Hardware Description Document

Color Display Controller

Contents

Figures

1. Introduction

The Color Display Controller is an FPGA based VGA controller, developed as a hardware design training project. The main goal is learning about basic communication protocols and design principles.

i. Functionality

The purpose of the controller is to display colored sections on a screen, via VGA. Initially, the screen is one solid color, but it can be split into two sections, vertically or horizontally, or four sections. Each section has a preset color, which can be changed by transmitting a new color code via UART. Additionally, buttons on the board can be used to swap colors between the sections or shuffle the colors in the selected section. Only one section is considered "selected" at once, and its corresponding number is indicated on a 7-segment display. The selected section can be changed with a button, which selects the next one in the series.

ii. Screen Layout

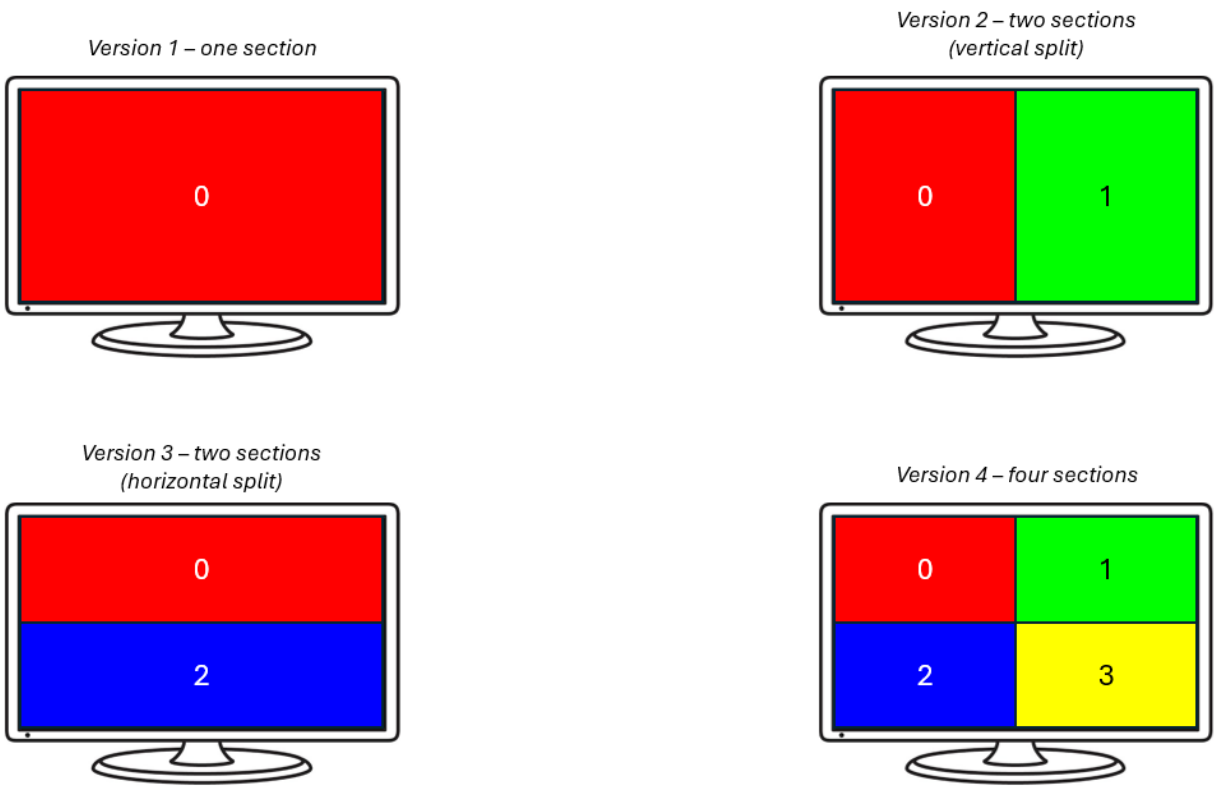


Figure 1 - Possible screen layout versions

There are four possible ways to split the screen, using physical switches. The colors shown in the figures are the presets for each section.

2. Input / Output

i. Input table

Input	Description
SW0	Splits the screen into two sections, vertically (version 2)
SW1	Splits the screen into two sections, horizontally (version 3)
BTNL	Swaps colors between the sections on the left and the sections on the right
BTNU	Swaps colors between the top sections and the bottom sections
BTNC	Changes the color in the selected section to the next one in the series Red->Yellow->Blue->Green
BTNR	Selects the next section on the screen
Rx	Receives via UART register addresses and configurations
Debug (SW14)	Switches to debug mode (shows debug output on board LEDs)
en_7s_frame (SW13)	Enables frame output on seven segment display
debug_color (SW12)	Enables the color code output on the 7-segment display (Requires en_7s_frame OFF)
debug_clr_reg (SW11)	Enables register color code output on 7-segment display (Requires debug_color ON)
rst (SW15)	Resets all modules
clk	100 MHz clock

ii. Output table

Module	Output	Description
VGA Controller	RED (4 bit)	Red bits for RGB color code
	GRN (4 bit)	Green bits for RGB color code
	BLU (4 bit)	Blue bits for RGB color code
	HSYNC	Horizontal SYNC signal
	VSYNC	Vertical SYNC signal
Seven Segment Controller	pos (8 bit)	Driven digit of seven segment display
	segments (8 bit)	Active segments for driven digit
Debug Interface	debug_reg	Shows the value of a register selected through UART
	debug_ch	Shows the selected section of the screen (current channel)
	debug_ frame	Shows the last valid data frame received through UART

