Smart Switching Regulator

Student: Niculescu Vlad

Mentor: Herbert Pötzl

Link to code: https://github.com/vladniculescu/GSOC

1 Prerequisites

In order to realize or develop this project, you will need:

1. Populated development board
2. Digilent Nexys Video FPGA board
3. Oscilloscope
4. Power Source (1.5A current capability)
5. Logic analyser (not mandatory, but useful)

2 Introduction

The main goal of this project is to design an adjustable converter for the Apertus AXIOM Beta camera system. The AXIOM requires multiple voltages, so an adjustable solution is preferred over having multiple converters. Multiple instances of this converter will be utilised in the Beta, each one working at a different output voltage, but with identical hardware components, and the performance criteria of the converter are efficiency, accuracy and ripple.

The whole converter will represent an individual block, which will communicate with the central processing through I2C. The input supply voltage is 5V.

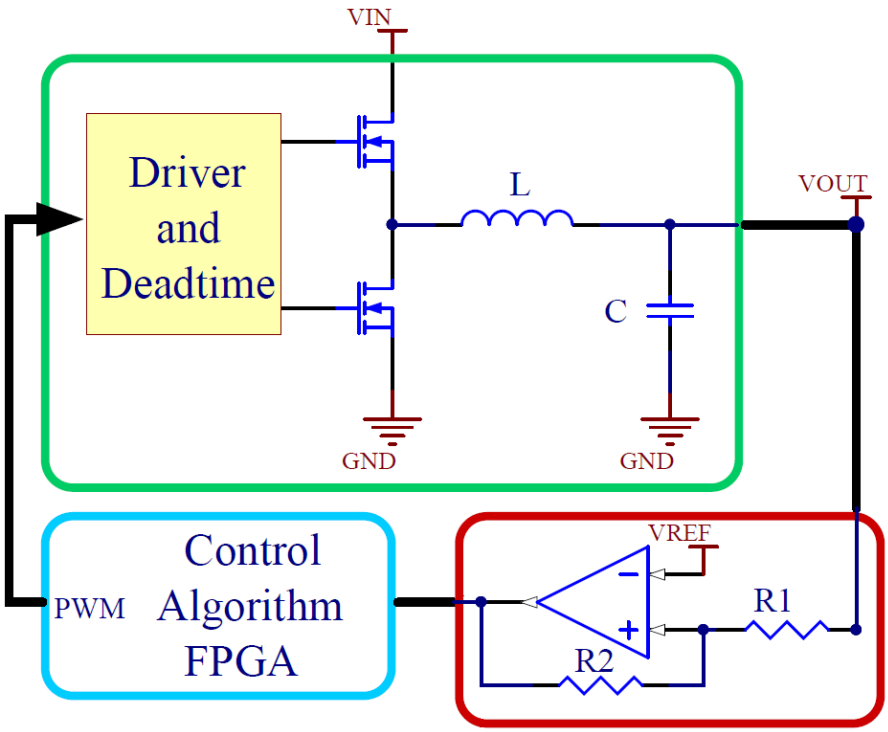

Figure 1: The Block Schematic

The synchronous configuration was preferred because of efficiency considerations. The chosen driver is controlled by a single PWM signal and this driver process the signal in order to properly drive the two transistors – it generates two complementary signals and also adds dead-time. The output voltage is maintained at a constant due to the feedback mechanism, this consists of a hysteresis comparator that indicates if the actual voltage is above or below a reference voltage.

The control algorithm runs on an FPGA and controls the converter accord-

ing to the information received from a comparator. Its main purpose is to maintain the output voltage as close as possible to the reference - This reference is generated by a 12-bit DAC and thus can be easily adjusted. The output voltage follows the reference voltage regardless of the drawn current. Both the DAC and the control algorithm act as I2C slaves which allow the I2C master to modify essential parameters. While the DAC's I2C allows for a reference change, the converter's I2C gives control over the converter's parameters, eg. switching frequency or speed of the feedback loop. Both slaves will be connected to the global I2C bus of the AXIOM system.

3 Designing the Converter

The first step in realising this project was to design a schematic for a test setup as well as building a prototype to test with. A rendering of the designed prototype board can be seen in figure 2.

