The RichArduino

ESE462 SP19 Red

Authors: Nathan Jarvis, Jackson Hyde,

Dean Choi, Jabari Booker

1.) Introduction:

The RichArduino is a FPGA-based alternative to the Arduino platform. The Arduino platform provides support for a swath of useful functions: I2C communication, digital I/O, etc. Because of this, the Computer Science and Engineering department of Washington University in St Louis uses Arduino to teach CSE 132, an introduction to Computer Engineering. However, the platform limits users to a specific processor as the microcontroller of the board. This does not allow for the exploration of microcontroller design for higher level classes. Therefore, we aim to create a new platform inspired by Arduino that allows developers to define a microcontroller for the board, the RichArduino .

The RichArduino replaces the Atmel 8-bit AVR microcontroller found on most Arduino boards with an FPGA. This FPGA is connected to a USB port, an HDMI port, and digital I/O pins. The RichArduino platform is designed to allow developers to program the FPGA with a microcontroller description that can drive and respond to these ports and pins.

2.) DVI Basics:

The RichArduino provides a Digital Visual Interface (DVI) character display. DVI is a video interface developed in the late 90's. It comes in several substandards: DVI-A (analog only), DVI-D (digital only), and DVI-I (analog and digital). DVI has been largely replaced by HDMI and DisplayPort, but it is relevant to us because HDMI has the same electrical specifications for many of its signals. Our design outputs DVI signals to an

HDMI connector. Since essentially all HDMI displays are backwards compatible with DVI, our RichArduino can display characters and images on an HDMI screen.

The display resolution we have chosen is 640x480. Our primary visual interface with the user will be text, comprised of 8x16 pixel characters. This means the effective character resolution of our screen is 80x30. Three control signals are used to interface with the HDMI driving logic, taken from an open source FPGA website [6]: xSync, ySync, and drawArea. Counters for the x and y dimensions of the display are used to time these signals. The fundamental operation of any video display system includes some downtime after transmitting each row and after completion of an entire frame. In this design, the downtime effectively creates a frame size of 800x524 of which only a portion is displayed. The interaction of these control signals relative to the entire frame size is shown in Figure 2.1.

The RSRC processor interacts with the character display via a two-port RAM of depth 4096 and data width 8. Of these 4096 memory locations, only 80x30=2400 are used to display characters. The contents of each memory location is an 8 bit ASCII character value. The logic divides the x-pixel counter by 8 and the y-pixel counter by 16 to create an address into this RAM. The 8-bit output of the text RAM helps address a character ROM which is initialized on bootup to hold the images of the ASCII characters. The addressing scheme uses the 8-bit output of the text RAM to pick a character, the bottom three bits of the x-pixel counter to pick an x-pixel, and the bottom four bits of the y-pixel counter to pick a y-pixel. The output of the ROM is thus either a 1 or a 0 to display either a white or black pixel for that location in the specified character.

To facilitate drawing graphics on the screen, we modified the ASCII ROM to hold two new shapes, a horizontal line and a horizontal line with a dot on top of it. These shapes were stored in the first and second addresses into the ROM, respectively. These shapes enable drawing one-dimensional axes with a point on them.

DVI and HDMI both encode their transmissions using Transition Minimized Differential Signaling (TMDS), a type of encoding that transforms 8 bit input words to 10 bit output codes (8b/10b). Thus, the physical-layer side of our DVI interface accepts 8 bit values for red, green, and blue and converts them via TMDS to 10 bit codes which are shifted out serially to the pins of the HDMI connector.

3.) USB Basics:

There are three main versions of USB connection. The USB 1.0 was developed in 1996, running at maximum speed of 12Mbit/s. Then USB 2.0 and 3.0 were founded in 2001 and 2011 respectively. The RichArduino uses a USB 2.0 specifications with a Type B connector (shown in figure 3.1) at a maximum throughput of speed of 480Mbit/s. The FT201x is an FTDI IC that offers a full speed compact bridge to I2C device from USB 2.0 connection. The device is strictly a slave in I2C bus and operates up to 3.4Mbit/s. was used to establish USB connection with the host computer. The specification of the FT201x chip is summarized below:

USB Chip: FT201x chip from FTDI

Power: Supports bus-powered configuration where it powers 5V Vcc and 3.3V internal IO pins and from the 5V power from the USB host. Up to 500mA can be drawn out from the USB bus. (shown in figure 3.2).

Data transfer: The USB chip stores data from the host into a 512 Byte FIFO RX buffer (Figure 3.3). The I2C Controller inside the FT201x operates as a slave and is addressed by the I2C master running on the FPGA. The master queries the number of bytes present in the RX buffer and, if four bytes constituting a whole word are present, initiates a read of four bytes. The actual I2C sequence is shown in figure 3.5. This process is repeated as necessary.

Clock and reset: FT201x chip generates 24MHz clock from the USB clock on CBUS0 and also sends reset signal to CBUS3. We measure all CBUS except CBUS0 are logic soft high by default, 2.8V. When the USB host computer drive CBUS3 low, it first drives it high to 3.3V then drops it to 0V for 2ms to ensure complete reset of the device.

Communication protocol: I2C

Communication speed: 100kHz

Slave Address: 0x22 by default. It can be changed by reprogramming the MTP memory section by issuing general call address 00000000 on either USB or I2C connection to FT201x chip.

Chip layout: Figure 3.4 (SSOP package)

4.) Architecture:

The RichArduino is built on a 4 layer board from ExpressPCB. This board includes an internal power, and ground layer as well as an external signal layer on either side of the board. These external signal layers each contain various surface components including resistors, capacitors, ferrite beads as well as various Integrated Circuits (IC) and connectors (see Figure 4.1). All resistors, capacitors, and ferrite beads

use a standard 0805 surface mount footprint on the board. Bypass capacitors are used as first order filters for the DC power supplies to the Spartan6 FPGA. AC signal noise is shorted to ground while a clean DC voltage is provided to the FPGA. In addition to filtering, bypass capacitors guard against voltage drops from spikes in currents draw. These components were added at the suggestion of Dr. William D. Richard because previous CSE462M boards were successful without them.

The RichArduino board draws power from the 5V USB bus. The LMP3990-3.3, LPM3990-1.2 IC's are linear regulators used to step down USB 5 Volts down to 3.3V and 1.2V respectively. The 3.3V supply is connected to the power plane and is used to as the auxiliary power supply(VccAUX) for the Spartan 6 FPGA. The RichArduino uses 3.3V I/O standard because a higher potential peripheral (5V) could damage the FPGA. The 1.2 V potential is used only for the core voltage (VccINT) of the Spartan6. The 1.2V is distributed through a secondary pseudo power plane. This secondary plane is a rectangular surface etched directly beneath the FPGA and allows for easier trace layout and connections.

The FT201x IC fulfills multiple roles on the RichArduino. The primary role of this IC is to provide an interface from the host computer's USB bus and the RichArduino's I2C bus. There is a single I2C bus on the RichArduino shared with I/O, FT201x, and the FPGA. The Spartan6 acts as the only master on this bus, all other peripherals are slaves. This unified bus can be split into two seperate I2C buses by uninstalling the 0 Ohm resistors R27 and R28. Further detail of I2C specifications and our specific implementation of the protocol can be found in the VHDL section of this report. The

FT201x also has 6 configurable IO pins . We use the CBUS0 pin on the chip to output a 24 MHz clock used in clocking the RichArduino. As a redundancy we included a 25 MHz crystal oscillator (ASFL1) as backup clock in case the FT201x clock did not work in implementation.

The RichArduino is programmed by a 32 Mb flash memory (W2532JV) over a Serial Peripheral Interface (SPI). The flash module is part of a Joint Test Action Group (JTAG) chain that includes the Spartan6. The flash module and FPGA can both be programmed over a JTAG interface using the IMPACT Software (refer to Figure 4.1). Designs programmed into the FPGA alone will be lost once power is cut from the IC. We loaded our design into the flash module so that the they would remain without power (refer Figure 4.2). On boot up this design is loaded into the FPGA.

The RichArduino FPGA design consists of 6 modules on a shared 32 bit data and address bus (see Figure 4.3). These modules include the RSRC, EEProm, SRAM, HDMI Module, I2C Module, and DIO Manager. All clocked modules use a 25 MHz clock generated by a Digital Clock Manager (DCM). The DCM uses the 24 MHz clock from the FT201x bus pin to produce the 25 MHz clock as well as a 100 MHz clock. The 100 MHz clock is used by the HDMI module to fulfill the DVI/HDMI specifications. A bus pin on the FT201x is also used of as the reset_l signal for the design. On reset from the GUI, the drivers write the reset pin low and then high to generate a reset signal for the design.

The RSRC module is an original processor design by Dr. William D. Richard and given to the Red Team as a completed component. It was changed in no way from the

original version(V0.806) provided by Dr. Richard at the beginning of the SP19 semester. The RSRC is a Harvard Architecture is a 32 bit, RISC processor using the fetch, decode, execute cycle(non-pipelined)[5].

EEPROM is a 16 KB read only memory containing the monitor program. The module is mapped to the bottom of the address space (see figure 4.4) so that when the RichArduino poll for new programs sent over USB on power up or reset. Refer to section 6 for greater detail on monitor program.

SRAM is a 16 KB memory used to hold and run downloaded programs. This memory is mapped to 0x1000 in memory. All downloaded programs must have a .org 0x1004 directive to properly branch with this address decode (the extra word is the space need for the program header).

The I^2C module allows the RSRC to communicate with the FT201x over an I^2C Bus. The module itself acts as the single I^2C master on the bus. The RSRC can read complete 32 bit words from this module. This functionality allows machine instructions to be loaded into SRAM from the host computer. A more in depth description of the I^2C module can be be found in section 9.

