4.1 Introduction

> *"* In a major matter, no details are small.
> *French Proverb*

COD Chapter 1 (Computer Abstractions and Technology) explains that the performance of a computer is determined by three key factors: instruction count, clock cycle time, and *clock cycles per instruction* (CPI). COD Chapter 2 (Instructions: Language of the Computer) explains that the compiler and the instruction set architecture determine the instruction count required for a given program. However, the implementation of the processor determines both the clock cycle time and the number of clock cycles per instruction. In this chapter, we construct the datapath and control unit for two different implementations of the LEGv8 instruction set.

This chapter contains an explanation of the principles and techniques used in implementing a processor, starting with a highly abstract and simplified overview in this section. It is followed by a section that builds up a datapath and constructs a simple version of a processor sufficient to implement an instruction set like LEGv8. The bulk of the chapter covers a more realistic **pipelined** LEGv8 implementation, followed by a section that develops the concepts necessary to implement more complex instruction sets, like the x86.

For the reader interested in understanding the high-level interpretation of instructions and its impact on program performance, this initial section and COD Section 4.5 (An overview of pipelining) present the basic concepts of pipelining. Current trends are covered in COD Section 4.10 (Parallelism via instructions), and COD Section 4.11 (Real stuff: The ARM Cortex-A53 and Intel Core i7 pipelines) describes the recent Intel Core i7 and ARM Cortex-A53 architectures. COD Section 4.12 (Going Faster: Instruction-level parallelism and matrix multiply) shows how to use instruction-level parallelism to more than double the performance of the matrix multiply from COD Section 3.9 (Going Faster: Subword parallelism and matrix multiply). These sections provide enough background to understand the pipeline concepts at a high level.

For the reader interested in understanding the processor and its performance in more depth, COD Sections 4.3 (Building a datapath), 4.4 (A simple implementation scheme), and 4.6 (Pipelined datapath and control) will be useful. Those interested in learning how to build a processor should also cover COD Sections 4.2 (Logic design conventions), 4.7 (Data hazards: Forwarding versus stalling), 4.8 (Control hazards), and 4.9 (Exceptions). For readers with an interest in modern hardware design, COD Section 4.13 (An introduction to digital design using a hardware design language ...) describes how hardware design languages and CAD tools are used to implement hardware, and then how to use a hardware design language to describe a pipelined implementation. It also gives several more illustrations of how pipelining hardware executes. (The hardware models in this chapter have been sourced by the authors and do not imply ARM-endorsed architectures.

A basic LEGv8 implementation

We will be examining an implementation that includes a subset of the core LEGv8 instruction set:

- The memory-reference instructions *load register unscaled* (`LDUR`) and *store register unscaled* (`STUR`)
- The arithmetic-logical instructions `ADD`, `SUB`, `AND`, and `ORR`
- The instructions *compare and branch on zero* (`CBZ`) and *branch* (`B`), which we add last

This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions. However, it illustrates the key principles used in creating a datapath and designing the control. The implementation of the remaining instructions is similar.

In examining the implementation, we will have the opportunity to see how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the computer. Many of the key design principles introduced in COD Chapter 1 (Computer Abstractions and Technology) can be illustrated by looking at the implementation, such as *Simplicity favors regularity*. In addition, most concepts used to implement the LEGv8 subset in this chapter are the same basic ideas that are used to construct a broad spectrum of computers, from high-performance servers to general-purpose microprocessors to embedded processors.

An overview of the implementation

In COD Chapter 2 (Instructions: Language of the Computer), we looked at the core LEGv8 instructions, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions. Much of what needs to be done to implement these instructions is the same, independent of the exact class of instruction. For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the `LDUR` and `CBZ` instructions, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction. The simplicity and regularity of the LEGv8 instruction set simplify the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes, except unconditional branch, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the operation execution, and conditional branches for comparison to zero. After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a conditional branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by four to get the address of the subsequent instruction.

PARTICIPATION ACTIVITY 4.1.1: Steps for LEGv8 instructions.

Consider the first three steps for a typical LEGv8 instruction.