3. RTL Diagram

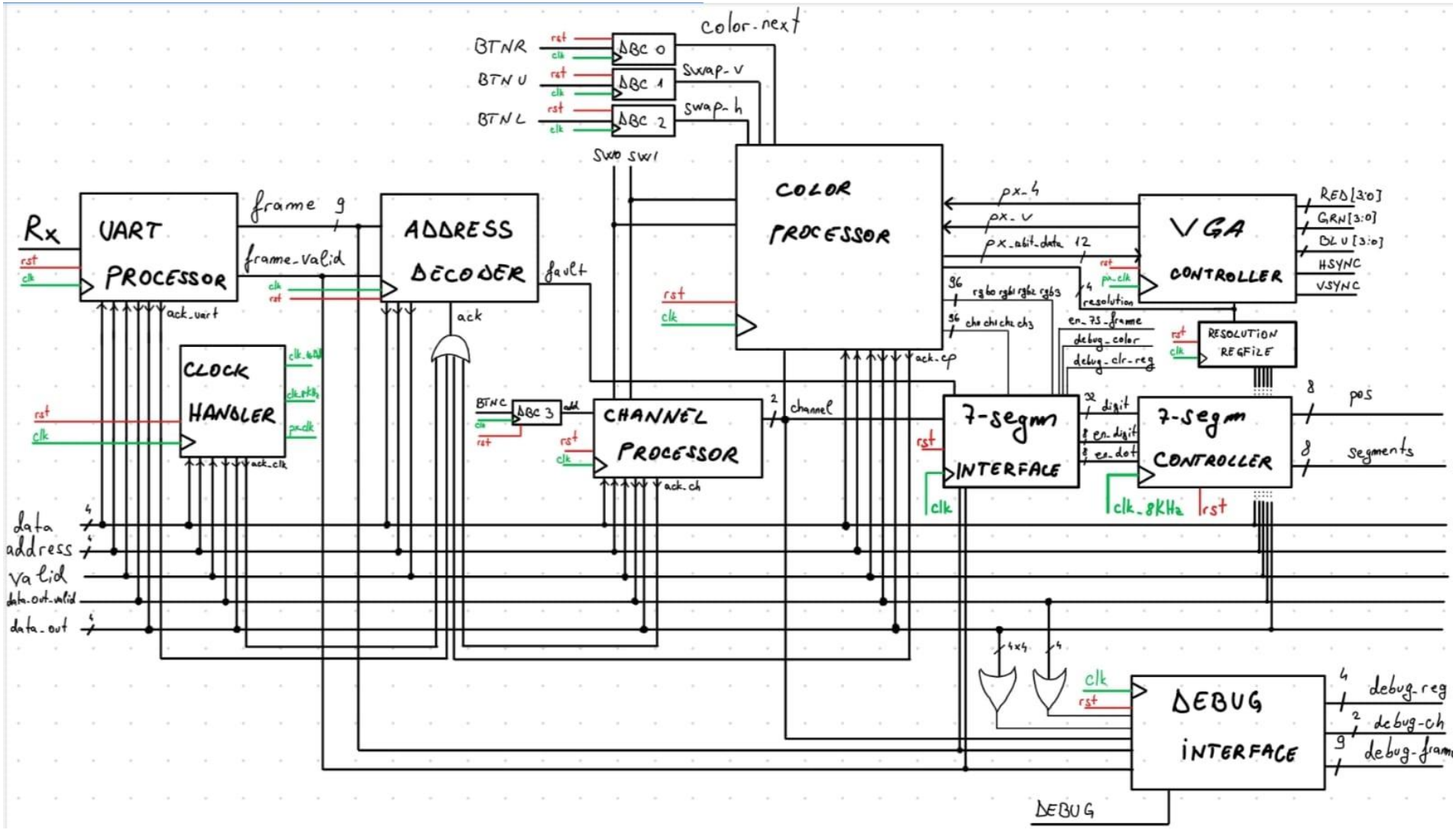


Figure 2 - RTL Diagram

4. Modules

i. UART module

The UART module is composed of two parts, the UART processor, that processes and validates the frames transmitted through Rx, and the regfile, that processes the register configurations for the module.

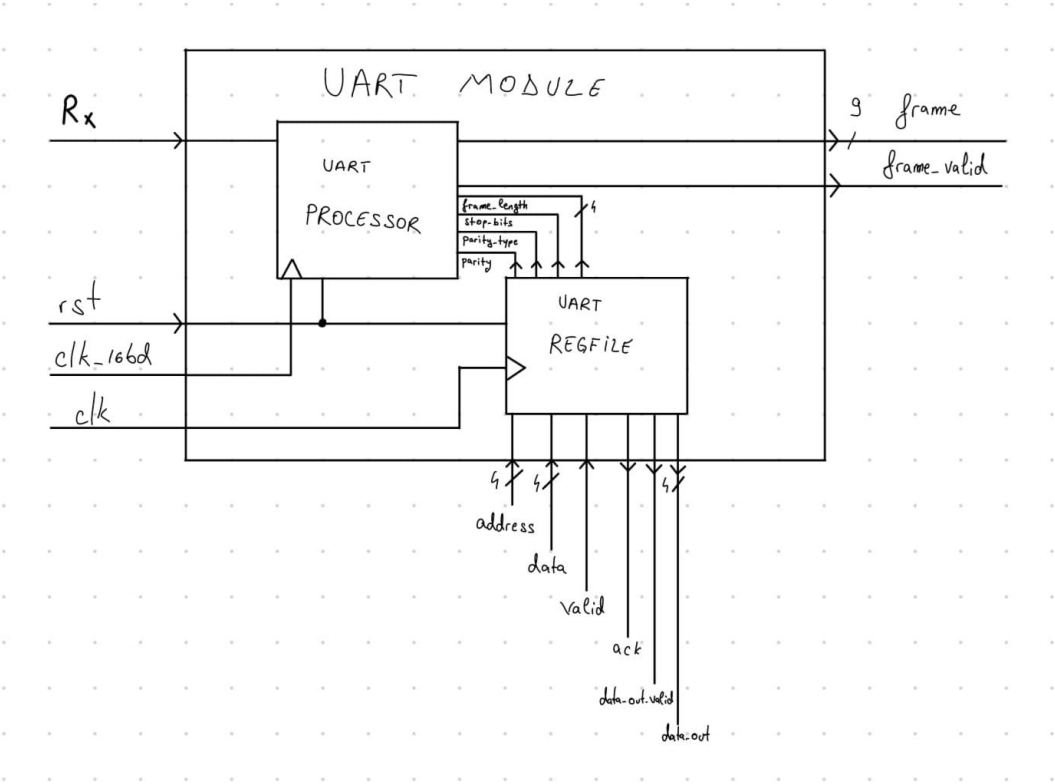


Figure 3 - UART module diagram

- **I/O**

Type	Name	Bits	Connects to	Description
Input	Rx	1	Input	UART serial signal
Input	data	4	Address Decoder	Data to be written in a register
Input	address	4		Address of the register being written
Input	valid	1		Valid signal for the register data
Output	ack	1		Acknowledge signal for the register data
Output	data_out	4	Debug Interface	Register data debug output
Output	data_out_valid	1		Valid signal for data_out
Output	frame	8	Address Decoder Debug Interface	Transmits an 8-bit frame to the address decoder for further processing
Output	frame_valid	1		Valid signal for frame
Input	clk	1	Input	100 MHz clock for the regfile

