Implementation of a Markov Decision Process

Christian Wagner, Electrical Engineering and Physics

Neil Mehta, Electrical Engineering and Mathematics

EE 354

Fall 2016

**Abstract:**

A Markov Decision Process is a mathematical model which provides a strategy for decision making in scenarios where outcomes are partially dependent on the actions of a decision maker and partially random. In other words, it is an algorithm employed to develop a “policy” (a policy here is defined as a set of optimal decisions for each position in a cost function) for a wide range of optimization problems. Our design seeks to develop a state machine that implements the algorithm for a Markov Decision Process, then displays the results (the policy on a grid of locations) utilizing VGA. In order to calculate the policy, it is necessary to do floating point addition and multiplication and later to decode these values back in to decimal, so to overcome these challenges, we developed modules to handle these functions as well.

**Introduction and Background:**

For a finite number of locations on a grid, there exists positive reward locations, negative reward locations, walls and open locations. Solving the policy entails finding the best direction of movement in order to reach the positive reward location from any other open location. The negative reward location should be avoided at all costs and walls have no policy defined on them since they are not open locations. In order to make a decision, two other factors come in to play: noise and discount. Noise is the associated with the probability of not moving in the direction which is chosen and discount is the effect to reward a particular movement has (less efficient paths to the reward state provide less reward). The policy is defined for each location on the grid as the direction of movement from that location which provides maximal reward. The number of iterations performed over locations a certain Manhattan distance away determines how quickly the utility (or expected reward by making a certain movement from a particular location) converges to a steady value. In order to implement the algorithm, floating point precision is needed. Furthermore, we employed a couple modules from previous assignments and project. We utilized the debouncer from a previous lab as well as the hvsync\_generator provided to us for the VGA implementation. For brevity’s sake, the debouncer and hvsync\_generator will not be discussed since these were left relatively untouched.

**Design:**

*Modules[[1]](#footnote-1):*

32 bit Floating Point Double Multiplier\*

16 bit Floating Point Double Multiplier\*

32 bit Floating Point Single Multiplier\*

16 bit Floating Point Single Multiplier

32 bit Floating Point Double Adder\*

16 bit Floating Point Double Adder

Floating Point to Decimal Decoder

Markov Decision Process State Machine

User Interface

Debouncer

H-sync/V-sync Generator

VGA module

*16 bit Floating Point Single Multiplier:*

The precision needed for our multiplication is handled sufficiently by the IEEE 754-2008 16 bit half-precision floating point standard. In this formation, the most significant bit is sign bit, the next five bits are the exponent, and the remaining 10 bits are the mantissa (fractional portion of a number in scientific notation). Our state machine takes in two half values, unpacks them in to their component blocks of bits, then multiplies the mantissa and normalizes the exponent of the product. Finally we shift the mantissa until we have a leading one and then repack all of the appropriate values in to a 16 bit output register and send an acknowledge signal to let the MDP state machine know the value it is receiving is valid. The one special case we needed to consider was if one or both of the numbers is zero then we skip to the done state and just output all zeroes.

