EE 3420 Lab Guide: Motor Drivers w/ NIOS II

Written by: Grant Seligman, Gabe Garves, and James Starks

SN754410 Quadruple H-Bridge

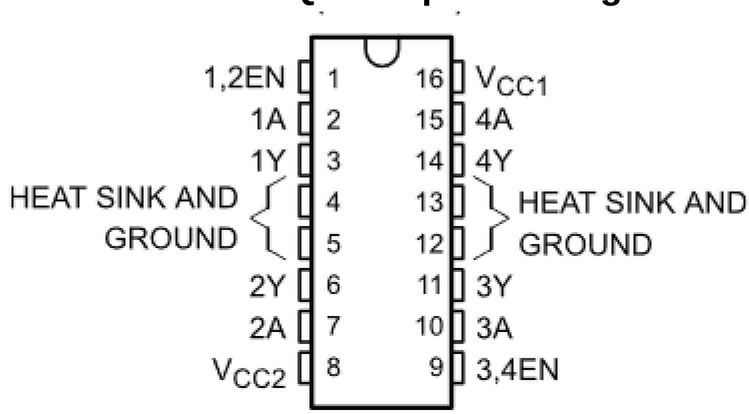


Figure 1 [1]

Part 1: DC Motor Driver

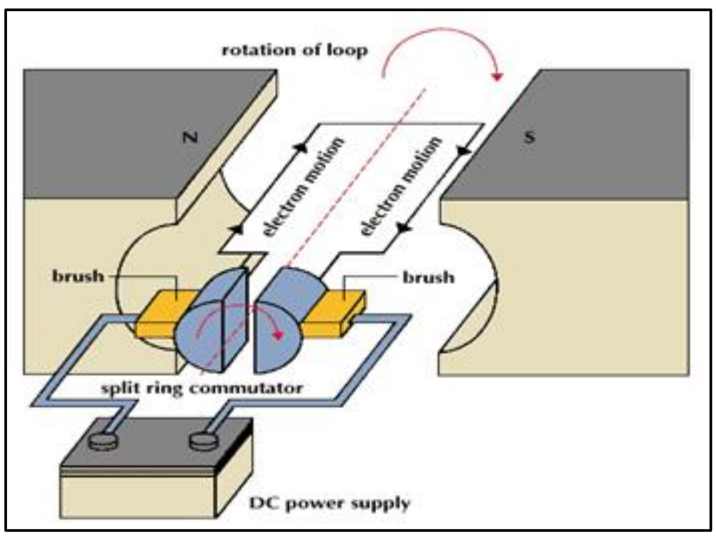


Figure 2 [2]

Example Overview:

Implement and design a system for the Arrow Max1000 board controlling the SN754410 Half-H to drive a DC motor. NIOS will be used as the controller for all the all the Verilog modules. We will look at both motors projects separately in this guide.

Verilog Breakdown:

Looking at **Figure 3** you can see the Verilog code that is used to control the speed of a DC motor. The speed controller is basically a PWM signal that, in this case, allows an external hardware module to control the duty cycle. The code works by iterating through a counter, and when the counter is greater than the set_speed, the state switches low. Due to the counter being 8-bits wide, and the input clock frequency being 256kHz, this drops the PWM signal frequency to 1kHz. This is because for one cycle to complete, count must iterate 256 times.

```
// Variable speed controller for DC motor.
/*
AUTHOR: GRANT SELIGMAN
DATE: 4/3/2020
EDITED BY: JAMES STARKS
DATE: 3/3/2020
FROM: TXST SENIOR DESIGN FALL 2019-SPRING 2020
FOR: TEXAS STATE UNIVERSITY STUDENT AND INSTRUCTIOR USE
DESCRIPTION: This module generates a PWM signal based on the
             input set_speed value. This is an 8bit value
             with 256 possible values. To calculate the duty
             of the PWM generated, divide set_speed/256.
*/
module speed_controller(motor_driver_en, set_speed, clk);
    // PWM signal for the motor driver enable pin
    output reg motor_driver_en;
    // Speed value provided by NIOS in the software layer
    input wire [7:0] set_speed;
    // Base input clock value to modifity
    input wire clk;

    reg [7:0] counter;

    initial begin
        counter = 8'b0;
        motor_driver_en = 1'b0;
    end

    always@(posedge clk) begin
        // Check to see if counter is higher than set_speed
        motor_driver_en = (counter > set_speed) ? 0 : 1;
        counter = counter + 1;
    end
endmodule
```

