Chapter 19

Sequential Examples

This chapter gives some additional examples of sequential circuits. We start with a simple FSM that reduces the number of 1s on its input by a factor of 3 to review how to draw a state diagram from a specification and how to implement a simple FSM in Verilog. We then implement an SOS detector to review factoring of state machines. Next we revisit our Tic-Tac-Toe game from Section 9.4 and build a datapath sequential circuit that plays a game against itself useing the combinational move generator we previously developed. We illustrate the use of table-driven sequential circuits and composing circuits from sequential building blocks like counters and shift registers by building a Huffman encoder and decoder. The encoder uses table lookup along with a counter and shift register while the decoder traverses a tree data structure stored in a table.

19.1 A Divide-by-Three Counter

In this section we will design a finite state machine that outputs a high signal on the output for one cycle for each three cycles the input has been high. More specifically, our FSM has a single input `in` and a single output `out`. When `in` is detected high for the third cycle (and sixth, ninth, etc...) `out` will go high for exactly one cycle. This FSM divides the *number* of pulses on the input by three. It does not divide the binary number represented by the input by three.

A state diagram for this machine is shown in Figure 19.1. At first it may seem that we can implement this machine with three states. Four, however are required. We need states A to D to distinguish having seen the input high for 0, 1, 2, or 3 cycles so far. The machine resets to state A. It sits in this state until the input is high on a rising clock edge at which time it advances to state B. The second high input takes the machine to C, and the third high input takes the machine to D where the output goes high for one cycle. We can't simply have this third hign input take us back to A because we need to distinguish having seen three cycles of high input — in which case the output goes high —

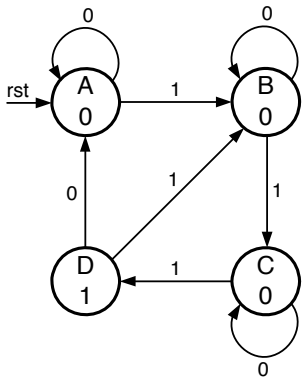

Figure 19.1: State diagram for a divie-by-three counter FSM. The four states represent having seen the input high for 0, 1, 2, and 3 cycles so far.

from having seen zero cycles of high input.[1]

The FSM always exits state D after one cycle. The input during this cycle determines the next state. If the input is low, the machine advances to state A to wait for three more high inputs before the next output. If the input is high, this counts as one of the three high inputs, so the machine advances to state B to wait for two more.

A verilog description of this divide-by-three FSM is shown in Figure 19.2. A *case* statement is used to implement the next-state function, including reset. A single assign statement implements the output function, making the output high in state D. The define statements used to define the states and their width are not shown. Waveforms from simulating this verilog model are shown in Figure 19.3.

19.2 An SOS Detector

Morse code, once widely used for telegraph and radio communication, encodes the alphabet, numbers, and a few punctuation marks into an on/off signal as patterns of dots and dashes. Spaces are used to separate symbols. A *dot* is a short period of *on*, a *dash* is a long period of *on*. Dots and dashes within a symbol are separated by short periods of *off*, and a space is a long period of *off*. The universal distress code, SOS, in Morse code is three dots (S), a space, three dashes (O), a space, and three dots again (the second S).

Consider the task of building a finite-state machine to detect when an SOS is received on the input. We assume that a *dot* is represented by the input being high for exactly one cycle, a *dash* is represented by the input being high

[1] See Exercise 19–3 for an approach that does require only three states.

```
//------------------------------------------------------------------------
//Divide by 3 FSM
//  in - increments state when high
//  out - goes high one cycle for every three cycles in is high
//    it goes high for the first time on the cycle after the third cycle
//    in is high.
//------------------------------------------------------------------------
module Div3FSM(clk, rst, in, out) ;
  input clk, rst, in ;
  output out ;

  wire out ;
  wire [`AWIDTH-1:0] state ; // current state
  reg  [`AWIDTH-1:0] next ;  // next state

  // instantiate state register
  DFF #(`AWIDTH) state_reg(clk, next, state) ;

  // next state function
  always @(state or rst or in) begin
    case(state)
      `A: next = rst ? `A : (in ? `B : `A) ;
      `B: next = rst ? `A : (in ? `C : `B) ;
      `C: next = rst ? `A : (in ? `D : `C) ;
      `D: next = rst ? `A : (in ? `B : `A) ;
      default: next = `A ;
    endcase
  end

  // output function
  assign out = (state == `D) ;
endmodule
```

Figure 19.2: Verilog description of the divide-by-three counter.

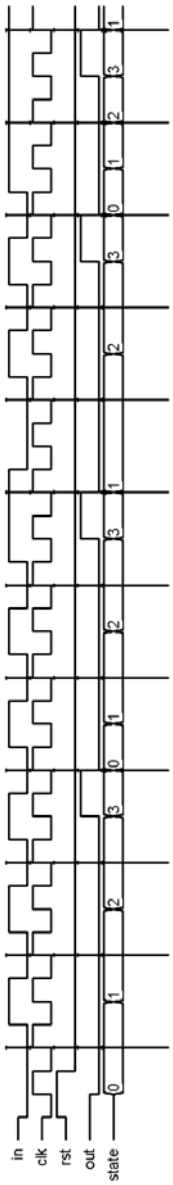

Figure 19.3: Waveforms from simulating the divide-by-three counter.

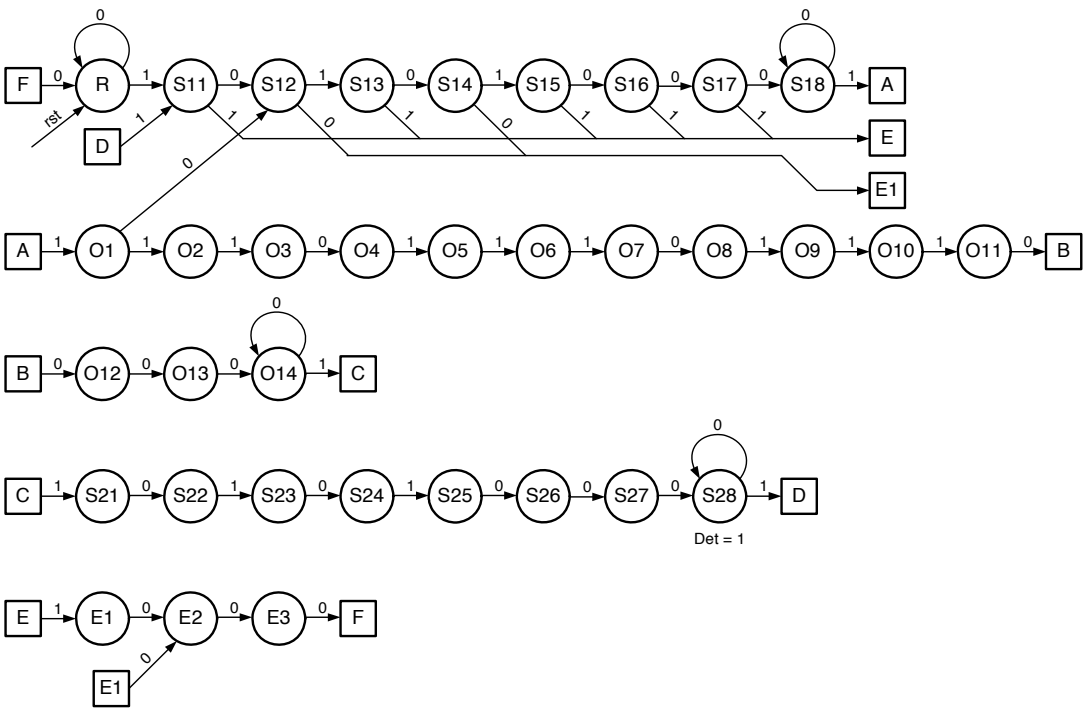

Figure 19.4: A state diagram for an SOS detector realized as a single, flat FSM. The square boxes indicate connections.

for exaclty three cycles, that dots and dashes within a symbol are separated by the input being low for exactly one cycle, and that a *space* is represented by the input being low for three or more cycles. Note that the input going either high or low for exactly two cycles is an illegal condition. With this set of definitions, one legal SOS string is 10101000111011101110001010100 0.

We can build an SOS detector as a single, flat state machine as shown with the state diagram of Figure 19.4. The FSM resets to state R. States S11 to S18 detect the first "S" and the associated space. States O1 to O11 detect the "O" and states O12 to O14 detect the space following the "O". Finally states S21 to S28 detect the second "S" and the subsequent space. State 28 outputs a "1" to indicate that SOS has been detected.

For clarity, many transitions are omitted from Figure 19.4. The transitions along the horizontal path from state R through state S28 represent the transitions that occur when an SOS is detected. If at any point along this path a 1 is detected when a 0 is expected the machine transitions to state E1. Similarly, if a 0 is detected when we expect a 1, the machine transitions to state E2. These transitions are shown for the first row (via boxes E and E1) and then omitted

to avoid cluttering the figure. States E1 to E3 are error handling states that wait for a space after an error condition and then restart the detection.

The transition from O1 to S12 handles the case where the input includes the string SSOS. After detecting the first S, we are expecting an O but instead receive a second S. If we were to transition to state E2 on receiving a 0 in O1, we would miss this second S and hence the SOS. Instead we must recognize the *dot* and go to state S12.

The transition from S28 to S11 (via the box labeled D) is needed to allow back-to-back SOSs with minimum sized spaces to be detected. After detecting SOS, including the subsequent space, in state S28, the next 1 may be the first dot of the next SOS and must be recognized by going to state S11.

