13 CENTRAL PROCESSING UNITS

13.1 INTRODUCTION

What to Expect

A CPU is one of the most important components in a computer, cell phone, or other "smart" digital device. CPUs provide the interface between software and hardware and can be found in any device that processes data, even automobiles and household appliances often contain a CPU. This chapter introduces CPUs from a digital logic perspective and touches on how those devices function.

- Describing the function of a CPU
- Designing a CPU
- Creating a CPU instruction set
- Deriving assembly and programming languages from an instruction set
- Describing CPU states

13.2 CENTRAL PROCESSING UNIT

13.2.1 *Introduction*

The CPU is the core of any computer, tablet, phone, or other computer-like device. While many definitions of CPU have been offered, many quite poetic (like the "heart" or "brain" of a computer); probably the best definition is that the CPU is the intersection of software and hardware. The CPU contains circuitry that converts the ones and zeros that are stored in memory (a "program") into controlling signals for hardware devices. The CPU retrieves and analyzes bytes that are contained in a program, turns on or off multiplexers and control buffers so data are moved to or from various devices in the computer, and permits humans to use a computer for intellectual work. In its simplest form, a CPU does nothing more than fetch a list of instructions from memory and then execute those instructions one at a time. This chapter explores CPUs from a theoretical perspective.

13.2.1.1 *Concepts*

A CPU processes a string of ones and zeros and uses the information in that binary code to enable/disable circuits or hardware devices in a computer system. For example, a CPU may execute a binary code and create electronic signals that first place data on the computer's data bus and then spins up the hard drive to store that data. As another example, perhaps the CPU detects a key press on a keyboard, transfers that key's code to memory and also sends a code to the monitor where specific pixels are activated to display the letter pressed.

Figure 13.1 illustrates a very simple circuit used to control the flow of data on a data bus.

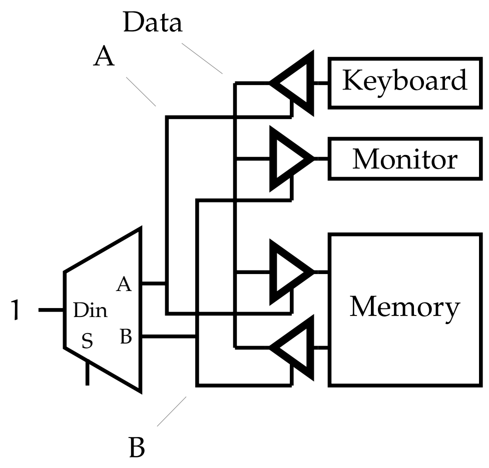

Figure 13.1: Simple Data Flow Control Circuit

In Figure 13.1, the demultiplexer at the bottom left corner of the circuit controls four control buffers that, in turn, control access to the data bus. When output A is active then input from the Keyboard is stored in Memory; but when output B is active then output from memory is sent to the monitor. By setting the select bit in the demultiplexer the circuit's function can be changed from reading the keyboard to writing to the monitor using a single data bus.

In a true CPU, of course, there are many more peripheral devices, along with an ALU, registers, and other internal resources to control. Figure 13.2 is a block diagram for a simplified CPU.

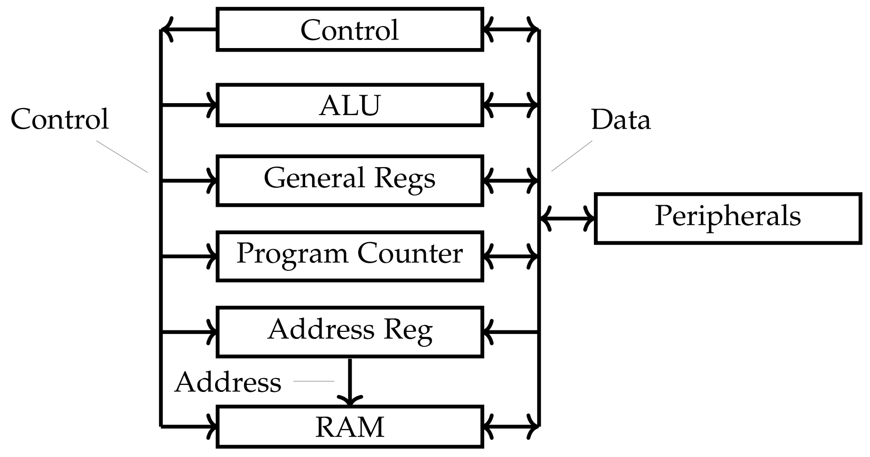

Figure 13.2: Simplified CPU Block Diagram

The CPU in Figure 13.2 has three bus lines:

- Control. This bus contains all of the signals needed to activate control buffers, multiplexers, and demultiplexers in order to move data.
- Data. This contains the data being manipulated by the CPU.
- Address. This is the address for the next instruction to fetch from RAM.

