**EE533: Network Processor Design & Programming**

**Lab #6: Pipeline Datapath on NetFPGA**

Instructor: Prof. Young Cho, PhD

Team Number:**#3**

***Project Partners:***

*Member #1:* **Sarthak Jain**

*Member #2:* **Archit Sethi**

*Member #3:* **Justin Santos**

*Designed, Created, and Submitted by Team #3*

***University of Southern California***

***Los Angeles, CA 90007***

Team #4 GitHub Repository Link: [*Team#3 Link*](https://github.com/EE533-TEAM3/EE533_LAB_PROJECTS/tree/main/LAB6)

Part-1: ALU Design & Implementation

1. 64-bit ALU with functions implemented Adder, Subtractor, Bitwise AND, Bitwise OR, Bitwise XNOR, Comparator, Logical Shifter

The Arithmetic Logic Unit (ALU) is like the brain of a processor—it performs all the number crunching and logical decisions. Think of it as a powerful calculator that not only adds and subtracts but also compares numbers, processes bitwise operations, and even detects patterns within data.

The ALU we’re looking at is designed to work with 64-bit numbers, meaning it can handle large data operations efficiently. It supports a variety of functions, from basic math to logical shifts and even pattern matching, making it ideal for applications like networking, encryption, or high-speed computing.

Let us explain the big picture: what does this ALU do?

This ALU takes in:

Two input numbers (alu\_in1\_i and alu\_in2\_i)

A control signal (aluctrl\_i) that tells it what operation to perform

A carry-in bit (cin) used in addition and subtraction

It produces:

A result (alu\_o) based on the operation selected

A carry-out bit (cout\_o) for arithmetic operations

Comparison flags (alu\_a\_gt\_b, alu\_a\_lt\_b, alu\_a\_eq\_b) to tell us how the two numbers relate

Now, let us step-by-step the ALU.

*Step-1: Understanding the Operations*

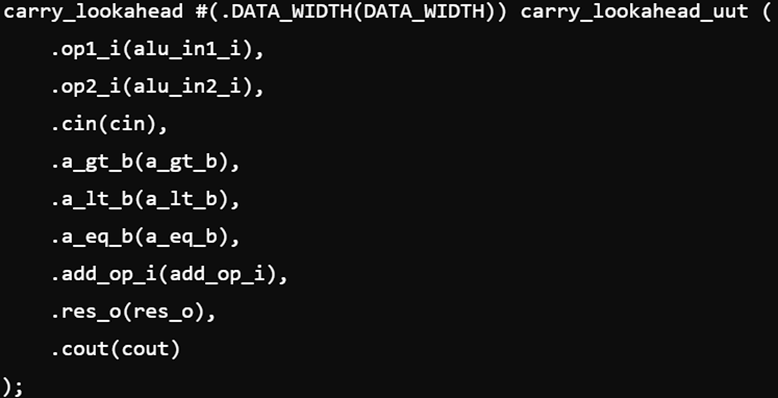
Every ALU operation is triggered by the control signal (aluctrl\_i), which acts like a command selector. Here’s what the ALU can do:

|  |  |  |
| --- | --- | --- |
| **Control Code** | **Operation** | **What It Does** |
| 1 | **Add** | Adds two numbers |
| 2 | **Subtract** | Subtracts second number from the first |
| 3 | **Bitwise AND** | Compares bits and keeps only the 1s |
| 4 | **Bitwise OR** | Compares bits and keeps 1s from either input |
| 5 | **Bitwise XNOR** | Inverts XOR—keeps bits that match |
| 6 | **Compare** | Checks if first number is >, <, or = second number |
| 7 | **Logical Shift Left** | Moves bits left, filling empty spots with 0s |
| 8 | **Logical Shift Right** | Moves bits right, filling empty spots with 0s |
| 9 | **Pattern Matching** | Checks if one number is a substring of another |

If none of the commands are given, the ALU defaults to returning ZERO.

*Step-2: Arithmetic Operations (Addition and Subtraction)*

Instead of using a basic adder, this ALU employs a Carry Lookahead Adder (CLA), which is a high-speed way to add numbers by predicting carry bits in advance.



If we’re adding, it just sums alu\_in1\_i and alu\_in2\_i normally.