The HDMI module implements 640X480 character display over DVI. The RSRC writes ASCII character values to the module that are displayed screen as actual characters. The HDMI module is explained in greater detail in section 3 and section 9.

The DIO Manager is used to control the 20 Digital IO pins on the RichArduino. Each pin is either an input(1) or output based(0) based upon the status of a configuration the corresponding bit in configuration register(.i.e config(1) controls

DIO_1). The status of an input pin is placed in the lowest bit of the data bus when it is read. Output pins will continuously drive the value of the least significant bit on the data bus from the most recent write to that pin.

5.) Assembler:

To assemble code, we use a two-pass assembler. On the first pass, the assembler searches for labels, creating map where the key of each entry is a label name, and the value is the associated value to that label. On the second pass, each assembly instruction is assembled to machine code, replacing labels with their associated values whenever they appear an instruction. The assembler supports all instructions and pseudo-ops described in *The SRC Instruction Set* excluding the *een*, *edi*, *rfi*, *svi*, and *ri* instructions and the *.db* and *.dh* pseudo-ops [1]. Binary values are not supported, but hexadecimal values are with a “0x” prefix to the hexadecimal value. Comments are marked with “//” rather than “;”.

6.) Monitor:

The monitor program is designed to run on boot up or reset of the RSRC to monitor the I2C for downloadable programs. The assembly code with comments is shown in figure 6.1. The monitor program has three main loops. The first loop is responsible for USB_setup. This is the step where we are constantly checking if the FIFO buffer of the FT201x chip has more than 4 bytes of data. This is ensuring that our design always acquire 32-bit long instruction without failure. Then the second loop is responsible for actually reading in this 32-bit instruction and saving to the SRAM.

The last loop is only used once in the beginning. When a user tries to download a program to our design, our GUI generates a header that contains the length of the program that user is trying to download. We read from the USB with the assumption that the first word sent. We save the header value to our program counter register. After that, we read instructions using first two loops until our program counter decrement to 0. At this point we branch to SRAM and run the downloaded program.

7.) Host GUI:

The RichArduino companion host GUI allows users to program or and interface with the RichArduino board via USB. The GUI is compatible with the 32- and 64-bit versions of Windows 7, 8, and 10 operating systems. The GUI is programmed using the Qt library (version 5.12.2) for the user interface and D2XX driver from FTDI (windows version 2.12.28) to interface with the FT201x usb controller on the RichArduino. Users can use the GUI to:

- Write, edit, save, assemble, and load assembly programs for the RichArcuino
- Assemble and save machine code to the host computer
- Reset the RichArduino

The programming environment of the host GUI detects programming errors, warns the user of word alignment, and informs the user when the board is disconnected and reconnected to the host.

8.) Shield Demo:

The Arduino shield used in our RichArduino project is a accelerometer (MMA8452Q). It has configurable 12-bit resolution or 8-bit resolution and measures

acceleration -2g to 2g where g is gravitational constant. The IC operates on 3.3V and has QFN package.

The accelerometer has multiple registers that can be written or read via I2C protocol. To access these registers the master first sends the slave address of the accelerometer, 0x1C, and then the address of desired register address. There are two types registers in the accelerometer that were used in RichArduino project: the data register and the control register. The data registers are 8 bit registers holding the acceleration readings. The x-acceleration readings are at 0x01 and it has 8-bit data for x-axis acceleration. When a read is performed from this address, the data register address automatically increments to the next address: 0x02, the y-acceleration data. 0x03 holds the z-acceleration data. This makes it simple to read the three desired acceleration magnitudes: we just have to read from address 0x01 and use the I2C's multiple byte read function to keep reading two more times.

The first control register has an address 0x2A. Each of the 8 bits in the register can turn on and off the certain modes of the accelerometer (register data structure is shown in figure 8.2). In our project, we left every mode to default, 0, and manipulated the two least significant bits which represents F_READ and active. The F_READ bit changes 12-bit data of each acceleration to 8-bit. The active bit is responsible for turning the accelerometer on and starting data acquisition. This allows our design to execute three reads instead of 6 reads on the I2C bus since the I2C can only transfer 8 bits of data at a time. The sequence of I2C communication with the accelerometer is shown in figure 8.3.

Using the I2C module in VHDL, we designed assembly code that first sets the control register of the accelerometer to the values we want, then repeatedly reads the x,y,z acceleration values, displays them, and creates a step count. The assembly code is shown in figure 8.4. The code first creates a graphical display of three lines for x,y, and z axis. The program then repeatedly reads the acceleration values and plots them along the lines as a special character. The character is displayed along the line as an offset of the beginning of the line based upon the read value. The graphical display is shown in figure 8.5.

Step count is a special function in y-acceleration. We first read y-acceleration and compare it to our threshold, 160 which is 4.5m/s^2, then set flag in r9 high if it is larger than the threshold. After that if the value of acceleration ever goes below the lower threshold, 96 which is -4.5m/s^2, we increment step count by 1 and set flag back to low.

9.) VHDL:

As described in our Architecture section, RichArduino has six modules that are on shared address and data bus: SRAM, EEPROM, RSRC, I2C, HDMI, and a DIO Manager. In this report we will discuss last three modules on the list since detailed description of SRAM, EEPROM, and RSRC were given in classes CSE362M and CSE462M by professor William D. Richard.

I2C

First, the basic I2C bus protocol. I2C bus protocol can run up to 400kHz in high-speed mode. It has two main bus connections named SDA and SCL, which are

held high by default. A SDA transports either data from the master or slave, depending on who has a control over the bus. A SCL is an internal clock for I2C bus communication.

The general read and write sequence in I2C protocol follows the pattern shown in figure 3.5. A start signal is a specific combination of SDA and SCL where host pulls SDA low before the falling edge of SCL clock. Slave address + command is sent in 8-bit data stream, 7-bit for address and 1-bit to indicate read or write. Then the slave of I2C communication pulls SDA low on next clock to indicate it received read or write command correctly. Then the either master or slave send 8-bit write or read data on SDA bus. Lastly, receiver of the data pulls SDA low again to indicate successful data transfer. Then the transaction ends with stop signal which is raising SDA at the falling edge of SCL. The detailed timing diagram is shown in figure 9.1.

The I2C module of our design implemented FSM to handle both read and write sequence as well as a special case called clock stretching. Clock stretching is added to support a I2C slave that requires slower clock cycle to process a I2C communication than the clock in SCL. The work of I2C module itself was mostly provided from a source code by Scott Larson on Digi-key website. The image of actual FSM can be found in figure 9.2.

The FSM first starts at Begin state, this is the Ideal state where it resets all the signal. Then in next clock cycle, it goes to the ready state. In this state, FSM waits to get enable signal (ena) to start the I2C transaction. When the I2C communication is initiated, it goes to the start state, which generates start sequence in SDA bus. Then it

goes to command state where it gets slave address and read or write command then send it to slave via I2C bus. Then it goes to slave acknowledge state to generate acknowledge error signal, ack_error, if the SDA channel wasn't pulled low. Then it goes to read or write state to send or receive 8-bit data then go to second acknowledge state to indicate rather slave or master received data successfully. If there is another I2C data transaction to be done, we directly branch back to start state, otherwise it goes to stop state which generates stop sequence and end I2C channel until new transaction is required in ready state.

The FSM has three signals that it uses to transit from state to state. First one is the enable signal (ena). The enable signal is responsible to indicate there is a I2C transaction request from the master. Bit count signal (bit_cnt) counts number of data bits from the I2C bus. I2C bus transfer data one bit at a time in serial. So, for read and write state (rd and wr) and command state it decrement bit_cnt from 7 to 0 to transfer full slave address and command in command state and read and write data in read and write state. Once FSM counts 0 and transmitted or received all 8 bit data, it branches to next state. The read or write signal (rw) simply indicates rather the I2C transaction was read or write. If it was a write, it puts data in data_wr register to SDA, If it is was read, it stores data from SDA to data_rd register.

The additional signal in I2C module is busy signal. In the middle of I2C data transaction, busy signal is set to high to indicate user there is transaction in process so don't input new I2C data request yet. The block diagram of I2C module in shown in figure 9.3.

To input slave address and command to I2C module, we wrapped I2C module with I2C_Top module that has registers to store slave addresses and commands that will be used in I2C module (block diagram shown in figure 9.4). These registers are i2c_ack_error, i2c_ena, i2c_addr, i2c_rw, i2c_data_wr, i2c_data_rd, and data.

The I2C_Top module consists of 5 stage FSM (shown in figure 9.5). The first stage is idle. In this stage it waits for I2C module to be enabled. The enable signal for I2C depends on the address on the shared address bus. It is "0000000000000001" with 1 is in position of the 17th bit. When the I2C_Top module is enabled it checks the “done” signal, which is set to 1 if there is any data from the previous I2C transaction that hasn’t been sent. If the done signal is high, it sets valid signal to high and send 1 to shared data bus and set done to low. When the I2C module is enabled again, idle state outputs the actual data from I2C transaction to shared data bus and set valid to low, indicating there is no data that hasn’t been sent.

In idle state branch to one of four states, USB_setup, USB, gyro_setup, and gyro. If the I2C module is enabled and the two least significant bits in shared address bus is 00, it branches to USB_setup. For 01 it branches to USB and for 10 it branches to gyro and for 11 it branches to gyro_setup.

For the four states we implemented special logic that counts busy signal from the I2C module (actual code shown in figure 9.6). In each state, we request multiple I2C transaction. When the one transaction is done, busy count incremented by 1 and it will go to slave address and command for the next transaction in the state.

