**Eloi Vives Bonet**

# Disseny i simulació VHDL/Simulink d'un controlador PID sobre FPGA.

**Treball Fi de Grau**

**dirigit pel Dr. Enrique Fernando Cantó Navarro**

**Grau en Enginyeria Electrònica Industrial i Automàtica**

UNIVERSITAT ROVIRA I VIRGILI

**Tarragona**

**2022**

# Índex

# 1 Introducció

En aquest treball es modelitzarà un motor DC (Direct Current) i es dissenyarà un controlador PID (Proporcional, Integral i Derivatiu) en una FPGA (Field Programmable Gate Array) la qual regularà la velocitat del motor. A l'hora de dissenyar el controlador s'estudiaran els efectes dels diversos elements del llaç de control.

Es partirà d'una pràctica (referència [4]) del màster de les tecnologies del vehicle elèctric. Aquesta pràctica consisteix a dissenyar un controlador PID en una FPGA amb l'ajuda de Simulink i system generator. Aquesta ja té creats els següents blocs.

- Preescaler
- Sensor
- Taula
- PWM
- Display

La planta a controlar és un motor DC (Direct Current), i segons les dades del fabricant aquest gira a 1200 rpm a 12 V. La velocitat d'aquest s'obté mitjançant un codificador rotacional incremental, que consisteix en un disc de 20 forats connectat a l'eix del motor, un LED i un fototransistor. Si l'alineament del disc ho permet la llum d'un LED és rebuda pel fototransistor, la tensió d'aquest es compara respecte a una tensió de referència per mitjà d'un LM393. S'actua sobre el motor per mitjà d'un driver (pont en H LM298N IC) usant PWM (Pulse Width Modulation) el valor del duty cycle aplicat és donat pel controlador PID a dissenyar.

En aquesta pràctica no s'aprofundeix, ni en la modelització del motor i en la dels diferents elements (PWM, sensor/taula), del llaç de control. A l'hora de simular el llaç de control s'utilitza un sensor i PWM ideals.

L'objectiu del treball és la simulació del PID amb el motor i els models del PWM i el sensor/taula, per així poder estudiar els efectes del PWM i el sensor/taula en el llaç de control.

L'entorn utilitzat per fer la simulació és Simulink, aquest s'ha ampliat amb system generator, una llibreria de blocs integrables (sumadors, restadors, multiplicadors, retards), els quals es comunicaran mitjançant fixed-point, que permet dissenyar i simular sistemes per FPGA de Xilinx.

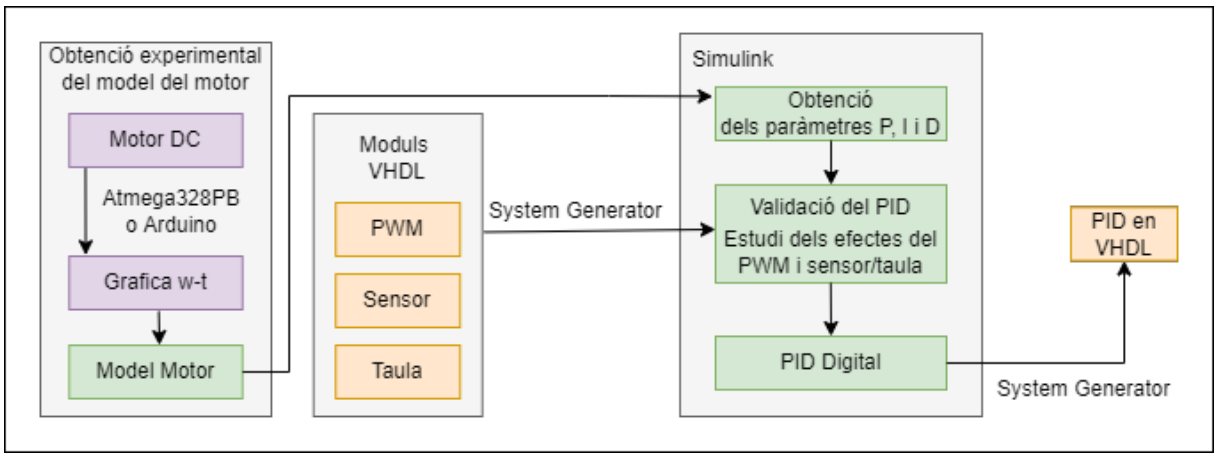



**Figura 1.1** Esquema de l'obtenció del mòdul PID en VHDL.

Com es pot interpretar de la figura anterior, el pla a seguir serà, primerament, modelitzar el motor. Es farà amb l'ajut de dos microcontroladors els quals enviaran informació de la velocitat del motor via serial. Seguidament es dissenyarà el PID digital amb l'ajuda del model de la planta. Amb el PID i el model s'observaran els efectes del PWM, el codificador, la taula i el sensor. Finalment s'obtindrà el PID, amb ajut de system generator, en llenguatge VHDL.

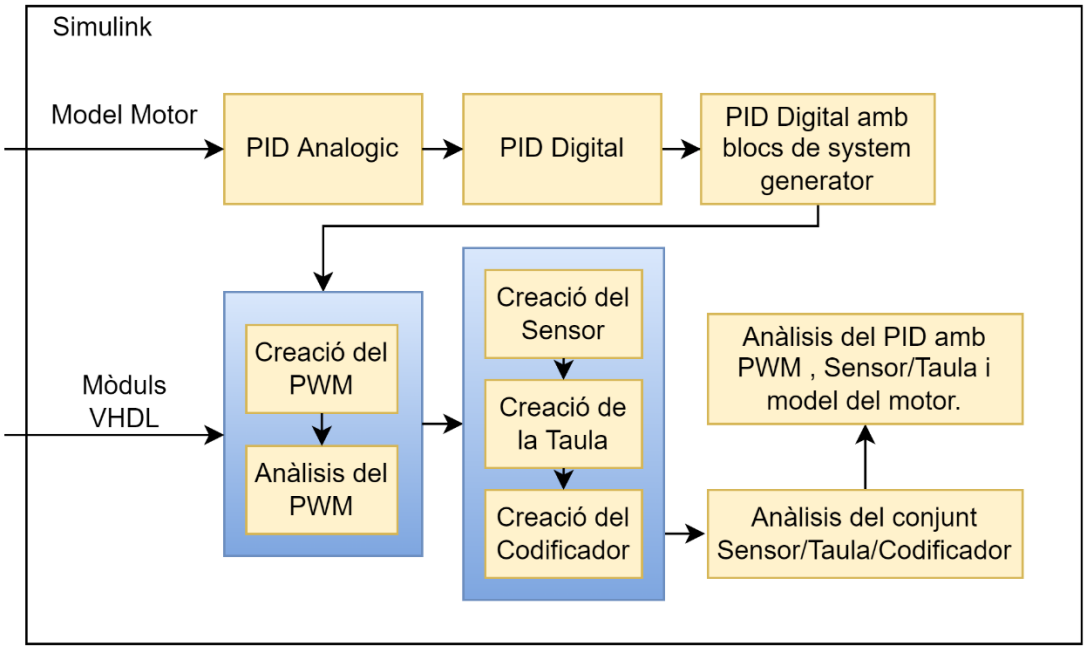



**Figura 1.2** Esquema del anàlisis dels diversos blocs en Simulink.

# 2 Informació preliminar

## 2.1 FPGA

Una FPGA és un dispositiu semiconductor que conté blocs de lògica, interconnexió i funcionalitat dels quals pot ser configurada 'in situ' mitjançant un llenguatge de programació especialitzat.

La lògica programable pot reproduir des de funcions tan senzilles com les que realitza una porta lògica fins a sistemes complexos en un xip.


**Figura 2.1** FPGA en circuit imprès [2]

Qualsevol circuit d'aplicació específica pot ser implementat en una FPGA, sempre que aquest disposi dels recursos necessaris. Les aplicacions on més comunament s'utilitzen les FPGA inclusiu els DSP (processament de senyals digitals), ràdio definit per software, sistemes aeroespacials i de defensa, prototipus de ASCs, sistemes d'imatges per a medicina, sistemes de visió per a computadors, reconeixement de veu, bioinformàtica, emulació de hardware de computadora. Hem de saber que el seu ús en altres àrees és cada vegada major, sobretot en aquelles aplicacions que requereixen un alt grau de paral·lelisme. [1]

Les FPGAs especialment troben aplicacions en qualsevol àrea o algorisme que pugui fer ús de l'alt grau de paral·lelisme ofert per la seva arquitectura, un dels quals és el trencament de codis, en particular atacs de força bruta a algorismes criptogràfics.

A més a més, cada cop són més usades en aplicacions convencionals d'alt rendiment on els nuclis computacionals com FFT o convolució són implementats en una FPGA en comptes d'un processador d'ús general.

## 2.2 Matlab

Matlab és un entorn de computació numèrica i un llenguatge de programació. Creat per la companyia MathWorks, Matlab permet manipular fàcilment matrius, dibuixar funcions i dades, implementar algorismes, crear interfícies d'usuari, i comunicar-se amb altres programes en altres llenguatges. Tot i que s'especialitza en computació numèrica, Simulink és una caixa d'eines opcional (toolbox). [9]

## 2.3 Simulink

Una de les extensions de Matlab és Simulink, el qual permet muntar tota mena de sistemes usant blocs. Alguns dels sistemes son:

- Temps continu
- Temps discret
- Codi embedded
- Sistemes físics

Un cop el sistema s'ha muntat, permet simular-lo, i en alguns casos compilar-lo, per així poder-se usar posteriorment. Aquest procés facilita enormement el procés de programació i debug. Els blocs es poden anar unint en subsistemes i aquests en subsubsistemes, i finalment permeten, en cas que es faci correctament, un entorn clar, visual i entenedor. S'han creat una sèrie d'extensions per Simulink, la que es farà servir en aquest treball és System Generator.

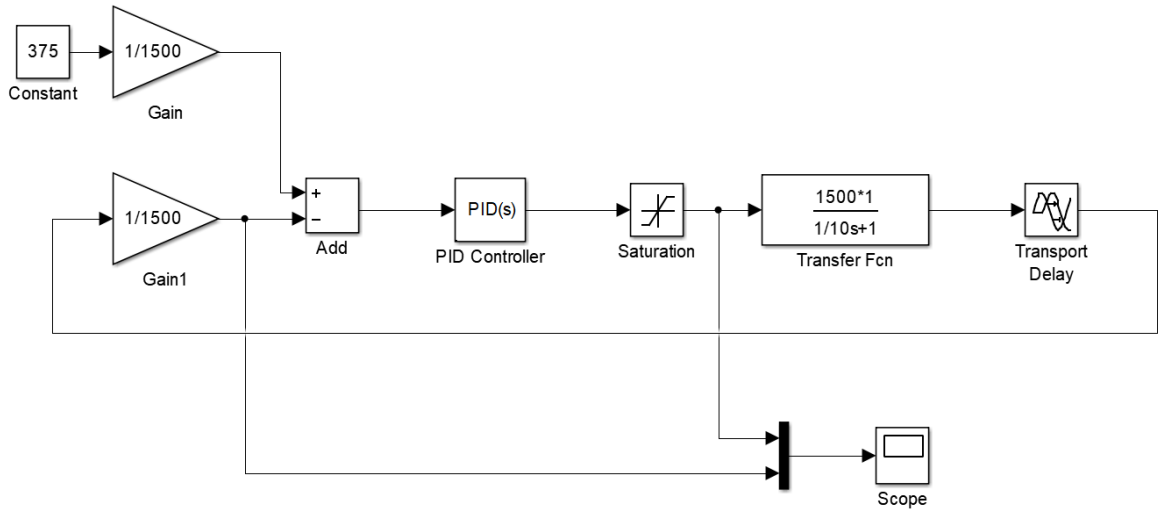



**Figura 2.2** Exemple de l'ide de Simulink.

### 2.3.1 System Generator

System generator és una llibreria la qual permet generar codi HDL a partir de models de Simulink o permet crear models Simulink a partir de codi HDL.

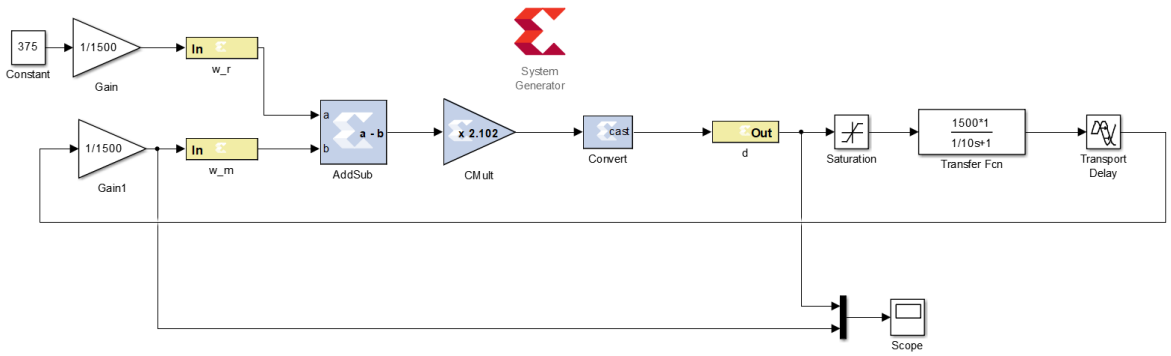



**Figura 2.3** Exemple de Simulink amb System Generator.

Dos dels blocs més importants de la llibreria son Gateway in i Gateway Out

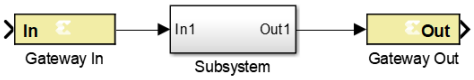



**Figura 2.4** Gateway in i Gateway Out.

Aquests blocs defineixen l'entrada (Gateway In) i sortida (GateWay Out) de la FPGA, entremig es col·loquen els elements de la FPGA que es volen modelar.

Gateway In permet definir el nombre de bits i el tipus (booleà, punt-fixe o punt-flotant) amb el qual es representa un senyal dins la FPGA. De la mateixa manera Gateway Out permet definir com representar les dades de la FPGA a la sortida del bloc. En el cas del PID s'usarà fixed point.

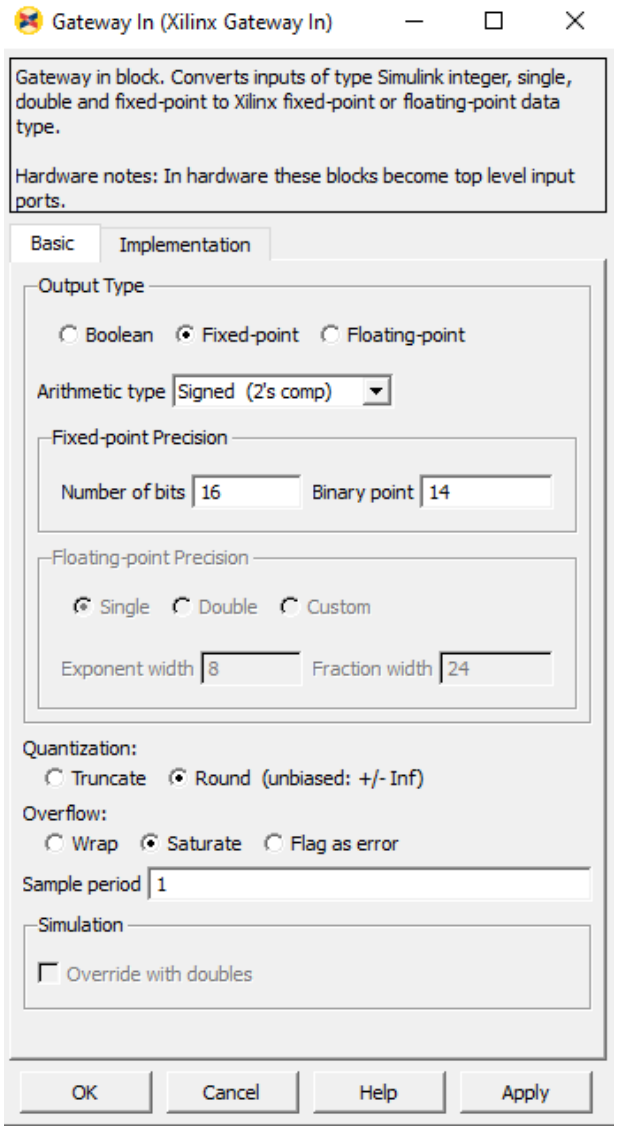



**Figura 2.5** Configuració de Gateway In.

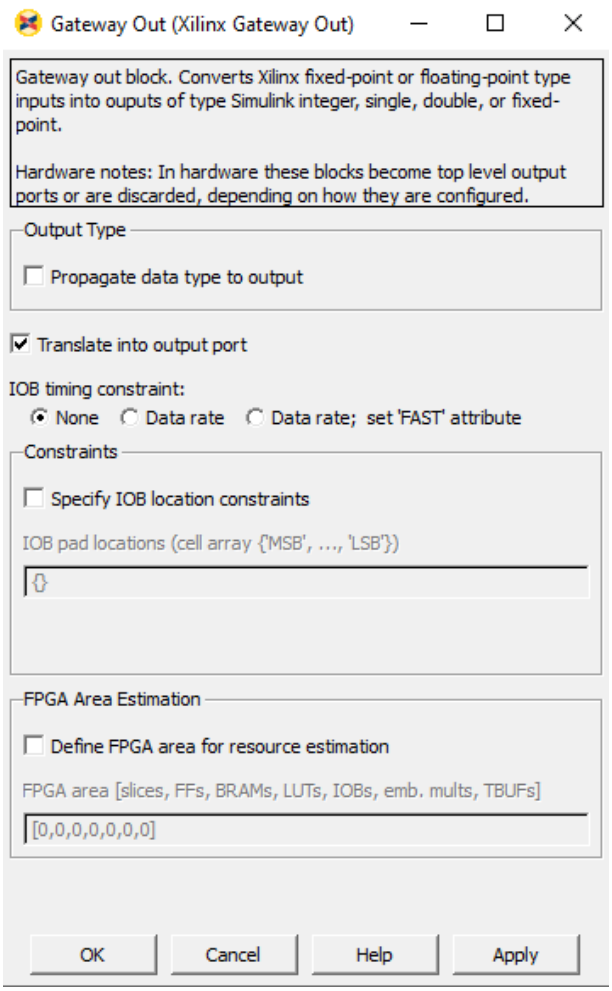



**Figura 2.6** Configuració de Gateway Out.

Un altre bloc molt important és System Generator aquest permet configurar tant l'autogeneració del codi VHDL com el període de simulació.


**Figura 2.7** System Generator.

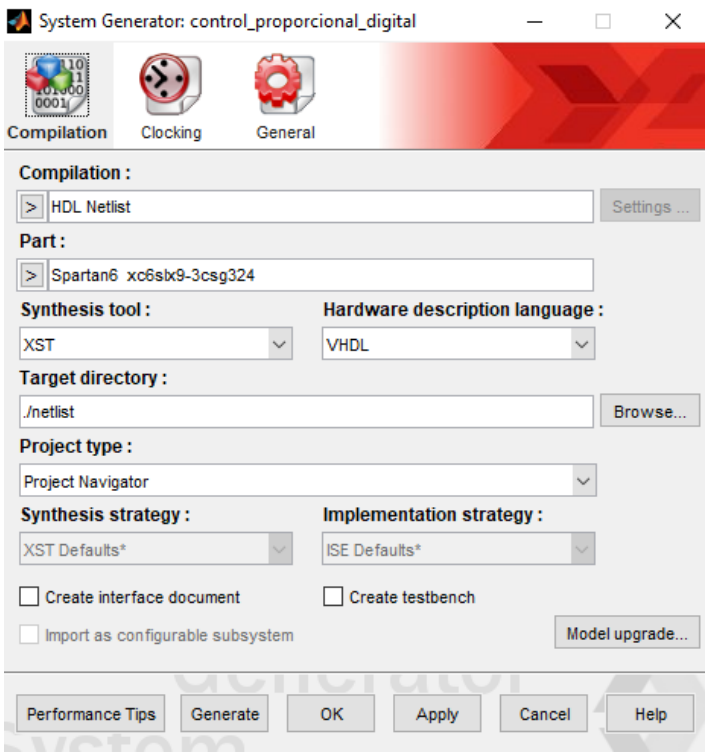



**Figura 2.8** Configuració System Generator.


**Figura 2.9** Configuració System Generator.

En la figura 2.8 es pot configurar per a quina FPGA i on s'ha de guardar el codi auto-generat, aquest serà el que està entre Gateway Ins i GatwayOuts.

