Implementation of DLX Architecture Modules for Non-Pipeline Execution using Verilog

Carlos Cruz Garza
Department of Electrical and Computer Engineering
The University of Texas at San Antonio
San Antonio, TX, USA
carlos.cruzgarza@my.utsa.edu

***Abstract*- Hennessy and Patterson originally designed the DLX CPU as a means to teach computer architecture principles using a 32-bit RISC microprocessor [1]. This processor follows the (0,3)-GPR architecture as well as the 5-stage pipeline execution cycle.[4] The ALU unit is a crucial component of any computer architecture, and it is fundamental for executing arithmetic and logical calculations. In the DLX, it also allows the flow of data between different registers. The objective of this project is to implement the Arithmetic and Logic Unit (ALU), control unit, and other modules of this fundamental architecture and its corresponding signals in Verilog. The ALU will include arithmetic operations such as ADD, MULT, SUB, and logical operations such as AND, OR, XOR, and shift operations. The control unit will be hard-wired as a Finite State Machine with the 5-stage execution cycle and its corresponding signals [2].**

Keywords—DLX, Computer Architecture, RISC, Verilog

I. Introduction

DLX architecture provides a deep fundamental understanding of RISC architecture. It follows a clean 5-stage pipeline as well as (0,3)-GPR addressing format [1]. The arithmetic and logic unit (ALU) is a fundamental part of the DLX. It is tasked with performing arithmetic and logical operations as well as transferring data between registers. The goal of this project is to implement the DLX ALU unit and test it in a testbench environment. After the ALU, other modules such as the control unit, register file, and other modules will be implemented to create a fully sequential DLX processor using Verilog.

II. Methodology

The DLX Architecture was explored extensively throughout the Computer Architecture class at the University of Texas at San Antonio. Supplemental studying was obtained using Hennessy and Patterson's *Computer Architecture: A Quantitative Approach* [1]. The goal of this project is to implement different functionalities of the CPU's architecture that are simplified down to work in a non-pipelined execution flow. There will be no memory controller implementation due to the nature of complexity of achieving such a module. Figure 1 shows all the modules required for implementation and how they interact with one another to make the DLX execute.

A. Overview

The DLX Architecture was explored extensively throughout the Computer Architecture class at the University of Texas at San Antonio. Supplemental studying was obtained using Hennessy and Patterson's *Computer Architecture: A Quantitative Approach* [1]. The goal of this project is to implement different functionalities of the CPU's architecture that are simplified down to work in a non-pipelined execution flow. There will be no memory controller implementation due to the nature of complexity of achieving such a module. Figure 1 shows all the modules required for implementation and how they interact with one another to make the DLX execute.

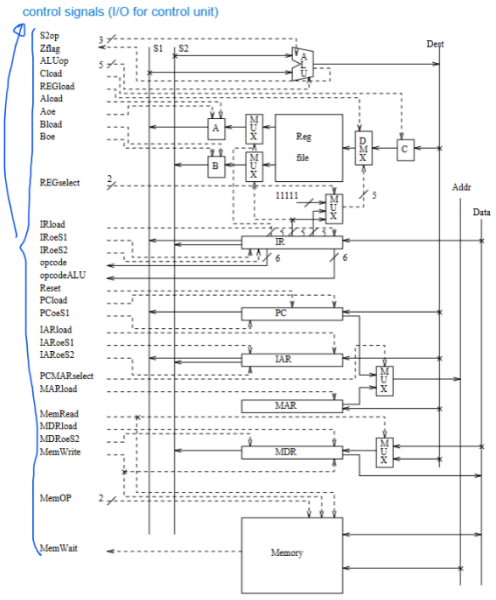

Figure 1: Data Flow of Modules in DLX Architecture [5]

B. System Modules

1) ALU: The modularization of the ALU in this implementation will be combinational to execute ALU operations in "one clock cycle". It will take as input the s1 and s2 buses, as well as the opcode for the ALU and the s2 bus. The output of this combinational circuit will be the output of the ALU, depending on the desired operation and the zflag, since it is a crucial signal for determining what to do with the output. Table 2 shows the synthesized combinational circuit required to perform all possible opcode operations in the DLX architecture