There are several blocks in the CPU in Figure 13.2:

- Control. This block contains the circuitry necessary to decode an instruction that was fetched from RAM and then activate the various devices needed to control the flow of data within the CPU.
- ALU. This is an ALU designed for the application that is using this CPU.
- General Registers. Most CPUs include a number of general registers that temporarily hold binary numbers or instructions.
- Program Counter. This is a register that contains the address of the next instruction to fetch from RAM.
- Address Register. This contains the address for the current RAM operation.
- RAM. While most computers and other devices have a large amount of RAM outside the CPU, many CPUs are constructed with a small amount of internal RAM for increased operational efficiency. This type of high-speed RAM is usually called cache (pronounced "cash").

In operation, the CPU moves the value of the Program Counter to the Address Register and then fetches the instruction contained at that RAM address. That instruction is then sent to the Control circuit where it is decoded. The Control circuit activates the appropriate control devices to execute the instruction. This process is repeated millions of times every second while the CPU executes a program.

13.2.1.2 *History*

CPUs from the early days of computing (circa 1950) were custom made for the computer on which they were found. Computers in those early days were rather rare and there was no need for a general-purpose CPU that could function on multiple platforms. By the 1960s *IBM* designed a family of computers based on the design of the *System/360*, or *S/360*. The goal was to have a number of computers use the same CPU so programs written for one computer could be executed on another in the same family. By doing this, *IBM* hoped to increase their customer loyalty.

CPUs today can be divided into two broad groups: Complex Instruction Set Computer (CISC) (pronounced like "sisk") and Reduced Instruction Set Computer (RISC). Early computers, like the IBM S/360 had a large, and constantly growing, set of instructions, and these types of CPUs were referred to as CISC. However, building the circuits needed to execute all of those instructions became ever more challenging until, in the late 1980s, computer scientists began to design RISC CPUs with fewer instructions. Because there were fewer CPU instructions circuits could be smaller and faster, but the trade off was that occasionally a desired instruction had to be simulated by combining two or more other instructions, and that creates longer, more complex computer programs. Nearly all computers, cell phones, tablets, and other computing devices in use today use a RISC architecture.

More recent developments in CPUs include "pipelining" where the CPU can execute two or more instructions simultaneously by overlapping them, that is, fetching and starting an instruction while concurrently finishing the previous instruction. Another innovation changed compilers such that they can create efficient Very Long Instruction Word (VLIW) codes that combine several instructions into a single step. Multi-threading CPUs permit multiple programs to execute simultaneously and multi-core CPUs use multiple CPU cores on the same substrate so programs can execute in parallel.

CPUs, indeed, all hardware devices, are normally designed using an Hardware Description Language (HDL) like Verilog to speed development and ensure high quality by using peer review before an IC is manufactured. It is possible to find Open Source Verilog scripts for many devices, including CPU cores[1], so designers can begin with

1 http://www.opencores.org

mature, working code and then "tweak" it as necessary to match their particular project.

13.2.1.3 *CPU Design Principles*

CPU design commonly follows these steps:

ANALYSIS OF INTENDED USE Designing a CPU starts with an analysis of its intended use since that will determine the type of CPU that must be built. A simple four-bit CPU — that is, all instructions are only four-bits wide — is more than adequate for a simple device like a microwave oven, but for a cell phone or other more complex device, a 16-bit, or larger, CPU is needed.

THE INSTRUCTION SET After the purpose of the CPU is defined then an instruction set is created. CPU instructions control the physical flow of bits through the CPU and various components of a computer. While an instruction set looks something like a programming language, it is important to keep in mind that an instruction set is very different from the more familiar higher-level programming languages like *Java* and C++.

CPU instructions are 16-bit (or larger) words that are conceptually divided into two sections: the operational code (*opcode*) and data. There are several classes of instructions, but three are the most common:

- R (REGISTER). These instructions involve some sort of register activity. The quintessential R-Type instruction is ADD, where the contents of two registers are added together and the sum is placed in another register.
- I (IMMEDIATE). These instructions include data as part of the instruction word and something happens immediately with that data. As an example, the LDI instruction immediately loads a number contained in the instruction word into a specified register.
- J (JUMP). These instructions cause the program flow to jump to a different location. In older procedural programming languages (like C and *Basic*) these were often called GOTO statements.

ASSEMBLY LANGUAGE A computer program is nothing more than a series of ones and zeros organized into words the bit-width of the instruction set, commonly 32-bit or 64-bit. Each word is a single instruction and a series of instructions forms a program in *machine code* that looks something like this:

```
0000000000000000        (13.1)
1001000100001010
1001001000001001
0111001100011000
0110000000100011
```

A CPU fetches and executes one 16-bit word of the machine code at a time. If a programmer could write machine code directly then the CPU could execute it without needing to compile it first. Of course, as it is easy to imagine, no one actually writes machine code due to its complexity.

