Battleship

Jaime Campos, Josh Cobian, Jacob Edmundson, Alan Ekstrand, Zach Simons

https://github.com/ZachSimons/Battleship

Project Final Report

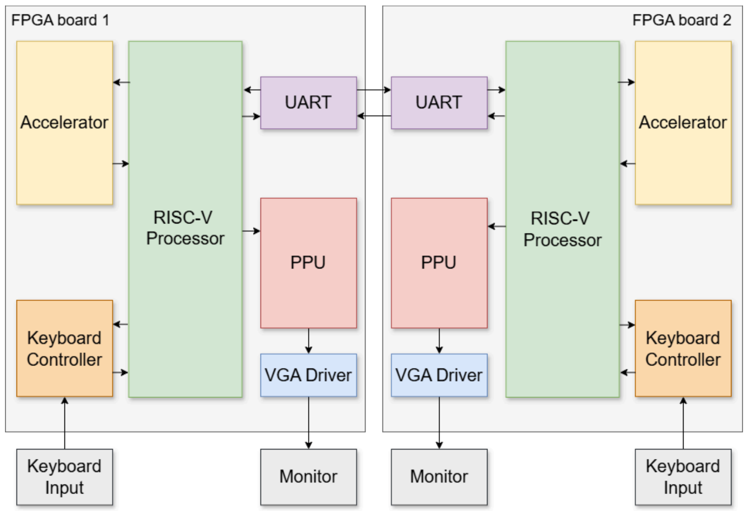
University of Wisconsin-Madison

Objective

Our goal for this project was to design a 32-bit RISC-V processor with custom instructions that we can use to run the two-player game “Battleship” on. This processor will interface with a number of external IO devices, including a VGA monitor, PS2 keyboard, and another FPGA board.

Background

We decided to emulate the game Battleship specifically because of its interaction with two players and turn-based play. In addition, since the game is simple enough to code and display on a screen (very little animation needed), we would be able to focus more on the hardware and processor design rather than spending a significant portion of time working on writing the game and creating the visuals. With this game, we would explore many types of IO and interfaces, including the VGA interface, PS2 interface, and UART.



High Level Diagram

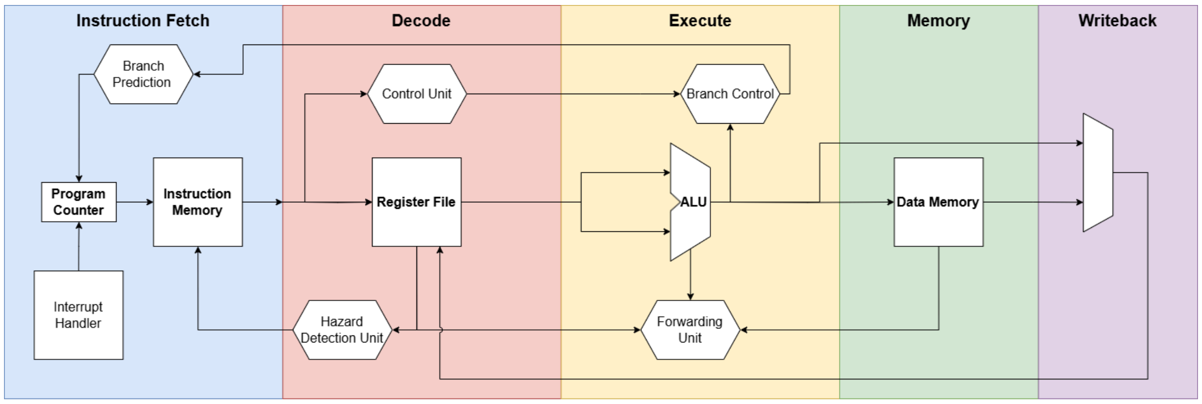
We chose the RV32I ISA since our game required relatively simple instructions, and we could easily extend the ISA by adding custom instructions that make the interrupts and interaction with the external devices simpler and more efficient. With a custom-built assembler, we were able to convert our C game code to machine code and execute it with our processor, displaying Battleship on the two screens and allowing players to duel each other.

