

Retardos, Macros y Subrutinas

Autor: Luis David Barahona Valdivieso

Fecha: 25/04/2025

índice

1. Macros.....	2
1.1. Planteamiento del problema e identificación de requerimientos.....	2
1.2. Diagrama de bloques y de flujo.....	2
1.3. Código en Ensamblador y C.....	3
1.4. Simulación de los casos en Microchip y Proteus.....	3
1.5. Implementación.....	3
1.6. Documentación en GITHUB.....	3
2. Subrutinas.....	3
2.1. Planteamiento del problema.....	3
2.2. Identificación de requerimientos.....	3
2.3. Diagrama de bloques y de flujo.....	3
2.4. Código en Ensamblador.....	7
2.5. Código en C.....	7
2.6. Simulación de los casos en Microchip y Proteus.....	7
2.7. Implementación.....	8
2.8. Documentación en GITHUB.....	8
3. Anexo.....	9
3.1. Plantilla base de un código en AVR Ensamblador.....	9
3.1.1. Plantilla general de un programa en AVR Ensamblador.....	9
3.1.2. Plantilla de una macro en AVR Ensamblador.....	9
3.1.3. Plantilla de una rutina en AVR Ensamblador.....	10
3.2. Plantilla base de un código en C.....	10
3.2.1. Plantilla general de un programa en C.....	10
3.2.2. Plantilla de una macro en C.....	11
3.2.3. Plantilla de un subrutina en C.....	11

1. Macros

1.1. Planteamiento del problema e identificación de requerimientos

Crear una macro capaz de permitir el desplazamiento aritmético a la derecha. Recordemos que este operador permite realizar divisiones con números que son potencias de 2.

Por ejemplo: Dividir 1024 / 8 es igual a $1024 >> 7$, pues $2^7 = 128$. En ambos casos el resultado debe generar 8.

1.2. Diagrama de bloques y de flujo

1.3. Código en Ensamblador

https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADa%204/ASR_Arithmetic_Shift_Right/ASR_Code_Assembler/AssemblerApplication1/main.asm

1.4. Código en C

C Con macro:

https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADa%204/ASR_Arithmetic_Shift_Right/ASR_Code_C_Macro/ASR_Code_C_Macro/main.c

C con rutina:

https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADa%204/ASR_Arithmetic_Shift_Right/ASR_Code_C_Rutina/ASR_C_Code/main.c

1.5. Simulación de los casos en Microchip y Proteus

SIMULACIÓN USANDO MACRO

CASO 1: Mientras $n \neq 0$ aún no hemos realizado los 7 desplazamientos necesarios

```
ASR_Code_C_Macro main.c
void ASR16(int16_t *value, uint8_t n) {
    // Macro para ASR de 8 bits
    #define ASR8(x) ((x >> 1) | (x & 0x80))

    void ASR16(int16_t *value, uint8_t n) {
        uint8_t *low = ((uint8_t *)value); // Byte bajo
        uint8_t *high = ((uint8_t *)value) + 1; // Byte alto

        n=n-1;
        while (n--) {
            ASR8(*high);
            uint8_t carry = *high & 0x01; // Bit menos significativo del alto
            *low = (*low >> 1) | (carry << 7); // ROR: rotar derecho el byte bajo
        }
    }
}
```

CASO 2: Cuando se $n=0$ (quiere decir que han pasado 7 desplazamientos), low debería ser 0x08, pues la división entre 1024 y 128 es 8.

```
ASR_Code_C_Macro main.c
void ASR16(int16_t *value, uint8_t n) {
    // Macro para ASR de 8 bits
    #define ASR8(x) ((x >> 1) | (x & 0x80))

    void ASR16(int16_t *value, uint8_t n) {
        uint8_t *low = ((uint8_t *)value); // Byte bajo
        uint8_t *high = ((uint8_t *)value) + 1; // Byte alto

        n=n-1;
        while (n--) {
            ASR8(*high);
            uint8_t carry = *high & 0x01; // Bit menos significativo del alto
            *low = (*low >> 1) | (carry << 7); // ROR: rotar derecho el byte bajo
        }
    }
}
```

