Mario Game in FPGA

1. Project Background

This is the program designed for 2025 FPGA class (Fudan University), built by Zhu Yanxiang, Shao Ziyue, Zhu Luyu, Zhu Dingchen, Han Yixuan. Code source is in github: https://github.com/Yanxiang-ZHU/FPGA_MARIO.git.

The goal of the project is to design a *game machine based on FPGA*, interacting with peripheral devices like *vga, keyboard and mouse*.

2. Introduction to the Project

2.1 Project Introction

This design implements a 2D side-scrolling Mario game based on FPGA, featuring classic gameplay such as character movement, *jumping, collecting gold coins, and destroying obstacles*. Players control the Mario character using both *buttons and the mouse*, navigating through a scrolling map while avoiding enemies (as the new_block module in the code implements the logic for monster collision). They score by toppling bricks (triggered when block_in == GY), and achieve multi-level jump height control through the GameEngine module. The game objective is to collect a specified number of gold coins (counted by the coins register) and reach the end of the level. The system displays the life value, gold coin count, and current level in real time using the MarioScore24x1 module (however, only one level was designed in this project).

2.2 Project Development Platform

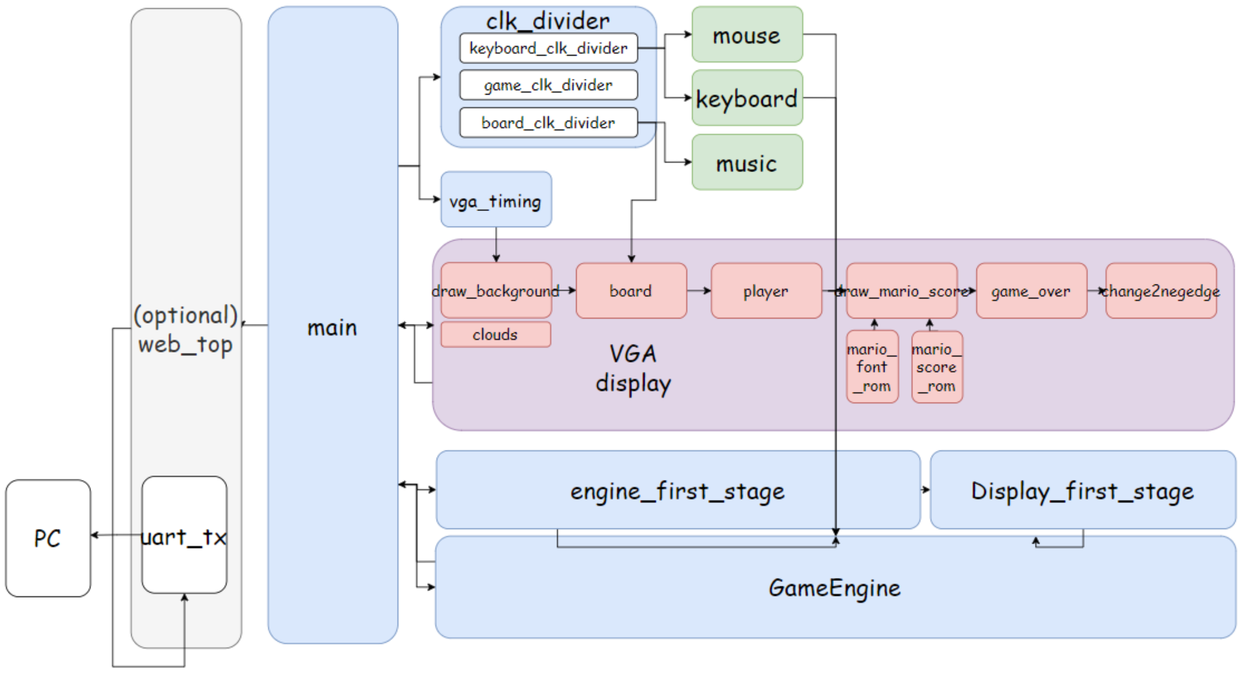
Hardware Platform: *Xilinx Artix-7 FPGA* (also attempted to use Altera DE2-115)

Development Tools: *Vivado 2023.2* (also attempted to use Quartus)

Hardware Expansion:

1. *VGA*: Connected to the FPGA through the following ports: vga_red, vga_green, vga_blue (for color signals), vga_hsync, vga_vsync (for synchronization signals). This setup enables the display of real-time graphical output such as images or UI interfaces on a monitor.
2. *Mouse*: Connected to the FPGA via the PS/2 interface using ps2_clk and ps2_data pins. The mouse provides user input support for pointer control or menu selection in the system.
3. *Buzzer*: A passive buzzer is connected to the FPGA. It is controlled by a square-wave signal generated by the FPGA to produce various tones for game sound effects or system notifications.

3. Module Architecture



3.1 Module Interface and System Architecture

The main module represents a complete Mario-style game implementation that integrates VGA graphics, audio synthesis, and input processing on a single FPGA platform. The interface design follows conventional gaming system architecture with dedicated clock and reset inputs, PS/2 peripheral support, and multi-bit VGA color outputs that provide 4096 distinct colors through 12-bit RGB encoding.

Input processing centers around a comprehensive control scheme that captures both traditional arrow-key navigation and modifier keys including space, shift, control, and escape. The PS/2 interface enables both keyboard and mouse input through dedicated clock and data lines, while the central button serves as a hardware reset mechanism. Audio output utilizes a single-wire PWM speaker interface that generates chiptune-style music and sound effects characteristic of retro gaming platforms.

3.2 Graphics Pipeline and Rendering System

The graphics subsystem implements a six-stage pipeline that transforms *geometric primitives into VGA-compatible signals*. Beginning with a 25MHz pixel clock generator, the VGA timing module produces standard 640×480 resolution with proper synchronization signals. The background renderer creates scrolling parallax effects through coordinate transformation, while the board renderer dynamically places game elements using dual-port RAM structures that enable real-time level modification.

Player sprite rendering utilizes *ROM-based* bitmap storage with support for directional animation and scaling. The score display system employs bitmap font rendering to overlay game statistics, lives, and level information. A final signal conditioning stage converts the pipeline output to negative-edge VGA signals, ensuring compatibility with standard display hardware while maintaining 60Hz refresh rates.