En la figura 2.9 es configura la simulació i la freqüència de rellotge de la FPGA.

L'últim bloc remarcable de la llibreria el Black box. Aquest bloc permet simular codi VHDL ja creat a Simulink. Un cop es crea el bloc, també es crea un arxiu de configuració d'aquest. Aquest arxiu permet configurar tant la freqüència del clock enable com genèric en cas que el bloc en tingui.

```
function setup_as_single_rate(block,clkname,cename)
  inputRates = block.inputRates;
  uniqueInputRates = unique(inputRates);
  if (length(uniqueInputRates)==1 & uniqueInputRates(1)==Inf)
    block.addError('The inputs to this block cannot all be constant.');
    return;
  end
  if (uniqueInputRates(end) == Inf)
     hasConstantInput = true;
     uniqueInputRates = uniqueInputRates(1:end-1);
  end
  if (length(uniqueInputRates) ~= 1)
    block.addError('The inputs to this block must run at a single
rate.');
    return;
  end
  theInputRate = uniqueInputRates(1);
  for i = 1:block.numSimulinkOutports
     block.outport(i).setRate(theInputRate);
  end
  block.addClkCEPair(clkname,cename,413);
  return;
```

En el codi anterior la funció "block.addClkCEPair" té com a entrada "413". Aquest nombre indica la relació que hi ha entre el nombre de cops que s'actualitzen les dades d'entrada i el nombre de cops que el bloc s'actualitza.

Els blocs amb què es crearà el PID són sumadors, restadors i multiplicadors, retards i acumuladors. Aquests seran configurats en punt fixe (fixed-point). [10]

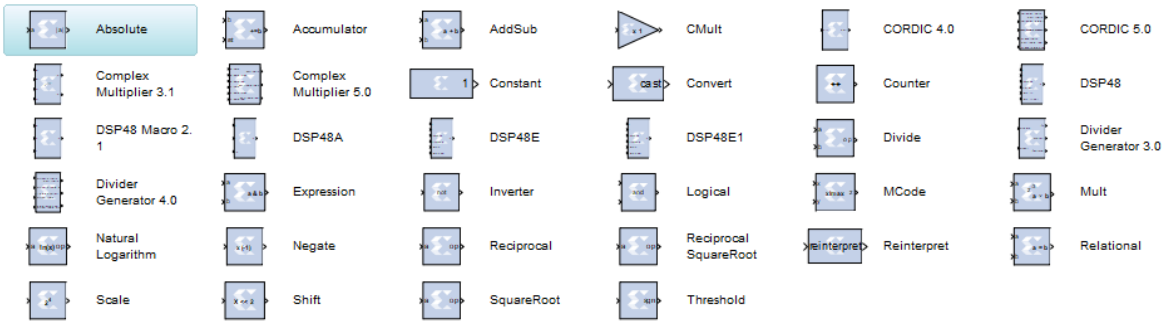



**Figura 2.10** Alguns dels blocs de System Generator.

## 2.4 Planta

La planta que es voldrà controlar consisteix en un motor, que té un codificador rotacional relatiu a l'eix que dona informació sobre la velocitat i posició del motor. El motor està alimentat mitjançant un pont en H, i tot i que el pont i el motor ho permeten només es farà anar el motor en una direcció, ja que el codificador actual no permet determinar la direcció de gir del motor, fent impossible dissenyar un control bidireccional en direcció.

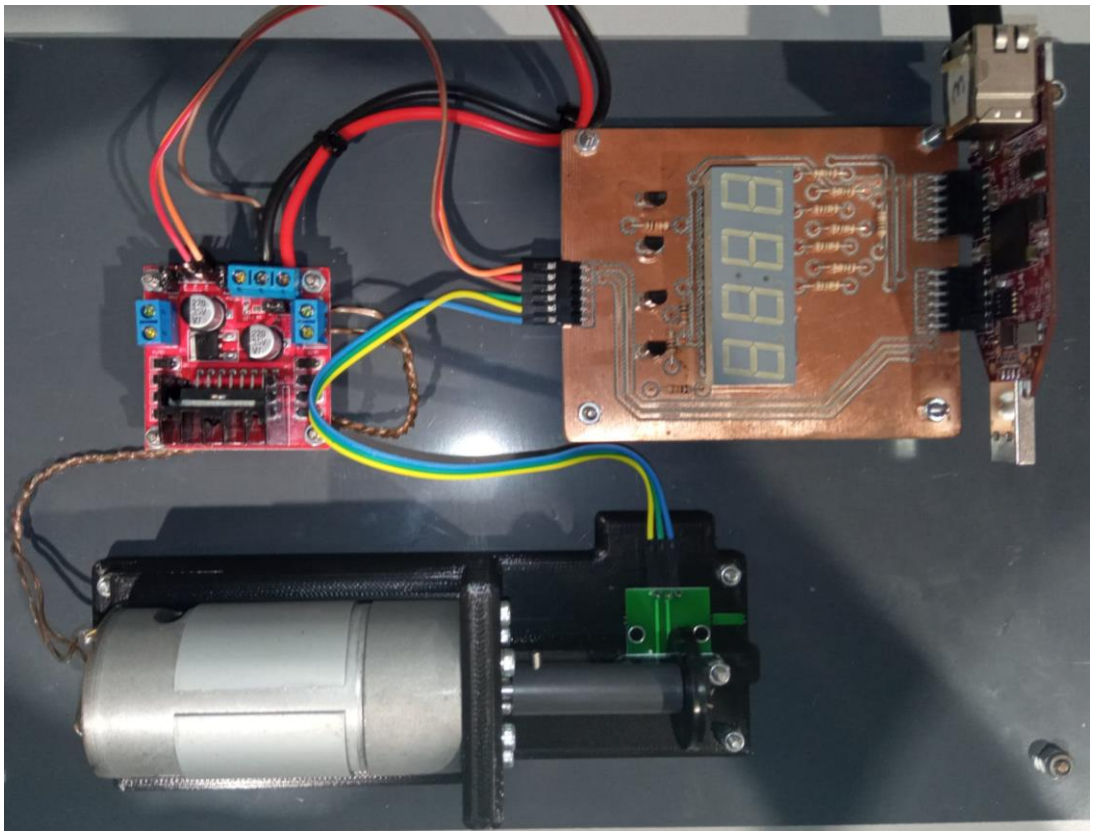


**Figura 2.11** Planta a controlar.

La planta és pot simplificar en un model amb blocs:

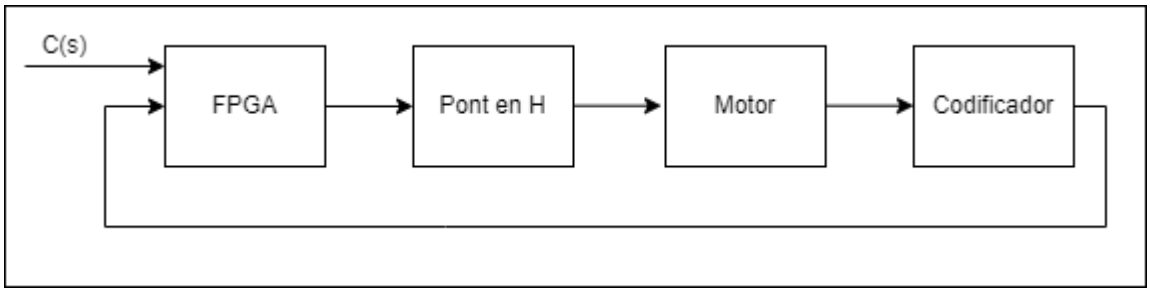



**Figura 2.12** Model de la planta.

### 2.4.1 Motor

Es farà servir un motor DC el qual gira a 1200 rpm, sense carrega, a 12V. S'usarà un PWM controlat pel PID dins la FPGA. Si D és el duty cylce (0≥ D ≥ 1), tenim:

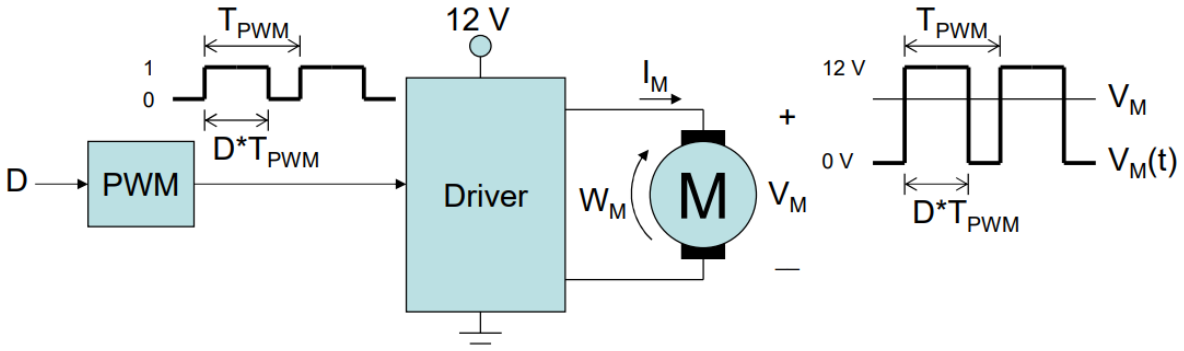



**Figura 2.13** Model del motor. [4]

Un motor DC és pot representar amb el circuit següent:

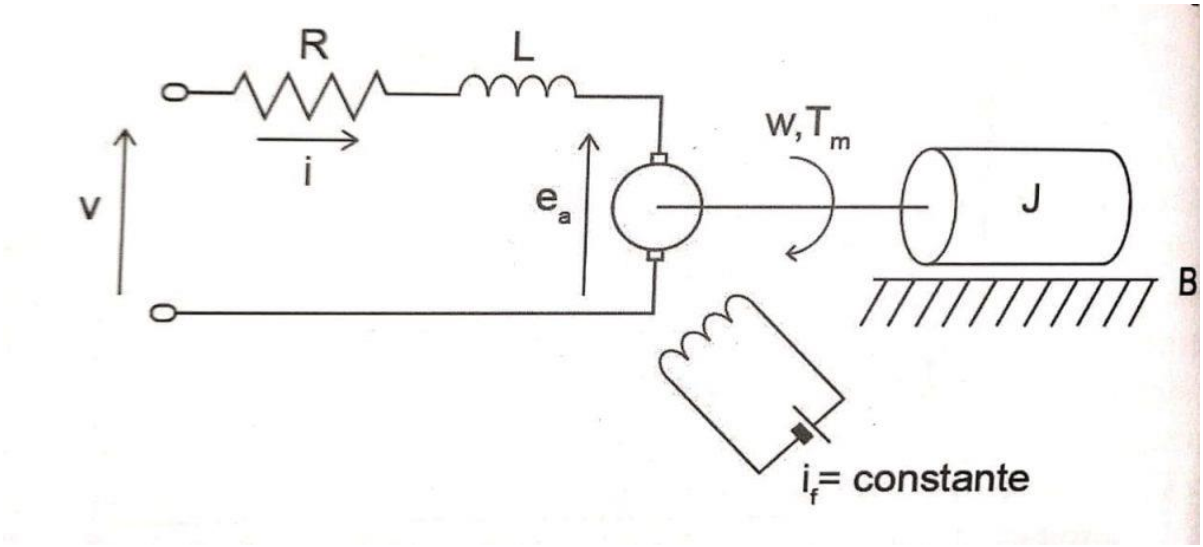



**Figura 2.14** Model electric d'un motor DC. [11]

Aplicant malles arribem al sistema d'equacions diferencials següent.

$$v(t) = Ri(t) + L\frac{di(t)}{dt} + E_a(t)$$

$$T_m(t) = B\omega(t) + J\frac{d\omega(t)}{dt}$$

$$T_m(t) = K_m i(t)$$

$$E_a(t) = K_a\omega(t)$$

**Equació 2.1** Sistema d'equacions diferencials del motor DC.

Per resoldre aquest sistema s'aplicarà la transformada de laplace.

$$\begin{gathered} \mathrm{Lsi(s)} = \mathrm{v(s)} - \mathrm{Ri(s)} - \mathrm{E_a(s)} \\ \mathrm{Js\omega(s)} = \mathrm{Tm(s)} - \mathrm{B\omega(s)} \\ \mathrm{T_m(s)} = \mathrm{K_m i(s)} \\ \mathrm{E_a(s)} = \mathrm{K_a \omega(s)} \end{gathered}$$

**Equació 2.2** Sistema d'equacions diferencials del motor.

Ara que el sistema d'equacions diferencials és un sistema d'equacions lineals resolem per mètode preferit.

$$\frac{\omega(s)}{v(s)} = \frac{K_m}{LJs^2 + (RJ + LB)s + K_m K_a}$$

**Equació 2.3** Funció de transferència velocitat angular – tensió del motor.

El nostre motor no ha de moure cap carrega i la inèrcia del rotor és pot considerar menyspreable respecte la resta de termes. Per tant els termes que continguin J és considerant nuls.

$$\frac{\omega(s)}{v(s)} = \frac{K_m}{LBs + K_m K_a}$$

**Equació 2.4** Funció de transferència velocitat angular – tensió simplificada del motor.

En aquest model s'inclourà un retard a causa del possible joc entre l'eix del motor i el codificador.

$$\frac{\omega(s)}{v(s)} = \frac{K_m}{LBs + K_m K_a} e^{-d_t \cdot s}$$

**Equació 2.5** Funció de transferència final del motor.

### 2.4.2 Driver

Per controlar el motor s'usarà el driver LM298N IC, donat que la placa no pot donar prou corrent al motor. Aquest driver és un pont en H format per transistors NPN.

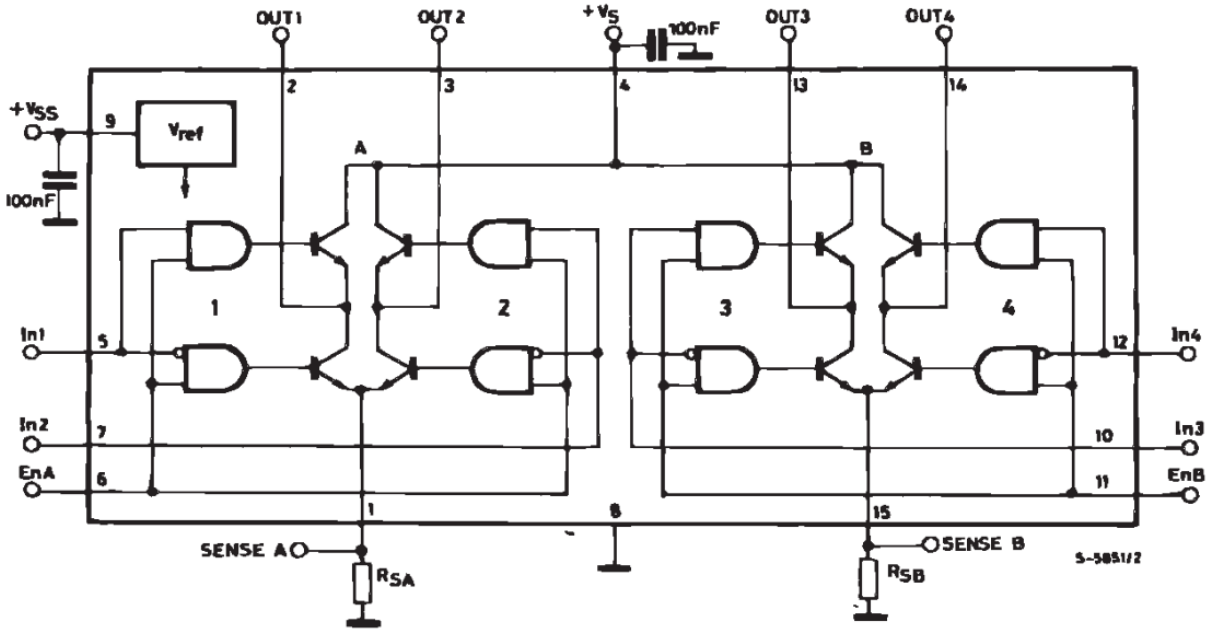
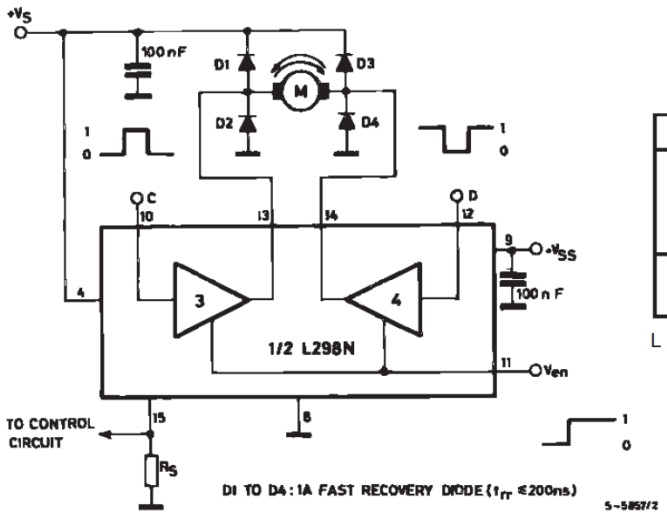



**Figura 2.15** Diagrama de blocs del driver. [5]

El driver té sortides per dos motors DC, però només s'usarà un. Cada sortida té tres pins que controlar, EN, in1 i in2. En cas que EN estigui desactivat (EN = 0), la sortida del driver serà sempre de 0V i, en cas que EN estigui activat (EN = 1), si in1 esta high i in2 low el motor experimentarà una tensió Vcc positiva, i en cas que in1 estigui low i in2 high el motor experimentarà una tensió negativa. Per la planta es deixarà in2 sempre a low i ENA sempre high, controlant així el duty cycle només amb in1, fent que el motor experimenti una tensió mitjana de D·Vcc.




| Inputs | | Function |
|---|---|---|
| $V_{en}$ = H | C = H ; D = L | Forward |
| | C = L ; D = H | Reverse |
| | C = D | Fast Motor Stop |
| $V_{en}$ = L | C = X ; D = X | Free Running Motor Stop |

L = Low H = High X = Don't care

**Figura 2.16** Control bidireccional del motor. [5]

### 2.4.3 Codificador

Consisteix en un comparador LM393 el qual emet polsos detectables per la FPGA. Es compararà una tensió de referència amb la sortida d'un fototransistor. La tensió que aquest doni dependrà de si la llum produïda per un díode (que està col·locat a l'altra banda del disc) és bloquejada o pot passar per un forat del disc.

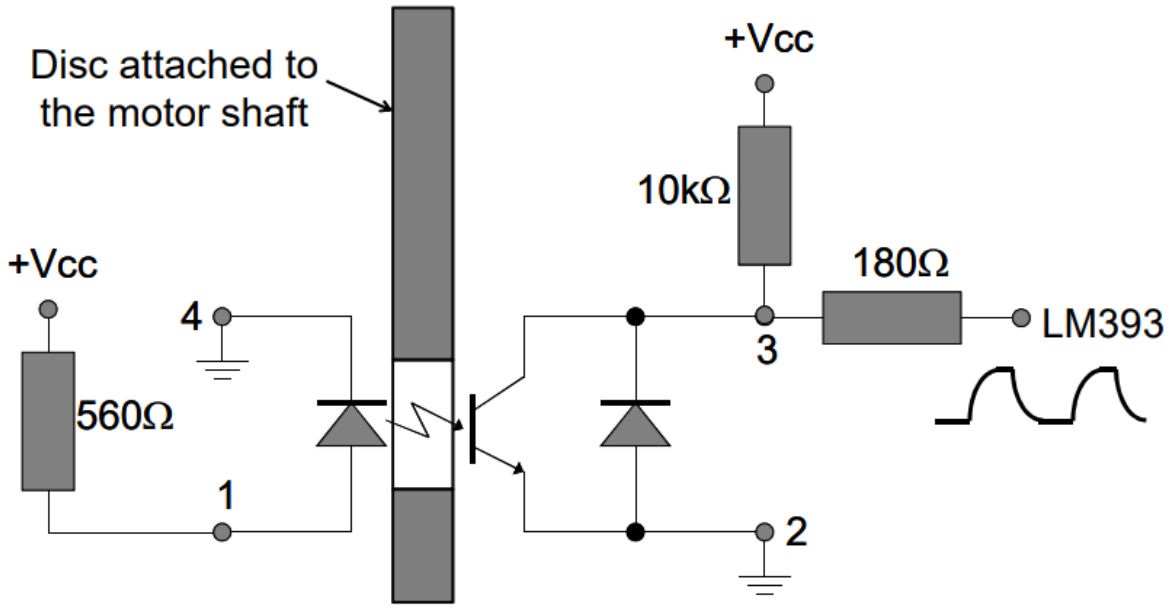



**Figura 2.17** Esquema del codificador. [4]

### 2.4.4 LX9 Microboard

Per poder desenvolupar el controlador PID s'usarà una placa de desenvolupament la qual conte una FPGA (XC6LX9-2CSG324C, Spartan-6 de Xilinx) i diversos perifèrics:

- Un port JTAG (Joint Test Action Group) per programar la FPGA.
- Quatre interruptors
- Quatre LEDs.
- Un polsador el qual s'usa per resetejar les entrades del controlador.
- Connectors J4 i J5 els quals s'usen per controlar un display 7 segments, llegir el codificador, i controlar el driver.

