ORacles: Final Project Submission

Ian Smith, Evelyn Kim, Anthony Lesik, Martin Korir

Introduction

The objective of this project was to demonstrate our team's ability to create a full-stack digital system by building a playable game on top of a custom RISC processor that is synthesized on an FPGA board, and building the VGA and PS/2 drivers so that we can display our game and provide input from a PS/2 keyboard. We designed our RISC processor off of the CR-16a [2] ISA but did not follow it to the letter. This was partly due to conflicting instructions from our lab modules, and personal preference.
In our RISC processor you will see modules for the instruction decoder / controller that implements a fetch/decode/execute/load state machine, ALU, RAM, register files, program counter, IR register, and our 2 to 1 muxes. Our top level module is named cpu_top and integrates all of these modules together.
Our PS/2 driver is more simple than most and we use it mainly to detect if the spacebar is compressed or not. We wire that output directly to register 13 in our regfile.
Our VGA controller and top level modules are much more complex and use a module per sprite to render them. It also uses an FSM to write a signal to memory for the game to use so that it may sync logic to frames and read position values from memory for each sprite. We did design our VGA so that it reads the sprites from its own separate memory. This was okay since we're only using 16 bit addressing which, even with two memories, keeps us within the 262k word limit (for 16 bit words) for our DE1-SOC FPGA board.

Below you will find a list that describes the layout of our project directories so you can easily navigate them.

- **cpu**:
    - **src**: Main folder for all of our non-testbench files.
    - **test**: Folder that contains our testbenches for our processor
- **game_code**: In this directory you will find our assembler, asm file we created to implement our game, the compiled version of that code, and the python script we used to turn png's into hex files that our useable by our VGA controller.
- **ps2**: In this directory you will find all of our PS/2 related files. Start with *space_key_detector.v*
- **resources**: This directory contains a block diagram for our ALU and regfile integration, and a pin out that you can import into Quartus to run our code.
- **Sprites**: This directory contains all of the sprites we used for our project in both their png and hex formats. It also contains the sprite hex files that we use in our VGA's direct memory (implemented in true_dual_port_ram_single_clock_vga.v).
- **vga**: In this directory you will find our VGA related modules. Start with vga_top.v. Our RAM that we use only with our VGA controller is in here as well.

- We also attached a README.md which will direct you on how to set up and run our project.

Abstract

Our team built a RISC processor based off of the CR-16a ISA [2], VGA and PS/2 drivers, an assembler, and a game that implements a simple jump game with collision detection and object wrapping. The CPU uses dual-port RAM preloaded with our game's assembled code and shares the second port with the VGA. Our VGA controller provides a signal that fires when a frame has finished rendering so our game logic is synced to each frame. It provides this signal by writing it into a known address which our game code can then read and use. The software implements jump physics, obstacle wrapping, AABB collision detection, and an internal, but unused, score. The software uses register-mapped input from the spacebar to drive a 96x96 walking character and cactus sprites and includes a background sprite as well. The project demonstrates our team's ability to synthesize complex logic using Verilog and integrate multiple modules together to form a complex system.

Detailed description of CPU

ALU

In our CPU implementation you will find modules for the ALU, decoder/controller, regfile, dual-port RAM, IR register, program counter, and our 2 to 1 mux. We also use a top level module called *cpu_top.v* that wires everything together and also keeps track of our flags by assigning them from the ALU and passing them to the decoder/control module. For our ALU we implemented the following instructions listed below:

- **NOP**: No operation, we simply do nothing if we receive this opcode.
- **AND**: Bitwise AND
- **OR**: Bitwise OR
- **XOR**: Bitwise XOR
- **ADD**: Add that sets carry and overflow flags. Does not use carry in.
- **ADDU**: Add that does not use carry in and does not set flags.
- **ADDC**: Add that uses carry and sets carry and overflow flags.
- **NOT**: Bitwise not
- **SUB**: Subtraction that sets carry and overflow flags and does not use carry in.
- **SUBC**: Subtraction that sets carry and overflow flags and does use carry in.
- **CMP**: This instruction got a little mangled. It is in the format of src, dest, but the comparison is backwards than what most people are used to. This is because we are setting the flags based on if the destination register is GREATER THAN