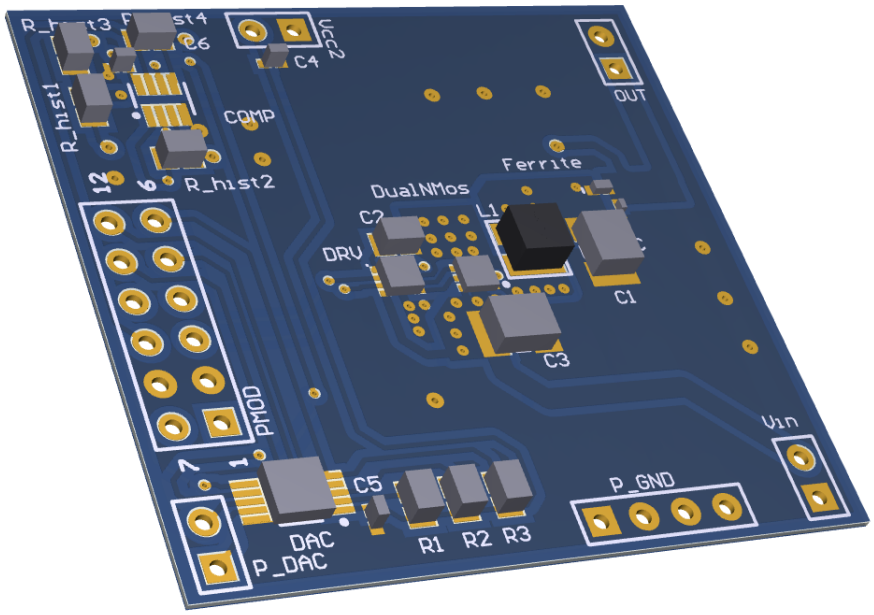

Figure 2: The PCB

Based on the design files, the PCB was fabricated at OSH Park and manually populated. The board was populated using the hot plate, because this was a good way to solder the parts in QFN packages.

After the board was populated, I started testing each module individually. The converter placed in the centre of the board was tested using a 'fixed duty

cycle PWM' signal. On applying such a signal, it was observed output will vary in accordance with the duty cycle of this PWM signal - This assures the proper functionality of the converter.

The DAC converter that uses an I2C interface was the next module tested. Using an USB to I2C Master board from 'Microchip Technology', certain configuration bytes were sent through I2C to configure the internal registers of the DAC. I searched in the datasheet for the command that would change the output voltage and then sent the commands that limit output voltages. I also measured the actual value at the DAC's output to check if this voltage has the expected value. Finally, I came to the conclusion that the DAC block works well.

Testing the comparator was the next stage of ensuring that the board functioned correctly. If this one had behaved correctly then the functionality of the board would have been validated, however, during this test, I discovered an electrical error in the schematic. The positive feedback of the comparator was wrongly connected, so the comparator did not act like a hysteresis one anymore. Consequently, I modified the board and managed to do it without damaging the traces by soldering two resistors on the pads of one. Ultimately this problem was fixed in the final iteration of the board, which is currently available on github (Layout Files/PCB_Project).

The hysteresis comparator acts like a feedback mechanism which gives the input for the control algorithm. The hysteresis was introduced to reduce the switching frequency of its output, however, the hysteresis was set to 80mV. A higher value would generate a higher ripple, too.

In order to obtain the desired value, a 1:62 ratio of the resistors was needed – in theory. To test if this value is also obtained in practice, a triangular signal was applied at the non-inverting input of the comparator, while a fixed voltage was applied at the inverting input. Figure 3 illustrates the triangular signal and also the comparator's output where the two thresholds are obvious.

4 The UART to I2C Bridge:

To simplify the hardware testing an UART to I2C bridge was designed to control the system from a PC via serial port. This bridge allows real-time communication between computer and the converter which means that the converter's parameters as well as the DAC's output voltage can be adjusted from the serial console of the computer. The FPGA development board used for this purpose was a 'Digilent Nexys Video', populated with a 'Xilinx Artix 7'.

The first step was to establish communication between computer and FPGA. The Nexys Video board uses a FTDI to connect the FPGA to computer's USB. The UART to I2C bridge was split into two components:

1. The UART control interface: This half is used to adjust the converter's settings

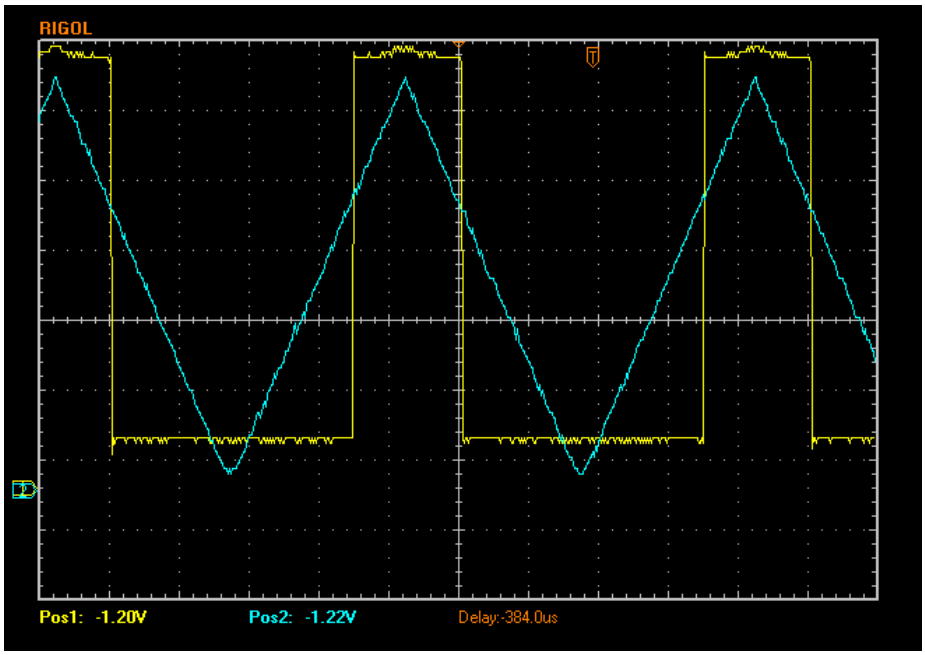

Figure 3: The comparator's output

2. The I2C Master: In order to configure the DAC, both components are required

Therefore, the first developed component was a UART module. This one consisted of a receiver and a transmitter module. The transmitter was tested using the LEDs on the development board. Basically, I sent different characters from the serial console and the eight LEDs illuminated according to the 8 bit ASCII number that corresponds to the sent character. The used serial terminal was Arduino IDE Serial Monitor. In order to use it, you have only to select the proper COM port and than to select the right baud rate, which was 9600 for this project.

Figure 4 shows how characters are sent from the serial console.

The second step was to develop an I2C Master module. This one takes the information received from the UART Receiver module, and sends it through I2C to the DAC to configure it. The received command is decoded and sent to the I2C bus (controlling the DAC). Both UART and I2C Master are implemented as Verilog modules.

How I2C works: In order to have an I2C communication it is necessary a master and at least one slave. The master is the one who starts the communication by pulling the SDA low and then pulling the SCL low. After the start sequence finished, the master starts sending eight bits of data through the SDA bus. The first data bit is outputted in the same moment the SCL firstly goes

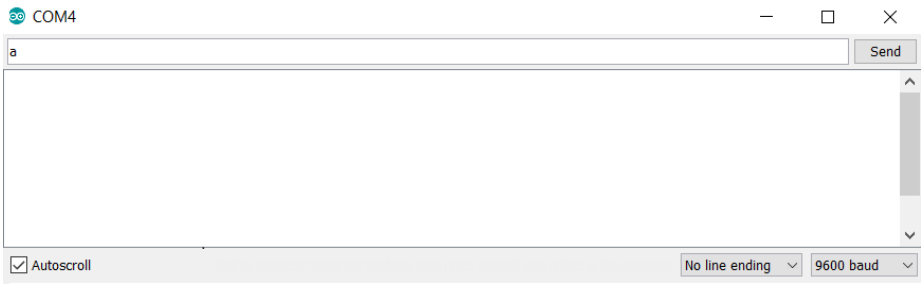

Figure 4: The Serial Console

down. After the eight bit were sent, the Master lets the SDA bus high and waits for the slave to pull it down. This fact is called acknowledge, because the slave confirms that it received all of the data bits. In an I2C communication, the data is outputted to bus on the negative edge of SCL and read from bus on the positive edge of this signal. For most of the I2C devices, three bytes of data are sent during a communication. The first one is the address of the device, so if multiple devices are connected to the same bus, just the one having the sent address will acknowledge. The second byte represents the address of the slave's register where data will be written or read. The slave devices are have a register file implemented inside, so this address can address any register. Finally, the third byte is the actual data that has to be stored in the slave's register, if a write operation has been requested. If the I2C operation is a read one, the slave will respond with the value from that location. The read/write operation is selected by the least significant bit from the first transferred byte. If this is set to 1, the operation is a read, otherwise it is a write. Consequently, the master sends the 7-bit address of the slave it wishes to communicate with, which is finally followed by a single bit representing whether it wishes to write (0) to or read (1) from the slave. Whenever a side sends a byte of data, the other side has to acknowledge. To stop the communication, the master firstly releases the SCL and then the SDA. This is the most widespread I2C structure. However, depending on slave's complexity, this structure can vary in length.

The I2C package structure of the used DAC consists of 9 bytes for each transmission. The first one is the DAC's address and the other four pairs of two bytes represent the numbers that give the output value for all of the four channels. These numbers are set by the user from the serial console and the bridge makes sure that those numbers arrive in the DAC's registers.

