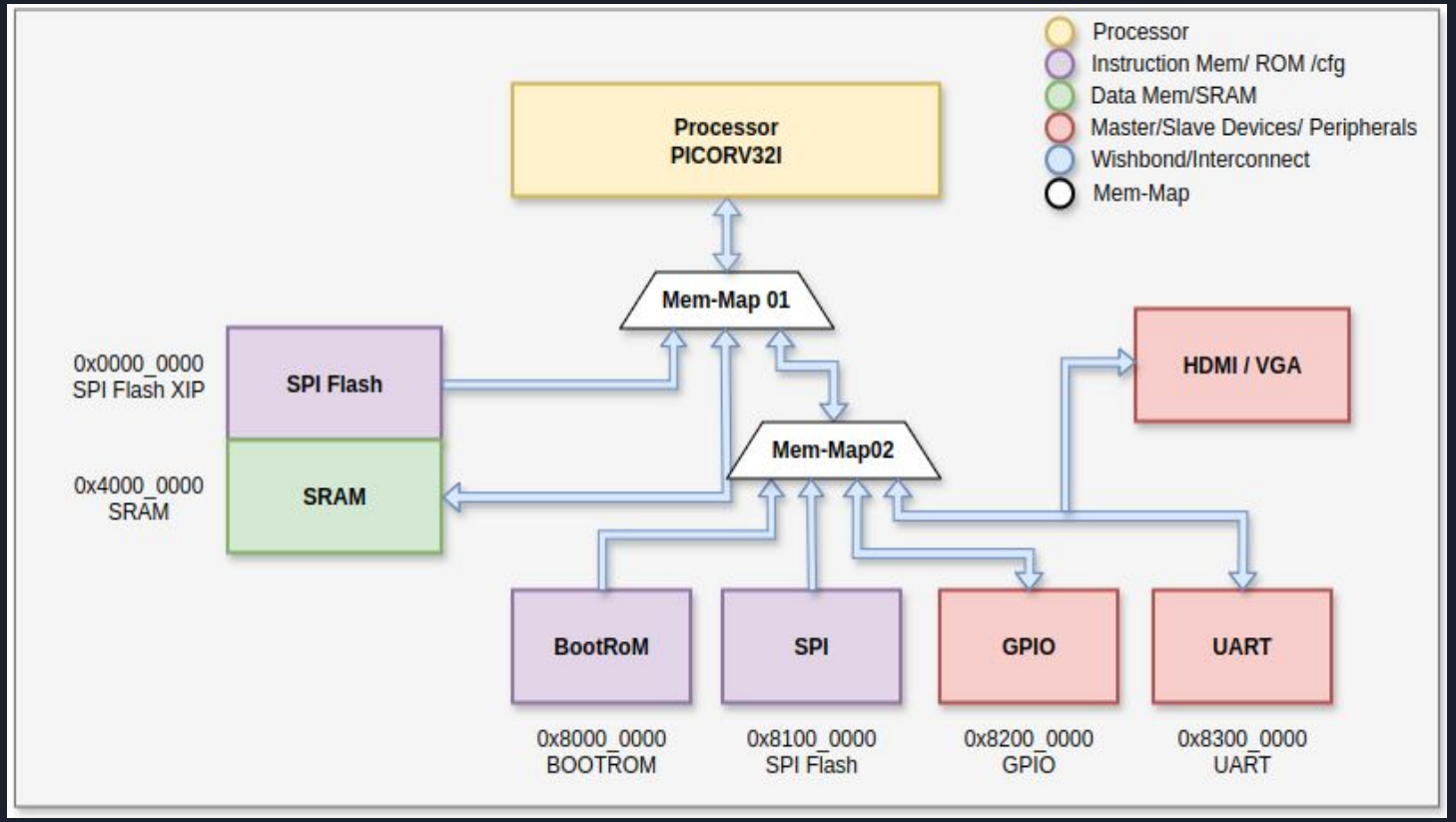
RISC-V SoC and Firmware Development

Lab: 04

Table Of Content

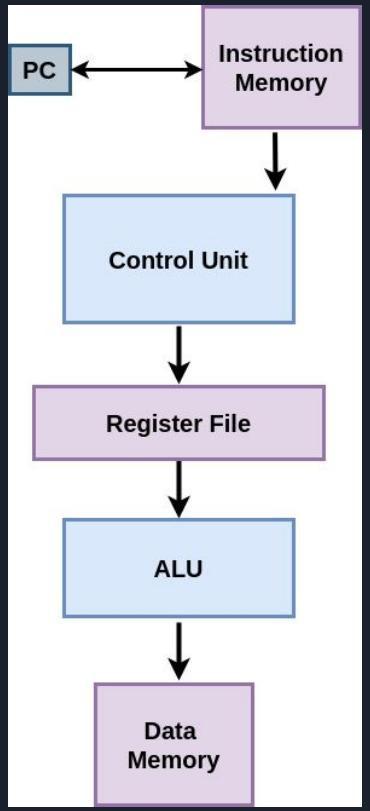
Processor Microarchitecture.



Processor Microarchitecture.

Microarchitecture refers to the way a computer's central processing unit (CPU) is designed and organized at the lowest level. It encompasses the specific implementation details of a CPU's intern components, such as the organization of the registers, the design of the arithmetic logic units (ALUs),size of instructions and the control units that manage the flow of data and instructions.

