EE 371 – Lab 6 Final Project Report
Pac-Man Style VGA Game on DE1-SoC

Student: Yashas Gogia
Student ID: *2366016*
Lab Number: Lab 6 – Final Project (VGA / Game)

---

1 Introduction

For this lab I implemented a simplified Pac-Man style game on the DE1-SoC board using SystemVerilog, VGA output, a Nintendo-style N8 controller, and on-chip memories. The final system:

- Draws a maze on a 290×410 VGA region using a pre-generated `tiles.mif`.
- Represents walkable paths using a separate node grid (`node_map_rom`) at a higher resolution than the tiles.
- Moves the player character in a Pac-Man style (always moving in one direction, with "sticky" direction changes when the player presses a new direction).
- Stores coins for each walkable node in a dual-port RAM, draws them on the VGA, and clears them as the player passes through.
- Counts collected coins and displays the score on the seven-segment HEX displays.
- Spawns an alien/ghost that moves around the maze using its own node-based controller. If the alien overlaps the player, it is game over (logical event).

The project heavily uses FSMs, careful handling of ROM/RAM latency, and a clean separation between control (FSMs, neighbour readers) and datapath (positions, addresses, RAM).

---

2 Design Procedure

2.1 High-Level System Overview

At the highest level, the design has these major blocks:

1. `video_driver`
   - Generates the VGA timing signals (x, y, HS, VS, etc.) for a 290×410 active area.
   - Receives final pixel color `r,g,b` and drives the VGA DAC pins.
2. `vga_background`

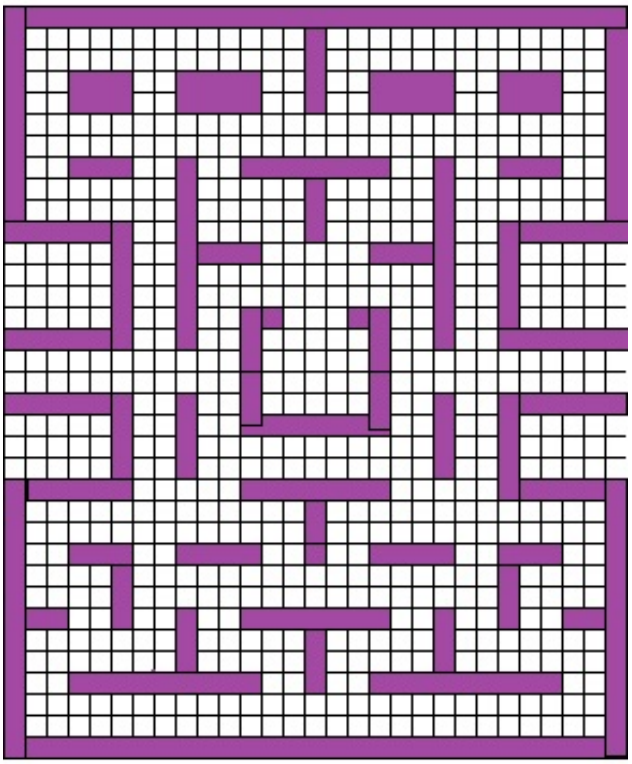


Figure 1: map

- Takes the current pixel coordinate (x, y) and looks up the corresponding tile color from `tiles.mif`.
- Outputs `bg_r, bg_g, bg_b` – this forms the static maze background.

3. `n8_driver` + `edge_detect` **modules**
   - Sample the N8 game controller using latch/pulse signals and a serial data line.
   - Produce stable button signals (up, down, left, right, A, B, etc.).
   - `edge_detect` converts these into one-cycle pulses for direction changes.

4. `player_ctrl` **(Pac-Man logic)**
   - Maintains the player's node position (`player_x`, `player_y`) on a 30×36 grid.
   - Talks to `node_neighbour_reader` to learn which directions are walkable around the current node.
   - Implements an always-moving, "sticky" Pac-Man movement scheme and a tunnel wrap at $(0, 17) \leftrightarrow (29, 17)$.
   - Writes to the coin RAM when the player overlaps a coin (sets that node's coin bit to 0).

5. `node_neighbour_reader`
   - Given the current node (x, y), sequentially reads UP, DOWN, LEFT, RIGHT neighbours from `node_map_rom`.
   - Handles ROM latency with an FSM (ADDR $\rightarrow$ READ $\rightarrow$ LATCH per direction).
   - Outputs `up_walkable`, `down_walkable`, `left_walkable`, `right_walkable` and a ready flag.

6. `node_map_rom`
   - A 1080×1-bit ROM (36 rows $\times$ 30 columns).
   - 1 means walkable node, 0 means wall.
   - This ROM is the "true" logical maze used by both the player and the alien.

7. `coin_map_ram`
   - A dual-port 1080×1-bit RAM.
   - Port A: write port used by the player to clear coins.
   - Port B: read port used by the VGA side to display coins.
   - Initially, every walkable node has a 1 (coin present).

8. **Alien (ghost) controller**
   - Similar structure to `player_ctrl`, with its own (`alien_x`, `alien_y`), its own `node_neighbour_reader`, and a movement policy that depends on the player's position.
   - If alien node equals player node, it triggers a "game over" condition (logical).

