Bus Communications Over SPI

EECE 344, Spring 2012, CSU Chico

Jeremiah Mahler jmahler@mail.csuchico.edu

1 Introduction

A bus is used to allow communication with multiple devices over a shared set of wires. The Serial Peripheral Interface (SPI) provides a means for transferring data between two devices. This project shows how an ARM board can communicate over SPI to a CPLD which defines a bus. Figure 1 gives an overview.

There are two main components used in this project which will be referred to in an abbreviated form throughout this document. The term "ARM" will refer to the ARM STM32L Discovery[4] board. And the term "CPLD" will refer to the Lattice MachXO[2] CPLD board. This project is specific to these development boards although it may be possible to substitute others with some modification.

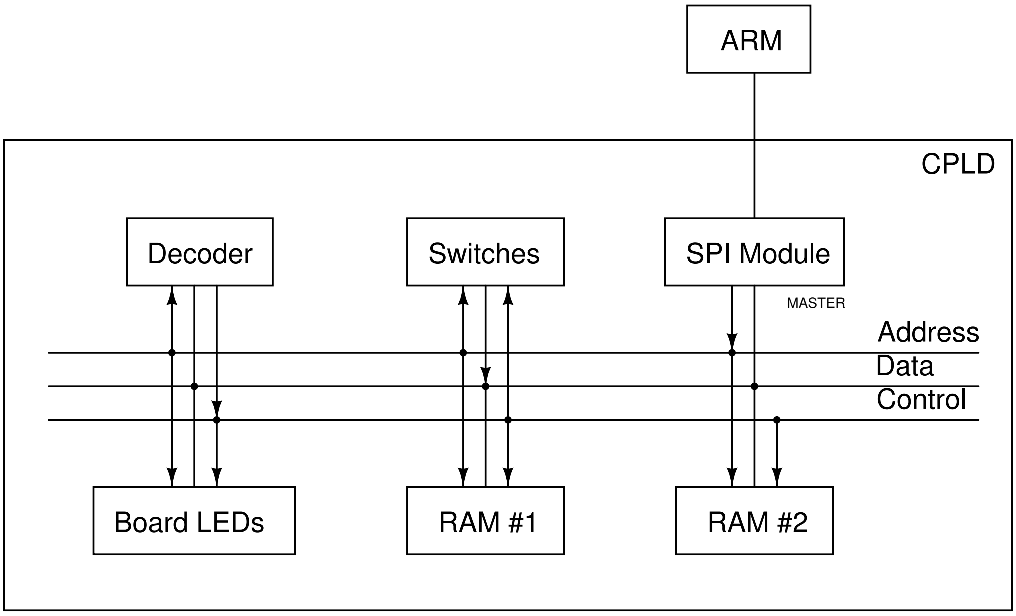
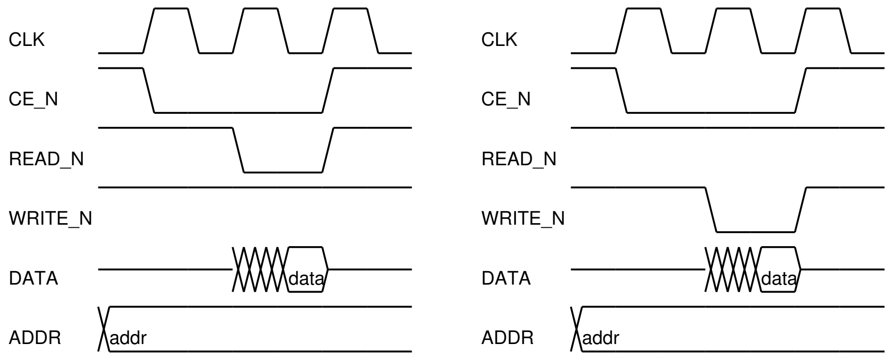


Figure 1: Conceptual overview of bus and related components. The ARM is only able to communicate with the bus through the SPI module. And all the components which are on the bus are controlled by the CPLD. Arrows designate input/output and if there are none it is bidirectional. Refer to table 1 for the specific address used.

Memory Map	
address (hex)	name
0x74	switches
0x6C	bar leds
0x50 - 0x5F	RAM #2
0x2F	board leds
0x00 - 0x0F	RAM #1

Table 1: Device memory map.