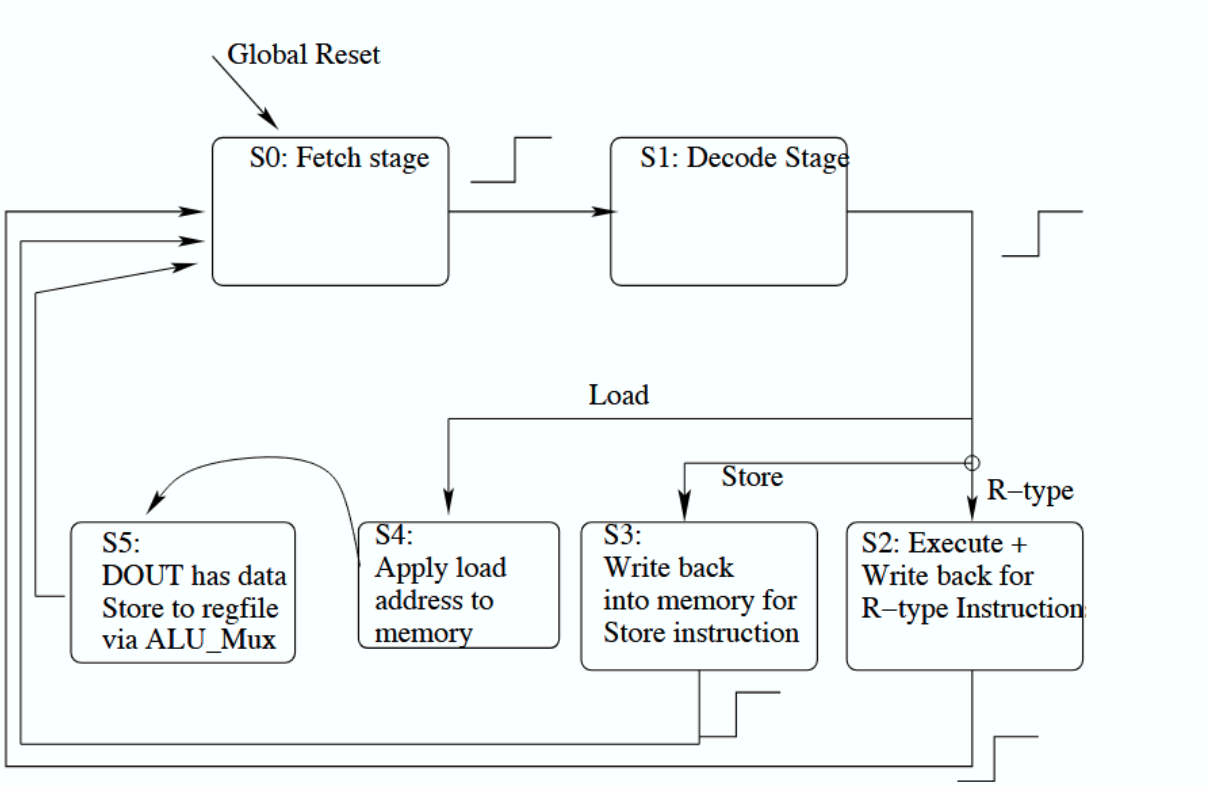
(instead of less than as usual) the source register. We did this because when we implemented our condition codes following the ISA that was given to us in lab 1, our CMP instruction was backwards, so we had to reverse it to make it work correctly. This proved to be a minor inconvenience but was relatively easy to work around since we kept track of this.

- **MOV**: Sets the destination register to the value of the source register. We use the assembler and decoder to also implement a MOVI instruction, but that is not defined in our ALU.
- **ASHU**: Arithmetic shift that can shift right or left depending on the sign of the source register. If the source register is negative, it shifts right. When shifting right, this instruction preserves the sign bit.
- **LSH**: Logical shift that can shift right or left depending on the sign of the source register. If the source register is negative, it shifts right. When shifting right, this instruction does not preserve the sign bit.

Control and Decoder

Our control/decoder module entirely comprises an FSM. You can find our FSM diagram below that was taken straight from the course power points. We implemented our decoder / control logic using this FSM. We use the 5th and 6th states to write, read, and store using an IR (Instruction register). This is needed because otherwise, our FSM would lose track of where we're trying to save the data that was loaded from memory. We created a separate module for our IR register that our top level module wires into

the decoder as an input.



Program Counter

Our program counter is relatively simple. We designed it to increment by 1 every clock cycle, or, using a signal called *pc_mux*, we can choose to increment it by a displacement amount (passed as a 16 bit value named *disp.* This displacement value allows us to modify our program counter so that we can support jump instructions. Our assembler implements using pc-relative values and not direct addresses, but our program counter does support direct addresses using the *pc_load* flag and the 16 bit *tgt_addr* value.