The communication protocol of the bridge consists of a string of characters that represent hexadecimal numbers. The I2C Master sends 8 bit packages, so, the command string consists of pairs of two bytes followed by a read/write flag, eg. I used a discrete slave I2C device to figure out if the Master works properly. I sent a sequence like "S49W71W00Rs", which reads the value in register 0x71 for a device with an address equal to 24. "S" denotes that a command for the DAC is going to be sent and "s" marks the final character in

this command. After a command successfully executes the bridge responds with a “D” character as acknowledgment. The “00” sequence behind the “R” doesn’t have any significance and merely signals that a byte is going to be received. However, because the structure involves two hexadecimal numbers followed by a letter, it is important that the “00” group or other group of two numbers to be there. The hexadecimal number “49” represents the concatenation of the hexadecimal “24” in binary with “1”, which corresponds to a read operation. The waveforms involved in this process are highlighted in Figure 5. A Saleae logic analyzer was used to test and debug the functionality of the I2C Master.

Figure 5: The I2C waveforms

In this way I separately tested the functionality of the two components of the bridge. After I was sure that both UART module and I2C Master worked, I integrated these two modules to obtain the desired bridge. Finally, I sent a string from the serial console to configure the DAC.

I wanted to obtain 2.05V at all of the four outputs of the DAC.

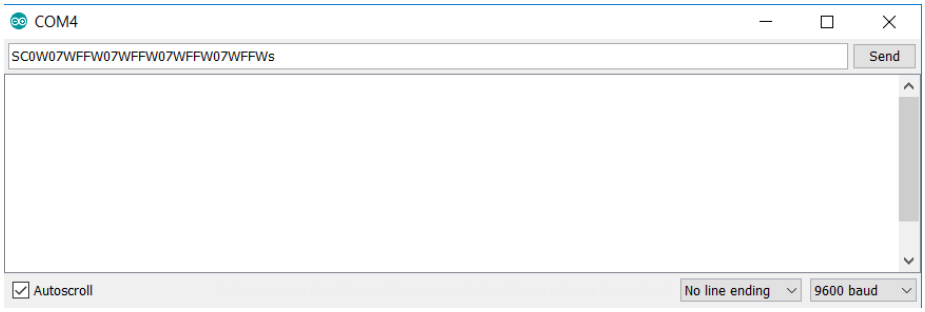

Figure 6: The I2C waveforms

Consequently, I sent the package in Figure 6.

Address	Ch1_MSb	Ch1_LSb	Ch2_MSb	Ch2_LSb	Ch3_MSb	Ch3_LSb	Ch4_MSb	Ch4_LSb

Table 1: The Package Structure for the DAC configuration

Table 1 the structure of the package that has to be sent in order to configure the DAC.

The MSb and LSb from 1 mean the most significant, respectively the least significant bit. There are four channels because the DAC is a 12-bit four channel

type. The ChX_MSbyte is a 4-bit number and ChX_LSbyte is an 8 bit one. Concatenating those two, the 12-bit value is obtained for each channel. So, returning to the example, "SC0W07WFFW07WFFW07WFFW07WFFWs" is sent from the serial console where S represents the start command, C0 the device address and W written after each two-number group denotes that the number has to be written (R is for read). The numbers are sent in hexadecimal format. Concatenating 07 and FF for each channel, 2047 is obtained. This is 2.05V for a 4.096V reference.

The Bridge can also adjust the converter's parameters. The structure of the command is very similar but this time the first letter of the string command will not be "S".

Command	Effect	Length
S	Adjusts the DAC's settings	Variable
X	Modifies the switching frequency	16 bit
T	Adjusts the loop speed	16 bit

Table 2: The commands of the Bridge

Table 2 Shows the adjustments that can be made using the Bridge. Those adjustments are executed from the serial console. The "X" and "T" characters signal that a command intended for the converter is going to be made. This one will only use the UART module.

The switching frequency is variable and can be adjusted between 0 and 780kHz with a 16-bit resolution. In order to modify it, a command like "Xxxxxs" should be sent from the serial console, where "xxxx" represents a 16-bit number in hexadecimal format. Consequently, "XFFFFs" will set the frequency to its maximum value. The same mechanism works for the loop responsiveness. A command like "Txxxxs" tells the algorithm how often to read the comparator. For a higher value of "xxxx" the loop will be slow, because the system will have to count to a high number until reading the comparator's output. Basically, it wants "xxxx" cycles and then reads the comparator.

As a conclusion, the bridge can adjust the DAC's settings, its switching frequency and the speed of the feedback. However, for the last two, only the first half of bridge is necessary, as the controller is a verilog module, not a separate I2C slave device.