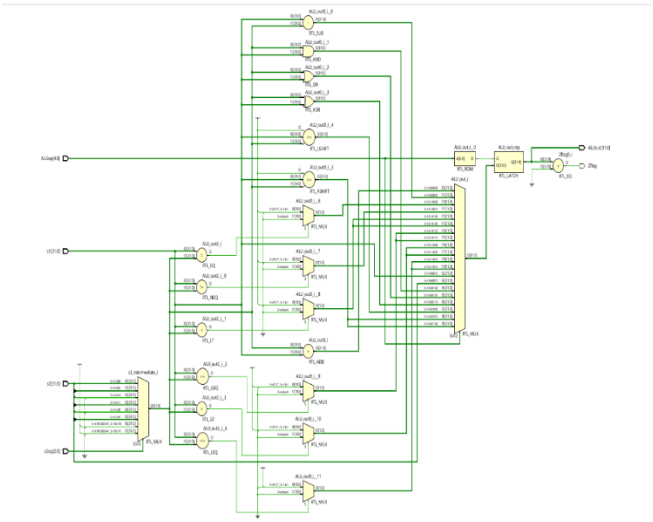

Figure 2: ALU Combinational Implementation

2) Control Unit: The control unit is designed as a Finite State machine following the states of the 5-stage execution flow that the DLX processor adapts [2]. Each stage is defined as follows: IF, ID, EX, MEM, WB. The end goal of this project was to have a full-fledged control unit that can decode and execute all the instructions created for the DLX by Patterson and Henessy [1]. The implementation of this control unit is hard-wired to execute one instruction, the addition of a register with an immediate value. This does not mean other instructions can't be implemented; however, for verification, this has been the only instruction that has been implemented

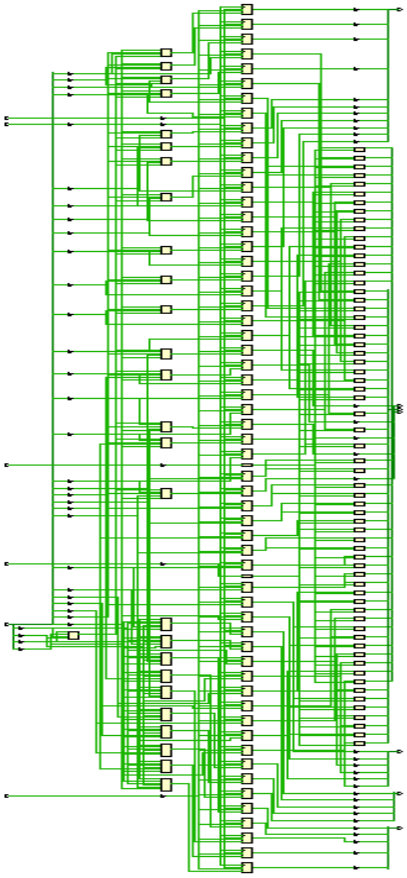

Figure 3: Control Unit Schematic

3) Register File: The register file contains all the registers and the logic to load data onto registers from the output of the C buffer DMUX. I also implemented circuitry logic to output the input of the A and B buffers, which connect to the S1 and S2 buses, respectively. The register file acts as temporary high-speed memory contained inside the CPU for faster memory access.

