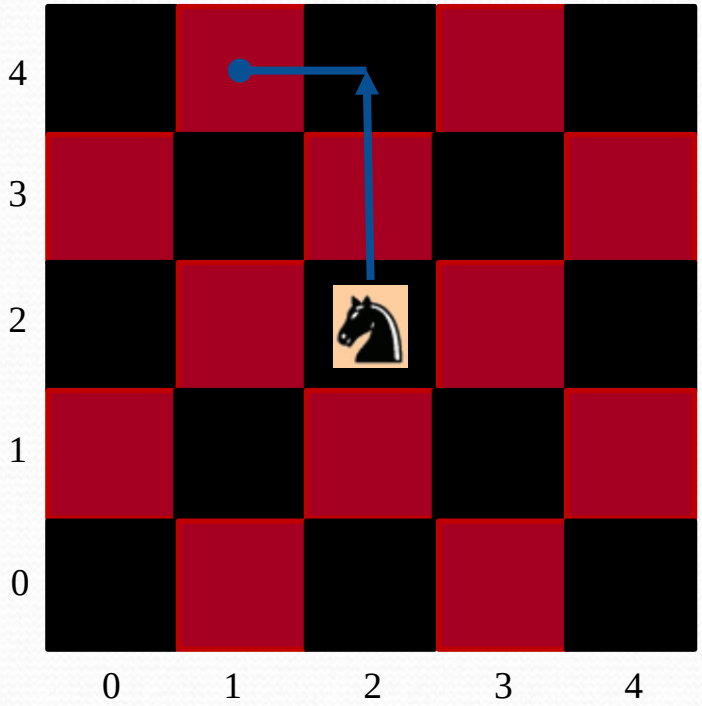
Exercise 23: TourLogic/TourCmd



A knight in chess can move in an "L" shape. Either 2 squares in the Y and 1 in the X as shown, or 1 square in the Y and 2 in the X. From the center square there would be 8 possible moves.

A classic problem for CS students to solve is the "Knight's Tour". From an arbitrary starting position "The Knight" needs to "visit" every square once and only once.

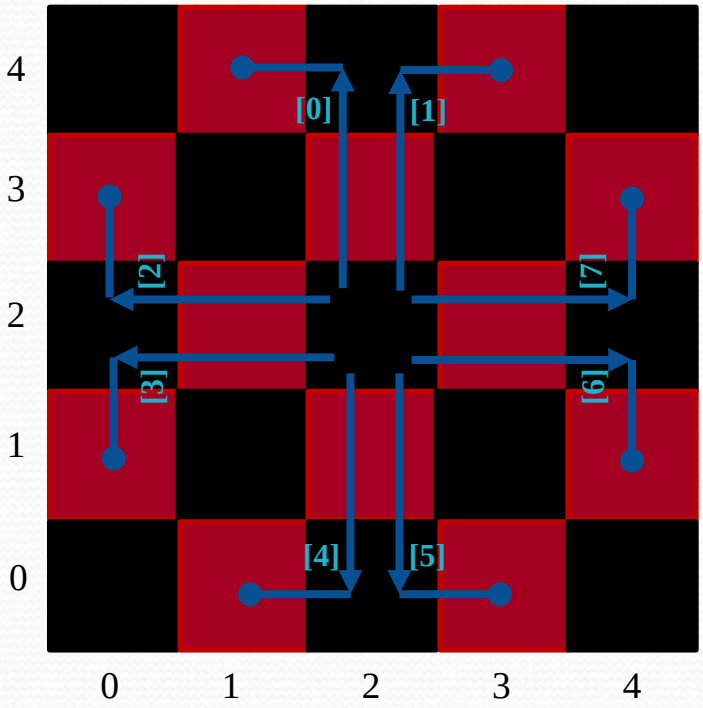
CS students solve this "virtually" *(in memory)*. As EE students we have way cooler skills than CS students so we will solve it in HW and make our little robot physically make the tour. Like a CS student though, we will solve it "virtually" first *(in memory)* and then re-play all the moves of our solution to make "The Knight" physically tour the board.