The Bridge is designed as a state machine. The states are: START_S, FILLBUF_S, SEND_S, WAIT_S, UPDATE_DIV_S, UPDATE_INTERVAL_S. In the first state, it continuously verifies if a new character has come through the serial bus. If so, it checks if this character corresponds to one of the defined functionalities. Then, it goes in the state that is specific for the received character.

```
START_S : begin
    goTX <= 0;
    memCnt <= 0;
    byteCnt <= 0;
    if (d_avail) begin
        if (data_rx == 83)        //S
            state <= FILLBUF_S;
        else if (data_rx == 88) //X
            state <= UPDATE_DIV_S;
        else if (data_rx == 84) //T
            state <= UPDATE_INTERVAL_S;
    end
end
```

For the converter's settings the command string length is fixed, but for the DAC's adjustment it can vary. The receipt of a converter's settings are each one handled by one state, which is 'UPDATE_DIV_S' for adjusting the switching frequency and 'UPDATE_INTERVAL_S' for adjusting the loop reading speed. This states present a counter that acts like an index for the current received hexadecimal number. In contrast to this, receiving a DAC's setting involves other mechanisms because it has a variable length. The string received from the serial console is stored in a buffer, which is done in the 'FILLBUF_S' state. When the "s" character has been received, the process goes into a decoding state, where the information stored in buffer is interpreted and sent to the I2C Master module, which is done in the 'SEND_S' state. The 'WAIT_S' state waits during the I2C sending process and receives feedback from the I2C Master module to check if the transfer succeeded.

This I2C Master module was designed just for development. Because it allowed me to control the converter and DAC in real time, I didn't need to re-synthesize the project at every parameter change. It won't be necessary when the converter will be integrated in the AXIOM System because it will be controlled by this one through the I2C Slave module. So the next step I took was to implement the Slave I2C for converter. This module is consisted of a I2C Slave that also has a register file. Some locations of this register file are directly controlling the converter's parameters. This allowed the converter to be configured by modifying some cells in the register file of an I2C Slave module. The converter's parameters are two 16-bit numbers. Judging by the fact that the register files has 256 locations of 8-bit numbers, it results that only four locations will be used to control the converter and the remaining 252 will remain unused. Figure 7 indicates the connections that are made between blocks and also which registers are used to adjust the parameters.

I tested this Slave module using an 'Arduino Uno' board. Firstly, I wrote a certain value in the first cell of the file register. Then, I read that cell to be sure that I received the value I wrote. For the actual converter, 'register 0' and 'register 1' store the LSB and MSB of the frequency division variable and 'register 2' and 'register 3' store the LSB and MSB of the feedback loop

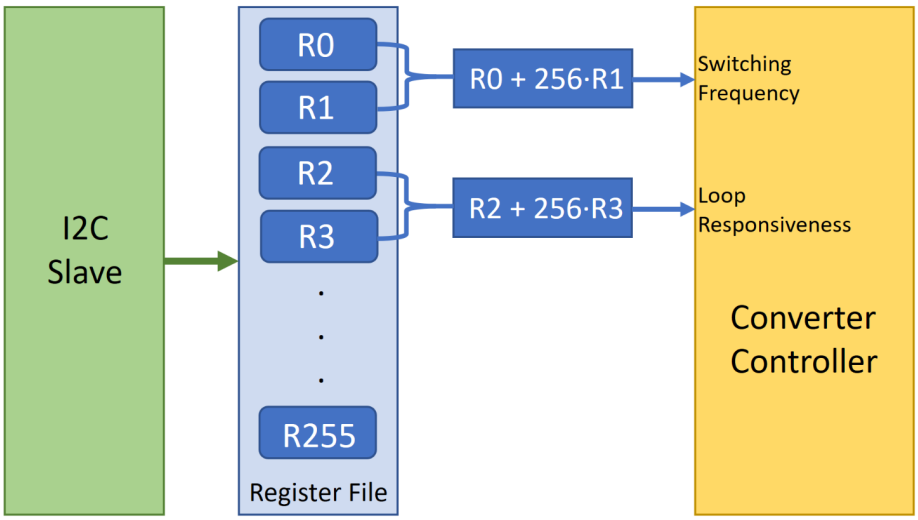

Figure 7: The I2C Slave block structure

responsiveness variable. eg. if the 'register 0' equals "FF" and 'register 1' equals "AB", then the switching frequency will be:

$$f_{SW} = \frac{register[1] \cdot 256 + register[0]}{65536} \cdot 50Mhz = \tag{1}$$

$$= \frac{171 \cdot 256 + 255}{65536} \cdot 50Mhz = 0.67 \cdot 50Mhz = 33.5Mhz \tag{2}$$

"171" and "255" are the decimal values of the hexadecimal "AB" and "FF".