Input	clk_16bd	1	Clock Handler	16xBaud Rate clock for the processor
Input	rst	1	Input	Hard reset signal

- **Registers**

Name	Address	Data	Hex	Description
parity	1001	0000	90	Disables the parity.
		0001	91	Enables the parity.
		1111	9F	Reads the register value.
parity_type	1010	0000	A0	Sets the parity to even.
		0001	A1	Sets the parity to odds.
		1111	AF	Reads the register value.
stop_bits	1011	0000	B0	Sets the number of stop bits to 1.
		0001	B1	Sets the number of stop bits to 2.
		1111	BF	Reads the register value.
frame_length	1100	0101	C5	Sets the frame length to 5.
		0110	C6	Sets the frame length to 6.
		0111	C7	Sets the frame length to 7.
		1000	C8	Sets the frame length to 8.
		1001	C9	Sets the frame length to 9.
		1111	CF	Reads the register value.
Soft reset	0000	0000	00	Resets all register values.

- **Functionality**

The UART processor receives frames from the Rx port, and validates them, sending only the data bits to the Address Decoder. A valid frame has the following format:

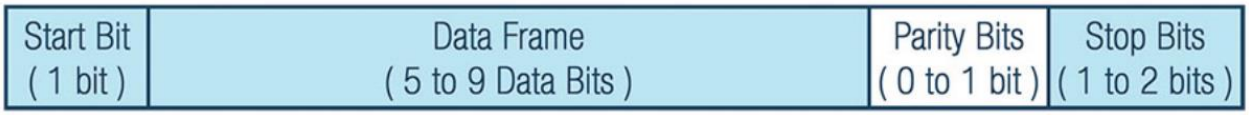


Figure 4 - UART frame format

An UART Frame consists of a start bit, 5 to 9 data bits, 0 or 1 parity bits and 1 or 2 stop bits.

Start Bit

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one (1) clock cycle. When the receiving UART detects the high to low

voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

Data Frame

The data frame contains the actual data being transferred. It can be five (5) bits up to eight (8) bits long if a parity bit is used. If no parity bit is used, the data frame can be nine (9) bits long. In most cases, the data is sent with the least significant bit first.

Parity

Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers.

After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number.

When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

Stop Bits

To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for one (1) to two (2) bit(s) duration.

(Peňa & Legaspi, 2020)

The processor functions using a clock with the frequency of 16 times the Baud Rate. Five different baud rates cand be set by configuring the clock handler. On this clock frequency, an UART data bit is transmitted every 16 clock cycles.

The reading and validation of the data frames is done by a Finite-State Machine, with 6 states: IDLE, START, READ, PARITY, STOP, DROP.

- IDLE

In this state, the Rx line is sampled every clock cycle, until detection of the start bit. Aon detection, the next state is set to START and a counter that increments every clock cycle is set to 0. This counter is then used to sample Rx every 16^{th} clock cycle.

At the same time, the IDLE state keeps the frame_valid signal at LOW.

- START

This state is active during the start bit, and is used to reset some registers necessary for the validation process. At the end of the start bit, the next state is set to READ.

- READ

This state samples Rx every 16 clock cycles and builds the data frame bit by bit, until the set frame length has been reached. After that, the next state is set to parity.

- PARITY

The parity state is used to validate the parity of the data frame. If there is no parity, it directly sets the next state as stop. Otherwise, the number of odd bits in the frame is calculated and compared with the parity bit. Those should be equal in case of even parity, and opposite in case of odd parity.

After 16 clock cycles, the next state is set to STOP.

- STOP

The stop state does the final validation of the frame. It counts the stop bits, and if the number is not equal to the set number, it sets the next state to DROP. If the parity was invalid, it also sets the next state to DROP. If everything is valid, frame_valid is set to HIGH, and the next state is set to IDLE.

- DROP

The drop state is used to reset the frame output to LOW if invalid and set the next state to IDLE.

ii. Address decoder

The address decoder receives the data frames from the UART processor, decodes the register addresses and makes the respective register configurations.

