School of Electrical and Computer Sciences
Indian Institute of Technology, Bhubaneswar

LIPSI Processor

Authors: Shishir Ravi Jois
Shrinivas B M
Supervised by: Dr. Srinivas Boppu
Affiliation: IIT Bhubaneshwar
Branch: ECE

Date: 20th April 2025

Project Link: https://github.com/Air-36/Lipsi_Project

Table of Contents

1. Introduction 3
2. The Design 3
3. The Datapath 3
4. The Instruction Set 5
5. Working of Instructions 6
6. Possible Improvements 12
7. Difficulties faced 12
8. Conclusion 12
9. References 13

Table of Figures

Figure 1 Datapath of the Lipsi Processor 4
Figure 2 State Diagram for ALU Register Operation. 6
Figure 3 State Diagram for Store Operation. 7
Figure 4 State Diagram for Branch and Link Operation. 7
Figure 5 State Diagram for Load Indirect Operation. 8
Figure 6 State Diagram for Store Indirect Operation. 8
Figure 7 State Diagram for ALU Immediate Operation. 9
Figure 8. State Diagram for Branch when Zero Operation. 9
Figure 9 State Diagram for Branch when Not Zero Operation. 10
Figure 10 State Diagram for ALU Shift Operation. 10
Figure 11 State Diagram for I/O Operation 11
Figure 12 State Diagram for Exit Operation. 11

1. Introduction

This paper introduces Lipsi, a lightweight microcontroller designed specifically for utility functions within an FPGA. Lipsi is suitable for implementing peripheral devices or state machines as part of a larger system-on-chip. Its primary design objective is to achieve a minimal hardware footprint, using just a single block RAM to store both instructions and data.

Lipsi follows the von Neumann architecture, where both instructions and data share the same memory space and use the same communication pathways. In Lipsi's case, this means a single block RAM is used to store both the program code and the data it processes. This design simplifies the hardware and reduces resource usage, which aligns with Lipsi's goal of being compact and efficient for small control tasks within an FPGA.

Lipsi is a multicycle CPU, i.e., all the instructions are performed over multiple clock cycles most taking around 3 cycles to execute. The execution of the instruction is broken into multiple steps mainly fetching the instruction, decoding and reading operands and executing the operation each of which takes one clock cycle. This approach allows the CPU to reuse functional units (like the ALU or memory) across different instruction steps, leading to a more compact and efficient hardware design.

2. The Design

Lipsi is an 8-bit processor designed as an accumulator-based machine, optimized specifically for FPGA block RAMs. The core design goal is to operate using only a single block RAM for both instructions and data, ensuring minimal resource usage.

The use of very less resources allows multiple processors to be run on a single FPGA. The RAM consists of 4096 bits or 512 bytes. This is equally divided into the data memory and the program memory. The selection between the data and the program memory is done by the MSB of the memory address (0 for data memory and 1 for program memory).

The lower 16 bytes of the data memory is allocated for the register file which can easily be accessed by the processor within one clock cycle. Using a single block RAM for instructions and data means that this memory is time shared between instruction fetch and data read. Therefore, Lipsi is a sequential and not a pipelined architecture.

3. The Datapath

The Lipsi processor includes a program counter (PC), on-chip memory, an arithmetic logic unit (ALU), and an accumulator register (A). In addition to these core components, the design requires one adder and three multiplexers. The instruction decode logic, which controls the multiplexers and ALU operations, is handled by an external controller which isn't the part of the datapath.

The on-chip memory is organized into three distinct sections:

1. Program area
2. Register file
3. Data memory

In typical FPGAs, a single block RAM is 512 bytes. Lipsi utilizes this memory by dividing it into two 8-bit addressable regions—one for **instructions** and one for **data**. Within the data region, the first **16 bytes** serve as a **register file**, while the remaining space is used for **general-purpose data storage**.

The datapath of the Lipsi processor comprises mainly of the following parts:

i. Memory
ii. ALU
iii. Accumulator
iv. Program Counter
v. Multiplexors and Adders