For USB_setup, we have 2 transactions. The first transaction sets i2c_ena register to 1 and i2c_address to general call command and i2c_rw to 0 and i2c_data_wr to 00001100. This is a command sequence to read byte available in FIFO buffer in FT201x chip. Then the second transaction is initiated with i2c_address equals 0100010 and i2c_rw to 1 to read the actual data of bytes available of FIFO. This follows the byte available check command explained in our USB basics section.

For USB, we have 4 transaction. A simple I2C read sequence can transfer 8 bit of data at a time. Since the RichArduino uses USB connection to receive 32bit instructions for application program, we implemented four repeated reads and concatenate each reads to one 32-bit register called data. The concatenation followed big endian order. By doing this we could store complete 32-bit instruction to our sram. The each transaction has i2c_address set to 0100010 and i2c_rw and i2c_ena to 1. Then it saves streamed data to i2c_data_rd and each 8-bit data gets stored to data register from 32 bit down to 25 bit to 7 bit down to 0 bit.

For gyro_setup state we have two transaction. Like described in our shield demo section, it first sets i2c_address to slave address of accelerometer, 0011100, and i2c_rw to 0 and i2c_data_wr to 00101010. Then in second transaction we put same address and i2c_data_wr to 00000011 to turn on the accelerometer to active state and set it to a fast read mode.

Lastly in gyro state, similar to USB state, it repeats read command three times and concatenate 8-bit data to data register. The first 8-bit is the x-axis acceleration value, the second is y-axis and the last 8-bit is the z-axis acceleration.

For both setup states, the ack_error signal from I2C module gets stored to i2c_ack_error register and store it to the second least significant bit of data register. So, after setup state, if the data is read “2”, the program can repeat the setup state again.

DVI

The DVI module’s core function has been explained at a high level in section 2. To implement this in VHDL, a few important changes had to be made to the translated VHDL provided by Dr. Richard. First, a DCM is used to generate the 25MHz pixel clock (“pixclk”) and the 250 MHz clock used for encoding the pixel data. The pixclk signal has to be driven out of the FPGA over differential output pins, however the global clock nets don’t have access to the OBUFDS drivers that create the differential output signals. To do this, a mirror of the clock signal has to be created on a non-global clock net using an ODDR2.

DIO Manager

The DIO manager allows the RSRC to interact with and use 20 digital Digital IO on the RichArduino Board. The base address for the Manager is 0x8000. The lower 5 bits of the address are used to address each IO pin as an offset from this base(i.e. DIO_5 is at 0x8005). The DIO Manager has a 20 bit status at address 0x801Fregister for the current status of each pin as either an input(1) or output(0). The status of each pin is held at the bit position of that IO pin (i.e. config(8) holds the position for IO pin 8). A write to this register will capture the lowest 20 bits of the data bus, reconfiguring all IO pins at once. An input pin will place the current value of the pin as the least significant bit on the bus and drive all the upper bits to zero. Output pins will continuously output

the least significant bit of the value written to them, 0 on reset. A user can read from an output configured pin, but it is considered a software error. The read will actually contain the last value written to that output pin as will as raise a flag in the second least significant position.

10.) Timing/Results:

See figure 8.5 and 9.1 for monitor output and the I2C timing diagram. The result of actual I2C module simulation can be found in figure 10.1. In the implementation of our FPGA design all timing constraints were met as shown by the timing report in figure 10.2.

In testing, we were able to successfully download programs from the GUI to run on the RichArduino. This success is proven by running various test programs uploaded to the board by the GUI. These test programs include flashing an LED based on a button input, a graphical display of the accelerometer readings, and a step counter.

11.) Conclusions:

In summary, the red team implementation of the RichArduno meets all specifications set out by the Red Team system requirements for the CSE462M SP19 project. The RichArduino acts as analogous Arduino Board, whereby we are able to both download and run programs sourced from a GUI onto a microcontroller. In addition to this basic Arduino paradigm, we incorporated both a DVI display output and a dedicated, hardware implemented, I^2C Bus. There is potential in the RichArduino platform for future expansions. Possible areas of expansion are support for analog

signaling, stack protocols, level translators(3.3V:5V) to interface with 5V IO, and full HDMI implementation.

12.) References:

12.1) Sources

[1] W. D. Richard, “The_RSRC_SRC_Instruction_Set.pdf.” .

[2] T. Downey, “Two-pass assemblers.” [Online]. Available: http://users.cis.fiu.edu/~downeyt/cop3402/two-pass.htm. [Accessed: 12-Feb-2019].

[3] C. B. Park and U. Kingdom, “D2XX_Programmer’s_Guide(FT_000071),” vol. 44, no. 0, pp. 1–103, 2011.

[4] The Qt Comapny, “Qt Documentation - Qt 5.12.” [Online]. Available: https://doc.qt.io/qt-5/. [Accessed: 12-Apr-2019].

[5] W. D. Richard, “The Really Simple RISC Computer (RSRC) V0.806”

[6] *HDMI*. [Online]. Available: https://www.fpga4fun.com/HDMI.html. [Accessed: 07-May-2019].