SIMULACIÓN USANDO RUTINA

CASO 1: Dentro de la rutina, realizando los desplazamientos

The screenshot shows the Microchip Studio interface with the project "ASR_C_Code" open. The code in main.c demonstrates an arithmetic shift right operation:

```

void asr16(int16_t *value, uint8_t n)
{
    //GOAL: Este código realiza un desplazamiento aritmético a la derecha usando rutinas

    #include <avr/io.h>
    #include <stdint.h>

    //RUTINA ASR (arithmetic Shift Right)
    void asr16(int16_t *value, uint8_t n)
    {
        for (uint8_t i = 0; i < n; i++)
        {
            // Realizar desplazamiento aritmético
            // Convertimos el número en una variable temporal
            int8_t temp = *value;

            // Desplazamiento aritmético: replicar el bit de signo
            *value = (temp >> 1) | (temp & 0x0000);
        }
    }

    int main(void)
    {
        // Simulación del comportamiento del código ensamblador
        // 1024/128=8, esto equivale a operar 1024>>3 = 8.
        int16_t num = 0b0000001000000000; // Equivale a R17:R16 = 0x04:0x00
        uint8_t shifts = 3; // Se necesitan 3 desplazamientos porque 2^3=8

        asr16(&num, shifts); // Desplazar 3 veces aritméticamente a la derecha

        while (1)
        {
            // Aquí podrías observar el resultado con un depurador o ejecutarlo a un puente
            PORTD = (uint8_t)(num & 0x00FF); // byte bajo
        }
    }
}

```

The Watch 1 window shows variables and their values:

Name	Type	Value
num	Unknown identifier	Error
shifts	Unknown identifier	Error
*value	int16_t registers@0x00fa	0x0400
n	uint8_t registers@R22	0x07
temp	Unknown identifier	Error

CASO 2: Dividir 1024/128=1024>>7=8

The screenshot shows the Microchip Studio interface with the project "ASR_C_Code" open. The code in main.c demonstrates division by 128:

```

int main(void)
{
    // Simulación del comportamiento del código ensamblador
    // 1024/128=8, esto equivale a operar 1024>>7 = 8.
    int16_t num = 0b0000001000000000; // Equivale a R17:R16 = 0x04:0x00
    uint8_t shifts = 7; // Se necesitan 7 desplazamientos porque 2^7=8

    asr16(&num, shifts); // Desplazar 7 veces aritméticamente a la derecha

    while (1)
    {
        // Aquí podrías observar el resultado con un depurador o ejecutarlo a un puente
        // Por ejemplo, si quisieras ver el resultado en PORTD:
        PORTD = (uint8_t)(num & 0x00FF); // byte bajo
    }
}

```

The Watch 1 window shows variables and their values:

Name	Type	Value
num	int16_t data@0x00fa ([R28]+1)	0x0008
shifts	Unknown location	Error
*value	undefined operand in unary op	Error
n	Unknown identifier	Error
temp	Unknown identifier	Error
i	Unknown identifier	Error

CASO 3: Dividir 1024/1024=1024>>10=1

The screenshot shows the Microchip Studio interface with the project "ASR_C_Code" open. The code in main.c demonstrates division by 1024:

```

//Dividir 1024/1024 = 1. Esto equivale a operar 1024>>10 = 1.
int16_t num = 0b0000001000000000; // Equivale a R17:R16 = 0x04:0x00
uint8_t shifts = 10; // Se necesitan 10 desplazamientos porque 2^10=1024

asr16(&num, shifts); // Desplazar 10 veces aritméticamente a la derecha

while (1)
{
    // Aquí podrías observar el resultado con un depurador o ejecutarlo a un puente
    // Por ejemplo, si quisieras ver el resultado en PORTD:
    PORTD = (uint8_t)(num & 0x00FF); // byte bajo
}

```

The Watch 1 window shows variables and their values:

Name	Type	Value
num	int16_t data@0x00fa ([R28]+1)	0x0001
shifts	Unknown location	Error
*value	undefined operand in unary op	Error
n	Unknown identifier	Error
temp	Unknown identifier	Error
i	Unknown identifier	Error

1.6. Implementación

1.7. Documentación en GITHUB

https://github.com/luisbarahona100/Mentorias/tree/main/Mentor%C3%A3da%204/ASR_Arithmetic_Shift_Right

