13 PROCESSOR

13.1 PURPOSE

A Central Processing Unit (CPU) is arguably one of the most important digital logic devices. CPUs are found in all computers and many other embedded logic devices. They are versatile circuits that can be used to control many processes and peripheral devices. The purpose of this lab is to lay the foundation of CPU operation.

13.1.1 *A Definition*

When asked to define "CPU" many students offer poetic definitions like "it is the brain of the computer." This may be somewhat artistic but is not very helpful in defining CPU for digital logic purposes. Here is a much better definition:

> A Central Processing Unit (CPU) is a hardware device that is designed to translate binary codes stored in software into signals that control hardware. Thus, a CPU is the interface between software and hardware.

The purpose of this lab is to demonstrate how binary codes can be used to manipulate hardware devices, like registers and adders, to move data through a circuit and accomplish a purpose. While the circuit developed in this lab is not a practical start for a CPU is does serve as an introduction to the concept of hardware manipulation by software codes.

13.2 PROCEDURE

This processor contains only three subcircuits connected by several bus lines and each of the three subcircuits are reasonably simple to understand.

13.2.1 *Arithmetic-Logic Unit*

This processor starts with a simple ALU, as in Figure 13.1.

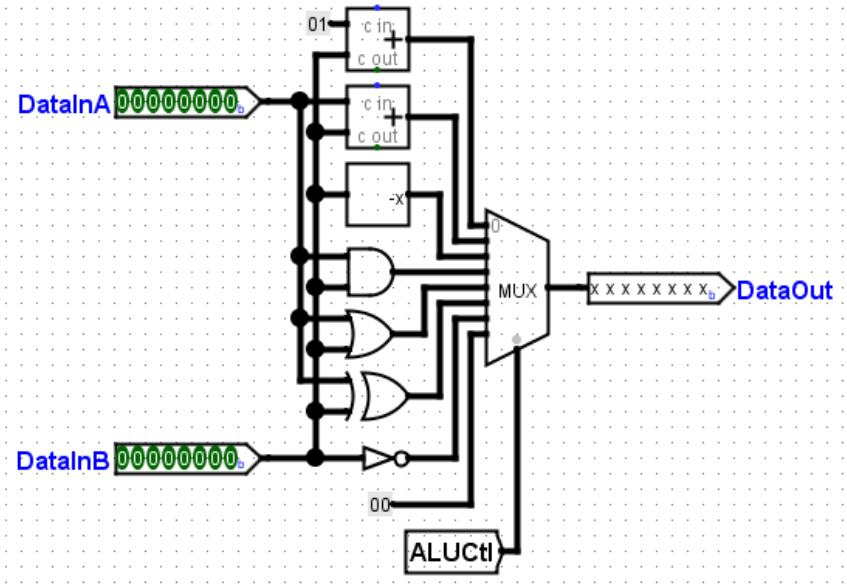

Figure 13.1: Simple ALU

To be sure, this ALU is not very complex but uses the same principles developed in Lab **??**, **??**. It contains only three arithmetic functions, increment, add, and negate; four logic functions, `AND`, `OR`, `XOR`, `NOT`; and one constant zero output. There are two data input ports but note that some of the functions only use the lower input, and one output port. The multiplexer determines which of the functions will be connected to the output and that is controlled by a signal named *ALUCtl*.

The ALU is then expanded somewhat to make it usable in a CPU. For simplicity, Figure 13.2 shows only the left side of the ALU.