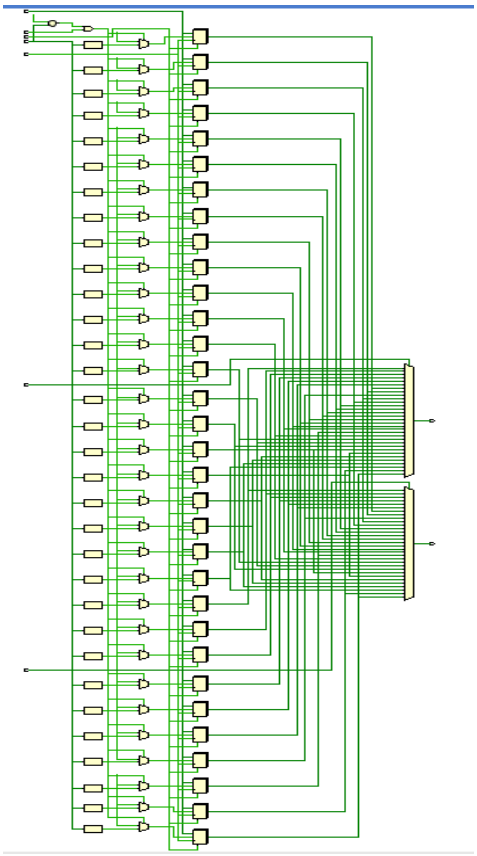

Figure 4: Register File

4) Program Counter: The program counter is responsible for storing the next instruction to be executed in the execution flow. Typically, it stores the instruction following the one that is being decoded, but it can jump to somewhere else in memory due to branch and jump instructions. It is a sequential circuit, so it takes in a clock signal as well as a reset signal. PCload is a control signal that, when enabled, will store whatever is being stored in the address bus into the PC register. Whenever PCoeS1 is enabled, the PC register will go from the tri-state buffer into the s1_bus, which will go to the ALU for the following execution of instructions.

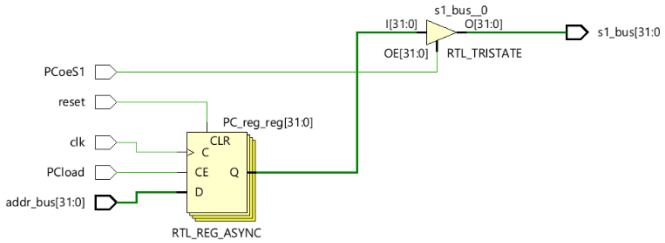

Figure 5: Program Counter Register

5) Memory Data Register: The memory data register stores the data that is about to be either loaded onto a register or written into memory. It takes the typical clock and reset signals as input. It also takes in as input an MDRload signal, which writes the MUX output onto its register. It outputs onto two different buses, the S2 bus when loading into a register and the data bus whenever a store instruction is called. These buses are controlled by tri-state buffers and are sent to the bus whenever their respective signal is called.

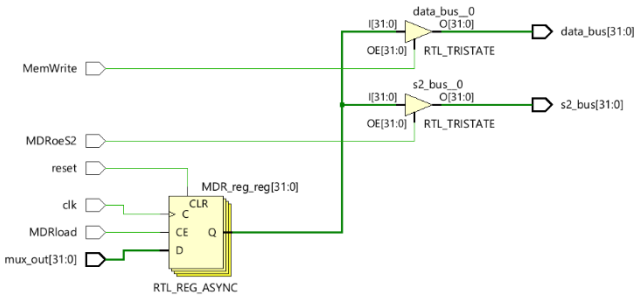

Figure 6: MDR Register

6) Memory Address Register: The memory address register stores the location of the instruction or data that is going to be loaded or stores by the CPU. It takes in the clock and a reset signal as input, as well as the destination bus and a MARload control signal. It always outputs it's internal register onto a wire that will feed into a MUX for the PCMARSelect signal to decide what to send to the memory controller.

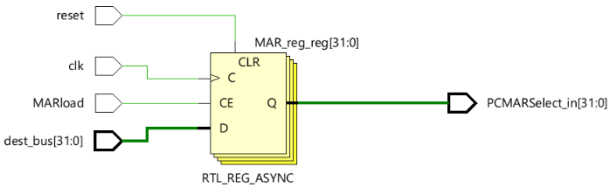

Figure 7: MAR Register

7) Instruction Address Register: The IAR acts as a temporary register that holds the value of the Program Counter. It is beneficial because the program counter is stored in the register file, so to avoid having to read from the register file module, the IAR also holds the address of the current instruction that is being executed.