A real chess board is 8x8, but I did not feel like making that large of a board, so our board will be the smallest board that has a solution (5x5 as shown). The 5x5 board does not have a solution from every starting point. Only the black squares have a solution. Your TourLogic has to work for any black square as a start point.

Exercise 23: TourLogic/TourCmd

- Break your team into 2 teams of 2
  - One will work on solving the Knight's Tour (***TourLogic***)
  - The other will work on the machine to "re-play" the solution (***TourCmd***) (take the moves stored in memory and makes them into movement commands to actually make the robot perform the tour).
  - Both teams need to "agree" on how the moves are encoded in memory. I will discuss that in the next slide.
  - The "solution to the tour" will be an array of 24 moves encoded in memory. For ***TourCmd*** to "re-play" these moves on the physical board how the moves are encoded has to be agreed upon between ***TourLogic*** & ***TourCmd***.

Exercise 23: TourLogic/TourCmd (~~Encoding of moves~~)



The most possible moves available from any square is 8. To encode 8 possible moves ~~you could use 3-bits~~, or you could use an 8-bit wide register and do 1-hot encoding.

Of course, there is the assignment of each encoding to its actual physical move. As shown the encoding [0] corresponds to +2 in the Y and -1 in the X.

This assignment of code to move is somewhat arbitrary, shown is the encoding I used.

1-hot encoding obviously requires more memory to store the move. This could result in larger area. This would make a case for 3-bit encoding.

When your Knight first makes a move to a new square you must compute *(and store)* all possible moves that could be made from that square. That lends itself to 1-hot encoding.

I honestly do not know the most optimal *(area wise)* solution. ~~I think 3-bit encoding might be a win in that regard~~. I did 1-hot encoding because I think it was simpler.

Exercise 23: TourLogic/TourCmd

- Both teams note the labeling of the "coordinates" (X/Y) of the board on the prior slide. This is how you should think of the board layout/coordinate relationship. *(if you think of the old cartesian plane in math…we are in quadrant I)*

- Now that you have agreed on the encoding you can break into 2 groups of 2 and go to your respective sections of this document.

- ***TourLogic*** is covered next.

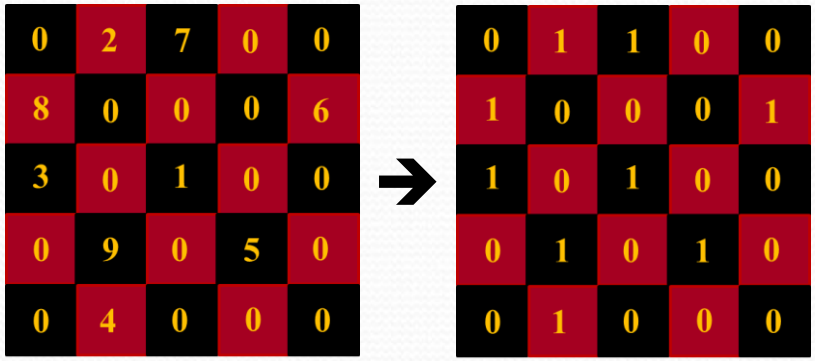
- ***TourCmd*** is later in this document.

Exercise 23: TourLogic *(Interface)*

Signal:	Dir:	Description:
clk,rst_n	in	System clock and asynch active low reset
x_start[2:0], y_start[2:0]	in	Position on 5x5 board that Knight will start tour. Only "black" locations have a solution to the tour.
go	in	A high pulse on this signal initiates your SM to find solution
done	out	Indicate solution found by 1 clock cycle pulse on **done**
indx[4:0]	in	Used to "re-play" solution (once found) to actually make "The Knight" move. There are 24 moves to a solution on a 5x5 board. Apply 5'b00000 to access initial move….apply 5'b10111 to access the last move.
move[7:0]	out	An encoding of the move. Since there are 8 possible moves a knight can make this could be encoded in 3-bits. My solution, however, used 1-hot encoding for the moves, so I used an 8-bit vector with each bit representing one of 8-moves *(only 1-hot though).* If you wish to encode as 3-bits you can.

