rotate in the counterclockwise direction, while a positive number configures

the motor to rotate in the clockwise direction.

To determine the reference point, the controller performs a homing procedure where it moves the camera to the bottom left corner at a very low current. When the motor cannot move any further due to the boundary of the movable area being reached, the current is shut off and the encoder value at that point is set as the origin position of 0 radians for both axis. After this outer point is found, the motor moves to the centre of both axis after which it starts waiting for commands over the communication interface.

The motor controller model is configured to perform a calculation every 100 microseconds by using timer interrupts, allowing a hundred times more integration steps compared to the default setup of every 10 milliseconds. A single timer interrupt is used to trigger the calculations for both motor controllers and these calculations are performed inside the interrupt service routine.

Besides the calculation being a hundred times more frequent, also the finish time of the control loop is decreased from 20 seconds to 1 second, to let all motor movements finish within a single second. Something that has decreased a bit in quality is the accuracy of the values on the in the motor controller model, which are changed from doubles to floats. For the DE0-NANO this is necessary since the DE0-NANO's FPU only supports single-precision floating point numbers. Doubles are still supported by software emulation, which is very slow and therefore undesired. By allowing an acceptable loss of accuracy the speed and simplicity of the model was increased substantially, resulting in an accurate and fast responding implementation. The code for the motor controller is in the zip file in the folder called "nios/software".

5D. Image processing

The image processing in this project is about detecting an object through its color in order to make it possible to track that object across the observable space. This is possible since each and every image that is being received from the camera is analysed. To get data from the camera, the GStreamer framework is used which makes it possible to get the video data with a C program. With GStreamer, a pipeline as in figure 13 is set up in order to prepare the data properly for use with OpenCV.

In the pipeline (called "video-player") there's a source, filter, decoder and sink element present. The source element is a v4l2src and is used to capture the video from the Logitech C250 webcam. A capsfilter is applied in order to enforce limitations on the data format which makes sure that the data is processed properly and conforms to the possibilities present. In the case of the Logitech C250 two formats are possible, namely MJPG and YUYV. MJPG is a format in which each frame is compressed as a JPEG image separately and YUYV is a format in which none of the frames is compressed or encoded. Here, the MJPG format is used since this format can be processed faster and is still of good enough quality to be able to be used for the image processing of this project. Since MJPG is used the decoder is a jpegdec which decodes jpeg images and the sink element is an appsink. Appsink is a sink plugin that makes it easier to get a handle on the GStreamer data in the pipeline. Through the appsink it is possible to get the data from the pipeline ready to use for further processing by OpenCV.

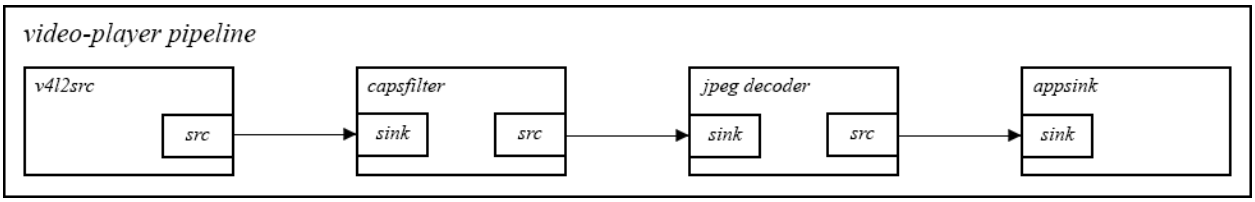

Figure 13: GStreamer pipeline

Every time a frame is received from the webcam and processed by the pipeline, the image is converted from RGB color format (figure 14) to an HSV color format (figure 15) and consecutively used for detection of the specified object. On this HSV frame a filter is applied which selects only the pixels that according to a given range of HSV values are supposed to be detected (figure 16). Once the desired pixels are known, an object is created from these pixels and the vertical and horizontal position of the object are determined. Since this is detection is done in every frame, it is possible to track the object with the webcam (figure 17).

