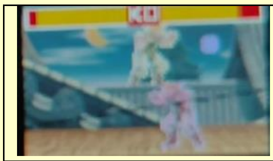
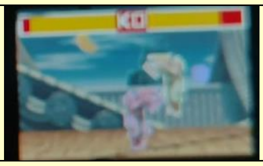
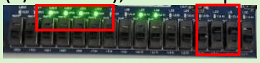
S3 06
Yow Keng Yee Samuel A0230054X, Leong Ko Ryan Jasper A0272790X, Lim Yee Kian A0272240R, Jonas Seet A0269849B

<table>
<tr><th colspan="3">Personal and Team Improvements</th></tr>
<tr><th>Student & Improvement Name</th><th>Improvement Description</th><th>Images/Photos</th></tr>
<tr>
<td>Team (S3-06)</td>
<td>A simplified version of street fighter with music was implemented. 2 Basys boards can be connected for a 2-player game, or a single board can play against a pseudo-random AI.
<b>Button mapping</b>
<table>
<tr><td>U</td><td>Jump Up</td><td></td><td></td></tr>
<tr><td>L / R</td><td>Move Left/Right</td><td>L &gt; D &gt; R &gt; C</td><td>Special Attack</td></tr>
<tr><td>C</td><td>Attack</td><td>U &gt; D &gt; L &gt; R &gt; L &gt; R &gt;</td><td>Bullet attack</td></tr>
</table>
</td>
<td><b>SW Mapping</b>
<table>
<tr><td>sw[0]</td><td>ON for 2s to reset game anytime</td></tr>
<tr><td>sw[1]</td><td>ON for master, OFF for slave. <b>During programming, set both boards to ON.</b></td></tr>
<tr><td>sw[2]</td><td>ON to enable audio</td></tr>
<tr><td>sw[7]</td><td>Player 2 receives AI inputs</td></tr>
<tr><td>sw[8]</td><td>Player 1 receives AI inputs</td></tr>
</table>
</td>
</tr>
<tr>
<td><b>Student A: Jonas Seet</b>
AI, Movement, Combos, Pseudo randomness</td>
<td><u><b>Pseudo-random AI:</b></u>
<ul>
<li><b>Pseudo random numbers are generated by a LFSR</b> (Linear-Feedback shift register (XOR gates version)) [1] (random number also used in rendering (see Student B))</li>
<li>AI will check the pseudo-random number to determine what move it should do. (attacks and movement are checked separately)</li>
<li>For added realism and challenge, AI gets more aggressive at low health. (faster check rate)</li>
</ul>
<u><b>Combo Moves:</b></u>
<ul>
<li>Combos are monitored in real-time.</li>
<li>Player must input a <b>set combination of inputs within a fixed time window</b> of the last input to execute one of two "special moves" (see: right).</li>
<li>Upon a valid input, a timer starts, and constantly checks for the next required input, and so on; until the combo executes/breaks.</li>
</ul>
<u><b>Player Movement Handler</b></u>
<ul>
<li>Translated raw player inputs into movement and combat, including debouncing.</li>
<li>Prevented “spamming” of player holding down attacks by making “held down” buttons only trigger attacks for one frame.</li>
</ul>
</td>
<td>
When SW8/SW7 is on, player 1/2 is controlled by the AI
<table>
<tr><td> </td><td> </td></tr>
<tr><td>Combo: Ranged bullet attack [UP->DOWN->LEFT->RIGHT->LEFT->RIGHT->ATTACK]</td><td>Combo: Super Punch [LEFT->DOWN->RIGHT->ATTACK]</td></tr>
</table>
*(To prevent “button mashing” to get combos, pressing any other button (not part of the combo sequence) will cause the <b>combo window to break</b>.)</td>
</tr>
<tr>
<td><b>Student B: Yow Keng Yee Samuel</b>
Graphics</td>
<td><ul>
<li>Adapted image drawing script[2] to use always blocks, which are used for image generation.</li>
<li>Rendered sprites[3] and background[4].</li>
<li>Wrote animations for normal attack, super attack, movement, and getting hit.</li>
<li>Wrote animations for the health bar when each character has a decrease in health.</li>
<li>Designed home/end screen, making use of 7 Segment displays, and btnC to reset game.</li>
<li>Wrote animations for circular bullet.</li>
</ul>
Bullet colour is based on a random 5-bit value (from Pseudo-random AI). This value is multiplied by 10, taken as hue value. A HSV to RGB[5] script was written which gives the bullet 32 possible colours. (Taking saturation and value as maximum values for max brightness)</td>
<td><table>
<tr><td> </td><td> </td></tr>
<tr><td>Sprite 2 (Left) is simply Sprite 1 (Right) with the green value bit shifted right.</td><td>Bullet rendered when super special attack performed, where bullet is a random bright colour.</td></tr>
</table>

