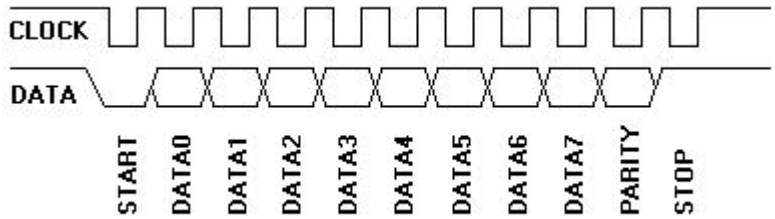
PS-2 Mouse:

The Protocol:

For out mini project we designed a serial port transmitter receiver, which uses the Baud rate protocol. The PS-2 port is similar to the serial port (performs the function of transmitting and receiving) except for the fact that instead of using baud rates the port uses an in-out clock to transmit and receive data. Data is sent and received synchronous to this clock. For the PS-2 protocol the data packets that are received and sent need to be of length 11 bits. Unlike the serial port protocol we have a bit assigned for the parity of the data byte.

Below is a diagram explaining the data packet and the timing related to the mouse clock while receiving a byte from the PS-2 mouse:



On mouse moves the mouse sends 3 bytes of information through the ps-2 port. The contents of the data packet is described below in the diagram:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X Movement							
Byte 3	Y Movement							

Byte 1 contains information on the direction the mouse moved wrt its previous position, status of the mouse buttons (0 for not pressed and 1 for pressed) and the x and y overflows which are set if the counters go past 255. Byte 2 and 3 contain information regarding the x movement and the y movement. The x movement and y movement are calculated based on the counter values in the mouse. The counters get incremented by a value of 4 for a movement of 1mm on the screen. Communicating with the mouse accordingly can change this resolution. The PS-2 mouse also allows scaling the value of the mouse movements. By default the scaling factor is set to1.

The PS-2 mouse on power on sends two bytes of data AA and 00. The AA signifies the mouse passed its self-test and the 00 stands for the mouse ID. Once this information is received and checked the driver issues a mouse reset and waits for acknowledgement (FA). After this the mouse variables such as scaling factor, resolution etc are set by transmitting the appropriate data to the mouse. After each transmission to the mouse the driver makes sure to check that acknowledgement is received if not it performs the action again. Once all the variables are set the enable signal is sent to the mouse asking it to start recording the movements of the mouse.

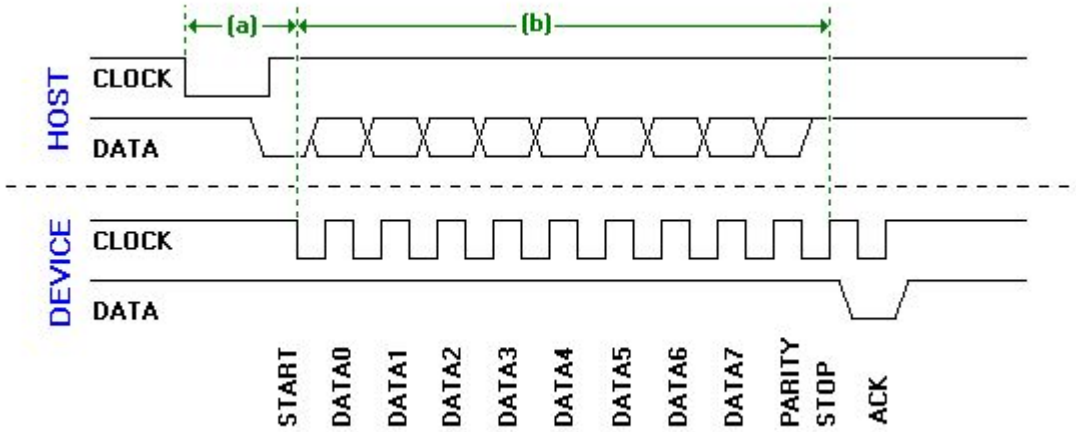
Jerry Mouse:

The design of the PS-2 interface was broken down into small individual modules. The main components were the transmitter, receiver and the control unit.

Transmitter:

The transmitter modules main function was to transmit data provided to it on the tx_data line serially to the mouse. There are specific timing guidelines that need to be followed while communicating with the PS-2 mouse. The process and timing for sending a byte of data to the mouse is outlined below:

1) Bring the **Clock** line low for at least 100 microseconds.
2) Bring the **Data** line low.
3) Release the **Clock** line.
4) Wait for the device to bring the **Clock** line low.
5) Set/reset the **Data** line to send the first data bit
6) Wait for the device to bring **Clock** high.
7) Wait for the device to bring **Clock** low.
8) Repeat steps 5-7 for the other seven data bits and the parity bit
9) Release the **Data** line.
10) Wait for the device to bring **Data** low.
11) Wait for the device to bring **Clock** low.
12) Wait for the device to release **Data** and **Clock**



(a) The time it takes the device to begin generating clock pulses after the host initially takes the **Clock** line low, must be no greater than 15ms;
(b) The time it takes for the packet to be sent must be no greater than 2ms.

Other than sending the data as outlined above the transmitter also needed to make sure that when its trying to transmit data the mouse should not be sending any data. This is made sure in our code by checking that that the clock is held high for enough time without a transition.

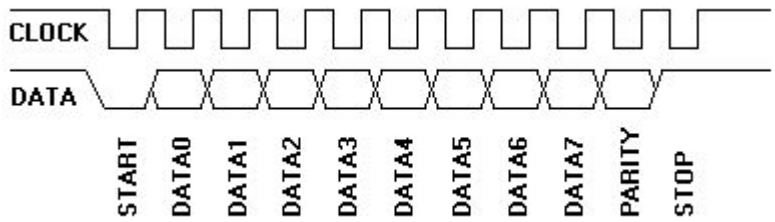
Note in the above figure after the transmitter transmits all the 11 bits the mouse sends an acknowledgement on the next falling edge of clock.

Functional Overview:
To transmit a data byte to the mouse the control unit provides the data through the tx_data line and provides the load signal. The transmitter on noticing the load waits for the m_clk to attain a stable state of 1 and then transmits the 11 bits of data serially. It finally checks to make sure that an acknowledgement is received. When the transmitter is transmitting a byte of data it sets the “active” signal to high letting the reciever know that is should neglect the activities on the m_clk and m_data line till the active signal is low again.

