FPGA Battleship

รายงานนี้เป็นส่วนหนึ่งของรายวิชา 01076112 Digital System Fundamentals

ภาคเรียนที่ 1 ปีการศึกษา 2567

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

กิตติกรรมประกาศ | Acknowledgements

โครงงาน FPGA Battleship เป็นโครงงานประจำรายวิชา Digital System Fundamental โดยมีจุดประสงค์เพื่อประยุกต์นำความรู้การต่อวงจรดิจิทัลที่เรียน มาพัฒนาเป็นโครงงาน โดยใช้บอร์ด FPGA เป็นเครื่องมือในการพัฒนา เดิมที ในขั้นแรกของการคิดและเสนอหัวข้อโครงงาน ผู้จัดทำยังไม่ได้มีความรู้เกี่ยวกับการพัฒนาโปรแกรม การต่อวงจรดิจิทัลด้วยวิธี Schematics ตลอดจนยังไม่มีความรู้ในการเขียน VHDL มากนัก อย่างไรก็ตาม ตลอดระยะเวลาการทำโครงงาน ผู้จัดทำได้รับความช่วยเหลือและคำแนะนำจากผู้คนมากมาย จนทำให้โครงงานสำเร็จลุล่วงได้ ผู้จัดทำจึงอยากกล่าวขอบคุณบุคคลที่มีส่วนช่วยเหลือในโครงงานดังต่อไปนี้

รศ. ดร. เจริญ วงษ์ชุ่มเย็น อาจารย์ประจำวิชา Digital System Fundamental ที่ให้ความรู้ด้านการต่อวงจรดิจิทัลด้วยวิธี Schematics ตลอดจนการประยุกต์นำวงจรดิจิทัลในบทเรียนมาประยุกต์ใช้ในการทำโครงงานตลอดทั้งภาคเรียน

พี่ ๆ TA ประจำวิชา ที่ให้คำแนะนำ ตลอดจนตอบคำถามหลายคำถามที่ผู้จัดทำสงสัยได้เป็นอย่างดี นอกจากนั้น ยังเป็นแรงผลักดันให้ผู้จัดทำในการทำโครงงานให้สำเร็จลุล่วงอีกด้วย

Network and Cloud Laboratory สำหรับสถานที่และสิ่งอำนวยความสะดวกในการทำโครงงานตลอด 1 สัปดาห์ ตลอดจนเป็นที่นอนพักผ่อน และร้านค้าสำหรับซื้ออาหารและเครื่องดื่มในขณะทำโครงงานตลอดทั้งคืน

โป๊ท สุวิจักษณ์ หยกพิทักษ์โชค สำหรับการลองผิดลองถูก และความรู้ในการเขียน VHDL ในการแสดงผลกราฟิกผ่านพอร์ต VGA ตลอดจนความรู้ในการใช้ IP ClockWizard ในการควบคุมสัญญาณนาฬิกา

ChatGPT ที่ช่วยตอบคำถามทุกอย่างที่ผู้จัดทำสงสัย Generate โมดูลต่าง ๆ ที่ใช้ในโครงงาน ตลอดจนช่วยในการแก้ Bug ต่าง ๆ ที่ผู้จัดทำสร้างขึ้น

และบุคคลอื่น ๆ ที่มีส่วนเกี่ยวข้องแต่ผู้จัดทำมิได้เอ่ยนาม ณ ที่นี้ ทุกท่านล้วนมีส่วนสำคัญอย่างยิ่งที่ทำให้โครงงานนี้สำเร็จลุล่วง ผู้จัดทำขอขอบคุณทุกท่านจากใจจริงสำหรับความช่วยเหลือ ความอดทน และกำลังใจที่มอบให้ตลอดระยะเวลาการทำโครงงาน การสนับสนุนจากทุกคนเป็นแรงผลักดันที่ทำให้ผู้จัดทำสามารถก้าวข้ามอุปสรรคต่าง ๆ และทำให้โครงงานนี้สำเร็จตามเป้าหมายที่วางไว้

คณะผู้จัดทำ

สารบัญ | Table of Contents

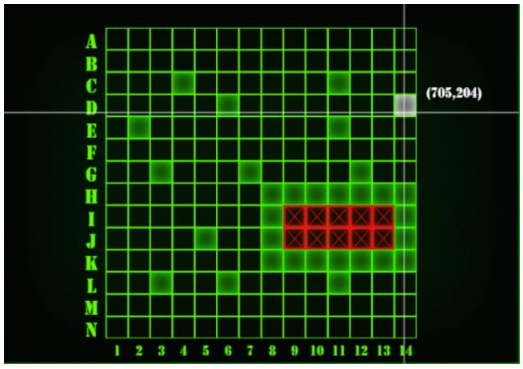
ที่มาของโครงงาน | Introduction

Sea Battleship เป็นเกมกระดานแนวกลยุทธ์ (Strategy) ที่ได้รับความนิยมอย่างมากในอดีตเนื่องจากมีกฎกติกาที่เข้าใจง่าย นอกจากนั้นยังเสริมสร้างทักษะกระบวนการคิดและกระบวนการวางแผนได้เป็นอย่างดี ปัจจุบันได้มีการปรับเปลี่ยนรูปแบบของเกมจากเกมกระดาน ให้มาอยู่ในรูปแบบดิจิทัล โดยสามารถเล่นผ่านเครื่องเกมต่าง ๆ รวมถึง PC ได้ด้วย

รูปที่ 1 บอร์ดเกม Sea Battleship

ที่มา https://www.walmart.com/ip/211020-Classic-Battleship-Game-Strategy-Board-Game-Sea-Battle-Toy-Retro-Series-for-Kids/236465351

FPGA Battleship เป็นโครงงานที่ได้รับแรงบันดาลใจมาจากเกม Sea Battleship แบบดั้งเดิม โดยจะพัฒนาเกมบนบอร์ด FPGA (Field Programmable Gate Arrays Board) จำนวน 2 บอร์ด แทนเครื่องเล่นเกมของผู้เล่น 2 คน โดยบอร์ด FPGA ทั้งสอง จะส่งข้อมูลของเกมแบบ PIPO (Parallel-in Parallel-out) เพื่อประมวลผลตรรกะ (Logic) ต่างๆ ของเกม ซึ่งนอกจากจะได้เกมฝึกทักษะการวางแผนและกลยุทธ์แล้วยังได้ฝึกทักษะการพัฒนาโปรแกรมบนบอร์ด FPGA ซึ่งทำให้สามารถเข้าใจหลักการทำงานของฮาร์ดแวร์ได้ดีมากยิ่งขึ้นด้วย



รูปที่ 2 ตัวอย่างเกม Sea Battleship ที่ถูกปรับปรุงให้สามารถเล่นได้บน PC ในรูปแบบออนไลน์

ที่มา https://www.battleshiponline.org/

กระบวนการหาข้อมูล | Researching Process

ในการทำโครงงาน มีความจำเป็นต้องทำงานกับ Hardware ไม่คุ้นเคย ตลอดจนมีการใช้โปรโตคอลต่าง ๆ ในการสื่อสารกันระหว่างบอร์ด FPGA จึงต้องมีการหาข้อมูลที่จำเป็นสำหรับการทำโครงงาน ดังต่อไปนี้

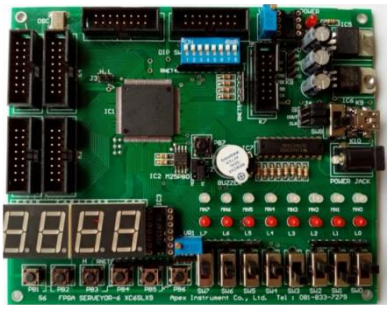
1. FPGA Surveyor-6 XC6SLX90

สำหรับอุปกรณ์ที่ใช้ในการพัฒนาโครงงาน คือบอร์ด FPGA รุ่น Surveyor-6 XC6SLX90 ซึ่งเป็นบอร์ดที่ใช้ชิพ FPGA (Field Programable Gate Array) รุ่น XILINX Spartan-6 XC6SLX9 ซึ่งชิพ FPGA แต่ละรุ่น ต่างมีคุณสมบัติ (Specification) ที่แตกต่างกัน และอาจนำมาสู่ข้อจำกัดในการพัฒนาโครงงานได้ จึงต้องมีการศึกษาคุณสมบัติของชิพ FPGA ที่ใช้ ซึ่งมีรายละเอียดคร่าว ๆ ดังต่อไปนี้

Feature	Spartan-6 XC6SLX9
Logic Cells	9,152
Configurable Logic Blocks (CLBs) - Slices - Flip-Flops - Max Distributed RAM (Kb)	 1,430 11,440 90
DSP48A1 Slices	16
Block RAM - 18 Kb - Max (Kb)	 32 576
CMTs	2
Memory Controller Blocks (Max)	2
Endpoint Blocks for PCIe	0
Maximum GTP Transceivers	0
Total I/O Banks	4
Max User I/O	200

จำนวน Flip-Flops และ Logic Slice ที่มีให้ใช้ในชิพ เป็นคุณสมบัติที่สำคัญในการพัฒนาโครงงาน คือต้องมีการจัดสรรทรัพยากรให้เพียงพอสำหรับ Hardware ที่มีอยู่อย่างจำกัด

บอร์ด FPGA รุ่น Surveyor-6 XC6SLX90 ได้มีการนำชิพ Xilinx Spartan-6 XC6SLX90 มาประกอบรวมกับอุปกรณ์ต่าง ๆ เช่น Push Button, Switch, LED, Seven Segments Display เพื่อให้มีความสะดวกต่อการศึกษาและใช้งาน โดยบอร์ด FPGA รุ่นดังกล่าวมีรายละเอียดดังต่อไปนี้



รูปที่ 3 บอร์ด FPGA รุ่น Surveyor-6 XC6SLX9

Components	**Quantity**
XILINX Spartan-6 XC6SLX9 FPGA Chip	1
Crystal Oscillator (20 MHz)	1
2 Digits Seven Segments Display	2
Push Button	6
- One Shot Push Button	2
- Active Push Button (Debounced)	2
- Active Push Button (Bounced)	2
Slide Switch	8
Dip Switch	8
Red LED	8
Logic Monitor LED	8
Buzzer	1
General Input-Output (GPIO) Pins	48 + 8

ในการทำโครงงาน จะใช้อุปกรณ์ Built-in ต่าง ๆ บนบอร์ด FPGA เช่น Push Button สำหรับควบคุม และ GPIO Pin สำหรับส่งข้อมูลการแสดงผลกราฟิกผ่านพอร์ต VGA และส่งข้อมูลระหว่าง FPGA ทั้งสอง

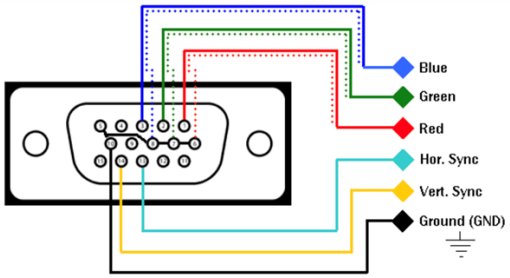
2. VGA Connector and Protocol

VGA (Video Graphics Array) เป็นมาตรฐานการแสดงผลกราฟิกที่ใช้ในการส่งสัญญาณภาพจากคอมพิวเตอร์ไปยังจอภาพ ซึ่งสาย VGA ออกแบบให้ใช้งานเพื่อส่งสัญญาณวิดีโอแบบ Analog ระหว่างคอมพิวเตอร์ กับจอมอนิเตอร์ โปรเจกเตอร์ หรือโทรทัศน์

โดยในโครงงาน ต้องมีการแสดงผลกราฟิกเกมผ่านหน้าจอมอนิเตอร์ ซึ่งต้องมีการศึกษาวิธีการเชื่อมต่อ ตลอดจนโปรโตคอลในการส่งข้อมูลผ่านพอร์ต VGA ซึ่งมีรายละเอียดดังต่อไปนี้

VGA Connector Pinout

พอร์ต VGA มาตรฐาน มีจำนวน 3 แถว แถวละ 5 Pins รวมทั้งสิ้น 15 Pins ดังรูปที่ 4



รูปที่ 4 รูปแบบการเชื่อมต่อของพอร์ต VGA (VGA Connector Pinout)

