first round. Each match in the tournament is performed using a magnitude comparator (Section 8.5). To break ties in favor of the lower number input, the magnitude comparator computes a signal $c_{igt0}$ that is true if $in_1 \lt in_0$. If they are tied, this signal is false, indicating that $in_0$ has won the match. A similar comparison is made between $in_3$ and $in_2$.

To select the competitors for the second round, two 2:1 multiplexers (Section 8.2) are used. Each multiplexer selects the winner of a first round match using the comparator output as the select signal.

A third magnitude comparator performs the second round match - comparing the two winners output by the multiplexers. The output of this second round comparator is the MSB of the priority arbiter. If this signal is true, the winner is $in_2$ or $in_3$; if its false, the winner is $in_0$ or $in_1$.

To get the LSB of the priority arbiter output, we select the output of the winning first round comparator. This is accomplished with a single-bit-wide 2:1 multiplexer controlled by the output of the final comparator.

### 9.4 Tic-Tac-Toe

In this section we develop a combinational circuit that plays the game of tic-tac-toe. Given a starting board position, it selects the square on which to play its next move. Being a combinational circuit, it can only play one move. However, it can easily be transformed into a sequential circuit (Chapter 14) that plays an entire game.

Our first task is to decide how to represent the playing board. We represent the input board position as two nine-bit vectors: one $xin$ encodes the position of the X's and the other $oin$ encodes the position of the O's. We map each nine-bit vector to the board as shown in Figure 9.10(a). The upper left corner is the LSB and the bottom right corner is the MSB. For example, the board shown in Figure 9.10(b) is represented by $xin = 100000001$ and $oin = 00011000$. For a legal board position, $xin$ and $oin$ must be orthogonal. That is $xin \land oin = 0$.

Strictly speaking, a legal board should also have $NO \geq NX \geq NO - 1$, where $NO$ is the number of bits set in $oin$ and $NX$ is the number of bits set in $xin$. If X goes first, the input should always have equal numbers of bits set in the two inputs. If O goes first, the input will always have one more bit set in $oin$ than in $xin$.

Our output will also be a nine-bit vector $xout$ that indicates which position our circuit will be playing. A legal move must be orthogonal to both input vectors. On the next turn $xin$ will be replaced by the OR of the old $xin$ and $xout$ and the opponent will have added a bit to $oin$.

Now that we have represented the board, our next step is to structure our circuit. A useful structure is as a set of ordered strategy modules that each apply a strategy to generate the next move. The highest priority module that is able to generate a move is selected. For example, a good set of strategy modules is:
module PriorityArbiter(in0, in1, in2, in3, out) ;
parameter n=8; // width of inputs
input [n-1:0] in0, in1, in2, in3 ;
output [1:0] out ;
wire [n-1:0] match0winner, match1winner ;
wire [1:0] out ;

// first round of tournament
MagComp #(n) round0match0(in1, in0, c1gt0) ; // compare in0 and in1
MagComp #(n) round0match1(in3, in2, c3gt2) ; // compare in2 and in3

// select first round winners
Mux2 #(n) match0(in0, in1, {c1gt0, ~c1gt0}, match0winner) ;
Mux2 #(n) match1(in2, in3, {c3gt2, ~c3gt2}, match1winner) ;

// compare round0 winners
MagComp #(n) round1(match1winner, match0winner, out[1]) ;

// select winning LSB index
Mux2 #(1) winningLSB(c1gt0, c3gt2, {out[1], ~out[1]}, out[0]) ;
endmodule

Figure 9.9: Verilog description of a four-input priority arbiter.
1. Win: If a move will complete three-in-a-row do it.

2. Don’t lose: If a move will block an opponent with two-in-a-row do it.

3. Pick first open square: Traversing the board in a particular order, pick the first square that is open.

A selection circuit combines the inputs from our modules and selects the highest priority module with an output. With this modular design, we can easily add more strategy modules later to refine the ability our our circuit.

The top-level module for our tic-tac-toe move generator is shown in Figure 9.11 and the Verilog for this module is given in Figure 9.12. It instantiates four modules: two instances of TwoInArray, and one instance each of Empty and Select3. The first TwoInArray module finds spaces (if any) where a play would cause us to win - that is spaces where there is a row, column, or diagonal with two Xs and no Os. The second TwoInArray module finds spaces (if any) where if we didn’t play, the opponent could win on their next play - spaces where a row, column or diagonal has two Os and no Xs. We use the same module for both the win and the block strategies because they require the same function - just with the Xs and Os reversed. The next module Empty finds the first empty space according to a particular ordering of the spaces. The ordering picks empty spaces in order of their strategic value. Finally, the module Select3 takes the three outputs of the previous modules and selects the highest-priority move.

