▶ 5.15 EXERCISES

For each exercise, unless otherwise indicated, assume that the clock frequency is much faster than any input events of interest, and that any inputs have already been debounced. An asterisk (*) indicates an especially challenging problem.

SECTION 5.2: HIGH-LEVEL STATE MACHINES

5.1 Draw a timing diagram to trace the behavior of the soda dispenser HLSM of Figure 5.3 for the case of a soda costing 50 cents and for the following coins being deposited: a dime (10 cents), then a quarter (25 cents), and then another quarter. The timing diagram should show values for all system inputs, outputs, and local storage items, and for the systems' current state.

5.2 Capture the following system behavior as an HLSM. The system counts the number of events on a single-bit input `B` and always outputs that number unsigned on a 16-bit output *C*, which is initially 0. An event is a change from `0` to `1` or from `1` to `0`. Assume the system count rolls over when the maximum value of *C* is reached.

5.3 Capture the following system behavior as an HLSM. The system has two single-bit inputs `U` and `D` each coming from a button, and a 16-bit output *C*, which is initially 0. For each press of `U`, the system increments *C*. For each press of `D`, the system decrements *C*. If both buttons are pressed, the system does not change *C*. The system does not roll over; it goes no higher than than the largest *C* and no lower than *C*=0. A press is detected as a change from `0` to `1`; the duration of that `1` does not matter.

5.4 Capture the following system behavior as an HLSM. A soda machine dispenser system has a 2-bit control input `C1 C0` indicating the value of a deposited coin. `C1C0 = 00` means no coin, `01` means nickel (5 cents), `10` means dime (10 cents), and `11` means quarter (25 cents); when a coin is deposited, the input changes to indicate the value of the coin (for possibly more than one clock cycle) and then changes back to `00`. A soda costs 80 cents. The system displays the deposited amount on a 12-bit output *D*. The system has a single-bit input `S` coming from a button. If the deposited amount is less than the cost of a soda, `S` is ignored. Otherwise, if the button is pressed, the system releases a single soda by setting a single-bit output `R` to `1` for exactly one clock cycle, and the system deducts the soda cost from the deposited amount.

5.5 Create a high-level state machine that initializes a 16x32 register file's contents to 0s, beginning the initialization when an input `rst` becomes `1`. The register file does *not* have a clear input; each register must be individually written with a 0. Do not define 16 states; instead, declare a local storage item so that only a few states need to be defined.

5.6 Create a high-level state machine for a simple data encryption/decryption device. If a single-bit input `b` is `1`, the device stores the data from a 32-bit signed input *I*, referring to this as an *offset* value. If `b` is `0` and another single-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 signed output *J*. If instead another single-bit input `d` is `1`, the device "decrypts" 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.

SECTION 5.3: RTL DESIGN PROCESS

For problems in this section, unless otherwise stated, when converting an HLSM to a controller and datapath, derive the controller's FSM as was done in Figure 5.16 (i.e., don't implement the FSM further), and only use datapath components from the datapath component library of Figure 5.21.

5.7 Create a datapath for the HLSM in Figure 5.98.

5.8 Create a datapath for the HLSM in Figure 5.63.

5.9 For the HLSM in Figure 5.14, complete the RTL design process:
(a) Create a datapath.
(b) Connect the datapath to a controller.
(c) Derive the controller's FSM.

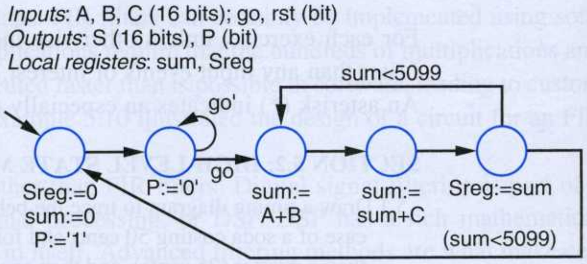
Inputs: A, B, C (16 bits); go, rst (bit)
Outputs: S (16 bits), P (bit)
Local registers: sum, Sreg
sum<5099
go'
go
Sreg:=0
sum:=0
P:='1'
P:='0'
sum:=
A+B
sum:=
sum+C
Sreg:=sum
(sum<5099)

Figure 5.98 Sample high-level state machine.