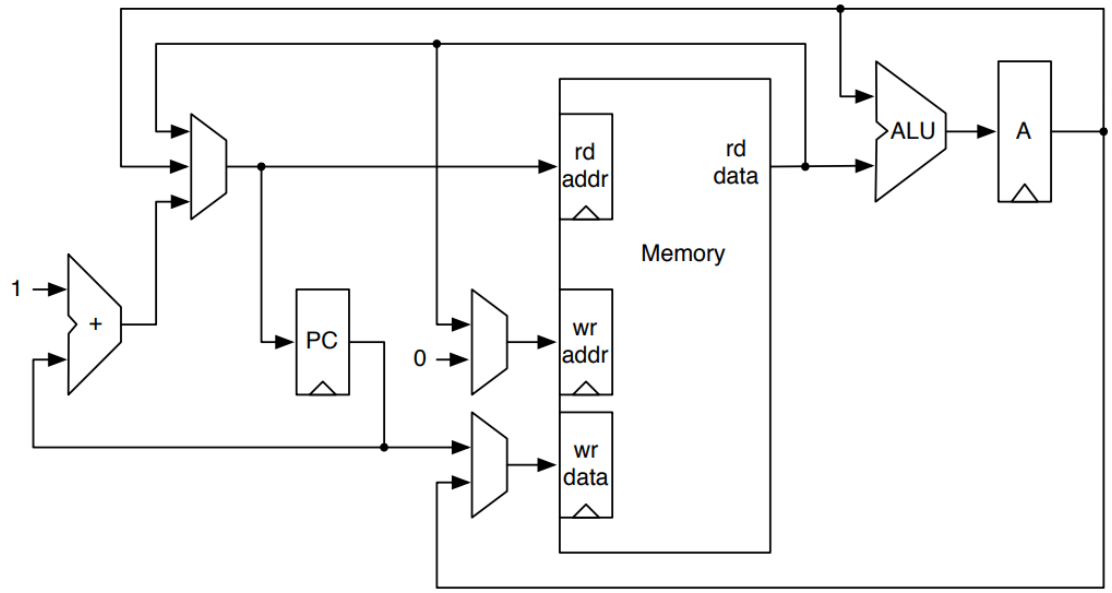

Figure 1 *Datapath of the Lipsi Processor*

4. The Instruction Set

We have included a total of 12 instructions as was the case in the original version of the Lipsi processor. These include the ALU register and immediate operations, store and load operations, branch operations, branch and link operations and the ALU shift operation. Most of the instructions are 1 byte in length except for the ALU immediate, branch and the I/O instructions which are of 2 bytes each.

Table 1 presents the complete set of instructions supported by the Lipsi processor, along with their corresponding encodings. In this context, A denotes the accumulator, which serves as the primary register for arithmetic and logic operations. The symbol f represents a specific ALU function to be performed. PC stands for the program counter, which tracks the address of the next instruction to execute. m[] refers to memory locations, while r indicates a register number, ranging from 0 to 15. n is used to denote an immediate constant value embedded within the instruction. a corresponds to an 8-bit address in memory, and IO signifies an input/output device used for peripheral communication.

Table 1

Encoding	**Instruction**	**Meaning**	**Operation**
0fff rrrr	*f* rx	ALU register	A = A *f* m[r]
1000 rrrr	st rx	store A into register	m[r] = A
1001 rrrr	brl rx	branch and link	m[r] = PC, PC = A
1010 rrrr	ldind (rx)	load indirect	A = m[m[r]]
1011 rrrr	stind (rx)	store indirect	m[m[r]] = A
1100 -fff nnnn nnnn	*fi* n	ALU immediate	A = A *f* n
1101 --00 aaaa aaaa	br	branch	PC = a
1101 --10 aaaa aaaa	brz	branch if A is zero	PC = a
1101 --11 aaaa aaaa	brnz	branch if A is not zero	PC = a
1110 –ff	sh	ALU shift	A = shift(A)
1111 aaaa	io	input and output	IO = A, A = IO
1111 1111	exit	exit for the tester	PC = PC

Most instructions take around 2 clock cycles to execute while the Load, store and I/O instructions take 2 clock cycles to execute excluding the fetch cycle. The load and store indirect instructions are necessary to store data in the memory apart from the register file. This has to be done indirectly as the instruction can only support up to 4 bits of data for the address of the memory (i.e., from 0 up to 15 which constitutes the register file).

Table 2 lists all ALU operations, including addition, subtraction, and logic operations.

