CHAPTER 5

REGISTER-TRANSFER LEVEL (RTL) DESIGN

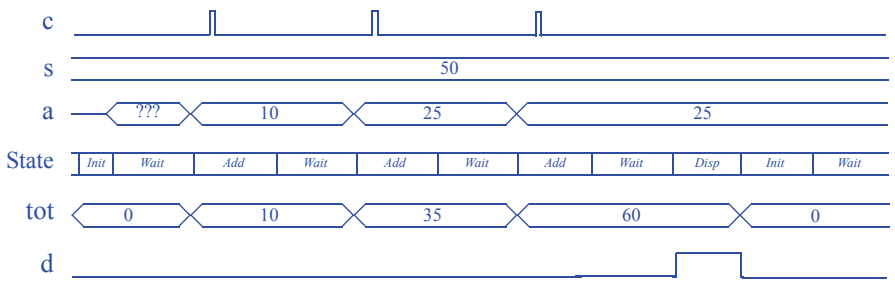
5.1 EXERCISES

For each exercise, unless otherwise indicated, assume that the clock frequency is much faster than any input events of interest, and that any button inputs have been debounced. Problems noted with an asterisk (*) represent especially challenging problems.

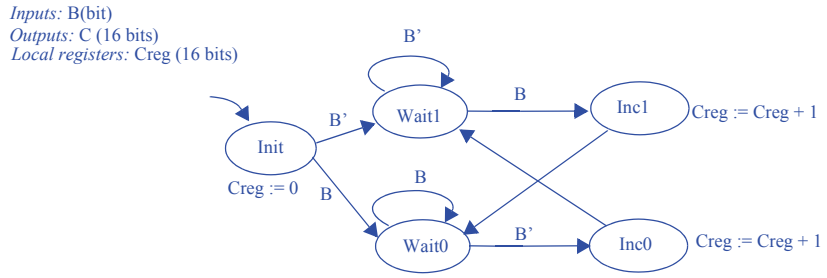
Section 5.2: High-Level State Machines

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.

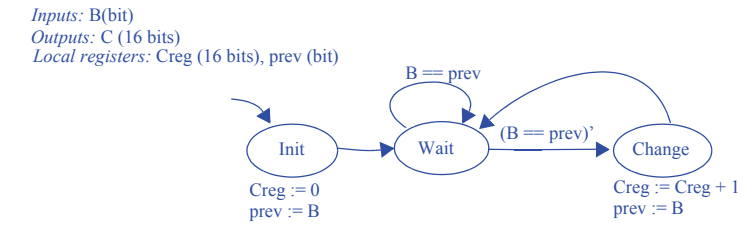
Note: figure not drawn to scale



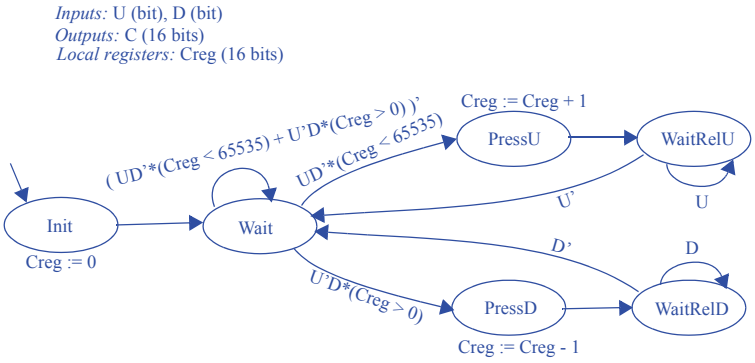
5.2 Capture the following system behavior as an HLSM. The system counts the number of events on a single-bit input `B` and always outputs that number unsigned on a 16-bit output C, which is initially 0. An event is a change from `0` to `1` or from `1` to `0`. Assume the system count rolls over when the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



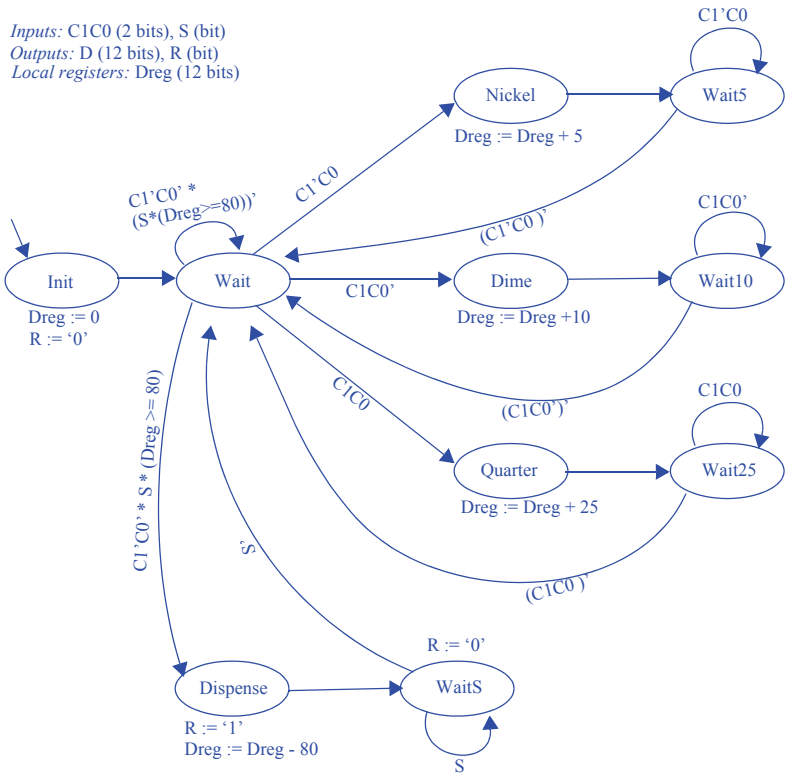
Alternative solution:



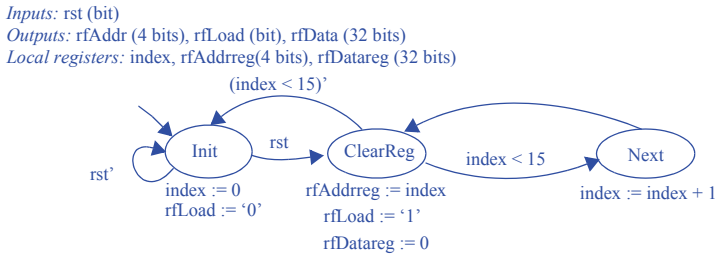
5.3 Capture the following system behavior as an HLSM. The system has two single-bit inputs `U` and `D` each coming from a button, and a 16-bit output C, which is initially 0. For each press of `U`, the system increments C. For each press of `D`, the system decrements C. If both buttons are pressed, the 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.



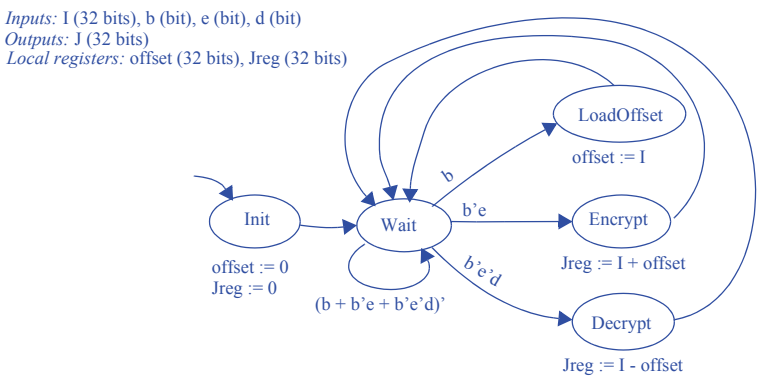
5.4 Capture the following system behavior as an HLSM. A soda machine dispenser system has a 2-bit control input `C1 C0` indicating the value of a deposited coin. `C1C0` = `00` means no coin, `01` means nickel (5 cents), `10` means dime (10 cents), and `11` means quarter (25 cents); when a coin is deposited, 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.



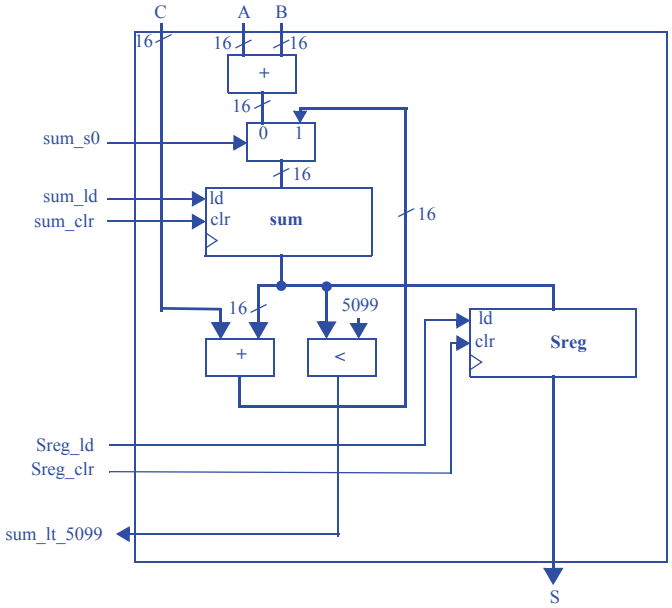
5.6 Create a high-level state machine for a simple data encryption/decryption device. If a single-bit input `b` is `1`, the device stores the data from a 32-bit signed input I, referring to this as an offset value. If `b` is `0` and another single-bit input `e` is `1`, then the device "encrypts" its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



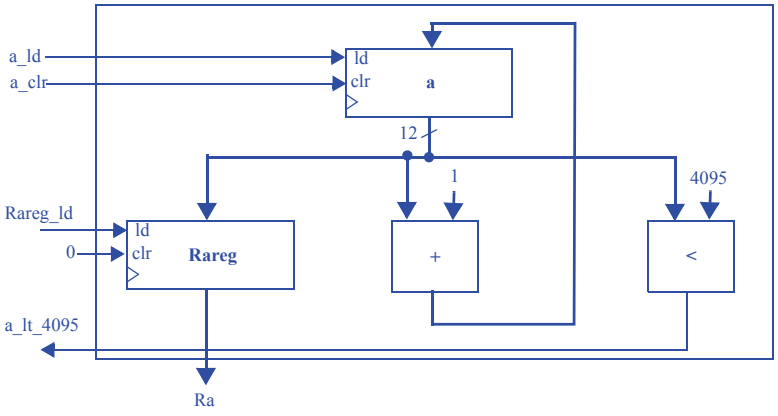
Section 5.3: RTL Design Process

5.7 Create a datapath for the HLSM in Figure 5.98.

(Note that "P" is not involved in the datapath; it will be a controller output.)



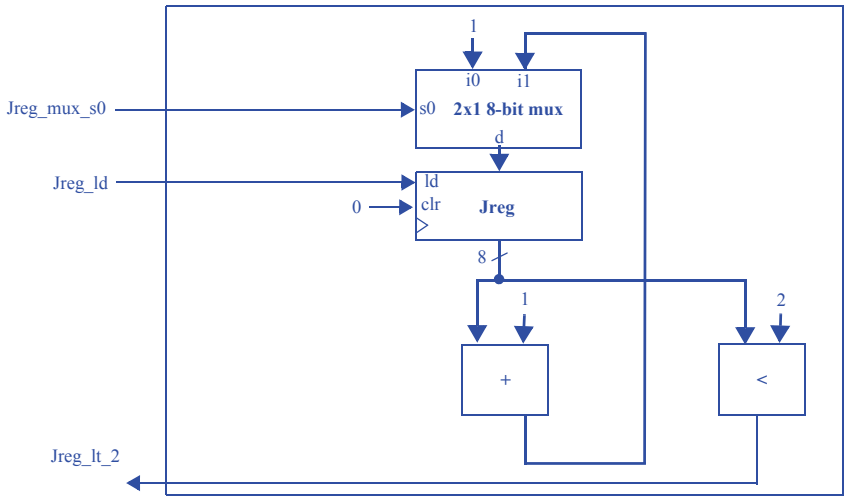
5.8 Create a datapath for the HLSM in Figure 5.63.



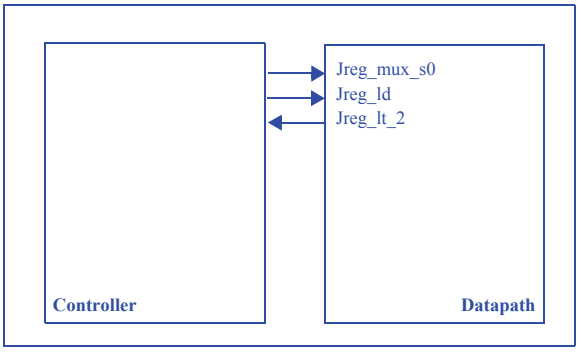
5.9 For the HLSM in Figure 5.14, complete the RTL design process:

a. Create a datapath.
b. Connect the datapath to a controller.
c. Derive the controller's FSM.

a) Create a datapath.

