Pong Pong Revolution:
Project Proposal

Daniel Lopuch and Zachary Remscrim

Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology
Cambridge, MA 02139

Pong Pong Revolution

Pong Pong Revolution (PPR) will be an enhanced Pong game controlled by the actual motions of the player. A video camera will look at the player running back and forth in front of a wall (or other neutral-colored surface), and the video input will be used to move the player's pong paddle. The game will be displayed onto an XVGA monitor. The 6.111 Labkit has all of the electronic components needed to implement PPR. The PPR developers request the usage of an SVideo-enable video camera for the PPR controller.

Block Diagram

Figure 1 below is a rough block diagram of all of the major modules and components of PPR.

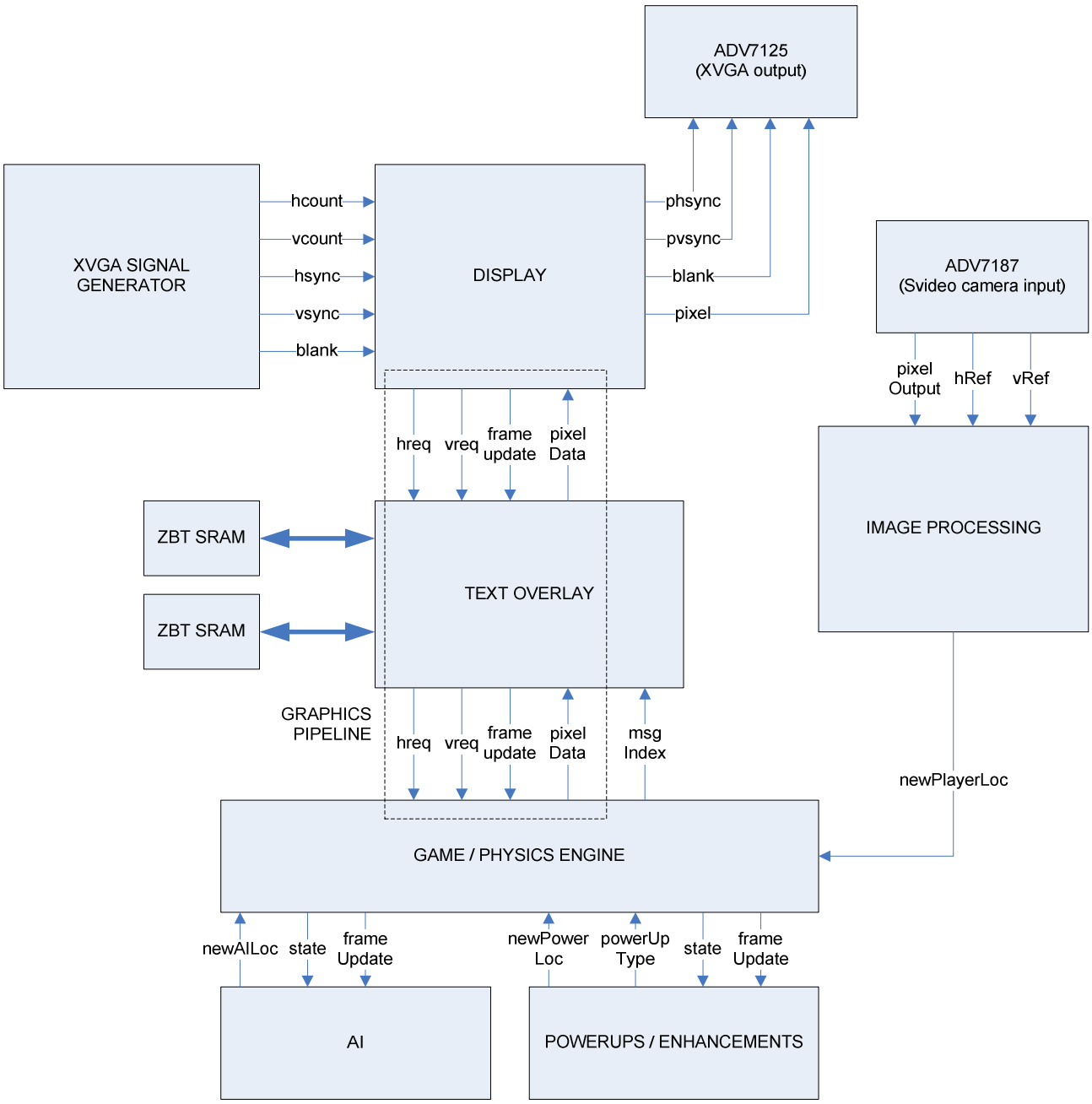

Figure 1: PPR Block Diagram

Module: Display
Authorship: Daniel Lopuch
Purpose: Interface between the lab5 XVGA module, the AD7125 chip, and the graphics pipeline.
Inputs:

- **hcount:** x-coordinate of hsync/vsync
- **vcount:** y-coordinate of hsync/vsync
- **hsync:** generated horizontal sync
- **vsync:** generated vertical sync
- **blank:** blank signal of hsync/vsync
- **pixel_in[2:0]:** pixel color from graphics pipeline (see Description for format)

Outputs:

- **phsync:** pipelined hsync output to AD7125 (**hsync** delayed be length of graphics pipeline)
- **pvsync:** pipelined vsync output to AD7125 (**vsync** delayed by length of graphics pipeline)
- **pblank:** pipelined blanking output to AD7125 (**blank** delayed by length of graphics pipeline)
- **pixel[2:0]:** pixel output from graphics pipeline
- **h_req:** x-coordinate request to graphics pipeline
- **v_req:** y-coordinate request to graphics pipeline
- **update_frame:** frame update signal to graphics pipeline; high for single 65mhz clock cycle on a new frame; logically, **vsync** && **~oldvsync**

Description:

Workings

The **display** module acts as the bridge between **xvga_generator** (which generates the appropriate xvga signals), the graphics pipeline, and the AD7125. It simply feeds the **hcount** and **vcount** inputs into the graphics pipeline and delays **hsync, vsync,** and **blank** by the length of the graphics pipeline so that the **pixel_in** input that comes out of the graphics pipeline is synchronized with the appropriate **hsync** and **vsync**. **hsync, vsync,** and **blank** are delayed into **phsync, pvsync,** and **pblank**, respectively.

Pixels

Pixels will be 3-bit colors, making pixel busses 3-bits wide. Bit[2] will represent Red. Bit[1] will represent Green. Bit[0] will represent Blue. White, black, yellow, purple, and teal are represented by mixing the appropriate color bits (ie yellow is represented by b110).

Testing:

Testing this module is relatively simple. In fact, most of it was already accomplished in the Lab 5 excercises. Some analysis and testing should be done to determine the length of the graphics pipeline. The length depends on the Text Overlay and Game Engine modules. Neither of the modules are currently expected to have any delay, but delay should be kept in mind as a source of potential problems.

Module: Text Overlay
Authorship: Daniel Lopuch
Purpose: Graphics pipeline module which overlays inspirational messages during gameplay.
Inputs:

- **xIn:** current x-coordinate input
- **yIn:** current y-coordinate input
- **frameUpdateIn:** frame update input
- **pixelIn[2:0]:** pixel input from later in the pipeline.
- **msgIndex[???]:** input from game engine that states which message to display; a 0 corresponds to no message overlay

Outputs:

- **xOut:** current x-coordinate pass-through to later in the pipeline
- **yOut:** current y-coordinate pass-through to later in the pipeline
- **pixelOut:** current pixel information fed back up the pipeline
- **frameUpdateOutput:** frame update pass-through to later in the pipeline

Description:

How It Fits In The Pipeline

xIn and **yIn** are always passed through to **xOut** and **yOut**, but this module also looks at **xIn** and **yIn**. If this module wants to draw some text at the current x and y, it asserts its own pixel information onto **pixelOut**. If it doesn't have anything to draw at the current x and y, it passes **pixelIn** through to **pixelOut**.

How It Draws Messages: A First Pass

The text overlay will have a character buffer with 16 "character slots" in the middle of the playing field, as shown in Figure 2.