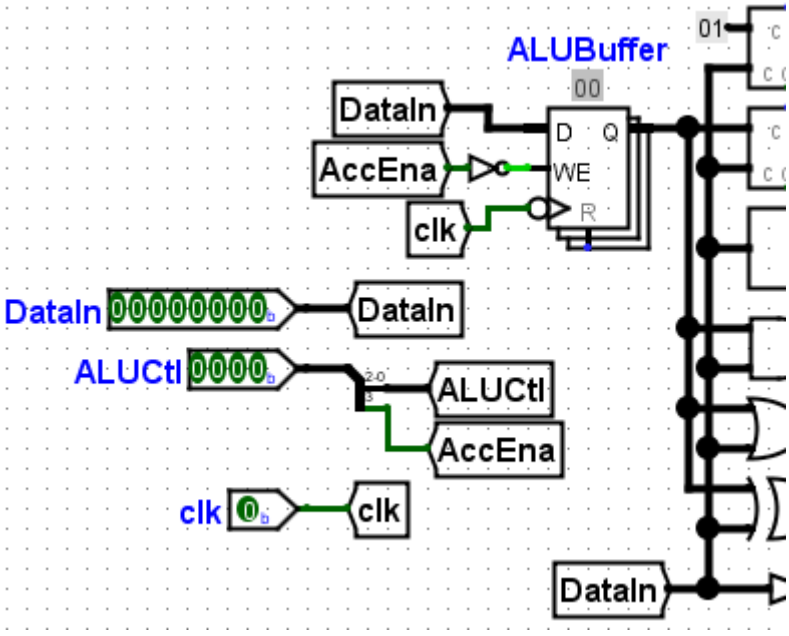

Figure 13.2: Left Side of ALU

Figure 13.3 shows the right side of the ALU.

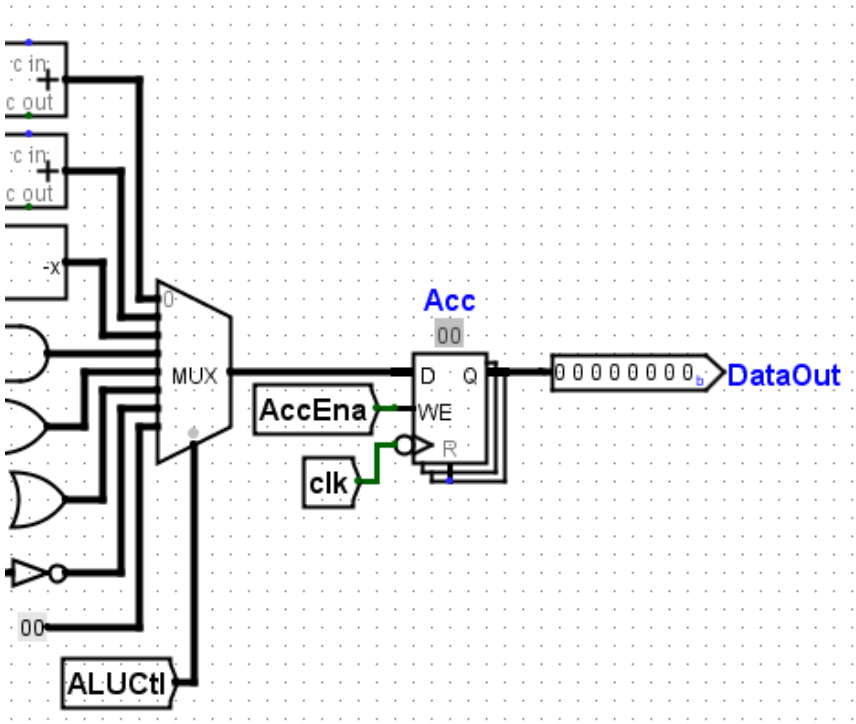

Figure 13.3: Full ALU

The simple ALU functions are found in the center of Figure 13.2. However, what started as *DataInA* has been replaced by a register

named *ALUBuffer*.[1] The *ALUBuffer's* inputs are from Tunnels (*Wiring* library) because those inputs are used in more than one location in the subcircuit.[2]

The ALU output is routed through a register named *Acc*, for *Accumulator*, which is the commonly-used name for the ALU output in a CPU circuit.

On the left side of the subcircuit are the three input ports. *DataIn* is an eight-bit number that is sent to both the *ALUBuffer* and the lower *DataIn* bus. The *ALUCtl* signal is split into two components. Bits 0-2 are sent to the multiplexer to select which of the eight functions will be output. Bit 3 of the *ALUCtl* signal is sent to the *AccEna* tunnel and when that is high the *Acc* register will be enabled but when that signal is low then the *ALUBuffer* register will be enabled. Finally, the clock input is sent to both registers.

13.2.2 *General Registers*

A CPU must have several general registers available to hold data temporarily while an instruction is being carried out. For example, it may be necessary to hold the *Acc* output until it is needed in a later step so that value can be stored in a register and then recovered when needed.

The processor circuit being built in this lab has four general registers. Figure 13.4 illustrates the `GenReg` subcircuit.

