Università degli Studi di Padova

Dipartimento di ingegneria dell'informazione

Università degli Studi di Padova

Dipartimento di ingegneria dell'informazione

Laurea magistrale in

Ingegneria Elettronica

PWM controller design with Zynq SoC

Relatore:
Prof. Simone Buso

Candidato:
Antonio Minighin

Anno Accademico : 2022/2023

Data: 13 Luglio 2023

Contents

List of Figures

Listings

Chapter 1

Introduction

Electronic systems are nowadays widely used in energy management, motor drives, consumer and industrial applications to monitor, optimize and automate processes. The key enabling technology, that allowed electronic systems to break into several markets, are processors, memories and digital logic capable of performing complex tasks, storing data and execute instructions. This has been made possible by the advancement of semiconductor manufacturing technology and system architectures. Some interesting features that came up with modern systems are:

- **Integration:** to combine many components into a single system which led to a rapid increase in the number of functionalities
- **Configurability:** to reconfigure the behavior of a system which allows it to perform different functions
- **Reliability:** to ensure equal performance under different conditions
- **Memory:** to store and retrieve information and use them in multiple ways

As a result, electronic systems have become smaller, faster, flexible and smarter, making them ideal to be used in several applications.

Each application requires a specific trade-off between performance, cost optimization and power efficiency, which is achieved only with specific devices. Based on the set of available resources and the processor speed, CPU can be grouped into MCU, DSP and APU. Microcontrollers are built with specific blocks optimized to interact with external systems, such as communication interfaces, timers, ADC, DAC and input output ports to send and receive signals. DSP are optimized to work on data streams to achieve high throughput on intensive and complex tasks. This can be done by using high speed communication interfaces and processing system with specialized architectures, like FPU and super-scalar pipeline with SIMD execution. Finally, APU are built with cache memory, high speed communication interfaces, DMA and even branch speculators to move data and execute complex programs or full operating systems. Modern devices, called SoC, integrate CPU, GPU, memory controllers, I/O controllers, and analog-mixed signal interfaces, to reach high performances with low power consumption, thanks to the tightly coupled components.

The semiconductor industry allows the customization of ASIC with the desired CPU and its peripheral components, or at least it allows choosing between different pre-configured options called ASSP. These integrated circuits are ready to be mounted on PCB and programmed to create complete electronic systems with few external components. However, the design of a full custom ASIC is expensive because NRE costs increased with VLSI, making SoC available only for applications with a significant volume size market. Already in the 80s, when the VLSI era began, the problem of NRE costs was solved with PL devices, ready-made products with interconnected configurable blocks used to implement different functions. With programmable devices, like GAL, CPLD and FPGA, NRE costs are absorbed by the manufacturing company and subdivided through the production volume, making them affordable also to niche markets. VLSI also required automatic tools the keep design complexity under control, bring down NRE costs, and follow the built-to-order model of production.

To deal with complexity, HDL were developed to describe the architectures at higher levels of abstraction, which are then compiled and implemented with the desired technology. At first, HDL were used just for circuit simulation to deal with high speed circuit and signals with fast transient. Only lately, compilers capable of synthesizing circuit structures were developed, while others were used to map and optimize the circuit description into hardware. As a consequence, even if behavioral simulation can describe every kind of digital circuit, there are standard and strict rules to code a digital circuit such that it can be implemented. Modern compilers can even understand C language and use HLS flow to produce an RTL design that can be implemented on the FPGA. The HLS is a very efficient design flow that well matches with the configurability of FPGA, and it can be used to accelerate complex algorithms on hardware, for example machine learning algorithm or 3D graphic algorithms. VHDL and Verilog languages are still used because the output is well predictable, and they allow debugging and to modify the circuit to meet timing and performance requirements.

Among all PL devices, FPGA devices emerged because they enable rapid design, functional verification, customizable hardware, and dynamic reconfiguration. FPGA are built with many CLB, usually embedding LUT and DFF, connected through a switching matrix and I/O ports in a complex fabric of routing lines. Xilinx solution, that will be produced up to 2035, is the Zynq All-Programmable SoC, an APU tightly coupled with an FPGA and other programmable blocks, to enhance the performance of reconfigurable devices with modern technology and architectures. This device can reach good performances on a huge number of applications, but its strength is the signal processing ability of both the processor and the programmable blocks.

The goal of this project is to develop a digital PWM controller system for applications where fast signal processing is required. Control applications require real-time and low latency system response both achievable only with highly specialized devices. Modern MCU are typically well suited for control systems because they integrate ad-hoc components that relieve the processor from repetitive tasks. MCU resources like ADC, timers, comparator, communication interface and even DMA, can be extremely optimized on performance and efficiency. However, in MCU and generic CPU based devices, the control algorithm always run in the processor because it depends on the specific application. When dealing with real time tasks, the algorithm can rapidly overload the CPU, especially in multitasking applications, where processing requirements rapidly increase. Furthermore, low latency forces the CPU to work fast for small-time intervals and rest for long periods. FPGA allows offloading common task like processing from software to hardware and increase system response by building ad-hoc data flow subsystem from the ADC to the controller output. Zynq makes possible to implement both software and hardware controller because it combines a PS tightly coupled with the PL and the ADC. Software and hardware can also adapt to perfectly interact each other, because they can be designed, tested and optimized concurrently in the same system.

PWM is the most common way to control switching devices because of its lower complexity when compared of other type of modulations, like PFM or PDM, which leads to relatively easy control algorithms. Power electronics systems are part of numerous applications, ranging from motor control to renewable energy systems, battery chargers, and industrial automation. Efficient power conversion is achieved through the utilization of switching techniques, specific circuit topologies, and control algorithms. For example, motor control applications require controlling speed, torque, and direction, with advanced algorithms, like vector control, or simpler one like PID control.

1.1 Zynq Technology

Zynq-7000 is a SoC divided in two parts, the PS and the PL, made of an APU coupled with an FPGA fabric and other programmable blocks. The PS is optimized to work on data and interact with different sources by using cache memory and

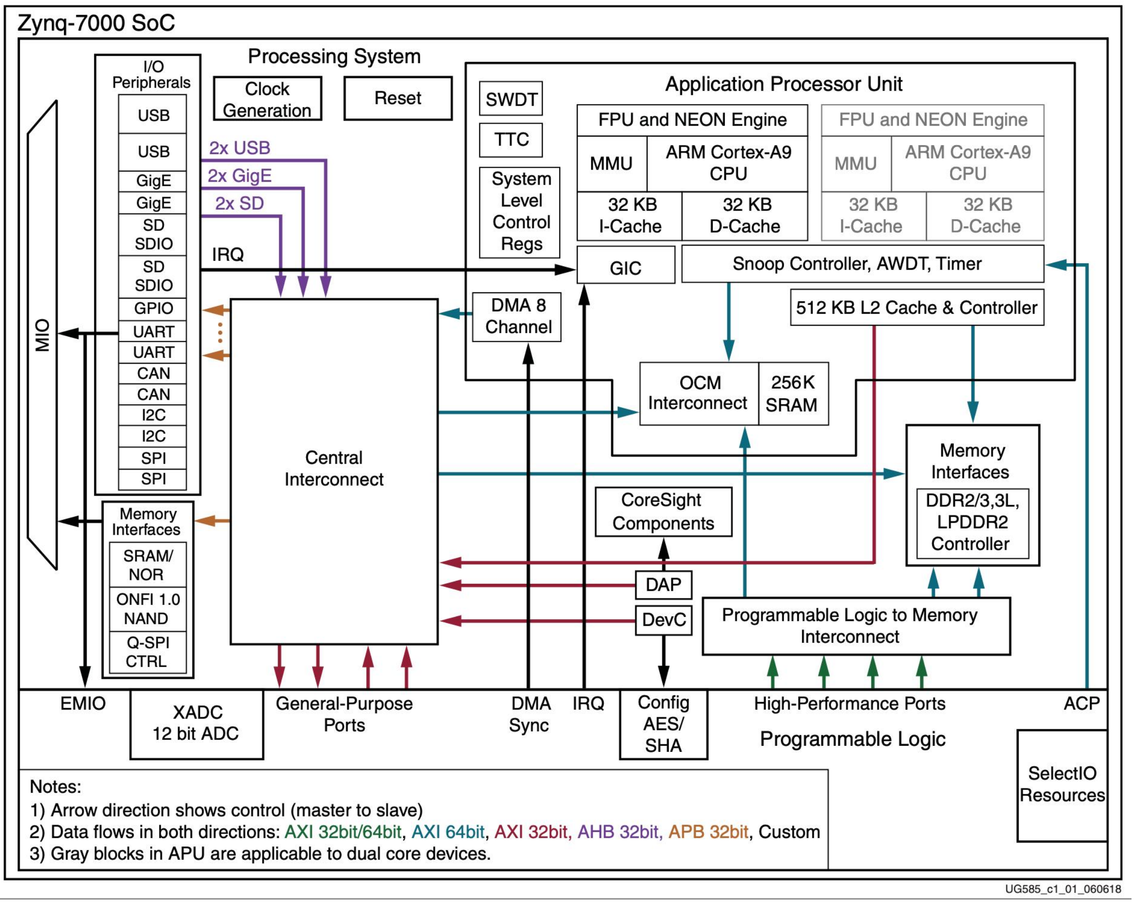


Figure 1.1: Zynq-7000 SoC block diagram showing the interconnections of the main components. Reference manual image [17]

several communication interfaces and controllers, in depth:

- **Processor:** A dual-core ARM-cortex A9 architecture with FPU and NEON engine
- **Memory:** Two level cache memory for instructions (I-Cache) and data (D-Cache), On-Chip-Memory OCM and SRAM
- **Controller:** A SCU for cache coherency, a GIC to dispatch and handling interrupts and a DDR controller for external memory
- **IO Ports:** Some IO peripherals and memory interfaces that shares a MIO port and 3 types of ports for PS and PL interaction, called General-Puropse GP ports, High-Performance HP ports and Acceleration Coherency Port ACP

The ARM-cortex A9 has a dual-issue, partially out-of-order pipeline extended with DSP instructions, FPU and NEON engine to perform SIMD operations. The two cores of Zynq-7000 can work in synchronous or asynchronous mode to build complex behaviors. The PL is based on the Artix-7 family FPGA fabric optimized for low power and high throughput operations, mainly composed of:

- **CLB:** configurable logic with 6 input LUT, shift registers and cascade adders
- **DSP48E1:** 4 inputs DSP block with pipeline pre-adder, signed multiplier and a final ALU with 48 bit accumulator
- **BRAM:** 36Kb dual port block RAM for read-while-write operations
- **XADC:** two 12 bit 1 Msample/s channel synchronous A/D converters with 2 fast dedicated lines and 16 multiplexed auxiliary input

PS and PL communicate over the AXI protocol implemented by the GP port with 2 managers and 2 subordinates or the HP port with 4 subordinates. The GP manager interfaces are divided into `M_AXI_GP0` and `M_AXI_GP1`, which are memory mapped, and accessible to the PS, respectively at the addresses `0004_0000` to `0007_FFFF` and `0008_0000` to `000F_FFFF`.

The PL is configured with a bitstream generated by the compiler from the HDL design and optimized for the FPGA technology, while the PS can run bare metal programs, RTOS, or Linux OS. Zynq systems are designed and programmed with Vivado and Vitis, which are respectively an IP block deign based CAD tool to configure the PL, and an IDE with the SDK to program the PS. The bitstream configuration process is always managed by the PS, which should always be activated first, and can reconfigure the PL on demand with the DFX, a reconfiguration process that can keep some PL parts active while it changes others.

The debugging of the PL part can be done through an ILA, a Xilinx soft IP that can be removed once the functional verification is completed. The ILA is a powerful debugger that can also interact with the PS to block and restart the software execution. However, it can degrade the system performance because it requires routeing more signals to specific elements, usually BRAM and FTM, that might be distant from the functional block.

1.1.1 AXI Interface

The AXI protocol allows transferring data between a manager and a subordinate by using a handshake rule and some sets of signals. These signals are grouped in address and data channels for read and write transactions. The address mechanism allows multiple subordinates to be connected to an AXI interconnect architecture, making them reachable from different managers. There are 3 variants of the AXI protocol, AXI, AXI-Lite and AXI-Stream, which are used to achieve fast communication or complex architectures. The AXI and AXI-Lite protocols are both memory mapped variants which differ only by some signals and the ability of performing burst data transfer. On the other hand, the AXI-Stream protocol enables unidirectional high speed data streams from a manager to a subordinate, by getting rid of

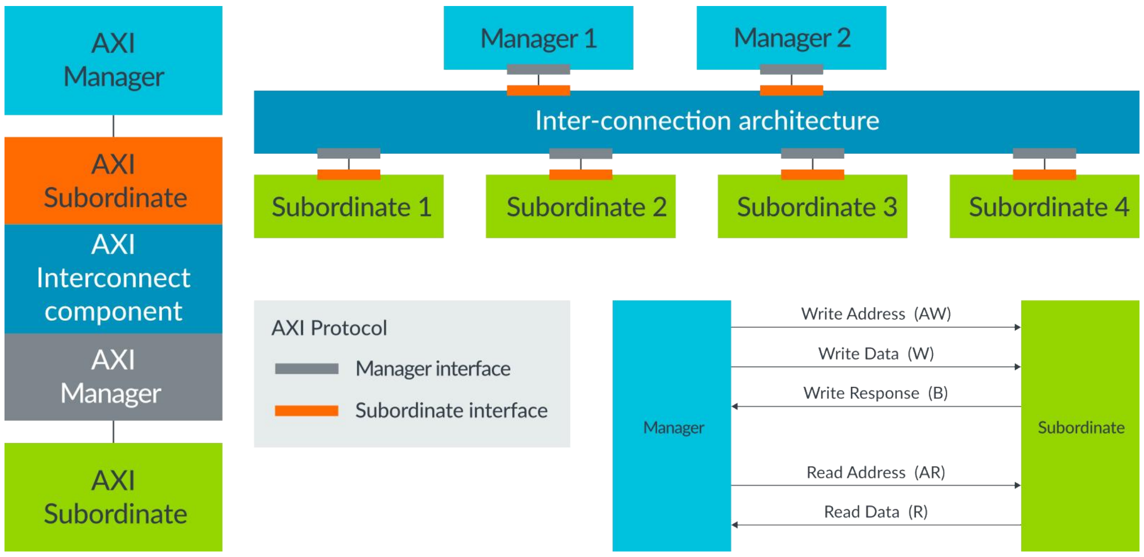


Figure 1.2: AXI functional blocks showing the connections and the channels of the communication protocol. Reference guide image [2]

the addresses and using only one channel with a restricted set of signals.

An AXI transaction is made of several transfers, containing an address and one or more data. The handshake is implemented in each channel by using the `READY` and `VALID` signals to confirm the transfer of the `ADDR` or the `DATA` signals. The `VALID` signal is controlled by the channel source and used to validate a transfer, while the `READY` signal is controlled by the channel destination and used to confirm a transfer. All signals are caught at the rising of the clock `ACLK`, therefore they must stay asserted at least for one clock period. The handshake procedure follows some simple rules:

1. A source must keep asserted `VALID` until the transfer ends
2. A source cannot wait for `READY` to be asserted before asserting `VALID`
3. A destination can wait for `VALID` to be asserted before asserting `READY`
4. A destination can deassert `READY` before `VALID` is asserted
5. A transfer ends when both `VALID` and `READY` are asserted

In the AXI-Stream protocol, the managers are always the source and the subordinates are the destination, therefore, the handshake rules can be seen as a good practice for an efficient chain of order. In fact, a mnemonic way to remember those rules is by thinking of rules number 1 and 2 as:
"The manager should schedule the operations and stick with the plan"
While number 3 and 4 can be thought as: "The subordinate is generally lazy but could be busy at any time"
These are general rules, but they can be used to build complex and efficient behavior without deadlocks in the processing chain.

1.1.2 XADC

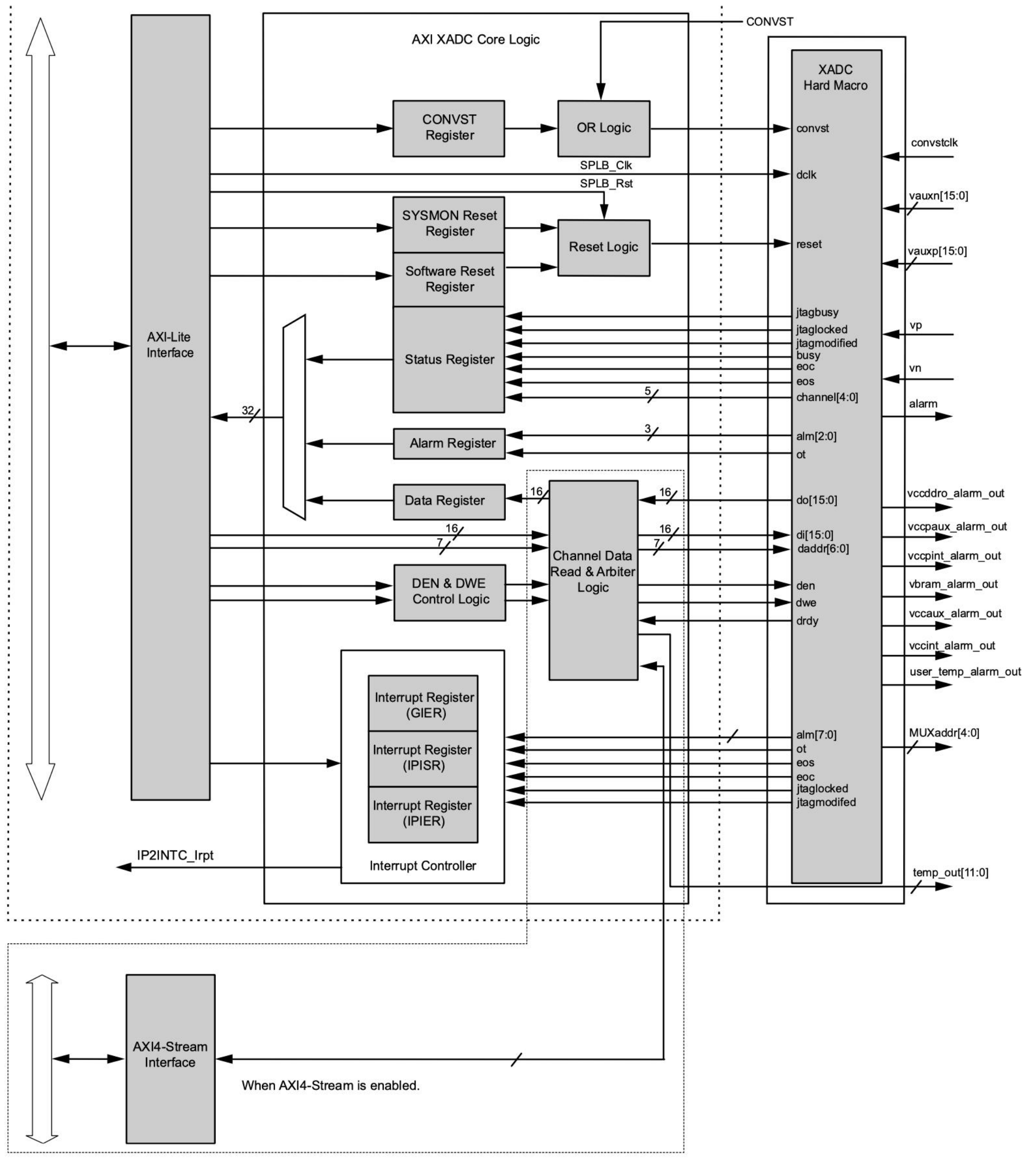


Figure 1.3: XADC with AXI interface wrapper diagram, showing the main registers, the signals and the buses. Product guide image [15]

The XADC is embedded in the PL part, and it can be controlled through the DRP both by the PS and the PL. The PS is directly connected to the DRP through the JTAG, but this communication is slow and used only to monitor the system parameters, like the chip temperature and the supply voltages from embedded sensors. To enable high speed communication with the PS, the XADC can be wrapped with an AXI-Lite subordinate interface and connected to the AXI GP port. In this case, the PS is the manager interface that can request data and or send commands ad lib. Moreover, since the GP port is the only port with a master interface, this connection is usually necessary. The same AXI wrapper can also implement an AXI-Stream manager, which can be used to build a processing chain in the PL and to stream data to the PS through the HP port. The two interfaces together can be used to fetch and stream data asynchronously from two independent interfaces and route them to different paths.