Figure 14: RGB frame

Figure 15: HSV frame

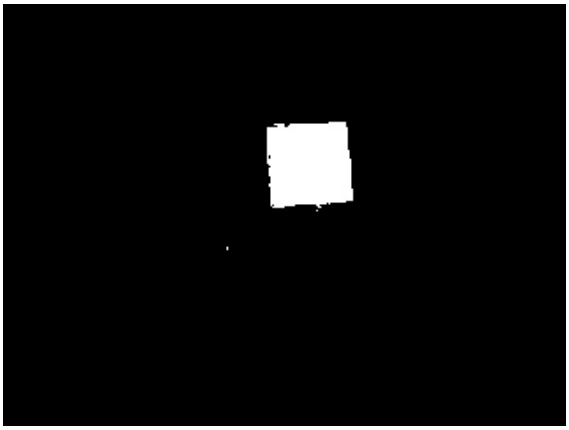

Figure 16: Detected object

Figure 17: Tracking object

From the position of the object in the webcam it is not immediately possible to send the needed angle adjustments to the controller. First the distance from the object to the center of the webcam

has to be determined since the camera should always have the center of the object in the center of the frame. Given the center pixel of the frame is always the same pixel (namely at width 160 and height 120), the distance from the center of the object to the center of the frame can be calculated with formulae:

$$distX = center\ frame(160) - center\ posX\ of\ object$$

$$distY = center\ frame(120) - center\ posY\ of\ object$$

Knowing these distances, one can then translate the pixel distances to degrees and hence inform the motor controller about the difference.

Translating these pixel values to degrees is not as easy as may seem since several steps have to be taken. First the aspect ratio of the diagonal size of the webcam's frames is determined given the horizontal (4) to vertical (3) aspect ratio, resulting in a diagonal aspect ratio of 5, compared to the other two sizes. When this is known, together with the Diagonal FOV of the webcam, the horizontal and vertical FOVs can be calculated. For the Logitech C250 the diagonal FOV is 63 degrees, which results in the following FOVs:

$$Horizontal\ FOV = atan(tan(\frac{Diagonal\ FOV}{2}) * (\frac{Horizontal\ aspect\ ratio}{Diagonal\ aspect\ ratio})) * 2$$

$$= (atan(tan(31.5) * (\frac{4}{5}))) * 2 = 26.12 * 2 = 52.24\ degrees$$

$$Vertical\ FOV = atan(tan(\frac{Diagonal\ FOV}{2}) * (\frac{Vertical\ aspect\ ratio}{Diagonal\ aspect\ ratio})) * 2$$

$$= (atan(tan(31.5) * (\frac{3}{5}))) * 2 = 20.19 * 2 = 40.38\ degrees$$

Knowing the total range, both in degrees and pixels, the translation of degrees per pixel can be made followed by a conversion to the desired radians per pixel:

$$\frac{Horizontal\ FOV}{width\ of\ frame\ in\ pixels} = \frac{52.24}{320} = 0.16325\ degrees\ per\ pixel$$

$$\Rightarrow \frac{0.16325\pi}{180} = 0.00284925\ radians\ per\ pixel$$

$$\frac{Horizontal\ FOV}{width\ of\ frame\ in\ pixels} = \frac{40.38}{240} = 0.16825\ degrees\ per\ pixel$$

$$\Rightarrow \frac{0.16825\pi}{180} = 0.00293652\ radians\ per\ pixel$$

Eventually these values are implemented in the code and the required radian value is transmitted to the motor controller. The entire code for the image processing is in the zip files in the folder JIWY_control/opencv.

5E. Communication

For the communication protocol it is decided to make use of UART. The choice for UART instead of GPMC is explained in detail in Chapter 4. UART synchronises between transmitter and receiver by using a start bit and stop bit rather than making use of a dedicated physical clock wire connecting the two. Specifically for our case this means that the UART communicates between the DE0-NANO and the Gumstix. However, in order to do this successfully it is also important to take into account the voltage level of both boards. Since the DE0-NANO functions at 3.3V and the Gumstix

at 1.8V a level shifter is needed to make sure everything runs smoothly. By putting a level shifter in between the two boards and adjusting the baud-rate to one another, the UART is able to send data to both boards.