Regfile

Our register file is the simple glue between the ALU, memory, and input. The regfile implements 16 registers, all 16 bits wide, with the ability to read from two registers at a time, and write to up to all 16 at a time using the *wr_en* signal and *reg_en* mask. On reset we clear everything to zero. We also used R13 as a direct data point for the PS/2 keyboard's spacebar. cpu_top.v feeds space_is_down from the PS/2 front-end, and the regfile overwrites R13 with that bit every cycle, regardless of reg_en. We do this so that

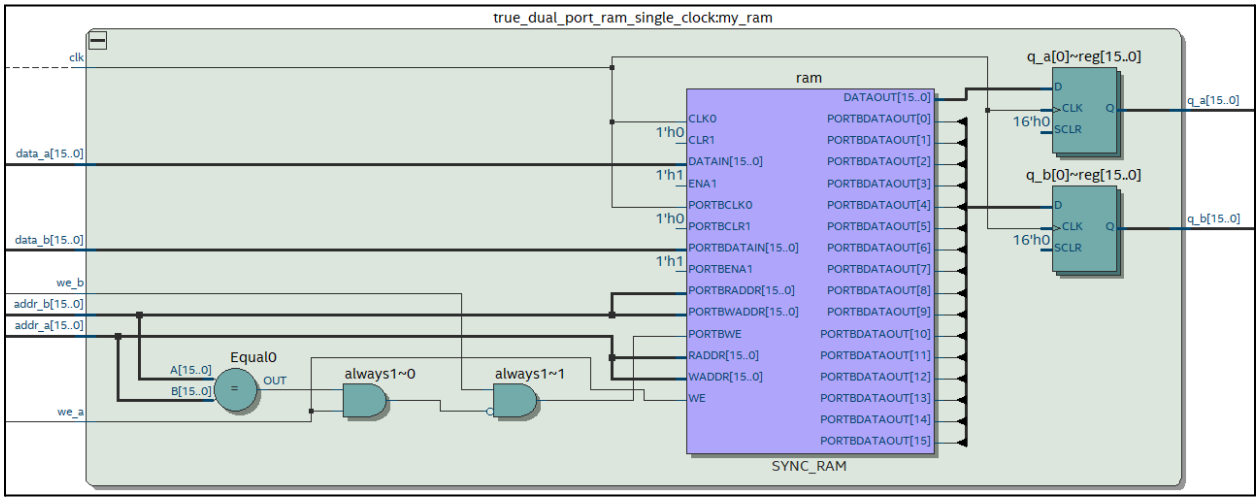
our game logic can easily poll the input from our PS/2 keyboard. All other registers follow the normal write path driven by the muxed ALU/memory data (regFileInput in cpu_top.v). We chose to leave out the top-level module *Regfile_ALU_Datapath.v* in our cpu_top.v module as we needed to have more control over our registers without having to deal with the complexity of going through a single module.

Instruction Register

Our IR is just a gated latch that holds the current instruction between the fetch and execute stages. It sits on the RAM A output (DOUT in cpu_top.v) and captures a 16-bit word on the rising edge only when ir_en is asserted by the control FSM. On reset it clears to 0. This keeps LOAD's two-step path intact: the decoder uses state S4 to put the address on the bus, then asserts ir_en in S4/S5 to freeze the fetched word so we don't lose the opcode when the PC advances. The latched value (ir_out) feeds back into the decoder as the stable instruction context for the subsequent cycle.

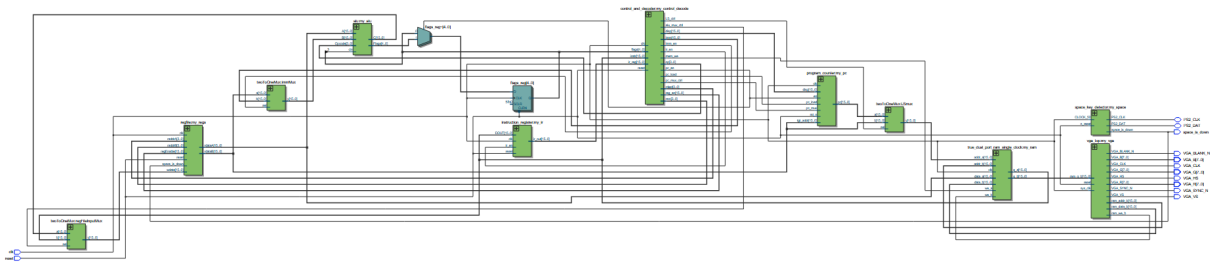
Dual-Port RAM

Our memory is a single-clock true dual-port block (*true_dual_port_ram_single_clock.v*) that feeds both the CPU and the VGA side. Port A is owned by the core: it writes *data_a* on *we_a* and returns either the newly written data or the stored word on reads. If both ports target the same address and try to write at the same time, Port A will take precedence. We preload the memory with $readmemh from *game_code/game.hex* which implements our game code. In *cpu_top.v* Port A is wired to the load/store mux and register file, and Port B is routed to vga_top for per-frame position fetches and the vblank flag. Both ports are 16 bits wide, and everything runs off the system clock for deterministic fetch/execute timing.