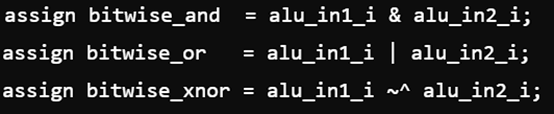
If we’re subtracting, it subtracts alu\_in2\_i from alu\_in1\_i by treating it like addition with a negative number.

This results in:

res\_o: The result of the operation.

cout: Carry-out, useful for multi-step calculations.

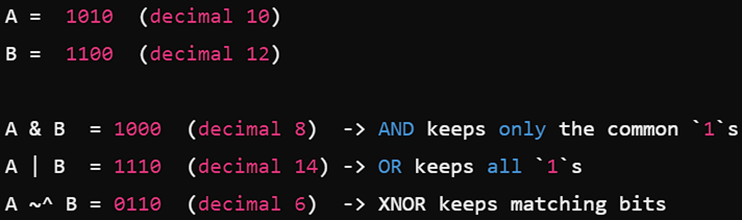
*Step-3: Logical Operations (Bitwise AND, OR, XNOR)*

**

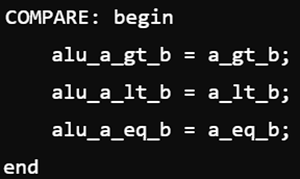
AND (&): Keeps only the bits that are 1 in both inputs.

OR (|): Keeps 1s from either input.

XNOR (~^): Keeps 1s where both bits match (opposite of XOR)



*Step-4: Comparison Operations (Greater than, Less than, Equal To)*

**

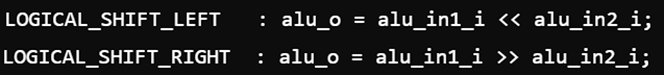
If alu\_in1\_i > alu\_in2\_i, it sets alu\_a\_gt\_b = 1.

If alu\_in1\_i < alu\_in2\_i, it sets alu\_a\_lt\_b = 1.

If they’re equal, alu\_a\_eq\_b = 1.

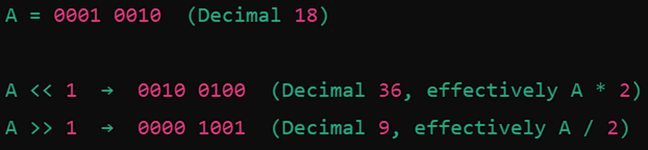
*Step-5: Shifting Operations*

Shifting is useful for multiplying/dividing numbers by powers of 2 or for manipulating bit patterns.



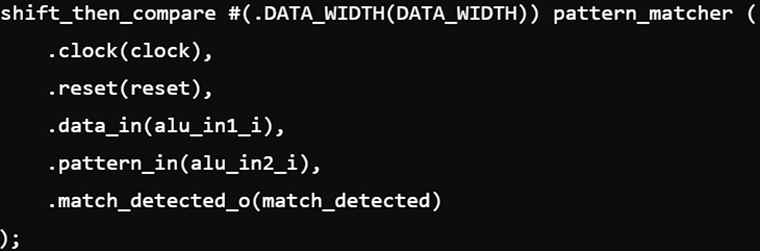
Left Shift (<<): Moves bits left, adding 0s on the right.

Right Shift (>>): Moves bits right, adding 0s on the left.



*Step-6: Pattern Matching (Shift-Then-Compare)*

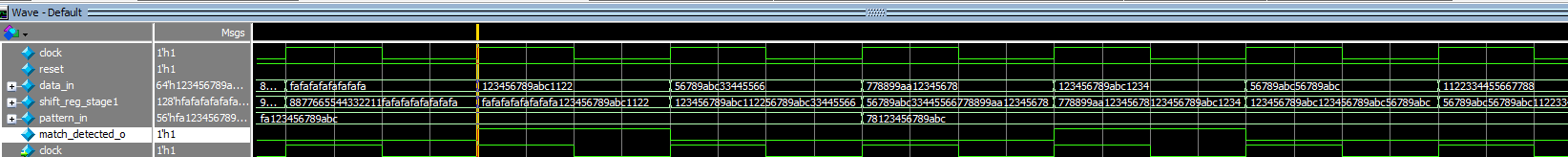
One of the cool features of this ALU is its pattern matching ability. This is useful for searching through network packets, text, or binary data.



alu\_in1\_i is treated as a long sequence of bits (like a paragraph).