While the flat FSM of Figure 19.4 works, it is not a very good solution for a number of reasons. First, it is not modular. If we were to change the definition of a dot to be the input going high for 1 or 2 cycles, the flat machine woudl need to be changed in eight places (every place a dot is recognized). Similar global change would be needed to accomodate a change to the definition of a dash or a space. Also, the machine would need to be complete reworked if the sequence we are detecting is different than SOS, say ABC. Second, the machine is large, 34 states, and would become even larger with more flexible definitions of dots and dashes. Finally, some aspects of the machine, like the transition from O1 to S12 are subtle.

The SOS machine is a perfect candidate for *factoring*. We can build FSMs to detect dots, dashes, and spaces, and then use the outputs of these FSMs to build FSMs that detect S and O. Finally, a simple top-level FSM detects SOS. A block diagram for a factored version of our SOS detection FSM is shown in Figure 19.5. The input bit stream (*sequence*) is input to three element detecting FSMs - *Dot*, *Dash*, and *Space*. Each of these FSMs has two outputs, one that indicates when the element has been detected, and one that indicates that the current input sequence *could be* part of that element. For example, the *Dot* FSM outputs *isDot* when a dot is detected, and *cbDot* when the current sequence could be a dot, but we need additional input before deciding.

The six signals out of the three element detectors feed a pair of character detectors, one each for S and O. Like the element detectors, each character detector also has an *is* and a *could be* output. The four signals out of the two character detectors are input to a top-level SOS FSM that indicates when SOS is detected.

Figure 19.6 shows the three FSMs that detect the elements Dot, Dash, and Space. The Dot FSM resets to state 0. Upon detecting a 1 on the input it indicates that the current sequence *could be* a dot by asserting output *cb* and transitions to state *Dot*. In state *Dot*, a 0 on the input results in a dot being detected with the *is* output asserted and returns the machine to state 0. Note that when the *is* output is asserted, the *cb* output is also asserted. If a 1 is detected in state *Dot*, the machine goes to state 1 to wait for a zero. The Dash and Space machines operate in a similar manner.

The character FSM for the character S is shown in Figure 19.7. The machine resets to state OTH (other) which is also the target of the default (def) tran-

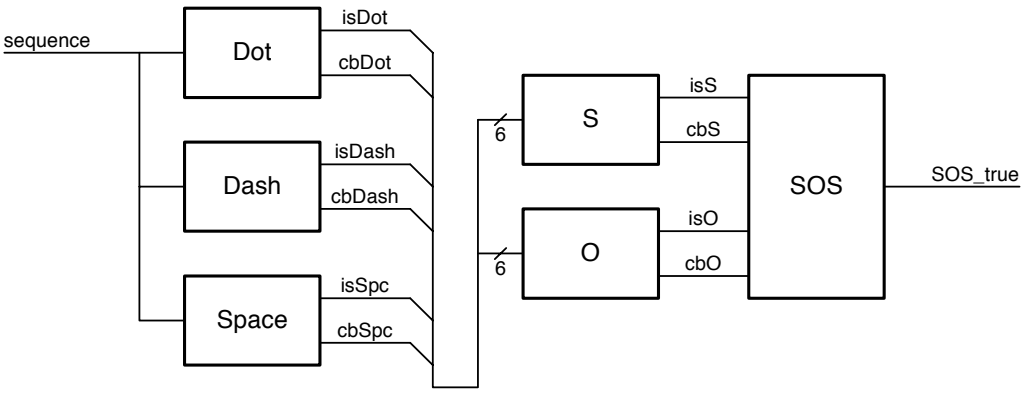

Figure 19.5: A block diagram of a factored SOS detector. The first rank of FSMs detects dots, dashes, and spaces. The second rank detects Ss and Os. The final SOS FSM detects the sequence SOS. Each sub-machine has two outputs: one indicates when the desired symbol has been detected (e.g., *isS*). The other indicates when the current sequence *could be* a prefix of the desired symbol (e.g., *cbS*).

sition which covers unexpected inputs. Upon detecting a space, the machine enters state SPC. Detecting the first dot moves the machine to state D1, and subsequent dots move the machine to states D2, and D3. Detecting a space in state D3 returns the machine to state SPC and asserts *is* to signal that an S has been detected.

If at any point during the sequence from SPC through D1, D2, and D3 and back to SPC the input could not be the element being waiting for (e.g., if *cbDot* is false in state D1), then the machine returns to state OTH. This is why we need the *could-be* outputs on the element detectors. They allow us to detect illegal elements between the elements we are looking for. Consider, for example the input sequence 00010110101000. The machine detects the space 000, and the first dot 10, but then returns to state OTH on the illegal element 110 because *cbDot* falls when the second 1 is detected. If we just wait for *isDot* we would erroneously determine that this sequence is an S because it has three dots. Without monitoring *cbDot* we wouldn't see that the three dots are not contiguous, and hence not an S.

The main SOS detecting FSM, shown in Figure 19.8, contains only three states. It waits in state ST (start) until an S is detected when it moves to state S1. From state S1 if an O is detected it moves to state O. However, if at any point in S1 input *cbO* goes false, indicating an illegal sequence between the S and the O, the machine returns to ST. If the machine detects a second S while in state O, it asserts its *is* output, detecting an SOS, and returns to ST. If input *cbS* goes false while instate O, detecting an illegal sequence between the O and

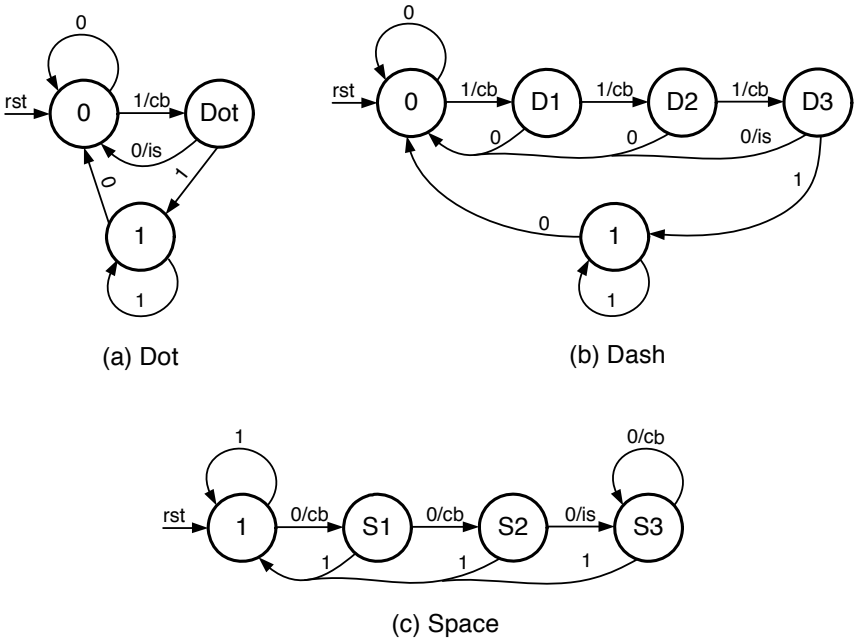

Figure 19.6: Element finite state machines. (a) Dot, (b) Dash, and (c) Space. Each outputs when the current sequence *could be* (cb) the respective element and when the element has been detected (is).

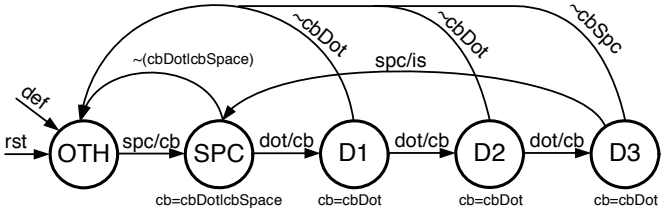

Figure 19.7: State diagram for S-detecting FSM.

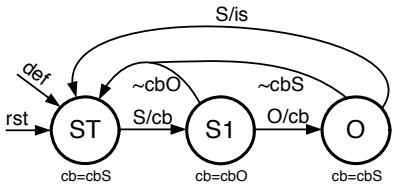

Figure 19.8: State diagram main SOS detecting FSM.

the S, the machine returns to ST without detecting SOS.

Waveforms showing operation of the factored SOS detector are shown in Figure 19.9. The waveforms show two good SOS detections with a SOT (T is a single dash) between them. Note the *cb* and *is* waveforms for each element, for the characters S and O, and for SOS itself. On the second 1 of the T's dash, *cbDot* falls causing *cbS* and *cbSOS* to fall in turn (combinationally).

Factoring the SOS detector gives us a much simpler system that is far easier to modify and maintain. Instead of 34 states in one brittle, monolithic state machine, we have a total of 20 states divided over six small, simple FSMs. The largest single FSM in the factored machine has a total of five states. Should we modify our specification to change the definition of a dot to be one or two 1s in a row, we would make one simple change to the Dot FSM.[2]

19.3 A Tic-Tac-Toe Game

In Section 9.4 we designed a combinational module that generated moves for the game of Tic-Tac-Toe. In this section we will use this module as a component in a sequential system that plays a game of Tic-Tac-Toe against itself.

A block diagram of the system is shown in Figure 19.10. The state in the system resides in the three registers on the left side of the figure. The 9-bit `Xreg` and `Oreg` hold bit maps that reflect the current positions of Xs and Os respectively. The 1-bit `xplays` register is true if X plays next and false if Y plays next. Reset is not shown in the figure. `Xreg` and `Oreg` reset to all zeros. `xplays` resets to true.