3.3 Game Engine and Memory Architecture

The core game engine operates at 600Hz to provide smooth physics simulation and responsive input handling. This high-frequency update cycle enables precise collision detection, gravity simulation, and player movement that feels natural despite the discrete nature of digital implementation. The engine interfaces with a sophisticated memory hierarchy that includes sprite ROMs, level data RAMs, and font storage systems.

Level management employs a dual-instantiation approach where one FirstStage module handles dynamic game logic while another provides read-only access for display rendering. This architecture prevents rendering artifacts during level modifications while maintaining deterministic timing. The distributed memory system utilizes multiple clock domains to optimize bandwidth usage, with a 100MHz system clock driving game logic and a 50MHz board clock managing display updates.

3.4 System Integration and Performance

Clock management represents a critical aspect of the design, with a *phase-locked loop generating multiple synchronized frequencies from a single input reference*. The 25MHz pixel clock drives VGA timing, while 100MHz and 600Hz derivatives handle processing and game updates respectively. This multi-domain approach maximizes performance while maintaining signal integrity across subsystem boundaries.

The modular architecture enables independent verification of graphics, audio, and control subsystems while providing clear interfaces for future expansion. Debug capabilities include LED status indicators and optional UART communication for development support. The complete system demonstrates effective resource utilization with pipelined rendering, shared memory structures, and optimized clock distribution that achieves real-time performance on modest FPGA hardware.

4. RTL Implementation

4.1 main.v

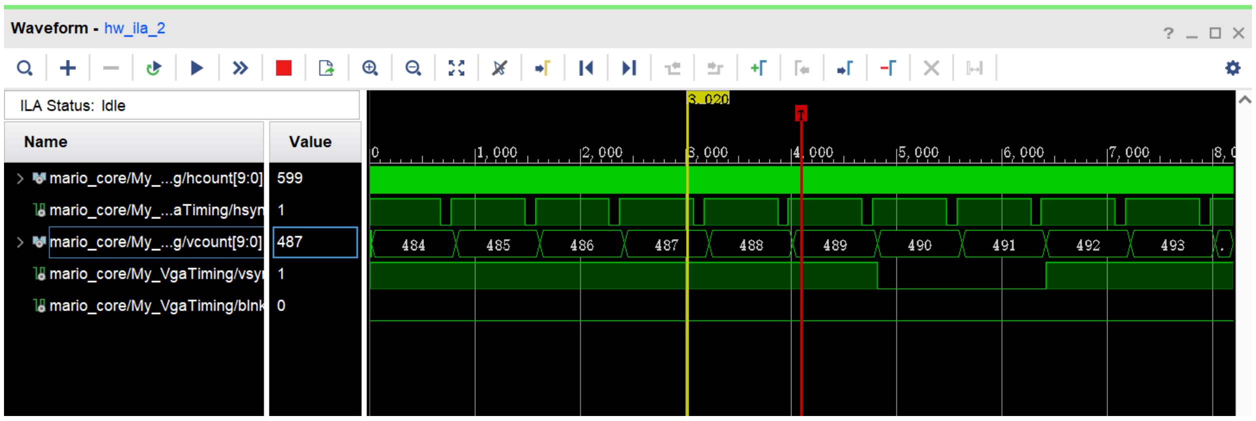
An FPGA-based top-level system for the classic Mario game is implemented. The game logic operations and image rendering are done on a Xilinx Artix-7 FPGA through a multi-stage pipelined architecture and hardware parallelisation design. The core of the system consists of a chain of five layers of modules: VGATiming generates *640x480@60Hz base signals;* Dynamic Background Renderer (DrawBackground) realises scrolling maps; Map Block Processing Module (Board) reads real-time map data through dual-port RAM; Character Rendering Engine (Player) combines with The character rendering engine (Player) implements Mario's movements in conjunction with sprite animation data stored in ROM; UI overlays (DrawMarioScore and Gameover) handle the scoring and end screens. The game logic is controlled by a state machine-driven physics engine (GameEngine), which updates the character movement, collision detection and life management system through a 600Hz clock, with a coordinate mapping formula plane_xpos*180 + plane_xpos_ofs to achieve precise addressing of the map tiles, and dual-clock-domain communication (25MHz rendering clock and 100MHz logic clock) for real-time communication. 100MHz logic clock) to ensure real-time response.

4.2 clk_divider.v

The clk_divider module is a parametric clock divider whose core function is to divide a 100MHz input clock (clk100MHz) to generate an output clock (clk_div) at a user-defined frequency (FREQ parameter). Through the counter cycle count (count register), when the count value reaches the preset LOOP_COUNTER_AT threshold (dynamically calculated by the FREQ parameter), the output clock level will be flipped to achieve a 50% duty cycle low frequency signal output. For example, when FREQ=100, 100Hz square wave will be output. rst reset signal will force the counter and output clock to be cleared to ensure the initial state of the frequency divider is stable.

4.3 vga_timing.v

This module implements a VGA 640x480@60Hz standard timing generator to provide industry standard drive timing for VGA displays by precisely controlling the *horizontal/vertical counters and synchronisation signals*. The module employs a dual counter architecture (hcount and vcount) to dynamically generate low-level active line sync signals (hsync stays low in the 656-752 pixel range) and field sync signals (vsync stays low in the 490-492 line range) based on a preset 800-pixel horizontal and 525-line vertical total period, while at the same time generating a low-level active line sync signal through the The hblnk and vblnk signals activate the fade control outside the valid display area (outside the 640x480 range) to ensure that RGB data is only output in the valid pixel range.



Name	Value
mario_core/My_...g/hcount[9:0]	599
mario_core/My_...aTiming/hsyn	1
mario_core/My_...g/vcount[9:0]	487
mario_core/My_VgaTiming/vsy	1
mario_core/My_VgaTiming/blnk	0

We generated the working waveform diagram of VGA, as shown in the above figure. The hcount is constantly cycling rapidly, while the vcount increments by 1 after the hcount completes one row of pixels. When the vcount reaches the bottom row of pixels of one frame of image, this vsync flips to the low level for a certain period of time.