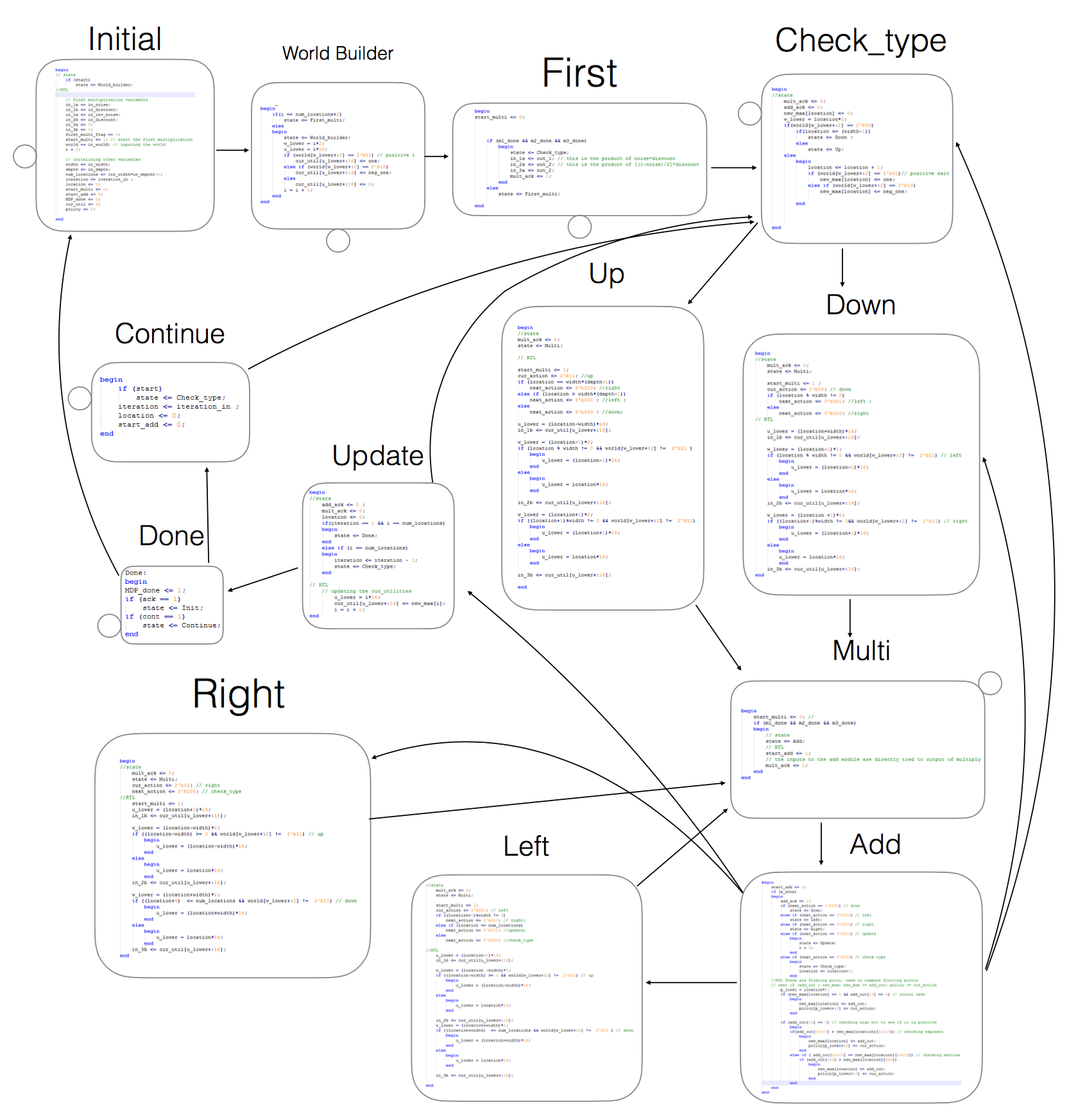
*16 bit Floating Point Double Adder:*

Similar to the multiplier, we take in three half values and unpack them in to their components when a start signal goes high. We set flags if any of them are zero, and then find the half value with the largest magnitude. We shift the mantissa such that the exponents are the same for all three values and then add in two states so that we can change the sign of the sum if necessary (since the sum of the two smaller values might be larger in magnitude than the that of the largest value). Then we normalize the exponent of the sum and shift the mantissa to get a leading one and repack the value and set an acknowledge signal high. The special case we considered here was if all the values are zero then we skip to the done state and output zero. If any one or two of the values are zero, we use the flags to make sure we don’t fall in to an infinite loop of shifting the mantissa when we normalize the exponents of those values (since the exponent would be zero, it would keep shifting).

*Floating Point to Decimal Decoder:*

Because of the difficulty of converting floating point back in to raw decimal, we opted to take our floating point numbers and compare them to multiples of 0.05 that are less than 1 (only the reward states have reward 1 or greater so those can be tied to the type of state). Thus our VGA outputs values that are round multiples of 0.05.

*Markov Decision Process State Machine:*

Our MDP state machine starts in an initial state where it creates a 3x4 grid and assigns default values for noise, discount, and number of iterations which it receives from the User Interface. From there it moves to a “World Builder” state which assigns the initial utilities and rewards to the different locations based on the type of location. Then we move to “First Multi” which performs the first set of multiplication of noise and discount. Next we move to “Check\_type” where we validate that the current location is open and if it is, we either go to “Up” or “Down” depending on where we are on the grid currently (if we’re in the first row, we can’t go up and if we’re in the bottom row, we can’t go down.). In these two states we begin to calculate utilities of moving in the up or down directions from our current location and then set the next action or direction of movement (either right, left, or down). From “Left,” ”Right,” ”Up,” and ”Down,” we move to “Multi,” where we wait for the multiplication to finish and then go to “Add” where we actually check the current action’s utility against all previously calculated utilities and update the policy if the current is larger than the old utility. Then we move to either the next direction we defined before or if all directions have been checked, we move to the “Update” state where we actually update the utilities that appear on the grid if the number of iterations is great than one and then go back to “Check\_type” to repeat the whole process to get more accurate utilities. If the number of iterations is one then we go to the “Done” state where we send an acknowledge signal which controls one of the LEDs on the board. We also can go to a “Continue” which leads to “Check\_type” upon receiving a start signal. The purpose of this state is so the user can enter allow the MDP to continue running (for a specified number of iterations) after it has already solved the policy.

