Pac-Man Design Document

Anastasiia Merkudanova (am6754) Austin Gnecco (asg2290)
Connor Marvin (cm4662) Shishir Sharma (sks2266)

4/17/2026

1 Introduction

In this project, we recreate Pac-Man, the 1980's arcade game, using the DE1-SoC board. The objective of the game is for the player to use a keyboard to control Pac-Man (the central character) and move him to consume all of the dots without being eaten by the ghosts, which move randomly around the screen attempting to catch him.

This project will utilize both code written in `C` and hardware implemented on the FPGA fabric written in Verilog. While the game logic is done on the microprocessor, the graphics are rendered on the FPGA and output to an external monitor.

2 System Block Diagram

As previously stated, all game logic is run on the microprocessor in `C`, which allows for ease of programming and interfacing with external peripherals. An input is received from the keyboard and is sent to the microprocessors over `USB` where it is decoded. Data for sprites and tiles is then passed through memory addresses to the hardware where it is rendered out to a `VGA` display.

3 Algorithms

3.1 Game Logic

The core of our game will be the logic controlling the score counting, movement of characters, and collision detection. The score will be incremented for each pellet that Pac-Man collides with. Pac-Man and the ghosts will all move at set rates, with a new position calculated 60 times every second. Pac-Man will be prevented from leaving the maze. The ghosts' behavior will be to chase Pac-Man while in their attack mode. When Pac-Man consumes the cherries, the ghosts will become vulnerable to Pac-Man and their behavior will become to run away. If the ghosts collide with Pac-Man in their vulnerable state, they will

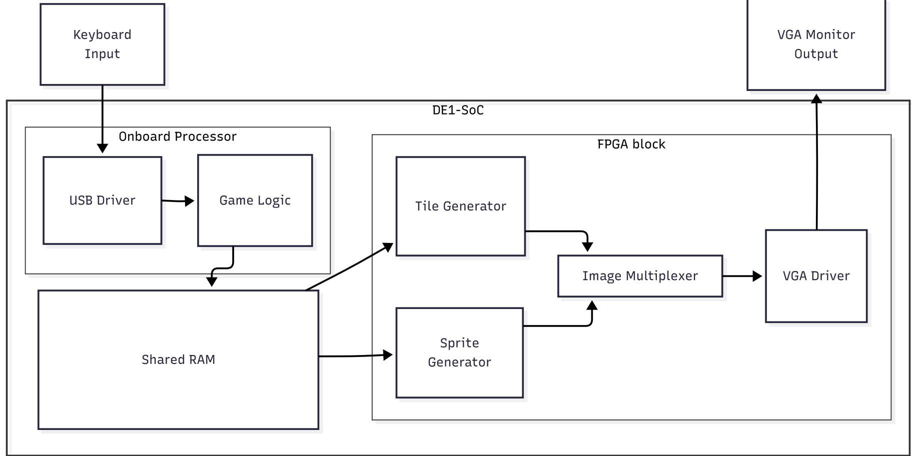

Figure 1: Pac-Man Block Diagram

respawn in the center of the map. Pac-Man will have three lives and one will be taken away for each collision with a non-vulnerable ghost. After the final death `GAME OVER` will be displayed, along with the score. The game ends either after the final death or when all of the pellets have been eaten.

3.1.1 Pac-Man

Pac-Man is the player-controlled entity, in this game. His behavior is driven entirely by user input, and the game updates his position accordingly. Outside of that, Pac-Man's animation, pellet collection, and collision checks are procedural once his location and movement direction are known.

3.1.2 Ghosts

The ghosts are not player controlled, and their behavior is pre-defined. Each ghost operates independently and is reactive to Pac-Man's current position, changes in direction, whether a power pellet has been eaten, and whether they have been captured or released.

3.1.3 Position Storage

A practical way to represent the maze is as a matrix or grid. Each location in the maze corresponds to a cell in that matrix, and the value stored in the cell tells the game what exists there. For example, one value can represent a wall, another an empty path, another a pellet, and another Pac-Man's current location. That makes movement and collision logic straightforward, because every algorithm only needs to check positions relative to that grid. Instead of doing complicated geometric calculations, the software just asks whether the