4.4 drawbackground.v

The drawbackground module is the background rendering controller in Mario games, and its core function is to generate composite visual effects of gradient skies and dynamic clouds on the FPGA screen. The module receives the VGA timing signals (hcount_in/vcount_in) and the horizontal offset (xoffset), and divides the screen area by the YRES - vcount_in - 1 < OFFSET condition: the upper half (Y-coordinate < 100) is rendered by the Clouds sub-module as clouds, and the lower half by the red and green channels (r = vcount_in >> 1), which are dynamically computed in vertical coordinates. green channels (r = vcount_in >> 1, g = vcount_in >> 1), and fixing the blue channel to its maximum value (b = 8'hff), creating a gradient sky from white at the top to blue-violet at the bottom. Cloud rendering is achieved by instantiating the Clouds module, using xoffset to control the horizontal displacement of the clouds, and ultimately overlaying the cloud colour (rgb_cl_out) on top of the gradient sky, and passthrough the sync signals (hsync_out/vsync_out) to ensure the correct display timing.

4.5 clouds.v

The clouds module is a cloud rendering controller for the Mario games. Its core function is to generate moving cloud effects on the FPGA screen through dynamic coordinate mapping and geometry detection. The module receives the raw pixel coordinates (hcount_in/vcount_in) and horizontal offset (xoffset) from the VGA timing generator, dynamically calculates the horizontal position of the clouds (xpos) from (hcount_in + xoffset) % XRES, and combines them with the predefined coordinates of the centre of the clouds (CLOUD_MAP_X/Y), radius (CLOUD_MAP_X/Y), and the centre of the clouds (CLOUD_MAP_X/Y). The cloud position (xpos) is dynamically calculated and combined with the predefined cloud centre coordinates (CLOUD_MAP_X/Y), radius (CLOUD_MAP_S) and colour (CLOUD_COLOR=24'h88_99_cc) to detect whether the current point belongs to a particular cloud on a pixel-by-pixel basis, using the circular region determination metric (xpos - X)^2 + (YRES-vcount-Y)^2 < S^2 . If the condition is met, the RGB output is set to a light blue cloud, otherwise the original input colour (rgb_in) is passed through, and the line-field sync signals (hsync_out/vsync_out) are passed through to ensure the correct timing of the picture.

4.6 board.v

The board.v module is the display controller for the Mario game scene and is responsible for mapping the game's logical coordinates to the screen pixels and rendering the corresponding colours. The module receives the raw pixel coordinates (hcount_in/vcount_in) from the VGA timing generator, dynamically calculates the block coordinates (block_xpos/block_ypos) corresponding to the current pixel by combining the player's horizontal position (plane_xpos) and offset (plane_xpos_ofs), and selects the corresponding 24-bit RGB colour value (e.g. block) according to the block type (block) via the case The block coordinates (block_xpos/block_ypos) of the current pixel are dynamically calculated, and the corresponding 24-bit

RGB colour value (e.g. light blue for square A, yellow for gold coin block GY) is selected according to the block type through the case statement, and then the processed video signal (rgb_out) and the synchronisation signals (hsync_out/vsync_out) are synchronised and outputted to the display. The module adopts a dual clock domain design: pclk is used for pixel-level signal pipelining, clk is used for coordinate calculation logic, and the reset signal (rst) clears all display states to ensure that the screen is initialised correctly.

4.7 player.v

This module implements a Mario game character rendering controller, which is responsible for generating the player character pixel data in real time and compositing it with the background image. The module dynamically switches the height of the character (40 or 80 pixels) via the size parameter, combines it with direction signals to control the character's orientation (left/right), and reads the pre-stored character sprite map data using rom_data. Core functions include: calculating ROM address based on xpos/ypos coordinates and scan position (hcount_in/vcount_in) to achieve pixel-level positioning of the character at the specified position on the screen; and pixel blending with the background rgb_in through ALFA_COLOR transparent colour detection (24'hA3_49_A4) to override the character colours in non-transparent areas only. Character colours. Video timing signals (hsync_in/vsync_in) are directly transmitted to maintain display synchronisation, and character height is switched without glitches via the player_height register to ensure a stable screen when the character's form changes.

4.8 draw_mario_score.v

The draw_mario_score module is a score and status information renderer for Mario games. Its core function is to draw Mario's life value, level progress and score letters in real time in a fixed area in the upper left corner of the screen (coordinates XPOS=40, YPOS=50, size WIDTH=552xHEIGHT=16). The module receives pixel data (char_pixels) from the character generator, converts the horizontal coordinates to character indices (char_xy) by bitwise operations ((hcount_in - XPOS -1) >> 3), and calculates the vertical line number (char_line = vcount_in - YPOS) in order to locate specific pixel positions in the character bitmap. When the current pixel is detected as belonging to the character valid area and the bit corresponding to char_pixels is high, the original input colour (rgb_in) is overridden by either yellow (24‘hff_ff_00 for highlighted life values) or white (24’hff_ff_ff for regular text) depending on the type of the character, and the synchronisation signal (hsync_out/ vsync_out) is transmitted to maintain the original input colour (rgb_in). vsync_out) to maintain VGA display timing.

4.9 gameover.v

The module implements a GAME OVER screen rendering system for Mario games, using VGA timing control and character dot matrix mapping technology to override the original screen when a game over signal is detected, and display the word ‘GAME OVER’ in white at a specified location on the screen. The module maps the character pixels to the hcount_in/vcount_in coordinates of the VGA output by using a predefined 11x10 dot matrix template (e.g. G_LETTER array) with 8 letters (G/A/M/E/O/V/R), combined with the calculation of the centre co-ordinates (X1_CENTER=170, Y1_CENTER=180, etc.) and the size scaling parameter (SIZE1=9). vcount_in coordinate space. When the end-of-game signal is activated, the module traverses the display area of the eight letters and dynamically calculates the scaled pixel position ((X1_OFFSET - hcount_in)/SIZE1) through the divider, and outputs a white colour (24'hffffffff) if it hits the “1” bit of the character dot matrix, or outputs a black background otherwise, to achieve the hardware-level anti-aliasing effect. The design keeps the VGA timing synchronised by pass-through signaling (hcount_out <= hcount_in) to avoid screen tearing.

