# 3710 Final Project

Charlotte Saethre, Kirra Kotensburg, Nicholas Ronnie, Sean Koo
UNIVERSITY OF UTAH - COMPUTER ENGINEERING
SALT LAKE CITY, UT

## ABSTRACT

For our final project in ECE 3710 we built a microprocessor that implements a baseline CR16 RISC architecture. We decided to use our microprocessor for a game application. The game we decided to implement was a version of the classic Space Invaders, which we call Stellar Trespasser's. This game involves a VGA output and PS/2 keyboard as the input for the user to move the player ship and shoot projectiles at enemies that attack. The results of this project turned out to be satisfactory with a baseline system that functions properly to execute the CR16 ISA - extended for our application purposes - and I/O subsystems that are required for the target application.

## KEYWORDS

CR16, RISC, Microprocessor, Space Invaders, Game Application

**ACM Reference Format:**
Charlotte Saethre, Kirra Kotensburg, Nicholas Ronnie, Sean Koo. 2023. 3710 Final Project. In *Stellar Trespassers, Dec 9 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/XXXXXX

## 1 INTRODUCTION

For our ECE 3710 semester long project we were to build a CR16 baseline system to process the CR16 ISA. With this baseline system we were to design an application that this microprocessor would be built for. The baseline system includes the following components: arithmetic unit, register file, datapath, and controller. These work together to process CR16 instructions. The application we decided to go with was a game application. The game we chose was something similar to the classic Space Invaders game. For this application we will need some kind of I/O sub systems such as input for the user to move a player ship and a way to draw sprites on the VGA display.

## 2 PROCESSOR DESIGN

In describing our baseline CR16 processor design we will start off with the top level module, cpu. The cpu module instantiates the exmem module which contains access to the external memory, the controller fsm which decodes our program instructions, the datapath which represents the flow of data while executing instructions, and the spriteVGA module that handles all of the graphics processing. We would also have a ps2 module at this level but opted to use the push buttons on the FPGA themselves. We have external inputs to the cpu that feed the push buttons states. We used three buttons on the board for left movement, right movement, and shooting projectiles. We used a mux in between our exmem and datapath to determine if we need to output to the program the push button data for external input when seeing an address come through the program that was our special keyboard address.

Our controller module is a basic finite state machine that accomodates the decoding of our baseline CR16 instruction set. We also include the NOP instruction which is mentioned in more detail below. The datapath module includes all of the logic for storing/loading to.from memory and into our registers. This module includes the alu (Arithmetic Logic Unit) and the regfile. For our 16-bit microprocessor we have 16 16-bit registers and a 16-bit address space.

Our exmem is a dual-port RAM. Port-A was used for the program data and also reading input from the push buttons. Port-B was strictly READ-ONLY memory that was used exclusively by the spriteVGA.

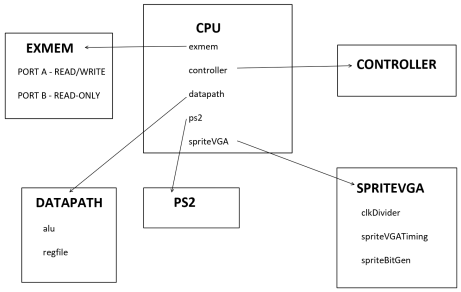


**Figure 1: High Level Block Diagram of CR16 system**

### 2.1 Target Application

The target application for our microprocessor is a game application. The game we wanted to replicate was Space Invaders. This makes graphics highly important and having some kind of external input readily available so our program can execute game logic. The game logic in which we needed to have is the ability to move a ship from left to right and allow a user to shoot projectiles on command that line up with enemy ships. We would also need to have enemy ships be generated in waves that try to reach the player ship after so many game loop cycles. Upon shooting an enemy ship on target the enemy and projectile should disappear detecting this collision.