9. **Coin counter / score display**
   - A simple counter that increments once per coin collected.
   - Score is shown on HEX displays (only 0–9 per digit).

10. **Top-level** `DE1_SoC` **module**
    - Connects everything: VGA, controller, player, alien, coin RAM, score, and HEX displays.
    - Handles global reset.

Figure 1 – Top-Level Block Diagram (Description)

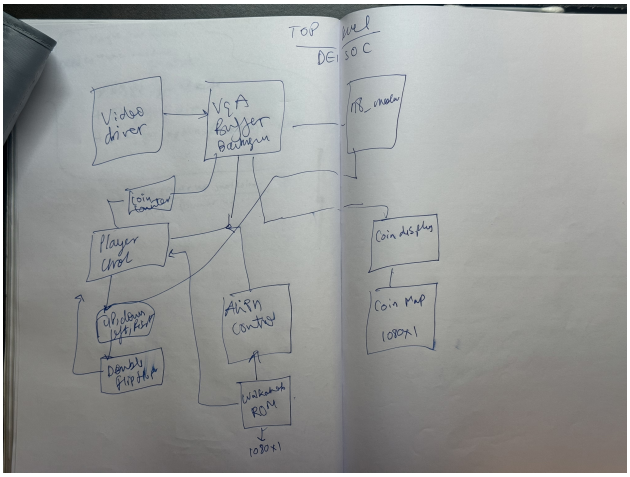


Figure 2: Top-Level Block Diagram

- `CLOCK_50` and reset feeding:
- `video_driver` (outputs x, y and VGA signals).
- `vga_background` (inputs x, y; outputs `bg_r, bg_g, bg_b`).
- `n8_driver` and `edge_detect` → `player_ctrl`.
- `player_ctrl` and `alien_ctrl` → their own `node_neighbour_reader` (each has its own instance and its own `node_map_rom` logically).
- `player_ctrl` → `coin_map_ram` (write port).
- `coin_map_ram` + x, y → coin overlay logic.
- Player and coin overlay → final scene overlay → `video_driver`.

2.2 VGA Subsystem and Coordinate System

The VGA core is mostly provided:

- Resolution: `WIDTH = 290`, `HEIGHT = 410`.
- Outputs: x in [0..289], y in [0..409].

I chose the maze to occupy:

- Horizontal: `MAP_X0 = 0`, width = `NODES_X` × `TILE_SIZE` = 30 × 10 = 300 pixels.

But since the visible region is 290 pixels wide (0..289), the rightmost part of the map is cropped; in practice the map and tiles were built to fit that visible width.

- Vertical: `MAP_Y0 = 30`, height = `NODES_Y` × `TILE_SIZE` = 36 × 10 = 360 pixels.

This leaves 30 pixels on top and 20 pixels on the bottom for UI / padding (the bottom gets cut at 410).

Background color for any (x, y) is fetched by:

1. Mapping (x, y) to a tile index (`tile_x`, `tile_y`),
2. `index = tile_y * 29 + tile_x` (because tiles image is 29 × 35),
3. Reading from `tiles.mif` using a ROM IP.

The important point is: tiles are for drawing, nodes are for logic. They have different dimensions

and MIFs, and I kept them separate to avoid the earlier bug where I assumed they were aligned.

2.3 Node Grid and Map ROM

The logical movement grid is:

- Columns: 0..29 (30 total),
- Rows: 0..35 (36 total),
- So total nodes = $30 \times 36 = 1080$.

Addressing:

$$\texttt{addr} = \texttt{node\_y} \times 30 + \texttt{node\_x}$$

where `node_y`: 0..35, `node_x`: 0..29.

`node_map_rom` is a 1-bit ROM with that layout:

- 1 = walkable intersection.
- 0 = wall.

This representation matches the way the player and alien controllers think: they move from node to node (intersection to intersection), not tile to tile.

Earlier issues happened when I tried to treat nodes as tiles and assume that visual walls lined up with logic. Separating the two fixed the consistency.

2.4 Player Controller (`player_ctrl`) – Always-Moving Pac-Man

Inputs and Outputs

Inputs:

- `clk`, `reset`,
- Direction pulses: `up_btn`, `down_btn`, `left_btn`, `right_btn`.

Outputs:

- `player_x`, `player_y` (6 bits each: 0..29, 0..35),
- Coin write control: `coin_wr_addr`, `coin_we`, `coin_din`.