4.10 change2negedge.v

The change2negedge module is a clock falling edge triggered sync buffer used to latch the input video sync signals (HSYNC/VSYNC) and RGB pixel data to the output on the falling edge of the clock. Its core function is to pass the input signals directly to the outputs (e.g. hsync_out <= hsync_in) by negedge clk triggering and to force all outputs to be cleared on reset (rst) to ensure that the display signals remain blacked out during system initialisation or abnormal states.

4.11 first_stage.v

This module is the map data control core of the Mario game, which implements dynamic map interaction and backup recovery mechanism through a three-level state machine. In normal mode (NORMAL_MODE), the player coordinates plane_ypos(4-bit) and plane_xpos(8-bit) are converted to the 12-bit RAM address (11-plane_ypos)*180 + plane_xpos, which realises the linear mapping of the 2D game coordinate system to the storage medium. Determine 34 predefined block types (e.g. obstacle A/D, prop S, etc.) by reading ram_read_data, and trigger blocking=1 collision signal when blocking class block is detected. Backup mode adopts mirror copy chain (ROM_STAGE_SIZE=2160) to write backup memory data into main RAM address by address, supports instant reconstruction of map state after game reset or death, combines with save_block signal to update map changes such as destroying blocks/props pickups in real time, and controls dual-port RAM synchronous operation through ram_we write enable signal. The dual-port RAM synchronisation is controlled by ram_we write enable signal, which guarantees 60Hz screen rendering without any lag. Hardware-level state switching mechanism (START_BACKUP→BACKUP) through the address counter saving_addr to achieve the full amount of data migration, the completion of the backuped flag is set, the design makes full use of the FPGA parallel bus characteristics to achieve a map recovery delay of less than 1ms.

4.12 game_engine.v

This Verilog module, as the *core control engine* of Mario game, realises the deep integration of player action response, physics simulation and game logic through a five-level state machine architecture. The module adopts a dual clock domain design (clk for logic processing and game_clk to control 60Hz screen refresh), combines player coordinates (player_xpos/ypos) with map offset (plane_xpos_ofs), and locates the player in real-time through a grid-based coordinate mapping (plane_xpos + (player_xpos + plane_xpos_ofs)/40). xpos_ofs)/40) to locate the map block where Mario is in real time, triggering collision detection (blocking_in signal) and cube destruction logic. Horizontal movement state machine (PL_LEFT/PL_RIGHT) combined with speed grading (PL_WALK/PL_RUN) to achieve delicate manipulation, and vertical state machine (PL_JUMP/PL_FALL) to simulate jumping trajectory through jump_height counter, and at the same time, use player_xpos % 40 to detect the landing state, forming a physics feedback Mechanism. The game state management module achieves life loss by decreasing player_life (cliff fall detection player_ypos == 0), double buffering of player_points to avoid refresh conflicts, and integrating a map backup and recovery system (restartgame signal triggers a position reset synchronised with stage_restarted). Background scrolling algorithm (bcgr_xpos offset calculation) and map board switching logic (MAX_OFFSET=39 boundary protection) to achieve multi-layer parallax effect, hardware-level timing optimisation (clk_hor_divider frequency division) to ensure smooth rendering on the Basys3 development board. The enemy AI module records the 8-direction neighbouring block status through modi_block register, and combines with new_block to dynamically generate the destruction effect.

4.13 mario_font_rom.v

An 8x16 pixel game-specific font memory was implemented as the core display driver module for the Mario game UI system. The 1280 pixel dot matrix data (16 rows x 8 columns per character) of 20 customised characters are mapped via a 12-bit address bus (high 8-bit character encoding + low 4-bit line numbering), with a combinational logic lookup mechanism for zero-latency reads. The module is pre-programmed with digital pipe style numbers (e.g. 7-segment display structure for character 0x00), game symbols (e.g. heart icon 0x0a) and Mario logo (special pixel arrangement of 0x10-0x14). Hardware level bitmap storage is achieved by case(addr) statement, and each 8-bit output corresponds to horizontal pixel lighting state in the VGA scan line. This module works with the UI controller of the game engine to convert the character code to screen position through the coordinate conversion of the upper module, and with the dynamic scanning and refreshing mechanism to achieve the stable display of game information such as ‘LIFE’ and ‘WORLD’.

4.14 mario_score24x1.v

This module implements a dynamic scoreboard controller for Mario game, which drives the screen display system to update by converting the game status data (number of lives, number of coins, level level) into character encoding in real time. The module adopts a combinational logic processing mechanism to decompose the input 12-digit number of coins into 3-digit decimal numbers by BCD algorithm, and maps them to the preset character code table based on the character coordinates (char_xy) to dynamically generate the ‘Mario’ logo, life icon (e.g., 8’h07 maps the number of lives), coin statistic (8’h23-8‘h25 maps the hundreds/tens/indices) and level information (e.g., 8’h25 maps the hundreds/decades/individual digits). and level information (e.g. 8‘h3e-8’h44 shows ‘LEVEL’). The core of the UI is designed to realise the character space layout through multi-level case statements. Combined with the parallel processing feature of Mario chip described in webpage 1, it ensures real-time output of corresponding pixel data during VGA scanning, and completes the visualisation of the game state information.

4.15 music.v

This module implements a Super Mario theme song player that drives a buzzer sound through a combination of ROM sheet music storage, crossover control and square wave generation techniques. The core contains three functional layers:

Sheet music storage: *the music_rom module uses a lookup table method to store a 54-bar note sequence* (e.g., 8'd34 corresponds to middle C), and reads the sheet music data cyclically by incrementing the address counter;

