

CSE 140L Spring 2018

Lab Assignment 1

Alarm Clocks

Due April 11, 2018, Wednesday, 9:00am

Introduction

The goal of this lab is to familiarize you with *LogicWorks*. Your task is to implement alarm clocks using a couple of different calendars. This assignment consists of three parts. In the first part, you will implement a conventional alarm clock. In the second part, you will implement an alternative version of the alarm clock that conforms to the rules of the French Republican Calendar. In the third part, you will extend the alarm clock you have built in Part 2 to keep track of the day of the week in the French Republican Calendar.

A library (*Lab1Sp18.clf*) containing several pre-designed components with which you can build your alarm clocks will be provided. We will also provide you the behavioral and structural diagrams for Part 1. In Part 3, it may be helpful to draw a behavioral diagram of how this alarm clock functions.

The deliverable requirements are specified at the end of each section. You will also need to answer some questions in a **report** according to the instructions. Electronic submission details are on the GradeScope page.

Part 1: Basic Alarm Clock

Behavior: At its simplest, an alarm clock has two functionalities – keeping track of the current time, and buzzing when the current time matches the alarm time. While it is possible to think of alarm clocks with a preset alarm time, the ensuing market fragmentation makes such a solution uneconomical – one would have to buy a new alarm clock any time he or she wants to change the alarm time! This inconvenience pushes us to add functionality that allows users to program the alarm time of the clock. Furthermore, electronic gadgets are prone to power loss and (unless you invest in nonvolatile memory) lose track of internal storage values in the process. Thus, we should extend our programmability to the current time as well as the alarm time. In any case, as we shall see shortly, setting the current time and setting the alarm time are very similar tasks, with similar implementations in hardware.

Keeping track of the current time, the clock's first and most basic functionality, is rather straightforward. At least in the initial formulation, we will content ourselves with keeping track of hours, minutes, and seconds, and leave higher-order signifiers of our temporal existence to more advanced implementations in subsequent parts. The relationship between the three units of time is captured in **Figure 1**, wherein the eccentric yet subtly elegant Babylonian system of 24 hours, 60 minutes and 60 seconds which we conventionally use is described. We spare you the small details by providing you a **ClockDev** component that already implements this functionality.

The buzzing part of the alarm is essentially nothing but an equality check between the minutes and hours of the current time and the alarm time (not too many people seem to care about alarms that buzz on a particular second, so the buzzer will not check for equality of seconds), complete with an **AlarmOn** switch that is used to turn the buzzer on and off, in case the user doesn't want to be rudely awakened on a day of leisure (See switch “S5” in **Figure 2** and **Figure 4**). The setting of the alarm time and the current time is shown in **Figure 3**. The user will enable the setting of the time with a pair of switches, **SetCurrentTime** (switch “S1”) and **SetAlarmTime** (Switch “S2”). If neither switch is on, the user will be

unable to set either; otherwise, the user will be able to set the current time if **SetCurrentTime** is on, or the alarm time if **SetAlarmTime** is on. While the user is setting either the current time or the alarm time, a pair of switches (actually “pushbuttons”, as we will see), **IncMinute** (Switch “S3”) and **IncHour** (Switch “S4”), will be used to increment the hours and minutes, respectively.

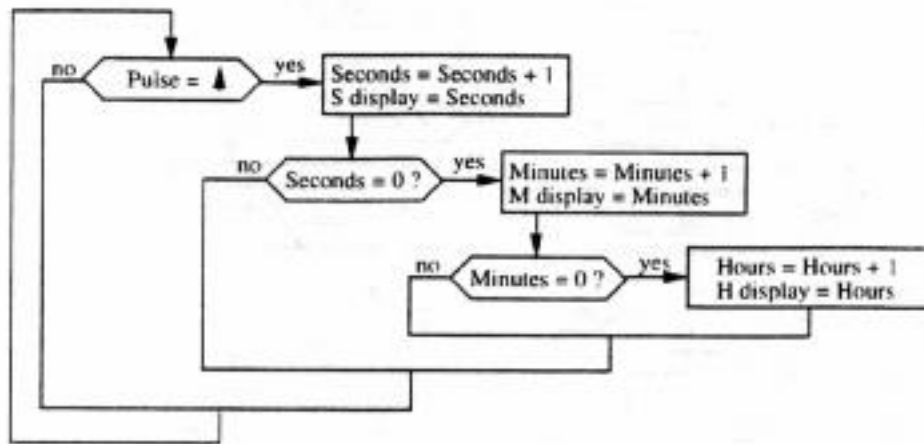


Figure 1: Clock Tick Behavior

Structure: Now that you see the tripartite behavioral description of the alarm clock, it is time to move on to a structural description. For the purpose of this exercise, the structural description can be thought of as being composed of two parts, namely, the components used to build the clock and their interconnections.