Internally, `player_ctrl` has:

1. **Direction state (enum):**

```
typedef enum logic [2:0] {
    DIR_NONE  = 3'd0,
    DIR_UP    = 3'd1,
    DIR_DOWN  = 3'd2,
    DIR_LEFT  = 3'd3,
    DIR_RIGHT = 3'd4
} dir_t;
```

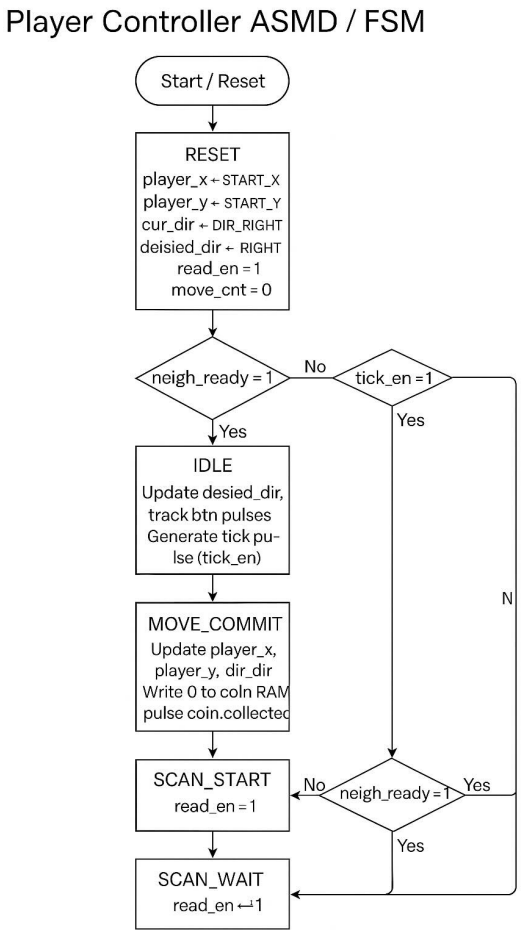


Figure 3: Player ctrl

cur_dir is the current actual movement direction, and desired_dir is the last requested direction from the controller.

2. **Slow movement divider:**

The game uses a 50 MHz clock, but moving every 20 ns would be way too fast. I added a 24-bit counter move_cnt and generated a pulse tick_en every N cycles. For example, MOVE_DIV = 5_000_000 gives about 10 moves per second:

```
if (move_cnt == MOVE_DIV - 1) begin
    move_cnt <= 0;
    tick_en  <= 1;
end else begin
    move_cnt <= move_cnt + 1;
    tick_en  <= 0;
end
```

3. **Neighbour flags from** node_neighbour_reader**:**

- up_walkable, down_walkable, left_walkable, right_walkable.
- neigh_ready indicates that these flags match the current (player_x, player_y).

4. **Tunnel behavior:**

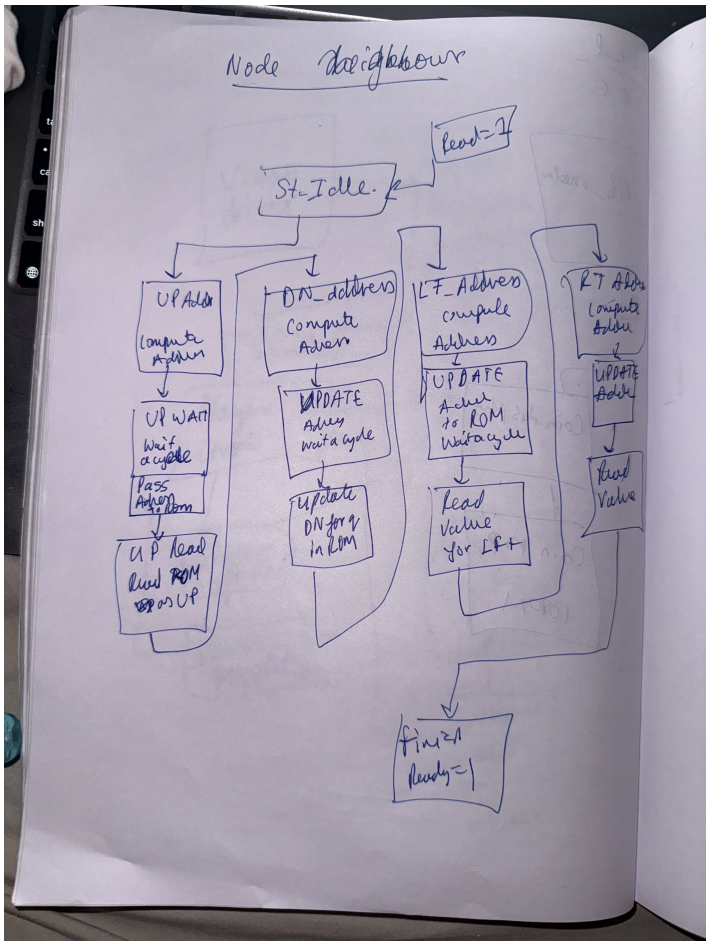
- If the player is at (0, 17) and tries to go left (desired or current direction), they teleport to (29, 17).
- If at (29, 17) and trying to go right, teleport to (0, 17).
- This warp is independent of ROM walkability (just like Pac-Man's side tunnel).

Movement Policy

On each cycle, combinational logic decides the next position. If neigh_ready and tick_en are both high:

1. Check for tunnel warp:
   - If at (0, 17) and desired_dir == DIR_LEFT or cur_dir == DIR_LEFT, move to (29, 17) and keep direction DIR_LEFT.
   - Similarly for the right side.
2. Else:
   - First, try desired_dir. If the corresponding walkable flag is 1 and we are in bounds, set candidate_dir to that.
   - If desired_dir is blocked (candidate_dir == DIR_NONE), try to keep moving in cur_dir if that direction is walkable.
   - If we found a candidate_dir, update next_x, next_y accordingly, update cur_dir_next, and set move_req = 1 so we know we actually moved.