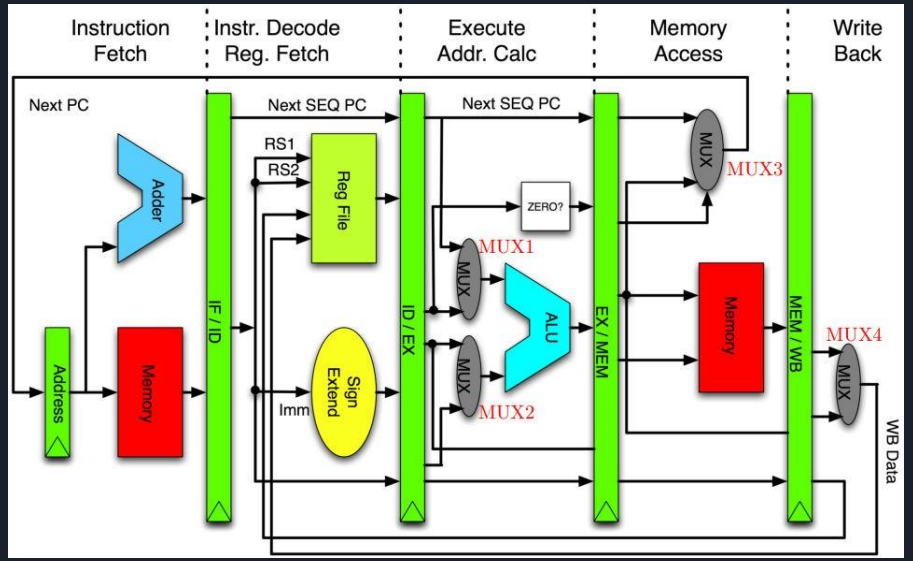
In other words, microarchitecture is concerned with the internal workings of a CPU, including the way instructions are executed, data is processed



Processor Microarchitecture.

The RISC-V architecture is divided into five stages:

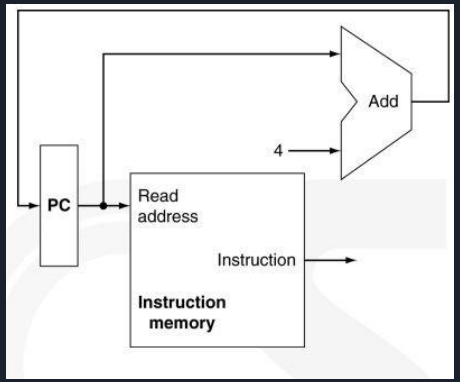
- Fetch
- Decode
- Execute
- Memory
- Writeback



Processor Microarchitecture.

Fetch stage:

The Instruction fetch stage contains two modules program counter and the instruction memory. Program counter acts as a pointer to the instruction memory it increments each cycle by +4, Indicating increment of a word(32 bits) and pointing to next instruction in memory.



00000000	93	00	00	02	addi x1, x0, 32
00000004	37	01	00	c0	lui x2, 0xc0000
00000008	83	c1	00	00	lbu x3, 0(x1)
0000000c	63	88	01	00	beq x3, x0, +16
00000010	23	00	31	00	sb x3, 0(x2)
00000014	93	80	10	00	addi x1, x1, 1
00000018	6f	f0	1f	ff	jal x0, -16
0000001c	6f	00	00	00	jal x0, 0

Processor Microarchitecture.

Decode stage:

After fetching, the instruction enters the decode stage where it is decoded based on its type (R, I, SB, etc.) determined by the 7-bit opcode.

The func3 and func7 fields distinguish operations within the same opcode (e.g., AND, OR, XOR).

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

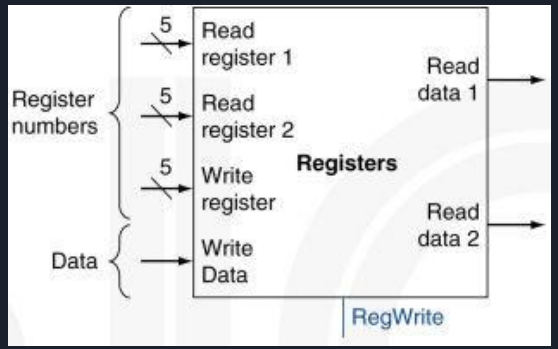
MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM
add	R	0110011	000	0000000
sub	R	0110011	000	0100000
sll	R	0110011	001	0000000
slt	R	0110011	010	0000000
sltu	R	0110011	011	0000000

Processor Microarchitecture.

Decode stage:

rs1, rs2, and rd are 5-bit fields that select read and write registers (32 registers),register x0 is always zero and cannot be written to.
imm field indicates the immediate value in instructions, which replaces rs1 with a 32-bit signed immediate

For sign ex ension the MSB of immediate is sign extended.



Processor Microarchitecture.

Decode stage:

Sign extension of -2047 decimal

(MSB=1):1000 0000 0000 -> 1111 1111 1111 1111 1111 1000 0000 0000

Sign extension of 1033 decimal

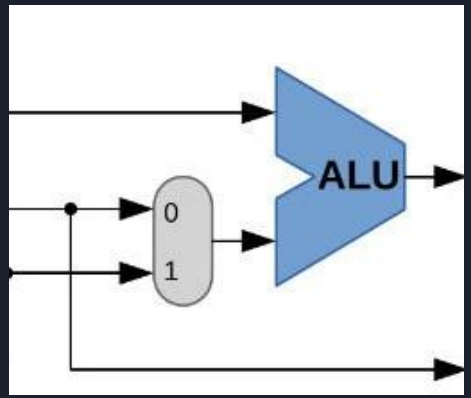
(MSB=0):0100 0000 1001 -> 0000 0000 0000 0000 0000 0100 0000 1001

Processor Microarchitecture.

Execution stage:

The execution stage performs arithmetic,logical calculations,calculating address for memory,jump and branch target.

The inputs of alu can be from counter registers,immediate,Program depending upon the type of instruction.

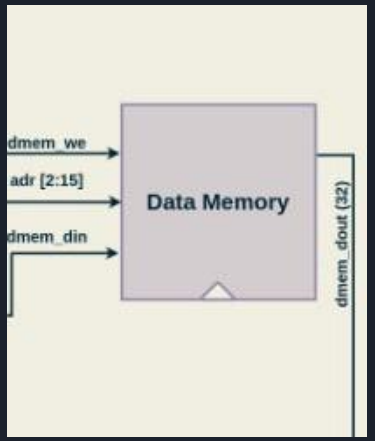


The alu contains the func3 and func7 from decode stage.The 3-bits from func3 three and 6th bit from func7 are concatenated,which tells alu to perform a particular action(add,sub,xor,and e.t.c)

Processor Microarchitecture.