2. Subrutinas

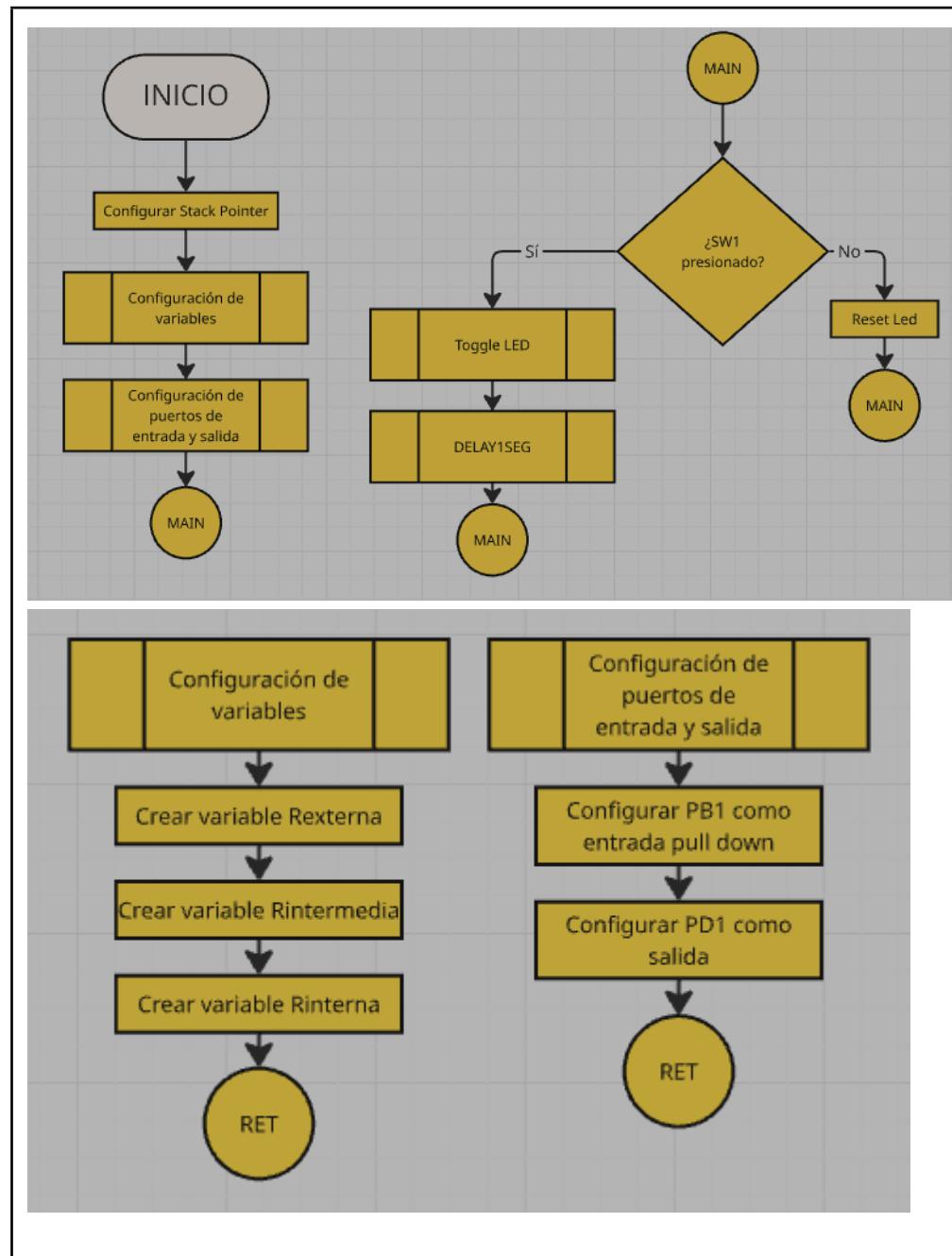
2.1. Planteamiento del problema

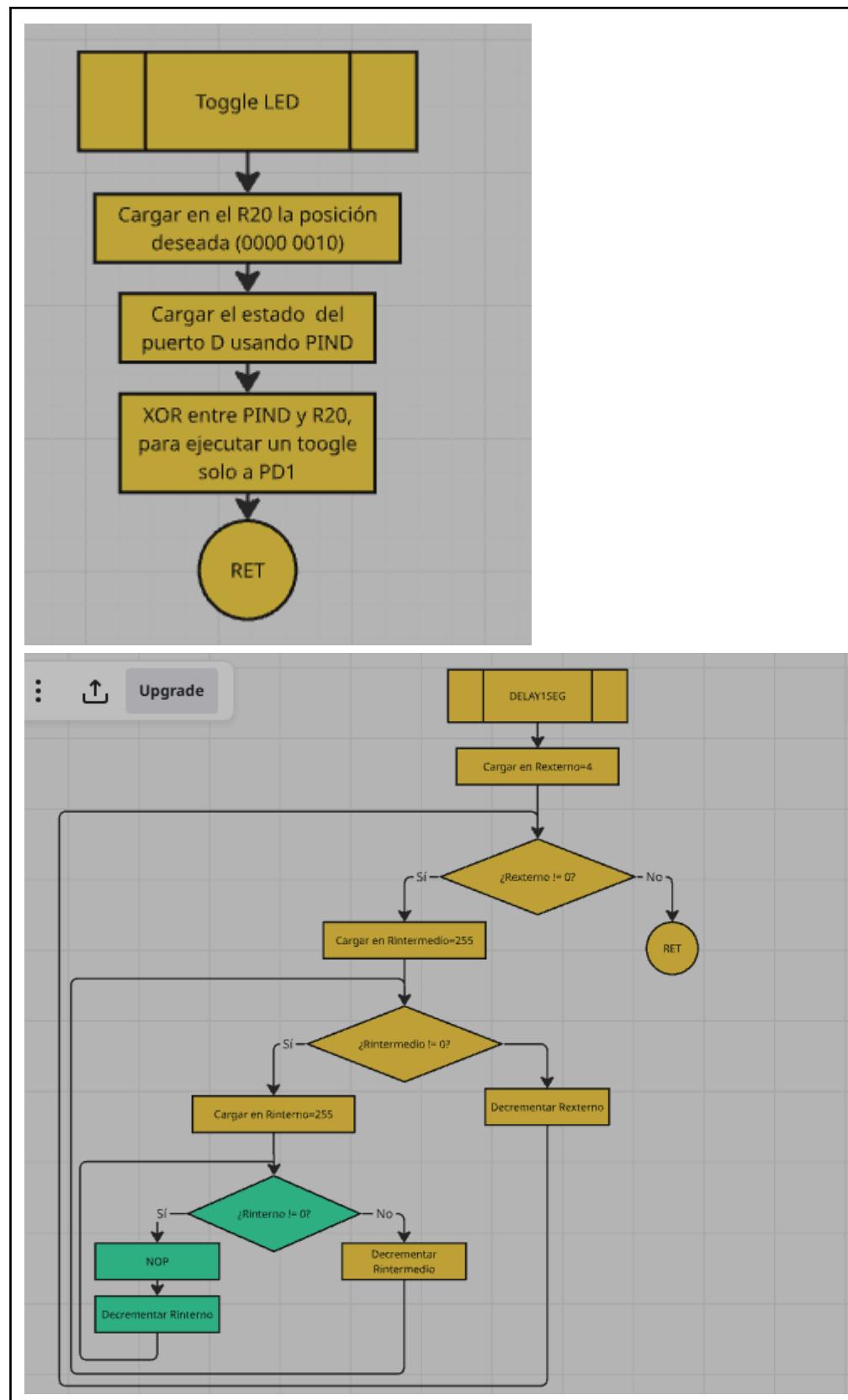
Prender y apagar un led cada segundo

2.2. Identificación de requerimientos

- Entrada : Switch en pull down
- Salida: Led

2.3. Diagrama de bloques y de flujo





M4

viernes, 2 de mayo de 2025 11:40

$R20$	$0000\ 0010$	β	$= P01$	XOR
$R21$	$0000\ 0020$		$= PIND$	
$\underline{R21}$		$0000\ 0010$	$\rightarrow PORTD$	

$R20$	$0000\ 0010$	β	$= P01$	XOR
$R21$	$0000\ 0040$		$= PIND$	
$\underline{R21}$		$0000\ 0040$	$\rightarrow PORTD$	

3 formas de decir "Quiero modificar solo P01"