RTL Diagram

Here is our RTL View of our overall CPU with the VGA and PS/2 controllers wired in:



For a better view of this diagram, I have included it in our resources directory as *cpu_top_block_diagram.pdf*.

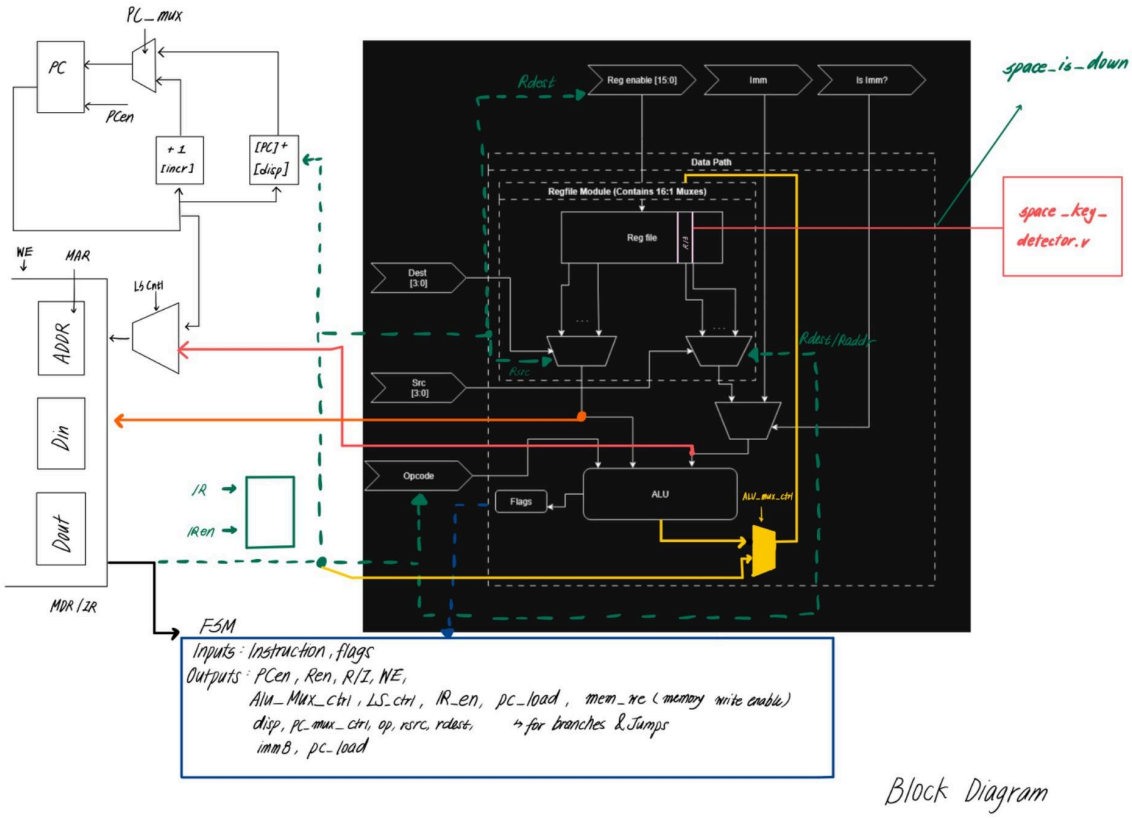
Detailed description of PS/2

For this project, we implemented keyboard input on the DE2 board using the PS2 interface. The PS2 controller, PS2_Data_In, PS2_Command_OUT, we used from official documentation from the University of Toronto ECE241 course materials which was provided by our course ppt [1]. Our design only required keyboard input, specifically the spacebar, so mouse features were not used. Since our design did not need to send commands back to the keyboard, the optional command lines were tied to constant values, and only the incoming scan-code signals were used. The controller provided two important outputs: one signal that indicates a new scan code has arrived, and one signal that contains the actual 8-bit scan code from the keyboard.

The keyboard sends a unique code whenever a key is pressed or released. A “make code” represents a key press, and a “break code” represents a key release. For the spacebar, the make code is 29 in hexadecimal. The break code always begins with the prefix F0. Using this information, we created a separate module to detect spacebar activity. The detector checks each incoming scan code. If the code F0 appears, the next code is interpreted as a release. If the code 29 appears without a preceding F0, it is interpreted as a key press. When the spacebar is pressed, the module sets a signal called space_is_down to 1. When the key is released, the signal returns to 0. The detector also generates a short one-cycle pulse whenever the spacebar is newly pressed. This pulse can be used for actions such as jumping in our game.