12.2) Images

Figure 2.1) DVI Module Control Signals

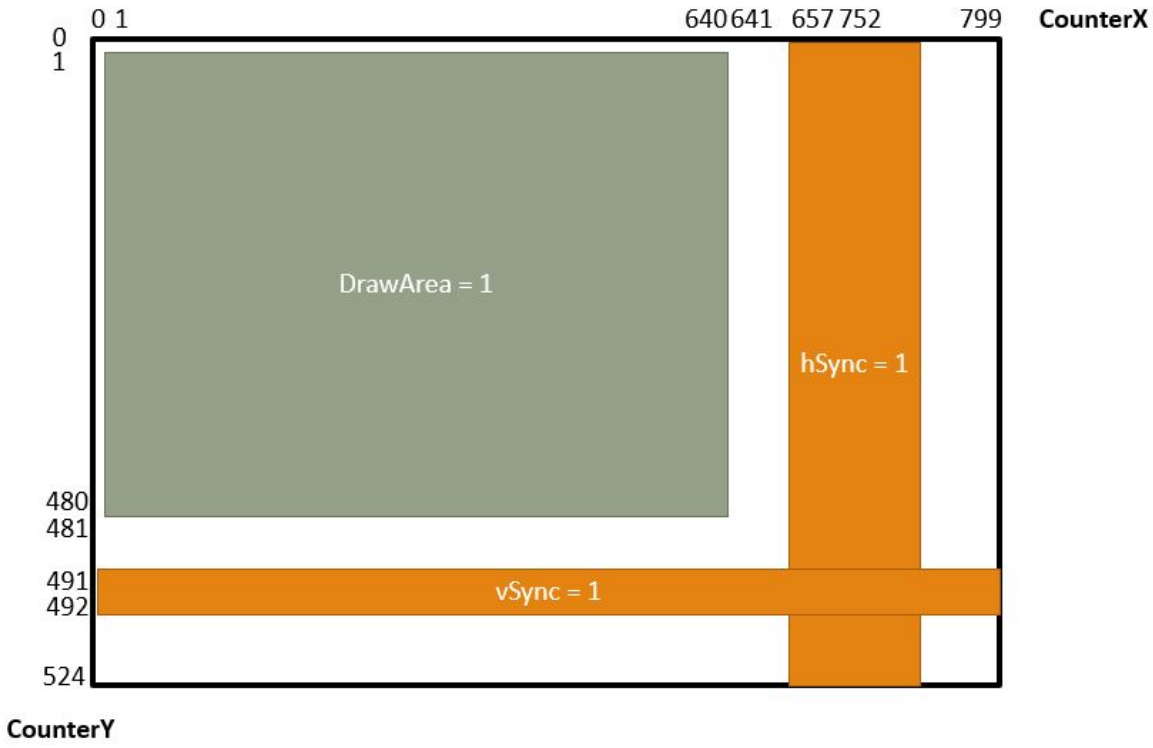


Figure 3.1) USB B-type header

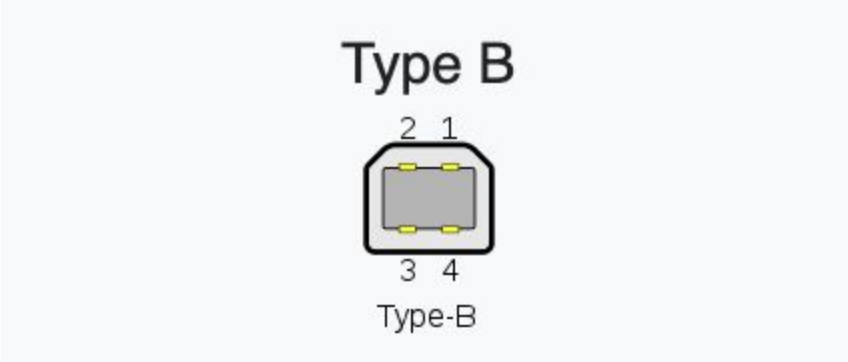


Figure 3.2) FT201x chip's bus powered configuration

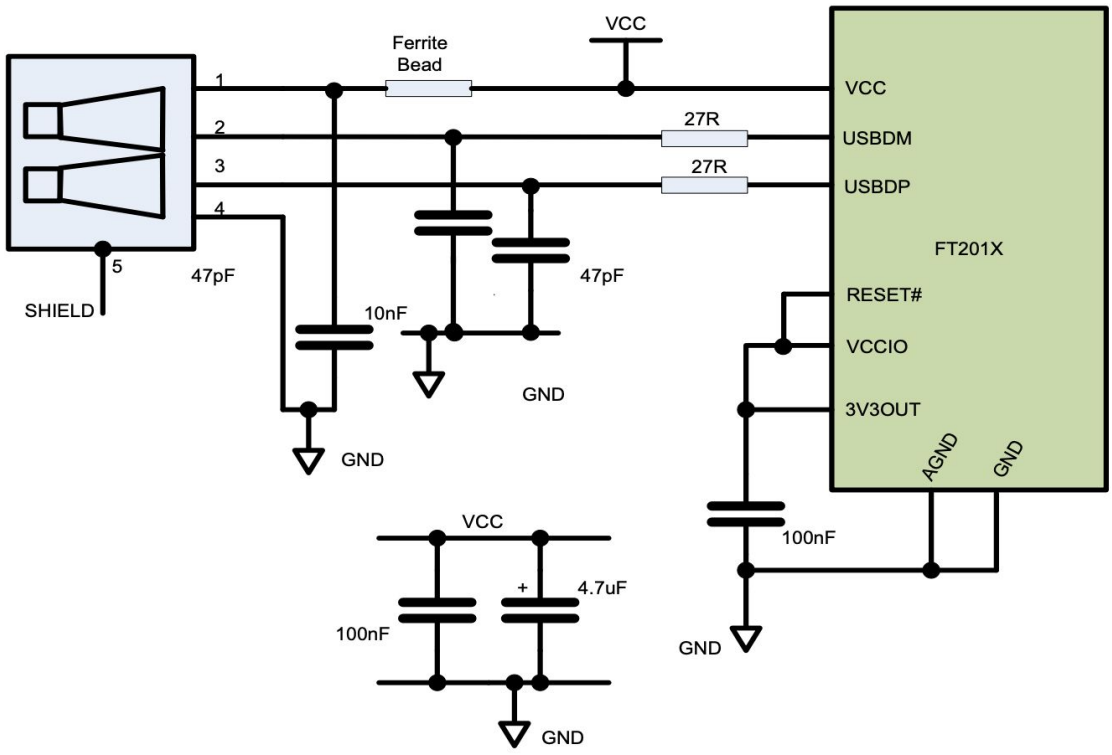


Figure 3.3) Data transfer block diagram from USB host to FT201x chip to SRAM of FPGA.

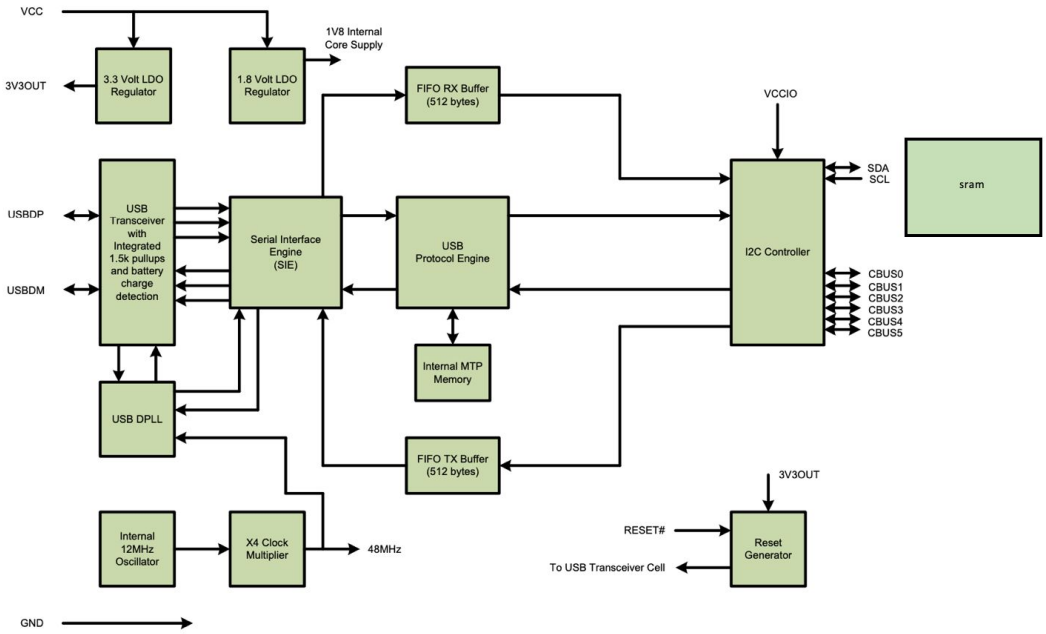


Figure 3.4) FT201x chip layout (SSOP package)

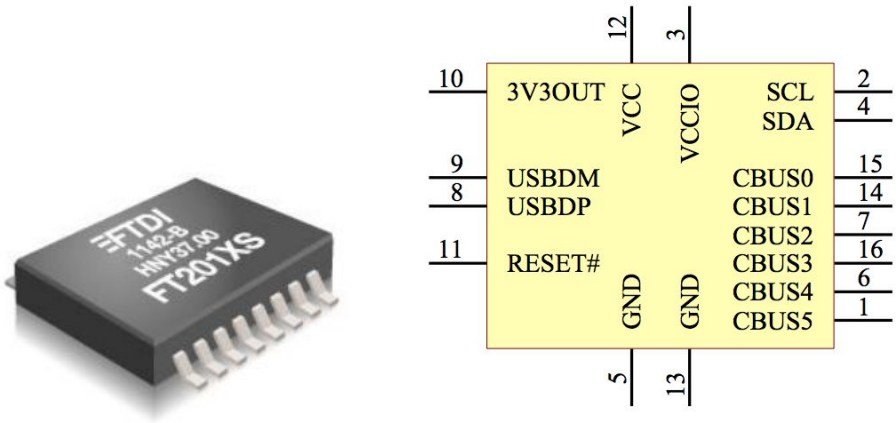


Figure 3.5) FT201x chip timing graph for I2C read and write and data available check on FIFO buffer.

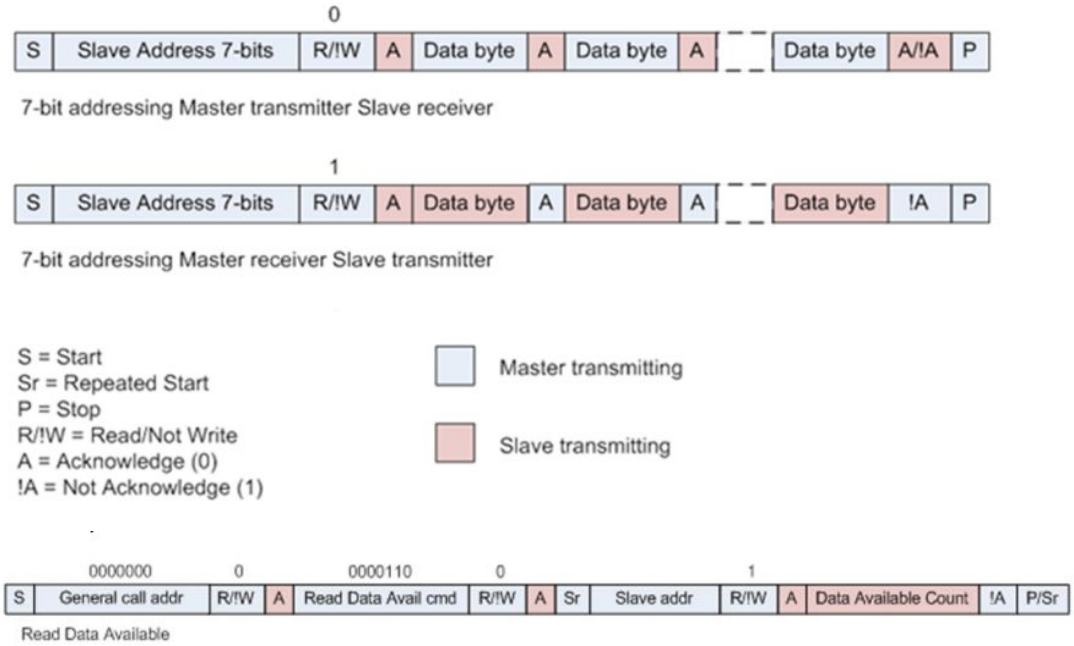


Figure 4.1) RichArduino Platform Design

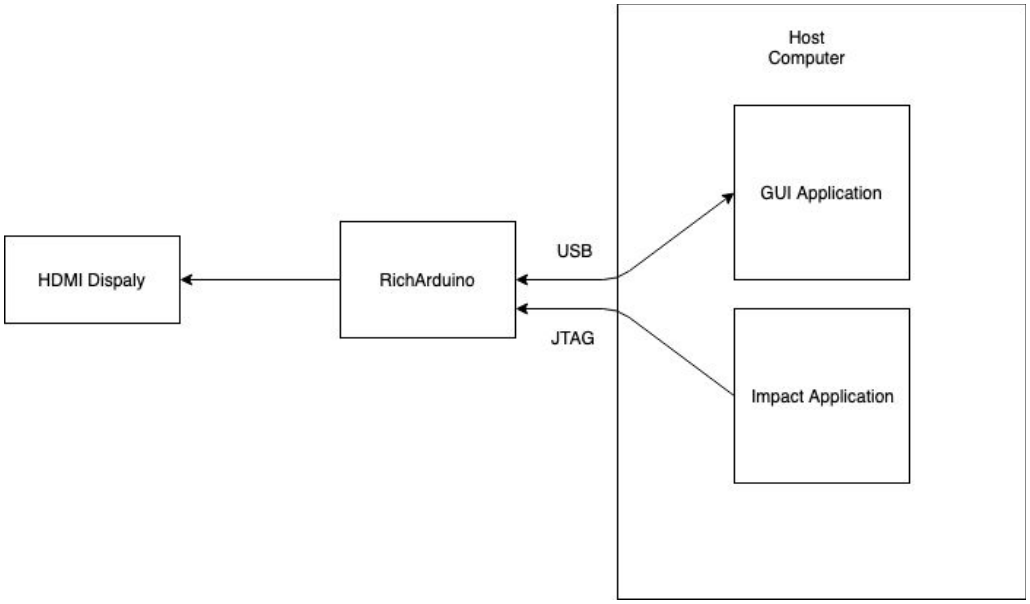


Figure 4.2) RichArduino Board Layout. Capacitors, Resistors, and Ferrite Beads are excluded from this board layout.