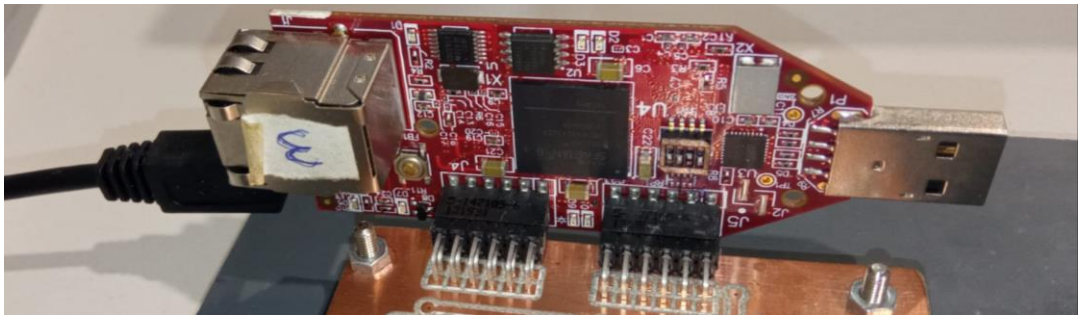


**Figura 2.18** Placa de desenvolupament.

La placa conte Spartan-6 de Xilinx una FPGA baix cost al la qual és pot dissenyar el softcore MicroBlaze. La FPGA XC6LX9-2CSG324C ha estat pensada per a tot tipus d'aplicacions de baix cost mantenint tot i així potencia i rendiment. La placa conte densitats de portes lògiques que varien des de 3840 a 147443 amb la meitat de consum que membres anteriors de la família Spartan. La FPGA conte una rica selecció de sistemes incorporats, entre altres cal destacar:

- 18 Kb (2 x 9Kb) de RAM (Random Access Memory)
- Controladors de memòria SDRAM (Synchronous Dynamic Random-Access Memory)
- Comptabilitat amb PCI (Peripheral Component Interconnect) exprés.
- Sistemes avançats de modes de control de potencia.
- Auto detecció d'opcions de configuració.
- Seguretat IP (Internet Protocol) amb AES (Advanced Encryption Standard) i protecció DNS (Domain Name System).

Aquestes opcions proveeixen una alternativa de baix cost a sistemes personalitzats ASIC. [12]

### 2.4.5 Placa Interface

La placa interface conte tant la sortida del display de 7 segments, controlat pel connector PMOD2, com el control de quin display s'il·lumina, la connexió del sensor i la sortida PWM que va al driver LM298N, controlats pel connector PMOD1. L'esquema elèctric de la Placa Interface és el següent.

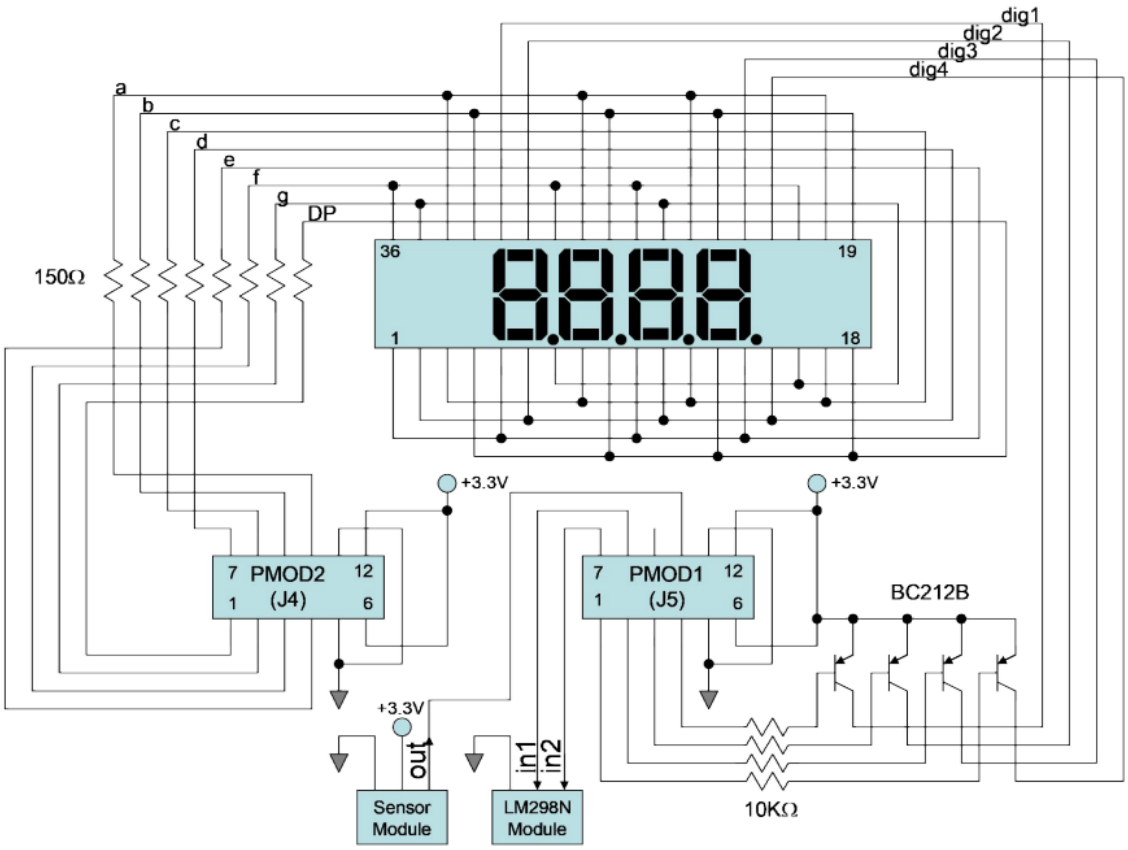



**Figura 2.19** Esquema elèctric de la Placa Interface. [4]

A l'entrada del mòdul display hi ha el nombre a mostrar, de mida està compres entre 0 i 0xFFFF, i cada display mostrarà mig bit. Així doncs, si el nombre a mostrar és 0x4321, el primer display mostrarà el nombre 4, el segon el nombre 3, etc. Els displays van amb lògica negada. Si volem mostrar el nombre 2 tindrem:

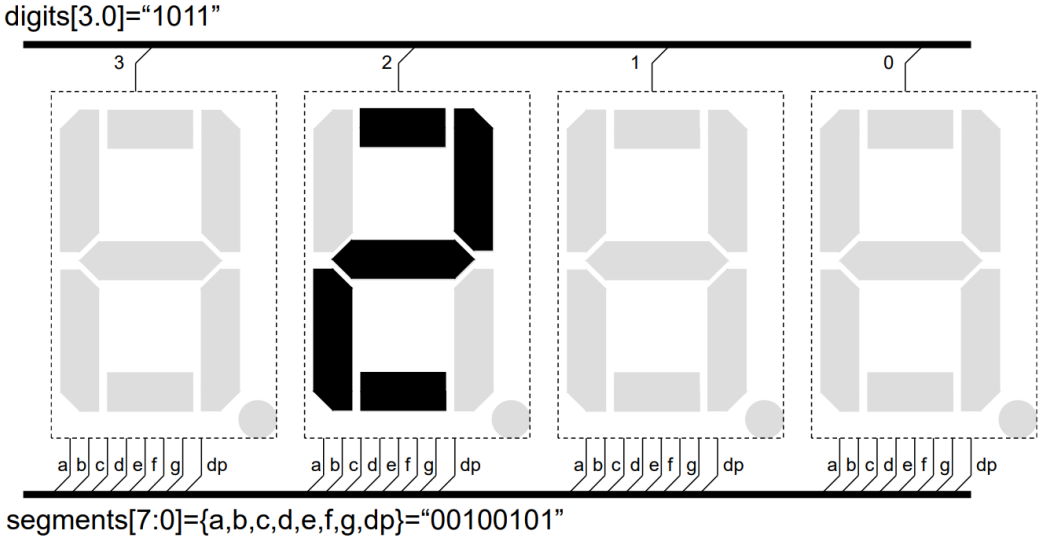



**Figura 2.20** Exemple del display. [4]

Per poder mostrar els diversos nombres s'optarà per canviar el display que s'està usant a alta freqüència donant així la sensació que tots estan oberts i mostrant el nombre que es desitja.

Es necessitarà un multiplexor per passar del nombre que es desitja mostrar a la combinació correcta de LEDs a encendre al display, per donar la sensació que el nombre està en aquest. Es necessitarà un comptador de 2 bits seguit d'un decoder per escollir a quin dels displays s'emet.

# 3 Caracterització del motor

Primerament, es dissenyarà un model del sistema a controlar, és a dir, el motor. Aquest model es pot obtenir amb els paràmetres elèctrics com resistència i inductància del rotor i estator. Tot i això, s'optarà per aconseguir-lo amb la corba de rpm-temps i assumint que el model que representa el motor té un pol a $-\frac{1}{\tau}$ i cert guany $k_m$ i cert retard temporal a causa del joc entre els diferents elements que componen el motor.

$$\frac{\omega(s)}{V(s)} = \frac{k_m}{1+\tau s} e^{-d_t \cdot s}$$

**Equació 3.1** Model del motor.

Per poder obtenir la corba de caracterització necessitarem que el microcontrolador disposi de una sèrie de perifèrics:

1. Timer, per poder mesurar el temps entre forat i forat del codificador.
2. UART, ja que es necessiten obtenir les dades del microcontrolador.
3. Digital I/O, per activar el motor amb precisió i poder llegir el valor de la sortida del codificador.

Per poder assegurar que les corbes obtingudes eren correctes s'han generat les corbes usant dos microcontroladors diferents.

## 3.1 Obtenció de la corba amb ATMEGA328PB

La primera corba s'ha obtingut mitjançat la placa atmega328pb xmini.

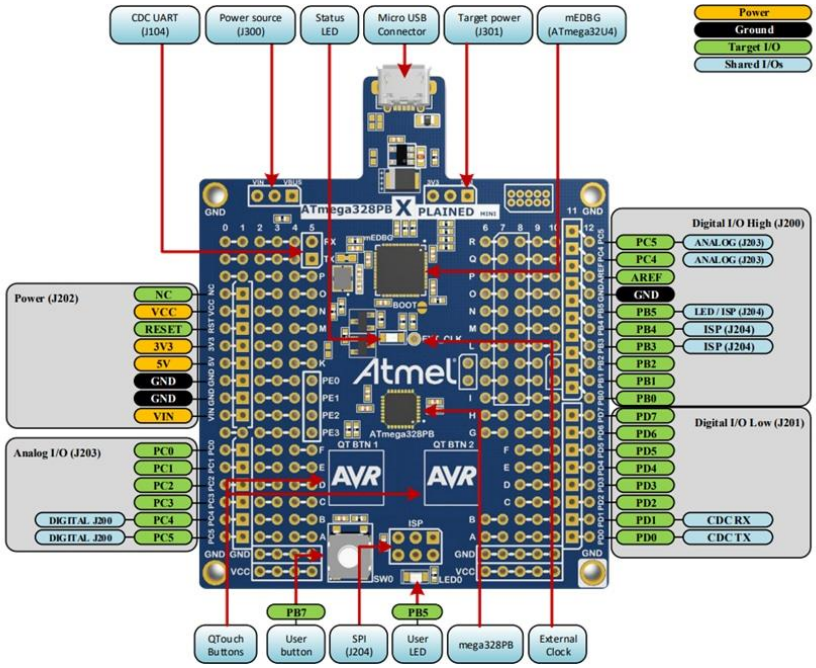



**Figura 3.1** Placa atmega328pb xmini [6]

S'ha escollit aquesta placa de desenvolupament perquè té les funcionalitats necessàries per poder realitzar la caracterització i es volia provar el software atmel START, el qual autogenera el codi dels perifèrics. El microcontrolador conté (entre altres):

1. Timer de 16 bits.
2. Diversos I/O.
3. UART.

Per evitar bugs es programaran els perifèrics i seran testejats de forma individual, i un cop aquests hagin passat el test s'integraran al main. Es proposen els següents testos: per la UART fer un "echo" (rep una paraula a Rx retornar-la per Tx), pels ports I/O es realitzarà un petit programa que faci que la sortida copiï el nivell lògic de l'entrada, i finalment pel timer s'enviarà un caràcter per la UART (la qual ja estarà testejada) que activarà el temporitzador. Un cop hagi passat cert temps s'enviarà un altre caràcter, i des de l'ordinador i mitjançant un script de Python es mesurarà si la diferència entre caràcters és l'esperada.

Un cop tots els mòduls estiguin testejats s'escriurà un main que en rebre un caràcter per la UART enviï el temps (en centenars de µs) que passa entre forat i forat del codificador. Aquesta tasca es realitzarà fins que es rebi un altre caràcter per la UART.

S'haurà de realitzar un programa en Python que enviï un caràcter per la UART, guardi les dades que es reben per aquesta, i un cop hagi passat el temps suficient perquè el motor estigui en estat estacionari s'enviarà un caràcter per parar de rebre dades. Finalment, es farà un script a part per processar les dades.

### 3.1.1 Timer

Per escollir quin mode volem de timer és necessiten tres definicions:

- BOTTOM: Quan el comptador arriba a BOTTOM És zero.
- MAX: El valor màxim del comptador 0xFF per 16 bits i 0xF per 8 bits.
- TOP: El valor més gran que pot arribar el comptador, pot ser MAX o un registre.

Amb aquestes definicions es pot escollir el mode d'operació del timer.

| Mode | WGM1[3] | WGM1[2] (CTC1)[1] | WGM1[1] (PWM1[1])[1] | WGM1[0] (PWM1[0])[1] | Timer/ Counter Mode of Operation | TOP | Update of OCR1x at | TOV1 Flag Set on |
|---|---|---|---|---|---|---|---|---|
| 2 | 0 | 0 | 1 | 0 | PWM, Phase Correct, 9-bit | 0x01FF | TOP | BOTTOM |
| 3 | 0 | 0 | 1 | 1 | PWM, Phase Correct, 10-bit | 0x03FF | TOP | BOTTOM |
| 4 | 0 | 1 | 0 | 0 | CTC | OCR1A | Immediate | MAX |
| 5 | 0 | 1 | 0 | 1 | Fast PWM, 8-bit | 0x00FF | BOTTOM | TOP |
| 6 | 0 | 1 | 1 | 0 | Fast PWM, 9-bit | 0x01FF | BOTTOM | TOP |
| 7 | 0 | 1 | 1 | 1 | Fast PWM, 10-bit | 0x03FF | BOTTOM | TOP |
| 8 | 1 | 0 | 0 | 0 | PWM, Phase and Frequency Correct | ICR1 | BOTTOM | BOTTOM |
| 9 | 1 | 0 | 0 | 1 | PWM, Phase and Frequency Correct | OCR1A | BOTTOM | BOTTOM |
| 10 | 1 | 0 | 1 | 0 | PWM, Phase Correct | ICR1 | TOP | BOTTOM |
| 11 | 1 | 0 | 1 | 1 | PWM, Phase Correct | OCR1A | TOP | BOTTOM |
| 12 | 1 | 1 | 0 | 0 | CTC | ICR1 | Immediate | MAX |
| 13 | 1 | 1 | 0 | 1 | Reserved | - | - | - |
| 14 | 1 | 1 | 1 | 0 | Fast PWM | ICR1 | BOTTOM | TOP |
| 15 | 1 | 1 | 1 | 1 | Fast PWM | OCR1A | BOTTOM | TOP |

**Figura 3.2** Diferents modes de funcionament del timer. [7]

El mode 4 és el que interessa, ja que el TOP es pot modificar mitjançant un registre i no necessitem PWM a cap pin.

Un cop s'ha seleccionat el mode de funcionament s'ha de determinar el valor de TOP. El datasheet del microcontrolador ens dona la fórmula.

$$f_{timer} = \frac{f_{clk}}{N \cdot (1 + OCRnA)} = \frac{f_{clk}}{N \cdot (1 + TOP)}$$

**Equació 3.2** Freqüència del timer.

On:

$f_{clk}$ és la freqüència del microcontrolador (16MHz).

$f_{timer}$ és la freqüència del timer.

$N$ el valor del preescaler.

Si reorganitzem l'equació 3.2:

$$TOP = \frac{f_{clk}}{N \cdot f_{timer}} - 1$$

**Equació 3.3** Freqüència del timer.

Segons l'equació 3.3, per un període de timer de 100 µs es necessiten el següents valors de TOP.

| N(Dec) | TOP(Hex) |
|---|---|
| 1 | 63F |
| 8 | 63 |
| 64 | B |
| 256 | 2 |
| 1024 | 0 |

**Taula 1** TOP en funció de N.

Com que el timer és de 16 bits, s'usarà un preescaler de 1 amb un TOP de 63F. Tot i que qualsevol opció anterior seria vàlida menys el preescaler de 1024, en el cas del timer de 8 bits l'opció de N = 1 tampoc seria vàlida.

Finalment s'habiliten les interrupcions.

### 3.1.2 I/O Port

Primerament es configurarà el pin en mode entrada o sortida

| DDxn | PORTxn | PUD (in MCUCR) | I/O | Pull-up | Comment |
|---|---|---|---|---|---|
| 0 | 0 | X | Input | No | Tri-state (Hi-Z) |
| 0 | 1 | 0 | Input | Yes | Pxn will source current if ext. pulled low |
| 0 | 1 | 1 | Input | No | Tri-state (Hi-Z) |
| 1 | 0 | X | Output | No | Output Low (Sink) |
| 1 | 1 | X | Output | No | Output High (Source) |

**Figura 3.3** Configuració dels I/O [7]

El pin del sensor estarà configurat seguint la configuració remarcada en color verd, ja que no necessita pull-up. Per altra banda, el pin que controla el pont en h segueix la configuració remarcada en vermell.
No s'usaran interrupcions, perquè el temps de processament del main és baix i es pot usar enquesta.

### 3.1.3 UART

A l'hora de configurar la UART només s'ha tingut en compte una cosa i és el baudrate. La resta de paràmetres s'han deixat en default i s'ha autogenerat el codi del perifèric. El baudrate s'ha escollit seguint el següent criteri.

$$B\left[\frac{bits}{s}\right] = \frac{1\,min}{60\,s}\cdot\frac{R\,rev}{1\,min}\cdot\frac{20\,forats}{1\,rev}\cdot\frac{5\,byte}{1\,forat}\cdot\frac{8\,bits}{1\,byte} = R\cdot\frac{40}{3}$$

**Equació 3.4** Calcul del Baudrate

Segons les especificacions del motor la velocitat màxima és 1500 RPM a 12 V. Això dona un baudrate de 20000, per tant, s'escollirà el baudrate estàndard directament més gran a aquest, és a dir, 38400 bits/s.

### 3.1.4 Codi Python

Per la comunicació via serial s'usarà la llibreria pyserial. Les dades a enviar són de tipus uint32_t i, per tant, el protocol que s'utilitzarà enviarà els quatre bytes de les dades de menor a major i un salt de la línia (\n). Python serà l'encarregat de llegir aquests bytes, ajustar-los i guardar-los, i finalment un altre programa llegirà les dades guardades, les processarà i dibuixarà una gràfica.Protocol.

Codi C.

```
Send_uint8(StartCaracter);
Send_uint8((uint8_t)(dT/0xFFFFFF)); //A
Send_uint8((uint8_t)((dT/0xFFFF)%0xFF)); //B
Send_uint8((uint8_t)((dT/0xFF)%0xFF)); //C
Send_uint8((uint8_t)(dT%0xFF)); //D
Send_uint8(EndCaracter);
```

Si un int32_t està compost de 4 bytes A, B, C i D (de major a menor), és separaren i s'envien aquests bytes.

