Micro P

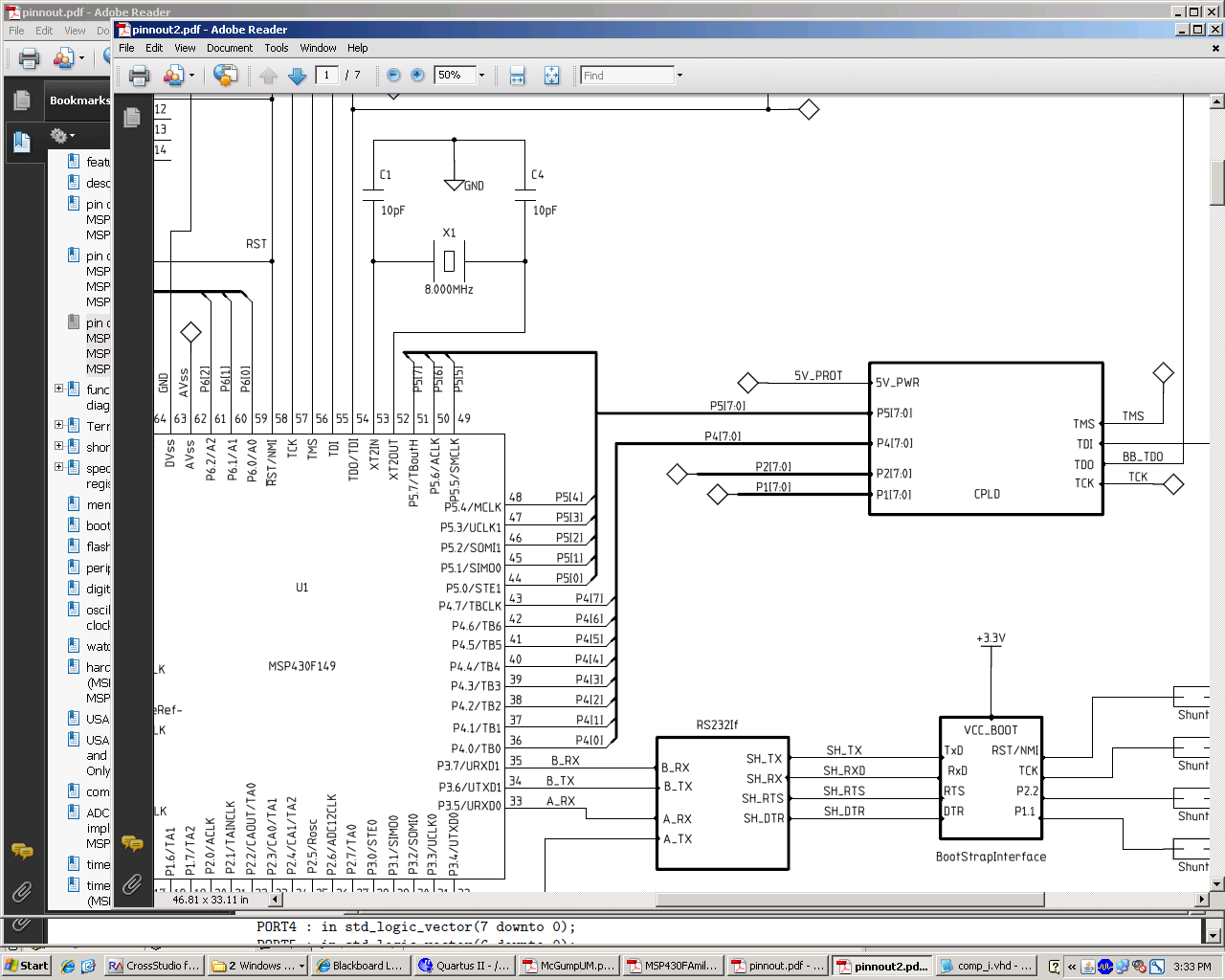
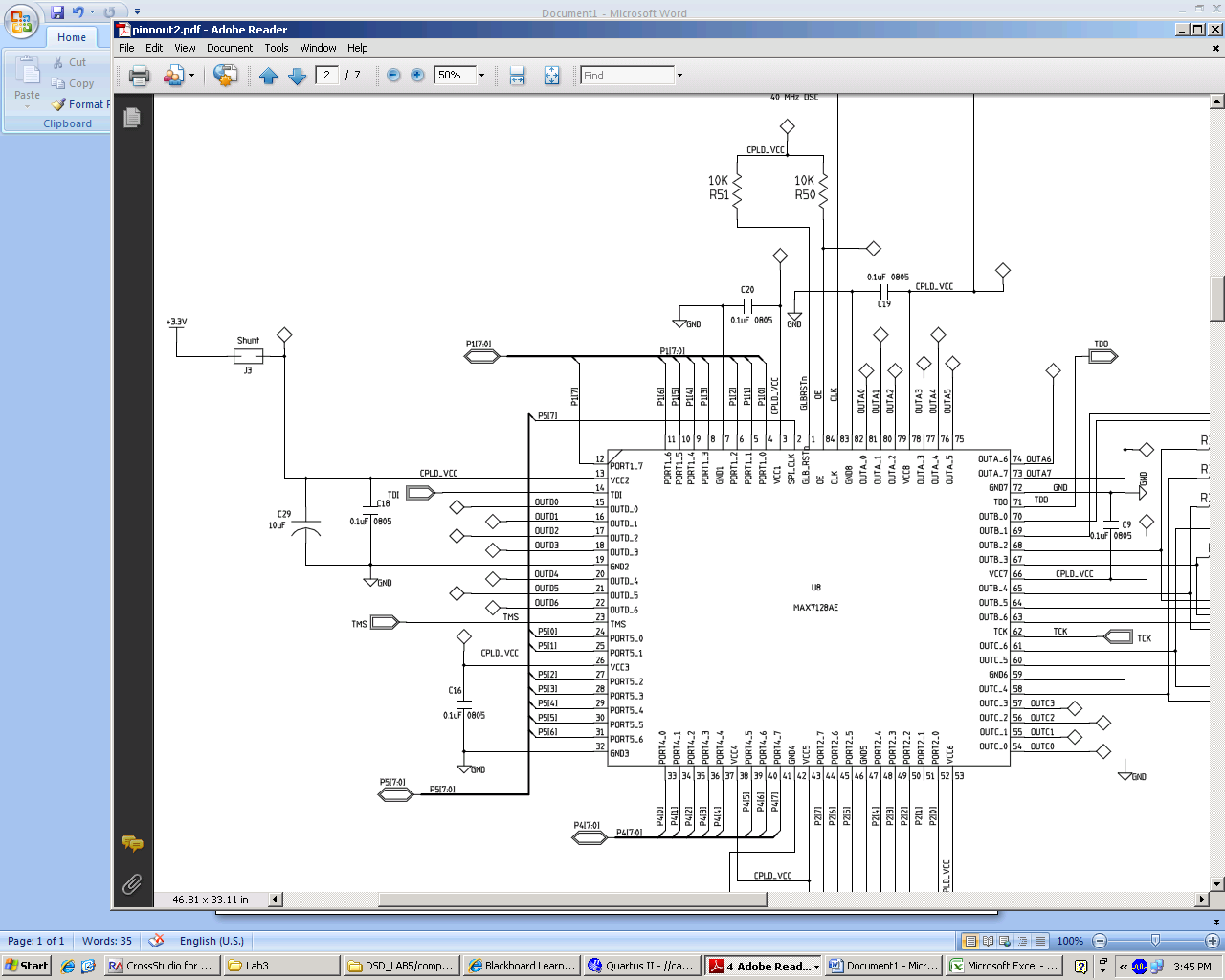
Lab notes for Lab 3: Setting up SPI BUS and the CC2500 chip