5 The control algorithm

The control algorithm uses a fixed switching frequency and varies only in response to the duty cycle of the PWM control signal. The parameters of this algorithm are the interval at which the duty-cycle is updated and the switching frequency. The switching frequency was experimentally determined to be 780kHz. A counter is incremented whenever the output voltage is higher than the reference and decremented when smaller than the reference. When the update time comes, the counter is evaluated and the duty-cycle is increased or decreased by 1, according to the value of this counter.

```
always @(posedge clock) begin
    ramp_prev <= ramp;

    if (ramp == 0 && ramp_prev == 63) begin
        test <= !test;
        if (fb) begin
            if (decimal == 65000 - fb_interval) begin
                decimal <= 65000;
                if (duty > 0)
                    duty <= duty - 1;
            end
            else
                decimal <= decimal - 1;
        end
        else begin
            if (decimal == 65000 + fb_interval) begin
                decimal <= 65000;
                if (duty < 63)
                    duty <= duty + 1;
            end
            else
                decimal <= decimal + 1;
        end
    end
end
```

Tuning the algorithm:

Finding a way to control the transistor was by far the most challenging stage of this project. In the beginning I didn't even think about including a variable that sets the feedback loop responsiveness. Despite the fact that a high efficiency (over 85%) was obtained even from the first iterations of the algorithm, the biggest challenge was reducing the output ripple. I used the oscilloscope to measure this ripple by evaluating the peak-to-peak amplitude of

the AC component. It basically has a high frequency component which comes from switching, and a lower frequency component generated by the action of the control loop (this is usually between 1 and 30 kHz). If the feedback loop actions too slow then the ripple will be higher because a low frequency component won't be rejected by the LC filter.

Figure 8: The output ripple for a slow feedback loop

Figure 8 highlights the output signal when such a situation occurs. The transitions between levels look like a square form - which indicates a poor filtering effect. In this case, the feedback loop responsiveness should be increased because a low-pass filter does not have any effect on a very low frequency signal. This was a mistake I made and I hardly figured out the cause.

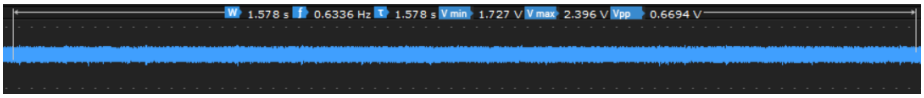

Figure 9: The output ripple for a fast feedback loop

However, if the speed of the feedback loop responsiveness is set to a high value, its effect will be harmful, too. Because the algorithm uses only a proportional component, if this one is too aggressive, it will determine an increase in the output voltage at every iteration. Consequently, it will somehow oscillate around the average position with a high amplitude. This was the first problem I encountered in developing the algorithm. In the beginning, I evaluated the comparator and adjusted the duty-cycle at the start of every PWM cycle. This proved to be too fast and triggered this kind of problem. In the end an equilibrium was found and I determined that the best value of the interval variable is 96. This means that the feedback loop is read every 96 cycles. Figure 9 highlights the improvement in ripple value.

Because the comparator's output turned out to be a little noisy, an averaging filter was introduced to reduce the noise.

```
always @(posedge clock) begin
    if (cnt == filter_length) begin
        cnt <= 0;
        cnt1 <= 0;
        cnt0 <= 0;
        comp_out_f <= fb_filtered;
    end
    else
        cnt <= cnt + 1;

    if (comp_out)
        cnt1 <= cnt1 + 1;
    else
        cnt0 <= cnt0 + 1;

    if (cnt1 > cnt0)
        fb_filtered <= 1;
    else
        fb_filtered <= 0;
end
```

This filter has two counters. During a fixed interval it counts how many ones and how many zeros are encountered. At the end of the interval, it outputs the most dominant value. "cnt1" and "cnt0" are the two counters. The averaging interval can be modified. However, it is not recommended to choose high values for it because doing so would cause a big delay. This averaging filter has the same impulse response as a finite impulse response low-pass filter, and consequently, a higher order will cause a higher delay, which is not helpful when it comes to real-time processes.

Figure 10: Filtered vs unfiltered comparator

In Figure 10, the effect of the filter is obvious, because the narrow high frequency spikes are rejected.

Finally, adjusting the switching frequency at 780kHz for the found speed of the feedback loop, I was able to obtain the waveform in Figure 11,.

The obtained ripple reading is around 110mV. This value is still poor when considering the requirements of this project. In order to reduce this ripple an additional hardware filter was introduced which consists of a ferrite bead and an 1uF capacitor. Figure 12 presents the output ripple after this additional circuit was added. The ripple was reduced to 30mV.

