Circuit Bees Group’s UrCPU

Breeze Tucker   
*Computer Science   
Ursinus College*Collegeville, USA  
brtucker@ursinus.edu

Michael Connors  
*Computer Science   
Ursinus College*Collegeville, USA  
miconnors@ursinus.edu

Bev Berrodin  
*Computer Science   
Ursinus College*Collegeville, USA  
evberrodin@ursinus.edu

# Introduction

The UrCPU addresses the need for a reliable and efficient microprocessor design. With a focus on memory reliability through the implementation of parity-check error correction code, this project aimed to create a versatile 20-bit microprocessor capable of storing and executing programs while ensuring data integrity. This project intended to implement a von Neumann style microprocessor that meets these requirements and provides opportunities for expanded functionality.

To tackle all of the complexities of designing the UrCPU, our team adopted a systematic approach that involved research, brainstorming, and development. We began by analyzing the project specifications and identifying key design requirements, such as the architecture specification, instruction set architecture, and error correction mechanisms. Through regular team meetings and discussions, we created a comprehensive solution strategy that encompasses the design and implementation of various hardware components, including registers, ALU, memory bank, and the control unit, using Verilog. Additionally, we devised test benches with representative and corner cases to verify the functionality and performance of each module. We aimed to deliver a fully functional microprocessor that not only meets the project requirements but also demonstrates innovative features and potential for future expansion. Ultimately, we found varying degrees of success across all aspects of this project.

# Hardware Design

## The overall design of this project was comprised of various hardware components, including registers, the control unit, and the Arithmetic Logic Unit (ALU). Each component was implemented to fulfill specific functions within the microprocessor. By using Verilog we were able to mostly complete our hardware design goals. We aimed to create a cohesive system that not only met the project's specifications but also demonstrated potential for future expansion and later optimizations. Overall, the project's design was partially successful in completing these goals.

## Registers use the D flipflop design discussed in class, specifically using the controlled variant to only write when given the write signal. This results in each register needing 20 flipflops and a shared Write Enable signal, for a total of 21 input bits. Only the Q output bits are kept, so there are 20 output bits as well, with 20 discarded notQ bits.

## The control unit takes advantage of a number of multiplexers, demultiplexers, and decoders in order to correctly route data. Multiplexers are used for incoming data, like values taken from registers, while demultiplexers are used for outgoing data, like data sent to write to a register. A decoder is additionally needed to enable writing on the correct register.

The Arithmetic Logic Unit, also known as the ALU, is dedicated for all numerical operations performed by the Ursinus Central Processing Unit. These operations are divided into five main categories, with each category getting its own verilog file for modules. These categories are program flow, logic, bit shifts, arithmetic, and comparison. Originally we had a unique file for each module, but we made the decision to group multiple modules into files in order to keep our code and files more organized. Most of the numerical operations will output a number that has been altered to be the answer of the required instruction performed on the input(s), with some even taking a carry into account even though it was not necessary for a three person group. The rest of the operations are dedicated to updating flag values.

We had issues with the compiling of our Verilog code within some of our modules which forced us to restrict the UrCPU’s capabilities. With a greater understanding of how Verilog is used and how we could use it to best suit our needs, we may have been better adjusted to complete all of our hardware design goals. However, we are satisfied with the modules which we were able to fully complete and optimize.

# Instruction Set

Every instruction is a 20-bit word divided evenly into four sections: a 5-bit opcode, two 5-bit input addresses, and a 5-bit write address in that order. As is, this allows for more than the required number of unique opcodes for our three person group, as well as the ability to index through our entire 32-bit register array. Access to the 64-bit memory is not directly possible with this convention, and would rely on passing an address already stored in a register.

The following instructions are implemented at a high level at time of writing: no operation: NOP, the logical functions: NOT, AND, OR, XOR, mathematical functions: INC, DEC, ADD, SUB, and comparisons: EQ, GT, LT. A complete list of planned opcodes can be found in opCodes.txt.

The following an example programs one might write for this architecture (commas added for readability):

01000,00000,00000,00000

#NOT x0, x0, x0 / x0 = NOT(x0)

01000,00001,00000,00001

#NOT x1, x0, x1 / x1 = NOT(x1)

01100,00000,00000,00000

#SHFTR x0, x0, x0 / x0 = SHFTR(x0)

01101,00001,00000,00001

#SHFTL x1, x0, x1 / x1 = SHFTL(x1)

01001,00000,00001,00010

#AND x0, x1, x2 / x2 = AND(x0, x1)

10101,00001,00010,00011

#SUB x1, x2, x3 / x3 = x1 - x2

# Implimentation

The implementation of the Arithmetic Logic Unit was where we began working on the UrCPU. It was definitely the right decision because it was not very hard, so it let us get used to Verilog’s syntax before moving onto the harder parts of the project. The ALU is also the most thoroughly tested part of our project, with each file having one or multiple testbenches. This really let us polish the ALU into a product that we are very proud of.

However, the other parts of the project were not so easy to implement. The control unit and the memory bank were the biggest roadblock in our way to creating a working central processing unit. We were forced to save them for last since they utilize many operations that would have to be implemented first like multiplexors and decoders. With their increased level of complexity and less time allocated to be able to solve them setting us back, the control unit and memory bank could be improved on in the future.

The biggest problem we faced was the deadline. With the course project being introduced to us at the start of the semester, we thought that we had plenty of time to work slowly. However, we did not accumulate all of the required knowledge to finish this project until very close to the end of the first semester. Due to the time constraints, not every aspect of the project was able to be implemented and thoroughly tested. With more time, it is certainly possible that we would have been able to solve the problems that we could not dedicate enough time to solve them.

##### Conclusion

Of the many challenges we ran into with this project, a lot of them were related to the limitations brought about by using Verilog, in addition to simply getting used to its syntax and conventions. One major example of this can be found in the mux/demux/encode/decode attempts made in FlowControl.v. The challenge here was differentiating the wires and identifying which was which. In conventional programming, this could be achieved by comparing a value to the iterator used in a for loop, but this is not the case for Verilog’s generate blocks, which do not give you access to a genvar (its iterator) during runtime. In the mux implementation, we see that you can simply index the input based on the selector, but this is not possible when indexing the output in the demux circuit.

Creating each module for each operation was very enjoyable, and it was fun to implement all of the operations in a vacuum. However, it was very difficult to try and link them together. The control unit and memory were definitely the most complicated parts of the project to implement since they are connected to most operations in some way. A possible extension would be to fully flesh out the missing parts from memory. It would also be helpful to have wide-scale testbenches. Most of our testbenches are to ensure that a specific module is working, but we currently do not have a broad testbench to make sure that everything works well in conjunction with one another.

The development of the UrCPU has been both challenging and enlightening. We successfully learned about the complexities of hardware design, by using Verilog to implement the UrCPU’s components such as registers, ALU, and the control unit. Despite our hurdles, particularly in refining Verilog syntax and addressing its limitations, we continued to deliver a mostly functional microprocessor capable of executing a range of instructions. However, time constraints certainly hindered our ability to fully realize the project's potential and conduct testing across all of its components. Moving forward, addressing the remaining challenges, such as refining the control unit and expanding memory functionalities, could enhance the UrCPU's versatility. Additionally, implementing more comprehensive test benches to further develop the system's overall functionality would be crucial for ensuring its full capability in real-world applications. Despite the obstacles faced, the UrCPU project has provided valuable insights into microprocessor design and Verilog programming, laying a foundational understanding of microprocessor design and implementation within our group and among our peers.

##### References

No outside references were used.