มหาวิทยาลัยมหิดล

รายงานวิจัยฉบับสมบูรณ์

โครงการโครงการพัฒนาเครื่องมือสำหรับมาตรวิทยาและฟิสิกส์อะตอมโดยใช้ field programmable gate array

โดย ธเนศ พฤทธิวรสิน

เมษายน 2562

สัญญาเลขที่ MRG6080221

รายงานวิจัยฉบับสมบูรณ์

โครงการโครงการพัฒนาเครื่องมือสำหรับมาตรวิทยาและฟิสิกส์อะตอมโดยใช้ field programmable gate array

ธเนศ พฤทธิวรสิน
ภาควิชาฟิสิกส์ คณะวิทยาศาสตร์ มหาวิทยาลัยมหิดล

สนับสนุนโดยสำนักงานกองทุนสนับสนุนการวิจัยและต้นสังกัด

Abstract (บทคัดย่อ)

Project Code : MRG6080221

Project Title : Development of compact field programmable gate array-based equipment for application in metrology and atomic physics

(ชื่อโครงการ) โครงการโครงการพัฒนาเครื่องมือสำหรับมาตรวิทยาและฟิสิกส์อะตอม โดยใช้ field programmable gate array

Investigator : Thaned Pruttivarasin Department of Physics, Mahidol University

(ชื่อนักวิจัย) ธเนศ พฤทธิวรสิน ภาควิชาฟิสิกส์ คณะวิทยาศาสตร์ มหาวิทยาลัยมหิดล

E-mail Address : thaned.pru@mahidol.edu

Project Period : 2 years (3 April 2017 – 2 April 2019)

(ระยะเวลาโครงการ) 2 ปี (3 เมษายน 2560 – 2 เมษายน 2562)

โครงการนี้ออกแบบและสร้างเครื่องกำเนิดความถี่วิทยุในย่าน 0 – 400 MHz โดยใช้เทคนิค direct-digital synthesis ควบคุมด้วย field-programmable gate array โดยผลงานที่ได้จะนำไปประยุกต์เป็นส่วนหนึ่งของโครงการมาตรฐานความถี่และเวลาที่เป็นความร่วมมือระหว่างมหาวิทยาลัยมหิดลและสถาบันมาตรวิทยา นอกจากนี้โครงการวิจัยยังได้สร้างเครื่องวัดสัญญาณรบกวนที่มีความแม่นยำสูงและประยุกต์นำไปวัดค่าคงที่ของ Boltzmann ซึ่งยังสามารถประยุกต์ใช้ในการเรียนการสอนด้านปฏิบัติการฟิสิกส์ชั้นสูงของนักศึกษาภาควิชาฟิสิกส์ในระดับปริญญาตรี

In this project, we design and build radio-frequency wave synthesizers based on direct-digital synthesis controlled by field-programmable gate arrays. The result is a complete ready-to-use system now being applied for the optical clock project at the National Institute of Metrology, Thailand and Mahidol University. We also build a low-noise amplifier to measure resistor Johnson noise to determine the Boltzmann's constant. The equipment is also suitable for be used in advanced undergraduate physics laboratory course.

Keywords : Boltzmann constant, radio-frequency synthesizer, low noise measurement, field-programmable gate array

Executive Summary

โครงการวิจัยนี้ประกอบด้วยงานสองส่วนหลัก

เครื่องกำเนิดความถี่วิทยุ

ส่วนแรกเป็นการออกแบบและสร้างเครื่องกำเนิดความถี่วิทยุในย่าน $0 - 400$ MHz โดยใช้เทคนิค direct-digital synthesis ควบคุมด้วย field-programmable gate array ผลที่ได้คือเครื่องกำเนิดสัญญาณวิทยุที่มีความเร็วสูงในการเปลี่ยนและปรับแต่งความถี่และความเข้ม พร้อมทั้งสามารถขยายจำนวนช่องสัญญาณได้มากที่สุดถึง 16 ช่อง ทำให้ลดขนาดพื้นที่และต้นทุนที่ใช้ในการสร้างการทดลองที่ต้องใช้เครื่องกำเนิดสัญญาณความถี่ที่มีจำนวนเครื่องเยอะ ผลงานที่ได้กำลังถูกนำไปประยุกต์เป็นส่วนหนึ่งของโครงการมาตรฐานความถี่และเวลาที่เป็นความร่วมมือระหว่างมหาวิทยาลัยมหิดลและสถาบันมาตรวิทยา

เครื่องวัดสัญญาณรบกวนเพื่อการหาค่าคงที่ของ Boltzmann

ในส่วนที่สองโครงการวิจัยนี้ได้สร้างเครื่องวัดสัญญาณรบกวนที่มีความแม่นยำสูง โดยใช้วงจรที่ประกอบด้วย JFET และ operational amplifiers เราได้ประยุกต์นำไปวัดค่าคงที่ของ Boltzmann ซึ่งยังสามารถประยุกต์ใช้ในการเรียนการสอนด้านปฏิบัติการฟิสิกส์ชั้นสูงของนักศึกษาภาควิชาฟิสิกส์ในระดับปริญญาตรี ผลงานในส่วนนี้ได้ถูกตีพิมพ์ในวารสารเชิงวิชาการในระดับนานาชาติ

สารบัญ

บทที่ 1

แหล่งกำเนิดคลื่นความถี่วิทยุ สำหรับฟิสิกส์อะตอม

1.1 เกริ่นนำ

ในการศึกษาโครงสร้างของอะตอม ส่วนใหญ่แล้วสิ่งที่เราทำคือการวัดค่าพลังงานระหว่างชั้นพลังงานของอะตอมด้วยแสงเลเซอร์ ตามปกติแล้ว แสงเลเซอร์จะมีความถี่ที่ถูกกำหนดด้วยแหล่งกำเนิดแสง ซึ่งประกอบด้วย optical resonator และ gain medium ที่อยู่ในแหล่งกำเนิด อันที่จริงเราสามารถที่จะเปลี่ยนความถี่ของแสงเลเซอร์ได้ด้วยการเปลี่ยนอุณหภูมิ หรือไม่ก็เปลี่ยนปริมาณของกระแสไฟฟ้าที่ส่งผ่านเข้าไปที่ตัวแหล่งกำเนิดของแสง โดยปกติแล้วความยาวคลื่นของเลเซอร์จะเปลี่ยนไปตามอุณหภูมิที่ประมาณ

$$\Delta\lambda \approx 0.3 \text{ nm/ K} \tag{1.1}$$

แต่การปรับความยาวคลื่นหรือความถี่ของแสงเลเซอร์ด้วยวิธีนี้ จะใช้เวลานานกว่าที่ความถี่ของเลเซอร์จะเสถียร ในการทดลองฟิสิกส์อะตอม เราจำเป็นต้องเปลี่ยนความถี่ของแสงเลเซอร์ในเวลาอันสั้น อย่างเช่นประมาณ 1-2 ms