Constraints: You can solve this anyway you wish, however, your solution must result in synthesizeable logic. There is no bonus given to solving it in fewer clock cycles, so although employing Warnsdorff's rule would be neat, I suggest a simple brute force solution. For a 5x5 board brute force results in a solution pretty quickly.

Exercise 23: TourLogic *(Hints..what memory do you need?)*



One thing you have to keep track of is where "The Knight" has been. For this I suggest starting with a 2D array of 5-bit vectors. You initialize this array with zeros everywhere and a 1 in the starting position. Then every time "The Knight" moves you mark the new spot with a move number. Having the actual move number is nice for debug, but once it is working you can save area by making this a single bit memory.

"The Knight" starts in a black square and has a number of possible first moves. The brute force solution involves picking one *(arbitrarily).* Now from the 2nd position there are a number of possible moves, again one is picked *(arbitrarily).* Eventually your possible choices at each new position diminish because "The Knight" has already visited neighboring squares. Eventually the Knight gets stuck without having completed the tour. One of your former decisions about what move to choose was bad. So you backup one move. Did you have other choices at this square? If so, take one of them instead. If not, you must backup yet another move. So what else do you have to keep track of?

- Your last move from any given move number (or square).
- The possible moves at that square, and which ones you have tried.

Exercise 23: TourLogic *(Hints..what memory do you need?)*

Below is a list of the memory structures (registers) I used:

- 2 dimensional array of 5-bit vectors to keep track of where "The Knight" had visited. This was later changed to a 2D array of bits because really just need to know if knight had been there or not, the 5-bit encoding of the move number was just handy for debug purposes.

- A 1D array *(size 24)* of last moves stored as 8-bit 1-hot vectors. You need to know your last move from any given position (or move number). This could also be done as a 2D array (5x5) based on board position, but a 24 deep 1D array *(based on move number)* turns out to be easier to deal with *(backing up is easier).* This also forms your answer *(re-play list)* once the solution is found.

- A 1D array of possible moves at any given position *(or move number).* Again stored as 8-bit 1-hot encoding of possible moves. Again, could be done as 2D array (5x5) but I found it easier to "backup" by making it a 24 deep 1D array.

- Other non-array registers:
  - Encoding of move to try *(8-bit 1-hot encoding)*
  - 5-bit encoding of what move number we are on. We need to make 24-moves.
  - xx, yy → What is current location of "The Knight"

Exercise 23: TourLogic *(Hints..perl programs)*

- Now try it:
  - Write a python/Java/C# program to solve the tour using the memory structures just outlined.
  - Now try to map that algorithm to states and think of control signals that control the needed memory structures. Remember you need to map this to synthesizeable hardware.
  - Now write verilog and a testbench and debug. This is not a simple problem…give it some thought/effort first on your own before looking at the following provided perl programs.
- Provided Examples: *(old men write in perl)*
  - **KnightsTour.pl** ➔ perl program that solves the tour, but written in a CS way. No intent to map to hardware.
  - **KnightsTourSM.pl** ➔ perl program written in a more hardware centric way that maps the solution to states. Do not mimic this code with your verilog. You should still have clear datapath registers vs control SM in your verilog. This example code has them "mashed" together.

Exercise 23: Testing TourLogic

- Luckily testing your verilog implementation of TourLogic is trivial *(debug is hard…but the testbench is easy)*
  - Simply instantiate TourLogic, apply a **x_start** & **y_start** that correspond to a black square, provide a **clk** and **rst_n** and apply go for 1 clock cycle. Look for **done**. *(It may take 10million clock cycles or more)*
  - For deug purposes you probably want to print out the status of the board with every new move. See the code segment below as an example.