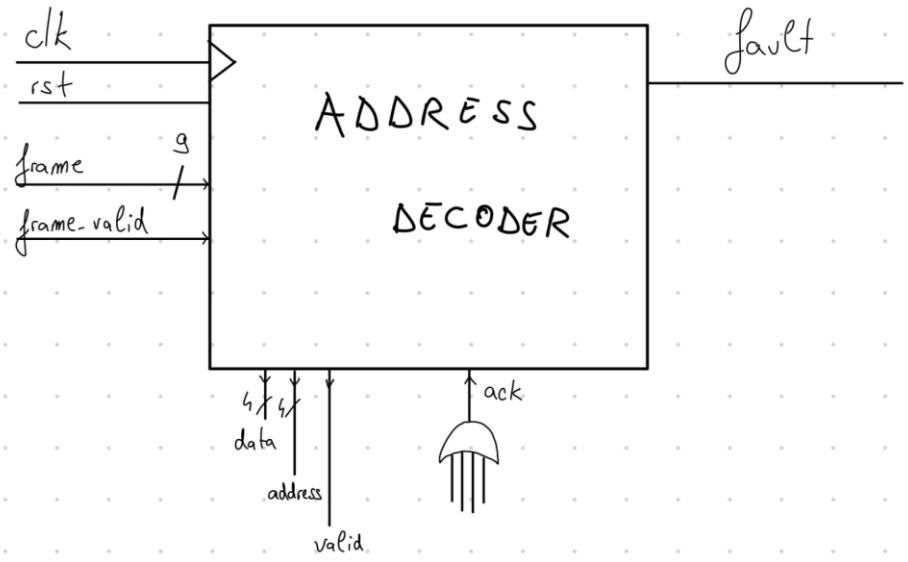


Figure 5 - Address Decoder Diagram

- **I/O**

Type	Name	Bits	Connects to	Description
Input	frame	8	UART processor	Transmits an 8-bit frame to the frame decoder for further processing
Input	frame_valid	1		Valid signal for frame
Output	data	4	UART processor Clock Handler Channel Processor Color processor	Data to be written in a register
Output	address	4		Address of the register being written
Output	valid	1		Valid signal for the register data
Input	ack	1		Acknowledge signal for the register data
Output	fault	1	Seven Segment Interface	Fault signal for invalid address (no acknowledge)
Input	clk	1	Input	100 MHz clock
Input	rst	1	Input	Hard reset signal

- **Functionality**

The address decoder works using the standard clock of the FPGA. It uses a finite-state machine with four states: WAIT, SPLIT, SEND, ACK. The data is transmitted on a bus using the signals address, data and valid, and expects an acknowledgement on the ack signal.

- WAIT

The wait state is an idle state that expects the frame_valid signal from the UART processor, and then sets the next state to SPLIT.

- SPLIT

The split state extracts the register address and the data from the frame, and sets the next state as SEND.

- SEND

The send state sets the valid signal to HIGH and waits for the ack signal for 8 clock cycles. It moves to the ACK state after acknowledgement or back to WAIT after 8 clock cycles, setting the valid signal to LOW.

- ACK

The ack state sets the valid back to LOW and moves back to WAIT.

iii. Channel processor

The channel processor is used to keep track of the current channel, and to switch to another active channel.

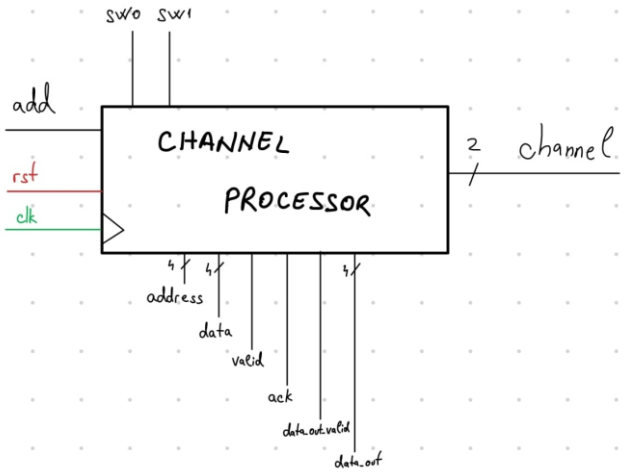


Figure 6 - Channel Processor

- **I/O**

Type	Name	Bits	Connects to	Description
Input	data	4	Address Decoder	Data to be written in a register
Input	address	4		Address of the register being written
Input	valid	1		Valid signal for the register data

Output	ack	1		Acknowledge signal for the register data
Output	data_out	4	Debug Interface	Register data debug output
Output	data_out_valid	1		Valid signal for data_out
Input	add	1	BTNC + DBC3	Switches to the next section
Input	SW0	1	Input	Screen vertical split
Input	SW1	1	Input	Screen horizontal split
Output	channel	2	Color processor Seven Segment Controller	Sends the channel code of the selected section
Input	clk	1	Input	100 MHz clock
Input	rst	1	Input	Hard reset signal

- **Registers**

Name	Address	Data	Hex	Description
ch	0010	0000	20	Selects channel 0
		0001	21	Selects channel 1, if available
		0010	22	Selects channel 2, if available
		0011	23	Selects channel 3, if available
		1111	2F	Reads the register value
Soft reset	0000	0000	00	Resets all register values

- **Functionality**

Unlike the other configurable modules, the channel processor does not have a separate regfile, due to the need for the register to be configurable from both UART and the module itself.

The module processes two signals: *address* and *add*. The address 0010 corresponds to the channel register, and when receiving this it, the module takes the data from the bus. If it is 1111, it outputs the current channel to the Debug Interface, otherwise, the data is further processed. If the channel that it indicates is active, it becomes the set channel.

The signal *add* is active when BTNC is pressed. BTNC is connected to a debouncer (DBC3), which outputs the signal add. When add is active, the channel is changed to the next active one, using the following logic:

- Current channel is 0
    - SW0 is active: Channel becomes 1
    - SW0 is inactive and SW1 is active: Channel becomes 2
    - Other: Channel remains 0
- Current channel is 1
    - SW1 is active: Channel becomes 2
    - Other: Channel becomes 0
- Current channel is 2
    - SW0 and SW1 are active: Channel becomes 3
    - Other: Channel becomes 0
- Current channel is 3: Channel becomes 0