When it is X's turn to play (`xplays` = 1) the multiplexers direct `Xreg` to the `xin` input of the move generator and `Oreg` to the `oin` input. The move generator generates the next move on `xout` and this is ORed with the current X position to generate the new x position that is stored back in `Xreg` at the end of the cycle. The write to `Xreg` is enabled by `xplays`. When `xplays` is false, the multiplexers switch the move generator inputs to generate a move for O and this move is written back to `Oreg` at the end of the cycle.

A Verilog description of the tic-tac-toe playing system is shown in Figure 19.11. After the declaration of the three state registers an assignment

[2]For some practice modifying this factored SOS detector, see Exercises 19–4 and 19–5.

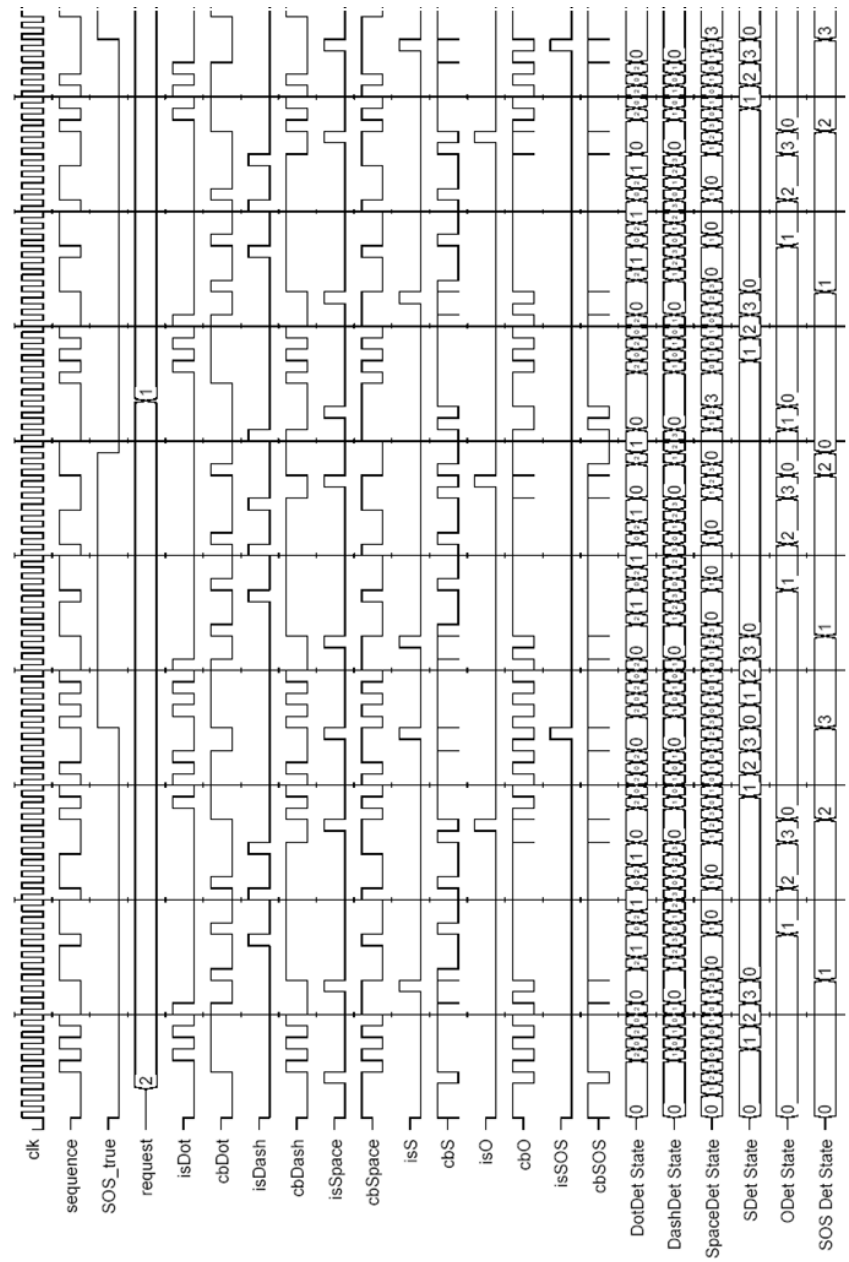

Figure 19.9: Waveforms showing operation of factored SOS detector.

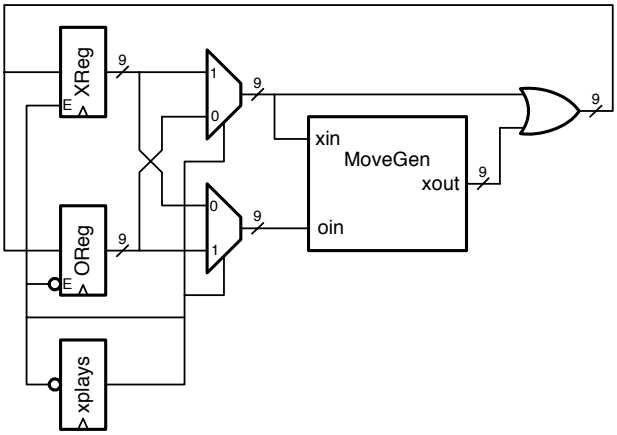

Figure 19.10: Block diagram of a tic-tac-toe playing system using the move generation module of Section 9.4.

statement toggles `xplays` each cycle. The input multiplexers are implemented as select statement in the argument list for the move generator. Two assign statements compute the next state for `Xreg` and `Oreg`. The OR of the move with the previous state is included in these statements.

19.4 A Huffman Encoder/Decoder

A Huffman code is an *entropy* code that encodes each symbol of an alphabet with a bit string. Frequently used symbols are encoded with short bit strings while infrequently used symbols are encoded with longer bit strings. To be able to distinguish short bit strings from the first parts of longer bit strings, each short bit string must not be used as a prefix for any longer bit string. The net result is *data compression*; that is, a typical sequence of symbols is encoded into fewer bits than would be required if all symbols were encoded with the same number of bits.

19.4.1 Huffman Encoder

For this example we will build a Huffman encoder and decoder for the letters of the alphabet, A-Z. The input to our encoder is a 5-bit code where A = 1 and Z = 26.[3] To prevent input characters from arriving faster than our encoder can handle them, our encoder generates an input ready *(irdy)* signal that indicates when the encoder is ready for the next input character. The output is a serial

[3]This corresponds to the low 5-bits of the ASCII code for both upper-case and lower-case letters.

```
//------------------------------------------------------------------------------
// Sequential Tic-Tac-Toe game
//   Plays a game against itself
//------------------------------------------------------------------------------
module SeqTic(clk, rst, xreg, oreg, xplays) ;
  input clk, rst ;
  output [8:0] xreg, oreg ;
  output xplays ;

  wire [8:0] nxreg, noreg, move ; // next state
  wire nxplays ; // next state

  // state
  DFF #(9) x(clk, nxreg, xreg) ;
  DFF #(9) o(clk, noreg, oreg) ;
  DFF xp(clk, nxplays, xplays) ;

  // x plays first, then alternate
  assign nxplays = rst ? 1 : ~xplays ;

  // move generator - mux inputs so current player is x
  TicTacToe movGen(xplays ? xreg : oreg, xplays ? oreg : xreg, move) ;

  // update current player
  assign nxreg = rst ? 0 : (xreg | (xplays ? move : 0)) ;
  assign noreg = rst ? 0 : (oreg | (xplays ? 0 : move)) ;
endmodule
```

Figure 19.11: Verilog description of the Tic-Tac-Toe playing system.

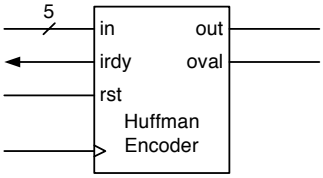

Figure 19.12: Block diagram symbol for our Huffman Encoder. The encoder accepts a 5-bit character on *in* each time *ready* goes high and generates a bit serial output stream on *out* when *valid* is high.

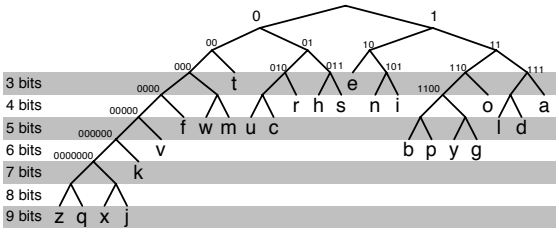

Figure 19.13: Huffman code for the alphabet shown as a binary tree. Starting at the root, each branch to the left denotes a 0 and each branch to the right a 1. Hence the letter W which is reached by a sequence of left, left, left, right, left is encoded as 00010.

bit stream of the encoded characters. To make it easy for the decoder to find the start of the bit stream, our encoder also generates an output valid *(oval)* signal that signals when the bits in the output stream are valid. A block diagram symbol for our encoder showing inputs and outputs is shown in Figure 19.12.

Figure 19.13 shows the code we will use for our example in tree form. The path from the root of the tree to a character gives the code for that character. The letter E, for example, is reached by going right, left, left and hence is represented by the three-bit string 100. The letter J is reached by going left 7 times and then right twice is represented by the nine-bit string 000000011. Very frequently occuring characters like T and E are represented with just three bits. Very infrequently occurring characters like Z, Q, X, and J are represented with nine bits. Representing the code as a tree makes it clear that a short string used to represent one symbol is not a prefix of a longer string used to represent another symbol since each leaf node of the tree terminates the path used to reach that leaf.

