Understanding the Logic of a VHDL 8-Bit Processor Design

1 Introduction

This document outlines the underlying logic of an 8-bit processor design implemented in VHDL, focusing on its architectural components and operational flow. The design incorporates various functionalities, including control signals, data manipulation, and communication protocols specific to 8-bit processing.

2 Architectural Overview

The processor architecture consists of several key components, as illustrated in Figure 1.

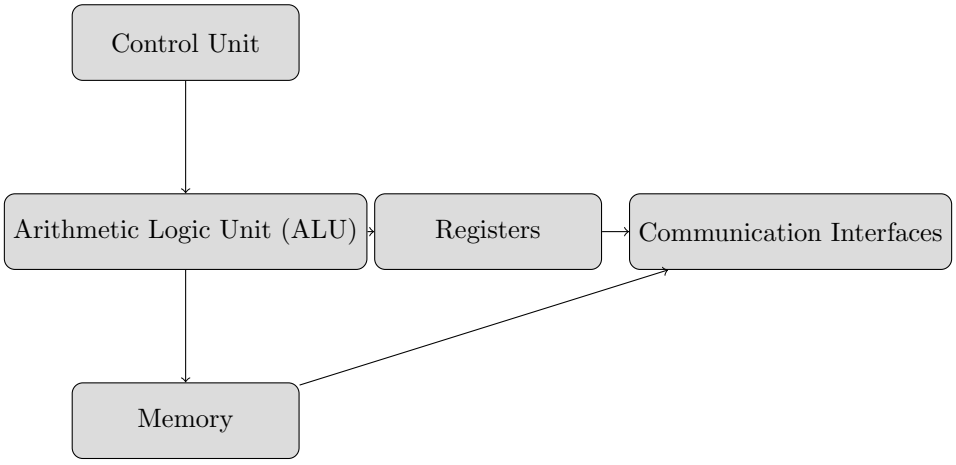

Figure 1: Basic 8-Bit Processor Architecture

The Control Unit directs operations by sending control signals based on the current state and input instructions.

The Arithmetic Logic Unit (ALU) performs arithmetic and logical operations on 8-bit data, utilizing flags to manage overflow and other conditions relevant to 8-bit operations.

Registers store intermediate 8-bit data and state information, including the program counter, accumulator, and status flags.

A simple memory model is employed for storing instructions and data, facilitating read and write operations with 8-bit data sizes.

The communication interfaces allow the processor to exchange data with external devices, including UART, I2C, and SPI.

2.1 Functional Logic

The processor operates based on a state machine architecture, where the state transitions are determined by control signals and the current operational context. The operational flow is depicted in Figure 2.

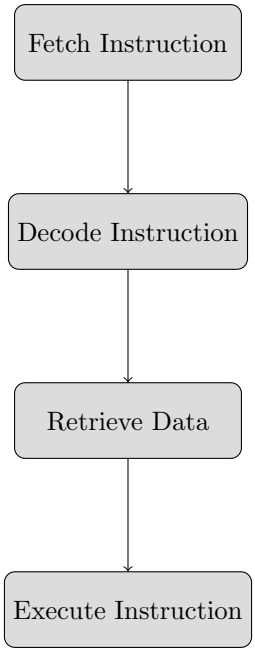

Figure 2: 8-Bit Processor Operational Flowchart

2.2 Initialization

Upon reset, all internal 8-bit registers and flags are initialized to a known state, ensuring the processor starts from a clean slate. This includes clearing memory, setting counters to zero, and preparing the control unit for the first instruction.

2.3 Instruction Fetching

The processor fetches instructions from memory sequentially. The program counter (PC) keeps track of the address of the current instruction. Upon fetching an instruction, the PC is incremented to point to the next instruction in the sequence.

2.4 Decoding and Execution

Once an instruction is fetched, it is decoded to determine the operation to be performed. The decoded instruction will trigger specific actions within the processor:

- **Arithmetic Operations:** The ALU processes instructions such as addition, subtraction, and logical operations, modifying the accumulator and updating status flags accordingly. Overflow detection is crucial in an 8-bit environment.
- **Data Movement:** The processor can move 8-bit data between registers, memory, and external devices, facilitating read and write operations as dictated by the instructions.
- **Control Flow:** Instructions can modify the flow of execution, allowing for conditional branching and jumps based on the state of the flags (e.g., zero, negative, overflow).

