cpu.sv

```
Inputs:
    clk             // Clock signal
    rst             // Reset signal
    irq             // Interrupt request
    nmi             // Non-maskable interrupt request
    joypad1 [7:0]   // Controller 1 input
    joypad2 [7:0]   // Controller 2 input
    ppu_read_data [7:0] // Data read from PPU

Outputs:
    ad1, ad2        // Audio outputs
    out0, out1, out2 // Controller outputs
    oe1, oe2        // Controller output enables
    hex_segA [7:0]  // Hexadecimal display segment A
    hex_segB [7:0]  // Hexadecimal display segment B
    hex_gridA [3:0] // Hexadecimal grid display A
    hex_gridB [3:0] // Hexadecimal grid display B
    ppu_write [7:0] // Data written to PPU
    ppu_write_data [7:0] // Data write channel to PPU
    tst             // Test output (not used)
```

Description: The `cpu` module represents a CPU core, closely resembling the 6502-2A03 architecture, commonly used in classic gaming systems. It includes logic for handling interrupts, controlling audio and visual outputs, processing controller inputs, and interfacing with a PPU (Picture Processing Unit). The module is structured with various internal registers, an ALU (Arithmetic Logic Unit), memory mapping, and control units to manage operations.

Purpose: This CPU module is designed to replicate the functionality of a classic gaming console's central processing unit, facilitating game logic, audio processing, controller input handling, and video output control. It's an integral part of a system aiming to emulate or reproduce the behavior of such classic gaming systems.

cpu_mmap.sv

```
arduino
Inputs:
    clk             // Clock signal
    rst             // Reset signal
    ADDR [15:0]     // Memory address
    DATA_IN [7:0]   // Data input
    joypad1 [7:0]   // Controller 1 input
    joypad2 [7:0]   // Controller 2 input
    ppu_read_data [7:0] // Data read from PPU
    RW              // Read/Write flag (0: read, 1: write)

Outputs:
    DATA_OUT [7:0]      // Data output
    test_registers [0:7][7:0] // Test mode registers
    ppu_write [7:0]     // Data written to PPU
    ppu_read [7:0]      // PPU read operation
```

```
    ppu_write_data [7:0] // Data write channel to PPU
```

Description: The `cpu_mmap` module serves as a memory management unit for a CPU, handling different types of memory and I/O operations. It includes logic for interfacing with instruction RAM, expansion ROM, save RAM, program RAM, PPU registers, APU I/O registers, and test mode registers. The module routes the data based on the memory address and the read/write operation specified.

Purpose: This module is designed to facilitate the memory mapping for a classic gaming console's CPU, managing various memory and I/O operations integral to the system's functionality. It is crucial for emulating or recreating the behavior of classic gaming systems.

control.sv

```
Inputs:
    clk             // Clock signal
    rst             // Reset signal
    opcode [7:0]    // Opcode for current instruction
    irq             // Interrupt request
    nmi             // Non-maskable interrupt request
    LC_Q            // Last carry out signal for branching
    C_Q, Z_Q, N_Q, V_Q // Flags for branching (Carry, Zero, Negative, Overflow)
    I_Q             // Interrupt disable flag

Outputs:
    A_EN, B_EN, X_EN, Y_EN, IR_EN, SP_EN   // Enable signals for various
registers
    DPL_EN, DPH_EN, PCL_EN, PCH_EN         // Enable signals for data pointer
low/high and program counter low/high
    DPL_SEL, DPH_SEL                       // Selector signals for data pointer
low/high
    PC_SEL [1:0], SP_SEL [1:0]             // Selector signals for program
counter and stack pointer
    N_EN, V_EN, I_EN, Z_EN, C_EN           // Enable signals for flags
    ALU_OP [3:0], ALU_SEL [3:0]            // Operation and selector signals for
ALU
    ADDR_SEL [1:0]                         // Address selector
    RW                                     // Read/Write flag
    BRK                                    // Break signal
    ACKNMI                                 // Acknowledge Non-Maskable Interrupt
```

Description: The `control` module is a sophisticated control unit for a CPU, designed to manage the flow of operations based on the current instruction's opcode. It handles state transitions, branching logic based on flags, interrupt requests, and generates the necessary control signals to drive various components of the CPU, such as registers, ALU, and memory access.