Memory stage

The memory stage is used to access data memory in order to read or write from memory,using load and store instructions.
If it's a load instruction, data is fetched from memory and placed in a register.
If it's a store instruction, data from a register is written to memory
NOTE:
 The memory stage is bypassed if the instruction is not load or store.

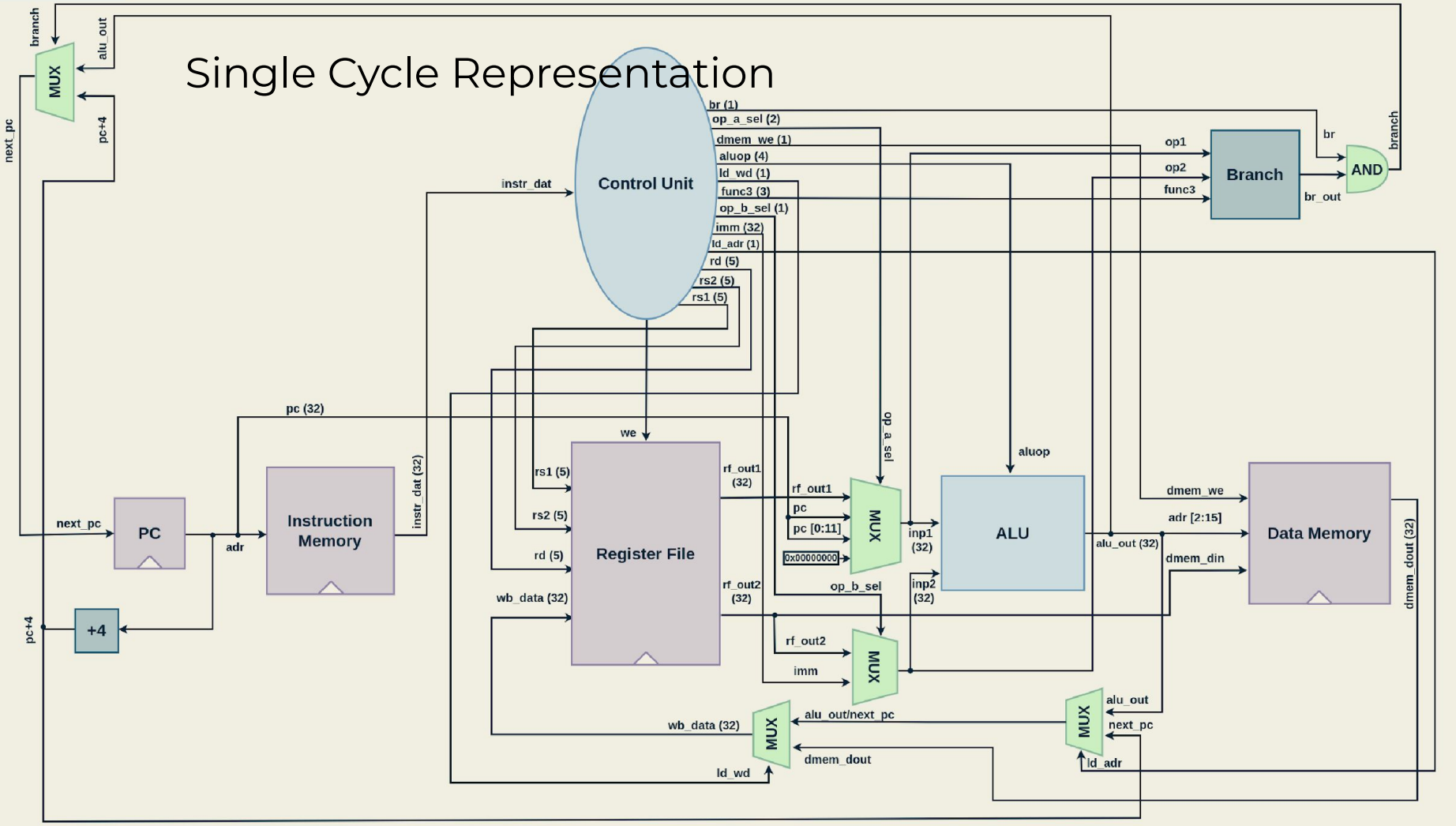


Processor Microarchitecture.

Writeback stage

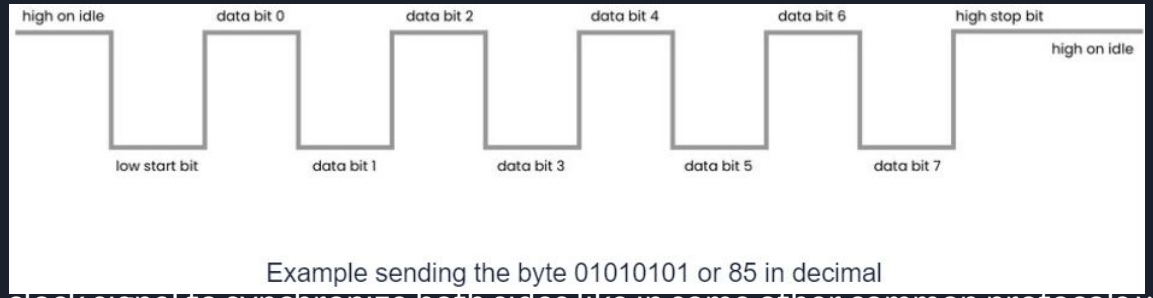
The writeback stage is responsible for writing the data into register.The Instructions, such as store instructions (SW, SH, SB)or branch instructions, do not write back to a register. For these, the write-back stage is effectively skipped as no data needs to be written to the register file.

Single Cycle Representation



UART Protocol Theory

With UART you send data from one device to another over a single wire. You send 1 start bit of data, then an agreed upon amount of data bits (usually 8) and finally a stop bit.
We will be looking at the simple case of 1 start bit, 8 data bits and a stop bit. The data is transferred the least significant bit first.