Table 2

Encoding	Name	Operation
0000	add	A = A+op
0001	sub	A = A−op
0010	adc	A = A+op+c
0011	sbb	A = A−op−c
0100	and	A = A ∩ op
0101	or	A = A ∪ op
0110	xor	A = A ⊕ op
0111	ld	A = op
1000	lrs	A = A >> 1
1001	lls	A = A << 1
1010	Lr	Left Rotate
1011	Rr	Right Rotate
1100	ars	A = A >>> 1

5. Working of Instructions

The State diagrams of each instruction that we have implemented are attached below. The type of state machine that we have followed is the Moore machine as the output depends only on the present state.

i. ALU Register:

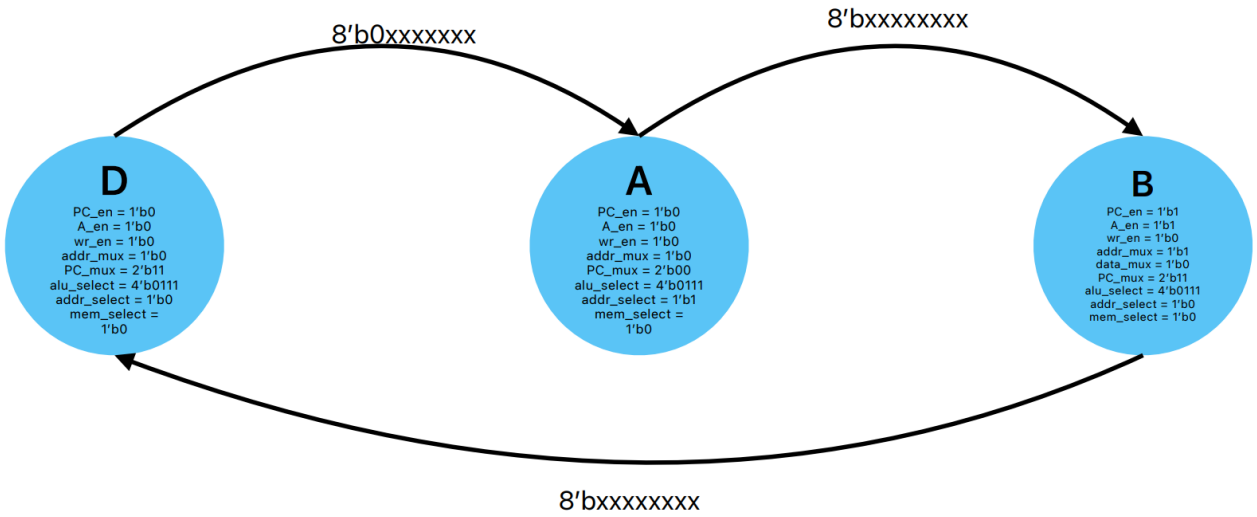

Figure 2 *State Diagram for ALU Register Operation.*

ii. Store Operation

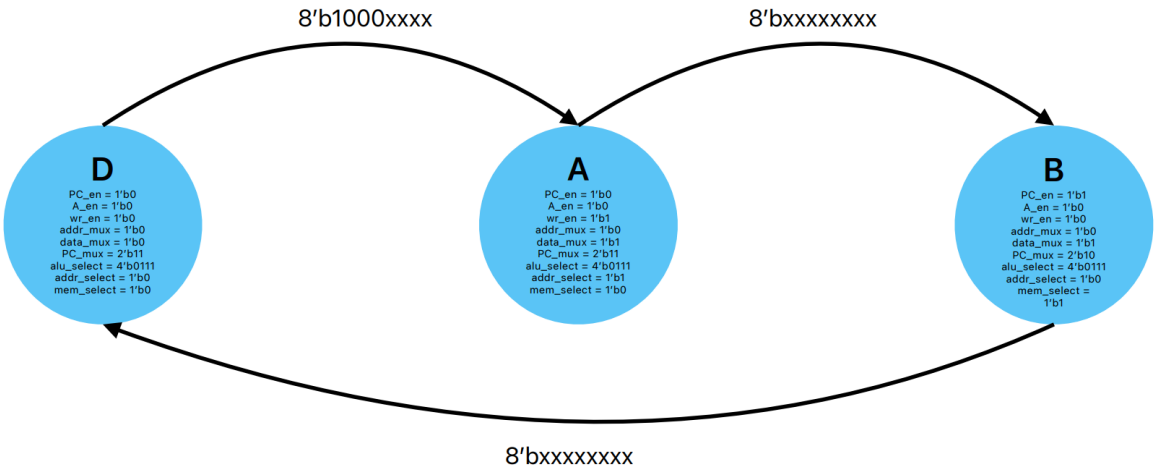