Use of 7 Segment Display (WIN, ----, LOSE) and rectangle (Green, -Red). Hold down btnC for 2s to reset game.</td>
</tr>
</table>

[1] https://docs.amd.com/v/u/en-US/xapp052
[2] https://github.com/BenedictChannn/EE2026-Project/blob/main/imageDrawing.py
[3] https://www.spriters-resource.com/arcade/streetfighter2/sheet/129870/
[4] https://www.youtube.com/playlist?list=PL8_5sYhn3XymSIUz3oio50XvenNZkCeCd
[5] Adapted from https://github.com/monkbroc/particle-hsv/blob/master/src/hsv.cpp

Student C: Leong Ko Ryan Jasper Game Mechanics	Physics Engine: • Management of characters positioning (updated at 20 ticks per second) and ability to **jump/move checks** and update to movement handler. • Implemented **gravity** through updating vertical displacement using **vertical velocity** and **acceleration.** • Implemented **collision detection** preventing players from overlapping and crossing the edges of the screen. • Player **Hit detection/ Facing direction**. Health Management: • Management of player health and damage calculation to different attacks • Output hit status to sprite module Game state management: • Management of **game states** when one player is KO. • Implemented **resets** for positioning and health	 Player at apex of jump  Change of direction $s = ut - \frac{1}{2}gt^2$ displacement equation Hit box visualisation - Red attack range - Black collision range
Student D: Lim Yee Kian Audio and 2 Board Communication	**Self-Implementation of in-game music module using AMP2 PMOD**[6] • SW[2] enables and disables audio. • Implemented background & critical health music, overlay sounds effects for punching and jumping. Audio stops when game ends. • Utilized different custom clocks to generate different tones[7] and used music states to vary tones and generate melody. • Takes in game state, player health, punching and jumping statuses of both characters to determine music output. **Communication between 2 Basys Board for 2 player controls** • SW[1] ON sets the board to Master, and OFF to Slave**. On set-up, set all boards to Master.** • PMOD connectors: Input data: JA; Output data: JXADC • Handle slave input controls and sending them to the player control handler on the master board. • Passing of game state from Master to Slave to display alternate 7 Seg win/lose message. • Master board: Led[10]: Up, led[11]: Left, led[12]: Right, led[13]: Down → show slave movement button presses received.	Background game music plays during gameplay (left). Sound effect overlays when either player is jumping (centre) or punching (right) Critical music plays when a player reaches critical health threshold (1/3 health), music stop when game ends  led[13:10] indicating slave movement received, sw[1] to choose master/slave, sw[2] to enable/disable audio 7 Segment game state information: playing (left), master won (centre), master lost (right) Wiring of the Master(Left) and Slave (Right)

Master			**Slave**		
Pin Type	**Pin**	**Purpose**	**Pin Type**	**Pin**	**Purpose**
Input	JA 1	player2UpBtn	Output	JXAC 1	btnU
	JA 2	player2DownBtn		JXAC 2	btnD
	JA 3	player2LeftBtn		JXAC 3	btnL
	JA 4	player2RightBtn		JXAC 4	btnR
	JA 7	player2AttackBtn		JXAC 7	btnC
Output	JXAC 1	winState[0]	Input	JA 1	slave_winState[0]
	JXAC 2	winState[1]		JA 2	slave_winState[1]
	JXAC 3	winState[2]		JA 3	slave_winState[2]

[6] **Background music generation:** Self-Implementation, inspired by: https://github.com/FPGADude/Digital-Design/tree/main/FPGA%20Projects/Star%20Wars%20Imperial%20March%20Song

[7] https://musescore.com/longlivethecommentfresh/mortal-kombat-theme-the-immortals