Separately, desired_dir is updated whenever a button pulse arrives (even if we cannot move yet), so that the player can queue up a direction change, which feels like real Pac-Man.



Coin Clearing Logic

When the player steps onto a node that has a coin, we want to clear that coin. The logic is:

- At the same time we commit (`player_x`, `player_y`) to `next_x`, `next_y`, we also compute:

```
coin_wr_addr = player_y * 30 + player_x;
coin_we      = (coin_was_present_here) ? 1 : 0;
coin_din     = 1'b0;
```

In the final design, I used a handshake so that:

- `coin_was_present_here` is derived from a comparison between the coin RAM's output at that address and the player's node.
- Once it is 1, the RAM write port sets that address's bit to 0 on the next cycle.

To avoid flicker or partial coins, I tied coin clearing to the one time a node is entered. If the player stays on the same node, we do not repeatedly assert `coin_we` for no reason.

2.5 Neighbour Reader (`node_neighbour_reader`)

This module is critical because of the ROM latency. The IP for `node_map_rom` has a registered output, which means:

- You set address on cycle N,
- You see the corresponding `q` on cycle N+1.

If you do not handle that explicitly, you end up with:

- Reading old data from the previous address,
- Storing the wrong flags (e.g., `up_walkable` becomes the value of some unrelated address),
- Movement bugs like sliding through walls or getting stuck on valid paths.

Interface

Inputs:

- `clk`, `reset`,
- `read_en` (1-cycle pulse from `player_ctrl` when we want to rescan neighbours),
- `node_x`, `node_y` (current node).

Outputs:

- `up_walkable`, `down_walkable`, `left_walkable`, `right_walkable`,
- `ready = 1` when we are idle and the four flags correspond to the current node.

The ROM instance is inside this module:

```
node_map_rom u_node_rom (
    .address (rom_addr),
    .clock   (clk),
    .q       (rom_q)
);
```

FSM Structure

I used an FSM with "address" and "read" states for each direction.

Conceptually:

- **S_IDLE**: `ready = 1`. Wait for `read_en`.
- For each direction UP, DOWN, LEFT, RIGHT:
  - ***_ADDR**: compute and set `rom_addr` for that neighbour.
  - ***_READ**: wait one cycle so that `rom_q` becomes valid.
  - *: latch `rom_q` into the corresponding walkable flag.

Example for UP:

1. **S_UP_ADDR**:
   - If `node_y > 0`, `rom_addr = (node_y - 1) * 30 + node_x`; else `rom_addr = base_addr` (treated as wall).
   - Next state: **S_UP_READ**.

2. **S_UP_READ**:
   - Next state: **S_UP** (ROM output is now valid).
3. **S_UP** (sequential block):
   - If `node_y > 0`, `up_walkable <= rom_q`; else `up_walkable <= 0`.
   - Next state: **S_DN_ADDR**.

This pattern is repeated for DOWN, LEFT, RIGHT, then we return to **S_IDLE** and assert `ready = 1`.

The key fix from earlier broken versions was to give the ROM one full clock cycle per address before sampling `rom_q`. Adding these intermediate states (`*_READ`) removed the timing bug where we were reading the previous address's data.

2.6 Coin RAM and Coin Display

Coin RAM Layout

`coin_map_ram` is also 1080×1-bit with the same layout:

$$\texttt{addr} = \texttt{coin\_node\_y} \times 30 + \texttt{coin\_node\_x}.$$

Write port (controlled by `player_ctrl`):

- `wraddress = coin_wr_addr`,
- `data = coin_din` (0 when eaten),
- `wren = coin_we`.

Read port (used by VGA overlay logic):

- `rdaddress = coin_rd_addr` computed from current (x, y).

Because this IP also has a registered output, I created a small pipeline so the coin is drawn in the correct pixel location.

Coin Addressing from VGA Coordinates

To bind coins to tiles/nodes:

1. Determine if the current pixel is inside the maze:

```
coin_in_map = 0;
if (x >= MAP_X0 &&
    x <  MAP_X0 + NODES_X*TILE_SIZE &&
    y >= MAP_Y0 &&
    y <  MAP_Y0 + NODES_Y*TILE_SIZE) begin

    coin_in_map = 1;
    coin_node_x = (x - MAP_X0) / TILE_SIZE;
    coin_node_y = (y - MAP_Y0) / TILE_SIZE;
end
```

2. Compute the read address:

```
coin_rd_addr = coin_node_y * NODES_X + coin_node_x; // NODES_X = 30
```

3. Pipeline the node coordinates and the ROM output:

```
always @(posedge CLOCK_50) begin
    if (reset) begin
        coin_node_x_r <= 0;
        coin_node_y_r <= 0;
        coin_in_map_r <= 0;
        coin_bit_r    <= 0;
    end else begin
        coin_node_x_r <= coin_node_x;
        coin_node_y_r <= coin_node_y;
        coin_in_map_r <= coin_in_map;
        coin_bit_r    <= coin_bit;  // coin_bit is the registered RAM output
    end
end
```

This keeps `coin_node_x_r`, `coin_node_y_r`, and `coin_bit_r` aligned so the coin is drawn at the correct tile center.