To make this keyboard input accessible to the CPU, we connected the spacebar signal into the register file as shown in the following diagram. Register number 13 was

reserved for this purpose and it's connected to the space_key_detector module. On every clock cycle, this register is overwritten with a 16-bit value that represents the current state of the spacebar(space_is_down). If the key is pressed, the register holds the value 1. If it is not pressed, the register holds the value 0. This prevents the assembly program from accidentally overwriting the register, while still allowing the CPU to read the key state at any time. In our assembly program, the game logic simply reads this register to determine whether the player is pressing the spacebar.



Block Diagram

Overall, our system uses the official PS2 controller to manage the physical keyboard interface, the custom detector module to interpret spacebar activity, and the register file to make this input available to the CPU. This modular design allowed us to integrate keyboard input cleanly into our CPU architecture and use it for interactive controls in the final project.

Detailed description of Software (game code)

Since we opted for a simpler game than initially intended due to time constraints, our game code ended up being relatively straightforward, consisting of only approximately 80 lines of assembly code. To implement a game similar to the Chrome browser dinosaur game, our code needed to handle several core components: the Y-axis position of the player object, the X-axis position of the obstacle object, simple game physics, collision detection, and game state management.

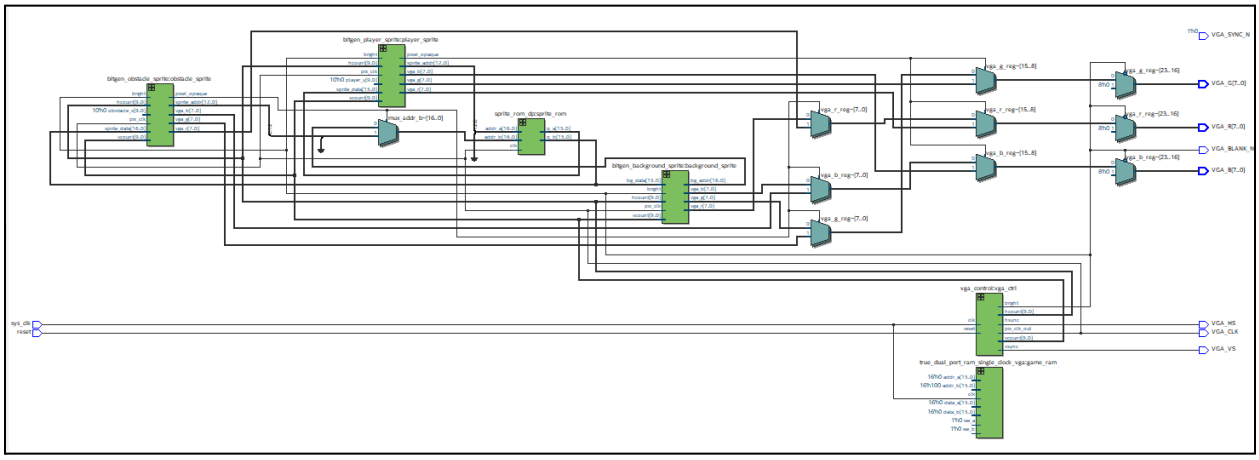
The assembly code begins with an initialization block where we populate registers with essential values, including sprite sizes, jump velocity, and the player's starting position. Following initialization, the program enters the main game loop, which executes the following steps in each iteration:

1. Check the dedicated input register for spacebar input. If the player is on the ground and the spacebar was pressed, set an upward velocity; otherwise, ignore the input.
2. Update the player's Y position based on its current velocity, apply gravity to the player object, and check whether it has reached the ground.
3. Update the obstacle's position by shifting it left by 6 pixels.
4. Perform collision detection between the player and obstacle by checking for overlap between the sprite bounding boxes. If a collision is detected, update the game status register.
5. Update the memory values that the VGA controller reads from, specifically the player's Y position and the obstacle's X position.

Given more time, we would have likely added additional features, such as a scoring system, a leaderboard, and increasing difficulty as the player progresses. However, even this relatively simple implementation required considerable effort to get working. One of the main challenges was distinguishing between game logic errors and instruction implementation bugs. Even with the relatively few layers we were working with, this project underscored the importance of exhaustive testing when building a system with multiple components.