iv. Seven Segment Interface

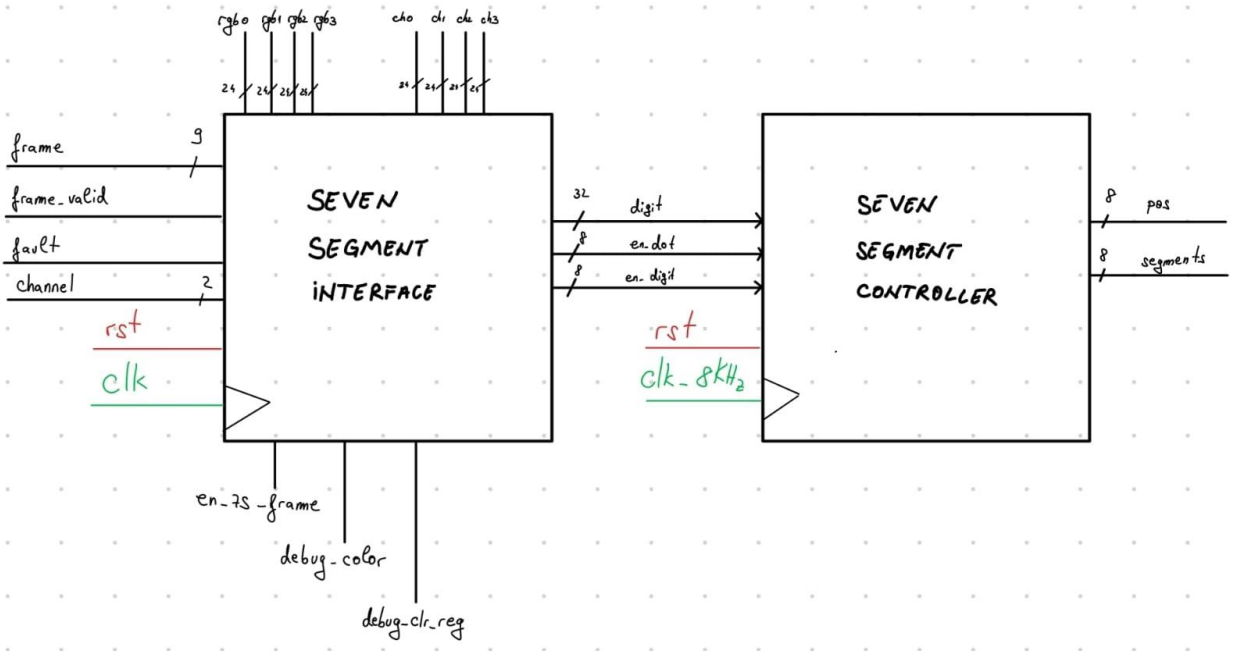


Figure 7 - Seven Segment Interface and Controller

- **I/O**

Type	Name	Bits	Connects to	Description
Input	en_7s_frame	1	Input	Debug frame mode
Input	debug_color	1		Debug color mode
Input	debug_clr_reg	1		Debug color register mode
Input	frame	9	UART module	9 bit received data frame
Input	frame_valid	1		Valid signal for frame

Input	channel	2	Channel Processor	Currently active channel
Input	rgb0-4	4x24	Color Processor Module	Color register codes for each channel
Input	ch0-4	4x24		Displayed color codes for each channel
Output	digit	32	Seven Segment Controller	8 digits to be displayed
Output	en_dot	8		Dots for every digit on the display
Output	en_digit	8		Enable signal for each digit
Input	clk	1	Input	100 MHz clock
Input	rst	1	Input	Hard reset signal

- **Functionality**

The seven segment interface sends data to be displayed to the seven segment controller. It has four modes, depending on the switches en_7s_frame, debug_color and debug_clr_reg. The modes are:

  - **All switches off**

  Digit 8 of the display is set to the current channel, digit 1 may show an 'F' if the last register address received through UART is invalid.

  - **en_7s_frame is on (debug frame mode)**

  All digits show the last data frame received through UART.

  - **debug_color is on and debug_clr_reg is off (debug color mode)**

  Digit 8 is set to the current channel, digits 1-6 show the hex code of the displayed color of that channel.

  - **debug_color and debug_clr_reg are on (debug color register mode)**

- Digit 8 is set to the current channel, digits 1-6 show the hex code of the color registers of that channel.

v. Seven Segment Controller

- **I/O**

Type	Name	Bits	Connects to	Description
Input	digit	32	Seven Segment Interface	8 digits to be displayed
Input	en_dot	8		Dots for every digit on the display
Output	en_digit	8		Enable signal for each digit

Output	pos	8	Output	Driven digit of seven segment display
Output	segments	8		Active segments for driven digit
Input	clk_8KHz	1	Clock Handler	8 KHz clock
Input	rst	1	Input	Hard reset signal

- **Functionality**

The seven segment controller generates the drive signals for the seven segment display. Based on an 8 KHz clock, for each of the 8 positions on the display, if the en_digit is asserted, it generates the segments using the respective bits of the digit signal and the en_dot signal. The controller can display all of the hexadecimal digits.