Codi python

```
while time.time() < timeout_start + timeout:
    data = puerto.read_until('\n'.encode('utf-8'), size=5)
    if data.decode('utf-8') == '':
        continue
    time_array.append(int(data.decode('utf-8')))
```

Data retorna un array de bytes, i un cop descodificats i transformats a int es guarden en un array.

## 3.2 Obtenció de la corba amb arduino uno.

La segona corba s'ha obtingut mitjançant un arduino uno.

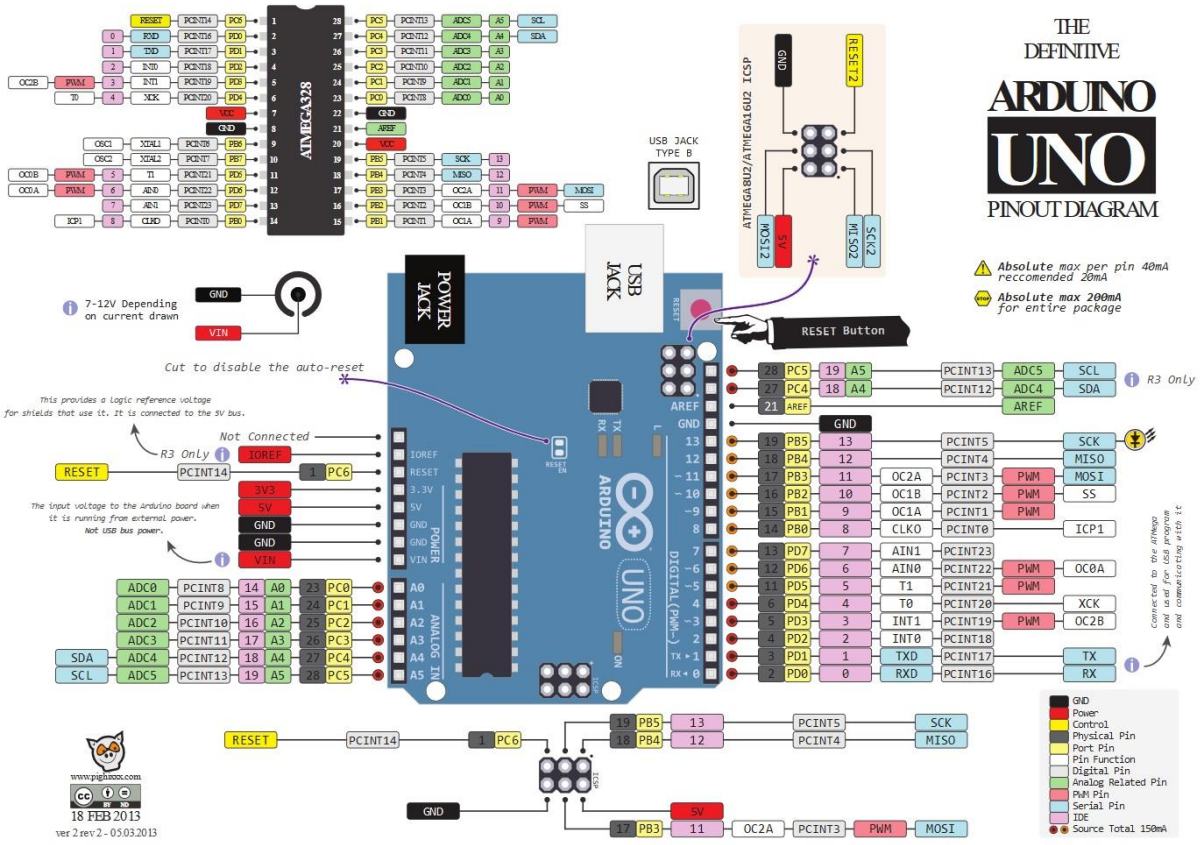



**Figura 3.4** Pinout Arduino Uno [8]

Aquest microcontrolador s'ha escollit donada la seva facilitat de ser programat i les eines que proporciona. L'estratègia usada per obtenir les dades és una diferent de l'anterior: aquest cop no s'usarà python per monitorar el serial port, sinó que s'usarà una eina de l'arduino, el monitor serial, el qual ens mostra per pantalla en format ascii tots els caràcters que es reben per serial i ens permet enviar caràcters de forma manual. Per tant, el codi de l'arduino serà similar al de l'atmega328pb, però aquest cop les dades no les guardarà Python, si no es quedaran guardades al monitor serial ja en format d'enter, i un cop guardades es copiaran a un Excel.

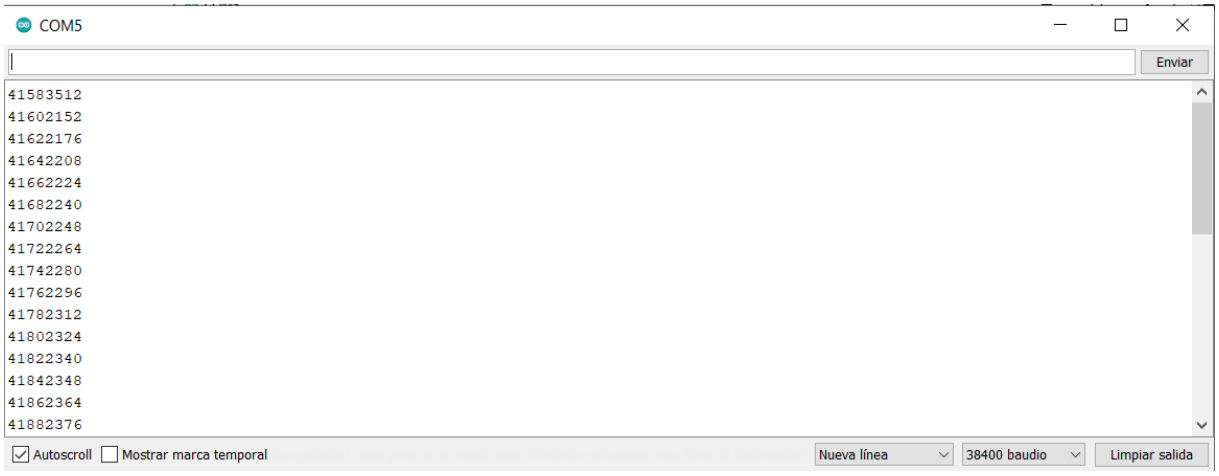



**Figura 3.5** Monitor serial

Aquest, pel serial, enviarà el valor d'un comptador de µs. Això no es farà en format hexadecimal, sinó que s'enviarà el valor decimal. Per tant el màxim nombre de bits que es poden enviar seran $[log(2^{32})] + 1 = 11$, ja que el comptador és un uint32_t i envia un caràcter addicional de salt de línia. Usant la equació 3.4 i canviant 5 bytes per 11 obtenim un baudrate de 44000 bits/s. Tal i com s'ha fet anteriorment , es farà ús el valor estàndard següent, en aquest cas 57600 bits/s.

## 3.3 Tractament de les dades

Hi ha dos tipus de dades principals: les de l'atmega328pb (les quals ens donen el temps entre forat i forat del codificador) i les de l'arduino (cada cop que es detecta un canvi ascendent a l'entrada del codificador, s'envia el temps des que el microcontrolador s'ha obert, en µs). Per solucionar aquest problema es transformaran les dades de l'arduino a les de l'atmega. Si es té una array de N elements, podem aplicar la següent funció.

$$dT[i] = T[i+1] - T[i], \qquad i \in [0, N-2]$$

**Equació 3.5** Calcul d'un increment i.

Obtenint així una nova array, aquest cop de la diferència entre forat i forat del codificador. Ara que les dades que s'estan tractant són iguals, es pot calcular la velocitat a la qual girava l'eix del motor.

$$dT \cdot \omega = d\theta$$

$$dT \cdot \omega \cdot l = d\theta \cdot l = 2\pi \rightarrow \omega = \frac{2\pi}{dT \cdot l}$$

**Equació 3.6** Velocitat angular en funció de la diferencia temporal entre forat i forat.

Es desitja la velocitat angular en rpm i no rad/s.

$$\omega \cdot \frac{60}{2\pi} = rpm = W \rightarrow \omega = W\frac{2\pi}{60} = \frac{2\pi}{dT \cdot l} \rightarrow W = \frac{60}{dT \cdot l}$$

**Equació 3.7** Velocitat angular en funció de la diferencia temporal entre forat i forat.

L'equació anterior serà aplicada a tota l'array dT (en segons) per crear l'array w on es guardaran les velocitats angulars a cada instant de temps, obtenint els següents resultats.

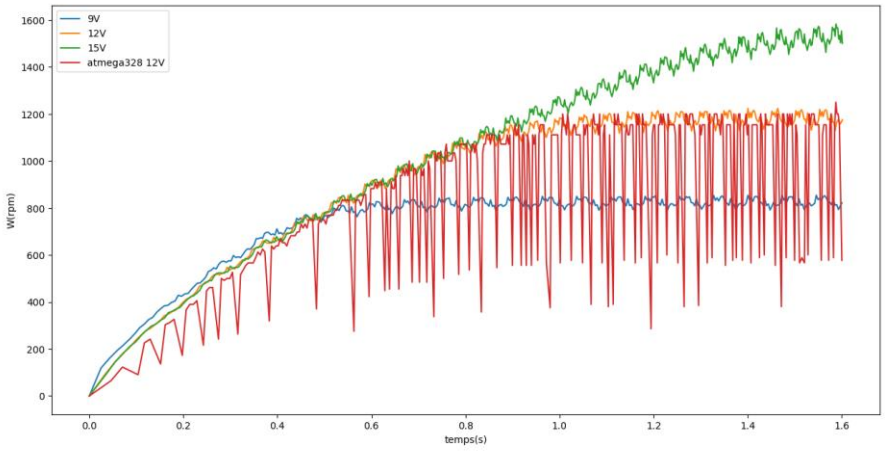



**Figura 3.6** Resposta del motor.

A la figura anterior es pot observar la resposta del motor a esglaons de diferent amplitud. La resposta de l'atmega32pb té una sèrie de pics, ja que el sensor no llegeix un forat en repetides ocasions, fet que feia que el comptador tingués el doble de temps i, per tant, anés a la meitat de velocitat. Això es pot arreglar via software si fem que en cas que un punt P[i] sigui més petit que el 60% del punt P[i-1] no col·loquem a la gràfica el punt P[i] sinó el punt P[i-1].

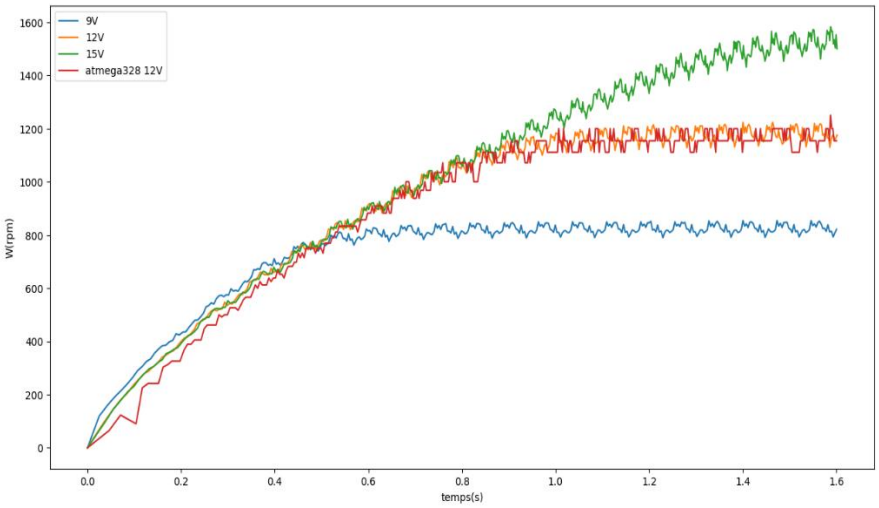



**Figura 3.7** Resposta temporal amb filtre per ATMEGA.

Es pot comprovar com coincideixen de forma quasi exacta la resposta de 12 V de l'arduino amb l'ATMEGA. També es pot observar un arrissat d'alta freqüència a causa de les vibracions de l'eix. Per poder eliminar-lo i poder buscar així de forma més precisa el valor del pol es farà la mitjana de cada punt amb els seus 5 punts anteriors.

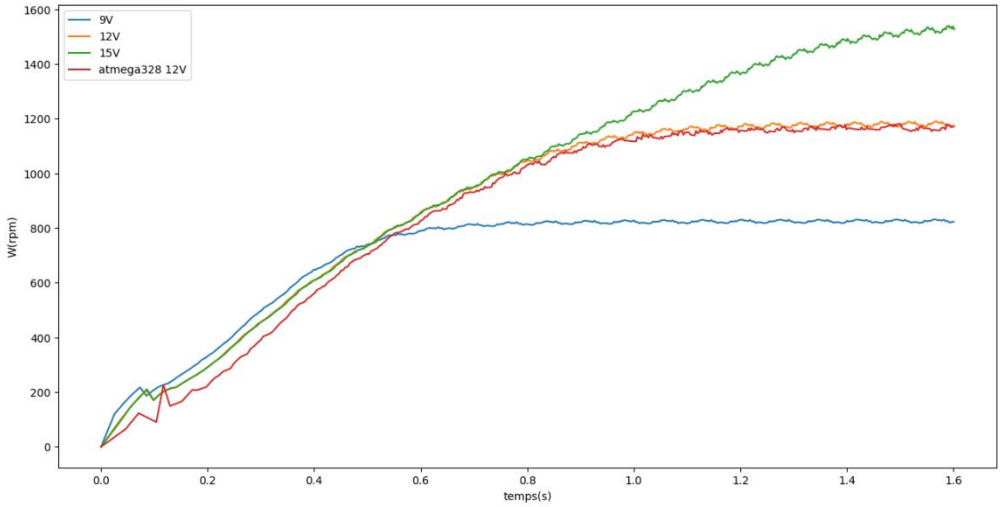



**Figura 3.8** Resposta temporal amb filtre per ATMEGA i mitjana.

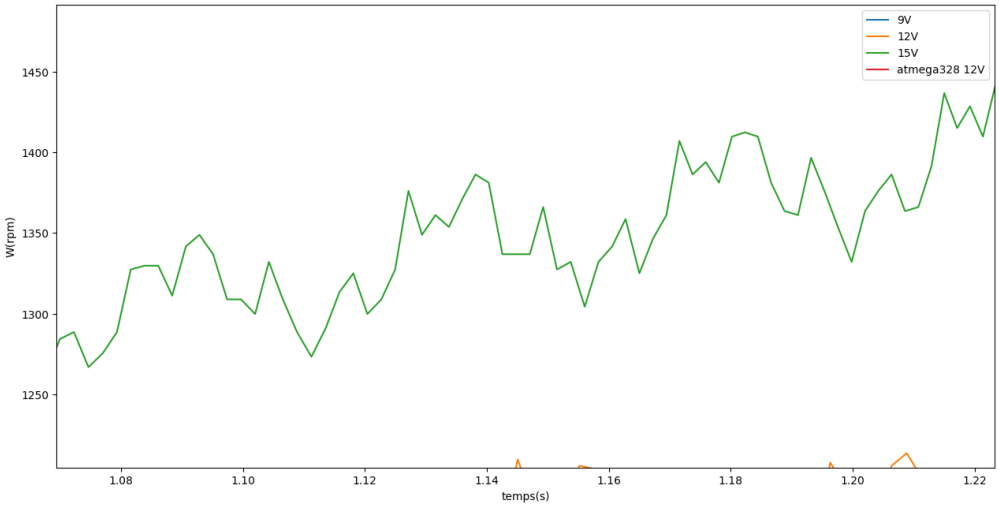



**Figura 3.9** Arrissat d'alta freqüència.

Podem observar a la figura 3.8 com tot i que el guany és lineal, la resposta està separada en una part exponencial amb cert retard i una rampa. Així i tot, per no complicar el model, es continuarà usant el model pol simple. Per aproximar la resposta al model s'utilitzarà el valor de τ de la resposta amb voltatge nominal, és a dir, el de la resposta de 1200 rpm i un valor de guany 100, ja que aproximadament a 12V tenim 1200 rpm.

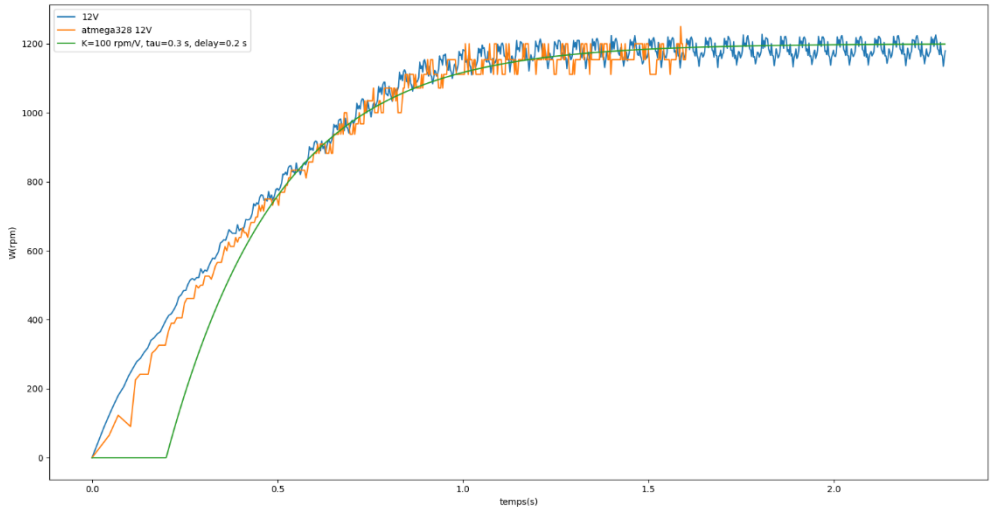



**Figura 3.10** Resposta temporal del motor amb el model.

Com es pot observar la resposta del model a aquesta tensió s'aproxima a la del motor, per tant s'usarà el següent model com a motor.

$$\frac{\omega(s)}{V(s)} = \frac{100}{1+0.3s} e^{-0.2s}$$

**Equació 3.8** Model del motor.

# 4 Disseny del controlador

El disseny del controlador es farà mitjançant l'eina Simulink i posteriorment System generator, una eina que permet modelar a Matlab codi en VHDL.

## 4.1 Creació de la planta

Per modelitzar la planta s'usarà el bloc de funció de transferència, multiplicat pel guany d'aquesta i finalment un time delay.

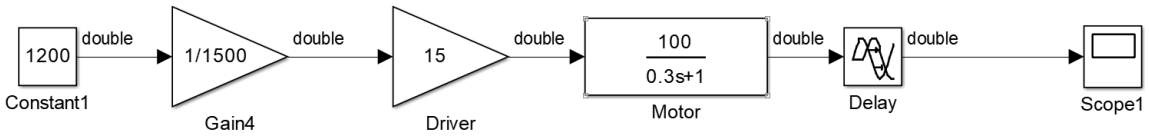



**Figura 4.1** Model del motor.

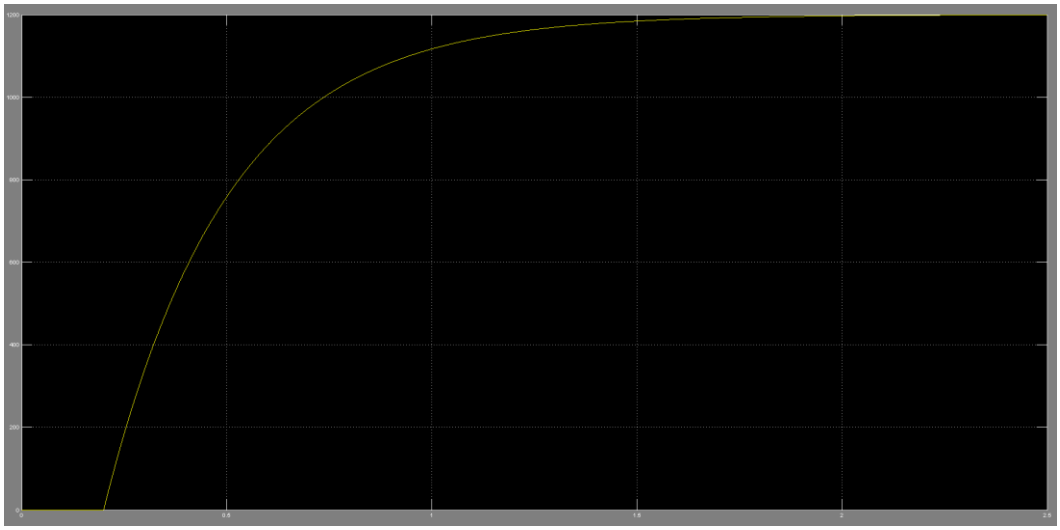


**Figura 4.2** Resposta del model a l'esglaó.

La resposta obtinguda és idèntica a la del model de la figura 3.10. Validant així el model del Matlab.