A block diagram of a Huffman encoder is shown in Figure 19.14. A five-bit input register holds the current symbol and is loaded with a new symbol each time *irdy* is asserted. The symbol is used to address a ROM that stores the string and string length associated with each symbol. For example, the ROM

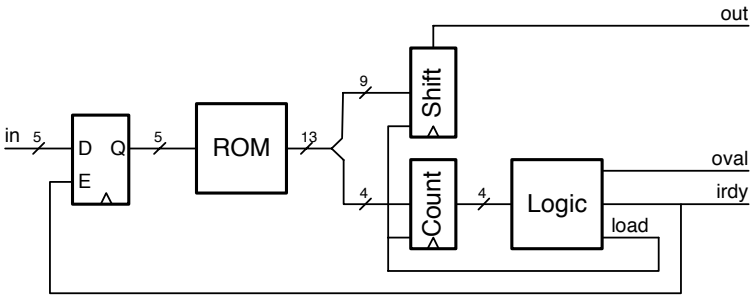

Figure 19.14: Block diagram of Huffman encoder. A ROM stores the length and string for each character. A counter counts down the length while a shift shifts out the string.

stores string 0011 001000000 for the symbol T. This indicates that the string representing T is three bits in length and the three bits are 001. Because the maximum length string is nine bits, we use four bits to represent the length and nine bits to represent the string. Strings shorter than nine bits are left-aligned in the nine-bit field so they can be shifted out to the left.

One cycle after a new symbol is loaded into the input register by *irdy*, signal *load* is asserted to load the length and string associated with that symbol into a counter and a shift register. The counter then counts down while the shift register shifts bits onto the output. When the counter reaches a count of 2 (second to last bit), *irdy* is asserted to load the next symbol into the input register, and when the counter reaches a count of 1 (last bit of this symbol), *load* is asserted to load the length and string for the next symbol into the counter and shift register.

A verilog implementation of our Huffman encoder is shown in Figure 19.15. We use the up/down/load counter from Section 16.1.2 to implement our counter and the left/right/load shift register from Section 16.2.2 to implement our shifter. Note that while we don't use the up function of the counter or the right function of the shift register, this will still result in an efficient implementation as the synthesizer will optimize away the unused logic. The table is implemented in module `HuffmanEncTable` (not shown) which is coded as a large case statement.

The control logic for the Huffman encoder is straightforward. One line of code asserts *irdy* on a count of two or zero - the latter is needed to load the first symbol after a reset. A DFF then delays *irdy* one cycle to generate signal *dirdy* that is used to load the counter and shifter. One line of code and a DFF are used to keep *oval* low after reset until the first time *dirdy* is asserted.

Figure 19.16 shows the result of simulating the Huffman encoder on the input string "THE". The three symbols 14 (T), 08 (H), and 05 (E) in hexadecimal are shown on *in*. The resulting output is 001 (T), 0110 (H), and 100 (E) shifted

```
//------------------------------------------------------------------------
// Huffman Encoder
//   in - character 'a' to 'z' - must be ready
//   irdy - when high accepts the current input character
//   out - bit serial huffman output
//   oval - true when output holds valid bits
//
//   input character accesses a table with each entry having
//   length[4], bits[9]
//------------------------------------------------------------------------
module HuffmanEncoder(clk, rst, in, irdy, out, oval) ;
  input clk, rst ;
  input [4:0] in ;
  output irdy, out, oval ;

  wire [3:0] length, count ;
  wire [8:0] bits, obits ;
  wire [4:0] char ;
  wire dirdy ; // irdy delayed by one cycle - loads count and sr
  wire oval ;

  // control
  wire out  = obits[8] ; // MSB is output
  wire irdy = ~rst & ((count == 2) | (count == 0)) ; // 0 count for reset
  wire noval = ~rst & (dirdy | oval) ;  // output valid cycle after load

  // instantiate blocks
  UDL_Count2 #(4) cntr(.clk(clk), .rst(rst), .up(1'b0), .down(~dirdy),
.load(dirdy), .in(length), .out(count)) ;
  LRL_Shift_Register #(9) shift(.clk(clk), .rst(rst), .left(~dirdy),
        .right(1'b0), .load(dirdy), .sin(1'b0), .in(bits), .out(obits)) ;
  DFF #(5) in_reg(clk, irdy ? in : char, char) ;
  DFF #(1) irdy_reg(clk, irdy, dirdy) ;
  DFF #(1) ov_reg(clk, noval, oval) ;
  HuffmanEncTable tab(char, length, bits) ;

endmodule
```

Figure 19.15: Verilog description of the Huffman encoder.

out bit serially on *out* starting with the first cycle in which *oval* is asserted. The value in the counter can be seen counting down from the string length (3 or 4) to 1 for each symbol while the value in the shift register *obits* is shifting the string associated with each symbol left.

19.4.2 Huffman Decoder

Now that we have encoded a character string using a Huffman code we will look at building the corresponding decoder. To decode Huffman-encoded bit string we simply traverse the encoding tree of Figure 19.13 traversing one edge for each bit of the input bit stream - the left branch for each zero, and the right branch for each one. When we encounter a terminal node during this traversal, we emit the corresponding symbol on the output and restart our traversal at the root of the tree.

To facilitate storing the decoding tree in a table, we relabel the nodes of the tree as shown in Figure 19.17. Each node is assigned an integer that serves as its address in the table. Note that the root does not need to be stored in the table, so we start labeling nodes at 0 with the left child of the root. At each entry in the table we store a type and a value. The type indicates whether this node is an internal node (type=0) or a terminal node (type=1). For a terminal node, the value holds the symbol to emit. For an internal node, the value holds the address of the left child of this node (which will always be an even number). The address of the right child can be found by adding one to the value.

To see how we traverse the table representing the tree to decode a bit string, consider decoding the bit string 001. We start at the root of the tree which has a left child with address 0 and the first 0 of the string directs us to this child. We read the entry for address zero and find that it is an internal node with a value of 2. The second bit of the string is a 0, so we proceed to address 2 (if this bit were a 1 we would have gone to address 3). We read the entry for address 2 and find that it is an internal node with a value of 6. The third bit of the string is a 1, so we proceed to one more than this value, address 7. Reading the entry for address 7 we find that it is a terminal node with a value of "T" (hex 14). We emit this value and reset our machine to start again from the root.

A block diagram of the Huffman decoder is shown in Figure 19.18 and Verilog code for the decoder is shown in Figure 19.19. The address of the current table node is held in the *node* register. When *type* is asserted — indicating a terminal node — *node* is set to the value of the next input bit (which selects one of the two children of the root to restart the search), the value field from the table is enabled into the output register, and *oval* is asserted on the following cycle. This outputs the current symbol and starts the machine at one of the children of the root depending on the first bit of the next symbol. If *type* is not assereted — indicating an internal node — the input value is combined with the value field from the table to select the left or right child of the current node — traversing the tree. The input value provides the least significant bit of the node address while the remaining bits come from the value field of the table. This simple concatenation is possible because all left children in the table have

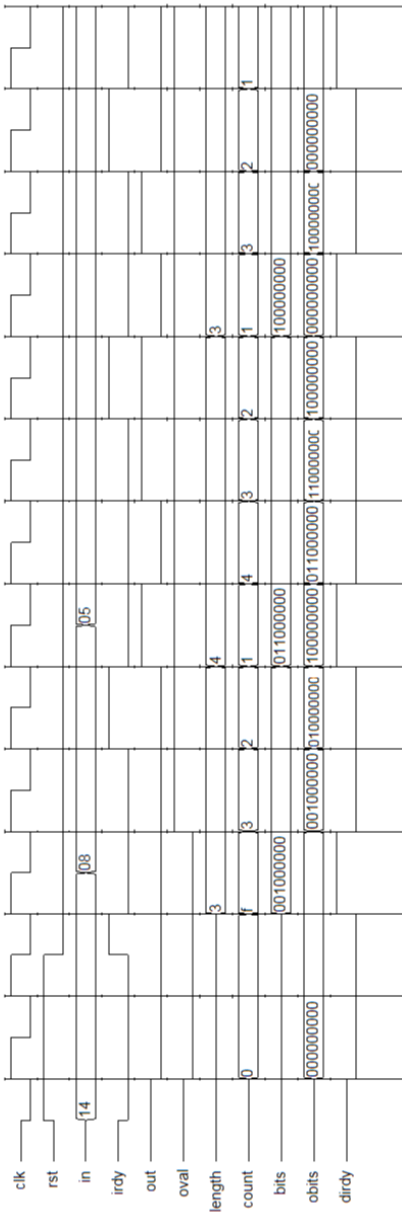

Figure 19.16: Waveforms from simulating the Huffman encoder on the string "THE".

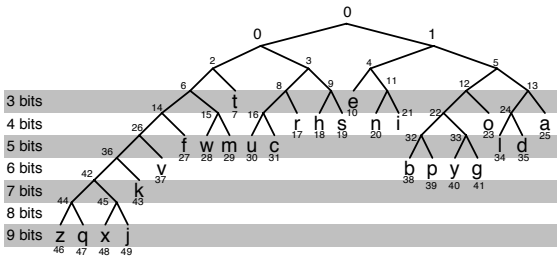

Figure 19.17: The Huffman code tree of Figure 19.13 relabeled to facilitate its storage in a decoding table. Each node of the tree is assigned a unique integer that serves as its address in the table.