Figure 3

Figure 4, which is the direction controller, is a simple 1-bit state machine. It checks for a 1 or 0, and depending on the student selection, the state outputs of 10 or 01 will be sent to the input pins on the SN754410. This flips the direction of through the DC motor causing it to switch directions.

```
// Variable speed controller for DC motor.
/*
AUTHOR: GRANT SELIGMAN
DATE: 4/3/2020
EDITED BY: JAMES STARKS
DATE: 3/3/2020
FROM: TXST SENIOR DESIGN FALL 2019-SPRING 2020
FOR: TEXAS STATE UNIVERSITY STUDENT AND INSTRUCTIOR USE
DESCRIPTION: This module sets the direction in which the
             motor will spin. The motor driver inputs
             connect the the inputs of the SN754410 A1 & A2.
             motor_driver_inputs[0] -> A1
             motor_driver_inputs[1] -> A2
             Look at the SN754410 documentation to see how
             Motor Driver Input (A) Truth
                                   |A1|A2|
                 --------------------------
                 Clockwise         |0 |1 |
                 Counter-Clockwise |1 |0 |
*/
module direction(motor_driver_inputs, select_direction);
    output reg [1:0] motor_driver_inputs;
    input wire select_direction;

    always@(select_direction) begin
        case(select_direction)
            // Clockwise
            1'b0:    motor_driver_inputs = 2'b10;
            // Counter-Clockwise
            1'b1:    motor_driver_inputs = 2'b01;
            // Default state is Clockwise
            default: motor_driver_inputs = 2'b10;
        endcase
    end
endmodule
```

Figure 4

FPGA Implementation:

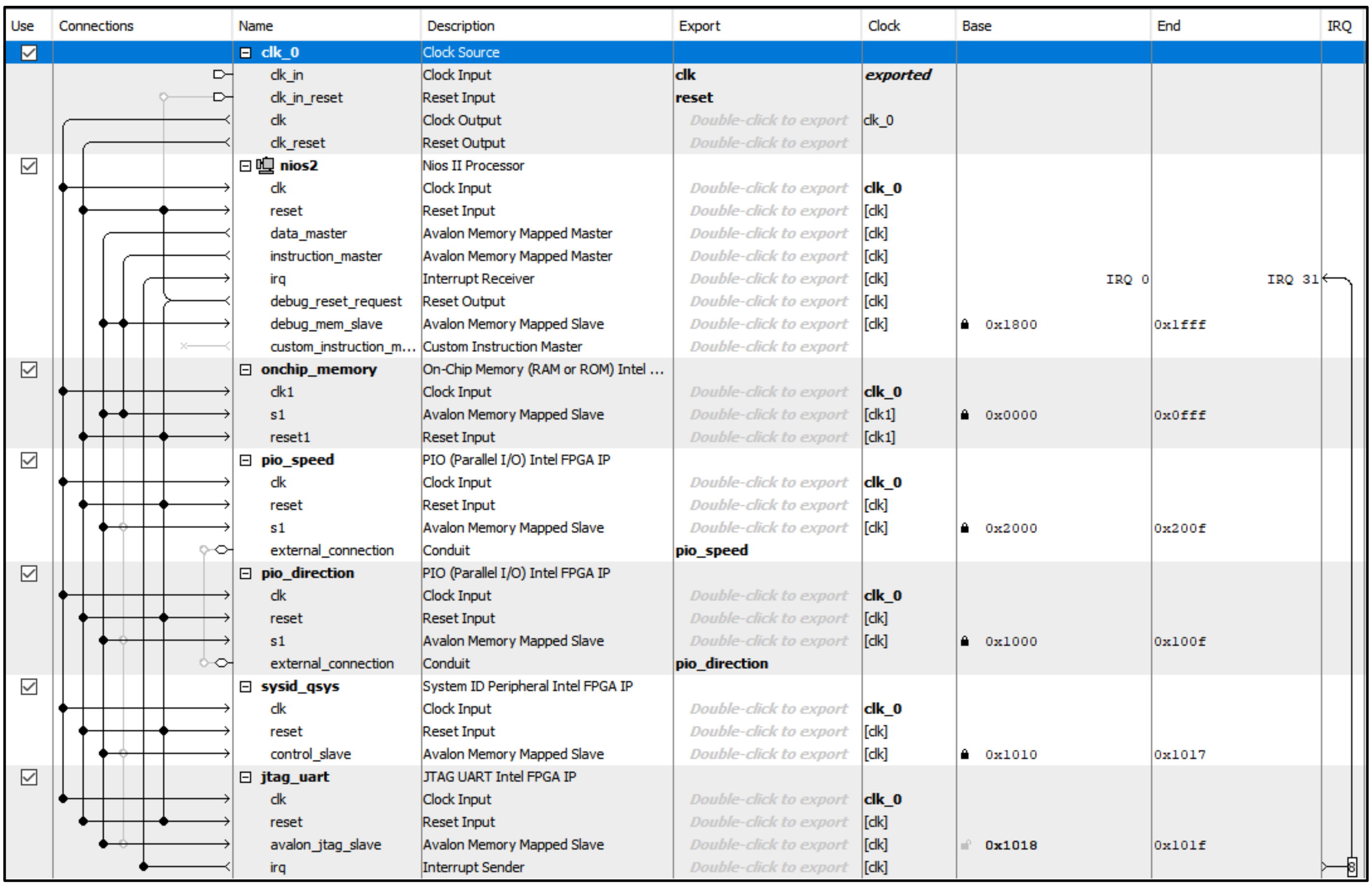
First you will need to create a new Quartus project and include these files…

- **speed_controller.v**
- **direction.v**

Create **Block Diagram/Schematic File** and create **Symbol Files** of all the included Verilog files.

Generate a 256kHz **ALTPLL** for the speed controller.

Design a **NIOS System** using the **Platform Design Tool** with **2 PIOs** to control the **Speed** and **Direction** modules (**Figure 5**).