Simon Foucher

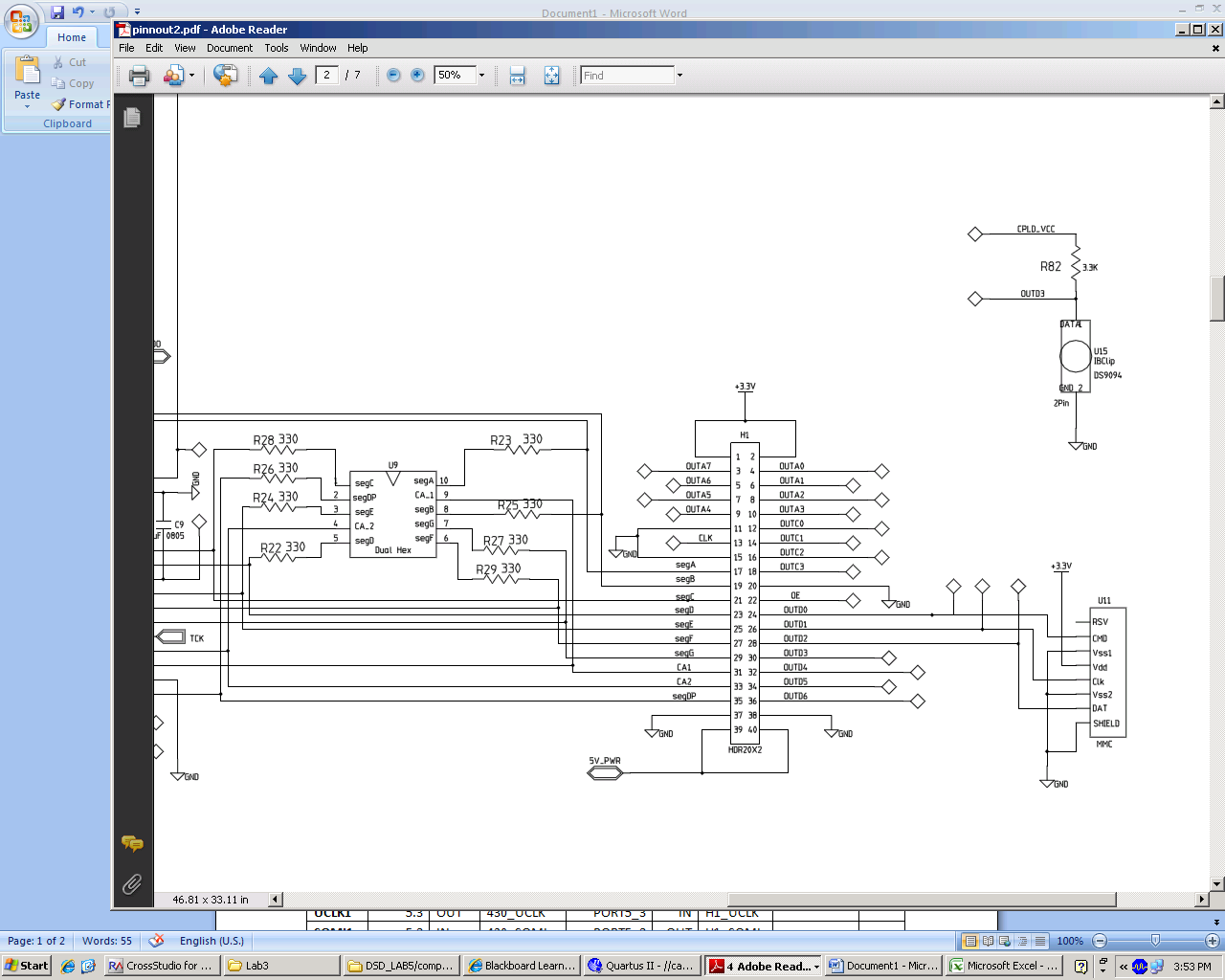
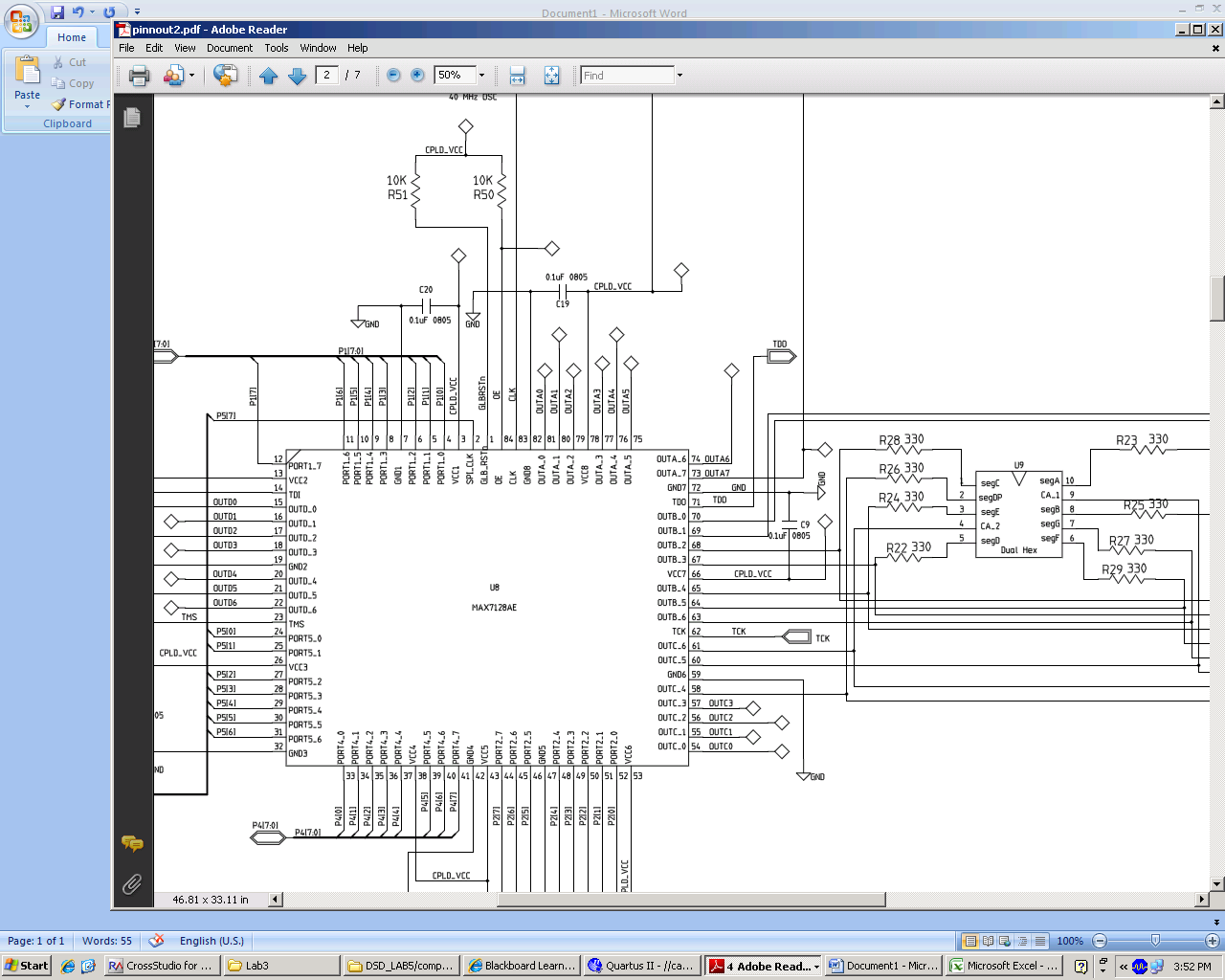
260 223 197