TourLogic_tb.sv

```
always
  #5 clk = ~clk;

//////////////////////////////////////////////////
// Look inside DUT for position to update.  When //
// it does print out state of board.  This is   //
// very helpful in debug.                       //
//////////////////////////////////////////////////
always @(negedge iDUT.update_position) begin: disp
  integer x,y;
  for (y=4; y>=0; y--) begin
      $display("%2d  %2d  %2d  %2d  %2d\n",iDUT.board[0][y],iDUT.board[1][y],
               iDUT.board[2][y],iDUT.board[3][y],iDUT.board[4][y]);
  end
  $display("--------------------\n");
end

endmodule
```

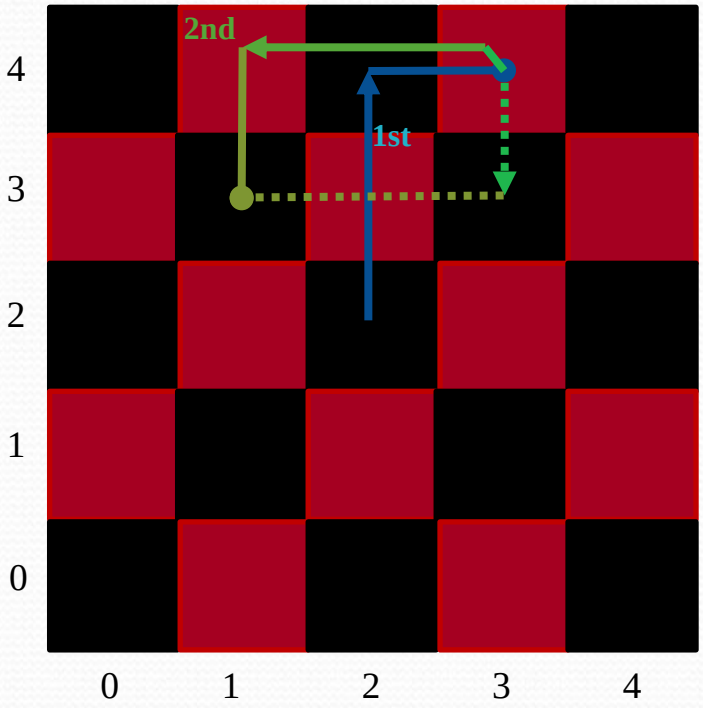
Exercise 23: TourLogic

- **Submit:**
  - TourLogic.sv
  - TourLogic_tb.sv
  - Proof it simulates correct and finds solution
  - Proof it synthesizes

One of many possible solutions
Starting from the center square

19	8	13	2	25
14	3	18	7	12
9	20	1	24	17
4	15	22	11	6
21	10	5	16	23

Exercise 23: TourCmd (No 180's allowed)



Every Knight move decomposes into two move commands. One in the Y, and one in the X.

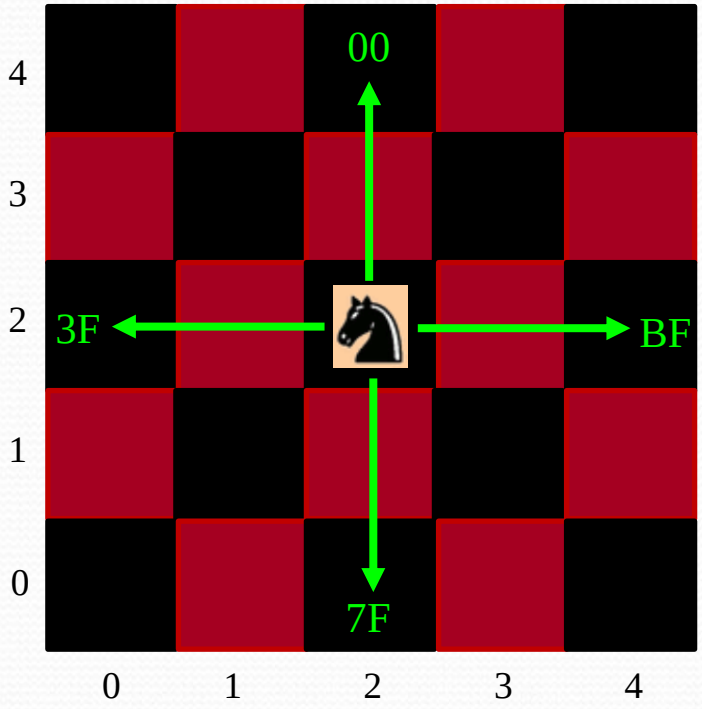
Imagine the first 2 moves you were re-playing were as shown. At the end of the first move the Knight would be facing right. To start the 2nd move it would have to spin 180 ° to face left.