The Xilinx LogiCORE IP for the AXI XADC is a ready-made wrapper for reading the results of the conversions or the status information of the XADC. This IP also embeds an interrupt controller that can be configured by 3 registers, the GIER, the IPISR and the IPIER, and attached to the GIC pin dedicated to the PL part. The GIER enables the `IP2INTC_irpt` line, which is driven by the IPISR interrupt event, if the corresponding bit of the IPIER is enabled. An interrupt event could be an alarm or the end of a conversion, which sets the corresponding bit in the IPISR to send an IRQ. When the IRQ is received, the IPISR must be cleared, otherwise the `IP2INTC_irpt` line will remain asserted and the IRQ pendent.

The XADC is clocked by the DCLK, which is used to communicate over the 16 bits DI and DO IO ports controlled by the DEN and the DWE signals. The ADCCLK is the main clock of the A/D conversion, it is obtained by a $4\times$ pre-scaling of the DCLK and must be in the range from 4MHz to 26 MHz, therefore the DCLK must be in the range of 16 MHz to 104 MHz. The A/D conversion process can be divided in two phases, an acquisition phase and a conversion phase, which are done by two separate circuits, a T/H amplifier and the digital converter. The conversion phase takes always 22 ADCCLK periods to convert the signal, during which the BUSY flag is asserted. The converter stops when the XADC deasserts the BUSY flag and the converted channel can be read from the 5 bits of the CHANNEL signal. After the conversion, the XADC takes other 16 DCLK cycles, equivalent to 4 ADCCLK cycles, to store the data in the relative register, and only at this point the EOC flag is asserted.

The XADC can be configured to work in continuous sampling mode or event driven mode, and can handle two simultaneous conversion to keep phase relation. In event driven mode, the sampling instant is determined by the signal CONVST rising edge, after which the conversion starts at most one DCLK cycle after, and the T/H amplifier is free to acquire the next sample. During the acquisition phase, the T/H amplifier charges a capacitor that will hold a constant value for the conversion phase. The T/H amplifier requires a minimum settling time to follow the input variations, which limits the time between the EOC of the previous channel and the start of the actual conversion. For a converter with N_{BIT}, the correct settling time t_s allows the T/H amplifier output to follow the input value with an error that is less than half LSB. The T/H amplifier can be approximated by a gain controlled generator with an input capacitance of $C = 3pF$, therefore the output follows a first order response and the IO error decreases exponentially with time

$$\begin{aligned} V_O &= (1 - e^{-t/\tau})V_I \\ e^{-t/\tau} &\leq \frac{1}{2^{N_{BIT}+1}} \\ t_s &\geq \tau \ln 2^{N_{BIT}+1} \end{aligned} \tag{1.1}$$

The time constant $\tau = RC$ is fixed by the source impedance of the circuit connected to the channel. The source impedance has a minimum value that depends on the selected channel, which is $100k\Omega$ for the auxiliary multiplexed inputs and 100Ω for the dedicated input.

1.1.3 DSP48E1

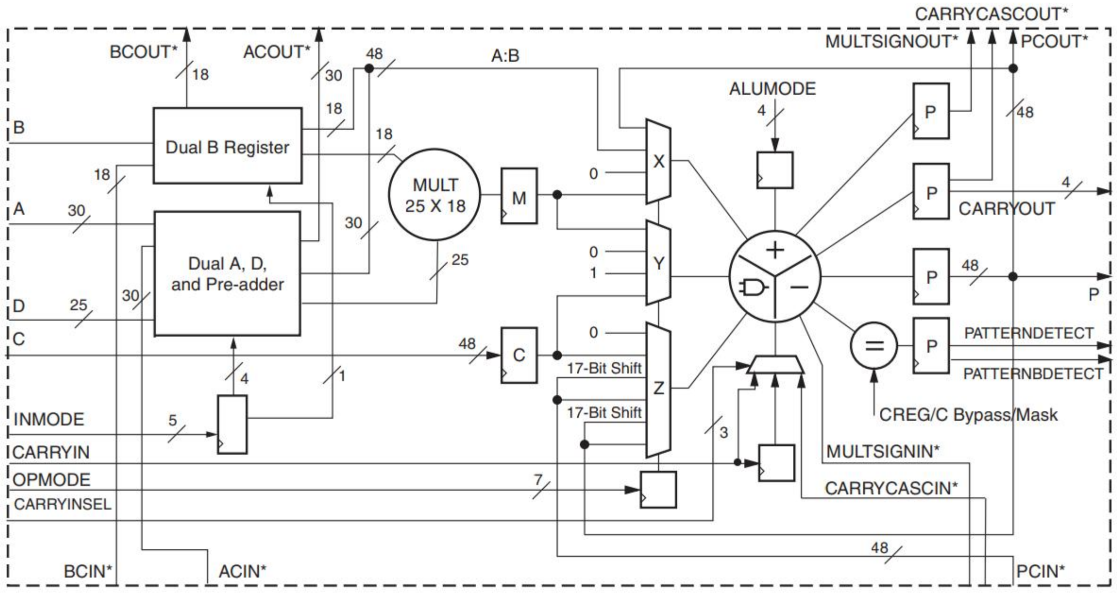


Figure 1.4: DSP48E1 simplified structure showing the pipeline, the data width and the possible configurations. Reference guide image [11].

The DSP48E1 is an optimized processing block for high speed and high resolution operations, that is built with a pipeline structure with a pre-adder, a multiplier and a ALU. There are 4 optionally buffered inputs with different width, A of 30 bits, B of 18 bits, C of 48 bits, and D of 25 bits, which are routed to different paths to build complex architectures. A and D feed a 25 bit pre-adder, that can also be used as a subtractor depending on the input mode selector. The pre-adder result has a width of 25 bits, without an extra bit to prevent the overflow of the sum, therefore both inputs must be saturated to 24 bits values. The pre-adder result is optionally buffered and routed to the 25x18 two's complement multiplier together with the 18 bits input B that can be optionally buffered twice for time delay matching. The 43 bits multiplier output is optionally buffered, signed extended to a 48 bit number and routed together with the 48 bit input C to the ALU. The result is stored in a 48 bit register, directly routed to the output or through a comparator with a preloaded matching value. The 30 bits of A can be concatenated with the 18 bits of B to build a 48 bit data that is routed directly to the ALU, capable of performing SIMD operations with the C input, like 2x24 bits or 4x12 bits sums. Another possibility is to route back the 48 bit bits of the output to the ALU and use the output register like an accumulator.

Chapter 2

PWM control technique

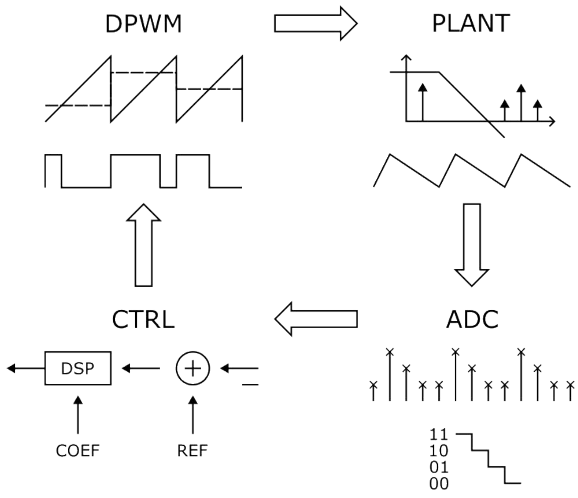
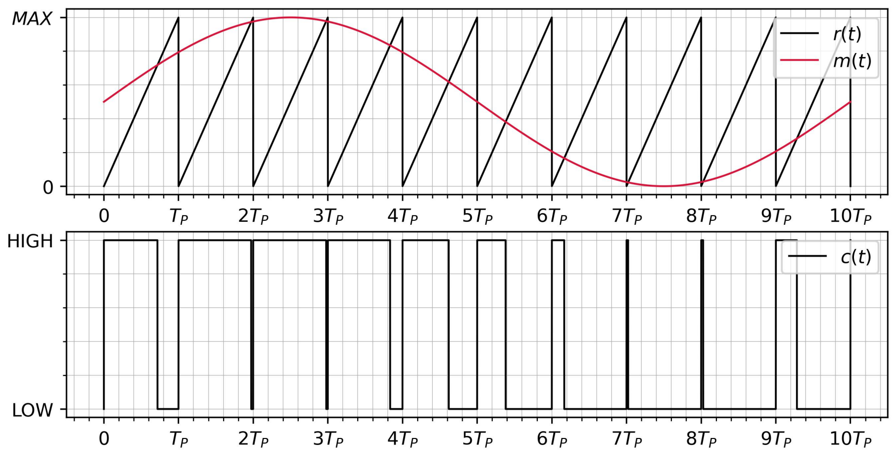


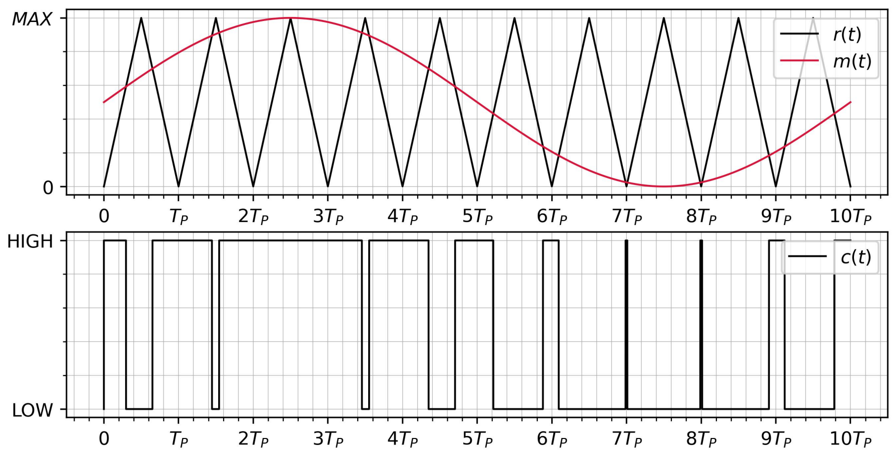
Figure 2.1: DPWM control technique principles

2.1 Pulse Width Modulation

The PWM is a technique to encode the signal information in the width of a series of pulses with fixed frequency and amplitude. A comparator and a ramp generator are the only two elements required to implement the PWM, making it simple and popular among all the other binary modulation techniques. The modulating signal $m(t)$ is continuously compared with the ramp, and the output of the comparator $c(t)$ is driven LOW when the carrier exceeds the modulating signal. The ramp $r(t)$ may be a sawtooth or a triangular wave with fixed period T_P, which allows to obtain a rectangular wave with different symmetry properties and with a fixed repetition rate F_P. Using the saw ramp the output rectangular waveform turns to be asymmetric and, depending on the counting sequence and how the comparator output is set at the beginning of each period, two type of modulating options are possible: trailing-edge and leading-edge modulation. Trailing-edge modulation is obtained driving the comparator output HIGH when the modulation period begins, which can be done by starting the counter from zero and counting upward. Similarly, the leading-edge modulation starts each period with the comparator output LOW, which is done by initializing the counter to its maximum value and let it count downward. By using a triangular ramp, the output rectangular waveform becomes symmetric, and both edges are modulated in one modulation period. In this case there is no more a trailing-edge and a leading-edge, but there are still two possible counting starting point that lead to a centered symmetric or lateral symmetric output waveform. Note that in practice, there is no need to implement two mechanisms to make the counter start from zero and count upward and then change direction or vice versa, because it only depends on the reference point of view.



(a) Trailing-edge PWM with saw ramp



(b) lateral-edge symmetric PWM with triangular ramp

Figure 2.2: PWM of a sinusoidal signal

Carrier Ramp	Modulator Input	Comparator Output	Trailing Edge	Leading Edge
$r(t)$	$m(t)$	$c(t)$	$c(nT_P) = HIGH$ $r(t) \to UP$	$c(nT_P) = LOW$ $r(t) \to DOWN$

Table 2.1: PWM signal names and output properties

Due to its low complexity the PWM can be realized with analog, digital or mixed signal electronics. A fully analog implementation requires a comparator with high slew rate and high bandwidth fed by a precise continuous time ramp. Analog systems can reach fast modulation periods, but they suffer from mismatches and offsets or signal interference and noise. On the other hand, digital solutions use reliable numeric comparators and counters to generate the ramps. Digital counter needs to be clocked at high speed to reach the same performance of a simpler analog ramp generator, but there is a breakeven point where the analog solution becomes more greedy in terms of resources. In fact, VLSI benefits the digital counter making it faster, while analog ramp generators require more and more power or complex architectures to operate at high frequencies.

Hybrid solutions are also possible; usually, the generation of a modulating signal is easier with digital electronics, which can be converted by a DAC and fed to the analog comparator. This is the classic way to interface a digital controller keeping the analog modulator, which is possible because the modulating signal is much slower than the ramp. On the other hand, the conversion of a digital ramp to feed the analog comparator requires high speed DACs, that are less competitive with respect to an analog ramp generator, making this solution not useful.

2.1.1 Modulator Model

A model is used to describe and analyze the behavior of the modulator, understand how to use it and how it interacts in the full system. To generalize the model, the comparator output $c(t)$ can be considered a signal that can assume only the values $HIGH = 1$ and $LOW = 0$. Then, the comparator output drives a switch that modulates the switching variable $v_s(t)$ performing a non-linear operation

$$v_x(t) = v_s(t)c(t) = \begin{cases} v_s(t) & when\, c(t) = 1 \\ 0 & when\, c(t) = 0 \end{cases} \tag{2.1}$$

The comparator on-off behavior can be modelled like in Fig. 2.3, as a normalization stage after which is introduced a distortion $e(t)$ due to the comparator behavior. This distortion is essential for the modulation, and it is correlated to both the carrier and the modulating signal in a non-linear manner, however there are some conditions

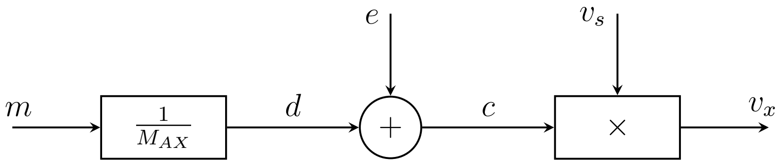


Figure 2.3: Modulator model showing the superimposed distortion introduced by the comparator and the switching stage.

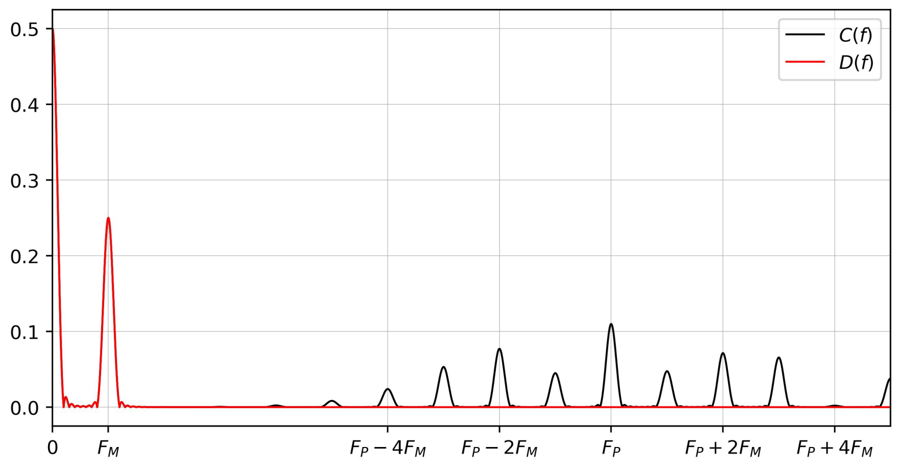
when the two contributions can be separated, and the original signal restored with a demodulation technique. By looking at the spectral decomposition in Fig. 2.4a, the comparator output can be analyzed by separating it in two components: the contribution $\bar{c}(t)$ due to the low frequency spectral components in the baseband, and that due to the high frequency spectral components $\hat{c}(t)$ around the modulating frequency F_P

$$c(t) = \bar{c}(t) + \hat{c}(t) = d(t) + e(t) \tag{2.2}$$

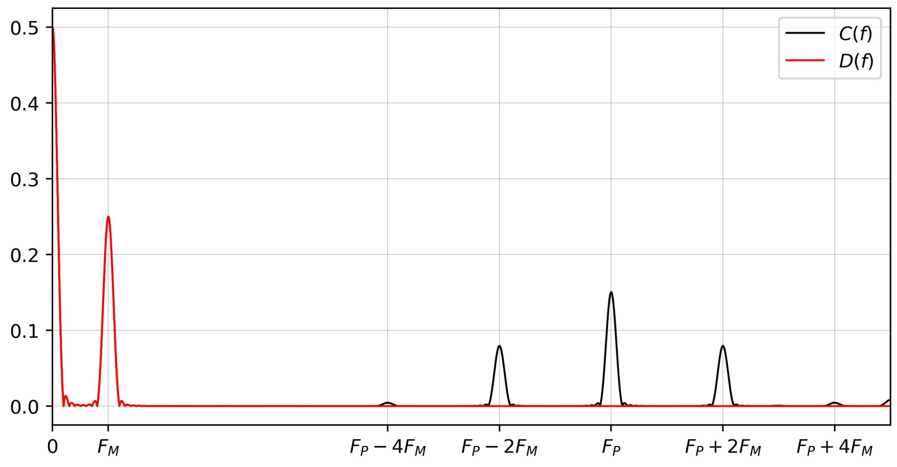
The modulation process is supposed to encode the modulating signal into the baseband frequency, therefore $\bar{c}(t) = d(t)$, and the distortion into the high frequency component $\hat{c}(t) = e(t)$. This happens only under two conditions: high modulation frequency (HMF) and input limited bandwidth (ILB). The HMF allows to shift the error spectrum far from that of the modulating signal, such that it doesn't alias into the baseband frequencies. The ILB condition prevents the PWM to spread the harmonics of the modulating signal, starting from the modulating frequency, towards the baseband. Fig. 2.4 shows that the generated harmonics are located at a relative distance kF_M from F_P, with F_M a generic tone of the modulating signal. Without considering other effects such as intermodulation harmonics, the error spectrum can be considered extinguished when $k = -4$.

The simulated model also shows that the triangular ramp gives higher performances with respect to the saw ramp with the same switching period, however, this may be misleading because the saw requires half bandwidth compared to that of the triangle. In practice, giving the same technology it is always possible to implement a saw ramp that goes two times faster than a triangular ramp, making both techniques useful for different situations.

