

Design and Implementation of a 32-bit ARM- Compatible RISC Processor Using Verilog

A PROJECT REPORT

Submitted by

Siddhapura Yash M

230283111043

BACHELOR OF ENGINEERING

in

Electronics & Communication



L.D. College of Engineering, Ahmedabad

Gujarat Technological University, Ahmedabad

July – 2025

ABSTRACT

This project presents the **design and implementation of a 32-bit ARM-compatible RISC processor** using **Verilog Hardware Description Language (HDL)**. The goal is to develop a simplified ARM-like CPU that supports a basic set of instructions typically used in embedded systems. Unlike pipelined architectures, this processor operates in a **non-pipelined, multi-cycle execution model**, which simplifies control logic and hardware complexity while still providing meaningful insight into how instruction execution works at the architectural level.

The processor is built using a **controller-datapath architecture**, where the **controller (FSM-based)** generates appropriate control signals for each instruction, and the **datapath** executes operations like arithmetic computation, memory access, and branching. Key modules include the **Instruction Register, Program Counter, Register File, Arithmetic Logic Unit (ALU), Comparator, Sign Extender, and Multiplexers**, all designed at the **RTL (Register Transfer Level)**. The processor supports essential instructions such as **data processing, load/store, and branching**, mimicking the behavior of a subset of the ARM instruction set.

Verification was performed through **simulation and testbenches**, where various instruction sequences were tested to ensure correctness of the control signals and functional output. This project not only demonstrates a working CPU model but also strengthens understanding of **digital design, finite state machines (FSMs), and instruction-level hardware behavior**. It serves as a valuable learning experience for anyone pursuing interests in **VLSI, computer architecture, or embedded system design**, and can be further extended with features like interrupt handling or pipelining in future work.

List of Figures :

Figure No.	Title	Page No.
Fig. 3.1	Top-Level CPU Architecture Overview	8
Fig. 4.5	General Instruction Format Overview	12
Fig. 4.6	Data Processing Instruction Format	13
Fig. 4.7	Memory Instruction Format	14
Fig. 4.8	Branch Instruction Format	15
Fig. 5.2	Detailed Description of the Controller-Datapath Diagram	21
Fig. 6.2	Description of the Detailed Control and Data Path Diagram	23
Fig. 7.2	Description of the Control Unit and Decoder Diagram	26

List of Tables :

Table No.	Title	Page No.
Table 4.11	ALU Control Logic Table	18
Table 4.12	Condition Code Table and Description	19
Table 4.13	Summary of addressing modes	20
Table 8.5	Assembly Program and Explanation	43

Table of Contents :

Section	Title	Page
	Acknowledgement	I
	Abstract	II
	List of Figures	III
	List of Tables	IV
	Table of Contents	V
1	INTRODUCTION	1
	1.1 Project Overview and Objectives	1
	1.2 Motivation for ARM CPU Design	1
	1.3 Introduction to ARM Architecture (Simplified)	1
	1.3.1 Key Features of ARM	1
	1.3.2 ARM Instruction Set Overview (Focus on a subset)	2
	1.4 Introduction to Verilog HDL	2
	1.4.1 Basics of Verilog (Modules, Ports, Data Types)	2
	1.4.2 Behavioral vs. Structural Modeling	2
	1.5 Project Scope and Limitations	2
	1.6 Document Organization	3
2	ARM ARCHITECTURE FUNDAMENTALS	4
	2.1 CPU Operation Cycle	4
	2.1.1 Instruction Fetch	4
	2.1.2 Instruction Decode	4
	2.1.3 Execution	4
	2.1.4 Memory Access	4
	2.1.5 Write-Back	5
	2.2 Register File	5
	2.2.1 General Purpose Registers (GPRs)	5
	2.2.2 Program Counter (PC)	5
	2.2.3 Status Register (CPSR - simplified flags)	5

	2.3 Instruction Set Architecture (ISA) Subset	6
	2.3.1 Data Processing Instructions	6
	2.3.2 Load/Store Instructions (LDR, STR)	6
	2.3.3 Branch Instructions (B, BL - simplified)	6
	2.3.4 Simplified ARM Instruction Formats	6
	2.4 Memory Organization	7
	2.4.1 Data Memory	7
	2.4.2 Instruction Memory	7
3	CPU ARCHITECTURE AND MODULE DESIGN	8
	3.1 Top-Level CPU Architecture Block Diagram	8
	3.2 Program Counter (PC)	8
	3.3 Instruction Memory	8
	3.4 Instruction Register (IR)	9
	3.5 I/P (Instruction Parse / Immediate Processor)	9
	3.6 General Purpose Register (GPR)	9
	3.7 ALU (Arithmetic Logic Unit)	9
	3.8 Data Memory	9
	3.9 I/O (Input/Output)	9
	3.10 Controller	10
	3.11 Clock and Reset Logic	10
	3.12 Design Consideration and Trade-offs	10
4	INSTRUCTION REGISTER AND FORMATS	11
	4.1 Introduction to the Instruction Register (IR)	11
	4.2 Role and Functionality of the IR	11
	4.3 IR and Instruction Decoding	11
	4.4 Introduction to ARM Instruction Formats	12
	4.5 General Instruction Format Overview	12
	4.6 Data Processing Instruction Format	13
	4.7 Memory Instruction Format	14
	4.8 Branch Instruction Format	15

	4.9 Detailed Bit Descriptions and Their Purpose	15
	4.10 Addressing Mode Summary	17
	4.11 ALU Control Logic Table	18
	4.12 Condition Code Table and Description	19
	4.13 Summary of Addressing Modes	20
5	CONTROLLER-DATAPATH INTERACTION	21
	5.1 Introduction to Controller-Datapath View	21
	5.2 Detailed Description of the Controller-Datapath Diagram	21
	5.3 Control Signal Generation and Application	22
6	DETAILED CONTROL AND DATA PATH	23
	6.1 Introduction to Detailed Datapath	23
	6.2 Description of ctrl_data_path.jpg	23
	6.3 Signal Routing and Multiplexing	24
7	CONTROL UNIT AND DECODER DESIGN	26
	7.1 Introduction to Decoder Logic	26
	7.2 Description of the Control Unit and Decoder Diagram	26
	7.3 Control Signal Derivation	28
	7.4 Detailed Bit Descriptions and Their Purpose	28
	7.5 Why Such Detailed Decoding?	30
8	VERILOG IMPLEMENTATION	31
	8.1 Module-by-Module Verilog Code	31
	8.1.1 extender.v	
	8.1.2 ALU.v	
	8.1.3 regfile.v	
	8.1.4 MUX.v	
	8.1.5 adder.v	
	8.1.6 flop.v	
	8.1.7 datapath.v	
	8.1.8 Decoder.v	
	8.1.9 condlogic.v	

	8.1.10 flopenr.v	
	8.1.11 controller.v	
	8.1.12 CPU.v	
	8.1.13 imem.v	
	8.1.14 dmem.v	
	8.1.15 top.v	
	8.1.16 memfile.dat	
	8.1.17 testbench.v	
	8.2 Memory Initialization	42
	8.3 Design Considerations and Trade-offs	43
	8.4 Result of Simulation in Log	43
	8.5 Assembly Program and Explanation	43
9	CONCLUSION AND FUTURE WORK	46
	9.1 Summary of Project Achievements	46
	9.2 Challenges Faced and Lessons Learned	46
	9.3 Future Enhancements	46
	9.4 References	47

Chapter 1: Introduction

1.1 Project Overview and Objectives

This project aims to design and implement a simplified Central Processing Unit (CPU) based on the ARM (Advanced RISC Machine) architecture using Verilog Hardware Description Language (HDL). The primary objective is to gain a comprehensive understanding of CPU architecture and the practical application of Verilog for digital hardware design. The project will involve the creation of various functional units, including an Instruction Fetch unit, Instruction Decode unit, Execute unit (with ALU), Memory unit, and Write-Back unit. Through this implementation, we intend to demonstrate the fundamental principles of a modern processor and its operation.

1.2 Motivation for ARM CPU Design

The ARM architecture dominates the embedded and mobile computing markets due to its power efficiency, performance, and scalability. Understanding ARM's design principles is crucial for aspiring hardware engineers and computer architects. This project provides a hands-on opportunity to delve into the core mechanisms that make ARM processors so prevalent. By designing a simplified version, we can appreciate the complexities involved in processor design, from instruction set implementation to pipeline management and hazard resolution, all within a manageable scope for a final year project.

1.3 Introduction to ARM Architecture (Simplified)

The ARM architecture is a family of Reduced Instruction Set Computer (RISC) instruction set architectures. Unlike Complex Instruction Set Computers (CISC), RISC architectures emphasize a smaller, highly optimized set of instructions, leading to simpler hardware designs, faster execution, and lower power consumption.

1.3.1 Key Features of ARM

1. **RISC Principles:** ARM processors adhere to RISC principles, featuring a fixed instruction size, a large number of general-purpose registers, and a load/store architecture. This means that data processing operations only occur on registers, and memory access is handled exclusively by dedicated load and store instructions.
2. **Load/Store Architecture:** All data operations are performed on operands held in registers. Data must be explicitly loaded from memory into registers before processing and stored back to memory from registers after processing. This simplifies the control unit and allows for efficient pipelining.
3. **Pipelining:** Most ARM processors employ pipelining, a technique that allows multiple instructions to be processed concurrently in different stages of execution, significantly improving throughput.

1.3.2 ARM Instruction Set Overview (Focus on a subset)

For this project, we will focus on implementing a simplified subset of the ARM instruction set. This subset will typically include:

1. **Data Processing Instructions:** Operations like ADD (addition), SUB (subtraction), AND (bitwise AND), ORR (bitwise OR), EOR (bitwise XOR), and MOV (move data). These instructions operate on register values and can optionally update condition flags.
2. **Load/Store Instructions:** LDR (load register from memory) and STR (store register to memory). These are the only instructions that interact directly with the data memory.
3. **Branch Instructions:** B (unconditional branch) and BL (branch with link, used for function calls). These instructions alter the program flow.

1.4 Introduction to Verilog HDL

Verilog is a Hardware Description Language (HDL) used for modeling electronic systems. It is widely used in the design and verification of digital circuits at various levels of abstraction, from the gate level to the behavioral level.

1.4.1 Basics of Verilog

1. **Modules:** The fundamental building blocks in Verilog, representing a distinct hardware component with defined inputs and outputs (ports).
2. **Ports:** Interfaces through which modules communicate with each other. They can be inputs, outputs, or inout.
3. **Data Types:** Verilog supports various data types, including reg (for storing values in sequential logic), wire (for connecting components), and integer for general-purpose variables.

1.4.2 Behavioral vs. Structural Modeling

1. **Behavioral Modeling:** Describes the circuit's behavior using procedural constructs like always blocks and initial blocks, focusing on *what* the circuit does rather than *how* it's implemented physically. This is often used for higher-level descriptions.
2. **Structural Modeling:** Describes the circuit's structure by instantiating and connecting lower-level modules (like gates or other custom modules). This approach focuses on *how* the components are interconnected. This project will utilize a combination of both, with individual units often described behaviorally and the top-level CPU described structurally.

1.5 Project Scope and Limitations

This project will focus on a simplified ARM CPU core. Key limitations include:

1. Implementation of a basic instruction set subset.
2. No complex features like pipeline, virtual memory, caches, interrupts, or advanced exception handling.
3. The memory will be modeled simply within the Verilog environment, not as an external memory system. These limitations ensure the project is feasible within the typical timeframe of a final year project while still providing a robust learning experience.

1.6 Document Organization

This document is organized into several chapters. Chapter 2 will detail the fundamental ARM architectural concepts relevant to our simplified core. Chapter 3 will cover the design of individual CPU modules. Chapter 4 will present the Verilog implementation. Chapter 5 will discuss simulation and verification methodologies. Chapter 6 will explore optional FPGA implementation. Finally, Chapter 7 will conclude the project and suggest future enhancements.

Chapter 2: ARM Architecture Fundamentals

This chapter lays the groundwork for understanding the fundamental architectural components and operational flow of our simplified, non-pipelined ARM CPU core. It covers the basic CPU operation cycle, the structure and function of the register file, the specific subset of the ARM instruction set architecture (ISA) that will be implemented, and the memory organization.