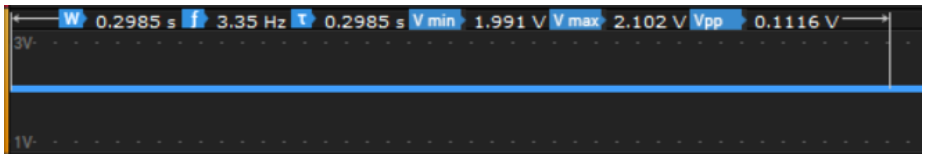

Figure 11: The output signal – 110mV ripple

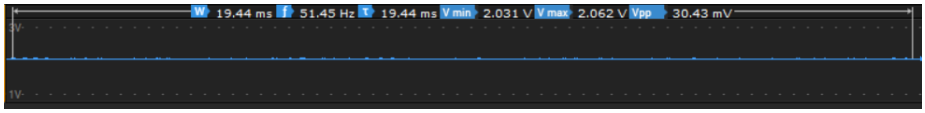

Figure 12: The output ripple after the second filter was added

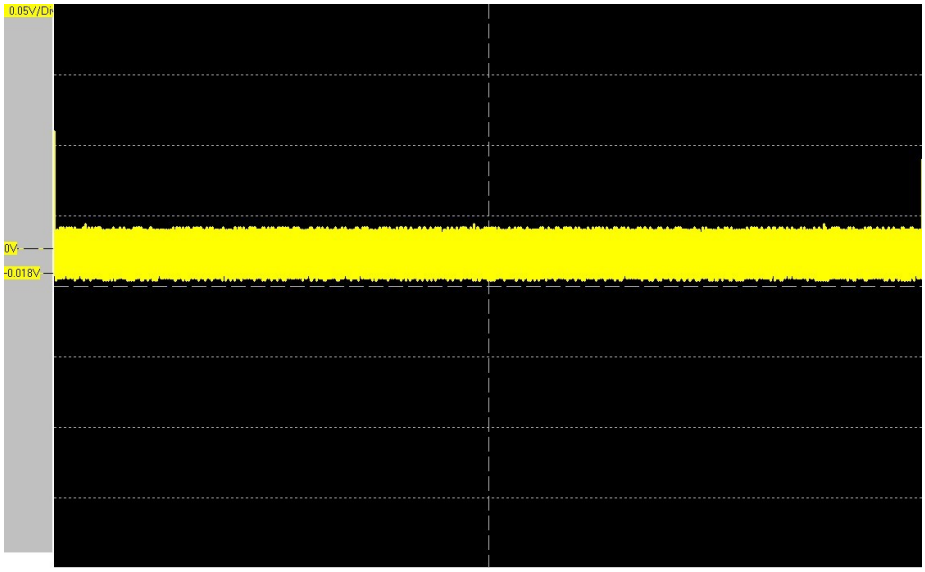

Figure 13: The output voltage measured with an oscilloscope

Figure 13 displays the same waveform exported from a Rigol DS1102E oscilloscope set to 50mV/div.

6 Measuring the efficiency

After I managed to achieve the most important objective (obtaining a low ripple), I went on to determine its efficiency. The efficiency was measured as being the ratio between the output-power and the input power. All the efficiency measurements were done for a 2V output voltage. One Ampere meter was put in series with the power supply and another one was put in series with the

load. For different load resistances, the input current, input voltage, output current and output voltage were measured. The efficiency was calculated with the formula:

$$e = \frac{U_{out} \cdot I_{out}}{U_{in} \cdot I_{in}} \tag{3}$$

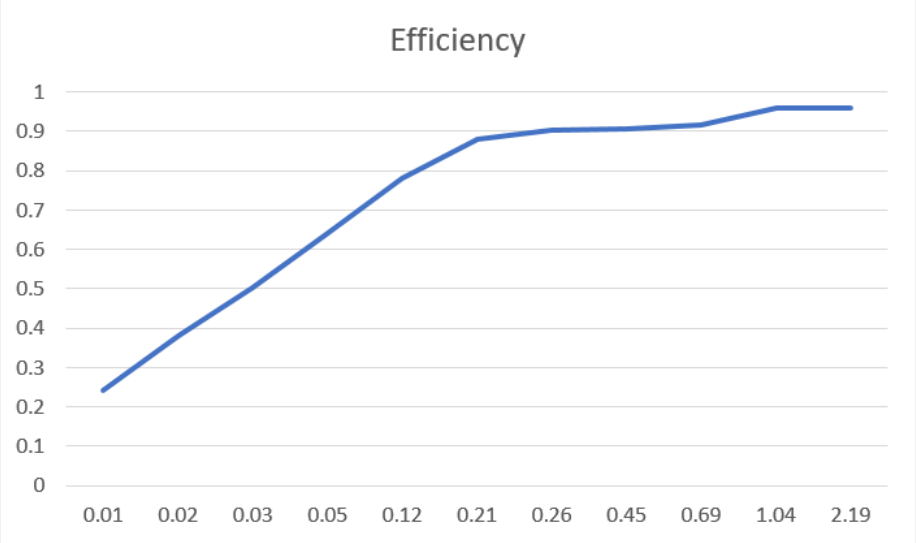

Figure 14: Variation of efficiency with current

What is important to notice in Figure 14, is that for load currents over 100mA, the efficiency goes over 70%. Moreover, for output currents higher than 250mA the efficiency exceeds 90%.