5.10 Given the HLSM in Figure 5.99, complete the RTL design process to achieve a controller (FSM) connected with a datapath.

5.11 Given the partial HLSM in Figure 5.75 for the system of Figure 5.74, proceed with the RTL design process to achieve a controller (partial FSM) connected with a datapath.

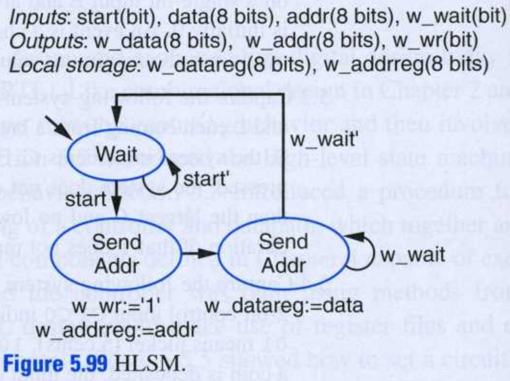
Inputs: start(bit), data(8 bits), addr(8 bits), w_wait(bit)
Outputs: w_data(8 bits), w_addr(8 bits), w_wr(bit)
Local storage: w_datareg(8 bits), w_addrreg(8 bits)
w_wait'
Wait
start'
start
Send
Addr
Send
Addr
w_wait
w_wr:='1'
w_addrreg:=addr
w_datareg:=data

Figure 5.99 HLSM.

5.12 Use the RTL design process to create a 4-bit up-counter with input `cnt` (`1` means count up), clear input `clr`, a terminal count output `tc`, and a 4-bit output *Q* indicating the present count. Only use datapath components from Figure 5.21. After deriving the controller's FSM, implement the controller as a state register and combinational logic.

5.13 Use the RTL design process to design a system that outputs the average of the most recent two data input samples. The system has an 8-bit unsigned data input *I*, and an 8-bit unsigned output *avg*. The data input is sampled when a single-bit input `S` changes from `0` to `1`. Choose internal bitwidths that prevent overflow.

5.14 Use the RTL design process to create an alarm system that sets a single-bit output `alarm` to 1 when the average temperature of four consecutive samples meets or exceeds a user-defined threshold value. A 32-bit unsigned input *CT* indicates the current temperature, and a 32-bit unsigned input *WT* indicates the warning threshhold. Samples should be taken every few clock cycles. A single-bit input `clr` when `1` disables the alarm and the sampling process. Start by capturing the desired system behavior as an HLSM, and then convert to a controller/datapath.

5.15 Use the RTL design process to design a reaction timer system that measures the time elapsed between the illumination of a light and the pressing of a button by a user. The reaction timer has three inputs, a clock input *clk*, a reset input *rst*, and a button input *B*. It has three outputs, a light enable output *len*, a 10-bit reaction time output *rtime*, and a *slow* output indicating that 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 that the clock input has a frequency of 1 kHz. Do not use a timer component in the datapath.

SECTION 5.4: MORE RTL DESIGN

For problems in this section, unless otherwise indicated, allowable datapath components are from Figure 5.21 and Figure 5.27, and controller design can end after deriving the FSM. Use the RTL design process for problems that state the need to "design" a system.

5.16 Create an FSM that interfaces with the datapath in Figure 5.100. The FSM should use the datapath to compute the average value of the 16 32-bit elements of any array *A*. Array *A* is stored in a memory, with the first element at address 25, the second at address 26, 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 overflow issues.

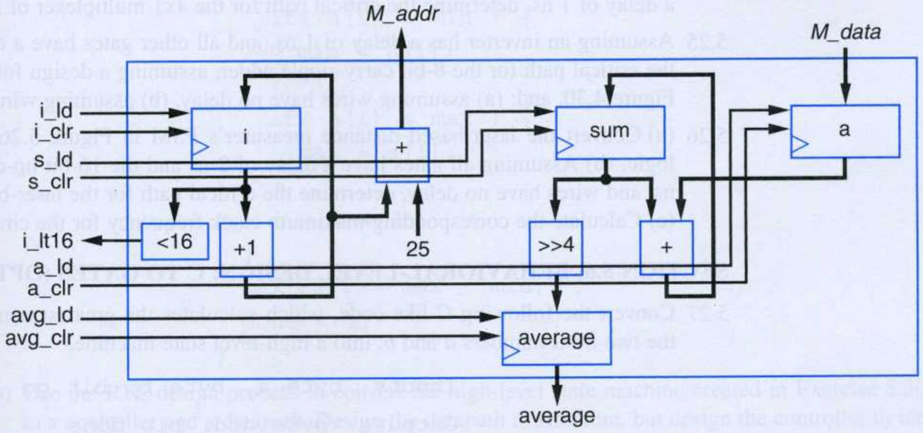

Figure 5.100 Datapath capable of computing the average of 16 elements of an array.