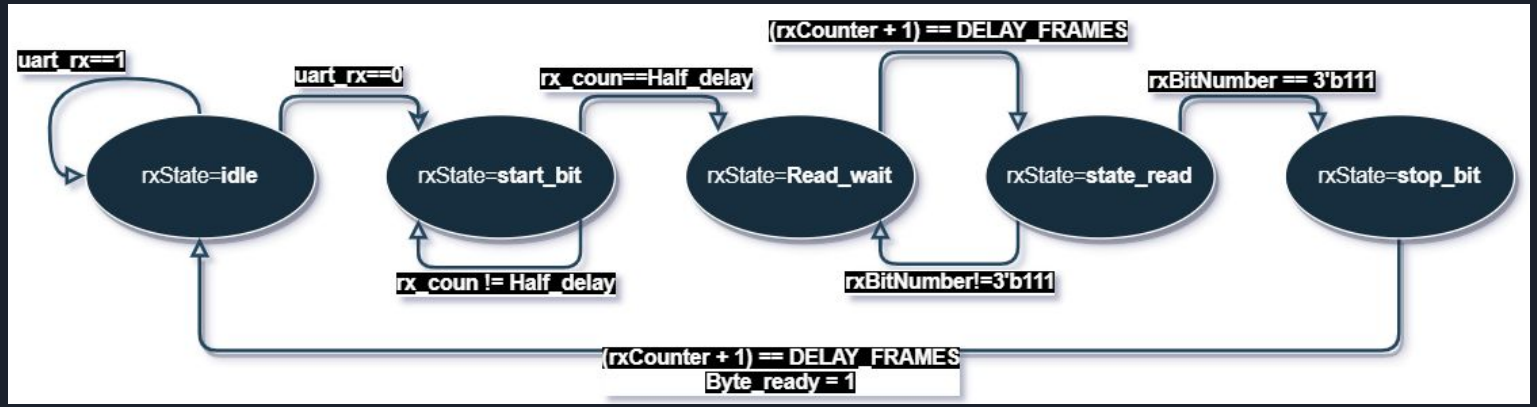
Example sending the byte 01010101 or 85 in decimal

There is no clock signal to synchronize both sides like in some other common protocols where both sides need to agree in advance on a frequency or "baud rate" which is the amount of bits per second and then each side needs to manage their own clocks to meet the desired frequency.

Receive State

We can start in an "idle" state when we see the “start bit” we can start receiving data and go to the "read data" state and then once we finish the bit we can go to the "stop bit" state finally returning back to "idle" ready to receive the next communication.

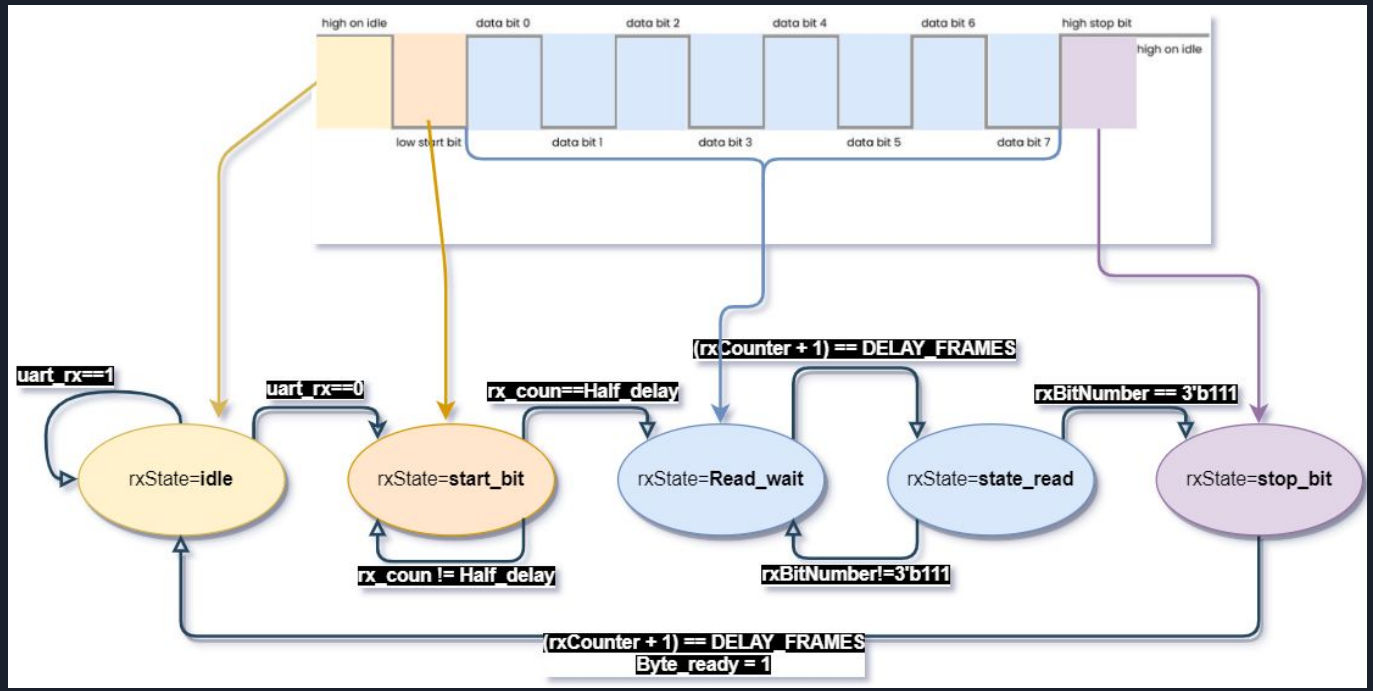
State Machine:



The rxState register can hold in-which state we currently are in
rxCounter for counting clock pulses because baud rates of 115200 bits per second, dividing the 27Mhz by this number = DELAY_FRAME = 234. So we saw 234 clock pulses is 1 UART bit frame
rxBitNumber which can keep track of how many bits we have read.

Receive State

Receive State Waveform analysis



Transmit State

The sending part works a lot like how we set up the receiver earlier, but with a key difference: instead of starting to count from the middle of the pulse, the sending side changes the line right at the start of each bit frame.