Purpose: This module is central to the CPU's functionality, acting as the brain of the processor. It interprets each instruction's opcode, manages the execution cycle, and controls the flow of data within the CPU, ensuring correct execution of instructions, handling interrupts, and managing branching and other complex operations.

ppu.sv

```
Inputs:
    clk                   // Clock signal
    rst                   // Reset signal
    ppu_write [7:0]       // Data written to PPU
    ppu_read [7:0]        // PPU read operation
    ppu_write_data [7:0]  // Data write channel to PPU

Outputs:
    ppu_read_data [7:0]   // Data read from PPU
    cpu_clock             // CPU clock signal
    cpu_clock_locked      // Indicates if CPU clock is locked
    nmi                   // Non-maskable interrupt signal
    hdmi_tmds_clk_n       // HDMI clock negative signal
    hdmi_tmds_clk_p       // HDMI clock positive signal
    hdmi_tmds_data_n [2:0] // HDMI data negative signals
    hdmi_tmds_data_p [2:0] // HDMI data positive signals
```

Description: The `ppu` (Picture Processing Unit) module is designed for handling graphics processing, particularly for generating video output signals. It interfaces with an HDMI transmitter, manages VGA signals, and coordinates with the CPU for graphics data processing. The module also generates clock signals and manages non-maskable interrupts (NMI).

Purpose: This module is essential in systems that require video output, such as gaming consoles or graphics processing units. It translates graphics data into video signals and ensures synchronization with the CPU's operations.

ppu_mmap.sv

```
Inputs:
    ppu_clk                // PPU clock signal
    rst                    // Reset signal
    ppu_read_addr [15:0]   // PPU read address
    cpu_write_addr [15:0]  // CPU write address to PPU
    cpu_write_data [7:0]   // Data written by CPU to PPU
    cpu_we                 // CPU write enable signal

Outputs:
    ppu_read_data [7:0]    // Data read by PPU
    palette [0:31][7:0]    // PPU palette data
```

Description: The `ppu_mmap` module functions as a memory management unit for the Picture Processing Unit (PPU) in a graphics system. It handles read and write operations between the CPU and PPU, manages character ROM, PPU RAM, and the color palette. The module maps addresses and data from the CPU to appropriate PPU memory spaces and reads data from PPU memory for graphics rendering.

Purpose: This module is essential for systems requiring advanced graphics processing, especially in gaming consoles or multimedia devices. It allows for efficient data transfer and manipulation between the CPU and PPU, crucial for rendering graphics and managing video output.

ppu_controller

```
Inputs:
    ppu_clk            // PPU clock signal
    rst                // Reset signal
    vga_x [9:0]        // Horizontal coordinate from VGA controller
    vga_y [9:0]        // Vertical coordinate from VGA controller

Outputs:
    x [8:0], y [8:0]   // 9-bit coordinates for PPU
    display            // Display enable signal
```

Description: The `ppu_controller` module is responsible for synchronizing the PPU (Picture Processing Unit) with the VGA (Video Graphics Array) controller. It maintains internal counters for horizontal (`h_counter`) and vertical (`v_counter`) positions and generates output coordinates (`x`, `y`) for the PPU. Additionally, it controls a `display` signal, which indicates when the PPU should be actively displaying graphics based on the current coordinates.

Purpose: This module plays a crucial role in the graphics pipeline of systems that use separate VGA and PPU components, particularly in gaming consoles or video rendering systems. It ensures that the PPU is synchronized with the VGA controller's timing, facilitating smooth and accurate graphics display.

ppu_color_mapper

```
Inputs:
    x, y [8:0]          // 9-bit coordinates for PPU
    ppu_clk             // PPU clock signal
    rst                 // Reset signal
    cpu_write [7:0]     // CPU write operation for PPU
    cpu_read [7:0]      // CPU read operation for PPU
    cpu_write_data [7:0]// Data written by CPU to PPU

Outputs:
    cpu_read_data [7:0] // Data read by PPU
    palette_index [5:0] // Index to the PPU palette
```

Description: The `ppu_color_mapper` module manages color mapping in a Picture Processing Unit (PPU). It takes coordinates, processes them, and determines the corresponding color index from the PPU's color palette. This module also handles data transfers between the CPU and PPU, including read and write operations for graphics data.