ที่มา https://www.nextpcb.com/blog/vga-connector-pinout

Pin No.	Description	Pin No.	Description	Pin No.	Description
1	RED Video (0.7 V_{pp})	**6**	RED GND	**11**	Monitor ID
2	GREEN Video (0.7 V_{pp})	**7**	GREEN GND	**12**	Monitor ID
3	BLUE Video (0.7 V_{pp})	**8**	BLUE GND	**13**	Horizontal Sync (HSYNC)
4	Monitor ID	**9**	+5V DC output	**14**	Vertical Sync (VSYNC)
5	GND	**10**	SYNC GND	**15**	Monitor ID

โดยในการประยุกต์ใช้บอร์ด FPGA สามารถใช้ GPIO Pin เป็น VGA Adapter ได้ ซึ่งจะได้มีการอธิบายโปรโตคอลที่ใช้ในการส่งข้อมูล ตลอดจนวิธีการเขียน VHDL ในการทำหน้าที่เป็น Graphic Driver ต่อไป

VGA Protocol

สำหรับโปรโตคอลในการส่งข้อมูลผ่านพอร์ต VGA จะแบ่งเป็น 2 ส่วน ได้แก่ Video Signal และ Synchronous Signal ซึ่งมีรายละเอียดดังต่อไปนี้

1. สัญญาณวิดีโอ (Video Signal)

สัญญาณวิดีโอประกอบด้วยสัญญาณสีหลัก 3 ช่อง คือ **Red (R)**, **Green (G)**, และ **Blue (B)** ซึ่งสัญญาณเหล่านี้ส่งออกมาตามลำดับของ Pixel ที่ต้องการแสดงผล

- สัญญาณสีแต่ละช่องเป็นแบบ **Analog** มีระดับแรงดันตั้งแต่ 0V (ไม่มีสี) ถึง 0.7V (สีสว่างสุด) แต่ละ Pixel ถูกสร้างจากค่าระดับแรงดันของทั้ง 3 สีผสมกัน
- การแสดงผลสีต่าง ๆ ขึ้นอยู่กับระดับของสัญญาณสีในแต่ละช่อง ทำให้สามารถแสดงผลได้หลายล้านสีโดยการผสมผสานของสัญญาณ RGB

2. สัญญาณซิงค์ (Synchronous Signal)

เพื่อให้สัญญาณภาพถูกจัดเรียง Pixel ได้อย่างถูกต้อง จำเป็นต้องมี Synchronous Signal 2 ประเภท คือ **Horizontal Sync (HSYNC)** และ **Vertical Sync (VSYNC)** ซึ่งทำหน้าที่กำหนดตำแหน่งในการวาดภาพ

- Horizontal Sync (HSYNC)
    - โดยปกติแล้ว การแสดงผลกราฟิกบนหน้าจอมอนิเตอร์ จะแสดงผลที่ละ Pixel เริ่มจาก Pixel ซ้ายสุด ไปยัง Pixel ขวาสุด (Pixel Row) และจาก Pixel บน ไปยัง Pixel ล่าง ตามลำดับ
    - HSYNC เป็นสัญญาณที่ใช้ในการ Sync แต่ละแถวของภาพ (Pixel Row) โดยในหนึ่งเฟรม (frame) หรือหนึ่งหน้าจอประกอบด้วยหลายแถว ซึ่งแต่ละแถวต้องเริ่มต้นและสิ้นสุดตามเวลาแน่นอน
    - สัญญาณ HSYNC เป็นสัญญาณที่บ่งบอกว่า การวาดภาพในแถว (Row) นั้น ได้สิ้นสุดลงแล้ว และให้เริ่มต้นวาดภาพในแถว (Row) ใหม่
    - ความถี่ (Frequency) ของสัญญาณ HSYNC ขึ้นอยู่กับความละเอียดของจอภาพ โดยจอภาพที่มีความละเอียดสูงขึ้น ก็ต้องใช้ความถี่ของสัญญาณ HSYNC ที่สูงขึ้นเช่นกัน

- Vertical Sync (VSYNC)
  - VSYNC ใช้ในการ Sync เมื่อจอภาพวาดภาพครบทั้งหน้าจอ (หนึ่งเฟรม) เพื่อให้จอภาพรู้ว่าถึงเวลาที่จะเริ่มวาดเฟรมใหม่จากแถว (Row) แรก
  - สัญญาณ VSYNC เป็นสัญญาณที่ทำให้การวาดภาพแต่ละเฟรมเกิดขึ้นต่อเนื่องตาม Refresh Rate ที่ต้องการ เช่น 60Hz หรือ 75Hz

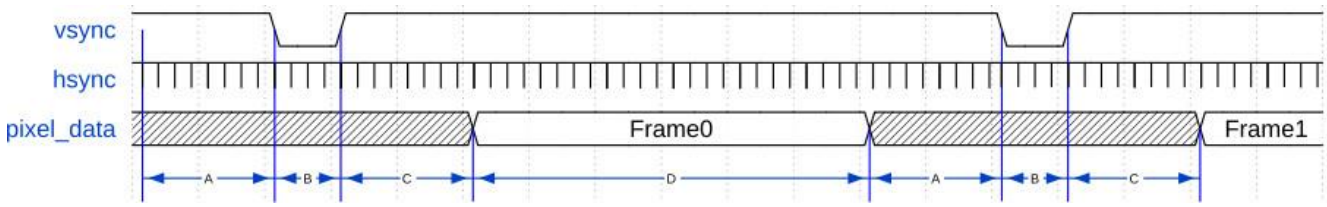
Front Porch และ Back Porch

เป็นช่วงสัญญาณที่ใช้เพื่อการ Sync แนวนอน (Horizontal) และแนวตั้ง (Vertical) โดยที่จะไม่มีการแสดงผลภาพในช่วงนี้ เนื่องจากช่วงสัญญาณนี้ทำหน้าที่เตรียมการสำหรับการเปลี่ยนแปลงจากบรรทัดหนึ่งไปยังอีกบรรทัดหนึ่ง (Horizontal) และจากเฟรมหนึ่งไปยังอีกเฟรมหนึ่ง (Vertical)

- Front Porch
  - เป็นช่วงสัญญาณที่เกิดหลังจากการแสดงผลภาพ (Active Video Area) และก่อนที่สัญญาณซิงค์ (Sync Pulse) จะเริ่มทำงาน
  - ทำหน้าที่ช่วยให้สัญญาณกลับไปยังตำแหน่งแรกของบรรทัดถัดไปก่อนที่ Sync Pulse จะเริ่มต้น
- Back Porch
  - เป็นช่วงสัญญาณที่เกิดหลังจาก Sync Pulse และก่อนการเริ่มต้นแสดงผล Pixel ในบรรทัดใหม่
  - ทำหน้าที่เป็นสัญญาณคั่นเวลาระหว่าง Sync Pulse และการเริ่มต้นแสดงผลของบรรทัดถัดไป

โดยสามารถแสดง Timing Diagram ของสัญญาณ HSYNC และ VSYNC ได้ ดังรูปที่ 5

- A - front porch - 16 pixel clocks
- B - sync pulse - 96 pixel clocks
- C - back porch - 48 pixel clocks
- D - visible line - 640 pixel clocks



รูปที่ 5 Timing Diagram ของ VGA Synchronous Signal

ที่มา https://gregchadwick.co.uk/blog/playing-with-the-pico-pt5/

VGA Timing

ในการแสดงผลภาพผ่าน VGA ต้องมีการใช้สัญญาณนาฬิกา ซึ่งความถี่ของสัญญาณนาฬิกาที่ใช้ในการขับ (Drive) แต่ละ Pixel จะแตกต่างกันไป ขึ้นอยู่กับความละเอียด (Resolution) และอัตราการรีเฟรช (Refresh Rate) ที่ต้องการ ซึ่งสามารถแสดงเป็นตารางได้ดังนี้

Format	Pixel Clock (MHz)	Horizontal (Pixel)				Vertical (Line)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29
1024x768, 70Hz	75.000	1024	24	136	144	768	3	6	29
1024x768, 75Hz	78.750	1024	16	96	176	768	1	3	28
1024x768, 85Hz	94.500	1024	48	96	208	768	1	3	36

Source: Rick Ballantyne, Xilinx Inc.

สำหรับโครงงาน FPGA Battleship จะเลือกใช้ความละเอียดในการแสดงผลที่ 640 x 480 Pixel โดยมีอัตราการรีเฟรช 60 Hz ดังนั้น จะใช้สัญญาณนาฬิกาในการขับ (Drive) VGA Adapter ที่ประมาณ 25 MHz และมี Parameter ที่ใช้ในการขับดังแสดงในตารางข้างต้น

3. Communication Protocol

โดยในโครงงานนี้ จะเลือกใช้ Communication Protocol การส่งข้อมูลโดยตรงผ่านสายแพแบบ **PIPO (Parallel In, Parallel Out)** คือวิธีการสื่อสารที่ส่งข้อมูลหลายบิตพร้อมกัน โดยส่งผ่านหลายสายสัญญาณในเวลาเดียวกัน

PIPO (Parallel In, Parallel Out)

เป็นรูปแบบการถ่ายโอนข้อมูลแบบขนาน ซึ่งข้อมูลทุกบิตของข้อมูลจะถูกส่งเข้าและส่งออกในครั้งเดียวพร้อมกันผ่านหลายสายสัญญาณ จึงทำให้สามารถถ่ายโอนข้อมูลได้รวดเร็วขึ้นมากเมื่อเทียบกับการส่งข้อมูลแบบอนุกรม โดยเฉพาะอย่างยิ่งในระบบที่ต้องการความเร็วสูงหรือที่ต้องการถ่ายโอนข้อมูลปริมาณมาก

หลักการทำงานของ Parallel-in Parallel-out

- การรับข้อมูลขนาน (Parallel Input)
  - ข้อมูลที่ต้องการรับเข้าในแบบขนาน (เช่น 8 บิต) จะถูกป้อนเข้าผ่านสายสัญญาณที่แยกกัน 8 เส้น ซึ่งจะนำข้อมูลแต่ละบิตเข้าสู่บัสขนานพร้อมกัน
- การเก็บข้อมูลในวงจรภายใน
  - ทุกบิตของข้อมูลที่รับเข้ามาในสัญญาณนาฬิกาเดียวกันจะถูกเก็บพร้อมกัน ทำให้ข้อมูลทั้งหมดที่ส่งเข้ามามีความสอดคล้องกันในทุกบิต
- การส่งข้อมูลขนาน (Parallel Output)
  - การส่งข้อมูลแบบขนานจะช่วยลดจำนวนรอบสัญญาณนาฬิกาที่ต้องใช้ในการส่งข้อมูลแต่ละบิตออกไป ทำให้ข้อมูลสามารถส่งออกในเวลาที่รวดเร็ว
- การทำงานในรอบสัญญาณนาฬิกาเดียว
  - PIPO ทำงานได้ในรอบสัญญาณนาฬิกาเดียว ทั้งในขั้นตอนการรับเข้าและการส่งออก เนื่องจากข้อมูลทุกบิตถูกประมวลผลพร้อมกัน

ตัวอย่างการทำงานของ Parallel-in Parallel-out

ในการส่งข้อมูลแบบ Parallel-in Parallel-out จะเริ่มต้นจากการอ่าน (Read) ข้อมูลจาก Register (Memory Unit) ของผู้ส่ง (Sender) แล้วส่งข้อมูลที่ต้องการผ่านพอร์ต GPIO อย่างไรก็ตาม เนื่องจากยังไม่มี

สัญญาณ Enable ทำให้ผู้รับ (Receiver) ยังไม่ได้เขียนข้อมูลที่ได้รับลงใน Memory ของตนเอง แต่จะเขียนข้อมูลลงใน Memory เมื่อมีสัญญาณ Enable เท่านั้น

เช่น หากต้องการส่งข้อมูล AttackCol(3:0) ขนาด 4 บิต ระหว่าง FPGA สองบอร์ด สามารถทำได้โดยการกำหนด GPIO Pin เป็น TX และ RX สำหรับส่งและรับข้อมูลตามลำดับ ซึ่งเนื่องจากเป็นการส่งข้อมูลแบบ Parallel จำนวน Pin จึงต้องสอดคล้องกับจำนวนบิตของข้อมูลที่ต้องการส่ง เมื่อได้รับสัญญาณ Enable จึงอ่านเขียนข้อมูลลงใน Memory หรือนำข้อมูลที่ได้ไปประมวลผลต่อไป