Each data message send over the communication bus is 32-bits in size, always requiring 4 bytes. The motor controller receives command messages over the communication bus which are immediately handled by a UART interrupt routine. The motor controller itself transmits messages over the communication bus containing information on the actual current angle of a specified motor. The message struct used for receiving and transmitting is the same. One bit (bit 31) indicates the motor (either pan or tilt) while all other bits are used for a (single precision) floating point value. The Gumstix sends the angle as the floating point value where the motor controller should move the motor to while the motor controller responds with updated motor position values. This way both ends of the communication interface have the required information for full operation with minimal overhead.

6. System integration

Having all units separately tested, the next step is to test the functioning of the system as a whole, after which it is decided whether the system is ready to be accepted and ready for deployment in the real world. While the system architecture is already given rather globally in Chapter 3, the exact functioning of the complete system has not yet been given. The overview of this is shown in figure 18. One can see here that the Pulse Width Modulators and Quadrature Encoders run on the FPGA of the DE0-NANO, the controller runs on the NIOS and the sole function of the Gumstix is to process the images of the webcam and send the angular data over UART to the controller on the NIOS.

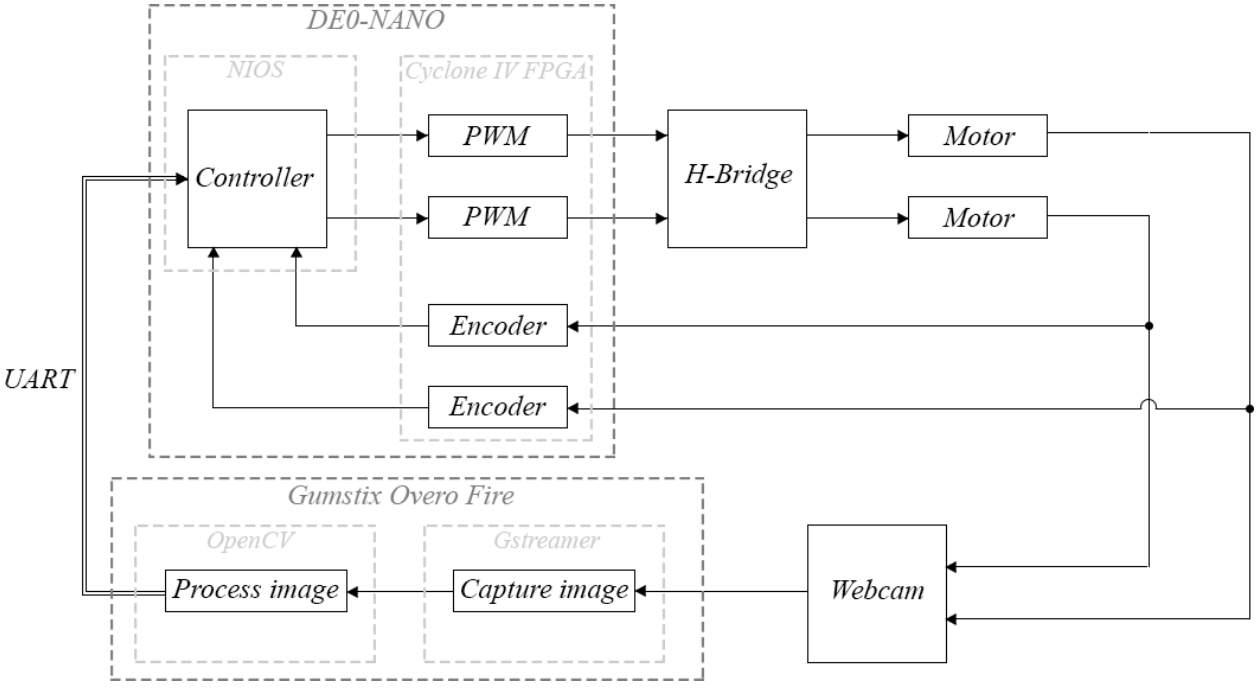

Figure 18: Functioning of the entire system

As can be concluded from this, the DE0-NANO performs all tasks concerning the control of

the motors. The schematic showing all components connected to the NIOS and their pins, can be seen in figure 19. It can be seen that the schematic has additional debounce components on the quadrature encoder inputs. It was found that the filtering of the encoder output pins was already sufficient, meaning that the components became unnecessary for this specific setup. Due to the very small hardware footprint and them not forming any form of negative impact on the signal it was decided to keep them in place, since this does enable other encoders to be used which are of less quality. A PIO output is also available and connected to the DE0's LEDs but were eventually left unused.