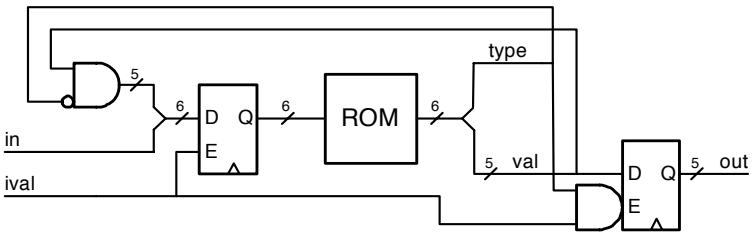

Figure 19.18: Block diagram of a Huffman decoder. The *node* register holds the address of the current tree node. The tree itself is stored in the ROM.

```
//------------------------------------------------------------------------
// Huffman Decoder - decodes bit-stream generated by encoder
//   in - bit stream
//   ival - true when new valid bit present
//   out - output character
//   oval - true when new valid output present
//------------------------------------------------------------------------
module HuffmanDecoder(clk, rst, in, ival, out, oval) ;
  input clk, rst, in, ival ;
  output [4:0] out ;
  output oval ;

  wire [5:0] node ; // pointers into table
  wire [4:0] value ; // output from table
  wire type ; // type from table
  wire ftype ; // fake a type on first ival cycle to prime pump

  wire [5:0] nnode = rst ? 6'd0
                         : (ival ? {(type|ftype) ? 5'b0 : value, in}
                                 : node) ;
  wire [4:0] nout = rst ? 5'd0
                        : ((ival & type) ? value : out) ;

  DFF #(6) node_reg(clk, nnode, node) ;
  HuffmanDecTable tab(node, {type, value}) ;
  DFF #(5) out_reg(clk, nout , out) ;
  DFF #(1) oval_reg(clk, ~rst & type & ival, oval) ;
  DFF #(1) ft_reg(clk, rst | (ftype & ~ival), ftype) ;
endmodule
```

Figure 19.19: Verilog description of the Huffman decoder.

even addresses. If *ival* goes low, the machine stalls, holding its present state until a valid input bit is available. Signal *ftype* in the Verilog model forces the machine to start from the root on the first valid input following reset.

Waveforms showing the combined operation of the Huffman encoder feeding the Huffman decoder are shown in Figure 19.20. The first 11 lines are the same as Figure 19.16 and represent the operation of the encoder encoding the symbol string "THE" into the bit string 0010110100. Signals `mid` and `mval` are output from the encoder and input (as `in` and `ival`) to the decoder. The state of the decoder is shown in the `node` variable and the `type` and `value` variables show what is read from the table at each node address. Note that each time `type` is asserted — indicating a leaf node — the search restarts on the next cycle with `node` at 0 or 1 (depending on `mid`). Also on the cycle following `type` the just

decoded symbol is output on `out` (values shown are hexadecimal) and `oval` is asserted to indicate a valid output.

19.5 A Video Display Controller

19.6 Exercises

19–1 *Divide-by-Four Counter:* Modify the counter from Section 19.1 to be a divide-by-four counter.

19–2 *Divide-by-Nine Counter:* Show how you can build a divide-by-nine counter using two divide-by-three counters. What happens to timing of the output pulse when you combine two counters?

19–3 *Divide-by-Three Mealy Machine:* Show how you can implement the divide-by-three counter of Section 19.1 using only three states if you allow the output to be a function of both the present state and the input. An FSM with a combinational path from input to output like this is called a *Mealy Machine* while an FSM where the output is a function only of the present state is called a *Moore Machine*.

19–4 *Modified SOS Machine:* Modify the factored SOS detector of Section 19.2 so that a dot is defined as one or two consecutive 1s and a dash is defined as 3 or 4 consecutive 1s.

19–5 *Further SOS Modifications:* Further alter the modified SOS machine from Exercise 19–4 so that the pauses between dots and dashes within a character may be 1 or 2 consecutive 0s, the spaces between characters are 3 or 4 consecutive 0s, and 5 or more consecutive 0s is an inter-word space. SOS should appear as a single word to be recognized.

19–6 *Huffman Encoder with Flow Control:* Modify the Huffman Encoder of Section 19.4.1 to accept an input valid signal *ival* that is true when a valid symbol is available on the input. A new symbol will be accepted only when both *ival* and *irdy* are asserted. Note that the output valid signal *oval* may need to go low after a string is shifted out if there is a wait for the next input signal.

19–7 *More Flow Control:* Take the Huffman encoder from Exercise 19–6 and extend it further to accept an output ready *ordy* signal that is true when the module connected to the output is ready to accept the next bit.

19–8 *One-Bit Strings:* Modify the Huffman encoder of Section 19.4.1 so it will work with a length of one. That is for codes where a symbol may be represented by a one-bit string.

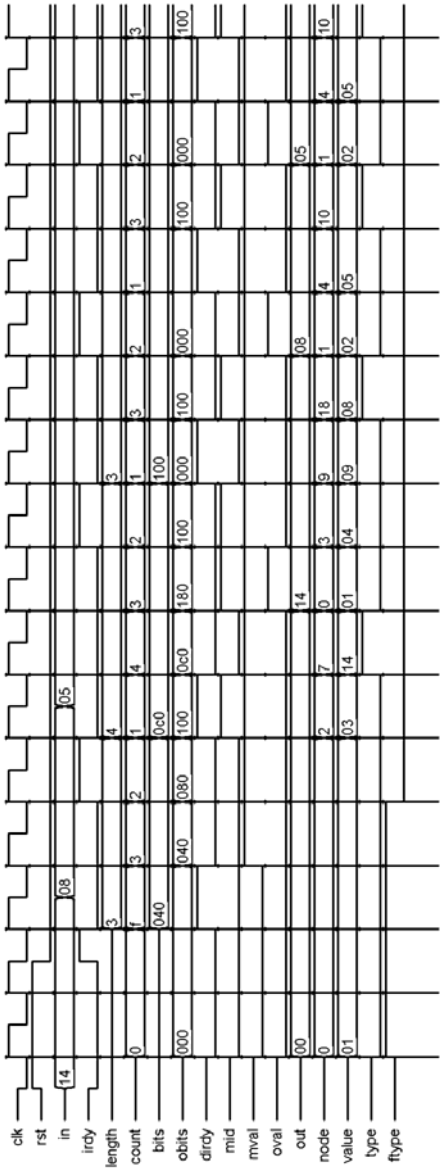

Figure 19.20: Waveforms of Huffman encoder and decoder, encoding the string "THE" into 0010110100 and then decoding this bit string back to "THE".

Chapter 20

System-Level Design

At this point in the course you now have the skills to design complex combinational and sequential logic modules. However, if someone were to ask you to design a DVD player, a computer system, or an internet router you would realize that each of these is not a single finite-state machine (or even a single datapath with associated finite-state controller). Rather, a typical system is a collection of modules each of which may include several datapaths and finite-state controllers. Once the system is decomposed to simple modules the design and analysis skills you have learned in the previous chapters can be applied. However, the problem remains to parition the system to this level where the design becomes manageable. This *system-level design* is one of the most interesting and challenging aspects of digital systems.

some idioms of system design

20.1 The System Design Process

The design of a system involves the following steps:

Specification: The most important step in designing any system is deciding - and clearly specifying in writing - what you are going to build. We discuss specifications in more detail in Section 20.2.

Partitioning: Once the system is specified, the main task in system design is dividing the system into manageable subsystems or modules. This is a process of divide and conquer. The overall system is divided into subsystems that can then be designed (conquered)separately. At each stage, the sybsystems should be specified to the same level of detail as the overall system was during our first step. As described in Section 20.3 we can partition a system by state or by task.

Interface Specification: It is particularly important that the interfaces between subsystems be described in detail. With good interface specifications, individual modules can be developed and verified independently.

When possible, interfaces should be independent of module internals — allowing modules to be modified without affecting the interface, or the design of neighboring modules.

Timing Design: Early in the design of a system, it is important to describe the timing and sequencing of operations. In particular, as work flows between modules, the sequencing of which module does a particular task on a particular cycle must be worked out to ensure that the right data comes together at the correct place and time. This timing design also drives the performance tuning step described below.

Module Design: Once the system is partitioned, modules and interfaces have been specified, and the system timing has been worked out, the individual modules can be designed and verified independently. Often the exact performance and timing (e.g., throughoput, latency, or pipeline depth) of a module is not known exactly until after the module design is complete. As these performance parameters are finalized, they may affect the system timing and require performance tuning to meet system performance specifications. The test of a good system design is if such independently designed modules can be assembled into a working system without rework.

Performance Tuning: Once the performance parameters of each module are known (or at least estimated), the system can be analyzed to see if it meets its performance specification. If a system falls short of a performance goal — or if the goal is to achieve the highest performance at a given cost — performance can be tuned by adding parallelism. This topic is treated in more detail in Chapter 21.

20.2 Specification

All too often people start desigining a system without a clear specification only to discover half-way (or further) through the design that they are building the wrong system. Much work is then discarded as they restart the design process. Another problem with vague specifications is that two designers may read the specification differently and design incompatible system parts.

A system design may start from an oral discussion of requirements. However, writing the specification down is a critical step to make sure that there are no misunderstandings about what is being designed. A written specification can also be used to validate that the right system is being designed by reviewing the specification with prospective customers and users of the system.

A good specification at a minimum must include a description of:

1. An overall description. What the system is, what it does, and how is it used.

2. All inputs and outputs: their formats, range of values, timing, and protocols.

Name	Direction	Width	Description
leftUp	input	1	when true moves the left paddle up.
leftDown	input	1	when true moves the left paddle down.
leftStart	input	1	when true starts the game or serves the ball from left to right.
rightUp	input	1	when true moves the right paddle up.
rightDown	input	1	when true moves the right paddle down.
rightStart	input	1	when true starts the game or serves the ball from right to left.
red	output	8	the intensity of the red color on the screen at the current pixel.
green	output	8	the intensity of the green color on the screen at the current pixel.
blue	output	8	the intensity of the blue color on the screen at the current pixel.
hsync	output	1	horizontal synchronization — when asserted starts a horizontal retrace of the screen.
vsync	output	1	vertical synchronization — when asserted starts a vertical retrace of the screen.