vi. Clock handler

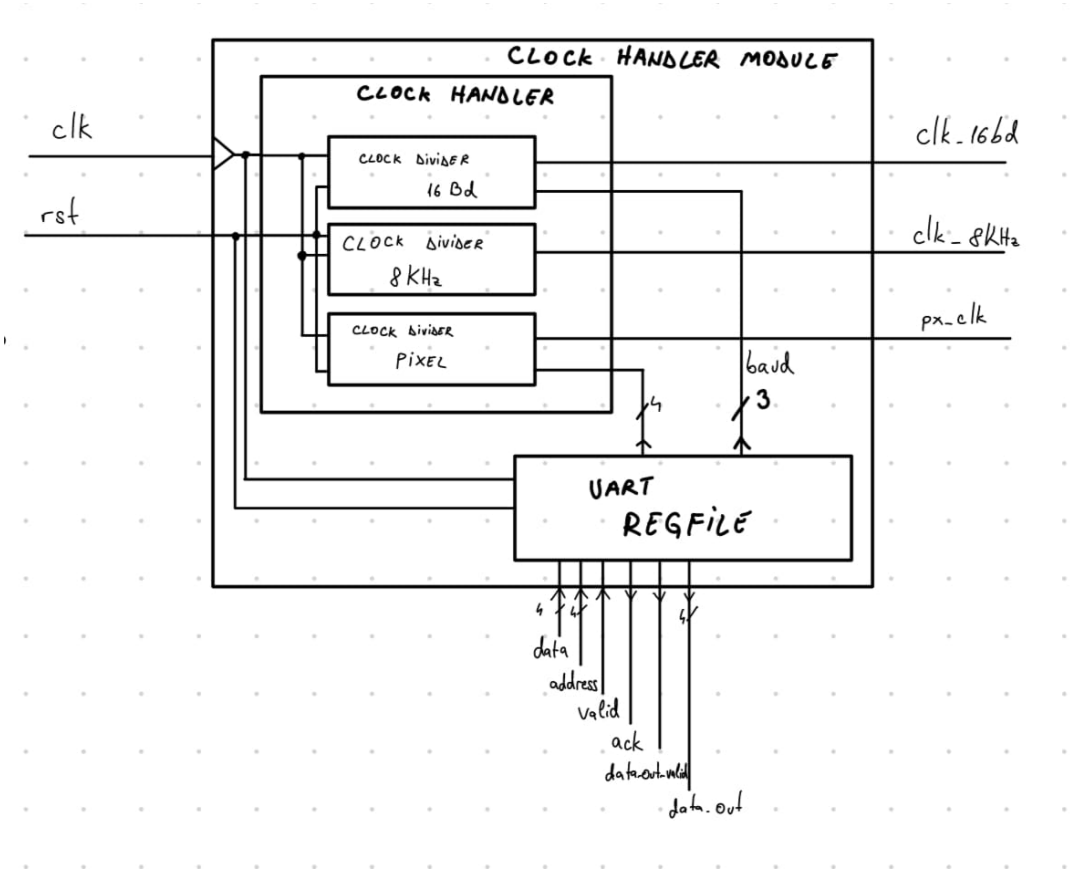


Figure 8 - Clock handler diagram

- **I/O**

Type	Name	Bits	Connects to	Description
Input	data	4	Address Decoder	Data to be written in a register
Input	address	4		Address of the register being written
Input	valid	1		Valid signal for the register data
Output	ack	1		Acknowledge signal for the register data
Output	data_out	4	Debug Interface	Register data debug output
Output	data_out_valid	1		Valid signal for data_out
Output	clk_16baud	1	UART processor	16xBaud Rate clock signal
Output	clk_8KHz	1	Seven Segment Controller	8KHz clock for the Seven Segment refresh rate of 1KHz
Output	px_clk	1	VGA Controller	Pixel clock
Input	clk	1	Input	100 MHz clock
Input	rst	1	Input	Hard reset signal

- **Registers**

Name	Address	Data	Hex	Description
baud	0001	0000	10	Sets the baud rate to 4800
		0001	11	Sets the baud rate to 9600
		0010	12	Sets the baud rate to 19.2k
		0011	13	Sets the baud rate to 57.6k
		0100	14	Sets the baud rate to 115.2k
		1111	1F	Reads the register value
px_ratio_pos	1110	0000	E0	Sets the pixel clock to 25MHz
		0001	E1	Sets the pixel clock to 50MHz
Soft reset	0000	0000	00	Resets all register values.

- **Functionality**

The Clock Handler is designed to integrate multiple clock dividers, for different frequencies.

This module contains a 16x Baud Rate divider and 8KHz divider and a pixel clock divider. Five of the most used baud rates can be set: 4800, 9600, 19.2k, 57.6k and 115.2k. The 16x baud rate clock divider functions using a ratio of clock cycles for each Baud Rate, right shifted with 4 bits. A counter going up to the ratio drives the divided clock. The 8KHz divider has a fixed ratio of 12500. The pixel clock divider has a ratio of either 4 or 2, for 25 and 50 MHz.