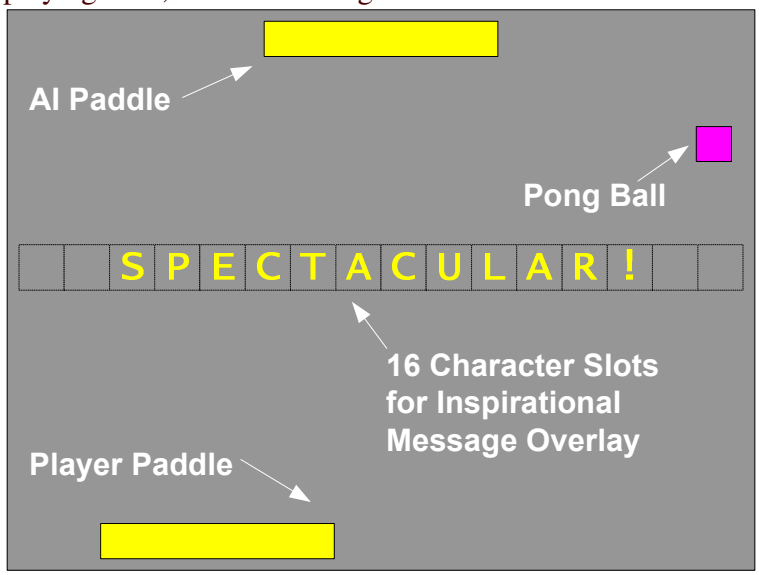

Figure 2: Character Slots will be positioned in the middle of the PPR Playing Field.

Each "character slot" is a small 25-bit RAM that stores a bitmap of the character currently programmed into it. Whenever **frameUpdateIn** is asserted (ie on a new frame), *TextOverlay* looks at what is asserted on **msgIndex**. It looks at the **msgIndex**-th position in a Message Lookup Table. This Message Lookup Table contains a list of 16 6-bit characters to be loaded into each of the character slots. For each of these 6-bit characters, the module looks up the

particular letter in a second Font Lookup Table. Each entry in the font table contains a 25-bit bitmap of the particular letter. The process of generating a message into each of the character slots is summarized in Figure 3 below:

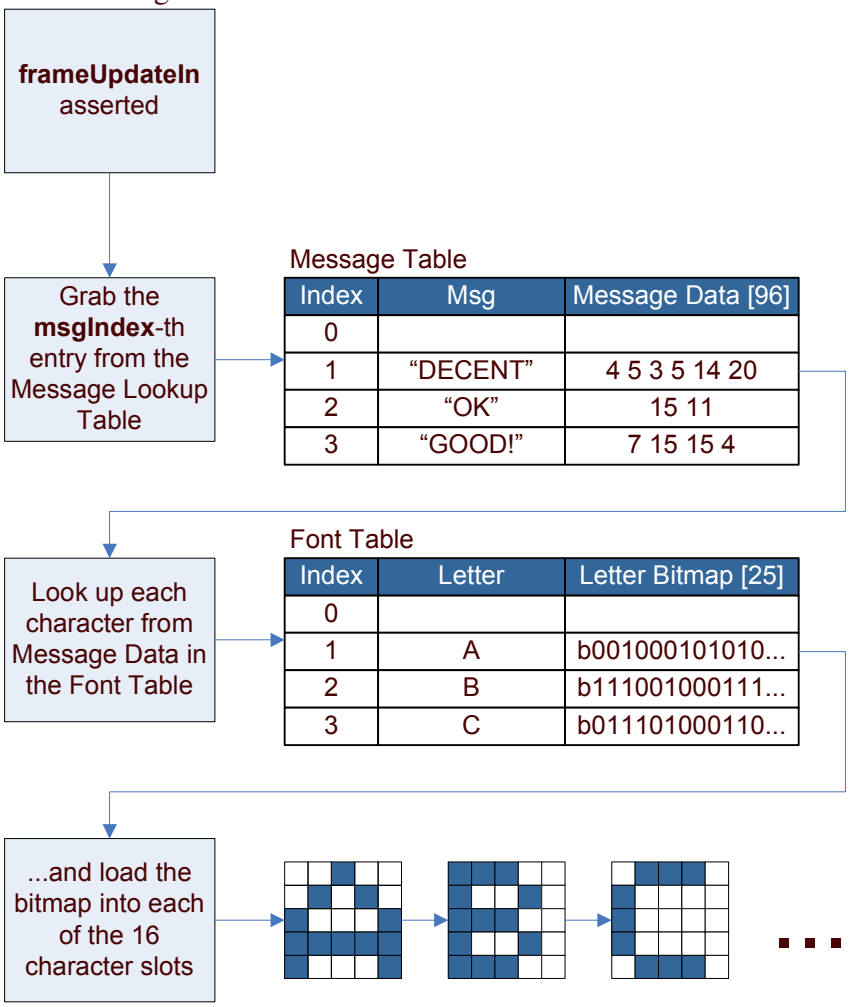

Figure 3: Generating a Message Bitmap

Memory and Timing Considerations

The 16 25-bit character buffers are planned to be implemented using the FPGA's onboard BRAM. The Message Table and the Font Table are planned to be implemented using the 6.111 Labkit's ZBT SRAM.

The Font Table, with each bitmap being only 25 bits wide, can be easily implemented with one letter per address slot. Each character index is 6-bits long (ie enough for A-Z and some special characters). The Font Table, therefore, needs 32 entries.

Each entry in the Message Table is effectively 96-bits wide (6-bits per character times 16 characters). Therefore, each message requires 2.5 entries in the 32bit-wide ZBT SRAM. The number of messages is still at this point undecided.

At first glance, it would seem timing may be a problem due to the delay in reading from the ZBT SRAM. However, we can take advantage of the fact that the messages are always displayed in the center of the screen – there is a significant time gap between a frame update

(when the new message is generated) and when the text overlay pixel information needs to be available in the graphics pipeline. For each text message, we need to make three reads to get the message and then another 16 reads to load the character bitmaps into each of the character slots. Since the ZBT SRAM can easily function at the 65mhz XVGA clock, we should be able to have the text bitmaps ready before the graphics pipeline even finishes drawing the first line of the screen. Put another way, although the message bitmap won't be ready immediately on a frame update, by the time the graphics pipeline begins requesting pixels that could be part of the text overlay, the text bitmaps will have been loaded with what feels like an eternity ago to a 65mhz clock.

Testing:

With the XVGA framework in place from Lab 5, the display overhead for testing this module is already done. The most testing time will be spent interfacing with the ZBT SRAM and developing (and troubleshooting) the correct timing.

Module: Image Processing
Authorship: Daniel Lopuch
Purpose: To read in the camera's SVideo input, figure out where on the camera's x-axis the player is, and to translate his position in the video image into a position for his pong paddle.
Inputs:

- **vidPixelIn[20:0]**: YCrCb pixel input bus from ADV7185 chip
- **vRef**: Vertical reference input from ADV7185
- **hRef:** Horizontal reference input from ADV7185

Outputs:

- **newPlayerLoc:** current x-position of player

Description:

This module will do simple image processing of a video camera image to determine where a player is and translate his position on the video image to a position for his paddle. The ADV7185 video decoder chip will output the pixel color information for each pixel in the video stream. The module will determine whether or not the pixel's color matches a particular criterion. If it does, the module will add the pixel's x-coordinate to a running average. When the ADV7185 has finished outputting one frame, the module will have calculated a centroid on the x-axis that (hopefully) corresponds to approximately the position of the player. The centroid will be outputted to the Game Engine module, which will use it to adjust the location of the player's paddle.

Testing:

As there have been no labs which have used the ADV7185 video decoder chip, the first priority of testing will be to determine how the chip works and to get some working video footage. The best way of doing this will be to simply read in the video image and retransmit it onto an XVGA display. This exercise should teach me how to read in the video signal and develop some way of synchronizing between the 27mhz SVideo signal and the 65mhz XVGA signal.

After this is done, some tests will be programmed to develop the best criteria for recognizing a player. If there is time, perhaps a more advanced image recognition algorithm can be implemented.

Module: Game/Physics Engine
Authorship: Zachary Remscrim
Purpose: This module is the central module of the device. Its purpose is to keep track of the current state of the game, update the state, and provide appropriate display data to the Display Module.
Inputs: The inputs to this module are **Clk, Reset, Update, NewPlayerLoc, NewAILoc, NewPowerLoc, Hcount, Vcount,** and **PowerUpType**.

