5.2 Create a high-level state machine for a simple data encryption/decryption device. If a bit-input $b$ is 1, the device stores the data from a 32-bit input $I$ as what is known as an offset value. If $b$ is 0 and another bit-input $e$ is 1, then the device “encrypts” its input $I$ by adding the stored offset value to $I$, and outputs this encrypted value over a 32-bit output $J$. If instead another bit-input $d$ is 1, the device should “decrypt” the data on $I$ by subtracting the offset value before outputting the decrypted value over $J$. Be sure to explicitly handle all possible combinations of the three input bits.

5.3 Create a high-level state machine for a digital bath-water controller. The system has a 3-bit input $\text{ratio}$ indicating the desired ratio of cold water to hot water, and a bit input $\text{on}$ indicating that the water should flow. The system has two 4-bit outputs $\text{hflow}$ and $\text{cflow}$, controlling the hot water flow rate and the cold water flow rate. The sum of these two rates should always equal 16. Your high-level state machine should determine the output values for $\text{hflow}$ and $\text{cflow}$ such that the ratio of hot water to cold water is as close as possible to the desired ratio, while the total flow is always 16. Hint: as there are only 8 possible ratios, a reasonable solution may use one state for each ratio.
5.11 Create an FSM that interfaces with the datapath in Figure 5.95. The FSM should use the datapath to compute the average value of the 16 32-bit elements of an array $A$. Array $A$ is stored in memory, with the first element at address 26, the second at address 27, and so on. Assume that putting a new value onto the address lines $M_{addr}$ causes the memory to almost immediately output the read data on the $M_{data}$ lines. Ignore the possibility of overflow.

![Datapath Diagram](image)

Figure 5.95: Datapath for computing the average of 16 elements of an array.

5.12 Using the RTL design method shown in Table 5.1, create an RTL design of a reaction timer circuit that measures the time elapsed between the illumination a light and the
The reaction timer has three inputs, a clock input \( clk \), a reset input \( rst \), and a button input \( B \), and three outputs, a light enable output \( len \), a 10-bit reaction time output \( rtime \), and a \( slow \) output indicating the user was not fast enough. The reaction timer works as follows. On reset, the reaction timer waits for 10 seconds before illuminating the light by setting \( len \) to 1. The reaction timer then measures the length of time in milliseconds before the user presses the button \( B \), outputting the time as a 12-bit binary number on \( rtime \). If the user did not press the button within 2 seconds (2000 milliseconds), the reaction timer will set the output \( slow \) to 1 and output 2000 on \( rtime \). Assume your clock input has a frequency of 1 kHz.

Design the datapath to structure, but design the controller to an FSM only, as was done in Figure 5.26. Hint: This is a control dominated RTL design problem.

**Inputs:** \( clk, rst, B \) (bit)

**Outputs:** \( len, slow \) (bit); \( rtime \) (11 bits)

**Local Registers:** \( wCount \) (14 bits); \( rCount \) (11 bits)

**Init**

\[ wCount = 0 \]

**Wait**

\[ wCount = wCount + 1 \]

**Count**

\[ wCount < 10000 \]

**Done**

\[ B' \]

**High-Level State Machine**

**Inputs:** \( clk, rst, B, rCount_lt_2000, wCount_lt_10000 \) (bit)

**Outputs:** \( len, slow, wCount, rCount, rtime \) (bit)
5.15 Using the RTL design method shown in Table 5.1, create an RTL design that computes the sum of all numbers positive numbers from a set of 16 separate 32-bit registers storing numbers in two’s complement form. Make the design as fast as possible by performing as many computations concurrently as possible. *Hint:* this is a data-dominated design.
Step 1 - Capture a high-level state machine

Since this problem is a data-dominated design, the problem’s high-level state machine is fairly simple:

Inputs: go (bit), R0...R15 (32 bits)
Outputs: sum (32 bits)

Step 2 - Create a datapath

![Datapath Diagram](image-url)
5.16 Using the RTL design method shown in Table 5.1, create an RTL design that outputs the maximum value found within a register file \( R \) consisting of 64 32-bit numbers.
5.1 Solutions to Exercises

5.1.7 Using the RTL design method shown in Table 5.1, create an RTL design that outputs a warning signal whenever the average temperature over the past four samples exceeds a user defined value. The circuit has a 32-bit input $CT$ indicating the current temperature reading, a 32-bit input $WT$ indicating the user-specified temperature at which the warning should be enabled, and a button input $clr$ that will disable the warning. When the average temperature exceeds the user-specified warning level, the circuit should assert the output $W$ to enable the warning. The warning output should remain high until the $clr$ button is pressed. *Hint:* You can use a right shift to implement the divide within your datapath.
Step 1 - Capture a high-level state machine

Inputs: CT, WT (32 bits); clr (bit)
Outputs: W (bit)
Local Registers: tmp0, tmp1, tmp2, tmp3, avg (32 bits)

Step 2 - Create a datapath
5.18 Using the RTL design method shown in Table 5.1, create an RTL design for a digital filter that outputs the average of the current 32-bit input and the previous 32-bit sample. *Hint:* You can use a right shift to implement the divide within your datapath.
The controller features two levels of gates, resulting in a delay of 4ns. Therefore the critical path is within the up-counter, or 5ns.

With a critical path of 5ns, the maximum clock frequency is $1,000,000,000 / 5 = 200$MHz.

Section 5.5: Behavioral-Level Design: C to Gates (Optional)

5.24 Convert the following C-like code, which calculates the greatest common divisor (GCD) of the two 8-bit numbers $a$ and $b$, into a high-level state machine.

```c
Inputs: byte a, byte b, bit go
Outputs: byte gcd, bit done
GCD:
while(1) {
    while(!go);
    done = 0;
    while ( a != b ) {
        if( a > b ) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
}
```
5.25 Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in Exercise 5.24 to a controller and a datapath. Design the data-path to structure, but design the controller to the point of an FSM only.
5.25 Use the RTL design method shown in Table 5.1 to convert the high-level state machine you created in Exercise 5.24 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.
**Step 1 - Capture a high-level state machine**

The high-level state machine was developed in Exercise 5.24.

**Step 2 - Create a datapath**

**Step 3 - Connect the datapath to a controller**
Step 4 - Derive the controller’s FSM

Inputs: go, done, a_gt_b, a_eq_b (bit)
Outputs: done, a_ld, a_sel, b_ld, b_sel, gcd ld (bit)

5.26 Convert the following C code, which calculates the maximum difference between any two numbers within an array A consisting of 256 8-bit values, into a high-level state machine.

Inputs: byte a[256], bit go
Outputs: byte max_diff, bit done

```
MAX_DIFF:
while(1) {
    while(!go);
    done = 0;
    i = 0;
    max = 0;
    min = 255; // largest 8-bit value
    while( i < 256 ) {
        if( a[i] < min ) {
            min = a[i];
        }
        if( a[i] > max ) {
            max = a[i];
        }
        i = i + 1;
    }
    max_diff = max - min;
    done = 1;
}
```