vii. Color Processor

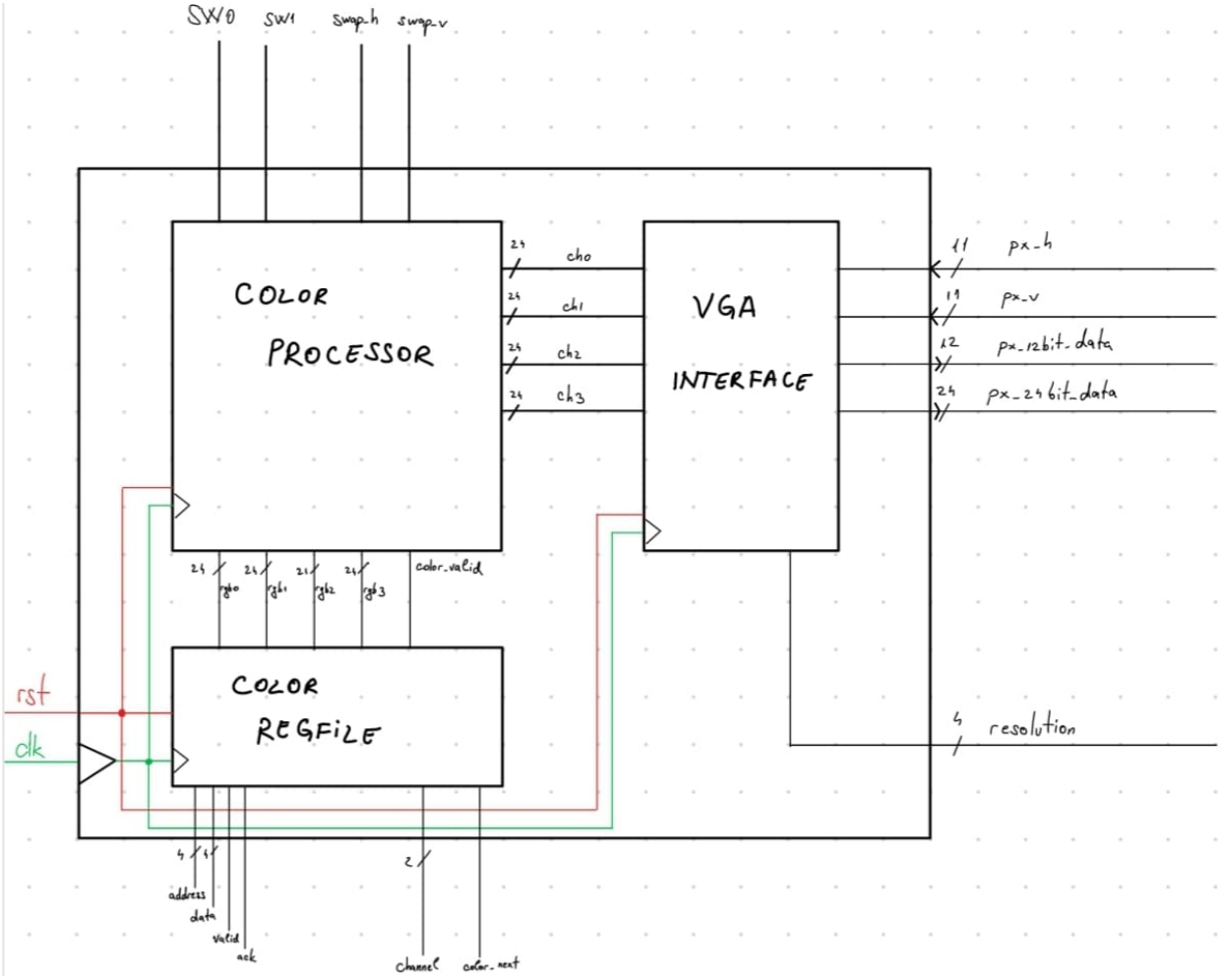


Figure 9 - Color Processor Module

- **I/O**

Type	Name	Bits	Connects to	Description
Input	SW0	1	Input	Splits the screen in two sections, vertically
Input	SW1	1	Input	Splits the screen in two sections, horizontally
Input	swap_h	1	BTNL + DBC2	Swaps the colors between two sections, horizontally
Input	swap_v	1	BTNU + DBC1	Swaps the colors between two sections, vertically
Input	color_next	1	BTNR + DBC0	Changes the color of a section to the next one in the preset series
Input	channel	2	Channel Processor	Current active channel

Input	data	4	Address Decoder	Data to be written in a register
Input	address	4		Address of the register being written
Input	valid	1		Valid signal for the register data
Output	ack	1		Acknowledge signal for the register data
Output	data_out	4	Debug Interface	Register data debug output
Output	data_out_valid	1		Valid signal for data_out
Input	px_h	11	VGA controller	Pixel column
Input	px_v	11		Pixel line
Output	px_24bit_data	24		Pixel color code (24 bit)
Output	px_12bit_data	12		Pixel color code (12 bit)
Input	clk	1	Input	100 MHz clock
Input	rst	1	Input	Hard reset signal

- **Functionality**

The Color Processor module is split in three parts: color processor, color regfile and VGA interface.

The regfile handles the color registers, one for every channel. The registers are split into six individually addressable sections. When a configuration for a color address is received through UART, it is applied on the register corresponding to the active channel. The regfile also sets the preset colors, and changes to the next preset for the active channel when BTNR is pressed.

The color processor gives the color code for each channel to the VGA interface. Even if not all channels are activated, the color processor provides four color codes:

- SW0 and SW1 are inactive: All codes are the same as channel 0
- SW0 active, SW1 inactive: Channel 2 is the same as channel 0, channel 3 is the same as channel 1
- SW0 inactive, SW1 active: Channel 1 is the same as channel 0, channel 3 is the same as channel 2
- SW0 and SW1 are active: All channels have individual codes

The color swap operations are also performed by the color processor. The changes are not set in the registers, only in the processors. When a new color is set through UART for a specific channel, the others remain unchanged, as the color_valid signal is only asserted for that channel.

The VGA interface uses the color codes for each channel and the current set resolution to provide a 12 bit and a 24 bit color code output for each requested pixel position.

- **Registers**

Name	Address	Data	Description
rgbx [23:20]	0011 + channel x	4-bit color code	Red MSB
rgbx [19:16]	0100 + channel x	4-bit color code	Red LSB
rgbx [15:12]	0101 + channel x	4-bit color code	Green MSB
rgbx [11:8]	0110 + channel x	4-bit color code	Green LSB
rgbx [7:4]	0111 + channel x	4-bit color code	Blue MSB
rgbx [3:0]	1000 + channel x	4-bit color code	Blue LSB
Soft reset	0000	0000	Reset all colors

viii. VGA Controller

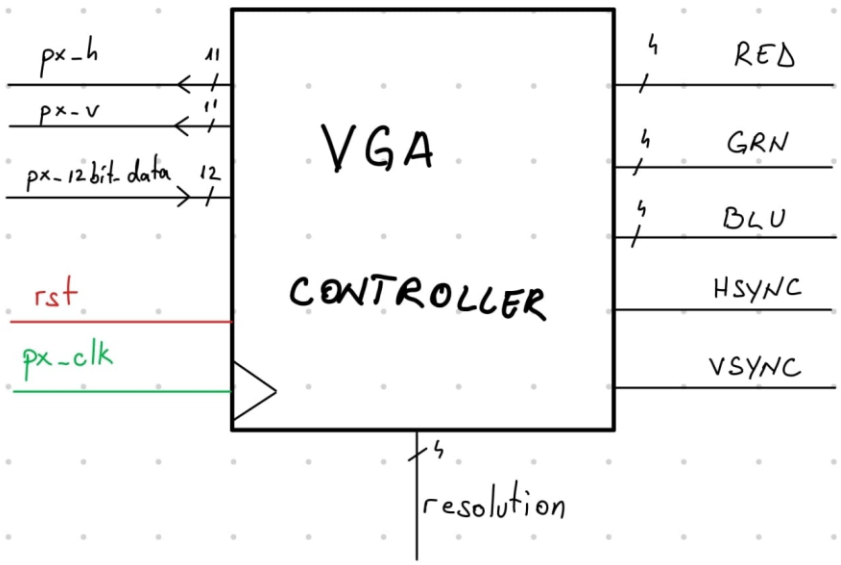


Figure 10 - VGA controller

- **I/O**

Type	Name	Bits	Connects to	Description
Output	px_h	11	Color Processor module (VGA Interface)	Pixel column
Output	px_v	11		Pixel line
Input	px_12bit_data	12		Pixel color code (12 bit)
Output	RED	4	Output	Monitor Red code
Output	GRN	4		Monitor Green code

Output	BLU	4		Monitor Blue code
Output	HSYNC	1		Monitor horizontal sync
Output	VSYNC	1		Monitor vertical sync

- **Functionality**

The VGA controller provides video data to a monitor through VGA. For every defined resolution, it generates the horizontal and vertical sync signals and provides the red, green and blue bits during the active section, and the code for black during the front porch, back porch and sync signals.