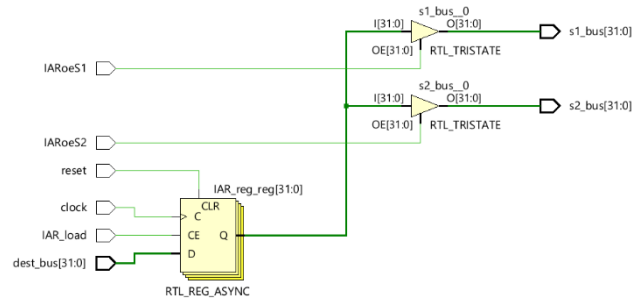

Figure 8: IAR Register

8) Instruction Register: The instruction register holds the current instruction that is being executed. From the instruction register, register values for the destination and sources are pulled. Opcode is pulled to determine the type of instruction, which is sent to the control unit to determine the execution path to take, since it is an FSM. The IR takes in the data bus as an input and gets loaded whenever the IRload

signal is enabled. Once through, rs1, rs2, rd, and opcode are extracted in order to decode the instruction in the second stage of the FSM.

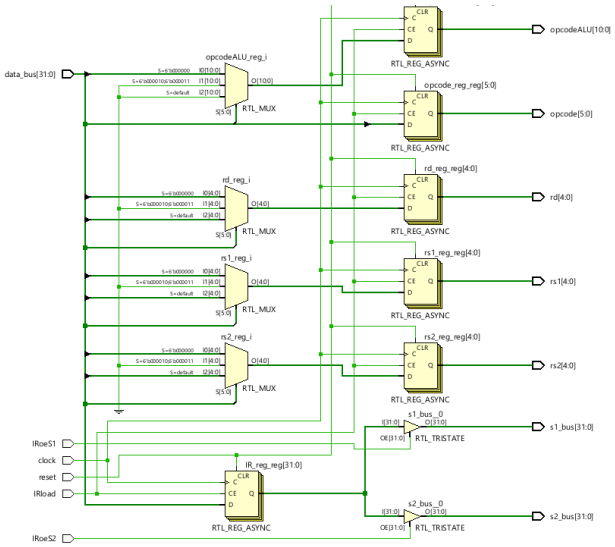

Figure 9: IR Circuit

C. Tools

All modules and testbenches were written using Verilog-2001 and in Xilinx Vivado 2023.1. All simulations, schematics, and timing analysis were performed inside Vivado's elaborate design suite.

D. Testing

1) Overview: Testbenches will be created to test the most crucial DLX modules for verification and timing analysis. The ALU will be tested by feeding it different opcodes and input values to evaluate the output. The instruction register will be tested by feeding it an instruction and making sure the instruction is being properly decoded. Lastly, the control unit will be tested by making sure it completes the correct execution cycle of an entire example (e.g., ADD). The other modules are registers with no outside functionality other than holding values or outputting signals, so their functionality can be tested for verification, but it will be omitted in this report.

2) ALU: Figures 10 & 11 show the timing analysis and output of test cases on a TCL console. These figures are crucial in verifying the desired outputs of the ALU unit. In the first 5ns, the first operation is being executed. The waveform clearly shows the addition of 0x0a and 0x14 to produce 0x1e. The Zflag wave remains low due to the output being non-zero. In the second operation, there is subtraction going on of 0x0 minus 0x0. What's interesting about this operation is that the Z flag properly went to one because the output of the ALU was zero, which raised the flag. Lastly, the last instruction compared the two values in the s1 and s2 buses and had an output of 1, meaning the values in the s1 and s2 buses are identical. With this analysis, the functionality and accuracy of the ALU modules implemented for the DLX architecture is ideal for different types of instruction executions.

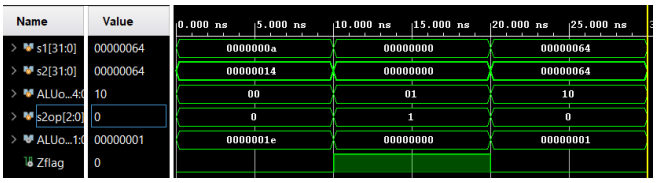

Figure 10: Timing Analysis of ALU

ALU Output	Signals			
	S1	S2	ALUout	Zflag
ADD	10	20	30	0
SUB	0	0	0	1
SEQ	10	10	1	0