Similar to the receiver, the sending side uses a few registers. One keeps track of the transmission process, another counts clock cycles, dataOut holds the current byte being sent, and txPinRegister stores the value for the uart_tx pin. The last two registers help keep tabs on which bit and byte are being sent.

In our example, we're sending a message stored in memory, so we need to know which byte is currently being sent. The "assign" statement links the uart_tx wire to our register, and the last two lines create a memory structure where each cell holds 8 bits, with our example having 12 cells in total.

Moving on, let's define the different stages of our sending process. Unlike the receiver, we don't include an extra "wait" stage because we're not dealing with a middle frame offset. However, we add an extra stage at the end to debounce the button, which we'll use to decide when to send data.

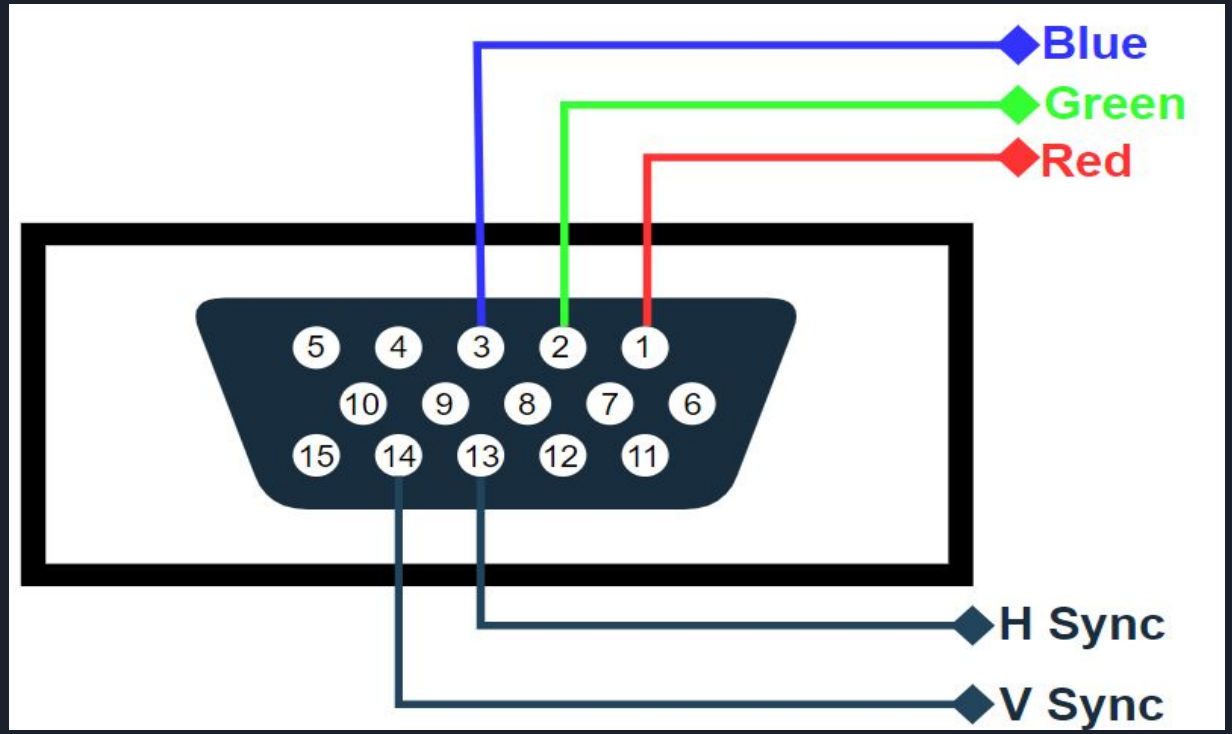
Note.: First, try creating your own state machine diagram and waveform analysis for transmitting data. After that, you can use the source code as a reference on Github ABDUL MUHEET GHANI

Transmit State

As an exercise, now you'll receive data by pressing buttons on the keyboard, and that data should be received by the FPGA, displaying an ASCII pattern on the LED.