(a) (b)

Figure 2: Bus read cycle (a) and write cycle (b).

2 Bus Interface

Before we can communicate with the bus over SPI we need to define how the bus itself works. There are two main operations that must be supported: read and write.

For a read cycle the first step is to drive the address value. Then the device is enabled and some time later read is enabled. At this point the device knows it should it should drive the data but it will take some time for it to become valid. After the data has been read the cycle is finished and the device is disabled. Figure 2 (a) shows this read cycle.

For a write cycle the first step is to drive the address value. Then the device is enabled and some time later write is enabled. At this point the data should be driven by the bus master but it will take some time to become valid. At the end of the write strobe, before it is disabled, the data should be latched by the device. Figure 2 (b) shows this write cycle.

A decoder will be used to control the chip enable for all the components to ensure only one is driving the bus at any given time (Figure 1).

3 SPI protocol

Since the ARM cannot connect directly to the bus the SPI is used as a communication channel. The read and write operations that need to be performed require a minimum of two bytes. A read, for example, would send the address in the first transaction. And in the second transaction the data read would be returned. A write is similar except that for the second transaction the data to be written is sent. The format of the bytes for each transaction is shown in Table 2.

8	7 1
rw bit	address
data	

Table 2: Format of two byte SPI transactions.

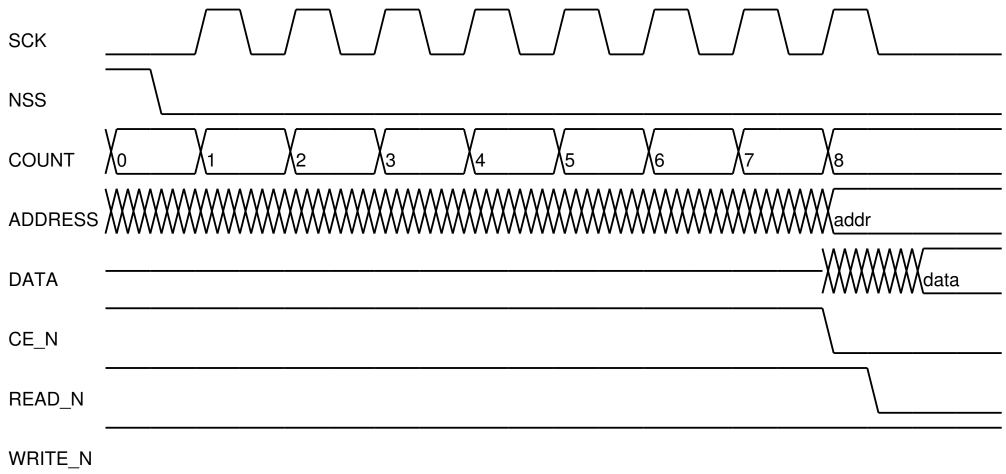
Note, this implementation is configured with the SPI settings shown in Table 3. Both the ARM and the CPLD must use these same settings in order to work properly with each other.

In order to perform a read/write the timing of the bus operations (Figure 2) needs to be integrated in to the timing of SPI operations. The result is showing in Figure 3 for the read operation and in Figure 4 for the write.

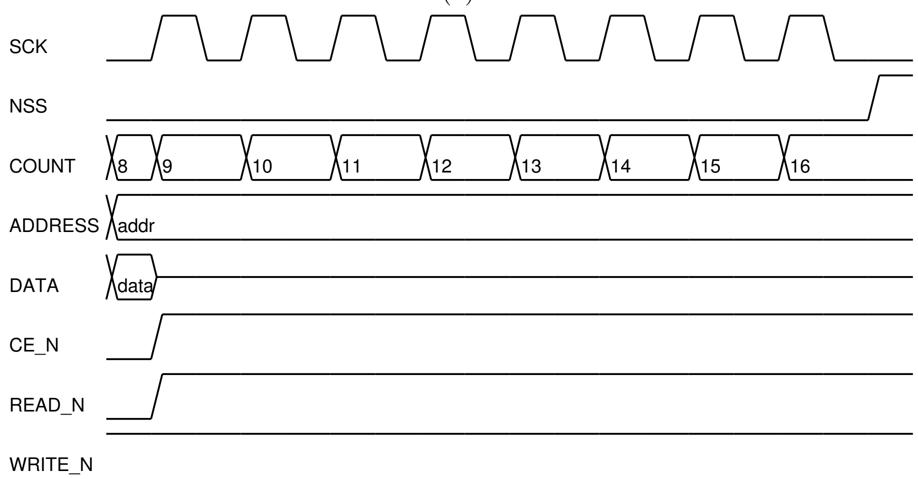
For a detailed timing diagram of the read and write operations refer to Figures 3 and 4.