Figure 3 *State Diagram for Store Operation.*

iii. Branch and Link

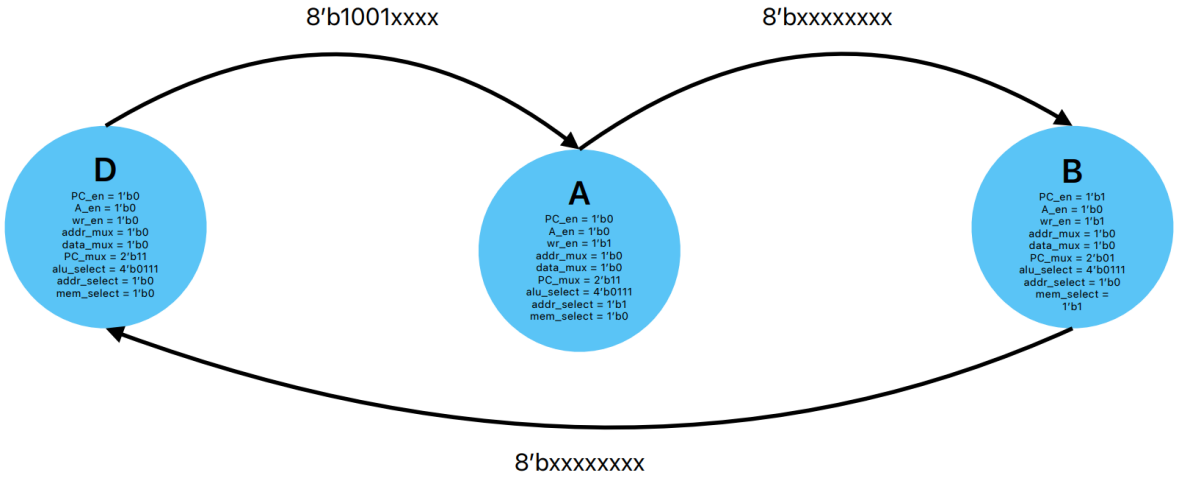

Figure 4 *State Diagram for Branch and Link Operation.*

iv. Load Indirect

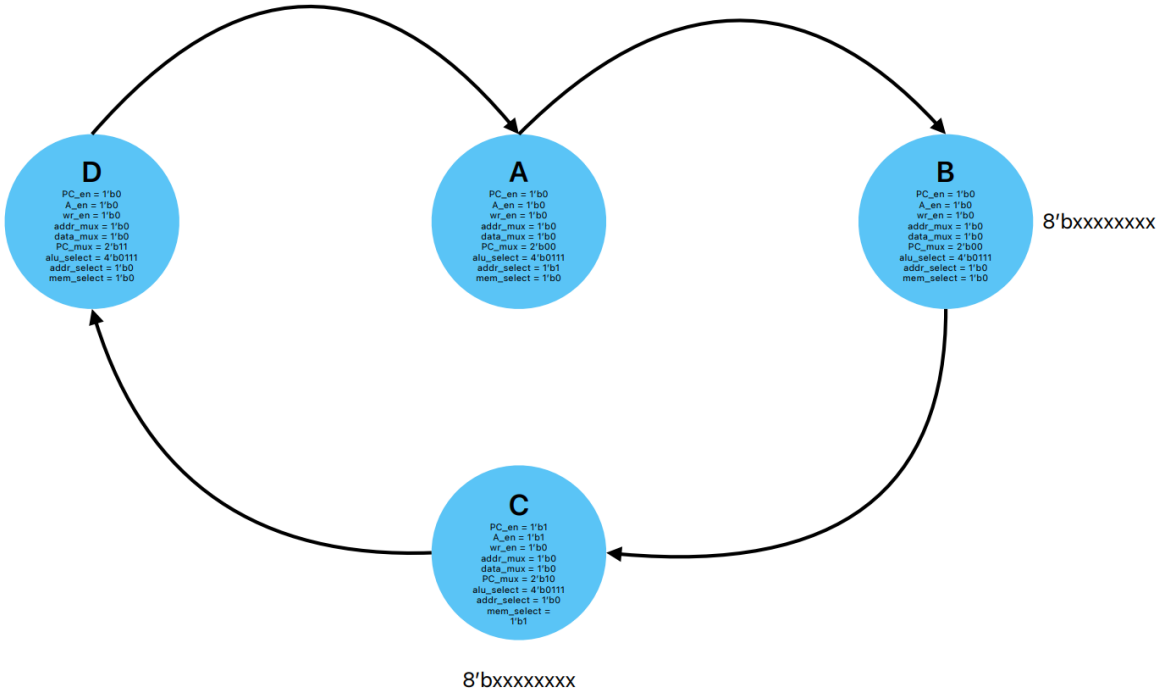

Figure 5 *State Diagram for Load Indirect Operation.*

v. Store Indirect