Technical description

Processor:

Our processor was a 32-bit RISC-V processor running a modified version of the RV32I ISA. We removed the instructions for communicating with the OS and added custom instructions for managing interrupts and communicating with peripheral hardware such as the accelerator, PPU, and other FPGA board.

The processor is a five-stage pipelined processor with fetch, decode, execute, memory, and write back stages. We implemented register file bypassing in order to avoid stalling for data hazards. In addition to register file bypassing, we also added EX-EX and MEM-EX forwarding paths. We implemented branch not taken branch prediction with jumps and branches resolved in execute. We also added a linear feedback shift register (LFSR) to generate random numbers for our code. The LFSR is seeded on reset using the current configuration of the switches on the FPGA board when the board is reset. On top of this, we have various custom instructions that our processor uses to send values off the processor, such as giving values to our accelerator, PPU, or UART module.



Processor Block Diagram

We decided for this processor to also only use Block RAM for the data and instruction memory. We did this primarily for access to low-latency memory. We also decided to split up the instruction memory and data memory, as it simplified memory routing within the processor.

We also decided to implement interrupts on our processor to assist in receiving keyboard data, as well as for communication between boards. We took precautions to avoid issues with interrupts by ensuring only one interrupt can be processed at a time to avoid interrupts overwriting each other, holding interrupts and their data around loads to avoid issues with stalling, and rerouting pc's when an interrupts occurs in proximity to a branch or jump to ensure the interrupt goes to the interrupt handler and returns to the correct location. We also have multiple ways of returning from interrupts in the form of returning to where the code left off, or returning to a given pc value, acting like a jump.

Accelerator:

The accelerator's purpose is to increase the speed at which our 'next-best-move' Monte Carlo algorithm can run. The algorithm generates five random ship locations, one for each ship, and then checks to make sure that they don't overlap with each other, or any invalid board space, such as a miss space. Because the algorithm runs in a loop, there are a lot of comparisons being made that take a large amount of instructions to complete. The accelerator replaces a portion of this by doing the comparisons in parallel as a single instruction. The accelerator consists of five ship registers, storing their locations, lengths, and orientations, as well as a game state register that is updated to reflect the current state of the board. Each update-ship-register instruction provides the data for two ships, allowing the total instruction count to send the ship data and get the response to only four instructions. It then uses the gamestate register as well as the ship registers to do parallel comparisons, outputting a 1 or 0 depending on if the given state is valid or not.

Software:

Our software was written in C. It manages the game logic of battleship, storing the location of your ships and the information that you know about the enemy board. It handled input from the keyboard via an interrupt handler to decode what key was pressed and respond accordingly. In addition, the interrupt handler controlled sending and receiving data from the other board in order to keep the games synchronized and communicate updates to the game state. Finally, the software is in charge of sending data to the PPU to communicate the current state of the game so that the correct sprites are drawn to the monitor.

In addition to the game logic, the software also ran a best move algorithm to give the user a suggested next target. This was a Monte Carlo algorithm that produced a heatmap on the grid to determine the probability of a ship being present on that square. The algorithm worked by generating a random assortment of the opponent's remaining ships and verifying that it was a legal configuration. It verified the legality of the configurations by checking that a) no ships overlapped with each other, b) no ships were placed on a square that was already known to be a miss, and c) all squares that were already known to be a hit had a ship placed on them. It repeats this process for a large number of random configurations and then determines which square was present in the most legal configurations.