option	value
MSB	first
CPOL	0
CPHA	0
NSS	slave select

Table 3: SPI configuration options

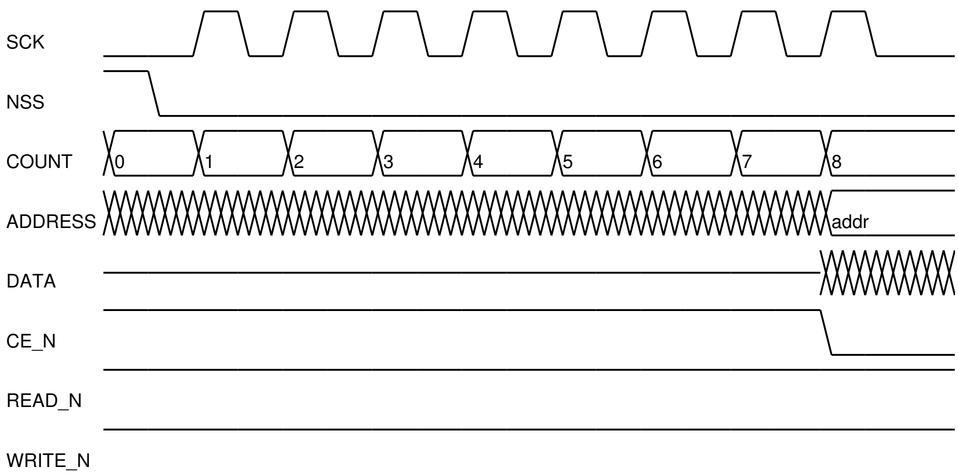


(a)

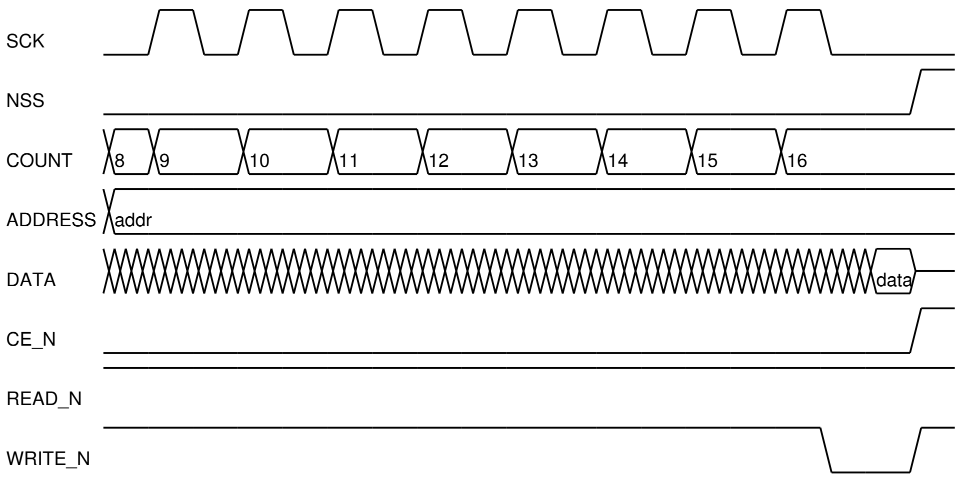


(b)

Figure 3: Timing diagram of SPI read cycle. Part (a) is the first 8-bits and part (b) is the second 8-bits continuing from (a).



(a)



(b)

Figure 4: Timing diagram of SPI write cycle. Part (a) is the first 8-bits and part (b) is the second continuing from (a). A write is not initiated until the end of the second byte because it has to read this byte entirely before it can be written.

4 User Interface

In order for the user to be able to perform read and write operations to the bus devices an interface needs to be defined. The only means of input is using an eight position DIP switch and a USER button. Outputs are provided by an LCD, a bar of eight LEDS, and a group of eight LEDs on board the CPLD.