2.1 CPU Operation Cycle

In a non-pipelined CPU, each instruction completes all its execution stages before the next instruction begins. This sequential processing simplifies control logic but limits performance. The typical stages an instruction goes through in a single clock cycle (for a single-cycle design) or across multiple clock cycles (for a multi-cycle design) are:

2.1.1 Instruction Fetch (IF)

In this initial stage, the Program Counter (PC) holds the address of the next instruction to be executed. The instruction at this address is fetched from the Instruction Memory and loaded into an Instruction Register (IR). The PC is then incremented to point to the subsequent instruction. For a non-pipelined design, this involves reading the instruction and preparing it for the next stage.

2.1.2 Instruction Decode (ID)

The fetched instruction, now in the Instruction Register, is decoded by the Control Unit. This involves identifying the instruction type (e.g., data processing, load/store, branch) and determining the necessary control signals for subsequent stages. Operands (values from registers) required by the instruction are read from the Register File. Immediate values within the instruction are also identified and possibly sign-extended if necessary.

2.1.3 Execution (EX)

During the execution stage, the main computation for the instruction takes place. This typically involves the Arithmetic Logic Unit (ALU), which performs arithmetic (addition, subtraction) and logical (AND, OR) operations on operands provided by the Register File or immediate values. For branch instructions, the ALU calculates the target address. For load/store instructions, the ALU calculates the memory address. The Shifter Unit might also be used here for data processing instructions that involve shifting.

2.1.4 Memory Access (MEM)

This stage is primarily used by load and store instructions.

1. **Load (LDR):** Data is read from the Data Memory at the address calculated in the Execution stage.

2. **Store (STR):** Data from a register (determined during Decode and possibly modified during Execute) is written to the Data Memory at the calculated address. Other instruction types (data processing, branch) typically bypass this stage or perform no operation on memory.

2.1.5 Write-Back (WB)

In the final stage, the result of the instruction is written back to the Register File. This result could be:

1. The output of the ALU (for data processing instructions).
2. The data read from memory (for load instructions). The Program Counter (PC) is also updated for branch instructions in this stage (or at the end of the fetch stage, depending on the specific design).

2.2 Register File

The Register File is a crucial component of the CPU, providing fast storage for data that the CPU frequently accesses.

2.2.1 General Purpose Registers (GPRs)

Our simplified ARM core will include a set of General Purpose Registers (GPRs), typically 16 registers (R0-R15), each capable of holding a 32-bit word. These registers are used to store operands for computations, intermediate results, and addresses. R15 is conventionally used as the Program Counter (PC).

2.2.2 Program Counter (PC)

The Program Counter (PC), often aliased as R15 in ARM, holds the memory address of the next instruction to be fetched. It is automatically incremented during the Instruction Fetch stage. For branch instructions, its value is updated with the target address.

2.2.3 Status Register (CPSR - simplified flags)

The Current Program Status Register (CPSR) holds important status flags that reflect the result of previous operations and control the CPU's behavior. For simplicity, we will focus on the most common condition flags:

1. **N (Negative):** Set if the result of an operation is negative.
2. **Z (Zero):** Set if the result of an operation is zero.
3. **C (Carry):** Set if an arithmetic operation generates a carry-out (for unsigned operations) or a borrow (for subtraction).
4. **V (Overflow):** Set if an arithmetic operation results in a signed overflow. These flags are used by conditional instructions to determine whether they should execute.

2.3 Instruction Set Architecture (ISA) Subset

To keep the project manageable, we will implement a simplified subset of the ARM ISA. Each instruction will be 32 bits long.

2.3.1 Data Processing Instructions

These instructions perform arithmetic and logical operations on register values. They typically have two source operands (registers or a register and an immediate value) and one destination register.

1. **ADD Rd, Rn, Operand2:** Adds Rn and Operand2, stores result in Rd.
2. **SUB Rd, Rn, Operand2:** Subtracts Operand2 from Rn, stores result in Rd.
3. **AND Rd, Rn, Operand2:** Bitwise AND of Rn and Operand2, stores result in Rd.
4. **ORR Rd, Rn, Operand2:** Bitwise OR of Rn and Operand2, stores result in Rd.
5. **MOV Rd, Operand2:** Moves Operand2 to Rd.
6. *Optional:* **CMP Rn, Operand2:** Compares Rn and Operand2 by subtracting them and setting condition flags, but does not store the result.

2.3.2 Load/Store Instructions

These are the only instructions that interact with memory.

1. **LDR Rd, [Rn, #offset]:** Loads a 32-bit word from memory address Rn + offset into register Rd.
2. **STR Rd, [Rn, #offset]:** Stores the 32-bit word from register Rd to memory address Rn + offset.

2.3.3 Branch Instructions

These instructions alter the sequential flow of program execution.

1. **B label:** Unconditional branch to the address specified by label.
2. **BL label:** Branch with Link. Unconditional branch to label, but stores the address of the instruction *after* the BL in the Link Register (R14) to allow for function return.

2.3.4 Instruction Formats (Simplified ARM instruction formats)

We will define simplified 32-bit instruction formats for the chosen subset. A common approach involves:

1. **Opcode:** Bits identifying the type of operation.

2. **Condition Code (Cond):** (Optional, for conditional execution) Bits indicating the condition under which the instruction executes.
3. **Source Register 1 (Rn):** Bits identifying the first source register.
4. **Source Register 2 (Rm) / Immediate Value:** Bits identifying the second source register or an immediate value.
5. **Destination Register (Rd):** Bits identifying the destination register.
6. **Offset/Address:** Bits for branch targets or memory offsets. The exact bit assignments will be detailed in Appendix A.

2.4 Memory Organization

Our CPU will interact with two distinct memory blocks: one for instructions and one for data.

2.4.1 Data Memory

The Data Memory will be a separate block of memory used to store and retrieve data values. It will be accessed by LDR and STR instructions. For simplicity, it can be modeled as a large array of registers in Verilog, with read and write ports.

2.4.2 Instruction Memory

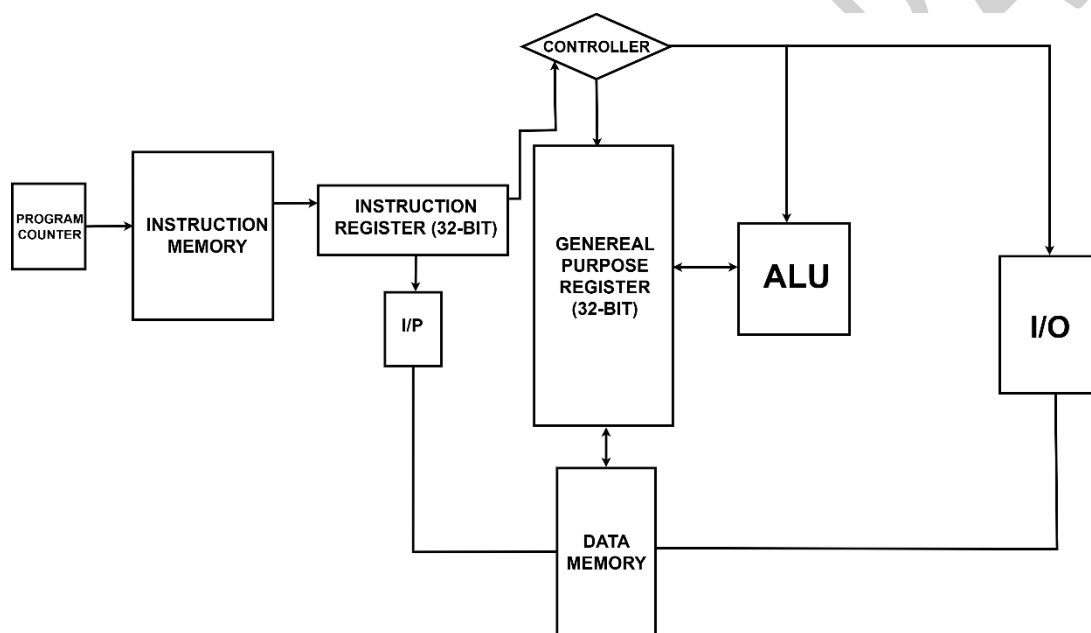
The Instruction Memory will store the program's instructions. It is read-only during normal execution and is accessed by the Instruction Fetch unit. Similar to Data Memory, it can be modeled as a Verilog array, typically initialized at the start of simulation using \$readmemb or \$readmemh with a file containing the machine code.

Chapter 3: CPU Architecture and Module Design

This chapter outlines the fundamental architecture of our simplified, non-pipelined ARM CPU core. We will introduce the main functional blocks and their roles, illustrating the core data flow and control mechanisms.

3.1 Top-Level CPU Architecture Overview

Our CPU is divided into a Datapath for data processing and a Control Unit for managing operations. Instructions are executed sequentially, completing one stage before the next begins. This simple design ensures clear understanding of fundamental CPU operations.



This diagram visually represents the core components and their interconnections within our simplified CPU. Each block plays a crucial role in fetching, decoding, executing, and storing instructions and data. The arrows indicate the primary data and control signal flow between these units.

3.2 Program Counter (PC)

The Program Counter is a 32-bit register that always holds the memory address of the instruction currently being executed. After an instruction is fetched, the PC typically increments by 4 to point to the next instruction in sequential program flow. However, for branch instructions, the PC is updated with a new target address calculated during the execution phase, altering the program's control flow.

3.3 Instruction Memory

This memory block serves as the storage for the program's machine code instructions. It is designed as a read-only component during the CPU's operational phase, meaning instructions

can be retrieved but not modified by the CPU itself. It provides the 32-bit instruction to the Instruction Register based on the address supplied by the Program Counter.

3.4 Instruction Register (IR)

The Instruction Register is a critical temporary storage unit for the 32-bit instruction that has just been fetched from the Instruction Memory. Its purpose is to hold this instruction stable for the entire duration that the Control Unit needs to decode it and other functional units prepare to execute it, ensuring consistent operation throughout the cycle.

3.5 I/P (Instruction Parse / Immediate Processor)

This unit is responsible for the initial analysis and preparation of the fetched instruction. It meticulously extracts various fields from the 32-bit instruction, such as the opcode (which defines the operation), source and destination register addresses, and any immediate values that are directly embedded within the instruction itself. Crucially, it also performs necessary sign extension for these immediate values, converting them to the full 32-bit width while preserving their numerical sign.

3.6 General Purpose Register (GPR)

The General Purpose Register file is a highly efficient bank of fast, 32-bit registers, typically comprising 16 or 32 such registers. These registers are fundamental for storing frequently accessed data, intermediate computational results, and memory addresses. They serve as the primary sources for operands required by the ALU and as destinations for the results of computations or data loaded from memory, enabling rapid data access.

3.7 ALU (Arithmetic Logic Unit)

The ALU stands as the computational core of the CPU. It is capable of performing a wide range of arithmetic operations, including addition and subtraction, as well as bitwise logical operations such as AND, OR, and XOR, on the data provided by the General Purpose Registers or immediate values. Beyond just producing a numerical result, the ALU also generates status flags (like Zero, Negative, Carry, Overflow) based on the outcome of its operations, which are crucial for conditional instruction execution.

3.8 Data Memory

This memory block is specifically designated for storing and retrieving program data, distinct from the instruction memory. Unlike the Instruction Memory, it supports both read (load) and write (store) operations, allowing the CPU to dynamically manipulate data during program execution. Memory addresses for data access are typically computed by the ALU, ensuring flexible and precise data manipulation.

3.9 I/O (Input/Output)

The I/O block represents the CPU's essential interface with the external world, encompassing peripherals and other hardware devices. It facilitates the bidirectional flow of data, enabling the CPU to send information to and receive information from external components. This

allows for interaction with users (e.g., via a display or keyboard) or communication with other systems.

3.10 Controller

The Controller is often referred to as the "brain" of the CPU due to its central role in managing all operations. It interprets the decoded instruction and generates all the necessary control signals that orchestrate the precise actions of every other component within the CPU. This includes directing the Program Counter updates, enabling register reads/writes, instructing the ALU on which operation to perform, and controlling memory access, ensuring that each instruction executes correctly and in the proper sequence.