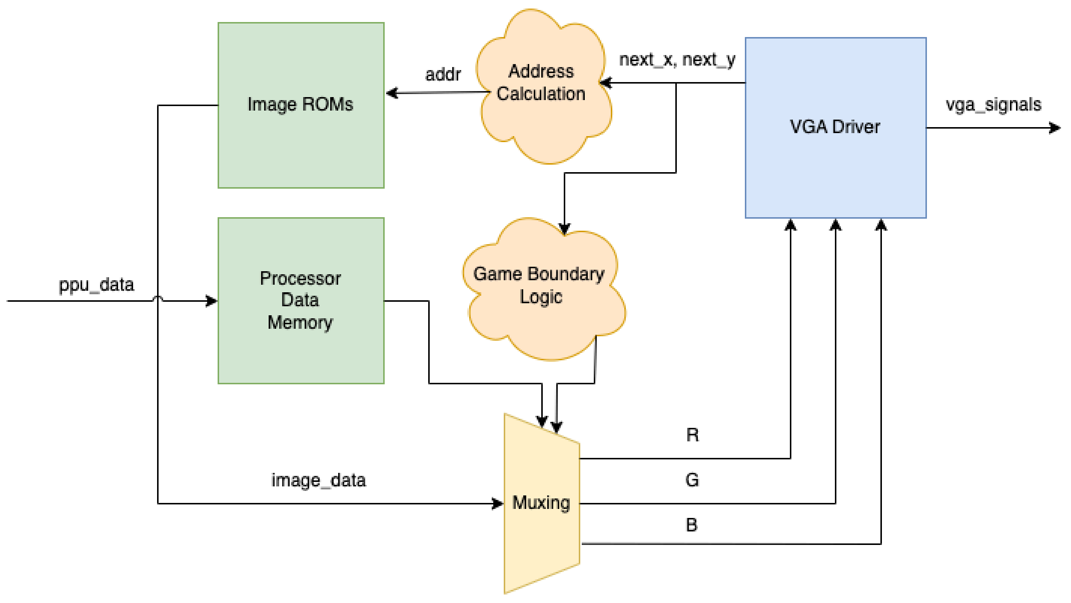
There were two different versions of the software, one that made use of the accelerator on our processor and one that did not. The accelerator would verify the legality of the ship configuration, significantly reducing the runtime of the algorithm.

PPU:

The PPU is the hardware that takes in data from the processor and outputs sprites onto a display through the FPGA's VGA port. To drive the VGA signals needed by the FPGA I borrowed a VGA Driver from a professor at Cornell. This driver was quite simple as all it needed was an 8-bit **color_in** signal, and outputted all the VGA signals along with **next_x** and **next_y,** which are the coordinates of the next pixel drawn.

I redesigned the driver so that instead of taking in just one 8-bit RGB signal, it takes in 3 8-bit signals for R, G, and B. This is so we are not stuck at using 8-bit color and can go up to 24-bit, which is the VGA interface's max color depth.

To design the rest of the PPU, I ended up using around 20% of the BRAM to store sprite data onto the board. This data would be accessed from addresses calculated according to the coordinates of the top left corner of the image, height, width, next_x, and next_y. After retrieving the RGB data from memory, we would have to choose which image to send to the VGA Driver. This required several muxes where the select signals would be based on the data sent from the processor, and what game boundary we are in (ex. drawing left board vs. right board).



Communication:

The way we decided to handle communication was through hardware interrupts. We did this to avoid polling, which wasted time where the processor could be running the prediction algorithm. When either a communication was received or a keyboard button was pressed, an interrupt would be sent to the processor, which would cause the processor to jump to the interrupt handler. Code would then run to handle the interrupt, and then the processor would return to the spot it saved when the interrupt originally happened. Only one interrupt can happen at a time to avoid unpredictable behavior regarding the interrupt handler. If two interrupts happened in quick succession, the second interrupt would just be dropped. This wasn't a problem as, if an interrupt was dropped, it could just be sent again at a later time.

Board to Board Communication:

After deciding to switch from using Ethernet because of its inefficiency and unjustifiable design complexity, we created a custom UART module to send the 3 bytes containing game state updates between FPGAs over GPIO. Our initial UART design was based on the SPART module, but due to challenges and instability encountered with sending 3 separate standard UART transactions, we modified the module to simply send one 3-byte data packet in a single transaction. For easy validation of the data being transmitted, each board displays the received data on its 7-segment display.

Keyboard:

The keyboard uses the PS/2 port connected directly to the FPGA. The way the PS/2 protocol works is that whenever a key is pressed, a clock pulse is sent to the FPGA along with a unique byte. We built a wrapper to interpret based on what byte was sent, output the direction the player moved or if they fired with a "done" signal. Once the done signal gets asserted, this will send an interrupt to the processor.

Challenges, workarounds, & lessons learned

Our biggest challenges were integrating all the different parts of our processor together. While each individual piece of the processor was tested, when these pieces were combined they did not always work together immediately. We learned that integration always takes longer than expected.