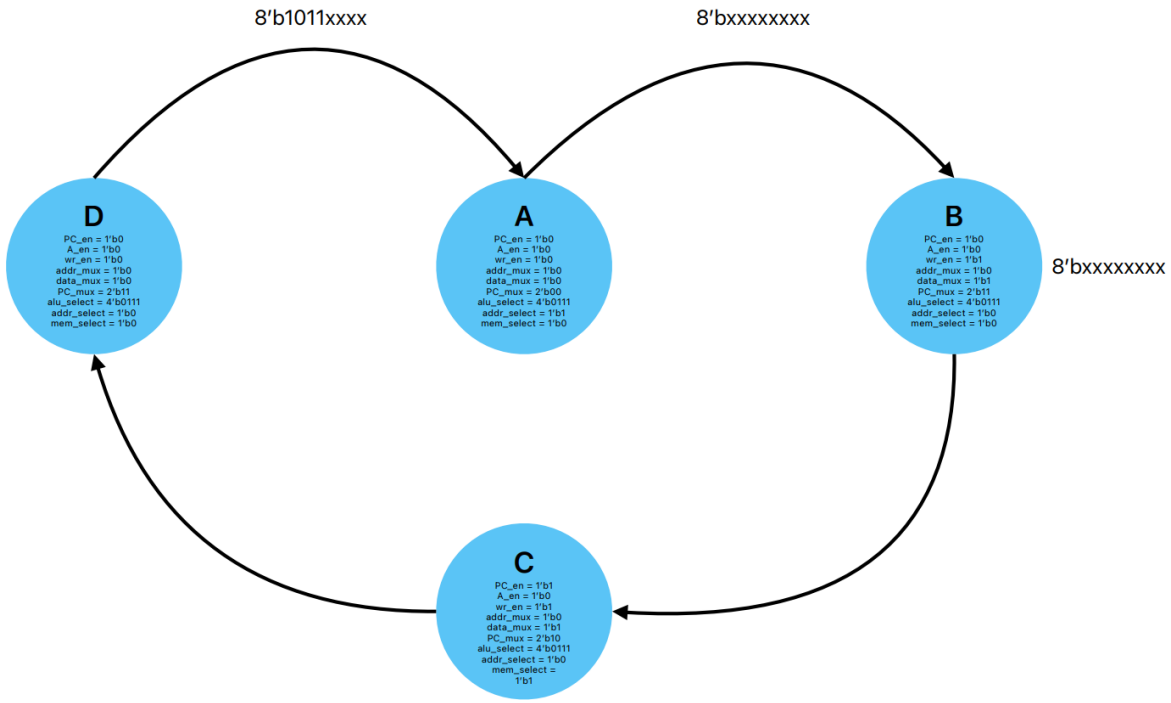

Figure 6 *State Diagram for Store Indirect Operation.*

vi. ALU Immediate

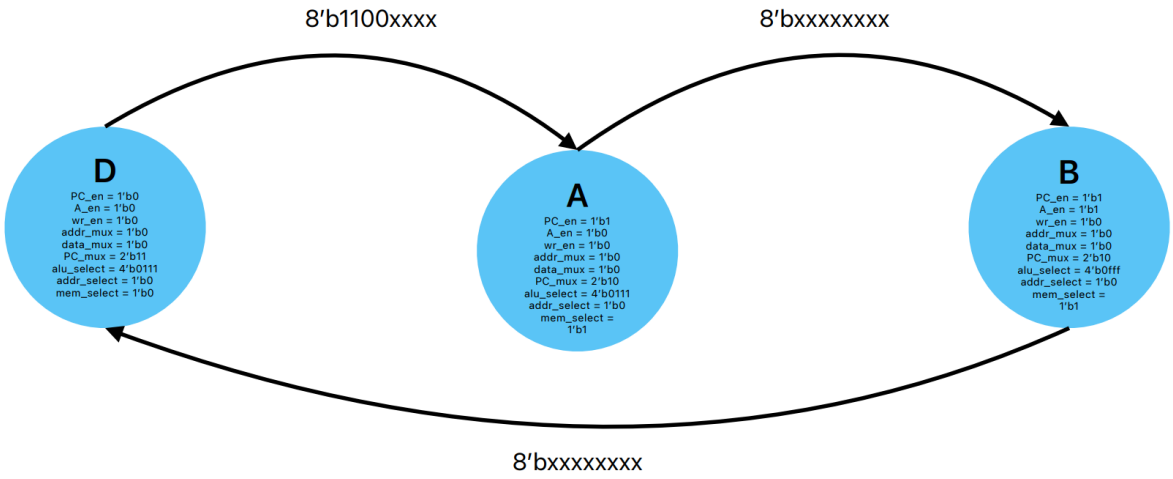

Figure 7 *State Diagram for ALU Immediate Operation.*

vii. Branch when Zero

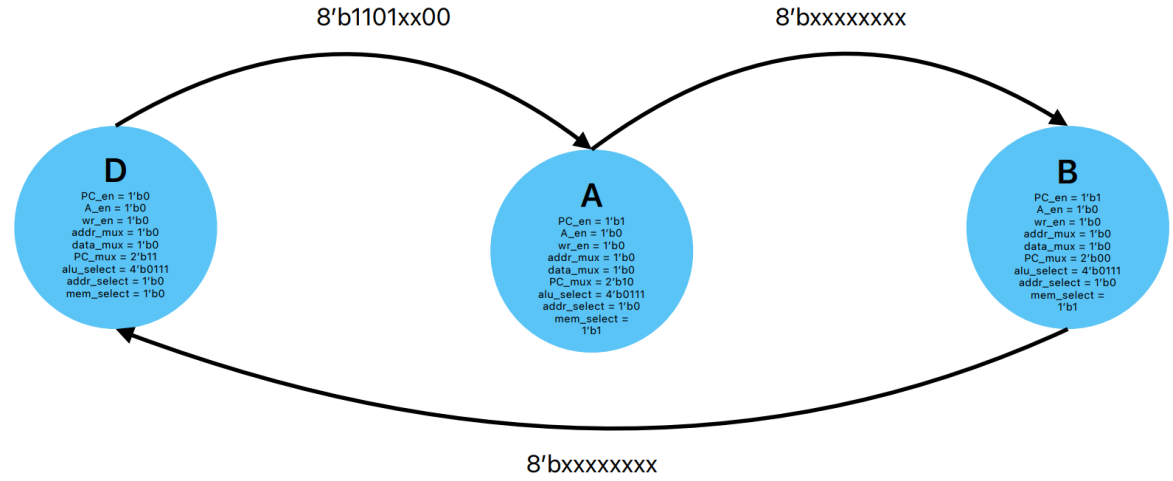

Figure 8. *State Diagram for Branch when Zero Operation.*