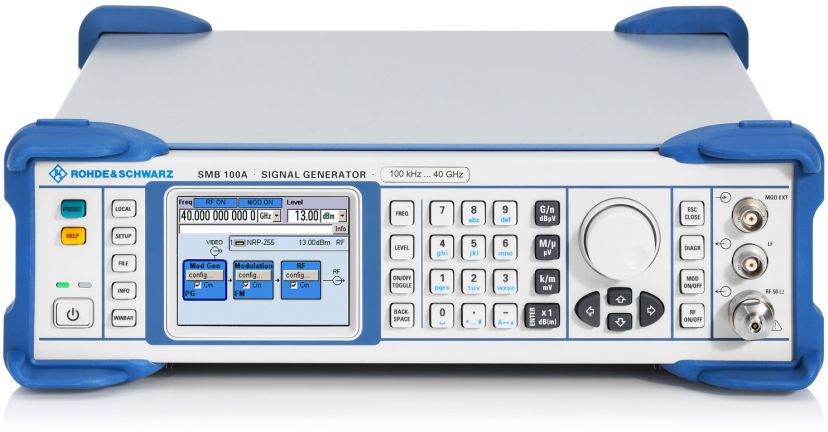
วิธีที่ดีกว่าคือการใช้ acousto optical modulator (AOM) ซึ่งเป็นผลึกที่มีดัชนีหักเหขึ้นอยู่กับคลื่นแม่เหล็กไฟฟ้าที่เราใส่เข้าไป โดยที่คลื่นแสงที่ผ่าน AOM จะมีความถี่เท่ากับ

$$f = f_0 + f_{\text{rf}} \tag{1.2}$$

โดยที่ f_0 เป็นความถี่ของแสงก่อนที่จะผ่านเข้า AOM และ f_{rf} เป็นความถี่ของแสงหลังจากที่ผ่าน AOM โดยทั่วไปแล้ว f_{rf} จะอยู่ในย่าน 20 MHz ถึง 400 MHz

ในการทดลองด้านฟิสิกส์อะตอม เราจะใช้ AOM หลายตัวในการควบคุมความถี่ของแสงให้มีความถี่ที่เราต้องการ ทุกๆ AOM ที่เราใช้ เราจะต้องมีแหล่งกำเนิดคลื่นวิทยุ (radio frequency) หนึ่งตัว (อย่างเช่นในรูปที่ 1.1) ในการทดลองหนึ่ง เราอาจจะต้องใช้แหล่งกำเนิดคลื่นวิทยุถึงสิบตัวเข้าด้วยกัน ซึ่งทำให้ต้นทุนในการ

จัดตั้งห้องทดลองนั้นสูง และยังใช้เนื้อที่มากอีกด้วย



รูปที่ 1.1 ตัวอย่างแหล่งกำเนิดสัญญาณวิทยุที่นิยมใช้กันในห้องทดลอง ซึ่งมีราคาสูง (ประมาณ 300,000 บาท) (รูปจากบริษัท Rohde & Schwarz)

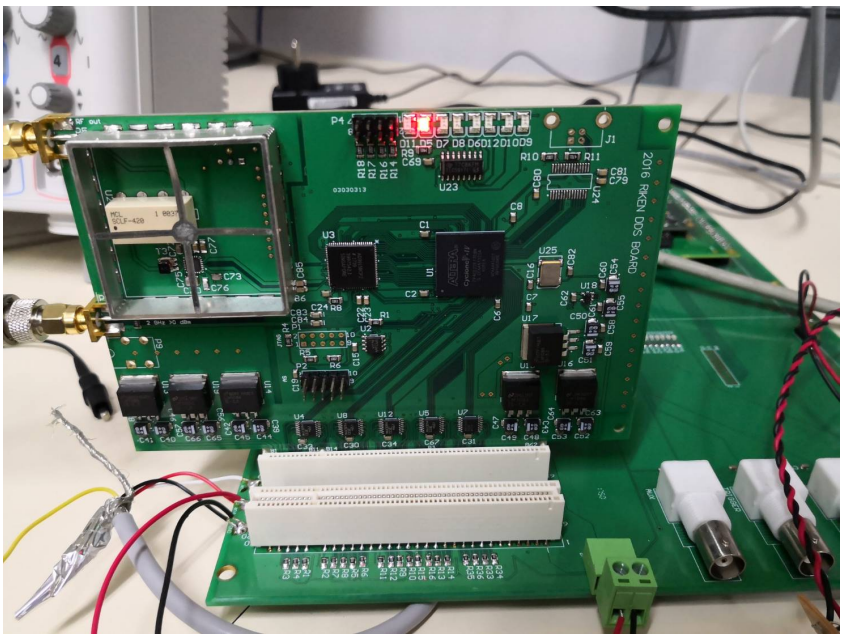
แหล่งกำเนิดคลื่นวิทยุที่ผลิตและสามารถซื้อได้นั้น มีประสิทธิภาพที่ดี สามารถปรับความถี่ได้อย่างละเอียด แต่มักจะมีราคาแพง และมีขนาดใหญ่

สิ่งสำคัญอย่างยิ่งที่เราต้องการคือการเปลี่ยนความถี่ด้วยความเร็ว ปกติแล้วแหล่งกำเนิดความถี่ที่เราซื้อได้นั้น จะมีฟังก์ชั่นการใช้งานที่เรียกว่า frequency-shift key (FSK) ซึ่งหมายถึงการที่เราโปรแกรมความถี่ของ generator ของเราไว้สองความถี่ อย่างเช่นให้เป็น f_1 และ f_2 เมื่อเราส่งสัญญาณไปยัง generator ของเรา (จะได้การเชื่อมต่อผ่าน GPIB, LAN หรือว่าเป็นเพียงสัญญาณ TTL ก็ตาม) ตัว generator ก็จะทำการสลับความถี่จาก f_1 ไปยัง f_2 หรือไม่ก็ f_2 กลับมาเป็น f_1 ซึ่งก็คือว่าเป็นไปตามที่เราต้องการ แต่โดยทั่วไป การสลับความถี่ในโหมด FSK นั้นจะใช้เวลามากกว่า 1-2 ms ซึ่งช้าเกินไปสำหรับการทดลองด้านฟิสิกส์อะตอม

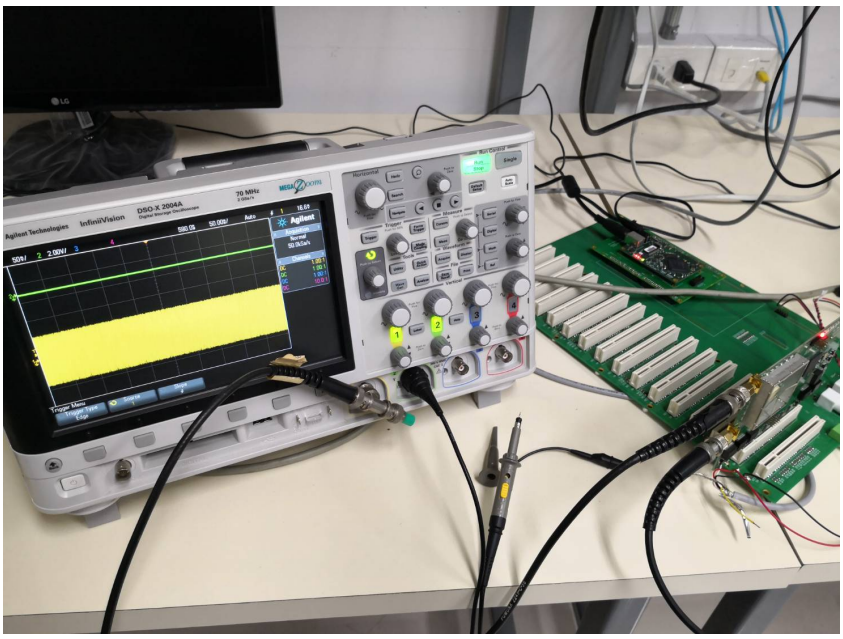
ด้วยข้อจำกัดเหล่านี้ของ generator ที่เราซื้อได้ในตลาด เราจึงมีแนวคิดที่จะสร้างวงจรที่กำเนิดความถี่ที่มีขนาดเล็กและมีประสิทธิภาพที่เทียบเท่าหรือดีกว่าที่มีขายอยู่ในท้องตลาด เป็นที่มาของโครงการวิจัยนี้