Coin Drawing Region

Each coin is a 4×4 yellow square around the center of its node:

```
coin_x_pix = MAP_X0 + coin_node_x_r * TILE_SIZE + TILE_SIZE/2;
coin_y_pix = MAP_Y0 + coin_node_y_r * TILE_SIZE + TILE_SIZE/2;

coin_here =
    coin_in_map_r && coin_bit_r &&
    (x >= coin_x_pix - 2) && (x <= coin_x_pix + 1) &&
    (y >= coin_y_pix - 2) && (y <= coin_y_pix + 1);
```

If `coin_here` is 1, the pixel is painted yellow (unless overwritten by Pac-Man).

Earlier bugs where coins appeared duplicated or shifted by one tile were due to mixing unregistered coordinates with registered RAM output. The pipeline fixed that.

heres a simulation:

2.7 Alien Controller

The alien controller is structurally similar to `player_ctrl`:

- Keeps its own (`alien_x`, `alien_y`),

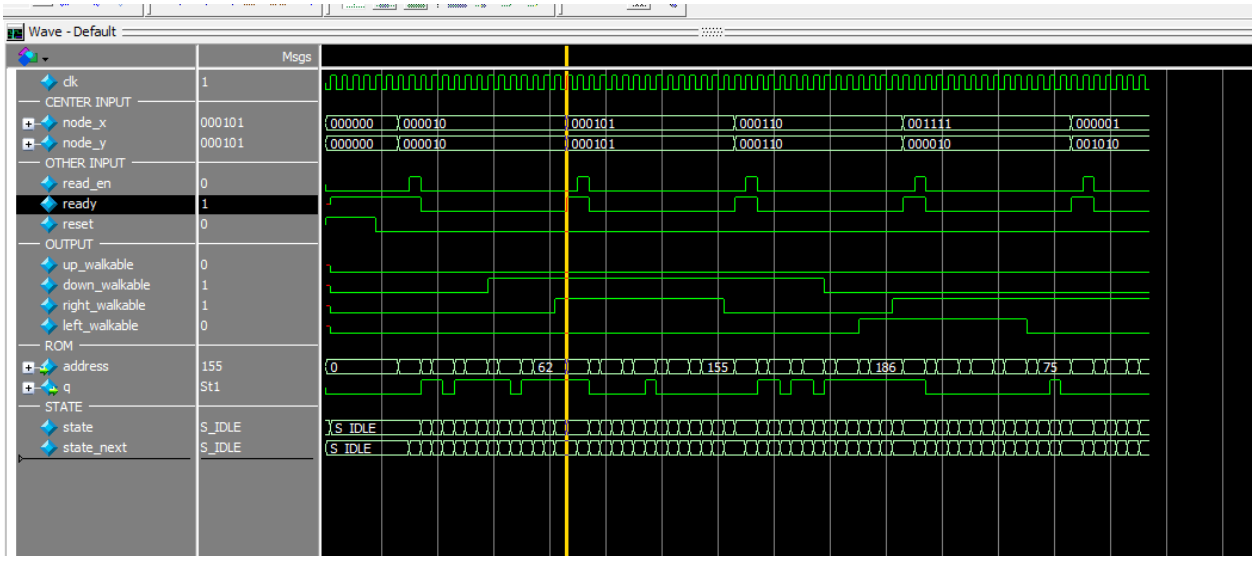


Figure 4: simulation

- Uses its own `node_neighbour_reader` and `node_map_rom`,
- Has its own movement direction state and clock divider (so it moves slower than the player),
- For "directive" behaviour, it does not just wander randomly; it chooses directions that roughly head towards the player:
  - It can bias horizontal movement when `player_x` is less/greater than `alien_x`,
  - Or vertical movement when `player_y` differs.

The logic is still fairly simple, but it is enough that the alien does not just jitter randomly. When `alien_x == player_x` and `alien_y == player_y`, the system can treat that as a collision.

2.8 Top-Level Integration (`DE1_SoC`)

The final `DE1_SoC` module wires everything:

- Handles `reset = ~V_GPIO[0]` and `CLOCK_50`.
- Instantiates `video_driver`, `vga_background`, `n8_driver`, `edge_detect`, `player_ctrl`, `alien_ctrl`, and `coin_map_ram`.
- Implements the overlay order:
  1. Start with tile background (`bg_r, bg_g, bg_b`).
  2. Draw coins on top if `coin_here`.
  3. Draw player on top if `pac_here`.
  4. (Optionally) draw alien on top with a different color.

The seven-segment displays are used to show either player coordinates or score. In the final configuration for scoring, I repurposed them to display a decimal score, with a small counter that increments each time a coin is eaten.

2.9 Creating the Maze: Hand-Drawn Map → Walkable Node Grid → MIF Files

A large and surprisingly non-trivial part of this project was building the actual maze that Pac-Man and the alien navigate. Instead of using a pre-made map from a textbook or demo code, I decided to design my own map from scratch, draw it manually, and convert it into two separate memory representations:

1. A tile-based color map (`tiles.mif`) for VGA drawing.
2. A walkable node grid (`node_map_rom.mif`) used exclusively for logic (movement, collisions, AI paths).

