Level 2

At this point, the three amplifier stages are ready for detailed component level design, while the power supply needs another level of refinement, as shown in Figure 5.4. The functional requirement for each of the elements in the power supply would be developed similarly. Functional decomposition stops at this point—all levels of the hierarchy are defined and the next step is the detailed design, where the actual circuit components are determined.

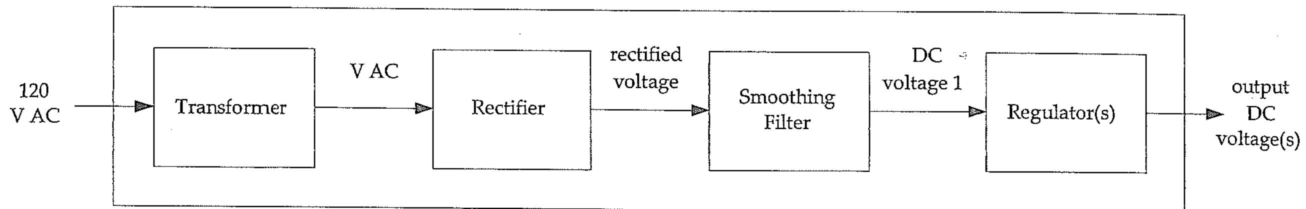

Figure 5.4 Level 2 design of the power supply.

5.5 Application: Digital Design

Functional decomposition is widely applied to the design of digital systems, where it is known as *entity-architecture* design. The inputs and outputs refer to the entity, and the architecture describes the functionality. The application of functional decomposition to digital systems is demonstrated in the following example. Consider the design of a simple digital stopwatch that keeps track of seconds and has the following engineering requirements.

The system must

- Have no more than two control buttons.
- Implement run, stop, and reset functions.
- Output a 16-bit binary number that represents seconds elapsed.

Level 0

The Level 0 diagram and functional requirements are shown in Figure 5.5.

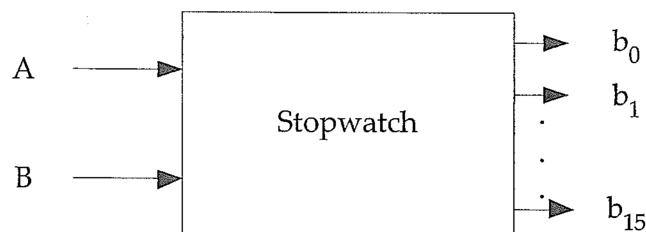

Figure 5.5 Level 0 digital stopwatch functionality.

Module	Stopwatch
Inputs	- A: Reset button signal. When the button is pushed it resets the counter to zero. - B: Run/stop toggle signal. When the button is pushed it toggles between run and stop modes.
Outputs	- b_{15}–b_0: 16-bit binary number that represents the number of seconds elapsed.
Functionality	The stopwatch counts the number of seconds after B is pushed when the system is in the reset or stop mode. When in run mode and B is pushed, the stopwatch stops counting. A reset button push (A) will reset the output value of the counter to zero only when the stopwatch is in stop mode.

Level 1

The Level 1 architecture in Figure 5.6 contains three modules: a seconds counter, a clock divider, and a finite state machine (FSM). The stopwatch counts seconds, thus the seconds counter module counts the seconds and outputs a 16-bit number representing the number of seconds elapsed. The clock divider generates a 1 Hz signal that triggers the seconds counter. The FSM responds to the button press stimuli and produces the appropriate control signals for the seconds counter. The system clock is included to clock both the FSM and the clock divider.