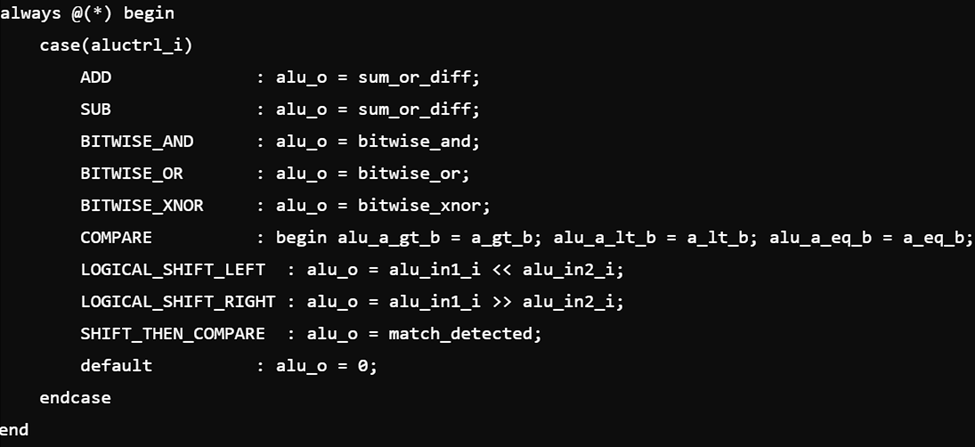
alu\_in2\_i is a smaller bit pattern (like a word).

The ALU checks if alu\_in2\_i appears inside alu\_in1\_i.

If the pattern is found, match\_detected becomes 1.

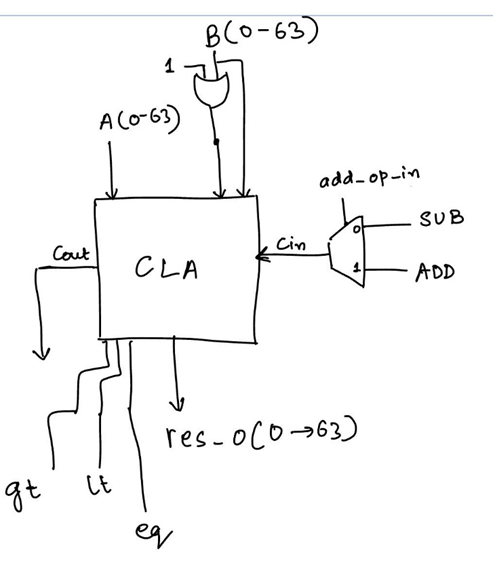
*Step-7: Execution Flow*

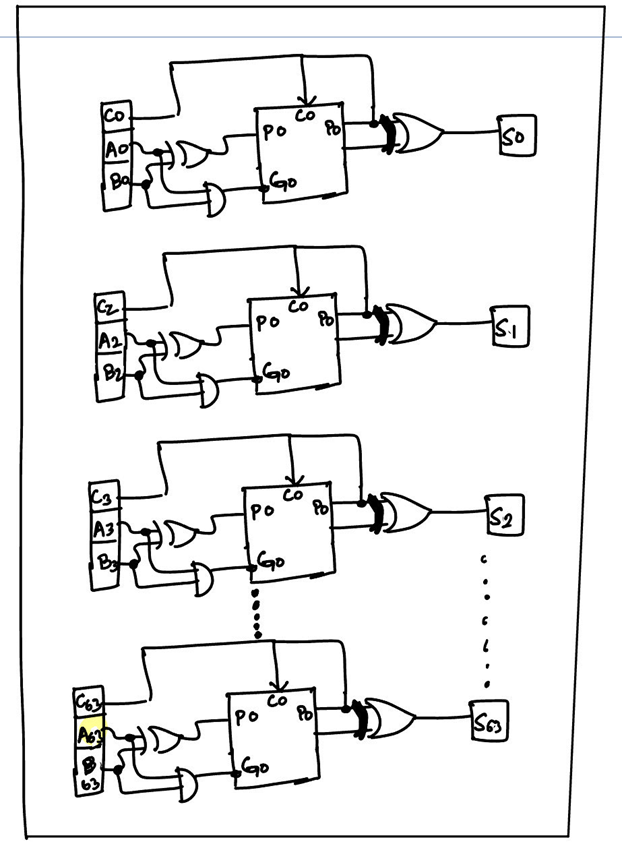
The ALU works inside an always block, which ensures that every time aluctrl\_i changes, the ALU picks the correct operation.