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
☑		clk_0	Clock Source					
		clk_in	Clock Input	clk	exported			
		clk_in_reset	Reset Input	reset				
		clk	Clock Output	Double-click to export	clk_0			
		clk_reset	Reset Output	Double-click to export				
☑		nios2	Nios II Processor					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31	
		debug_reset_request	Reset Output	Double-click to export	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x1800	0x1fff	
		custom_instruction_m…	Custom Instruction Master	Double-click to export				
☑		onchip_memory	On-Chip Memory (RAM or ROM) Intel …					
		clk1	Clock Input	Double-click to export	clk_0			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]	0x0000	0x0fff	
		reset1	Reset Input	Double-click to export	[clk1]			
☑		pio_speed	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x2000	0x200f	
		external_connection	Conduit	pio_speed				
☑		pio_direction	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x1000	0x100f	
		external_connection	Conduit	pio_direction				
☑		sysid_qsys	System ID Peripheral Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x1010	0x1017	
☑		jtag_uart	JTAG UART Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x1018	0x101f	
		irq	Interrupt Sender	Double-click to export	[clk]			8

Figure 5

Look at **Figure 6** to see how everything is wired up and then **Start Compilation**.

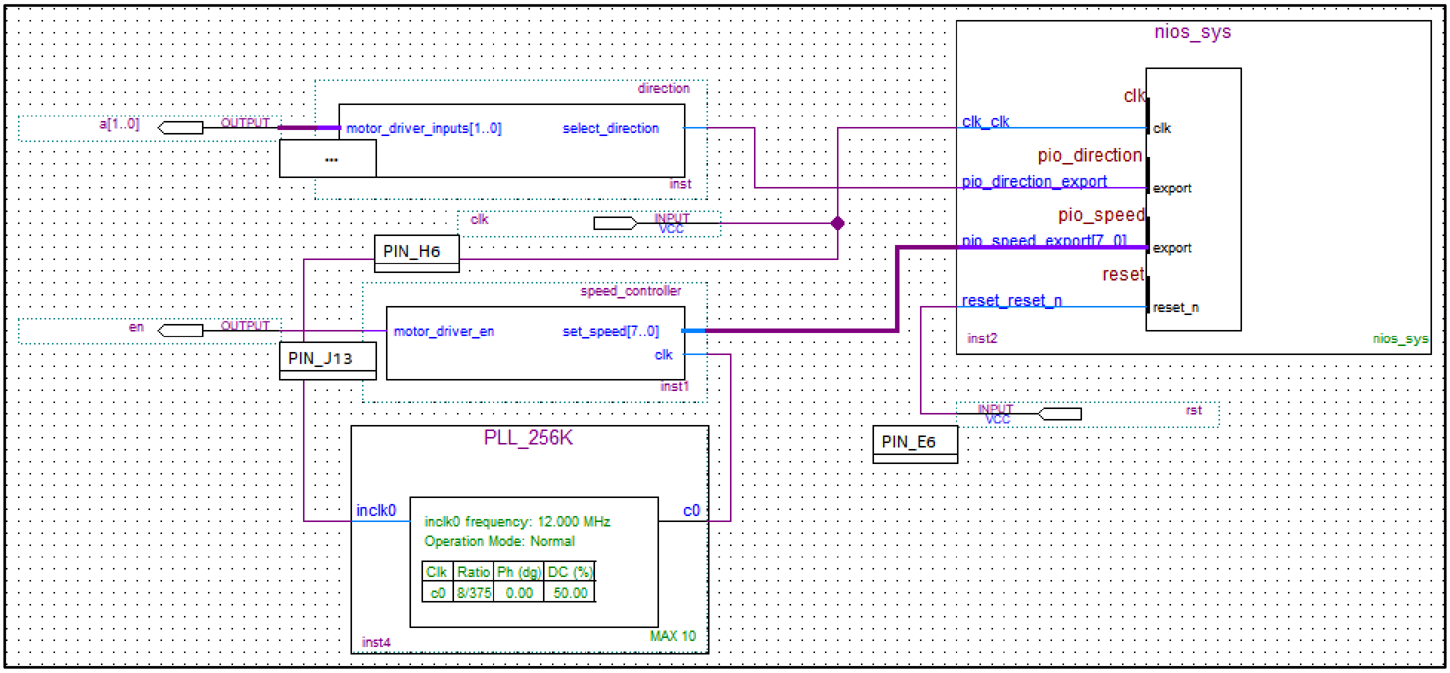


Figure 6

After everything compiles with no errors, open the **Pin Planner** and use **Figure 7** as a guide and set the **Pin Locations**.