Table 20.1: Inputs and Outputs of Pong System

3. All user visible state. This includes configuration registers, mode bits, and internal memories.

4. All *modes* of operation.

5. All notable *features* of the system.

6. All interesting *edge cases*, i.e., how the system handles marginal cases.

The remainder of this section gives specifications for three example systems: a "pong" game, a DES cracker, and a music player. We will then perform the system-level design of these three examples in the remainder of the chapter.

20.2.1 Pong

Overall Description: Pong is a video game. It displays a ping-pong-like game on a VGA video screen. Users control the game using push-buttons to move the paddles and serve. Games are played to 11 points. The player winning the last point serves. The screen is considered to be a 64 by 64 grid for purposes of the game — point (0,0) is top left.

Inputs and Outputs: The inputs and outputs of our Pong system are specified in Table 20.1. Note that our digital module produces as output digital red, green, blue, and sync outputs for the display. A separate analog module combines these signals to produce the analog signals to drive the display.

Name	Width	Description
rightPadY	6	y-position of the top of the right paddle.
leftPadY	6	y-position of the top of the left paddle.
ballPosX	6	x-position of the ball.
ballPosY	6	y-position of the ball.
ballVelX	1	x velocity of the ball (0=left, 1=right).
ballVelY	2	y velocity of the ball (00=none, 01=up, 10= down).
rightScore	4	score of right player.
leftScore	4	score of left player.
mode	2	current mode - idle, rserve, lserve, play.

Table 20.2: User visible state of the Pong system

Name	Description
idle	score is zero, awaiting first serve. First start button pressed zeros score and starts game with serve from that direction. (e.g., lstart serves from left to right).
play	ball in play. Ball advances according to velocity. Hitting top or bottom of court reverses y velocity. Hitting paddle reverses x velocity. Missing left or right paddle enters rserve or lserve mode respectively and increments appropriate score.
lserve	waiting for left player to serve. When lstart is pressed ball is served from left to right.
rserve	waiting for right player to serve. When rstart is pressed ball is served from right to left.

Table 20.3: Modes of the Pong system

State: The visible state of the Pong system is shown in Table 20.2. Most of this state represents the positions of game elements on the video screen. This state is visible in the sense that it can be seen on the display. The user cannot directly read or write this state.

Modes: The modes of the pong system are shown in Table 20.3.

While we have given many details of the pong video game, this specification is by no means complete. Things left unspecified include the value of the ball position and velocity on a serve, how the y-velocity of the ball changes when hitting a paddle, and the height of a paddle. A complete specification should leave nothing to the imagination. In Exercise 20–1 the reader is given the task of completing the specification of the pong game. In practice, specifications are typically completed in an iterative manner with additional details specified on each iteration. Specification reviews are often held where a specification is presented to a group and reviewed critically to identify missing or incorrect items.

Name	Direction	Width	Description
cipherText	input	8	Cipher text to be cracked. This text is input one byte at a time. A byte is accepted on each clock for which cipherTextValid and cipherTextReady are asserted.
cipherTextValid	input	1	Asserted when cipherText has the next valid byte of cipher text to load.
cipherTextReady	output	1	Asserted when the system is able to accept a byte of cipher text.
start	input	1	Start key search. Asserted when loading of cipher text is complete to direct the system to start search of key space.
found	output	1	Asserted when the key is found.
key	output	56	When found is asserted the recovered key is output on this signal.

Table 20.4: Inputs and Outputs of the DES cracker system

20.2.2 DES Cracker

Overall Description: The DES cracker is a system that accepts a portion of ciphertext encrypted using the data encryption standard and searches the space of possible keys to find the key that was used to encrypt the ciphertext. The system assumes that the original plaintext is plain ASCII text that uses only capital letters and numbers.

Inputs and Outputs: The inputs and outputs of the DES cracker are shown in Table 20.4

State:

Modes:

20.2.3 Music Player

Music player

20.3 Partitioning

Much of system design is partitioning a system into modules. While many consider this to be an art, most systems are partitioned by state, task, or interface. With state partitioning, the system is divided into modules associated with different pieces of state (user visible or strictly internal). Each module is responsible for maintaining its portion of system state and communicating appropriate *views* of this state to other modules in the system.

With task partitioning, the function performed by a system is divided into tasks and separate modules are associated with each task. A common form

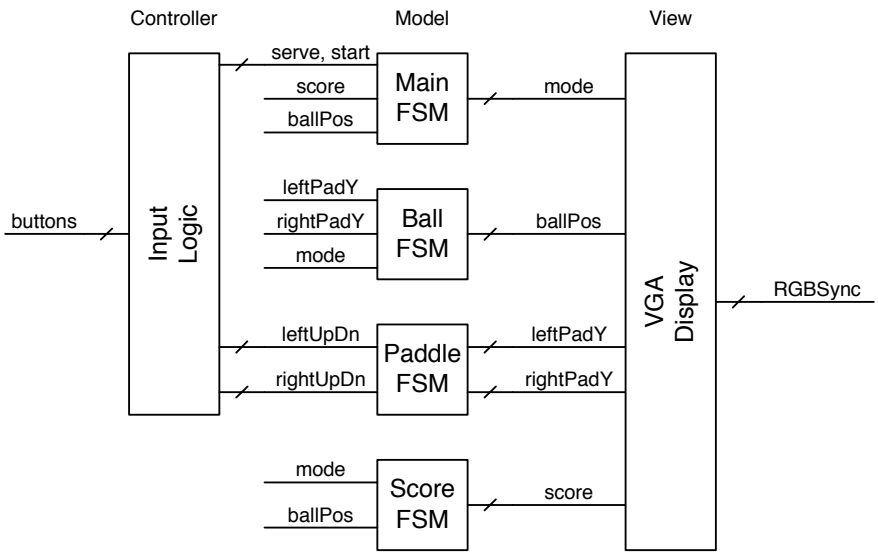

Figure 20.1: The Pong game is partitioned using the model-view-controller decomposition. The score and position of the ball and paddles constitute the *model*. This model is viewed by a VGA display module. A controller module conditions input buttons to affect the model. Note that the model is further partitioned by state into separate modules for ball, paddles, and score.

of task partitioning is model-view-controller partitioning. The system is partitioned into a model module — that contains most of the system funciton — a view module — that is responsible for all output (views of the model) — and a controller module — that is responsible for all input (controlling the model).

Finally, with interface partitioning, a separate module is associated with each interface (or related set of interfaces) of a system. Most systems employ some combination of these three partitioning techniques.

20.3.1 Pong

Figure 20.1 shows how the pong video game system is partitioned along two axes. Horizontally the system is partitioned by task, into model, view, and controller. The model portion of the system is further partitioned vertically by state with the ball state, the paddle state, the score, and the mode in separate modules. The figure also shows the interfaces defined between the modules. In most cases a module simply exports all or part of its state (e.g., score, ballPos).

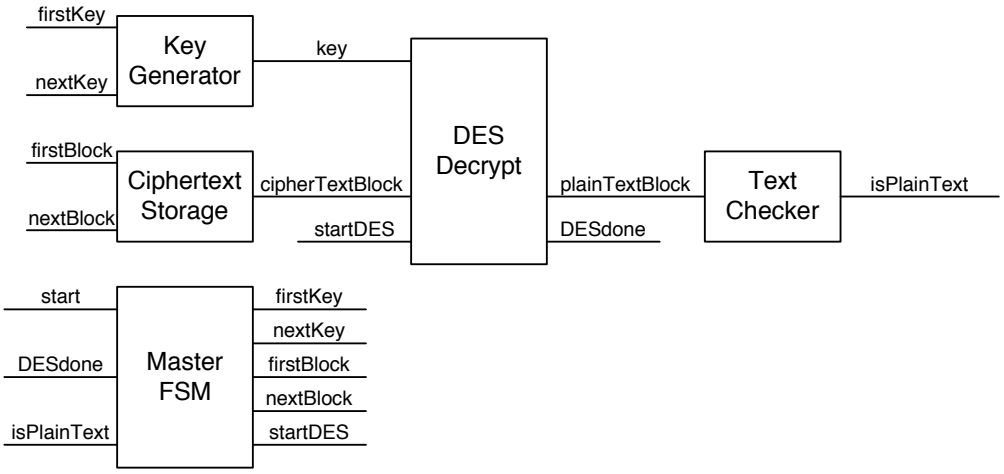

Figure 20.2: A DES *cracker* is partitioned by task. The modules perform the subtasks of generating keys, sequencing the ciphertext, decrypting the ciphertext to plaintext, and checking the plaintext to see if it is indeed plain text. A master FSM module controls overall timing and sequencing.

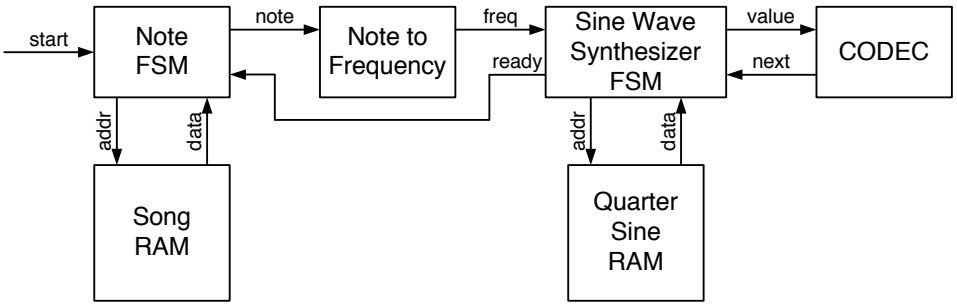

Figure 20.3: A simple music synthesizer is partitioned by task. A note FSM determines the next note to play. A note-to-frequency block converts the note into a frequency. A sine-wave synthesizer FSM synthesizes a sine wave of the specified frequency.