VGA System Overview

Our VGA system acts as the visual output interface for the game. It is responsible for rendering all the graphical elements to the display at 640×480 resolution and 60 Hz refresh rate. The system begins with our VGA controller (**vga_control.v**) that generates exact timing signals which are then fed to our different sprite modules. These sprite modules access a shared dual-port ROM (**sprite_rom_dp.v**) that contains all our graphical assets and each module calculates when it should be visible based on the current pixel position. The **vga_top**.v module integrates all of these modules and manages the communication with game state RAM through **cpu_top.v** to pass the sprite positions. Shown below is an RTL View image of our VGA subsystem.



VGA Top Module

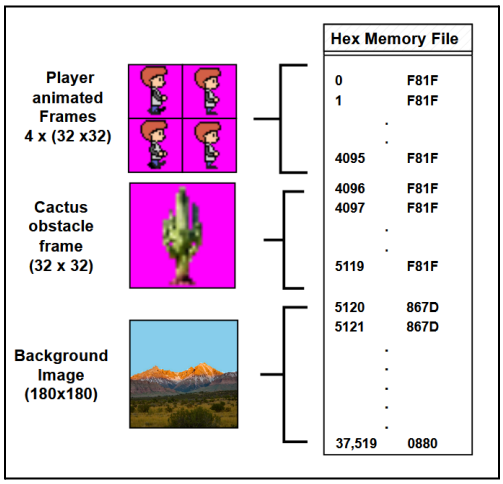
The vga_top.v module is the top module for the VGA subsystem as it instantiates all the bitgen modules, the dual-port sprite ROM, and vga_control. It also manages communication with the game state RAM and contains a FSM that synchronizes position updates with the display. The FSM is triggered by a vertical blanking signal that occurs when the display completes drawing one full frame. During the vertical blanking, the FSM first writes a flag to RAM address 0xFFFE to notify the game logic that the VGA system is ready to receive new position data. It then reads the player's Y position from address 0x0100 and the obstacle's X position from address 0x0101. These position values are captured into local registers and held stable throughout the next frame for the sprite movement. The module also implements the ROM port multiplexing strategy, where port A is dedicated to the player sprite while port B switches between the obstacle and background.

VGA Controller

Our vga_control.v module acts as the timing generator for the VGA display. It produces all the synchronization signals required for 640×480 resolution at 60 Hz refresh rate. We used the timing values that were provided in the Canvas module slides as well as online references[3]. This module outputs the synchronization signals hsync, vsync, our 25 MHz clock, hcount, and vcount to be used by our bitgen modules [3]. The bright signal is used to indicate when the current pixel coordinates falls within the visible display area which allows our other modules to know when to output valid color data instead of blanking the display.

Sprite ROM Architecture

Our sprite ROM is set up to use our **true_dual_port_memory.v** to create a dual-port memory containing all our graphics for the game. As shown in the image below, player animation frames occupy addresses 0 through 4095 for four 32×32 images. The 32×32 obstacle sprite takes up the next 1024 lines of storage space. Lastly the background tiles begin at address 5120 and extend to the end at 37,519. All our sprites were pngs that we converted to RGB565 hex format using a simple python script [4]. To handle transparency we chose to use magenta (F81F) as our transparent color. This is because the default way of using black (0000) caused us to lose details in the sprites like outlines and shadows as they were treated as transparent and lost from the final rendering[5].



The dual-port memory allows us to access the two different sprites within a single clock cycle. We do this by having port A be dedicated to the player sprite to make sure the character can always get its pixel data without contention. Then port B is shared between the obstacle and background sprites using a multiplexer controlled by the **obstacle_opaque** signal. When the obstacle is drawing a visible (non-transparent) pixel, port B fetches obstacle data from ROM. Otherwise, it fetches background data. With this approach we are able to optimize our hardware utilization and maintain the rendering priority needed for correct sprite layering.