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate
a[1]	Output	PIN_J12	5	B5_N0	PIN_J12	3.3-V LVTTL		8mA (default)	2 (default)
a[0]	Output	PIN_L12	5	B5_N0	PIN_L12	3.3-V LVTTL		8mA (default)	2 (default)
altera_reserved_tck	Input	PIN_G2	1B	B1_N0	PIN_G2	3.3 V Sc... Trigger		8mA (default)	
altera_reserved_tdi	Input	PIN_F5	1B	B1_N0	PIN_F5	3.3 V Sc... Trigger		8mA (default)	
altera_reserved_tdo	Output	PIN_F6	1B	B1_N0	PIN_F6	3.3-V LVTTL		8mA (default)	2 (default)
altera_reserved_tms	Input	PIN_G1	1B	B1_N0	PIN_G1	3.3 V Sc... Trigger		8mA (default)	
clk	Input	PIN_H6	2	B2_N0	PIN_H6	3.3-V LVTTL		8mA (default)	
en	Output	PIN_J13	5	B5_N0	PIN_J13	3.3-V LVTTL		8mA (default)	2 (default)
rst	Input	PIN_E6	8	B8_N0	PIN_E6	3.3-V LVTTL		8mA (default)	

Figure 7

NIOS II Setup:

Like in the previous lab guides, start to create a **New NIOS II Application and BSP from template** and select the **Small Hello World** template. Next generate the **BSP** project and you can start editing the application project.

Look at **figures 8 through 11** to get an idea on how to create this application.

You may think the functions in this application are redundant because they are typically included with the standard C I/O and Math libs, but since this project has been created using the Altera small C library those libraries are not available.

A brief overview of the code below…

- **Figure 8** shows the included libraries and the custom **pow** function from the math lib used for calculating X^Y and returns an unsigned 8-bit value.

- **Figure 9** is a custom **gets** used to read in ***N*** ASCII encoded values, characters, or until '***n***' value is entered and returns the count of characters retrieved.

- **Figure 10** is a crude custom **strtol** function that assumes the ASCII value array passed is decimal formatted and converts and rebases the values to be read as standard integer values of 8-bits in length.

- **Figure 11** is the main function of the program that asks the user to input the a value from 0-255, duty cycle of the PWM enable will be speed/256*100. User is then Prompted what direction they want the motor to spin. The desired settings are then written to the hardware modules and the console prints the applied settings.

main.c

```c
//NIOS II DC Motor Lab
/*
 * AUTHOR: JAMES STARKS
 * DATE: 4/26/2020
 * FROM: TXST SENIOR DESIGN PROJECT FALL 2019-SPRING 2020
 * FOR: TEXAS STATE UNIVERSITY STUDENTS AND INSTRUCTOR USE
 * DESCRIPTION: DC motor controller lab.
 *
 * You may notice some of the included subroutines are normally
 * included with with in some of the standard C libraries, but
 * since we are using the the BSP small C libraries, not all
 * functionalities are included.
 *
 * The purpose of this program is to get user input and pipe it
 * to a speed controller and direction controller. These controllers
 * are memory mapped Verilog modules included in the Quartus
 * project.
 *
 */
#include "system.h"
#include "sys/alt_stdio.h"
#include "altera_avalon_pio_regs.h"

/*
 * pow_u8
 * ----------------
 * Computes x to the y using alt_u8 (unsigned chars).
 *
 * x: Base
 * y: Exponent
 *
 * Returns: x^y as uint8
 *
 */
alt_u8 pow_u8(alt_u8 x, alt_u8 y)
{
    // Anything to the 0 is 1
    if(y == 0)
    {
        return 1;
    }
    // Loop y-1 times because we start at 0
    else
    {
        alt_u8 value = x;
        for(int i = 0; i < y-1; i++)
        {
            value = value * x;
        }
        return value;
    }
}
```

Figure 8

```c
/*
 * gets_u8
 * ----------------
 * Use the alt_getchar() subroutine to get max_len
 * characters and store into u8 array.
 *
 * str: Input alt_u8 array
 * max_len: Max acceptable length for str
 *
 * Returns: If less than max_len characters were
 *          entered, return the count.
 *
 */
alt_u8 gets_u8(alt_u8 str[], alt_u8 max_len)
{
    alt_u8 ch;
    alt_u8 count = 0;
    // Read input buffer until '\n'
    while((ch = alt_getchar()) != '\n')
    {
        // Add input character until max characters reached
        if(count < max_len)
        {
            str[count++] = ch;
        }
        // Add null terminator at the end
        str[count] = '\0';
    }
    return count;
}
```

Figure 9

```c
/*
 * strtoi_u8
 * ----------------
 * Convert char array into alt_u8,
 *
 * str: Input alt_u8 array
 * len: Length of input array (We could have looped
 *      till the null terminator, but knowing the len is
 *      important when knowing which sig fig wer're on.
 *
 * Returns: alt_u8 value of input string.
 *
 */
alt_u8 strtoi_u8(alt_u8 str[], alt_u8 len)
{
    alt_u8 value = 0;
    alt_u8 y = 0;
    // Start for the top of the array (least significant value)
    // and work down.
    for(alt_8 i = len-1; i >= 0; i--)
    {
        /* Rebase the ASCII value so it becomes an int with str[i]-48.
         *
         * Pow_u8 is used to scale the significant figures, y is used
         * to keep track of which sig fig index the loop's on.
         */
        value += (str[i]-48)*pow_u8(10, y);
        y++;
    }
    return value;
}
```