Looking at equation (2.1), it is clear that the spectrum of the modulated variable will be the convolution $V_x(f) = C(f) * V_s(f)$. This complicates again the possibility of separating the spectral components, however in many practical situations there are conditions and properties that simplifies it. For example, when the switching variable is constant $v_s(t) \equiv V_s$, the switch operates as a constant gain for the



(a) Sinewave spectrum with saw ramp modulation



(b) Sinewave spectrum with triangular ramp modulation

Figure 2.4: Spectrum of a pulse width modulated sinewave

comparator output.

$$v_x(t) = V_s c(t) \tag{2.3}$$

On the other hand, when the modulating signal is constant $m(t) \equiv M$, the comparator output has a constant mean value, that is the ratio of the pulse width T_W and the modulation period. This ratio is called the duty-cycle D, and it is also related to the modulating signal

$$D = \frac{T_W}{T_P} = \frac{M}{M_{AX}} \tag{2.4}$$

In general, if the switching variable is relatively slow $v_s(t) = \bar{v}_s(t)$ with small amplitude and limited bandwidth, its spectrum combined with the distorted spectrum doesn't alias in the baseband, and it is possible to separate the modulated variable in the low spectral component $\bar{v}_x(t)$ and the high spectral component $\hat{v}_x(t)$

$$\begin{aligned} \bar{v}_x(t) &= \bar{v}_s(t)d(t) \\ \hat{v}_x(t) &= \bar{v}_s(t)e(t) \end{aligned} \tag{2.5}$$

All these properties can be used to linearize a model for the low frequency components, under the hypothesis of HFM and ILB, and find a transfer function around its operating point $Q = (D, V_s)$. This can be done by considering all signals made of a constant component and a small variation around it, and then truncating the Taylor series expansion to the first order

$$\begin{aligned} d(t) &= D + \tilde{d}(t) \\ \bar{v}_s(t) &= V_s + \tilde{v}_s(t) \\ \bar{v}_x(t) &= V_x + \tilde{v}_x(t) \\ f(d, \bar{v}_s) \approxeq f(D, V_s) &+ \frac{\partial f}{\partial d}\bigg|_Q (d - D) + \frac{\partial f}{\partial v_s}\bigg|_Q (\bar{v}_s - V_s) \end{aligned} \tag{2.6}$$

As a result, the expression of the modulated variable becomes that of two input system with an offset determined by the operating point.

$$\bar{v}_x(t) = d(t)\bar{v}_s(t) = DV_s + V_s\tilde{d}(t) + D\tilde{v}_s(t) \tag{2.7}$$

Again, it should be noticed that, if one of the two variable is kept constant $\tilde{d}(t) = 0$ or $\tilde{v}_s(t) = 0$, the other is simply gained, and this holds without approximations.

2.1.2 Digital PWM

The DPWM realization closely resembles its continuous-time analog counterpart, with the key difference being the finite resolution of time and amplitudes. Typically, the modulating signal is generated using an M_{BIT} counter that increments or decrements at each clock cycle T_{CLK}. The counter maximum value M_{AX} falls within the range of $[0 : 2^{M_{BIT}} - 1]$, and the time resolution is dictated by the clock period T_{CLK}. As a result, the modulating frequency F_P is constrained by the counter number of bits.

$$\begin{aligned} F_P &= \frac{F_{CLK}}{M_{AX} + 1} \\ T_P &\leq 2^{M_{BIT}} T_{CLK} \end{aligned} \tag{2.8}$$

These equations hold for the modulation with the sawtooth ramp, but the modulation period should be doubled when using the same counter resolution with a triangular ramp.

The set of possible values for the modulating signal is also limited by M_{AX}, leading to the concept of the equivalent number of bits $ENOB$. This qualitative value is defined as the base-2 logarithm of the number of possible values that the modulating signal can assume. This definition combined with the modulating period constraint (2.8), shows that there are no ways rather than increasing the clock frequency to keep both fast modulation and high resolution

$$ENOB = \log_2(M_{AX} + 1) = \log_2(\frac{F_{CLK}}{F_P}) \tag{2.9}$$

Although the $ENOB$ well defines the performance of the DPWM, there are other dynamics aspects that impacts on the system behavior, like the dead-time of the comparator update and the acquisition time instant. In most of the digital implementations, the modulating signal is updated once per modulation period, which leads to an input-output delay effect. In a continuous-time PWM this effect doesn't occur because the comparator always compares the instantaneous value of the modulating signal. On the other hand, the DPWM compares the value fixed at the beginning of the modulating period introducing a variable delay T_D. When using the saw ramp, this delay depends on the operating point $T_D = DT_P$, while using the triangular ramp, this delay becomes constant and equal to $T_D = T_P/2$ because the delay of the first edge is compensated by an anticipation of the second edge.

To mitigate the delay effect, the update frequency F_U must be increased, making the digital DPWM more similar to a continuous-time version. By updating the counter more frequently, a variation of the modulating signal is followed with less delay, and the DPWM becomes more reactive. This improvement isn't free, because the processing of each new value and the transfer to the comparator should be completed within each updating period. A fast processing system with fast communication interfaces may sustain the update rate, however the only way to keep up with high frequency update is to use ad-hoc DSP subsystems.

2.2 Control System

Control systems are designed to regulate the output of a process by operating in open-loop mode or a closed-loop mode. Open-loop control system works without feedback from the process, relying solely on the process model, therefore they are used when the process is well predictable, stable and without environmental disturbances. Sometimes a control system works in open-loop mode because a process state can't be measured, or it sends feedback signals only while it is running. On the other hand, closed-loop control systems operate by sensing the output and by adjusting the process input until the desired set point is reached. The feedback allows taking into consideration also environmental changes and disturbances that are affecting the process, and possibly compensate them.

In closed-loop control systems, the process output y is measured and compared to the desired set point u. The negative feedback control uses the difference of these signals to feed the controller that modifies the process input until the error e becomes zero. The controller is usually designed in the forward path such that the closed-loop transfer function $A_F(s)$ can be forced to follow the desired feedback behavior when the loop gain $T(s)$ is high enough.

$$\begin{aligned} A_F &= \frac{A}{1+A\beta} = \frac{1}{\beta}\frac{T}{1+T} \\ A_T &= \lim_{T\to\infty} A_F = \frac{A}{T} = \frac{1}{\beta} \end{aligned} \tag{2.10}$$

This ideal feedback behavior is called the theoretical gain $A_T(s)$, defined as the ratio between the open-loop gain $A(s)$ and the loop gain, that turns to be noise gain, the inverse of the feedback gain $\beta(s)$. On the other hand, when the loop gain drops to zero, the closed-loop gain will follow the open-loop gain, behaving like no feedback is applied.

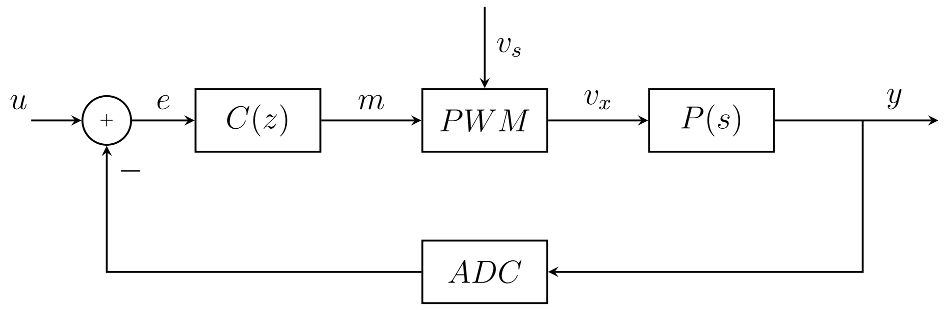


Figure 2.5: Digital PWM control system with discrete time controller

Fig. 2.5 shows the digital PWM control system which falls in the category of a mixed-signal non-linear system. It is composed of a digital controller $C(z)$, a PWM that acts on the analog process $P(s)$, and the digitalized output that feeds back the controller. The purpose of a PWM control system is to control the average value of the process output $\bar{y}(t)$, by controlling the average value of the modulated variable $\bar{v}_x(t)$. In some particular application, it is also required to control the process against the variation of the switched variable $v_s(t)$. An example could be a DC/DC converter, where the output voltage should be kept stable even if the input voltage drops. A more complex example is the PFC, that forces the output average current to follow an input proportional to the voltage $u = kv_s$. PWM systems are always MIMO systems, but, except from specific applications, one of the two inputs is always chosen as the control signal while the other is considered a disturbance. In fact, it is always possible to build a controller that eliminates the disturbance, while it can track the input reference.

2.2.1 Controller Design

The most straightforward method to design a controller, which can be implemented by a digital system through an algorithm, is to emulate the behavior of a continuous time controller and use a discretization process that matches the frequency response. The two main transfer functions of the PWM are evaluated from the PWM linearization, and they are called the control-to-output gain $G(s)$ and the open-loop susceptibility $N(s)$. The choice of the names is due to the path of the relative signals, the first is the signal that controls the process while the second is usually considered a noisy signal injected in the process.

$$\begin{aligned} G(s) &\triangleq \frac{\tilde{y}(s)}{\tilde{d}(s)}|_{\tilde{v}_s=0} = V_s P(s) \\ N(s) &\triangleq \frac{\tilde{y}(s)}{\tilde{v}_s(s)}|_{\tilde{d}=0} = DP(s) \end{aligned} \tag{2.11}$$

In the PWM model, the comparator introduces a scale factor M_{AX}^{-1} which normalizes the modulating signal to the auxiliary variable $d(t)$. The physical dimension of this quantity depends on the PWM implementation, but it is always related to the ramp or the comparator input signals. In an analog modulator the ramp is usually a voltage compared to another voltage signal, therefore M_{AX} is measured in Volts. In the DPWM, it is related to the resolution of the counter 2^{ENOB} which is adimensional, and the dimension is introduced by the gain V of the switching stage. The contribution of these two coefficients, the comparator scale factor and the switching gain, is combined in the total PWM scale factor α.

In a digital control system, also the ADC introduces a normalization scale factor g, because it maps the value withing the FSR into a number of B_{BIT}. Furthermore, the conversion delay, the modulating update mechanism of the DPWM, and the fact that its value is held for an entire sampling period, will affect the loop-gain with a total delay effect $e^{-s\Gamma}$ which degrades the phase margin. In the ideal situation, the ADC samples and converts the signal in an infinitesimal amount of time, just before the updating of the modulating signal, which then holds the converted value for a sampling period T. However, in real systems the data conversion takes a finite amount of time, which then requires to be processed and finally transfered to the modulator.

PWM Scale Factor	ADC Scale Factor	Discrete Delay FX	Open-Loop Gain	Loop Gain
$\alpha = \dfrac{V_s}{M_{AX}}$	$g = \dfrac{2^{B_{BIT}}}{F_{SR}}$	$e^{-s\Gamma}$	$A = C\alpha P$	$T = A\beta$

Table 2.2: PWM controller loop gain parameters

After the loop is closed, two transfer functions can be identified: the closed-loop gain $A_F(s)$ and the noise-loop gain $N_F(s)$

$$\tilde{y} = \frac{A}{1+T}\tilde{u} + \frac{N}{1+T}\tilde{v}_s = A_F\tilde{u} + N_F\tilde{v}_s \tag{2.12}$$

Both transfer functions have a particular behavior when the loop gain is high, $T >> 1$ in the complex sense, and they also share the stability property. In fact, under the hypothesis of high gain, the closed-loop transfer function will follow the theoretical gain, while the noise-loop gain drops to zero

$$\begin{aligned} \lim_{T\to\infty} A_F(s) &= \frac{1}{\beta} \\ \lim_{T\to\infty} N_F(s) &= 0 \end{aligned} \tag{2.13}$$

The stability property is determined by the loop gain, more precisely, if the denominator $1 + T$ reaches zero in the complex plane then the output becomes unstable. This happens because the loop introduces positive feedback in the system, by delaying some harmonics in phase with the input without attenuating or boosting them. In a negative feedback control system, this happens when the phase of the loop gain is inverted $\angle T(j\omega) = -180^o$ while its module drops to the unity $|T(j\omega)| = 1$. The objective of the controller is to keep the loop gain high when possible and to adjust the phase around the crossing frequency to prevent positive feedback.

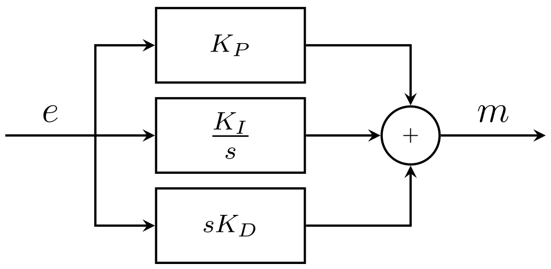


Figure 2.6: PID controller block diagram

A well known controller that implements all these features is the PID controller of Fig. 2.6, that has a pole in the origin which ensure zero asymptotic error for stationary inputs, and two zeros to rise the phase margin before the loop gain reaches the unity

$$C_{PID}(s) = K_P + \frac{K_I}{s} + K_D s = \frac{s^2 K_D + s K_P + K_I}{s} \tag{2.14}$$

Each coefficient is used for a specific purpose, the proportional term K_P sets the intensity of the response, while the integral term K_I determines how to react if the error is accumulated over time. The derivative term K_D is used to increase the response intensity when fast transients occur, but it should be used with care, especially when noise is injected in the loop. With this in mind, the system response can be set tuning the parameters and looking at the response in time. When a model of the process is available, the PID can also be designed in the frequency domain by placing the zeros in the correct position to rise the phase margin and adjust the bandwidth.

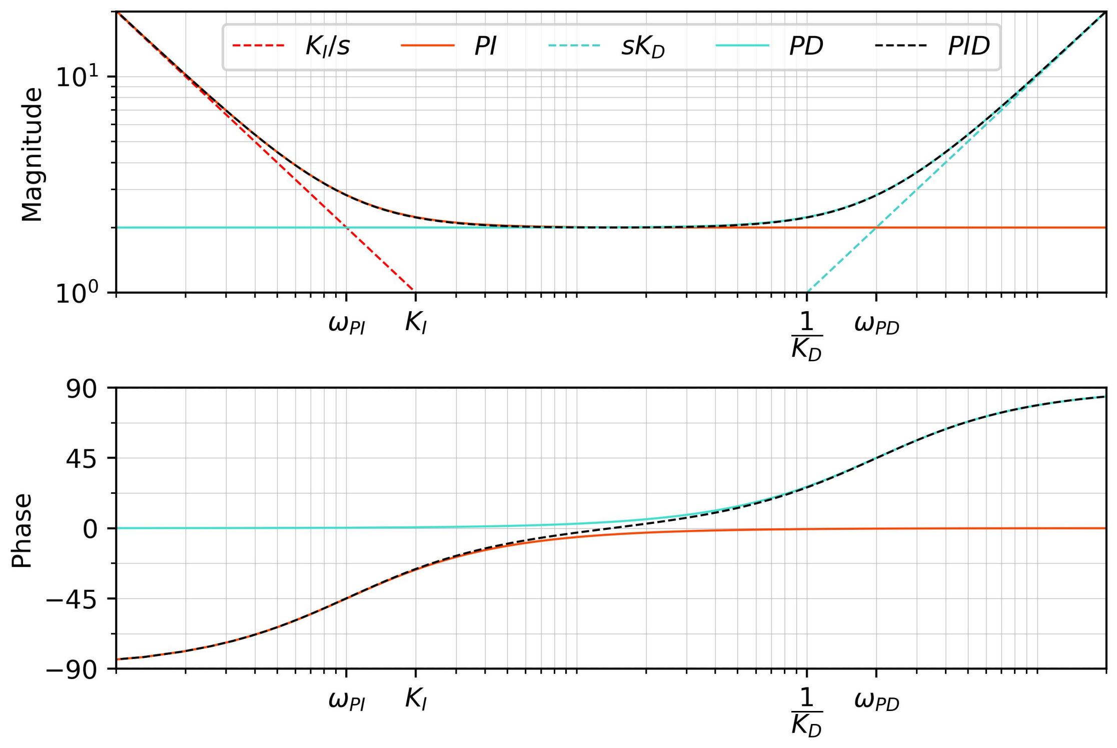


Figure 2.7: Bode plot of the PID designed with the asymptotic technique

The design can be done by looking at the asymptotic behavior of the integrative and the derivative action, that allows to set the PI and PD action independently. At low frequencies, the PI action is dominant over the derivative action, therefore the PID follows the term K_I/s which drops below the unity when $\omega \geq K_I$. When the integrative action becomes comparable with the proportional action $K_P = K_I/s$, the first zero appears at ω_{PI}. If the derivative action is still zero $sK_D \approxeq 0$ when the integrator contribution drops to zero $K_I/s \approxeq 0$, the PI response asymptotically lies down to the proportional gain K_P. Without a derivative action $K_D = 0$, the PI controller design is finished with only two parameters which are used to adjust the gain and the phase.

$$\begin{gathered} \omega_{PI} = \frac{K_I}{K_P} \\ \lim_{\omega \to \infty} |C_{PI}(j\omega)| = K_P \\ \angle C_{PI}(j\omega) = -\frac{\pi}{2} + \tan^{-1}(\frac{\omega}{\omega_{PI}}) \end{gathered} \tag{2.15}$$

If a derivative action is required to boost the phase margin, this shouldn't interfere with the integrative action, therefore its contribution should start at higher frequencies. The derivative action $K_D s$ rises above the unity for $\omega \geq 1/K_D$, and when it becomes comparable to the proportional action $K_P = K_D s$, the second zero appears at the frequency ω_{PD}

$$\begin{gathered} \omega_{PD} = \frac{K_P}{K_D} \\ \angle C_{PID}(j\omega) \approxeq -\frac{\pi}{2} + \tan^{-1}(\frac{\omega}{\omega_{PI}}) + \tan^{-1}(\frac{\omega}{\omega_{PD}}) \end{gathered} \tag{2.16}$$

This design is based on several approximations, however it returns good results when the derivative action starts one decade above the integrative action $1/K_D \geq 10K_I$ like in Fig. 2.7

2.2.2 Discretization

The discretization method is used to transform the continuous-time controller $C(s)$ in a discrete-time controller $C(z)$ that can be implemented through an algorithm in a digital system. There are methods aimed to conserve the time domain response like the ZOH, and others that tries to conserve the frequency response. Time domain conservation is used to translate the overall process with its interface, usually the DAC and the ADC, in a discrete-time system and then build a controller in the discrete domain. When the controller has been designed in the continuous-time domain, frequency response conservation is used to guarantee that the discretization doesn't destabilize the control system. The purpose of frequency conservation is to approximate the map $z = e^{sT}$, which is the relation between the Zeta transform of a discrete-time signal $y(nT)$ and the Laplace transform of the sampled signal $y_\delta(t)$. The sampled signal $y_\delta(t)$ is modelled by the linear combination of a continuous time signal and the periodic repetition of a Dirac impulse $\delta(t)$

$$y_\delta(t) = \sum_{-\infty}^{\infty} y(t)\delta(t-nT) \tag{2.17}$$

To find a relation between the two domains, these methods try to approximate the derivative operator s with a function of z. The naive method is called backward Euler which can be seen as a way of approximating the derivative in the point $t = nT$ with the incremental ratio of the actual sample and the previous one

$$\begin{gathered} \frac{df(t)}{dt}|_{t=nT} = \frac{f(nT) - f(nT-T)}{T} \\ s = \frac{1-z^{-1}}{T} \end{gathered} \tag{2.18}$$

A more accurate method is called the Tustin method, which approximates the integral operator $1/s$ with the trapezoidal integration

$$\begin{gathered} \int_{nT-T}^{nT} f(\tau)d\tau = F(nT) - F(nT-T) = [f(nT) + f(nT-T)]\frac{T}{2} \\ s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}} \end{gathered} \tag{2.19}$$

For these methods the frequency conservation is not always guaranteed, and it needs to be checked by substituting $z = e^{j\omega T}$ and compare it to $s = j\omega$. Looking at the way the Euler method approximates the map, it works well below the sampling frequency $f << F = 1/T$, while the Tustin method gives good results also at higher frequencies because the error term drops faster to zero

$$\begin{gathered} s = \frac{1-e^{-j\omega T}}{T} \approx j\omega \iff \frac{\omega^2 T^2}{2} \approx 0 \\ s = \frac{2}{T}\frac{e^{j\omega T}-1}{e^{j\omega T}+1} = \frac{2}{T}j\tan(\pi f T) \approx j\omega \iff \frac{\omega^3 T^3}{3} \approx 0 \end{gathered} \tag{2.20}$$

Both methods introduce a frequency distortion called warping, however, the Tustin methods allows to compensate this effect with a pre-warping operation to obtain the right target frequency $\bar{f}$

$$f = \frac{\arctan(\pi \bar{f} T)}{\pi T} \tag{2.21}$$

The PID controller in the discrete domain is obtained by using the Euler method $E_{PID(z)}$ or the Tustin method $T_{PID}(z)$. A straightforward way is to discretize both the integrator K_I/s and the differentiator $K_D s$, and then rearrange the Zeta transfer function to obtain a block diagram which can be implemented by an algorithm

$$\begin{aligned} E_{PID}(z) &= K_P + \frac{K_I T}{1-z^{-1}} + K_D \frac{1-z^{-1}}{T} \\ T_{PID}(z) &= K_P + \frac{K_I T}{2} \frac{1+z^{-1}}{1-z^{-1}} + K_D \frac{2}{T} \frac{1-z^{-1}}{1+z^{-1}} \end{aligned} \tag{2.22}$$

All Zeta transfer functions can be written in the canonical form, by finding all coefficients a_n and b_m, and implement the autoregressive moving-average (ARMA) structure.

$$\sum_{n=0}^{N} a_n y(kT - nT) = \sum_{m=0}^{M} b_m x(kT - mT) \tag{2.23}$$

The ARMA structure is a generic compact way of implementing every linear discrete response, and it can be directly mapped to an algorithm. The full PID can be grouped and computed with the ARMA algorithm, however, keeping the discretized integrator and the differentiator separated allows to implement more sophisticated control algorithms like the anti-windup integrator, useful to block the saturation effect of non-linear systems.