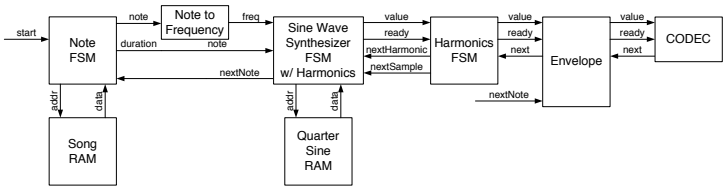

Figure 20.4: An expanded version of the music synthesizer uses a sine-wave synthesizer module that returns the value of multiple harmonics for each note, a harmonics FSM that combines these harmonics, and an envelope FSM that modulates the resulting waveform with an attack-decay envelope.

20.3.2 DES Cracker

20.4 Modules and Interfaces

modules and interfaces
- valid/ready flow control
- an example

20.5 System-Level Timing

20.6 System-Level Examples

20.7 Exercises

20–1 *Pong Specification.* The specification of the pong video game in Section 20.2.1 is purposely incomplete. Some of the missing specifications are listed in the text. Identify as many unspecified issues as you can and give a specification of each.

Chapter 21

Pipelines

A pipeline is a sequence of modules, called *stages* that each perform part of an overall task. Each stage is like a station along an assembly line — it performs a part of the overall assembly and passes the partial result to the next stage. By passing the incomplete task down the pipeline, each stage is able to start work on a new task before waiting for the overall task to be completed. Thus, a pipeline may be able to perform more tasks per unit time (i.e., it has greater *throughput*) than a single module that performs the whole task from start to finish.

load balance

elastic

21.1 Basic Pipelining

Suppose you have a factory that assembles toy cars. Assembling each car takes four steps. In step 1 the body is shapred from a block of wood. In step 2 the body is painted. In step 3 the wheels are attached. Finally, in step 4, the car is placed in a box. Suppose each of the four steps takes 5 minutes. With one employee, your factory can assemble one toy car each 20 minutes. With four employees your factory can assemble one car every 5 minutes in one of two ways. You could have each employee perform all four steps — producing a toy car every 20 minutes. Alternatively, you could arrange your employees in an assembly line with each employee performing one step and passing partially completed cars down to the next employee.

In a digital system, a pipeline is like an assembly line. We take an overall task (like building a toy car) and break it into subtasks (each of the four steps). We have a separate unit, called a pipeline stage (like each employee along the assembly line), perform each task. The stages are tied together in a linear manner so that the output of each unit is the input of the next unit — like the employees along the assembly line passing their output (partially assembled cars) to the next employee down the line.

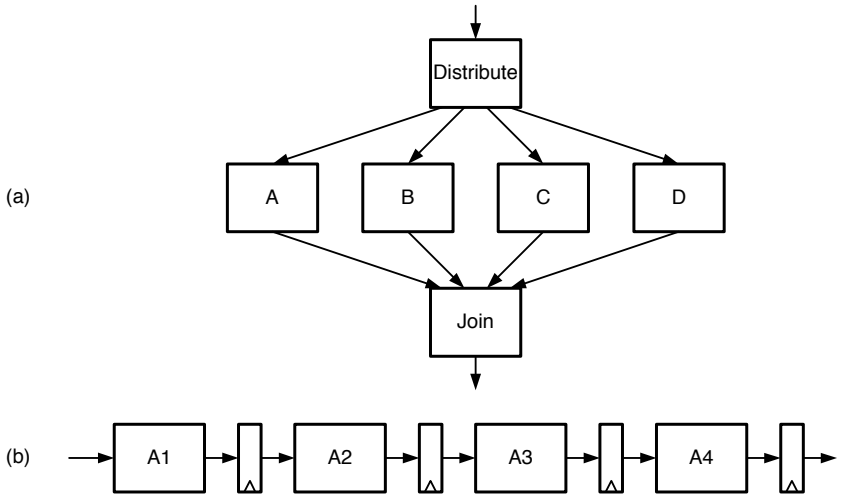

Figure 21.1: Throughput of a module can be increased by (a) using parallel copies of the module, or (b) pipeling a single copy of the module.

The *throughput* Θ of a module is the number of problems a module can solve (or tasks a module can perform) per unit time. For example, if we have an adder that is able to perform one add operation every 10ns, we say that the throughput of the adder is 100Mops. The latency T of a module is the amount of time it takes the module to complete one task from begininng to end. For example, if our adder takes 10ns to complete a problem from the time the inputs are applied to the time the output is stable, its latency is 10ns. Latency is just another word for delay. For a simple module, throughput and latency are reciprocals of one another: $\Theta = \frac{1}{T}$. If we accelerate modules through pipelining or parallelism, however, the relation becomes more complex.

Suppose we need to increase the throughput of a module with $T = 10$ns, $\Theta = 100$Mops by a factor of four. Further suppose the module is already highly optimized so that we are unlikely to get much increase in throughput by redesigning the module. Just as with our toy car factory, we have two basic options. First, we could build four copies of our module as shown in Figure 21.1(a). Modules A through D are identical copies of our original module. The distribute block distributes problems to the four modules, and the join block combines the results. Here we can start four copies of our problem in parallel as shown in Figure 21.2(a). Our latency is still $T = 10$ns, because it still takes 10ns to complete one problem from beginning to end. Our throughput, however, has been increased to $\Theta = 400$Mops since we are able to solve four problems every 10ns.

An alternative method of increasing throughput is to *pipeline* a single copy of the module as shown in Figure 21.1(b). Here we have taken a single module

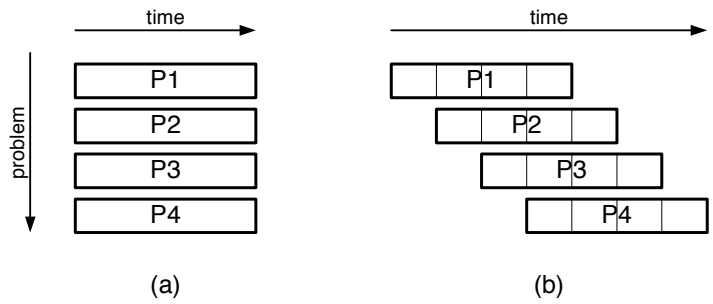

Figure 21.2: Timing diagram showing execution of four problems $P1, \ldots, P4$ on the parallel and pipelined configurations of Figure 21.1.

A and divided it into four parts $A1, \ldots, A4$. We assume that we were able to do this partition evenly so that the delay of each of the four submodules Ai is $T_{Ai} = 2.5$ns. (Such an even division of modules is not always possible as discussed below in Section 21.3). We insert a register between adjacent submodules. Each register holds the result of the preceding submodule on one problem freeing that submodule to begin working on the next problem. Thus, as shown in Figure 21.2(b), this pipeline can operate on four problems at once in a staggered fashion. As soon as submodule $A1$ finishes work on problem $P1$, it starts working on $P2$ while $A2$ continues work on $P1$. Each problem continues down the pipeline, advancing one stage each clock cycle, until it is completed by module $A4$. If we ignore (for now) register overhead, our latency is still $T = 10$ns (2.5ns $\times$ 4 stages) and our throughput has been increased to 400Mops.

Compared to using parallel moudules, pipelining has the advantage that it multiplies throughput without the cost of duplicating modules. Pipelining, however, is not without its own costs. First, pipelining requires inserting registers between pipeline stages. In some cases, these registers can be very expensive. Also, a pipelined implementation has more register overhead than a corresponding parallel design.

Now consider the affect of register overhead. Suppose each register has a total overhead $t_{\rm reg} = t_s + t_{dCQ} + t_k = 200$ps. For a single module or a parallel combination (Figure 21.1(a)), we only pay this overhead once, increasing our latency to $T = 10.2$ns and reducing the throughput of four parallel modules to $\Theta = 4/10.2 = 392$Mops. Pipelining the module, on the other hand, incurs the register overhead once per stage. Thus our latency is increased to $T = 10.8$ns and our througput is reduced to $\Theta = 1/2.7 = 370$Mops.

In practice, many implementations use parallel pipelined modules. One pipelines a module until the cost of the registers and the register overhead becomes expensive and then obtains further throughput increases by using multiple copies of the pipelines working in parallel.

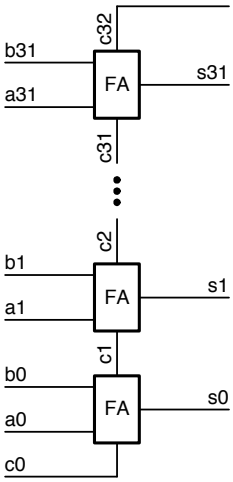

Figure 21.3: A 32-bit ripple carry adder. If each stage requires 100ps, an add can be performed in 3.2ns.

21.2 Example: Pipelining a Ripple-Carry Adder

As a more concrete example of pipelining, consider a 32-bit ripple-carry adder as shown in Figure 21.3. If each full-adder module has a delay from carry-in to carry-out of $t_{dcc} = 100$ps, then in the worst-case (where the carry must propagate all the way from the LSB to the MSB), the delay of the adder is $t_{d\text{add}} = 32t_{dcc} = 3.2$ns. This single adder is capable of performing one add every 3.2ns for a throughput of 312.5 million adds per second. Suppose our goal is to achieve a throughput of 1 billion adds per second — starting a new add every 1ns. We can pipeline this adder to achieve this goal.