The bar of LEDs and the group of LEDs on the CPLD both operate similarly. They act as an eight bit register which can be read from and written to. Refer to Table 1 for their specific addresses.

The LCD is the primary means of user feedback. It instructs the user when to enter a command or data and it displays the results. Figure 5 gives an overview of the states of the system.

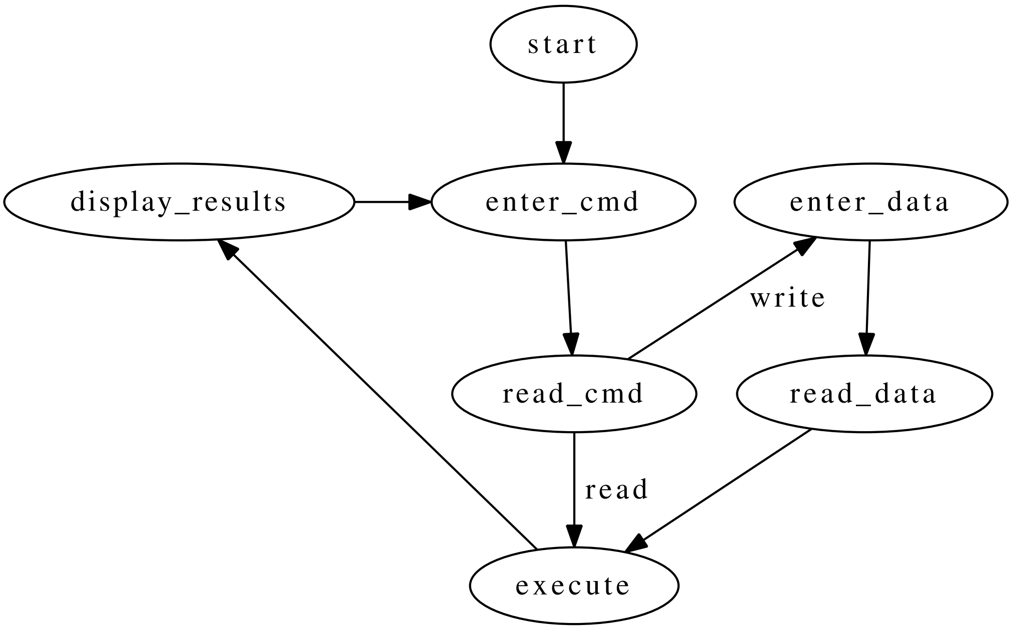


Figure 5: State diagram of ARM board operation. The states 'display_results', 'enter_cmd' and 'enter_data' wait for the USER button to be pressed before continuing. 'read_cmd' reads the input switches to get the command (address and rw bit). If the command is a write 'read_data' reads the switches again to get the data. And 'execute' performs the read/write command.

As an example the following dialog shows how to read from the switches. At each step the USER button is pressed to proceed to the next step. A value of '–' is used to denote when the switches have no bearing on the results.

LCD	switches	description
`CMD`	0xFF	read from address 0x74 (switches)
`74 R F4`	–	the value 0xF4 was read from address 0x74

As another example the following shows how to write to the external LEDs.

LCD	switches	description
`CMD`	0x6C	write to address 0x6C (bar leds)
`DATA`	0x35	the data 0x35 will be written to the address
`6C W 35`	–	value 0x35 was written to address 0x6C

A identical procedure can be performed to read/write to any of the addresses in the memory map (Table 1).

RAM			CPLD			
pin	label	description	function	Mach XO Ball	Header	Pin
12	A0	mem_address	PL17D	L4	J4	36
11	A1	mem_address	PL12D	L2	J4	35
10	A2	mem_address	PL17C	L5	J4	34
9	A3	mem_address	PL12C	K2	J4	33
8	A4	mem_address	PL15C	M2	J4	26
7	A5	mem_address	PL10C	G1	J4	21
6	A6	mem_address	PL10D	H1	J4	23
5	A7	mem_address	PL8D	H3	J4	19
27	A8	mem_address	PL15D	N2	J4	28
26	A9	mem_address	PL11C	J3	J4	29
23	A10	mem_address	PL19A	N4	J4	38
25	A11	mem_address	PL11D	K3	J4	31
4	A12	mem_address	PL8C	G3	J4	17
28	A13	mem_address	PL16D	R2	J4	32
3	A14	mem_address	PL7C	E1	J4	13
31	A15	mem_address	PL7D	F1	J4	15
2	A16	mem_address	PL6D	D1	J4	11
13	DQ0	mem_data	PL7A_LV_T	F2	J3	25
14	DQ1	mem_data	PL17A_LV_T	K5	J3	32
15	DQ2	mem_data	PL18A_LV_T	M5	J3	38
17	DQ3	mem_data	PL9A_LV_T	H4	J3	37
18	DQ4	mem_data	PL8A_LV_T	G4	J3	31
19	DQ5	mem_data	PL16A_LV_T	J4	J3	26
20	DQ6	mem_data	PL15A_LV_T	L3	J3	20
21	DQ7	mem_data	PL5A_LV_T	B1	J3	19
32	Vcc	suppy voltage			J9	5
16	Vss	ground			J3	36