5.17 Design a system that repeatedly computes and outputs the sum of all *positive* numbers within a 512-word register file *A* consisting of 32-bit signed numbers.

5.18 Design a system that repeatedly computes and outputs the *maximum* value found within a register file *A* consisting of 64 32-bit unsigned numbers.

5.19 Using a timer, design a system with single-bit inputs U and D corresponding to two buttons, and a 16-bit output *Q* which is initially 0. Pressing the button for U causes *Q* to increment, while D causes a decrement; pressing both buttons causes *Q* to stay the same. If a single button is held down, *Q* should then continue to increment or decrement at a rate of once per second as long as the button is held. Assume the buttons are already debounced. Assume Q simply rolls over if its upper or lower value is reached.

5.20 Using a timer, design a display system that reads the ASCII characters from a 64-word 8-bit register file *RF* and writes each word to a 2-row LED-based display having 32 characters per row, doing so 100 times per second. The display has an 8-bit input *A* for the ASCII character to be displayed, a single-bit input `row` where `0` or `1` denotes the top or bottom row respectively, a 5-bit input *col* that indicates a column in the row, and an enable input `en` whose change from `0` to `1` causes the character to be displayed in the given row and column. The system should write *RF*[0] through *RF*[15] to row 0's columns 0 to 15 respectively, and *RF*[16] to *RF*[31] to row 1.

5.21 Design a data-dominated system that computes and outputs the sum of the absolute values of 16 separate 32-bit registers (*not* in a register file) storing signed numbers (do not consider how those numbers get stored). The computation of the sum should be done using a single equation in one state. The computation should be performed once when a single-bit input go changes from 0 to 1, and the computed result should be held at the output until the next time go changes from 0 to 1.

SECTION 5.5: DETERMINING CLOCK FREQUENCY

5.22 Assuming an inverter has a delay of 1 ns, all other gates have a delay of 2 ns, and wires have a delay of 1 ns, determine the critical path for the full-adder circuit in Figure 4.30.

5.23 Assuming an inverter has a delay of 1 ns, all other gates have a delay of 2 ns, and wires have a delay of 1 ns, determine the critical path for the 3x8 decoder of Figure 2.62.

5.24 Assuming an inverter has a delay of 1 ns, all other gates have a delay of 2 ns, and wires have a delay of 1 ns, determine the critical path for the 4x1 multiplexer of Figure 2.67.

5.25 Assuming an inverter has a delay of 1 ns, and all other gates have a delay of 2 ns, determine the critical path for the 8-bit carry-ripple adder, assuming a design following Figure 4.31 and Figure 4.30, and: (a) assuming wires have no delay, (b) assuming wires have a delay of 1 ns.

5.26 (a) Convert the laser-based distance measurer's FSM in Figure 5.26 to a state register and logic. (b) Assuming all gates have a delay of 2 ns and the 16-bit up-counter has a delay of 5 ns, and wires have no delay, determine the critical path for the laser-based distance measurer. (c) Calculate the corresponding maximum clock frequency for the circuit.

SECTION 5.6: BEHAVIORAL-LEVEL DESIGN: C TO GATES (OPTIONAL)

5.27 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.

```
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;
      }
   }
   gcd = a;
   done = 1;
}
```

5.28 Use the RTL design process to convert the high-level state machine created in Exercise 5.27 to a controller and a datapath. Design the datapath to structure, but design the controller to an FSM and then stop.

5.29 Convert the following C-like 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;
}
```

5.30 Use the RTL design process to convert the high-level state machine created in Exercise 5.30 to a controller and a datapath. Design the datapath to structure, but design the controller to tan FSM and then stop.

5.31 Convert the following C-like code, which calculates the number of times the value *b* is found within an array *A* consisting of 256 8-bit values, into a high-level state machine.