The best way to determine the required components is by looking at the behavioral description and identifying functionalities that map to known hardware components. This being your first hardware exercise, we will help you a bit and develop coarse-grained components that you can plug in directly instead of going to the trouble of composing them out of more fine-grained parts. (See **Reference 2** for a complete list.) In that spirit, looking at the behavioral diagram of **Figure 1**, one realizes that a component is needed that will keep track of the time and increment it appropriately with each passing unit of time. We will give you a **ClockDev** component that will perform this functionality and do a little bit more to boot. The **ClockDev** component will increment the seconds on a pulse. When the seconds reach 59, instead of being incremented to 60, they will roll over to 0 with the minutes being incremented thereupon. Similarly, the minutes, when incremented past 59, will roll over to 0 and trigger the hours to be incremented, which will in turn roll over from 23 to 0 as necessary. For the first part of this exercise, no further increments of other counters are required upon rolling over of the hours.

The behavioral description in **Figure 2** that illustrates the buzzing functionality necessitates an equality check between the current time and alarm time. We will provide you with a **7Comp** circuit that implements this functionality. (All you need for this part is a comparator of 6 bits, but we are providing you a slightly larger one so that you can handle with the same comparator the comparisons needed in later parts as well.)

Figure 3 illustrates the behavioral description for setting the correct time, a functionality that can be incorporated into the **ClockDev**. To complete the implementation of **Figures 2** and **3**, you need an **AlarmDev** component which in many ways mimics the **ClockDev** component. We will supply you with an **AlarmDev** as well.

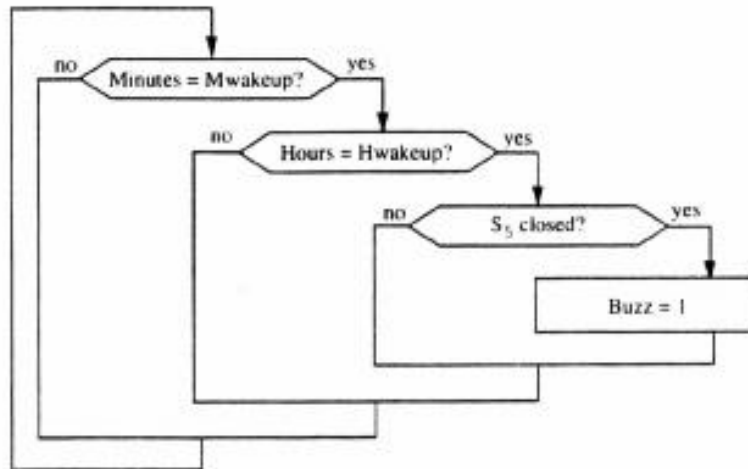


Figure 2: Alarm Buzz Behavior

We also provide you with a **MUX-2x7**. A MUX, short for multiplexer, is a circuit component which acts as a “decider” among possible output alternatives. The **MUX-2x7** receives 2 inputs, each of which are 7-bits, and outputs one of them based on a third input S, or “Select”. (See the Component Library section of **Reference 2** for a more detailed description of the **MUX-2x7**.) The reason for including this component is perhaps not as evident from the behavioral description that we have provided you. One issue is that while an Alarm Clock normally would display the current time, when you are setting the alarm time, it would be desirable (indeed necessary) to have the clock display the Alarm Time so that you know how close you are getting to the desired alarm time. So, a MUX will be needed to switch between displaying the current time (most of the time) and displaying the alarm time (when the user is setting the alarm time). (You may notice that the MUX is oversized for the conventional alarm clock, since this component as well is designed to be reusable in subsequent parts without modification.)