# Hardware connections

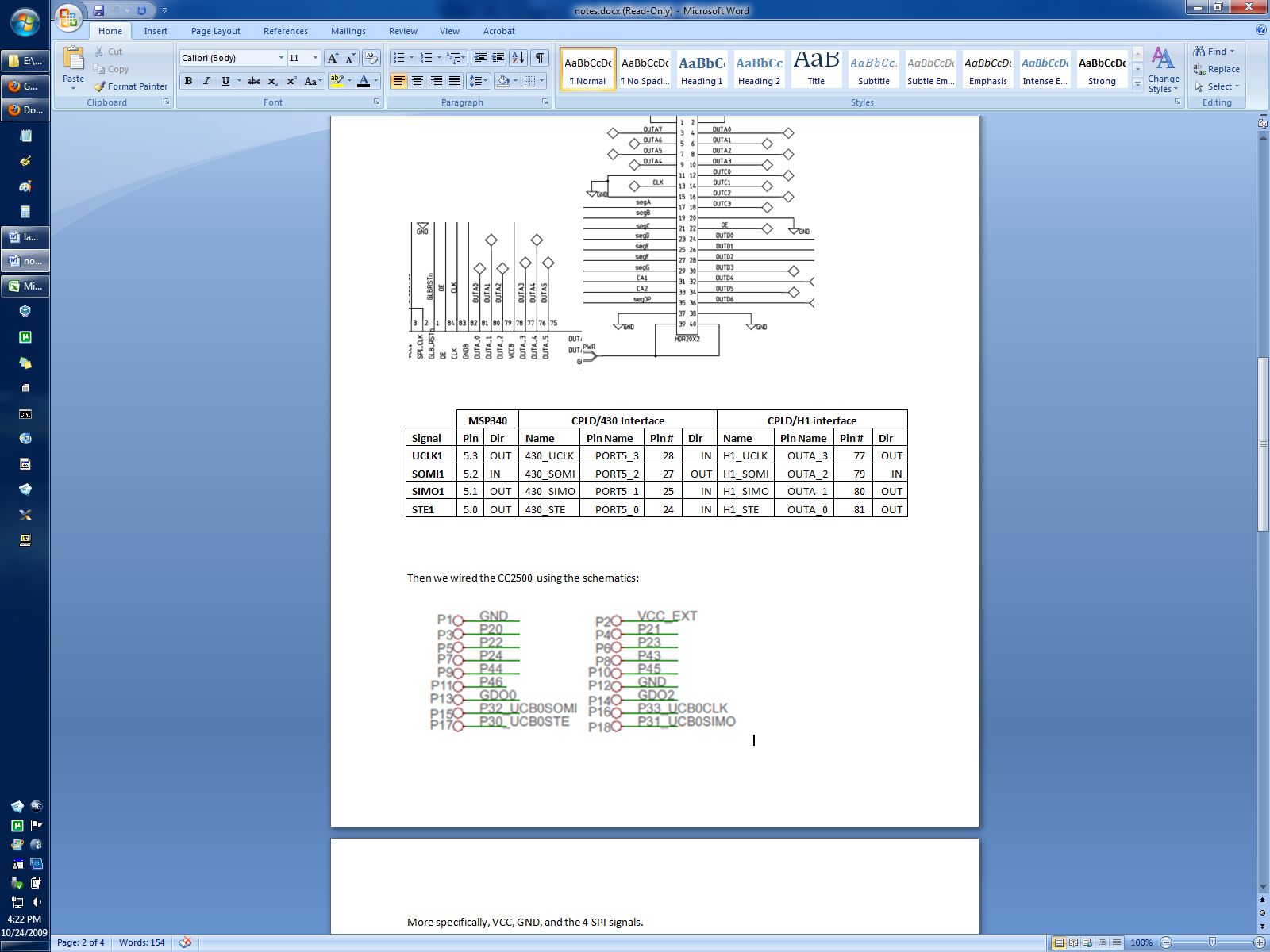
The first task was to connect the proper pins. We examined the pinnout diagrams both from the MSP430 and the CPLD. Since the UART0 was already configured to communicate with the RS232 interface, we chose the UART1 as an SPI interface. Looking at the MSP430, we found that we were interested in PIN 5.[0..3]. Then we located the connections for those pins on the CPLD and noted their pin number.