The integrator and the differentiator can be visualized from the block diagrams of Fig. 2.8 and Fig. 2.9. Both the Euler integrator $E_I(z)$ and the Tustin integrator $T_I(z)$, use an accumulation mechanism, while the Tustin integrator also delays the input to keep trace how it changes.

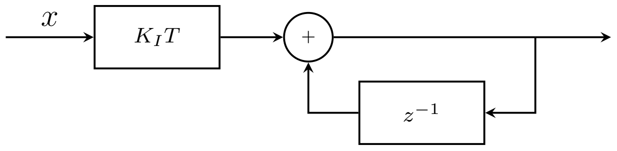
$$\begin{aligned} E_I(z) &= \frac{K_I T}{1-z^{-1}} \\ y_n &= k_i x_n + y_{n-1} \end{aligned} \tag{2.24}$$

$$\begin{aligned} T_I(z) &= \frac{K_I T}{2} \frac{1+z^{-1}}{1-z^{-1}} \\ y_n &= k_i(x_n + x_{n-1}) + y_{n-1} \end{aligned} \tag{2.25}$$

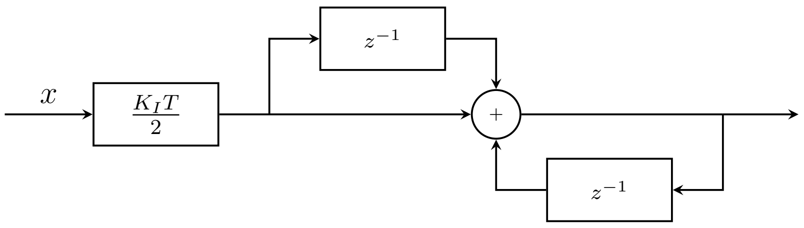
Similarly, the differentiator discretized with the Tustin methods $T_D(z)$ keeps trace of how its output changes, differently to that discretized with the Euler method $E_D(z)$.

$$\begin{aligned} E_D(z) &= K_D \frac{1-z^{-1}}{T} \\ y_n &= k_d(x_n - x_{n-1}) \end{aligned} \tag{2.26}$$

$$T_D(z) = K_D \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}$$
$$y_n = k_d(x_n - x_{n-1}) - y_{n-1} \tag{2.27}$$

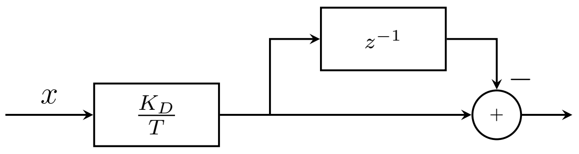


(a) Integrator K_I/s discretized with Euler method

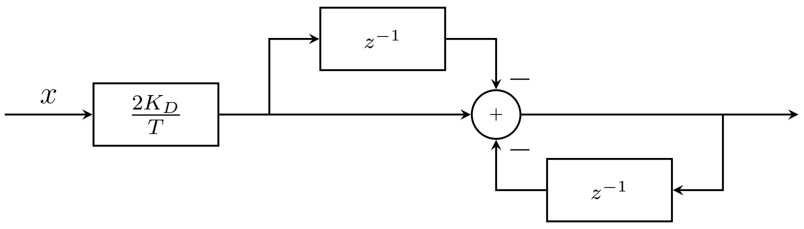


(b) Integrator K_I/s discretized with Tustin method

Figure 2.8: Integrator block diagrams after discretization



(a) Differentiator sK_D discretized with Euler method



(b) Differentiator sK_D discretized with Tustin method

Figure 2.9: Differentiator block diagrams after discretization

2.3 Analog to Digital conversion

The digital to analog conversion, as well as the inverse operation, is one of the most delicate part of the signal acquisition and discretization. It may be divided in three main components: sampling, quantization and holding operation. The sampling operation converts the continuous time signal $y(t)$ in the discrete time signal $y(nT)$. The quantization translates each value of the input in a numeric value with a fixed number of bits B_{BIT}. The holding logic is a function that can be used to represent a discrete-time to a continuous-time conversion, useful to model the final stage of a DAC or even a digital logic that holds its value for a clock period. These elements are used to model the digital part of the system and also the interface with the analog process. Figure [2.10] shows a possible combination of these elements, but in general they need to be rearranged to better describe the full system.

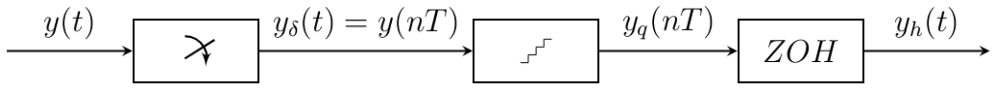


Figure 2.10: Analog to digital interface model

2.3.1 Sampling

The sampling operation can be modelled by means of the original signal $y(t)$, the continuous time sampled signal $y_\delta(t)$ and the discrete-time signal $y(nT)$. The continuous-time sampled signal is the original signal with a combination of Dirac impulses $\delta(t)$

$$y_\delta(t) = y(t) \sum_{n=-\infty}^{+\infty} \delta(t - nT) = \sum_{n=-\infty}^{+\infty} y(nT)\delta(t - nT) \tag{2.28}$$

By using this definition, the Laplace transform of the sampled signal $y_\delta(t)$ is equivalent to the Zeta transform of the discrete signal $y(nT)$ evaluated for $z = e^{sT}$

$$\begin{aligned} y_\delta(s) &= \sum_{n=-\infty}^{+\infty} \int_{-\infty}^{+\infty} y(t)\delta(t - nT)e^{-st}dt \\ &= \sum_{n=-\infty}^{+\infty} y(nT)e^{-snT} = \sum_{n=-\infty}^{+\infty} y(nT)z^{-n} = y(z) \end{aligned} \tag{2.29}$$

This is a unique relation between the sampled signal and the discrete time signal, but the relation between the original signal and the sampled one is not trivial. In fact, the Fourier transform of the sampled signal is the convolution in the frequency domain of a combination of Dirac impulses $\delta(f)$ with the original signal spectrum. This comes from the property of the Fourier series of the Dirac combination, which is the inverse of a Dirac combination in the frequency domain

$$\sum_{n=-\infty}^{+\infty} \delta(t - nT) = \frac{1}{T} \sum_{k=-\infty}^{+\infty} e^{j2\pi kFt} = \mathcal{F}^{-1}\Big\{\frac{1}{T} \sum_{k=-\infty}^{+\infty} \delta(f - kF)\Big\} \tag{2.30}$$

A unique relation can be found only if the original spectrum is band limited and the sampling frequency is at least twice the signal bandwidth $F \geq 2B_W$. If so, the sampled spectrum becomes the repetition of the original spectrum centered at kF

$$y_\delta(s) = \int_{-\infty}^{+\infty} y(t)\frac{1}{T}\sum_{k=-\infty}^{\infty} e^{j2\pi kFt}e^{-st}dt = \frac{1}{T}\sum_{k=-\infty}^{\infty} y(s - j2\pi kF) \tag{2.31}$$

This fact is better known as the Shannon-Nyquist theorem that describes how to reconstruct a continuous time signal from a discrete time signal. In practice there are other effects that may corrupt the correct sampling process, like the jitter of the sampling instant or some noise with high spectral components. The consequence of the jitter is a superimposed noise due to the uncertainty of the sampling instant, which is greater for signals with a fast rate of change. On the other hand the high spectral components of the noise injected into the sampling process can alias with the signal spectrum. For these reasons, an anti-aliasing filter that limits the signal frequency is always recommended before the sampling procedure.

2.3.2 Quantization

In digital systems the values are encoded with a finite number of bits introducing an error over the true value of the discrete time signal $y(nT)$. This amplitude discretization is called quantization and the error is called quantization error. The quantization error is correlated to the signal and the sampling instant, however it can be modelled as a continuous value discrete time random variable $e_q(nT)$ with uniform probability that can assume values in the range $[-\Delta/2, +\Delta/2]$, where the quantization interval Δ is determined by the number of bits B_{IT} and the full scale range F_{SR} of the quantizer

$$\Delta = \frac{F_{SR}}{2^{B_{IT}} - 1} \tag{2.32}$$

Apparently, the only way to reduce the error effect is to increase the quantizer number of bits, but there is another clever way of doing it, which is based on the estimation of the true value from multiple closely related samples. This method can be derived from a full characterization of the quantization error and by looking at the spectral decomposition of a random signal.

First, all continuous value random variables are characterized by their probability density function $P_{DF}(e)$, a function that integrated over an interval $[a, b]$ gives the probability that the variable takes a value of that interval. This function can be used to evaluate the mean μ_1 and the variance σ^2 using the k^{th}-moment definition μ_k. A uniform distribution has a constant PDF, inversely proportional to the cardinality of the interval of possible values, from this fact it is possible to find all the parameters of the quantization error

$$P_{DF}(e) = \frac{1}{\Delta} \tag{2.33}$$

$$\mu_k = \int_{-\infty}^{+\infty} e^k P_{DF}(e) de \tag{2.34}$$

$$\begin{aligned} \mu_1 = \int_{-\infty}^{+\infty} e P_{DF}(e) de = \frac{1}{\Delta} \int_{-\Delta/2}^{\Delta/2} e de = 0 \\ \sigma^2 = \mu_2 - \mu_1^2 = \int_{-\infty}^{+\infty} e^2 P_{DF}(e) de = \frac{1}{\Delta} \int_{-\Delta/2}^{\Delta/2} e^2 de = \frac{\Delta^2}{12} \end{aligned} \tag{2.35}$$

Second, the variance can also be derived from the autocorrelation function $R_{ee}(\tau)$, which is the measure of the similarity between a signal with a delayed copy of itself. When the delay is zero $\tau = 0$, the autocorrelation becomes the second moment $\mu_2 = R_{ee}(0)$, and it represents how likely the signal changes around its mean. The precise formulation depends on the signal type, continuous, discrete, finite or periodic, but more important is the consequence of the Wiener-Khinchin theorem which states that power spectral density $P_{SD}(f)$ is the Fourier transform of the autocorrelation function

$$\begin{aligned} P_{SD}(f) &= \int_{-\infty}^{\infty} R_{ee}(\tau) e^{-j2\pi f \tau} d\tau \\ R_{ee}(\tau) &= \int_{-\infty}^{\infty} P_{SD}(f) e^{-j2\pi f \tau} df \end{aligned} \tag{2.36}$$

Vice versa, the autocorrelation is a measure of the signal power, in particular, if evaluated in zero, it becomes the average power $R_{ee}(0) = P_E$ which for a discrete time signal can also be evaluated by integrating the power spectral density up to the sampling frequency

$$P_E = \int_{-F/2}^{+F/2} P_{SD}(f) df \tag{2.37}$$

The samples of a true random signal are completely uncorrelated one with the other, which means that the autocorrelation of a random signal must be zero except when the delay is zero $R_{ee}(\tau) = 0 \; \forall \, \tau \neq 0$.

$$R_{ee}(\tau) = \mu_2 \delta(\tau) \tag{2.38}$$

For this reason the P_{SD} of a random signal is constant at all frequencies because it is the Fourier transform of a Dirac delta function. It follows an important property of the quantization error, in fact, since the integration of the P_{SD} gives the variance σ^2 which is fixed by the quantizer level Δ, then the P_{SD} must be inversely proportional to the sampling frequency F

$$P_{SD}(f) = \frac{\Delta^2}{12F} \tag{2.39}$$

This means that it is possible to oversample the signal and use a digital filter to keep the signal spectral components while reducing the P_{SD} and the average error power P_E.

2.3.3 Holding

The holder is a block useful to model a digital to analog interface that returns a continuous time signal $y_h(t)$ keeping constant the discrete time signal $y(nT)$ for a whole sampling period. The constant holding function, also called zero order hold, is described by means of two Heaviside step functions $h(t)$ forming a rectangular impulse that lasts exactly one sampling period

$$y_h(t) = \sum_{n=0}^{+\infty} y(nT)\big[h(t-nT) - h(t-nT-T)\big] \tag{2.40}$$

From this definition it is possible to find a transfer function with respect to the continuous time sampled signal $y_\delta(t)$ which is related to the Zeta transform of the discrete time signal $y(nT)$ through the map $Y_\delta(s) = Y(z = e^{sT})$

$$\begin{aligned} Y_h(s) &= \sum_{n=0}^{+\infty} y(nT)\big[\frac{e^{-snT} - e^{-s(n+1)T}}{s}\big] \\ &= \frac{1-e^{-sT}}{s}\sum_{n=0}^{+\infty} y(nT)e^{-snT} = \frac{1-e^{-sT}}{s}Y_\delta(s) \end{aligned} \tag{2.41}$$

By looking at the frequency response of the ZOH transfer function, it is possible to estimate the delay of the digital system response, which is constant and equal to half of the sampling period $T/2$

$$\begin{aligned} Z_{OH}(s) &= \frac{1-e^{-sT}}{s} \\ Z_{OH}(j\omega) &= \frac{e^{j\omega\frac{T}{2}} - e^{-j\omega\frac{T}{2}}}{e^{j\omega\frac{T}{2}}j\omega} = \frac{2sin(\omega\frac{T}{2})}{e^{j\omega\frac{T}{2}}\omega} \\ \angle Z_{OH}(f) &= \angle e^{-j\omega\frac{T}{2}} = -f\pi T \\ |Z_{OH}(f)| &= \frac{sin(\pi f T)}{\pi f} \end{aligned} \tag{2.42}$$

Chapter 3

Prototype

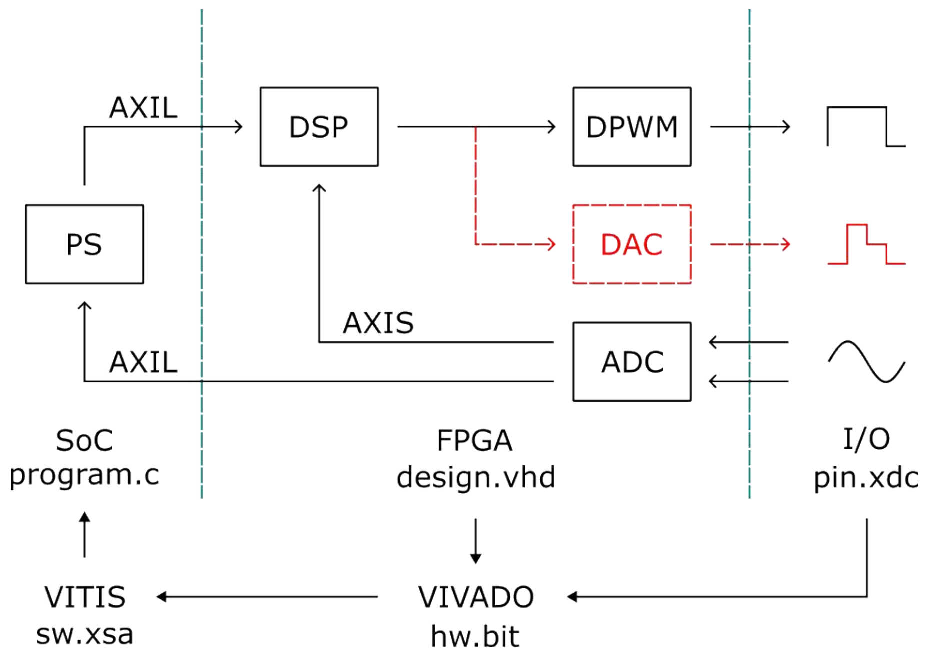


Figure 3.1: Block diagram of the prototype, showing the main data flow and the design procedure

3.1 DPWM Implementation

The FPGA based DPWM is designed to be fully programmable from software, such that the PS can configure all parameters by writing to some configuration registers. The physical parameters, like the number of bits of the counter, are configurable only during the bitstream generation, while the variable parameters, like the counter M_{AX} value, are configurable from software during run-time. To make them soft-configurable, the components in the FPGA are connected to the PS through an AXI interface that maps the DPWM registers in the PS memory.

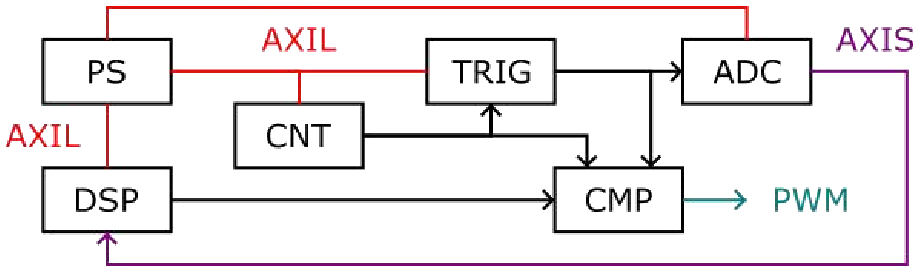


Figure 3.2: DPWM block diagram showing the main connections

The design is subdivided in functional blocks, a counter to generate both the saw ramp and the triangular ramp, the main PWM comparator with a compare register that stores the modulating value and a trigger generator to synchronize the acquisition and the update events. All DPWM blocks are described in VHDL, mainly to make them portable but also because the underneath structure can be inferred from the code. These two features of the VHDL code, allows the DPWM to be implemented on other technologies rather than Xilinx FPGA, while it can be optimized, tested and debugged in all its components.

3.1.1 Basic DPWM

```
COUNTER_LOGIC : process(clk_i)
begin
    if rising_edge(clk_i) then
        if counter_r >= unsigned(max_i) then
            counter_r <= to_unsigned(0, counter_r'length);
        else
            counter_r <= counter_r + 1;
        end if;
    end if;
end process;
```

Listing 3.1: counter VHDL

```
COMPARATOR_LOGIC : process(counter_r, comp_i)
begin
    if counter_r < unsigned(comp_i) then
        pwm_o <= '1';
    else
        pwm_o <= '0';
    end if;
end process;
```

Listing 3.2: comparator VHDL

The basic DPWM is made of two main blocks, the counter and the comparator, both described at behavioral level to let the compiler infer the best structure. The counter is a sequential circuit made with an M_{BIT} register, that is updated at every clock rising edge and forced to restart from zero when it reaches the `max_i` value. The `max_i` value is programmable in the range $[0 : 2^{M_{BIT}} - 1]$, allowing for an accurate control over the modulation period. The counter can optionally include an enabling logic and reset logic, but the basic version is just a free running counter that starts as soon as `max_i` is set to a value different from zero. The comparator is a combinational circuit that constantly compares the counter with the `comp_i` value and asserts the `pwm_o` line when it matches or exceeds the comparator value. The comparator can optionally be inverted to implement the leading edge modulator, but the basic version is just a trailing edge modulation. The `comp_i` value can be set in the range $[0 : M_{AX}]$, such that the PWM output has a duty cycle of zero when `comp_i = 0` and a duty-cycle of one when `comp_i = max_i + 1`. Values greater than M_{AX} are possible but inherently saturated by the comparator logic, resulting always in a 100% duty-cycle. This effect allows the comparator to work safely up to $2^{M_{BIT}} - 1$, but greater values should be saturated to prevent the overflow, either by the software or with a dedicated hardware.

```vhdl
MATCH_COMPARATOR : process(counter_i, instant_i)
begin
    if counter_i = instant_i then
        trigger_o <= '1';
    else
        trigger_o <= '0';
    end if;
end process;
```

Listing 3.3: trigger generator VHDL

The counter is routed to the trigger generator to synchronize all components that interact with the modulator. Differently from a generic enabling signal asserted for long periods, the trigger signal is a pulse that lasts a single clock cycle or a small fixed period. Since the counter assumes a specific value only for a single clock period in each counting sequence, the trigger can be generated by a comparator activated on the match between the counter and a specific value.