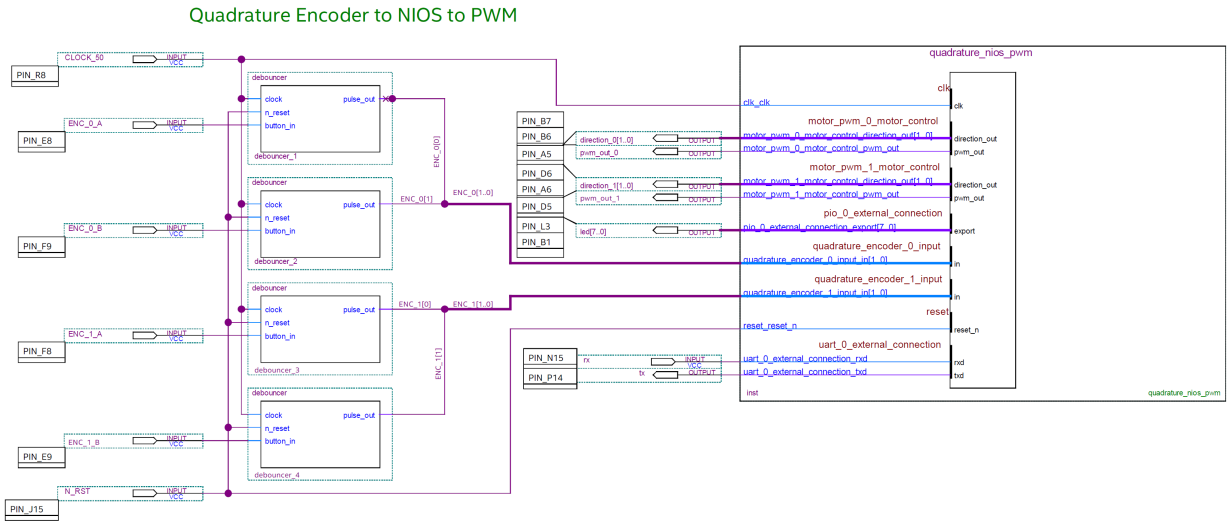

Figure 19: Motor controller schematic in Quartus

The NIOS core architecture has been expanded compared to the default components (as explained during the tutorials) which consist of a clock, memory, CPU, jtag, system timer and sysid. The components which have been added are (figure 20):

- *Floating Point Hardware 2*
  The first component at the top is the Floating Point Hardware 2, also known as the floating point unit. It is set to handle specific opcodes and is connected to the CPU via two custom instruction slave connections. In the Eclipse C programming interface for the NIOS core the FPU library is compiled with optimizations enabled for "size", meaning that the library will be compiled to an as small as possible memory footprint.

- *UART Intel FPGA IP*
  The second component in the list of figure 20 is the UART Intel FPGA IP, referred to as the UART component. The UART's RX & TX pins are routed to external pins which can be seen on the schematic of figure 19. The UART component generates an interrupt on an incoming message with a priority of 4.

- *quadrature_encoder_x*
  The next two components, named quadrature_encoder_x, have been custom made to connect the quadrature encoder which has been designed in VHDL to the NIOS. They require a clock and a reset line while they can be accessed over the avalon memory bus by the CPU. Each quadrature encoder component has two inputs which are externally available.

- *motor_pwm_x*
  The next two components, named motor_pwm_x, have been custom made to allow the easy control of external motors. They also require a clock and a reset line being able to configured over the avalon memory bus by the CPU. Each component has two output pins to set the direction and one output pin for the PWM signal. All three output pins are externally available.