Figure 10

```
int main()
{
    alt_u8 speed;
    alt_u8 direction;
    alt_u8 new_length;

    // Main program loop
    while(1)
    {
        speed = 0;
        direction = 0;
        // Allocate character array of length 3.
        alt_u8 str[3];

        // Prompt the user to enter decimal integer between 0 and 255.
        alt_printf("Enter speed[0-255(0-fully off, 255-fully on)]");
        // Call gets_u8 passing max_len value of 3 because there's only 3 sig figs in 255.
        new_length = gets_u8(str, 3);
        // Convert user input from ASCII into uint8.
        speed = strtoi_u8(str, new_length);

        // Same as above, but for direction.
        alt_printf("Enter direction [1-0(1-Counterclockwise, 0 Clockwise)]");
        new_length = gets_u8(str,1);
        direction = strtoi_u8(str, new_length);

        // Write to speed and direction modules to control the motor.
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_SPEED_BASE, speed);
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_DIRECTION_BASE, direction);

        // Print out the settings applied to the motor controller modules
        // Speed - Set PWM duty cycle in the speed_controller.v module
        // Direction - Set the direction in the direction.v module
        alt_printf("\n--------Applied Settings--------");
        alt_printf("\nSpeed: 0x%x", speed);
        alt_printf("\nDirection: %x", direction);
        alt_printf("\n--------------------------------\n\n");

    }

    return 0;
}
```

Figure 11

Figure 12 is what the final project should look like on a breadboard.

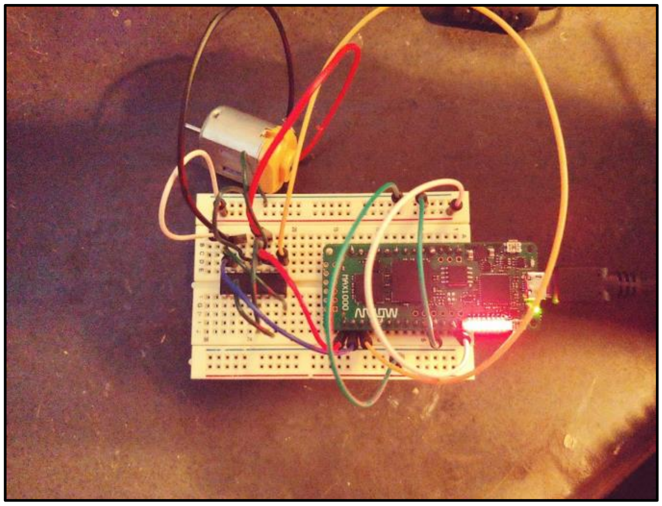


Figure 12

Part 2: Stepper Motor

Example Overview:

Implement and design a system for the Arrow Max1000 board controlling the SN754410 Half-H to drive a stepper motor.

Verilog Breakdown:

Looking at **Figure 13,** we can see four pins and a 5 V source center tapped on both windings. When one of the pins is set low (while others remain high) you get current flowing through half of the winding that creates a magnetic field. The order at which you set the pins low can change the magnetic field to spin the motor clockwise or counter clockwise. The order for clockwise would go setting A to low, then B to low, then C, and finally D. For counter clockwise just reverse the order. Remember, only set one pin low at a time.

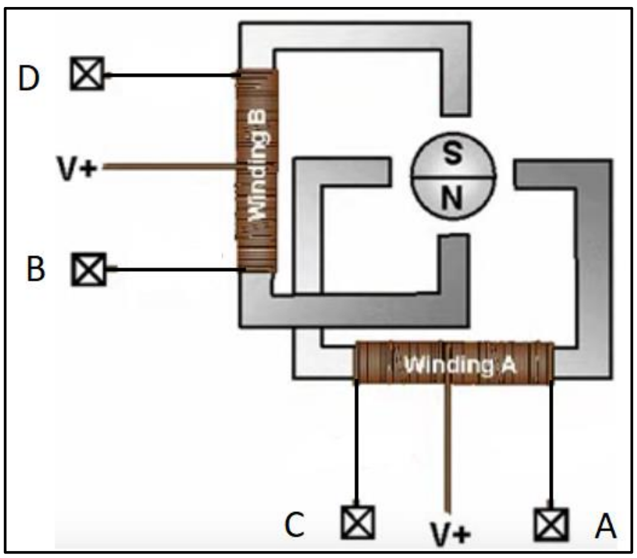


Figure 13 [2]

Figure 14 is a simple bidirectional state machine used to control which coils in the stepper motor are energized. When a 1 is seen on the direction the motor spins counter-clockwise, and clockwise when 0 is on direction. The speed at which the motor spins is dependent on the clock frequency into this module.

```
1  /*
2  AUTHOR: GABE GARVES
3  DATE: 3/25/2020
4  FROM: TXST SENRIOR DESIGN PROJECT FALL 2019-SPRING 2020
5  FOR: TEXAS STATE UNIVERSITY STUDENT AND INSTRUCTOR USE
6  DESCRIPTION: This module is a state machine used for controlling
7               a stepper motor. If direction is high, the state
8               machine reverses direction.
9               [3:0]Drive breakdown
10              Index values in drive            [3][2][1][0]
11              Stepper Motor Winding terminals   A   B   C   D
12 */
13 module stepper_controller(drive, clk, direction);
14     output reg [3:0] drive;     //Signals driving the unipolar motor windings.
15     input clk, direction;       //Clock can also be PWM signal. btn toggles spin direction
16 
17     always @ (posedge clk) begin
18         case (drive)
19             4'b1110: drive = (direction) ? 4'b1101 : 4'b0111;
20             4'b1101: drive = (direction) ? 4'b1011 : 4'b1110;
21             4'b1011: drive = (direction) ? 4'b0111 : 4'b1101;
22             4'b0111: drive = (direction) ? 4'b1110 : 4'b1011;
23             default: drive = 4'b1110;
24         endcase
25     end
26 endmodule
```