PORTD: P06 P05 P04 P03 P02 P01 P00

$XOR \approx \text{tugle si } B=1$

A	B	y
0	0	0
0	1	1
1	0	1
1	1	0

$0000\ 0010 = 1$

```
delay2AVR.py
requirements.txt
RetardoAVREnsamblador.pdspj
RetardoAVREnsamblador.pdspj.DESKTOP-6MAFF1OJ...
```

Calculator app showing: $4 \times 250 \times 250 \times 4 \times 0.000001 = 1$

```
61 return None
62
63
64 resultado = calcular_retardo_max_10_nops(1e6, 1.0) # f=1MHz, delaydeseado= 1 segundo
65 print(resultado)
```

Terminal output:

```
mens/barahona_2025_marzo/Mentorías/Mentoria 4/Retardos/delay2AVR.py"
Delay deseado: 1000000.00 us
Delay real: 1000000.00 us
Error: 0.00 us
Ciclos necesarios: 1000000.0
{'Registros': [4, 250, 250], 'NOPS_internos': 4, 'Delay_real': 1.0, 'Error': 0.0}
PS C:\Users\luisdavidbarahona\Documents\barahona_2025_marzo\Mentorías\Mentoria 4\Retardos> []
```

$f_{clk} = 1MHz = 10^6 Hz = 10^6 \text{ ciclos/segundo}$

$T_{clk} = 1/f_{clk} = 1\mu\text{seg} \rightarrow \text{duración del ciclo}$

ciclos consumidos

$$T_{delay} = R_{ext} \cdot R_{inter} \cdot R_{int} \cdot NI \cdot T_{clk}$$

$$= \frac{1}{4} \cdot \frac{1}{255} \cdot \frac{1}{255} \cdot 4 \cdot 1 \cdot 10^{-6}$$

$T_{delay} = 1.0404 \text{ segundos}$

$T_{delay} \text{ max con 3 reg} = 255 \cdot 4 \cdot 1 \cdot 10^{-6} = 66.52 \text{ seg}$