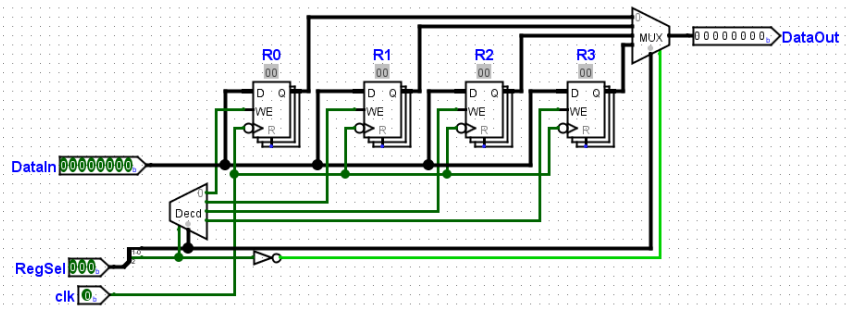

Figure 13.4: General Registers

The `GenReg` subcircuit does not require any novel digital logic concepts. Starting on the left side of the circuit:

- *DataIn* is connected to the data bus and is the main input port for the registers. Note that *DataIn* is connected to the *Data* port on all four registers.

1 IMPORTANT NOTE: All registers in this Processor circuit are triggered on the Falling Edge of the clock. The reason for this will become evident when the circuit is tested.

2 Tunnels are used extensively in this circuit to simplify the diagrams and aid in tracing signals.

- The register that actually stores the input data is determined by the Decoder (*Plexers* library) in the lower left corner of the subcircuit. The two low-order bits from the *RegSel* signal activate one of the output lines from the Decoder and that line is tied to the Write Enable port of the register. On the next clock pulse that register will lock in the data present on the *DataIn* port.
- The outputs from all of the registers are wired to a Multiplexer (*Plexers* library). The select bits from the Decoder that are used to select the storage register are also used to select the register output line which is, in turn, wired to the *DataOut* port.
- The high-order bit from the *RegSel* control signal is used to determine if data are stored to or read from a register. When that bit is high the decoder is active and will select a storage register but when that bit is low the output multiplexer will be activated and send a register's stored value to the output port.

13.2.3 *Control*

The `Control` subcircuit in this device is very simple and could, in all actuality, be eliminated. However, in a true CPU the `Control` subcircuit is rather complex and critical to the operation of the circuit so a `Control` subcircuit is included in this lab as an example. Figure 13.5 illustrates the `Control` subcircuit.

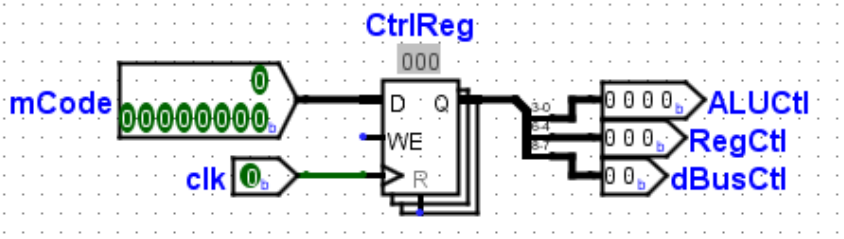

Figure 13.5: Control Subcircuit

The `Control` subcircuit includes a nine-bit input named *mCode* (for "Microcode"). That input is latched by a register[3] and the output of that register is split into three components.

BITS 0-3 These are the ALU control bits and they are sent to the `ALU` subcircuit.

BITS 4-6 These are the register control bits and are sent to that subcircuit.

3 Note, as an exception to the other registers in the Processor circuit, the register in the control subcircuit must be set to trigger on the leading edge of the clock rather than the falling edge.

BITS 7-8 These are the *dBus* ("Data Bus") control bits. The data bus is found in the `main` circuit and carries the data to each of the subcircuits. The dBus control is just a multiplexer that controls which subcircuit's output has control of the data bus.

13.2.4 *Main*

The `main` circuit ties the three subcircuits together with three control busses and one data bus. Figure 13.6 illustrates the `main` circuit.

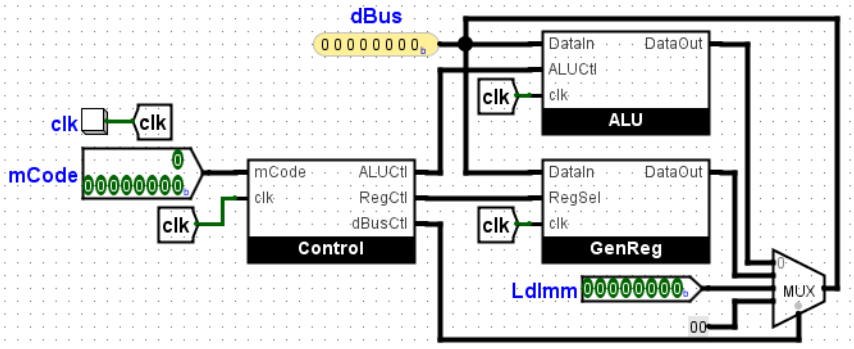

Figure 13.6: Main Circuit

There are no novel digital logic functions used in this circuit. The first input is *mCode* which is the microcode used to control the flow of data in the dBus ("data bus"). the other input, *LdImm* ("Load Immediate") can contain an eight-bit number that is to be loaded into one of the registers for processing. In a full CPU that input would be wired to a Random Access Memory (RAM) device.