รูปที่ 6 ตัวอย่างการรับส่งข้อมูลระหว่าง FPGA แบบ Parallel

สำหรับในโครงงาน FPGA Battleship มีการรับส่งข้อมูลระหว่างบอร์ด FPGA จำนวนมาก เช่น ตำแหน่งของเรือที่ต้องการโจมตี, ผลลัพธ์จากการโจมตี เป็นต้น จึงเลือกใช้การส่งข้อมูลแบบ Parallel เพื่อให้ข้อมูลทั้งหมดถูกส่งถึงพร้อมกัน นอกจากนั้น ยังช่วยลดความยุ่งยากในการ Implement ระบบในการส่งข้อมูลแบบ Serial ได้อีกด้วย

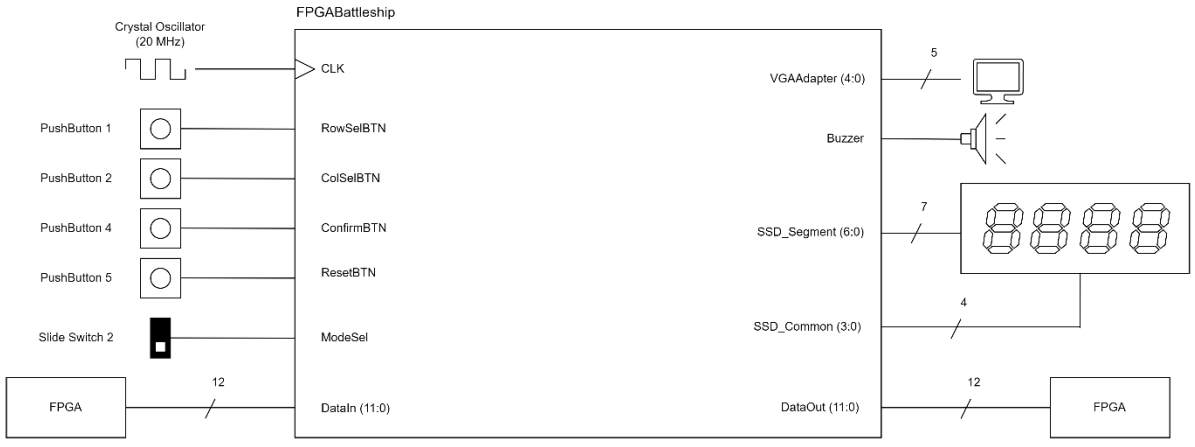
กระบวนการออกแบบ | Design Process

Top-Down Design

ในกระบวนการออกแบบ ผู้จัดทำเริ่มต้นโดยใช้กระบวนการ Top-Down Design โดยกำหนดภาพรวมของระบบว่าต้องมี Input-Output อะไรบ้าง ซึ่งสามารถสรุปเป็นตารางได้ดังนี้

Input	Output
- Push Button สำหรับควบคุมตำแหน่ง Cursor ของเกม จำนวน 2 ปุ่ม (Row และ Column)	- VGA Adapter
- Push Button สำหรับ Confirm ตำแหน่งใช้ในการวางเรือและโจมตีเรือฝ่ายตรงข้าม	- Seven Segments Display
- Push Button สำหรับ Reset Memory	- Buzzer
- Slide Switch สำหรับเปลี่ยนโหมดการเล่น	
- Clock	
Communication Chanel (DataIn & DataOut)	

ซึ่งสามารถเขียน Top Level Module ได้ดังภาพ



รูปที่ 7 Top-Down Design (Top Level Module) ของโครงงาน

จากนั้นจึงค่อย ๆ ทำการ Break Down และพัฒนาโมดูลต่าง ๆ ทีละ Layer ตามกระบวนการ Top Down Design ต่อไป

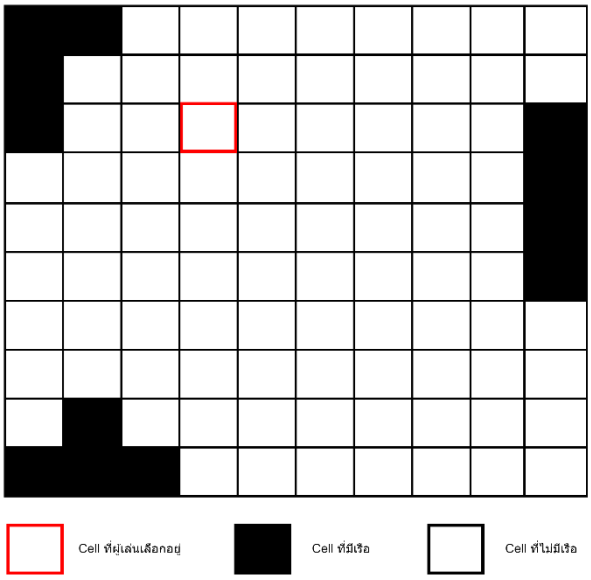
ในการพัฒนาโมดูล จะพิจารณาถึงความละเอียดและซับซ้อนของโมดูล หากเป็นโมดูลที่มีรายละเอียดมาก เช่น Memory Unit หรือ VGA Adapter จะใช้ VHDL ในการพัฒนา แต่หากเป็นวงจรที่ไม่ซับซ้อนมาก เช่น วงจร Multiplexer อาจพิจารณาใช้การต่อวงจรแบบ Schematics

Graphic User Interface (GUI)

ในการออกแบบหน้าต่าง Graphic User Interface จะอ้างอิงจากเกม Sea Battleship แบบดั้งเดิม คือ

- โหมดวางเรือ (Editing Mode)

  มีพื้นที่ตาราง Grid ขนาด 10 × 10 ช่อง และมี Cursor ให้ผู้ใช้สามารถเลือกตำแหน่งในการวางเรือได้ ดังภาพ

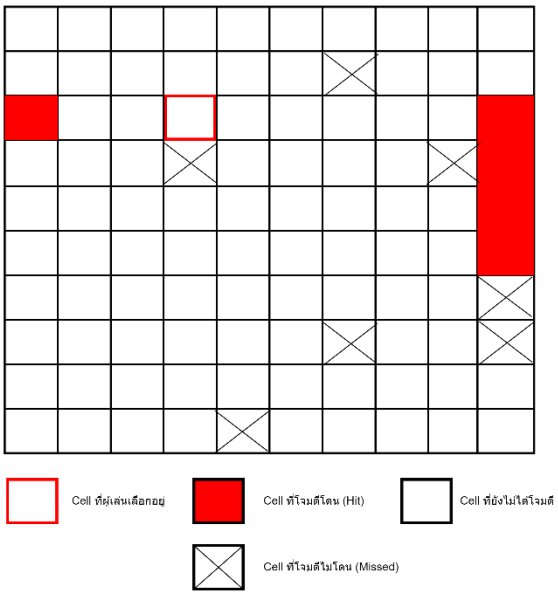


รูปที่ 8 หน้าต่าง Graphic User Interface (GUI) ในโหมดวางเรือ (Editing Mode)

ในโหมดวางเรือ (Editing Mode) นี้ ผู้ใช้จะใช้ปุ่ม Push Button 1 และ Push Button 2 ในการเลื่อนตำแหน่งของ Cursor สีแดงไปตามตาราง Grid เพื่อเลือกตำแหน่ง Row และ Column ที่ต้องการวางเรือ หากผู้เล่นได้ตำแหน่งที่ต้องการวางเรือแล้ว จะกดปุ่ม Push Button 4 เพื่อยืนยันตำแหน่ง อย่างไรก็ตาม หากตำแหน่งนั้นมีเรือวางอยู่แล้ว แต่ผู้เล่นวางเรือซ้ำ จะเป็นการนำเรือในตำแหน่งนั้นออกแทน ผู้เล่นสามารถวางเรือได้สูงสุด 20 ลำ หากผู้เล่นพยายามวางเรือเกินจำนวนที่กำหนด จะไม่สามารถวางได้ และมีเสียง Buzzer เพื่อเตือนผู้เล่น

- โหมดโจมตี (Attack Mode)

มีพื้นที่ตาราง Grid ขนาด 10 × 10 ช่อง และมี Cursor ให้ผู้ใช้สามารถเลือกตำแหน่งในการยิงเรือของฝ่ายตรงข้ามได้ ดังภาพ



รูปที่ 9 หน้าต่าง Graphic User Interface (GUI) ในโหมดโจมตี (Attack Mode)

โดยในโหมดโจมตี (Attack Mode) หลังจากผู้เล่น เลือกตำแหน่งที่ต้องการโจมตีด้วยปุ่ม Push Button 1 และ Push Button 2 แล้ว สามารถกดปุ่ม Push Button 4 เพื่อทำการโจมตีเรือฝ่ายตรงข้ามได้ โดยหลังจากโจมตีเรือฝ่ายตรงข้ามเสร็จสิ้น Cell ที่ผู้เล่นโจมตี จะเปลี่ยนสถานะตามผลลัพธ์ของการโจมตี ดังแสดงในรูปที่ 9

ผู้เล่นไม่สามารถโจมตีซ้ำในตำแหน่งที่โจมตีไปแล้วได้ หากผู้เล่นพยายามโจมตีซ้ำ จะไม่สามารถโจมตีได้ และมีเสียง Buzzer เตือน

นอกจากนั้น จะมีการแสดงจำนวนเรือที่ผู้เล่นโจมตีโดนบน Seven Segment Display เพื่อให้ผู้เล่นทราบสถานะของเกมอีกด้วย

เนื่องจากเป็นการพัฒนามาจากบอร์ดเกมแบบดั้งเดิม ดังนั้น จึงให้อิสระแก่ผู้เล่น โดยผู้เล่นทั้งสองสามารถตกลงกันว่าจะวางเรือกี่ลำก็ได้ (แต่ไม่เกิน 20 ลำ) หรือมีข้อกำหนดหรือรูปแบบการวางอย่างไรก็ได้ ขึ้นอยู่กับกลยุทธ์ของผู้เล่นทั้งสอง

Communication

ในการส่งข้อมูลหากันระหว่างบอร์ด FPGA 2 บอร์ด จะใช้การส่งข้อมูลโดยตรงแบบ PIPO (Parallel In, Parallel Out) ซึ่งจะใช้สายส่งข้อมูลทั้งหมดจำนวน 24 เส้น ทำให้มีการใช้พินบนบอร์ด FPGA ดังนี้

K1 CONNECTOR	
FPGA	**Description**
P5	ใช้ส่งสัญญาณการโจมตีไปยังฝ่ายตรงข้าม
P7	ใช้รับสัญญาณการโจมตีที่มาจากฝ่ายตรงข้าม
P9	ใช้เพื่อส่งสัญญาณว่ามีการโจมตีเข้ามาจากฝ่ายตรงข้าม
P11	ใช้รับข้อมูลว่ามีการโจมตีเข้ามาจากฝ่ายตรงข้าม
P14 P16	ใช้สำหรับส่ง Flag เพื่อตอบกลับว่าโจมตีโดนเรือหรือไม่
P21 P23	ใช้สำหรับรับ Flag การโจมตีจากฝ่ายตรงข้าม

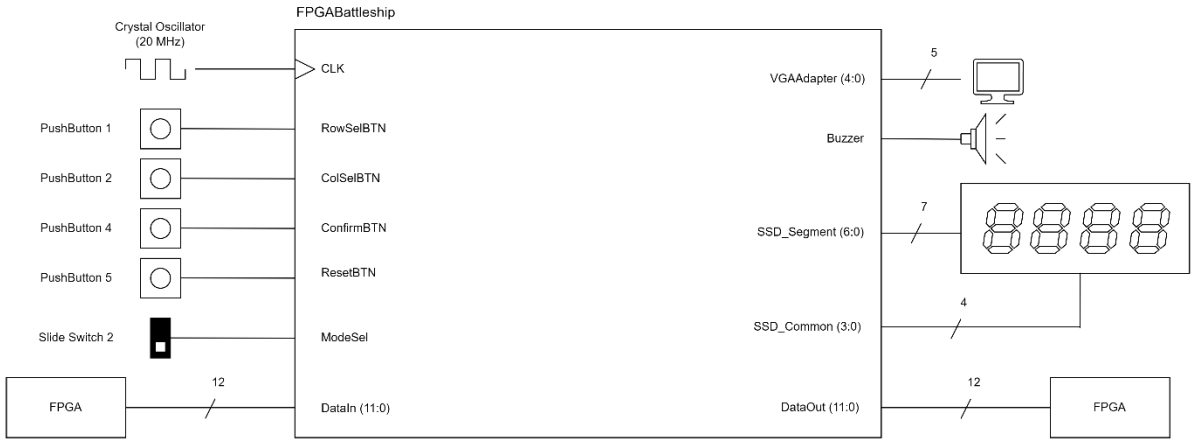
K2 CONNECTOR	
FPGA	**Description**
P6 P8 P10 P12	ใช้สำหรับส่งพิกัดการโจมตีในแนวนอน Row ไปยังฝ่ายตรงข้าม
P15 P17 P22 P24	ใช้สำหรับรับพิกัดการโจมตีในแนวนอน Row ที่มาจากฝ่ายตรงข้าม