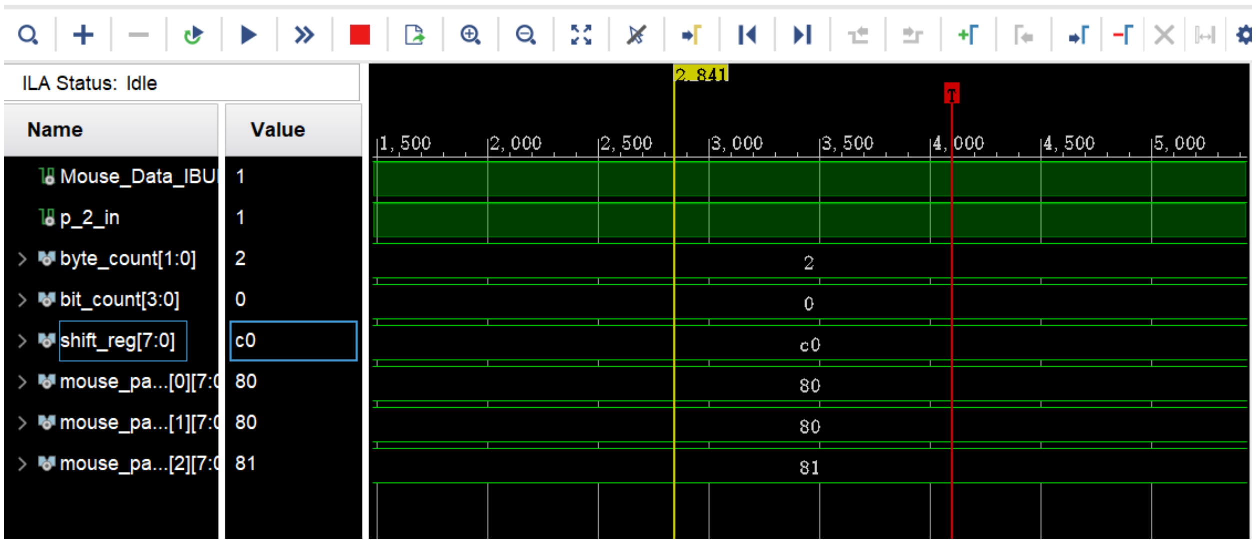
Tone generation: the divide_by12 module decomposes the note encoding into octaves (octave) and scales (note), and generates the target frequency by combining the 12 crossover coefficients of clkdivider (e.g., 9'd511 corresponds to the A tone at 440Hz);

Timing control: the high 9 bits of the 32-bit counter tone drive ROM address switching, the low 22 bits combine counter_note and counter_octave to achieve note beat control (about 1/4 second per beat), and finally output a square wave signal to the speaker pin to drive the buzzer. The module implements inter-note rest via the tone[21:18]! = 0 condition for inter-note rest control, reproducing the hardware-level synthesis of the game's classic melody.

4.16 new_block.v

This module implements a dynamic cube state transition controller, designed for Tetris-type games, which handles cube interactions in real time through combinatorial logic. *The module decides whether to generate a new block, trigger a score, or modify the block type based on the current block type* (e.g., gold GY, destructible D), direction of movement (direction), and relative coordinates (relative_xpos/relative_ypos). For example, when a gold coin cube is detected (block_in == GY), the base cube B is generated and marked with the scoring signal new_point; the destructible cube D is converted to the destructive state DY based on the positional interval (20 pixel division) and the direction signal.

4.17 Mouse.v



Name	Value
Mouse_Data_IBU	1
p_2_in	1
byte_count[1:0]	2
bit_count[3:0]	0
shift_reg[7:0]	c0
mouse_pa...[0][7:0	80
mouse_pa...[1][7:0	80
mouse_pa...[2][7:0	81

The PS/2 mouse operates by transmitting data packets to the FPGA via two lines: Mouse_Clk and Mouse_Data. Each mouse action—such as moving or clicking—triggers the mouse to send a 3-byte packet. The FPGA captures this data on the falling edge of the Mouse_Clk signal, collecting one bit at a time into a shift register. Once 11 bits are received (including start, data, parity, and stop bits), a full byte is formed and stored. After three such bytes are received, the FPGA decodes the first byte to determine the status of the left, right, and middle buttons. Based on this status, it updates internal flags and performs simple logic—such as incrementing or decrementing a displayed number depending on mouse clicks. We only *use the mouse to click* in our project (just like the button logic).

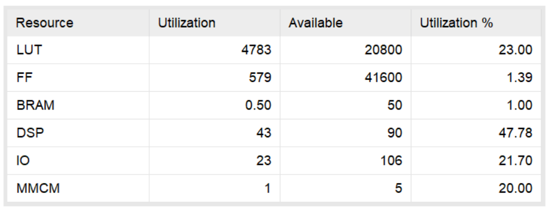
4.18 Keyboard.v

Initially we planned to use keyboard to control the game, but sadly found that our keyboard not support the PS2 agreement. So it appears like whatever key button we click, nothing happens to the game. Even thought, we will still introduce the keyboard module here.

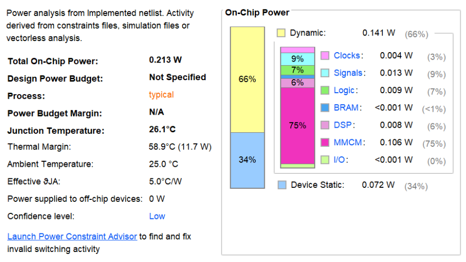
It also captures serial data from the keyboard clock (PS2_CLK) and data (PS2_DATA) lines, reconstructs 11-bit scan codes, and interprets them to identify specific key presses and releases. Our module supports modifier and control keys such as L_ALT, R_ALT, L_CTRL, R_CTRL, SPACE, L_SHIFT, R_SHIFT, and navigation keys like ESC, D_ARROW, L_ARROW, and R_ARROW. It handles extended and break codes defined in the PS/2 protocol to distinguish between left/right keys and press/release events. A state machine monitors the clock edges and accumulates bits, checking for errors through parity. Upon receiving a full scan code, the module updates output signals accordingly.

5. Project Result

We can analyze the result from synthesis and implementation (including placing and routing).