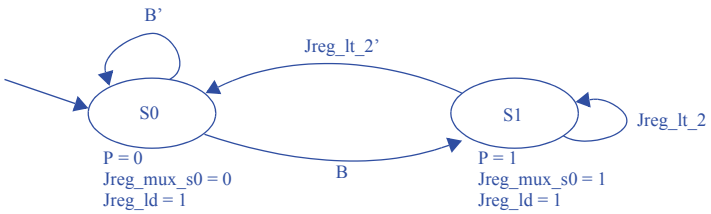


b) Connect the datapath to a controller.

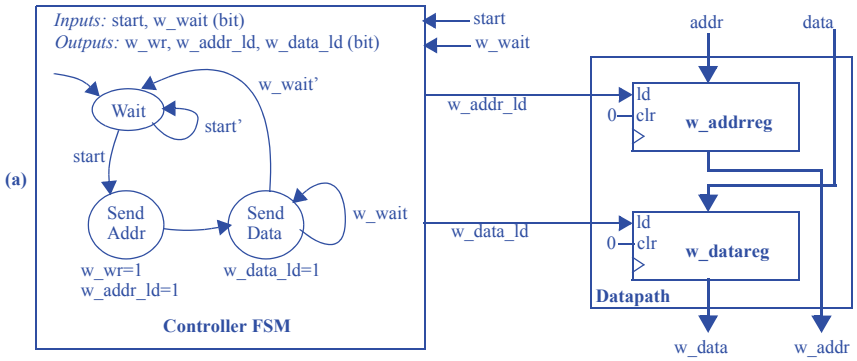


c) Derive the controller's FSM.

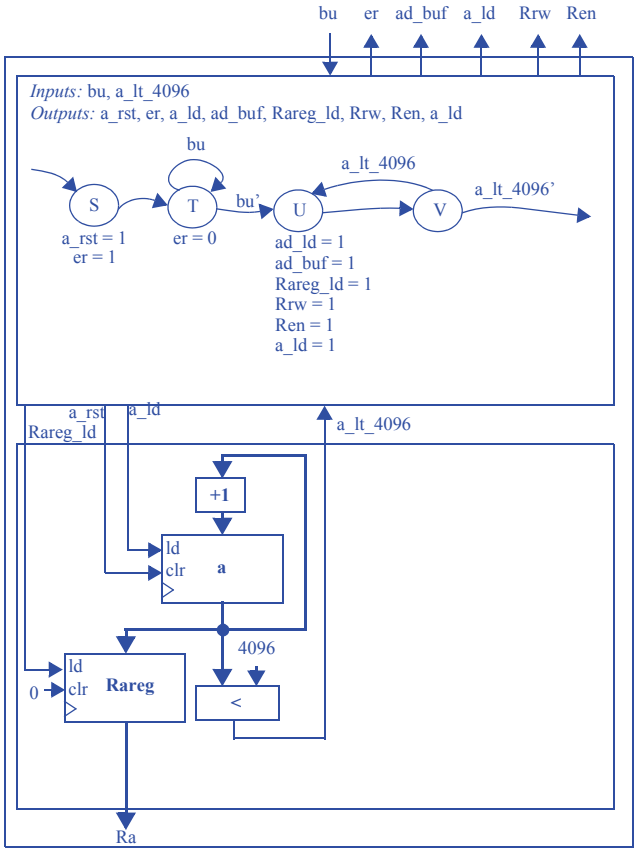
Inputs: B, Jreg_lt_2
Outputs: P, Jreg_mux_s0, Jreg_ld



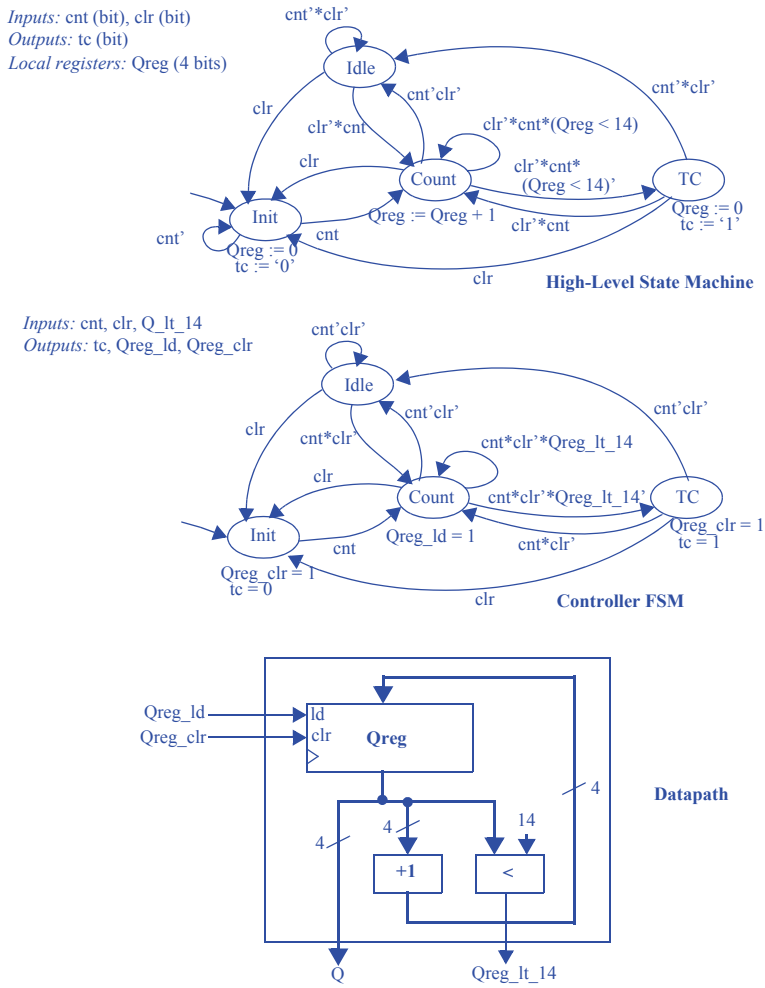
5.10 Given the HLSM in Figure 5.99, complete the RTL design process to achieve a controller (FSM) connected with a datapath.



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.



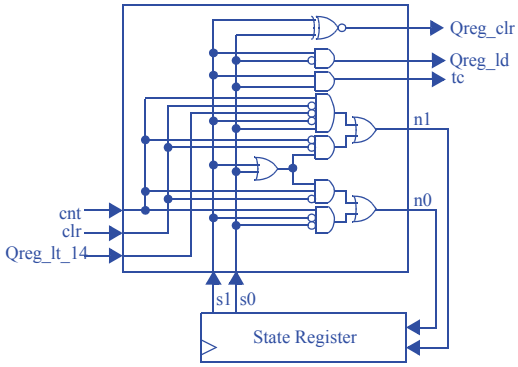
5.12 Use the RTL design process to create a 4-bit up-counter with input `cnt` (`1` means count up), clear input `clr`, a terminal count output `tc`, and a 4-bit output Q indicating the present count. Only use datapath components from Figure 5.21. After deriving the controller's FSM, implement the controller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



	Inputs					Outputs				
	s1	s0	cnt	clr	Qreg_lt_14	n1	n0	tc	Qreg_ld	Qreg_clr
Init	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	1	0	0	0	0	1
	0	0	0	1	0	0	0	0	0	1
	0	0	0	1	1	0	0	0	0	1
	0	0	1	0	0	0	1	0	0	1
	0	0	1	0	1	0	1	0	0	1
	0	0	1	1	0	0	1	0	0	1
	0	0	1	1	1	0	1	0	0	1
Count	0	1	0	0	0	1	0	0	1	0
	0	1	0	0	1	1	0	0	1	0
	0	1	0	1	0	0	0	0	1	0
	0	1	0	1	1	0	0	0	1	0
	0	1	1	0	0	1	1	0	1	0
	0	1	1	0	1	0	1	0	1	0
	0	1	1	1	0	0	0	0	1	0
	0	1	1	1	1	0	0	0	1	0
Idle	1	0	0	0	0	1	0	0	0	0
	1	0	0	0	1	1	0	0	0	0
	1	0	0	1	0	0	0	0	0	0
	1	0	0	1	1	0	0	0	0	0
	1	0	1	0	0	0	1	0	0	0
	1	0	1	0	1	0	1	0	0	0
	1	0	1	1	0	0	0	0	0	0
	1	0	1	1	1	0	0	0	0	0
TC	1	1	0	0	0	1	0	1	0	1
	1	1	0	0	1	1	0	1	0	1
	1	1	0	1	0	0	0	1	0	1
	1	1	0	1	1	0	0	1	0	1
	1	1	1	0	0	0	1	1	0	1
	1	1	1	0	1	0	1	1	0	1
	1	1	1	1	0	0	0	1	0	1
	1	1	1	1	1	0	0	1	0	1

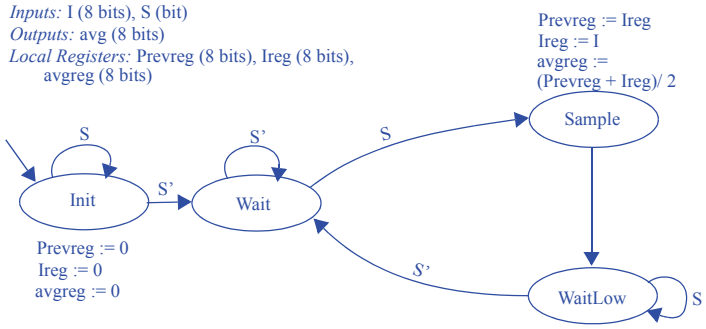
n1 = (s1 + s0)cnt'clr' + s1's0*cnt*clr'Qreg_lt_14'
n0 = s1's0'cnt + (s1 + s0)cnt*clr'
tc = s1s0
Qreg_ld = s1's0

Qreg_clr = s1’s0’ + s1s0

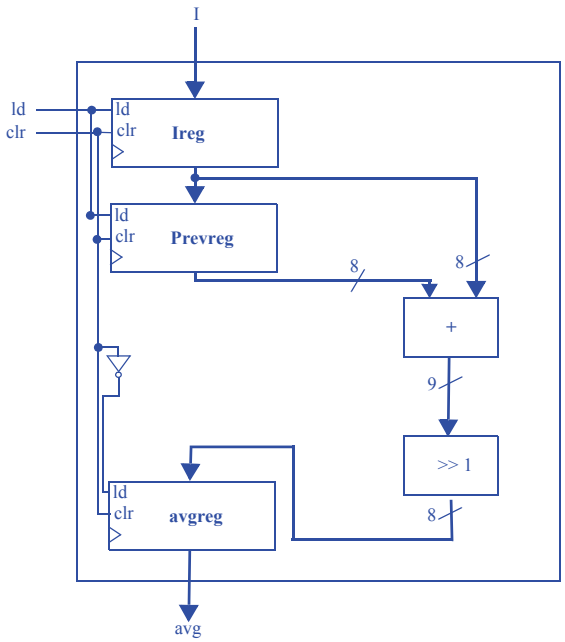


5.13 Use the RTL design process to design a system that outputs the average of the most recent two data input samples. The system has an 8-bit unsigned data input I, and an 8-bit unsigned output avg. The data input is sampled when a single-bit input `S` changes from `0` to `1`. Choose internal bitwidths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

Step 1 - Capture a high-level state machine

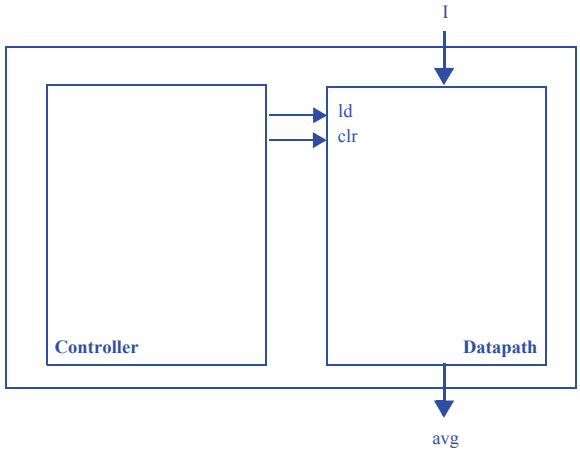


Step 2 - Create a datapath



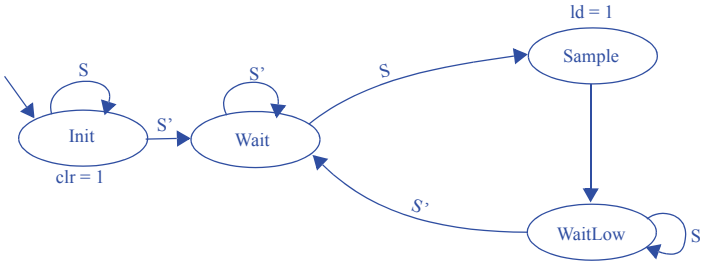
Note: A solution more consistent with the chapter's methdology would use a separate clear and ld signal for each register. In this particular example, a single clr and a single load line happens to work.