For reasons I don't want to bother explaining *(has to do with gyro accuracy)* We should limit all moves to 90 heading adjustments.

This can be easily accomplished by always decomposing the encoded moves to be in Y first *(or X first…does not matter just be consistent).* The dotted line shown would be the 2nd move if the Y component of the move was always executed first. By always consistently moving in the Y first *(or X first (arbitrary decision))* we ensure all heading changes are limited to 90 °.

Note the labeling of the (X/Y) coordinates of the board.

Exercise 23: TourCmd (Headings)



The move command has an 8-bit field for specifying the heading. This means the robot is capable of following 256 different headings, however, our Knight only makes orthogonal moves. You can think of the heading field of the command as an 8-bit signed number. With a CCW rotation from north being positive, and clockwise rotation from north being negative.

For our purposes there are only 4 heading fields we will use:

North (+Y) is 00
West (-X) is 3F
South (-Y) is 7F
East (+X) is BF

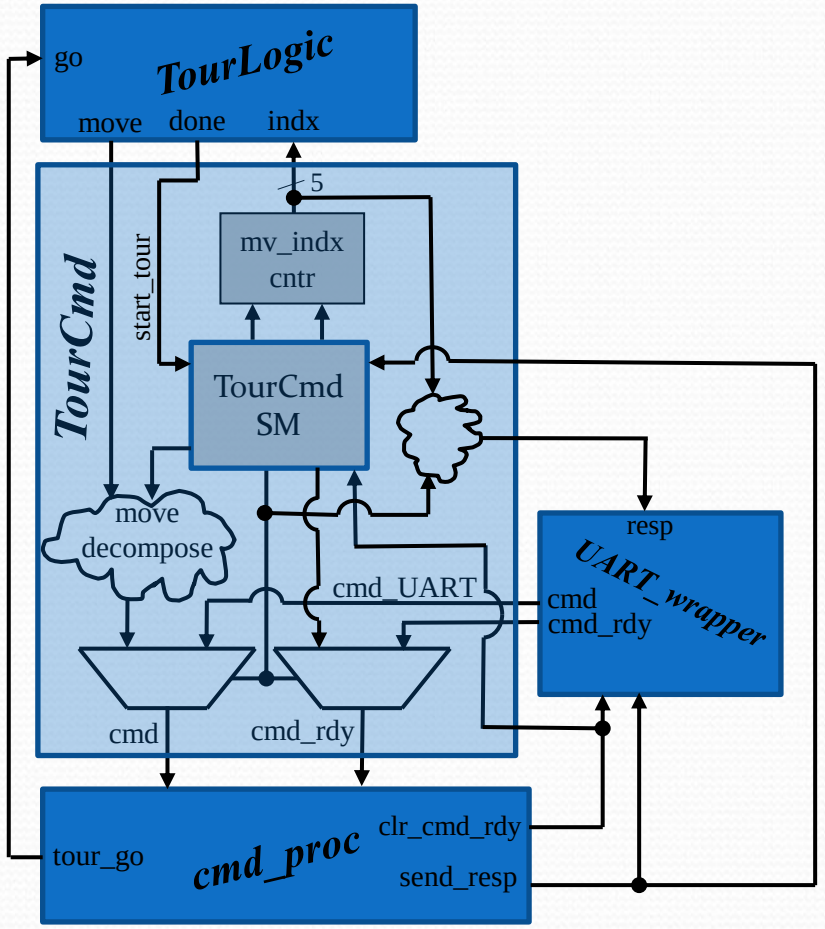
CCW from north are positive headings

CW from north are negative headings

South for us could be represented by 7F or 80, but recall we are "F" extending non-zero heading fields to form a 12-bit heading. So 80 would become 80F. Just stick with 7F to represent South.