Figure 14

Lines 19-22 in **Figure 14** *are tertiary case statements. They are basically a one-line if statement. In the first case statement, direction is checked and if it is high the next state will go to 4'b1101. If direction is low the next state will be 4'b0111. These case statements are more efficient at the hardware level than if/else-if/else statements.*

Figure 15 is the module that controls the clock frequency into the stepper controller. The module essentially acts like a variable clock divider that uses a 1-6bit input to divide the clock down. The output frequency can be calculated with in_clk/division/2 = out_clk.

```
/*
AUTHOR: JAMES STARKS
DATE: 4/24/2020
FROM: TXST SENRIOR DESIGN PROJECT FALL 2019-SPRING 2020
FOR: TEXAS STATE UNIVERSITY STUDENT AND INSTRUCTOR USE
DESCRIPTION: This module is exactly like the previous stepdown
             clock dividers except the division factor is a
             16bit value that can be set by another module (in
             this case, NIOS).
*/
module variable_stepdown(out_clk, division, in_clk);
    output reg out_clk;
    input wire [15:0] division;
    input wire in_clk;

    reg [15:0] count;

    initial begin count = 16'b0; out_clk = 0; end

    always@(posedge in_clk) begin
        count = count + 1;
        if (count == division) begin
            out_clk = ~out_clk;
            count = 0;
        end
    end
endmodule
```

Figure 15

You may be wondering why this code isn't just dividing the clock input directly. Well division in gate logic is very costly to space and inefficient. It is better to have the NIOS C software layer to the division and just send the quotient to this hardware module.

FPGA Implementation:

First you will need to create a new Quartus project and include these files…

- **variable_stepdown.v**
- **unipolar_stepper.v**

Create **Block Diagram/Schematic File** and create **Symbol Files** of all the included Verilog files.

Generate a 1MHz **ALTPLL** for the speed controller.

Design a **NIOS System** using the **Platform Design Tool** with **2 PIOs** to control the **Speed** and **Direction** modules (**Figure 16**).