- **Clk** is the system clock, it is a 1 bit value.
- **Reset** is the global reset, it is also a 1 bit value and is active high.
- **Update** is the 1 bit signal produced by the Display Module that indicates that it has moved to the next frame of video. In this game, time is kept using this signal; the game updates its state whenever Update is asserted high. This signal remains high for only one clock cycle after being asserted.
- **NewPlayerLoc** is the new location of the player's paddle; the paddle will be moved to this location on the next assertion of Update, if a collision does not occur. The manner in which collisions are dealt with is discussed below in the "Description" section. This signal consists of a pair of values, an x and y coordinate. Each of these values will be N bit signals, where N is the minimum number of bits needed to encode the largest pixel value on the screen. For example, for a 1024x768 display, the x and y coordinates would each be 10 bit values. This signal is generated by the Image Processing Module.
- **NewAILoc** and **NewPowerLoc** are identical to NewPlayerLoc with the exception of the fact that they represent the new locations of the AI's paddle and of a powerup, respectively; they are produced by the AI Module and the PowerUp Module, respectively.
- **Hcount** and **Vcount** are signals produced by the Display Module to indicate which pixel it is currently requesting information about. Hcount corresponds to the horizontal position of the desired pixel, while Vcount corresponds to the vertical position of the desired pixel. As was the case for NewPlayerLoc, these signals are each N bit signals, where N is the minimum number of bits needed to encode the largest pixel value on the screen.
- **PowerUpType** is a signal produced by the PowerUp module to specify the type of the powerup. For example, a given value of PowerUpType may correspond to a powerup that enlarges a player's paddle.

Outputs: The outputs of this module are **Color, State,** and **MessageNumber**.

- **Color** specifies the color that should be displayed for the pixel whose position corresponds to the Hcount and Vcount inputs. If a game object (such as a paddle or the ball) occupies the location specified by Hcount and Vcount, then Color will be equal to the color of that game object. If no game object occupies that location, then Color will be set to the color of the background. Color is a COLOR_DEPTH bit number, where COLOR_DEPTH is a parameter that specifies the number of bits used to encode the color. For example, if COLOR_DEPTH is 3, then this is a 3 bit number. This output is sent to the Display Module.
- **State** specifies the current state of the game. It consists of the current positions of the player's paddle, the AI's paddle, the ball, and any powerups; state also includes the current heading of the ball and the powerup. This signal requires a number of bits given by (number of game objects)*(number of bits needed to encode vertical dimension of screen + number of bits needed to encode horizontal dimension of screen)+(number of bits needed to encode heading of ball and powerup). This output is used by the AI Module and the PowerUp Module.
- The **MessageNumber** output specifies which message should be displayed by the Text Overlay Module. The message that should be displayed at any time is a function of the player's current performance.

Description:

- For each frame, this module will need to compute the value of MessageNumber. This will be a relatively simple computation that uses the player's current score, the number of times they have successfully hit the ball, etc., to determine which message should be displayed.
- After receiving the new positions of all the game objects, from the AI Module, PowerUp Module, etc., this module must resolve any collision. A collision occurs whenever one game object attempts to occupy the same space as another. The results of this collision will depend on which objects are involved. For example, if a ball collides with a paddle, it will bounce off the paddle. This module will also make sure that no object goes off screen or passes through another object. The specific operations involved will be simple additions and comparisons. In order to facilitate this, sub-modules such as Blob or Intersects may be used. Blob is used to represent a rectangular game object, given the x and y coordinates of its top left corner. Intersects is used to determine if two blobs intersect one another.
- The module will require very little in terms of internal memory. The current positions of various game objects, as well as the values of certain variables needed to determine the value of MessageNumber, need to be stored
- This module will need to output a value for Color each time that Hcount or Vcount changes. Once per frame, it will need to output a value for State as well as a value for Message Number.

Testing:

- In order to fully test this module, multiple stages of testing must be performed. Firstly, it is essential that the module be subjected to a representative set of input values in a simulation. This can be accomplished through the use of a Verilog Testbench which provides inputs similar to the expected inputs. Specifically, a standard incrementing pattern of Hcount and Vcount, a periodic update signal, as well as an assortment of inputs from AI, PowerUp, and Image Processing Modules. Once the module has been debugged and is able to work properly in simulation, it will then be implemented on the FPGA. After implementation, it will be tested and debugged further.