Exercise 23: TourCmd (Usurp control of commands)

The ***TourCmd*** block will make "The Knight" physically re-play the solution to the tour by usurping control of the ***cmd_proc*** block. Meaning instead of commands coming from ***UART_wrapper*** *(via Bluetooth)* they will instead come from ***TourCmd***'s decomposition of "the moves" solved by ***TourLogic***.



The commands from ***UART_wrapper*** do not go directly to ***cmd_proc***, but are rather routed through ***TourCmd***. Initially commands are coming from ***UART_wrapper*** *(via Bluetooth).* When the command to perform the tour comes in then **tour_go** is asserted for one clock cycle. This makes ***TourLogic*** compute the necessary moves, and assert **done** *(which kicks off the TourCmd SM)*. Now ***TourCmd*** usurps control of commands, and decomposes each move of the solution into a vertical and horizontal move.

At the end of every move when ***cmd_proc*** asserts **send_resp** a resp of 8'hA5 will be sent if this is the final move of the tour *(mv_indx==5'd23).* Otherwise if it was an intermediate move of the tour a response of 8'h5A is sent.

Exercise 23: TourCmd (Verbal Description of States)

- Stay in IDLE state as until **start_tour** asserted. In the IDLE the mux select should be such that all commands to ***cmd_proc*** come from ***UART_wrapper.***
  - When **start_tour** asserted zero **mv_indx** and transition to state where you are making a segment of the "L" shape move.
- Form a vertical *(or horizontal)* move cmd to send to ***cmd_proc*** that is a function of the move read from ***TourLogic***. Assert **cmd_rdy** to ***cmd_proc*** to get it working on the move.
  - Leave this state when **clr_cmd_rdy** is asserted.
- Keep the cmd to ***cmd_proc*** consistent with the vertical *(or horizontal)* move until ***cmd_proc*** asserts **send_resp**.
  - When **send_resp** comes you know the first move segment is done and you can move on to the horizontal *(or vertical)* move segment.
- Form a horizontal *(or vertical)* move cmd to send to ***cmd_proc*** that is a function of the move read from ***TourLogic***. Assert **cmd_rdy** to ***cmd_proc*** to get it working on the move.
  - Leave this state when **clr_cmd_rdy** is asserted.