Step 3 - Connect the datapath to a controller



Step 4 - Derive the controller's FSM

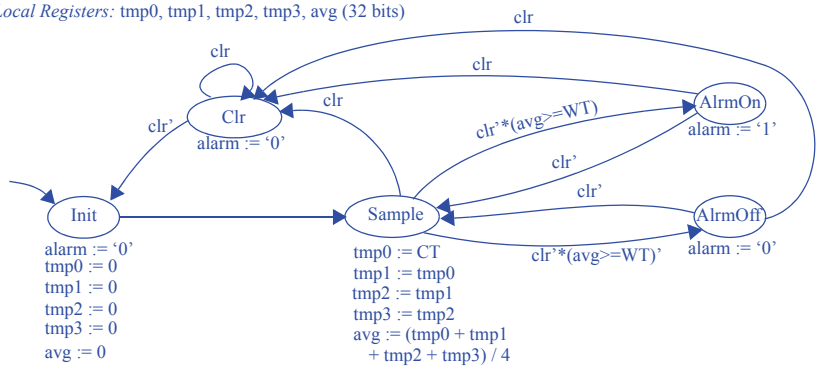
Inputs: S
Outputs: ld, clr



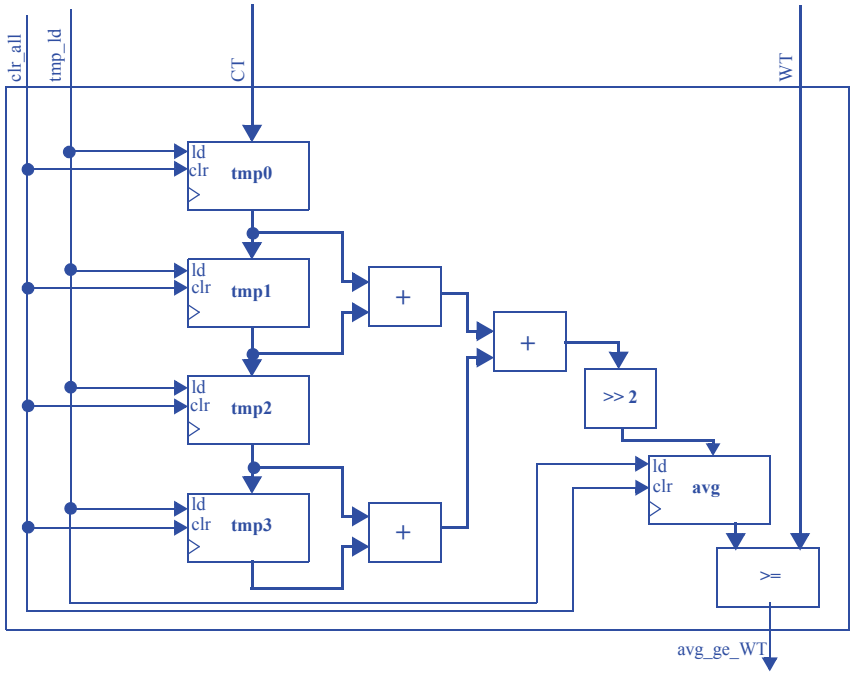
5.14 Use the RTL design process to create an alarm system that sets a single-bit output `alarm` to `1` when the average temperature of four consecutive samples meets or exceeds a user-defined threshold value. A 32-bit unsigned input CT indicates the current temperature, and a 32-bit unsigned input WT indicates the warning threshold. 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.

Step 1 - Capture a high-level state machine

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

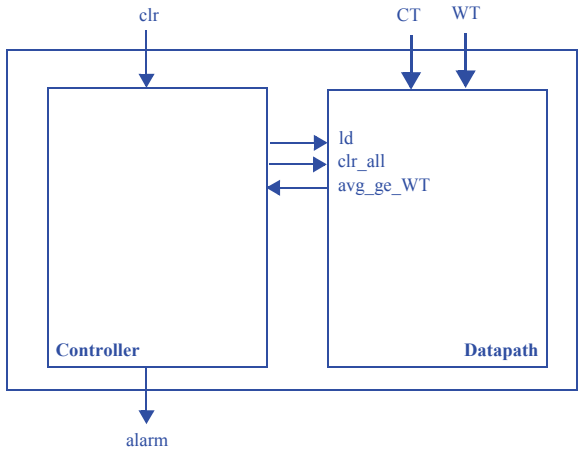


Step 2A - Create a datapath



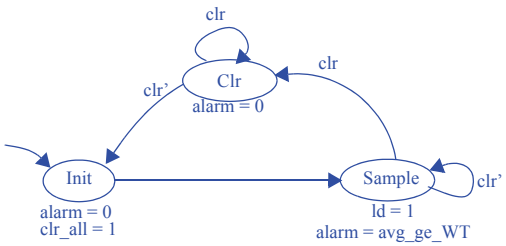
Note: A solution more consistent with the chapter's methdology would use a separate clear and ld signal for each register. In this particular example, a single clr and a single load line happens to work.

Step 2B- Connect the datapath to a controller



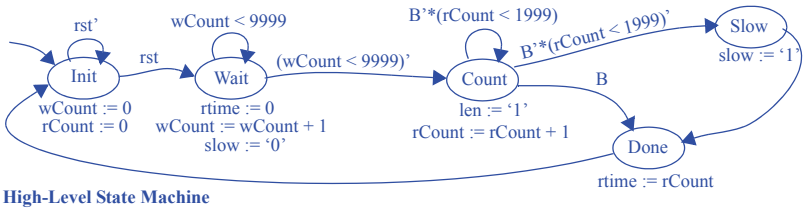
Step 2C - Derive the controller's FSM

Inputs: clr, avg_lt_WT
Outputs: alarm, clr_all, ld

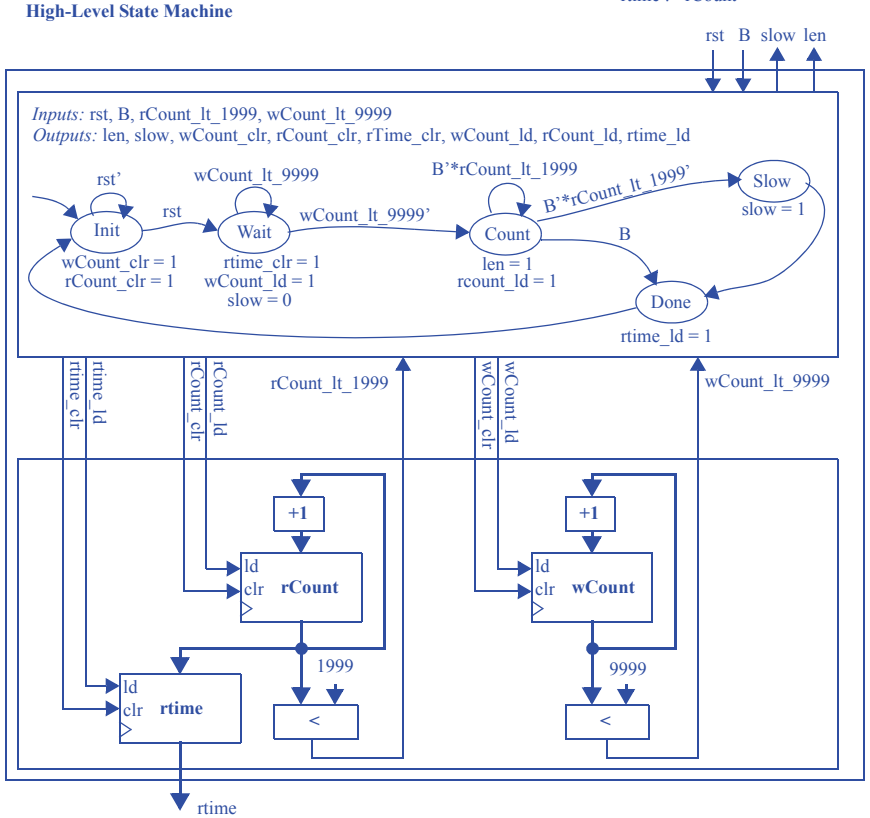


5.15 Use the RTL design process to design a reaction timer system that measures the time elapsed between the illumination of a light and the pressing of a button by a user. The reaction timer has three inputs, a clock input clk, a reset input rst, and a button input B. It has three outputs, a light enable 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.

Inputs: rst, B (bit)
Outputs: len, slow (bit); rtime (11 bits)
Local Registers: wCount (14 bits); rCount (11 bits)

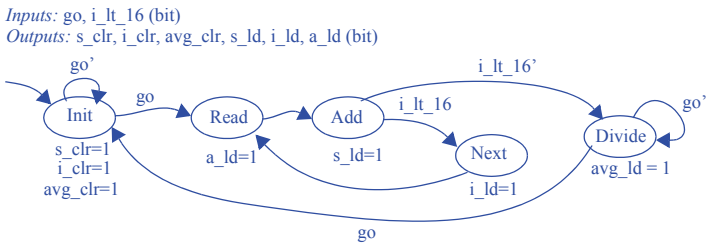


High-Level State Machine



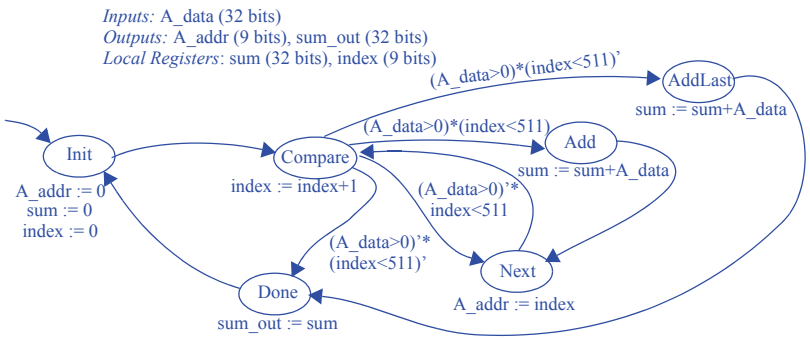
Section 5.4: More RTL Design

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.

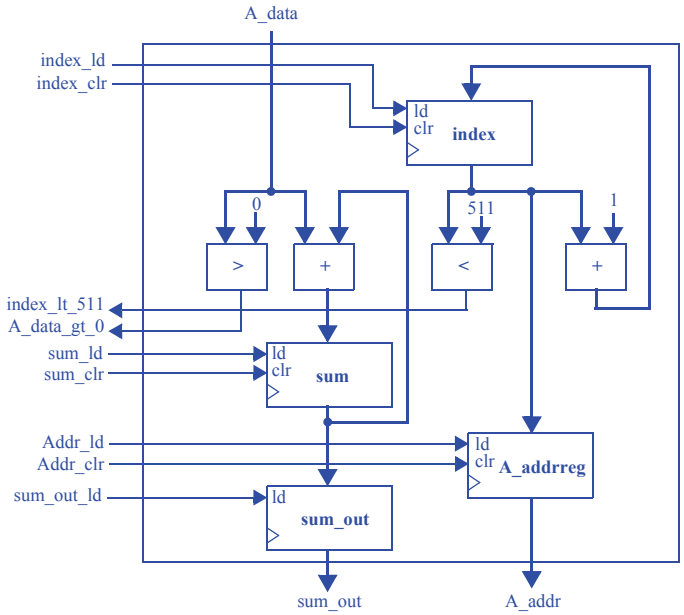


5.17 Design a system that repeatedly computes and outputs the sum of all positive numbers within a 512-word register file A consisting of 32-bit signed numbers.