Module: PowerUp
Authorship: Zachary Remscrim
Purpose: The purpose of this module is to implement power ups, which are game objects that give the player that picks them up bonuses, such as a larger paddle. Each instance of this module represents a single power up.
Inputs: The inputs to this module are **Clk, Reset, State,** and **Update.**

- **Clk** is the system clock, it is a 1 bit value.
- **Reset** is the global reset, it is also a 1 bit value and is active high.
- **State** contains the current state of the game (locations of the paddles, the ball, current powerups, etc.). This module will use the information to determine the new location of an existing powerup, or, if the powerup was just collected by a player, determine the location and type of a new powerup.
- **Update** is the 1 bit signal produced by the Display Module that indicates that it has moved to the next frame of video. In this game, time is kept using this signal; the game updates its state whenever Update is asserted high.

Outputs: The outputs of this module are **NewPowerLoc** and **PowerUpType**.

- **NewPowerLoc** is the new location of a power up. Assuming that the power up has not been claimed by a player, the new location will be determined by taking the existing location and adding a small adjustment to it in the direction of the powerup's current heading. If the powerup

was claimed by the player, a new powerup will be generated and its location will be used for NewPowerLoc. This signal consists of an x and y value, specifying the location.

- **PowerUpType** is a number which represents the type of a given powerup. Types of powerups include: enlarge paddle, slow down ball, etc.

Description:

- Each frame, the module will need to compute the value of NewPowerLoc. This is a relatively simple computation. If the powerup was not claimed by a player, the new position can be determined by the addition of a small value in the direction of the powerups current heading to its current position. If the power up was just claimed, the location of a new powerup will need to be calculated. This will be done through the use of simple arithmetic operations performed on segments of the State input.
- PowerUpType will need to be computed whenever a new power up is generated, which happens during only a small fraction of frames. It's value will need to be stored and outputted until a new value is computed.
- Very little internal memory will be needed since this module need only store the current location and heading of the powerup as well as certain parameters which govern the generation of new powerups.

Testing:

- Due to the relatively simple nature of this module, only a moderate amount of testing is required. Much as was the case for the Game Module, this module should first be tested in simulation. It should be subjected to a varying set of states, to assure that it will provide the proper output for that state. After this has been accomplished, the module will be implemented on the FPGA, and further testing will be done.

Module: AI

Authorship: Zachary Remscrim

Inputs: The inputs to this module are **Clk, Reset, State,** and **Update.**

- **Clk** is the system clock, it is a 1 bit value.
- **Reset** is the global reset, it is also a 1 bit value and is active high.
- **State** is a signal produced by the Game module that specifies the current state of the game (i.e. position of game objects). This module will use this information to determine where to move the AI's paddle.
- **Update** is the 1 bit signal produced by the Display Module that indicates that it has moved to the next frame of video. In this game, time is kept using this signal; the game updates its state whenever Update is asserted high.

Output: The output of this module is **NewAILoc**.

- **NewAILoc** is the position that the AI's paddle will be moved to on the next update.

Description:

- In order to determine the best location to move its paddle, the AI will need to consider several factors. Specifically, the current location of the ball, as well as the current locations of powerups, affect the AI's decision. The AI will attempt to move its paddle in such a way that it will get as many powerups as possible, while still leaving sufficient time to move into position to hit the ball. In order to make this determination, many calculations will need to be performed. The actual number of calculations will be determined by the desired level of accuracy; there will, of course, be an upper limit due to the fact that the AI will need to return a new position once per frame. An example of a calculation that would need to be performed is determining the location of another game object after a certain period of time, and determining if the AI paddle can reach that object.

- This module will need to have a throughput of one new position per frame. The module will be implemented in such a way that it will attempt to refine its choice of new position until time runs out, but will always save an intermediate calculation so that it will have a new position available to be outputted at any time.
- The module will need a small to moderate internal memory to store intermediate calculations as well as various parameters that govern its behavior.

Testing:

- As was the case for both the Game and PowerUp modules, testing will begin with a Verilog Testbench. Here, it should be verified that the module gives reasonable output when given a variety of states. Furthermore, it should be verified that the module gives a reasonable result if interrupted by an assertion of Update during its calculation. After this phase of testing, its implementation on the FPGA will be tested by examining its behavior when competing against a human player.