The DPWM is configured and controlled from the PS, using a 32-bit configuration registers connected to the AXI GP port. Each register is accessible in one read/write transaction, therefore, to minimize traffic in the AXI bus, the parameters should be grouped in few registers when possible. Two configuration parameters are enough for the basic DPWM, one is the counter `max_i` value and the other is the `instant_i` value for the trigger generator. If the DPWM has a 16 bit counter, these two values can be stored in a single 32 bit register, such that they can be accessed in a single read/write operation. Even if the configuration parameters don't require to be changed frequently and with strict timing constraints, groups can be made to minimize FPGA usage and also to force a correct setup. For example, by grouping the `max_i` value and the `instant_i` value in one register, these are forced to change together to prevent errors like `max_i < instant_i`. On the other hand, the main control parameters, like `comp_i`, that needs to change frequently without errors, should be stored in dedicated registers accessible from a dedicated bus. Since the DPWM configuration is usually done once, and doesn't change during normal operations but only when the DPWM stops, the `comp_i` value is stored in dedicated register accessible from a single AXI bus for both configuration and runtime parameters to reduce FPGA usage.

3.1.2 Up-Down Counter

```vhdl
UPDOWN_COUNTER : process(clk_i)
begin
    if rising_edge(clk_i) then
        if (resetn_i = '0') or (reloadn_s = '0') then
            counter_r <= to_unsigned(0, counter_r'length);
        else --if (enable_s = '1') then
            case updown_s is
            when '0' =>
                counter_r <= counter_r - 1;
            when others =>
                counter_r <= counter_r + 1;
            end case;
        end if;
    end if;
end process;
```

Listing 3.4: updown counter VHDL

An important feature of the DPWM is the possibility to switch between a saw ramp or a triangular ramp to implement both symmetric and asymmetric modulation. To generate a triangular ramp, the counter should be able to count downward and upward, and change its direction whenever it reaches the `max_i` value or zero. This feature must be software selectable from a configuration bit attached to the input `sawtri_i`. The counting direction is managed by a control unit through the signal `updown_s`, which is set to `1` when the counter must count upward and `0` when it must count downward. The up-down counter also needs some sort of reset mechanism, either controlled externally from the input `reset_i` or by its own control machine through the reset signal `reload_s`. The reset mechanism is necessary for two reasons, to reset the counter when it reaches the `max_i` value, and to keep the counter to

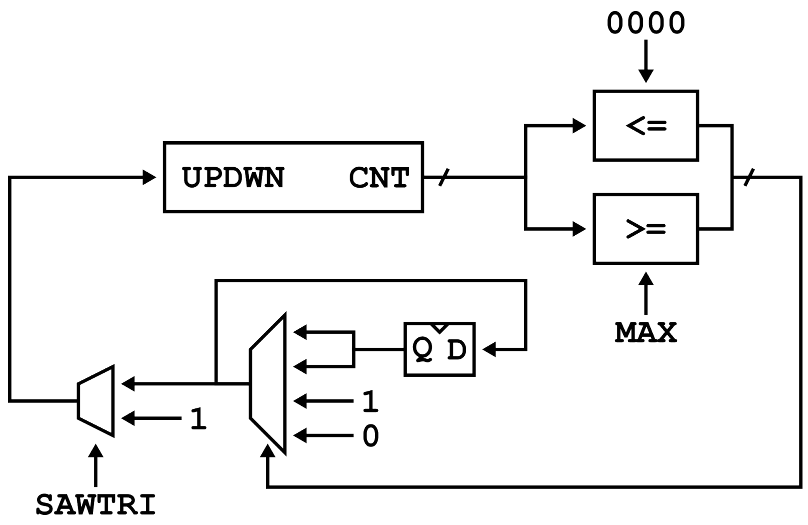


Figure 3.3: Updown counter block diagram with range detection and status latch logic

zero and ready to start. This second feature can be implemented with an `enable_s` signal, that also allows starting the counter, however it turns to be a redundant and not useful operation which complicates the control mechanism.

```
OVERFLOW_LOGIC : process(counter_r, max_i)
begin
    if counter_r >= unsigned(max_i) then
        over_s <= '1';
    else
        over_s <= '0';
    end if;
end process;

UNDERFLOW_LOGIC : process(counter_r)
begin
    if counter_r <= to_unsigned(0, counter_r'length) then
        under_s <= '1';
    else
        under_s <= '0';
    end if;
end process;

range_s <= under_s & over_s;

UPDOWN_LOGIC : process(range_s, sawtri_i, updown_r)
begin
    case sawtri_i is
    when '0' =>
        updown_s <= '1';
    when others =>
        case range_s is
        when "01" =>
            updown_s <= '0';
        when "10" =>
            updown_s <= '1';
        when others =>
            updown_s <= updown_r;
        end case;
    end case;
end process;

UPDOWN_REG : process(clk_i)
begin
    if rising_edge(clk_i) then
        if (resetn_i = '0') or (reloadn_s = '0') then
            updown_r <= '1';
        else
            updown_r <= updown_s;
        end if;
    end if;
end process;
```

Listing 3.5: updown control logic VHDL

To choose the direction, the control unit checks the status of the counter, if the `max_i` value is reached, it rises the overflow flag `over_s`, and if the zero is reached it rises the `under_s` flag. These two conditions are then concatenated with the `&` operator, not to be confused with the bitwise `and`, into the `range_s` flag used by the control logic. The up-down control logic only cares about the `sawtri_i` selector, the range in which the counter is, and the previous status registered in the flip-flop `updown_r`. If the `sawtri_i` bit is not active, the control logic always tells the counter to count upward, otherwise it checks the counter range and, whenever it overflows or underflows, it forces the counter to change its direction to return inside the allowed range. When the direction is corrected then it is kept until another out-of-range event occurs, therefore the up-down control logic works like a switch driven by the conditions.

```
RELOAD_LOGIC : process(range_s, sawtri_i)
begin
    case sawtri_i is
    when '0' =>
        case range_s is
        when "01" =>
            reloadn_s <= '0';
        when "11" =>
            reloadn_s <= '0';
        when others =>
            reloadn_s <= '1';
        end case;
    when others =>
        case range_s is
        when "11" =>
            reloadn_s <= '0';
        when others =>
            reloadn_s <= '1';
        end case;
    end case;
end process;
```

Listing 3.6: reload control logic VHDL

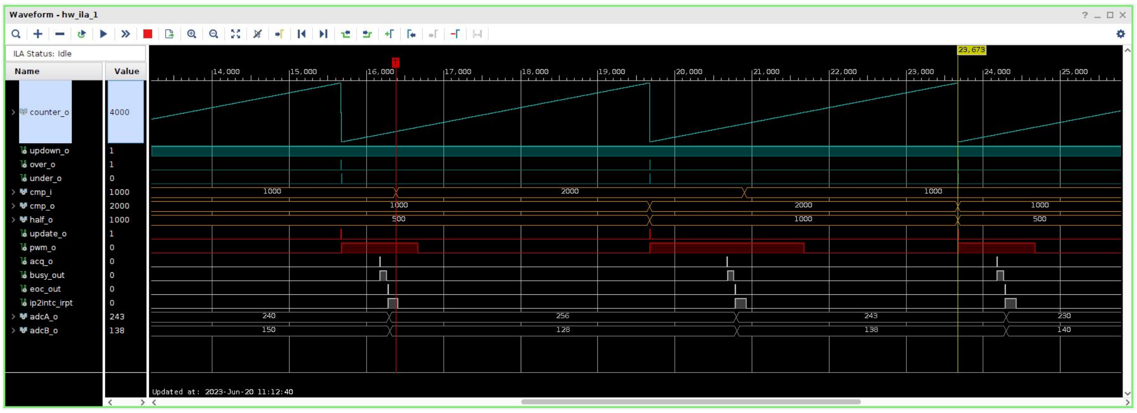
The counter reload logic is an essential part of the up-down counter, mainly for the saw mode, and it is also part of the control sequence to properly start the counter. When the counter is working in saw mode, the active low reload signal `reloadn_s` is activated if the counter overcomes the upper range limit `range_s = 01`. The `range_s` status signal can also describe the situation when `max_i = 0`, corresponding to the case when `range_s = 11`, which forces the counter to reload. This operation may seem trivial, but it helps to keep the counter steady, otherwise it may oscillate between `0` and `1`. When the selected mode is the triangular ramp, the counter doesn't reload, except when `max_i=zero` and the range collapses.

3.1.3 Trigger generation

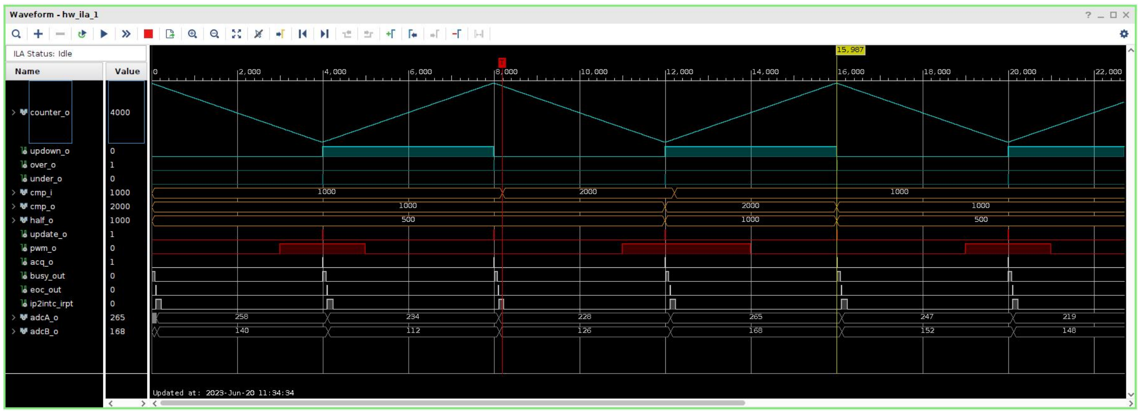
The DPWM comparator value `comp_i` is usually controlled directly by the user or indirectly by an automatic control system. In both cases its value can change in any instant of the modulation period, which in most of the cases can't be predicted. To synchronize the comparator update event to the modulation period there must be a buffer register that is updated at regular intervals.

The buffer register is realized with DFFs, and it may be optionally bypassed if not required or for any other reason. The update event is controlled by the trigger generator that enables the register to store the value at the next clock cycle. Since the enabling mechanism of the DFF is synchronous, the update trigger should last at least one clock period and possibly less than two to prevent the latch of a second value.

The trigger generator for the comparator update depends on the type of ramp. When using the saw ramp, the trigger is generated when the counter reaches the `max_i` value, such that the register is effectively updated when the counter is reloaded



(a) Sawtooth ramp DPWM with the comparator update at the beginning of each modulation period, and the sample acquisition in the middle point of the ON period



(b) Triangular ramp DPWM with the comparator update and the sample acquisition at the maximum and minimum counter instant

Figure 3.4: DPWM working modes showing the sample acquisition, the signal processing, and the update of the comparator. The sequence starts with the acquisition trigger, the ADC data is converted and elaborated inside the interrupt handler, but the comparator register is updated only at the next update event

to zero. On the other hand, when using the triangular ramp, the comparator can be updated when it reaches zero, when it reaches the maximum value, or also in both cases. This selection is realized with an enabling stage made of AND gates controlled by the `peak_i` and `valley_i`, which let the triggers propagates if enabled. The enabling stage is followed by a merging stage made of an OR gate that allows a single line to drive the update events. Finally, a multiplexer is added to prevent the double update in case both `peak_i` and `valley_i` are enabled when using a saw ramp. This multiplexer isn't strictly needed to work properly, because the user could simply disable the update at the zero matching condition. Even if both triggers occur, they will be merged in a single trigger that last two clock cycles, resulting in a non-critical soft error. Moreover, that multiplexer will introduce a latency, limiting

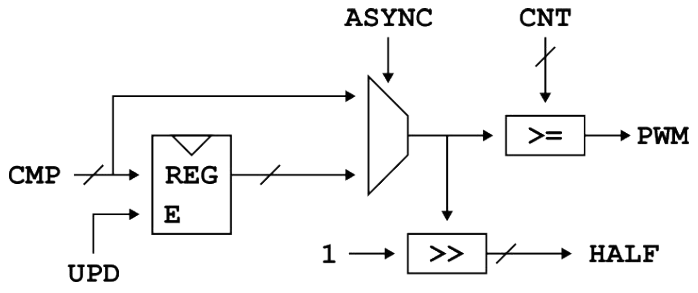


Figure 3.5: Comparator block diagram with buffer register for synchronization

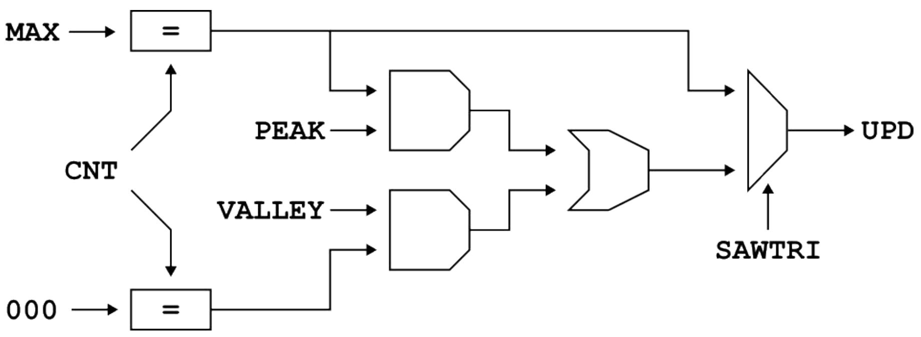


Figure 3.6: Trigger generator for the comparator update optimized for the up-down counter

the maximum achievable speeds. Although not needed, the final multiplexer is left in the design, to clarify the update instant in saw ramp mode, and to force a safer configuration.

In DPWM control systems, the sampling instant should be synchronized with the modulation period to fully exploit the modulation properties. Depending on the modulation type and the expected feedback signal, there are some strategic instants used to extract more information than a pure amplitude value. Moreover, depending on the controller behavior, the sampling instant could be set as close as possible to the comparator update event to minimize the delay introduced by the control system. These instant are usually at the beginning or at the end of the counting sequence, at the rising edge or at the falling edge of the PWM output or at the midpoints of the on-off periods.

The triggers for the acquisition events are realized using multiple comparators that compare the counter value with the instant value. The comparators are routed to an enabling stage made of AND gates, which is used to individually enable the triggers with a set of control signals. Finally, the enabled triggers are merged into a single line with a cascade of OR gates.

Since comparators are abundant in the trigger generation, it would be better if they are optimized for the target technology. N-bit matching comparators in LUT based devices are realized by subdividing the two inputs into groups with the same

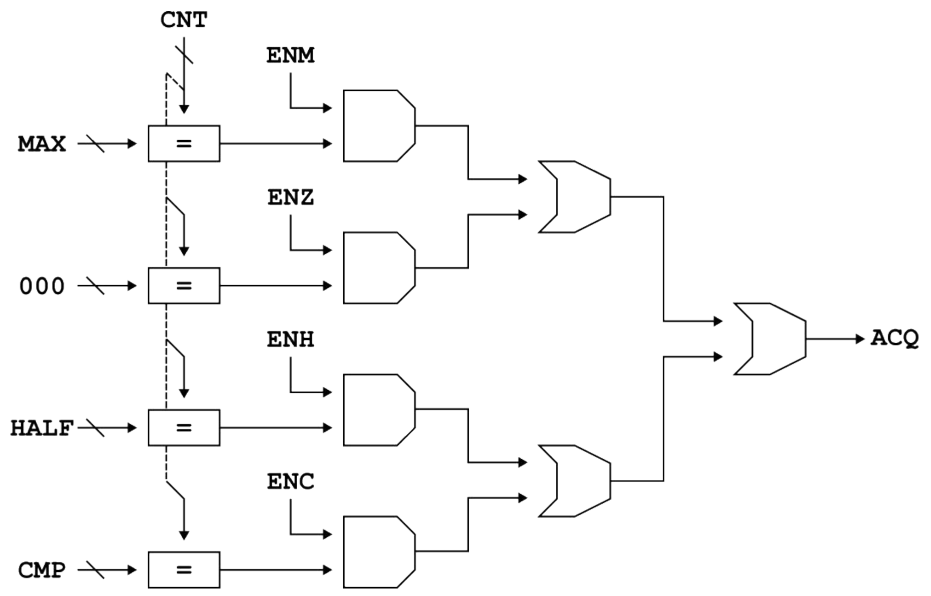


Figure 3.7: Trigger generator for synchronized acquisition events

number of bits, and then comparing them using a *divide and conquer* method. These groups are made with bits that belong to the same position of both inputs, otherwise a comparison would be meaningless. With 6 bit input LUTs, like those embedded in the Artix-7 CLB slices, it is possible to realize a comparison between 2 groups of 3 bits. These partial comparisons are merged into cascaded stages of LUTs that are used to trigger the output if all the previous results are true. For example, a 16-bit comparator, realized with 6 input LUTs, requires a total of 7 LUTs, 6 of them used as partial comparators, and the last one to merge their results.

A single Artix-7 CLB is made of 2 slices with 4 LUTs each, therefore comparators ranging from 12 bits up 18 bits are well optimized for this technology. When many comparators are required there are other solutions to restrict the number of implemented comparators and save resources. For example, using a multiplexer controlled by a selector to route the acquisition registers to a single comparator. However, if comparators are already implemented in other blocks, and their outputs can be reused either by the user or by the compiler, a second different solution turns into a waste of resources instead of saving them.

3.2 Controller implementation

The control system includes the ADC and the controller, implemented either via software or with the hardware DSP. The XADC can convert up to 2 simultaneous signals which are always stored in the relative registers and streamed through the AXI-Stream bus. At each conversion the XADC sends an IRQ to the PS to inform it that a new data is ready. Meanwhile, a sequence of data is streamed to a dedicated

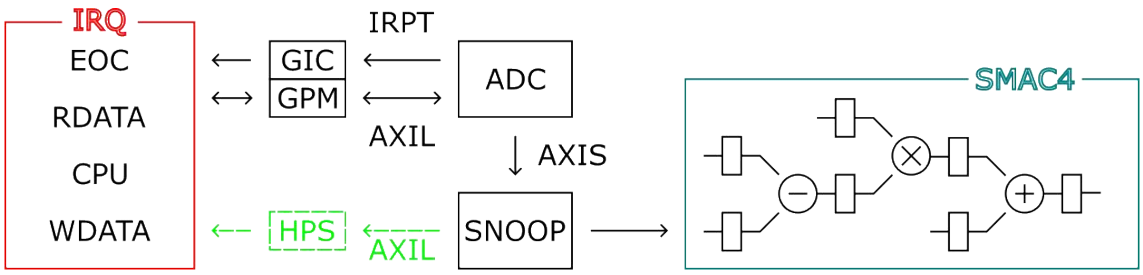


Figure 3.8: Software controller routine compared to hardware controller data flow

block, that checks the data packet IDs and saves those it needs. By doing so the converted signals can be elaborated both in the PS or in the PL, enabling different control architectures. The software controller runs periodically inside an interrupt handler, and it allows building complex behaviors like different anti-windup schemes. On the other hand, the one built with the hardware DSP blocks tends to be simpler, but it allows for deterministic low latency control schemes.