1.2 หลักการออกแบบวงจรการทำงาน

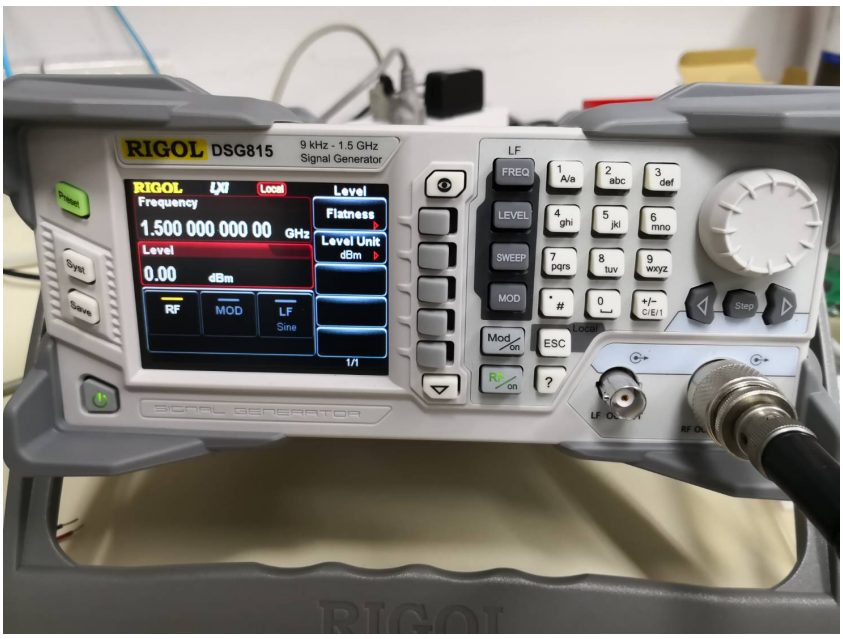
เราใช้ integrated circuit (IC) ที่ผลิตโดย Analog Device คือ AD9915 ซึ่งเป็น IC ที่สามารถสร้างความถี่วิทยุได้ตั้งแต่ 0 - 500 MHz โดยที่สามารถปรับความละเอียดของความถี่ได้ถึงระดับ nHz แต่การควบคุม IC ตัวนี้นั้น มีความซับซ้อน เราจึงต้องใช้ IC อีกตัวหนึ่งในการควบคุม และเป็นตัวกลางระหว่างตัว AD9915 กับคอมพิวเตอร์ของเรา IC ที่ว่านี้คือ field programmable gate array ที่เราสามารถโปรแกรมให้ทำงานได้ตามใจเราได้



รูปที่ 1.2 แผ่นวงจร DDS ที่ทำหน้าที่ให้กำเนิดสัญญาณวิทยุ อันที่จริงจะเรียกว่าเป็น generator ก็ไม่น่าจะถูกต้องซะทีเดียว เพราะว่าหน้าที่ของ DDS chip เบอร์ AD9915 นั้นเป็นตัวคูณหรือหารสัญญาณที่เราใช้อ้างอิงจาก 1.5 GHz (โดยตัวมันเอง ชิปตัวนี้ไม่สามารถที่จะให้ความถี่วิทยุออกมาโดยตรงได้) ไฟ LED ที่เห็นด้านบนหลายๆดวงนั้น เป็นไฟที่เอาไว้ใช้ debug ในตอนที่พัฒนาโปรแกรม ในตอนใช้งานจริงไฟเหล่านี้จะแสดงให้เห็นถึงสถานะการทำงานของวงจร DDS ในแต่ละอัน ซึ่งจะทำให้เราสามารถมองเห็นได้อย่างรวดเร็วว่าการทำงานของวงจร DDS ในแต่ละวงจรที่เสียบอยู่ในแต่ละช่วงของ PCI slot ของ motherboard นั้นยังทำงานดีอยู่หรือไม่ ในวงจรดั้งเดิมนั้น ไฟของ LED เป็นสีเขียว แต่เราพบว่าหลอดไฟสีแดงนั้นทำงานได้ดีกว่าโดยเฉพาะในการทดลองเรื่องฟิสิกส์อะตอมที่ค่อนข้างตอบสนองต่อแสงที่มีความยาวคลื่นสั้นได้ดีกว่า ซึ่งทำให้ผลกาทดลองมีความผิดพลาดเกิดขึ้น สายไฟต่างๆที่เห็นนั้นเป็นสายไฟที่มาจากแหล่งกำเนิดไฟ โดยปกติเราจะใช้ไฟ +8V และ +5V สำหรับวงจร DDS



รูปที่ 1.3 การทดลองสัญญาณออกจาก DDS ด้วย oscilloscope สังเกตว่าเราต้องใช้ 50 Ω ในการ terminate สัญญาณก่อนที่จะวัดด้วย oscilloscope ทั้งนี้เพราะว่า input impedance ของ oscilloscope นั้นจะเป็น high impedance (ซึ่งเหมาะสมกับการตรวจวัดวงจรทั่วไป) แต่ว่าสัญญาณวิทยุโดยมาตรฐานแล้วจะใช้ termination ที่ 50 Ω (หรือ 75 Ω สำหรับสัญญาณ video) การที่ไม่ terminate สัญญาณอาจจะทำให้สัญญารสะท้อนกลับซึ่งทำให้วงจร DDS นั้นเกิดความเสียหายได้



รูปที่ 1.4 แหล่งกำเนิดสัญญาณที่ 1.5 GHz ที่เราใช้เป็นสัญญาณอ้างอิงสำหรับ DDS ทุก channel

รายละเอียดของวงจรและการทำงานเบื้องต้นนั้น สามารถอ่านได้จาก งานตีพิมพ์ [1]

สำหรับโครงการวิจัยนี้ เป็นการสร้างระบบกำเนิดความถี่วิทยุนี้ โดยใช้ทรัพยากรในประเทศไทย และสร้างร่วมมือกับสถาบันมาตรวิทยา เพื่อที่จะใช้งานในโครงการนาฬิกาอะตอมต่อไปในอนาคต