- *Interval Timer Intel FPGA IP*
  After the motor_pwm_x components the Interval Timer Intel FPGA IP component can be seen, named timer_0. This timer is used to generate the timed interrupt periods on which the calculations are performed in the motor controllers. The timer component generates an interrupt every 100 microseconds with a priority of 3, which gives it a higher priority than the UART component, ensuring that its IRQ routine will not be interrupted by incoming UART messages.

- *PIO Intel FPGA IP*
  Finally a PIO Intel FPGA IP component is available which is connected to the DE0-NANO's eight LEDs. This component is not an important part of the system's functionality but was mainly a useful way of expressing the state of the motor controller.

All other components besides NIOS's (on-chip) memory have been kept unchanged compared to the configuration of the tutorials. The On-Chip Memory Intel FPGA component has its total memory size increased to 46080 bytes.

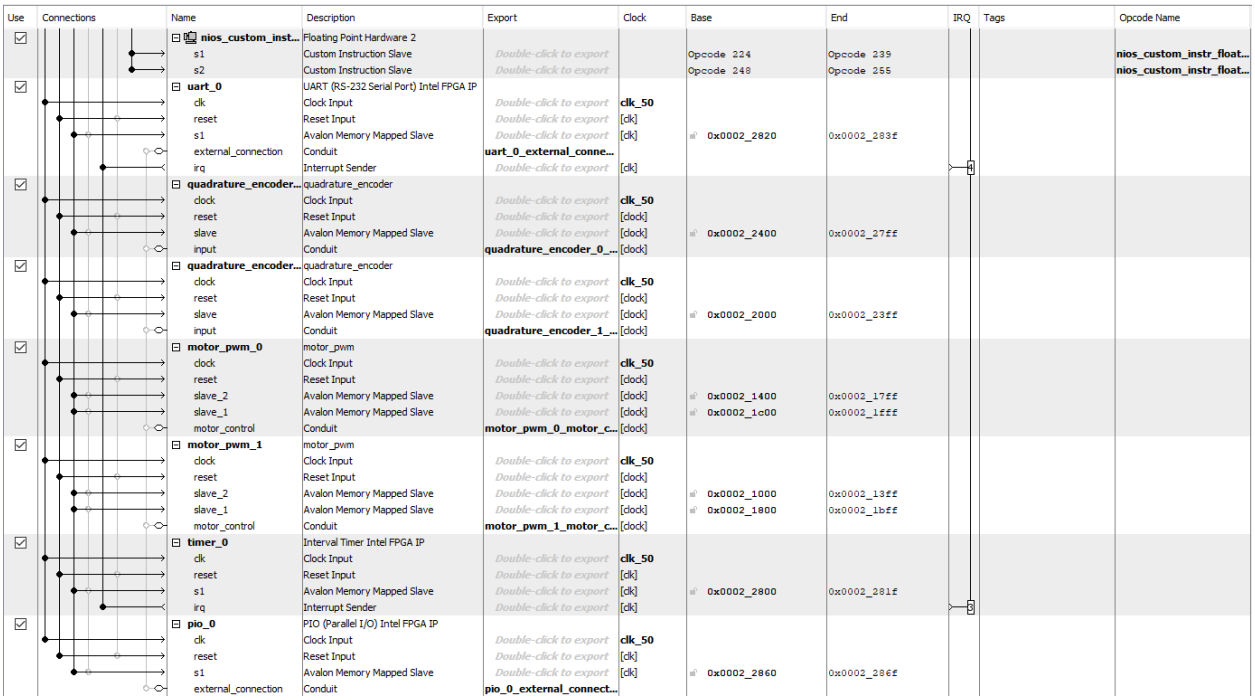

Figure 20: Additional NIOS components

When performing tests on the entire system nearly everything has been proven to work, with the exception of one problem. Given the time for this project, and the error faced, it was impossible to make the system deployment ready. The problem discovered is namely that when driving both

pan and tilt with the controller simultaneously as a full-blown setup, based on image processing data, the JIWY starts moving in inexplicable ways. The reason this is confusing is because when the controller is manually given coordinates (not generated by the image processing), it does work with both pan and tilt simultaneously. This would suggest a problem with the calculation of the coordinates from the image processing. However, in the case the controller receives image processing data and drives only the pan, or just the tilt, everything seems to work fine and the JIWY, fairly accurately, follows the object as expected (although just over one axis). In the next chapter, discussion and conclusion, possible causes for this error are identified and solutions are proposed on how this error can be avoided in the future.