```
Inputs: byte a[256], byte b, bit go
Outputs: byte freq, bit done
FREQUENCY:
while(1) {
   while(!go);
   done = 0;
   i = 0;
   freq = 0;
   while( i < 256 ) {
      if( a[i] == b ) {
         freq = freq + 1;
      }
   }
   done = 1;
}
```

5.32 Use the RTL design process to convert the high-level state machine you created in Exercise 5.31 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

5.33 Develop a template for converting a `do{ }while` loop of the following form to a high-level state machine.

```
do {
  // do while statements
} while (cond);
```

5.34 Develop a template for converting a `for()` loop of the following form to a high-level state machine.

```
for(i=start; i<cond; i++)
{
   // for statements
}
```

5.35 Compare the time required to execute the following computation using a custom circuit versus using a microprocessor. Assume a gate has a delay of 1 ns. Assume a microprocessor executes one instruction every 5 ns. Assume that n=10 and m=5. Estimates are acceptable; you need not design the circuit, or determine exactly how many software instructions will execute.

```
for (i = 0; i<n, i++) {
   s = 0;
   for (j = 0; j < m, j++) {
      s = s + c[i]*x[i + j];
   }
   y[i] = s;
}
```

SECTION 5.7: MEMORY COMPONENTS

5.36 Calculate the approximate number of DRAM bit storage cells that will fit on an IC with a capacity of 10 million transistors.

5.37 Calculate the approximate number of SRAM bit storage cells that will fit on an IC with a capacity of 10 million transistors.

5.38 Summarize the main differences between DRAM and SRAM memories.

5.39 Draw a circuit of transistors showing the internal structure for all the storage cells for a 4x2 DRAM (four words, 2 bits each), clearly labeling all internal components and connections.

5.40 Draw a circuit of transistors showing the internal structure for all the storage cells for a 4x2 SRAM (four words, 2 bits each), clearly labeling all internal components and connections.

5.41 Summarize the main differences between EPROM and EEPROM memories.

5.42 Summarize the main differences between EEPROM and flash memories.

5.43 Use an HLSM to capture the design of a system that can save data samples and then play them back. The system has an 8-bit input D where data appears. A single-bit input S changing from 0 to 1 requests that the current value on D (i.e., a sample) be saved in a nonvolatile memory. Sample requests will not arrive faster than once per 10 clock cycles. Up to 10,000 samples can be saved, after which sampling requests are ignored. A single-bit input P changing from 0 to 1 causes all recorded samples to be played back—i.e., to be written to an output Q one sample at a time in the order they were saved at a rate of one sample per clock cycle. A single-bit input R resets the system, clearing all recorded samples. During playback, any sample or reset request is ignored. At other times, reset has priority over a sample request. Choose an appropriate size and type of memory, and declare and use that memory in your HLSM.

SECTION 5.8: QUEUES (FIFOS)

5.44 For an 8-word queue, show the queue's internal state and provide the value of popped data for the following sequences of pushes and pops: (1) push A, B, C, D, E, (2) pop, (3) pop, (4) push U, V, W, X, Y, (5) pop, (6) push Z, (7) pop, (8) pop, (9) pop.

5.45 Create an FSM describing the queue controller of Figure 5.79. Pay careful attention to correctly setting the `full` and `empty` outputs.

5.46 Create an FSM describing the queue controller of Figure 5.79, but with error-preventing behavior that ignores any pushes when the queue is full, and ignores pops of an empty queue (outputting 0).

SECTION 5.9: MULTIPLE PROCESSORS

5.47 A system *S* counts people that enter a store, incrementing the count value when a single-bit input `P` changes from `1` to `0`. The value is reset when `R` is `1`. The value is output on a 16-bit output *C*, which connects to a display. Furthermore, the system has a lighting system to indicate the approximate count value to the store manager, turning on a red LED (`LR=1`) for 0 to 99, else a blue LED (`LB=1`) for 100 to 199, else a green LED (`LG=1`) for 200 and above. Draw a block diagram of the system and its peripheral components, using *two* processors for the system *S*. Show the HLSM for each processor.

5.48 A system *S* counts the cycles high of the most recent pulse on a single-bit input `P` and displays the value on a 16-bit output *D*, holding the value there until the next pulse completes. The system also keeps track of the previous 8 values, and computes and outputs the average of those values on a 16-bit output *A* whenever an input `C` changes from `0` to `1`. The system holds that output value until the next change of `C` from `0` to `1`. Draw a block diagram of the system and its peripheral components, using two processors and a global register file for the system. Show the HLSM for each processor.