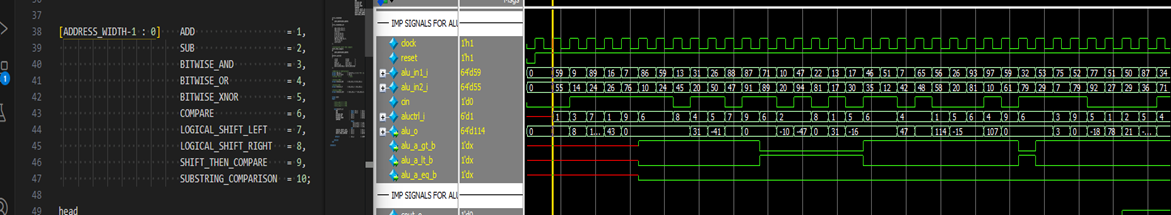


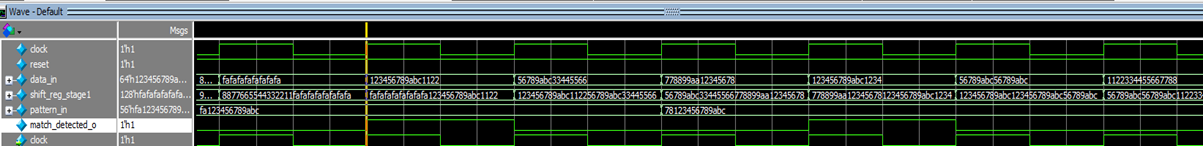
* It **checks the control signal (aluctrl\_i)** and selects the corresponding operation.
* If no valid operation is found, it **defaults to returning zero**.

1. Block Diagram of 64-bit ALU





1. Simulations Waveform of 64-bit ALU



Part-2: Register File & Memory Design

1. Register File Implementation

A Register File is a small, high-speed storage unit inside a processor that holds frequently used data. It provides quick access to operands for arithmetic and logical operations, reducing the need to fetch data from slower main memory.

This Verilog implementation defines a 64-bit, 64-register file that supports:

Reading two registers simultaneously

Writing to a register on the negative clock edge

Configurable output register behavior

2. Internal Register Storage

reg [DATA\_WIDTH-1 : 0] register [DATA\_WIDTH-1 : 0];

This line declares an array of registers. Since DATA\_WIDTH = 64, this means:

• The register file consists of 64 registers, each 64 bits wide.

3. Writing to Registers

always @(negedge clock)

begin

if(reg\_write)

register[waddr] <= wdata;

end

• Write happens on the negative edge of the clock (``).

• If reg\_write is enabled, the data wdata is stored in register waddr.

4. Reading from Registers

If REGISTERED\_OUTPUT = 1 (Output Stored in Flip-Flops)

always @(posedge clock)

begin

data\_reg1 <= (r1addr != 0) ? register[r1addr] : 'd0;

data\_reg2 <= (r1addr != 0) ? register[r2addr] : 'd0;

end

• Reads are stored in flip-flops on the positive edge of the clock (posedge clock).

• This adds a cycle delay to the output, making it useful for pipelined architectures.

If REGISTERED\_OUTPUT = 0 (Combinational Read)

assign r1data\_o = (r1addr != 0) ? register[r1addr] : 'd0;

assign r2data\_o = (r2addr != 0) ? register[r2addr] : 'd0;

•The output is directly assigned from the register file.

•Faster, but may cause glitches if registers change mid-cycle.

5. Why This Design Is Important

Dual Read, Single Write

• Supports reading two registers at once, which is essential for executing arithmetic/logical operations in parallel.