7. Discussion and conclusion

Throughout the past months the entire design flow (following the V-model) and design space exploration have been gone through. First, the requirements have been gathered and a goal has been defined, followed by a mapping of the system architecture. Next, the design space is explored in detail for each subsystem, all established through five different criteria. Based on the results of the DSE, every subsystem is implemented and tested properly after which everything is combined in one entire system. This system is close to operating properly, however, unfortunately one error is expected to be preventing the system to be fully functional.

This one error, faced when driving both the pan and tilt with the controller, is mentioned in the previous chapter and is also the main challenge still unanswered for this project. Unfortunately, due to limited time availability it was impossible to do the proper testing in order to confirm the prediction and implement corresponding improvements. Without proper testing the best guess is that the coordinates which are derived from the image processing cause a change in position which is too large and is furthermore unstable. The change in the x and y direction from the image processing might be simply too extreme, causing the camera to always move towards the visible area boundaries.

Once the control mistake is fixed, a working implementation of visual servoing is anticipated. Of course this does not mean the current implementation cannot be improved. In the case more time would be available for developing the system, several improvements can be implemented. The most obvious improvements are to implement the more complex but also more accurate and faster executing solutions resulting from the DSE. When more time is available, the development time has a lower weight which could change the entire outcome of the DSE. Next to the improvements in the development, also evaluation and testing can be done more elaborately in the future to ensure that everything works even smoother. An example of this would be to smoothen the movement of the camera using filters, when following an object.

All and all, almost everything is functioning as desired. Despite the fact that the complete system is not yet entirely working, the evaluation and testing results of the subsystems do give hope that the solution for a working system is close.

8. Learning points on methods / DSE approach of ESL

Reflecting back on the course we can say that for the both of us it has been a great learning opportunity on several aspects. First of all, with the COVID pandemic being omnipresent throughout the rest of the year it was amazing to finally be able to work in the lab and actually get the practical hands on experience that is so important for Embedded System Design. Also great about this course is that it combines a bit of all core courses of the master which helps us see the bigger

picture of why each course fits the curriculum so well. Take for example the IP components in Quartus, these are a great addition to the education we already got during the Embedded Systems Architecture 1 course.

Something else which is good to have experienced through the labs is the hassle with all the hardware and combining it with the software. On paper and in simulations it always speaks for itself, however in reality when working with real hardware components there are always things that you did not take into account. Some hardware components were broken for example, or were tuned differently than expected (for example the bounce in the encoder which was not present).

Next to smaller learning points, two notable ones are about Design Space Exploration and the Modern Iterative Design Approach. Since Design Space Exploration has not been mentioned to such an extent in previous courses, most of it was completely new and really showed us a great way to make decisions about how one determines the architectural details of a system being developed. The DSE approach helped us to understand how to do model-driven design the right way with among others limiting the solution space, creating aspect models and easing the decision process.

The last, but not least important, major learning point is about the Modern Iterative Design Approach. One of us was already introduced to this approach a bit through the Systems Engineering and RTSD courses. However, for the other this was completely new. By structuring the design flow concurrently, and with the introduction to the V-model as well as the design pyramid it was much clearer to us which steps to follow in order to arrive at the desired outcome. By iterating through the various steps of the flows, the complete overview of the design process was considerably better manageable than how we would have approached it before Embedded Systems Laboratory.