Receiver:

The receiver performs the function of receiving data serially from the mouse and transmitting it to the control unit as a byte of data. The basic process and timing that the mouse uses while sending a data is outline below:

1) Waits for **Clock** = high.
2) Delays 50 microseconds.
3) Clock still = high?
   No—go to step 1
4) Data = high?
   No--Abort (and read byte from host)
5) Delays 20 microseconds (=40 microseconds to the time **Clock** is pulled low in sending the start bit.)
6) Outputs Start bit (0) \ After *sending each of these bits, test*
7) Outputs 8 data bits > ***Clock*** *to make sure host hasn't pulled it*
8) Outputs Parity bit / low *(which would abort this transmission.)*
9) Outputs Stop bit (1)
10) Delays 30 microseconds (=50 microseconds from the time **Clock** is released in sending the stop bit)



To stick with the above protocol our receiver monitors the data line continuously checking for a start bit. If a start bit is detected then it starts collecting the serial bits received on falling edges of the m_clk and saving them in a shift register. Once the 11-bit packet has been received it puts the data on the rx_data line and sends a signal to the control unit informing that a fresh packet has been received. It ignores any new data packets till the control unit reads rx_data. Since the transmitter also uses the m_data line to transmit data the receiver checks the active signal is low before it starts receiving data.

Functional Overview:

The receiver receives a packet of data serially from the mouse. It checks the parity and stop bit to make sure the packet received is right. If they are wrong it aborts the receiving operation. Once it receives a data packet it places the data byte on the rx_data line and sets the rda signal to high. It ignores any new packet received from the mouse till the data has been read.

Control unit:

The control unit acts as the brain controlling the receiver and transmitter accordingly. On reset the control unit send a reset signal to the mouse controller and waits for an acknowledgement and mouse id. After it receives them it transmits the enable signal to the mouse and waits for acknowledgement. Once this is received the mouse has been reset and enabled in the sense it starts tracking the mouse movements. The control unit then loops around the states of receiving three packets of data from the mouse and transmitting it to the processor.

Functional Overview:

Control unit acts the interface between the processor and the mouse receiver, transmitter. The control unit collects three packets of data from the mouse receiver and issues the interrupt signal to the processor. The processor jumps to the interrupt handler and the interrupt handler performs a mouse read to get an update on the mouse movements. On getting the read signal the Control unit places the 3 bytes on the data bus and issues the ready signal. The control unit waits for the data to be read before it can clear the interrupt and start collecting data from the receiver again.

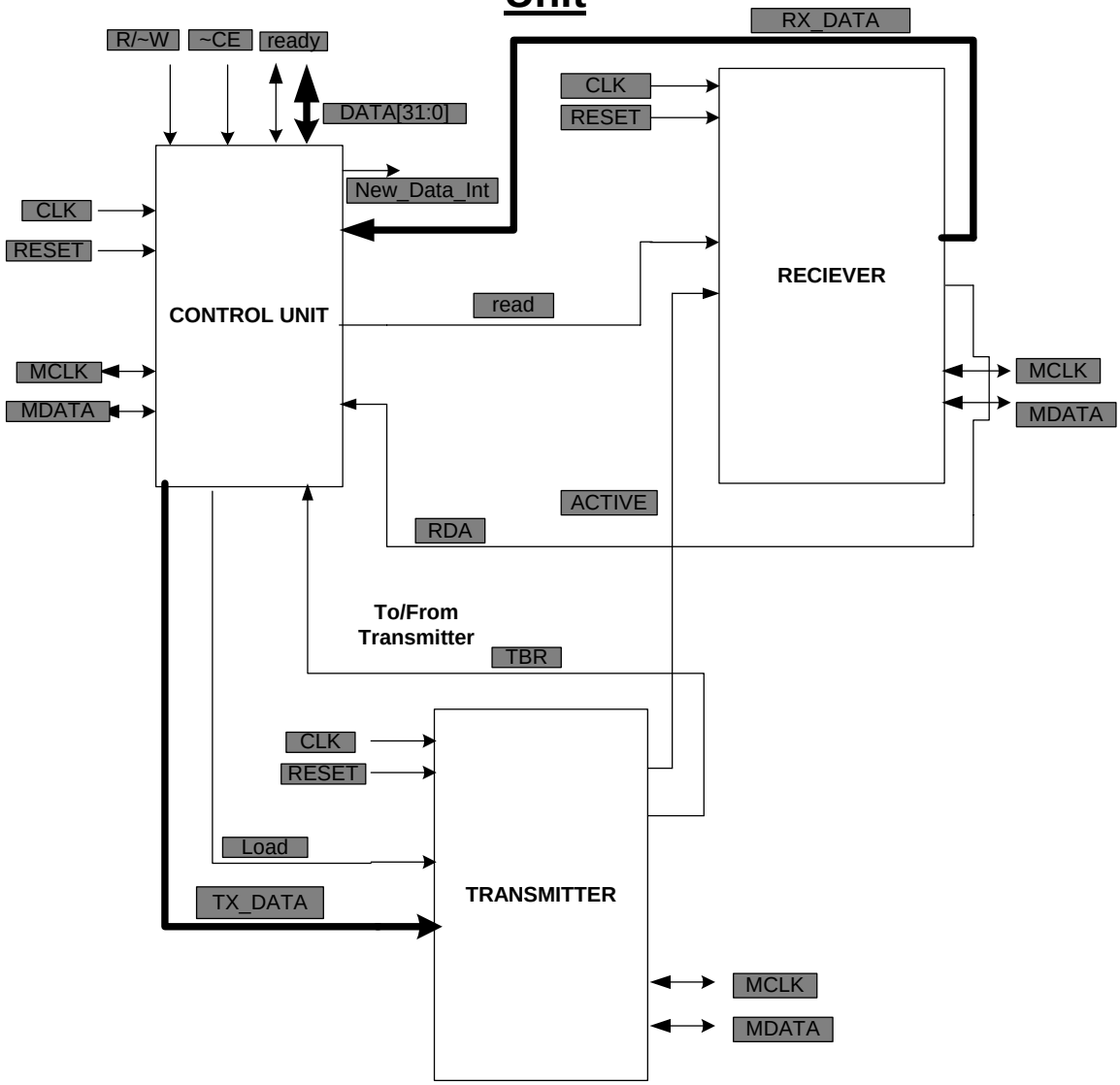
Comments:

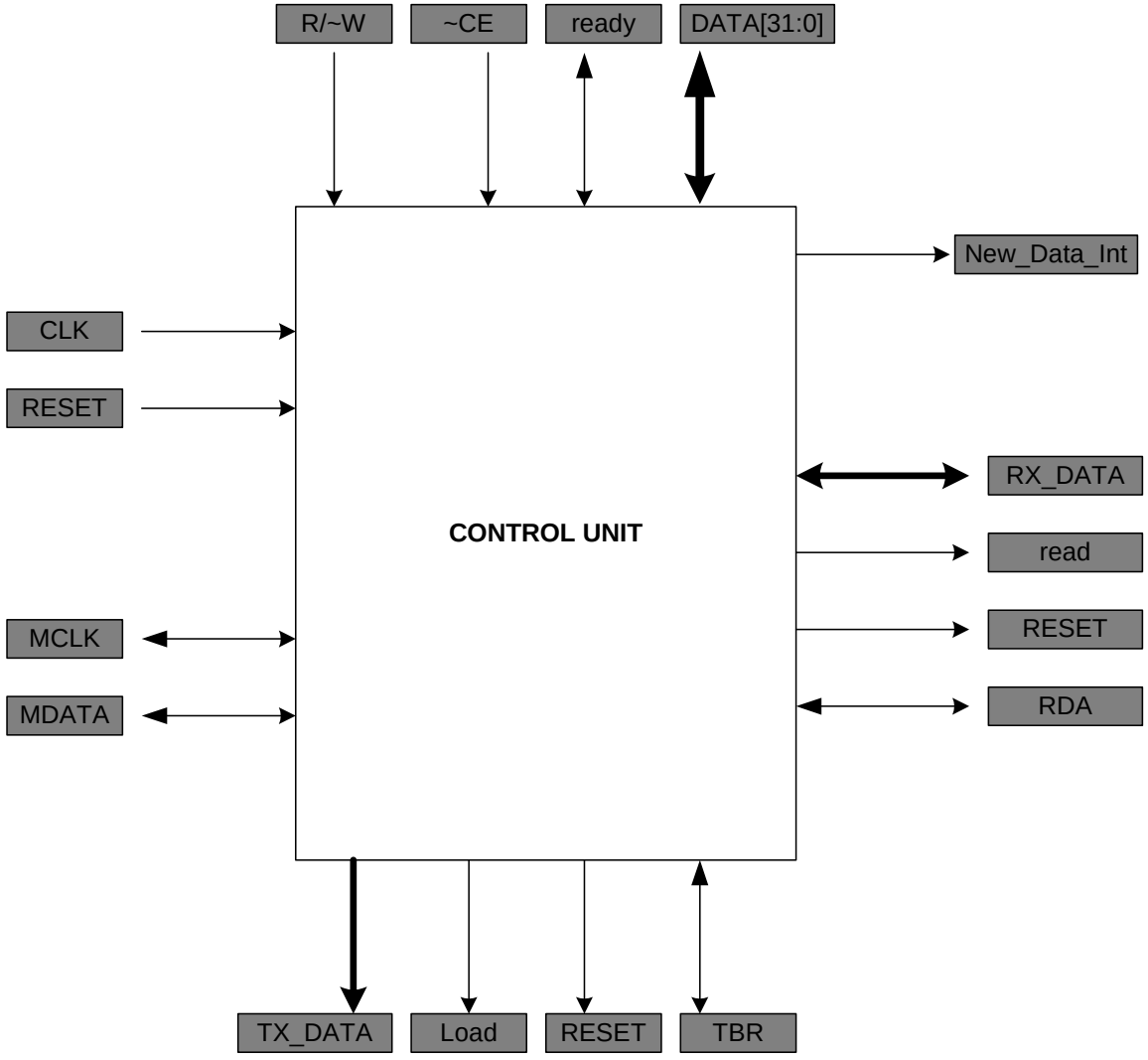
The control unit initially after reset acts as a driver communicating with the mouse. Usually the driver for a mouse is written completely in software to get a good control of the mouse. But, the mouse we designed for our project can only read from the mouse using software but is not capable of writing or transmitting to the mouse. This is a decision we took early during the design phase because we wanted to have the mouse working independent of the processor. With the way our current mouse is designed the initialization is all hardwired. Hence on reset our control unit logic initializes the mouse and brings it to a known state of waiting to receive data on mouse moves. Transition from this phase to a completely software controlled mouse isn't going to be much. We would have to just add a couple more signals capable of controlling the transmitter. Due to the final time crunch we did not add this feature, which we could have easily added had we got some extra time.

Testing of The mouse Interface:

The mouse controller was tested thoroughly. Each module the receiver, transmitter and the control unit were tested uniquely. Hardware test benches were written for the transmitter and the control unit and force file for the receiver. (See topic on HW Testing and Appendix for the test benches). After getting it working in functional simulation we created a bit file and loaded it on the FPGA board to make sure the mouse functions as we expect it to. The mouse did not behave as expected on reset and the changes to our control unit are outlined in section HW testing. After the changes we again loaded it on the FPGA and made sure the mouse interface was transmitting and receiving data correctly using the logic analyzer. This in-depth testing reduced any unexpected functioning.