Table 4: Pin assignments between RAM chip and the CPLD which are common to both chips. See Table 5 for those pins which are unique for each chip.

5 Pin Assignments, Schematics

The pin assignments define all the interconnecting wires between the ARM, CPLD and other components.

To locate a pin on the CPLD requires two designations[2, Pg. 11-14]. The first is the header which has names such as J9, J7, etc. And the second is the number of the pin. The board will have pin numbers at the beginning and end of a header to denote the orientation.

On the CPLD the pins correspond to headers (J9, J7, etc) and pins within those headers[2, Pg. 11-14]. The header and pin numbers are printed at the end of each header.

When specifying the pin constraints in Diamond[3] the header and pin number are not available. Instead the "Mach XO Ball" must be specified. And this value is included in the following pin assignments.

The pin assignments are given in Tables 4, 5, 6 as well as the schematic in Figures 6, 7.

In Diamond the pins were configured with standard options: low voltage 3.3 volt CMOS with no pull up or pull down. Output pins to drive the leds were configured for 8 mA drive current.

The input switches to the CPLD can be interfaced by connecting one end to ground and the other end to the pin along with a pull up resistor to Vdd. A resistor value between 1k and 10k should be acceptable. And the pull up voltage for Vdd can be sourced from a pin on the board (Table 6).

