The Design and Implementation of a 20-Bit Microprocessor

Mia Sideris
Ursinus College
CS 274
Collegeville, U.S
misideris@ursinus.edu

Julian Faust
Ursinus College
CS 274
Collegeville, U.S
jufaust2@ursinus.edu

Dante Ottaviani
Ursinus College
CS 274
Collegeville, U.S
daottaviani@ursinus.edu

***Abstract*—We will discuss the design and implementation of a 20-bit microprocessor that satisfies a number of core requirements that allow for its effective coordination and performance in respect to the clock time. The UrCPU's strategy of design seeks to minimize redundancy without posing a cost on the functionality of the circuitry. With this technique, we have combined the fundamental elements of a simple microprocessor, including general, incremental, status, and pointer registers, one arithmetic logic unit, one internal bank of memory, and one control unit. As a result, less internal memory will be utilized in the interactions between these components, and the UrCPU's performance capabilities can be maximized to the furthest extent possible, while considering the trade-off of sizing constraints.**

I. Introduction

The purpose of this project is to design and effectively implement a simple 20-bit microprocessor in a hardware description language, Verilog. We will refer to this 20-bit microprocessor as the Ursinus Central Processing Unit (UrCPU). The UrCPU is reliable on memory through the use of error-correction code with a majority multiplicity rule of three. Thus, the general capabilities of the UrCPU include storing short programs in memory, executing programs by fetching, transforming, and storing pieces of data, and correcting memory errors. The UrCPU encompasses a multitude of specifications which provide the foundation for these capabilities. Such specifications include six general purpose registers, six program pointer/segment registers, one status register, three increment registers, one arithmetic logic unit (ALU), one bank of internal memory, and one control unit. Each of these components are connected within a singular top-level module with a respective testbench file. Additionally, each specification of the UrCPU is implemented within individual Verilog modules, each complemented with representative and corner cases. As a result, each component's output can be evaluated with greater ease in an organized manner. Finally, a script will drive the execution pipeline that will include individual tests to run the code in console, and generate signals obtained with the use of GTKWave.

A generalized strategy of implementing the UrCPU begins with roughly sketching a design of the microprocessor in a way that will connect each of the specified elements without redundancy. For example, it is important to consider the data path connections between each listed register in a way that will minimize the number of connections. In other words, we must consider the economics of implementing one connection over another, and whether we can implement the same concept using alternative methods that are less costly, or mindful of the amount of memory usage. Consequently, the next step to designing the microprocessor is to implement the ALU where arithmetic and logical operations will be performed based on the inputs it receives from the respective registers or based on the values that have been stored temporarily within the bank of internal memory. The ALU must be able to perform a plethora of operations that can handle the various outputs from the registers, while also being efficient in respect to the clock time. Thus, each operation design is carefully implemented with minimal instructions. Finally, due to the complexity and significance of the control unit, it was a sensible design strategy to begin its implementation early in the process of creating the microprocessor. Being that the processor is responsible for generating the control signals that facilitate the operations that take place in other areas of the microprocessor, such as the ALU. Thus, it is crucial to surround the UrCPU's development around our initial understanding of the control unit's design.

II. Hardware Design Decisions

As previously discussed, the primary design strategy for the UrCPU is to remain mindful of the amount of memory being consumed by each operation. Not only will redundancy result in slower clock times for various test cases, but exceptional amounts of code will increase the chances of logical errors being made throughout the design process. It is rather efficient to reuse the various aspects of our code in other areas where the same logic can be applied, without drastically changing the functionality of the operation.

Consider the register specifications as previously mentioned. Each register (general, segment/pointer, status, increment) was designed in such a way that a flag would act as a signal each time a particular condition was met, like the overflow flag implemented in the status register. The overflow flag would signal whenever the output of some operation expected to surpass the established range of data. Similarly, an underflow flag was also implemented within this register, which would act as a signal whenever the output is expected to be a value too small to be represented by the preset range of data. Given that these two registers go logically hand-in-hand with one another in terms of functionality purposes, we can use the same design method for both. Instead of completely re-writing the logic behind both signals, we can implement the same design method, and only find it necessary to alter a fraction of the design itself. Thus, reusability is a key factor in determining the design method for each register in order to minimize errors and verify the connections between the registers and the input data from the ALU in a more effective way.