5.49 A keypad needs to be interfaced with a computer. The keypad has a 4-bit output *K* representing the encoding of the key that was pressed and a single-bit output `E` that changes from `0` to `1` when a key is pressed. The computer has a corresponding 4-bit input *CK* and single-bit input `CE`. However, sometimes the computer is busy with other tasks and takes some time to receive the key, so it has an output `Crec` that it sets to `1` for one clock cycle when the key value has been received. Design a systems *S* in between the keypad and computer that can buffer up to 32 key values while the computer is busy. Show a block diagram of *S* composed of two processors and a queue, along with interfaces to the keypad and computer, and show HLSMs for each processor.

SECTION 5.10: HIERARCHY—A KEY DESIGN CONCEPT

5.50 Compose a 20-input AND gate from 2-input AND gates.

5.51 Compose a 16x1 mux from 2x1 muxes.

5.52 Compose a 4x16 decoder with enable from 2x4 decoders with enable.

5.53 Compose a 1024x8 RAM using only 512x8 RAMs.

5.54 Compose a 512x8 RAM using only 512x4 RAMs.

5.55 Compose a 1024x8 ROM using only 512x4 ROMs.

5.56 Compose a 2048x8 ROM using only 256x8 ROMs.

5.57 Compose a 1024x16 RAM using only 512x8 RAMs.

5.58 Compose a 1024x12 RAM using 512x8 and 512x4 RAMs.

5.59 Compose a 640x12 RAM using only 128x4 RAMs.

5.60 *Write a program that takes a parameter *N*, and automatically builds an N-input AND gate from 2-input AND gates. Your program merely needs to indicate how many 2-input AND gates exist in each level, from which one could easily determine the connections.

▶ DESIGNER PROFILE

Chi-Kai started college as an engineering major, and became a Computer Science major due to his developing interests in algorithms and in networks. After graduating, he worked for a Silicon Valley startup company that made chips for computer networking. His first task was to help simulate those chips before the chips were built. For over 10 years now, he has worked on multiple generations of networking devices that buffer, schedule, and switch ATM network cells and Internet Protocol packets. "The chips required to implement networking devices are complex components that must all work together almost perfectly to provide the building blocks of telecommunication and data networks. Each generation of devices becomes successively more complex."

When asked what skills are necessary for his job, Chi-Kai says "More and more, breadth of one's skill set matters more than depth. Being an effective chip engineer requires the ability to understand chip architecture (the big picture), to design logic, to verify logic, and to bring up the silicon in the lab. All these parts of the design cycle interplay more and more. To be truly effective at one particular area requires hands-on knowledge of the others as well. Also, each requires very different skills. For example, verification requires good software programming ability, while bring up requires knowing how to use a logic analyzer—good hardware skills."

High-end chips, like those involved in networking, are quite costly, and require careful design. "The software design process and the chip design process are fundamentally different. Software can afford to have bugs because patches can be applied. Silicon is a different story. The one-time expenses to spin a chip are on the order of $500,000. If there is a show-stopping bug, you may need to spend another $500,000. This constraint means the verification approach taken is quite different—effectively: there can be no bugs." At the same time, these chips must be designed quickly to beat competitors to the market, making the job "extremely challenging and exciting."

One of the biggest surprises Chi-Kai encountered in his job is the "incredible importance of good communication skills." Chi-Kai has worked in teams ranging from 10 people to 30 people, and some chips require teams of over 100 people. "Technically outstanding engineers are useless unless they know how to collaborate with others and disseminate their knowledge. Chips are only getting more complex—individual blocks of code in a given chip have the same complexity as an entire chip only a few years ago. To architect, design, and implement logic in hardware requires the ability to convey complexity." Furthermore, Chi-Kai points out that "just like any social entity, there are politics involved. For example, people are worried about aspirations for promotion, financial gain, and job security. In this greater context, the team still must work together to deliver a chip." So, contrary to the conceptions many people have of engineers, engineers must have excellent people skills, in addition to strong technical skills. Engineering is a social discipline.