Another major challenge was our use of IP memory blocks. We were concerned about the limited amount of memory available on the FPGA boards, so we decided to use IP memory because they are very efficient. However, the IP memory blocks introduced extra stalling into our processor since the IP memory blocks do not have a single cycle design. This caused issues with other aspects of the processor, such as what to do when an interrupt occurs while the processor is stalling.

Contributions of individuals

Jaime Campos:

I had the task of building the PPU which would display sprites given data from the processor, and the Keyboard Driver which sent direction and firing data to the processor. For the PPU I borrowed some sprites from the NES Version of Battleship. However, these sprites weren't perfect and had to do some photoshop to fix the board and fit each of the ships on the grid. I used water and shoreline sprites designed by Jacob and tiled them into the background of the game. The reason for tiling was due to memory limitations of loading in a full 640x480 image into ROM. I also integrated Jacob's turn, title, and game end sprites into the PPU so they could get displayed.

Josh Cobian:

My primary responsibility was putting together the custom RISC-V processor. I helped with deciding the ISA and Creating the custom instructions. I then wrote both the memory stage and the initial fetch stage within the processor. For the memory stage I had to determine what type of memory to use, how much we planned on using and whether we wanted it byte addressable. For the fetch stage I mainly created the initial functionality with branching, PC register, and the instruction memory. For processor integration worked on forwarding, Memory Stalling, Hazard Detection, Branch Prediction and implementing custom instructions. I also spent a lot of time debugging and writing test benches for the processor using the Java assembler. After the processor was integrated I then worked on debugging the UART communication and helped build a version that was extremely reliable. Once that was completed, I mainly worked on the code and provided insight into the processor whenever necessary.

Jacob Edmundson:

My contributions shifted continuously during the semester as the team's immediate needs changed. In the beginning, I was exploring Ethernet as our initial board-to-board communication method, but we realized it would be extremely inefficient due to the Ethernet standards' minimum packet length of 64 bytes, and we only planned to send 3 bytes per turn. Because of this inefficiency and design time requirement, I decided to shift my focus to the processor, as it was the cornerstone of the project. Once the processor was assembled, I created the test bench for it and started debugging. I spent the majority of my time improving the test bench and debugging the processor, adjusting the logic as needed. I also created an automated test script that made running all of our tests much easier. After the processor was mostly tested, I did most of the high-level RTL integration, and then I worked on creating the UART module. With most RTL done, I helped create game sprites (background, title, turn, game end screens).

Alan Ekstrand:

My primary responsibility was creating the software for our processor. I wrote the battleship code running on the processor in C. I created two different versions of the code, one with the accelerator and one without the accelerator. I used the RISC-V GNU Compiler Toolchain to compile the C code into RV32I assembly code with our custom instructions added as inline assembly directives. After compilation, I would manually check the assembly code in order to verify that the compiler did not have any conflicts with the inline assembly instructions that we added. Finally, in order to assemble our assembly code with the custom instructions in our ISA, I wrote an assembler that was capable of interpreting RV32I extended with our custom instructions. This assembler was written in Python and outputs a hex file containing the machine code that could run on our processor. This file was then loaded into our instruction memory when the FPGA board was programmed, allowing our processor to run the battleship code.

In addition to writing and debugging the code, I also worked on the processor. I created the execute stage of our processor. This included the ALU and all its operations, handling bit shifts, and branching logic. Once the processor was created, I worked on testing and debugging on the processor. Finally, I worked to integrate the processor with the other elements of our project, including the keyboard, PPU, UART, and the accelerator.

Zach Simons:

My primary contributions were designing the RISC-V processor and the accelerator. I started by helping design the extensions to the ISA, working on decode, forwarding, hazards, and implementing custom instructions. These consisted of writing a register file with RF bypassing, an instruction decoder, and the connecting decode logic, as well as a hazard detection module and forwarding logic. During each large stage of development, I also wrote testbenches to make sure the additions I was making worked as intended. Further on, I worked across the whole processor for integration and bug fixing, which was a substantial portion of the project. I also worked on implementing the interrupts on the processor. On the accelerator side, I wrote the whole accelerator which consisted of the register updating logic as well as the combination logic to handle the parallel comparisons. On top of these I contributed to the overall testing and debugging of the combined project which included writing testbenches, assembly test programs, and debugging various systems across the project such as our UART module.