Both representations had to align visually and logically despite being different sizes. Getting this correct required several iterations and a custom Python pipeline.

Hand-Drawing the Map in MS Paint (Tiles)

The first step was to design the maze layout exactly as I wanted it. I opened MS Paint and manually drew:

- 29 columns × 35 rows of tiles.
- Each tile represented 10×10 pixels on screen.
- Walls were drawn in blue.
- Floor/path areas drawn black.
- I left two UI bands:
  - 30 pixels at the top for score.
  - 30 pixels at the bottom.

This image became the visual layer of the game.

However, only drawing the tiles is not enough. Tile boundaries do not necessarily equal movement nodes. Pac-Man moves along intersections, not tile centers. That meant I needed a second, logically consistent grid.

Designing the Walkable Node Grid System (Movement Logic Grid)

To solve the alignment issue between visuals and movement, I created a 30×36 logical grid, where each cell represents a node (an intersection point or walkable junction):

- Node grid size: 30 columns × 36 rows = 1080 nodes.
- The node coordinate → screen pixel mapping is:

$$\texttt{pixel\_x} = \texttt{MAP\_X0} + \texttt{node\_x} \times 10, \quad \texttt{pixel\_y} = \texttt{MAP\_Y0} + \texttt{node\_y} \times 10.$$

Nodes represent:

- Path intersections.
- Horizontal/vertical centers of corridors.
- The tunneling nodes at row 17 (col 0 ↔ col 29).

I then manually went through my MS Paint tile map and marked each node as either:

- 1 = walkable.

- 0 = wall / non-walkable.

This forms the true gameplay logic and is consumed by:

- The player controller.
- The alien controller.
- The coin placement system.

The nodes must match the tile map visually so the player never appears to walk through walls. That consistency is what required all the careful scripting.

Python Scripts for Converting Tile Image → MIF File

Once the tile image was finalized in MS Paint, I wrote a Python script (using `numpy` + `PIL`) to convert it into a `.mif` file readable by Quartus.

Key script features:

- Loads PNG/JPG of the map.
- Divides into 10×10 pixel tiles.
- Reads dominant color per tile (blue wall, black floor, etc.).
- Maps each tile to a palette index.
- Outputs a `.mif` file for a ROM of depth 1015 (29×35) and width 8 bits.

Why `numpy`?

- Fast pixel processing.
- Easy slicing of 2D arrays.
- Clean way to isolate each tile block.

Why MIF?

- The VGA background module uses `altsyncram` (ROM IP), which requires `.mif` for initialization.

Python Script for Generating the Walkable Node MIF File

The walkable grid required another script because MS Paint tiles do not map 1:1 to nodes.

I created a Python pipeline where I could:

- Load the tile map image.
- Check tile colors to infer whether the intersection is walkable.
- Produce another `.mif` file: 1080 entries, 1 bit each.
- Use index = `node_y * 30 + node_x`.

This file became `node_map_rom.mif`.

Why Two MIF Files? Why Not One?

Originally, I attempted to use a single MIF file for both visuals and logic. This failed badly because:

Tiles ≠ Nodes

- Tiles are 29×35.
- Nodes are 30×36.
- Tiles are 10×10 pixels, but nodes represent intersection points.
- A wall tile does not necessarily mean its corners are blocked.

The map needed to reflect Pac-Man's corridor logic, which is not pixel-aligned. This is why many early bugs appeared:

- Sliding problems.
- Getting stuck in corners.
- Walking through visible walls.
- Movement not matching visuals.

Separating the maps completely fixed these issues.

Links to Python Conversion Tools

The scripts can be linked or attached as supplementary material, for example:

- Tile → MIF generator: `https://colab.research.google.com/drive/1cHMQyLd1mg9Yhx5bDUfw7dr7LCym3Y3?usp=sharing`
- Node grid generator: `https://colab.research.google.com/drive/1xn1WFWVl4MvxZHuNGmpIK_WmjJPTEXHN?usp=sharing`

Verifying MIF Alignment in Simulation

Before loading onto hardware, I wrote a simple ModelSim testbench that:

1. Instantiated the ROM with my MIF.
2. Dumped all 1015 (tiles) or 1080 (nodes) locations.
3. Printed the grid to console.
4. Let me visually check patterns like:

```
1111100000000111111
0000100000000100000
0000101111111100000
```

This matched the map's corridor structure. I repeated the test for:

- Tile map (color indices).
- Node map (walkable flags).

Only after confirming correctness did I proceed with hardware integration.

---

3 Results

3.1 Differences from Original Proposal

In the initial proposal, my plan was roughly:

- Display a static Pac-Man style maze using VGA.
- Move a single player sprite based on N8 input across tiles or nodes.
- Possibly show a simple score or just coordinates.