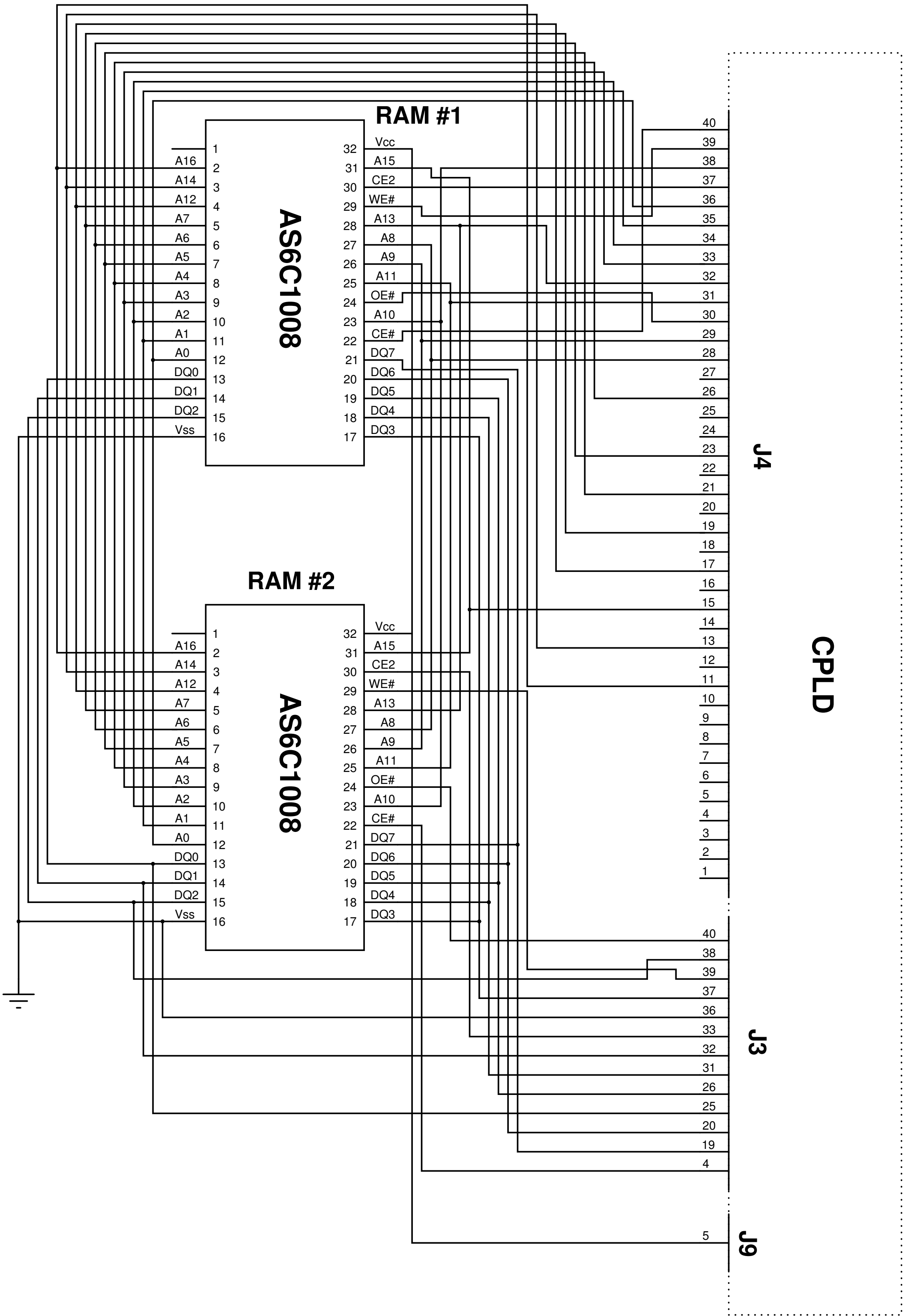


Figure 6: Schematic diagram of wires between both RAM chips and the CPLD.