Purpose: This module is crucial for rendering graphics in systems with separate PPU units, commonly found in gaming consoles and multimedia devices. It facilitates the transformation of graphics data into color outputs on the screen, ensuring accurate and vibrant display of images.

vga_color_mapper

```
Inputs:
    x, y [9:0]            // 10-bit coordinates for VGA
    palette_index [5:0]   // Index to the VGA palette

Outputs:
    r, g, b [7:0]         // Red, Green, and Blue color outputs
```

Description: The `vga_color_mapper` module converts palette indices into corresponding RGB values for display on a VGA screen. It uses the provided X and Y coordinates, along with a palette index, to determine the appropriate color values for each pixel.

Purpose: This module is an essential component in systems that generate video output, particularly those that use VGA for display. It is responsible for translating palette indices, typically from a PPU, into actual colors displayed on the screen.

ppu_color_rom.sv

```
Inputs:
    palette_index [5:0] // Index to the VGA palette

Outputs:
    r, g, b [7:0]       // Red, Green, and Blue color outputs
```

Description: The `vga_color_rom` module is a read-only memory component that maps palette indices to RGB color values. It utilizes a predefined palette ROM to translate a 6-bit palette index into corresponding 8-bit values for red, green, and blue (RGB) components.

Purpose: This module is used in video processing systems, particularly those involving VGA output, to convert numerical color indices into actual color values displayed on the screen. It is a key component in rendering color graphics in systems that use indexed color palettes.

vga_controller.sv

```
Inputs:
    pixel_clk        // Pixel clock signal
    rst              // Reset signal

Outputs:
    hs               // Horizontal sync pulse
    vs               // Vertical sync pulse
    active_nblank    // Signal indicating active display or blanking interval
    x, y [9:0]       // Coordinates (Horizontal x, Vertical y)
```

Description: The `vga_controller` module generates synchronization signals required for a VGA (Video Graphics Array) display. It operates based on a pixel clock, managing horizontal and vertical synchronization pulses and tracking pixel coordinates for display. The module controls when pixels should be active or in a blanking interval, adhering to VGA timing standards.

Purpose: This module is essential for interfacing with VGA displays, ensuring that pixels are drawn at the correct coordinates on the screen. It is widely used in systems that require a VGA output, such as computers, gaming consoles, and other multimedia devices.

alu

```
arduino
Inputs:
    P, Q [7:0]          // Operands for arithmetic operations
    C_IN                // Carry input
    OP [3:0]            // Operation code

Outputs:
    V, Z, N, I          // Flags: Overflow, Zero, Negative, Interrupt
    C_OUT               // Carry output
    Y [7:0]             // Result of operation
```

Description: The `alu` (Arithmetic Logic Unit) module performs various arithmetic and logic operations based on the operands `P` and `Q` and the operation code `OP`. It supports operations such as addition, subtraction, logical operations (AND, OR, EOR), and shift operations. The module also sets appropriate flags based on the operation's result.

Purpose: This module is a core component of the CPU 6502, handling the arithmetic and logic computations required for processing instructions.

microinstruction_rom

```
Inputs:
    index [6:0]        // Index for the microinstruction ROM

Outputs:
    microinstr [36:0]  // Microinstruction output
```

Description: The `microinstr_rom` module serves as a read-only memory for microinstructions in a CPU architecture. It maps a 7-bit index to a corresponding 37-bit microinstruction. The microinstruction format includes various fields that control the CPU's behavior at a microarchitectural level.

Microinstruction Format:

- **[36:31]** - 6 bits: Enables for A, B, X, Y, IR, SP
- **[30:27]** - 4 bits: Enables for DPL, DPH, PCL, PCH
- **[26:21]** - 6 bits: Selects for DPL, DPH, PC (2 bits), SP (2 bits)
- **[20:16]** - 5 bits: Flags (N, V, I, Z, C) enables
- **[15:12]** - 4 bits: ALU operation code
- **[11:8]** - 4 bits: ALU input select
- **[7:6]** - 2 bits: Address bus selection
- (Further fields might be present but are not displayed in the provided snippet)