viii. Branch when Not Zero

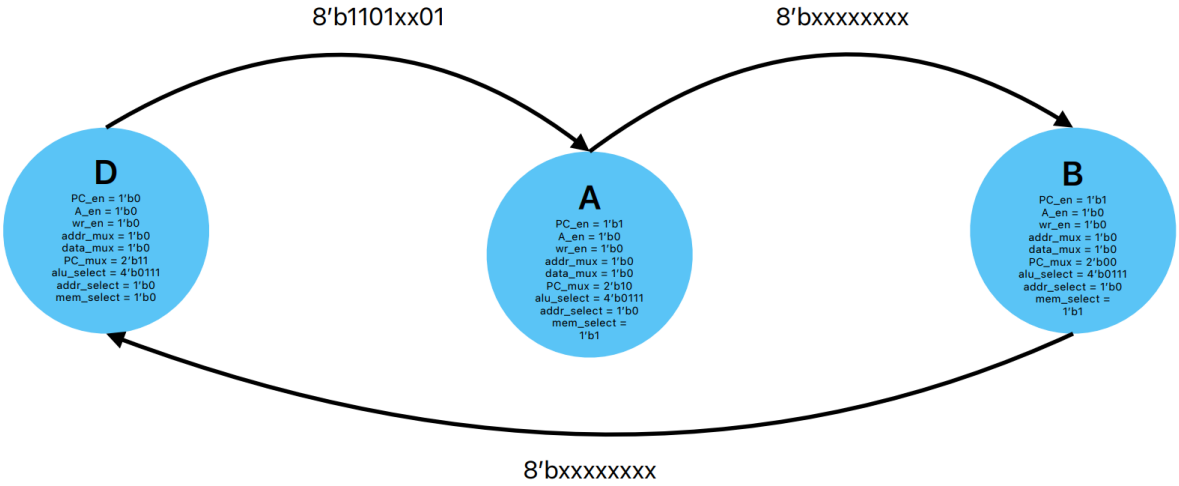

Figure 9 *State Diagram for Branch when Not Zero Operation.*

ix. ALU Shift

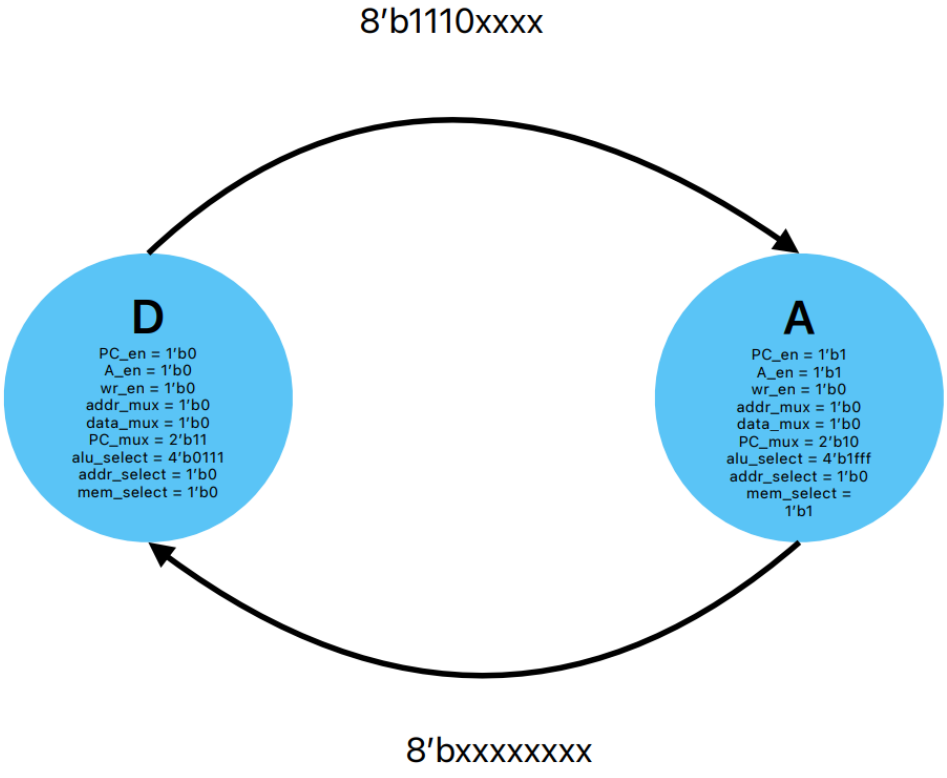

Figure 10 *State Diagram for ALU Shift Operation.*

x. I/O Operation

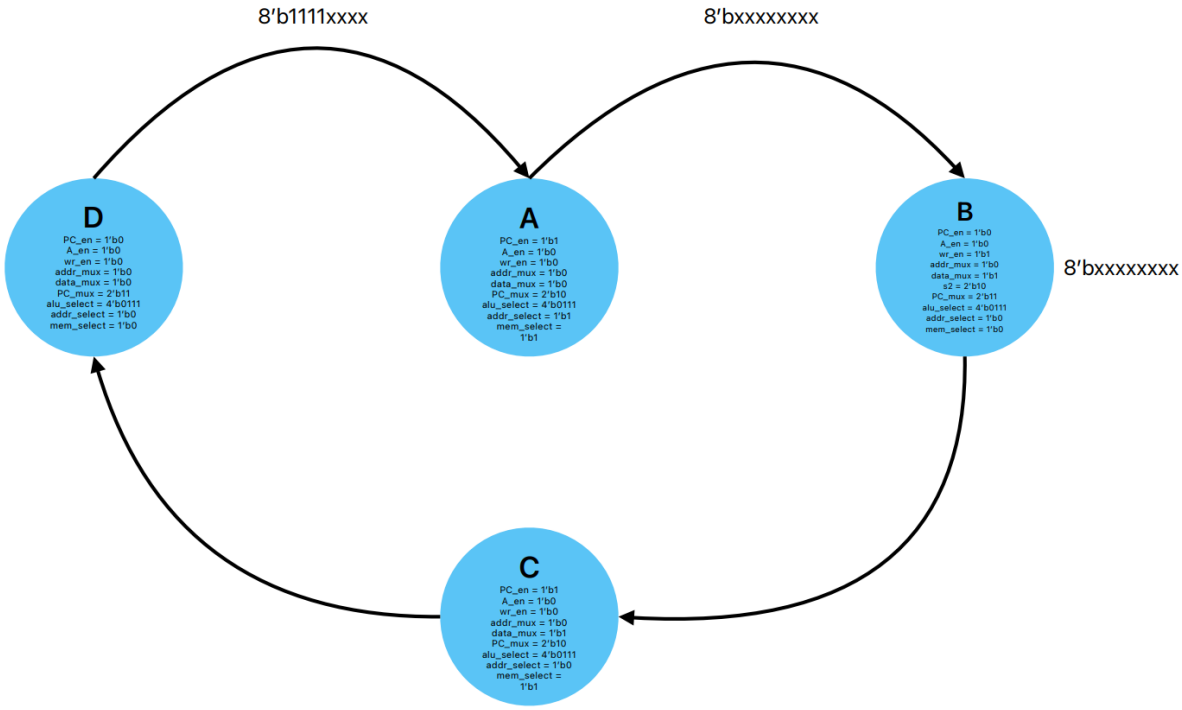

Figure 11 *State Diagram for I/O Operation*

.