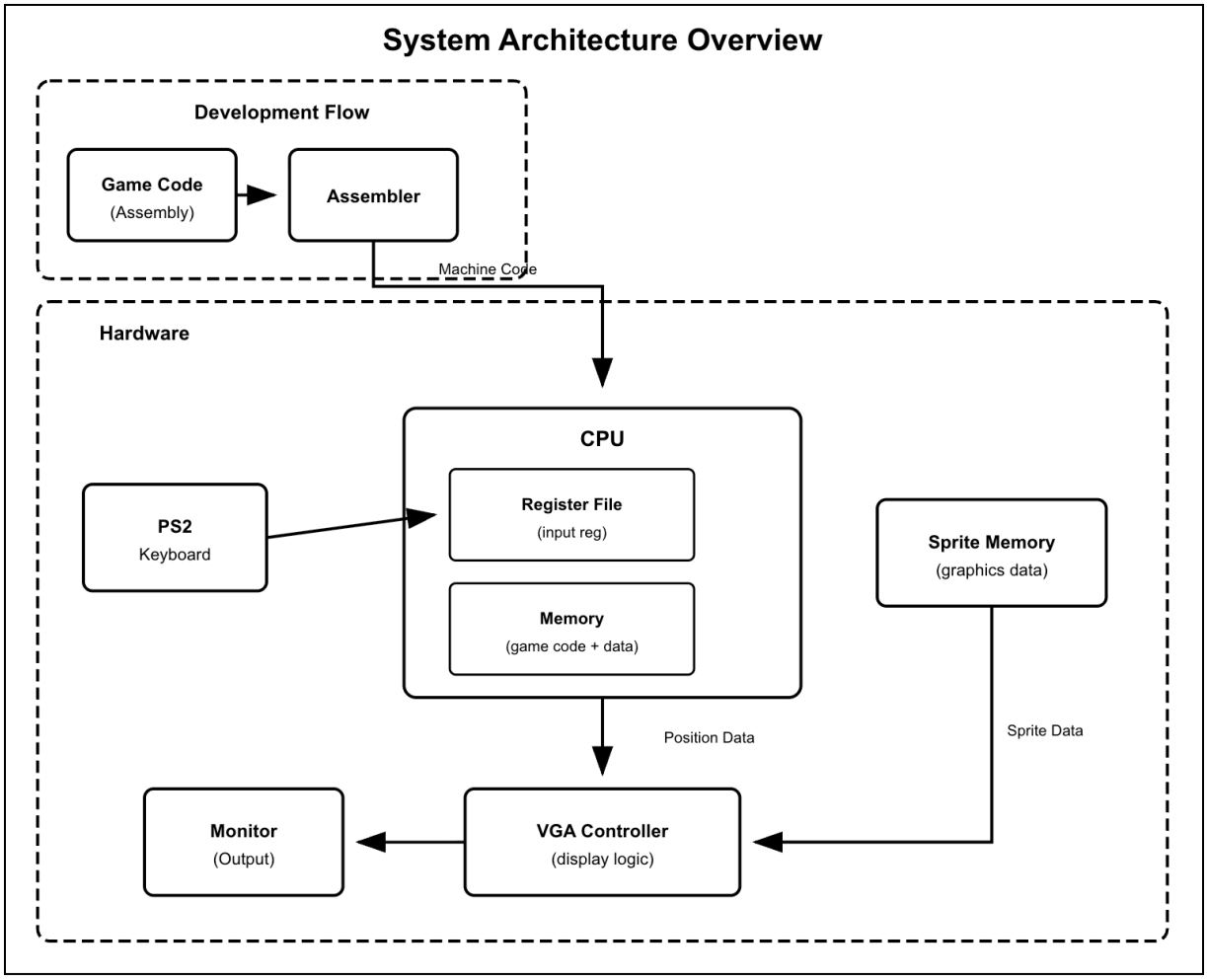
Bitgen Modules

Our player sprite module (**bitgen_player_sprite**) uses a character positioned at a fixed X coordinate on the center of the screen and its vertical position is set by the CPU game logic. The sprite reads the **player_y** input signal from the game state RAM which changes based on the jump physics variables we have set. This module cycles through four animation frames at approximately 5FPS using an internal counter to create a running animation effect. Each frame is 32×32 pixels and scaled by a factor of 3 to be more visible on the screen.

The obstacle sprite module (**bitgen_obstacle_sprite**) displays a cactus positioned at a dynamic X coordinate that moves horizontally across the screen from right to left. The obstacle has a fixed vertical position (y = 200) keeping it grounded at a consistent height throughout gameplay. The **obstacle_x** value is continuously updated by the CPU game logic so that when the obstacle moves off the left edge of the screen, the game logic resets its position to the right side.

The background sprite module (**bitgen_background_sprite**) draws a full-screen scrolling background that fills the entire 640×480 display area. This sprite is set at the furthest layer back so that it only appears where neither the player nor obstacle sprites have opaque pixels.

Overall system integration



The diagram above illustrates the overall system architecture of our project. On the development side, game code written in assembly is converted to machine code with our custom assembler, which is then loaded into the CPU's memory. The CPU executes this machine code to run the game logic, handling tasks such as processing player input, updating object positions, applying physics, and detecting collisions. During gameplay, the CPU continuously updates position data for the player and obstacle objects, storing these values in memory. The VGA controller reads this position data from the CPU's memory and combines it with sprite graphics to render the game to the monitor.