   
**Figure 1. Left: Pinnout of the MSP430, UART1 SPI pins are 5.0 to 5.3. Right, looking at the CPLD map, the ports are numbered 77, 79, 80 and 81.**

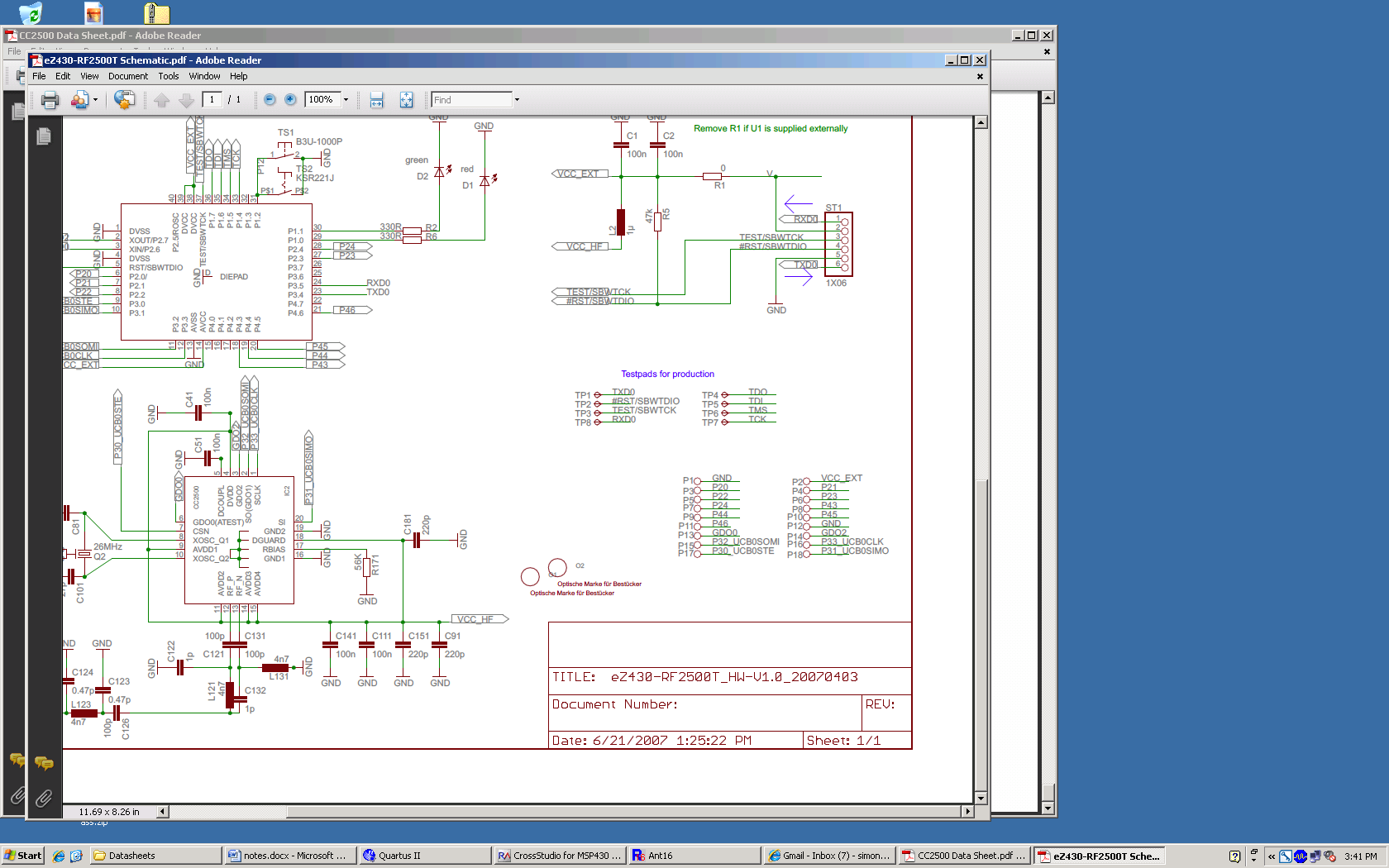
With this information gathered, we focused our attention on the other side of the CPLD, and chose an output port. The decision was made to pick OUTA. We located the pin numbers once again and looked up where they were connected on the H2 board pin array.

  
**Figure 2. Left: Pin numbers for the slave side of the CPLD and Right: corresponding location on the H2 array.**

With this information gathered, we created the following table in order to guide the VHDL code we would need as well as the pin assignment required.



After the CPLD connections made, we wired the CC2500 using the schematics provided:



More specifically, VCC, GND, and the 4 SPI signals were connected.

# Program flow

The program was designed as a finite state machine. When the program starts, before entering in the FSM, we initialize the UART0 which is used to communicate with a computer terminal, UART1, which is used in SPI mode, and finally, we reset the CC2500 chip to put it in IDLE mode.

After this has been done, we run a function that reads the content of all the registers and displays their content on the screen, and enter the FSM.

The FSM was built using a while(1) loop and a switch case selecting between 5 states: INIT, GRAB\_USER\_INPUT, MOD\_REG1, MOD\_REG2 and SET\_UP\_RX. While in each state, the UART0 interface listens for user keyboard input and records it in a buffer. The buffer is composed of 3 slots, therefore recording the past 3 keystrokes. Whenever a key is struck, the UART will set a flag; ‘KEYPRESSED’, to notify the FSM of an event.

Here is an outline of the states.