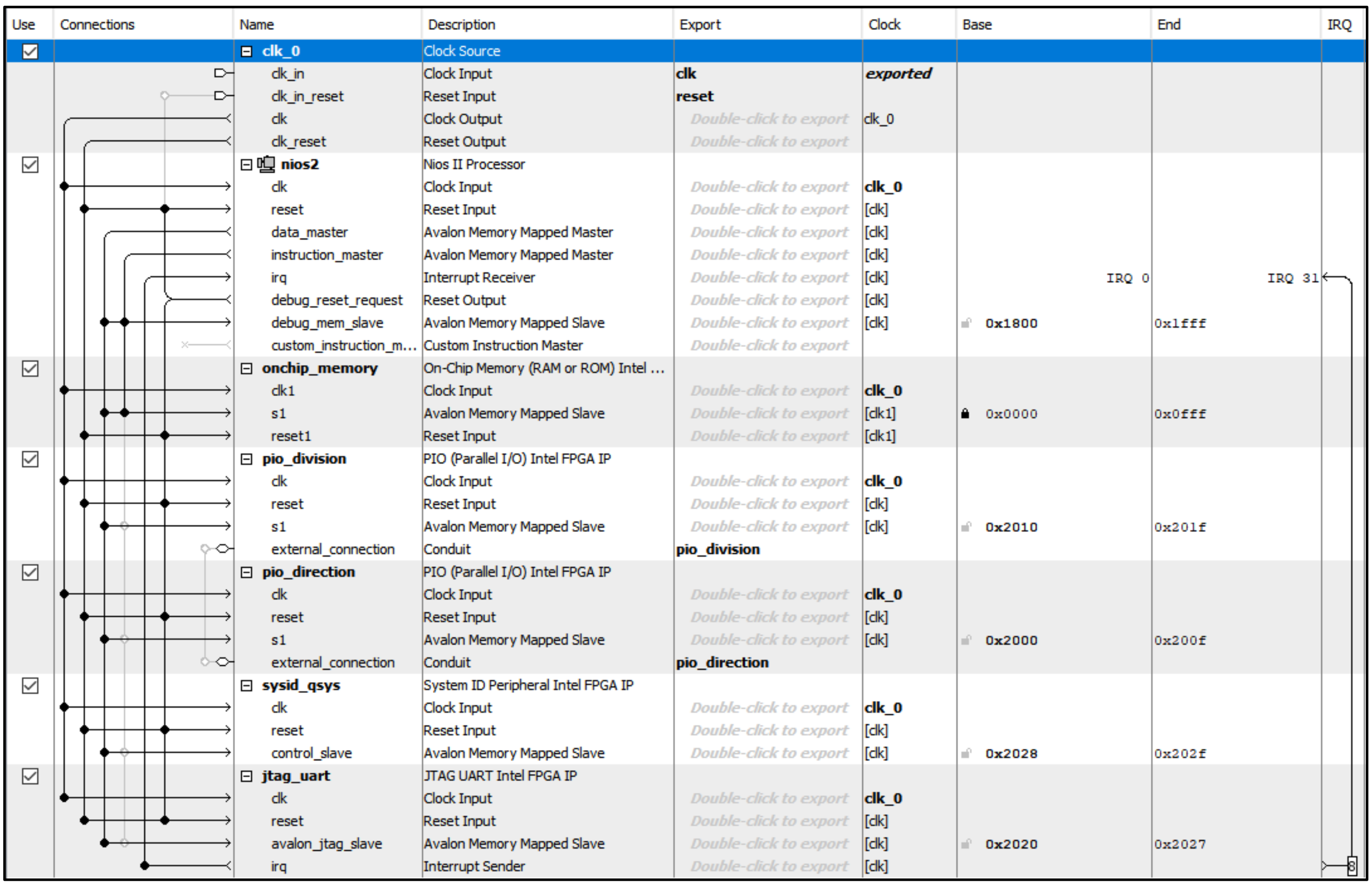


Figure 16

Look at **Figure 17** to see how everything is wired up and then **Start Compilation**.

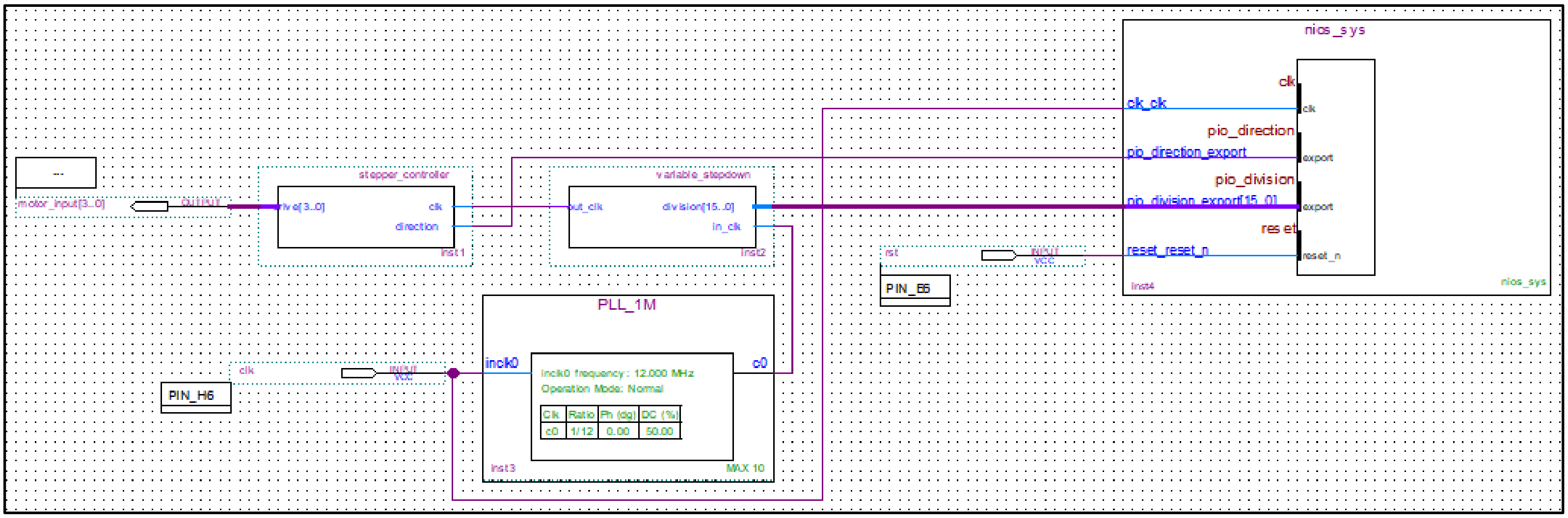


Figure 17

After everything compiles with no errors, open the **Pin Planner** and use **figure 18** as a guide and set the **Pin Locations**.

	Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate
in	altera_reserved_tck	Input	PIN_G2	1B	B1_N0	PIN_G2	3.3 V Sc... Trigger		8mA (default)	
in	altera_reserved_tdi	Input	PIN_F5	1B	B1_N0	PIN_F5	3.3 V Sc... Trigger		8mA (default)	
out	altera_reserved_tdo	Output	PIN_F6	1B	B1_N0	PIN_F6	3.3-V LVTTL		8mA (default)	2 (default)
in	altera_reserved_tms	Input	PIN_G1	1B	B1_N0	PIN_G1	3.3 V Sc... Trigger		8mA (default)	
in	clk	Input	PIN_H6	2	B2_N0	PIN_H6	3.3-V LVTTL		8mA (default)	
out	motor_input[3]	Output	PIN_K11	5	B5_N0	PIN_K11	3.3-V LVTTL		8mA (default)	2 (default)
out	motor_input[2]	Output	PIN_J13	5	B5_N0	PIN_J13	3.3-V LVTTL		8mA (default)	2 (default)
out	motor_input[1]	Output	PIN_J12	5	B5_N0	PIN_J12	3.3-V LVTTL		8mA (default)	2 (default)
out	motor_input[0]	Output	PIN_L12	5	B5_N0	PIN_L12	3.3-V LVTTL		8mA (default)	2 (default)
in	rst	Input	PIN_E6	8	B8_N0	PIN_E6	3.3-V LVTTL		8mA (default)	

Figure 18

NIOS II Setup:

Like before, start to create a **New NIOS II Application and BSP from template** and select the **Small Hello World** template. Next generate the **BSP** project and you can start editing the application project.