Resource	Utilization	Available	Utilization %
LUT	4783	20800	23.00
FF	579	41600	1.39
BRAM	0.50	50	1.00
DSP	43	90	47.78
IO	23	106	21.70
MMCM	1	5	20.00



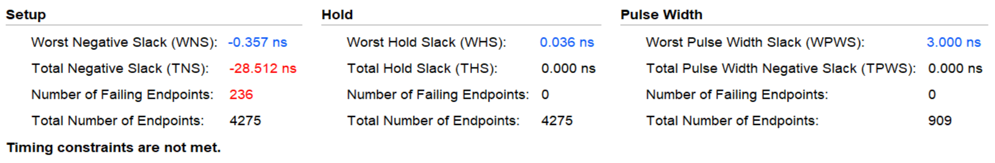
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.213 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**26.1°C**
Thermal Margin:	58.9°C (11.7 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		0.141 W	(66%)
	Clocks:	0.004 W	(3%)
	Signals:	0.013 W	(9%)
	Logic:	0.009 W	(7%)
	BRAM:	<0.001 W	(<1%)
	DSP:	0.008 W	(6%)
	MMCM:	0.106 W	(75%)
	I/O:	<0.001 W	(0%)
Device Static:		0.072 W	(34%)



Setup

Worst Negative Slack (WNS):	-0.357 ns
Total Negative Slack (TNS):	-28.512 ns
Number of Failing Endpoints:	236
Total Number of Endpoints:	4275

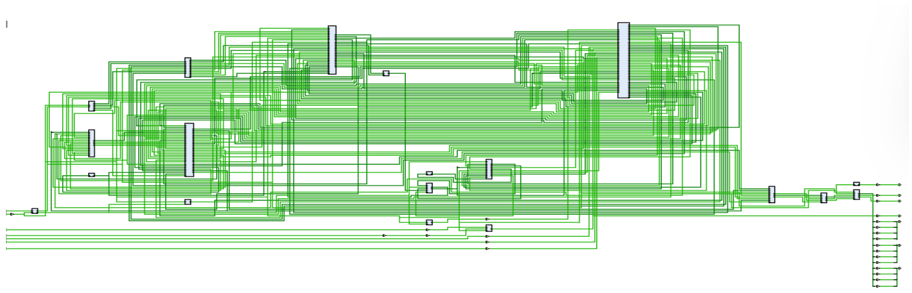
Hold

Worst Hold Slack (WHS):	0.036 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	4275

Pulse Width

Worst Pulse Width Slack (WPWS):	3.000 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	909

Timing constraints are not met.

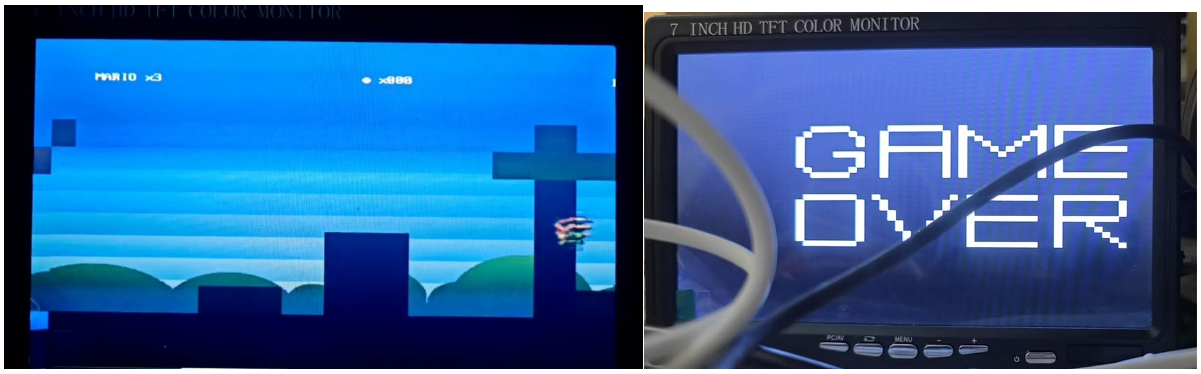


The resource utilization chart indicates that the design consumes 23.00% of LUTs (4783 out of 20800), 1.39% of BRAM (0.50 out of 50), 47.78% of DSPs (43 out of 90), 21.70% of I/O (23 out of 106), and 20.00% of MMCM (1 out of 5). The significant DSP usage (47.78%) suggests heavy reliance on digital signal processing blocks, which could be optimized if the design permits. *LUT utilization at 23.00% leaves room for additional logic if needed*, while the low BRAM usage indicates minimal memory-intensive operations.

Timing analysis reveals that timing constraints are not met, with a worst negative slack (WNS) of -0.357 ns and a total negative slack (TNS) of -28.512 ns across 236 failing endpoints out of 4275 total endpoints. Hold slack is met with a WNS of 0.036 ns and no failing endpoints, and pulse width slack is also satisfactory with no failures. The negative timing slack indicates that the design operates slower than the specified clock frequency, necessitating adjustments such as pipelining or reducing combinational path delays.

Power analysis shows a total on-chip power consumption of 0.213 W, with 66% dynamic power (0.141 W) and 34% device static power (0.072 W). The dynamic power is dominated by MMCM (75% or 0.106 W) and DSPs (6% or 0.008 W), suggesting that clock management and DSP operations are significant power contributors. The junction temperature of 58.9°C with an ambient temperature of 25.0°C indicates a thermal margin of 5.0°C/W, which is within safe limits but warrants monitoring under sustained operation.

The placement layout visualizes a dense interconnect structure, with green lines representing routed nets. The complexity suggests efficient resource allocation, though congestion in certain areas could contribute to the timing issues. Optimizing placement or rerouting critical paths may improve timing performance.



This picture *shows the game graphics of the Mario game*. It has a total of three lives. Currently, it hasn't eaten the gold yet. Eating the gold will increase the count, and then different blocks will appear in the air. Some blocks can be knocked down from below. If you fall into the river, you will die and the game will restart from the refresh point. If all three lives are lost, *the screen will display "Game Over"*.

* The game video can be seen in video platform bilibili: **https://b23.tv/RugymKW**.

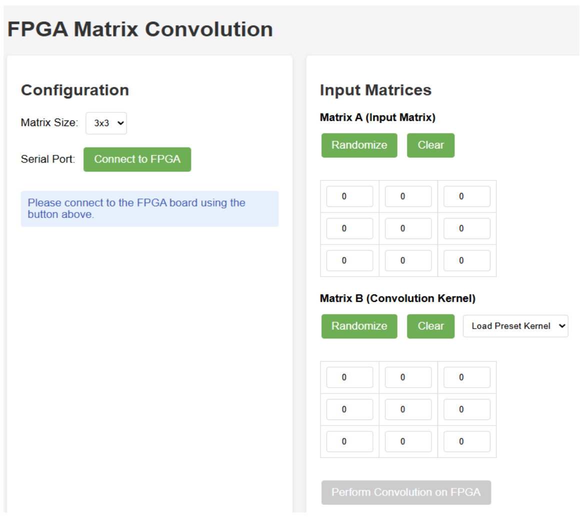
6. Other Attempts

6.1 The web server for Matrix Computing in FPGA

We also attempted to *combine the problem 2*, which required us to *construct a web server for FPGA*. For visualization, we coded for a simple matrix computing module to give the feedback as we should have several results to verify our works.