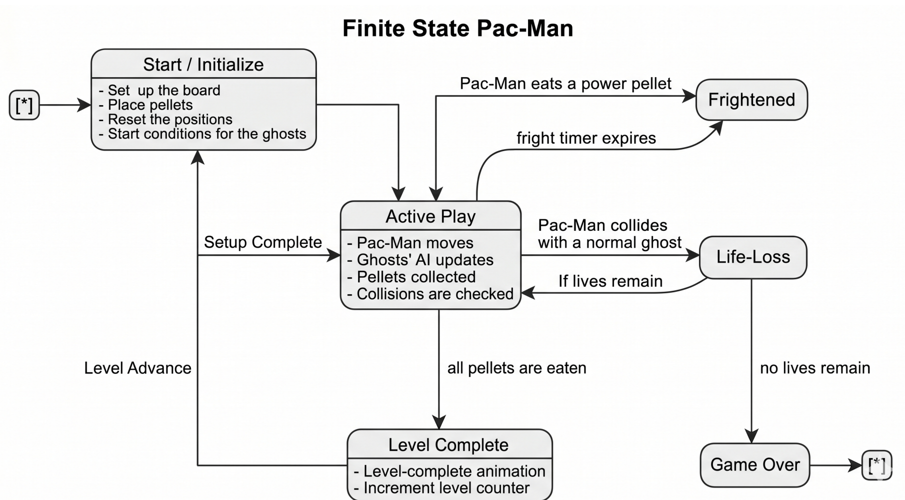

Figure 2: Pac-Man State Machine

next tile is open, whether it contains a pellet, or whether a ghost occupies it. Each ghost can use Pac-Man's current matrix position as an input and then apply its own targeting rule to decide where to move next. The same matrix can also encode the walls and blocked spaces, which prevents invalid movement and keeps all path decisions tied to the actual maze layout. Pellets are already naturally represented in the matrix structure of the maze. Each tile can indicate whether a pellet is present, and when Pac Man moves into that tile, the pellet is removed and the score increases.

Handling multiple entities on the same board introduces overlap issues. Since ghosts can occupy the same tile, the display system cannot rely on a single matrix value to represent everything. Instead, the base maze can remain one matrix while dynamic entities like Pac Man and ghosts are tracked separately. When rendering, a priority system determines what is drawn on top. For example Pac Man can take highest priority, followed by ghosts, then pellets and background. This avoids conflicts while keeping the logic clean.

Movement between tiles requires more precision than just grid positions. Ghosts and Pac Man do not jump instantly from one tile to another but instead move smoothly across pixels. This means each entity needs both a tile position and a sub tile offset. By tracking how many pixels have moved toward the next tile, the system can interpolate positions and determine when a transition is complete. This also allows partial sprite rendering, where part of a sprite appears in one tile and the rest in the next. Rather than storing every possible split sprite, it is more efficient to store full sprites and render only the visible portions based on the offset. This keeps memory usage low while still allowing smooth motion.

A finite state machine describing the game behavior is shown in Figure 2.

3.2 Graphics Rendering

The graphics will be rendered using a tile and sprite system as is laid out in the provided *VGA Tile Graphics on an FPGA* tutorial. The maze and score will be displayed using tiles, while Pac-Man and the ghosts will be displayed using sprites. The pellet tiles will be replaced with blank tiles as Pac-Man consumes them, and the score tile will be updated accordingly. All other tile assets will remain static during the game. The sprites and the tiles are both rendered at 32×32 resolution.

At the start of each frame, the graphics hardware reads the data from each sprite, which consists of its x and y coordinates and its requested image. Then it reads from the tiles, which are listed in order in memory (so coordinates are not needed) and the requested image is generated in that location. Sprites are always shown on top of the tiles, and there is not any need to be able to render the tiles on top of the sprites. The tiles and sprites can both be rotated; the first two bits of each image address are the rotation bits, allowing for a horizontal flip and vertical flip respectively.

3.3 Sound Generation

The audio system will be implemented using a memory-mapped streaming architecture managed by the HPS running Linux. The game's sound effects—including the movement loops, ghost sirens, and pellet consumption—are stored as 32-bit PCM .wav files on the SD card. During gameplay, these files are read into a software buffer where they undergo digital summation, allowing for the simultaneous playback of multiple concurrent tracks. This mixed data is then transferred to the Audio IP core via the Avalon Slave Port. The core manages the stream through four independent FIFOs, each holding up to 128 32-bit words, ensuring the left and right channels remain perfectly synchronized. To bridge the parallel processing of the HPS with the serial requirements of the physical board, the core automatically serializes the data for the onboard Wolfson CODEC. Proper operation is maintained by auxiliary hardware: the Audio Clock core provides the required sampling frequencies, while the Audio and Video Config core initializes the CODEC registers via an I^2C interface

4 Resource Budget

The table below shows the on-chip memory usage for the `FPGA` graphics. The maze borders consist of 4 bend pieces and two straight pieces, for a total of 6 images. The dots and power pellets that Pac-Man eats each require 2 images. Each of the four ghosts characters have 4 images during normal operation (one for each direction of travel). When the ghosts are vulnerable to getting eaten they have a different set of 4 images for a grand total of 32. Finally, the blue ghosts and their animations will require a total of 4 images. Pac-Man itself is simple, requiring only 4 images to model the movement of the mouth, with the image being able to be rotated to change direction. Displaying the "HIGH

SCORE" caption will require some of the 26 characters, and numbers 0-9 are needed to represent the score.