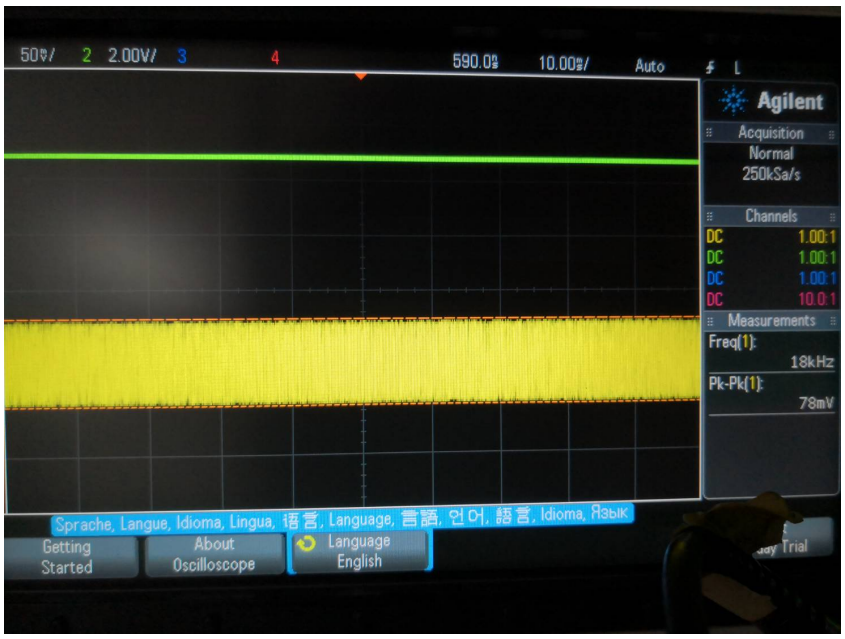
การทดสอบวงจรกำเนิดความถี่วิทยุ สังเกตว่าเราใช้ PCI slot ที่ใช้ในคอมพิวเตอร์เพื่อความสะดวกในการเพิ่มและเปลี่ยน channel แผ่นวงจรกำเนิดความถี่วิทยุ

สิ่งสำคัญคือแหล่งกำเนิดวิทยุที่ใช้เป็นตัวอ้างอิงที่ 1.5 GHz ดังที่แสดงในรูปที่ 1.4 ซึ่งความถี่ในย่าน GHz นั้น ในปัจจุบันมีการใช้งานอย่างแพร่หลาย ไม่ว่าจะเป็นสัญญาณจากโทรศัพท์เคลื่อนที่ หรือว่าสัญญาณ wifi สำหรับเครื่องข่ายสัญญาณในร่ม ที่เราเลือก 1.5 GHz นั้นเป็นเพราะว่านี่เป็นความถี่สูงสุดที่แหล่งกำเนิดสัญญาณของเราจะให้ได้

ในอนาคตถ้ามีการเปลี่ยนแหล่งกำเนิดความถี่ที่เราเอาเป็นตัวอ้างอิง เราก็สามารถจที่จะปรับแต่งได้โดยตรงใน software

1.3 ปัญหาที่พบ

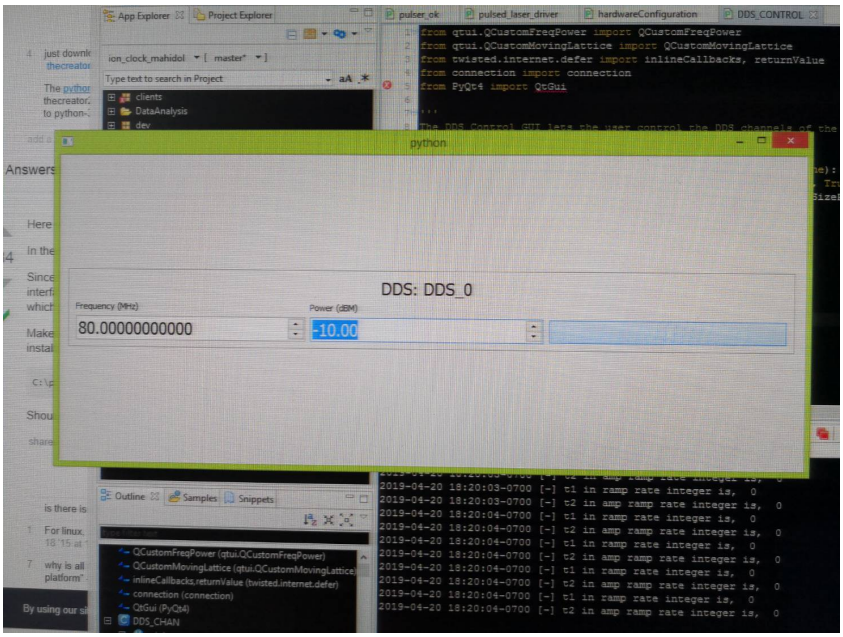
รูปที่ 1.5 ปัญหาที่เกิดจาก ground loop ที่มาจาก oscilloscope เราจะเห็นว่าคลื่นความถี่วิทยุที่ปล่อยออกมาจากวงจรนั้น แทนที่จะเป็นคลื่นต่อเนื่อง กลับมีช่วงที่คลื่นนั้นหายไป



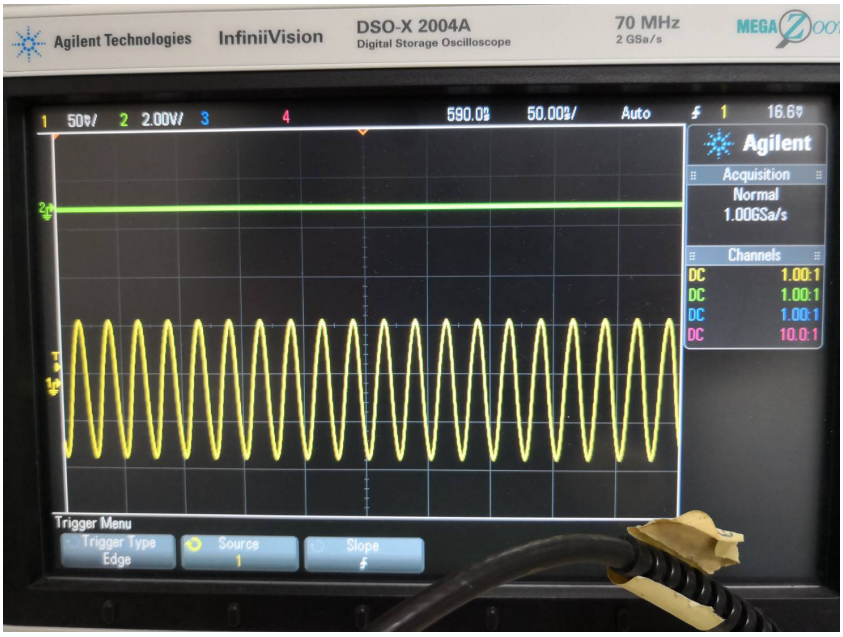
รูปที่ 1.6 เมื่อแยกแหล่งจ่ายไฟของ oscilloscope ออกมาโดยใช้ transformer แล้ว เราจะเห็นว่าคลื่นนั้นเป็นคลื่นต่อเนื่องอย่างที่มันควรจะเป็น

รูปที่ 1.7 สัญญาณรบกวนของความถี่วิทยุที่วัดได้บน oscilloscope จะสังเกตว่าสัญญาณรบกวนนั้นมี noise อยู่ที่ 50 Hz ซึ่งเป็นความถี่ของไฟในห้องทดลอง ทำให้เรามีสมมุติฐานว่า น่าจะเกิดจาก ground loop ในห้องทดลอง