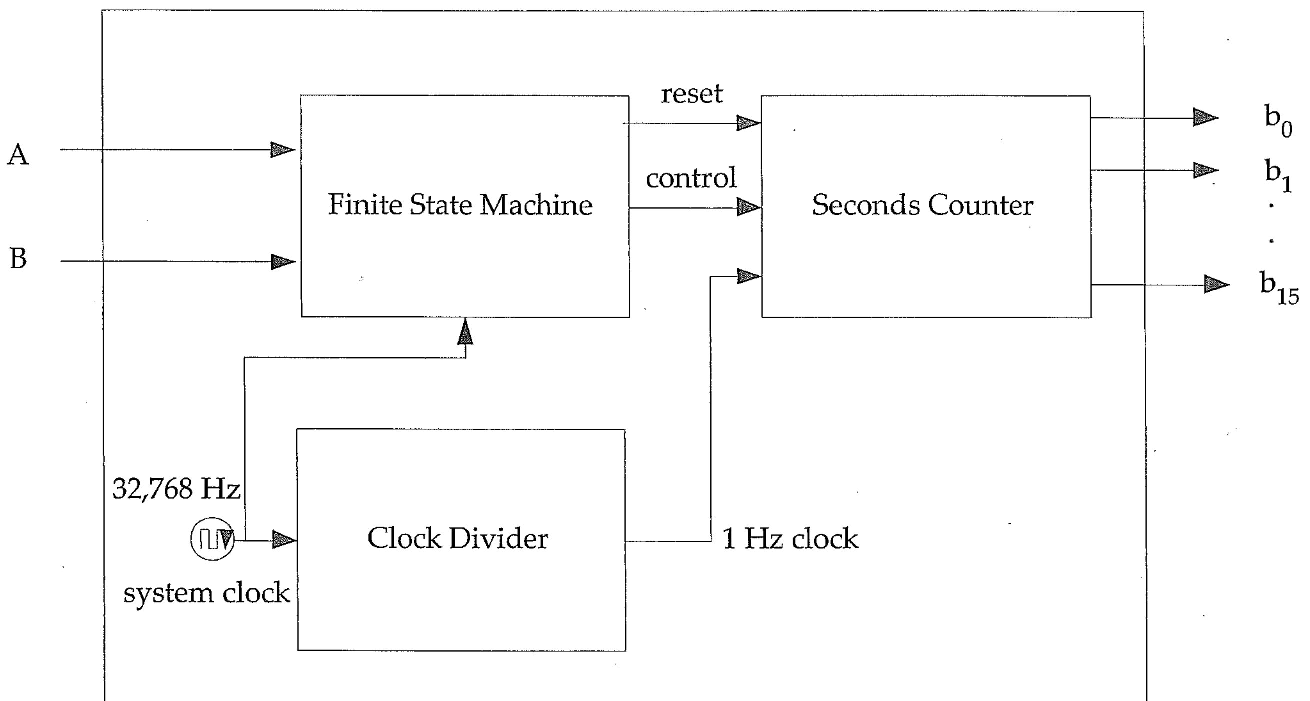

Figure 5.6 Level 1 design for the digital stopwatch.

The functionality of the Level 1 modules is described as follows, starting with the finite state machine.

Module	Finite State Machine
Inputs	- A: Signal to reset the counter. - B: Signal to toggle the stopwatch between run and stop modes. - Clock: 1 Hz clock signal.
Outputs	- Reset: Signal to reset the counter to zero. - Control: Signal that enables or disables the counter.
Functionality	

The functionality of the finite state machine is described with a tool that is probably familiar to the reader, the state diagram. State diagrams are covered in more detail in Chapter 6. The state diagram describes stimulus-response behavior, and shows how the system transitions between states according to logic signals from the button presses.

Next, consider the clock divider.

Module	Clock Divider
Inputs	- System clock: 32,768 Hz.
Outputs	- Internal clock: 1 Hz clock for seconds elapsed.
Functionality	Divide the system clock by 32,768 to produce a 1 Hz clock.

The value of 32,768 Hz was selected for the system clock for several reasons. It is a power of 2 that is easily divisible by digital circuitry to produce a 1 Hz output signal. It is also well above the clock rate needed for detecting button presses, and there is a wide selection of crystals that can meet this requirement.

Finally, consider the seconds counter.

Module	Seconds Counter
Inputs	- Reset: Reset the counter to zero. - Control: Enable/disable the counter. - Clock: Increment the counter.
Outputs	- b_{15}–b_0: 16-bit binary representation of number of seconds elapsed.
Functionality	Count the seconds when enabled and resets to zero when reset signal enabled.

The system decomposition would end here, assuming that the design is to be implemented with off-the-shelf chips. The next step would be to determine components at the detailed design level. However, if it were an integrated circuit design, the description would continue until the transistor level is reached.