Table 1: On-chip Memory Usage for FPGA Graphics

Name	Type	Image	Size (bits)	# of Frames	Total size
Pac-Man	Sprite		16 × 16	4	8192
Ghosts	Sprite		16 × 16	4	32,768
Map Pieces	Tile		32 × 32	6	49,152
Dots	Tile		32 × 32	2	16,384
Characters	Tile		32 × 32	36	294,912

5 Hardware/Software Interface

The hardware and software interface through memory mapped I/O, where the software will write to and read from a series of addresses that are mapped to physical ports in the hardware. Conceptually, the FPGA component puts the pattern/sprite related data inside the shared RAM. The exposed interface is a byte-addressable RAM, where the addresses are allocated for different functions - either information for the pattern generator table or the sprite attribute table. The software is able to change the layout of tiles and position of sprites on the screen by simply updating the byte stored at certain addresses inside the byte-addressable RAM. The software driver could hide the intricacies by providing user-space programs with a simple graphics API. To change the position of a certain graphics element from the software programmer's perspective, one could simply pass the address and the x and y coordinates of the sprites. By not encoding storing the position of each tile but rather reading them in sequential order out of memory, a large amount of data is saved, as 11 bits would be needed to number each tile.

Table 2: Communication Parameters

Communication	X coords	Y coords	Image Data	Rotation
Sprite Data	10 bits	9 bits	3 bits	2 bits
Tile Data	N/A	N/A	6 bits	2 bits

The layout of the shared memory between the CPU and the FPGA is shown

in Table 3 below. This is not a space efficient memory structure as most of the time only 8 of the 16 bits on each line are used. However, the DE1-SoC board has significantly more memory than this project will use and using this configuration allows us to write the 9 and 10 bit Y and X positions of the sprites in a single 16 bit write, rather than having to divide up the data between rows.

Table 3: Sprite and Tile Address Map

Address	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0x00							Px9	Px8	Px7	Px6	Px5	Px4	Px3	Px2	Px1	Px0
0x01								Py8	Py7	Py6	Py5	Py4	Py3	Py2	Py1	Py0
0x02											Pi5	Pi4	Pi3	Pi2	Pi1	Py0
0x03							G1x9	G1x8	G1x7	G1x6	G1x5	G1x4	G1x3	G1x2	G1x1	G1x0
0x04								G1y8	G1y7	G1y6	G1y5	G1y4	G1y3	G1y2	G1y1	G1y0
0x05											G1i5	G1i4	G1i3	G1i2	G1i1	G1y0
0x06							G2x9	G2x8	G2x7	G2x6	G2x5	G2x4	G2x3	G2x2	G2x1	G2x0
0x07								G2y8	G2y7	G2y6	G2y5	G2y4	G2y3	G2y2	G2y1	G2y0
0x08											G2i5	G2i4	G2i3	G2i2	G2i1	G2y0
0x09							G3x9	G3x8	G3x7	G3x6	G3x5	G3x4	G3x3	G3x2	G3x1	G3x0
0x0a								G3y8	G3y7	G3y6	G3y5	G3y4	G3y3	G3y2	G3y1	G3y0
0x0b											G3i5	G3i4	G3i3	G3i2	G3i1	G3y0
0x0c							G4x9	G4x8	G4x7	G4x6	G4x5	G4x4	G4x3	G4x2	G4x1	G4x0
0x0d								G4y8	G4y7	G4y6	G4y5	G4y4	G4y3	G4y2	G4y1	G4y0
0x0e											G4i5	G4i4	G4i3	G4i2	G4i1	G4y0
0x0f									t0i7	t0i6	t0i5	t0i4	t0i3	t0i2	t0i1	t0i0
0x10									t1i7	t1i6	t1i5	t1i4	t1i3	t1i2	t1i1	t1i0
0x11									t2i7	t2i6	t2i5	t2i4	t2i3	t2i2	t2i1	t2i0
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
0x19d									t1997i7	t1997i6	t1997i5	t1997i4	t1997i3	t1997i2	t1997i1	t1997i0
0x19e									1198i7	1198i6	1198i5	t1198i4	t1198i3	t1198i2	t1198i1	t1198i0
0x19f									t1199i7	t1199i6	t1199i5	t1199i4	t1199i3	t1199i2	t1199i1	t1199i0
0x1a0																ready

In this table, the first portion of the sprite memory addresses signifies the character: P is Pac-Man, G1 is Ghost 1, G2 is Ghost 2, etc. The lower case letter then denotes what the data is: x for x position, y for y position, i for image, r for rotation and t for tile. Finally, the last number tells what bit number the data is.

Finally, control needs to be synchronized between the FPGA and the SoC. Since the frames will be updated 60 times per second, we have decided to put the FPGA in charge of the timing. When the program starts, the FPGA raises the READY bit in the last row of the memory (0x1a0). The SoC sees this bit and begins computing the sprite positions for the next frame. Since the software calculations are done very quickly, the table is nearly immediately updated. After 1/60th of a second, the graphics hardware reads the sprite and tile positions, outputs the graphics over VGA to the display, and then raises the ready flag to prepare the data for the next frame.