As the project evolved, the final implementation changed in several ways:

1. **Separated tiles vs. nodes**

   Originally I thought I could directly tie movement to the tiles used for drawing. After running into alignment and wall collision issues, I redesigned the system to have:

   - A separate node grid in `node_map_rom` for logic.
   - A 29×35 tile map in `tiles.mif` for visuals only.

2. **Added coins using a dual-port RAM**

   This was an extra feature beyond the minimal spec:

   - Every walkable node can have a coin.
   - Coins are drawn at the correct intersections and cleared as Pac-Man moves through.

3. **Implemented Pac-Man-style continuous movement**

   Instead of moving exactly one node per button press, the player:

   - Always keeps moving in the current direction.
   - Can pre-queue direction changes at intersections, like the real game.

4. **Added an alien (ghost)**

   The proposal did not necessarily include an AI opponent. The final project has:

   - A second moving sprite with node-based motion.
   - Its own neighbour reader and ROM.
   - Simple directive behaviour towards the player.

5. **Proper score counting**

   Instead of just coordinates, I added a dedicated score counter that maps onto HEX displays (0–9 per digit), increasing every time a coin is eaten.

3.2 Functional Testing

I tested the design in three stages: simulation, HEX display debugging, and on-board behaviour.

Simulation

1. `node_neighbour_reader` testbench I wrote a testbench that instantiated the real `node_map_rom` with the project's MIF. It provided a list of (`node_x`, `node_y`) positions (e.g., (2, 2), (5, 5), (6, 6), (15, 2), etc.) and pulsed `read_en`. For each, the testbench:

- Waited until `ready` went high again.
- Printed out `up_walkable`, `down_walkable`, `left_walkable`, `right_walkable`.
- Compared them against expected values manually derived from the map.

This confirmed the FSM state sequence (ADDR → READ → LATCH) was correctly aligned with the ROM latency.

2. Player movement testbench In simulation, I forced button pulses and observed:

- Player position (`player_x`, `player_y`).
- Direction state (`cur_dir`, `desired_dir`).
- Requests to neighbour reader (`read_en`).
- Responses (`up_walkable`, etc.).

I verified:

- Up/Down/Left/Right moves decremented/incremented coordinates correctly.
- The player did not move into nodes where the ROM bit was 0.
- The tunnel warp triggered only at (0, 17) ↔ (29, 17).

3. Coin clearing simulation I initialized coin bits to 1 at known nodes, moved Pac-Man through those nodes, and verified that `coin_we` pulsed exactly once per node entry and that the RAM bit changed from 1 → 0.

On-Board Testing (DE1-SoC)

1. Maze alignment By displaying `player_x` and `player_y` on HEX displays, I could verify that:

- Starting position matched a visually open intersection.
- Moving around the map at node level matched intersections on the drawn maze.

2. Collision with walls I attempted to walk into walls from many angles:

- The player never stepped onto a node marked as wall in `node_map_rom`.
- The earlier bug where Pac-Man got stuck at corners and could not slide was fixed by the neighbour reader + sticky direction logic.

3. Coins Initially, I saw misaligned and duplicated coins. After the pipeline fix:

- Every walkable intersection that should have a coin had exactly one 4×4 coin, centered at the intersection.
- As Pac-Man passed through a node, the coin disappeared and did not reappear.

4. Alien behaviour I confirmed the alien:

- Stayed on valid nodes only.
- Moved slower than Pac-Man (separate clock divider).
- Roughly chased the player (not purely random).
- Triggered collision when they overlapped.

3.3 Performance and Limitations

Movement speed is controlled by divider constants. With the current values:

- Pac-Man motion is responsive but not too fast.
- Alien motion is slower and more predictable.

Main limitations:

- AI is still simple; it does not do full path-finding.
- Score range is limited by HEX display digits.
- There is no explicit game-over screen yet; collision is detected logically but not yet drawn as a separate state.

---

4 Experience Report

4.1 Implementation Challenges

This lab was more of a miniature system design project than a single algorithm. A few specific pain points:

1. Node vs. Tile Confusion

I originally tried to tie movement directly to the 29×35 tile grid used for `vga_background`. That caused:

- Misalignment between visual walls and logical walls.
- Cases where Pac-Man could move through what looked like a wall, or get blocked in an open corridor.

The fix was uncomfortable but necessary: I defined a separate 30×36 node grid (`node_map_rom`) and treated that as the only source of truth for movement, even though the tiles visually resemble the grid.

2. ROM Latency and Latching Neighbour Flags

The biggest debugging time sink was handling the synchronous ROM outputs correctly:

- At first, I changed `rom_addr` and latched `rom_q` in the same state, assuming it was a combinational ROM.
- On hardware, this caused `up_walkable` to hold the wrong values and Pac-Man got stuck, especially at corners.
- In ModelSim, I could see that in the READ state I was still reading the previous address's data.

Fix: introduce explicit ADDR $\rightarrow$ READ $\rightarrow$ LATCH sequence, with `rom_q` captured only in the last state. Once I did that consistently, the wall collisions behaved exactly as expected.

3. Shared ROM vs. Independent Readers

At one point I tried to have both the player and alien use the same `node_map_rom` instance. That idea was bad:

- Each FSM wanted to control `rom_addr` at the same time, and I did not have an arbiter.
- The ROM output was essentially garbage from the perspective of each controller.