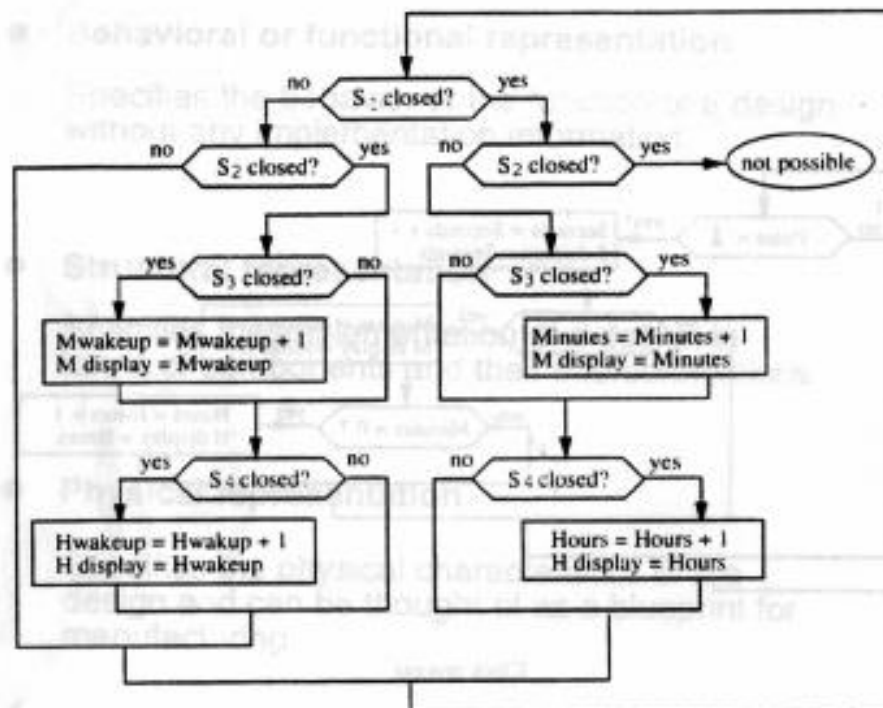


Figure 3: Time and Alarm Set Behavior

SetCurrentTime and **SetAlarmTime** together determine the functionality that the user is interested in. In **Figure 3**, these two switches are modeled by S_1 and S_2 , respectively. When both **SetCurrentTime** and **SetAlarmTime** are off (we shall use the notation “ $(S_1, S_2) = (0, 0)$ ”), the clock should passively tick and display the current time. If $(S_1, S_2) = (1, 0)$, the user should be able to set the current time (the display should show the current time). If $(S_1, S_2) = (0, 1)$, the user should be able to set the alarm time (the display should show the alarm time, so that the user can see it as it changes.) $(S_1, S_2) = (1, 1)$ will behave identically to $(S_1, S_2) = (0, 0)$, i.e. increment neither. The switches are modeled by the **Binary Switch**, which is provided by the component libraries native to *LogicWorks*. The selection of functionality can be implemented by using a **decoder**, which can also be found in the native libraries.

Now that we have taken a high-level look at the components, it's time to guide you through their interconnections to complete our structural analysis. One way to think about it is to start from the outputs. *What is the expected computation of the circuit?* The final outputs of the alarm clock are supposed to be a time (typically the current time, but occasionally the alarm time) and a signal indicating the status of the buzzer. You will therefore need some LCD displays and a binary probe. Then, we can ask *where do these display components receive information from?* The LCDs, for example, must output either the current time or the alarm time depending on the switches **SetCurrentTime** and **SetAlarmTime**. Thus, the inputs of the LCD come from multiplexers which will choose between clock time and alarm time based on the system configuration. We ask ourselves again about the multiplexers: where do they receive inputs from? After mulling the question recursively, we come down to the base case at primary inputs – the clock, buttons and switches which are used to control the clock by the user.

A high-level structural schematic that embodies this discussion is presented in **Figure 4**.