3.2.1 ADC configuration

The XADC is wrapped with the Xilinx AXI IP connected to the M_AXI_GP1 port and controlled from the PS. This allows to partially reconfigure the XADC from software after the bitstream generation, as well as query the XADC as needed, to read the converted signals or the status registers. The XADC works in event sampling mode and the conversion start input `convst_in` is controlled by the trigger generator. The conversion starts the first clock cycle after the `convst_in` rising edge, and it takes 22 ADC clock periods to convert the sampled signal. After other 4 ADC clock periods, the data is stored in the register, and it can be read from the PS with an AXI read transaction.

```c
// XADC base address position and register offsets
#define AXI_XADC_BASE_ADDRESS 0x7FFF8000U
// XADC register offset
#define STATUS_OFFSET 0x04U
#define VAUX1_OFFSET  0x244U
#define VAUX9_OFFSET  0x264U
#define GIER_OFFSET   0x5CU
#define IPISR_OFFSET  0x60U
#define IPIER_OFFSET  0x68U
// Inline definition for read and write
#define XADC_mRead(BaseAddress, RegOffset) \
  Xil_In32(BaseAddress+RegOffset)
#define XADC_mWrite(BaseAddress, RegOffset, Value) \
  Xil_Out32(BaseAddress+RegOffset, Value)

```

```c
int main() {
  uint16_t adc = XADC_mRead(AXI_XADC_BASE_ADDRESS, VAUX9_OFFSET);
}
```

Listing 3.7: XADC definitions C

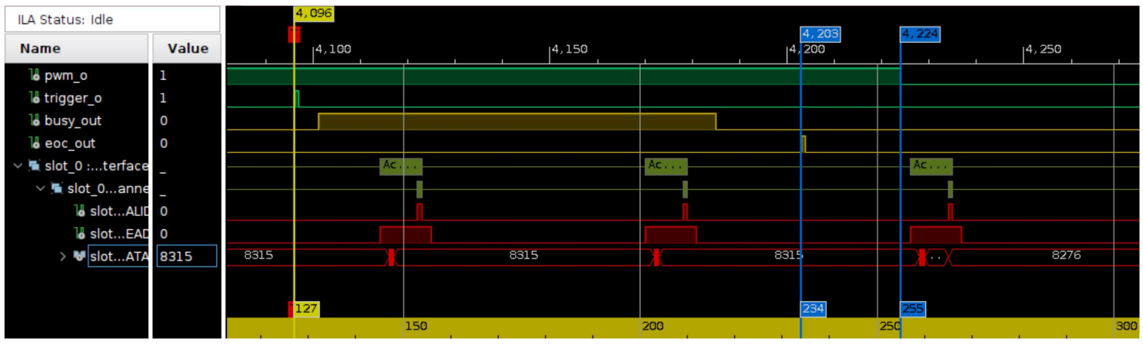


Figure 3.9: XADC conversion with polling mode AXI read transaction shows that a full conversion takes 107 $DCLK$ cycles

The AXI Read transaction is usually asynchronous to the ADC conversion, and it is difficult to understand if the data in the relative register is the new one or the old one. This can be solved by looking at the `channel[4:0]` lines or by using a flag to indicate if the data has already been read. However, these methods require to continuously poll the XADC or a status register not to miss the data, which most of the time is not useful. A more efficient way is to ask the processor intervention at the end of each conversion with an interrupt, such that the PS can execute an AXI Read transaction to the XADC register as soon as the data is converted, and fetch the data.

```c
void setupInterruptXADC(void){
  u32 reg = 0;
  reg = (1 << GIER);
  XADC_mWrite(AXI_XADC_BASE_ADDRESS, GIER_OFFSET, reg);

  reg = (1 << EOC);
  XADC_mWrite(AXI_XADC_BASE_ADDRESS, IPIER_OFFSET, reg);
}
```

Listing 3.8: XADC interrupt setup C

To implement the interrupt system, the `IP2INTC_irpt` signal is routed to the PS `IRQ_F2P` pin and configured to interrupt the processor. The `IP2INTC_irpt` is enabled by the GIER, and the end-of-conversion event that triggers the interrupt is enabled in the IPIER. This configuration is done by the software, possibly with a safe procedure, for example by checking if other events are enabled to disable them.

```c
#define INTC_DEVICE_INT_ID 61
```

```c
void setupGIC(void){
    // Initialize the GIC
  static XScuGic InterruptController;
  static XScuGic_Config *GicConfig;
  GicConfig = XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
  XScuGic_CfgInitialize(&InterruptController, GicConfig, GicConfig
   ->CpuBaseAddress);

  //Assign the main handler for the interrupt exception
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
      (Xil_ExceptionHandler)XScuGic_InterruptHandler, &
   InterruptController);
    //Enable the exceptions
  Xil_ExceptionEnable();

  //Connect a handler to the device ID
  XScuGic_Connect(&InterruptController, INTC_DEVICE_INT_ID,
      (Xil_ExceptionHandler)DeviceDriverHandler, (void *)&
   InterruptController);
  //Enable the device ID interrupt
  XScuGic_Enable(&InterruptController, INTC_DEVICE_INT_ID);
}
```

Listing 3.9: GIC setup C

The GIC in the PS should be configured to receive interrupts from the `IRQ_F2P` pins, which are a total of 16, each with a unique ID. Since the GIC is a non-vectored controller, a unique primary handler is associated to each CPU. Every time an interrupt exception rises, this handler polls the GIC to get the ID of the device that asked for the interruption. This is done by the `Xil_ExceptionRegisterHandler` that makes the `XScuGic_InterruptHandler` the main handler. All other device handlers are connected and enabled to the main one by using `XScuGic_Connect`, `XScuGic_Enable` and the unique `INTC_DEVICE_INT_ID`.

```c
void DeviceDriverHandler(void *CallBackRef){
  u32 reg = XADC_mRead(AXI_XADC_BASE_ADDRESS, IPISR_OFFSET);
  XADC_mWrite(AXI_XADC_BASE_ADDRESS, IPISR_OFFSET, reg);
  adc = XADC_mRead(AXI_XADC_BASE_ADDRESS, VAUX_OFFSET) >> 4;
}
```

Listing 3.10: XADC handler C

By doing so, the interrupt system is configured and the XADC handler is going to be executed at the end of each conversion. However, since the `IP2INTC_irpt` is one for all the XADC events, the only way to know which event has triggered the interrupt is to poll the XADC with a AXI read transaction to the IPISR. This operation can be neglected in case there is only one interrupt event active, like if the system works only with the EOC event, but even in this case, there is no way rather than polling the status register to check which channel has triggered the EOC event. Moreover, the bit in the IPISR must be cleared with an AXI write transaction that toggles the bit, otherwise the interrupt remains active, and the processor jumps into it as soon as it exits the handler. If the first instruction is used to clear the IPISR, then other events of the same nature can re-trigger `IP2INTC_irpt` and are guaranteed to be served when the processor exits the handler. On the other hand, if the IPISR is cleared with the last instruction, the processor can complete the tasks in the handler with a lower latency.

In all the cases, the processor enters the interrupt handler with a variable delay that depends on the CPU active tasks. The interrupt delay from the EOC is approximately 50 T_{CLK} periods long, and in the best case it takes other 50 clocks cycles to complete the handler routine. The initial delay can be reduced only using the FIQ line of the processor or other software solutions, while the access delay can be reduced by reorganizing the code to execute the most important read and write operations first, keeping the same amount of time to complete the routine.

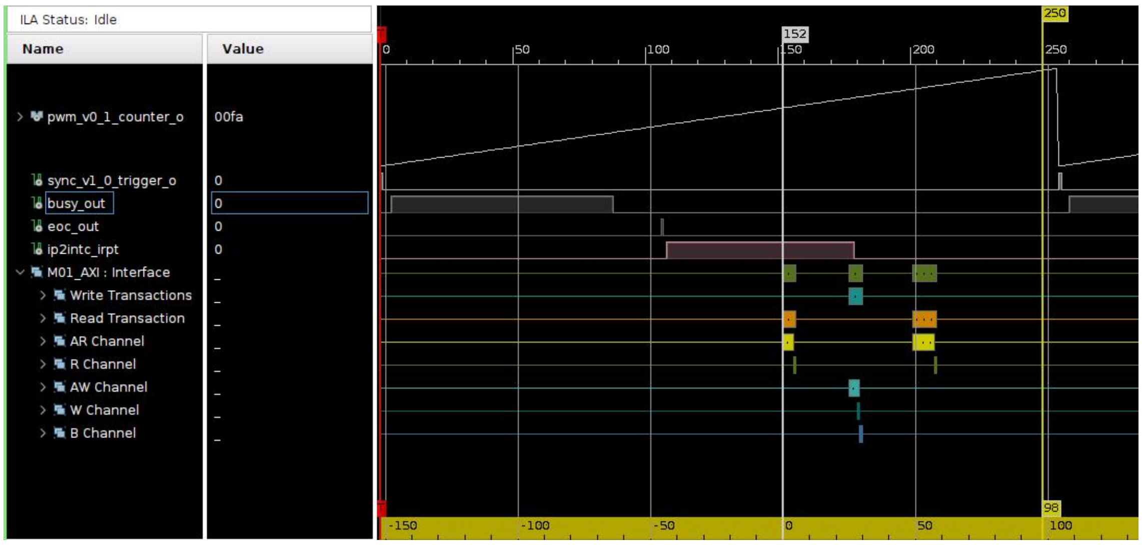


Figure 3.10: XADC interrupt handler showing that the EOC interrupt is served and cleared approximately after 50 T_{CLK} cycles, and the ADC value read after other 50 clock cycles

The AXI-Stream XADC interface allows streaming data with lower latency from the EOC event, but it requires a subsystem capable of manage and dispatch the converted data. In fact, the XADC tries to stream data by asserting the `VALID` line as soon as a channel is converted, but if the subordinate doesn't assert the `READY` line, the data is pushed into a FIFO until it is full and the XADC stops working.

Moreover, by embedding the channel number in the transmission ID, the XADC stream interface uses only one transmission line for all the channels. Therefore, it's up to the subordinate to check the ID and route the data to the right path, or throw it away to flush the FIFO.

```vhdl
id_c <= std_logic_vector(to_unsigned(SNOOP_ID, ID_WIDTH));

-- s_axis_tready <= '1'; --Always ready is a BAD PRACTICE
s_axis_tready <= s_axis_tvalid; -- The subordinate waits the
    manager to assert TVALID
DATA_EXTRACTION : process(s_axis_aclk)
begin
    if rising_edge(s_axis_aclk) then
        if (s_axis_tvalid = '1') and (s_axis_tid = id_c) then
            data_r <= s_axis_tdata;
            id_r <= s_axis_tid;
        end if;
    end if;
end process;

adc_o <= data_r when (enout_i = '1') else (others => '0');
```

Listing 3.11: AXI Snooper VHDL

To manage the XADC stream, a solution is the AXI snooper, a controller that asserts the `READY` line, checks the transmission ID, stores the data in a register and throws away those that are useless. The implemented AXI snooper will also allow to enable the output or force it to zero, for example to open or close the feedback loop. Differently from a "sniffer" that checks the transmission line externally without interfering with the operations, the "snooper" is actively involved. This is a basic requirement to work with the AXI-Stream protocol because the data stream is always pulled by the subordinate, not pushed by the manager. To pull the data stream, the `READY` line can be left asserted to pretend that the subordinate is always ready and prevent the stall of the XADC. However, this method isn't safe, even if compliant with the AXI stream rules, because the AXI snooper can miss some data, typically when the subordinate is effectively busy while the manager sends them. A better approach is to use the lazy property of the AXI subordinate, which allows it to flag the ready status only when the manager has already validated a data. The AXI snooper could also be upgraded to accept multiple IDs and store the different data in more than one register, or snoop the transmission, store the data and re-stream it.

3.2.2 Software Controller

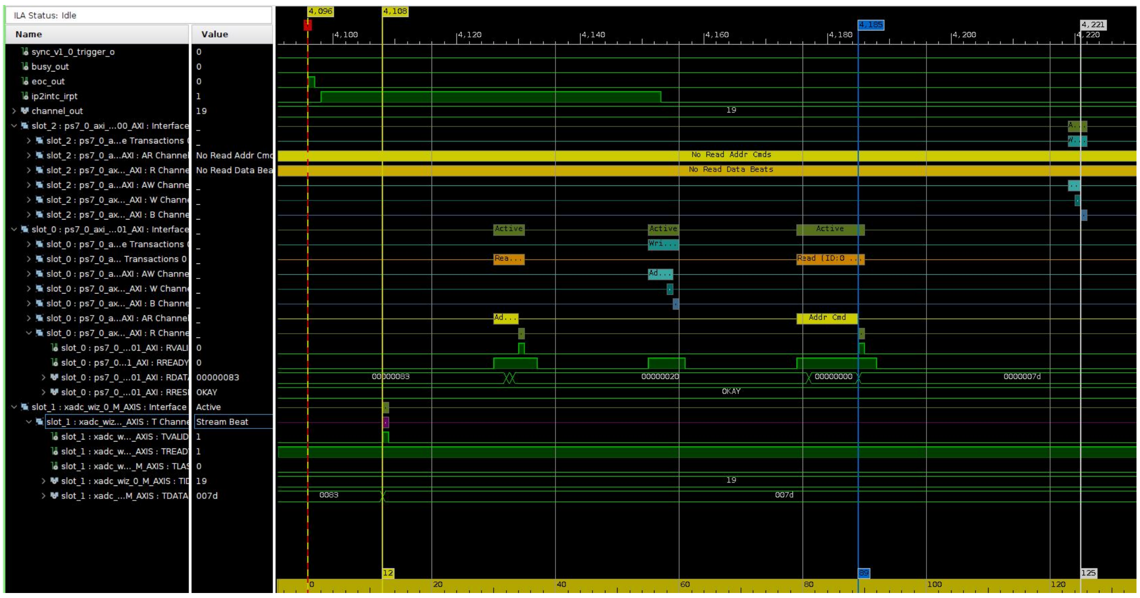


Figure 3.11: AXI-stream compared with AXI-Lite, showing that the data is streamed 12 T_{CLK} after the EOC, against the ≈ 90 clock cycles required by the PS to fetch the data, and the 125 required to execute the algorithm

```c
static uint16_t adc = 0;
static uint16_t ref = 0x777;
static int16_t err = 0;

static int32_t out = 0;             // controller output
static int32_t out_max = 0xFFFF;   //

void DeviceDriverHandler(void *CallBackRef){
    // Set adc = 0 to open the feedback loop
    adc = XADC_mRead(AXI_XADC_BASE_ADDRESS, VAUX_OFFSET) >> 4;
    err = (int16_t) (ref - adc);
    out = controlAlgorithm(err);
    // Limiter
    if (out < 0) {out = 0;}
    if (out > out_max) {out = out_max;}
    // Write the PWM comparator and clear the pending interrupt
    AXIREG_mWriteReg(AXI_REG0_BASEADDR, AXI_CMP_REG, out);
    XADC_mWrite(AXI_XADC_BASE_ADDRESS, IPISR_OFFSET, 0);
}
```

Listing 3.12: Software Controller C

To implement the software controller, the XADC should be configured to send the IRQ to the processor at each EOC event, such that the piece of code inside the handler is executed periodically. The handler first reads the data from the XADC with an AXI transaction, and then computes the error from the set point. The converted channels are always stored in a 16 bit register and left aligned, such that the `adc[15:4]` bits are the effective bits while the 4 less significant bits are random bits that can be discarded or used for dithering. Also, the set point `ref` is stored in a 16-bit unsigned integer, but it must be limited by the comparator maximum value `ref <` $2^{M_{BIT}}$ to prevent the overflow of the PWM comparator, and it should be limited by the counter maximum value to prevent the saturation.

By doing so, the error turns to be the difference of two 16-bit unsigned integers, which should be stored in a 17 bit signed integer. However, by using 16 bits the compiler can infer the use of special DSP instructions to accelerate the control algorithm that returns the value `out` for the PWM comparator. Since the output of the control algorithm may be a negative number, or it may exceed the maximum PWM comparator value, it should be limited to prevent the overflow. Finally, the saturated value is sent to the PL with an AXI write transaction, the interrupt is cleared, and the processor exit the handler returning to normal operation.

To disable the feedback control, the `adc` variable must be set to zero and the algorithm bypassed, either by modifying the code or by using other run-time methods. A naive method is to use a soft-switch with a `bool open` variable controlled in the main program to switch between `y=ref if(open == 1)` or to execute the control algorithm `if(open == 0)`. This method requires to check the open condition inside the interrupt handler, that shouldn't be a problem for an APU with a branch speculator, but there are more sophisticated ways, like dynamically changing the content of the interrupt handler using function pointers.

```
static uint16_t kp = 56590;     // 1.727 x 2^15
static uint16_t kiT = 16685;    // 0.5092 x 2^15
static uint16_t kdF = 21989;    // 0.6710 x 2^15
static int32_t xi = 0;          // integrator input
static int32_t yi = 0;          // integrator output
static int32_t xd = 0;          // differentiator input
static int32_t yd = 0;          // differentiator output
static int32_t xold = 0;        // delayed input
static int32_t yold = 0;        // delayed output

void pidEuler(void) {
    // Integrator
    xi = (kiT*err) >> 15;
    yi = yi + xi; //yold = yi is embedded;
    // Differentiator
    xd = (kdF*err) >> 15;
    yd = xd - xold;
    // PID Output
```

```c
    out = yi + yd + ((kp*err) >> 15);

    // Update
    xold = xd;
}

void pidTustin(void) {
    // Integrator
    xi = (ki2*err) >> 15;
    yi = yi_1 + xi + xi_1;
    // Differentiator
    xd = (kd2*err) >> 15;
    yd = xd - xd_1 - yd_1;
    // PID merged output
    out = yi + yd + ((kp*err) >> 15);

    // update variables
    xi_1 = xi;
    yi_1 = yi;
    xd_1 = xd;
    yd_1 = yd;
}
```

Listing 3.13: PID control algorithms C

The PID control algorithm is realized by using the ARMA structure for both the integrator and the differentiator, and then add them to the proportional part. The coefficients of the PID are rational numbers coded in Q-format and stored in 16 bits variables, and the input error `err` is a 16-bit signed integer. When a rational `Qm.n` number is multiplied by an integer, the decimal point is moved by `n` positions to the left, and it might require an adjustment by a post-shift to the right. The multiplication doesn't implicitly require the post-shift but only more bits to represent the result, however, if the result is accumulated, some extra bits are needed to prevent the overflow.

This effect is clearly visible in the integrator part of the Euler PID, that adds its previous value to the product. The result of the multiplication `xi = kiT*err` is stored in a 32-bit signed integer variable, and then added to the accumulator which is also stored in a 32-bit signed integer register. This means that a post-shift operation is necessary before the accumulation, otherwise the overflow would be straightforward. In the basic implementation, all the 15 lower bits are discarded, and the upper bits are used for accumulation. This solution simplifies the sum between all the partial results because it doesn't require another point alignment. However, only the integer part of all multiplication is kept while the decimal part is always thrown away. A better solution would be to reduce the bit for the accumulation and use them for the decimal part. This requires to adjust the point when merged in the final sum with some shifts, but the algorithm gains some resolution.

A secondary problem of the integrator is how it behaves around the saturation limits when closed in a feedback loop. Let's suppose the system has just been turned on, the process output is zero and also the set point is at zero. The ADC reads zero as the first value, the error is zero and the controller sends a zero value to the modulator. Ideally nothing happens, and the situation is perfectly stable, however, the ADC may read 1 as the second value because of some noise, the error becomes negative, and the controller accumulates a negative error. At this point the negative error is permanently stored in the accumulator because the ADC can't read a value below zero, and cycle after cycle it accumulates the noise contribution. When the set point rises to the desired value, the controller output doesn't react until the integrator enters again in the linear region of the modulator. A similar problem happens whenever the controller output saturate the modulator during a transient and the system doesn't react linearly to the controller, or when something is clamping or limiting the process output. This effect is called windup, and, to solve it, it requires a mechanism that removes the accumulation effect when the saturation condition is detected.

```c
static acc_max = 0x0FFF;

void antiwindup(void) {
    // Integrator
    xi = (kiT*err) >> 15;
    yi = yold + xi;
    // Saturation detector
    if ((yi <= acc_max) && (yi >= 0)) {
        yold = yi; // store the integrator output
    }
}
```

Listing 3.14: Anti-windup C

There are multiple anti-windup schemes, the simplest is to compute the integrator response but disable the accumulator if the saturation is detected. This method works well against the noise accumulation effect and doesn't affect the controller behavior inside the saturation limits.