3.11 Clock and Reset Logic

These are fundamental and indispensable signals for the CPU's synchronous operation. The clk (clock) signal is a periodic pulse that synchronizes all sequential operations, ensuring that all components act in unison and data propagates correctly. The reset signal, when asserted, initializes the entire CPU to a known, starting state, typically clearing all registers and setting the Program Counter to a predefined initial address, preparing the CPU for a fresh execution.

3.12 Design Considerations and Trade-offs

Our non-pipelined CPU design significantly simplifies the control logic compared to more complex architectures, as each instruction completes all its execution stages fully before the next one begins. This approach, while easier to implement and debug due to its straightforward flow, inherently results in lower performance (fewer instructions executed per unit of time) compared to pipelined CPUs. The chosen instruction set subset balances providing essential functionality with maintaining manageable complexity for this project.

Chapter 4: Instruction Register and Formats

This chapter provides a comprehensive overview of the Instruction Register (IR), its fundamental role in the CPU's instruction processing pipeline, and a detailed breakdown of the ARM instruction formats used in our simplified CPU. Understanding these formats and how they are decoded is crucial for both writing assembly programs and implementing the CPU's control logic.

4.1 Introduction to the Instruction Register (IR)

The Instruction Register (IR) serves as a vital temporary storage unit within the CPU's control path. Its primary function is to hold the binary representation of the instruction that has just been fetched from the Instruction Memory. This immediate storage is critical because it provides a stable and unchanging version of the instruction for the entire duration of the clock cycle during which it is being decoded and its operands are prepared for execution by subsequent CPU stages. Without the IR, the instruction might change while other parts of the CPU are trying to interpret it, leading to unpredictable behavior and erroneous execution.

4.2 Role and Functionality of the IR

The Instruction Register is a 32-bit register, mirroring the fixed instruction size of our simplified ARM architecture. Its core functionality is straightforward: on the active clock edge, it loads the 32-bit instruction provided by the Instruction Memory. Once loaded, the instruction remains stable in the IR, providing a consistent input to the decoding logic, specifically the Instruction Parse / Immediate Processor and the main Control Unit. This stability is essential for these units to accurately interpret the instruction's opcode, function fields, and operand addresses without data hazards that could arise from a constantly changing instruction input from the memory bus. The IR effectively acts as the interface between the instruction memory and the control unit, ensuring a reliable instruction stream for processing.

4.3 IR and Instruction Decoding

A key aspect of the Instruction Register's role is its ability to uniformly hold **all types of instructions** used by the CPU, regardless of their specific format or function. In an ARM-like architecture, instructions are fixed-width 32-bit words, and each part of the instruction (specific bit fields) has a precise meaning. Once loaded into the IR, these different bits are then simultaneously presented to the decoding logic. This decoding logic, which includes components like the Instruction Parse / Immediate Processor and the main Control Unit, analyzes the various fields within the 32-bit instruction held in the IR. Based on this analysis, it generates the appropriate control signals that direct the datapath, ensuring the correct operations are performed for the specific instruction type (e.g., R-type, I-type, Load/Store, Branch). The IR's consistent output allows the decoder to work efficiently across all supported instruction formats.

4.4 Introduction to ARM Instruction Formats

The ARM architecture is characterized by its fixed 32-bit instruction length, which simplifies instruction fetching and decoding. Each instruction is meticulously divided into various fields, with specific bit ranges allocated for opcode, operands, and control information. This structured format allows the CPU's Control Unit to efficiently interpret the instruction's intent and generate the necessary control signals to orchestrate the datapath operations. Our simplified core implements a subset of these formats, focusing on data processing, memory access, and branch instructions.

4.5 General Instruction Format Overview

The 32-bit instruction is broadly divided into several common fields that appear across different instruction types. Understanding these common fields is essential before delving into specific instruction formats:

31:28	27:26	25	24:21	20	19:16	15:12	11:0
Cond	op	I	cmd	S	Rn	Rd	Src2

1. **Bits — cond (Condition Field):** This 4-bit field specifies the condition under which the instruction should be executed. This enables conditional execution of nearly all instructions in ARM, reducing the need for explicit branch instructions in many cases. Examples include 0000 for Equal (if Zero flag is set), 0001 for Not Equal (if Zero flag is clear), and 1110 for Always (unconditional execution).
2. **Bits — op (Opcode Class):** This 2-bit field identifies the broad category of the instruction, allowing the main decoder to quickly determine the general instruction format and which other fields are relevant for further interpretation. For instance, 00 might denote Data Processing or miscellaneous instructions, 01 for Load/Store operations, 10 for Branch instructions, and 11 for unimplemented or reserved opcodes.
3. **Bit [25] — I (Immediate Bit):** This single bit is crucial for selecting the nature of the second operand (Src2). If I is 0, Src2 is interpreted as a register value. If I is 1, Src2 is an immediate value. This bit directly controls a multiplexer in the datapath, choosing between a register or an immediate value for ALU operations.
4. **Bits — cmd (Operation Code / Function):** This 4-bit field specifies the exact operation to be performed, such as ADD, SUB, AND, or ORR. This field is typically sent to the ALU control logic to dictate which arithmetic or logical operation the ALU should execute. For example, 0000 could mean AND, 0100 for ADD, 0010 for SUB, and 1100 for ORR.
5. **Bit [20] — S (Set Condition Codes):** This single bit determines whether the instruction updates the CPU's condition flags (N, Z, C, V) in the Current Program Status Register (CPSR). If S is 1, the flags are updated; if 0, they are not. This is

particularly useful for instructions like CMP (compare) or TST (test), which primarily set flags for subsequent conditional operations.

6. **Bits — Rn (First Operand Register):** This 4-bit field denotes the first source register. It typically provides the left operand in ALU operations, allowing selection from 16 distinct general-purpose registers (R0-R15).
7. **Bits — Rd (Destination Register):** This 4-bit field specifies the destination register where the result of the instruction will be stored. Like Rn, it allows addressing one of 16 registers. In comparison instructions (e.g., CMP), this field might be unused as the result is only reflected in the flags, not stored in a register.
8. **Bits [11:0] — Src2 (Second Operand or Shift/Immediate Field):** This 12-bit field is highly versatile and its interpretation depends on the I bit. It can represent a second operand register (Rm), an immediate constant, or a shifted register value. Its precise interpretation is critical for providing the correct second operand to the ALU or memory address calculation unit.

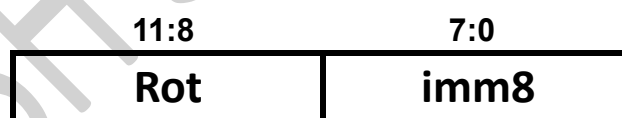
4.6 Data Processing Instruction Format

Data processing instructions perform arithmetic and logical operations. Their format is characterized by op bits being 00.



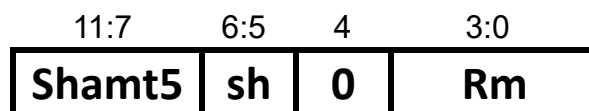
The Src2 field (bits 11:0) has two interpretations based on the I bit:

1. **If I = 1 (Immediate Operand):**



Here, imm8 (8 bits) represents an 8-bit immediate value, and Rot (4 bits) specifies a rotate amount. The 8-bit immediate is rotated right by Rot * 2 bits to form a 32-bit immediate operand.

2. **If I = 0 (Register Operand with Optional Shift):**



In this case, Rm (4 bits) is the second operand register. Shamt5 (5 bits) specifies the shift amount, and sh (2 bits) indicates the shift type (Logical Shift Left, Logical Shift Right, Arithmetic Shift Right, Rotate Right). Bit [4] is typically 0 for this format.

Example: ADD R5, R6, R7 (Add R6 and R7, store in R5)

Assuming an "Always" condition (1110), op=00 (Data Processing), I=0 (Register operand), cmd=0100 (ADD), S=0 (Don't set flags), Rn=6 (R6), Rd=5 (R5), and Src2 refers to R7 with no shift (00000 00 0 0111).

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	0100	0	6	5	0	0	0	7

4.7 Memory Instruction Format

Memory instructions (Load/Store) are used for data transfer between registers and memory. Their format is identified by op bits being 01.

31:28	27:26	25:20				19:16		15:12	11:0	
Cond	01	I	P	U	B	W	L	Rn	Rd	Src2

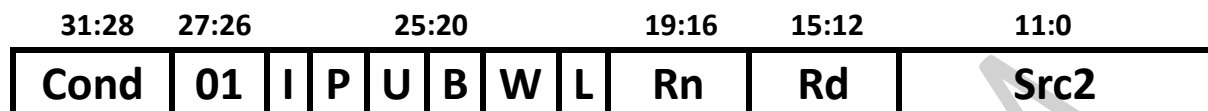
1. **I (25):** Immediate offset. 0 means offset is a register in Src2; 1 means offset is a 12-bit immediate (imm12).
2. **P (24):** Pre/Post indexing. 1 for Pre-indexing (offset applied before memory access); 0 for Post-indexing (offset applied after access, base register updated).
3. **U (23):** Up/Down. 1 to add offset to base; 0 to subtract offset.
4. **B (22):** Byte/Word. 1 for byte transfer; 0 for word (32 bits) transfer.
5. **W (21):** Write-back. 1 means update the base register (Rn) with the new address after offset application; 0 means no write-back.
6. **L (20):** Load/Store. 1 for Load (LDR); 0 for Store (STR).
7. **Rn (19:16):** Base register, holding the starting memory address.
8. **Rd (15:12):** Destination register for LDR or source register for STR.
9. **Src2 / imm12 (11:0):** Contains the offset. If I=0, it's a shifted register offset (Shamt5, sh, Rm). If I=1, it's a 12-bit immediate offset (Imm12).

L	B	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

P	W	Index Mode
0	0	Post-index
0	1	Not-Supported
1	0	Offset
1	1	Pre-index

Example: STR R11, [R5], #26 (Store R11 to memory at address R5, then R5 = R5 + 26)

This is a Post-indexed store with an immediate offset. Cond=Always (1110), op=01 (Load/Store), I=1 (Immediate offset), P=0 (Post-index), U=1 (Add offset), B=0 (Word), W=1 (Write-back), L=0 (Store), Rn=5 (R5), Rd=11 (R11), imm12=26 (0x01A).



4.8 Branch Instruction Format

Branch instructions alter the sequential flow of program execution. Their format is identified by op bits being 10.



1. **Cond (31:28):** Condition for execution, similar to data processing instructions.
2. **10 (27:26):** Fixed bits identifying a branch instruction.
3. **L (25):** Link bit. 0 for a normal branch (B), 1 for a Branch with Link (BL), which saves the return address in the Link Register (LR, R14) for function calls.
4. **Imm24 (23:0):** A 24-bit signed immediate offset. This value is shifted left by 2 bits (multiplied by 4) to get a byte offset, and then added to the PC to compute the branch target address.

4.9 Detailed Bit Descriptions and Their Purpose

This section provides a consolidated explanation of the purpose and significance of each bit field across the various instruction formats.

1. **Op (1:0) - 2 bits:** This 2-bit field allows for up to 4 primary instruction types. This coarse grouping enables the main decoder to quickly determine the general instruction format (e.g., R-type, I-type, Load/Store, Branch/Jump) and subsequently identify which other fields within the instruction are relevant for further decoding.
2. **Funct (5:0) - 6 bits:** Providing 64 possible combinations, this 6-bit field allows for a rich set of specific operations within a given instruction type, particularly for R-type (register-to-register) instructions. It enables the definition of distinct arithmetic and logical operations (e.g., add, subtract, AND, or OR) that share a common opcode but differ in their functional execution.