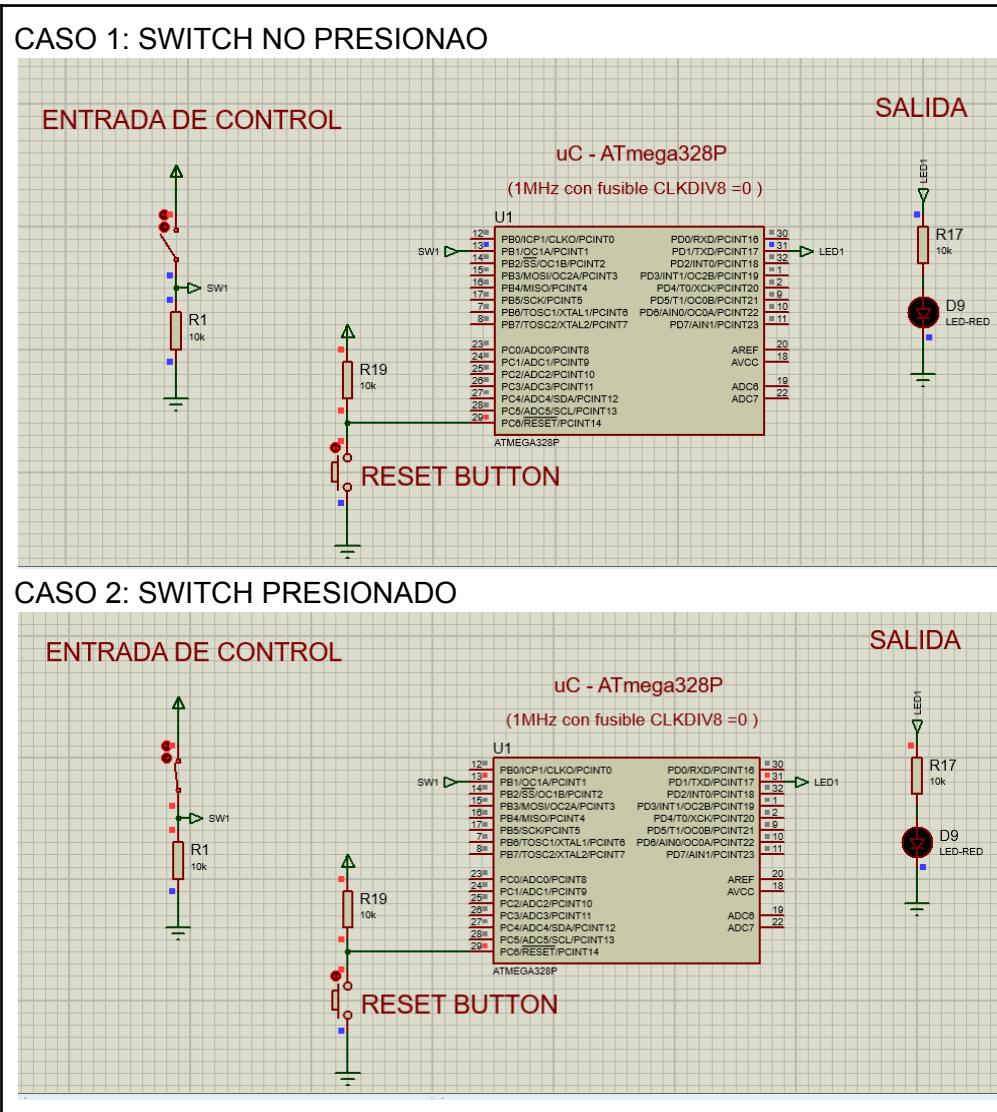
2.4. Código en Ensamblador

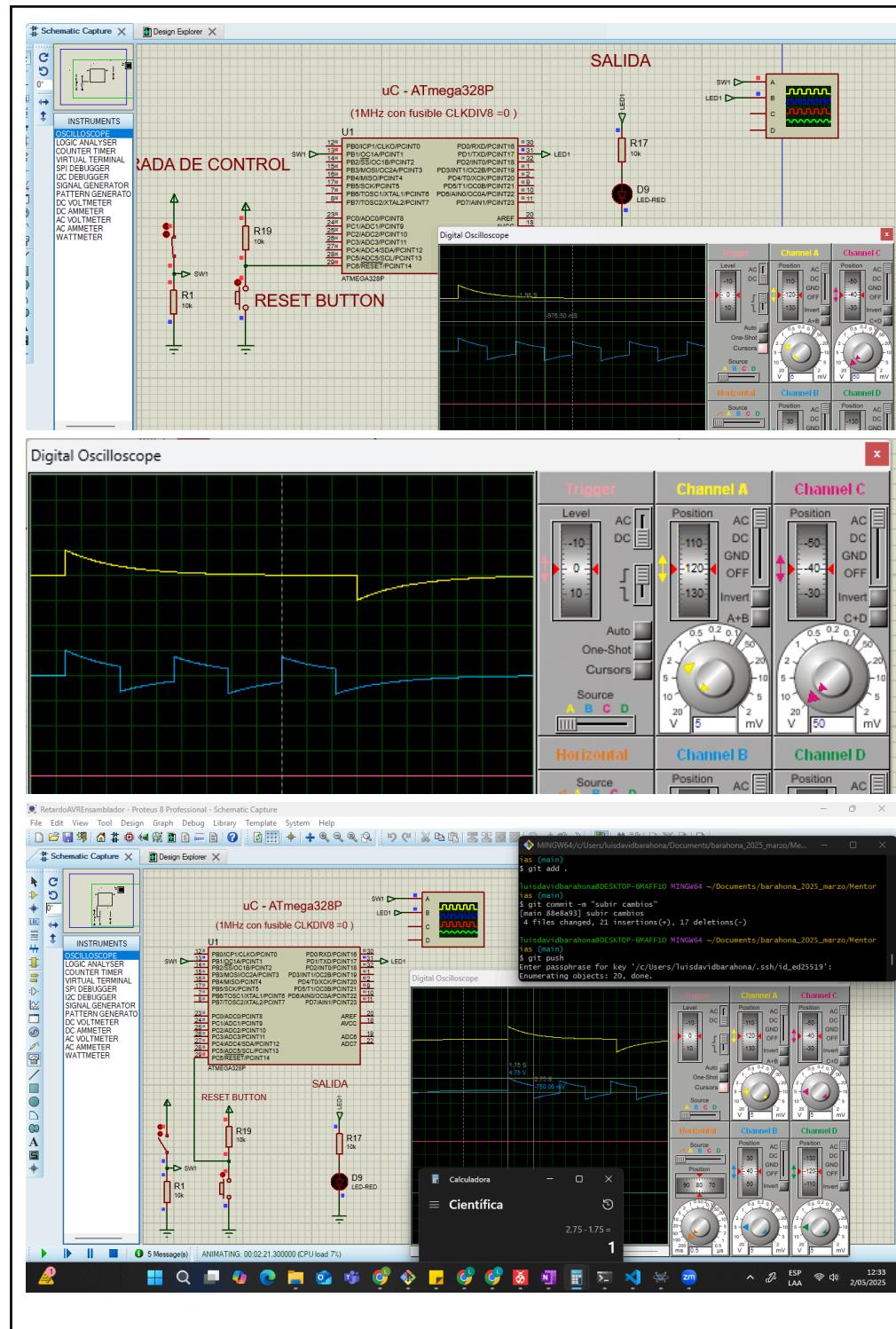
<https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADA%204/Retardos/RetardoAVREnsamblador/RetardoAVREnsamblador/main.asm>