Figure 11: TCL Console Output

3) Control Unit: Figures 12 & 13 show the timing analysis as well as the output per state of the control unit implementation of the DLX architecture. The controller follows the DLX 5-stage execution cycle[2], and in this implementation, they were hard-wired into a finite state machine where each state corresponds to an individual stage in the execution cycle. The waveform shows the proper activation of signals as the state machine progresses every clock cycle through the pipeline. The limitation of this implementation is the hard-wiring aspect. Since adding a different execution stage for every single instruction that the DLX can perform, I implemented the execution of just one instruction and tested the cycle of that instruction instead. In the Instruction Fetch (IF) stage, the program counter is sent out to the memory bus to retrieve the instruction in memory, which is then loaded onto the instruction register. In the instruction decode phase, the corresponding values are loaded into the A and B registers. Lastly, in this stage, the program counter is incremented to point towards the next instruction. The EX, MEM, and WB stages are unique to the ADDI instruction that is being implemented in the test of the control unit. Figure 12 shows the state of each signal per state machine state.

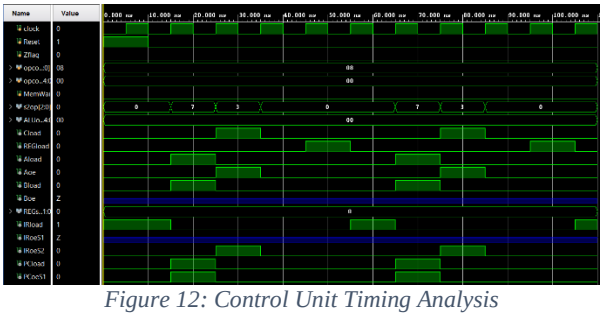

Figure 12: Control Unit Timing Analysis

Signal Analysis	Clock Edge				
	IF	IF	EX	MEM	WB
ALUop	00000	00000	00000	00000	00000
S2op	000	111	011	000	000
Aload	0	1	0	0	0
Aoe	0	0	1	0	0
Bload	0	1	0	0	0

Boe	Z	Z	Z	Z	z
Cload	0	0	1	0	0
REGselect	00	00	00	00	00
REGload	0	0	0	0	1
IRload	1	0	0	0	0
IRoeS1	Z	Z	Z	Z	Z
IRoeS2	0	0	1	0	0
PCload	0	1	0	0	0
PCoeS1	0	1	0	0	0
PCMARsel	0	0	0	0	0
MemRead	1	0	0	0	0
MemWrite	Z	Z	Z	Z	Z
MDRload	Z	Z	Z	Z	Z
MDRoeS2	Z	Z	Z	Z	Z
MemOP	00	00	00	00	00

Figure 13: Control Unit TCL Console Output

4) Instruction Register: Figures 14 & 15 show the timing analysis and instruction decoding performed by the DLX_IR modules created using Verilog. The purpose of this register was to store the current instruction and to perform the instruction decoding once it received the control signals from the control unit. The waveform shows the behavior of the most crucial signals, such as IRload, opcode, rs1, rs2, rd, and opcodeALU. The circuit first differentiates between what type of instruction is contained (I-type, R-type, J-type) by examining the opcode signal. If the opcode remains at zero, we know it will be an R-type. If the opcode maps to a jump instruction, it recognizes it as J-type. Otherwise, it will perform I-type instructions as the default state. A testbench was created and displayed on a tcl console to verify that the instruction is being executed correctly. The first instruction (0x00221801) is immediately recognized as an R-type instruction since it has an opcode of 0. After recognition, the rs1, rs2, rd, and opcodeALU registers are properly filled with the correct part of the instruction for execution. The second instruction, (20A60008), is recognized as an I-type because it doesn't pertain to a jump instruction, and the opcode is not 0. This again allows for proper handling of rs1, rs2, and rd registers for proper instruction execution. For jump instructions, the signals are left as empty once the opcode is recognized to be a jump instruction. The waveform also confirms that IroeS2 and IRoeS1 properly load the output of the IR onto the s1 and s2 buses.