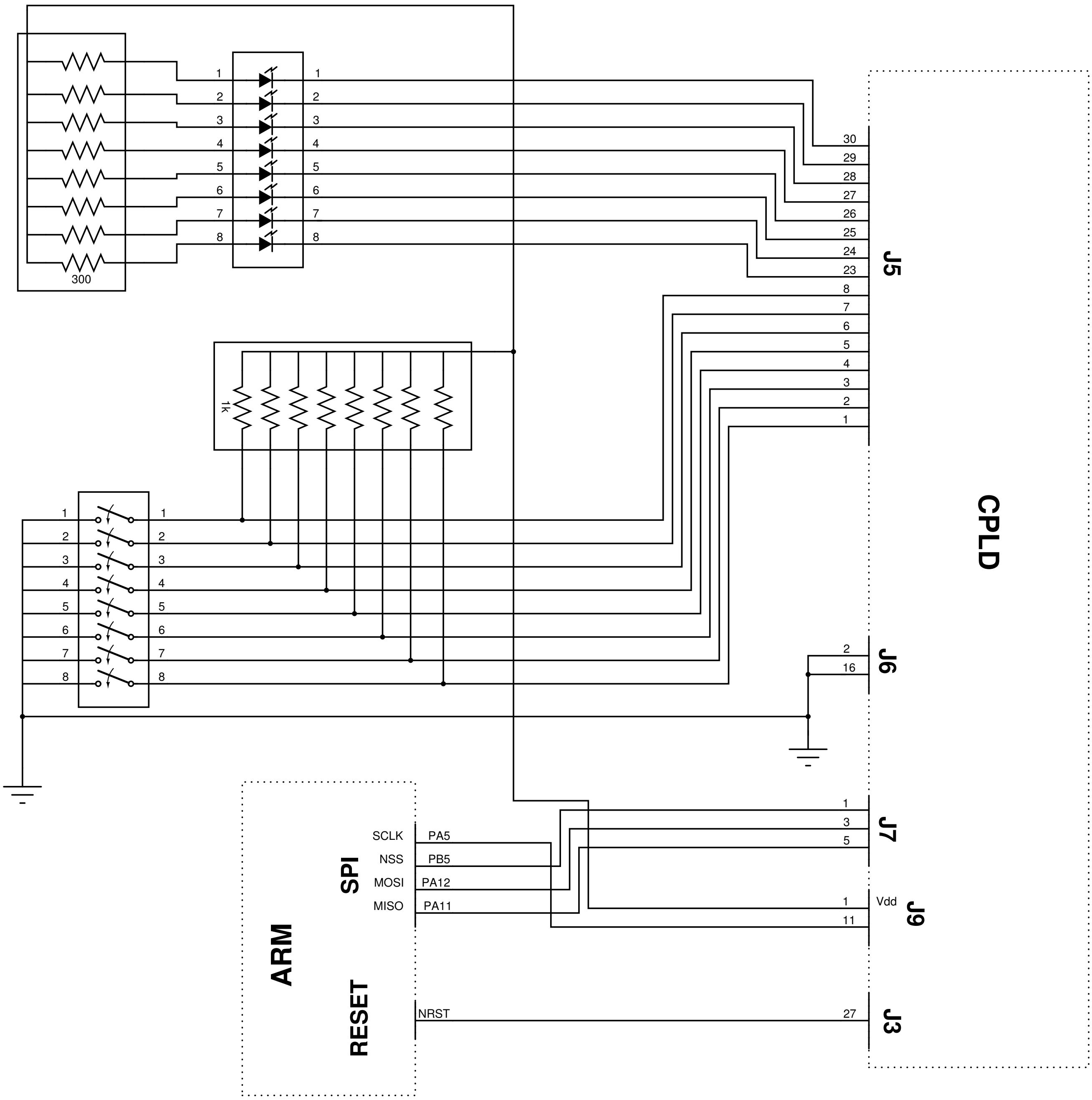


Figure 7: Schematic diagram of wires between the ARM and the CPLD.

RAM #1				CPLD			
pin	label	variable	description	function	Mach XO Ball	Header	Pin
30	CE2	mem1_ce2	chip enable	PL14C	N1	J4	37
22	CE#	mem1_ceh_n	chip enable	PL19B	N3	J4	40
29	WE#	mem1_we_n	write enable	PL14D	P1	J4	39
24	OE#	mem1_oe_n	output enable	PL16C	R1	J4	30

RAM #2				CPLD			
pin	label	variable	description	function	Mach XO Ball	Header	Pin
30	CE2	mem2_ce2	chip enable	PL8B	G5	J3	33
22	CE#	mem2_ceh_n	chip enable	PL11B	J2	J3	4
29	WE#	mem2_we_n	write enable	PL19B	H5	J3	39
24	OE#	mem2_oe_n	output enable	PL18B	M4	J3	40

Table 5: Pin assignments between RAM chip and the CPLD which are unique to each RAM chip.

The output LEDs are connected to the CPLD using a series resistor. Vdd would connect to the resistor which connects to the led (forward biased) which connects to the pin. The value of the resistor should limit the current to approximately 10 mA. A value of 300 Ω is a typical value.

To reset the boards their reset pins must be configured. The ARM board provides a `NRST` pin which is at Vcc when enabled and goes low when the reset button is pushed[4, Pg. 17, 20]. This can then be connected to the CPLD to cause it to reset using `GSRN`[1, Pg. 13, 46, 50, 53; 2, Pg. 8]. Table 6 lists the pins that were used.

SPI						
Verilog		**ARM**	**CPLD**			
name	description	pin	function	Mach XO Ball	Header	Pin
SCLK	SPI clock	PA5	PT9B	D7	J9	11
NSS	SPI slave select	PB5	PR4C	F13	J7	1
MOSI	SPI master out slave in	PA12	PR4D	F12	J7	3
MISO	SPI master in slave out	PA11	PR5C	B16	J7	5
switches						
Verilog		**input switches**	**CPLD**			
name	description	pin	function	Mach XO Ball	Header	Pin
switches[0]	input switch 8	8	PT2C	B2	J5	1
switches[1]	input switch 7	7	PT9A	D8	J5	2
switches[2]	input switch 6	6	PT2D	B3	J5	3
switches[3]	input switch 5	5	PT9C	E8	J5	4
switches[4]	input switch 4	4	PT3A	A2	J5	5
switches[5]	input switch 3	3	PT9D	E9	J5	6
switches[6]	input switch 2	2	PT3B	A3	J5	7
switches[7]	input switch 1	1	PT10A	A10	J5	8
	power, Vdd, pull up				J9	1
	ground				J6	2
LEDs						
Verilog		**output LEDs**	**CPLD**			
name	description	pin	function	Mach XO Ball	Header	Pin
bar_leds[0]	output led 8	8	PT15D	B5	J5	23
bar_leds[1]	output led 7	7	PT12B	A12	J5	24
bar_leds[2]	output led 6	6	PT6E	E7	J5	25
bar_leds[3]	output led 5	5	PT12C	B11	J5	26
bar_leds[4]	output led 4	4	PT6F	E6	J5	27
bar_leds[5]	output led 3	3	PT12D	B12	J5	28
bar_leds[6]	output led 2	2	PT16C	A5	J5	29
bar_leds[7]	output led 1	1	PT13C	C11	J5	30
	power, Vdd, pull up				J9	1
	ground				J6	16
reset						
Verilog		**ARM**	**CPLD**			
name	description	pin	function	Mach XO Ball	Header	Pin
reset_n	active low reset	NRST	PL7B	G2	J3	27