3. **Rd (3:0) - 4 bits:** Allows addressing 16 distinct registers (R0-R15). This is a common number of general-purpose registers in simpler ARM-like architectures, providing sufficient on-chip storage for frequently accessed data.
4. **Cond (3:0) - 4 bits:** Allows for 16 different conditional execution types. These conditions determine whether an instruction will execute based on the state of the ALU flags (e.g., branch if equal, branch if less than, branch if greater than or equal, always, never).
5. **ALUFlags (3:0) - 4 bits:** Each bit represents a specific status flag generated by the ALU (e.g., Z-flag for zero, N-flag for negative, C-flag for carry, V-flag for overflow). These bits are set by the ALU based on the result of its last operation and are crucial inputs for the Conditional Logic, enabling conditional branching and execution.
6. **FlagW (1:0) - 2 bits:** This 2-bit signal controls different flag write scenarios. It allows for flexibility, such as enabling the writing of all flags, writing only specific flags (e.g., only the zero flag), or completely disabling flag writes for instructions that should not affect the status register.
7. **PCSrc (1 bit):** A single-bit control signal that typically selects between PC+4 (sequential instruction fetching) and the calculated branch target address (for a taken branch). This acts as a select signal for a multiplexer that determines the next value of the Program Counter.
8. **RegW (1 bit):** A single-bit Register Write Enable signal. When '1', data is enabled to be written to a register in the General Purpose Register file. When '0', writes are disabled, preventing unintended modifications to registers.
9. **MemW (1 bit):** A single-bit Memory Write Enable signal. When '1', data is enabled to be written to the Data Memory. When '0', write operations are disabled, preventing unintended data corruption in memory.
10. **MemtoReg (1 bit):** A single-bit control signal that acts as a selector for a multiplexer. If '1', it directs data read from memory (by a Load instruction) to be written to a register. If '0', it selects the ALU result or the output of the extender (for immediate values) to be written to a register.
11. **ALUSrc (1 bit):** A single-bit control signal that selects the second operand for the ALU. A '1' might select an immediate value (from the instruction, after extension) as the ALU operand, while a '0' selects a value read from a register.
12. **ImmSrc (1:0) - 2 bits:** This 2-bit signal allows for up to 4 different immediate processing types. This is essential for handling various instruction formats that use immediate values differently (e.g., sign-extend a 16-bit value, zero-extend a 16-bit value, shift left by 2 for branch targets, or no extension). It ensures correct formatting of immediate values before they are used by the ALU or other units.
13. **RegSrc (1:0) - 2 bits:** This 2-bit signal provides flexibility by allowing for up to 4 different ways to select register inputs to the ALU or other data path components.

For example, it might select R1/R2, R1/immediate, or PC/R2, providing versatility in operand sourcing based on the instruction type.

14. **ALUControl (1:0) - 2 bits:** This 2-bit signal allows for up to 4 different basic ALU operations. For more complex ALUs, this signal would typically be wider (e.g., 3-4 bits for 8-16 operations). It precisely dictates the exact arithmetic or logical function the ALU should perform for the current instruction.

4.10 Addressing Mode Summary

The ARM architecture supports various addressing modes, which define how the operand's effective memory address is calculated or how the operand itself is derived.

1. **Register Only:**
 1. **Example:** ADD R3, R2, R1
 2. **Description:** The operation $R3 \leftarrow R2 + R1$ is performed. Both operands are directly taken from registers.
2. **Immediate-shifted Register:**
 1. **Example:** SUBB R4, R5, R9, LSR #2
 2. **Description:** The operation $R4 \leftarrow R5 - (R9 \gg 2)$ is performed. The second operand is a register value that is first shifted (Logical Shift Right by 2 bits in this example) before being used in the operation.
3. **Immediate:**
 1. **Example:** SUB R3, R2, #25
 2. **Description:** The operation $R3 \leftarrow R2 - 25$ is performed. One of the operands is a constant value directly embedded within the instruction.
4. **Base (Memory Addressing):**
 1. **Immediate Offset:**
 1. **Example:** STR R6, [R11, #77]
 2. **Description:** Data from register R6 is stored to the memory address calculated as $\text{mem}[R11 + 77] \leftarrow R6$. The offset is a constant value.
 2. **Immediate-shifted Register Offset:**
 1. **Example:** LDR R8, [R9, R2, LSL #2]
 2. **Description:** Data is loaded into R8 from the memory address calculated as $R8 \leftarrow \text{mem}[R9 + (R2 \ll 2)]$. The offset is a register value that is first shifted (Logical Shift Left by 2 bits) before being added to the base register.

5. **PC-Relative (Branching):**

1. **Example:** B LABEL1
2. **Description:** The Program Counter is updated to branch directly to LABEL1. The target address is calculated relative to the current PC value using an offset specified in the instruction.

4.11 ALU Control Logic Table

This table illustrates the specific mapping from various input control signals (derived from instruction decoding) to the ALUControl and FlagW outputs. These outputs directly govern the operation performed by the ALU and whether the condition flags in the CPSR are updated. The ALUOp signal, combined with the Funct[4:1] (cmd) and Funct[0] (S) bits from the instruction, determines the exact ALU operation and flag write behavior.

Here is the ALU Control Logic table:

<i>ALUOp</i>	<i>Funct 4:1 (cmd)</i>	<i>Funct[0] (S)</i>	<i>Type</i>	<i>ALUControl 1:0</i>	<i>FlagW 1:0</i>
0	X	X	NOT DP	00	00
1	0100	0	ADD	00	00
		1	ADD	00	11
	0010	0	SUB	01	00
		1	SUB	01	11
	0000	0	AND	10	00
		1	AND	10	10
	1100	0	ORR	11	00
		1	ORR	11	10

The table operates as follows:

1. **ALUOp (1 bit):** This high-level control signal differentiates between non-data processing (0) and data processing (1) instructions.
2. **Funct[4:1] (cmd) (4 bits):** For data processing instructions (ALUOp = 1), these bits specify the core operation (e.g., 0100 for ADD, 0010 for SUB, 0000 for AND, 1100 for ORR).
3. **Funct[0] (S) (1 bit):** This bit indicates whether the condition flags should be updated (1) or not (0) after the ALU operation.
4. **Type:** Describes the type of operation (e.g., NOT DP for non-data processing, ADD, SUB, AND, ORR).

5. **ALUControl[1:0] (2 bits):** These bits are the direct control signals sent to the ALU, instructing it which specific function to perform (e.g., 00 for ADD, 01 for SUB, 10 for AND, 11 for ORR).
6. **FlagW[1:0] (2 bits):** These bits control the writing of the condition flags. A value of 11 typically means all flags (N, Z, C, V) are updated, while 00 means no flags are updated. This depends on the S bit.

This table is a critical component of the ALU Decoder within the Control Unit, ensuring that the correct low-level commands are sent to the ALU based on the decoded instruction.

4.12 Condition Code Table and Description

This table details the various condition codes used in ARM instructions and the corresponding states of the ALU flags that must be met for the condition to be true. These codes are crucial for enabling conditional execution of instructions and conditional branching, allowing the program flow to adapt based on the results of previous computations.

The table operates as follows:

Cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	Z'
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	C'
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	N'
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	V'
1000	HI	Unsigned higher	Z' \wedge C
1001	LS	Unsigned lower or same	Z \vee C'
1010	GE	Signed greater than or equal	N = V
1011	LT	Signed less than	N \neq V
1100	GT	Signed greater than	Z' \wedge (N = V)
1101	LE	Signed less than or equal	Z \vee (N \neq V)
1110	AL	Always / unconditional	Ignored

Explanation of CondEx (Condition Expression) Symbols:

1. **Z**: Zero flag (set if result is zero)
2. **N**: Negative flag (set if result is negative)
3. **C**: Carry flag (set if carry-out occurred)
4. **V**: Overflow flag (set if signed overflow occurred)
5. **X'**: Logical NOT of flag X
6. **X AND Y**: Logical AND of flags X and Y
7. **X OR Y**: Logical OR of flags X and Y
8. **X XOR Y**: Logical XOR of flags X and Y

This table is fundamental for understanding how conditional instructions (which use the Cond field) determine whether they should execute based on the state of the CPU's status flags.

4.13 Summary of addressing modes :

Operand Addressing Mode	Example	Description
Register		
Register Only	ADD R3, R2, R1	$R3 \leftarrow R2 + R1$
Immediate-shifted register	SUBB R4, R5, R9, LSR #2	$R4 \leftarrow R5 - (R9 \gg 2)$
Immediate		
Immediate	SUB R3, R2, #25	$R3 \leftarrow R2 - 25$
Base		
Immediate offset	STR R6, [R11, #77]	$\text{mem}[R11 + 77] \leftarrow R6$
Immediate-shifted register offset	LDR R8, [R9, R2, LSL #2]	$R8 \leftarrow \text{mem}[R9 + (R2 \ll 2)]$
PC-Relative	B LABEL1	Branch to LABEL1

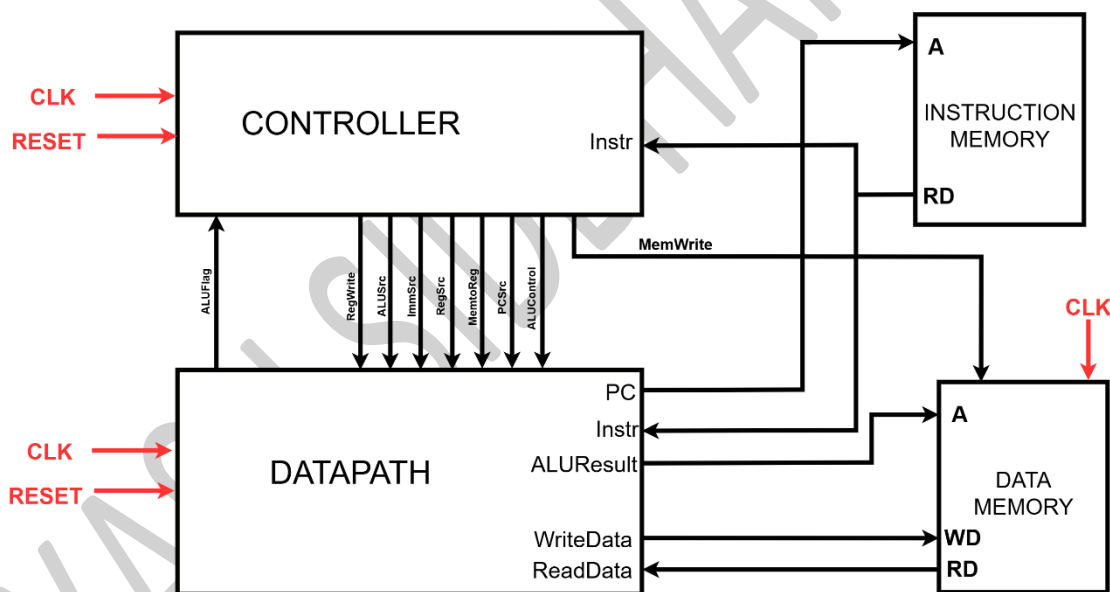
Chapter 5: Controller-Datapath Interaction

This chapter elaborates on the critical interaction between the CPU's two main logical divisions: the Controller and the Datapath, as visually represented in the CPU-TOPVIEW.drawio.jpg diagram. Understanding this interaction is fundamental to comprehending how instructions are translated into physical operations within the processor.

5.1 Introduction to Controller-Datapath View

While previous chapters presented individual CPU components, this chapter delves into the fundamental architectural concept of separating the computational elements (Datapath) from the decision-making and sequencing elements (Controller). The CPU-TOPVIEW.drawio.jpg specifically illustrates how these two major units communicate, with the Controller issuing commands and the Datapath executing them, and providing feedback. This clear separation is a cornerstone of modern CPU design, simplifying complexity and aiding in systematic development.

5.2 Detailed Description of the Controller-Datapath Diagram



This diagram distinctly separates the CPU into two primary blocks: the **CONTROLLER** and the **DATAPATH**. The **CONTROLLER** block serves as the command center, receiving the current instruction (**Instr**) directly from the **Instruction Memory** and crucial status flags (**ALUFlag**) as feedback from the Datapath.