Purpose: This module is a key component in a microprogrammed CPU, where each microinstruction dictates the operation of the CPU at a fine-grained level. It is essential for the execution of complex instructions, breaking them down into simpler operations that the CPU's control unit can directly execute.

microcode_rom

```
arduino
Inputs:
    opcode [7:0]     // Opcode for the instruction
    cycle [2:0]      // Current cycle in instruction execution

Outputs:
    index [6:0]      // Index to the microinstruction ROM
```

Description: The `microcode_rom` module functions as a lookup table that translates an instruction's opcode and the current cycle number into an index for the microinstruction ROM. This module maps each combination of opcode and cycle to a specific microinstruction index, directing the CPU's control logic to the appropriate microinstruction for each step of instruction execution.

Purpose: This module is crucial in microprogrammed control units within CPUs. It determines the sequencing of microoperations necessary to execute higher-level instructions.

top_level

```
arduino
Inputs:
    Clk                  // Main clock signal
    rst                  // Reset signal
    joypad1, joypad2 [7:0] // Joypad inputs

Outputs:
    hex_segA, hex_segB [7:0]     // Hexadecimal segment outputs
    hex_gridA, hex_gridB [3:0]   // Hexadecimal grid outputs
    hdmi_tmds_clk_n, hdmi_tmds_clk_p            // HDMI TMDS clock signals
(negative and positive)
    hdmi_tmds_data_n, hdmi_tmds_data_p [2:0]    // HDMI TMDS data signals
(negative and positive)
```

Description: The `top_level` module is the main module of a system that integrates various submodules, including a Picture Processing Unit (PPU) and a Central Processing Unit (CPU). It coordinates the operations of these submodules and interfaces with external components like joypads and HDMI output.

Purpose: This module acts as the central hub for the system, managing the flow of data between the CPU, PPU, and external interfaces. It is essential for systems requiring coordinated processing and output, such as gaming consoles or multimedia devices.

register

```
less
Parameters:
    WIDTH [int]                // Width of the register in bits
    INITIAL_VALUE [WIDTH-1:0]  // Initial value of the register

Inputs:
    clk                        // Clock signal
    rst                        // Reset signal
    en                         // Enable signal
    D [WIDTH-1:0]              // Data input

Outputs:
    Q [WIDTH-1:0]              // Data output
```

Description: The `register` module represents a generic register with a specified bit width (`WIDTH`). It stores data (`D`) when enabled (`en`) and can be reset (`rst`) to an initial value (`INITIAL_VALUE`).

Purpose: This module is a fundamental building block in digital systems, used for storing data across clock cycles. It is crucial in designs that require temporary storage of data, such as CPUs, memory units, and various digital circuits.

shift_register

```
Parameters:
    WIDTH [int]                // Width of the shift register in bits
    INITIAL_VALUE [WIDTH-1:0]  // Initial value of the shift register

Inputs:
    clk                        // Clock signal
    rst                        // Reset signal
    load_en                    // Load enable signal
    D [WIDTH-1:0]              // Data input for loading
    shift_en                   // Shift enable signal
    SD                         // Serial data input for shifting

Outputs:
    Q [WIDTH-1:0]              // Parallel data output
    SQ                         // Serial data output (most significant bit)
```

Description: The `shift_register` module is a generic shift register that can load parallel data or shift data serially. It shifts data from the least significant bit to the most significant bit, with an option to load new data or retain its value based on control signals.

Purpose: Widely used in digital systems for data manipulation, serialization/deserialization of data, and as temporary buffers. It is essential in designs requiring sequential data processing, such as communication interfaces and data storage systems.

mux2

```
csharp
Parameters:
    WIDTH [int]            // Bit width of the inputs and output

Inputs:
    sel                    // Selector signal
    in0, in1 [WIDTH-1:0]   // Input signals

Outputs:
    out [WIDTH-1:0]        // Output signal
```

Description: `mux2` is a 2-to-1 multiplexer with a parameterizable width. It selects between two input signals (`in0` , `in1`) based on the value of the selector (`sel`) and outputs the selected input.