A few additional design decisions are worth noting. The PS2 keyboard input is directly connected to a dedicated register in the CPU's register file, allowing the game code to read player input with minimal latency and without additional memory-mapped I/O overhead. Additionally, sprite data is stored in a separate memory module rather than within the CPU's main memory. This separation was motivated both by space constraints, as storing sprite graphics alongside game code would have increased memory requirements, and by a general separation of concerns, since the sprite data is static and only needs to be accessed by the VGA controller for rendering purposes.

Each member's responsibilities

Evelyn - In this project, my main responsibility was implementing the connection between the PS2 keyboard interface and our custom CPU. I integrated the PS2 controller with the rest of the system and ensured that the spacebar input could be correctly interpreted and used by the CPU. In addition to the PS2 interface work, I also contributed to testing and debugging the core CPU components, including the ALU, the register file, and the memory modules. My role involved verifying that each part of the CPU behaved as expected and making adjustments during debugging sessions to ensure stable and correct operation. I also worked on the initial draft and base code for the instruction register and the store instruction logic for the FSM for branch instruction.

Anthony - My main responsibilities in the chip design portion of the project were implementing the top-level CPU module that connected all components of the chip, as well as assisting in the design and testing of the control FSM that coordinated how data was transferred throughout the design. During the development of the game, I was responsible for creating the assembler and implementing the game logic in assembly.

Ian - My responsibilities for this project included implementing the PS/2 controller alongside Evelyn, collaborating on the control and decode FSM, and contributing to the design of the ALU. I implemented the dual-port RAM and handled the write-conflict logic, worked on integrating the VGA system with the game code, and collaborated on the game logic itself. I also wrote several testbenches and supported project management by encouraging consistent meeting times and helping define clear, concise tasks that incorporated the input of every team member.

Martin - My main responsibility in this project was designing and implementing the VGA subsystem that drew all visual elements of the game. I began with initial testing of VGA displays by creating simple test patterns such as RGB color bars to verify proper timing and signal generation in our vga controller. After I got that working, I implemented the

VGA related modules such as vga_control, vga_top, and the three sprite bitgen modules that handled individual sprite rendering. Using Piskel, I also made all of our sprite images and used a Python script to convert them into the Hex RGB565 format. Lastly, I designed the ROM port multiplexing strategy to share memory between sprites and created the priority-based multiplexer that implemented proper sprite layering.

Lessons Learned

Through this project, we learned how important it is to understand both low-level protocols and high-level system design. Working with the PS2 interface showed us the need to interpret scan codes correctly and connect hardware modules in a clean and reliable way. We also realized how critical timing is in digital design, especially when handling signals like keyboard inputs, register updates, and controller states. Another key lesson was the value of systematic debugging. Testing the ALU, register file, memory, and controller/decoder taught us to isolate problems carefully, check waveforms, and verify each module step by step.

We also ran into design issues when designing our CPU. When we started we tried to stay consistent with the CR-16a ISA [2] as defined per the instructions, but also ended up trying to implement according to the different ISA that was linked in the Lab 1 module. This resulted in our code having a backwards CMP instruction, and a src, dest order for each instruction except for the LOAD instruction. This highlights the importance of ensuring that everyone in the group is operating under the same instructions and each decision is clearly defined and documented.

In terms of VGA, this project taught us how to draw onto VGA displays as long as you can get the precise timing signals and clock frequency [3]. We also learned how to do chroma-keying with a non-black color in order to draw our sprites with correct transparency. When drawing multiple sprites, we learned that resource sharing through multiplexing can significantly reduce hardware utilization while maintaining performance. Lastly, modular design with clear interfaces between components really helped when testing and debugging the system.

References

[1] “PS/2 Controller” Department of Electrical and Computer Engineering, University of Toronto. Accessed: Dec. 2025. Online: https://www.eecg.utoronto.ca/~jayar/ece241_08F/AudioVideoCores/ps2/ps2.html

[2] “CompactRISC - CR16A Programmer’s Reference Manual,” The College of Engineering, University of Utah, https://my.eng.utah.edu/~cs3710/handouts/cr16a-prog-ref.pdf (accessed Dec. 8, 2025).

[3] Pio Assembly VGA driver for RP2040 (raspberry pi pico). VGA. (n.d.). https://vanhunteradams.com/Pico/VGA/VGA.html

[4] Adam@SheekGeek, & Roberto. (2020, July 3). RGB565 to RGB888 color conversion. SheekGeek. https://sheekgeek.org/2020/adamsheekgeek/rgb565-to-rgb888-color-conversion

[5] Playing with the pico part 6 - SNES like sprites and tilemap with VGA. Greg Chadwick - Playing with the Pico Part 6 - SNES like sprites and tilemap with VGA. (n.d.). https://gregchadwick.co.uk/blog/playing-with-the-pico-pt6/