For the communication part, we used the UART module (including UART_Receiver & UART transmitter) to build a link between the FPGA hardware and the PC part(We can also call it , the front-end and the software part).

For the front-end part, we used the Python language to build the web server and used the HTML language to build our personal web server. The web is shown below in Fig .



Matrix Size: 3x3

Please connect to the FPGA board using the button above.

Matrix A (Input Matrix)

0	0	0
0	0	0
0	0	0

Matrix B (Convolution Kernel)

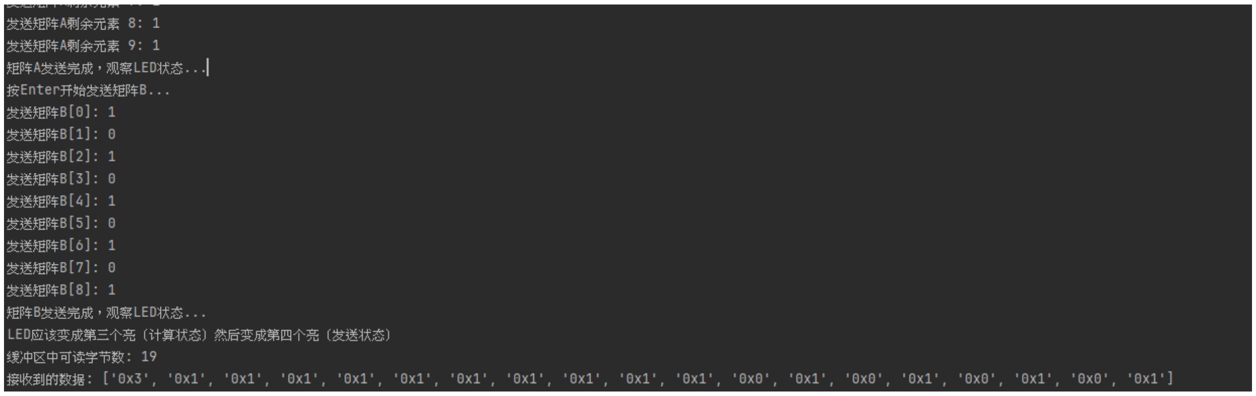
0	0	0
0	0	0
0	0	0

Here, we could use the different configuration command to customize the size of the filter (3* 3, 5*5, 7*7 and 9*9). After click on the “Connect to FPGA” button , we could select different port in the PC host machine. In this experiment, the FPGA port is defined as “COM6”. Then, we could use the Input Matrices part to customize our matrices for computing in FPGA. We can use the “Perform Convolution on FPGA” button to send the input data to FPGA through UART port.

For the hardware part , we use the top.v file to control the whole process. The top module has 3 submodule, uart_rx, uart_tx and convolution_processor. We use the UART module to receive and send information and use convolution _processor to compute the data received from the web server.

In the top module , we define several states: IDLE, CONFIG(to receive the size of the matrix), RECEIVE_A, RECEIVE_B, COMPUTING , SEND.

First we use Python to test the validation of our code and the FPGA port. We send the data one by one from the local host. And the result is shown below .

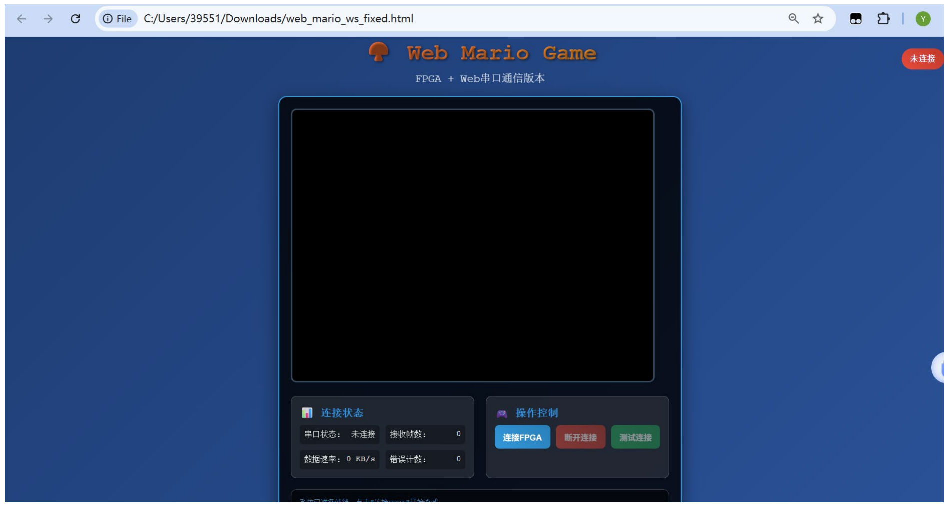


```
发送矩阵A剩余元素 8: 1
发送矩阵A剩余元素 9: 1
矩阵A发送完成，观察LED状态...
按Enter开始发送矩阵B...
发送矩阵B[0]: 1
发送矩阵B[1]: 0
发送矩阵B[2]: 1
发送矩阵B[3]: 0
发送矩阵B[4]: 1
发送矩阵B[5]: 0
发送矩阵B[6]: 1
发送矩阵B[7]: 0
发送矩阵B[8]: 1
矩阵B发送完成，观察LED状态...
LED应该变成第三个亮（计算状态）然后变成第四个亮（发送状态）
缓冲区中可读字节数: 19
接收到的数据: ['0x3', '0x1', '0x1', '0x1', '0x1', '0x1', '0x1', '0x1', '0x1', '0x1', '0x1', '0x0', '0x1', '0x0', '0x1', '0x0', '0x1', '0x0', '0x1']
```

From this picture , we can see that the *FPGA could receive all the data we sent*. After that , we use the received data to perform the convolution computing. Due to the limited time, we haven't debugged the whole project yet. But we confirm that we would achieve the success soon . While the whole project is thought to be quite interesting , we will further extend more function . Maybe we can use this design as prototype to develop an online tool to transplant the local operand to the cloud one day.