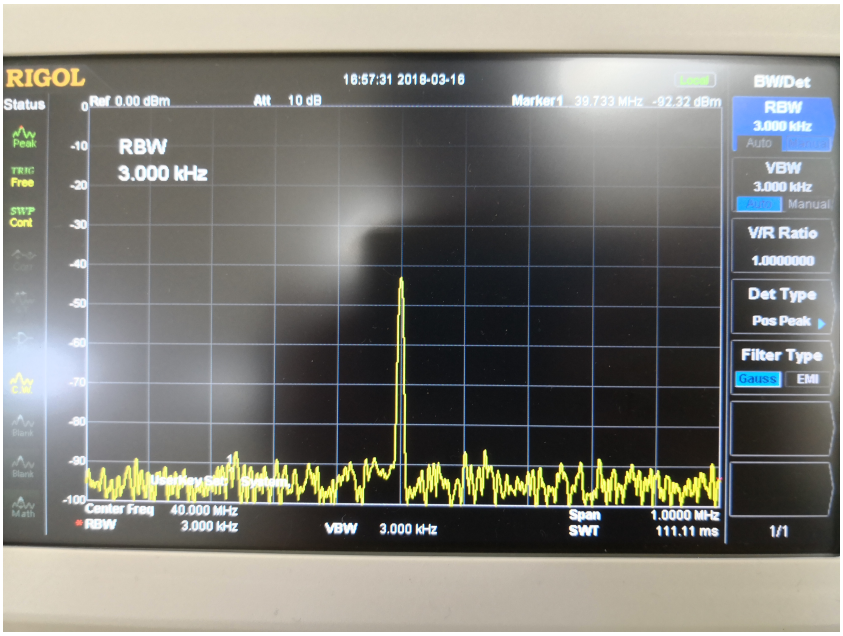
รูปที่ 1.8 หม้อแปลงที่ซื้อมาจาก aliexpress ตามความตั้งใจเดิมเราซื้อหม้อแปลงตัวนี้มาเพื่อนำมาใช้กับอุปกรณ์ที่นำมาจากประเทศสหรัฐอเมริกา ซึ่งใช้แรงดันไฟฟ้า 120 V แต่โชคดีที่หม้อแปลงตัวนี้มี option ที่ให้แรงดันไฟฟ้า 220 V ด้วย ซึ่งทำให้หน้าตัด ground loop จากวงจรของเราจากไฟบ้าน (ไฟในห้องทดลอง) ทำให้สัญญาณรบกวนที่มาจาก line power นั้นหายไป



รูปที่ 1.9 โปรแกรมที่ใช้ควบคุมความถี่ของคลื่นวิทยุ ด้วยโปรแกรม Python



รูปที่ 1.10 สัญญาณความถี่วิทยุที่ปราศจากสัญญาณรบกวน



รูปที่ 1.11 สัญญาณความถี่วิทยุที่วัดจาก spectrum analyzer จะเห็นว่าสัญญาณมีความสะอาดที่ดีใช้ได้

ด้วยความร่วมมือกับสถาบันมาตรวิทยา เราได้ผลิตตัววงจรกำเนิดคลื่นวิทยุ (ซึ่งต่อไปนี้จะเรียกว่า วงจร DDS) ได้ดังรูปและพบว่า วงจรทำงานได้ดี สามารถเชื่อมต่อกับระบบคอมพิวเตอร์ได้อย่างที่ออกแบบ (ในส่วน

ของ software นั้น โครงการมีแผนที่จะปรับเวอร์ชั่นของโค้ดคอมพิวเตอร์จาก Python 2.7 ให้เป็น 3.5 แต่เห็นว่ายังไม่มีความจำเป็นและอาจจะใช้เวลามากเกินไป จึงเป็นโครงการที่จะทำให้อนาคต)

อย่างไรก็ดี คลื่นความถี่วิทยุที่ผลิตออกมานั้น กลับมีสัญญารบกวนค่อนข้างเยอะ อย่างที่แสดงในรูปต่อไป ซึ่งแสดงให้เห็นถึง คลื่นความถี่วิทยุที่วัดได้ (สีเหลือง) เทียบกับความต่างศักย์ของแหล่งกำเนิดไฟฟ้าของวงจรทั้งสองอัน (สีเขียวและสีฟ้า) จะเห็นว่ามีคลื่นรบกวนที่เป็นคาบๆ เราพบว่าคาบนั้นตรงกับคาบของไฟบ้าน (ประมาณ 50 Hz) ซึ่งบ่งว่า สัญญาณรบกวนนี้น่าจะมาจาก ground loop ในห้องทดลอง

สิ่งหนึ่งที่เราสังเกตก็คือ การที่เราต่อสาย USB 2.0 จากแผ่นวงจรของเราเข้ากับคอมพิวเตอร์นั้น วงจรกับคอมพิวเตอร์ของเราจะแชร์ ground ของระบบเข้าด้วยกัน การเชื่อมต่อนี้เป็นทางที่ทำให้ ground ของระบบทั้งหมด (วงจรและทั้งห้องทดลอง) เป็น loop ใหญ่ ซึ่งจะทำให้เกิดสัญญาณรบกวนได้ง่ายมาก

เราจึงมีแนวคิดที่จะทำให้ ground loop นี้หายไป

ในตอนแรกเราคิดว่าสัญญาณรบกวนนี้มาจาก stability ของแหล่งกำเนิดไฟฟ้าของเรา แต่เมื่อเราลองเปลี่ยนแหล่งกำเนิดไฟฟ้า ก็พบว่าสัญญาณรบกวนนี้ยังอยู่

สุดท้ายเราได้ลองใช้อุปกรณ์ที่ตัด ground ของวงจรไฟฟ้ากับคอมพิวเตอร์ นั่นก็คือ USB isolator ที่ผลิตโดย OLIMEX (สามารถอ่าน datasheet ได้จาก

http://www.farnell.com/datasheets/1848390.pdf?_ga=2.240070211.1502224237.1555060890-1484995229.1554792329)

โดยอุปกรณ์ชิ้นนี้จะส่งผ่านข้อมูล USB ผ่านการเหนี่ยวนำทางไฟฟ้าแทนที่จะต่อสัญญาณทางไฟฟ้าโดยตรง การใช้อุปกรณ์ตัวนี้ทำให้สัญญาณรบกวนนั้นหายไปหมดสิ้น

1.4 ผลที่ได้จากงานวิจัย

โดยรวมแล้ววงจรกำเนิดคลื่นวิทยุสำหรับการทดลองฟิสิกส์อะตอมนั้น ใช้งานได้ และพร้อมที่จะเป็นส่วนหนึ่งของโครงการนาฬิกาอะตอมต่อไป

หมายเหตุ โครงการนาฬิกาอะตอม ในปัจจุบัน หัวหน้าโครงการวิจัยนี้ ได้รับทุนวิจัยจากศูนย์ความเป็นเลิศด้านฟิสิกส์ รหัสโครงการ ThEP-61-PHY-MU3 ตั้งแต่ มิถุนายน 2561 ถึง พฤษภาคม 2564 โดยร่วมมือการสถาบันมาตรวิทยา (ดร.ปิยะพัฒน์ พูลทอง) ในการดำเนินงานโครงการวิจัย สำหรับวงจรกำเนิดคลื่นวิทยุนั้น ทางมหิดลและมาตรวิทยา มีวงจรที่พร้อมใช้งานห้องทดลองละอย่างน้อย 1 ชุด ที่ผ่านการทดสอบแล้ว