**INIT**: Initialization state. We clear the key buffer, display a user prompt and move to state GRAB\_USER\_INPUT.

**GRAB\_USER\_INPUT**: This could be defined as the ‘menu’ state, where the user can enter a command. As the user presses keys, the key is scanned within the ISR and is interpreted as 3 things:

* A valid hex key (0-9, a-f, A-F): the key is pushed down the buffer such that the user’s previous 2 hex entries are recorded (for a total of 1 byte)
* A valid command key: There are 4 valid command keys. Whenever a command key is recorded, it is not passed down the buffer, and the FSM interprets the command to act accordingly. Here are the 4 commands:
  + ‘ENTER’: The system transforms both values contained within the key stroke buffer, converts them into a single 1 byte HEX value. If this value is a valid register address, moves to state MOD\_REG1, otherwise, the keystroke is ignored. This value will be used as an address in the next state
  + ‘ESCAPE’: Go back to INIT state; the key buffer is cleared and we get right back into this state
  + ‘TAB’: The system performs a read on all the registers and displays their content to the user. Then, the state goes back to INIT
  + ‘SPACE’: The system goes into SET\_UP\_RX mode
* A key not listed above: The system ignores the entry

**MOD\_REG1:** Read the register selected by the user in the previous state and display its content, prompt the user for a new value, then move to MOD\_REG2.

**MOD\_REG2:** Very similar to GRAB\_USER\_INPUT state. The system will only record HEX and command keys, all the same commands are present. The only difference is the behavior of the ENTER key. Here, the key buffer is read and converted to a single 1 byte HEX value, this value is written in the requested register, and finally the system performs a read on that register and displays the read value on the screen. Afterwards, the FSM goes back into INIT mode.

**SET\_UP\_RX:** We fire a strobe to put the chip in RX mode, read the state and display it on the screen. Afterwards, we repeatedly read the Rx FIFO and look for a valid packet header. Since we know that the header will start with the length of the message 0xE, and the address 0xFF, we scan the data until we receive the proper header. Once we find a proper header, we read ‘packet length’ times the FIFO and display its content if what we read was a displayable ASCII character. Afterwards, we go back into INIT mode.

# IO.c relevant functions

**resetCS2500():** This function is used to reset the CC2500 chip. (the CS2500 is a typo from CC2500!) It follows the data sheet instructions:

1. Set SCLK=1 and SI=0
2. Strobe CSn low / high.
3. Hold CSn high for at least 40 ?s relative to pulling CSn low
4. Pull CSn low and wait for SO to go low (CHIP\_RDYn).
5. Issue the SRES strobe on the SI line.
6. When SO goes low again, reset is complete and the chip is in the IDLE state.

**readRegs():** Performs a sequential read on all the registers by calling the ‘readReg’ function and displaying the read content on the screen in an esthetically nice menu.

**readReg(int address):** Function is called to read the register content at the provided address.

* First, we gain control of the CS pin and set it to low in order to signal that we want to initialize a read.
* We set the read bit high on the ‘address’ integer using an OR operator and set the READ flag to 0, to signal that we expect a status
* Fire the read command to the SPI interface and wait for the RECEIVED flag to go high (set in the ISR). In the ISR, the byte received will be placed in the global variable STATUS, since READ was low.
* Set READ = 1 and fire a second read command to the SPI, and wait for RECEIVED flag to go high again. This time, the ISR will place the received data into the global variable ‘DATA’.
* Finally, set the CS pin to high to signal that we are done reading and exit function.