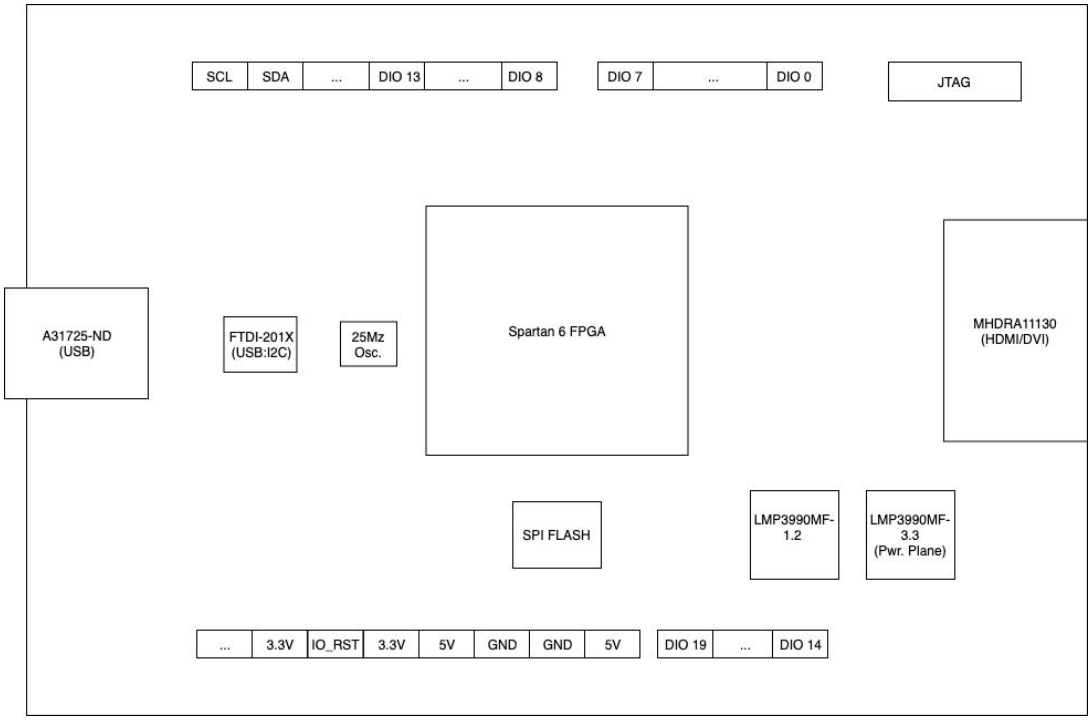
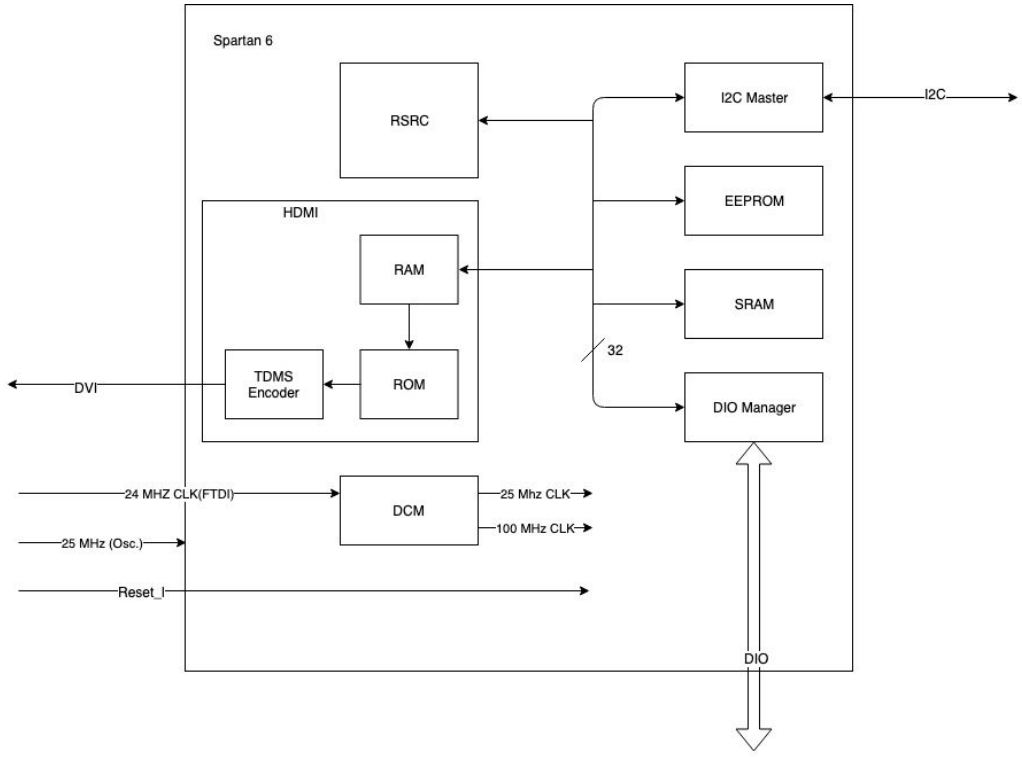


Figure 4.3) RichArduino Internal Architecture. This architecture includes the 6 main modules (HDMI, RSRC, I2C Master, EEPROM, SRAM, DIO manger) that all share a 32 bit address and data bus. All clocks are routed through global clock pins on the FPGA. The I2C bus is shared bus with the I2C bus on the RichArduino header pins. This can be split into two separate buses by removing the 0 Ohm resistors R27 & R28



Notes

The I2C bus is a shared bus with the FTDI 201X chip and I2C header pins.

Figure 4.4) RichArduino Address Spaces. Hard lines indicate the allocated memory space of a given module through an address decode. Dashed lines indicate the upper limit of addressable address space.

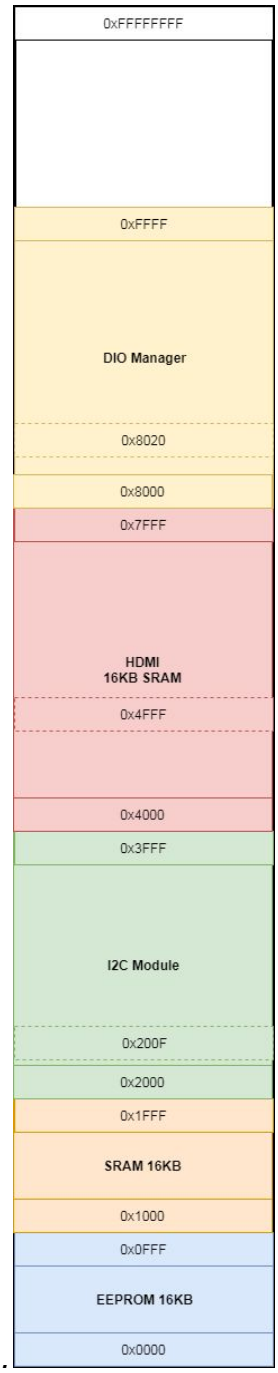


Figure 6.1) Source code for monitor program

```
la r31, LOOP          //Loop address
la r7, LOOP2          //Loop2 address
la r25, LOOP3         //Loop3 for header checking
la r8, 3              //4byte count
la r17, 0             //program count
la r14, 0             //initialize header valid to 0
la r18, 0             //initialize header data to 0
la r30, 1
shl r30, r30, 13      //I2C address
la r1, 0              //register to store compared values
la r26, 1
shl r26, r26, 12      //Sram address that don't change
addi r26, r26, 4
la r27, 1
shl r27, r27, 12      //Sram address that change
LOOP: ld r1,(r30)     //load valid bit
andi r2, r1, 1        //check valid bit
brzr r31, r2
ld r1, (r30)          //read byte count
sub r2, r8, r1        //compare to 3
brpl r31, r2          //branch if >= 3
LOOP2: ld r1, 1(r30)  //read valid bit
andi r2, r1, 1        //check valid bit
brzr r7, r2
ld r1, (r30)          //read data
addi  r18, r1, 0      //store data to r18
st r1, (r27)          //store to Sram
addi r27, r27, 4
brzr r25, r14         //branch if no valid header
sub r17, r17, r14     //subtract 1 from program counter
brnz r31,r17              //keep reading if pc is not zero
br r26                //go to sram if pc is zero
LOOP3: la r14, 1      //valid header exist
addi r17, r18, 0          //loading valid header to r17
br r31                //branch back to beginning to read actual program
stop
```

Figure 8.1) Address for data register of accelerometer. It has two different addressing scheme based on F_READ mode.

Name	Type	Register Address	Auto-Increment Address		Default	Hex Value	Comment
			F_READ=0	F_READ=1			
STATUS(1)(2)	R	0x00	0x01		00000000	0x00	Real time status
OUT_X_MSB(1)(2)	R	0x01	0x02	0x03	Output	—	[7:0] are 8 MSBs of 12-bit sample.
OUT_X_LSB(1)(2)	R	0x02	0x03	0x00	Output	—	[7:4] are 4 LSBs of 12-bit sample.
OUT_Y_MSB(1)(2)	R	0x03	0x04	0x05	Output	—	[7:0] are 8 MSBs of 12-bit sample.
OUT_Y_LSB(1)(2)	R	0x04	0x05	0x00	Output	—	[7:4] are 4 LSBs of 12-bit sample.
OUT_Z_MSB(1)(2)	R	0x05	0x06	0x00	Output	—	[7:0] are 8 MSBs of 12-bit sample.
OUT_Z_LSB(1)(2)	R	0x06	0x00		Output	—	[7:4] are 4 LSBs of 12-bit sample.