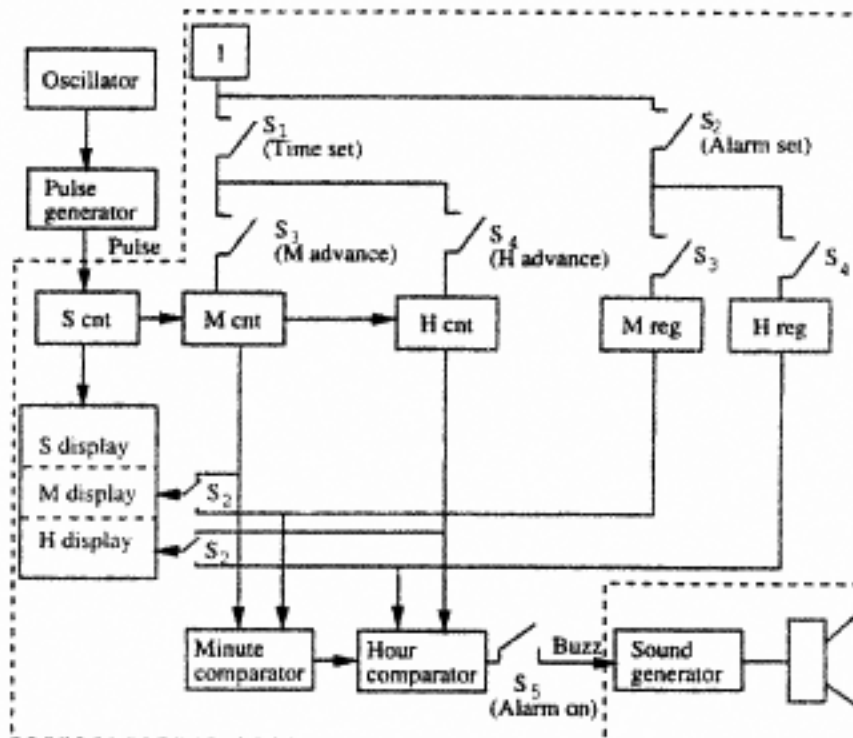


Figure 4: Alarm Clock Structural Schematic

Input/Output: The inputs to this simple alarm clock we're developing are: **AlarmOn** (a binary switch), **SetCurrentTime** and **SetAlarmTime** (two binary switches), and the **IncHour** and **IncMinute** pushbuttons. (See **Reference 2** for details about the **SPDT Pushbutton**.) You should also have a binary switch "Clear" hooked to the CLR inputs of the **ClockDev** and **AlarmDev**. The outputs will be the current time (or alarm time) on the **LCD Displays** and the state of the buzzer (a binary probe). No additional inputs or outputs should be necessary for this device, nor should they be included.

Implementation hints:

- You should use a Binary Switch to activate the alarm functionality (**AlarmOn**). (Refer to switch S5 in Figure 2 and Figure 4.)
- You should implement the logic that will enable the user to set the alarm time and the current time. You should use two switches (Refer to switches S1 and S2 in Figure 3 and Figure 4) and a decoder component, which you can get from the library, to implement the following cases:
 - If S1=1 and S2=0, then the user can set the current time;
 - If S1=0 and S2=1, then the user can set the alarm time;
 - Otherwise, no user modifications to current time or alarm time take place.
- Add a Binary Probe labeled BUZZER that will go high when the alarm is on, and the hours and the minutes of the clock match the alarm setting. If the user turns the alarm off, the BUZZER signal should go down.
- To display the time, you should be using **LCD displays** (provided by *LogicWorks*) connected to an **LCD Interface**. Rather than implementing separate displays for **AlarmDev** and **ClockDev**, there should be one single set of LCDs, as in our regular alarm clocks, which shows the alarm time only when we select that switch to set the alarm time. You can use a **MUX-2x7** to select between the inputs.

Part 1 Deliverables:

- A LogicWorks circuit file for the conventional Alarm Clock named **Lab1Part1.cct**
- In the report, explain how you tested your alarm clock.

Part 2: French Republican Calendar Alarm Clock

Our conventional calendar retains irregular periods which stem from the fact that human societies have over the years tried to synchronize time according to the rotation of the earth around the sun. An interesting alternative approach to this issue seems to be the quest for simplicity, which has been espoused by the French revolutionaries of the late 18th century who, in addition to being busy with decapitating each other via the guillotine and helping humanity transition from the realm of monarchy to that of a bourgeois democracy, were also busy with trying to modernize calendars.

In their zeal for scientific precision, the French revolutionaries carried their love of the metric system which they propounded into the measurement of hours and minutes. The French Republican calendar was much more regular with every one of the 12 months being 30 days which are divided into three ten-day weeks. The deficit of 5 days at the end of each year (or 6 days in leap years) were national holidays. Each day was divided into ten *decimal hours*, each hour into 100 *decimal minutes*, and each decimal minute into 100 *decimal seconds*.

Behavior: Your task is to implement an Alarm Clock that obeys the rules of the French Republican calendar. The operation of an alarm clock under the French Republican calendar is essentially identical to the Western calendar that you have implemented in Part 1. Again, the user must be able to set the current time and alarm time in the same manner.