Most of the work in our tic-tac-toe implementation is done by the TwoInArray module shown in Figure 9.13. This module creates eight instances of the TwoInRow module (Figure 9.14). Each TwoInRow module checks one line (row, column, or diagonal). If the line being checked has two bits of a true and no bits of b true, a 1 is generated in the position of the open space. The module consists of three four-input AND gates, one for each of the three positions being checked. Note that we only check one bit of b in each AND gate since we are assuming that the inputs are legal, so that if a bit of a is true, the corresponding bit of b is false.

Three instances of TwoInRow check the rows producing their result into the nine-bit vector rows. If a bit of rows is true, playing an a into the corresponding space will complete a row. Similarly three instances of TwoInRow check the three
Figure 9.11: High-level design of the tic-tac-toe module. Three strategy modules accept the inputs, xin and oin, and compute possible moves to win, not lose, or pick an empty square. The Select3 module then picks the highest priority of these possible moves to be the next move.
module TicTacToe (xin, oin, xout) ;
    input [8:0] xin, oin ;
    output [8:0] xout ;
    wire [8:0] win, block, empty ;
    TwoInArray winx(xin, oin, win) ; // win if we can
    TwoInArray blockx(oin, xin, block) ; // try to block o from winning
    Empty emptyx(~(oin | xin), empty) ; // otherwise pick empty space
    Select3 comb(win, block, empty, xout) ; // pick highest priority
endmodule

Figure 9.12: Top-level Verilog description for our tic-tac-toe move generator.
module TwoInArray(ain, bin, cout) ;
    input [8:0] ain, bin ;
    output [8:0] cout ;

    wire [8:0] cout ;
    wire [8:0] rows, cols ;
    wire [2:0] ddiag, udiag ;

    // check each row
    TwoInRow topr(ain[2:0], bin[2:0], rows[2:0]) ;
    TwoInRow midr(ain[5:3], bin[5:3], rows[5:3]) ;
    TwoInRow botr(ain[8:6], bin[8:6], rows[8:6]) ;

    // check each column
    TwoInRow leftc({ain[6], ain[3], ain[0]},
                    {bin[6], bin[3], bin[0]},
                    {cols[6], cols[3], cols[0]}) ;
    TwoInRow midc({ain[7], ain[4], ain[1]},
                   {bin[7], bin[4], bin[1]},
                   {cols[7], cols[4], cols[1]}) ;
    TwoInRow rightc({ain[8], ain[5], ain[2]},
                   {bin[8], bin[5], bin[2]},
                   {cols[8], cols[5], cols[2]}) ;

    // check both diagonals
    TwoInRow dndiagx({ain[8], ain[4], ain[0]}, {bin[8], bin[4], bin[0]}, ddiag) ;
    TwoInRow updiagx({ain[6], ain[4], ain[2]}, {bin[6], bin[4], bin[2]}, udiag) ;

    // OR together the outputs
    assign cout = rows | cols |
        {ddiag[2], 1'b0, 1'b0, 1'b0, ddiag[1], 1'b0, 1'b0, 1'b0, ddiag[0]} |
        {1'b0, 1'b0, udiag[2], 1'b0, udiag[1], 1'b0, udiag[0], 1'b0, 1'b0} ;
endmodule
// TwoInRow
// Indicates if a row (or column, or diagonal) has two pieces of type a
// and no pieces of type b. (a and b can be x and o or o and x)
// Inputs:
//  ain, bin - (3 bits) row of types a and b.
// Outputs:
//   cout - (3 bits) location of empty square if other two are type a.
//------------------------------------------------------------------------------
module TwoInRow(ain, bin, cout);

input [2:0] ain, bin;
output [2:0] cout;
wire [2:0] cout;

assign cout[0] = ~bin[0] & ~ain[0] & ain[1] & ain[2];
endmodule

Figure 9.14: Verilog description of the TwoInRow module. This module outputs a 1 in the empty position of a row that contains two bits of a and no bits of b.

columns for two bits of a and no bits of b, producing results into nine-bit vector cols. The final two instances of TwoInRow check the two diagonals producing results into three-bit vectors ddiag and udiag for the downward sloping and upward sloping diagonals.