## 4.2 Disseny del PID analògic.

Primerament es dissenyarà un model ideal del nostre llaç de control. Aquest constarà d'un sensor, per obtenir la velocitat; un controlador, el qual gestionarà el voltatge aplicat al motor en funció de la consigna i el valor del sensor; i finalment un PWM, el qual amb l'ajuda del driver aplicarà el voltatge mitjà assignat pel controlador al motor.

Els paràmetres del PID seran generats mitjançant l'eina PID tuner, aquesta eina linealitzarà la planta i permet escollir la resposta transitòria que l'usuari prefereixi.

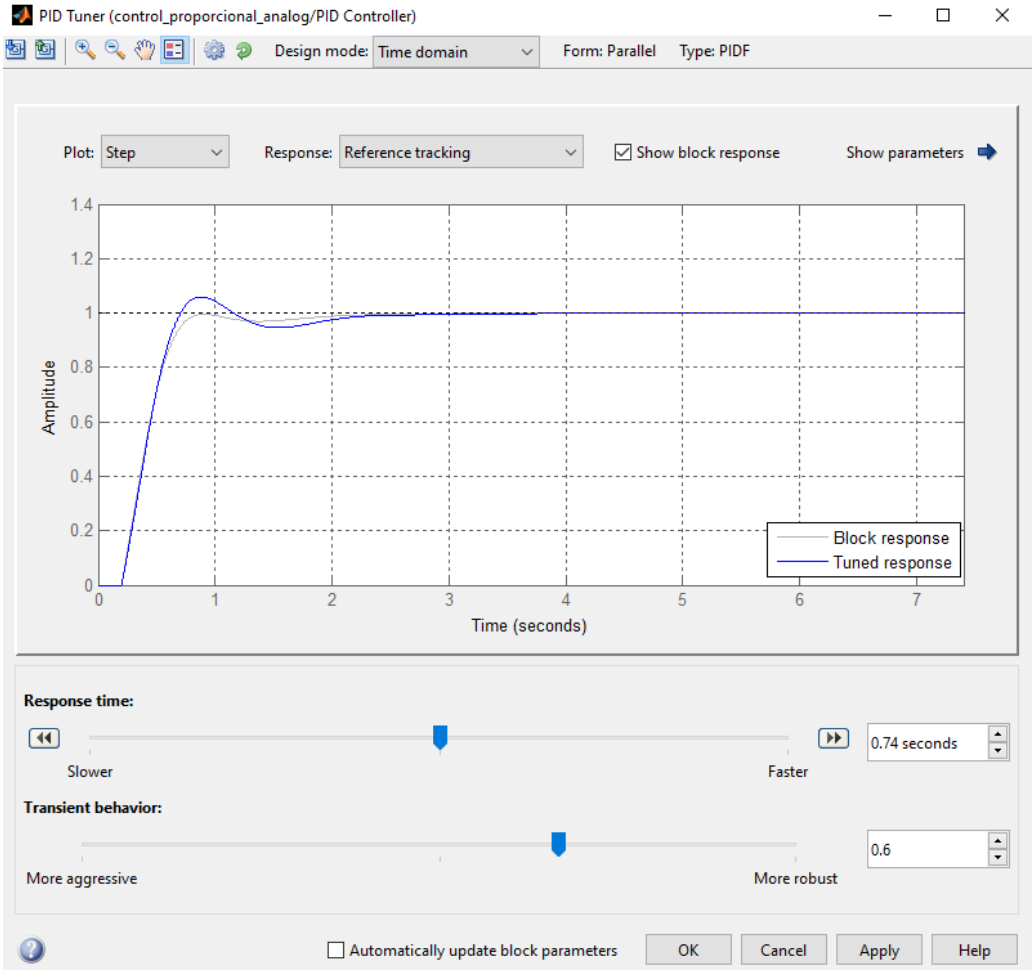



**Figura 4.3** Eina PID tuner Matlab.

Un cop s'ha escollit una resposta transitòria adient es guarden els canvis i s'obtenen els paràmetres del PID.

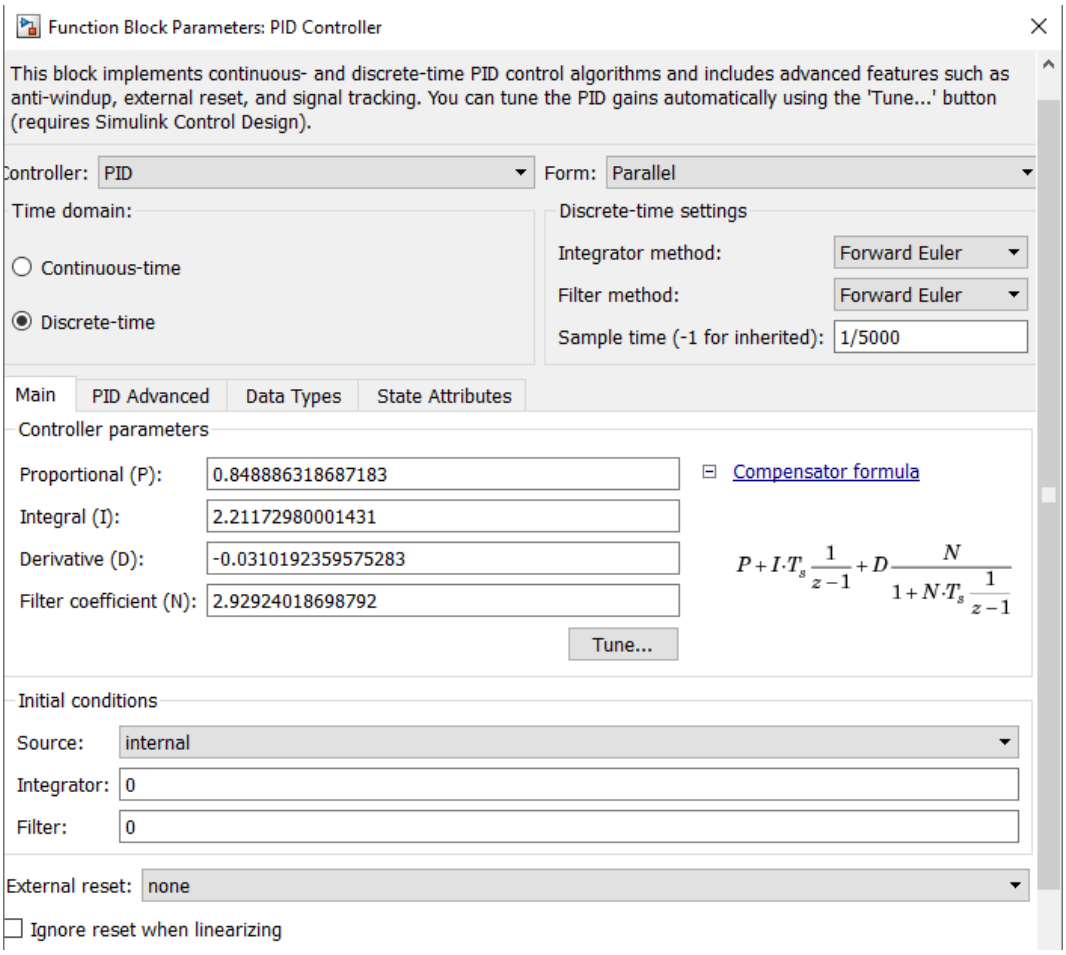



**Figura** 4.4 Eina PID Matlab.

Amb els paràmetres del PID es comprova la resposta temporal.

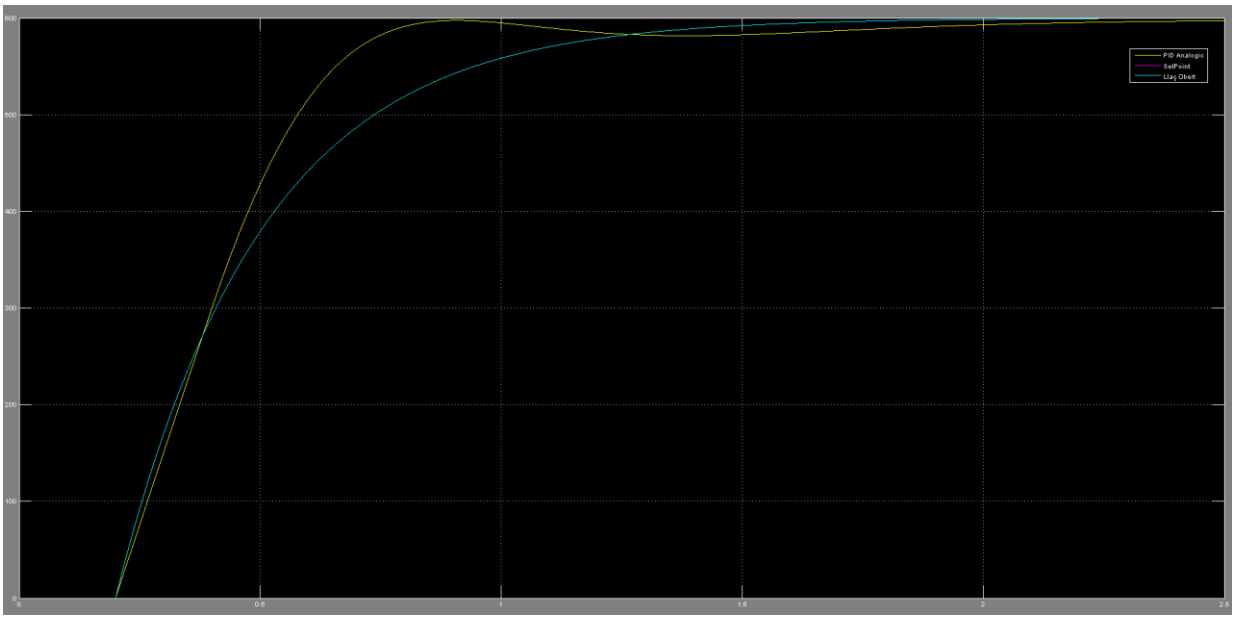


**Figura 4.5** Resposta del PID a un esglaó de 600.

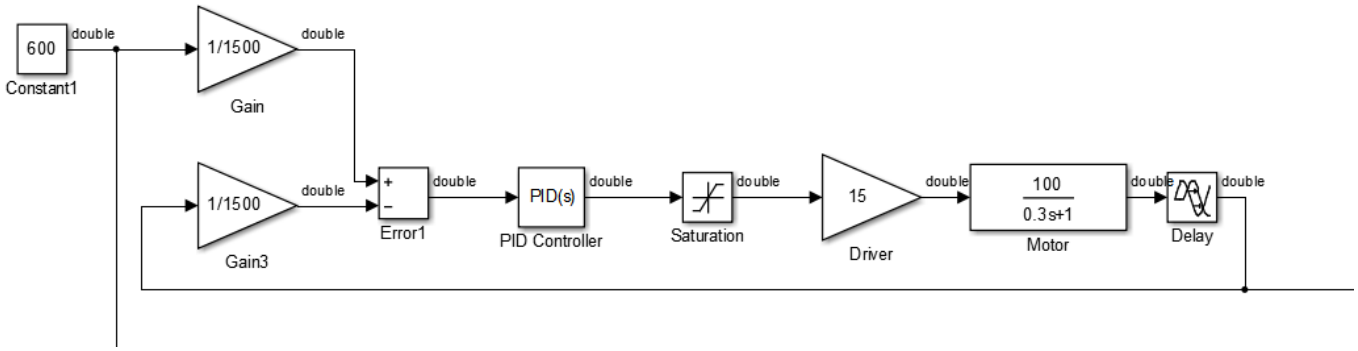



**Figura 4.6** Model PID analògic.

Es pot comprovar com no només tenim millor resposta temporal amb el PID que amb llaç obert, sinó que a més a més en cas que el voltatge del driver canviï (passi de 15 V a 12 V, per exemple) el motor arribarà a girar a la velocitat de la consigna, sempre que aquesta no sigui inferior al voltatge del driver multiplicada per 100.

## 4.3 Disseny del PID digital.

Per dissenyar un PID digital semblant a l'analògic, primer s'ha d'observar la funció de transferència d'aquest.

$$\frac{C(s)}{E(s)} = P + I\frac{1}{s} + D\frac{N}{1 + N\frac{1}{s}} = P + I\frac{1}{s} + D\frac{Ns}{s + N}$$

**Equació 4.1** Funció de transferència del PID.

On E(s) es l'error (consigna – sensor) i C(s) la sortida del PID.

Primer es modelarà la P, on simplement multiplicarem l'error per la constant P. Seguidament l'integrador (el qual consistirà en una memòria a la qual se li suma el valor de l'error a cada iteració i al final és multiplicada per I·dt, on dt és el temps de mostreig), i finalment el derivador. Per modelitzar-lo, la N ha de ser prou gran perquè la funció de transferència convergeixi a una derivada per baixes freqüències i poder modelar el derivador com l'error actual menys l'error en la iteració anterior multiplicat per D/dt.

Per generar e(t) hi ha dues entrades SetPoint i InSensor. Aquestes tenen un marge d'entrada de 0 a 255 (2^8), discrets i amb resolució d'1 (tot valor per sota es tronca a 0 i superior a 255 queda com a 255). Els guanys 1 i 2 s'han modificat tenint en compte aquest marge de valors.

Finalment, el valor de sortida (OutPID) té un rang de 0 a 511 i, un cop normalitzat, va de 0 a 1 amb una resolució d'1/511.

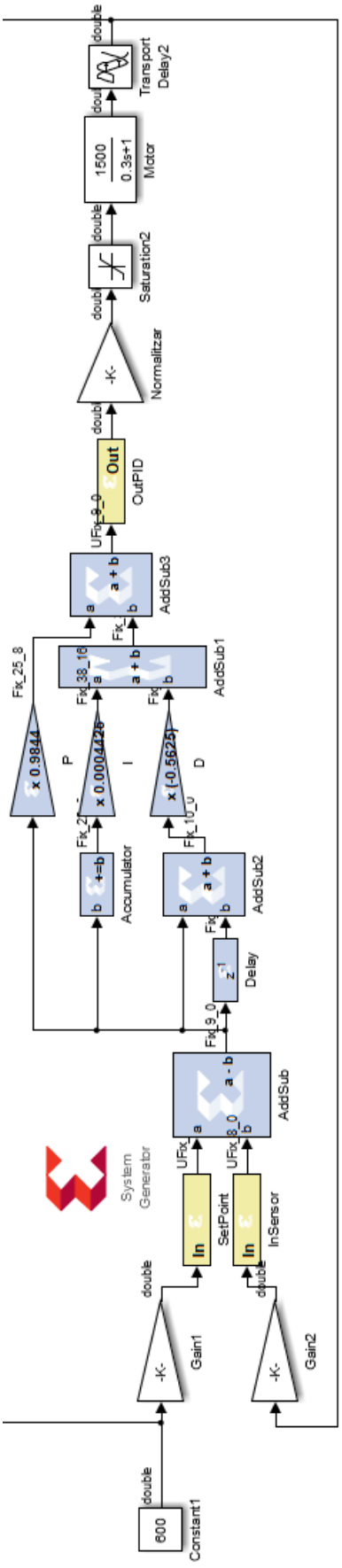



**Figura 4.7** Model de PID digital.

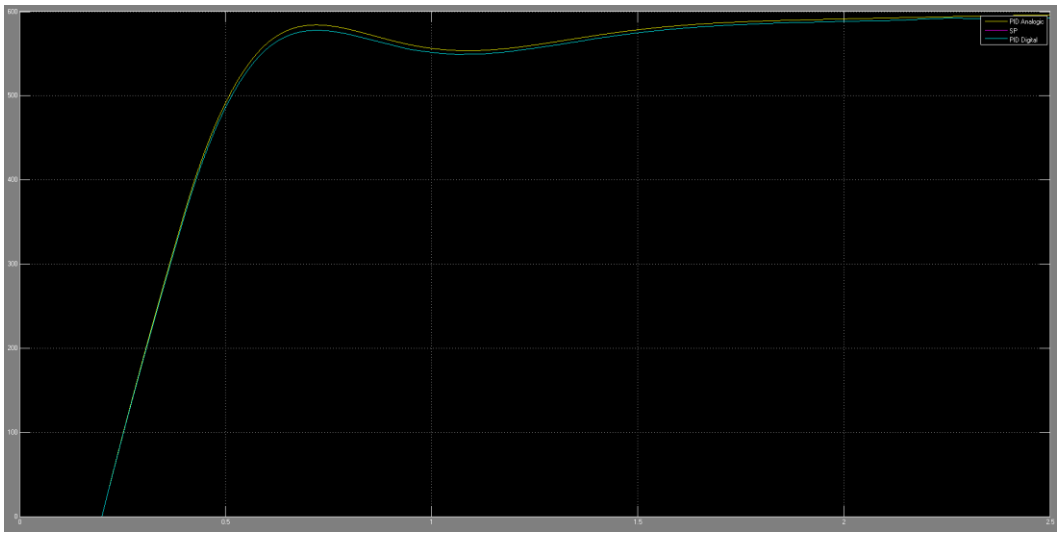



**Figura 4.8** Resposta del PID digital a un esglaó de valor 600.

Es pot observar com les diferències entre el model analògic i digital són ínfimes i es poden atribuir a l'aproximació del derivador i el fet que l'error detectat pel sistema està quantitzat en intervals de 2^(-8).

### 4.3.1 PWM

Per poder modelitzar el PWM i la resta de components s'usarà el bloc Black box, el qual agafa codi VHDL i el transforma en un bloc de Matlab.

El bloc generarà una ona polsada i periòdica amb freqüència $f_{PWM}$. Cada pols estarà en estat On un temps, on $0 \leq T_{on} \leq T_{PWM}$. El valor de $T_{on}$ vindrà donat pel parametre D o duty cylce.

$$T_{on} = D \cdot T_{PWM}$$

El bloc tindrà $N_{PWM} + 1$ bits, els quals representaran el valor del duty cycle. S'ha escollit aquest valor, ja que si disminuïm el nombre de bits no es podran representar tots els valors de duty cycle. Per exemple, amb $N_{PWM}$ només es poden representar els següents valors de duty cycle:

$$D_{Max} = \frac{2^{N_{PWM}} - 1}{2^{N_{PWM}}}$$

Afegint aquest bit addicional el problema desapareix. Es voldrà que el duty cycle (d) segueixi la funció següent:

$$0 \leq d \leq 2^{N_{PWM}+1} - 1, \qquad D(d) = \begin{cases} \dfrac{d}{2^{N_{PWM}}}, & 0 \leq d \leq 2^{N_{PWM}} - 1 \\ 1, & d > 2^{N_{PWM}} - 1 \end{cases}$$

Es desitja trobar un valor de preescaler P de clk que compleixi que la freqüència del PWM sigui de 5000 Hz, ja que les simulacions s'han realitzat a aquesta freqüència. Cada període del PWM està dividit en $2^{N_{PWM}}$ intervals i el bloc s'ha d'actualitzar en cada un d'aquests intervals.

$$f_{PWM} = \frac{f_{clk}}{P \cdot 2^{N_{PWM}}} \leftrightarrow P = \frac{f_{clk}}{f_{PWM} 2^{N_{PWM}}}$$

Finalment, es pot representar el PWM com a una màquina d'estats finits amb dos estats, 0 i 1, els quals representen el valor de la sortida. Es vol que la maquina passi a l'estat 1 quan s'inicia un nou cicle i el comptador intern, cnt, és igual a 0. Per altra banda, un cop cnt=d, es vol passar de 1 a 0. Això serà cert menys excepte el cas especial de d=0, pel qual es desitjarà que la sortida es mantingui a zero, fent així la nova condició per passar de 0 a 1 cnt=0 i d!=0, o de manera equivalent: cnt=0 i cnt!=d, estalviant així un comparador.