3.2.3 Hardware Controller

```vhdl
architecture RTL of DSP_Pipeline is
    signal a : signed(DATA_WIDTH - 1 downto 0);
    signal b : signed(PARAMS_WIDTH - 1 downto 0);
```

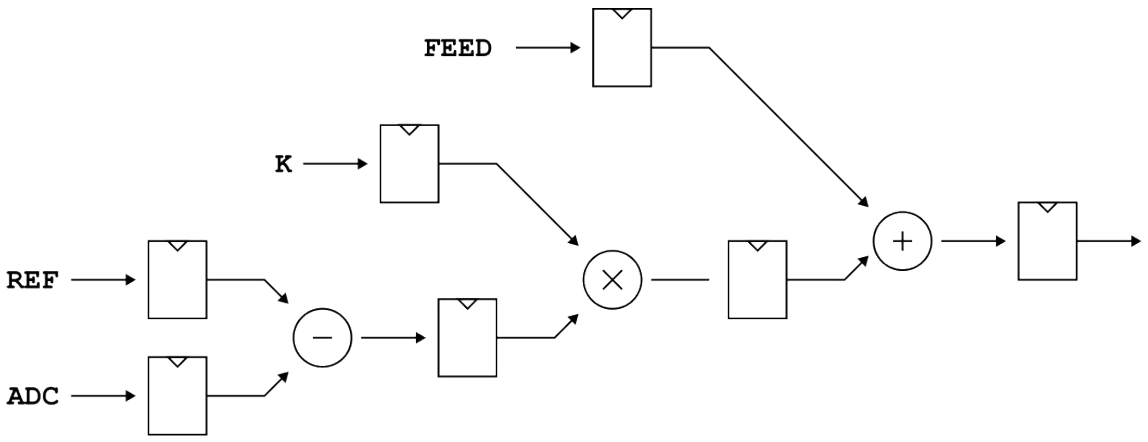


Figure 3.12: Pipeline DSP standard block with 25-bit pre-adder, 18x25 bits signed multiplier and 48-bit final adder

```
    signal c : signed(DATA_WIDTH + COEF_WIDTH downto 0);
    signal d : signed(DATA_WIDTH - 1 downto 0);
    signal sub : signed(DATA_WIDTH downto 0);
    signal mult, p : signed(DATA_WIDTH + COEF_WIDTH downto 0);
begin

    PIPELINE : process(clk_in)
    begin
        if rising_edge(clk_in) then
            if (start_in = '1') then
                a <= signed(ref_in);
                b <= signed(k_in);
                c <= signed(feed_in);
                d <= signed(adc_in);
                sub <= a - d;
                mult <= sub*b;
                p <= mult + c;
            end if;
        end if;
    end process;
    y_out <= std_logic_vector(p);

end RTL;
```

Listing 3.15: DSP basic block

The software controller behavior can be emulated in the hardware to gain some speed and reduce the latency. This can be done using some DSP blocks to build a complex processing unit and routing the ADC converted data to it. DSP blocks are correctly inferred if signed arithmetic is used for inputs and all the operations: "When coding for inference and targeting the DSP block, it is recommended to use signed arithmetic, and it is a requirement to have one extra bit of width for the pre-adder result so that it can be packed into the DSP block". Therefore, the MSB of both the data and coefficients always represent the sign bit, differently from what was done in software. To use full range 16 bit unsigned values, DATA_WIDTH and COEF_WIDTH should be set to 17 bits, and the extra bit fixed to zero, either externally during the software configuration or bitstream generation.

The DSP pre-adder allows to compute the feedback error from the difference of the reference `ref_in` and the ADC converted data `adc_in`. The error is stored in the `sub` register with one bit more to prevent the overflow, and then multiplied by the coefficient `k_in`. The product is stored in the `mult` register, and used together with the `feed_in` input to feed the ALU. The pipeline should be used whenever possible, because most of the arithmetic operations always requires buffer registers both at the input and at the output. To produce the correct output, the pipeline DSP takes from 2 to 4 clock cycles, depending on which input changes, but it can be overclocked with respect to the main system clock. In fact, the DSP pipeline is clocked at high speed, such that the four clock cycles to execute the pipeline are negligible with respect to the sampling period $4T_{CLK} << T$.

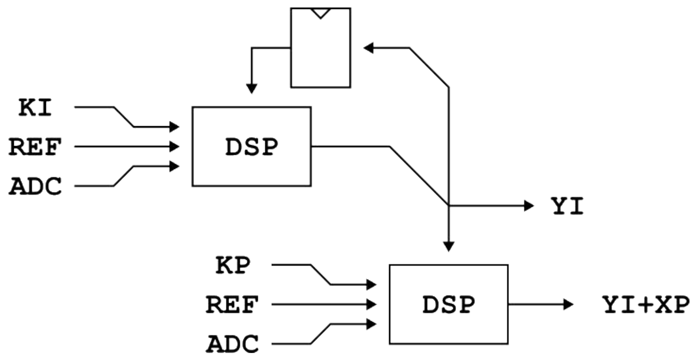


Figure 3.13: PI controller realized with the pipeline DSP

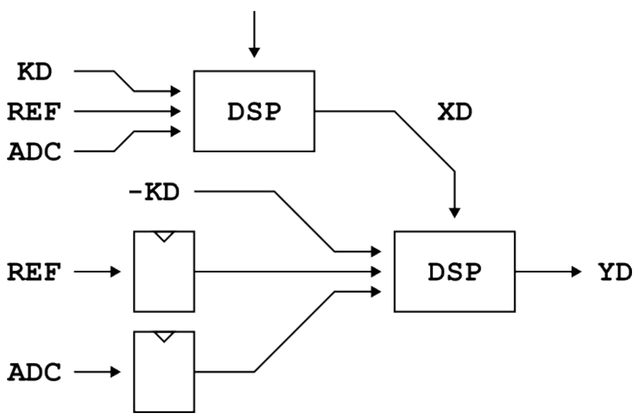
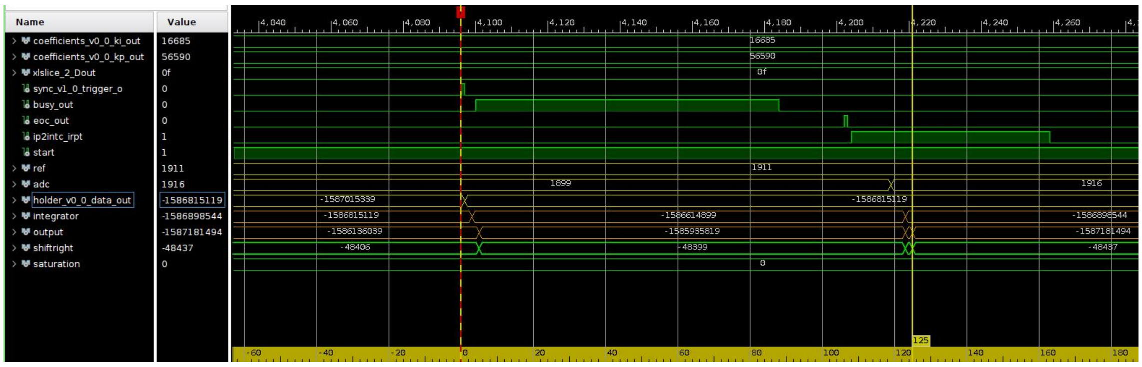


Figure 3.14: Differentiator realized with the pipeline DSP

This configuration of the DSP can be used to implement both the integrator or the differentiator, by using the `feed_in` input to feedback the output or feedforward the input. The sample delay z^{-1} is realized with a register that holds a constant value for an entire sampling period, that must be triggered in the right instant to sample and hold the correct data. Since the registers inside the FPGA are made with D-Type flip-flops, they must be triggered from the enable input `en_in` or synchronized by a clock. The clock is usually shared among the components of a specific region with a dedicated trace, therefore the best solutions are always those with few clock sources that leads to a compact and reliable design. To get rid of multiple clocks, the holder is triggered from the enable input, which is synchronous and must stay on at least one clock period. The DSP pipeline takes one clock cycle to store the `feed_in` input plus another to accumulate the result in the output register. When the `feed_in` is used in feedback configuration, like for the integrator, if the sample and hold register stays active for more than two clock cycles, it will accumulate more than once. Also, when the DSP is used as differentiator, it might be dangerous to keep the register enabled for more than one clock cycle, because it may latch a non-stable data. The solution is to build a circuit that creates a pulse that last exactly one clock cycle, like the trigger generator with match comparators, or using a DFF with the input D and the inverted output $\bar{Q}$ going through an AND port.



Name	Value
coefficients_v0_0_ki_out	16685
coefficients_v0_0_kp_out	56590
xlslice_2_Dout	0f
sync_v1_0_trigger_o	0
busy_out	0
eoc_out	0
ip2intc_irpt	1
start	1
ref	1911
adc	1916
holder_v0_0_data_out	-1586815119
integrator	-1586898544
output	-1587181494
shiftright	-48437
saturation	0

Figure 3.15: DSP PI controller sequence, showing the pipeline execution when the data is sampled and after ADC the conversion

There are several possible implementations of the hardware PID controller but some of them are better suited for the DSP block. The naive version of the PID could be realized with one DSP for the integrator, the two for the differentiator and one with the unconnected `feed_in` to work as a gain block. Then, these 3 outputs are shifted to adjust the point position and routed to a 3 inputs adder. However, this implementation uses just 2 of the 4 ALU in the DSP blocks, many shifters and one additional adder. To take the best out of the DSP blocks, these could be chained, such that all 48 bit ALU are used to compute the partial sums. By doing so, the

PID could be realized with just 4 DSP blocks and some intermediate shifters to adjust the points before feeding the ALU. Moreover, to get rid of multiple shifters, all coefficients can be described with the same `Qm.n` notation, such that the decimal point of all products is already aligned, and they can be directly added together. Finally, the output is shifted by `n` positions to the right and the result constrained to an unsigned number limited by the DPWM comparator number of bits. [1]

The PID in the FPGA is connected to AXI GP port, and configured by some registers that store the coefficients and the post-shift value. A separate control register allows starting the DSP blocks and closing the feedback by enabling the AXI snooper output. Finally, the hardware controller input is connected to the PS, while the output is connected to the DPWM through a switch useful to bypass the hardware controller.

[1]If the same `Qm.n` notation is used for K_P and K_D coefficients, the proportional term can be merged into the DSP differentiator with a single coefficient $(K_P + K_D)$, therefore the final PID requires just 3 DSP blocks

Chapter 4

System Validation

The DPWM can be used in many configurations, by choosing the number of bits, the sampling and update frequency, or the modulation frequency. Also, the digital controller may be designed in several ways, for example to reach high bandwidth or a good time response. Testing all configurations would be unfeasible and most of them will give similar results. Therefore, some specific tests should be used to verify the limits of the device while others for the typical use case.

4.1 Zynq board setup

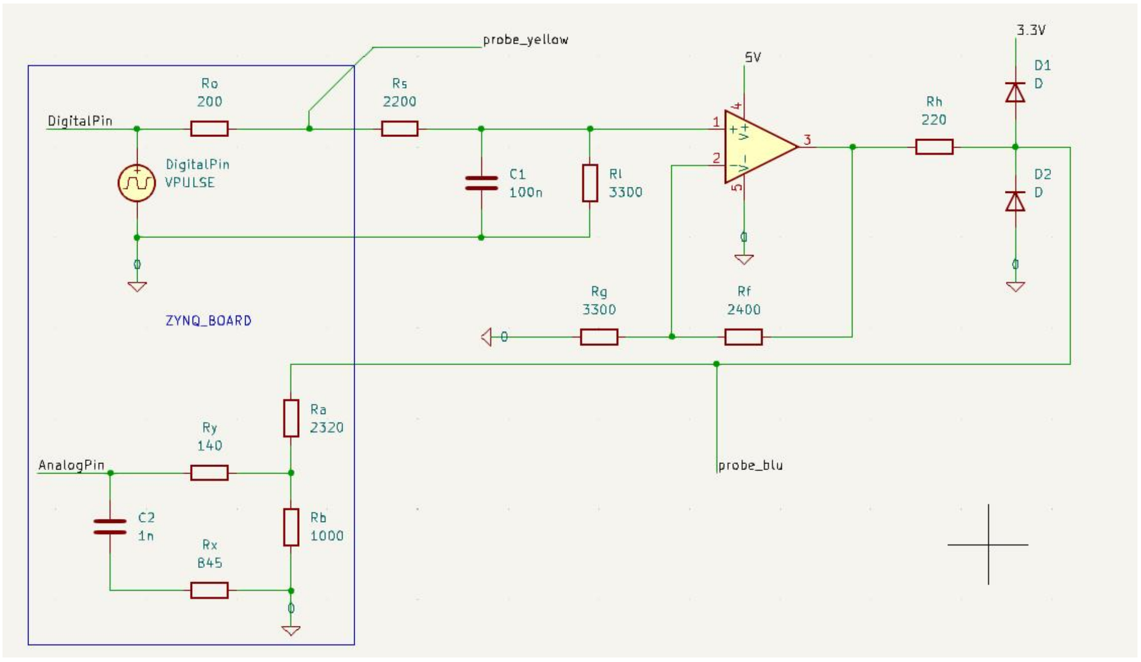


Figure 4.1: Zynq board setup with anti-aliasing filter and single pole test circuit

To verify the DPWM control system, a simple setup is made using some OPAMP to emulate the process response, while the DPWM controller is implemented using the PYNQ-Z2 board. This board has some analog inputs for the signal acquisition, all with a basic anti-aliasing RC network, and digital outputs that switches between $0V$ and $V = 3.3V$. The anti-aliasing network attenuates the input voltage because the ADC can handle only voltages from $0V$ to $1V$, and the board is thought to work between $0V$ and $3.3V$. With the configuration of Fig. 4.1 the range is extended to $F_{SR} = 3.3V$, but it requires to be driven by a low impedance source. To clamp the voltages at the analog input to a maximum value of $V_{MAX} = 3.3V$, two Schottky diodes are placed after a resistor $R_H = 220\Omega$ and connected to the $0V$ and the $3.3V$. This configuration of the analog input is always driven by an amplifier capable of pre-compensate the attenuation without introducing other loading effects. The bandwidth of the anti-aliasing filter would have been $B'_W = 94.522kHz$ but considering the R_H contribution it is reduced to $B_W = 93.482kHz$. The bandwidth is more or less 10 times smaller than the maximum sampling frequency 1 MSample/s, which is a standard choice for simple RC anti-aliasing networks. Considering the current limiter resistor, a small attenuation is introduced $a = 0.94$, lowering the overall gain before the analog input.

$$a = \frac{R_A + R_B}{R_A + R_B + R_H} \tag{4.1}$$

The clock frequency for the DPWM counter is $F_{CLK} = 100MHz$ because it is the maximum clock the PS can provide to the PL without using special PLL. Moreover, higher clock frequencies may lead to failures because the design is not optimized for high speed and because the ILA is going to decrease the reachable performance. At fixed clock frequency, the modulation frequency F_P will depend only on the maximum counter value M_{AX}. Moreover, since the switching voltage is also fixed $V_S = 3.3V$, the maximum counter value will also determine the voltage resolution.

$$V_{RES} = \frac{V_S}{M_{AX} + 1} \tag{4.2}$$

The testable modulation frequencies are limited by the anti-aliasing filter, in fact, it will degrade both the magnitude and phase from relatively low frequencies. The roll off of the frequency response can be either neglected or taken into consideration depending on its contribution, but for comparable results the ratio between the modulation frequency and the anti-aliasing bandwidth F_P/B_W should be kept small.

$$\begin{aligned} |A_{AF}(F_P)| &= \frac{1}{\sqrt{1 + (F_P/B_W)^2}} \\ \angle A_{AF}(F_P) &= -\tan^{-1}\left(\frac{F_P}{B_W}\right) \end{aligned} \tag{4.3}$$

Table [4.1] shows that the PYNQ-Z2 with 100 MHz clock allows testing modulation frequencies up to 24 kHz. Above this limit the anti-aliasing filter will interfere with the tests, making them more board dependent and less useful to characterize the design.

$M_{AX} + 1$	256	1024	4096	16384	65536
$ENOB$	8	10	12	14	16
$V_{RES}\,[mV]$	12.89	3.223	0.8057	0.2014	0.05035
$F_P\,[kHz]$	390	97	24	6.1	1.5
$T_P\,[\mu s]$	2.56	10.24	40.96	163.84	655.36
$\lvert A_{AF}\rvert\,[dB]$	-12	-3	-0.3	-0.02	-0.001
$\angle A_{AF}\,[Deg]$	-76.4	-45.9	-14.5	-3.69	-0.92

Table 4.1: Parameters of the PYNQ-Z2 DPWM at fixed clock frequency $F_{CLK} = 100\,MHz$ and fixed switching voltage $V = 3.3V$

The DPWM voltage resolution V_{RES}, together with the voltage resolution of the ADC V_{FSR}, will act as two independent scale factors for the control system. Since the F_{SR} and the switching voltage V_S are the same, the total scale factor will be determined only by the choice of the relative bits.

$$\alpha = \frac{2^{B_{BIT}}}{2^{ENOB}} \tag{4.4}$$

To normalize the total scale factor $\alpha = 1$, the ADC converted data can always be increased with an arbitrary number of bits. However, to get good results, these bits should be random bits, such that the converted data dithers, instead of being always rounded to the floor or the ceiling by adding all zeros or all ones. This is the default mechanism of the XADC, that stores the converted data in a 16 bit register with the true 12 bits left aligned and 4 less significant random bits.

A simple first order system with only one pole in $\omega_o = \tau_o^{-1}$ can be used to test various configuration of the DPWM control system. This can be realized with an RC network and an OPAMP in non inverting configuration. The RC network fixes the pole and the attenuation while the non-inverting amplifier is used to set gain.

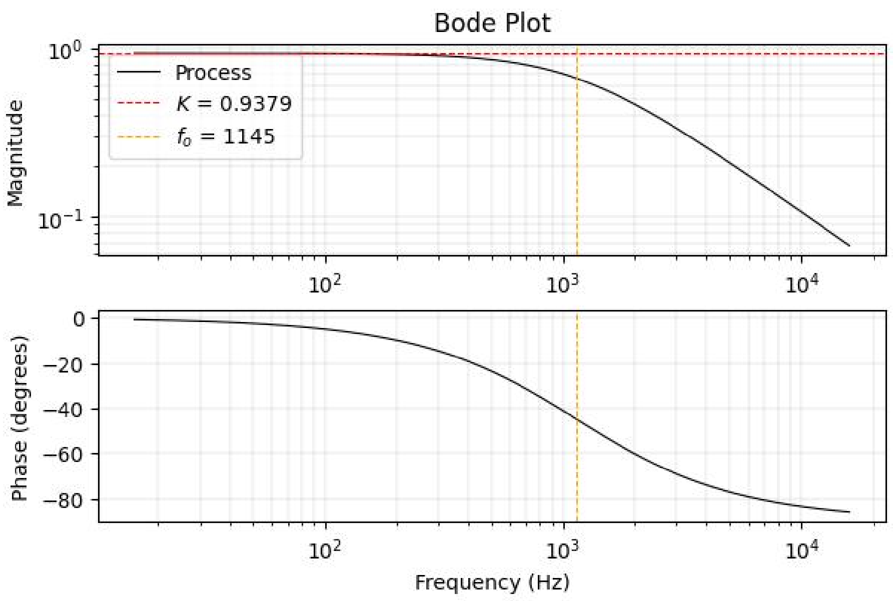


Figure 4.2: Bode plot of the single pole system emulated with the non inverting OPAMP

$$
\begin{aligned}
P(s) &= \frac{K}{s\tau_o + 1} \\
A_{TT} &= \frac{R_L}{R_S + R_L} \\
G_{AIN} &= 1 + \frac{R_F}{R_G} \\
\tau_o &= C(R_L//R_S)
\end{aligned}
\tag{4.5}
$$

A bandwidth around 1 kHz is good to test many configurations because it is sufficiently smaller than the modulation frequency. The RC network is realized with a total series resistor $R_S = R'_S + R_O = 2.4k\Omega$ and a load resistor $R_L = 3.3k\Omega$ that fixes the attenuation to $A_{TT} = 11/19$. The capacitor of the RC network is chosen to be $C = 100nF$, leading to a time constant of $\tau_o = 138,95\mu s$ and a cut-off frequency of $f_o = 1.145kHz$. The gain of the non-inverting amplifier is chosen to compensate the input attenuation $G_{AIN} = A_{TT}^{-1}$, this can always be done by using $R_F = R_S$ and $R_G = R_L$.