Consequently, the ALU is designed in such a way that it operates in full-word mode. The ALU consists of five foundational classes of operations: program flow, logic, bit shifts, and arithmetic. Each class is used to execute varying numerical operations depending on the data input. Within each class, there are numerous operations that are available to apply to the data input. For example, in the logic class, we have “NOT”, “AND”, “XOR”, and “OR.” These logical statements can be used universally to implement a circuit using minimal connection pathways for any given operation. The output of each operation will eventually be stored in a separate register, ultimately concluding the execution by resulting in a singular output.

The bank of internal memory contains 64 words of internal memory with parity-based error correction code multiplicity of three. Assuming full-word access, we have implemented the following operations: MRR Rd Ro (load register to register), LDC R C (load constant), LDD Rd Ro (load direct), and STD Rd Ro (store direct). Execution units from the ALU generally require access to data and operands, to which the bank of internal memory can provide swift access to intermediate results needed by the execution units. Thus, this design method can improve the overall performance of our UrCPU by remaining readily accessible with any given operation. Additionally, the internal memory will store data that is required by the control unit to execute instructions and instill effective coordination.

Finally, the control unit stands as an integral component of any microprocessor. It is designed to direct the operation of the entire processor based on the instructions it receives from the internal memory. Once the instructions are received and interpreted, the control unit will generate the necessary control signals to bring together the outputs from the other remaining components of the UrCPU. To accomplish this, the control unit operates within five primary instruction stages: instruction fetch, instruction decode, instruction execute, memory access, and write back. When the information is in fetch, the data is stored into an internal buffer, taken from memory. Then, all data paths are set, and the instruction will execute when all data is loaded in. Only when the data is fully loaded is, will the execution begin. Finally, the necessary register and data path will be activated to write the final result to its respective destination.

III. INSTRUCTION SET ARCHITECTURE DESIGN

Despite our 20-bit microprocessor having notable sizing constraints, our instruction set architecture design seeks to maximize the number of operations the UrCPU is able to compute. Each operation is designed in a straightforward manner that minimizes the need for additional lines of code, and thus further logical errors. This method is utilized even in the most fundamental logical operations contained within the ALU among many others. Consider the brief Verilog program for the logical conjunction of an “AND” circuit. Observe both the initiation file, and the testbench file:

```
module AND(

    input [19:0] a,
    input [19:0] b,
    output reg [19:0] c

);

    always @* begin
    c = a & b;
    end

endmodule

module AND_tb;

reg [19:0] a;
reg [19:0] b;
wire [19:0] c;

AND and_gate(.a(a), .b(b), .c(c));

initial
begin

    $dumpfile("AND_tb.vcd");
    $dumpvars(1, and_gate);
    $monitor(a, b, c);
    a = 1'b0;
    b = 1'b0;

    #5
    a = 1'b0;
    b = 1'b1;

    #5
    a = 1'b1;
    b = 1'b0;

    #5
    a = 1'b1;
    b = 1'b1;

end

endmodule
```

Notice the brevity of this particular logical operation. In addition to several other seemingly straightforward and minimally designed operations (OR, NOT, XOR), the UrCPU can be described as being composed of undemanding logical statements which can be utilized together in a manner that allows for the capability of more intricate operations, without explicitly having to code in additional complexity. In some sense, these simplistic operations pose as the foundation of the microprocessor’s elaborate computing abilities. Consider the various notable laws of logic, including DeMorgan’s Law, the Distributive Law, and the Associative Law. Instead of incorporating the intricacy of each law, we can instead focus on developing the basic components that make the arrangement of each law possible, and thus outputting the same result with less memory usage.

Consequently, the limited number of bits presented within our opcode—though constricting the number of unique operations—encourages a more specialized design approach to the UrCPU. Thus, we are granted the opportunity to design instructions that are more efficient at tackling specific applications, instead of instructions that cover a greater number of operations, but less complexity. Given the specifications of this project, we need not develop the microprocessor to that extent. However, depending on the task assigned to the UrCPU, this method of design stands to be more effective at handling the task as opposed to the alternative method.