Based on the interpretation of the instruction's opcode and the current state of these flags, the Controller generates a comprehensive set of precise control signals. These signals include, but are not limited to, **RegWrite** (to enable writing to the Register File), **ALUSrc** (to select the second operand for the ALU, either a register value or an immediate), **ImmSrc** (to specify the type of immediate extension), **RegSrc** (to select the source registers for reading), **MemtoReg**

(to determine if the data written back to a register comes from the ALU or memory), PCSrc (to control the Program Counter's next value for branches), ALUControl (to dictate the specific operation the ALU performs), and MemWrite (to enable writing to Data Memory).

These meticulously generated signals are then fed directly to the **DATAPATH**, which encompasses the core execution units such as the Program Counter, the General Purpose Registers, and the Arithmetic Logic Unit (ALU). The Datapath, in turn, actively interacts with the **Instruction Memory** (fetching Instr for the Controller) and the **Data Memory** (performing WD writes and RD reads, using addresses specified by A).

Furthermore, the Datapath outputs results like ALUResult (the outcome of ALU operations) and WriteData (data destined for memory), and receives ReadData from memory. Both the Controller and Datapath are meticulously synchronized by a global clock signal (CLK) and can be brought to a known initial state by a RESET signal, ensuring predictable and reliable operation.

5.3 Control Signal Generation and Application

This diagram profoundly emphasizes the unidirectional flow of control signals from the Controller to the Datapath, which is the essence of how the CPU operates. For every single instruction, the Controller undertakes a meticulous analysis of the instruction's opcode and various function fields. Concurrently, it evaluates the current state of the ALU flags (such as Zero, Negative, Carry, and Overflow), which provide vital information about the outcome of preceding operations.

Based on this comprehensive analysis, the Controller asserts specific control signals that precisely dictate the behavior of each and every component within the Datapath. For instance, the MemWrite signal is activated only when a store instruction is to be executed, ensuring data is written to memory at the correct time. The ALUControl signals are set to specify the exact arithmetic or logical operation the ALU must perform for the current instruction.

Similarly, PCSrc is a critical control signal that determines whether the Program Counter should simply increment for sequential execution or load a new address for a branch instruction, especially for conditional branches where its selection depends on the ALU flags. This intricate and highly coordinated network of control signals is paramount; it ensures that the correct operations are performed on the correct data, at the correct time, and in the correct sequence, thereby enabling the CPU to execute instructions accurately and sequentially. The continuous feedback loop provided by ALUFlag to the Controller is vital, allowing the Controller to make dynamic, conditional decisions that are essential for handling various instruction types and program flow changes.

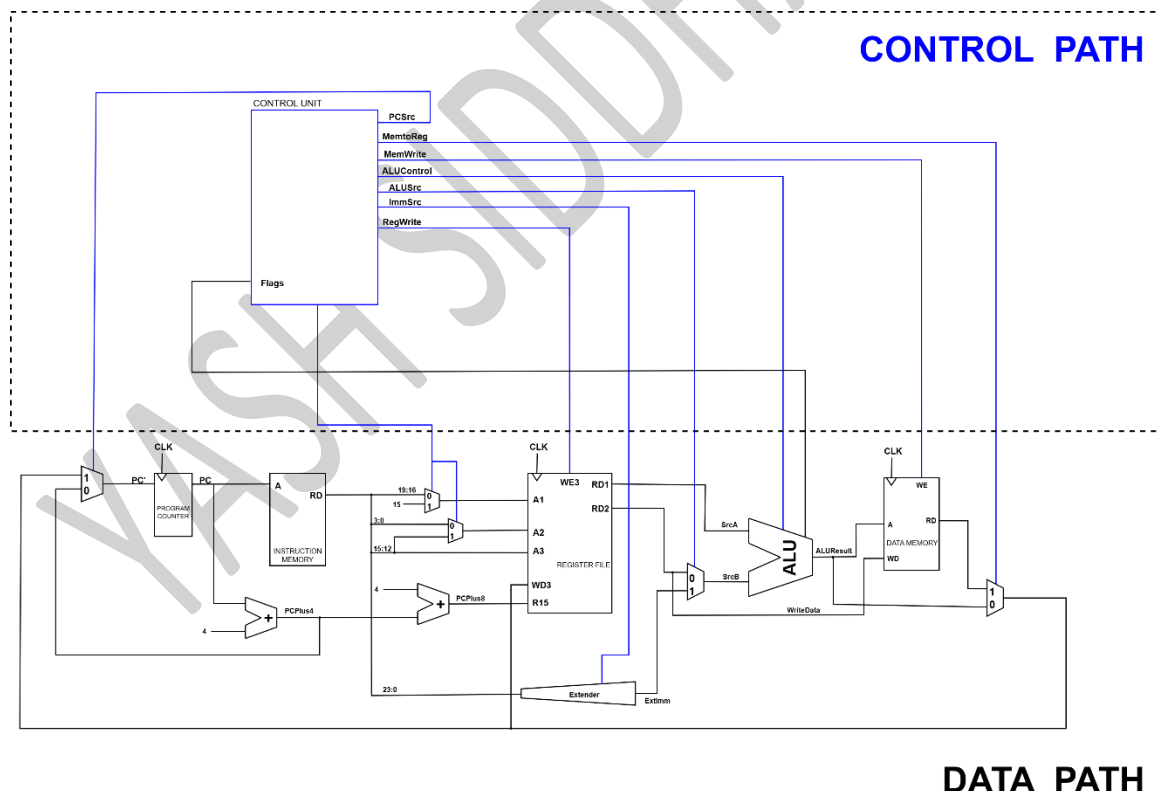
Chapter 6: Detailed Control and Data Path

This chapter provides an in-depth examination of the CPU's internal workings by analyzing the accompanying diagram. This diagram is crucial for understanding the precise routing of data and the application of control signals at a granular level within our non-pipelined ARM CPU.

6.1 Introduction to Detailed Datapath

While previous chapters offered conceptual and high-level views, the detailed datapath diagram presents the comprehensive interconnectivity of the CPU's functional units. It explicitly shows the various data paths and the specific control signals that govern their operation. This level of detail is absolutely essential for the actual Verilog implementation, as it meticulously illustrates the placement and function of multiplexers, extenders, and the intricate connections between registers. These elements collectively enable each instruction to complete its full cycle of operations within the CPU. Effectively, this diagram serves as the bridge, merging the abstract concept of control signals with the physical reality of data flow within the hardware.

6.2 Description of the Detailed Control and Data Path Diagram



This diagram meticulously illustrates the combined **CONTROL PATH** and **DATA PATH** of the CPU, providing a comprehensive view of how instructions are executed at a low level. The **CONTROL UNIT** section, prominently displayed at the top, is responsible for generating a

precise set of control signals. These signals (e.g., PCSrc, MemtoReg, MemWrite, ALUControl, ALUSrc, ImmSrc, RegWrite) are derived based on the specific instruction being executed and the current state of the Flags (condition codes) received from the ALU. This intricate logic ensures that every operation is correctly sequenced and controlled.

The **DATA PATH**, located below the Control Unit, illustrates the dynamic flow of 32-bit data throughout the processor. Instructions are initially fetched from the **INSTRUCTION MEMORY**, with their addresses provided by the **PROGRAM COUNTER (PC)**. Once an instruction is in the system, its various fields are meticulously used to read required data from the **REGISTER FILE**, producing two primary read data outputs (RD1 and RD2). An **Extender** component is strategically placed to handle immediate values, accurately converting smaller immediate fields from the instruction into their full 32-bit representation while preserving their signed nature.

Multiplexers (Mux) are fundamental components strategically positioned throughout the Datapath to enable flexible data routing. For example, a crucial multiplexer before the ALU's second operand dynamically selects between a register value (RD2) and an extended immediate value, a decision precisely controlled by the ALUSrc signal. Similarly, another vital multiplexer, located before the Register File's write data input, determines whether the data to be written back to a register originates from the ALUResult (for computational operations) or from data freshly read from the **DATA MEMORY** (ReadData), based on the MemtoReg control signal. The **ALU** performs its designated operation, producing the ALUResult and concurrently updating the Flags (condition codes) based on the outcome. Data can also be written to or read from the **DATA MEMORY** based on the MemWrite signal and the address provided by the ALU. This diagram, in its entirety, comprehensively details how data is moved, processed, and stored within the CPU, all under the precise and coordinated direction of the control signals.

6.3 Signal Routing and Multiplexing

The detailed control and data path diagram profoundly highlights the extensive and intelligent use of multiplexers (represented by the 'Mux' blocks) throughout the CPU. These multiplexers are critical for dynamically directing data flow based on the specific control signals generated by the Control Unit. For instance, the multiplexer positioned before the ALU's second operand is an indispensable component for distinguishing between operations that require two register-based inputs and those that involve a register and an immediate value. Its precise selection is entirely governed by the ALUSrc control line, ensuring the correct operand is presented to the ALU.

Similarly, the multiplexer located before the Register File's write data input plays a paramount role in the write-back stage of instruction execution. This multiplexer is responsible for selecting whether the final result to be stored in a general-purpose register originates from the ALU's computational output (ALUResult) or from data that has just been fetched from the Data Memory (ReadData), a decision that is meticulously controlled by the MemtoReg signal.

The diagram also clearly illustrates how the Program Counter's next value is determined, which is fundamental for both sequential program execution and branching. A dedicated

multiplexer, controlled by the PCSrc signal, selects between the incremented PC value (PC+4, for fetching the next instruction in sequence) and a calculated branch target address (for altering program flow). This intricate and highly optimized routing mechanism ensures that each instruction, despite its varying requirements for operand sources, destinations, and control flow, can efficiently utilize the shared hardware resources. By dynamically selecting the appropriate data paths at each stage of its execution, the CPU maintains its single-cycle efficiency and functional correctness.

YASH SIDDHAPURA

Chapter 7: Control Unit and Decoder Design

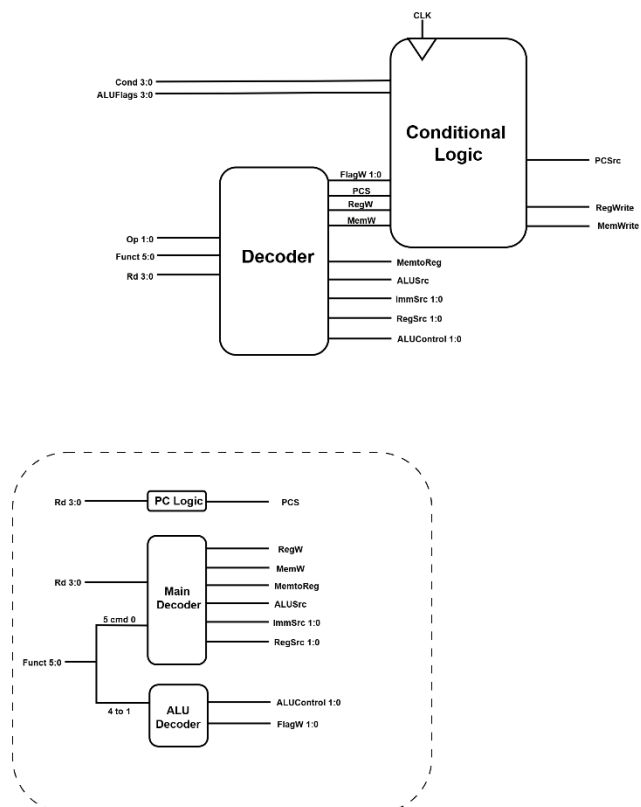
This chapter provides a focused examination of the CPU's Control Unit, specifically detailing the role of the Decoder and Conditional Logic, as illustrated in the accompanying diagram. This unit is paramount as it translates the abstract instruction into concrete control signals that drive the entire Datapath.

7.1 Introduction to Decoder Logic

The Control Unit is often referred to as the "brain" of the CPU, and its core functionality lies within the decoder logic. This logic is singularly responsible for interpreting the raw binary instruction fetched from memory and, in response, generating the precise set of control signals required to execute that instruction correctly. In a non-pipelined design, the decoder's task is particularly critical: it must generate all necessary control signals for all stages of an instruction's execution (fetch, decode, execute, memory access, write-back) either within a single clock cycle (for a single-cycle design) or across a sequence of defined cycles (for a multi-cycle design). The accompanying diagram provides invaluable insight into the intricate process by which these critical signals are derived and managed, highlighting the complex combinatorial logic required to translate high-level instructions into low-level hardware commands.