K3 CONNECTOR	
FPGA	**Description**
P124 P127 P132 P134	ใช้สำหรับส่งพิกัดการโจมตีในแนวตั้ง Column ไปยังฝ่ายตรงข้าม
P138 P140 P142 P1	ใช้สำหรับรับพิกัดการโจมตีในแนวตั้ง Column ไปยังฝ่ายตรงข้าม

กระบวนการพัฒนา | Development Process

ในหัวข้อนี้ จะอธิบายกระบวนการพัฒนาโครงงาน โดยจะอธิบายไปตาม Top Down Design ที่ได้ออกแบบ และจะ Break Down รายละเอียดแต่ละโมดูลที่ใช้ในโครงงาน ทั้งโมดูลที่พัฒนาด้วย VHDL และโมดูลที่ได้จากการต่อวงจรแบบ Schematics

Top Level Module (1st Layer)



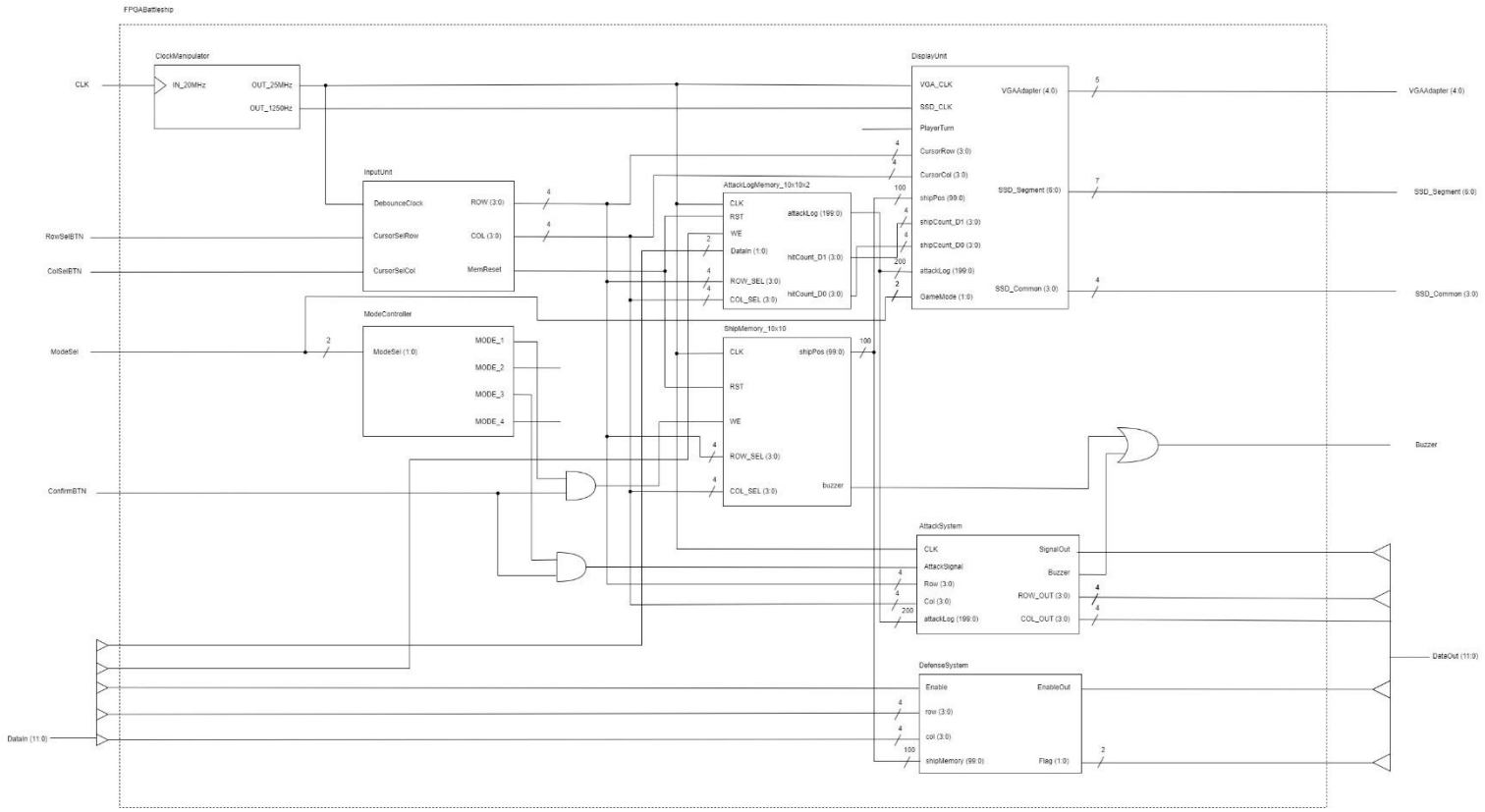
รูปที่ 10 Top-Down Design (Top Level Module) ของโครงงาน

สำหรับ Top Level Module (1st Layer) ได้กำหนดขอบเขต Input และ Output ของระบบไว้กว้าง ๆ ประกอบด้วย

Port	Description
Input Ports	
CLK	สัญญาณนาฬิกาความถี่ 20 MHz จาก Built-in Crystal Oscillator
RowSelBTN	เชื่อมต่อกับปุ่ม Push Button เพื่อใช้เลือกตำแหน่งของ Cursor
ColSelBTN	
ConfirmBTN	เชื่อมต่อกับปุ่ม Push Button เพื่อใช้ยืนยัน Action ต่าง ๆ ของผู้เล่น
ModeSel	เชื่อมต่อกับ Slide Switch เพื่อสลับ Mode ในการเล่น ระหว่าง Editing Mode และ Attack Mode
DataIn (11:0)	เชื่อมต่อกับ FPGA อีกบอร์ดหนึ่งเพื่อแลกเปลี่ยนข้อมูลที่จำเป็น ซึ่งได้อธิบายแล้วในกระบวนการออกแบบ (Design Process)

Port	Description
Output Ports	
VGAAdapter (4:0)	เชื่อมต่อกับ VGA Connector โดยมี Bus Width = 5 Bits ใช้สำหรับ RED Video, GREEN Video, BLUE Video, HSYNC และ VSYNC
Buzzer	เชื่อมต่อกับ Buzzer บนบอร์ด เพื่อเป็นสัญญาณเตือนต่าง ๆ ให้ผู้เล่น
SSD_Segment (6:0)	เชื่อมต่อกับ Segment ของ Seven Segment Display เพื่อแสดงผลจำนวนเรือที่ผู้เล่นยิงโดน
SSD_Common (3:0)	เชื่อมต่อกับ Common ของ Seven Segment Display เพื่อแสดงผลจำนวนเรือที่ผู้เล่นยิงโดน
DataOut (11:0)	เชื่อมต่อกับ FPGA อีกบอร์ดหนึ่งเพื่อแลกเปลี่ยนข้อมูลที่จำเป็น ซึ่งได้อธิบายแล้วในกระบวนการออกแบบ (Design Process)

FPGABattleship (2nd Layer)



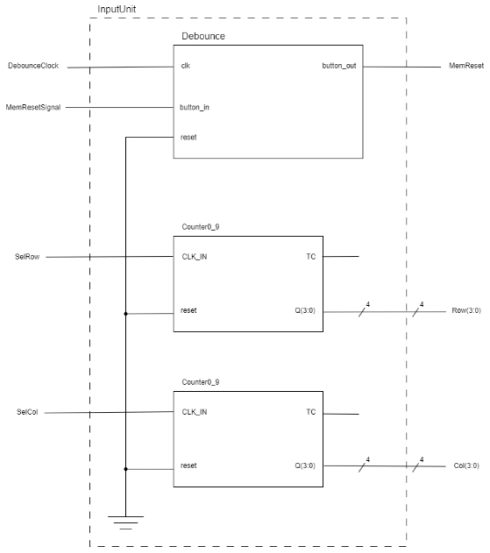
รูปที่ 11 โมดูล FPGABattleship (2nd Layer) ของโครงงาน

ภายในโมดูล FPGABattleship (2nd Layer) มีโมดูลส่วนประกอบและรายละเอียดดังต่อไปนี้

Module	Description
ClockManipulator	ทำหน้าที่แปลงสัญญาณนาฬิกาจาก Built-in Crystal Oscillator (20MHz) เป็นสัญญาณนาฬิกาความถี่ 25 MHz และ 1250 Hz เพื่อใช้ขับ VGA และ Seven Segment Display ตามลำดับ
InputUnit	ทำหน้าที่รับ Input จากผู้ใช้ และแปลงให้เป็นตำแหน่ง Row และ Column ของ Cursor นอกจากนั้นยัง Debounce ปุ่ม Push Button 5 ด้วย
ModeController	ทำหน้าที่เป็นวงจร 2_4_Decoder เพื่อเลือกโหมดการเล่นเกม
ShipMemory_10x10	ทำหน้าที่เปรียบเสมือนหน่วยความจำ (Memory) ที่ใช้เก็บตำแหน่งการวางเรือของผู้เล่น โดยภายในเป็น Matrix ขนาด 10x10x1
AttackLogMemory_10x10x2	ทำหน้าที่เปรียบเสมือนหน่วยความจำ (Memory) ที่ใช้เก็บประวัติและผลลัพธ์ในการโจมตีเรือฝ่ายตรงข้าม ภายในเป็น Matrix ขนาด 10x10x2
DisplayUnit	ทำหน้าที่เป็น Graphic Driver โดยอ่านข้อมูลจาก Memory แล้วนำไปแสดงผลผ่าน VGA Adapter โดยภายในมีวงจร Multiplexer ที่ทำหน้าที่เลือกฉากที่ถูกต้องกับโหมดของเกมไปแสดงผล
AttackSystem	ทำหน้าที่เป็นผู้ส่งสัญญาณโจมตี ตลอดจนตำแหน่งที่ผู้เล่นโจมตีเรือฝ่ายตรงข้ามผ่าน GPIO Pin เพื่อให้บอร์ด FPGA เป้าหมายทำการประมวลผลต่อไป
DefenseSystem	ทำหน้าที่รับสัญญาณการโจมตีและตำแหน่งที่ทำการโจมตีจาก FPGA อีกบอร์ดหนึ่งมาประมวลผล และคืนค่าผลลัพธ์ว่าโจมตีสำเร็จหรือไม่ให้บอร์ด FPGA ผู้โจมตีประมวลผลต่อไป

Design Document ฉบับนี้ จะได้แสดงรายละเอียดโมดูลที่สำคัญ และจะขอละเว้นการลงรายละเอียดในโมดูลพื้นฐาน เช่น Multiplexer, Decoder, Seven Segment Display Driver ไว้ในฐานที่เข้าใจ

InputUnit (3rd Layer)



รูปที่ 12 โมดูล InputUnit (3rd Layer) ของโครงงาน

โมดูล InputUnit ทำหน้าที่รับและประมวลผล Input จากผู้ใช้ ซึ่งประกอบไปด้วยการกดปุ่ม Push Button 1 และ Push Button 2 เพื่อเลื่อนตำแหน่งของ Cursor และการกดปุ่ม Push Button 5 เพื่อ Reset Memory

เนื่องจากตาราง Grid มีขนาด 10 × 10 ในการเลื่อน Cursor จึงได้ใช้วงจร mod-10 counter เป็นวงจรในการนับตำแหน่งของ Cursor โดยให้ User Input (Push Button 1 และ Push Button 2) เป็นสัญญาณ Clock และให้ Output เป็น BCD (Binary Coded Decimal) เพื่อนำไปแสดงผลโดยโมดูล DisplayUnit ต่อไป