Use ALU for comparisons **Use ALU for operation execution**

Fetch instruction from memory **Read register(s)** **Use ALU for address calculation**

1st step

2nd step

3rd step for memory-reference instructions

3rd step for arithmetic-logic instructions

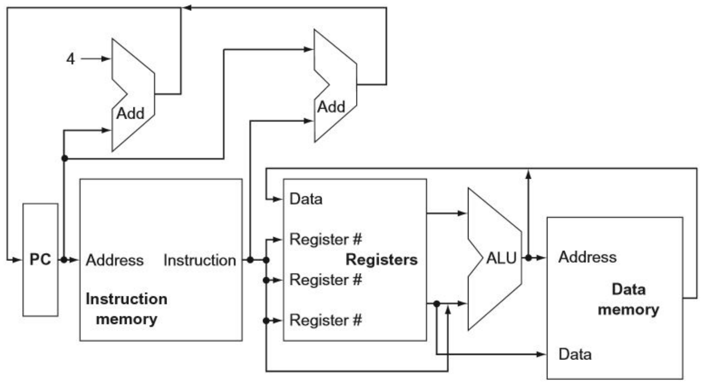
3rd step for branch instructions

Reset

The figure below shows the high-level view of a an LEGv8 implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.

Figure 4.1.1: An abstract view of the implementation of the LEGv8 subset showing the major functional units and the major connections between them (COD Figure 4.1).

All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare to zero (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch offset are summed) or from an adder that increments the current PC by four. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.



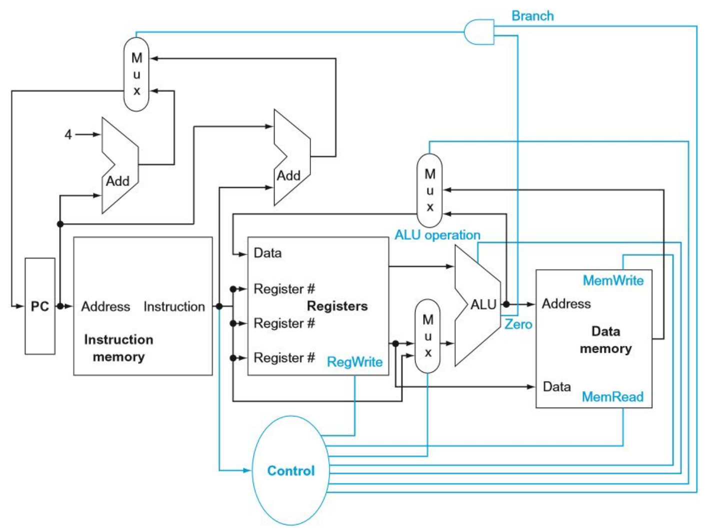
First, in several places, the figure above shows data going to a particular unit as coming from two different sources. For example, the value written into the PC can come from one of two adders, the data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate field of the instruction. In practice, these data lines cannot simply be wired together; we must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination. This selection is commonly done with a device called a *multiplexor*, although this device might better be called a *data selector*. COD Appendix A (The Basics of Logic Design) describes the multiplexor, which selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.

The second omission in the figure above is that several of the units must be controlled depending on the type of instruction. For example, the data memory must be read on a load and write on a store. The register file must be written only on a load or an arithmetic-logical instruction. And, of course, the ALU must perform one of several operations. (COD Appendix A (The Basics of Logic Design) describes the detailed design of the ALU.) Like the multiplexors, control lines that are set based on various fields in the instruction direct these operations.

The figure below shows the datapath of COD Figure 4.1 (An abstract view of the implementation of the LEGv8 subset ...) with the three required multiplexors added, as well as control lines for the major functional units. A *control unit*, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors. The top multiplexor, which determines whether PC + 4 or the branch destination address is written into the PC, is set based on the Zero output of the ALU, which is used to perform the comparison of a `CBZ` instruction. The regularity and simplicity of the LEGv8 instruction set mean that a simple decoding process can be used to determine how to set the control lines.

Figure 4.1.2: The basic implementation of the LEGv8 subset, including the necessary multiplexors and control lines (COD Figure 4.2).

The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottom-most multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.



In the remainder of the chapter, we refine this view to fill in the details, which requires that we add further functional units, increase the number of connections between units, and, of course, enhance a control unit to control what actions are taken for different instruction classes. COD Sections 4.3 (Building a datapath) and 4.4 (A simple implementation scheme) describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of the above figures. In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

While easier to understand, this approach is not practical, since the clock cycle must be severely stretched to accommodate the longest instruction. After designing the control for this simple computer, we will look at pipelined implementation with all its complexities, including exceptions.

PARTICIPATION ACTIVITY 4.1.2: Check yourself: Five classic components of a computer.

How many of the five classic components of a computer appear in the figure above (COD Figure 4.2 (The basic implementation of the LEGv8 subset ...))?

1) Input
- True
- False

2) Memory
- True
- False

3) Control
- True
- False

4) Datapath
- True
- False

5) Output
- True
- False

Provide feedback on this section