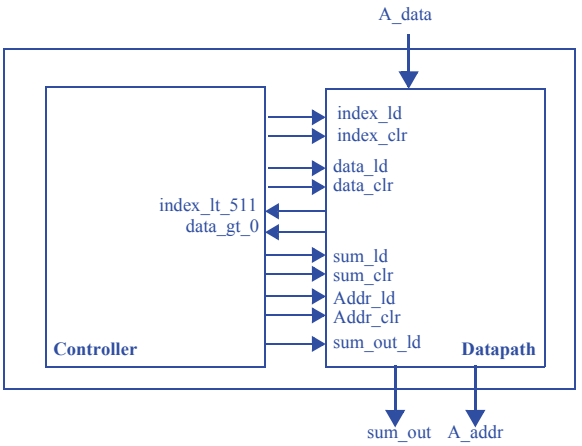
Step 1 - Capture a high-level state machine



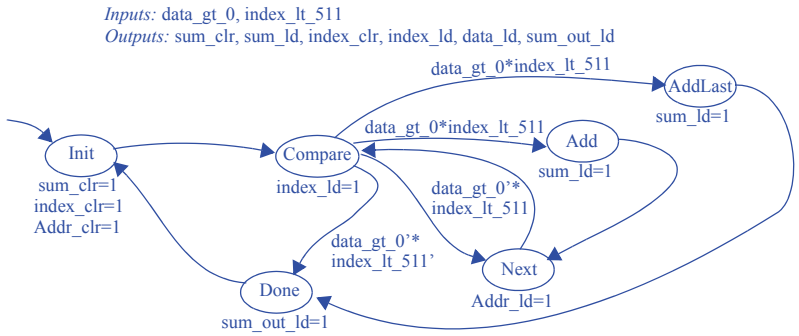
Step 2A - Create a datapath



Step 2B - Connect the datapath to a controller

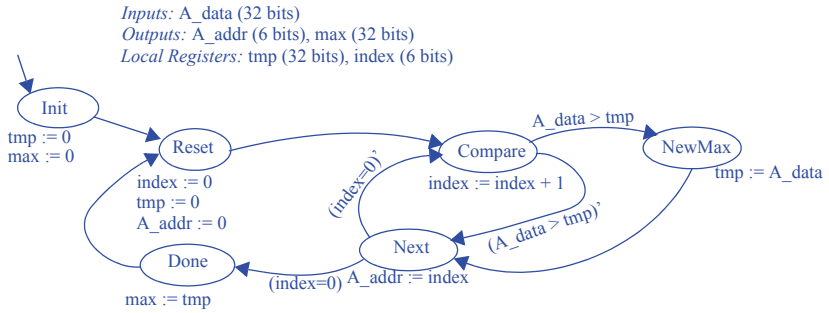


Step 2C - Derive the controller's FSM

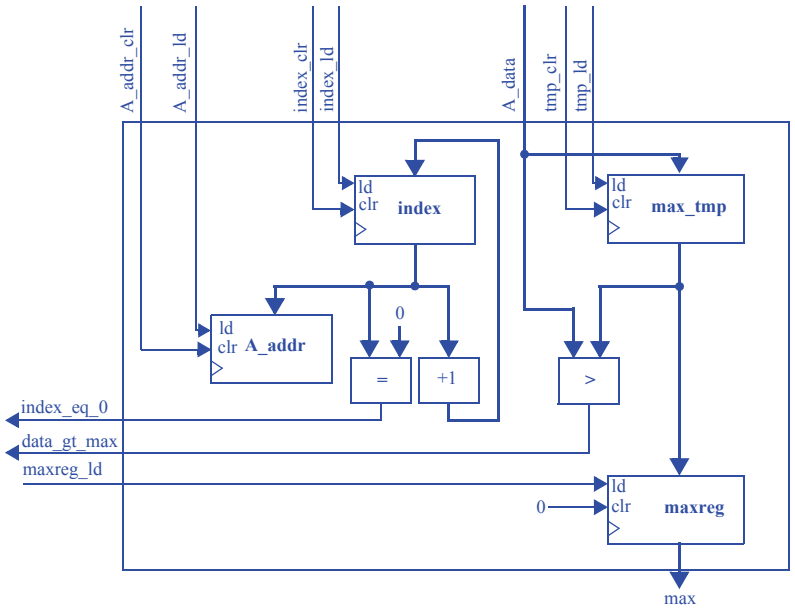


5.18 Design a system that repeatedly computes and outputs the maximum value found within a register file A consisting of 64 32-bit unsigned numbers.

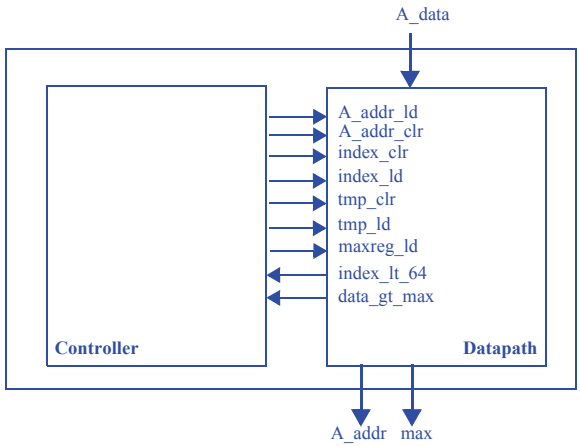
Step 1 - Capture a high-level state machine



Step 2A - Create a datapath

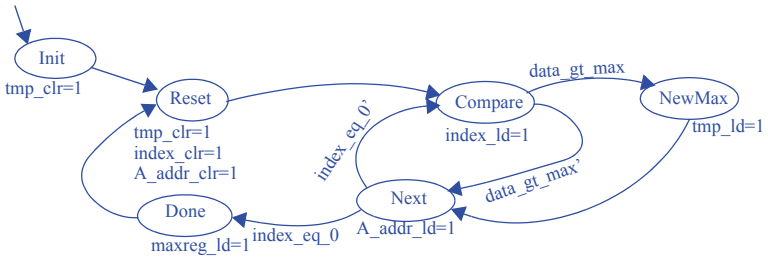


Step 2B - Connect the datapath to a controller



Step 2C - Derive the controller's FSM

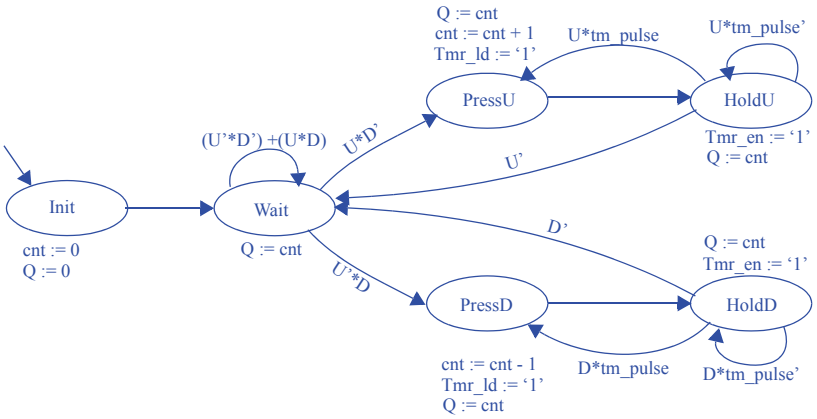
Inputs: index_eq_0, data_gt_max
Outputs: A_addr_ld, A_addr_clr, index_clr, index_ld, tmp_clr, tmp_ld, maxreg_ld



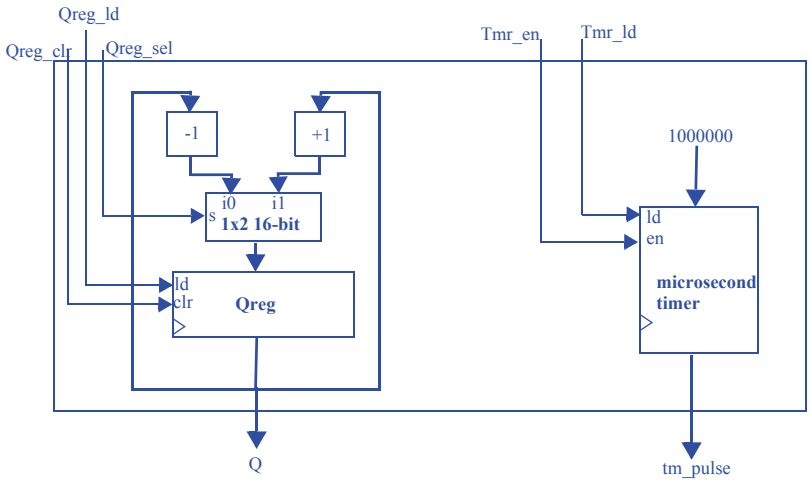
5.19 Using a timer, design a system with single-bit inputs `U` and `D` corresponding to two buttons, and a 16-bit output Q which is initially 0. Pressing the button for `U` causes Q to increment, while `D` causes a decrement; pressing both buttons causes Q to stay the same. If a single button is held 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.

Step 1 - Capture a high-level state machine

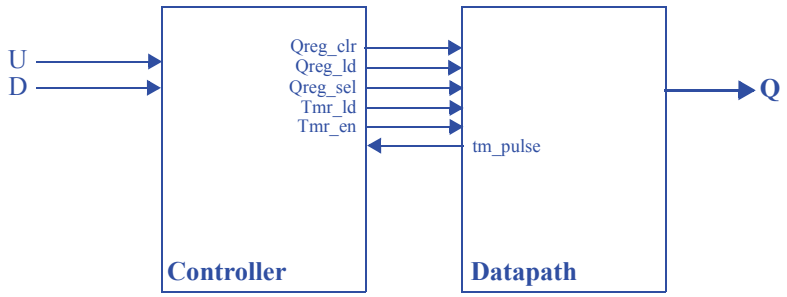
Inputs: U, D, tm_pulse (bit)
Outputs: Q (16 bits), Tmr_ld, Tmr_en (bit)
Local Registers: cnt (16 bits)



Step 2A - Create a datapath

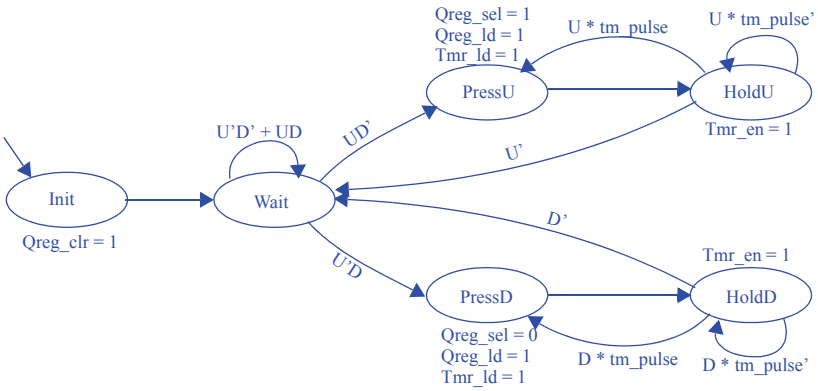


Step 2B - Connect the datapath to a controller



Step 2C - Derive the controller's FSM

Inputs: U, D, tm_pulse
Outputs: Qreg_clr, Qreg_ld, Qregsel, Tmr_ld, Tmr_en



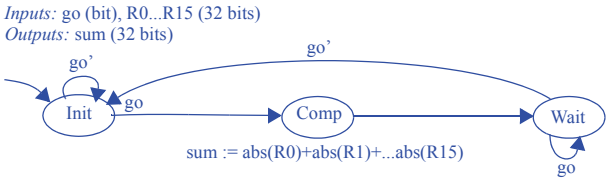
5.20 Using a timer, design a display system that reads the ASCII characters from a 64-word 8-bit register file RF and writes each word to a 2-row LED-based display having 32 characters per row, doing so 100 times per second. The display has an 8-bit input A for the ASCII character to be displayed, a 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.

Do not assign this exercise; it contains an error.

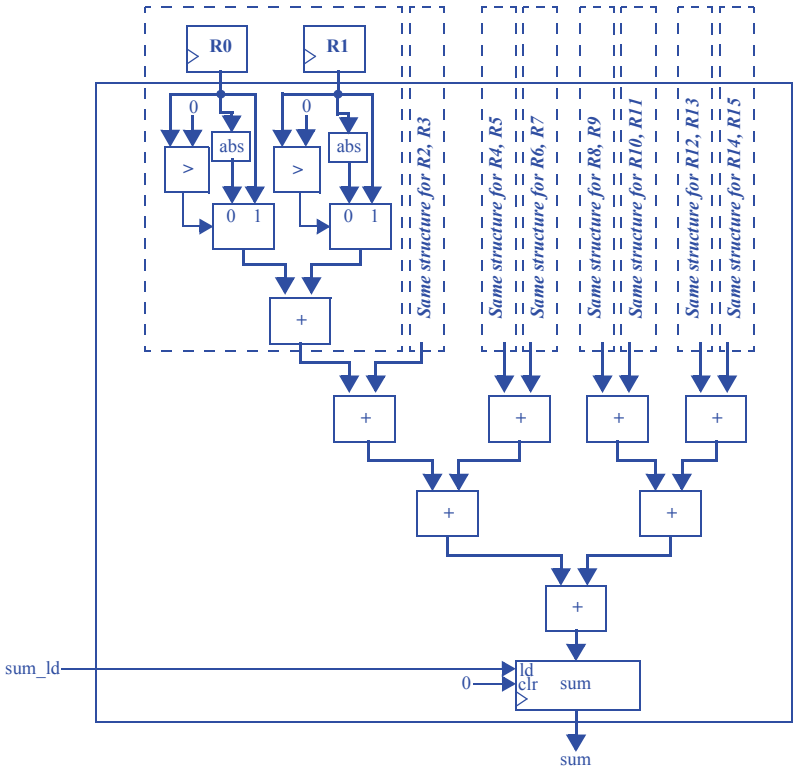
5.21 Design a data-dominated system that computes and outputs the sum of the absolute values of 16 separate 32-bit registers (not in a register file) storing signed numbers (do not consider how those numbers get stored). The computation of the sum should be done using a single equation in one state. 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`.