IV. Implementation

The advantages of the UrCPU's design include efficiency, and reduced complexity. Due to the primary design strategy being code efficiency, we have minimized the number of instructions necessary to perform any given task. Through the use of several complier optimizations, like loops and register allocation, we have maximized the performance of our microprocessor by reducing the amount of memory access patterns. Additionally, reduced code complexity minimizes the probability of running into logical errors, given the assumed direct relationship between greater code density, and probability of encountering error, especially in the development of more complex instructions.

However, the trade-offs associated with this implementation strategy inhibit the capacity of operations our microprocessor is capable of performing. Furthermore, it can also be argued that the amount of time consumed by developing a few specialized instructions can be equivalent, if not greater, than the time consumed by opting for a generalized/general purpose design strategy. Specialized design methods call for great attention-to-detail and properly organized instructions. Thus, a challenge that was heavily encountered throughout the design process of the UrCPU was maintaining orderly programs while working with a multitude of files containing various components of the microprocessor. Arguably the greatest difficulty of all was amassing each and every file and delivering it under a singular top-level module, while keeping each individual module properly documented, and making sure not to confuse certain input declarations that have similar names with one another.

A possible solution to this challenge is to declare each input and output variable within every file in a unique manner. In other words, if one or more components of the UrCPU call for similar inputs, but not the exact same input value, then the programmer must hold the responsibility to carefully denote each variable in respect to the explicit end goal of the instruction. One could accomplish this through the use of capitalized letters, or abbreviated terms that relate to the specific instruction. Thus, it would also be beneficial to incrementally include each portion of a particular component, instead of all at one time. For example, when the logical and bit-sift portion of the ALU is complete, it would be wise to first make these individual additions to the final file, rather than add every specification of the ALU all at once. This way, the final connection of all components becomes more manageable than before.

V. Discussion and Conclusions

Throughout the design and implementation of the UrCPU, several notable achievements were encountered. Due to the number of specifications that a microprocessor commands, heavy code documentation has played a great deal in keeping our files organized and easily readable to look back on. Code documentation is an essential element to any elaborate project and allows third parties to easily understand one's logic despite knowing little about the academic concepts behind it. Documentations also makes it easier for the programmer to look back on portions of code where the logic can be further extended at a later date, without having to glance through each data path connection.

Further extensions to this project could include making more operations available to the microprocessor. Instead of being limited to a design focused on specialization, more time dedicated to this project can be spent developing logical/arithmetic operations that the current microprocessor lacks. Thus, this will expand the functionality of the UrCPU by enhancing the versatility through adding a wider range of operations. However, it can also be noted that the trade-off being specialization and general-purpose design can never be perfectly addressed; there will always be a decision to be determined about which aspects to include or sacrifice.

The potential to scale for the UrCPU holds immense opportunity. Scaling of the microprocessor can refer to many instruction design methods such as performance scaling, application scaling, or market scaling. Performance scaling allows for the execution of a greater number of tasks for the microprocessor to handle at a given time. Application scaling refers to making the microprocessor more versatile in terms of adaptability with various numerical inputs. The more cases the microprocessor can handle, the greater its robustness. Finally, market scaling refers to making the microprocessor perform niche needs to the "consumer" of the final product. It could be a possibility that one needs the microprocessor only to handle a few specific instructions effectively, instead of a wide range of cases which they may not need to delve into in the first place.

VI. References

[1] P. Deepchandra, "What is Scaling of MOSFET, Scaling Factor, and Types of Scaling,"Hackatronic: What is Scaling of MOSFET, Scaling Factor, and Types of Scaling (hackatronic.com) May 09, 2021.

[2] T. Vincent, "DRY (Don't Repeat Yourself) — Avoiding Redundancy In Software Programming," Medium: DRY (Don't Repeat Yourself) — Avoiding Redundancy In Software Programming | by Vincent T. | 0xCODE | Medium November 23, 2021.