Figure 8.2) Configuration of 8-bit control register 1 of accelerometer that is used to set F_READ and active in our project.

0x2A: CTRL_REG1 Register (Read/Write)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ASLP_RATE1	ASLP_RATE0	DR2	DR1	DR0	LNOISE	F_READ	ACTIVE

Figure 8.3) I2C timing graph of read and write sequence for accelerometer

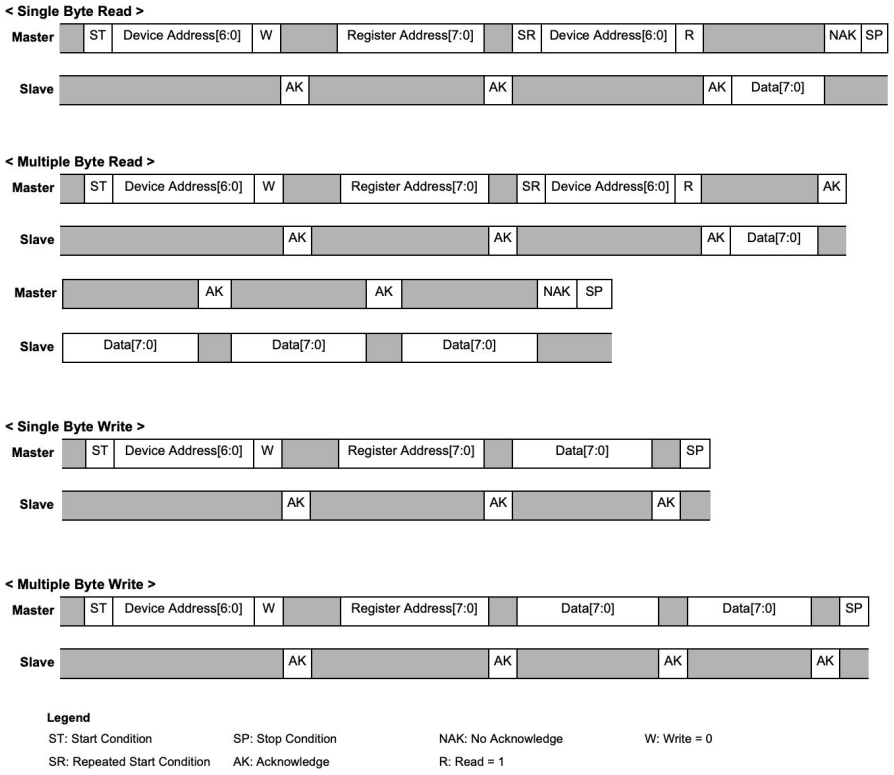


Figure 8.4) Source code for accelerometer-demo application program.

```
.org 4100
la r1, 0
la r5, 0
la r31, 1
shl r31, r31, 14        //DVI address
la r30, LOOP
la r28, LOOP2
la r27, LOOP3
la r26, 90
la r2, 1
la r9, (r31)
addi r9, r9, 15164
LOOP: st r1,(r31)       // blank the screen
addi r31, r31, 4
sub r10, r9, r31
brpl r30, r10
la r1, 88           // begin drawing title and axes
la r30, 1
shl r30, r30, 14        // DVI address
LOOP3: addi r31, r30, 0
addi r31, r31, 2560     // down five lines
addi r31, r31, 24       // over 6 chars
st r1, (r31)
addi r31, r31, 8
addi r9, r31, 256
LOOP2: st r2, (r31)
addi, r31, r31, 4
sub r8, r9, r31
brnz r28, r8
addi r1, r1, 1
sub r8, r26, r1
addi r30, r30, 2560
brpl r27, r8
la r31, LOOP4
la r7, LOOP5
la r17, NOTLESS
la r21, NOTMORE
la r23, 48   //step count
la r30, 1
la r1, 0
la r24, 8
shl r30, r30, 13

la r8, 4
la r29, 0
la r28, 1
la r13, 103 //g
la r14, 48 //0
la r15, 45 //-
la r16, 50 //2
shl r28, r28, 14
addi r25, r28, 1024     // down two lines
addi r25, r25, 104      // correct offset to start drawing title
la r22, 34      //draw "
st r22, (r25)
addi r25, r25, 4
la r22, 51      //draw 3
st r22, (r25)
addi r25, r25, 4
la r22, 68      //draw D
st r22, (r25)
addi r25, r25, 4
la r22, 34      //draw "
st r22, (r25)
addi r25, r25, 4
la r22, 32      //draw space
st r22, (r25)
addi r25, r25, 4
la r22, 65      //draw A
st r22, (r25)
addi r25, r25, 4
la r22, 99      //draw c
st r22, (r25)
addi r25, r25, 4
la r22, 99      //draw c
st r22, (r25)
addi r25, r25, 4
la r22, 101     //draw e
st r22, (r25)
addi r25, r25, 4
la r22, 108     //draw l
st r22, (r25)
addi r25, r25, 4
```

```
la r22, 101     //draw e
st r22, (r25)
addi r25, r25, 4
la r22, 114     //draw r
st r22, (r25)
addi r25, r25, 4
la r22, 97      //draw a
st r22, (r25)
addi r25, r25, 4
la r22, 116     //draw t
st r22, (r25)
addi r25, r25, 4
la r22, 105     //draw i
st r22, (r25)
addi r25, r25, 4
la r22, 111     //draw o
st r22, (r25)
addi r25, r25, 4
la r22, 110     //draw n
st r22, (r25)
addi r25, r25, 4
la r22, 32      //draw space
st r22, (r25)
addi r25, r25, 4
la r22, 77      //draw M
st r22, (r25)
addi r25, r25, 4
la r22, 97      //draw a
st r22, (r25)
addi r25, r25, 4
la r22, 103     //draw g
st r22, (r25)
addi r25, r25, 4
la r22, 110     //draw n
st r22, (r25)
addi r25, r25, 4
la r22, 105     //draw i
st r22, (r25)
addi r25, r25, 4

la r22, 116     //draw t
st r22, (r25)
addi r25, r25, 4
la r22, 117     //draw u
st r22, (r25)
addi r25, r25, 4
la r22, 100     //draw d
st r22, (r25)
addi r25, r25, 4
la r22, 101     //draw e
st r22, (r25)
addi r28, r28, 2592     //  beginning of x axis
addi r27, r28, 2560     //  beginning of y axis
addi r26, r27, 2560     //  beginning of z axis
la r3, 0                //  previous z offset
la r4, 0                //  previous y offset
la r5, 0                //  previous x offset
la r11, 1 //pipe
la r12, 2 //dot
st r14, 636(r26) // draw 0
st r14, 636(r28)
st r14, 636(r27)
st r13, 516(r26) // draw g for -2g
st r13, 516(r28)
st r13, 516(r27)
st r13, 580(r26) // draw g for -g
st r13, 580(r28)
st r13, 580(r27)
st r15, 508(r26) // draw - for -2g
st r15, 508(r28)
st r15, 508(r27)
st r15, 576(r26) // draw - for -g
st r15, 576(r28)
st r15, 576(r27)
st r13, 700(r26) // draw g for g
st r13, 700(r28)
st r13, 700(r27)
st r13, 764(r26) // draw g for 2g
st r13, 764(r28)
st r13, 764(r27)
st r16, 760(r26) // draw 2 for 2g
```

```
st r16, 760(r28)
st r16, 760(r27)
st r16, 512(r26) // draw 2 for -2g
st r16, 512(r28)
st r16, 512(r27)
la r25, 1               //  begin writing: Steps:
shl r25, r25, 14
addi r25, r25, 12948    //  put Steps: 25 lines down and in the middle
la r16, 83
st r16, (r25)           //  store S
la r16, 116
addi r25, r25, 4
st r16, (r25)           //  store t
la r16, 101
addi r25, r25, 4
st r16, (r25)           //  store e
la r16, 112
addi r25, r25, 4
st r16, (r25)           //  store p
la r16, 115
addi r25, r25, 4
st r16, (r25)           //  store s
la r16, 58
addi r25, r25, 4
st r16, (r25)           //  store :
la r16, 32
addi r25, r25, 4
st r16, (r25)           //  store space
addi r25, r25, 4        //  move the step location pointer over a character
la r20, 150             //  acceleration threshold to start watching for a step
la r22, 70          //  acceleration threshold to count a step if we were positive
la r10, 50
la r18, 1
shl r18, r18, 15        //  dio manager address
la r13, DONTRESETSTEPS
LOOP4: ld r1, 3(r30)    //  setup accel loop
andi r2, r1, 1          //  check if setup is done
brzr r31, r2            //  r31 holds loop4
ld r1, (r30)            //  ???
andi r2, r1, 2          //  ???
```

```
brnz r31, r2            //  ???
la r9, 0                //  r9 is used to indicate we have seen a y-accel value greater than threshold
LOOP5: ld r1, 2(r30)    //  loop for storing values to screen
andi r2, r1, 1          //
brzr r7, r2             //  r7 holds loop5
ld r1, (r30)            //  r1 holds the accel data
st r11,(r26)            //  z loop -- store a pipe to the previous z loc
sub r26,r26,r3          //  get rid of the previous z offset
shr r1, r1, 8           //  shifts the z data into bottom eight digits of r1
andi r3, r1, 255        //  r3 holds just the z data
addi r3, r3, 128        //  add a 1 to the eight bit -- negative values now have magnitude 0-127 and positive values have 128-255
andi r3, r3, 255        //  mask off overflow
add r26,r26,r3          //  add this value (0-255) to the offset for the z axis -- creates the new z loc
st r12,(r26)            //  store a pipe w/ dot to the new z loc
st r11, (r27)           //  y loop -- store a pipe to the previous y loc
sub r27, r27, r4        //  get rid of the previous y offset
shr r1, r1, 8           //  shifts the y data into bottom eight digits of r1
andi r4, r1, 255        //  r4 holds just the y data
addi r4, r4, 128        //  add a 1 to the eight bit -- negative values now have magnitude 0-127 and positive values have 128-255
andi r4, r4, 255        //  mask off overflow
add r27, r27, r4        //  add this value (0-255) to the offset for the y axis -- creates the new y loc
st r12, (r27)           //  store a pipe w/ dot to the new y loc
sub r19, r20, r4        //  if y accel is not greater than the positive threshold
brpl r21, r19           //  branch to NOTMORE
la r9, 1                //  load r9 with 1 to indicate we have seen a value higher than 1
NOTMORE: sub r19, r4, r22       //  r19 gets negative threshold minus value
brpl r17, r19           //  the value is not lower than negative threshold
add r23, r23, r9        //  the value is less than the negative threshold so add the value of r9 to the step counter
la r9, 0                //  the value has been less than the threshold so load the "seenPositive register to zero"
NOTLESS: ld r15, (r18)  //  r15 holds the value of the voltage below the button.. if 1, need to reset step count
brzr r13, r15           //  test the value of the voltage below the button
la r23, 48              //  value of voltage is 1, so reset the step counter to ASCII zero
la r9, 0                //  also reset the "have seen a high"
DONTRESETSTEPS: st r23, (r25)   //  write value of step count to the DVI slot
st r11, (r28)           //  x loop
sub r28, r28, r5
shr r1, r1, 8
andi r5, r1, 255
addi r5, r5, 128
andi r5, r5, 255
add r28, r28, r5
st r12, (r28)
br r7                   //  branch to LOOP5
stop
```