4.1.1 Step response characterization

Using a first order system with a single pole, a closed feedback system will always be stable, therefore a simple PI controller is enough to characterize the delay effect. One possible configuration is a PI controller designed with the zero that compensate the pole of the process, such that it is possible to set the gain to reach the desired bandwidth. This can be done by forcing $K_I = \omega_o K_P$ such that the crossing frequency

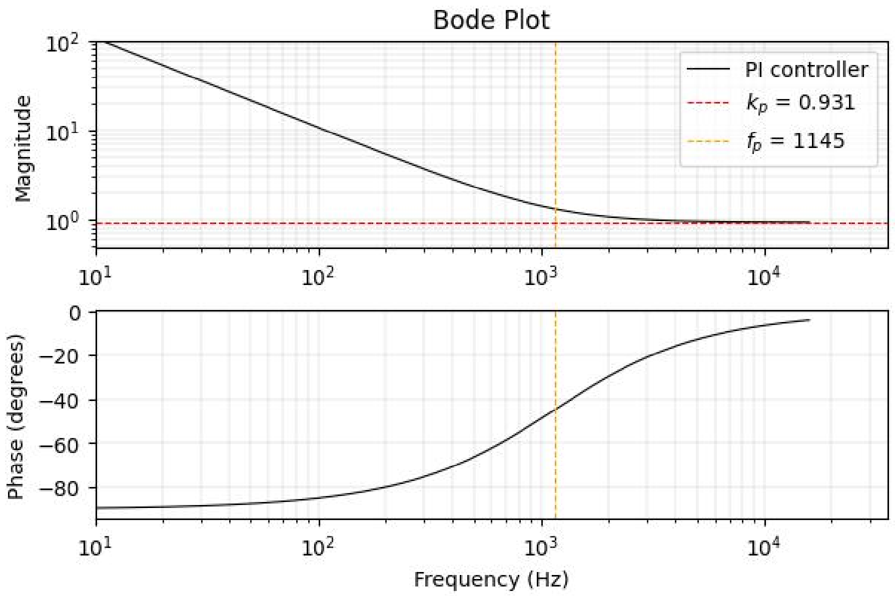


Figure 4.3: PI design such that $f_p = f_o$ and $f_c = 1\,kHz$

becomes $f_c = f_o K K_P$. With this configuration, a continuous time controller always gives a theoretical phase margin of $\phi_M = 90^o$, however, due to the delay effect of the DPWM the phase margin is degraded, as depicted in Fig. 4.4. The degradation of the phase margin depends on the delay from the acquisition instant to the compare instant, which is variable in most of the cases.

Using a sawtooth ramp and keeping a fixed modulating period $T_P = 40.96\mu s$ and one sample per period $T = T_P$, the control system is tested to reach gradually the limit of the crossing frequency. The parameters in table [4.2] are used to close the loop at different crossing frequency and verify the controller behavior.

f_c $[kHz]$	1	2.5	4.5
K_P	0.931	2.327	4.189
K_I $[rad/s]$	6699	16748	30147
$K_I T$	0.2744	0.6860	1.2348

Table 4.2: PI controller parameters to compensate the pole $f_p = f_o$ as a function of the crossing frequency f_c, using a sawtooth ramp DPWM with $T = T_P = 40.96\mu s$

When the acquisition instant is set to be close to the update instant, the delay depends only on the modulating ramp or the operative point. Using a saw ramp and the acquisition synchronized with the middle point of the on-time period, the total delay can be greater than a full modulating period. With this technique a bandwidth of 1 kHz still gives a monotonic step response like in Fig. 4.5, while a

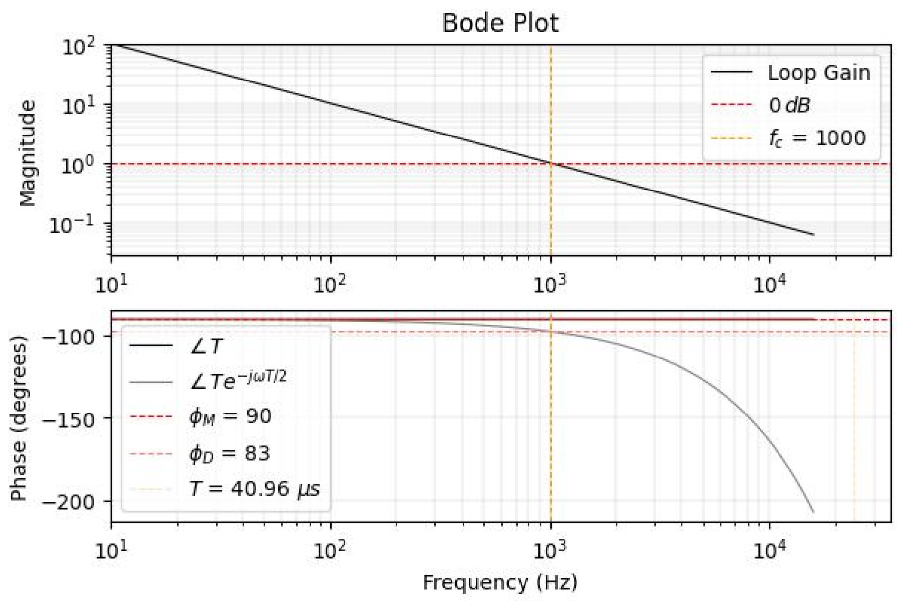


Figure 4.4: Phase margin degradation ϕ_D of the loop gain due to the delay effect

2.5 kHz bandwidth will produce an overshoot due to the fast phase degradation.

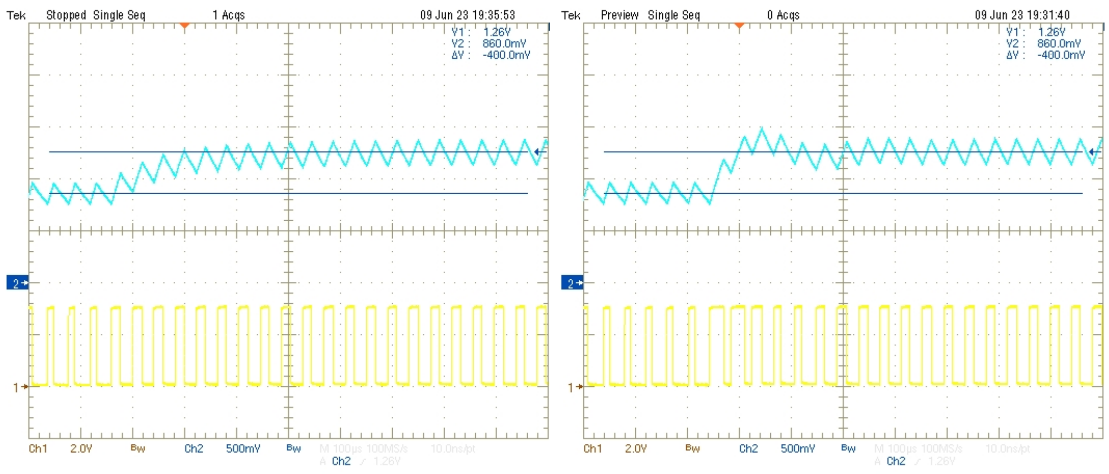


Figure 4.5: Step response of the asymmetric modulation and synchronized acquisition with the loop closed at a crossing frequency of $1\,kHz$ and at $2.5\,kHz$

The overshoot is due to the delay from the acquisition instant, performed at the midpoint of the on period, to the compare instant after the update, which is more than a full modulation period, more precisely $T_D = T_P(1 + D/2)$ By removing the synchronized acquisition option and trigger it as close as possible to the update instant, the crossing frequency can be increased, but the system won't track the average value. A second option, that keeps tracking the average value, is to synchronize the acquisition instant at the middle of the off-time, which, considering the trailing edge modulation, is always closer to the update instant. This requires to evaluate the new comparator value, together with the new acquisition instant, before the end of counting sequence where the comparator is updated. For this solution, the hardware controller becomes essential to push the acquisition limit close to the update instant, otherwise, with a software controller, this limit would be significantly far from the starting of the new modulation period.

To keep tracking the average value, it is possible to use the triangular waveform and trigger the acquisition at the beginning of the modulation period, which corresponds to a peak or a valley of the waveform. To get comparable results with the previous experiment, the modulation frequency with the triangular waveform should always be the same $T_P = 24\,kHz$, this is achieved by halving the counter maximum value, however, this doubles the DPWM scale factor of the forward gain. The forward gain can be compensated by dividing the ADC scale factor, that, even if it reduces the total ADC resolution, it will be kept coherent with that of the DPWM $2^{ENOB} = 2^{B_{BIT}}$.

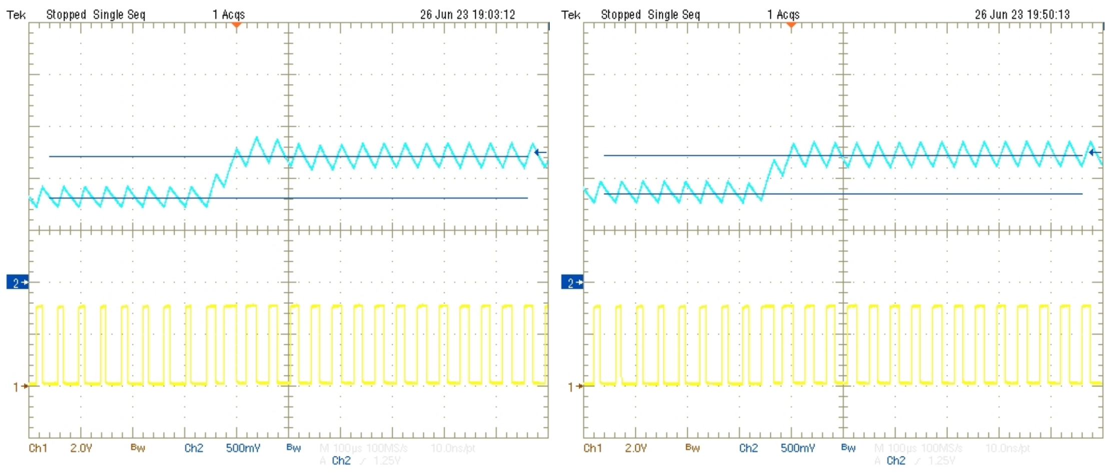
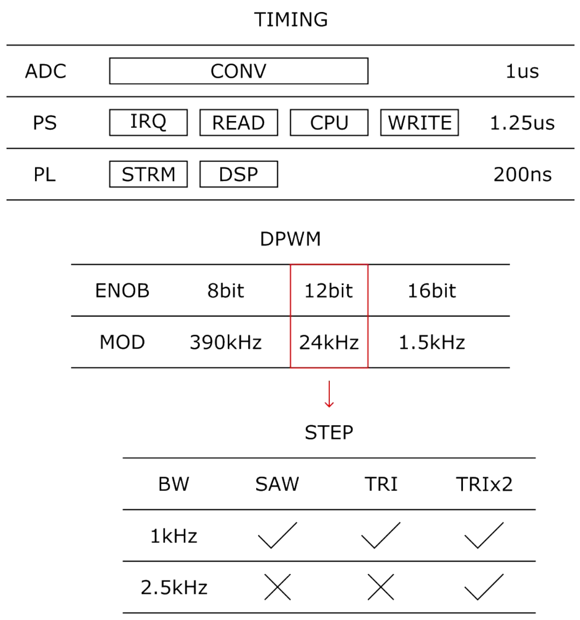


Figure 4.6: Step response of the symmetric modulation with the loop closed at a crossing frequency of $2.5\,kHz$, using a single acquisition per period and a double acquisition per period technique

A DPWM with a triangular ramp can be updated both at the peak or at the valley of the waveform, therefore the update period is every half period $T_U = T_P/2$, which is reduced by a factor of 2 with respect to the sawtooth ramp modulation, where $T_U = T_P$. By using a single acquisition per period, either at the valley or at the peak, the step response is slightly improved because the total delay, evaluated from the acquisition instant to the comparing instant after the update, is always less than a modulation period $T_D < T_P$. Finally, using the double acquisition at the peak and at the valley, the delay is halved and the system turns to be more reactive.

4.2 Conclusions



TIMING

ADC	CONV				1us
PS	IRQ	READ	CPU	WRITE	1.25us
PL	STRM	DSP			200ns

DPWM

ENOB	8bit	12bit	16bit
MOD	390kHz	24kHz	1.5kHz

STEP

BW	SAW	TRI	TRIx2
1kHz	✓	✓	✓
2.5kHz	×	×	✓

Figure 4.7: Summary of settings and experimental results

The Zynq technology enables rapid prototyping and validation of modules, that make it a valuable tool in the field of system design and development. The design flow allows for continuous integration of functionalities and continuous optimization of modules, resulting in a better understanding of system requirements and less complex debugging. Even if the Zynq has a higher price compared to application specific microcontroller, most of the modules turns to be portable across programmable devices, saving NRE costs and time.

The DPWM modules implemented in the FPGA is clocked by the PS at a frequency $F_{CLK} = 100MHz$ and can reach high modulation frequency $F_P = 390kHz$ using a reduced resolution of 8 bit. To increase the resolution, keeping the same modulation frequency, the DPWM should be clocked by the CMT, that can synthesize a clock frequency up to $MMCM\_F_{OUTMAX} = 800MHz$. However, most of the components inside the Artix-7 FPGA can't handle this clock rate, but are designed to work at lower frequencies that depend on the speed grade of the device, as described in the switching characteristic data sheet [12]. With a series of optimizations, it would be possible to double the clock frequency, to get 1 extra bit of resolution or double the modulation period keeping the same resolution. Even thought these optimizations are possible, they are useless by themselves, because the speed of a closed loop control system is always dictated by the slowest operation.

The ILA debugger allows to measure the time of each operation, and it shows that the two most time expensive operations are the ADC conversion and the PS to

PL communication to complete the IRQ. The ADC takes a total conversion time of $T_{ADC} = 1\mu s$ when the ADC is clocked at the maximum speed $104MHz$, while the IRQ handler is completed in a variable amount of time, that mainly depends on the 4 AXI read and write transaction, two to read and clear the interrupt and two to fetch and send back the data to the PL. From experimental results, when the AXI Manager GP port is clocked at 100 MHz, the communication between PS and PL takes around $25T_{CLK} = 250ns$ to perform each read or write transaction, therefore the IRQ handler can be computed in $T_{PS} = 1.25\mu s$. Then, the minimum modulation period should be sufficiently long to fit both the conversion and the IRQ handler, starting from the acquisition instant such that $T_P \geq T_{ACQ} + T_{ADC} + T_{PS}$. When the acquisition instant coincides with the beginning of the modulation period $T_{ACQ} = 0$, the DPWM modulation frequency can reach its limits, but this configuration introduces a long delay before the update instant of the comparator, degrading the controller performances. Moreover, in many cases, it is required to sample the feedback signal in other instant of the modulation period, like in the middle of the ON period, which might be closer to the end of the modulation period.

By implementing the signal processing in the hardware with less than 10 cascaded DSP blocks, the elaboration time can be reduced to one order of magnitude lower than the ADC conversion time, making possible to reach higher modulation frequency limits and sample the feedback system much closer to the update instant. This improvement enables different techniques to make the DPWM control loop more reactive, to fully exploit the duty cycle range and to reach higher modulation frequencies.

Glossary

ADC Analog to Digital Converter.
ADCLK ADC Clock.
ALU Arithmetic Logic Unit.
APU Application Processor Unit.
ASIC Application Specific Integrated Circuit.
ASSP Application Standard Product.
AXI Advanced eXtensible Interface.

CAD Computer-Aided Design.
CAN Control Area Network.
CLB Configurable Logic Block.
CONVST Conversion Start.
CPU Central Processor Unit.

DAC Digital to Analog Converter.
DCLK DRP Clock.
DEN Data Enable.
DFF D-Type Flip-Flop.
DI Data Input.
DMA Direct Memory Access.
DO Data Output.
DPWM Digital PWM.
DRP Dynamic Reconfiguration Port.
DSP Digital Signal Processor.
DWE Data Write Enable.

EMIO Extended Multiplexed IO.
EOC End-Of-Conversion.

FPGA Field Programmable Gate Array.
FPU Floating Point Unit.

GIC	Global Interrupt Controller.
GIER	Global Interrupt Enable Register.
GP	General-Purpose.
GPU	Graphic Processing Unit.
HDL	Hardware Description Language.
HP	High-Performance.
IDE	Integrated Development Environment.
ILA	Integrated Logic Analyzer.
IP	Intellectual-Property.
IPIER	IP Interrupt Enable Register.
IPISR	IP Interrupt Status Register.
IRQ	Interrupt Request.
JTAG	Joint Test Action Group.
LUT	Look-Up Table.
MCU	Micro Controller Unit.
NRE	Non-Recurring Engineering.
PL	Programmable Logic.
PS	Processing System.
PWM	Pulse Width Modulation.
RAM	Random Access Memory.
SDK	Software Development Kit.
SIMD	Single Instruction Multiple Data.
SoC	System-on-Chip.
SPI	Serial Peripheral Interface.
VHDL	VHSIC Hardware Description Language.
VLSI	Very Large Scale Integration.

Bibliography

[1] Arm. *AMBA AXI Protocol Specification.* URL: https://developer.arm.com/documentation/ihi0022/latest.

[2] Arm. *Learn the architecture - An introduction to AMBA AXI (V3.0).* URL: https://developer.arm.com/documentation/102202/0300/?lang=en.

[3] Arm. *Learn the Architecture - Introducing Neon (V1.0).* URL: https://developer.arm.com/documentation/102474/0100/?lang=en.

[4] Arm. *Learn the architecture - Introducing the Arm Architecture (V2.1).* URL: https://developer.arm.com/documentation/102404/0201/?lang=en.

[5] Louise H. Crockett et al. *The Zynq Book. Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC.* Ed. by Strathclyde Academic Media. 2014.

[6] Louise H. Crockett et al. *The Zynq Book Tutorials. for Zybo and ZedBoard.* Ed. by Strathclyde Academic Media. 2015.

[7] Adam Taylor. *The MicroZed Chronicles.* Adiuvo Engineering site. URL: https://www.adiuvoengineering.com/microzed-chronicles-archive.

[8] Xilinx. *7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide (UG480).* URL: https://docs.xilinx.com/r/en-US/ug480_7Series_XADC.

[9] Xilinx. *7 Series FPGAs Configurable Logic Block User Guide (UG474).* URL: https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB.

[10] Xilinx. *7 Series FPGAs Data Sheet: Overview (DS180).* URL: https://docs.xilinx.com/v/u/en-US/ds180_7Series_Overview.

[11] Xilinx. *7 Series FPGAs DSP48E1 User Guide (UG479).* URL: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1.

[12] Xilinx. *Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics (DS181).* URL: https://docs.xilinx.com/v/u/en-US/ds181_Artix_7_Data_Sheet.

[13] Xilinx. *Vivado Design Suite User Guide: Synthesis (UG901)*. URL: https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis.

[14] Xilinx. *Vivado Design Suite: AXI Reference Guide (UG1037)*. URL: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide.

[15] Xilinx. *XADC Wizard v3.3 LogiCORE IP Product Guide (PG091)*. URL: https://docs.xilinx.com/v/u/en-US/pg091-xadc-wiz.

[16] Xilinx. *Zynq-7000 SoC Data Sheet: Overview (DS190)*. URL: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview.

[17] Xilinx. *Zynq-7000 SoC Technical Reference Manual (UG585)*. URL: https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM.

[18] Xilinx. *Zynq-7000 SoC: Embedded Design Tutorial (UG1165). A Hands-On Guide to Effective System Design*. URL: https://docs.xilinx.com/v/u/2019.2-English/ug1165-zynq-embedded-design-tutorial.