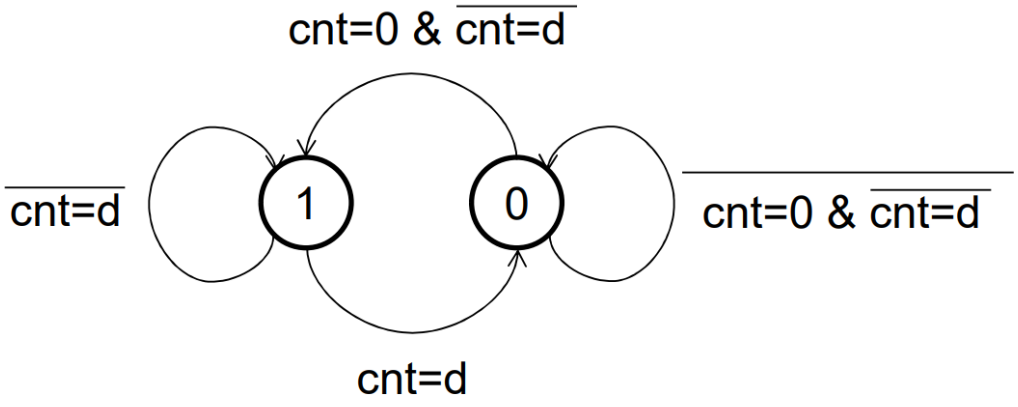



**Figura 4.9** Maquina d'estats del PWM.

Tenint el PWM dissenyat s'ha de modificar part del PID, ja que per cada $2^{N_{PWM}}$ iteracions del PWM el PID només n'ha de fer una. Aquesta reducció d'iteracions es pot aconseguir amb dos blocs: downsample i Clock Enable Probe. Finalment s'afegeixen 4 enables, de tal forma que el bloc només actualitzi la sortida, l'acumulador i el delay del derivador si el Clock enable està activat.

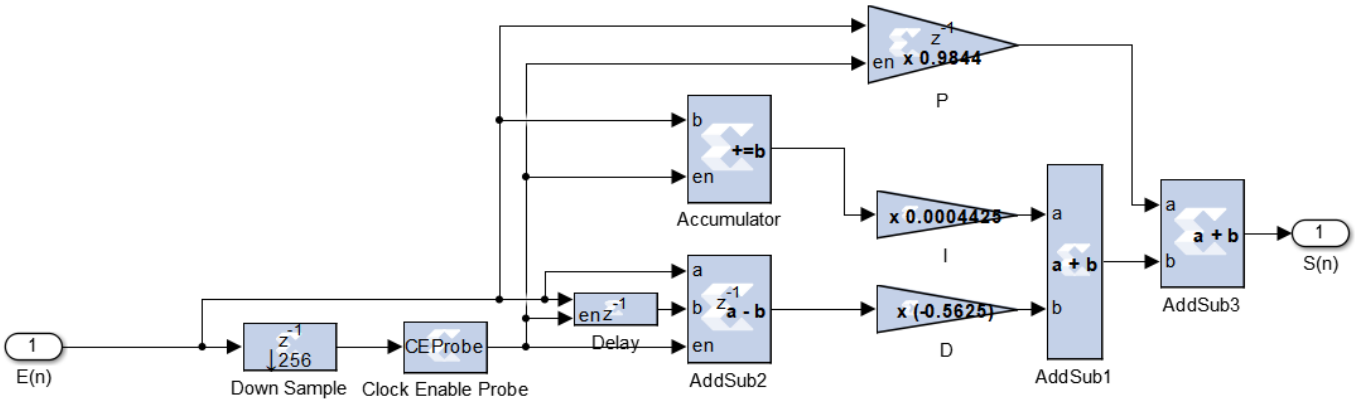



**Figura 4.10** PID modificat amb downsample.

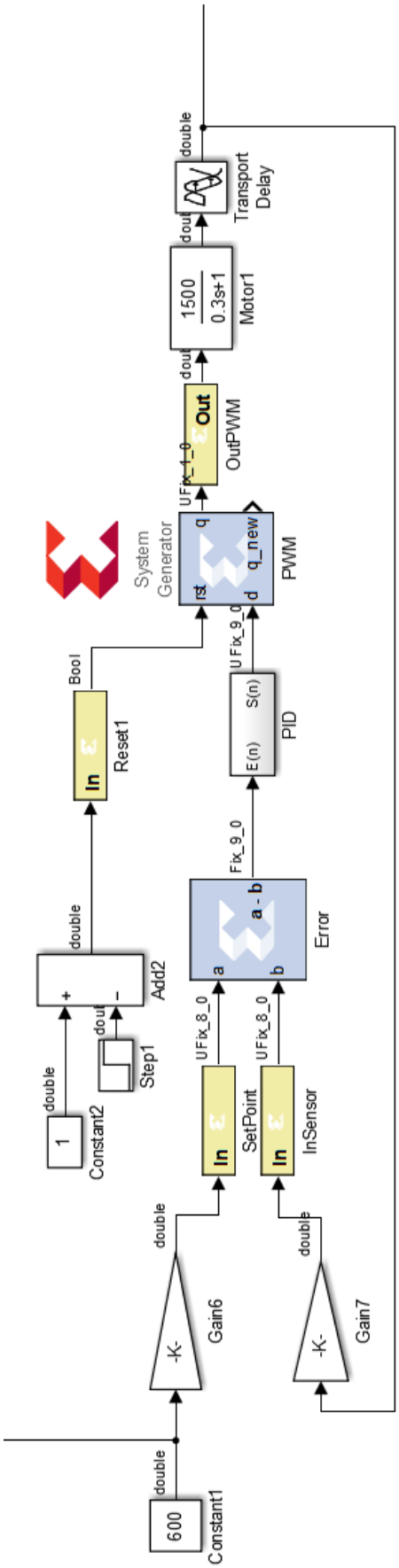



**Figura 4.11** Model del PID digital amb PWM.

Com es pot observar a la figura 4.11, s'ha afegit un reset per inicialitzar el PWM aquest només si està en 1 la primera iteració de la simulació o $1/(5000 \cdot 2^{N_{PWM}})\ s$.

La simulació es realitzarà amb $N_{PWM} = 8$, ja que per resolucions més petites la quantització del duty cycle a la sortida començaria a tenir efectes a la planta.

$$D \in [0, 2^{N_{PWM}}\ ], \overline{V_o} = \frac{1}{T}\int_0^{DT} \frac{V_{Driver}}{2^{N_{PWM}}} dt = \frac{D}{2^{N_{PWM}}} \cdot V_{Driver}$$

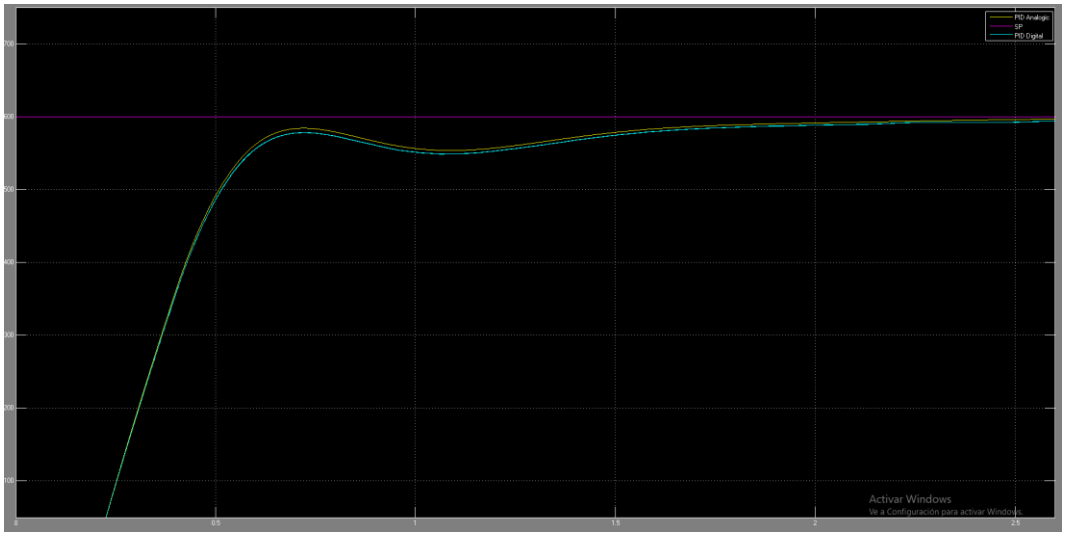



**Figura 4.12** Model del PID digital amb PWM.

Es pot observar com el resultat és igual a l'anterior, i això és gràcies a dos factors: que el nombre de bits usat és prou gran, i que la freqüència del PWM és prou elevada. Es pot calcular la resposta del motor al primer harmònic del PWM.

$$|PWM(s)Motor(s)|_{s=2\pi i\cdot 5000} = 15 \cdot PWM(s)\left.\frac{100}{1+0.3s}\right|_{s=2\pi i\cdot 5000} = |PWM(2\pi i \cdot 5000)| \cdot 0.159$$

L'efecte del primer harmònic (i posteriors és negligible), això és deu a què l'harmònic DC tindrà el valor del duty cycle i la resta seran de valor inferior. L'harmònic DC estarà multiplicat per 1500 (guany de la planta en DC) i la resta d'harmònics com a molt estaran multiplicats per 0.159, aquesta és una diferència de 4 ordres de magnitud.

### 4.3.2 Sensor.

El sensor dona el nombre de polsos de temps Tps de la iteració anterior que han passat entre forat i forat de l'encoder.

Per poder aconseguir aquest bloc primerament s'ha d'aconseguir generar un pols cada cop que hi hagi un canvi ascendent a l'entrada. Aquest pols es genera usant dos registres els actualitzem amb un pols de clk (reescalat prèviament) comparant la sortida d'aquests registres es pot obtenir el pols desitjat.

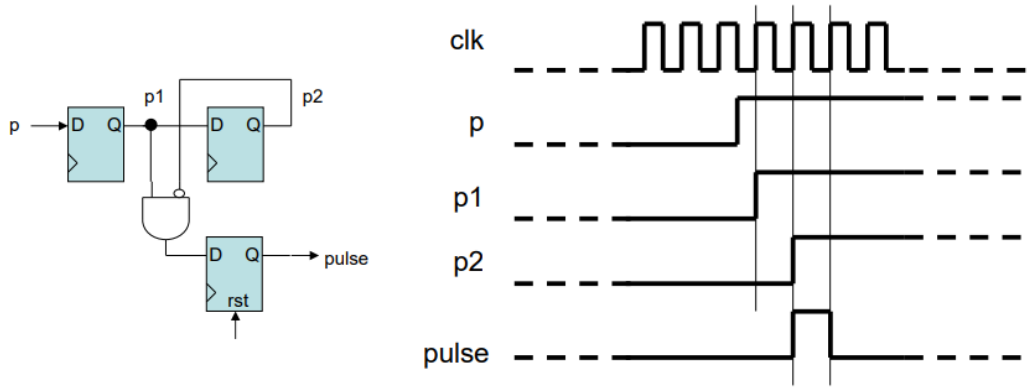



**Figura 4.13** Detector de canvis. [4]

Seguidament es crearà un comptador, el qual anirà sumant de forma periòdica conforme a un clock preescalat. Cada cop que es polsi reset o es detecti un pols, aquest comptador es ficarà a zero i el seu valor serà copiat a la sortida fins al fi de la següent iteració.

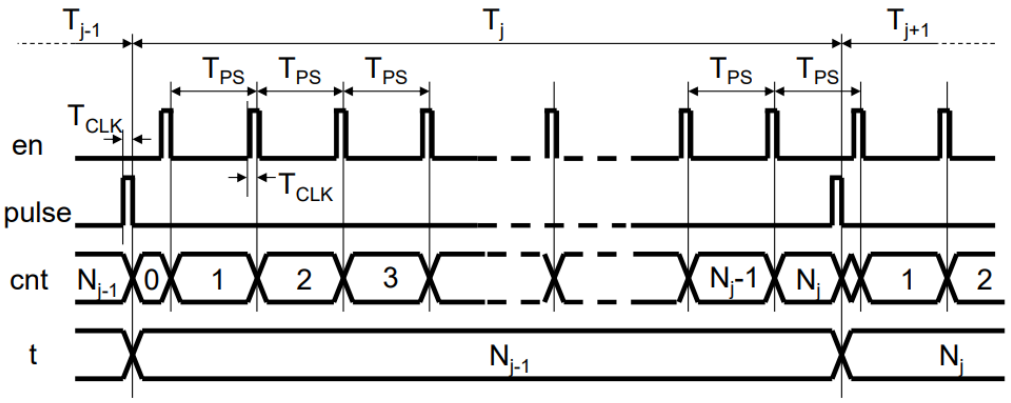



**Figura 4.14** Evolució temporal del sensor.

En cas que passi temps suficient com perquè el comptador vagi a fer overflow, es mantindrà el valor màxim del comptador.

$$cnt(t) = \begin{cases} \left\lceil \frac{t}{T_{ps}} \right\rceil, & t < T_{ps}(2^{N_{sens}} - 1) \\ 2^{N_{sens}} - 1, & t \geq T_{ps}(2^{N_{sens}} - 1) \end{cases}$$

Només queda determinar el temps del preescaler: el valor d'aquest estarà en funció del nombre de bits i de la velocitat angular mínima que es vol obtenir. De l'equació 3.6 s'obté:

$$T_{ps}cnt(t_{max}) = \frac{N_{ps}}{f_{clk}}(2^{N_{sens}} - 1) \geq \frac{60}{W_{min} \cdot l} \rightarrow N_{ps} \geq \frac{60 \cdot f_{clk}}{W_{min} \cdot l \cdot (2^{N_{sens}} - 1)}$$

### 4.3.3 Taula.

Com s'ha vist anteriorment la velocitat del motor (la qual necessita el PID) és inversament proporcional al temps entre polsos. Una manera senzilla d'implementar la relació temps velocitat sense un circuit divisor és mitjançant una ROM.

$$\mathrm{W(t)} = \frac{60 \cdot \mathrm{f_{clk}}}{\mathrm{N_{ps}} \cdot l \cdot \mathrm{t}} = \mathrm{k}\frac{1}{\mathrm{t}}, \mathrm{k} = \frac{60 \cdot \mathrm{f_{clk}}}{\mathrm{N_{ps}} \cdot l}$$

Si es representa la sortida de la taula com R[t] i el numero de bits d'aquesta és Q, $2^Q - 1$ serà el valor més gran que aquesta podrà donar i per tant ha de representar $W_{MAX}$.

$$R[t] = \frac{\mathrm{W(t)}}{W_{MAX}}(2^Q - 1) = \frac{\mathrm{k}}{t \cdot W_{MAX}}(2^Q - 1)$$

Per t màxim (o W mínim) es vol que el valor de la taula sigui 0, i per t mínim es vol que el valor de la taula sigui $2^Q - 1$

$$t_{Sat} = \left\lfloor \frac{k}{W_{MAX}} \right\rfloor$$

$$R[t] = \begin{cases} 2^Q - 1, & 0 < \mathrm{t} < t_{Sat} \\ \mathrm{k}\dfrac{1}{W_{MAX} \cdot \mathrm{t}}(2^Q - 1), & t_{Sat} \leq \mathrm{t} < 2^{N_{sens}} - 1) \\ 0, & t \geq (2^{N_{sens}} - 1) \end{cases}$$

La resolució mínima de la taula pot ser un problema. Aquesta es trobarà entre la màxima distancia entre dos valors consecutius. Aquesta distancia serà:

$$Res_{min,TOP}[bits] = R[t_{Sat}] - R[t_{Sat} + 1] = 2^Q - 1 - \left\lfloor \mathrm{k}\frac{1}{\left\lfloor \frac{k}{W_{MAX}} \right\rfloor + 1}(2^Q - 1) \right\rfloor$$

$$Res_{min,BOT}[bits] = R[2^{N_{sens}} - 2] - R[2^{N_{sens}} - 1] = R[2^{N_{sens}} - 2] - 0 = \left\lfloor \frac{\mathrm{k}}{W_{MAX}} \frac{(2^Q - 1)}{(2^{N_{sens}} - 2)} \right\rfloor$$

$$Res_{min}[bits] = max(Res_{min,TOP}[bits], Res_{min,BOT}[bits]), Res_{min}[rpm] = \frac{W_{MAX}}{(2^Q - 1)} Res_{min}[bits]$$

### 4.3.4 Codificador

El codificador usat al laboratori consisteix en un disc amb forats equidistants, al girar l'eix del motor aquests giren amb ell. Si es col·loca un LED en una cara del disc i un fotodetector en l'altra, s'obté un senyal polsant la qual conte informació sobre la velocitat angular del motor.

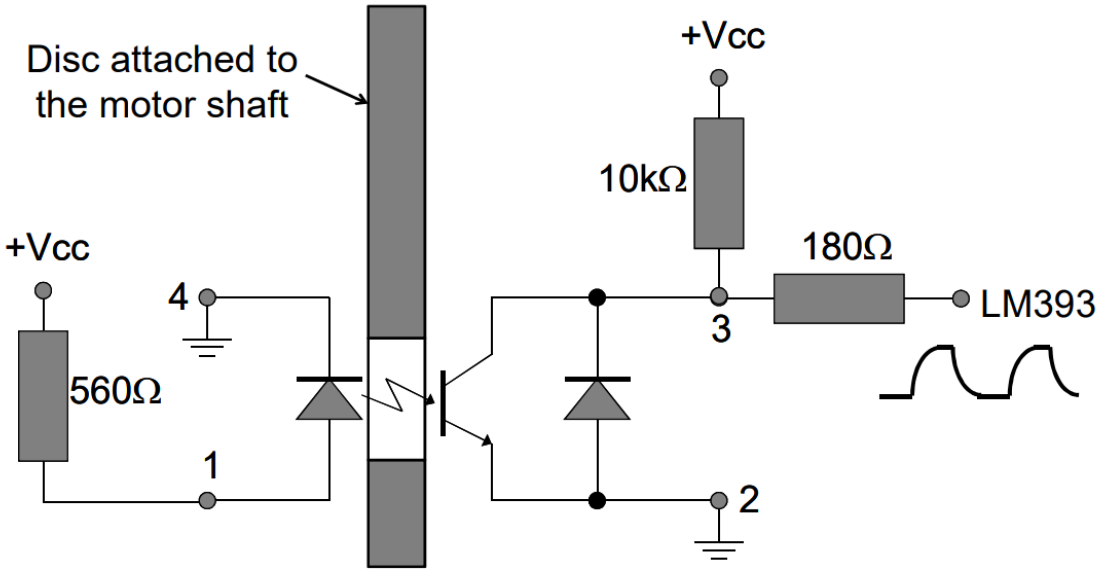



**Figura 4.15** Esquema del codificador. [4]

Primerament es necessita treballar sobre la posició de l'eix, i no sobre la velocitat. Per tant, integrarem la velocitat angular (en rpm). Seguidament es vol fer que aquest angle sigui periòdic, i això es farà mitjançant l'opció mod. El valor pel qual es dividirà és 60/l, ja que la senyal s'ha de repetir 20 vegades per segon. En cas de 60 rpm es volen l polsos per segon, i també s'ha de passar de minuts a segons. Per tant, la senyal s'ha de dividir entre 60. Finalment, la senyal periòdica s'ha de comparar per banalitzar-la, donat que la informació està a la freqüència i no en el temps que la senyal està oberta o tancada, es pot escollir un valor arbitrari, com 60/l·0.5.

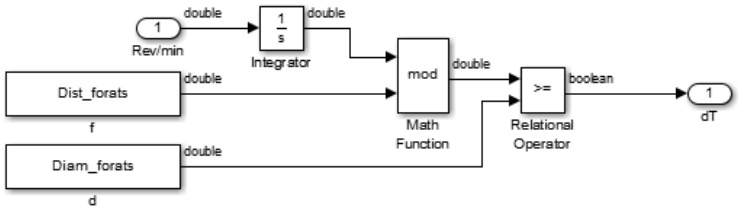



**Figura 4.16** Model del codificador al Simulink.

### 4.3.5 Simulació

Finalment, podem comprovar si els 3 últims elements funcionen correctament.