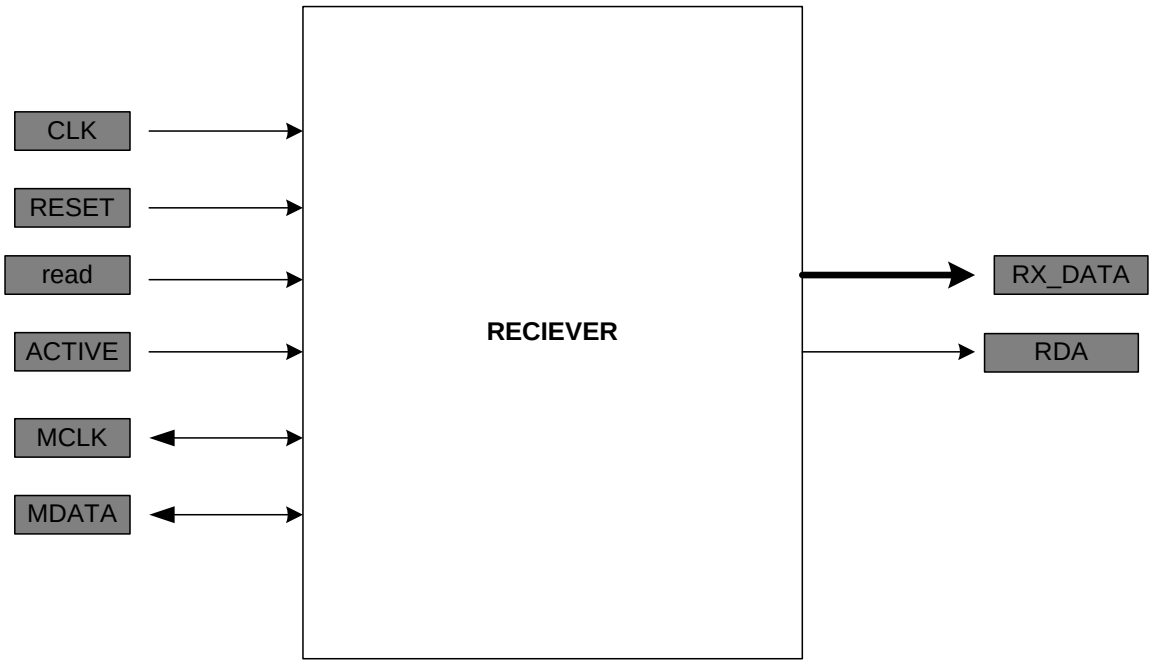
Control Flow in the Control Unit



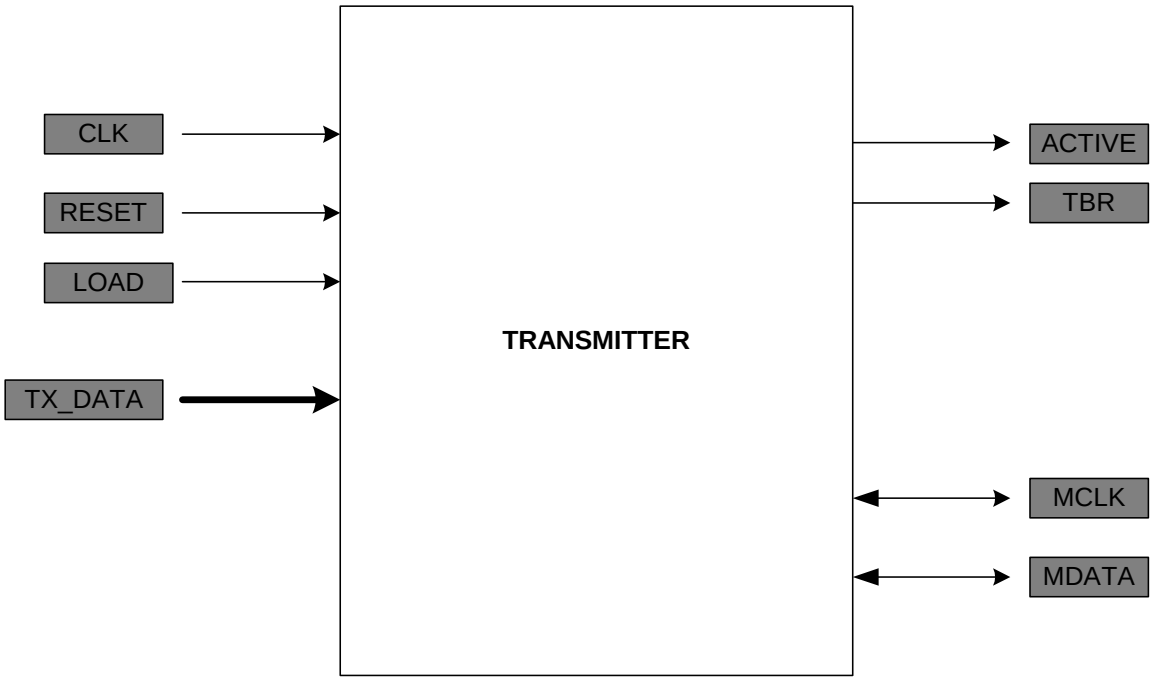


To/From Transmitter

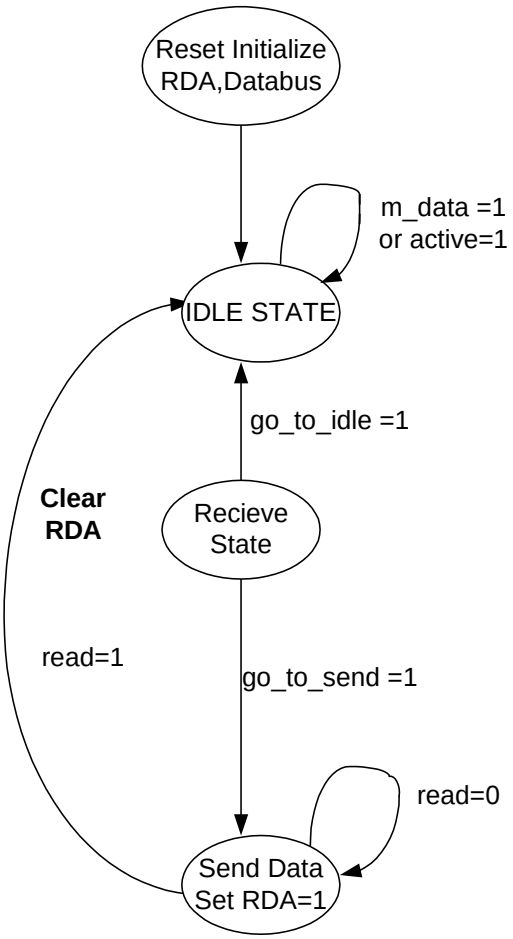
Reciever Block Diagram



Transmitter Block Diagram



Running On Processors Clock.



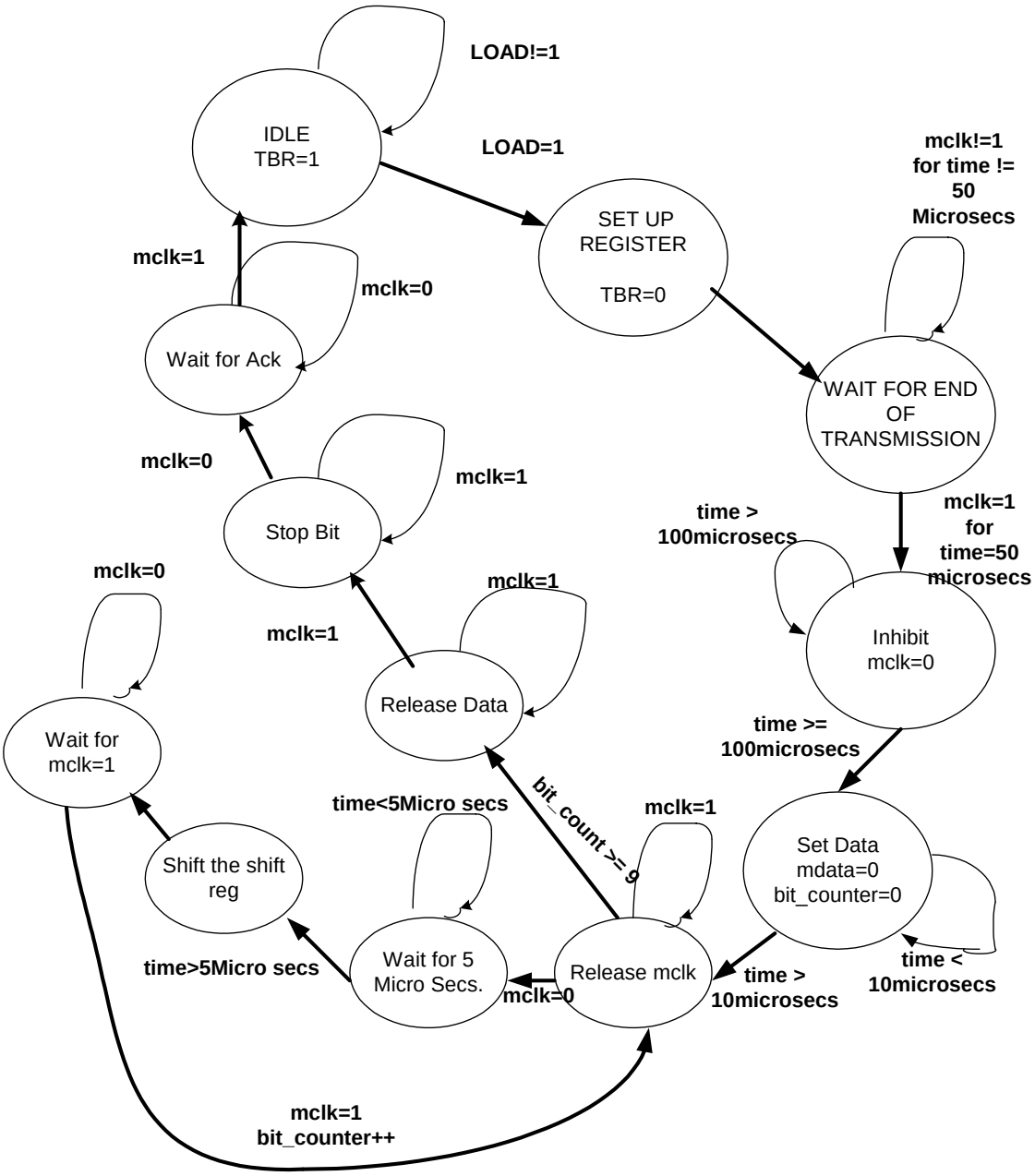
Running On Mouse Clock.