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:

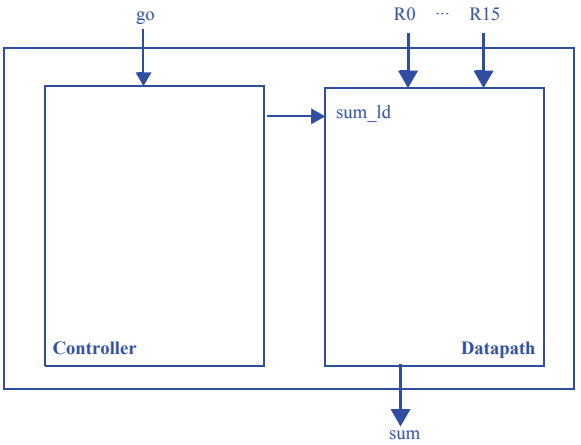


Step 2A - Create a datapath

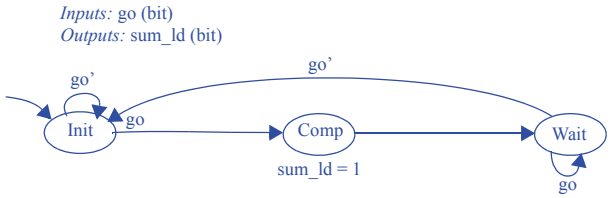


Note: the abs component may be found in Exercise 4.38

Step 2B - Connect the datapath to a controller



Step 2C - Derive the controller's FSM



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.

The critical path of the full adder lies along the path from any of the inputs to the co output. The critical path features two gates with a total delay of 4ns and three segments of wire with a total delay of 4ns, for a total critical path delay of 7ns.

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.

The critical path of the decoder lies along one of the decoder's inverted inputs to one of its outputs: 1ns (wire) + 1ns (inverter) + 1ns (wire) + 2ns (AND gate) + 1ns (wire) = 6ns.

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.

The critical path of a 4x1 multiplexer involves an inverter (1ns), an AND gate (2ns), and an OR gate (2ns), resulting in a total critical path delay of 5ns.

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.

(a) Assume the 8-bit carry-ripple adder consists of 8 full-adders chained together. Each full-adder features a critical path delay of 4ns (an AND gate and a XOR gate). Thus, the total critical path delay for the 8-bit carry-ripple adder is 8*4ns = 32ns.

(b) Each full-adder's critical path features one internal wire between an AND and XOR gate and two wires that connect the full-adder's inputs and outputs. For the entire 8-bit carry-ripple adder, the 8 internal wires contribute 8ns to the critical path delay. Wires connecting full-adders together contribute 7ns to the critical path delay.

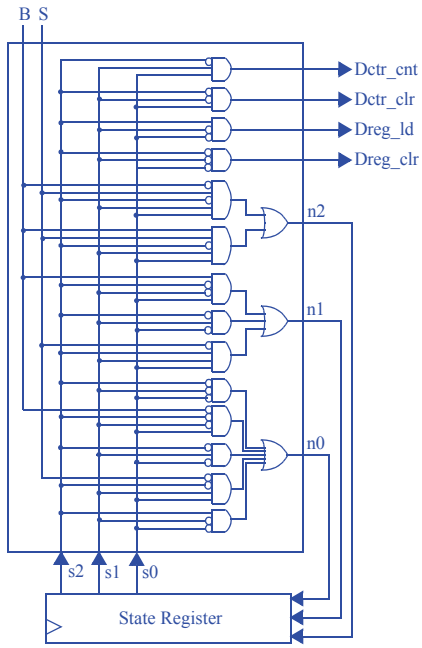
The initial ci and final co contribute 2ns to the critical path delay. Thus, the total critical path delay is 32ns (for gates) + 8ns + 7ns + 2ns = 49ns.

5.26 (a) Convert the laser-based distance measurer's FSM, shown in Figure 5.21, 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.

(a)

Inputs					Outputs							
s2	s1	s0	B	S	n2	n1	n0	L	Dreg_clr	Dreg_ld	Dctr_clr	Dctr_cnt
0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	1	0	0	1	0	1	0	0	0
0	0	0	1	0	0	0	1	0	1	0	0	0
0	0	0	1	1	0	0	1	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0	1	0
0	0	1	0	1	0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0	0	0	0	1	0
0	0	1	1	1	0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	1	1	0	0	0	0
0	1	0	0	1	0	1	1	1	0	0	0	0
0	1	0	1	0	0	1	1	1	0	0	0	0
0	1	0	1	1	0	1	1	1	0	0	0	0
0	1	1	0	0	0	1	1	0	0	0	0	1
0	1	1	0	1	1	0	0	0	0	0	0	1
0	1	1	1	0	0	1	1	0	0	0	0	1
0	1	1	1	1	1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	0	0	1	0	0
1	0	0	0	1	0	0	1	0	0	1	0	0
1	0	0	1	0	0	0	1	0	0	1	0	0
1	0	0	1	1	0	0	1	0	0	1	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0

n2 = s1's1s0B'S + s2's1s0BS
n1 = s2's1's0B + s2's1s0' + s2's1s0S'
n0 = s2's1's0' + s2's1's0B' + s2's1s0' + s2's1s0S' + s2s1's0'
Dreg_clr = s2's1's0'
Dreg_ld = s2s1's0'
Dctr_clr = s2's1's0
Dctr_ctr = s2's1s0



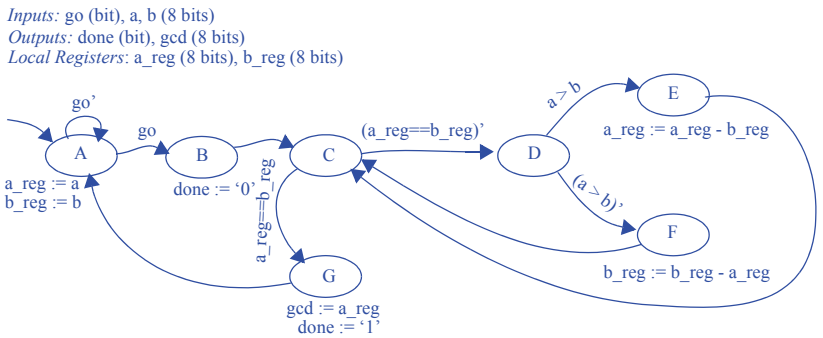
(b) The controller features two levels of gates, resulting in a delay of 4ns. Therefore the critical path is within the up-counter, or 5ns.

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

Section 5.5: 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;
}
```

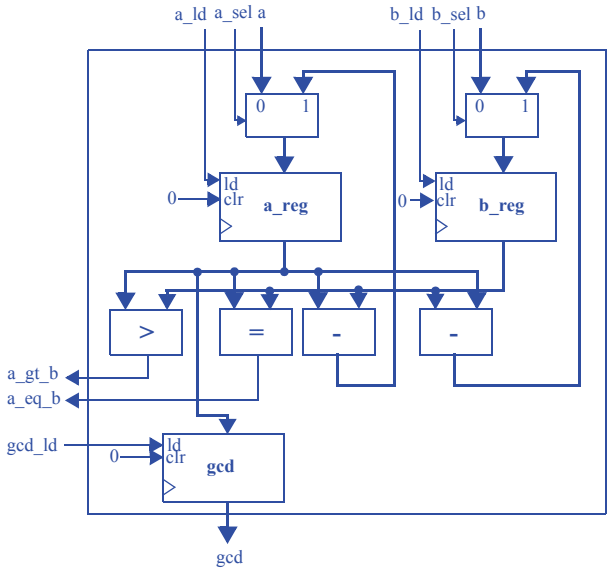


5.28 Use the RTL design process to convert the high-level state machine you created in Exercise 5.27 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

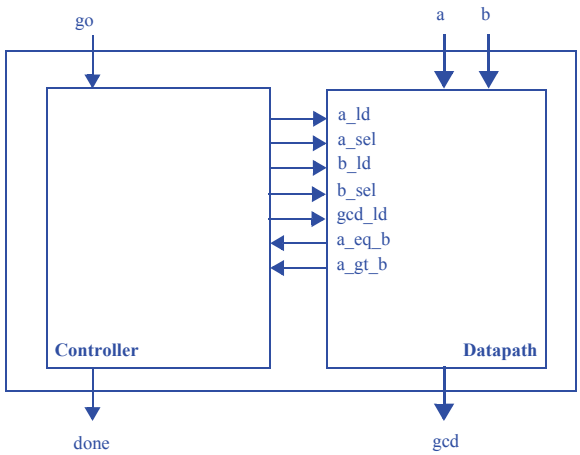
Step 1 - Capture a high-level state machine

The high-level state machine was developed in Exercise 5.27.

Step 2 - Create a datapath

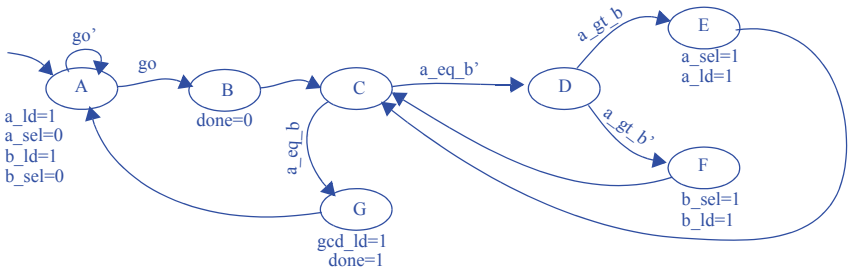


Step 3 - Connect the datapath to a controller



Step 4 - Derive the controller's FSM

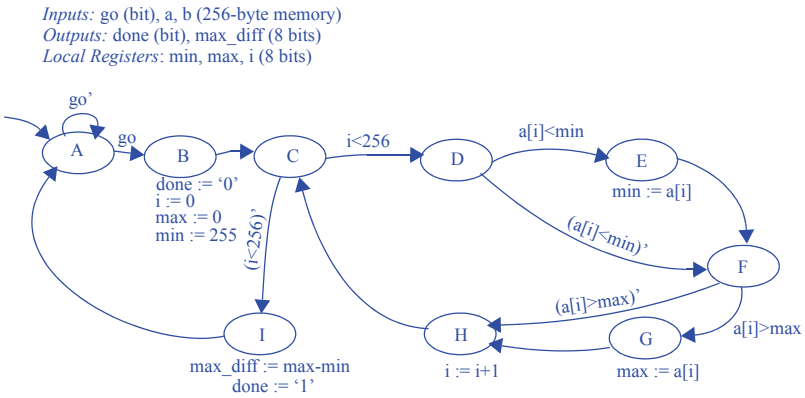
Inputs: go, done, a_gt_b, a_eq_b (bit)
Outputs: done, a_ld, a_sel, b_ld, b_sel, gcd_ld (bit)



5.29 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[1] > max ) {
      max = a[i];
    }
    i = i + 1;
  }
  max_diff = max - min;
  done = 1;

}
```

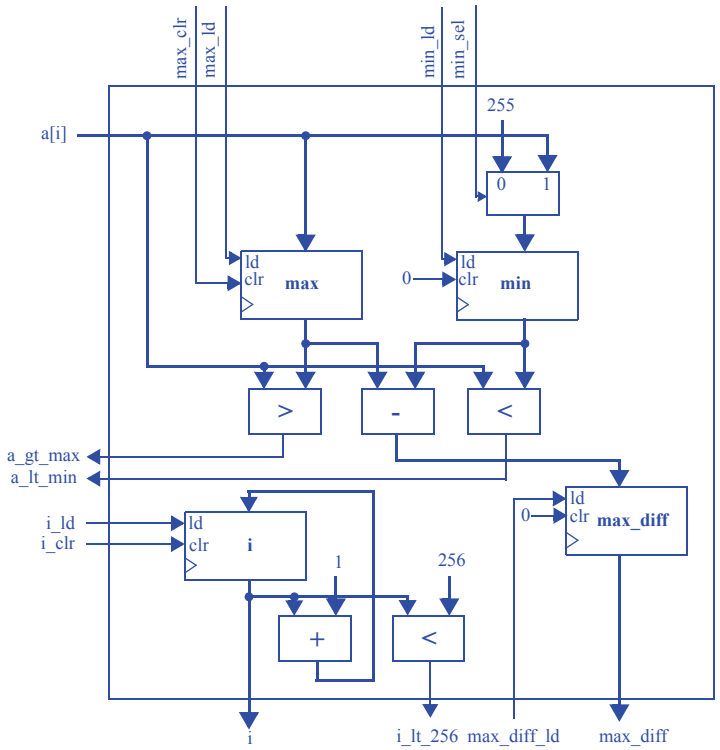


5.30 Use the RTL design process to convert the high-level state machine you created in Exercise 5.29 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

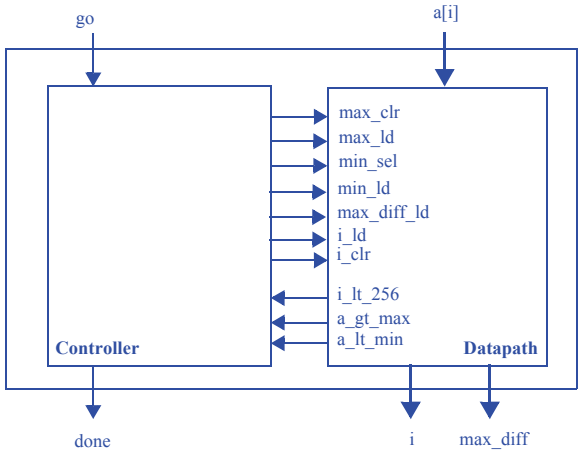
Step 1 - Capture a high-level state machine

The high-level state machine was developed in Exercise 5.29.