Muxes

mux4

```
Parameters:
    WIDTH [int]            // Bit width of the inputs and output

Inputs:
    sel [1:0]              // 2-bit selector signal
    in0, in1, in2, in3 [WIDTH-1:0] // Input signals

Outputs:
    out [WIDTH-1:0]        // Output signal
```

mux8

```
Parameters:
    WIDTH [int]            // Bit width of the inputs and output

Inputs:
    sel [2:0]              // 3-bit selector signal
    in0 to in7 [WIDTH-1:0] // Input signals

Outputs:
    out [WIDTH-1:0]        // Output signal
```

Description: `mux8` is an 8-to-1 multiplexer. It chooses from eight input signals based on a 3-bit selector and provides the selected input at its output.

mux16

```
csharp
Parameters:
    WIDTH [int]            // Bit width of the inputs and output

Inputs:
    sel [3:0]              // 4-bit selector signal
    in0 to inF [WIDTH-1:0] // Input signals (16 total)

Outputs:
    out [WIDTH-1:0]        // Output signal
```

Description: `mux16` is a 16-to-1 multiplexer. It selects from sixteen input signals based on a 4-bit selector. The chosen signal is then outputted.

Purpose: These multiplexer modules are fundamental in digital logic design, allowing for the selection of different data paths or control signals based on the value of a selector. They are widely used in CPUs, data routing, control units, and various other digital systems.

cpu_top_level.sv

```
Inputs:
    Clk                  // Main clock signal
    rst                  // Reset signal
    joypad1, joypad2 [7:0] // Joypad inputs

Outputs:
    hex_segA, hex_segB [7:0]     // Hexadecimal segment outputs
    hex_gridA, hex_gridB [3:0]   // Hexadecimal grid outputs
```

Description: The `cpu_top_level` module serves as the top-level container for the CPU (`cpu`) module in a system. It is responsible for providing the CPU with necessary signals such as clock, reset, and inputs from joypads, and retrieving outputs for hexadecimal displays.

Purpose: This module functions as the primary interface for the CPU, facilitating the integration of the CPU with other system components or peripherals. It primarily handles input/output management, clock signal distribution, and reset control.

HexDriver.sv

```
Inputs:
    clk, reset,
    [3:0] in[4],
Outputs:
    [7:0] hex_seg,
    [3:0] hex_grid
```

Description: Given 4 x **4-bit** data `[3:0] in[4]`, it will generate correct `hex_seg` and `hex_grid` which will be used to display data on the FPGA board.

Purpose: Display the 8-bit logic processor's register on the FPGA's Segment LED display.

VGA-HDMI IP

```
Inputs:
    pix_clk, pix_clkx5, pix_clk_locked
    reset_ah, red, green, blue,
    hsync, vsync, vde,

    aux0_din, aux1_din, aux2_din, ade

Outputs:
    [2:0] TMDS_CLK_P
    [2:0] TMDS_CLK_N
    TMDS_DATA_P
    TMDS_DATA_N
```

Description: The `hdmi_tx_0` module acts as a VGA to HDMI converter. It takes in VGA signals such as red, green, blue color channels along with sync signals (hsync and vsync) and converts them into differential HDMI outputs. The pixel clock and its multiplied version (5x) drive the conversion process. A separate locked signal indicates the status of the pixel clock. The module also provides auxiliary data inputs, which are unused in this context and are set to default values.

Purpose: This module serves as a bridge between VGA interfaces and HDMI outputs, facilitating the conversion of VGA signals into HDMI format.

Clocking Wizard (Vivado IP)

Description: The Clocking Wizard in Vivado is a customizable IP core designed to generate multiple clock frequencies derived from a primary reference clock. It utilizes Phase-Locked Loops (PLLs) and/or Digital Clock Managers (DCMs) depending on the FPGA family and user configuration. The Clocking Wizard allows users to specify the desired output frequencies, phase shifts, and duty cycles. Furthermore, it provides features to optimize jitter performance and introduces protection mechanisms against clock glitches.

Purpose: The primary goal of the Clocking Wizard is to generate various clock frequencies required for a design from a single reference clock. This aids in maintaining synchronization across different parts of an FPGA design and provides flexibility in clock configuration to meet specific design requirements. It is an essential IP in Vivado for designs that demand multiple clock domains or specific clock characteristics.

Testbenches:

`ppu_test`

Testbench Setup:

- **No Inputs/Outputs**: As a testbench, `ppu_test` does not have input and output ports. It is designed to simulate and test the behavior of the `top_level` module.