On **Reset** Clear Buffer Register,counter and initialize variables go_to_send,go_to_recieve and go_to_idle.

If in **Recieve State:**
Check Start bit if incorrect set ***go_to_idle=1***.
Recieve a new bit every negative edge of mouse clock.
When stop bit seen set flag ***go_to_send=1*** if wrong stop bit set ***go_to_idle=1***

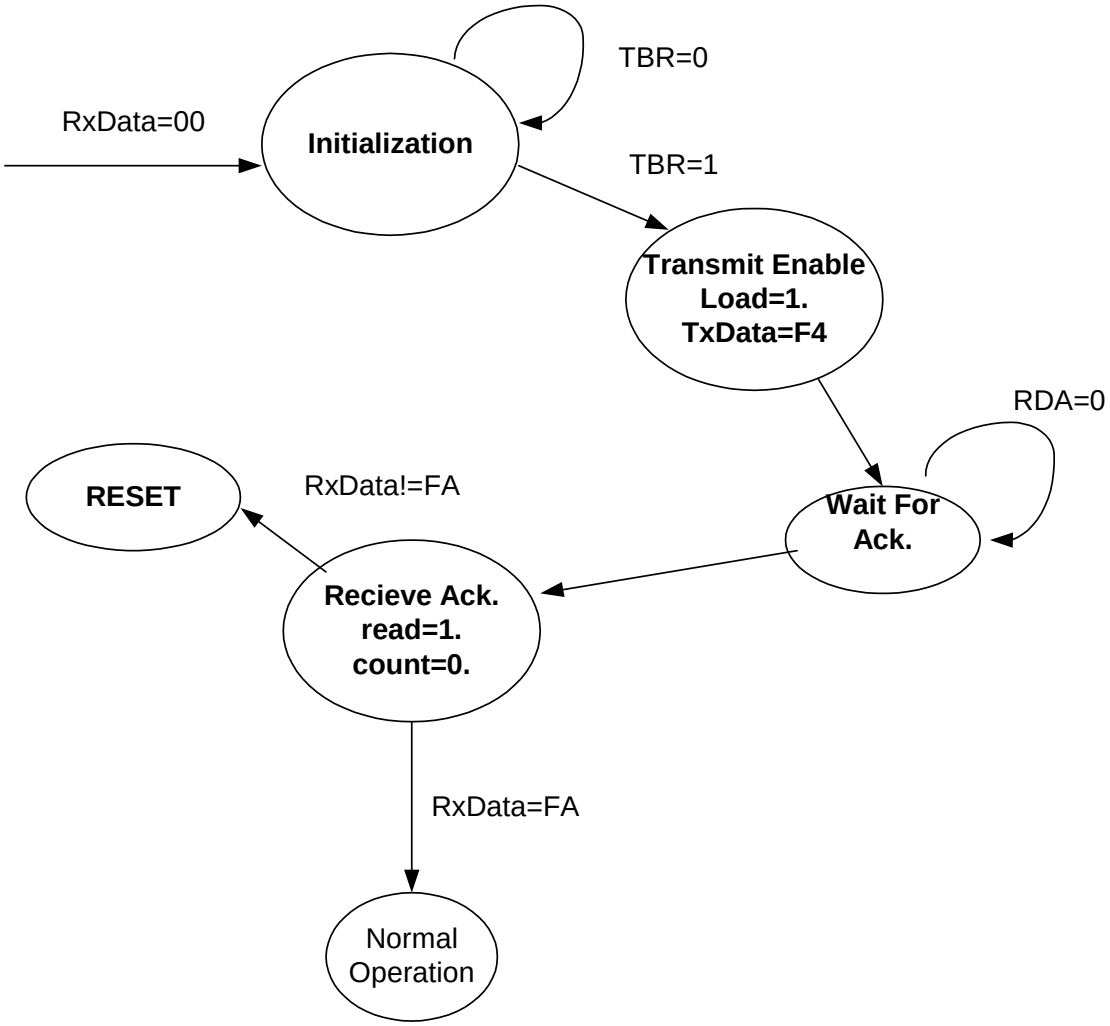
If in **Send State:**
Initialize variables go_to_idle,go_to_send, go_to_recieve.