5.6 Application: Software Design

Software also lends itself to functional decomposition, since virtually all computing languages provide the capability to call functions, subroutines, or modules. Functional software design simplifies program development by eliminating the need to create redundant code via the use of functions that are called repeatedly.

Structure charts are specialized block diagrams for visualizing functional software designs. The modules used in a structure chart are shown in Figure 5.7. The larger arrows indicate connections to other modules, while the smaller arrows represent data and control information passed between modules. Five basic modules are utilized:

1) *Input modules.* Receive information.
2) *Output modules.* Return information.
3) *Transform modules.* Receive information, change it, and return the changed information.
4) *Coordination modules.* Coordinate or synchronize activities between modules.
5) *Composite modules.* Any possible combination of the other four.

This approach to software design, also known as *structured design,* was formalized in the 1970s by IBM researchers [Ste99].

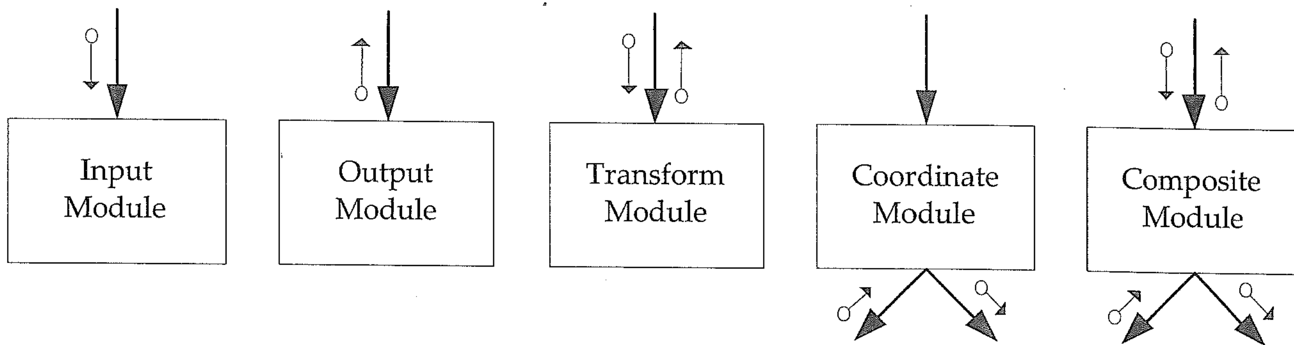

Figure 5.7 Module types for functional software design. The larger arrows indicate connections between modules and the smaller arrows represent data and control.

The following example demonstrates the application of functional decomposition to a software design with the following requirements.

The system must

- Accept an ASCII file of integer numbers as input.
- Sort the numbers into ascending order and save the sorted numbers to disk.
- Compute the mean of the numbers.
- Display the mean on the screen.

This is a fairly simple task that could easily be done in a single function, but doing so would not allow components of the design to be easily reused, tested, or troubleshot. The engineering requirements themselves provide some guidance in terms of how to arrange the functionality of the modules (*form follows function*). The architecture in Figure 5.8 contains a `main` module that calls three submodules. In this design `main` is a coordinating module that controls the processing and calling of the other modules, a common scenario. It was also decided that all user interaction would take place within `main`. The order of the processing is not described by structure charts. In our program, `main` calls `ReadArray`, `SortArray`, and `ComputeMean` in sequential order. `main` passes the filename (`fname`) to `ReadArray`, which reads in the array and the number of elements in it, and returns this information to `main`. The choice of passing in the filename was deliberate; the user could have been prompted for the filename in `ReadArray`, but doing so might limit future reuse of the function since you may not always want to do so when reading an array of data. `SortArray` is then called, which accepts the array of numbers and the number of elements in the array, and returns the sorted values in the same array. Finally, `ComputeMean` is executed, which accepts the sorted array and the number of elements, computes the mean value, and returns it to `main`.