1.5 โครงการในอนาคต

มีโครงการที่จะเปลี่ยนโค้ดจาก Python 2 เป็น Python 3

บทที่ 2

การวัด noise ละเอียดสูงเพื่อการหาค่า Boltzmann's constant

2.1 เกริ่นนำ

ตัวต้านทานที่มีความต้านทาน R นั้น เวลาที่เราวัดความต่างศักย์คร่อม จะเกิด voltage noise (สัญญาณรบกวนในรูปแบบของความต่างศักย์) เท่ากับ

$$V^2 = 4k_B T R \Delta f \tag{2.1}$$

โดยที่ T เป็นอุณหภูมิของตัวต้านทาน (มีหน่วยเป็น kelvin) k_B เป็น Boltzmann's constant และ Δf คือ bandwidth ของ noise ที่เราต้องการวัด

สาเหตุที่ต้องคูณ Δf (ซึ่งมีหน่วยเป็น Hz) เข้าไปนั้น เนื่องจากการพิสูจน์หา voltage noise นั้น เราจะพิจารณาถึง oscillation mode ของคลื่นแม่เหล็กไฟฟ้าที่เราวัดได้ โดยที่แต่ละ mode นั้นไม่ขึ้นต่อกัน (independent) ดังนั้น ปริมาณของ noise ที่เราวัดได้จะมากหรือน้อยขึ้นอยู่กับว่า เราวัดช่วงความถี่ตั้งแต่กี่ Hz ถึงกี่ Hz

เราอาจจะสงสัยว่า แล้วทำไมเวลาที่เราวัด voltage noise ด้วย voltmeter แล้วปริมาณ noise ที่ได้ไม่กลายเป็นอนันต์ เพราะว่าเราวัดถึงความถี่ที่สูงที่สุดที่เป็นไปได้ เหตุผลก็คือ ในระบบใดๆก็ตาม จะมี stray capacitance อยู่ (สมมุติว่ามีค่าเท่ากับ C) เราจึงมี roll-off ที่เกิดจาก RC filter พอป้องกันไม่ให้สัญญาณที่มีความถี่สูงๆนั้นมีค่า finite ทำให้เวลารวมกันแล้วไม่เป็น infinity

อย่างไรก็ดีในการวัดค่า noise นั้น ที่เราใช้นั้น จะไม่ได้กว้างมาก (อย่างมากก็ไม่เกิน MHz) เนื่องจากปกติ noise พวกนี้จะมีความต่างศักย์ที่น้อย เราจึงต้องผ่านวงจรขยายสัญญาณเพื่อที่จะทำให้สัญญาณนั้นใหญ่หรือโตพอที่จะวัดด้วยอุปกรณ์ในห้องทดลองได้ (อย่างเช่น oscilloscope หรือ digital voltmeter)

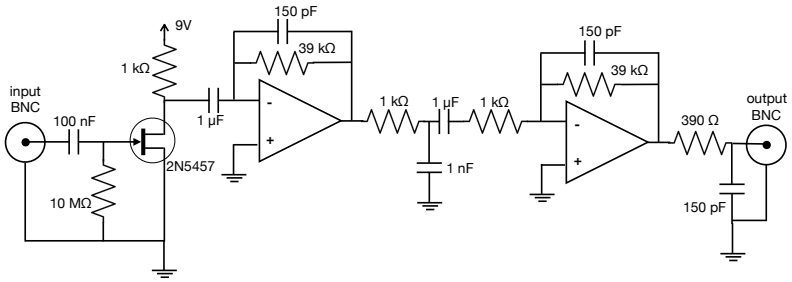
จุดมุ่งหมายหลักของเราคือการวัด สำหรับตัวต้านทานที่มีค่า R ต่างๆกัน แล้ววิเคราะห์หาค่า k_B

2.2 หลักการออกแบบวงจรการทำงาน

วงจรขยายที่เราใช้นั้นได้รับแรงบันดาลใจมาจากวงจรของ Geller[4] แต่เนื่องจากวงจรต้นฉบับนั้น ใช้อุปกรณ์บางตัวที่เลิกผลิตไปแล้ว เราจึงต้องทำการหาอุปกรณ์ตัวใหม่มาทดแทน โดยเฉพาะ JFET ที่แต่เดิมใช้เบอร์ BF445 ซึ่งเราได้เปลี่ยนเป็น 2N5457 ซึ่งก็ทำงานได้ดีพอๆกัน

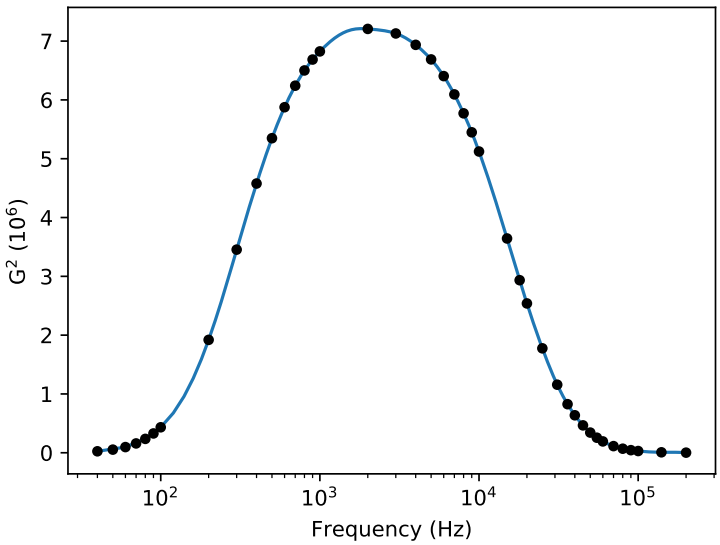
JFET นั้นเป็น ทรานซิสเตอร์ที่ใช้สนามไฟฟ้าในการเปิดปิดการไหลของอิเล็กตรอน ทำให้มี stray capacitance ที่ต่ำมาก เหมาะสำหรับวงจรขยายแบบที่เราต้องการ ถ้า capacitance ต่ำ จะทำให้ความเร็วของวงจรขยายนั้นมีมากขึ้น เราจึงอยากให้ stray capacitance ของอุปกรณ์ที่ใช้เป็นส่วนหนึ่งของวงจรอิเล็กทรอนิกส์นั้นต่ำ จริงๆแล้วเราอาจจะใช้ operational amplifier (op-amp) แทนก็ได้ แต่ว่า op-amp ที่มี spec ใกล้เคียงกับ JFET ตัวนี้นั้น หาได้ค่าข้างยาก และทำให้วงจรมีความซับซ้อนมากเกินความจำเป็น เราจึงตัดสินใจที่จะใช้ JFET คงเดิมไว้ อย่างไรก็ดี JFET ที่ใช้ในวงจรดั้งเดิม [4] นั้นคือตัว BF445B ได้เลิกผลิตไปแล้ว ทำให้เราตัดสินใจที่จะหา JFET ตัวอื่นมาแทน

ในตอนที่ออกแบบวงจรเราก็ได้พยายามหา BF445B มาทดสอบการใช้งาน ก็พบว่าเราหามาได้ จากการทดสอบก็ไม่พบว่าการทำงานของวงจรเก่าและใหม่ต่างกันเท่าใดนัก



รูปที่ 2.1 วงจรขยายสัญญาณที่ใช้

โดยที่วงจรที่ใช้นั้นแสดงให้เห็นในรูปที่ 2.1 เราจะเห็นว่า วงจรนั้นมีส่วนขยายอยู่ 3 ส่วนด้วยกันคือ 1.) JFET เบอร์ 2N5457 ซึ่งมีข้อดีคือ input capacitance นั้นมีน้อย 2.) op amp ตัวแรก 3.) op amp ตัวที่สอง วงจรทั้งหมดนั้น มีอัตราขยายสูงสุดอยู่ที่ประมาณ 2500 เท่าที่ความถี่ประมาณ 1 kHz โดยที่เราวัด gain profile ได้ตามรูปที่ 2.2