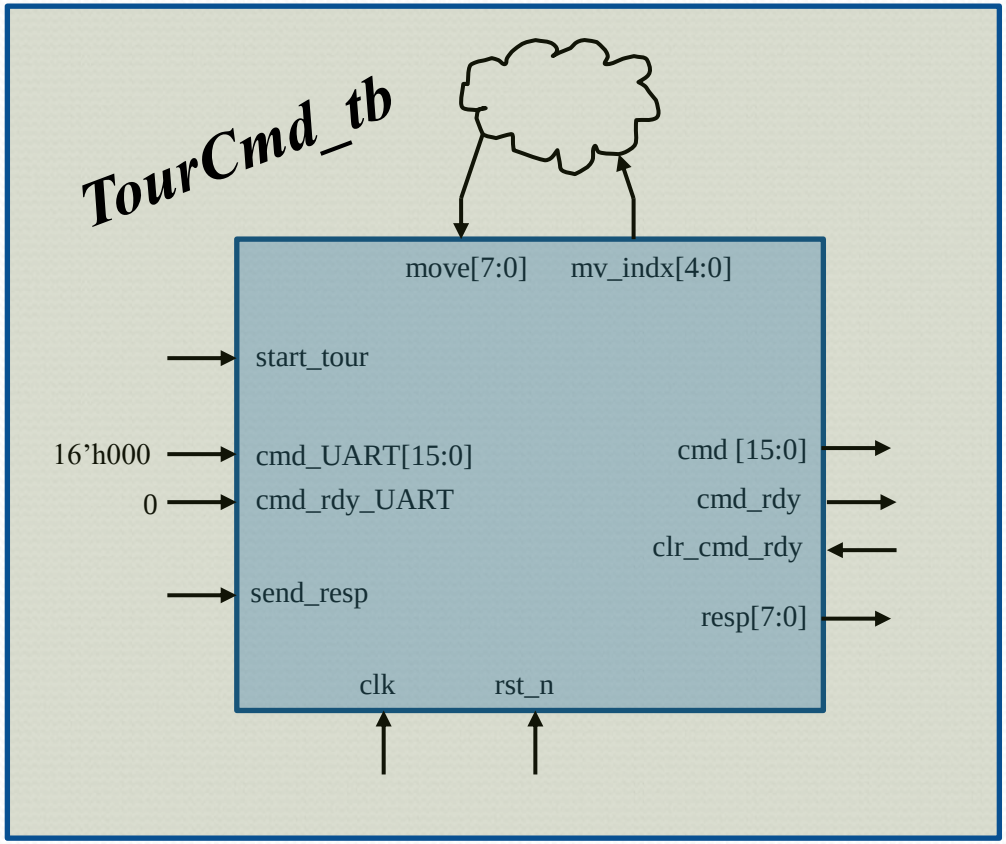
With fanfare

- Keep the cmd to ***cmd_proc*** consistent with the horizontal *(or vertical)* move until ***cmd_proc*** asserts **send_resp**.
  - When **send_resp** comes you are done with the 2nd move segment and either advance **mv_indx** and go to the 2nd state or you are totally done and return to IDLE.

Exercise 23: TourCmd (interface)

Signal:	Dir:	Description:
clk, rst_n	in	
start_tour	in	Comes from ***TourLogic***'s done signal, kicks off SM
mv_indx[4:0]	out	Used to access move of solution from ***TourLogic***'s memory
move[7:0]	in	Encoded move, this might be [2:0] if you did not use 1-hot encoding
cmd_UART[15:0]	in	"Normal" command path input from UART_wrapper
cmd_rdy_UART	in	cmd_rdy from UART_wrapper.
cmd[15:0]	out	Command to ***cmd_proc***. Will be usurped by ***CmdTour*** SM.
cmd_rdy	out	cmd_rdy to ***cmd_proc***. Will be from ***UART_wrapper*** or SM.
clr_cmd_rdy	in	From ***cmd_proc***. Used to knock down cmd_rdy from SM *(also routed to **UART_wrapper**)*
send_resp	in	Lets us know ***cmd_proc*** is done processing latest command. If cmd was from ***UART_wrapper***, or was last command of the tour then we send a response of 8'hA5, otherwise we send 8'h5A if intermediate move of the tour.
resp[7:0]	out	Response to send to host, either 0xA5 or 0x5A *(see above)*.

Exercise 23: TourCmd (simple testbench)



Your first testbench can be a simple one to wring out the basic operation of the SM. Apply **clk**, **rst_n**, and then hit it with a pulse on **start_tour**. It should present the first **cmd** *(vertical move cmd perhaps)* of the encoded move present on **move**. Then assert **clr_cmd_rdy** at the testbench level. Then after a few clocks assert **send_resp** so it goes on to the 2nd command *(horizontal move?)* of the encoded move. You can continue on testing for a couple of moves. Make the **move** a simple function of **mv_indx** *(see cloud in diagram)*

Test it as thoroughly as you can. This block will be tested more completely when you move to "full chip" testing where a model of the Knight & the Board are provided.

Turn In: **TourCmd.sv**, **TourCmd_tb.sv**, and proof you ran the testbench.