In operation, only a single bit of this adder is *busy* at any point in time. For example, 1ns after the inputs are applied only bit 10 is active. Bits 0 through 9 have completed operation, and bits 11-31 are waiting on their carry inputs. (They have computed p and g, but cannot compute s until the carry input is available). By pipelining the adder n ways, we can get n of the 32-bits working at a given point in time.

The first step in pipelining a unit is to divide it into submodules. Figure 21.4 shows our 32-bit adder module divided into four submodules each of which performs an 8-bit add. Each 8-bit adder accepts an 8-bit slice of each input vector, a[i+7:i] and b[i+7:i], along with a carry in ci and generates an 8-bit slice of the sum s[i+7:i] and a carry out c[i+8]. The delay (from carry-in to carry-out) of each of these 8-bit adders is $t_{d8} = 8t_{dcc} = 800ps$ so it will fit in a 1ns clock cycle leaving 200ps for register overhead.

To allow our partitioned adder to work on muliple problems at the same

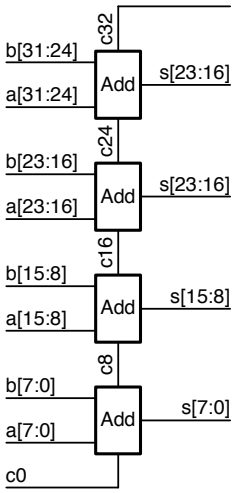

Figure 21.4: Dividing the 32-bit ripple-carry adder into four 8-bit adders.

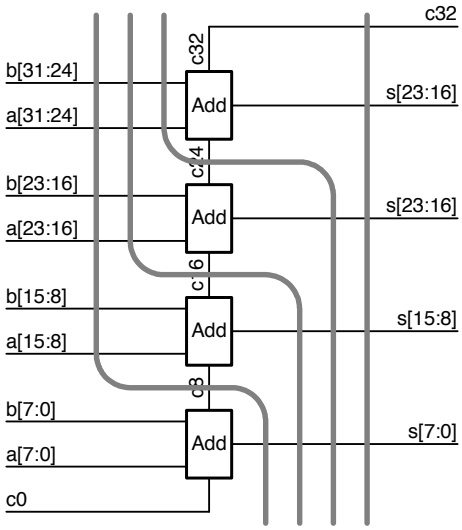

Figure 21.5: To pipeline the 32-bit ripple-carry adder, we insert registers to separate the stages at locations shown by the thick, gray lines. All paths from input to output must pass through each pipeline register.

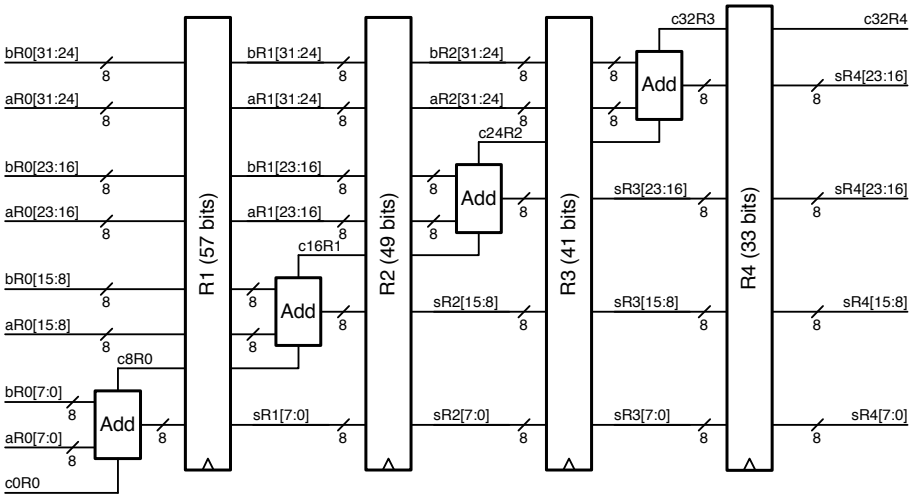

Figure 21.6: The pipelined 32-bit ripple-carry adder. Registers have been inserted and the schematic redrawn to make each pipeline stage one horizontal region. Signal names are augmented with the name of the pipeline register producing them.

time, we insert *pipeline registers* between the submodules. The thick, gray lines in Figure 21.5 show where the registers are to be inserted. The circuit redrawn after inserting the registers is shown in Figure 21.6. The combination of a submodule and its register is referred to as a *pipeline stage*. Pipeline register $R1$ is inserted after the first 8-bit adder. This register captures the partial result of the add after 800ps — this includes the low eight-bits of the sum sR0[7:0], bit 8 of the carry c8R0, and the upper 3-bytes of the input vectors aR0[31:8] and bR0[31:8] for a total of 57 bits.

We label all signals before $R1$ (including the primary inputs) with the suffix $R0$ to denote that they are in the 0^{th} pipeline stage and to distingish these signals from signals in other pipeline stages. In a similar manner, the outputs of $R1$, and all other signals in first pipeline stage, are labeled with a suffix $R1$. Note that signals sR1[7:0] and sR2[7:0] are different signals. While they are both the low byte of a sum, at any point in time they are the low bytes of different sums. By labeling signals with their pipeline stage we can easily spot the common pipeline error of combining signals from different stages. We know that an expression `fooR1 & barR2` is an error because signals from different pipeline stages should not be combined.

The second stage of the pipeline adds the second byte of data aR1[15:8] and bR1[15:8], using c8R1 as the carry in, giving sR1[15:8] and c16R1. The outputs of this second byte adder are captured by $R2$ along with sR1[7:0] and aR1[31:16] and bR1[31:16], a total of 49 bits. In a similar manner the third and

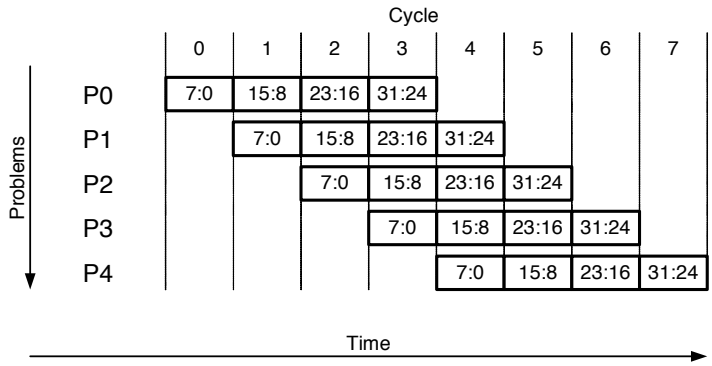

Figure 21.7: Pipeline diagram showing the timing of the pipelined 32-bit ripple-carry adder. The diagram illustrates on which cycle each sum byte of problems $P0, \ldots, P4$ is computed.

fourth pipeline stages add the third and fourth bytes of data. At the output of the fourth stage, register $R4$ captures the full 32-bit product sR3[31:0] and the carry out c32R3. The output of $R4$ is the result of the add: s[31:0] and c32.

Timing of the pipeline is illustratd in the *pipeline diagram* of Figure 21.7. The figure shows five problems, $P0, \ldots, P4$, proceeding down the pipeline. Problem Pi enters the pipeline on cycle i. Byte j (bits [8j+7:8j]) of the sum of Problem i is computed during cycle $i + j$. The result of the add for problem Pi appears on the output four clock cycles later, on cycle $i + 4$. At the time the result for a problem $P0$ appears on the output during cycle 4, four subsequent problems are in process in the various pipeline stages.

Pipelining our ripple-carry adder has increased both its throughput and its latency. If we assume that register overhead $t_o = t_s + t_{dCQ} + t_k$ is 200ps for each register, then the delay of each pipeline stage is 1ns — 800ps for the 8-bit adder, and 200ps for register overhead. Hence we can operate our pipeline at 1GHz, achieving our throughput goal of $\Theta = 1$Gops. The latency of our pipeline is $T = 4$ns, compared to the original 3.2ns for the unpiplined adder. The difference is the overhead of the four registers.

In general, if the delay of the combinational logic in the longest pipeline stage is $t_{\max}$, then the total delay of a pipeline stage is:

$$t_{\text{pipe}} = t_{\max} + t_o = t_{\max} + t_s + t_{dCQ} + t_k. \tag{21.1}$$

From t_{pipe} we see the latency of n stages is:

$$T = n\left(t_{\max} + t_o\right) = n\left(t_{\max} + t_s + t_{dCQ} + t_k\right), \tag{21.2}$$

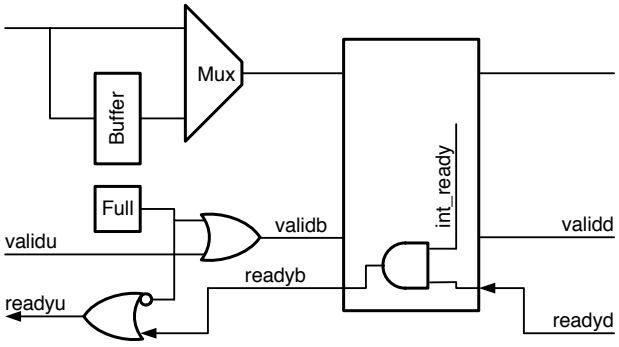

Figure 21.8:

and the throughput is:

(21.3)

21.3 Load Balancing

21.4 Variable Loads

21.5 Double Buffering