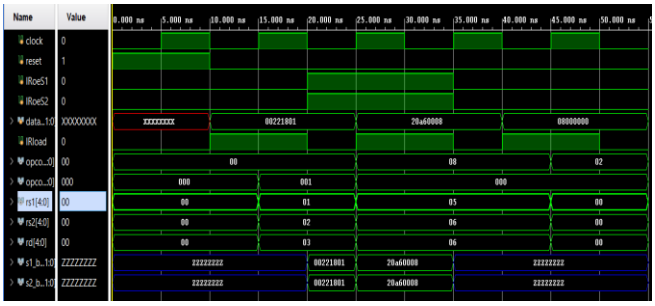

Figure 14: Instruction Register Timing Analysis

Signal Analysis	Signal				
	opcode	Rs1	Rs2	rd	opcodeALU
R-type	000000	1	2	3	0x01
I-type	001000	5	6	6	0x00
J-type	000010	0	0	0	0x00

Figure 15: Instruction Register TCL Console Output

III. Results

A. DLX Modules Results

As shown in Section II, all critical DLX processor modules were successfully implemented and tested using Verilog and Xilinx Vivado. Implementation of the ALU module as a combinational circuit provided speed, as it was able to execute one operation every clock cycle. Results from the timing analysis and tcl console showed proper handling of different test case operations, where the output successfully matched the theoretical output needed. The Control Unit, treated as a finite state machine, was able to successfully take in and send out the proper control signals to execute the ADDI instruction in a 5-stage cycle fashion [2]. The states included: Instruction Fetch, Instruction Decode, Execution, Memory Access, and Write Back. These are the classical stages expected from the DLX architecture. Instruction decoding was successfully implemented and tested through the implementation of the Instruction Register. Depending on the opcode, the instruction type was able to properly be decoded and, in turn, sent out the parts of the instruction to their respective destination.

B. Integration of all DLX Modules

The plan for this project was to fully integrate the modules of the DLX architecture to have a fully functioning 5-stage non-pipelined DLX processor [2]. Although the result was not achieved. The modules created show crucial understanding of the DLX architecture and have the ability to be implemented into the full-flown architecture.

IV. Conclusion

This project successfully implements the building blocks required to execute a simplified, non-pipelined DLX processor architecture using Verilog. Key components were implemented, such as the Arithmetic & Logic Unit (ALU), the Control Unit (CU), the Instruction Register (IR), the Program Counter (PC), the Register File (RF), and other registers to properly be able to show the data flow and the 5-stage execution cycle of the 32-bit DLX microprocessor. Although the full, large-scale system integration was not achieved, the key components were successfully modularized and tested to bring them all together in the future. This approach aligns with other approaches to implementing the DLX architecture, such as the work by Fagin and Chintrakulchai, who demonstrated the importance of hardware-based validation and testing of educational material such as the DLX processor model [3].

V. References

[1] Patterson, D. A., & Hennessy, J. L. (1996). *Computer Architecture: A quantitative approach*. Kaufmann.

[2] Dilakanont, N. (n.d.). *MLDesign Technologies, inc.*. MLDesign Technologies, Inc - Palo Alto USA/CA.

https://www.mldesigner.com/mldesigner/examples/computer-architecture/dlx-processor/

[3] Fagin, B., & Chintrakulchai, P. (1992). *Prototyping the DLX microprocessor | IEEE conference publication | IEEE Xplore*. IEEE Xplore. https://www.mldesigner.com/mldesigner/examples/computer-architecture/dlx-processor/

[4] "DLX." *Wikipedia*, Wikimedia Foundation, 2 Apr. 2025, en.wikipedia.org/wiki/DLX

[5] Lin, Wei-Ming, "DLX_CPU", EE_5123 *Computer Architecture*, The University of Texas at San Antonio, Received in Spring 2025, Course handout.

[6] "Instruction Register." *GeeksforGeeks*, GeeksforGeeks, 27 Feb. 2024, www.geeksforgeeks.org/instrcution-register

[7] "What is a Register in CPU." *MS.Cods*, ms.codes/en-gb/blogs/computer-hardware/what-is-a-register-in-cpu. Accessed 8 May 2025.