สำหรับการกดปุ่ม Push Button 5 ต้องทำการ Debounce ปุ่มก่อน เนื่องจากไม่ได้มีการ Pre-Debounce ซึ่งหากไม่ทำการ Debounce อาจทำให้พฤติกรรมของปุ่มไม่ถูกต้องได้ สำหรับโครงงานนี้ ได้เขียน VHDL สำหรับ Debounce ปุ่ม Push Button ดังต่อไปนี้

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Debounce is
    Port (
        clk         : in std_logic;      -- System clock
        reset       : in std_logic;      -- Reset signal
        button_in   : in std_logic;      -- Raw button input
        button_out  : out std_logic      -- Debounced button output
    );
end Debounce;
```

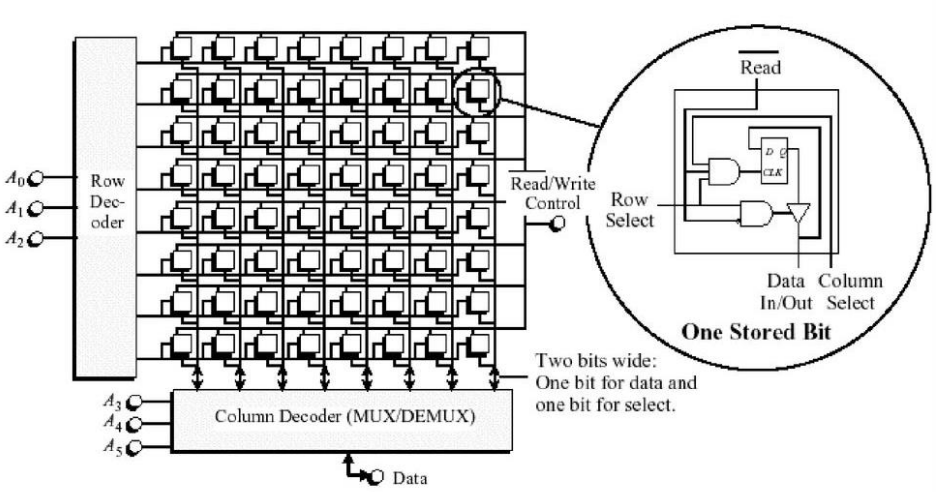
```
architecture Behavioral of Debounce is
    constant debounce_time : integer := 500000;

    signal counter      : integer range 0 to debounce_time := 0;
    signal stable_state : std_logic := '0';
    signal button_sync  : std_logic := '0';
begin
    -- Synchronize button input to avoid metastability
    process(clk, reset)
    begin
        if reset = '1' then
            button_sync <= '0';
        elsif rising_edge(clk) then
            button_sync <= button_in;
        end if;
    end process;

    -- Debounce process
    process(clk, reset)
    begin
        if reset = '1' then
            counter <= 0;
            stable_state <= '0';
            button_out <= '0';
        elsif rising_edge(clk) then
            if button_sync = stable_state then
                counter <= 0;
            else
                if counter < debounce_time then
                    counter <= counter + 1;
                else
                    stable_state <= button_sync;
                    button_out <= button_sync;
                    counter <= 0;
                end if;
            end if;
        end if;
    end process;

end Behavioral;
```

โดยเมื่อสัญญาณจาก Push Button 5 ผ่านการ Debouce เรียบร้อยแล้ว จะส่งสถานะ State ของปุ่มออกทางพอร์ต button_out สำหรับใช้งานต่อไป

ShipMemory_10x10 (3rd Layer)

โมดูล ShipMemory_10x10 เป็นโมดูลที่ทำหน้าที่เปรียบเสมือน Memory คือ ใช้ในการเก็บข้อมูลตำแหน่งของเรือที่ผู้เล่นวางลงบนตาราง Grid ซึ่งหากต้องการ Implement โดยใช้โครงสร้างแบบ Memory Matrix ด้วยการต่อวงจรแบบ Schematics นั้น จำเป็นต้องต่อวงจรที่ซับซ้อนอย่างมาก (อาจต้องใช้ Flip-Flops จำนวน 100 ตัว สำหรับเก็บข้อมูล 100 บิต) ดังนั้น สำหรับการสร้างโมดูล ShipMemory_10x10 จึงใช้ VHDL ในการสร้าง

โดยสำหรับการทำ Memory ในโครงงานนี้ จะมีลักษณะคล้าย Memory Matrix ของจริง คือ จะมี Address Bus (ในกรณีนี้ คือ Row และ Column) DataIn และ Write Enable (WE) อย่างก็ตาม จะไม่ได้ทำระบบ Read Enable เนื่องจากจะให้ Memory ส่งค่าออกทุกรอบสัญญาณนาฬิกาอยู่แล้ว



รูปที่ 13 โครงสร้างของ 2D Memory ที่จะ Implement ในการจัดเก็บตำแหน่งของเรือที่ผู้เล่นวาง

ที่มา https://slideplayer.com/slide/17338485/

ซึ่งสามารถจำลองโครงสร้าง Memory ดังกล่าวใน VHDL ได้ โดยใช้ std_logic_vector(99 down to 0) เมื่อมีเรือวางอยู่ ให้ข้อมูลใน Address นั้นเป็น ‘1’ ในทางกลับกัน หากไม่มีเรือใน Address นั้น ให้เป็น ‘0’ แสดงได้ด้วย VHDL Code ดังนี้

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ShipMemory_10x10 is
    Port (
        clk             : in std_logic;
        rst             : in std_logic;
        row_sel         : in std_logic_vector(3 downto 0);
        col_sel         : in std_logic_vector(3 downto 0);
        write_enable    : in std_logic;
        shipPos         : out std_logic_vector(99 downto 0);
        maxShipLimit    : out std_logic
    );
end ShipMemory_10x10;
```

```
architecture Behavioral of ShipMemory_10x10 is
    -- Memory array to represent the 10x10 grid (each cell is 1 bit)
    signal memory : std_logic_vector(99 downto 0) := (others => '0');

    signal cell_address : integer range 0 to 99;
    constant max_ships : integer := 20;  -- Max number of ships allowed
    signal shipCount : integer := 0;  -- Current number of ships placed

    -- Buzzer control signals
    signal maxShipLimit_int : std_logic := '0';  -- Internal buzzer signal
    signal buzzer_timer : integer := 0;   -- Timer for 100ms buzzer signal
    constant buzzer_duration : integer := 2500000; -- 100ms at 25MHz clock

begin
    cell_address <= to_integer(unsigned(row_sel)) * 10 +
to_integer(unsigned(col_sel));

    process(clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                memory <= (others => '0');
                shipCount <= 0;  -- Reset ship count on reset
                maxShipLimit_int <= '0';
                buzzer_timer <= 0;
            elsif falling_edge(write_enable) then
                if memory(cell_address) = '0' then  -- Place a ship
                    if shipCount < max_ships then
                        memory(cell_address) <= '1';
                        shipCount <= shipCount + 1;
                    end if;
                elsif memory(cell_address) = '1' then  -- Remove a ship
                    memory(cell_address) <= '0';
                    shipCount <= shipCount - 1;
                end if;

                if shipCount >= max_ships then
                    maxShipLimit_int <= '1';
                    buzzer_timer <= buzzer_duration;
                end if;
            end if;

            -- Handle buzzer timing
            if buzzer_timer > 0 then
                buzzer_timer <= buzzer_timer - 1;
                if buzzer_timer = 1 then
                    maxShipLimit_int <= '0';
                end if;
            end if;
        end if;
    end process;
    shipPos <= memory;

    maxShipLimit <= maxShipLimit_int;

end Behavioral;
```

เมื่อจำนวนเรือที่ผู้เล่นวางถึงจำนวนเรือสูงสุดที่สามารถวางได้ (maxShipLimit) จะมีการส่งสัญญาณเตือนผู้เล่นผ่านพอร์ต maxShipLimit: out std_logic ซึ่งจะนำไปใช้ในการสั่ง Buzzer ต่อไป

AttackLogMemory_10x10x2 (3rd Layer)

โมดูล AttackLogMemory_10x10x2 เป็นโมดูลสำหรับใช้เป็น Memory เช่นเดียวกันกับ ShipMemory ในขณะที่ Data Cell ของ ShipMemory_10x10 จะเก็บข้อมูลเพียง 1 บิต (‘0’ เมื่อไม่มีเรือวางอยู่ ‘1’ เมื่อมีเรือวาง) แต่โมดูล AttackLogMemory_10x10x2 จะเก็บข้อมูลจำนวน 2 บิต ต่อ 1 Memory Cell ซึ่งแสดงสถานะของการโจมตีเรือฝ่ายตรงข้ามดังต่อไปนี้

ข้อมูล	สถานะ
‘00’	ยังไม่ได้ยิงเรือในตำแหน่งนั้น
‘01’	ยิงแล้ว ไม่มีเรือฝ่ายตรงข้ามในตำแหน่งนั้น (Missed)
‘10’	ยิงแล้ว โดนเรือฝ่ายตรงข้าม (Hit)
‘11’	Invalid Status

โดยค่าสถานะของแต่ละ Memory Cell จะถูกส่งต่อไปยัง DisplayUnit เพื่อแสดงผล Cell นั้น ๆ ให้สอดคล้องกับข้อมูลการโจมตีเรือต่อไป

เช่นเดียวกับ ShipMemory_10x10 ในการ Implement โมดูล Memory ขนาดใหญ่ด้วยวิธี Schematics มีความยุ่งยากและซับซ้อนเป็นอย่างมาก จึงได้ใช้ VHDL ในการสร้างโมดูล โดยสามารถแสดง VHDL Code ได้ดังนี้

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity AttackLogMemory_10x10x2 is
    Port (
        clk             : in  std_logic;
        rst             : in  std_logic;
        write_en        : in  std_logic;
        row             : in  std_logic_vector(3 downto 0);
        col             : in  std_logic_vector(3 downto 0);
        data_in         : in  std_logic_vector(1 downto 0);
        data_out        : out std_logic_vector(199 downto 0);
        hit_count_tens  : out std_logic_vector(3 downto 0);
        hit_count_units : out std_logic_vector(3 downto 0)
    );
end AttackLogMemory_10x10x2;

architecture Behavioral of AttackLogMemory_10x10x2 is
    -- Define a 10x10 memory matrix, where each cell has 2 bits
    type memory_type is array (0 to 9, 0 to 9) of std_logic_vector(1 downto
0);
    signal memory : memory_type := (others => (others => "00"));
```

```
    -- Signal to hold the hit count
    signal hit_count : integer := 0;
begin

    -- Write process to handle writing to memory and reset functionality
    process (clk)
        variable row_index : integer;
        variable col_index : integer;
    begin
        if rising_edge(clk) then
            if rst = '1' then
                -- Reset all cells to "00" and clear hit count
                for i in 0 to 9 loop
                    for j in 0 to 9 loop
                        memory(i, j) <= "00";
                    end loop;
                end loop;
                hit_count <= 0;
            elsif write_en = '1' then
                -- Convert row and col (4-bit vectors) to integer indices
                row_index := to_integer(unsigned(row));
                col_index := to_integer(unsigned(col));

                -- Ensure row and col are within bounds (0 to 9)
                if row_index >= 0 and row_index <= 9 and col_index >= 0 and
col_index <= 9 then
                    if memory(row_index, col_index) = "10" and data_in /=
"10" then
                        hit_count <= hit_count - 1;
                    elsif memory(row_index, col_index) /= "10" and data_in
= "10" then
                        hit_count <= hit_count + 1;
                    end if;

                    -- Update the memory cell
                    memory(row_index, col_index) <= data_in;
                end if;
            end if;
        end if;
    end process;

    -- Data output process to read all memory cells
    process (memory)
        variable temp_out : std_logic_vector(199 downto 0);
    begin
        -- Convert 2D memory array to 1D output vector (flattening)
        for i in 0 to 9 loop
            for j in 0 to 9 loop
            temp_out((i*10 + j)*2 + 1 downto (i*10 + j)*2) := memory(i, j);
            end loop;
        end loop;
        data_out <= temp_out;
    end process;
```

```
    -- BCD conversion process for the hit count
    process(hit_count)
        variable tens_place : integer;
        variable units_place : integer;
    begin
        -- Calculate tens and units places
        tens_place := hit_count / 10;
        units_place := hit_count mod 10;