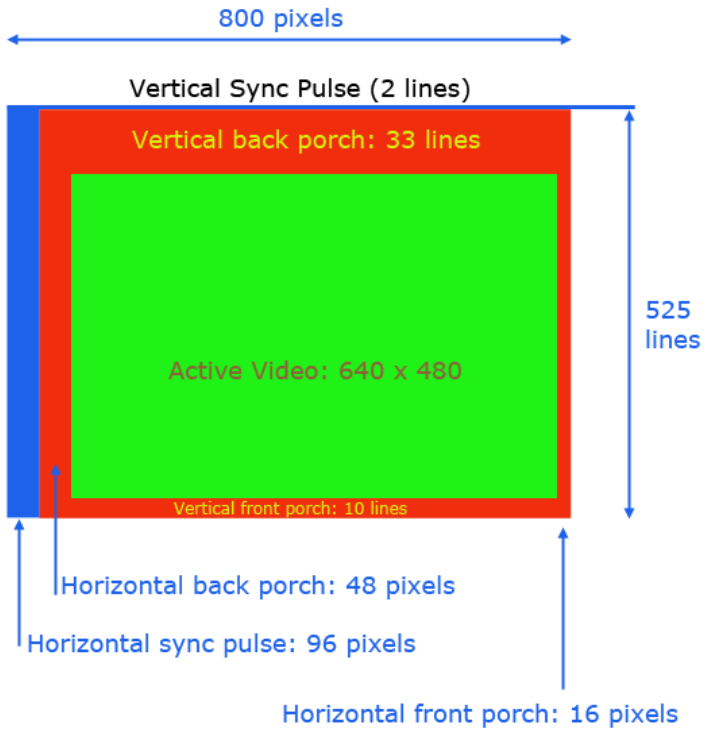
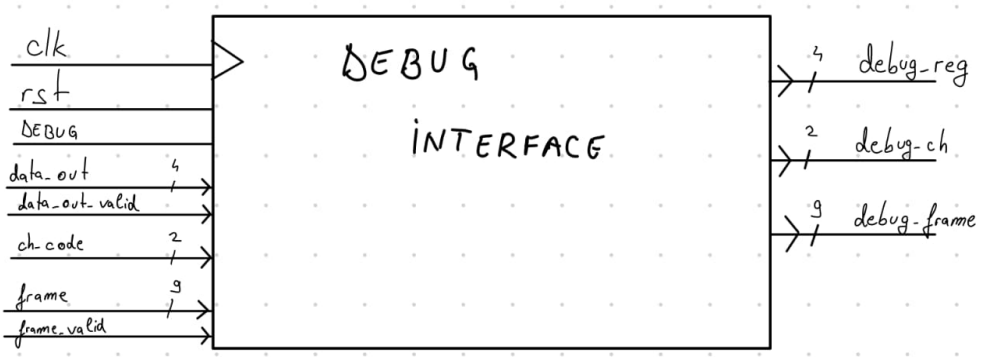


Figure 11 - VGA Timing for 640 x 480 resolution (Stack Exchange, n.d.)

ix. Debug Interface



- **I/O**

Type	Name	Bits	Connects to	Description
Input	Debug	1	Input	Activate debug mode
Input	data_out	4	UART processor Clock Handler Channel Processor Color processor	Register data
Input	data_out_valid	1		Register data valid signal
Input	ch_code	2	Channel Processor	Channel data
Input	frame	9	UART processor	Frame data
Input	frame_valid	1		Frame data valid signal
Output	debug_reg	4	Output	Debug output for register data
Output	debug_ch	2		Debug output for channel data
Output	debug_frame	9		Debug output for frame data

- **Functionality**

The debug interface is used to output certain data from the modules to the 16 LEDs of the FPGA. There is a 4 bit register data output, a 2 bit channel output and a 9 bit frame output.

Register data				NC	Channel		Last received valid data frame									

x. Resolution Regfile

The resolution regfile is used to set the screen resolution, for both the VGA controller and the VGA interface.

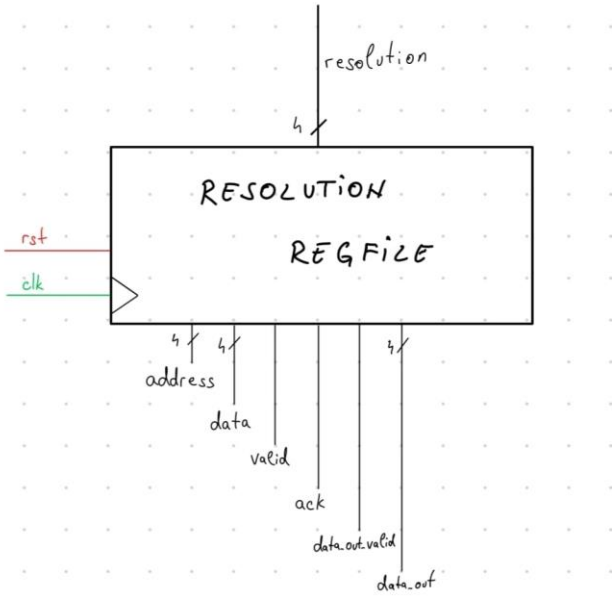


Figure 12 - Resolution Regfile

- **Registers**

Name	Address	Data	Hex	Description
resolution	1101	0000	D0	Resolution 640 x 480
		0001	D1	Resolution 800 x 600
		0010	D2	Resolution 1024 x 768 *not implemented
		0011	D3	Resolution 1152 x 864 *not implemented
		0100	D4	Resolution 1280 x 720 *not implemented
		0101	D5	Resolution 1280 x 800 *not implemented
		0110	D6	Resolution 1280 x 1024 *not implemented
		0111	D7	Resolution 1400 x 1050 *not implemented
		1000	D8	Resolution 1400 x 900 *not implemented
		1001	D9	Resolution 1600 x 900 *not implemented
		1010	DA	Resolution 1680 x 1050 *not implemented
		1011	DB	Resolution 1920 x 1080 *not implemented
		1111	DC	Reads the register value
Soft reset	0000	0000	00	Resets all register values

Bibliography

Peňa, E., & Legaspi, M. G. (2020, December). *UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter*. Retrieved from analog.com: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

Stack Exchange. (n.d.). Retrieved from https://i.stack.imgur.com/tGTx3.png