2.5 Communication Protocols

The processor includes interfaces for communication with external components, utilizing several protocols:

- **SPI (Serial Peripheral Interface):** Used for connecting to external devices such as LCD screens and external memory modules. This allows the processor to upload assembly code to the memory and display information on the LCD.

- **I2C (Inter-Integrated Circuit):** Employed for controlling sensors like the MPU6050 (a motion tracking device) and digital-to-analog converters (DACs). I2C provides a simple way to communicate with multiple devices over a two-wire interface.
- **UART (Universal Asynchronous Receiver-Transmitter):** This protocol enables communication between the processor and a PC, allowing for data exchange, debugging, and control of the processor from a host computer.

2.6 Processing States

The processor implements a series of processing states that define its operation:

- **Fetch State:** In this state, the processor fetches the next instruction from memory by using the program counter (PC) to specify the address. The address is sent to the memory, and the processor initiates the fetch process. Once the memory is ready, the instruction is loaded into the instruction register for further processing.
- **Decode State:** During the decode state, the fetched instruction is analyzed to determine its type and the required operation. The control unit interprets the instruction and generates the appropriate control signals. This state may also involve determining operand addresses or preparing immediate values needed for execution.
- **Retrieve State:** In the retrieve state, the processor retrieves the necessary operands or data required for instruction execution. The processor sets the address input based on the program counter (PC) and initiates memory access by asserting a start signal. It waits for the memory to indicate that the data is ready. Once ready, the processor reads the data into the appropriate registers, ensuring that all required data is available before proceeding to execute the instruction.
- **Execute State:** In the execute state, the processor performs the operation specified by the decoded instruction. This may involve arithmetic or logical operations performed by the Arithmetic Logic Unit (ALU), data movement between registers, or interaction with memory. The results of the execution are updated in the registers or memory as specified by the instruction.

3 Assembly Code Conversion to Binary

In addition to the processor's functionality, a Python script has been developed to convert assembly code into binary format and upload it to the SPI external memory. The script processes assembly instructions, maps them to their corresponding binary representations, and handles both immediate values and register operations.

The Python script operates as follows:

- **Command Mapping:** Each assembly command is matched to a specific binary byte value using regular expressions. This allows the program to recognize various operations, including arithmetic, data transfer, and control commands.
- **Assembly to Binary Conversion:** The assembly code is parsed line by line. For each line, the corresponding binary representation is determined based on the command and any constants or registers specified. This includes handling string outputs for the `PRNT` command.
- **Binary File Generation:** The resulting binary data is written to a file (e.g., `compile.bin`), which is subsequently uploaded to external SPI memory using a command-line utility (e.g., `openFPGALoader`).

The following is a simplified version of the Python code used for this process:

```
import re
import subprocess

# Define commands with regex patterns and their binary equivalents
commands = {
    'ADD': '0001',
    'SUB': '0010',
    'MOV': '0011',
    'PRNT': '0100',
    # Add additional commands as necessary
```

```
}

def assemble_line(line):
    """Convert a single line of assembly code to binary."""
    match = re.match(r'(\w+)\s*(.*)', line)
    if match:
        command, operands = match.groups()
        binary = commands.get(command)
        if binary:
            # Additional processing for operands if necessary
            return binary
    return None

def assemble_code(assembly_code):
    """Convert a complete assembly code to binary."""
    binary_output = []
    for line in assembly_code.splitlines():
        binary_line = assemble_line(line.strip())
        if binary_line:
            binary_output.append(binary_line)
    return binary_output

def write_binary_file(binary_output, filename='compile.bin'):
    """Write the binary output to a file."""
    with open(filename, 'wb') as f:
        for binary in binary_output:
            f.write(bytes([int(binary, 2)]))

# Example assembly code
assembly_code = """
MOV A, 5
ADD A, B
PRNT A
"""

# Assemble the code
binary_output = assemble_code(assembly_code)

# Write to binary file
write_binary_file(binary_output)

# Upload binary to SPI memory
subprocess.run(['openFPGALoader', 'compile.bin'])
```

The integration of this conversion process allows users to write assembly code for the processor, which can then be easily converted to binary and uploaded to external memory for execution. This significantly enhances the usability of the processor design.

4 Conclusion

This document provides an overview of an 8-bit processor design implemented in VHDL, emphasizing its architecture, operational flow, and the associated assembly code conversion process. The combination of VHDL implementation with a Python-based assembly code converter allows for an efficient workflow in developing and testing assembly programs on the designed processor.