TRANSMITTER STATE DIAGRAM



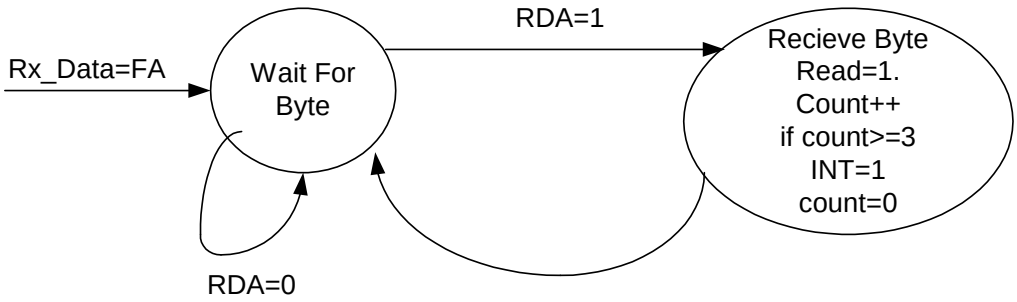
Control Unit

Initialization



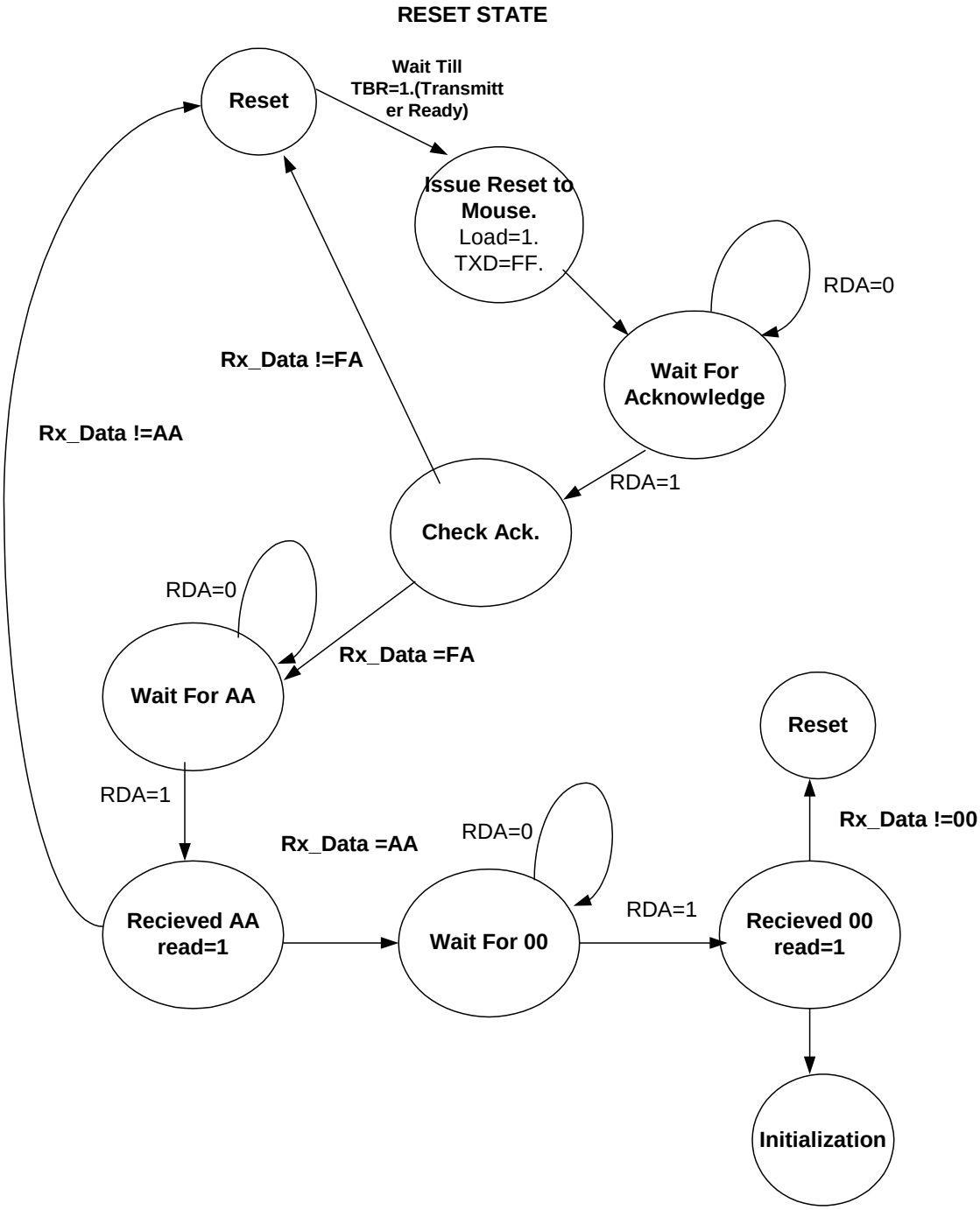
Control Unit

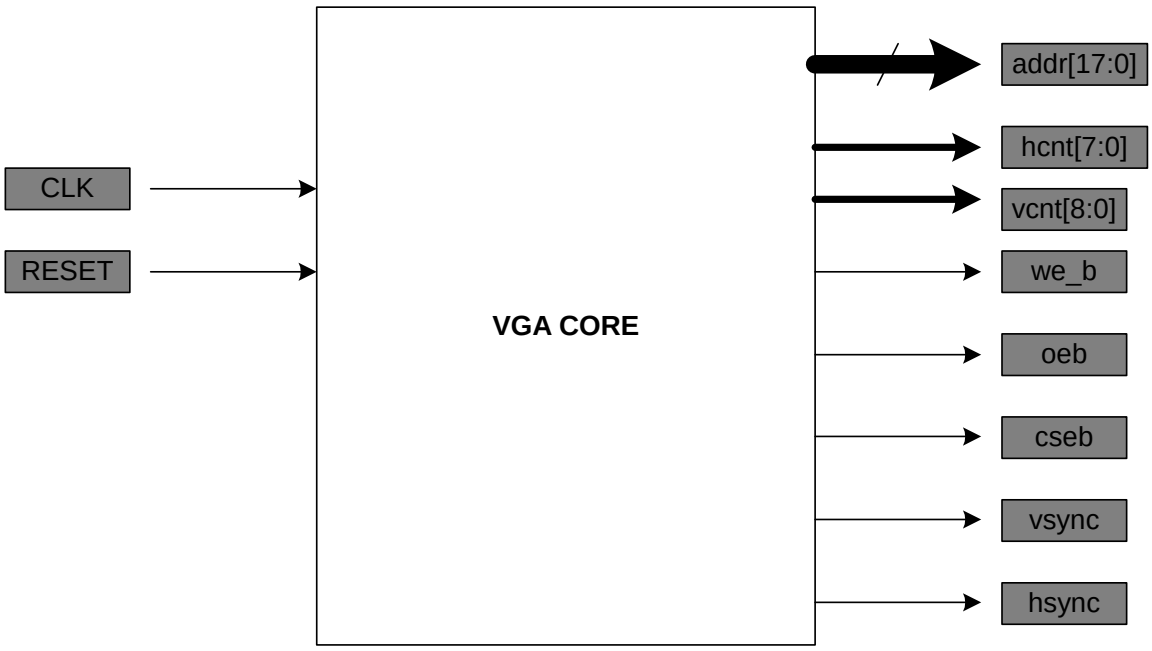
Normal Operation



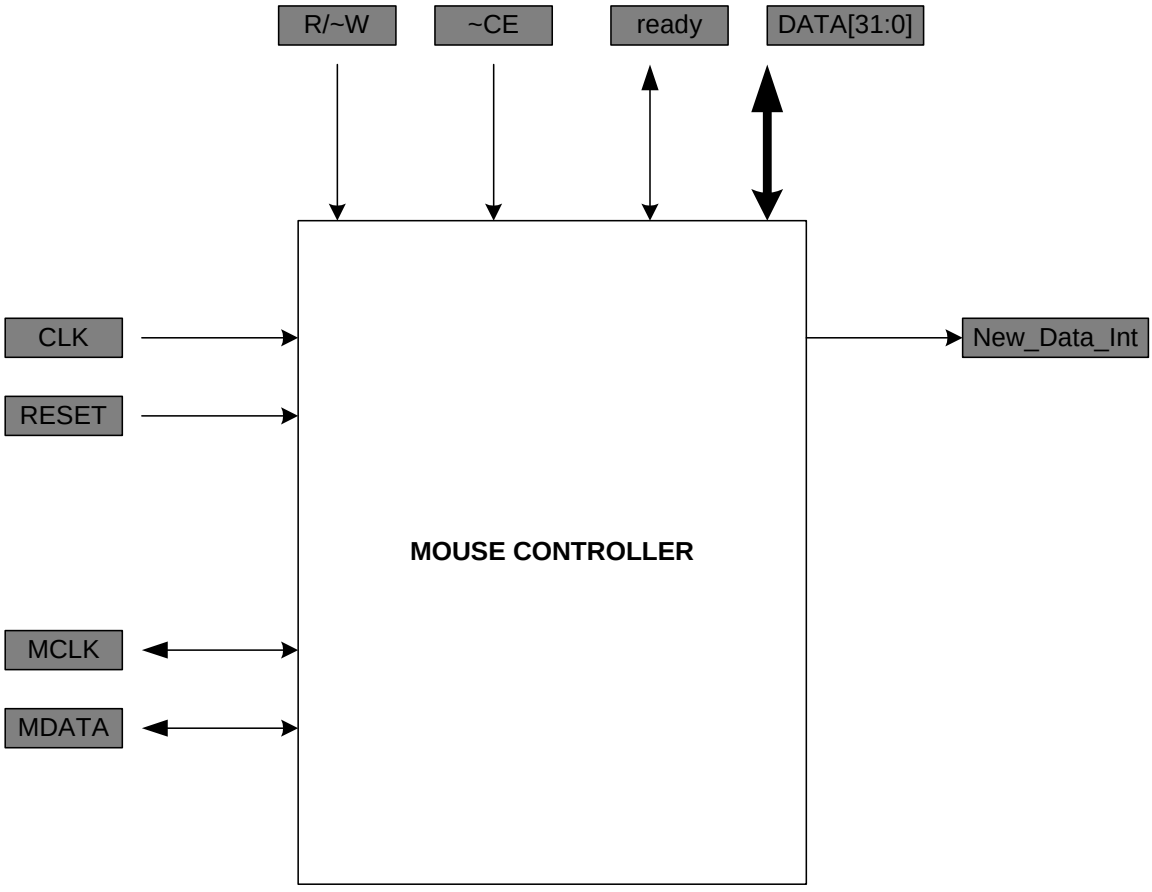
At Any time if R/~w =1 and ~CE=0 then drive data and set INT=0 and set ready=1 for 1 clock cycle.

Control Unit





Mouse Controller Block Diagram



VGA: All the colors displayed by a monitor can be derived by a combination of the three primary colors Red, Blue and Green. To draw a pixel on the monitor the vga controller sends a digital 6-bit (2^6=64 colors) value to the VGA Monitor. This 6-bit value is converted to an Analog signal by the Digital