Internal Signals:

- `Clk` (Logic): Simulates the clock signal.
- `rst` (Logic): Simulates the reset signal.
- `joypad1`, `joypad2` (Logic [7:0]): Simulate inputs from joypads.
- `hex_segA`, `hex_segB` (Logic [7:0]): For observing hexadecimal segment outputs from `top_level`.
- `hex_gridA`, `hex_gridB` (Logic [3:0]): For observing hexadecimal grid outputs from `top_level`.
- `hdmi_tmds_clk_n`, `hdmi_tmds_clk_p` (Logic): For observing HDMI TMDS clock signals.
- `hdmi_tmds_data_n`, `hdmi_tmds_data_p` (Logic [2:0]): For observing HDMI TMDS data signals.

Instantiated Modules:

- `top_level top_level_inst`: Instance of the `top_level` module with all connections made using the wildcard `.*` to automatically connect signals with the same name.

Procedural Blocks:

- `clk_gen`: A procedural block to generate a clock signal. Toggles the `Clk` signal every 1 time unit.
- `clk_init`: An initial block to set the initial state of the clock (`Clk`) to 0.
- `test`: An initial block to simulate the test scenario. It sets the reset signal high, then low after 2 time units, simulates joypad inputs, and runs the simulation for a specified duration before finishing.

Purpose of Testbench:

- **Simulation**: To simulate the `top_level` module, ensuring that it behaves as expected when subjected to various inputs and conditions.
- **Verification**: To verify the functionality of the integrated modules within `top_level`, particularly their response to clock, reset, and joypad inputs.

Test Scenarios

- **Reset Test**: Observes the behavior of the system upon reset.
- **Joypad Input Test**: Verifies how the system responds to specific joypad inputs.
- **Clock Signal Test**: Checks the system's behavior across multiple clock cycles.
- **Duration**: The test runs for 1,000,000,000 time units, after which the simulation terminates.

`cpu_test`

Testbench Setup:

- **No Inputs/Outputs**: As a standard practice for testbenches, `cpu_test` does not have input and output ports. It is structured solely to simulate and validate the behavior of the `cpu` module.

Internal Signals:

- `clk` (Logic): Simulates the main clock signal.
- `rst` (Logic): Simulates the reset signal.
- `irq`, `nmi` (Logic): Simulate interrupt request and non-maskable interrupt signals.
- `ad1`, `ad2` (Logic): Audio output signals for verification.
- `out0`, `out1`, `out2`, `oe1`, `oe2` (Logic): Controller output and output enable signals for testing.
- `tst` (Logic): Test signal (not used).
- `ppu_write`, `ppu_read` (Logic [7:0]): Signals for simulating PPU write and read operations.
- `ppu_write_data`, `ppu_read_data` (Logic [7:0]): Data signals for PPU write and read operations.
- `joypad1`, `joypad2` (Logic [7:0]): Simulate inputs from joypads.
- `hex_segA`, `hex_segB` (Logic [7:0]): For observing hexadecimal segment outputs.
- `hex_gridA`, `hex_gridB` (Logic [3:0]): For observing hexadecimal grid outputs.

Instantiated Modules:

- `cpu cpu_inst`: Instance of the `cpu` module with all connections made using the wildcard `.*` to automatically connect signals with the same name.

Procedural Blocks:

- `clk_gen`: Generates a clock signal by toggling the `clk` signal every 1 time unit.
- `clk_init`: Initializes the `clk` signal to 0 at the beginning of the simulation.
- `test`: Simulates the reset and joypad input scenarios. Sets the reset signal high, then low after 2 time units, simulates joypad inputs, and runs the simulation for a specified duration before terminating.

Special Procedures:

- **NMI Simulation**: An always block periodically toggles the `nmi` signal to simulate non-maskable interrupts.

Test Scenarios

- **Interrupt Test**: Verifies CPU's response to non-maskable interrupts (`nmi`).
- **Reset Test**: Observes CPU behavior upon reset.
- **Joypad Input Test**: Tests CPU's handling of joypad inputs.
- **Clock Signal Test**: Examines CPU's functionality across multiple clock cycles.
- **Simulation Duration**: Runs for 10,000 time units before concluding.