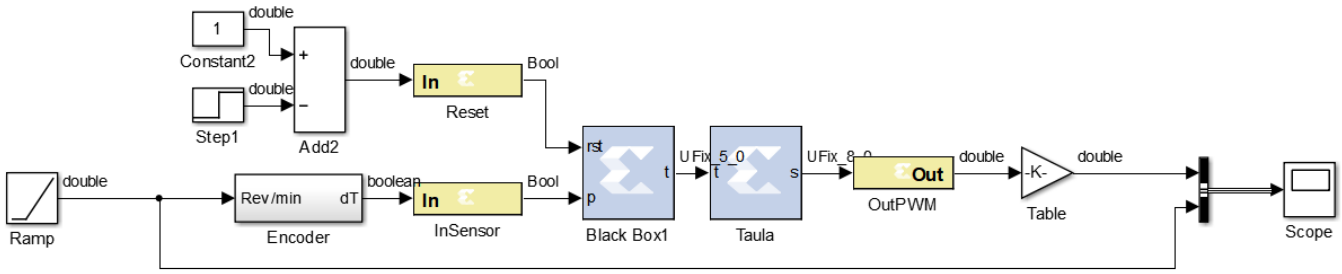



**Figura 4.17** Test de la taula sensor i codificador al Simulink.

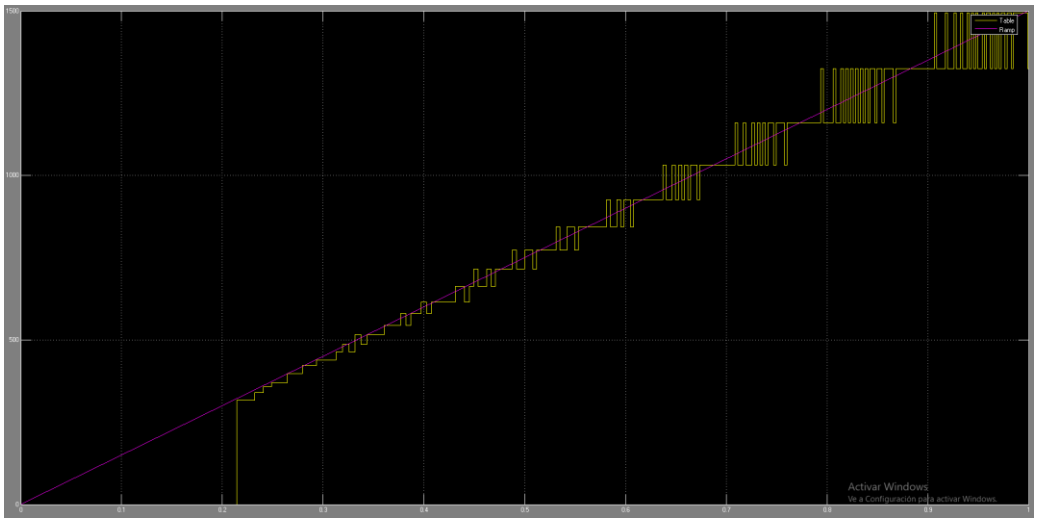


**Figura 4.18** Resposta de la taula sensor i codificador al Simulink.

A la figura anterior es pot comprovar la resposta de l'encoder, sensor i taula a una rampa de pendent 1500 durant un segon. Les dades utilitzades per calcular els paràmetres d'aquests han estat: L=20, Wmin=300 rpm, Wmax=1500 rpm, fclk =5000·2^8 Hz, Nsens = 5, Ntaula=8. La resposta és l'esperada i el conjunt d'elements segueixen la rampa.

Una vegada que s'ha comprovat que els blocs funcionen correctament s'implementen al llaç de control per observar el seu efecte.

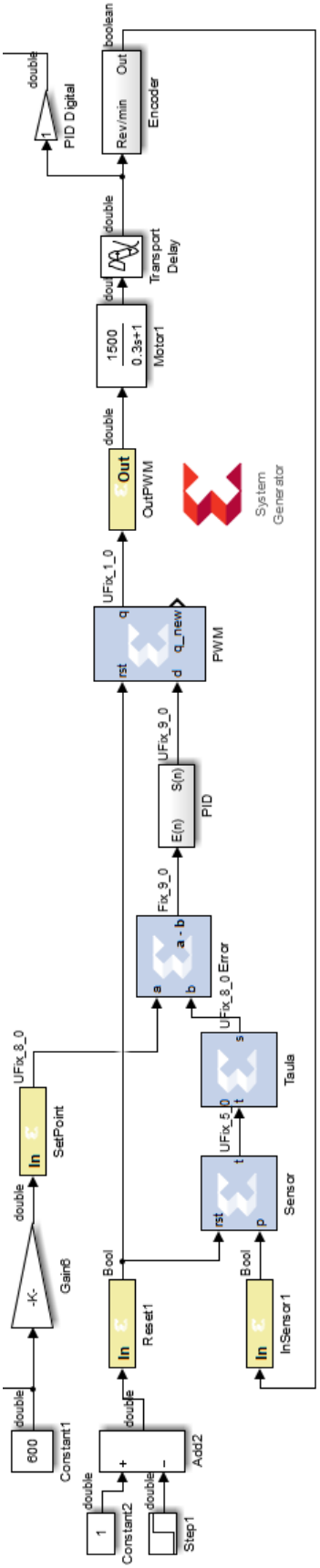



**Figura 4.19** Model final de la planta.

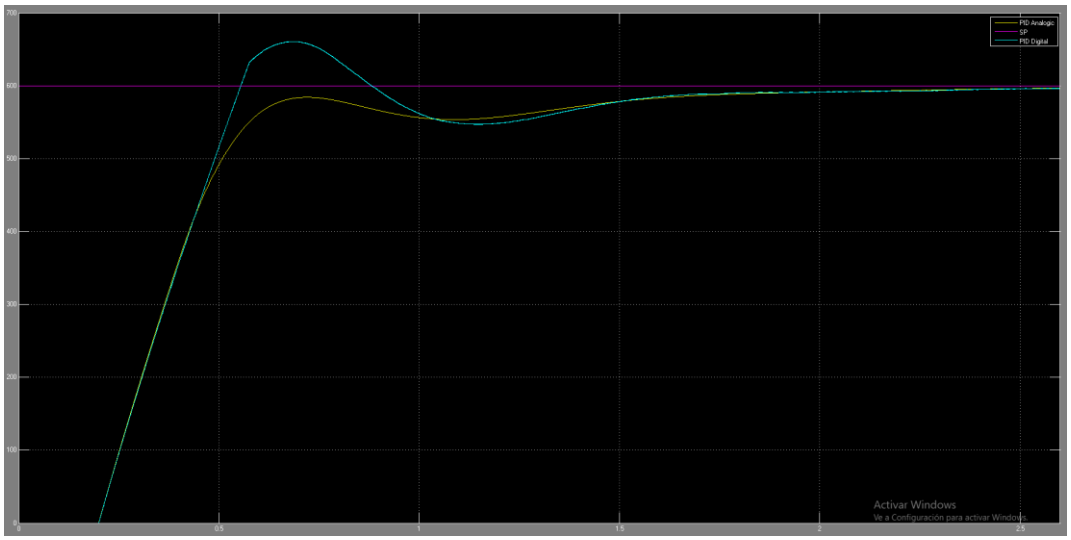



**Figura 4.20** Resposta del model de la planta a un esglaó de valor 600.

Tot i que el PID analògic i digital no són idèntics, això es deu a la resolució del sensor, ja que el motor gira molt lent i aquest es satura i, per tant, el PID "pensa" que no està actuant sobre el motor, segueix proporcionant la mateixa potència que quan aquest estava parat a l'inici. Aquest efecte apareix si es treballa a altes velocitats o per sota de la velocitat mínima Wmin, ja que en aquests punts d'operació la resolució del sensor i la taula és baixa.

# 5 Programació de la FPGA

El programa de la FPGA consistirà en la taula, sensor i PWM. Aquests ja s'han programat prèviament per fer les simulacions en Matlab. També s'implementaran dues funcionalitats, la primera és usar el display 7 segments per mostrar tant la consigna com el duty cylce actual, la segona és poder canviar la consigna mitjançant la botonera de la placa.

Donat que el mòdul top amb control P ja està creat a la pràctica s'utilitzarà el codi de la pràctica i es canviarà el control P pel PID.

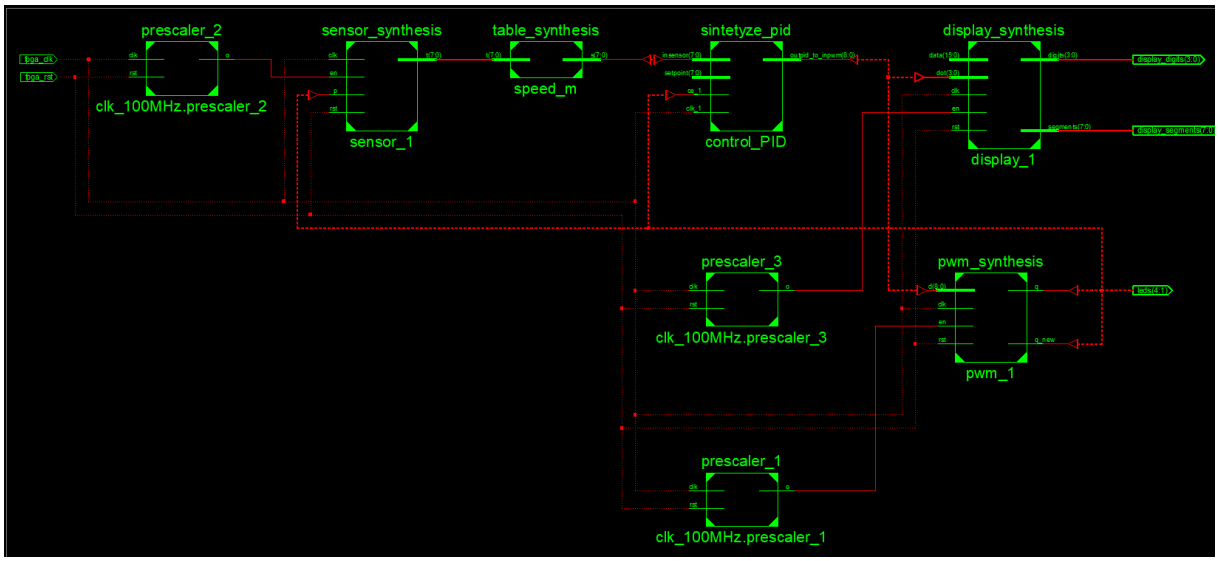



**Figura 5.1** Diagrama de blocs de la programació en FPGA.

## 5.1 PID

Per passar el PID de la simulació del Matlab al PID programat en VHDL per la FPGA s'ha usat System generator. Primerament, s'ha retornat al primer PID (figura 4.7), és a dir el que no està preescalat, i s'ha compilat mitjançant system generator a VHDL.

Un cop és té el PID compilat en VHDL s'importa al projecte i s'ha d'afegir la direcció de les macros generades pel system generator.

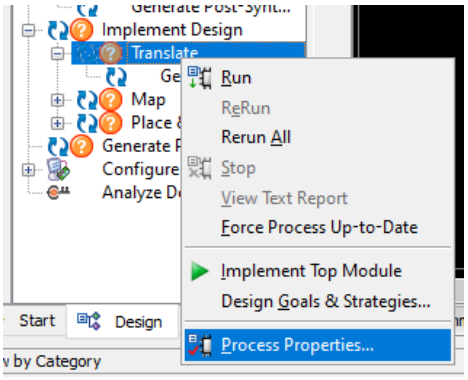



**Figura 5.2** Configuració macros system generator.

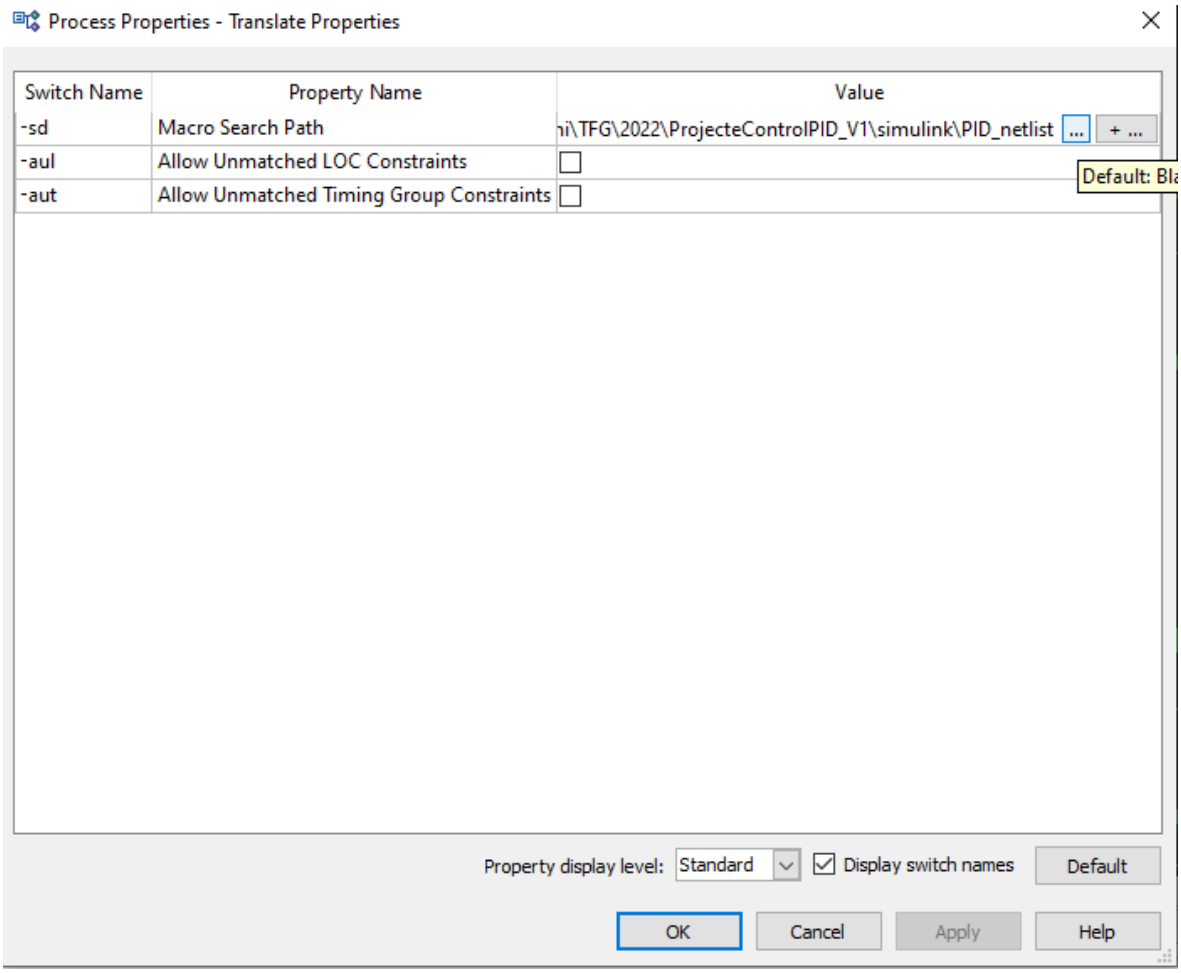



**Figura 5.3** Configuració macros system generator.

# 6 Resultats experimentals.

Donat que a l'hora d'obtenir les dades no es disposava del codi anterior de l'atmega ni de l'arduino s'ha realitzat un nou codi similar al de l'arduino però aquest cop a l'atmega328p. Primerament, s'espera a detectar un pols del PWM i es comença a enviar, s'envia el temps en mili-segons cada cop que es detecta un flanc ascendent del codificador. Finalment, les dades es llegeixen per serial i es guarden i processen igual a les de l'apartat 3.3, obtenint les dades següents.

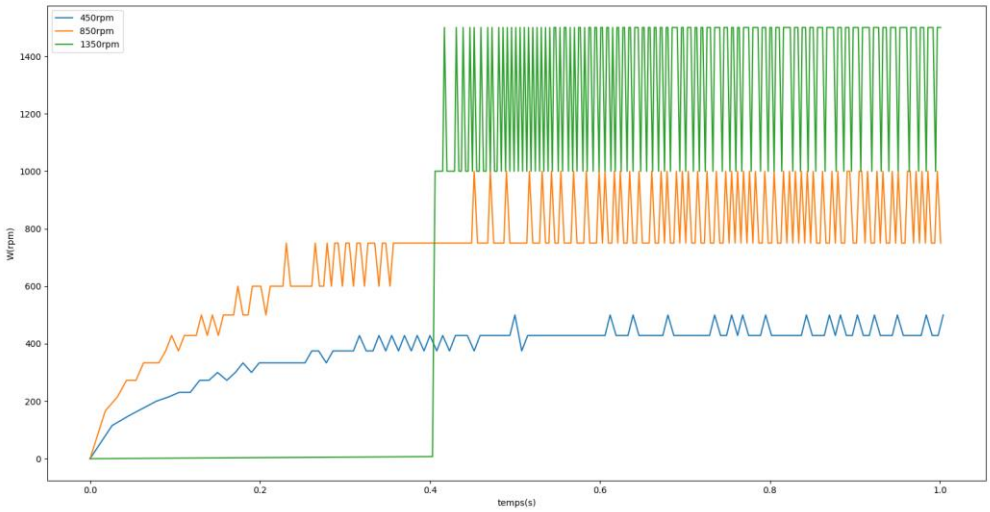



**Figura 6.1** Resposta transitòria temporal amb diferents consignes.

Els pics són deguts a la poca resolució del timer. Després de fer les mesures s'ha observat com la diferencia entre el temps T[n] i T[n+1] per velocitats elevades com 1350 rpm oscil·len entre 2 i 3 ms, clar indicatiu de poca resolució. Així i tot, es pot observar com el controlador fa arribar la velocitat del motor a la consigna. Aquest resultat es pot veure més clarament si s'aplica la mitja dels últims k elements a la gràfica anterior fent que l'element n sigui igual a la suma dels elements n, n-1, n-2, ..., n-k+1, n-k, i dividit entre k. En aquest cas s'usarà k=2.

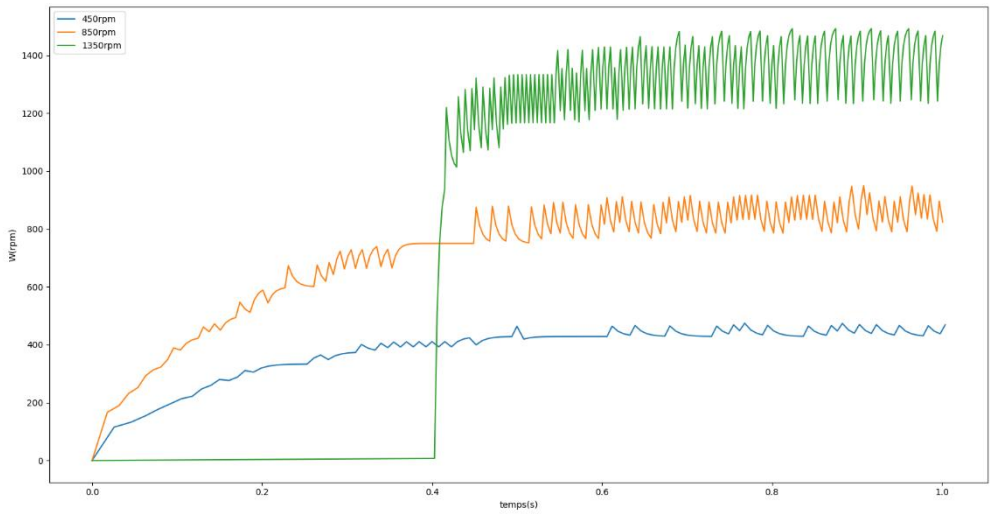



**Figura 6.2** Resposta transitòria temporal filtrada amb diferents consignes.

És fàcil observar a la figura 6.2 com la resposta de controlador a les diferències consignes en estat estacionari és correcta i produeix un error nul, gràcies al PI. Així i tot, la resposta transitòria no ofereix el sobre-pic de la figura 4.20. Aquest efecte probablement és donat perquè el retard temporal real sigui molt més petit que l'usat en el model. Tampoc es poden observar els efectes del PWM, això és per la baixa resolució. Tanmateix, aquests efectes serien molt difícils d'apreciar donades les clares vibracions de l'eix.