Appendices

A. VHDL code for the Quadrature Encoder

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity quadrature_encoder is
6   port
7   (
8    clk :  in std_logic;
9    n_rst : in std_logic;
10   enc_in: in std_logic_vector(1 downto 0);
11   count : inout std_logic_vector(31 downto 0)
12  );
13 end quadrature_encoder;
14
15 architecture behaviour of quadrature_encoder is
16 begin -- Signal decoder.
17  process (clk, n_rst, enc_in)
18  variable counter : signed(31 downto 0) := (others => '0');
19  variable prev_input_state: std_logic_vector(1 DOWNTO 0) := enc_in;
20  begin
21    if (n_rst = '0') then
22     counter := (others => '0');
23     prev_input_state := enc_in;
24    -- Only enter this if statement if the input state has changed.
25    elsif rising_edge(clk) and enc_in /= prev_input_state then
26     case prev_input_state is
27      when "11" =>
28       if enc_in = "10" then
29        counter := counter - 1;
30       end if;
31       if enc_in = "01" then
32        counter := counter + 1;
33       end if;
34      when "01" =>
35       if enc_in = "11" then
36        counter := counter - 1;
37       end if;
38       if enc_in = "00" then
39        counter := counter + 1;
40       end if;
41      when "10" =>
42       if enc_in = "00" then
43        counter := counter - 1;
44       end if;
45       if enc_in = "11" then
46        counter := counter + 1;
47       end if;
48      when "00" =>
49       if enc_in = "01" then
50        counter := counter - 1;
51       end if;
52       if enc_in = "10" then
53        counter := counter + 1;
54       end if;
55      when others =>
56       ASSERT false REPORT "previous state not known" SEVERITY note;
57     end case;
58     prev_input_state := enc_in;
59    end if;
60    -- Route the counter to the output.
61    count <= std_logic_vector(counter);
62  end process;
63 end behaviour;
```

B. VHDL code for the Pulse Width Modulator

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity pwm_module is
6   generic (
7    DATA_WIDTH : natural := 32;  -- word size of the output register
8     DUTY : natural := 16  -- word size of the output register
9   );
10  port
11  (
12   clk :  in std_logic;
13   n_rst : in std_logic;
14   period_cc: in std_logic_vector(DATA_WIDTH-1 downto 0); -- in clock cycles
15   duty_cycle : in std_logic_vector(DUTY-1 downto 0); -- in clock cycles
16   direction_given : in std_logic_vector(1 downto 0);
17   enable : in std_logic;
18   dir : out std_logic_vector(1 downto 0);
19   pwm_out : out std_logic
20  );
21 end pwm_module;
22
23 architecture behaviour of pwm_module is
24 begin
25  -- PWM module.
26  process (clk, n_rst)
27   variable cc_count : unsigned(DATA_WIDTH-1 downto 0) := (others => '0');
28   variable signed_period_cc : unsigned(DATA_WIDTH-1 downto 0);
29   variable signed_duty_cycle : unsigned(DUTY-1 downto 0);
30  begin
31     if(n_rst = '0') then
32       cc_count := (others => '0');
33       signed_period_cc := (others => '0');
34       signed_duty_cycle := (others => '0');
35    pwm_out <= '0';
36     elsif(rising_edge(clk)) then
37       signed_period_cc := unsigned(period_cc);--x"000186a0";
38       signed_duty_cycle := unsigned(duty_cycle);--x"C350";
39       if(enable = '1') then
40         if(cc_count < signed_period_cc-1) then
41           if(cc_count < signed_duty_cycle-1) then -- this if case is not needed if waiting
                 one period before starting the pwm signal is okay
42             pwm_out <= '1';
43           elsif(cc_count >= signed_duty_cycle-1) then
44             pwm_out <= '0';
45           else
46             ASSERT false  REPORT "No such operation exists" SEVERITY failure; -- not
                   synthesized, just for debugging
47           end if;
48           cc_count := cc_count + 1;
49         elsif(cc_count >= signed_period_cc-1) then
50           cc_count := (others => '0');
51           pwm_out <= '1';
52         else
53           ASSERT false  REPORT "No such operation exists" SEVERITY failure; -- not
                 synthesized, just for debugging
54         end if;
55       else
56     pwm_out <= '0';
57   end if;
58       dir <= direction_given;
59     end if;
60   end process;
61 end behaviour;
```