Step 2 - Create a datapath

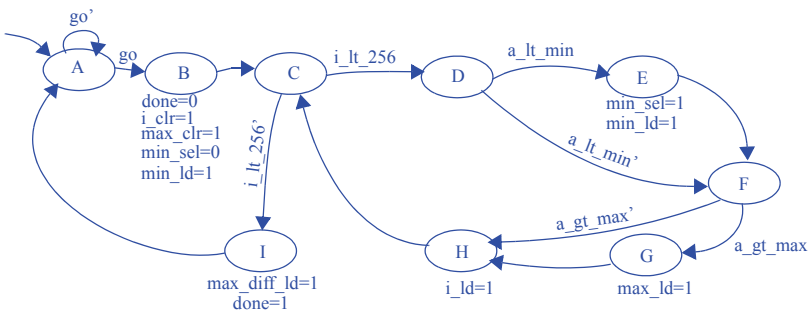


Step 3 - Connect the datapath to a controller



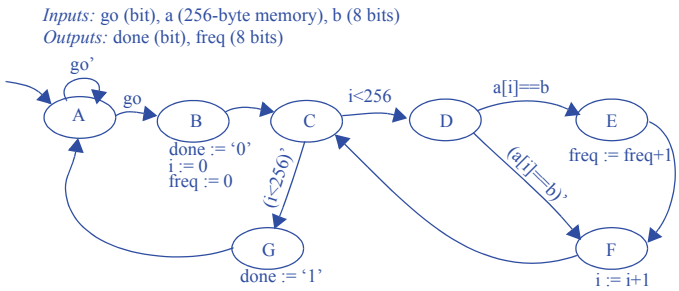
Step 4 - Derive the controller's FSM

Inputs: go, i_lt_256, a_gt_max, a_lt_min (bit)
Outputs: done, max_clr, max_ld, min_sel, min_ld, max_diff_ld, i_ld, i_clr (bit)



5.31 Convert the following C 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;
    }
    i = i + 1;
  }
  done = 1;
}
```

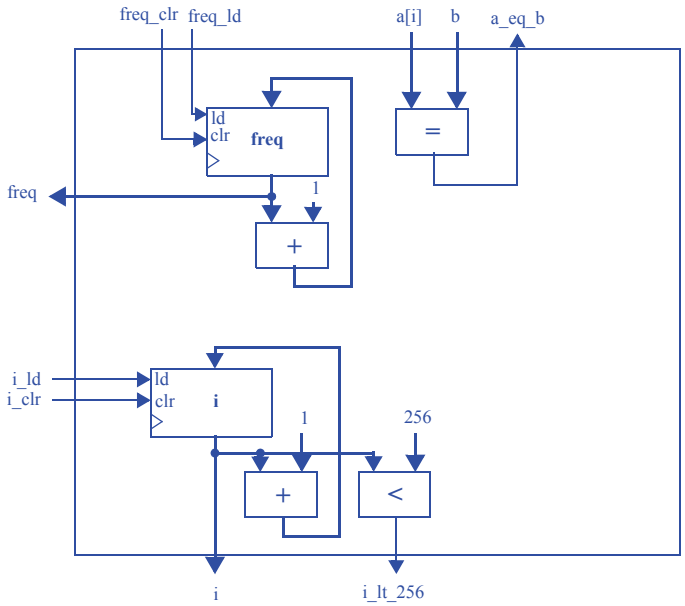


5.32 Use the RTL design process to convert the high-level state machine you created in Exercise 5.31 to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only.

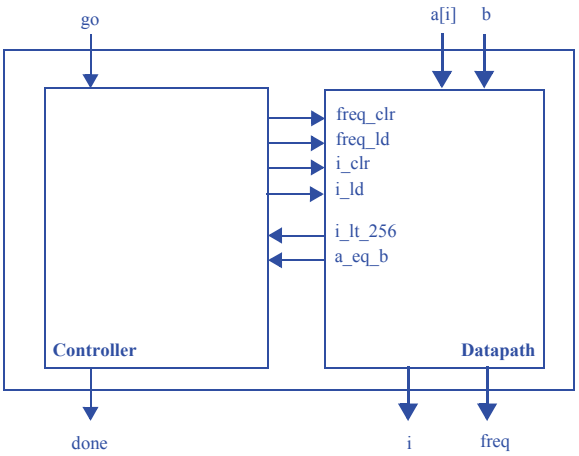
Step 1 - Capture a high-level state machine

The high-level state machine was developed in Exercise 5.31.

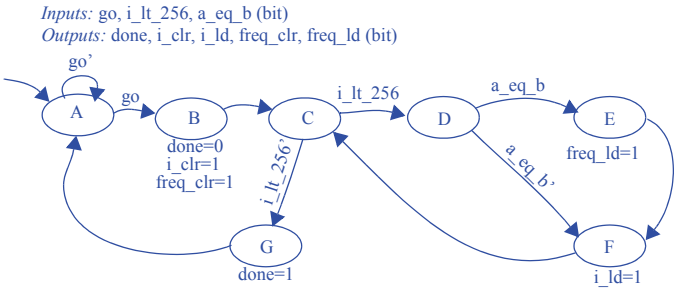
Step 2 - Create a datapath



Step 3 - Connect the datapath to a controller

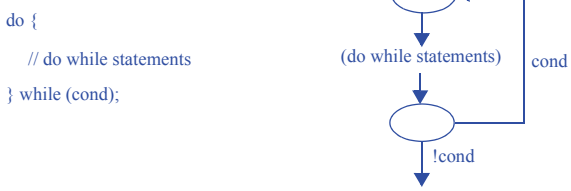


Step 4 - Derive the controller's FSM



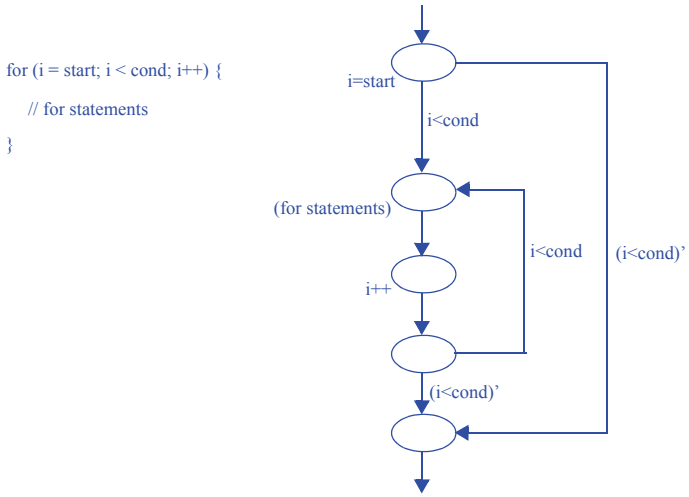
5.33 Develop a template for converting a `do{ }while` loop of the following form to a high-level state machine.