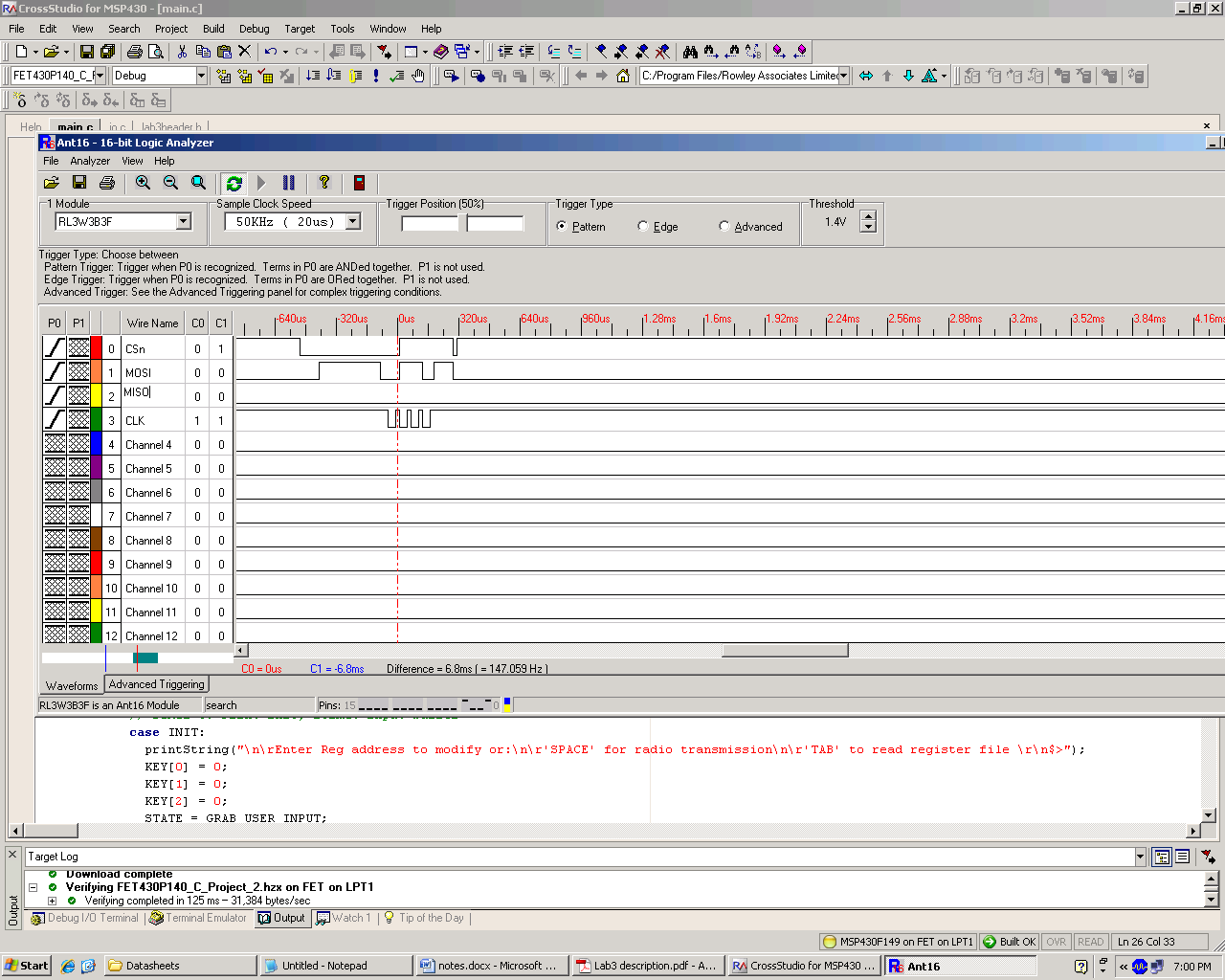
**writeReg(int address, int data):**

FILL ME\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

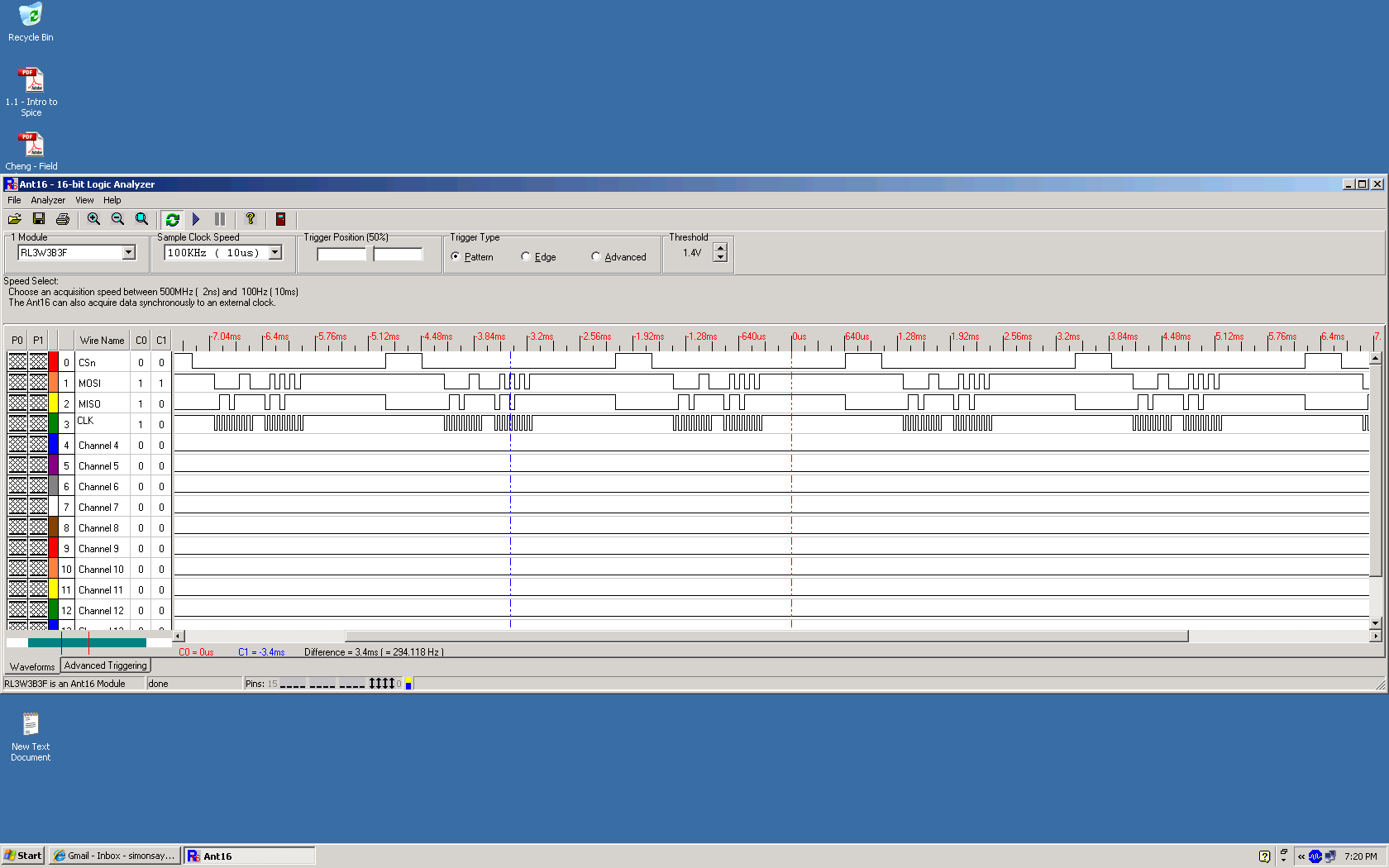
**SPI ISR:** if the flag READ = 0, read UART1 RX Buffer and copy its content to the global variable STAUS. If the flag READ = 1, store the buffer content into DATA. Afterwards, set the flag RECEIVED = 1 to notify the system that we are done.