Figure 8.5) The monitor output of the accelerometer application program.

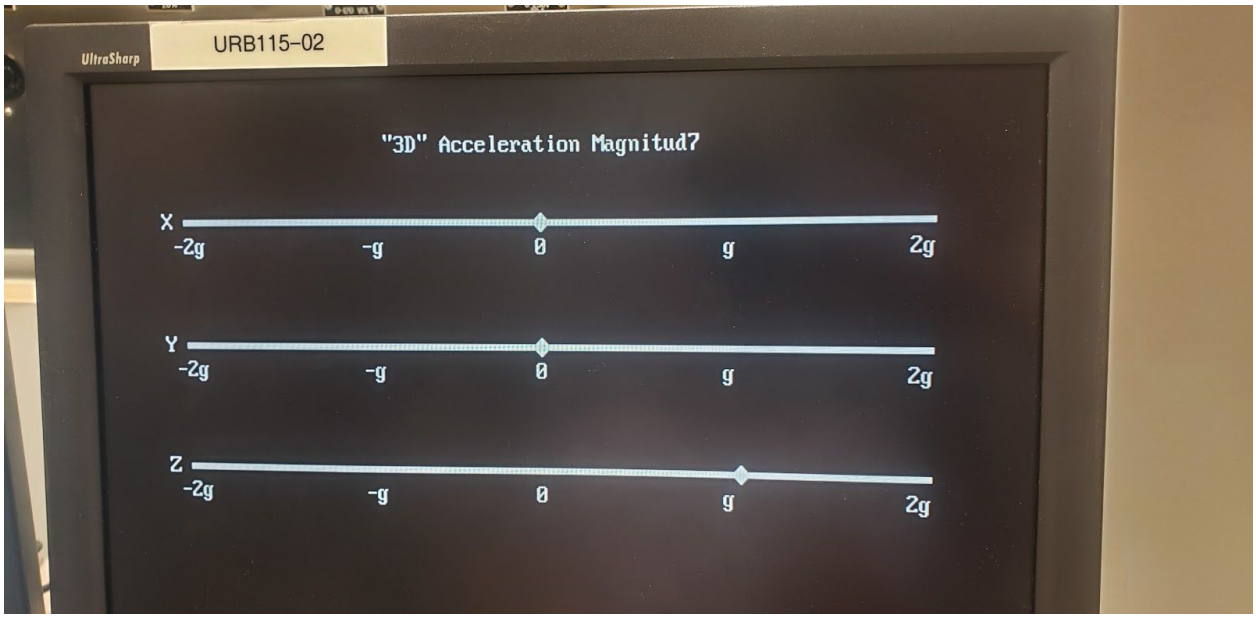


Figure 9.1) Timing diagram of I2C bus protocol for read or write.

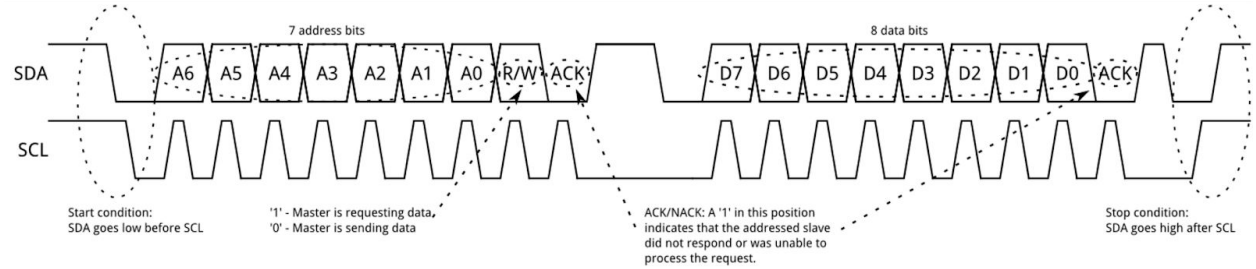


Figure 9.2) I2C module FSM

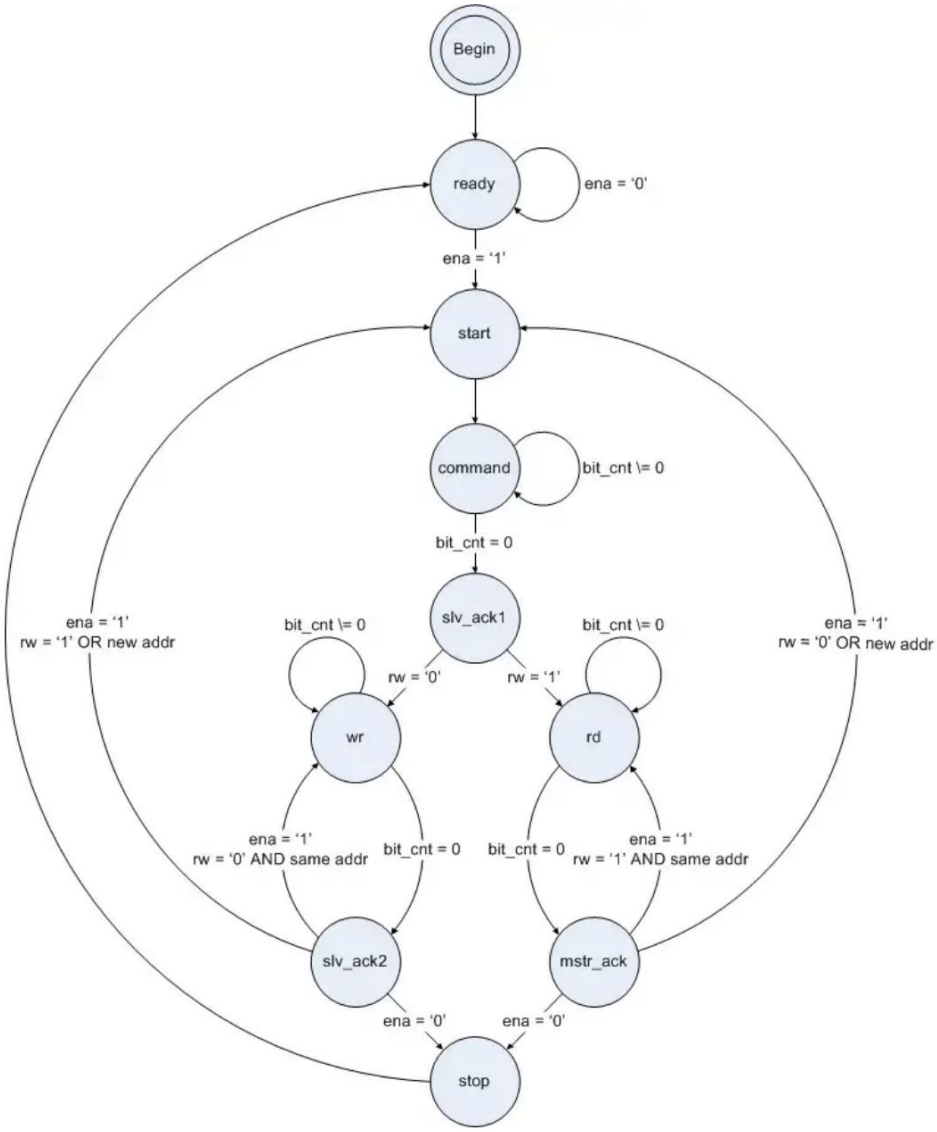


Figure 9.3) Block diagram of I2C module.

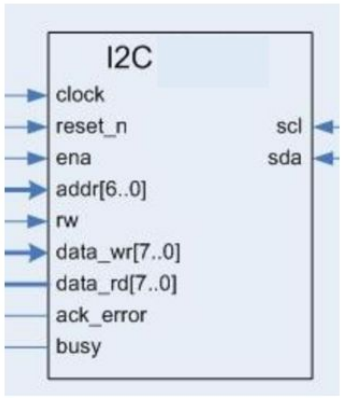


Figure 9.4) Block diagram of I2C top module connected with I2C module.