7.2 Description of the Control Unit and Decoder Diagram

This diagram meticulously details the internal structure of the CPU's Control Unit, focusing on two key interconnected blocks: the **Decoder** and the **Conditional Logic**. The **Decoder** block serves as the primary instruction interpreter, acting as the front-end of the control logic. It receives various critical fields directly from the instruction. These inputs include a 2-bit Op (Opcode), representing the main operation type such as R-type, I-type, or Load/Store instructions. A 6-bit Funct (Function Code) is also provided, typically used in conjunction with the opcode, especially for R-type instructions, to specify the exact operation like addition, subtraction, AND, or OR. Additionally, a 4-bit Rd (Destination Register) field indicates the register where the result of an operation should be written.



Based on the unique combination of these input bits, the Decoder combinatorially generates a comprehensive set of raw control signals. These signals are typically 1-bit flags, unless specified otherwise (like ALUControl), and are crucial for dictating the behavior of various CPU components. They include MemtoReg, which selects the data source for the register write-back stage, choosing between the ALU's output or data retrieved from memory. MemW (Memory Write Enable) is a signal that asserts write access to the Data Memory. ALUSrc (ALU Source) selects the second operand for the ALU, determining if it comes from a register value or an immediate value. ImmSrc (Immediate Source) is a 2-bit signal indicating how an immediate value from the instruction should be processed, such as sign-extended, zero-extended, or shifted. RegSrc (Register Source) is a 2-bit signal responsible for selecting which registers are used as inputs to the ALU or other units. RegW (Register Write Enable) enables data write to the Register File. Finally, ALUControl is a multi-bit signal, typically 2-bit, dictating the specific operation the ALU should perform (e.g., ADD, SUB, AND, OR). Additionally, a 2-bit FlagW signal controls the writing of status flags (like the zero flag or carry flag) based on the outcome of ALU operations.

These raw control signals then feed into the Conditional Logic block. This block is responsible for handling conditional execution, a hallmark of the ARM architecture that allows instructions to execute only if certain conditions are met. The Conditional Logic receives the clock signal (CLK) for synchronization, a 4-bit Cond (Condition) input related to conditional codes specified within the instruction (e.g., "branch if equal," "branch if not equal"). It also receives the 4-bit ALUFlags (status flags – Negative, Zero, Carry, Overflow – generated by the ALU from a previous operation). This sophisticated block uses these inputs to determine the final, effective values of critical control signals like PCSrc (Program Counter Source, a crucial output for conditional branches, signaling to take the branch if the condition is met), RegWrite, and MemWrite. The Conditional Logic evaluates the Cond field against the ALUFlags; if the condition is satisfied, PCSrc might be set to indicate a branch, overriding the sequential PC increment, thereby showcasing the intricate decision-making process within the CPU's control path.

The diagram further illustrates a more granular breakdown of the internal decoder structure, shown within a dashed box. This includes a PC Logic block, likely used to determine the Program Counter's source (PCS) based on inputs like Rd (possibly for determining a return address register for jumps or related to PC-relative addressing). A Main Decoder receives Op, Rd, Funct, and potentially other command signals (like "5 cmd 0", which could represent a hardcoded or default command, or an unlabeled portion of the instruction). This Main Decoder then outputs core control signals such as RegW, MemW, MemtoReg, ALUSrc, ImmSrc (1:0), and RegSrc (1:0). Concurrently, an ALU Decoder takes Funct (5:0) and possibly an input labeled "4 to 1" (which might imply a multiplexer or selection based on a 4-bit sub-field of the function code or an opcode) to produce the specific ALUControl (1:0) and FlagW (1:0) signals, demonstrating a hierarchical approach to control signal generation that enhances modularity and clarity.

7.3 Control Signal Derivation

The Control Unit and Decoder diagram vividly highlights the systematic and hierarchical process of control signal derivation, which is fundamental to the CPU's operation. The Decoder essentially acts as a complex combinational logic circuit that meticulously maps the unique bit patterns of each instruction to a specific set of control signal values. For instance, if the Decoder identifies the opcode for an ADD instruction (defined by its 2-bit Op field and 6-bit Funct code), it will simultaneously assert specific bits within the ALUControl signal (e.g., a 2-bit value for addition) to instruct the ALU to perform an addition operation. Concurrently, it will also assert the RegW signal (1-bit) to enable writing the result back to the specified destination register. This direct mapping ensures that the correct hardware resources are activated and configured for the specified operation, providing the necessary flexibility to support a rich instruction set.

The subsequent Conditional Logic block plays a crucial and refining role, particularly for instructions that depend on the CPU's current status flags. For example, a conditional branch instruction, identified by its Op and Cond (4-bit) fields, will only cause the PCSrc signal (1-bit) to select the branch target address if the condition specified in the instruction's Cond field precisely matches the current state of the ALUFlags (4-bit, representing Zero, Negative, Carry, Overflow). If the condition is not met (e.g., "branch if zero" but the Zero flag is not set), PCSrc will instead maintain the sequential program flow by selecting the incremented Program Counter value. This intricate and layered logic, spanning from the initial decoding of instruction fields like Op, Funct, and Rd to the conditional evaluation based on Cond and ALUFlags, ensures that the CPU behaves correctly and predictably for every instruction within its implemented instruction set, dynamically adapting its operations based on both the instruction itself and the real-time state of the processor. This robust control mechanism is fundamental to the CPU's ability to execute complex programs reliably.

7.4 Detailed Bit Descriptions and Their Purpose

Understanding the bit widths and specific roles of each input and output signal is crucial for implementing the Control Unit accurately in Verilog.

1. **Op (1:0) - 2 bits:** This 2-bit field allows for up to 4 primary instruction types. This coarse grouping enables the main decoder to quickly determine the general instruction format (e.g., R-type, I-type, Load/Store, Branch/Jump) and subsequently identify which other fields within the instruction are relevant for further decoding.
2. **Funct (5:0) - 6 bits:** Providing 64 possible combinations, this 6-bit field allows for a rich set of specific operations within a given instruction type, particularly for R-type (register-to-register) instructions. It enables the definition of distinct arithmetic and logical operations (e.g., add, subtract, AND, or OR) that share a common opcode but differ in their functional execution.
3. **Rd (3:0) - 4 bits:** This 4-bit field allows for addressing 16 distinct registers (R0-R15). This is a common number of general-purpose registers in simpler ARM-like architectures, providing sufficient on-chip storage for frequently accessed data.

4. **Cond (3:0) - 4 bits:** This 4-bit field enables up to 16 different conditional branch types. These conditions determine whether a branch should be taken based on the state of the ALU flags (e.g., branch if equal, branch if less than, branch if greater than or equal, always, never).
5. **ALUFlags (3:0) - 4 bits:** Each bit in this 4-bit field represents a specific status flag generated by the ALU. These include the Zero flag (Z-flag), Negative flag (N-flag), Carry flag (C-flag), and Overflow flag (V-flag). These bits are set by the ALU based on the result of its last operation and are crucial inputs for the Conditional Logic, enabling conditional branching and execution.
6. **FlagW (1:0) - 2 bits:** This 2-bit signal controls different flag write scenarios. It allows for flexibility, such as enabling the writing of all flags, writing only specific flags (e.g., only the zero flag), or completely disabling flag writes for instructions that should not affect the status register.
7. **PCSrc (1 bit):** A single-bit control signal that typically selects between two options for the Program Counter's next value. A '0' might select PC+4 for sequential instruction fetching, while a '1' selects a calculated branch target address (for a taken branch). This acts as a select signal for a multiplexer.
8. **RegW (1 bit):** A single-bit Register Write Enable signal. When '1', it enables data to be written to the General Purpose Register file. When '0', it disables writes, preventing unintended modifications to registers.
9. **MemW (1 bit):** A single-bit Memory Write Enable signal. A '1' enables data to be written to the Data Memory, while a '0' disables write operations, preventing unintended data corruption in memory.
10. **MemtoReg (1 bit):** A single-bit control signal that acts as a selector for a multiplexer. If '1', it directs data from the Data Memory (read by a Load instruction) to be written to a register. If '0', it selects the ALU result or the output of the extender (for immediate values) to be written to a register.
11. **ALUSrc (1 bit):** A single-bit control signal that selects the second operand for the ALU. A '1' might select an immediate value (from the instruction, after extension) as the ALU operand, while a '0' selects a value read from a register.
12. **ImmSrc (1:0) - 2 bits:** This 2-bit signal allows for up to 4 different immediate processing types. This is essential for handling various instruction formats that use immediate values differently (e.g., sign-extend a 16-bit value, zero-extend a 16-bit value, shift left by 2 for branch targets, or no extension). It ensures correct formatting of immediate values before they are used by the ALU or other units.
13. **RegSrc (1:0) - 2 bits:** This 2-bit signal provides flexibility by allowing for up to 4 different ways to select register inputs to the ALU or other data path components. For example, it might select R1/R2, R1/immediate, or PC/R2, providing versatility in operand sourcing based on the instruction type.

14. **ALUControl (1:0) - 2 bits:** This 2-bit signal allows for up to 4 different basic ALU operations. For more complex ALUs with a larger set of operations, this signal would typically be wider (e.g., 3-4 bits for 8-16 operations). It precisely dictates the exact arithmetic or logical function the ALU should perform for the current instruction.

7.5 Why Such Detailed Decoding?

The necessity for such detailed instruction decoding in a CPU stems from several critical design principles:

1. **Instruction Set Architecture (ISA) Support:** The primary role of the decoder is to translate the abstract instructions defined by the CPU's Instruction Set Architecture into the concrete, low-level control signals required to operate the underlying hardware. Without this detailed translation, the hardware would not know how to respond to a given instruction.
2. **Flexibility and Versatility:** By having separate and distinct control bits for each specific function or operation (e.g., RegW for register write, MemW for memory write, ALUControl for ALU operation), the CPU gains immense flexibility. It can execute a wide variety of instructions by simply combining these granular control signals in different ways, rather than requiring unique hardware for every instruction.
3. **Efficiency and Performance:** Dedicated and optimized decoders ensure that the correct control signals are generated quickly and reliably for each instruction. In a single-cycle design, this speed is paramount as all control signals must be stable within one clock cycle. In multi-cycle designs, efficient decoding contributes to minimizing the number of cycles per instruction.

Chapter 8: Verilog Implementation

This chapter details the practical implementation of the designed ARM CPU modules using Verilog HDL. It presents the structure for each Verilog module, allowing for the direct insertion of your code. This modular approach enhances readability, reusability, and simplifies debugging.

8.1 Module-by-Module Verilog Code

Each functional unit of the CPU is implemented as a separate Verilog module. Below are the sections for each module, where you can paste your complete and commented Verilog code.