```
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;
}
```

Based on our answer for Exercise 5.34, we naively assume that each "for" construct requires 4 states, not including any statements. We'll also assume that "s=0" requires one state, "s = s + c[i] * x[i + j]" requires one state, and "y[i] = s" requires one state.

The inner loop statement is executed 5 times per outer loop iteration, which means we go through ((2 states + 1 state/inner statement) * 5 iterations) + 2 states = 17 states for the entire inner loop at each outer loop iteration. That means the outer

loop's inner statement is comprised of 19 states. We execute the outer loop 10 times, for a total of ((2 states + 19 states/inner statement) * 10 iterations) + 2 states = 212 states.

We'll assume that one state takes at most the same amount of time as one microprocessor instruction. This gives us 212 * 5ns = 1060 ns for the hardware implementation.

On the microprocessor, if we assume we are allowed base + offset addressing, we must first compute i+j for the inner loop's inner statement, then fetch x[i + j], then fetch c[i], then multiply, and then add. This equates to 5 instructions per inner loop statement. The for loop itself requires two extra instructions, for incrementing j and branching. For 5 iterations, this gives us (5 instr./inner statement * 5 iterations + 1 increment * 5 iterations + 1 branch * 5 iterations) = 35 instructions / inner loop.

Thus, each outer loop iteration requires 35 + 2 = 37 instructions. We then have a total of (37 instr./inner statement * 10 iterations + 1 increment * 10 iterations + 1 branch * 10 iterations) = 390 instructions. This gives us 390 instructions * 5ns/instruction = 1950 ns for the software implementation.

We can see that even with very rough estimates, hardware is clearly much faster than software.

Section 5.6: 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.

10 million transistors / 1 transistor/DRAM bit storage cell = 10 million DRAM bit storage cells.

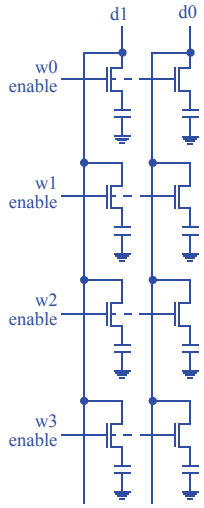
5.37 Calculate the approximate number of SRAM bit storage cells that will fit on an IC with a capacity of 10 million transistors.

10 million transistors / 6 transistors/SRAM bit storage cell = 1,666,666 SRAM bit storage cells, or about 1.67 million SRAM bit storage cells.

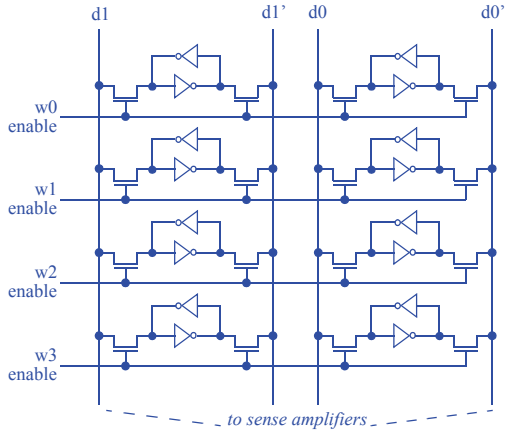
5.38 Summarize the main differences between DRAM and SRAM memories.

DRAM memories use a single transistor and capacitor per bit, while SRAM memories require six transistors per bit. SRAM is thus less compact and more expensive than a DRAM that can store the same number of bits. However, SRAMs typically feature faster access times than DRAMs as DRAMs require a periodic refresh of its contents, a process which blocks DRAM accesses.

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



5.41 Summarize the main differences between EPROM and EEPROM memories.

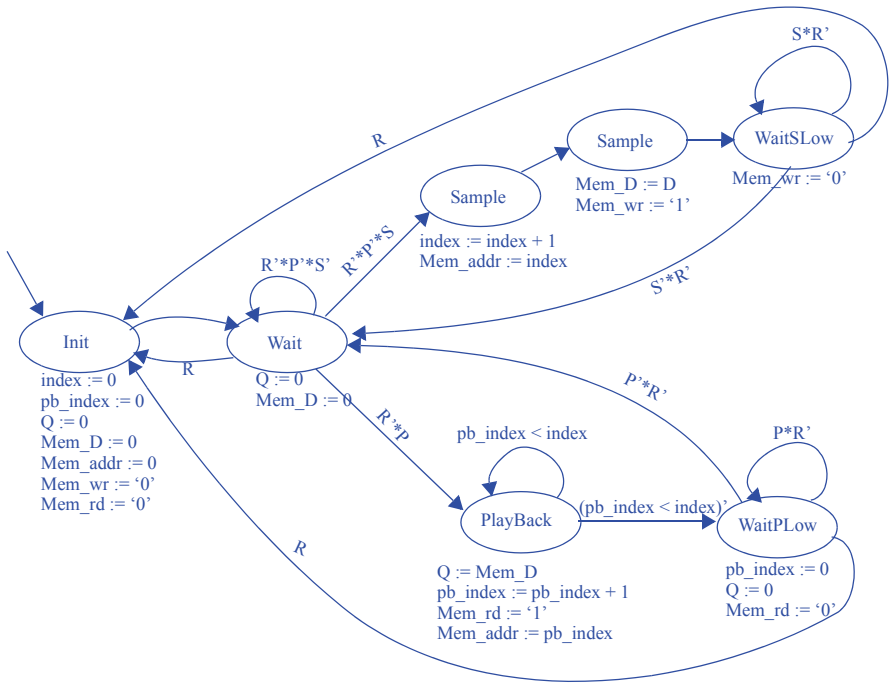
An EPROM is erased en masse by shining ultraviolet light on the memory (typically through a window in the memory's packaging). An EEPROM is erased through a high-voltage signal, and specific words can be erased.

5.42 Summarize the main differences between EEPROM and flash memories.

Whereas an EEPROM may permit erasing one word at a time, a flash memory is a type of EEPROM which permits erasing larger blocks of memory at a time (or perhaps the entire memory).

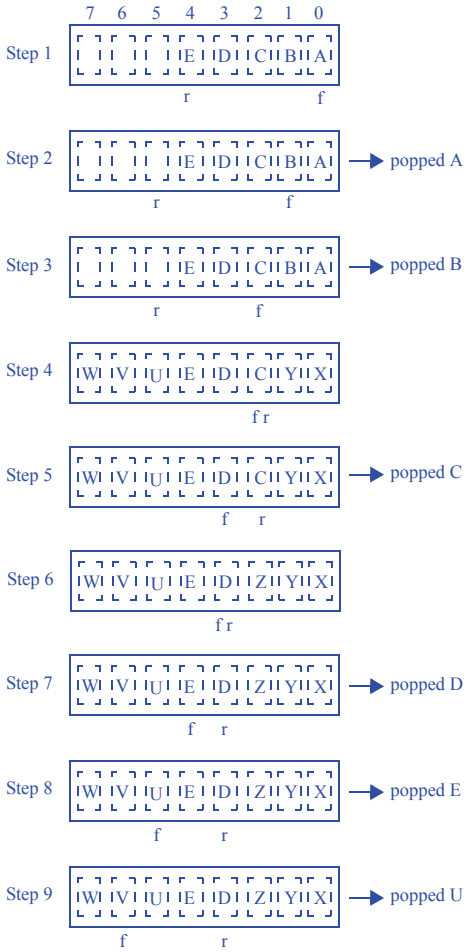
5.43 Use an HLSM to capture the design of a system that can save data samples and then play them back. The system has an 8-bit input D where data appears. A single-bit input S changing from 0 to 1 requests that the current value on D (i.e., a sample) be saved in a nonvolatile memory. Sample requests 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.

Inputs: S, P, R (bit); D, Mem_D (8 bits)
Outputs: Q (8 bits); Mem_D (8 bits) [both an input and an output]; Mem_addr (14 bits); Mem_wr, Mem_rd (bit)
Local Registers: index (14 bits), pb_index (14 bits)

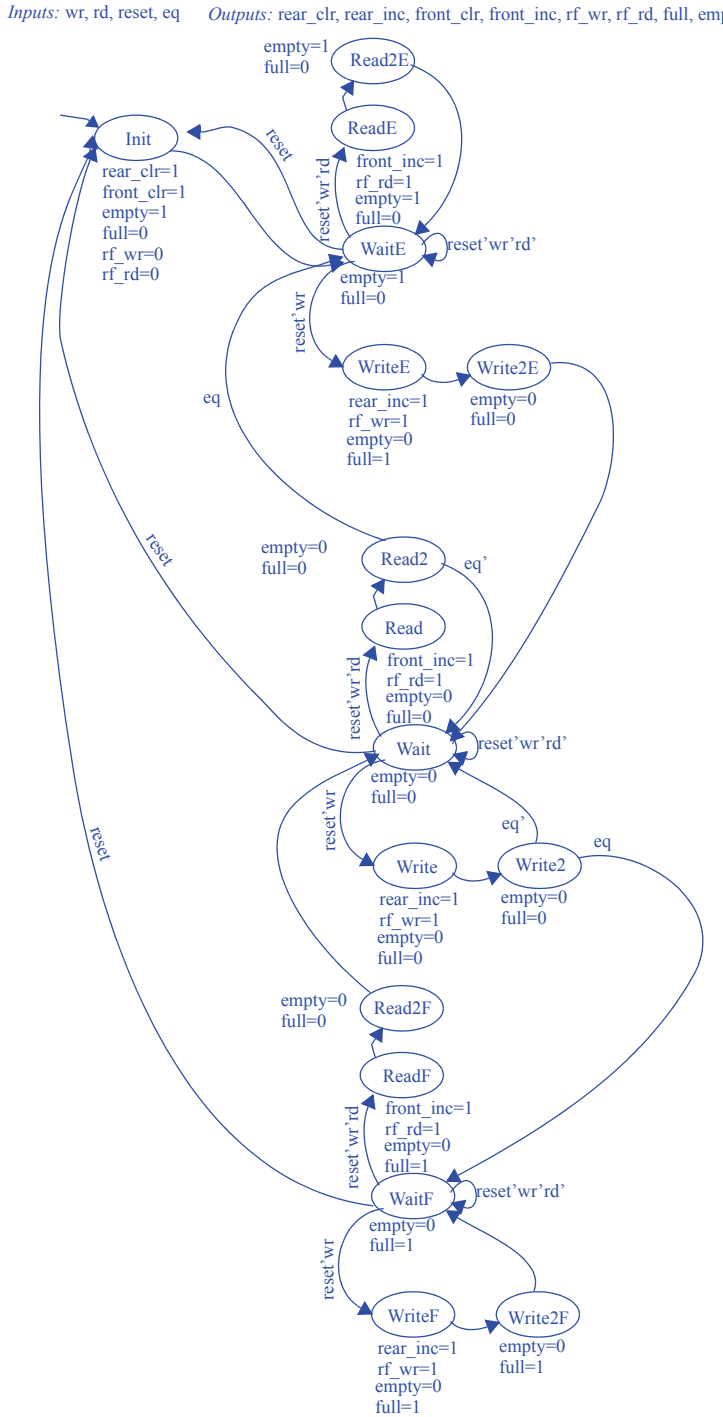


Section 5.7: Queues (FIFOs)

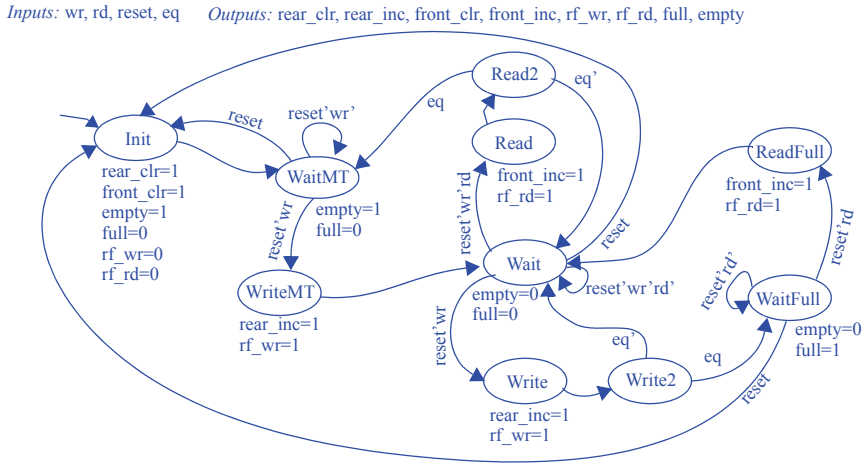