        -- Convert to BCD and assign to output ports
        hit_count_tens <= std_logic_vector(to_unsigned(tens_place, 4));
        hit_count_units <= std_logic_vector(to_unsigned(units_place, 4));
    end process;

end Behavioral;
```

สังเกตว่า โมดูล AttackLogMemory_10x10x2 จะให้ Output เป็น BCD ของจำนวนเรือที่ผู้เล่นยิงโดน (hitCount) เพื่อนำไปแสดงผลบน Seven Segment Display ใน DisplayUnit ต่อไปด้วย

AttackSystem (3rd Layer)

AttackSystem เป็นโมดูลที่ทำหน้าที่ส่งข้อมูลการโจมตีของผู้เล่น (คือ ตำแหน่งที่ผู้เล่นต้องการโจมตี) ไปยังบอร์ด FPGA อีกบอร์ดหนึ่ง โดยจะมีการตรวจสอบจาก AttackLogMemory_10x10x2 ก่อน ว่าตำแหน่งที่ผู้เล่นต้องการโจมตีนั้น เคยถูกโจมตีมาก่อนแล้วหรือยัง หากเป็นตำแหน่งที่ผู้เล่นเคยโจมตีแล้ว จะไม่สามารถโจมตีได้ และจะส่งสัญญาณไปสั่งให้ Buzzer ดังเพื่อแจ้งเตือนผู้เล่น

โมดูล AttackSystem เขียนด้วย VHDL Code ดังนี้

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity AttackSystem is
    Port (
        row             : in  std_logic_vector(3 downto 0);
        col             : in  std_logic_vector(3 downto 0);
        attackSignal    : in  std_logic;
        clk             : in  std_logic;
        attackLog       : in  std_logic_vector(199 downto 0);
        row_out         : out std_logic_vector(3 downto 0);
        col_out         : out std_logic_vector(3 downto 0);
        attackSignal_out : out std_logic;
        buzzer          : out std_logic
    );
end AttackSystem;

architecture Behavioral of AttackSystem is
    signal button_reg       : std_logic := '0';
    signal attack_detected  : std_logic_vector(1 downto 0);
    signal buzzer_active    : std_logic := '0';
    signal buzzer_timer     : integer := 0;
    signal pulse_count      : integer := 0;
```

```
    constant BUZZER_DURATION  : integer := 2500000;
    constant DELAY_DURATION   : integer := 2500000;
begin

    -- Attack processing and buzzer control
    process(clk)
    begin
        if rising_edge(clk) then
            -- Determine the attacked cell's previous state from attackLog
            attack_detected <= attackLog((to_integer(unsigned(row)) * 10 +
to_integer(unsigned(col))) * 2 + 1
                                         downto (to_integer(unsigned(row))
* 10 + to_integer(unsigned(col))) * 2);

            -- Detect button press and handle attack logic
            if attackSignal = '1' and button_reg = '0' then
                if attack_detected = "01" or attack_detected = "10" then
  -- Cell already attacked (missed or hit), start the buzzer timer
                    buzzer_active <= '1';
                    buzzer_timer <= BUZZER_DURATION;
                    pulse_count <= 2;
                    attackSignal_out <= '0';
                else
                    -- Cell not yet attacked, proceed with attack
                    row_out <= row;           -- Pass row data to output
                    col_out <= col;           -- Pass col data to output
                    attackSignal_out <= '1';  -- Set enable signal high
                end if;
            else
                attackSignal_out <= '0';
            end if;

            -- Buzzer timing control for two pulses
            if buzzer_active = '1' then
                if buzzer_timer > 0 then
                    buzzer <= '1';
                    buzzer_timer <= buzzer_timer - 1;
                elsif pulse_count > 1 then
                    buzzer <= '0';
                    buzzer_timer <= DELAY_DURATION;
                    pulse_count <= pulse_count - 1;
                else
                    buzzer <= '0';
                    buzzer_active <= '0';
                end if;
            end if;

            -- Update the button state for edge detection
            button_reg <= attackSignal;
        end if;
    end process;

end Behavioral;
```

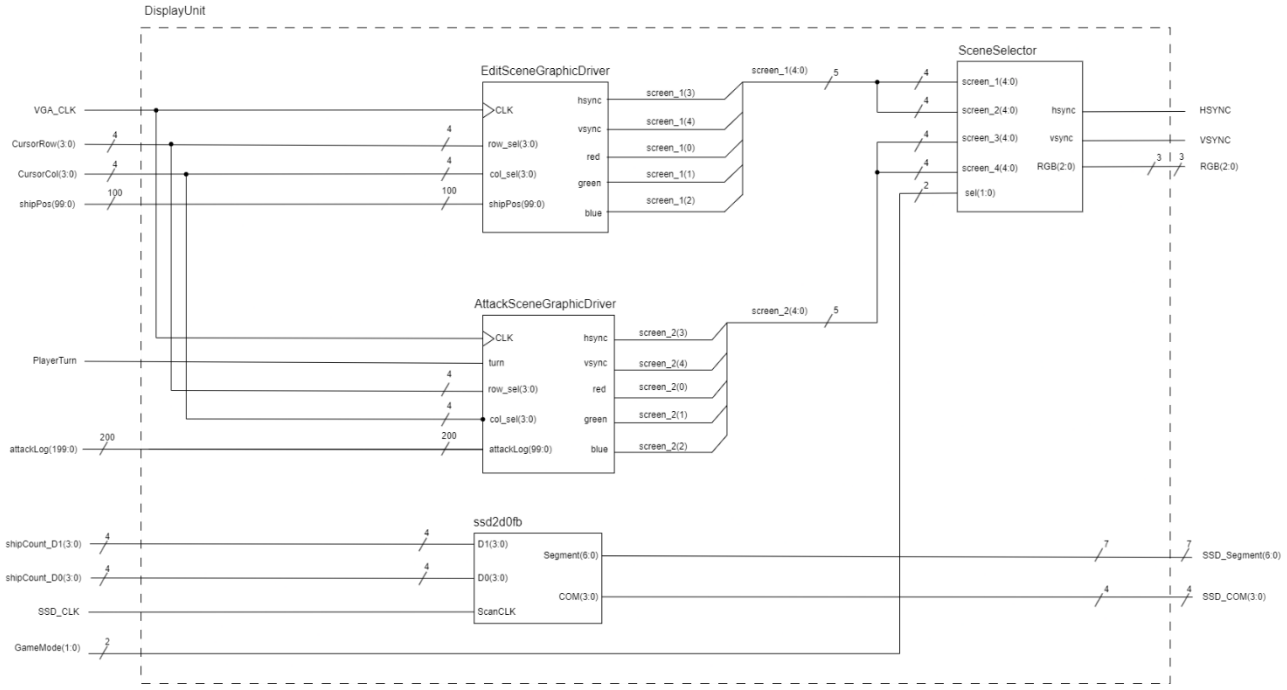
DefenseSystem (3rd Layer)

DefenseSystem เป็นโมดูลที่ใช้สำหรับการประมวลผลการโจมตีจากบอร์ด FPGA ของฝ่ายตรงข้าม ว่าตำแหน่งที่ฝ่ายตรงข้ามพยายามยิง มีเรือวางอยู่จริงหรือไม่ โดยตรวจสอบกับตำแหน่งเรือของตนเองที่เก็บไว้ใน ShipMemory_10x10 จากนั้น จะส่งผลลัพธ์ (Flag) กลับไปให้ FPGA ของผู้ที่ทำการโจมตี เก็บลงใน AttackLogMemory_10x10x2 เพื่อให้ผู้โจมตีเห็นผลลัพธ์ของการโจมตี

DefenseSystem เขียนด้วย VHDL Code ดังนี้

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DefenseSystem is
    Port (
        row         : in  std_logic_vector(3 downto 0);
        col         : in  std_logic_vector(3 downto 0);
        shipMemory  : in  std_logic_vector(99 downto 0);
        enable      : in  std_logic;
      -- 2-bit output flag ("10" for hit, "01" for miss)
        flag        : out std_logic_vector(1 downto 0);
        enable_out  : out std_logic
    );
end DefenseSystem;

architecture Behavioral of DefenseSystem is
    signal cell_address : integer range 0 to 99;
begin

    process(row, col, enable)
    begin
        if enable = '1' then
            -- Calculate the cell address based on row and column inputs
            cell_address <= to_integer(unsigned(row)) * 10 +
to_integer(unsigned(col));

            -- Check if there's a ship at the calculated address
            if shipMemory(cell_address) = '1' then
                flag <= "10";          -- Success (hit) if there's a ship
            else
                flag <= "01";          -- Miss if there's no
ship
            end if;

            enable_out <= '1';         -- Set enable_out high for further
processing
        else
            flag <= "00";              -- Default flag value when enable is
not high
            enable_out <= '0';         -- Set enable_out low when not
enabled
        end if;
    end process;

end Behavioral;
```

DisplayUnit (3rd Layer)



รูปที่ 14 โมดูล DisplayUnit (3rd Layer) ของโครงงาน

DisplayUnit เป็นโมดูลที่ใช้สำหรับเลือก Mode ในการแสดงผลภาพของตัวเกม และรวมไปถึงการแสดงผลจำนวนเรือที่ผู้เล่นโจมตีโดนผ่าน 7-Segment ซึ่ง Mode ของตัวเกมจะประกอบไปด้วย 2 mode คือ การจัดวางเรือในฝ่ายตนเอง และการโจมตีเรือของฝ่ายตรงข้าม การสลับ Mode ของตัวเกมจะถูกควบคุมผ่าน Slide Switch 2 ซึ่งมีหน้าที่ในการส่งสัญญาณ selector ไปยังโมดูล SceneSelector เมื่อได้รับสัญญาณแล้วก็จะทำการเลือกแสดงผลภาพ Mode ที่กำหนด และส่งสัญญาณ R, G, B, Hsync, Vsync ผ่านพอร์ต VGA เพื่อแสดงผลภาพออกทางจอมอนิเตอร์

EditSceneGraphicDriver (4th Layer)

EditSceneGraphicDriver เป็นโมดูลที่ใช้สำหรับแสดงผลตัวเกมใน Mode ของการจัดเรือ ตัวโมดูลจะมีการรับ CursorRow(3:0) และCursorCol(3:0) เข้ามาเพื่อทำหน้าที่ในการแสดงผลตำแหน่ง Cursor ที่อยู่ปัจจุบันของผู้เล่น และมีการรับ shipPos(99:0) เพื่อแสดงตำแหน่งของเรือที่ผู้เล่นจัดวางในตาราง Grid ขนาด 10 × 10 ช่อง

EditSceneGraphicDriver เขียนด้วย VHDL Code ดังนี้

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity EditSceneGraphicDriver is
    Port (
        clk         : in std_logic;                     -- 25MHz input clock
        hsync       : out std_logic;
        vsync       : out std_logic;
        red         : out std_logic;                    -- Red color output
(1-bit)
        green       : out std_logic;                    -- Green color output
(1-bit)
        blue        : out std_logic;                    -- Blue color output
(1-bit)
        row_sel     : in std_logic_vector(3 downto 0);-- 4-bit row
selection
        col_sel     : in std_logic_vector(3 downto 0);-- 4-bit column
selection
          shipPos     : in std_logic_vector(99 downto 0) -- Ship Data
    );
end EditSceneGraphicDriver;

architecture Behavioral of EditSceneGraphicDriver is

    -- VGA 640x480 at 60Hz Timing Constants
    constant h_display      : integer := 640;
    constant h_front_porch  : integer := 16;
    constant h_sync_pulse   : integer := 96;
    constant h_back_porch   : integer := 48;
    constant h_total        : integer := 800;
constant v_display      : integer := 480;
    constant v_front_porch  : integer := 10;
    constant v_sync_pulse   : integer := 2;
    constant v_back_porch   : integer := 33;
    constant v_total        : integer := 525;

    signal h_count : integer range 0 to h_total - 1 := 0;
    signal v_count : integer range 0 to v_total - 1 := 0;

begin
    -- Horizontal and Vertical Counter
    process(clk)
    begin
        if rising_edge(clk) then
            if h_count = h_total - 1 then
                h_count <= 0;
                if v_count = v_total - 1 then
                    v_count <= 0;
                else
                    v_count <= v_count + 1;
                end if;
            else
                h_count <= h_count + 1;
            end if;
        end if;
    end process;

    -- Horizontal Sync Pulse
    hsync <= '0' when (h_count >= h_display + h_front_porch and h_count <
h_display + h_front_porch + h_sync_pulse) else '1';
```

```
    -- Vertical Sync Pulse
    vsync <= '0' when (v_count >= v_display + v_front_porch and v_count <
v_display + v_front_porch + v_sync_pulse) else '1';

    -- RGB Color Outputs (Blank 10x10 Grid with Selected Cell Filled Based
on Data)
    process(clk, h_count, v_count, row_sel, col_sel, shipPos)
      constant cell_size : integer := 40;   -- Cell size is 40x40 pixels
      constant grid_start_x : integer := (h_display - 400) / 2;
      constant grid_start_y : integer := (v_display - 400) / 2;

      variable selected_row : integer;
      variable selected_col : integer;
      variable cell_x : integer;
      variable cell_y : integer;
    begin
        -- Convert 4-bit row and column inputs to integers
        selected_row := to_integer(unsigned(row_sel));
        selected_col := to_integer(unsigned(col_sel));

        -- Calculate cell position within the 10x10 grid
        cell_x := (h_count - grid_start_x) / cell_size;
        cell_y := (v_count - grid_start_y) / cell_size;

        -- Default color: black for background
        red   <= '0';
        green <= '0';
        blue  <= '0';

        if (h_count < h_display and v_count < v_display) then
            -- Check if within the 10x10 grid area
            if (h_count >= grid_start_x and h_count <= grid_start_x + 400
and
                v_count >= grid_start_y and v_count <= grid_start_y + 400)
then
                if (cell_x = selected_col) and (cell_y = selected_row) then
                    -- Check for cell border
                    if ((h_count - grid_start_x) mod cell_size = 0 or
                        (v_count - grid_start_y) mod cell_size = 0 or
                        (h_count - grid_start_x + 1) mod cell_size = 0 or
                        (v_count - grid_start_y + 1) mod cell_size = 0)
then
                        -- Draw the cursor border in red
                        red   <= '1';
                        green <= '0';
                        blue  <= '0';
                    else
                        if shipPos((cell_y * 10) + cell_x) = '1' then
                            red   <= '1';
                            green <= '1';
                            blue  <= '1';
                        else
                            red   <= '0';
                            green <= '0';
                            blue  <= '0';  -- Black if no ship
                         end if;
                    end if;

                elsif ((h_count - grid_start_x) mod cell_size = 0 or
(v_count - grid_start_y) mod cell_size = 0) then
```

```
                    -- Draw grid lines (white outline for all cells)
                    red   <= '1';
                    green <= '1';
                    blue  <= '1';
                else
                    -- Set color for cells based on mode and shipPos data
                    if shipPos((cell_y * 10) + cell_x) = '1' then
                        -- White for cells with ship data in EDIT mode
                        red   <= '1';
                        green <= '1';
                        blue  <= '1';
                    else
                        -- Black background for other cells
                        red   <= '0';
                        green <= '0';
                        blue  <= '0';
                    end if;
                end if;
            else
                -- Outside the grid area, set background color to black
                red   <= '0';
                green <= '0';
                blue  <= '0';
            end if;
        else
            -- Set color output for non-display area (black)
            red   <= '0';
            green <= '0';
            blue  <= '0';
        end if;
    end process;

end Behavioral;
```