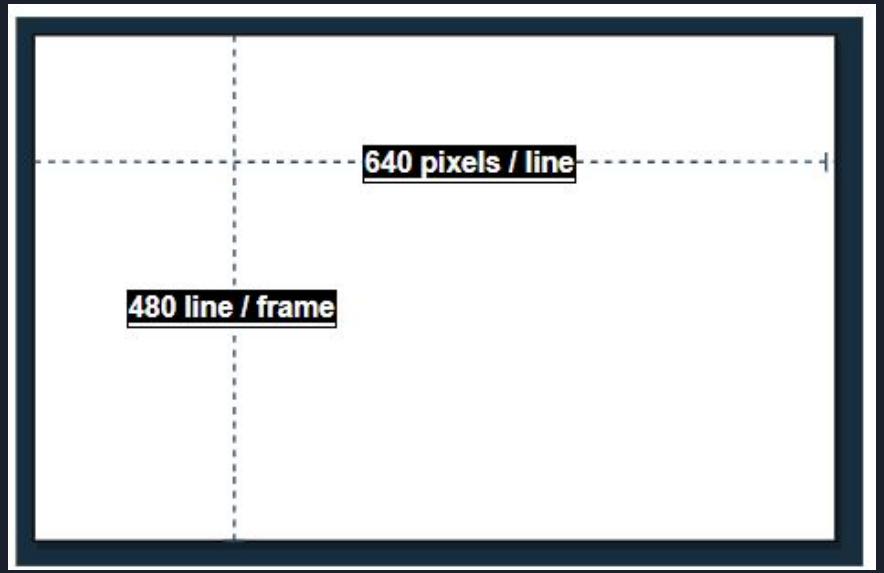
VGA Basics

VGA Pin Configuration

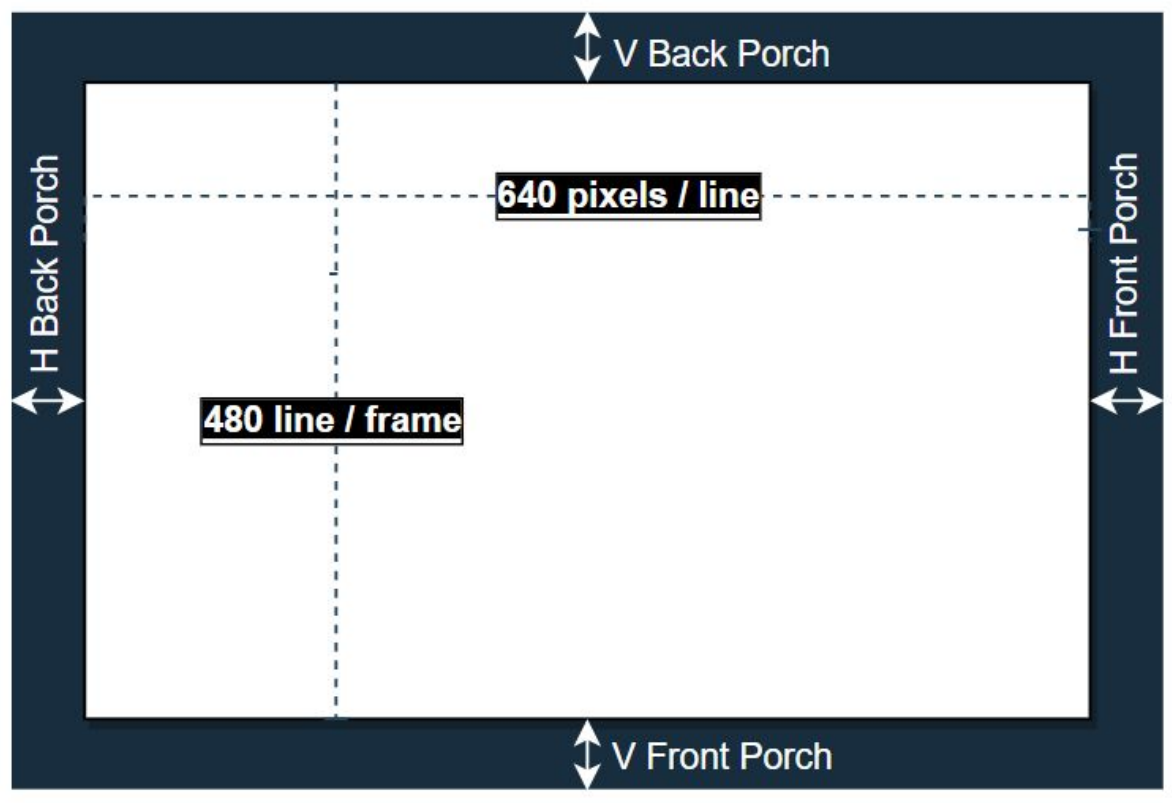


VGA Basics

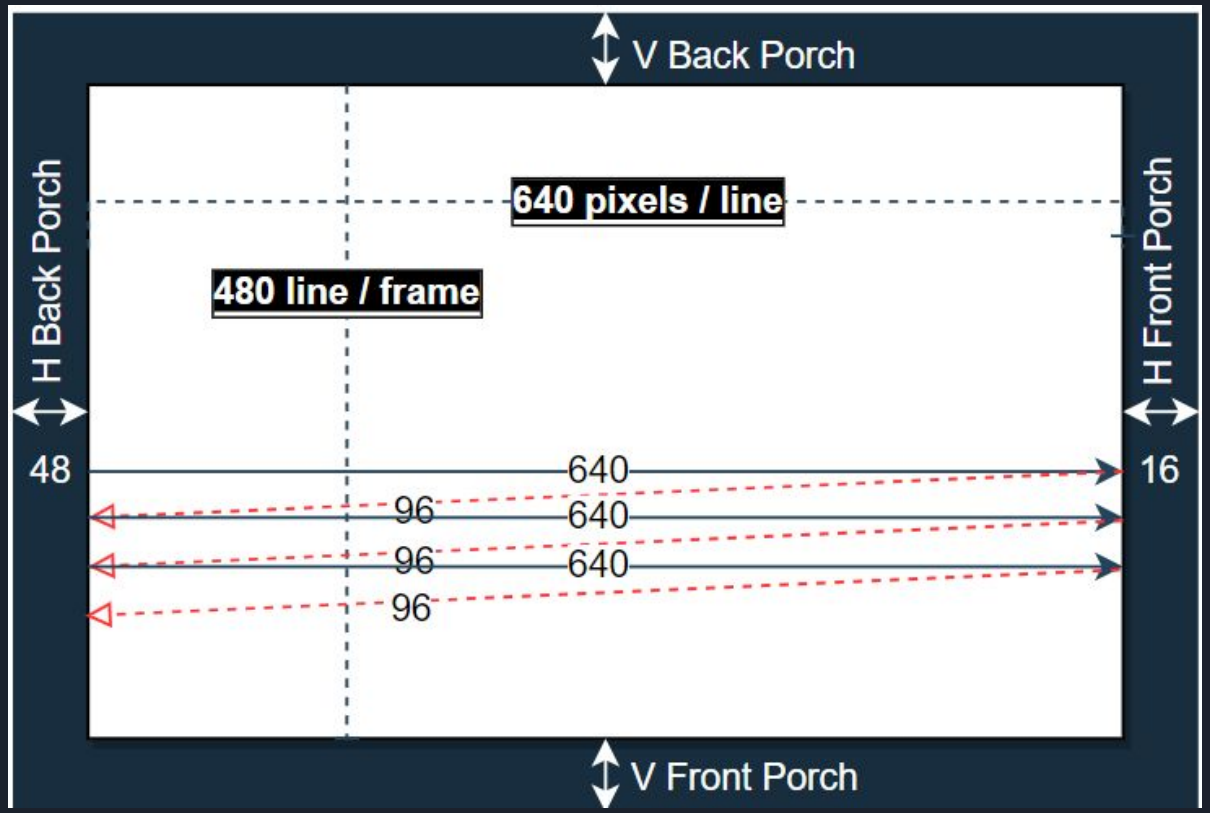
In this example, we are considering a screen resolution of 640x480. All further information will be based on this resolution. First, let's understand what 640x480 means: 640 represents the visible pixels per line, and 480 denotes the number of lines, as shown in the diagram below:



VGA Basics


V Back Porch
H Back Porch
640 pixels / line
H Front Porch
480 line / frame
V Front Porch

VGA Basics

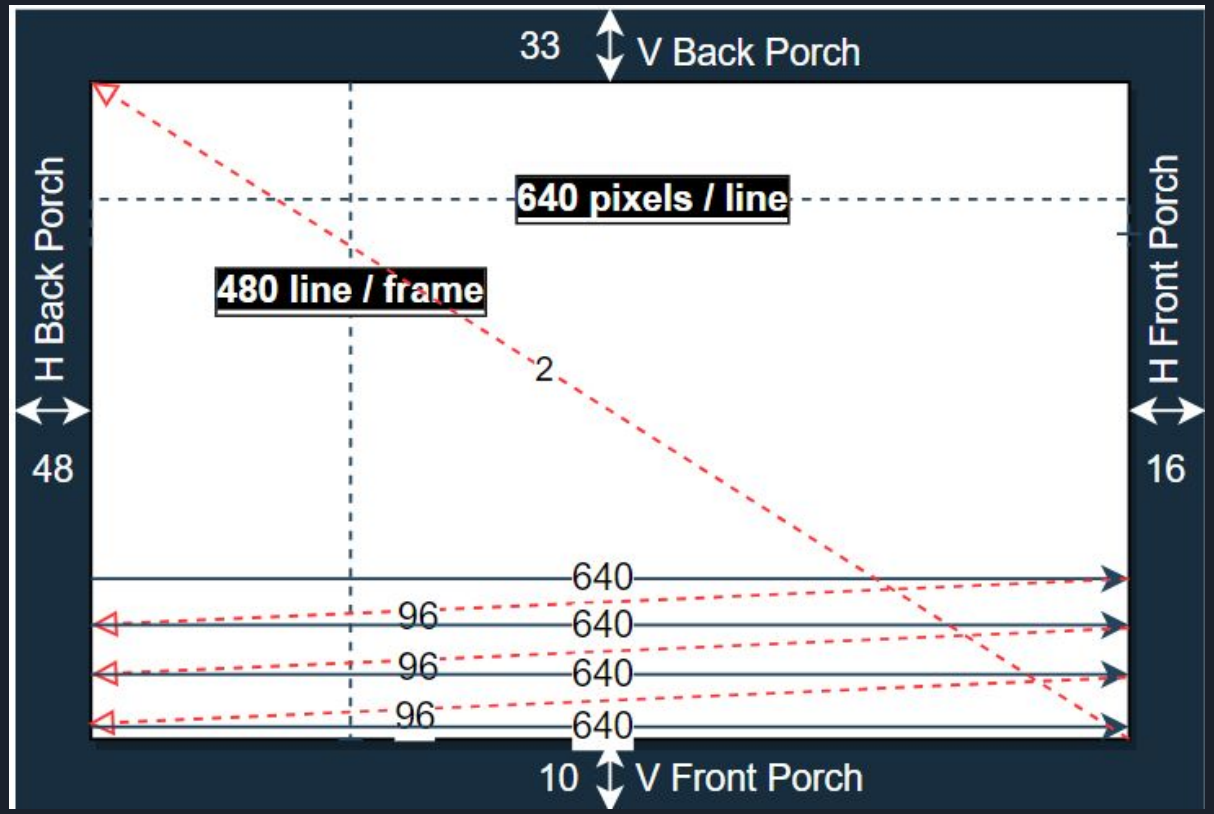

V Back Porch
640 pixels / line
480 line / frame
H Back Porch
H Front Porch
48
640
16
96
640
96
640
96
V Front Porch

VGA Basics

In the previous diagram, we initially calculate the total number of cycles required to complete the pixels for one line. The HSync signal returns to the beginning of the next line, necessitating a horizontal back porch of 48 cycles. With 640 horizontal pixels, 16 cycles are allocated for the horizontal front porch, and an additional 96 cycles are required to return to the start of the next line.

The calculation is as follows:
48 (horizontal back porch) + 640 (horizontal pixels) + 16 (horizontal front porch) + 96 (return to the next line) = 800, which equals one horizontal cycle (1H cycle).

VGA Basics


33
V Back Porch
640 pixels / line
480 line / frame
2
H Back Porch
48
H Front Porch
16
640
96
640
96
640
96
640
10
V Front Porch

VGA Basics

In the previous diagram, we initially calculate the total number of cycles required to complete the 480 lines per frame. The VSync signal returns to the beginning of the frame, necessitating a vertical back porch of 33 cycles. With 480 vertical line, 10 cycles are allocated for the vertical front porch, and an additional 2cycles are required to return to the start of the frame.

The calculation is as follows:
33 (vertical back porch) + 480 (vertical lines) + 10 (vertical front porch) + 2 (return to the beginning of the frame) = 525.

VGA Basics

VGA Horizontal Timing Analysis

In the diagram below, we observe that the first 96 clocks are allocated for returning to the start of the next line. Throughout this entire period, the HSync signal remains at 1. Following this, there's a horizontal back porch of 48 cycles where the HSync signal switches to 0. This marks the start of the visible display for 640 clock cycles, during which we can modify the values of red, green, and blue. Afterward, the horizontal front porch begins for 16 clocks, during which time both HSync and RGB return to low again.