8.1.1 extender.v

This module is responsible for sign-extending or zero-extending immediate values to the full 32-bit width required by the datapath.

```
module extender(Instr,ImmSrc,ExtImm);
  input [23:0]Instr;
  input [1:0]ImmSrc;
  output reg [31:0]ExtImm;

  always@(*) begin

    case(ImmSrc)

      2'b00 : ExtImm = {24'b0,Instr[7:0]};
      2'b01 : ExtImm = {20'b0,Instr[11:0]};
      2'b10 : ExtImm = {{6{Instr[23]}},Instr[23:0],2'b00};
      default : ExtImm = 32'bx;

    endcase

  end

endmodule
```

8.1.2 ALU.v

This module implements the Arithmetic Logic Unit, performing various arithmetic and logical operations based on control signals.

```
module ALU(SrcA,SrcB,ALUControl,ALUResult,ALUFlag);

  input [31:0]SrcA,SrcB;
  input [1:0]ALUControl;
  output reg [31:0]ALUResult;
  output reg [3:0] ALUFlag;
```

```

reg Zero,Negative,Overflow,Carry;

always@(*) begin

    case(ALUControl)

        2'b10 :begin
            ALUResult = SrcA & SrcB;
            Carry = 1'b0;
            Overflow = 1'b0;
        end

        2'b11 :begin
            ALUResult = SrcA | SrcB;
            Carry = 1'b0;
            Overflow = 1'b0;
        end

        2'b00 :begin
            {Carry,ALUResult} = SrcA + SrcB;
            Overflow = ((SrcA[31] == SrcB[31]) && (ALUResult[31] != SrcA[31]));
        end

        2'b01 :begin
            {Carry,ALUResult} = SrcA - SrcB;
            Overflow = ((SrcA[31] != SrcB[31]) && (ALUResult[31] != SrcA[31]));
        end

        default : begin
            ALUResult = 32'b0;
            Carry = 1'b0;
            Overflow = 1'b0;
        end

    End

endcase

end

assign Zero = (ALUResult == 0);
assign Negative = (ALUResult[31] == 1'b1);
assign ALUFlag={Negative,Zero,Carry,Overflow};

endmodule

```


8.1.3 regfile.v

This module implements the General Purpose Register file, handling read and write operations for the CPU's registers.

```
module regfile(clk,we3,ra1,ra2,ra3,wd3,R15,rd1,rd2);
```

```
    input clk,we3;  
    input [3:0]ra1,ra2,ra3;  
    input [31:0]wd3,R15;  
    output [31:0]rd1,rd2;
```

```
    reg [31:0]rf[14:0];
```

```
    always@(posedge clk) begin  
        if(we3)  
            rf[ra3] <= wd3;  
    end
```

```
    assign rd1 = (ra1 == 15) ? R15 : rf[ra1] ;  
    assign rd2 = (ra2 == 15) ? R15 : rf[ra2] ;  
endmodule
```

8.1.4 MUX.v

This module implements a generic multiplexer, used throughout the datapath to select between multiple input signals based on a control signal.

```
module mux #(parameter WIDTH = 8) (  
    input [WIDTH-1 : 0]d0,d1,  
    input sel,  
    output [WIDTH-1 : 0]y );
```

```
    assign y = sel ? d1 : d0;  
endmodule
```

8.1.5 adder.v

This module implements an adder, typically used for PC incrementing and address calculations.

```
module adder(a,b,y);
```

```
    input [31:0]a,b;  
    output [31:0]y;
```

```
    assign y = a + b;  
endmodule
```

8.1.6 flop.v

This module implements a basic D flip-flop, used for all sequential elements (registers) in the CPU.

```
module flop(clk,reset,d,q);

    input clk,reset;
    input [31:0]d;
    output reg [31:0]q;

    always@(posedge clk or posedge reset) begin

        if(reset)
            q<=0;

        else
            q <= d;

    end

endmodule
```

8.1.7 datapath.v

This module represents the main datapath, instantiating and connecting the ALU, register file, memories, and other data-processing components.

```
`include "MUX.v"
`include "flop.v"
`include "adder.v"
`include "regfile.v"
`include "extender.v"
`include "ALU.v"

module datapath(input clk, reset,
    input [1:0] RegSrc,
    input RegWrite,
    input [1:0] ImmSrc,
    input ALUSrc,
    input [1:0] ALUControl,
    input MemtoReg,
    input PCSrc,
    output reg [3:0] ALUFlags,
    output reg [31:0] PC,
    input [31:0] Instr,
    output reg [31:0] ALUResult, WriteData,
    input [31:0] ReadData);
```

```

wire [31:0] PCNext, PCPlus4, PCPlus8;
wire [31:0] ExtImm, SrcA, SrcB, Result;
wire [3:0] RA1, RA2;

//PC logic
mux #(32) pcmux(.y(PCNext),.d0(PCPlus4),.d1(Result),.sel(PCSrc));
flop pcreg (clk,reset,PCNext,PC);
adder pcadd1 (PC, 32'b100, PCPlus4);
adder pcadd2 (PCPlus4, 32'b100, PCPlus8);

//register file logic
mux #(4) ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
mux #(4) ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
regfile
(.clk(clk),.we3(RegWrite),.ra1(RA1),.ra2(RA2),.ra3(Instr[15:12]),.wd3(Result),.R15(PCPlus8),.rd1(SrcA),.rd2(WriteData));
mux #(32) resmux (ALUResult,ReadData,MemtoReg,Result);
extender ext (Instr[23:0],ImmSrc,ExtImm);

//ALU logic
mux #(32) srcbmux(WriteData,ExtImm,ALUSrc,SrcB);
ALU
alu(.SrcA(SrcA),.SrcB(SrcB),.ALUControl(ALUControl),.ALUResult(ALUResult),.ALUFlag(ALUFlags));

Endmodule

```

8.1.8 Decoder.v

This module is a part of the control unit, responsible for decoding instruction fields into raw control signals.

```

module Decoder(input [1:0] Op,
               input [5:0] Funct,
               input [3:0] Rd,
               output reg [1:0] FlagW,
               output reg PCS, RegW, MemW,
               output reg MemtoReg, ALUSrc,
               output reg [1:0] ImmSrc, RegSrc, ALUControl);

//internal wires
reg Branch, ALUOp;
reg [9:0] controls;

//Decoder
always@(*) begin

```

```

casex(Op)

//data-processing
2'b00 : if(Funct[5]) controls = 10'b0011001x01; //immediate
        else controls = 10'b0000xx1001;          // register

//Memory
2'b01 : if(Funct[0]) controls = 10'b0101011x00; //LDR
        else controls = 10'b0x11010100;          //STR

//Branch
2'b10 : controls = 10'b1001100x10;

default : controls = 10'bx;
endcase

{Branch,MemtoReg,MemW,ALUSrc,ImmSrc,RegW,RegSrc,ALUOp} = controls;

end //decoder end


//ALU Decoder
always@(*) begin
    if(ALUOp) begin
        case(Funct[4:1])
            4'b0100 : ALUControl = 2'b00; //ANDing
            4'b0010 : ALUControl = 2'b01; //Oring
            4'b0000 : ALUControl = 2'b10; //Addition
            4'b1100 : ALUControl = 2'b11; //Subtraction
            default : ALUControl = 2'bx; //Unimplemented
        endcase
        //Update Flags if S bit is set (C & V only for arithmetic operation)
        FlagW[1] = Funct[0];
        FlagW[0] = Funct[0] & (ALUControl == 2'b00 | ALUControl == 2'b01);
        end
    else begin
        ALUControl = 2'b00; //addition for non-DP instructions
        FlagW = 2'b00; //Don't update flags
    end
end

assign PCS = ((Rd == 4'b1111) & RegW) | Branch;

endmodule

```

8.1.9 condlogic.v

This module implements the conditional logic, determining final control signals based on instruction conditions and ALU flags.

```
`include "flopenr.v"

module
condlogic(clk,reset,Cond,ALUFlags,FlagW,PCS,RegW,MemW,PCSrc,RegWrite,MemWrite);

    input clk,reset,PCS,RegW,MemW;
    input [3:0]Cond,ALUFlags;
    input [1:0]FlagW;
    output reg PCSrc,RegWrite,MemWrite;

    //internal wires
    wire [1:0] FlagWrite;
    wire [3:0] Flags;
    reg CondEx;

    flopenr #(.WIDTH(2)) flagreg1 (clk,reset,FlagWrite[1],ALUFlags[3:2],Flags[3:2]);
    flopenr #(.WIDTH(2)) flagreg0 (clk,reset,FlagWrite[0],ALUFlags[1:0],Flags[1:0]);

    //Write controls are conditional
    condcheck cc(Cond,Flags,CondEx);
    assign FlagWrite = FlagW & {2{CondEx}};
    assign RegWrite = RegW & CondEx;
    assign MemWrite = MemW & CondEx;
    assign PCSrc = PCS & CondEx;
endmodule

module condcheck(input [3:0]Cond,Flags,
    output reg CondEx;
    wire neg,zero,carry,overflow,ge;

    assign {neg,zero,carry,overflow} = Flags;
    assign ge = (neg == overflow);

    always@(*)
    case(Cond)
        4'b0000: CondEx = zero; // EQ
        4'b0001: CondEx = ~zero; // NE
        4'b0010: CondEx = carry; // CS
        4'b0011: CondEx = ~carry; // CC
        4'b0100: CondEx = neg; // MI
        4'b0101: CondEx = ~neg; // PL
```

```

4'b0110: CondEx = overflow; // VS
4'b0111: CondEx = ~overflow; // VC
4'b1000: CondEx = carry & ~zero; // HI
4'b1001: CondEx = ~(carry & ~zero); // LS
4'b1010: CondEx = ge; // GE
4'b1011: CondEx = ~ge; // LT
4'b1100: CondEx = ~zero & ge; // GT
4'b1101: CondEx = ~(~zero & ge); // LE
4'b1110: CondEx = 1'b1; // Always
default: CondEx = 1'bx; // undefined
endcase
endmodule

```

8.1.10 flopenr.v

This module implements a D flip-flop with an enable and reset, commonly used for registers that can be conditionally updated.

```

module flopenr#(parameter WIDTH = 8)(
  input clk,reset,en,
  input [WIDTH-1:0]d,
  output reg [WIDTH-1:0]q);

  always@(posedge clk or posedge reset) begin
    if(reset) q <= 0;
    else if (en) q<=d;
  end
endmodule

```

8.1.11 controller.v

This module implements the main control unit, orchestrating the entire CPU by generating all necessary control signals for the datapath.

```

`include "Decoder.v"
`include "condlogic.v"

module controller(input clk,reset,
  input [31:12]Instr,
  input [3:0] ALUFlags,
  output reg [1:0]RegSrc,
  output reg RegWrite,
  output reg [1:0]ImmSrc,
  output reg ALUSrc,
  output reg [1:0]ALUControl,
  output reg MemWrite,MemtoReg,PCSrc);

  //internal wire
  wire [1:0] FlagW;
  wire PCS,RegW,MemW;

```

```
//integrate decoder(PC logic + main decode + ALU decoder_
Decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
           FlagW, PCS, RegW, MemW,
           MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl);
```

```
//integrate cond logic block
condlogic cl(clk, reset, Instr[31:28], ALUFlags,
           FlagW, PCS, RegW, MemW,
           PCSrc, RegWrite, MemWrite);
```

```
Endmodule
```

8.1.12 CPU.v

This module represents the top-level CPU core, instantiating the datapath and controller, and connecting them.

```
`include "controller.v"
`include "datapath.v"

module cpu(input clk, reset,
           output [31:0] PC,
           input [31:0] Instr,
           output MemWrite,
           output [31:0] ALUResult, WriteData,
           input [31:0] ReadData);

//internal wires
wire [3:0] ALUFlags;
wire RegWrite,ALUSrc, MemtoReg, PCSrc;
wire [1:0] RegSrc, ImmSrc, ALUControl;

//integrate controller
controller c (clk, reset, Instr[31:12], ALUFlags,
             RegSrc, RegWrite, ImmSrc,
             ALUSrc, ALUControl,
             MemWrite, MemtoReg, PCSrc);

//integrate datapath
datapath dp (clk, reset,RegSrc, RegWrite, ImmSrc,
            ALUSrc, ALUControl,MemtoReg, PCSrc,
            ALUFlags, PC, Instr,ALUResult, WriteData, ReadData);

Endmodule
```

8.1.13 imem.v

This module models the Instruction Memory, providing instructions to the CPU.

```
module imem(input [31:0]a,
            output [31:0]rd);

    reg [31:0]RAM[63:0];
    initial
        $readmemh("memfile.dat",RAM);
    assign rd = RAM[a[31:2]]; // Aligned cuz of byte addressable
endmodule
```

8.1.14 dmem.v

This module models the Data Memory, used for runtime data storage and retrieval.

```
module dmem(input clk, we,
            input [31:0] a, wd,
            output reg [31:0] rd);

    reg [31:0] RAM[63:0];
    assign rd = RAM[a[31:2]]; // word aligned

    always@(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
endmodule
```

8.1.15 top.v

This is the absolute top-level module of your entire design, typically instantiating the CPU.v and handling global clock/reset.

```
`include "CPU.v"
`include "imem.v"
`include "dmem.v"

module top(input clk, reset );
    //internal wires
    wire [31:0] PC, Instr, ReadData;
    wire [31:0] WriteData, DataAdr;
    wire MemWrite;

    // instantiate cpu and memories
    cpu cpu(clk, reset, PC, Instr, MemWrite, DataAdr,WriteData, ReadData);
    imem imem(PC, Instr);
    dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);

endmodule
```