5.44 For an 8-word queue, show the queue's internal state and provide the value of popped data for the following sequences of pushes and pops: (1) push A, B, C, D, E, (2) pop, (3) pop, (4) push U, V, W, X, Y, (5) pop, (6) push Z, (7) pop, (8) pop, (9) pop.



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



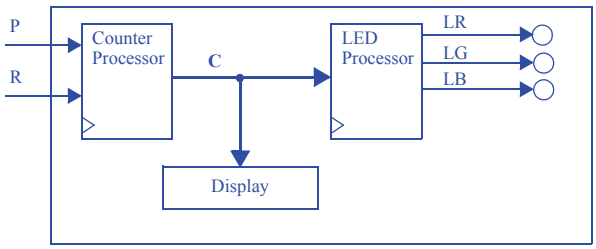
5.46 Create an FSM describing the queue controller of Figure 5.79, but with error-preventing behavior that ignores any pushes when the queue is full, and ignores pops of an empty queue (outputting 0).



Section 5.9: Multiple Processors

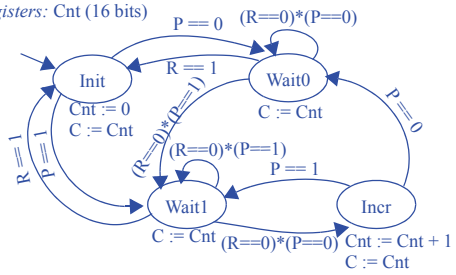
5.47 A system S counts people that enter a store, incrementing the count value when a single-bit input `P` changes from `1` to `0`. The value is reset when `R` is `1`. The value is output on a 16-bit output C, which connects to a display. Furthermore, the system has a lighting system to indicate the 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.

System Diagram:

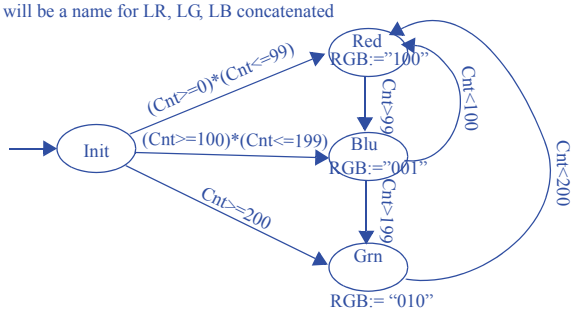


]

Counter HLSM:
Inputs: P, R (bit)
Outputs: C (16 bits)
Local Registers: Cnt (16 bits)

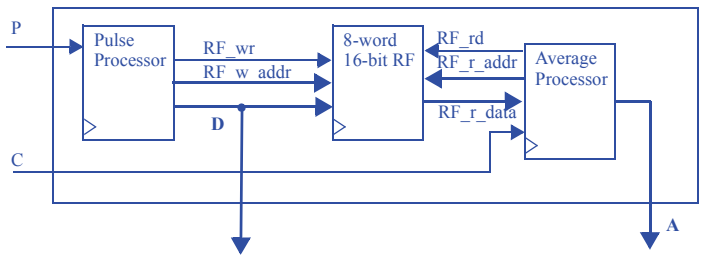


LED HLSM:
Inputs: Cnt (16 bits)
Outputs: LR, LG, LB (bit)
Note: RGB will be a name for LR, LG, LB concatenated

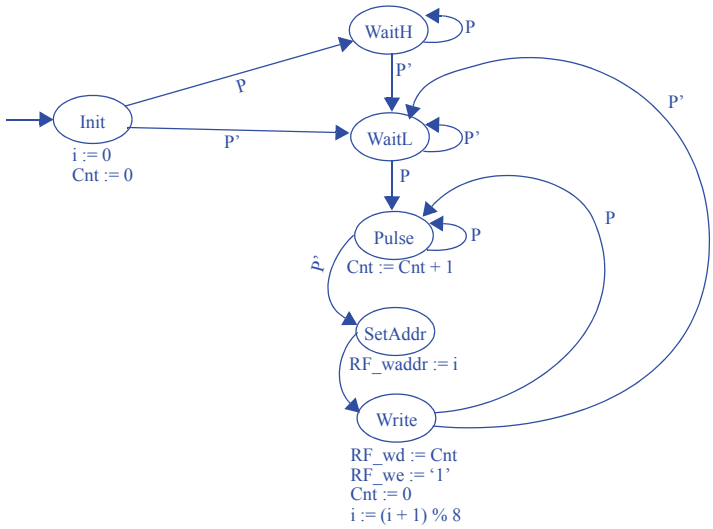


5.48 A system S counts the cycles high of the most recent pulse on a single-bit input `P` and displays the value on a 16-bit output D, holding the value there until the next pulse completes. The system also keeps track of the previous 8 values, and computes and outputs the average of those values on 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.

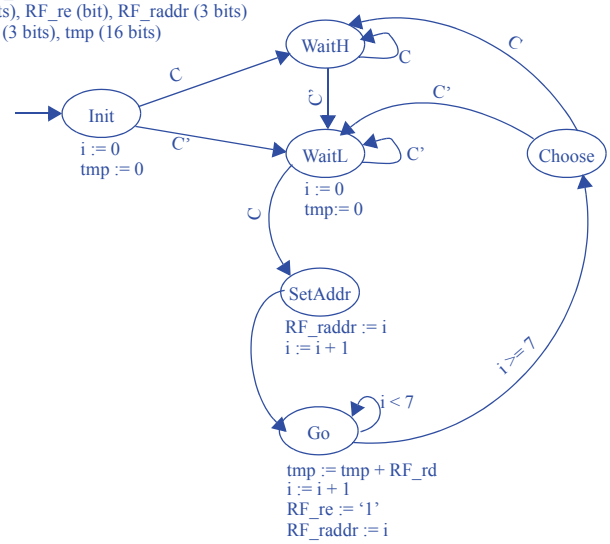
System Diagram:



Pulse HLSM:
Inputs: P (bit)
Outputs: RF_waddr (3 bits), RF_we (bit), RF_wd (16 bits)
Local Registers: i (3 bits), Cnt (16 bits)

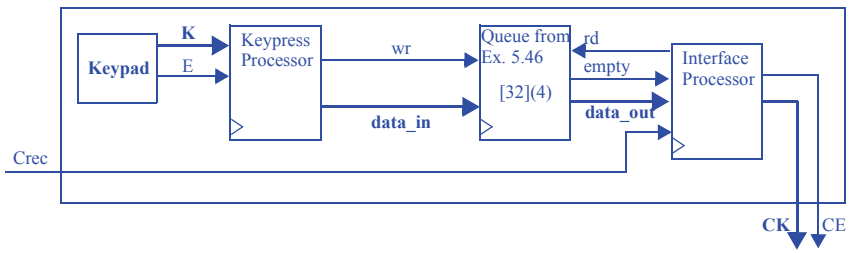


Average HLSM:
Inputs: C (bit), RF_rd (16 bits)
Outputs: A (16 bits), RF_re (bit), RF_raddr (3 bits)
Local Registers: i (3 bits), tmp (16 bits)

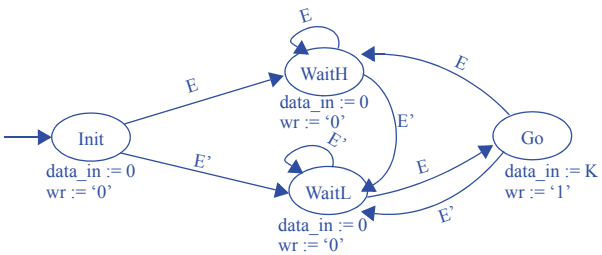


5.49 A system S counts people that enter a store, incrementing the count value when a single-bit input `P` changes from `1` to `0`. The value is reset when `R` is `1`. The value is output on a 16-bit output C, which connects to a display. Furthermore, the system has a lighting system to indicate the 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.

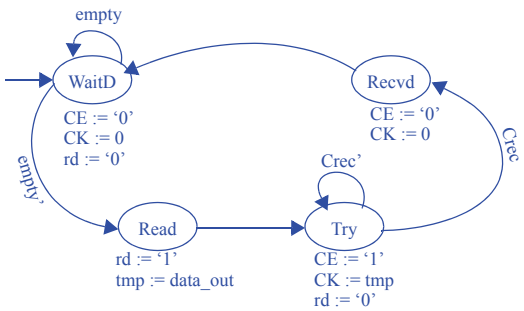
System Diagram:



Keypress HLSM:
Inputs: K (4 bits), E (bit)
Outputs: data_in (4 bits), wr (bit)

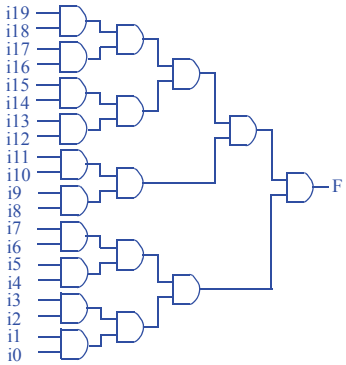


Interface HLSM:
Inputs: empty (bit), Crec (bit), data_out (4 bits)
Outputs: rd (bit), CE (bit), CK (4 bits)
Local Registers: tmp (4 bits)

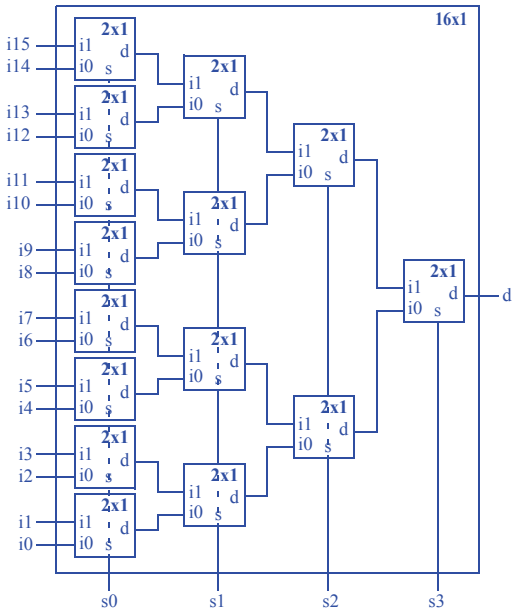


Section 5.10: Hierarchy—A Key Design Concept

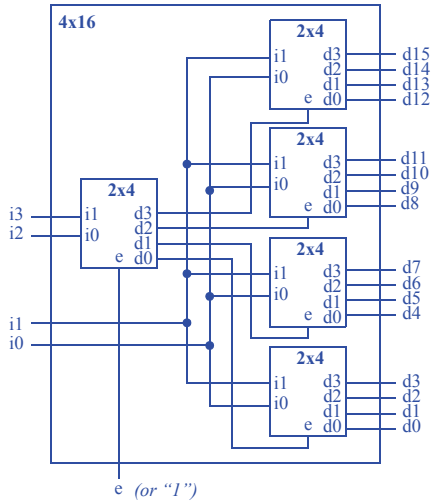
5.50 Compose a 20-input AND gate from 2-input AND gates.



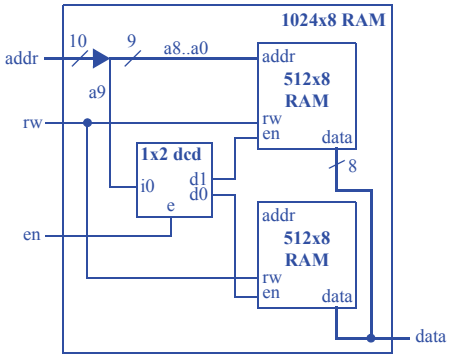
5.51 Compose a 16x1 mux from 2x1 muxes.



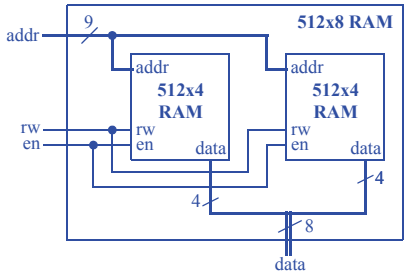
5.52 Compose a 4x16 decoder with enable from 2x4 decoders with enable.



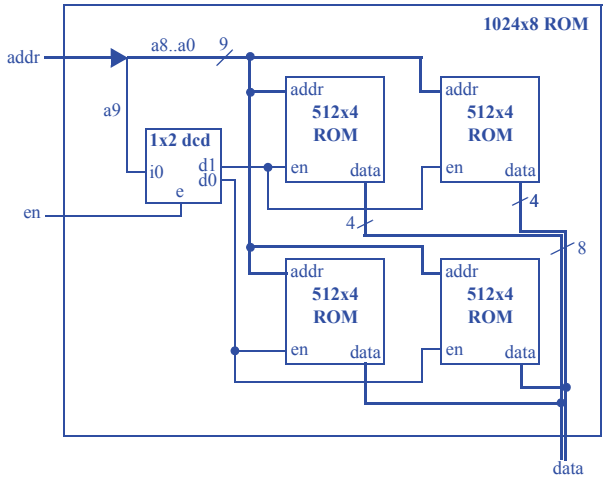
5.53 Compose a 1024x8 RAM using only 512x8 RAMs.



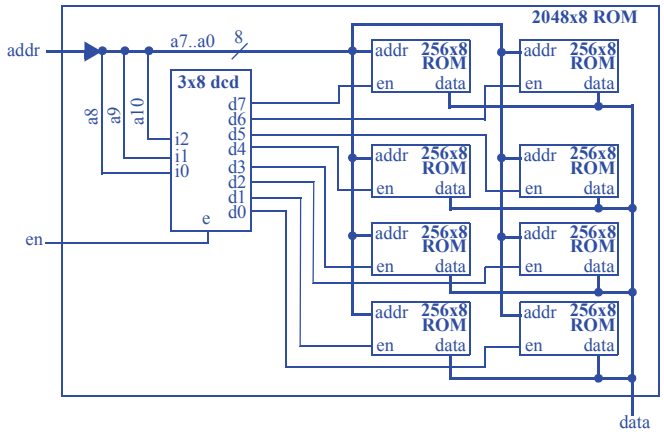
5.54 Compose a 512x8 RAM using only 512x4 RAMs.



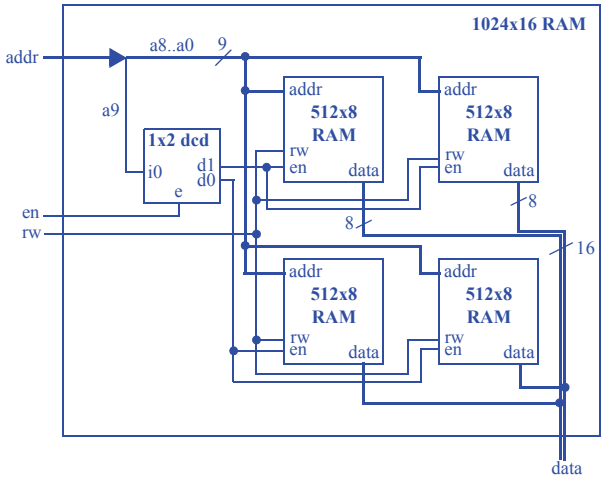
5.55 Compose a 1024x8 ROM using only 512x4 ROMs.



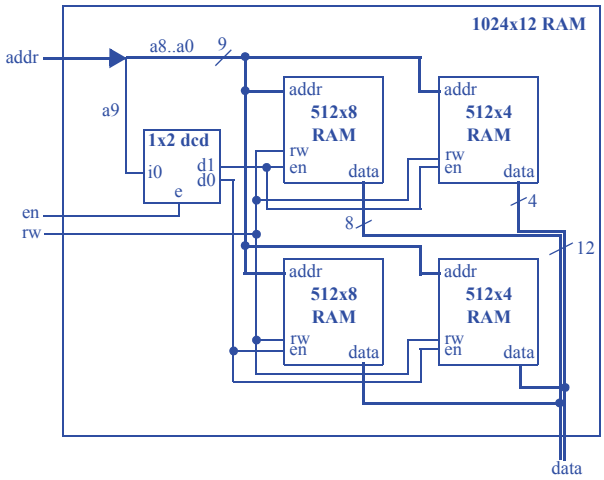
5.56 Compose a 2048x8 ROM using only 256x8 ROMs.



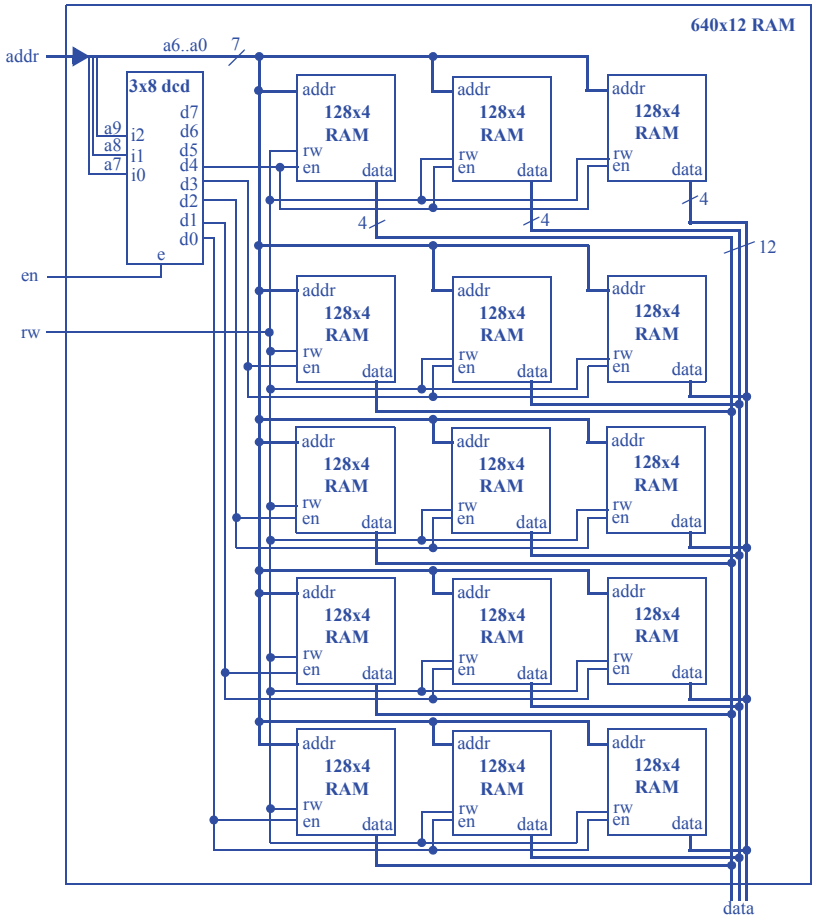
5.57 Compose a 1024x16 RAM using only 512x8 RAMs.



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 need indicate how many 2-input AND gates exist in each level, from which we could easily determine the connections.

Solution not shown for challenge problems. The general solution involves a while loop that continues until an iteration involves just 1 AND gate. Each iteration should place X/2 gates, where X is initially N and where X is set to X/2 in each iteration. Care must be taken when a level has an odd number of inputs.