I resolved this by having each `node_neighbour_reader` wired to its own ROM instance (or at least treating them conceptually independent), so they could operate without interfering.

4. Coin RAM Registered Output and Ghost Coins

Early coin display attempts had weird artefacts:

- Extra coin pixel to the right.
- Duplicate coins between tiles.
- Flickering behaviour.

The root cause was mixing a registered RAM output (`q`) with non-registered node coordinates. The fix was to pipeline both the node coordinates and the coin bit together, so that at any pixel the coin decision is based on aligned data.

5. Always-Moving Pac-Man vs. Single-Step Movement

Switching from "move once per button press" to "always moving with direction changes" introduced subtle bugs:

- If I tried to change direction into a blocked tile, Pac-Man could stop moving entirely instead of continuing straight.
- Sliding along walls also broke initially, because I was resetting `cur_dir` incorrectly when `desired_dir` was blocked.

The solution was the sticky logic:

- Try `desired_dir` first if walkable; otherwise keep `cur_dir` if that is still walkable.
- Only change `cur_dir` if we can actually step into the desired direction.

6. Debugging with HEX Displays

Showing `player_x` and `player_y` on HEX0–HEX3 was surprisingly helpful:

- It let me confirm which node Pac-Man thought he was on.
- I could correlate that with visible map intersections and quickly spot off-by-one errors.

4.2 Reflections on Map Construction and Conversion Pipeline

One of the most unexpectedly time-consuming parts of the entire project was the map creation pipeline. Making a custom Pac-Man maze is easy visually, but turning that map into something a hardware design can consume reliably is a much deeper technical process.

Tile Map vs. Node Map Mismatch

Initially, I assumed I could infer walkability directly from the tile map. This created several serious bugs:

- Player sliding incorrectly.
- Player stopping at wrong places.
- Player walking inside walls.
- Misaligned collision checks.
- Impossible corners.

Once I accepted that game logic cannot rely on tile art, I redesigned everything using a dedicated node map with precise 30×36 indexing.

Python MIF Conversion

I underestimated how tricky it is to:

- Read image files reliably.
- Slice them exactly into 10×10 blocks.
- Avoid off-by-one errors.
- Maintain consistent indexing order (row-major).
- Output MIF files that Quartus understands (radices, `WIDTH/DEPTH`, etc.).

The rigid formatting requirements of `.mif` forced me to write careful, explicit loops and sanity checks.

Alignment Testing Took Many Iterations

I would load the tile map into hardware and see the walls look correct, but the player logic would be offset by one tile. Fixing this required:

- Printing node indices on HEX displays.
- Using colored debug overlays on VGA.
- Checking ModelSim dumps.
- Re-running MIF generator scripts.

It took multiple revisions until the visual map and logical grid aligned perfectly.

Once It Worked, Everything Else Fell Into Place

After the maps and MIFs were correct:

- Player movement became smooth.
- Sliding worked as expected.
- Tunnel wrap lined up correctly.
- Coins displayed in the correct node centers.
- The alien AI respected walls perfectly.
- All movement and collision bugs basically disappeared.

The correctness of the entire game hinged on getting the MIF files and grid alignment right.

Challenges:

- The hardest part as a team was aligning our mental models of the coordinate systems (x/y in pixels vs. `node_x`/`node_y` vs. tile indices).
- Once we agreed on a consistent mapping and documented it, everything else became much easier.

If I worked alone: I would simply state that all design, implementation, and debugging were done individually.

4.3 Lessons Learned

- Synchronous memories are not free "arrays": IPs like `altsyncram` or generated ROMs usually add a register on the output. Ignoring that detail led directly to subtle bugs in both wall collision and coin rendering.
- Visual debugging is powerful, but numbers matter: Watching the VGA output helped, but what really nailed bugs was correlating it with HEX-displayed node indices and ModelSim waveforms.
- Pac-Man style movement is a small but real control problem: On paper "always move in a direction unless blocked" sounds trivial. Implementing it cleanly with queued direction changes, warp tunnels, and sliding along walls ended up being a non-trivial FSM design that I now understand much better.

---

5 Formatting, Code, and Demo

5.1 Formatting / Figures

In the final PDF, I will:

- Convert the described top-level block diagram, neighbour reader FSM, and Pac-Man controller FSM into clean figures.
- Label them as Figure 1, Figure 2, etc. with captions.
- Ensure all signals and modules mentioned in the text match the names in the code.

5.2 SystemVerilog Code

I will upload:

- `DE1_SoC.sv` (top level),
- `player_ctrl.sv`,
- `alien_ctrl.sv`,
- `node_neighbour_reader.sv`,
- `node_map_rom.v` (or whatever Quartus generated),
- `coin_map_ram.v`,
- `vga_background.sv`,
- `video_driver.sv`,
- `n8_driver.v`, `edge_detect.sv`,

- Any testbenches (`node_neighbour_reader_tb.sv`, etc.).

5.3 DEMO

video link:(https://drive.google.com/file/d/1U1415-k03LlAY7HNaKb28D4PAVEdLPTT/view?usp=sharing)