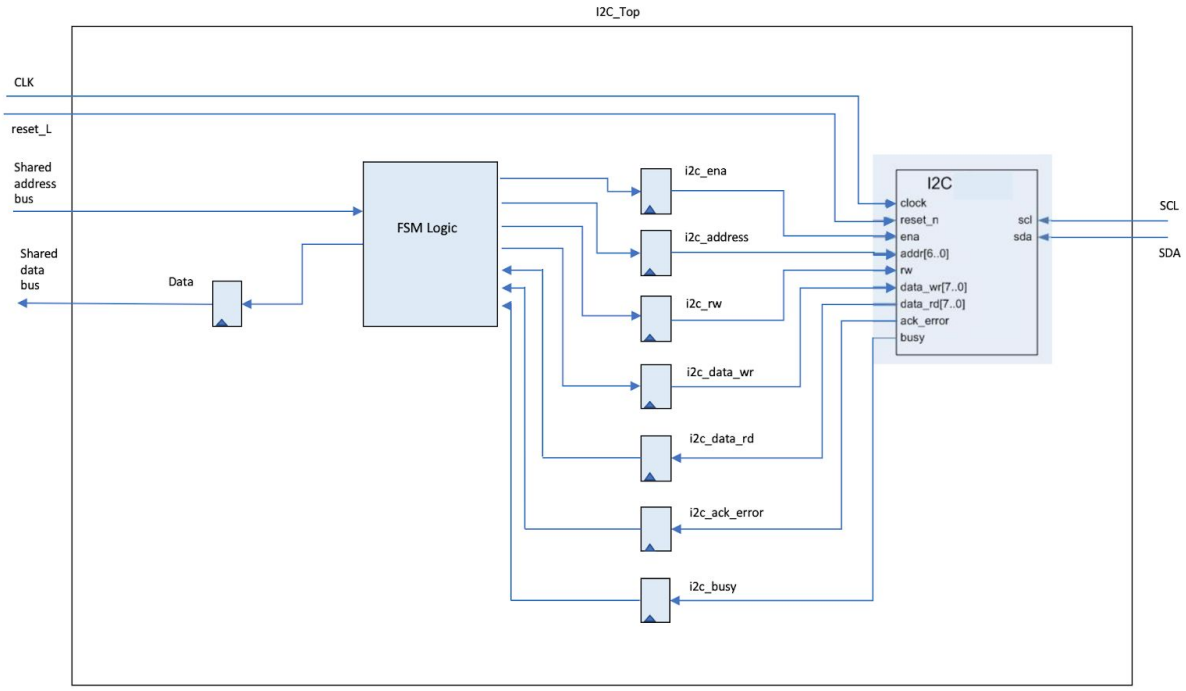


Figure 9.5) FSM for I2C top module

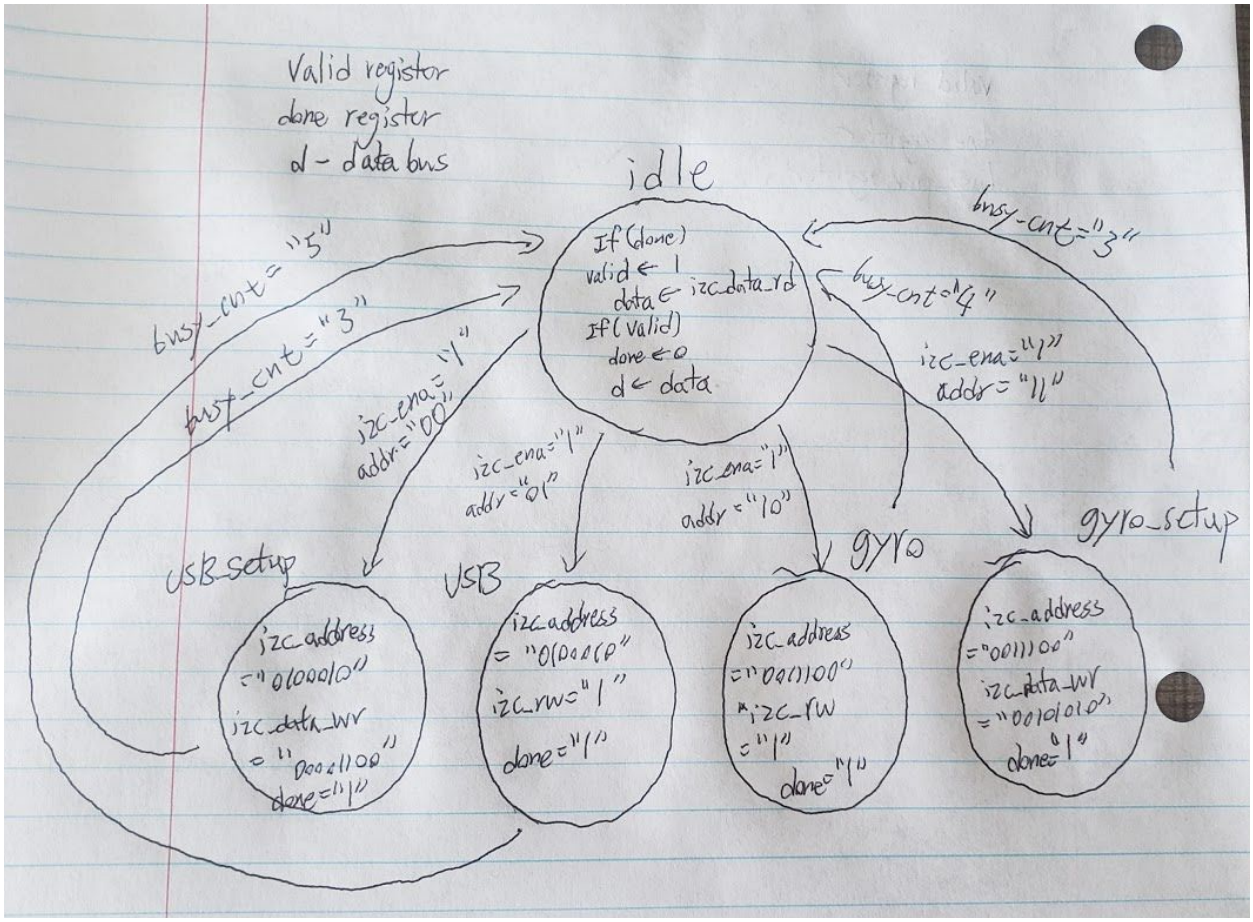


Figure 9.6) Code snippet of the busy count logic used in I2C top module to create control of I2C transaction for USB_setup, USB, gyro_setup, and gyro.

```
busy_prev <= i2c_busy;
IF(busy_prev = '0' AND i2c_busy = '1') THEN
  busy_cnt <= busy_cnt + 1;
END IF;
```

Figure 10.1 I2C running module in simulation. SDA goes low before SCL indicating the start of transaction on the bus. SCL and SDA are put at high impedance because the I2C is a pull up bus.

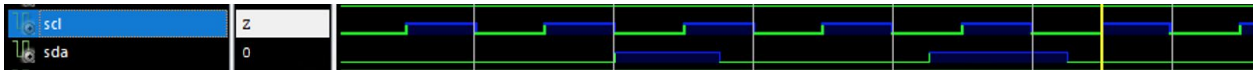


Figure 10.2 Timing report of RichArduino Implementation on the Spartan6 FPGA. The report shows a positive slack for all modules, proving that our design met timing.

Asterisk (*) preceding a constraint indicates it was not met.
This may be due to a setup or hold violation.

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
TS_mydcm1_clkout1 = PERIOD TIMEGRP "mydcm1_clkout1" TS_cbus_0 / 4.16666667 HIGH 50%	SETUP	0.498ns	9.501ns	0	0
	HOLD	0.033ns		0	0
TS_Inst_HDMI_test_mydcm_2_inst_clkout2 = PERIOD TIMEGRP "Inst_HDMI_test_mydcm_2_inst_clkout2" TS_mydcm1_clkout1 / 2 .5 HIGH 50%	SETUP	1.113ns	2.886ns	0	0
	HOLD	0.000ns		0	0
TS_cbus_0 = PERIOD TIMEGRP "cbus_0" 41.66 6 ns HIGH 50%	MINLOWPULSE	21.666ns	20.000ns	0	0
TS_mydcm1_clkout0 = PERIOD TIMEGRP "mydcm1_clkout0" TS_cbus_0 / 1.04166667 HIGH 50%	SETUP	18.439ns	21.560ns	0	0
	HOLD	0.391ns		0	0
TS_Inst_HDMI_test_mydcm_2_inst_clkout0 = PERIOD TIMEGRP "Inst_HDMI_test_mydcm_2_inst_clkout0" TS_mydcm1_clkout1 / 0 .25 HIGH 50%	SETUP	35.262ns	4.737ns	0	0
	HOLD	0.391ns		0	0
TS_Inst_HDMI_test_mydcm_2_inst_clkout1 = PERIOD TIMEGRP "Inst_HDMI_test_mydcm_2_inst_clkout1" TS_mydcm1_clkout1 / 0 .25 PHASE 19.99968 ns HIGH 50%	MINPERIOD	38.269ns	1.730ns	0	0

Figure 11.1 RichArduino Companion GUI. A) Assembly file path field; B) Programming field; C) Activity Output Log; The "Reset" button performs a soft reset for the RichArduino. The "Reconnect" button re-establishes a connection with the RichArduino board. The "Upload" button saves and uploads the current assembly code in the programming field. The "Open" button opens the file path described in the assembly file path field. The "Browse and Open" button opens file manager to choose an assembly file to open. The "Save .asm" button saves changes to the currently open assembly file. The "Save As .asm" button saves assembly code to a new file. The "Save .bin" button saves the assembled machine code of the currently assembly code to a new file.