# 7 Conclusions

Considero que el treball ha estat un èxit per les següents raons:

1. Tot i que com s'ha vist el model usat tenia massa retard, el controlador aconsegueix un error en estat estacionari quasi nul (amb una mica d'oscil·lació a causa de la taula de la FPGA), i amb un transitori el doble de ràpid que amb llaç obert.
2. S'han pogut observar clarament els efectes dels diversos elements del llaç a les mitjançant simulacions.
3. El codificador junt amb el bloc del sensor i la taula, tenen una gran importància en el llaç, i la resolució d'aquests elements afectarà tant a l'error en estat estacionari del PID com a la precisió d'aquest.
4. Per altra banda, el PWM quasi no té efecte en el llaç, ja que la freqüència de commutació és molt elevada. La planta actua com a filtre passa baixos, fent que els harmònics (menys el DC) del PWM siguin menyspreables. En cas que la freqüència del PWM fos més baixa, es podrien observar efectes no desitjats, com oscil·lacions o arrissat a la planta.
5. Pel que fa al temps simulació, aquest augmenta depenent dels elements que s'afegeixen. Si només s'afegeix el PID no es nota un augment significatiu en el temps d'espera, però en cas que també s'afegeixi el PWM el temps de la simulació augmenta de forma significativa. La causa d'aquest augment és que no només s'actualitza el PID 5000 vegades per segon, sinó que cada cop que aquest actualitza el PWM ha de fer tot el seu cicle que necessita $2^{N_{PWM}}$ actualitzacions.
6. Si s'hagués d'automatitzar el procés de modelitzar el motor, la forma més senzilla seria amb python i arduino, donat que les dos plataformes tenen gran quantitat de llibreries i facilitarien el procés. Així i tot s'ha usat microchip Studio, ja que en algunes ocasions no es disposava d'arduino.
7. Tant system generator com l'ISE de Xilinx han estat eines de gran utilitat donant moltes possibilitats per debugar. Poder testejar el codi simulant-lo en el llaç de control és molt útil per trobar comportaments inesperats, que en cas de no ser simulats poden portar problemes posteriors.

Per finalitzar m'agradaria incloure possibles millores del treball:

1. Dissenyar un llaç doble, el primer en corrent i el segon en tensió per obtenir així una resposta més ràpida i robusta.
2. Crear un softcore a la FPGA i posar el PID com a perifèric. Així poden controlar els valors de les constants del controlador mitjançant el softcore. Tot seguit controlar el softcore mitjançant comandes per la UART (Universal Asynchronous Reciever Transmiter) podent així canviar els paràmetres del controlador de forma senzilla i sense necessitat de recompilar tot el projecte.
3. Estudiar les diferències, en la resposta de la planta, a l'usar floating-point o fixed-point.

# 8 Índex d'il·lustracions

# 9 Índex de taules

# 10 Índex d'equacions

# 11 ANNEX

## 11.1 Referències

[1] Wikipedia. (23 de Març del 2022) *FPGA* https://ca.wikipedia.org/wiki/Matriu_de_portes_programable_in_situ

[2] FPGA en circuit imprès [Fotografia]. https://academy.bit2me.com/wpcontent/uploads/2019/03/fpga2.jpg

[3] Wikipedia. (4 de Març del 2021) *Simulink*. https://ca.wikipedia.org/wiki/Simulink

[4] Contingut provinent de l'assignatura *Embedded Systems and Communications Laboratory (17695112). Campus virtual URV. 2020-2021.* https://campusvirtual.urv.cat/pluginfile.php/3463206/mod_resource/content/6/practicas.pdf

[5] STMicroelectronics. *L298. Dual full-bridge driver.* https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf

[6] Microchip Technology Inc. (2017) *ATmega328PB Xplained Mini.* http://ww1.microchip.com/downloads/en/devicedoc/50002660a.pdf

[7] Microchip Technology Inc. (2017) *ATmega328PB.* http://ww1.microchip.com/downloads/en/DeviceDoc/40001906A.pdf

[8] Arduino Uno Pinout [Fotografia]. https://www.electrogeekshop.com/arduino-uno-pinout-una-sencilla-introduccion-a-su-esquema-5-5-2/

[9] Wikipedia. (12 de Abril del 2022) *MATLAB*. https://ca.wikipedia.org/wiki/MATLAB

[10] AMD XILINX. *Vivaldo 2020.1 – System Generator for DSP.* https://www.xilinx.com/support/documentation-navigation/design-hubs/2020-1/dh0014-vivado-system-generator-hub.html

[11] Sergio Andrés Castaño Giraldo. 2021. *Modelo de Motor DC*. https://controlautomaticoeducacion.com/analisis-de-sistemas/modelo-de-motor-dc/

[12] Xilinx. *Spartan-6 Family Overview.* 2011. https://www.mouser.es/datasheet/2/903/ds160-1591533.pdf

## 11.2 Codis

### 11.2.1 Codi inicialitzacio de la simulació.

```
control_simulacio
encoder
PID
```

#### 11.2.1.1 PID

```
P = 0.985555234140336;
I = 2.23649108678945;
D=-0.000117125390232721;
N=2000.70044786414791;
Ts = 1/5000;
PID_bits = 16;
```

#### 11.2.1.2 Encoder

```
n_forats = 20;
Duty_Forats = 0.9;
Dist_forats = 60/(n_forats); %En rad/n_forats
Diam_forats = Dist_forats*Duty_Forats;
```

#### 11.2.1.3 Control Simulació.

```
clear
N_PWM_bits = 8;
F_PWM = 5000;
T_Mostreig = 1/(F_PWM*(2^N_PWM_bits));
T_FPGA = 10;%100 MHz;
T_sim = 0.4;
D= 1000;
```

### 11.2.2 Codi Matlab Taula

```
function table_sens_config(this_block)

  % Revision History:
  %
  %   10-Jun-2022  (16:43 hours):
  %     Original code was machine generated by Xilinx's System Generator
after parsing
  %     C:\Users\Eloi\OneDrive -
URV\Uni\TFG\Simulacions\2022\projecte_amb_blocs_testejats\Tutorial_0\table.
vhd
  %
  %

  this_block.setTopLevelLanguage('VHDL');

  this_block.setEntityName('table_sens');

  % System Generator has to assume that your entity  has a combinational
feed through;
  %   if it  doesn't, then comment out the following line:
  this_block.tagAsCombinational;
```

```
  this_block.addSimulinkInport('t');

  this_block.addSimulinkOutport('s');


  % ------------------------------
  if (this_block.inputTypesKnown)
    % do input type checking, dynamic output type and generic setup in this
code block.

    % (!) Port 't' appeared to have dynamic type in the HDL -- please add
type checking as appropriate;
  q_port = this_block.port('t');
  q_port.setType('UFix_5_0');
  q_port.useHDLVector(true);
  % (!) Port 's' appeared to have dynamic type in the HDL
    q_port = this_block.port('s');
  q_port.setType('UFix_8_0');
  q_port.useHDLVector(true);
  % --- you must add an appropriate type setting for this port
  end  % if(inputTypesKnown)
  % ------------------------------

  % System Generator found no apparent clock signals in the HDL, assuming
combinational logic.
  % ------------------------------
   if (this_block.inputRatesKnown)
     inputRates = this_block.inputRates;
     uniqueInputRates = unique(inputRates);
     outputRate = uniqueInputRates(1);
     for i = 2:length(uniqueInputRates)
       if (uniqueInputRates(i) ~= Inf)
         outputRate = gcd(outputRate,uniqueInputRates(i));
       end
     end  % for(i)
     for i = 1:this_block.numSimulinkOutports
       this_block.outport(i).setRate(outputRate);
     end  % for(i)
   end  % if(inputRatesKnown)
  % ------------------------------

    % (!) Set the inout port rate to be the same as the first input
    %     rate. Change the following code if this is untrue.
    uniqueInputRates = unique(this_block.getInputRates);

  % (!) Custimize the following generic settings as appropriate. If any
settings depend
  %      on input types, make the settings in the "inputTypesKnown" code
block.
  %      The addGeneric function takes  3 parameters, generic name, type
and constant value.
  %      Supported types are boolean, real, integer and string.
  this_block.addGeneric('NBITS_TIME','integer','5');
  this_block.addGeneric('NBITS_SPEED','integer','8');
  this_block.addGeneric('FACTOR_K','integer','9297');
  this_block.addGeneric('MAX_SPEED','integer','1500');

  % Add addtional source files as needed.
  %  |-------------
  %  | Add files in the order in which they should be compiled.
```

```
  %  | If two files "a.vhd" and "b.vhd" contain the entities
  %  | entity_a and entity_b, and entity_a contains a
  %  | component of type entity_b, the correct sequence of
  %  | addFile() calls would be:
  %  |    this_block.addFile('b.vhd');
  %  |    this_block.addFile('a.vhd');
  %  |-------------

  %    this_block.addFile('');
  %    this_block.addFile('');
  this_block.addFile('C:/Users/Eloi/OneDrive -
URV/Uni/TFG/Simulacions/2022/projecte_amb_blocs_testejats/Tutorial_0/table.
vhd');

return;
```

### 11.2.3 Codi Matlab Sensor

```
function sensor_config(this_block)

  % Revision History:
  %
  %   24-Jul-2021  (20:43 hours):
  %     Original code was machine generated by Xilinx's System Generator
after parsing
  %     C:\Users\Eloi\OneDrive - URV\Uni\Curs2020\Semestre_1\Electronica
Digital\Laboratoris\Lab5\Tutorial_0\sensor.vhd
  %
  %

  this_block.setTopLevelLanguage('VHDL');

  this_block.setEntityName('sensor');

  % System Generator has to assume that your entity  has a combinational
feed through;
  %   if it  doesn't, then comment out the following line:
  this_block.tagAsCombinational;

  this_block.addSimulinkInport('rst');
  this_block.addSimulinkInport('p');

  this_block.addSimulinkOutport('t');


  % ------------------------------
  if (this_block.inputTypesKnown)
    % do input type checking, dynamic output type and generic setup in this
code block.

    if (this_block.port('rst').width ~= 1);
      this_block.setError('Input data type for port "rst" must have
width=1.');
    end

    this_block.port('rst').useHDLVector(false);

    if (this_block.port('p').width ~= 1);
      this_block.setError('Input data type for port "p" must have
width=1.');
```

```
    end

    this_block.port('p').useHDLVector(false);

  % (!) Port 't' appeared to have dynamic type in the HDL
  % --- you must add an appropriate type setting for this port
  q_port = this_block.port('t');
  q_port.setType('UFix_5_0');
  q_port.useHDLVector(true);
  end  % if(inputTypesKnown)
  % ------------------------------

  % ------------------------------
   if (this_block.inputRatesKnown)
     setup_as_single_rate(this_block,'clk_Sens','ce_Sens')
   end  % if(inputRatesKnown)
  % ------------------------------

    % (!) Set the inout port rate to be the same as the first input
    %     rate. Change the following code if this is untrue.
    uniqueInputRates = unique(this_block.getInputRates);

  % (!) Custimize the following generic settings as appropriate. If any
settings depend
  %      on input types, make the settings in the "inputTypesKnown" code
block.
  %      The addGeneric function takes  3 parameters, generic name, type
and constant value.
  %      Supported types are boolean, real, integer and string.
  this_block.addGeneric('NBITS','integer','5');

  % Add addtional source files as needed.
  %  |-------------
  %  | Add files in the order in which they should be compiled.
  %  | If two files "a.vhd" and "b.vhd" contain the entities
  %  | entity_a and entity_b, and entity_a contains a
  %  | component of type entity_b, the correct sequence of
  %  | addFile() calls would be:
  %  |    this_block.addFile('b.vhd');
  %  |    this_block.addFile('a.vhd');
  %  |-------------

  %    this_block.addFile('');
  %    this_block.addFile('');C:\Users\Eloi\OneDrive
  this_block.addFile('C:/Users/Eloi/OneDrive -
URV/Uni/TFG/Simulacions/2022/projecte_amb_blocs_testejats/Tutorial_0/sensor
.vhd');

return;


% ------------------------------------------------------------------

function setup_as_single_rate(block,clkname,cename)
  inputRates = block.inputRates;
  uniqueInputRates = unique(inputRates);
  if (length(uniqueInputRates)==1 & uniqueInputRates(1)==Inf)
    block.addError('The inputs to this block cannot all be constant.');
    return;
  end
  if (uniqueInputRates(end) == Inf)
```

```
      hasConstantInput = true;
      uniqueInputRates = uniqueInputRates(1:end-1);
  end
  if (length(uniqueInputRates) ~= 1)
    block.addError('The inputs to this block must run at a single rate.');
    return;
  end
  theInputRate = uniqueInputRates(1);
  for i = 1:block.numSimulinkOutports
     block.outport(i).setRate(theInputRate);
  end
  block.addClkCEPair(clkname,cename,413);
  return;

% ------------------------------------------------------------
```

### 11.2.4 Codi Matlab PWM

```
function pwm_config(this_block)

  % Revision History:
  %
  %   19-Apr-2022  (15:54 hours):
  %     Original code was machine generated by Xilinx's System Generator
after parsing
  %     C:\Users\Eloi\OneDrive -
URV\Uni\TFG\2022\projecte_amb_blocs_testejats\Tutorial_0\PWM.vhd
  %
  %

  this_block.setTopLevelLanguage('VHDL');

  this_block.setEntityName('pwm');

  % System Generator has to assume that your entity  has a combinational
feed through;
  %   if it  doesn't, then comment out the following line:
  this_block.tagAsCombinational;

  this_block.addSimulinkInport('rst');
  this_block.addSimulinkInport('d');

  this_block.addSimulinkOutport('q');
  this_block.addSimulinkOutport('q_new');

  q_port = this_block.port('q');
  q_port.setType('UFix_1_0');
  q_port.useHDLVector(false);
  q_new_port = this_block.port('q_new');
  q_new_port.setType('UFix_1_0');
  q_new_port.useHDLVector(false);

  % ------------------------------
  if (this_block.inputTypesKnown)
    % do input type checking, dynamic output type and generic setup in this
code block.

    if (this_block.port('rst').width ~= 1);
      this_block.setError('Input data type for port "rst" must have
width=1.');
```

```
    end

    this_block.port('rst').useHDLVector(false);

    % (!) Port 'd' appeared to have dynamic type in the HDL -- please add
type checking as appropriate;

  end  % if(inputTypesKnown)
  % -----------------------------

  % -----------------------------
   if (this_block.inputRatesKnown)
     setup_as_single_rate(this_block,'clk_PWM','ce_PWM')
   end  % if(inputRatesKnown)
  % -----------------------------

    % (!) Set the inout port rate to be the same as the first input
    %     rate. Change the following code if this is untrue.
    uniqueInputRates = unique(this_block.getInputRates);

  % (!) Custimize the following generic settings as appropriate. If any
settings depend
  %      on input types, make the settings in the "inputTypesKnown" code
block.
  %      The addGeneric function takes  3 parameters, generic name, type
and constant value.
  %      Supported types are boolean, real, integer and string.
  this_block.addGeneric('NPWM','integer','8');

  % Add addtional source files as needed.
  %  |-------------
  %  | Add files in the order in which they should be compiled.
  %  | If two files "a.vhd" and "b.vhd" contain the entities
  %  | entity_a and entity_b, and entity_a contains a
  %  | component of type entity_b, the correct sequence of
  %  | addFile() calls would be:
  %  |    this_block.addFile('b.vhd');
  %  |    this_block.addFile('a.vhd');
  %  |-------------

  %    this_block.addFile('');
  %    this_block.addFile('');
  this_block.addFile('C:/Users/Eloi/OneDrive -
URV/Uni/TFG/Simulacions/2022/projecte_amb_blocs_testejats/Tutorial_0/PWM.vh
d');

return;


% -------------------------------------------------------------

function setup_as_single_rate(block,clkname,cename)
  inputRates = block.inputRates;
  uniqueInputRates = unique(inputRates);
  if (length(uniqueInputRates)==1 & uniqueInputRates(1)==Inf)
    block.addError('The inputs to this block cannot all be constant.');
    return;
  end
  if (uniqueInputRates(end) == Inf)
     hasConstantInput = true;
     uniqueInputRates = uniqueInputRates(1:end-1);
```

```
  end
  if (length(uniqueInputRates) ~= 1)
    block.addError('The inputs to this block must run at a single rate.');
    return;
  end
  theInputRate = uniqueInputRates(1);
  for i = 1:block.numSimulinkOutports
     block.outport(i).setRate(theInputRate);
  end
  block.addClkCEPair(clkname,cename,theInputRate);
  return;

% ------------------------------------------------------------
```

### 11.2.5 Codi VHDL PWM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity pwm is
generic(
NPWM: integer:=8 );
port(
clk_PWM: in std_logic;
rst: in std_logic;
ce_PWM: in std_logic;
d: in std_logic_vector(NPWM downto 0);
q: out std_logic;
q_new: out std_logic );
end entity;
architecture a1 of pwm is
signal cnt_0, cnt_d: std_logic;
signal q_int: std_logic:='0';
signal cnt: unsigned(NPWM-1 downto 0);


begin
    q <=q_int;

    cnt_0<='1' when cnt=0 else '0';  --comparator cnt_0='1' when cnt=0
    cnt_d<='1' when cnt=unsigned(d) else '0';  --comparator cnt_d='1' when
cnt=d
counter: process begin
wait until rising_edge(clk_PWM);
    if rst='1' or (cnt+1=0 and ce_PWM = '1') then
        cnt<=(others=>'0');
    elsif ce_PWM = '1' then
        cnt<=cnt+1;
    end if;
end process;
fsm: process begin
wait until rising_edge(clk_PWM);
    if q_int='0' and cnt_0='1' and cnt_d='0' and rst = '0' then
        q_int<='1';
    end if;
    if (q_int='1' and cnt_d='1') or rst = '1' then
        q_int<='0';
    end if;
end process;
q_new<='1' when ce_PWM='1' and cnt+1=0 else '0';
end architecture;
```

### 11.2.6 Codi VHDL Sensor

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity sensor is
generic(
NBITS: integer:=4 );
port(
clk_Sens: in std_logic;
rst: in std_logic;
ce_Sens: in std_logic;
p: in std_logic;
t: out std_logic_vector(NBITS-1 downto 0));
end entity;

architecture a1 of sensor is
signal pulse: std_logic:='0';

begin

detector: process
variable p_reg1,p_reg2: std_logic;
begin
wait until rising_edge(clk_Sens);
    if rst = '0' and p_reg1 = '1' and p_reg2 = '0' then
        pulse <= '1';
    else
        pulse <= '0';
    end if;
        p_reg2 := p_reg1;
        p_reg1 := p;
end process;


counter: process
variable cnt: unsigned(t'range):=(others=>'0');
constant ZERO: unsigned(cnt'range):=(others=>'0');
constant MAX: unsigned(cnt'range):=(others=>'1');
begin
wait until rising_edge(clk_Sens);
    if rst = '1' then
        t <= (others=> '1');
        cnt := ZERO;
    elsif pulse = '1' then
        t <= std_logic_vector(cnt);
        cnt := ZERO;
    elsif not(cnt = MAX) and ce_Sens = '1' then
        cnt := cnt + 1;
    end if;

end process;
end architecture;
```

### 11.2.7 Codi VHDL Taula

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity table_sens is
```

```
generic(
NBITS_TIME: integer:=4; --NSENSOR -P
NBITS_SPEED: integer:=8; --NTABLE -Q
FACTOR_K: integer:=37500; --K
MAX_SPEED: integer:=1000 ); --WMAX
port(
t: in std_logic_vector(NBITS_TIME-1 downto 0);
s: out std_logic_vector(NBITS_SPEED-1 downto 0) );
end entity;
architecture a1 of table_sens is
constant MAX_TIME: integer:=(2**NBITS_TIME)-1;
constant MIN_TIME: integer:=FACTOR_K/MAX_SPEED;
subtype T_TIME is unsigned(NBITS_TIME-1 downto 0);
subtype T_SPEED is unsigned(NBITS_SPEED-1 downto 0);
type T_SPEED_VECTOR is array(0 to (2**NBITS_TIME)-1) of T_SPEED;--ROM
datatype


function F_TIME_TO_SPEED(t: integer) return T_SPEED is
constant SPEED_FS: integer:=(2**NBITS_SPEED);
variable speed_int: integer;
variable speed: T_SPEED;
begin
   if 0 <= t and t <= MIN_TIME then
            speed_int:= SPEED_FS - 1;
    elsif MIN_TIME < t and t < MAX_TIME then
            speed_int:= (SPEED_FS*FACTOR_K)/(MAX_SPEED*t);
    else
            speed_int:= 0;
    end if;
speed:=conv_unsigned(speed_int,NBITS_SPEED);
return speed;
end function;
function F_TABLE return T_SPEED_VECTOR is
variable table: T_SPEED_VECTOR;
begin
for idx in T_SPEED_VECTOR'range loop
table(idx):=F_TIME_TO_SPEED(idx);
end loop;
return table;
end function;
constant SPEED_TABLE: T_SPEED_VECTOR:=F_TABLE; --implements a ROM
signal idx: integer:=0;
signal speed: T_SPEED;
begin
idx<=conv_integer(unsigned(t));
speed<=SPEED_TABLE(idx); --Reads the ROM
s<=std_logic_vector(speed);
end architecture;
```