Table 6: Definition of the pin assignments between the ARM board, the CPLD, and other devices. Notice that the switch and LEDs are reversed. This was done so that the orientation from LSB to MSB is from right to left.

6 Development

While the end result of this project is simple its development was far from easy.

All the code to control the CPLD was written in Verilog. While Verilog is a rich language with numerous capabilities[5] only a small fraction of it can be synthesized in to functional code. It was often trivial to devise a solution but the code had to be significantly reworked in order to get it to synthesize[1].

As an example of one of one of the common synthesis problems, suppose it was desired to increment a variable on the rising edge and falling edge of a clock. The Verilog simulator (Icarus) will allow this and it will simulate properly. But this code can't be synthesized because it can't be turned in to a flip flop.

As another example, suppose it was desired to reset all the variables on a falling edge of the NSS signal, and then mutate the variables during subsequent rising edges of the SCK signal. Again this will simulate properly but it cannot be synthesized because two different always blocks cannot modify the same variable.

Beyond synthesis errors there seemed to be an endless supply of problems which were quite intricate and difficult to diagnose. Without a full arsenal of debugging techniques this project is nearly impossible. For the simple problems "trial and error" could be used but more often this was impractical. A logic analyzer with a minimum of four channels was invaluable for diagnosing errors with SPI communications such as glitches and shifts. A C debugger was crucial for stepping through the ARM code to verify that it is behaving as expected.

An example of one of the problems that had to be diagnosed were glitches on the SPI. This normally first appears as incorrect or erratic values being received by the master (MISO). Using a logic analyzer this problem can be clearly seen by spikes which rise and fall at a single clock edge. If it is operating normally it should only rise or fall at a single clock edge. The next logical step was to investigate the CPLD since it is driving the output. With the MISO signal wire disconnected from the ARM it will be seen that the wire is being driven properly and there are no glitches. But how can the ARM disrupt the signal if it is only an input? While it is not exactly clear how this signal is disrupted, it is caused by the GPIO sample speed being configured for too slow of a speed. A trial and error approach to this problem would have been nearly impossible. Only by using a methodical approach along with the proper tools was this problem able to be solved in a practical amount of time.

7 Conclusion

While this project was a success in creating bus communications over SPI its development was far more difficult than expected. Writing synthesizeable Verilog code and debugging the numerous intricate problems were the biggest hindrances to a timely completion of this project.

[1] The term "synthesize" is used to describe when the Verilog code is processed under strict rules by a program such as Diamond or Synplify. Simulating the code under Icarus Verilog is not considered "synthesizing" because it is far less strict.

8 References

[1] Lattice Semiconductor Corportation. Machxo family data sheet, 2010. DS1002 Version 02.9, July 2010.

[2] Lattice Semiconductor Corportation. Machxo2280 breakout board evaluation kit users guide, 2011. March 2011, Revision: EB66_01.0.

[3] Lattice Semiconductor. Lattice diamond design software. `http://www.latticesemi.com/products/designsoftware/diamond/index.cfm?source=topnav`, 2012. [Online; accessed 23-March-2012].

[4] ST. Um1079 user manual - stm32l-discovery, 2012. Doc ID 018789 Rev 2.

[5] D.E. Thomas and P.R. Moorby. The Verilog Hardware Description Language. Number v. 2 in The Verilog Hardware Description Language. Kluwer Academic Publishers, 2002.