# SCREEN CAPTURES

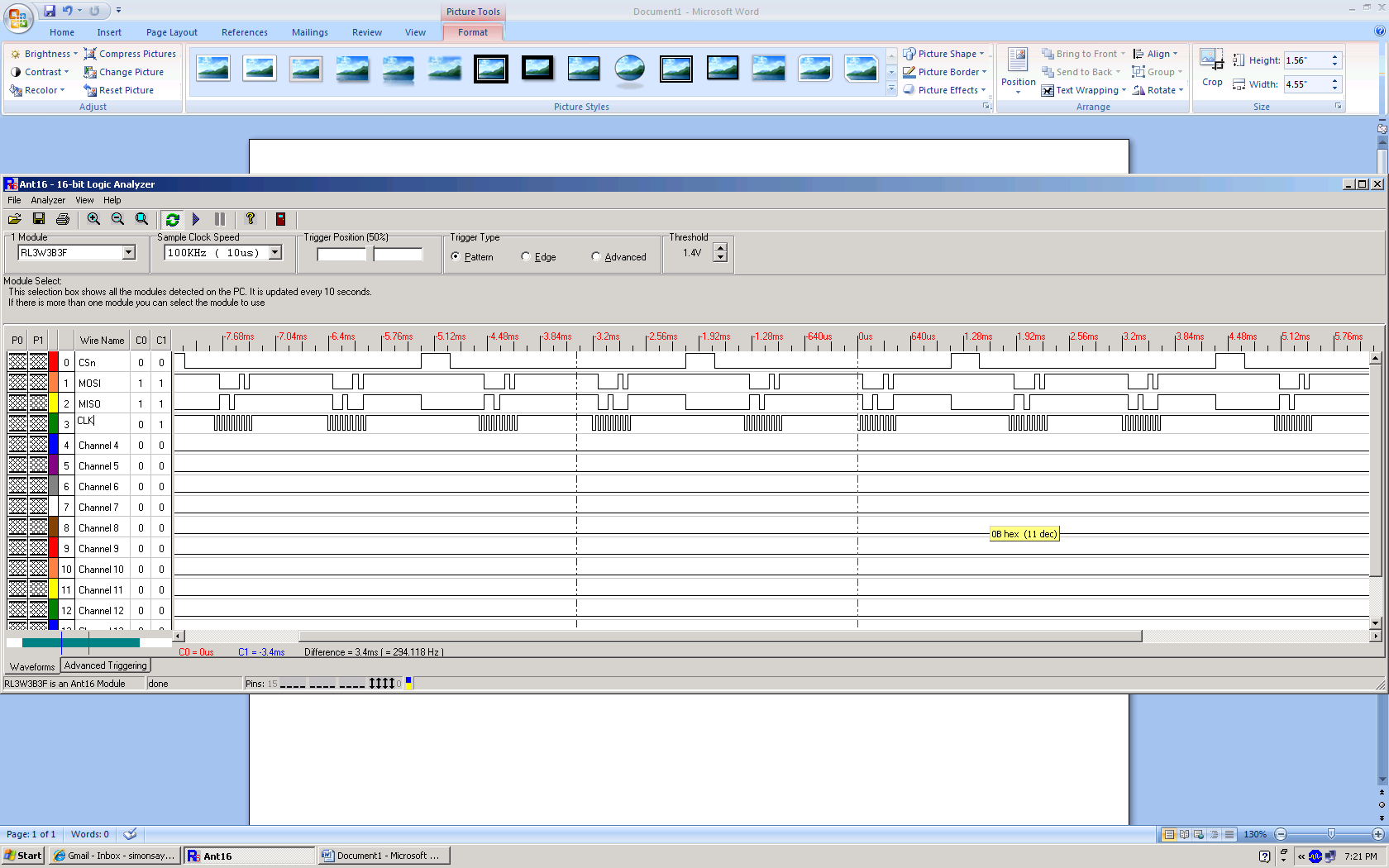
**–A strobe command to reset the chip (SRES)**

****

**–A write to PKTLEN with a value of 0x55**

****

**–A read from the SYNC0 register which we expect to be 0x91 (reset value) if the register was not modified.**

****