The next level higher than machine code is called *Assembly*, which uses easy-to-remember abbreviations (called "mnemonics") to represent the available instructions. Following is the assembly language program for the machine code listed above:

Label	Mnemonic	Operands	Comment
START	NOP	0 0 0	No Operation
	LDI	1 0 a	R1 ¡- 0ah
	LDI	2 0 9	R2 ¡- 09h
	SHL	3 1 4	R3 ¡- R1 ¡¡ 8
	XOR	0 2 3	Acc ¡- R2 XOR R3

Each line of assembly has four parts. First is an optional *Label* that can be used to indicate various sections of the program and facilitates "jumps" around the program; second is the mnemonic for the code being executed; third are one or more operands; and fourth is an optional comment field.

Machine code is CPU specific so code written for one type of computer could not be used on any other type of computer.

Once the program has been written in Assembly, it must be "assembled" into machine code before it can be executed. An assembler is a fairly simply program that does little more than convert a file containing assembly code into instructions that can be executed by the CPU. The assembly program presented above would be assembled into Machine Code 13.1.

PROGRAMMING LANGUAGES Many high level programming languages have been developed, for example *Java* and *C++*. These languages tend to be easy to learn and can enable a programmer to quickly create very complex programs without digging into the complexity of machine code.

Programs written in any of these high-level languages must be either interpreted or compiled before they can be executed. Interpreters are only available for "scripting" languages like *PERL* and *Python* and they execute the source code one line at a time. In general, interpreters cannot optimize the code so they are not efficient; but they enable a programmer to quickly "try out" some bit of code without having to compile it. A compiler, on the other hand, converts the entire program to machine code (normally referred to as "object" code by a compiler) and then creates an executable file. A compiler also optimizes the code so it executes as efficiently as possible.

In the end, there are dozens of different programming languages, but they all eventually reduce programming instructions to a series of ones and zeros which the CPU can execute.

13.2.1.4 *State Machine*

Because a CPU is little more than a complex FSM, the next step in the design process is to define the various states and the operations that take place within each state. In general, an operating CPU cycles endlessly through three primary states:

- FETCH an instruction from memory. Instructions take the form of 16-bit numbers similar in appearance to 1001000100001010.
- DECODE the instruction. The instruction fetched from memory must be decoded into something like "Add the contents of Register 2 to Register 3 and save the results in Register 1."
- EXECUTE the instruction.

In general, the way that a CPU functions is to fetch a single instruction from glsram and then decode and execute that instruction. As an example, consider the Assembly example introduced above:

```
Label Mnemonic Operands    Comment
START NOP      0 0 0       No Operation
      LDI      1 0 a       R1  <- 0ah
      LDI      2 0 9       R2  <- 09h
      SHL      3 1 4       R3  <- R1 << 8
      XOR      0 2 3       Acc <- R2 XOR R3
```

Each line is an instruction and the CPU would fetch and execute each instruction from RAM in order. The purpose of this short code snip is to load a 16-bit register with a number when only 8 bits are available in the opcode. The eight high order bits are loaded into Register one and the eight low order bits are loaded into Register two. The high order bits are shifted to the left eight places and then the two registers are XOR'd together:

1. NOP: This is a "no operation" instruction so the CPU does nothing.
2. LDI: The number $0A_{16}$ is loaded into register one. This is the value of the high-order bits desired in the 16-bit number.

3. LDI: The number 09_{16} is loaded into register two. This is the value of the low-order bits desired in the 16-bit number.

4. SHL: Register three is loaded with the value of register one shifted left eight places.

5. XOR: The Accumulator is loaded with the value of register two XOR'd with 4egister three. This leaves the accumulator with a 16-bit number, $0A09_{16}$ that was loaded eight bits at a time.

13.2.2 *CPU States*

The first task that a CPU must accomplish is to fetch and decode an instruction held in memory. Figure 13.3 is a simplified state diagram that shows the first two CPU states as *Fetch* and *Decode*. After that, all of the different instructions would create their own state (for simplicity, only three instructions are shown in the Figure 13.3).

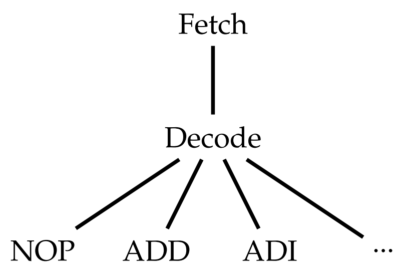

Figure 13.3: CPU State Diagram

The CPU designer would continue to design states for all instructions and end each state with a loop back to Fetch in order to get the next instruction out of RAM.

Designing a CPU is no insignificant task and is well beyond the scope of this book. However, one of the labs in the accompanying lab manual design a very simple processor that demonstrates how bits are moved around a circuit based upon control codes.