13.2.5 *Testing the Circuit*

The circuit should be tested by inputting these signals and observing the output.

13.2.5.1 *Copy LdImm To R0*

Enter some value in the *LdImm* input port, set the *mCode* input to 101000000 (the first three values in the table below), and then pulse the *clk*. When completed, the *dBus* and *R0* should both contain the value of the *LdImm* port.

dBus	Reg	ALU	dBus	Notes
10	100	0000	LdImm	R0 <- LdImm

Table 13.1: R0 <- LdImm

13.2.5.2 *Copy LdImm To R1*

Enter some value in the *LdImm* input port, set the *mCode* input to 101010000 (the first three values in the table below), and then pulse the *clk*. When completed, the *dBus* and *R1* should both contain the value of the *LdImm* port.

dBus	Reg	ALU	dBus	Notes
10	101	0000	LdImm	R1 <- LdImm

Table 13.2: R1 <- LdImm

13.2.5.3 *Copy LdImm To ALUbuf*

Enter some value in the *LdImm* input port, set the *mCode* input to 100000000 (the first three values in the table below), and then pulse the *clk*. When completed, the *dBus* and *ALUbuf* should both contain the value of the *LdImm* port.

dBus	Reg	ALU	dBus	Notes
10	000	0000	LdImm	ALU <- LdImm

Table 13.3: ALU <- LdImm

13.2.5.4 *Increment Ro*

Use the LdImm function to initialize Ro.

Incrementing the value in Ro requires two steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, *Ro* will contain the original value of the *Ro*+1.

dBus	Reg	ALU	dBus	Notes
01	000	1000	R0	Acc <- R0+1
00	100	0000	Acc	R0 <- Acc

Table 13.4: Ro <- Inc(Ro)

13.2.5.5 *Add Ro And R1, Store In Ro*

Use the LdImm function to initialize Ro and R1.

Adding the values of *Ro* and *R1* and storing the result in *Ro* requires three steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the sum of the original values of *Ro* and *R1* will be stored in *Ro*.

dBus	Reg	ALU	dBus	Notes
01	001	0001	R1	ALU <- R1
01	000	1001	R0	Acc <- R0 + R1
00	100	0001	Acc	R0 <- Acc

Table 13.5: R0 <- R0 + R1

13.2.5.6 *Subtract R1 From R0, Store In R0*

Use the LdImm function to initialize R0 and R1.

Subtracting the value of *R1* from *R0* and storing the result in *R0* requires four steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the difference of the original values of *R0* and *R1* will be stored in *R0*.

dBus	Reg	ALU	dBus	Notes
01	000	0010	R0	ALUbuf <- R0
01	001	1010	R1	Acc <- ~R1
00	100	1001	R0-R1	dBus <- Acc
00	100	0111	dBus+1	R0 <- R0 - R1

Table 13.6: R0 <- R0 - R1

13.2.5.7 *Copy R0 to R1*

Use the LdImm function to initialize R0.

Copying the value of *R0* to *R1* requires four steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the value of *R0* will be stored in *R1*.

dBus	Reg	ALU	dBus	Notes
00	000	1111	0	dBus <- 0
00	000	0100	0	ALU <- dBus
01	000	1100	Acc	Acc <- ALU OR R0
00	101	0111	Acc	R1 <- Acc

Table 13.7: R1 <- R0

13.2.5.8 *Swap R0 And R1*

Use the LdImm function to initialize R0 and R1.