As our application heavily relies on graphics and external input it was necessary to focus our time and energy on getting a working VGA framework unit and look into ways of getting the necessary input from the user. This required us to have VGA and PS/2 modules.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CR16 '23, Dec 9, 2023, Salt Lake City, UT, USA*
© 2023 Association for Computing Machinery.
ACM ISBN XXXXXXXXXXX…$15.00
https://doi.org/10.1145/XXXXXX

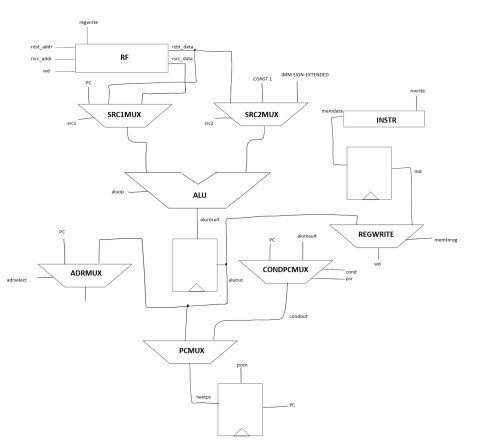


**Figure 2: Diagram of datapath**

Our baseline system needed to be extended to include the NOP instruction which is necessary for creating a delay so game objects can move on the screen at a rate that can be seen by the human eye.

## 2.2 Additional Features

An additional instruction we decided to include on top of the baseline system was a NOP instruction. The NOP instruction is basically a no-operation instruction, it does nothing. Our specific use case for including a NOP instruction was to delay the game between updating game state so that the VGA can display the game objects in the right places at a rate that can be seen by the human eye. This is usually achieved with an FPS (frames per second) that is equal to 30. The way we implemented the NOP instruction in our controller module was when we encountered the NOP instruction we started a count and stayed in this state, basically idling, until we hit a certain count value - this count value was chosen to achieve the 30 FPS goal.

The other additional features to our system include the VGA subsystem and PS/2 subsystem.

# 3 APPLICATION DESIGN

## 3.1 IO

*3.1.1 Sprite VGA Output.* The game design as a whole operates using a 20 x 15 grid, with each grid location capable of displaying a single sprite. The VGA module has control of one of the two ports that enable access to the memory unit. This allows the VGA module to retrieve data concerning what should be displayed on the screen independent of the state and behavior of the CPU module. During the initialization phase and the repeating game loop, the VGA module reads in from agreed-upon data locations in memory for the following items: Enemy Count, Enemy Data, Projectile Count, Projectile Data, Ship Data, and IsGameOver. A grid of sprites is constructed over a short period, starting from a blank register of size 20 x 15. When a visible Enemy, Projectile, or Ship is loaded from memory, the corresponding grid location is set to represent the sprite located in that location. Once a complete grid of sprites is constructed, representing the current sprites that are stored in memory, this temporary register representing what should be displayed is latched into a different register that is referenced for the displaying of sprites on the screen. H Count and V Count values are utilized to determine which grid spot and sub-position within that grid spot the H Count and V Count are at. The color that is displayed is communicated using the 2-bit value of the sprite, while the choice to either draw that specific pixel at all is communicated using a 1 or a 0 within the 32x32 representation of each sprite that is initialized earlier. The additional feature of a Game Over screen was implemented using a specific memory address that stores the IsGameOver bit in the MSB and the players score in the remaining 15 bits. Once the Game Loop has determined that the player has lost, the MSB of this memory address is set to 1, and the VGA switches the way it displays data on the screen. A preprogrammed GAME OVER grid of sprites is loaded into the displayed sprites register, and the player score is seen in the bottom right using a binary representation, with a projectile representing a 1, and a enemy representing a 0.

*3.1.2 PS/2 Input.* The goal with our game input was to use right and left arrow keys to move the ship and use the spacebar to shoot projectiles from a PS/2 keyboard. The reason we used PS/2 is because this protocol is much more straightforward then dealing with USB. Although we created a PS/2 module we ended up not using in our final game application as it had some issues and inconsistently worked. The PS/2 protocol involves a clock that is from the keyboard itself and data that comes in serially (1 bit at a time). We sample the data on the falling edge of the ps2 clock. The data packet for the key being pressed is 11 bits that contains 8 bits of data. The encoding for the keys is a 2-hex value. How we designed the module to take in data from the keyboard was to use a buffer that is 8 bits wide. As we sample data on the falling edge we start a counter that collects the ps2 data and puts it into our buffer. When we get to the count that is after the last data bit we set a flag to high and enter another always block to check what the code was that we saw on the previous iteration. As a button is pressed the ps2 protocol sends the 2-hex code continuously and then when it is released it sends a key-release code, which we check for to indicate a key has been pressed. Depending on which key code we collected we send out a 1, a 2, or a 3, for left being pressed, right being pressed, or spacebar being pressed, respectively.