After checking the rows, columns, and diagonals, the final assign statement combines the results into a single 9-bit vector by ORing together the individual components. The rows and cols vectors are combined directly. The 3-bit diagonal vectors are first expanded to 9-bits to place their active bits in the appropriate positions.

The Empty module, shown in Figure 9.15 uses an arbiter (Section 8.4) to find the first non-zero bit in its input vector. Note that the top-level has ORed the two input vectors together and taken the complement so each 1 bit in the input to this module corresponds to an empty space. The input vector is permuted, using a concatenation statement, to give the priority order we want (middle first, then corners, then edges). The output is permuted in the same order to maintain correspondence.

The Select3 module, shown in Figure 9.16 is also just an arbiter. In this case, a 27-bit arbiter scans all three inputs to find the first bit. This both selects the highest priority non-zero input and also selects the first set bit of this input. The 27-bit output of the arbiter is reduced to 9 bits by ORing the bits corresponding to each input together.
module Empty(in, out);
    input [8:0] in;
    output [8:0] out;

    RArb #(9) ra({in[4], in[0], in[2], in[6], in[8], in[1], in[3], in[5], in[7]}),
           {out[4], out[0], out[2], out[6], out[8], out[1], out[3], out[5], out[7]});
endmodule

Figure 9.15: Verilog description of a Empty module. This module uses an arbiter to find the first empty space searching first the middle space, then the four corners, then the four edges.

module Select3(a, b, c, out);
    input [8:0] a, b, c;
    output [8:0] out;
    wire [26:0] x;

    RArb #(27) ra({a,b,c},x);

    wire [8:0] out = x[26:18] | x[17:9] | x[8:0];
endmodule

Figure 9.16: Verilog description of the Select3 module. A 27-input arbiter is used to find the first set bit of the highest priority strategy module. A three-way OR combines the arbiter outputs.
It is worth pointing out that the entire tic-tac-toe module is at the bottom level built entirely from just two module types: `TwoInRow`, and `RArb`. This demonstrates the utility of combinational building blocks.

A simple test bench for the tic-tac-toe module is shown in Figure 9.17. The test bench instantiates two copies of the TicTacToe module. One plays X and the other plays O. The test bench starts by checking the module that plays X, called `dut` in the test bench, with some directed testing. The five vectors check empty, win, and block strategies and check row, column, and diagonal patterns.

After the five directed patterns, the test bench plays a game of TicTacToe by ORing the outputs of each module into its input to compute the input for the next round. The results of the game (obtained by writing a script to massage the output of the `$display` statements) is shown in Figure 9.18.

The game starts with an empty board. The `empty` rule applies and X plays to the center - our highest priority empty space. The `empty` rule applies for the next two turns as well and O and X take the top two corners. At this point X has two in a row, so the `block` rule applies and O plays to the bottom left corner (position 6) completing the first row of the figure.

The second row of the figure starts with the `block` rule causing X to play on the left edge (position 3). O then blocks X in the middle row. At this point `empty` causes X to take the remaining corner. In the last two moves `empty` causes O and X to fill the two remaining open spaces. The game ends in a draw.

The verification performed by this test bench is by no means adequate to verify proper module operation. Many combinations of inputs have not been tried. To thoroughly verify the module a checker is required. This would typically be implemented in a high-level programming language (like "C") and interfaced to the simulator. Proper operation would then be verified by comparing the simulation results to the high-level language model. One hopes that the same mistake would not be made in both models.

Once a checker is in place, we still need to pick the test vectors. After a bit more directed testing (e.g., win, block, near-win, and near-block on all eight lines) we could take two approaches. We could exhaustively test the module. There are $2^{18}$ input cases. Depending on how fast our simulator runs we may have time to try them all. Alternatively, if we don’t have time for exhaustive testing, we could apply random testing, randomly generating input patterns and checking the resulting outputs.

### 9.5 Exercises

- **9–1 Multiple of 5 circuit.** Using an approach similar to the multiple-of-3 circuit of Section 9.1, design a multiple-of-5 circuit that outputs true iff its 8-bit input is a multiple of 5. (Optional, code your design in Verilog and exhaustively verify it with a test bench.)

- **9–2 Calendar circuit.** Recode the `NextDayOfWeek` module so it will work with arbitrary definitions of the constants ‘SUNDAY’, ‘MONDAY’, , , , ‘SATURDAY.’