Implementation hints:

You can implement the Republican Calendar Alarm Clock by mimicking the **ClockDev** and **AlarmDev** components. You can view the inner workings of these components by double clicking either component inside of your design in *LogicWorks*. Below is a description of how the **ClockDev** component works.

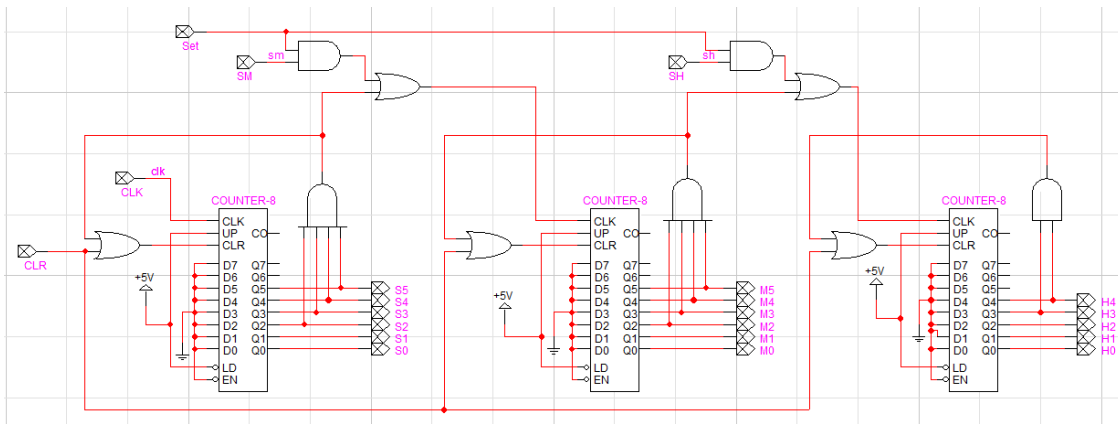


Figure 5: The implementation of ClockDev for a conventional calendar

As shown in **Figure 5**, the **ClockDev** component contains 3 separate 8-bit counters (see **Reference 2** for a detailed description of **Counter-8**). The counters, from left to right, keep track of the seconds, minutes, and hours. Let's focus on inputs to the seconds counter. The input **LD** is used to signal whether to load the specific value at inputs **D0...D7** into the counter. (Note the small circle at the **LD** input. This means the input signal semantics are flipped. You can find more information on this convention at the end of **Reference 2**.) Since that input is tied to **+5V** (denoting logic 1), which is inverted to a logic 0, the **LD** operation is disabled and the inputs **D0...D7** are unused (which are arbitrarily tied to **ground**, denoting logic 0). The **UP** input is tied to a logic 1, which indicates that the counter should increment instead of decrement. The **EN** input indicates whether the counter is enabled, i.e. should increment on the next rising edge of the **CLK** input. **EN** is tied to logic 0 (note the circle at the input once again), meaning that the counter is enabled and will increment. (Please read carefully the counter descriptions in **Reference 2** to follow the somewhat non-intuitive use of **LD** and **EN** signals in the LogicWorks Library.)

Now look at the outputs, **Q0...Q7**. Since only 6 bits are necessary to represent the numbers 0 through 60, **Q6** and **Q7** are unused. **Q2**, **Q3**, **Q4**, and **Q5** are all wired to a 4-input **AND** gate. This **AND** gate cleverly signals when the seconds counter has reached 60. Since each **Q** output represents a binary digit, if **Q2**, **Q3**, **Q4**, and **Q5** are all set to logic 1, then the value in the counter is $2^2 + 2^3 + 2^4 + 2^5 = 4 + 8 + 16 + 32 = 60$. In this scenario, the **AND** gate will output a logic 1, which feeds back to the asynchronous **CLR** input through an **OR** gate, immediately resetting the counter back to 0. This effectively creates a counter that counts from 0 to 59 before rolling back to 0 (though it does very temporarily hold the value 60). Finally, this **AND** gate is also wired to the **CLK** input of the minute counter through another **OR** gate. When the seconds counter changes from 59 to 60, the **AND** gate output changes from logic 0 to logic 1, triggering the minute counter to increment as well. The minute and hour counters are wired similarly.

Part 2 Deliverables:

- A LogicWorks circuit file for the Republican Calendar alarm clock named **Lab1Part2.cct**
- In the report, explain how you modified the two components.
- Clearly, the time standards are different across the Western Calendar and the French Revolutionary Calendar. In the report, show your work on calculating how many conventional seconds a decimal second would correspond to, and similarly how many conventional minutes a decimal minute would correspond to.