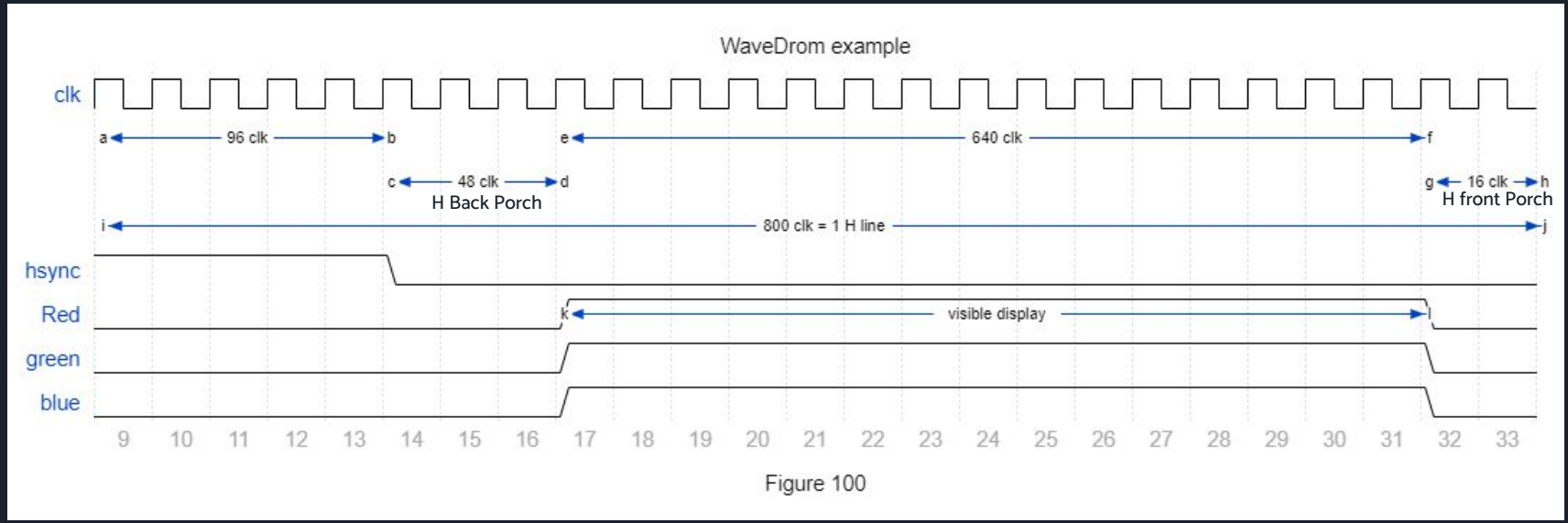


Figure 100

VGA Basics

VGA Vertical Timing Analysis

In the diagram below, we observe that the first 2 hsync are allocated for returning to the start of the frame. Throughout this entire period, the VSync signal remains at 1. Following this, there's a vertical back porch of 33 hsync where the VSync signal switches to 0. This marks the start of the visible display for 480 hsync cycles, during which we can modify the values of red, green, and blue. Afterward, the vertical front porch begins for 10 hsync, during which time both VSync and RGB return to low again

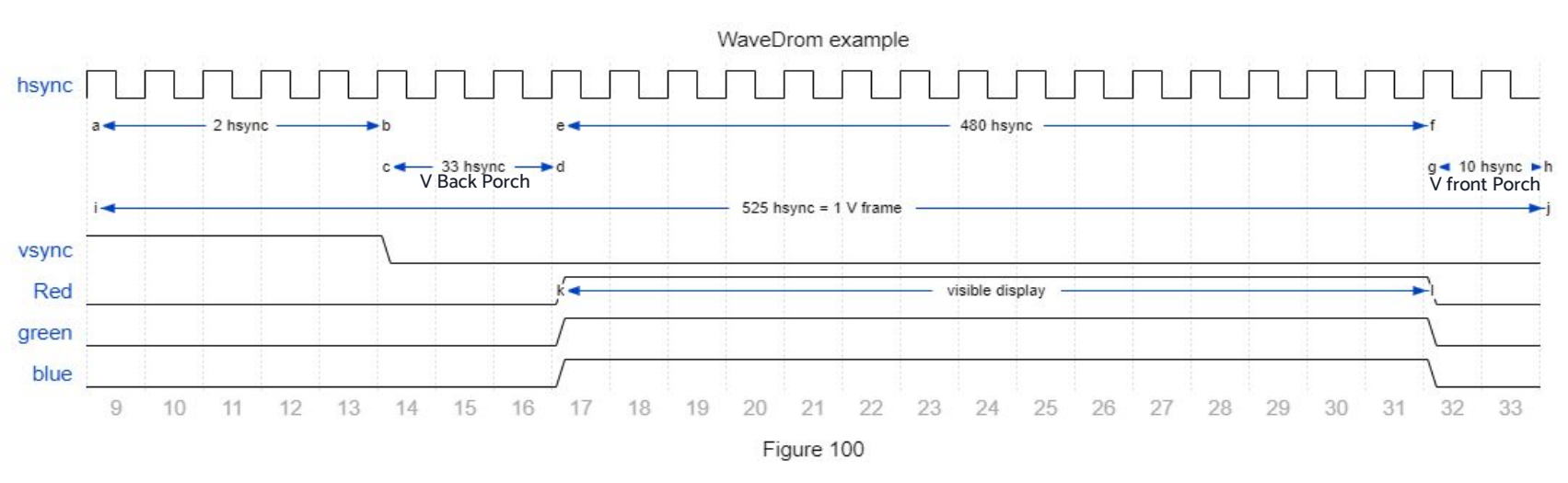


Figure 100

VGA Basics

Task

The task is to display a basic output on the screen. The screen will be divided into three sections vertically. With 480 pixels along the y-axis, dividing it by 3 gives 160 pixels per section. The display will showcase red from 1 to 160, green from 161 to 320, and blue from 321 to 480, just as illustrated in the diagram below. Clk frequency should be 25Mhz. For tang nano 9k default clock is 27MHz so we need to convert it to 25Mhz clk frequency by adding PLL and clk divider IP from gowin.

Source Code Reference Github Abdul Muheet Ghani

TESTIMONIAL

Author:

Abdul Muheet Ghani, Research Associate at MERL-UITU.

Under The Supervision Of:

Dr.Ali Ahmed (Team Lead MERL).

Sponsored By: Edmund from Symbiotic EDA. for sponsoring FPGA.

Thanks: Lushay Lab (They've provided crucial resources and guidance throughout the project)