Swapping the values of *R0* and *R1* requires 12 steps. Set the *mCode* input to the first three values in the table below and pulse the *clk* for each of the steps. When completed, the values of *R0* and *R1* will exchanged.

dBus	Reg	ALU	dBus	Notes
00	000	1111	0	dBus <- 0 (Move R0 to R2)
00	000	0100	0	ALU <- dBus
01	000	1100	Acc	Acc <- ALU OR R0
00	110	0111	Acc	R2 <- Acc
00	000	1111	0	dBus <- 0 (Move R1 to R0)
00	000	0100	0	ALU <- dBus
01	001	1100	Acc	Acc <- ALU OR R1
00	100	0111	Acc	R0 <- Acc
00	000	1111	0	dBus <- 0 (Move R2 to R1)
00	000	0100	0	ALU <- dBus
01	010	1100	Acc	Acc <- ALU OR R2
00	101	0111	Acc	R1 <- Acc

Table 13.8: Ro <-> R1

13.3 ABOUT PROGRAMMING LANGUAGES

The codes that were input for the last example (swap *Ro* and *R1*) would create the following program.

```
000001111
000000100
010001100
001100111
000001111
000000100
010011100
001000111
000001111
000000100
010101100
001010111
```

This group of instructions would be considered "CPU Microcode," which is a very highly specialized form of programming. It is the code that is built into a CPU circuit and it determines what gates, registers, and other devices are active for each step of the code. When Intel, AMD, Motorola, or other manufacturers create a new CPU, one of their main challenges is creating the microcode that will, for exam-

ple, "add the contents of register one to the contents of register two and store the result in register zero." The microcode must be able to activate and deactivate various devices within the CPU so data appear on the appropriate bus at the right time in order to achieve the objective. Normally, microcode steps must be executed over several clock cycles in order to do a single job. For example, in one clock cycle the contents of register one may be placed on the data bus, the next clock cycle will load that data into the ALU register, and so forth until the entire process is complete.

Microcode is usually stored in ROM that is built into the CPU. This is typically called "firmware" since it is a string of ones and zeros, like software, but it cannot be changed, like hardware.

It is important to keep in mind the difference between instructions contained in a software program (like Word) and those contained in microcode. A single instruction in software is interpreted and executed by the CPU using, perhaps, dozens of microcode steps. As an example, the software may want to move a single byte from RAM to the video card. The CPU may process that instruction by first moving the byte from RAM to register one and then moving it from there to the video card's input register and then activating the video card input function. Those moves may require several clock cycles as various multiplexers and other devices are activated in the correct sequence to move the data to its destination.

A software program, like Word, is nothing more than a series of ones and zeros, organized into groups, commonly 64 in modern computers. Each group of bits forms a single "word" of information; or a single instruction which would then be used by the CPU to trigger a microcode sequence. When viewed at the level of ones and zeros, a software program is said to be in "machine code," and could look something like the following (note, only the first 32 bits of each word are shown).

```
10010100101100101001101011001010
01101001101011000111101011101011
00011011110010000111010111100101
```

If a programmer could master machine code, then those programs would be as concise and efficient as possible since they would be written in machine code the CPU can execute directly. 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" code. Assembly uses easy-to-remember abbreviations to represent the various CPU instructions available; and it looks something like this:

```
INP
STA FIRST
INP
STA SECOND
LDA FIRST
SUB SECOND
OUT
HLT
FIRST DAT
SECOND DAT
```

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 converts a file containing assembly codes into machine codes that can be executed by the CPU.

Many programming languages have been developed that are considered "higher" than Assembly; for example, C++, Java, and Visual Basic. These languages tend to be easy to master and can enable a programmer to quickly create very complex programs. Programs written in each of these languages must be compiled, or changed into machine code, before they can be executed. Here is an example Java program:

```
public class HelloWorldExample{
  public static void main(String args[]){
  System.out.println("Hello World !");
  }
}
```

In the end, while there are dozens of different programming languages, they are all designed to be reduced into a series of machine codes which the CPU can then execute.

13.4 CHALLENGE

Using the examples in the "Testing the Circuit" section, create the microcode necessary to carry out these functions:

1. Store the value contained in *LdImm* in *R2* (*R2* <- *LdImm*). (Assume that *LdImm* is pre-loaded with the value to store.)
2. Store the value contained in *LdImm* in *R3* (*R3* <- *LdImm*). (Assume that *LdImm* is pre-loaded with the value to store.)
3. Store the 2s complement of the value in *Ro* back into *Ro* (*Ro* <- *~Ro*). The subtraction example will help with this function.
4. Store the bitwise NOT of the value in *Ro* back into *Ro* (*Ro* <- *Ro'*).

13.5 DELIVERABLE

To receive a grade for this lab, build the Processor circuit and then complete the Challenge. Be sure the standard identifying information is at the top left of the Processor `main` circuit, similar to:

```
George Self
Lab 13: Processor
April 5, 2018
```

Save the Processor circuit in a file with this name: *Lab13_Processor*. Complete the code required in the Challenge and store that in a text file with the name *Lab13_Code.txt*. Submit both files for grading.