to Analog converter. The analog signal drives an electron gun, which moves from the extreme left to the right bottom and based on the analog signal value an appropriate pixel is drawn on the screen.

As stated before the VGA guns runs from the top left corner to right bottom corner. The gun is controlled by 2 signals namely the horizontal sync (hsync) and vertical sync (vsync). The negative pulse on the hsync mark the start and end of a new line and a negative pulse on the vsync mark the start and end of a new frame buffer. There are some timing protocols to be met while communicating with the VGA monitor and is illustrated below:

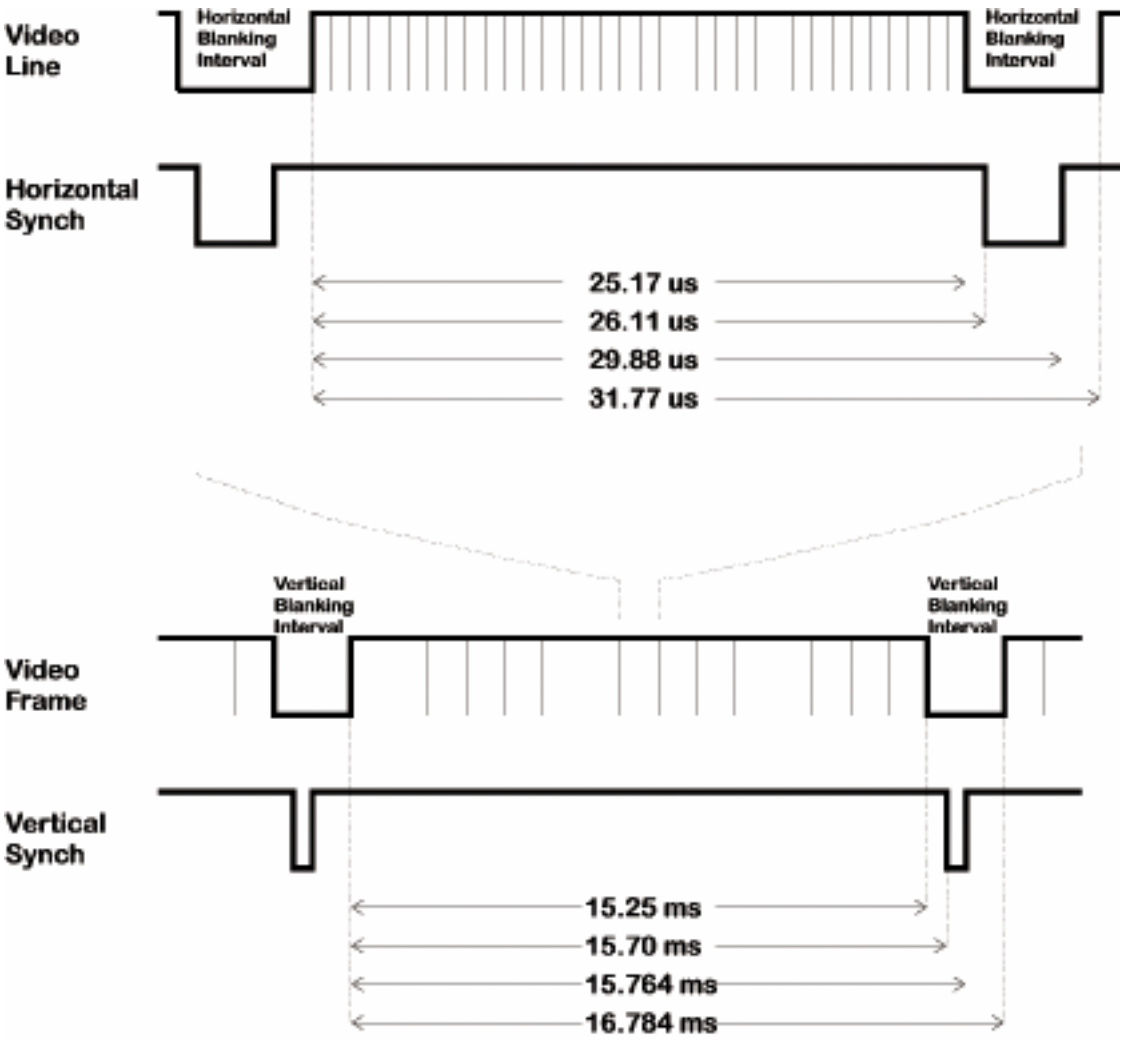


Figure 2: VGA signal timing.