2.5. Código en C

https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADA%204/Retardos/RetardoAVR_CodeC/RetardoAVR_CodeC/main.c

2.6. Simulación de los casos en Microchip y Proteus





2.7. Implementación 2.8. Documentación en GITHUB

CODE AVR ENSAMBLADOR:

<https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADA%204/RetardoAVREnsamblador/RetardoAVREnsamblador/main.asm>

CODE AVR C:

https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADa%204/Retardos/RetardoAVR_CodeC/RetardoAVR_CodeC/main.c

SCRIPT PYTHON PARA CALCULAR EL VALOR DE LOS R:

<https://github.com/luisbarahona100/Mentorias/blob/main/Mentor%C3%ADa%204/Retardos/delay2AVR.py>

3. Anexo

3.1. Plantilla base de un código en AVR Ensamblador

3.1.1. Plantilla general de un programa en AVR Ensamblador

```
None

; --- Directivas iniciales ---
.include "m328Pdef.inc"

; --- Definción de macros ---

; --- Redireccionamiento al posición 0x00 de la memoria de programa
.org 0x00
    rjmp RESET

; --- Variables temporales ---
.def temp = r16

; --- Configuración de puertos e inicialización ---
RESET:

; --- Programa principal ---
MAIN:

; --- Subrutinas ---
OTRAS_ETIQUETAS:
```

3.1.2. Plantilla de una macro en AVR Ensamblador

```
None
;CREACIÓN DE MACRO
.macro toggle_bit
```

```
.endm

;USO DE MACRO
toggle_bit
```

- Es definido antes del MAIN

3.1.3. Plantilla de una rutina en AVR Ensamblador

```
None

;CONFIGURAR STACK POINTER
.include "m328Pdef.inc"
.org 0x00
    rjmp RESET
RESET:
    LDI R19, LOW(RAMEND)
    OUT SPL, R19
    LDI R19, HIGH(RAMEND)
    OUT SPH, R19

; USAR RUTINA
MAIN:
    RCALL RUTINA_SUMAR
    RJMP MAIN

;DEFINIR RUTINA
RUTINA_SUMAR:
    ;Lógica de la suma
RET
```

3.2. Plantilla base de un código en C

3.2.1. Plantilla general de un programa en C

```
C/C++
#include <avr/io.h>
```

```
#include <util/delay.h>
#include <avr/interrupt.h>
#include "DEF_ATMEGA328P.h"
#include <avr/sfr_defs.h>

int main(){

while (1){

}

}
```

- Se recomienda abrir el archivo debuggs/.lss para poder observar el código ensamblador equivalente al code C escrito.

3.2.2. Plantilla de una macro en C

```
C/C++
// Macro para alternar el estado de un bit en un puerto
#define TOGGLE_BIT(port, pin) ((port) ^= (1 << (pin)))

//USO
TOGGLE_BIT(PORTB, PB0); // Alterna PB0
```

3.2.3. Plantilla de un subrutina en C

```
C/C++
int function rutina (param1, param2){
    #LOGICA: SUMAR param1 y param2

}

#uso

int resultado=0;
resultado = rutina(1,2);
```

- En C, una rutina es la función que conocemos de toda la vida.