รูปที่ 2.2 Gain profile ที่วัดจากการใช้ function generator ใส่สัญญาณ sine เข้าไปใน input ของ amplifier

วิธีที่เราวัด gain profile นี้ คือเราใช้ function generator ที่ทราบ ความถี่และ amplitude ที่แน่นอน ใส่เข้าไปใน input ของวงจร amplifier แล้วค่อยๆปรับความถี่ไปเรื่อยๆ ในระหว่างที่ปรับความถี่เราก็จดค่า output voltage แต่ละค่า ข้อมูลที่ได้ก็นำมาพล็อตกราฟได้ดั่งรูป

gain profile นี้จำเป็นอย่างยิ่งในการหาค่า k_B เนื่องจาก frequency bandwidth หรือที่เราเรียกว่า Δf ในสูตรนั้นเขียนได้เป็น

$$\Delta f = \int_0^\infty \frac{[G(f)]^2}{1+(2\pi f RC)^2} df \tag{2.2}$$

ซึ่งหมายความว่าเราต้องวัด gain profile หรือ ฟังก์ชั่น $G(f)$ ในการคำนวณ และสำหรับทุกๆค่า R ที่เราใช้นั้น เราต้องคำนวณหาอินทิกรัลนี้ใหม่ทุกครั้ง

ค่า เป็น stray capacitance ของระบบวงจรขยายของเรา ขึ้นกันอะไรบ้างนั้น จริงๆแล้วถ้าจะถามว่าอะไรที่ไม่ contribute กับค่า stray capacitance นี้น่าจะง่ายกว่า (คำตอบก็คือทุกอย่างก็ contribute กับ stray capacitance นี้ทั้งนั้น)

การออกแบบวงจรที่ดีนั้นทำให้ stray capacitance นั้นน้อย เราอยากให้ค่า น้อยๆเพราะว่าจะทำให้ roll-off อันเนื่องมาจาก นั้นเกิดที่ความถี่สูงๆ (ทำให้วงจรของเรา “เร็ว” นั่นเอง)

โดยทั่วไปแล้ว เราจะลด stray capacitance ด้วยการ ทำให้ขนาดของ integrated circuit (ที่เราเรียกว่าชิป) ของเราเล็กลง อย่างใช้เปลี่ยนไปใช้ package แบบ SMD (surface mounted device)

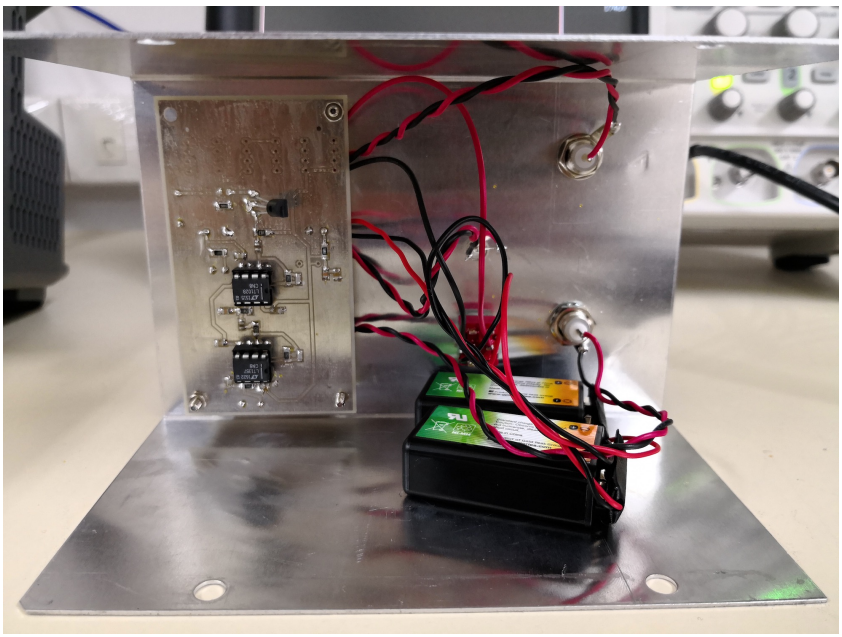
สำหรับในวงจรที่โครงการวิจัยนี้ เราไม่จำเป็นต้องทำให้ stray capacitance นั้นเล็กจนเกินไป เพราะสุดท้ายเราจะตั้งใจเพิ่ม stray capacitance เพื่อ demonstrate ให้เห็นถึงผลของ roll off ในวงจร amplifier อีกด้วย

2.3 การสร้างวงจร

วงจรที่สร้าง เราใช้โปรแกรม Eagle ในการออกแบบแผ่นวงจรพิมพ์ แล้วก็มีหลายเวอร์ชั่นที่เราปรับปรุงเรื่อยมา ณ ที่นี้เราจะขอสรุปข้อดีข้อเสียที่เราพบในการสร้างวงจรด้วยวิธีต่างๆ

- สิ่งหนึ่งที่เราต้องพิจารณาคือ วงจรแบบนี้ เรา (หรือนักเรียนเวลาใช้เรียน) จะต้องทำการเปลี่ยนตัวต้านทาน input ของวงจรบ่อยๆ ตอนแรกที่เราคิดกันคือใช้ตัวหนีบ (ภาษาอย่างไม่เป็นทางการเรียกว่าปากจระเข้) เป็นตัวหนีบตัวต้านทาน แต่เราพบว่า นักเรียนจะชอบบิดสายไฟไปมาทำให้สายขาดในเวลาอันสั้น (1 เทอม สายขาดไปประมาณ 5 ที)

- ในเวอร์ชั่นแรก ด้วยความที่อยากจะลด stray capacitance เราเลยใช้ op amp ทุกตัวแบบ SMD ปรากฏว่าเมื่อใช้งานไปเรื่อย op amp ก็มีเสียบ้าง (แม้ว่าจะไม่บ่อย แต่ก็ทำให้เปลี่ยนลำบาก) เวอร์ชั่นสุดท้ายเลยเปลี่ยนเป็น DIP แทน ทำให้เปลี่ยนง่ายกว่า เราไม่ได้วัดว่า แบบ SMD กับ DIP ทำให้ stray capacitance นั้นเปลี่ยนไปมาขนาดไหน แต่ stray capacitance ในกรณีที่ใช้ DIP นั้นอยู่ในย่านที่เรารับได้ (ประมาณ 10 pF) เราจึงตัดสินใจว่าจะใช้แบบ DIP เพื่อความสะดวกในการซ่อมบำรุง รูปที่ 2.3 นี้เป็นรูปแบบของวงจรเวอร์ชั่นสุดท้าย