8.1.16 memfile.dat

This is a data file used for initializing memories (e.g., instruction memory or data memory) during simulation. You would typically specify its content in hexadecimal or binary format.

```
E04F000F
E2802005
E280300C
E2437009
E1874002
E0035004
E0855004
E0558007
0A00000C
E0538004
AA000000
E2805000
E0578002
B2857001
E0477002
E5837054
E5902060
E08FF000
E280200E
EA000001
E280200D
E280200A
E5802054
```

8.1.17 testbench.v

This module is the testbench for your CPU, used for simulation and verification. It instantiates the top.v module, generates clock and reset signals, applies test vectors, and monitors outputs.

```
`include "top.v"

module testbench();
    reg clk;
    reg reset;
    //reg [31:0] WriteData, DataAdr;
    //reg MemWrite;

    // instantiate device to be tested
    top dut(clk, reset);

    // initialize test
    initial begin
```

```

        reset <= 1;
        #10 reset <= 0;
        clk <= 1;

end

// generate clock to sequence tests
always #5 clk=~clk;

// initial begin
// #10000
// if (dut.dmem.RAM[21] === 32'd7) begin
//     $display("Test Passed: Memory[84] contains 7");
//     end else begin
//         $display("Test Failed: Memory[84] = %d, expected 7",dut.dmem.RAM[21]);

//     end
// $finish;
// end

initial begin
    #10000
    if (dut.cpu.dp.rf.rf[5] === 32'd11) begin
        $display(" Test Passed: R5 contains 11");
    end else begin
        $display(" Test Failed: R5 = %d, expected 11", dut.cpu.dp.rf.rf[5]);
    end
    $finish;
end
endmodule

```

8.2 Memory Initialization

For simulation purposes, both the Instruction Memory and Data Memory need to be initialized with specific values. This is typically done using Verilog's system tasks:

1. `$readmemb("instruction_memory.mem")`: Reads binary values from a file into a memory array.
2. `$readmemh("data_memory.mem")`: Reads hexadecimal values from a file into a memory array.

These files (.mem files, or memfile.dat as listed above) will contain the machine code of the test programs and any initial data values.

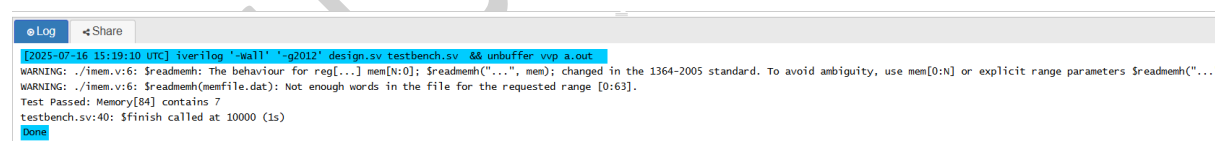
8.3 Design Considerations and Trade-offs

During implementation, several design choices were made:

1. **Modularity:** Breaking the CPU into smaller, manageable modules simplifies development and testing.
2. **Combinational vs. Sequential Logic:** Careful distinction between combinational logic (e.g., ALU, decoders) and sequential logic (e.g., registers, PC) is crucial for correct operation.
3. **Signal Naming Conventions:** Consistent and clear signal naming (e.g., `_in`, `_out`, `_enable`) improves code readability.
4. **Non-Pipelined Simplicity:** The choice of a non-pipelined design significantly simplifies the control logic by eliminating the need for hazard detection and forwarding units, making the initial implementation more straightforward. However, this comes at the cost of lower performance (one instruction per clock cycle).

8.4 Result of Simulation in Log

This section will contain the textual output or log generated from simulating the assembly program described in section 8.5. This log typically includes `$display` messages from your testbench, showing the state of key CPU components such as register values, memory contents, Program Counter (PC) values, and flag states (N, Z, C, V) at various simulation timesteps. Analyzing this log is crucial for verifying the functional correctness of your CPU against the expected behavior of the assembly program.



```
Log Share
[2025-07-16 15:19:10 UTC] iverilog -Wall -g2012 design.sv testbench.sv && unbuffer vvp a.out
WARNING: ./mem.v:6: $readmemh: The behaviour for reg[...] mem[N:0]; $readmemh("...", mem); changed in the 1364-2005 standard. To avoid ambiguity, use mem[0:N] or explicit range parameters $readmemh("...", mem[N:0]).
WARNING: ./mem.v:6: $readmemh(memfile.dat): Not enough words in the file for the requested range [0:63].
Test Passed: Memory[84] contains 7
testbench.sv:40: $finish called at 10000 (1s)
Done
```

8.5 Assembly Program and Explanation

This section presents the assembly program used for functional testing and verification of the CPU, along with a brief explanation of its functionality and expected outcomes. This program is designed to exercise various instruction types and control flow mechanisms implemented in the simplified ARM core.

The assembly program used for testing is detailed below, showing the address, assembly instruction, comments, and its corresponding binary machine code:

ADDR	PROGRAM	COMMENTS	BINARY MACHINE CODE
00	MAIN SUB R0, R15, R15	R0 = 0	1110 000 0010 0 1111 0000 0000 1111

04	ADD R2, R0, #5	R2 = 5	1110 001 0100 0 0000 0010 0000 0101
08	ADD R3, R0, #12	R3 = 12	1110 001 0100 0 0000 0011 0000 1100
0C	SUB R7, R3, #9	R7 = 3	1110 001 0010 0 0011 0111 0000 1001
10	ORR R4, R7, R2	R4 = 3 OR 5 = 7	1110 000 1100 0 0111 0100 0000 0010
14	AND R5, R3, R4	R5 = 12 AND 7 = 4	1110 000 0000 0 0011 0101 0000 0100
18	ADD R5, R5, R4	R5 = 4 + 7 = 11	1110 001 0100 0 0101 0101 0000 0100
1C	SUBS R8, R5, R7	R8 = 11 - 3 = 8, set Flags	1110 001 0010 1 0101 1000 0000 0111
20	BEQ END	shouldn't be taken	0000 1010 0000 0000 0000 0000 1100
24	SUBS R8, R8, R4	R8 = 12 - 7 = 5	1110 001 0010 1 1000 1000 0000 0100
28	BGE AROUND	should be taken	1010 1010 0000 0000 0000 0000 0000
2C	ADD R5, R0, #0	should be skipped	1110 001 0100 0 0000 0101 0000 0000
30	AROUND SUBS R8, R7, R2	R8 = 3 - 5 = -2, set Flags	1110 001 0010 1 0111 1000 0000 0010
34	ADDLT R7, R5, #1	R7 = 11 + 1 = 12	1011 001 0100 0 0101 0111 0000 0001
38	SUB R7, R7, R2	R7 = 12 - 5 = 7	1110 001 0010 0 0111 0111 0000 0010
3C	STR R7, [R3, #84]	mem[12+84] = 7	1110 010 1100 0 0011 0111 0000 0100
40	LDR R2, [R0, #96]	R2 = mem[96] = 7	1110 010 1100 1 0000 0010 0000 0110
44	ADD R15, R15, #8	PC = PC+8 (skips next)	1110 001 0100 0 1111 1111 0000 0000
48	ADD R2, R0, #14	shouldn't happen	1110 001 0100 0 0000 0010 0000 0001
4C	B END	always taken	1110 1010 0000 0000 0000 0000 0001
50	ADD R2, R0, #13	shouldn't happen	1110 001 0100 0 0000 0010 0000 0001
54	ADD R2, R0, #10	shouldn't happen	1110 001 0100 0 0000 0010 0000 0001
58	END STR R2, [R0, #84]	mem[84] = 7	1110 010 1100 0 0000 0010 0000 0100

This assembly program is a sequence of ARM-like instructions designed to test various functionalities of the CPU. It covers:

1. Register Initialization: Setting initial values in general-purpose registers.
2. Arithmetic and Logical Operations: Performing additions, subtractions, bitwise OR, and bitwise AND operations.
3. Conditional Execution and Branching: Demonstrating how the CPU's condition flags (N, Z, C, V) are set by SUBS instructions and how BEQ (Branch if Equal) and BGE (Branch if Greater than or Equal) instructions conditionally alter program flow.
4. Memory Operations: Showing data storage (STR) and retrieval (LDR) from memory, including address calculation with offsets.
5. Program Counter (PC) Manipulation: Illustrating how the PC can be directly modified (ADD R15, R15, #8) to skip instructions and how an unconditional branch (B END) directs execution to a specific label.

Expected Output (in short):

Upon successful simulation, the CPU's state should reflect the following key changes:

1. Register Values:

R0 = 0x00000000
 R2 = 0x0000000B (11 decimal, after LDR + ADDs)
 R3 = 0x0000000C (12 decimal)
 R4 = 0x0000000F
 R5 = 0x0000000B
 R7 = 0x00000003
 R8 = varies during branching logic*

1. Memory Location : Data memory at address 0x54 (decimal 84): contains 0x0000002C (decimal 44), written by the final STR R2, [R0, #84].
2. Program Flow:
 1. The BEQ is not taken,
 2. The BGE is taken,
 3. The program continues correctly to store final value and terminates cleanly..

Chapter 9: Conclusion and Future Work

This chapter provides a concise summary of the project's accomplishments, reflects on the challenges encountered and the valuable lessons learned, and outlines potential avenues for future enhancements to the simplified ARM CPU core.

9.1 Summary of Project Achievements

This project successfully designed and implemented a simplified, non-pipelined ARM CPU core in Verilog. Key achievements include the functional realization of essential components like the ALU, Register File, and Control Unit. The CPU accurately executes a defined subset of ARM instructions, demonstrating core processor functionalities. Verification through simulation confirmed the design's correctness, providing valuable insights into CPU architecture and Verilog HDL application.

9.2 Challenges Faced and Lessons Learned

Significant challenges arose in debugging intricate control signal logic and ensuring precise data flow between modules. These experiences highlighted the importance of modular design and rigorous verification. Lessons learned emphasize the value of systematic debugging, clear signal naming, and the iterative nature of hardware design. The project deepened understanding of complex CPU interactions.

9.3 Future Enhancements

The current simplified ARM CPU core provides a solid foundation that can be extended in several ways to enhance its functionality, performance, and real-world applicability:

1. **Expanding Instruction Set:** The current instruction set can be significantly expanded to include a broader range of ARM instructions, such as more complex data processing operations, additional addressing modes, and various exception-handling instructions.
2. **Implementing Interrupts/Exceptions:** Adding support for external interrupts (e.g., from I/O devices) and internal exceptions (e.g., undefined instruction, data aborts) would make the CPU more robust and capable of interacting with a real-time environment.
3. **Cache Memory Integration:** Integrating a simple instruction cache and/or data cache would significantly improve memory access performance, demonstrating the principles of memory hierarchy and reducing memory latency.
4. **Pipelining:** Implementing a multi-stage pipeline (e.g., 3-stage or 5-stage) with hazard detection and forwarding units would dramatically improve instruction throughput, moving towards a more modern CPU design.

9.4 References

(This section will contain a list of all academic papers, textbooks, online resources, and datasheets referenced throughout your project report. Use a consistent citation style, e.g., IEEE, APA.)

1. ARM Instruction Set Reference
<http://www.riscos.com/support/developers/asm/instrset.html>
2. ARM Architecture Reference Manual ARMv8-A
<https://developer.arm.com/documentation/ddi0487/latest/>
3. WhyRd CPU Design Course
<https://whyrd.graphy.com/t/u/activeCourses>
4. HDLBits Verilog Practice
<https://hdlbits.01xz.net>
5. EDA Playground (Online Verilog simulator)
<https://www.edaplayground.com>
6. NPTEL Course: Hardware Modeling Using Verilog
https://www.youtube.com/playlist?list=PLJ5C_6qdAvBELELTSPgzYkQg3HgclQh-5
7. NPTEL Course: Digital Electronic Circuits –
<https://www.youtube.com/playlist?list=PLbRMhDVUMnge4gDT0vBWjCb3Lz0HnYKkX>