Forward Calculation of Frame Buffer Size:

Monitor Runs at **60 Hz.**
=> Time for refreshing per frame= **16.784 ms**

Number of Horizontal Lines: 527(Vertical Count)
=> Time Per Horizontal Line= **31.77 ms**

Clock Running at 12 MHz and time per Horizontal Line **31.77 ms**
=>Number of Pixels Per Horizontal Line: **381.**

Reverse Calculation:

Number of Clock cycles per horizontal line: 381 (Horizontal Count)
Number of Clock cycles per video frame: 527(Lines) * 381 = 200787.
Clock Running at 12 Mhz = 83.33 Nano Seconds per clock cycle.
=> 83.33 ns* 200787 per frame buffer= 0.167 Seconds
VGA Frame Refresh rate in Frequency 1/0.167 Sec = 59.76 Hz

Jerry's Implementation of VGA:

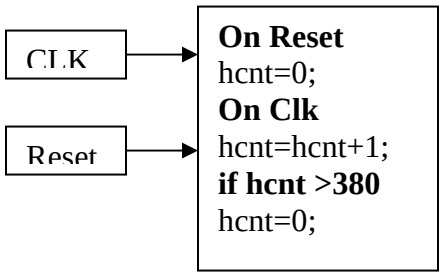
To implement the VGA during the initial rounds of discussion we assigned a contiguous block of memory addresses for the frame buffer. The address of the memory location is incremented for every pixel and the new pixel value is displayed on the monitor. (Look at Memory Map)

The horizontal pixel count and the vertical line number are kept track by two counters hcnt and vcnt. Every clock cycle the hcnt is incremented and for every 380 counts of hcnt the vcnt is incremented by one. We provide buffering on both the sides of the frame buffer in order to center the image and be on the safer side by not being too close to the blanking period.

The VGA core we implemented was a verilog translation of the core provided in the vga hand sheet provided to us. Also due to the difference in the input clock speed the vga core built by us differs in the counter values.

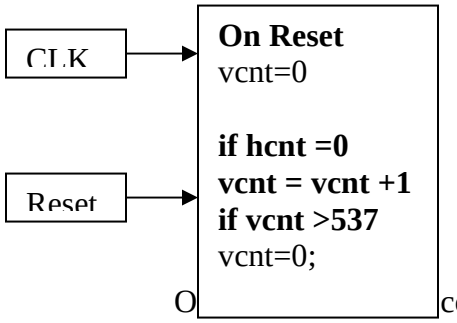
The overall working of the VGA is described in the below flow chart. The code is broken into 5 simple always statements each of which controlling one out put signal.

Horizontal Count Always Block:



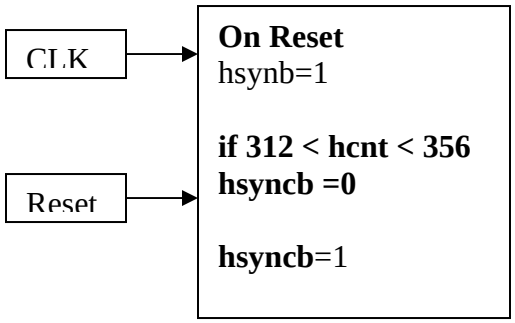
On reset the horizontal count is set to zero. On every positive edge of the 12 Mhz clock the horizontal count is increased. When the count reaches 380 its reset to zero indicating the beginning of a new line.

Vertical Count Always Block:



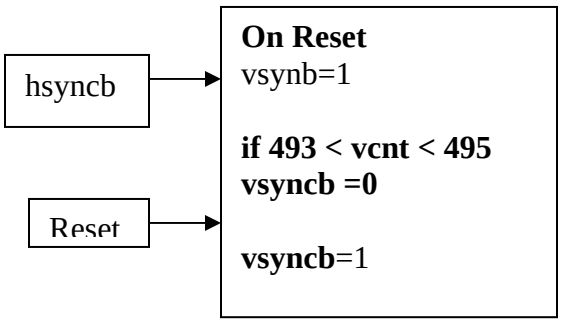
O count is set to zero. Once every 380 horizontal counts the vertical count is incremented by one. When the count reaches 380 its reset to zero indicating the beginning of a new line.

Horizontal Sync Always Block:



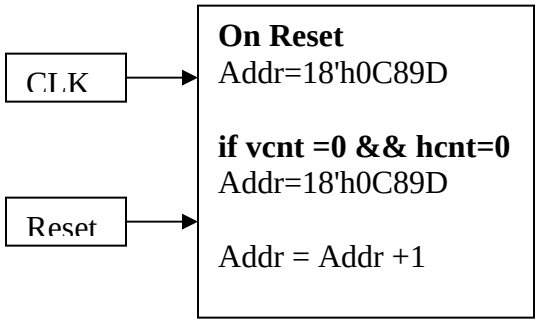
On reset the horizontal sync signal is set to 1. In order to provide the Blanking signal the horizontal sync signal is set 0 when the hcnt is in the range of 312 and 356.

Vertical Sync Always Block:



On reset the vertical sync signal is set to 1. In order to provide the Blanking signal the vertical sync signal is set 0 when the vcnt is in the range of 493 and 495.

Address Calculation Always Block:



The address is calculated using the values in the horizontal and vertical counter. The MATLAB program organizes the pixels in the memory accordingly. The starting address of the frame buffer is 4C89D. The 19th bit is always set to one and we set the lower 18 bits accordingly while calculating the address. When the vcnt and hcnt cycle round to zero again we reset the address back to the starting address.

Testing of the VGA:

The VGA was tested thoroughly. Hardware test bench was written for the VGA core. (See topic on HW Testing and Appendix for the test benches). After getting it working in functional simulation we created a bit file and loaded it on the FPGA board to make sure

the vga functions as we expect it to. There were some initial problems in our code. The counters were off by a value of 1 or 2 due to which the whole image was getting shifted. We had to go back to the functional simulation and re-check to find the bug. Also, the colors displayed by the monitor were not what we expected based on the rgb values. It turned out the counter error was actually giving us the difference in color. Once we fixed the bug and couple more of minor bugs we were able to get the VGA working error free. Due the thorough testing during the initial stages we did not face any unexpected functional error during the final stages of getting the whole project together.