Part 3: Day of the Week in French Republican Calendar Alarm Clock

The basic French Republican Calendar alarm possesses many features, but some people may wish to keep track of the day of the week in addition to the current hours, minutes, and seconds (and perhaps only be woken up on a particular day of the week). Remember that there are 10 days in a week in the French Republican Calendar. Your job will be adding this functionality. This will require an additional **LCD Interface** wired to another **LCD Display** to show the day of the week (represented as an integer), as well as at least a 4-bit counter to keep track of the current day (Why at least 4-bit?). You may use **Counter-4** in the native *LogicWorks* library to represent a decimal day of the week.

The current time and the alarm time will maintain separate counters for the day of the week. Thus, the alarm should be settable to a specific day of the week. You will need a second **Counter-4** to keep track of the alarm time's day of the week. In addition to setting a specific alarm day which would need to be matched by the current day for the buzzing to take place at the correct day, we would like to maintain the ability to set an alarm to buzz **every day** at the designated time instead of only on a specific day. This will require slightly different semantics for the alarm counter, detailed in the next paragraph.

The counters of the current time and alarm time will be 0-indexed to represent the days of the week, running from 0 (tenth day or *day of rest*) through 9 (ninth day). Instead of incrementing to 10, the current time's counter will roll over to 0, in a manner similar to how the minutes (and seconds) roll from 59 over to 0. However, the alarm's day of the week counter will also include (and cycle through) the value 10. If the alarm time's day of the week is set to the value 10, then it will signal the alarm to ignore the day of the week when evaluating whether to trigger the alarm buzzer (i.e. buzz independent of the day of the week).

As with Part 1, you should hook your "Clear" input to initialize your counters to the appropriate values. The reset, i.e. rollover, value for the counters should be the value 0. Also, the displaying semantics should identically match previous parts, in which the current day should be displayed by default, and the alarm day should only be displayed if the user is setting the alarm.

The details of the implementation of the days of the week feature are detailed below, where (Current Time) denotes the behavior implemented for the current time, and (Alarm Time) denotes the behavior for the alarm time version:

- Similar to the hours, minutes, and seconds, a single **LCD Interface** and **LCD Display** will be used to display the day of the week for both the current time and the alarm time. Just like before, a **MUX** will be used to determine which value to display.
- (Current Time) Should count from 0 to 9
- (Current Time) Can be incremented via a **SPDT Pushbutton** or through a rollover of the hours
- (Alarm Time) Should count from 0 to 10
- (Alarm Time) Can be incremented via a **SPDT Pushbutton**
- (Alarm Time) If set to 10, then the day of the week is ignored when evaluating the logic for the BUZZER

Implementation hints:

The operation of the day of the week feature is very similar to the operation of the hours, minutes, and seconds that are already in place. You may implement the day of the week by mimicking the implementation of the counters inside of the **ClockDev** and **AlarmDev** components.

Part 3 Deliverables:

- A LogicWorks circuit file for the Republican Calendar Alarm Clock with the day of the week named **Lab1Part3.cct**.
- In the report, explain how you modified the components to support this feature, as well as challenges you came across in reaching your correct solution.

Reference 1: Implementation Issues

Testing your implementation: You should make sure your implementations work properly. Firstly, you should test that the switches are implemented correctly. Set the switches for adjusting the current time, and then observe that you can increment hours and minutes. Do the same thing for the alarm time and make sure you can increment hours and minutes for the alarm setting. This step also tests whether you have properly connected the **LCD interfaces** to the outputs.

In Part 1, you should set different clock and alarm times and then wait and observe that the alarm turns on when it is supposed to. Pay attention to how long the BUZZER signal is high. The BUZZER should be high for exactly 1 minute and then should go low. Turning the **AlarmOn** switch to Off should immediately set the BUZZER to low; you may want to verify this. In Part 2, you should verify that the clock follows the rules of the French Revolutionary Calendar. The alarm functionality should also be examined. In Part 3, check that you can increment the day of the week independently. For the current time's day of the week, verify that it increments correctly when the hours roll over. Reexamine the correctness of the alarm with an attention to the day of the week.

LogicWorks Hints: To implement the alarm clock, you will need to use some logic components, such as comparators, multiplexers, display units, etc. There are various libraries provided by LogicWorks. Each library contains components relevant to the name of library. For example, the library **Simulation IO** contains HexDisplay, BinarySwitch, BinaryProbe, clock, etc. The library **Simulation Logic** contains adders, counters, buffers, registers, decoders, etc. The **Simulation Gates** library contains AND, OR, NOR, XOR, NOT and other gates. We will additionally be supplying components relevant to particular labs; please check the lab files for useful components.