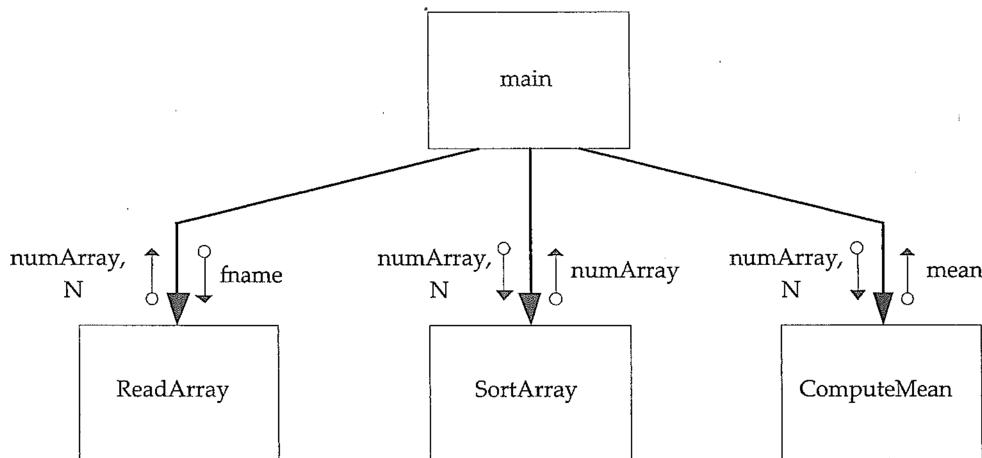

Figure 5.8 Structure chart design of sorting and mean computation program.

The functional requirements for each module in the structure chart are detailed in Table 5.1. The structure chart provides a visual relationship between modules in the design, but also has some disadvantages. It is difficult to visualize designs as the complexity of the software increases. This can be addressed by expanding sublevels in the design as necessary in different diagrams. Structure charts also lack a temporal aspect that indicates the calling order. Most software systems have many layers in the hierarchy and highly complex calling patterns. In this example, `main` calls three modules in a well-defined order, but if there were another level in the hierarchy, there is no reason why it could not be called by a module at any other level. That leads to some of the unique problems associated with software design. Functional design works well for small to moderately complex software, but tends to fall short when applied to large-scale software systems. As such, it has given way to the object-oriented design approach.

5.7 Application: Thermometer Design

The final example includes both analog and digital modules and the objective is to design a thermometer that meets the following engineering requirements.

The system must

- Measure temperature between 0 and 200°C.
- Have an accuracy of 0.4% of full scale.
- Display the temperature digitally, including one digit beyond the decimal point.
- Be powered by a standard 120 V, 60 Hz AC outlet.
- Use an RTD (resistance temperature detector) that has an accuracy of 0.55°C over the range. The resistance of the RTD varies linearly with temperature from 100 Ω at 0°C to 178 Ω at 200°C. (Note: this requirement does not meet the abstractness property identified in Chapter 3, since it identifies part of the solution. This requirement is given to provide guidance in this example.)

Table 5.1 Functional design requirements for the number sort program.

Module name	main()
Module type	Coordination
Input arguments	None.
Output arguments	None.
Description	The main function calls ReadArray() to read the input file from disk, SortArray() to sort the array, and ComputeMean() to determine the mean value of elements in the array. User interaction requires the user to enter the filename, and the mean value is displayed on the screen.
Modules invoked	ReadArray, SortArray, and ComputeMean.

Module name	ReadArray()
Module type	Input and output
Input arguments	- fname[]: character array with filename to read from.
Output Arguments	- numArray[]: integer array with elements read from file. - N: number of elements in numArray[].
Description	Read data from input data file and store elements in array numArray[]. The number of elements read is placed in N.
Modules invoked	None.

Module name	SortArray()
Module type	Transformation
Input arguments	- numArray[]: integer array of numbers. - N: number of elements in numArray[].
Output Arguments	- numArray[]: sorted array of integer numbers.
Description	Sort elements in array using a shell sort algorithm. Saves the sorted array to disk.
Modules invoked	None.

Module name	ComputeMean()
Module type	Input and output
Input arguments	- numArray[]: integer array of numbers. - N: number of elements in numArray[].
Output arguments	- mean: mean value of the elements in the array.
Description	Computes the mean value of the integer elements in the array.
Modules invoked	None.