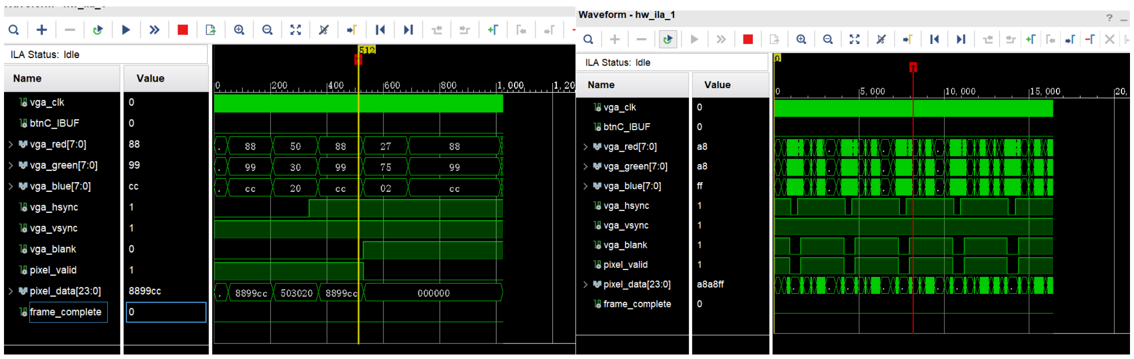
6.2. Uploading VGA Images to Web Server

After successfully testing bidirectional data transmission between a web interface and an FPGA, as well as displaying graphics on a VGA peripheral through a basic gaming machine, we proceeded to integrate a web-based visualization system. Our objective is to design a top-level module in RTL, develop a Python-based RX module for the PC, and create a web page to display the output.



The envisioned web interface allows users to manage the connection between the PC and the FPGA. Upon establishing a stable connection, the system parses data received from the UART RX port (handled by the Python module), converts the UART data into frame-based pixel data, and renders it dynamically on the web page.

To realize this functionality, it was necessary to convert VGA signal outputs—including vga_red, vga_green, vga_blue, vga_clk, vga_blank, vga_hsync, and vga_vsync—into pixel data.

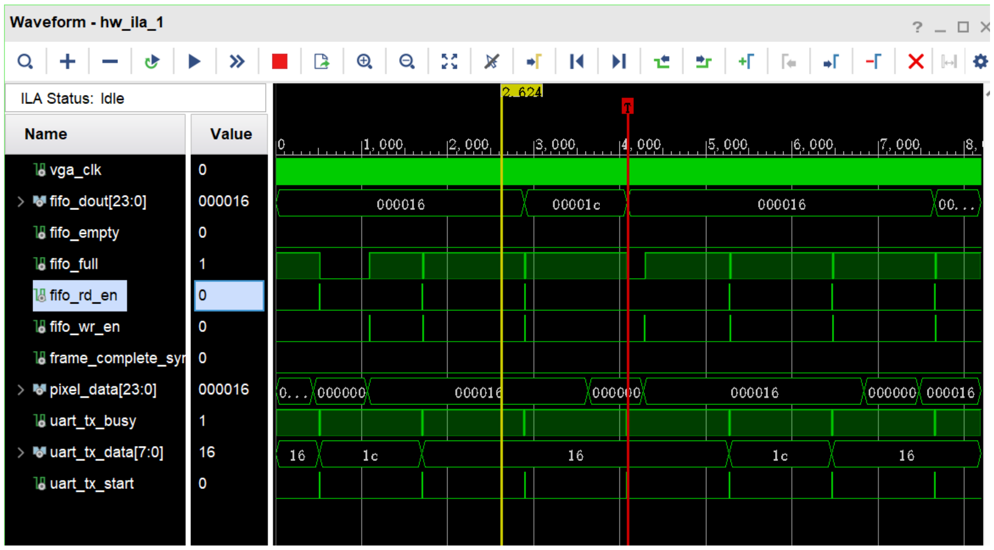


Name	Value
vga_clk	0
btnC_IBUF	0
vga_red[7:0]	88
vga_green[7:0]	99
vga_blue[7:0]	cc
vga_hsync	1
vga_vsync	1
vga_blank	0
pixel_valid	1
pixel_data[23:0]	8899cc
frame_complete	0

Name	Value
vga_clk	0
btnC_IBUF	0
vga_red[7:0]	a8
vga_green[7:0]	a8
vga_blue[7:0]	ff
vga_hsync	1
vga_vsync	1
vga_blank	1
pixel_valid	1
pixel_data[23:0]	a8a8ff
frame_complete	0

The VGAtoPixel module was designed for this purpose, selectively capturing valid pixel values by analyzing synchronization signals (hsync and vsync) and discarding blanking intervals.

Due to the significant difference in clock domains between VGA output and UART transmission, an asynchronous FIFO was introduced. The FIFO's write interface is driven by the VGA domain, with w_clk connected to vga_clk and w_data assigned to the generated pixel stream.



However, the system *encountered a bottleneck at this stage*. The UART interface, even at an increased baud rate of 921600, *proved insufficient for timely transmission of high-resolution video*. For example, only about 1 in 200 frames of a 640×480 display could be transmitted in real time, resulting in a slideshow-like experience on the web—more akin to PowerPoint slides than smooth video playback.

To overcome this limitation, we are now exploring the use of *Ethernet-based transmission*. Preliminary calculations indicate that Ethernet offers data rates thousands of times faster than UART at 921600 baud, which would be sufficient to transmit full frames in real time and replicate the VGA display quality on the web interface.

6.3 Quartus and DE2 Board

For this FPGA course, we were provided with an Altera DE2 FPGA board for experiments. We downloaded Quartus for the experiments. Due to our lack of proficiency in using Quartus, the connections of many IP cores were quite complex. So, we first attempted to develop the project using Basys3 in Vivado. Later, we also studied the use of Nios (through Quartus), hoping to transmit data via a network cable to the internet and play games online. However, *due to the tight development schedule,* we were unable to complete it. This is an area where we can make progress in the future.