AttackSceneGraphicDriver (4th Layer)

AttackSceneGraphicDriver เป็นโมดูลที่ใช้สำหรับแสดงผลตัวเกมใน Mode ของการโจมตีเรือฝ่ายตรงข้าม ตัวโมดูลจะมีการรับ CursorRow(3:0) และCursorCol(3:0) เข้ามาเพื่อทำหน้าที่ในการแสดงผลตำแหน่ง Cursor ที่อยู่ปัจจุบันของผู้เล่น และมีการรับ attackLog(199:0) เข้ามาเพื่อแสดงผลเรือที่โจมตีโดน (ช่องสีแดง) และเรือที่โจมตีไม่โดน (เครื่องหมายกากบาท)

AttackSceneGraphicDriver เขียนด้วย VHDL Code ดังนี้

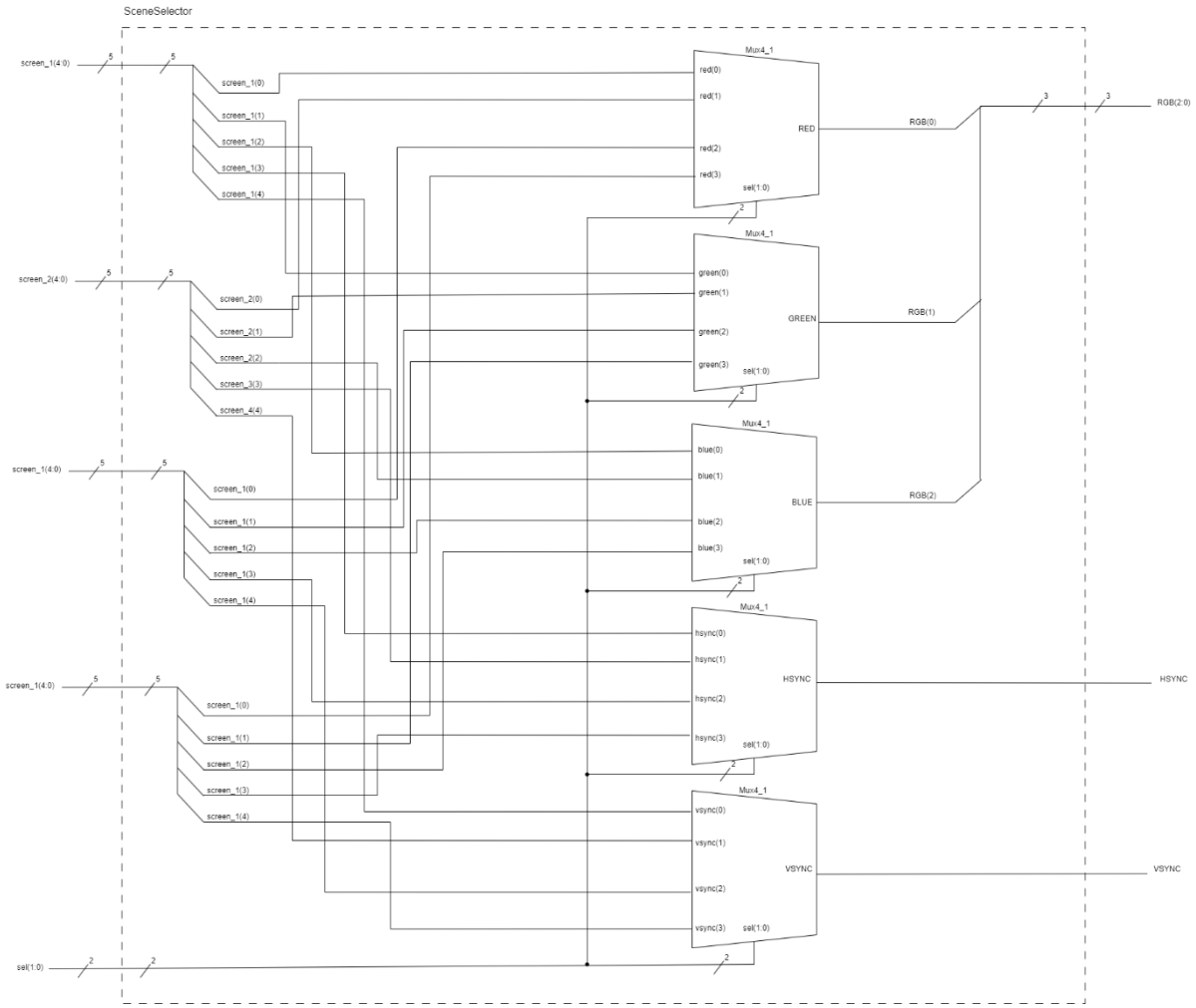
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity AttackSceneGraphicDriver is
    Port (
        clk         : in std_logic;                     -- 25MHz input clock
        hsync       : out std_logic;                    -- Horizontal sync
output
        vsync       : out std_logic;                    -- Vertical sync
output
        red         : out std_logic;                    -- Red color output
(1-bit)
        green       : out std_logic;                    -- Green color output
(1-bit)
        blue        : out std_logic;                    -- Blue color output
(1-bit)
        row_sel     : in std_logic_vector(3 downto 0);-- 4-bit row
selection
        col_sel     : in std_logic_vector(3 downto 0);-- 4-bit column
selection
        attackLog   : in std_logic_vector(199 downto 0); -- Attack Log (00
for non-attack, 01 for missed, 10 for success, 11 for invalid)
        turn        : in std_logic                      -- Turn indicator (0:
opponent's turn, 1: player's turn)
    );
end AttackSceneGraphicDriver;

architecture Behavioral of AttackSceneGraphicDriver is
    -- VGA 640x480 at 60Hz Timing Constants
    constant h_display      : integer := 640;  -- Horizontal display area
    constant h_front_porch  : integer := 16;   -- Horizontal front porch
    constant h_sync_pulse   : integer := 96;   -- Horizontal sync pulse
width
    constant h_back_porch   : integer := 48;   -- Horizontal back porch
    constant h_total        : integer := 800;  -- Total horizontal pixels
constant v_display      : integer := 480;  -- Vertical display area
    constant v_front_porch  : integer := 10;   -- Vertical front porch
    constant v_sync_pulse   : integer := 2;    -- Vertical sync pulse width
    constant v_back_porch   : integer := 33;   -- Vertical back porch
    constant v_total        : integer := 525;  -- Total vertical lines

    signal h_count : integer range 0 to h_total - 1 := 0;
    signal v_count : integer range 0 to v_total - 1 := 0;

begin
    -- Horizontal and Vertical Counter
    process(clk)
    begin
        if rising_edge(clk) then
            if h_count = h_total - 1 then
                h_count <= 0;
                if v_count = v_total - 1 then
                    v_count <= 0;
                else
                    v_count <= v_count + 1;
                end if;
            else
                h_count <= h_count + 1;
            end if;
        end if;
```

```
    end process;

    -- Horizontal Sync Pulse
    hsync <= '0' when (h_count >= h_display + h_front_porch and h_count <
h_display + h_front_porch + h_sync_pulse) else '1';

    -- Vertical Sync Pulse
    vsync <= '0' when (v_count >= v_display + v_front_porch and v_count <
v_display + v_front_porch + v_sync_pulse) else '1';

    -- RGB Color Outputs (Display the Attack Log Based on the Attack State)
    process(clk, h_count, v_count, row_sel, col_sel, attackLog, turn)
        constant cell_size : integer := 40;   -- Cell size is 40x40 pixels
        constant grid_start_x : integer := (h_display - 400) / 2;  --
Center the grid horizontally (400 = 10 cells * 40 pixels)
        constant grid_start_y : integer := (v_display - 400) / 2 + 20;  --
Center the grid vertically, 40 pixels down for text
        variable selected_row : integer;
        variable selected_col : integer;
        variable cell_x : integer;
        variable cell_y : integer;
        variable cell_state : std_logic_vector(1 downto 0);
    begin
        -- Convert 4-bit row and column inputs to integers
        selected_row := to_integer(unsigned(row_sel));
        selected_col := to_integer(unsigned(col_sel));

        -- Calculate cell position within the 10x10 grid
        cell_x := (h_count - grid_start_x) / cell_size;
        cell_y := (v_count - grid_start_y) / cell_size;

        -- Default color: black for background
        red   <= '0';
        green <= '0';
        blue  <= '0';

        -- Display "Opponent's turn..." or "It's your turn..." at the top
        if (h_count < h_display and v_count < 20) then
            if turn = '0' then
                -- Opponent's turn: Display in red
                red   <= '1';
                green <= '0';
                blue  <= '0';
            else
                -- Player's turn: Display in green
                red   <= '0';
                green <= '1';
                blue  <= '0';
            end if;

        elsif (h_count < h_display and v_count < v_display) then
            -- Check if within the 10x10 grid area
            if (h_count >= grid_start_x and h_count <= grid_start_x + 400
and
                v_count >= grid_start_y and v_count <= grid_start_y + 400)
then

                -- Determine the attack state of the current cell from
attackLog
                cell_state := attackLog((cell_y * 10 + cell_x) * 2 + 1
downto (cell_y * 10 + cell_x) * 2);
```