*User Interface:*

Upon reset, our User Interface generates width = 4 columns, depth = 3 rows, and iterations = 10 by default. This is the “world” that gets fed to the MDP. The initial state checks for the user to press the up, down, left, or right buttons. If the left button is pressed, the next state is MDP where the MDP state machine is run. If the up button is pressed, we move to a state that allows the user to assign the number of iterations using the switches on the board. Pressing up again returns the interface to the state machine. If the down button is pressed, we move to a state where the user can change the number of columns of the grid (so long as it is even and less than or equal to 8) again by using the switches and pressing down to assign the value, this then enters a similar state where the user can change the number of rows. Pressing down a last time returns the interface to the initial state. If the right button is pressed we go to a state where the user can change the selected location using switches. From here the user can move back to the initial state by pressing down or move to the “Type” state by pressing right again. In the type state, the user is allowed to change the type of location of the currently selected position on the grid. The user can select between open location, positive reward, negative reward, or wall and these changes are updated in the array that is passed to the “World Builder” state of the MDP state machine.

*VGA:*

We start by initializing registers to hold the x and y positions on the 640x480 screen for ever possible location where we may want to place a figure (this includes anywhere we would like to place the numbers representing utilities, the dots representing the direction of the policy or walls if the type of the position on the grid is a wall). These values are hardcoded in. We also create two registers (x and y position) for a point that will move as we draw the values on the screen called draw\_point. We do the same for each line of the grid and specify the CounterX and CounterY ranges to display the grid. In a non-clocked always block, we have a for loop that runs through all of the locations and moves the draw\_point according to which of the 10 possible points in a position of the grid we would currently like to draw. The ten possible points represent the possible sign symbol, the three digits for our utility, the decimal point, the four possible positions of the policy and the upper left hand corner in case we would like to place a wall at that position of the grid. At each location, we perform the necessary logic based on arrays received from the MDP state machine to determine what kind of character we would like to put there. We have 12 bit registers for each type of character possible (0-9, sign, the four policy locations, and walls). We also made wires containing all of these characters in terms of draw\_point i.e. they are defined as relative distances away from the draw\_point. These are the actual shapes displayed on the screen. When we would like to assign a character we simply move the draw\_point to the appropriate location, determine what character we would like to place and assign that wire defined in terms of the current\_draw point in to the appropriate 12 bit register at the appropriate index (position of the grid). When we are finished, we place these 12 bit registers on the appropriate R, G, B wires to produce the colors we would like. We are using all possible RGB wires to produce a brighter, cleaner looking display. Finally we update the vga\_r, vga\_b, vga\_g registers in a clocked always block at 125 MHz. The VGA module also acts as our top module, so it instantiates the Debouncers for the buttons and instantiates the User Interface and feeds the latter with the debounced up, down, left, right buttons. The display is updated when the MDP completes its calculations, thus we needed an acknowledge signal from the User Interface to tell the VGA module when to display, so this was included in the port list of the User Interface instantiation inside the VGA module. Furthermore, the reset and switchers are also fed in to this port list so the User Interface may react appropriately to user input. On SSD3 we placed the current state of the User Interface, on SSD2 we placed the type of the current position of the grid, on SSD0 we placed the current value of the switches.

**Test Methodology:**

We began testing for each individual component as they were completed. For all modules except the VGA we created test benches with some input values. For the VGA, we initially put eights at every location on the screen just to check their locations. Later we moved to creating a fake User Interface and fake MDP state machine with some hardcoded values in order to test the VGA module because had we tested the VGA module with the actual UI and MDP, synthesis time would have been excessive. After the VGA module was working with the fake UI and MDP, we then implemented the real state machines with the VGA and made sure that it was working exactly the same way. In order to test the output values from the decoder, it was necessary to write a C++ script to convert 16 bit floating point in to decimal as there were not online converters and ModelSIM does not recognize this format.

**Conclusion and Future Work:**

The Markov Decision Process problem is an interesting one as it represents an example of artificial intelligence and learning. In our implementation, we achieved the core of the design through implementation of our state machine. However, one aspect which we attempted but did not succeed in was making the VGA module more robust to user input. Because of the draw back of having to hardcode in x and y positions one which to draw figures, our method would not necessarily work for all of the functionality which we built in to the actual MDP state machine such as the capability to handle grids of different size. This issue arose because of our strategy of statically drawing figures on the screen before they are displayed. An alternative which we attempted was scanning the screen as the values and their positions change on the screen (as would the grid). Although values and grid appeared in the proper places on the screen, the scanning frequency was not cooperative with the hvsync signal and the result was either nice solid figures and grid lines that were scanned slowly across the screen such that it was obvious to the viewer or such dim values that they were barely recognizable as the figures we attempted to draw. If we had been able to achieve the scanning mechanism with VGA, our implementation would be totally robust to any user changes allowed by the User Interface.

1. Modules demarcated with “\*” were not included in the final design. [↑](#footnote-ref-1)