To be able to start the simulation, you need to put a "clock" in your design sheet (not to be confused with the alarm clock we are creating!). In this case, you will need to connect the "clock to counters" (i.e. the **ClockDev** component) to increment time at every clock tick. Also, be sure that your circuit file contains only a single "clock" component. A single clock can control any number of devices.

Be aware that even if some lines seem to be connected, they may not be actually connected. One way to check is by moving the components to see whether the lines connecting them move together with the component. Another way to connect pins instead of drawing wires between them is to assign names to the pins, and the simulator will assume that pins with the same names are connected.

During your simulation, you can use **binary probes** or **hex display** to monitor the lines you want to observe. Sometimes the value of a line can be "unknown" (X or "grayed digits"). If this is observed on the clock/alarm/counter outputs, clicking on the "Clear" binary switch should resolve the problem. You can also use the keyboard shortcut *Ctrl-f* to toggle the values on all wires in the design.

Reference 2: Component Libraries

In this lab, you will have to use components from multiple libraries. We will provide you with a library with pre-constructed components, and you will also require components from the native libraries that already come with *LogicWorks*.

Components in Provided Library (Lab1Sp18.clf)

LCD Interface: This component will be used to connect your design with a set of 7-segment displays so that it will be easier for you to observe the simulation. You will use one “LCD Interface” component for each set of numbers you want to display (i.e. Hours, Minutes, Seconds).

Inputs:

- 0..6 - Input bus that will carry a number between [0,127].

Outputs:

- a1...g1 - Output bus consisting of 7-segment designators to be connected to the LSD (Least Significant Digit), i.e. right decimal digit of your display;
- a2...g2 - Output bus consisting of 7-segment designators to be connected to the MSD (Most Significant Digit), i.e. left decimal digit of your display.

ClockDev: This is the main component through which you will be able to set the current time.

Inputs:

- CLR – Sets current time to 00:00:00 (0 hours, 0 minutes and 0 seconds);
- CLK – Input from a pulse generator;
- Set – When set to 1, it will allow user to change the current time;
- SH (Set Hours) – If Set = 1, the hours of the clock will increment by 1 when SH is toggled to 1;
- SM (Set Minutes) – If Set = 1, the minutes of the clock will increment by 1 when SM is toggled to 1.

Outputs:

- H4...H0 – The hours represented by a 5-bit integer. H0 is LSB while H4 is MSB;
- M5...M0 – The minutes represented by a 6-bit integer. M0 is LSB while M5 is MSB;
- S5...S0 – The seconds represented by a 6-bit integer. S0 is LSB while S5 is MSB. Note that here S denotes Seconds, not Switches.

AlarmDev: You will be able to set the alarm time with this component. The inputs and outputs are very similar to the **ClockDev** component, except that the **AlarmDev** does not have the CLK input and the seconds output (S5...S0). CLK is not needed here since we do not want to increment the alarm time after we set it. Neither is the seconds output necessary as the alarm time is typically set in hours and minutes only. The remaining input and output pins are exactly the same as the **ClockDev** component.

Inputs:

- CLR – Sets alarm time to 00:00 (0 hours and 0 minutes, and seconds are not included);
- Set – When set to 1, it will allow the user to change the alarm time;
- SH (Set Hours) – If Set = 1, the hours of the alarm time will increment by 1 when SH is toggled to 1;
- SM (Set Minutes) – If Set = 1, the minutes of the alarm time will increment by 1 when SM is toggled to 1.

Outputs:

- H4...H0 – The hours represented by a 5-bit integer. H0 is LSB while H4 is MSB;
- M5...M0 – The minutes represented by a 6-bit integer. M0 is LSB while M5 is MSB.

7Comp: This is a 7-bit comparator that you can use to compare two 7-bit signals.

Inputs:

- A6...A0 – One of the two 7-bit inputs;
- B6...B0 – The other 7-bit input.

Output:

- Z – Outputs a '0' if A and B are not equal, and a '1' if they match exactly.

MUX-2x7: This is a 2-to-1 multiplexer which has 7-bit inputs. You will need this component to select either the current time or the alarm time to display.