Configurable Output

• REGISTERED\_OUTPUT = 1: Smoother data flow, less glitching.

• REGISTERED\_OUTPUT = 0: Faster response, no latency.

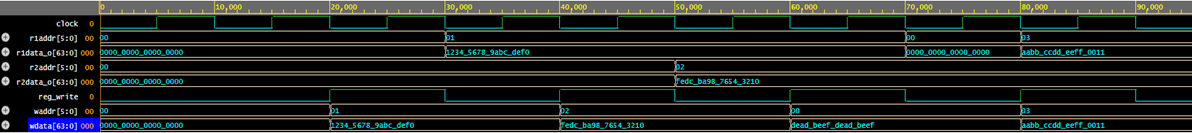
Efficient Storage

• Uses a 64-entry array, keeping access times low.

Synchronous Write, Asynchronous Read

• Writes happen on the clock edge to avoid race conditions.

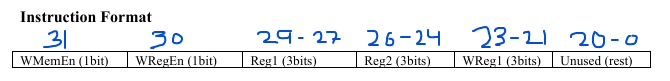
• Reads are instantaneous for quick access.

1. Simulation Waveform of Register File

Part-3: Pipelined Datapath Implementation

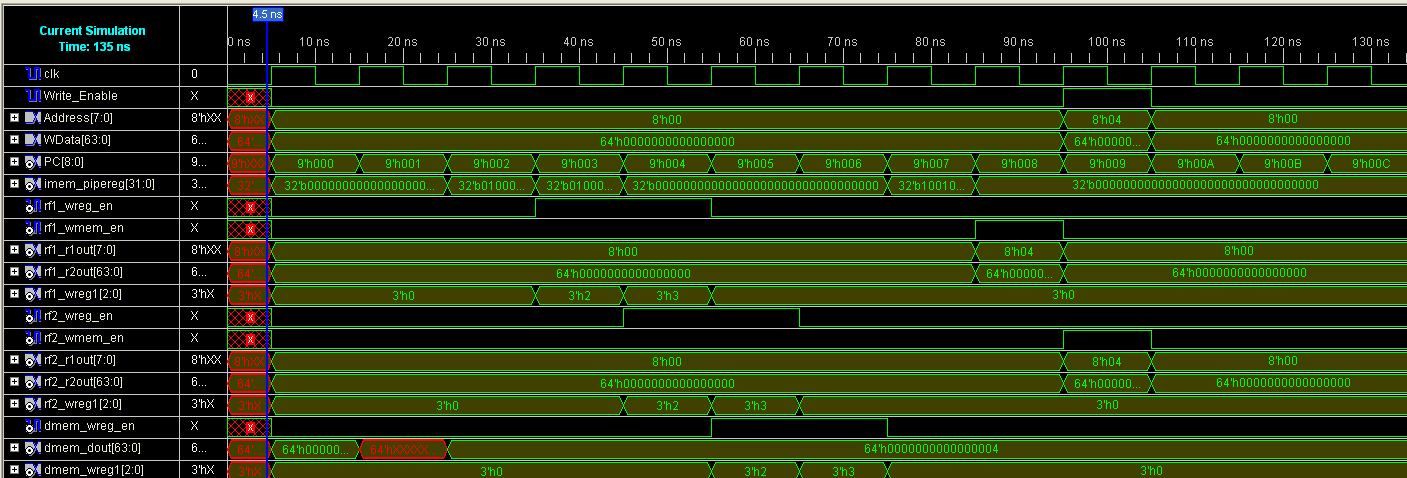
1. Skeleton Design of 5-stage Pipeline Datapath Unit (DPU)

Below is the way we partition the instructions, indicating the bits that will be used for enable signals and register file inputs.



1. Simulation Waveform of Pipelined Datapath

This snippet shows the data pipeline going through all the instructions in the Instruction Memory and all the bits of imem\_pipereg propagating to the different inputs of the modules and subsequent pipeline registers. We see that our instructions are being propagated through each cycle, and are properly triggering the enable signals as shown with the toggles of our wreg\_en and wmem\_en signals.



This snippet shows that by the end of our “dummy” instructions, we get 0x4 to be stored in Address 4 of our data memory.