7 Measuring the accuracy

DAC voltage [V]	Output voltage [V]
1.5	1.47
1.753	1.749
2.05	2.08
2.249	2.229
2.49	2.53
2.99	3.03
3.49	3.53
3.99	3.988

Table 3: Reference Voltage vs Output Voltage

Table 3 shows that the difference between reference voltage and output voltage is less than 40mV.

8 Area occupied

The area occupied by each functional block is described below:

Part type	Area [mm2]	Code
Driver	4.6	NCP81151B
Dual MOS	4.6	NTLUD4C26N
2 x Capacitors 1210	25.2	
Capacitor 0805	4.05	
Inductor	12.21	
Ferrite bead	1.3	742792023
Capacitor 0603	1.3	
Total	53.2	
Total are on PCB	148.0	

Table 4: Area occupied by Buck Converter

Table 4 shows the area occupied by each element of the buck converter.

Important: These parts were chosen for prototyping. For a final solution, the capacitors in a 1210 package can be replaced 0805.

Part type	Area [mm2]	Code
DAC	11.1	MCP4728
3 x Resistance 0805	12.1	
1 x Capacitor 0402	0.3	
Total	23.5	
Total are on PCB	60.0	

Table 5: Area Occupied by DAC's circuit

Table 5 shows the area occupied by each element of the DAC's circuit.

Important: The 0805 resistances can be replaced with some in 0402.

Part type	Area [mm2]	Code
Comparator	9.0	MAX9107
4 x Resistance 0805	16.1	
1 x Capacitor 0402	0.3	
Total	25.4	
Total are on PCB	73.5	

Table 6: Area Occupied by Comparator's circuit

Table 6 shows the area occupied by each element of the Comparator's circuit. Important: The 0805 resistances can be replaced with some in 0402.

9 Future improvements

When it comes to future improvements I think there are still many other things which can be improved, especially in terms of ripple and efficiency: Replacing the comparator with an ADC might bring some advantages. Basically, the comparator is a 1-bit ADC and the information provided by it is very poor. Consequently, the complexity of the control algorithm is limited due to the poor feedback information. By knowing the precise value of the output voltage, a more accurate control can be achieved. However, some research should be done here as those advantages might not justify the additional outlay and, in addition, the space occupied by an ADC.

Integrating a current measurement circuit. The output ripple and efficiency are powerfully dependent on the load current. By knowing the value of the instantaneous current the algorithm's parameters can be adapted even during running. A look-up table can be developed inside the FPGA so that the algorithm will know what parameters to use depending on the load current.

10 Building the project

In order to test the project, several steps should be followed. If you want to test the converter through the UART to I2C bridge, you should:

1. Get the board and equip it
2. Download the files in the **UART to I2C Bridge** folder
3. Create a new project in your HDL synthesis tool (I used Xilinx Vivado)
4. Import all the ".v" files from the just downloaded folder as design sources
5. Import "constr.xdc" as constraints file.
6. Generate the bit-file and upload it
7. Connect the board to the JB PMOD connector of the board
8. Connect the computer to the UART port of your FPGA.
9. Open the serial console and configure the DAC. Example: send "SC0W07WFFW07WFFW07WFFW07WFFWs" to set the output voltage to 2.05V

If you want to test the converter through the I2C Slave module, you first have to know that you will need a separate I2C Master to command the converter and the DAC (which are two slave devices on the same bus). For instance, you

can use Arduino Uno, because it allows you to send custom I2C commands. You should:

1. Get the board and equip it
2. Download the files in the **Converter I2C Slave** folder
3. Create a new project in your HDL synthesis tool (I used Xilinx Vivado)
4. Import all the ".v" files from the just downloaded folder as design sources
5. Import "constr.xdc" as constraints file.
6. Manually connect the SDA and SCL of the DAC and converter's slave to your I2C bus.
7. Generate the bit-file and upload it
8. Connect the board to the JB PMOD connector of the board.
9. Configure the DAC and converter from yout I2C Master.

Note: If you have other FPGA board, you can also realize the setup. You have to manually connect the FPGA I/Os to the board pins. Use the schematic and layout from *"Layout Files/PCB_Project.pdf"* to identify the function of each pin of the board.

References

[1] Texas Instruments, *Application Report* **SLVA477B–December 2011–Revised August 2015**, "Basic Calculation of a Buck Converter's Power Stage".

[2] Texas Instruments, *Application Report* **SLVA390–February 2010**, "Calculating Efficiency".

[3] Texas Instruments, *Application Report* **SLVA630A–January 2014–Revised October 2014**, "Output Ripple Voltage for Buck Switching Regulator".

[4] Analog Devices, *Application Note* **AN-1144**, "Measuring Output Ripple and Switching Transients in Switching Regulators".

[5] https://e2e.ti.com/support/power_management/simple_switcher/w/simple_switcher_wiki/2243.understanding-measuring-and-reducing-output-voltage-ripple