Inputs:

- A6...A0 – One of the two 7-bit inputs;
- B6...B0 – The other 7-bit input;
- S – Select signal. When S=0, A6...A0 will be transmitted to the outputs, whereas when S=1, B6...B0 will be transmitted to the outputs.

Output:

- Q6...Q0 – 7-bit output bus.

LogicWorks Native Libraries:

Binary Switch: Toggles its output when the user clicks it. The starting output is usually 0.

Binary Probe: Shows the value on the wire that it is connected to. Possible values are 0, 1, C (conflict), X (unknown), and Z (high impedance).

SPDT Pushbutton: Outputs the passive input unless the user is holding the left mouse click over the button, in which case it outputs the active input. This button can be used when a logic value is needed instantaneously, such as clear or increment signals.

Decoder-4: Takes select signals as input, and resets the output corresponding to the select signals. All other outputs are set. Details about its enable signal are further discussed at the end.

Inputs:

- S0 and S1 – Select Signals;
- EN – Enable signal.

Outputs:

- Q0...Q3 – Output ports. If EN = 1, all outputs are 1. If EN = 0, for $i = 0$ to 3, $Q_i = 0$ if $S_1S_0 == i$, else $Q_i = 1$.

LCD Display: 7-segment display to be connected to the **LCD Interface** provided.

Inputs:

- a...g – Input bus containing the on/off status of each segment in the display;
- dot – Input controlling the on/off status of the decimal point in the display. It is unused for this lab and may be tied to ground (logic 0).

Counter-4: Keeps track of a 4-bit integer value and allows it to increment under certain conditions. Note that the semantics of EN and LD are inverted from what one would conventionally expect.

Inputs:

- CLK – An input which determines when the counter increments. Specifically, if CLK transitions from 0 to 1 (and the CLR input is 0), the current value of Q will increment;
- UP – Specifies whether to count up or down (you should set this to 1 as you will be performing up counting only in this lab);
- CLR – If CLR is set (i.e. CLR = 1), the value of Q will be set to 0. Furthermore, Q cannot increment until CLR is reset (i.e. CLR = 0);
- LD – Asynchronous signal enabling a load from D (discussed in the next line);
- D3...D0 – The value to be loaded if LD = 0;
- EN – When set to 1, the counter will **not** increment. When reset to 0, the counter will increment.

Outputs:

- Q3...Q0 – The current value of the integer being kept track of by the counter;
- C0 - “Carry-out”, or rollover, which turns to 0 when UP = 1 and the current value is 1111 or when UP = 0 and the current value is 0000; the output value is 1 otherwise.

Counter-8: Keeps track of an 8-bit integer value and allows it to increment under certain conditions. Note that the semantics of EN and LD are inverted from what one would conventionally expect.

Inputs:

- CLK – An input which determines when the counter increments. Specifically, if CLK transitions from 0 to 1 (and the CLR input is 0), the current value of Q will increment;
- UP – Specifies whether to count up or down (you should set this to 1 as you will be performing up counting only in this lab);
- CLR – If CLR is set (i.e. CLR = 1), the value of Q will be set to 0. Furthermore, Q cannot increment until CLR is reset (i.e. CLR = 0);
- LD – Asynchronous signal enabling a load from D (discussed in the next line);
- D7...D0 – The value to be loaded if LD = 0;
- EN – When set to 1, the counter will **not** increment. When reset to 0, the counter will increment.

Outputs:

- Q7...Q0 – The current value of the integer being kept track of by the counter;
- C0 - “Carry-out”, or rollover, which turns to 0 when UP = 1 and the current value is 11111111 or when UP = 0 and the current value is 00000000; the output value is 1 otherwise.

Some of the inputs of these components may be different than the standard convention. For example, typically an enable signal (EN) will enable the component if it is set (i.e. 1) and disable the component if it is reset (i.e. 0). This is opposite to the convention used in the LogicWorks native library for the **Counter-4**, **Counter-8**, and **Decoder-4** components. If you look at the components' symbols in *LogicWorks*, you will notice that there is a small circle at the EN inputs. This signifies that the EN semantics are complemented, meaning that an input 0 enables the device and an input 1 disables it. Note that the LD inputs of **Counter-4** and **Counter-8** have the small circle as well, resulting in a similar interpretation. The Q outputs of **Decoder-4** also have the circle. This indicates that all outputs are at logic 1 except for the selected output, which is at logic 0. This is contrary to a typical decoder, which has all outputs at logic 0 except for the selected output.