Looking at **Figure 19** only the main function was included, this is because the functions above from the DC motor lab were pretty much lifted to this project, with one exception. Since 16-bits are used in the variable clock, **strtoi** must be adapted (**See line 135)**. Even though you will find the stepper stops functioning reliably around 700Hz, or a division factor of ~2857, having these extra bits means the module can more accurately divide down to the desired frequency.

Note: On line 130 there is a bit of math, this converts the requested frequency into a division factor that the variable clock uses. Since the variable step down takes a 1MHz input clock, the formula for calculating the division factor from frequency is:

$$Division\ Factor\ = \frac{Input\ Clock\ Frequecny}{(Wanted\ Frequency\ *\ 0.5)}$$

$$Division\ Factor\ = \frac{1E6}{(Wanted\ Frequency\ *\ 0.5)}$$

$$Division\ Factor\ = \frac{500E3}{(Wanted\ Frequency)}$$

Lastly, like before the options that are applied to the variable step down and stepper controller are displayed to the user, then the program loops. You may notice on the options display, it also outputs the “actual frequency,” this is because most of the time user input frequency doesn’t divide nicely, and integer division truncates any decimal places. So, in the off chance the user enters a frequency that doesn’t divide nicely, the user can see the actual value.

main.c

```c
 */
int main()
{
    alt_u16 freq;
    alt_u16 div;
    alt_u8  direction;
    alt_u8  new_length;

    // Main program loop
    while(1)
    {
        freq = 0;
        direction = 0;
        div = 0;
        alt_u8 str[5];

        // Prompt user for a frequency
        alt_printf("Enter frequency [0-65535]");
        new_length = gets_u8(str, 5);
        freq = strtoi_u16(str, new_length);

        // The div well be written to the variable_stepdown.v module, which
        // expect a uint16 to divde the clock by. The input clock to the
        // variable_stepdown.v module is 1MHz. Look into that module to
        // see how the frequency is dropped by the division constant.
        div = 500000/freq;

        // Same as above, but for direction.
        alt_printf("Enter direction [1-0(1-Counterclockwise, 0 Clockwise)]");
        new_length = gets_u8(str,1);
        direction = strtoi_u16(str, new_length);

        // Write to division constant and direction modules to control the motor.
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_DIVISION_BASE, div);
        IOWR_ALTERA_AVALON_PIO_DATA(PIO_DIRECTION_BASE, direction);

        // Print out the settings applied to the motor controller modules
        // Requested Freq - Requested frequency as entered by the user.
        // Actual Freq - Due to truncation in integer division, actual
        //                freq maybe different. This uint16 is then piped
        //                to variable_stepdown.v module.
        // Direction - Set the direction in the direction.v module
        alt_printf("\n--------Applied Settings--------");
        alt_printf("\nRequested Freq: 0x%x", freq);
        alt_printf("\nActual Freq: 0x%x", (500000/div));
        alt_printf("\nDirection: %x", direction);
        alt_printf("\n---------------------------------\n\n");

    }

    return 0;
}
```

Figure 19

Figure 20 is what the final project should look like on a breadboard.

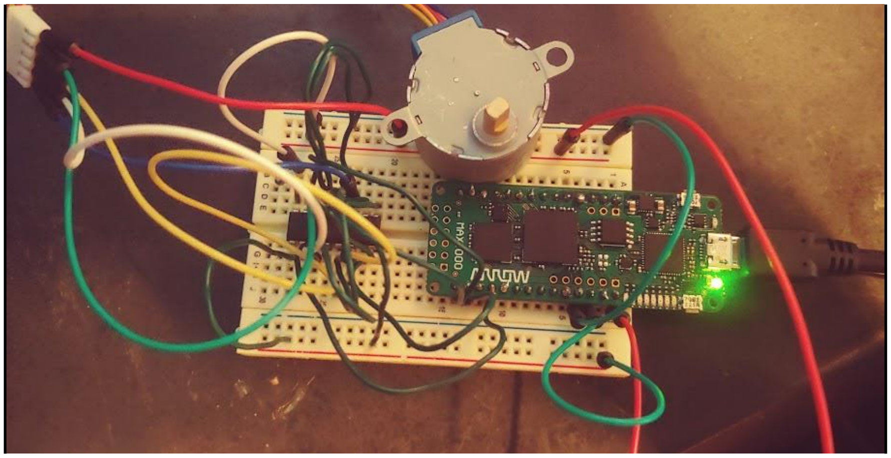


Figure 20

References

[1] "sn754410.pdf." Accessed: Apr. 29, 2020. [Online]. Available: http://www.ti.com/lit/ds/symlink/sn754410.pdf.

[2] *Unipolar and Bipolar Stepper Motors*. .https://www.youtube.com/watch?v=vxxnPJBxG3M