```
                -- Check if the current pixel is within the selected cell
(cursor) and draw the attack status within it
                if (cell_x = selected_col) and (cell_y = selected_row) then
                    -- Draw the cell border (red for selected cell) around
the cell
                    if ((h_count - grid_start_x) mod cell_size = 0 or
                        (v_count - grid_start_y) mod cell_size = 0 or
                        (h_count - grid_start_x + 1) mod cell_size = 0 or
                        (v_count - grid_start_y + 1) mod cell_size = 0)
then
                        red   <= '1';
                        green <= '0';
                        blue  <= '0';
                    else
                        -- Display the attack status of the selected cell
                        case cell_state is
                            when "00" =>  -- Non-attack: Blank cell (black)
                                red   <= '0';
                                green <= '0';
                                blue  <= '0';
                            when "01" =>  -- Missed attack: Cross line
(white cross)
                                if ((h_count - grid_start_x) mod cell_size
= (v_count - grid_start_y) mod cell_size) or
                                   ((h_count - grid_start_x) mod cell_size
= cell_size - (v_count - grid_start_y) mod cell_size - 1) then
                                    red   <= '1';
                                    green <= '1';
                                    blue  <= '1';
                                end if;
                            when "10" =>  -- Successful attack: Red cell
                                red   <= '1';
                                green <= '0';
                                blue  <= '0';
                            when "11" =>  -- Invalid state: Blue cell
                                if ((h_count - grid_start_x) mod cell_size
= (v_count - grid_start_y) mod cell_size) or
                                   ((h_count - grid_start_x) mod cell_size
= cell_size - (v_count - grid_start_y) mod cell_size - 1) then
                                    red   <= '1';
                                    green <= '1';
                                    blue  <= '1';
                                end if;
                            when others =>
                                red   <= '0';
                                green <= '0';
                                blue  <= '0';
                        end case;
                    end if;
                elsif ((h_count - grid_start_x) mod cell_size = 0 or
(v_count - grid_start_y) mod cell_size = 0) then
                    -- Draw grid lines (white outline for all cells)
                    red   <= '1';
                    green <= '1';
                    blue  <= '1';
                else
                    -- Draw cell based on attack state for other cells
                    case cell_state is
                        when "00" =>  -- Non-attack: Blank cell (black)
                            red   <= '0';
```

```
                            green <= '0';
                            blue  <= '0';
                        when "01" =>  -- Missed attack: Cross line (white
cross)
                            if ((h_count - grid_start_x) mod cell_size =
(v_count - grid_start_y) mod cell_size) or
                               ((h_count - grid_start_x) mod cell_size =
cell_size - (v_count - grid_start_y) mod cell_size - 1) then
                                red   <= '1';
                                green <= '1';
                                blue  <= '1';
                            end if;
                        when "10" =>  -- Successful attack: Red cell
                            red   <= '1';
                            green <= '0';
                            blue  <= '0';
                        when "11" =>  -- Invalid state: Blue cell
                            if ((h_count - grid_start_x) mod cell_size =
(v_count - grid_start_y) mod cell_size) or
                               ((h_count - grid_start_x) mod cell_size =
cell_size - (v_count - grid_start_y) mod cell_size - 1) then
                                red   <= '1';
                                green <= '1';
                                blue  <= '1';
                            end if;
                        when others =>
                            red   <= '0';
                            green <= '0';
                            blue  <= '0';
                    end case;
                end if;

            else
                -- Outside the grid area, set background color to black
                red   <= '0';
                green <= '0';
                blue  <= '0';
            end if;

        else
            -- Set color output for non-display area (black)
            red   <= '0';
            green <= '0';
            blue  <= '0';
        end if;
    end process;

end Behavioral;
```

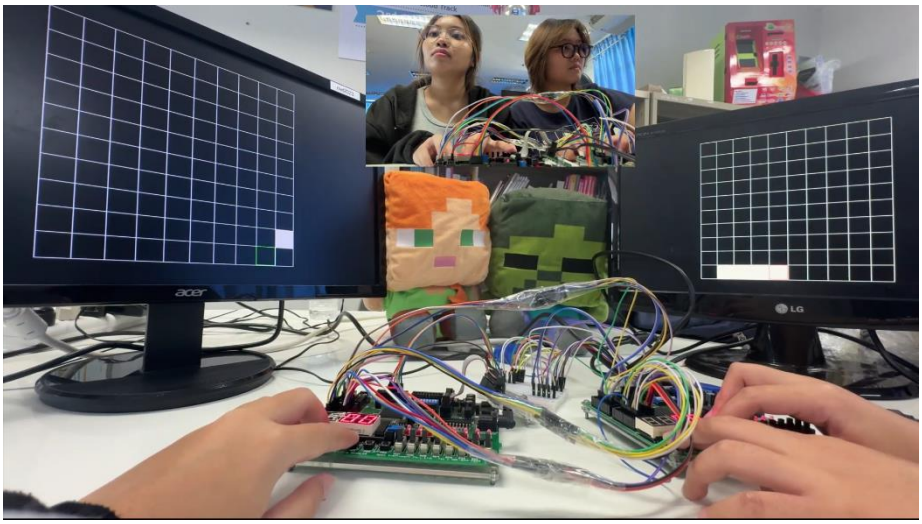
SceneSelector (4th Layer)



รูปที่ 15 โมดูล SceneSelector (4th Layer) ของโครงงาน

SceneSelector เป็นโมดูลสำหรับเลือกช่องสัญญาณในการแสดงผลภาพ ซึ่งในตัวโมดูลจะมีการใช้ Multiplexer 4-to-1 (MUX 4:1) เพื่อเป็นตัวกำหนดสัญญาณที่จะแสดงผลภาพ โดยจะรับ input จากโมดูล AttackSceneGraphicDriver และEditSceneGraphicDriver เข้ามา เพื่อเลือก Mode การแสดงผลของตัวเกม ผ่านสัญญาณ selector จาก Slide switch 2

กระบวนการทดสอบ | Testing Process



รูปที่ 16 การทดสอบเล่นเกม FPGA Battleship หลังพัฒนาเสร็จ

หลังการพัฒนาเสร็จสิ้น ได้มีการทดลองเล่นเกมจริง พบว่า ผู้เล่นสามารถวางเรือและโจมตีเรือของฝ่ายตรงข้ามได้ตามตรรกะที่ออกแบบไว้ อย่างไรก็ตาม ยังคงมีบางส่วนที่ยังคงต้องมีการปรับปรุง โดยเฉพาะ การสร้างประสบการณ์ที่ดีให้แก่ผู้เล่น (User Experience: UX) ดังต่อไปนี้

ข้อเสนอแนะ

1. การควบคุมด้วยปุ่ม Push Button เพียงสองปุ่ม อาจทำให้ผู้เล่นเกิดความสับสนในการควบคุมได้ ควรมีการพัฒนาให้ใช้ D-Pad (ปุ่ม 4 ทิศทาง) ในการควบคุม

2. หน้า Graphic User Interface (GUI) ยังไม่ได้แสดงรายละเอียดต่าง ๆ ของเกมได้ดีเท่าที่ควร อาจมีการปรับปรุงให้แสดงรายละเอียดต่าง ๆ ของเกมเพิ่มเติม เช่น จำนวนเรือที่ยิงโดน เป็นต้น

3. ยังไม่มีระบบ Turn ในการเล่น ทำให้บางครั้งผู้เล่นสับสนว่าถึง Turn ของตนเองแล้วหรือยัง (ซึ่งจะได้อธิบายที่มาของปัญหานี้ต่อไป)

4. ผู้เล่นยังสามารถหลับ Mode ของเกมไปใน Mode วางเรือหลังเกมเริ่มขึ้นแล้วได้ ซึ่งไม่ควรมีการให้มีการสลับเกิดขึ้น หลังเกมเริ่ม ผู้เล่นควรทำได้เพียงดูรูปแบบการจัดวางของตนเองเท่านั้น ไม่สามารถแก้ไขได้

ปัญหาที่พบ

ระหว่างการจัดทำโครงงาน ผู้จัดทำได้พบปัญหาต่าง ๆ มากมาย ซึ่งหลายปัญหาส่งผลต่อชิ้นงานจริง ทำให้มีความจำเป็นต้องปรับเปลี่ยน หรือปรับลด Feature บางอย่างลงไป ในหัวข้อนี้จะได้อธิบายปัญหาต่าง ๆ ดังต่อไปนี้

1. ไม่สามารถสร้าง Design ขนาดใหญ่เกินกว่าทรัพยากรที่มีในชิพ FPGA ได้

ปัญหานี้เป็นปัญหาสำคัญ เนื่องจากเป็นปัญหาที่เกิดขึ้นเมื่อพัฒนาโครงงานไปเรื่อย ๆ จนถึงจุดหนึ่งที่ผู้จัดทำใช้ทรัพยากรของ FPGA จนหมด (ในกรณีนี้ ใช้ Slices Multiplexer จนหมด) ทำให้ไม่สามารถ Implement Design ต่อไปได้ ทำให้ฟังก์ชันต่าง ๆ ที่ต้องการพัฒนาเช่น ระบบ Turn, ระบบ แสดงผลการโจมตีของฝ่ายตรงข้ามบนหน้าจอของตนเอง ไม่สามารถทำได้ เนื่องจากระบบที่กล่าวมาข้างต้น มีความจำเป็นต้องใช้ Conditional Statement ที่ซับซ้อนเป็นจำนวนมาก ซึ่งไม่สามารถทำได้เนื่องจากจำนวนทรัพยากรที่ชิพ FPGA มีให้ไม่เพียงพอนั่นเอง

ผู้จัดทำ ได้ทดลองแก้ไขปัญหาข้างต้น ด้วยการลดความซับซ้อนของ Multiplexer และตัด VHDL Code ส่วนที่ไม่จำเป็นไปเป็นจำนวนมาก แต่ปัญหาก็ได้หาคลี่คลายไม่ ผู้จัดทำจึงมีความจำเป็นต้องตัด Feture ที่ได้กล่าวไปข้างต้นออก

2. ไม่สามารถใช้งาน Built-in Hardware บนบอร์ด FPGA ได้

ในระหว่างการพัฒนาโครงงาน พบว่า ปุ่ม Push Button 3 และ 4 บนบอร์ด FPGA ไม่สามารถใช้งานได้ ซึ่งก่อนหน้าที่ผู้จัดทำจะทราบสาเหตุ ได้มีการทดลองแก้ไขการออกแบบหลายครั้ง เพราะไม่คิดว่าปัญหาจะเกิดขึ้นที่ Built-in Hardware ของ FPGA

อย่างไรก็ตาม หลังผ่านกระบวนการทดสอบหลายครั้ง จึงทราบว่า แท้จริงแล้วปัญหาเกิดขึ้นกับ Hardware จึงทำการเปลี่ยนบอร์ด FPGA ใหม่ ทำให้สามารถดำเนินโครงงานได้จนสำเร็จ

แหล่งอ้างอิง | Reference

AMD Xilinx**.** (2011)**. Xilinx DS160 Spartan-6 Family Overview.** สืบค้นเมื่อ 11 พฤศจิกายน 2024. จาก https://docs.amd.com/v/u/en-US/ds160

Pena and Legaspi. (2020). **UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter.** สืบค้นเมื่อ 3 ตุลาคม 2024. จาก https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

NextPCB. (ไม่ปรากฏปีที่เขียน). **VGA Connector Pinout - Basic Introduction is Here.** สืบค้นเมื่อ 3 ตุลาคม 2024. จาก https://www.nextpcb.com/blog/vga-connector-pinout

Milton Bradly Company. (1990). **BATTLESHIP For 2 Players**. สืบค้นเมื่อ 3 ตุลาคม 2024. จาก https://www.hasbro.com/common/instruct/battleship.pdf

Battleship Game. สืบค้นเมื่อ 3 ตุลาคม 2024. จาก https://www.battleshiponline.org/

GregChadWick. (ไม่ปรากฏปีที่เขียน). **Playing with the Pico Part 5 - Producing VGA Video.** สืบค้นเมื่อ 11 พฤศจิกายน 2024. จาก https://gregchadwick.co.uk/blog/playing-with-the-pico-pt5/

Nathan Ickes. (2004). **VGA Video.** สืบค้นเมื่อ 11 พฤศจิกายน 2024. จาก https://web.mit.edu/6.111/www/s2004/NEWKIT/vga.shtml

Miles Murdocca and Vincent Heuring. (1999). **Principles of Computer Architecture.** สืบค้นเมื่อ 11 พฤศจิกายน 2024. จาก https://slideplayer.com/slide/17338485/

รูปภาพ Diagram ทั้งหมดในเอกสารฉบับนี้ ถูกสร้างด้วย AppDiagram.io