xi. Exit Operation

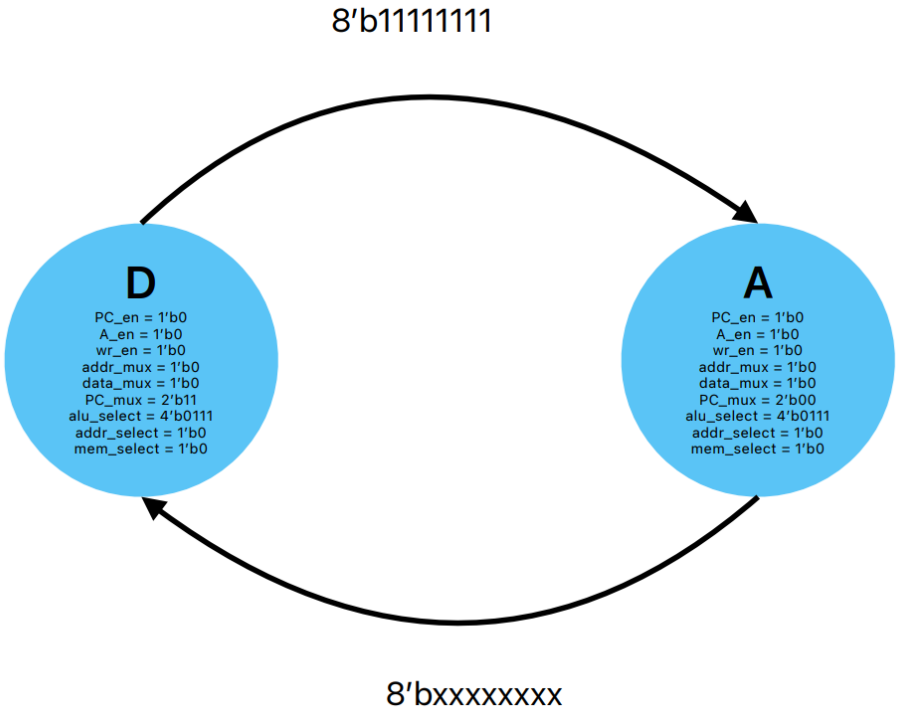

Figure 12 *State Diagram for Exit Operation.*

6. Possible Improvements

The overall cycle count for most instructions can be reduced by introducing dedicated states for each instruction, rather than reusing common states as done in the current implementation. This approach enables finer control over execution timing and can lead to improved performance.

Further optimization is possible for store-type instructions by enabling parallelism in memory read and write operations. Such architectural enhancements allow overlapping of data movement, thereby reducing the effective number of cycles per instruction.

Logic element utilization can also be significantly improved through efficient instruction encoding strategies. For instance, subtraction can be implemented using the existing addition circuitry, eliminating the need for a separate subtract unit and thereby reducing hardware redundancy.

Additionally, support for custom instructions may be incorporated to enhance flexibility and performance for application-specific requirements. These instructions can be tailored to accelerate common computation patterns or peripheral control.

Finally, machine code can be conveniently uploaded to the FPGA via a UART-based bootloader. This mechanism offers a lightweight and flexible solution for code deployment without requiring reprogramming of the FPGA fabric.

7. Difficulties faced

Designing Lipsi, which is a multi-cycle processor, involved building a detailed finite state machine. This process was challenging due to limited prior experience and the learning curve involved in understanding how processors function. It took time to figure out how each instruction moves through different stages and how these stages connect within the control flow.

Assigning control signals to different parts of the processor—such as memory, the accumulator, and enable lines—required careful planning to make sure everything worked as expected. One of the most difficult parts was debugging: testing each instruction on its own and then making sure the processor behaved correctly when multiple instructions were executed in sequence.

8. Conclusion

In conclusion, the design and implementation of the Lipsi multi-cycle processor provided valuable insight into processor architecture and control logic. Despite the initial challenges, especially in managing the control signals and debugging the state machine, the project deepened our understanding of instruction execution, hardware coordination, and system-level integration. This experience has laid a strong foundation for more advanced digital design and processor development in the future. The potential improvements suggested above would increase the functionality as well as the ease of use of the processor.

9. References

1. https://github.com/schoeberl/lipsi