We were able to test this module by lighting up LEDs on the FPGA that correspond to each of the buttons being pressed. When connecting our system to the PS/2 module we found that it didn't always move the ship consistently. And when it did it had a bug where it would move in the opposite direction and jump back to the current location of the ship before actually moving in the right direction based on the key that was pressed. We tried to debug these issues and found that we ran out of time to get it working consistently so we decided to settle on using the push buttons on the FPGA itself as this worked consistently and we wanted to focus more on writing game logic at this point.

# 4 APPLICATION IMPLEMENTATION

*4.0.1* ***Initialization.*** During our initialization section at the top of our program, we reserved registers r10 to r15 for holding memory

addresses for our game objects, the PC counter for the game loop start, and the game loop count. We chose to do it this way to ensure that we could use these memory addresses throughout the rest of the program. We initially placed the player ship in the bottom left corner of the screen and set up four enemy ships at the top of the screen for our initial wave of enemies.

*4.0.2* ***Game Loop.*** Inside our game loop, we generate new waves of enemies and then also check for user input to see if the ship needs to be moved left or right or if we need to shoot projectiles. We also check for project enemy collisions to eliminate both upon impact.

*4.0.3* ***Enemy Generation.*** At the beginning of the game loop, we check for the game loop counter being at certain values to create a new enemy wave pattern. We use 16 bits and then 4 bits to make up a pattern of where enemies will appear for the 20 x-coordinates that we have available. So, at different loop counts, we will be generating new waves of enemies. The enemies will appear at the x position determined by the pattern we create for that iteration. We also include logic to advance the enemies in the vertical direction so they approach the player at the bottom of the screen.

*4.0.4* ***Collision Detection.*** We check the position of the projectile and see if there is an enemy in the same position. When we find a projectile enemy collision we turn these two game objects to be invisible and update the projectile and enemy counts that we are keeping track of throughout the game. We use the counts to determine how many projectiles/enemies are visible on the screen so the VGA can draw them.

*4.0.5* ***Game Over.*** Once any of the enemies reaches the bottom of the screen we jump into a game-over section of the assembly that ends the game. The display will place Sprites on the screen to spell out GAME OVER on the screen. This is also where we made room for a score to be displayed in binary at the bottom using the alien ship to represent 0 and the projectiles to be 1. This aspect could not be fully implemented, so our final project has a hard coded example of a player score instead.

# 5 CONCLUSION

The main features of the project include the CPU module that contains all of the logic to execute the CR16 ISA, the VGA module to process our graphics, and the kb3.asm as our main program for the Stellar Trespassers game.

## 5.1 What was done well

Designing and implementing the instruction set for our system went well. We were able to add in a few extra instructions like No-op to keep the fps of the game at a reasonable rate. The timing module for the VGA took time before working but provided no issues after that. The bit generation module was a creative design and not what was originally planned. Turning the whole display into a 20x15 grid simplified placement for our sprites. We worked on the assembly code in pieces so that we could test some functionality and build off of what we got working. This made it a bit easier to find problems inside the code as well since we could often narrow down the errors to the new section being added.

## 5.2 What could have been better

*5.2.1* ***The PS2.*** Incorporating the PS2 could have gone better. We managed to get it integrated with our CPU, it could move the player's sprite when the 'a' and 'd' keys were pressed (acting as arrow keys for the user). However, it worked very inconsistently. Often we would have to do a hard reset of the whole system and recompile our code in Quartus to get it to function. Then when it was working there was a strange glitch where the first time a directional key was pressed, the sprite would move forward one space and then move back to its starting position. We spent a long time looking for the error in the code before deciding to move on and use the FPGA's push buttons for user input instead.

*5.2.2* ***Reset.*** Throughout our project, the reset button never functioned correctly. When it was pressed the display would reset incorrectly and unpredictably. All the sprites would appear in random places, or disappear entirely. We added reset logic into the assembly code and scoured our Verilog, but couldn't find the issue and get this aspect working in time.

*5.2.3* ***Score.*** We wanted to track the player's score and have it displayed at game over. This was more of an issue with time and this feature of our game was a stretch goal for us. Given more time this likely would've been properly incorporated.

*5.2.4* ***Closing Statement.*** Despite the setbacks we ran into, we did manage to get more of the game's functionality completed than expected at some points. The player ship moves and shoots when prompted. Alien ships invade and come in waves. Projectiles hit the aliens and make them disappear. And we got the game over screen working. Altogether, the game does function as intended.

# REFERENCES

https://projectf.io/posts/hardware-sprites/