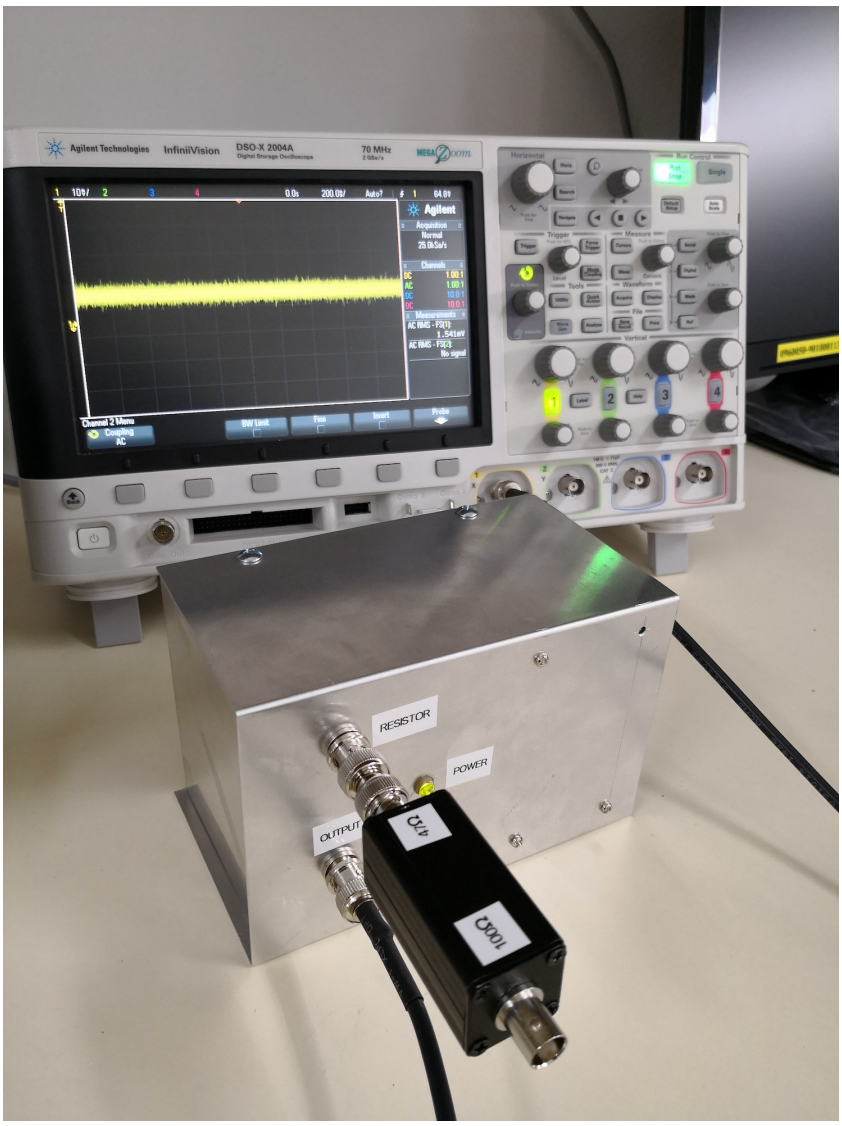


รูปที่ 2.3 โครงสร้างภายในของวงจร

- สิ่งที่สำคัญอย่างยิ่งยวด เนื่องจากวงจรขยายของเรานั้นมีอัตราขยายที่ค่อนข้างสูง จำเป็นอย่างยิ่งที่จะใส่ในกล่องโลหะที่ปิดสนิทเพื่อที่จะลด pick-up ของสัญญารบกวนที่มาจากภายนอก ตอนที่ทำการวัดก็ควรจะอยู่ในห้องที่เงียบๆด้วย เพราะคลื่นเสียงสามารถเหนี่ยวนำให้เกิดการสั่นสะเทือนของข้อต่อต่างๆ ทำให้เกิดสัญญาณรบกวนเพิ่มขึ้น

- การต่อ ตัวต้านทานแต่ละค่านั้น เราเปลี่ยนมาใช้เป็นกล่องที่มีตัวต่อ BNC ทำให้สะดวกต่อการเปลี่ยนค่าดังที่แสดงในรูปที่ 2.4 และรูปที่ 2.5

รูปที่ 2.4 กล่องตัวต้านทานที่ใส่ BNC connector ทำให้สะดวกสบายในการเปลี่ยนค่า ทำให้เครื่องไม่ต้องผ่านการบำรุงรักษาเยอะเกินไป



รูปที่ 2.5 ตัวอย่างการวัดจริง

รูปที่ 2.6 ตัวอย่างการวัดในอีกมุมหนึ่ง

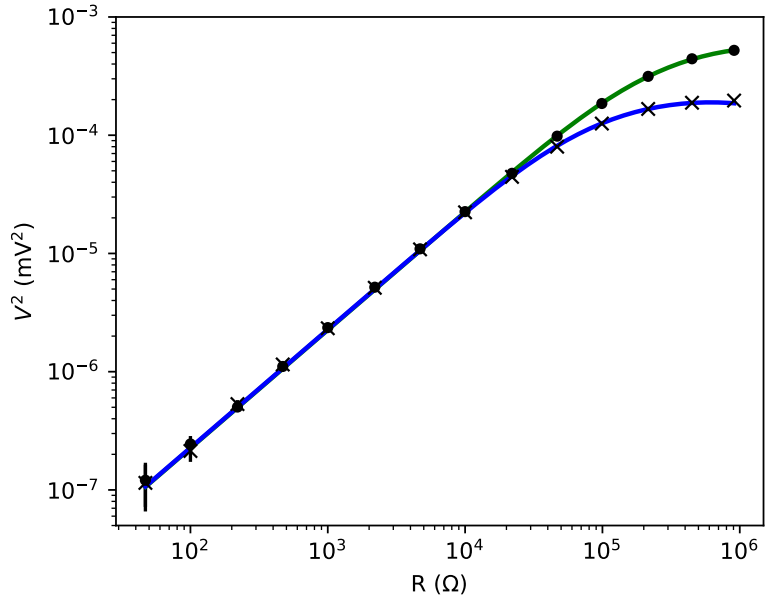
- เนื่องจากการที่ ตัวต้านทาน นั้นบรรจุอยู่ในกล่องอะลูมิเนียมเล็กๆ เรามีความคิดที่จะใส่ thermistor เข้าไปแทน แล้ววัด ค่า noise เทียบกับอุณหภูมิ ซึ่งจะทำให้เราสามารถวัดค่า absolute temperature scale ได้ด้วย (มีข้อมูลเบื้องต้นแล้ว แต่ยังไม่พร้อมสำหรับการตีพิมพ์ คาดว่าจะสามารถตีพิมพ์ได้ประมาณกลายปี 2562)

- แบตเตอรี่ที่ใช้นั้น จะหมดเร็วมากถ้าเกิดไม่ได้ปิดสวิชต์ ซึ่งเราก็กำชับนักเรียนที่ใช้งานว่าต้องปิดสวิชต์ทุกครั้งหลังจากใช้งานเสร็จแล้ว

2.4 ผลการทดลอง

ผลการทดลองที่ได้ (ไม่ว่าจะทำเอง หรือเป็นนักเรียนเป็นคนเก็บข้อมูล) ก็ให้ผลที่ค่อนข้างดี รูปที่ 2.7 ต่อใบแสดงให้เห็นถึงค่า noise สำหรับค่าความต้านทานต่างๆกัน จะเห็นว่าแทนที่จะเป็นเส้นตรง กลับเริ่มที่จะโค้งตอนที่ R มีค่าเยอะ อันนี้เป็นผลมาจาก roll-off ที่กล่าวไปในข้างต้น ซึ่งก็เป็นสิ่งดีเพราะจะทำให้เราสามารถ fit ข้อมูลและหาค่า C ออกมาได้ด้วย C ค่า ที่หาได้อยู่ที่ ประมาณ 40 pF (ค่าที่ละเอียดสามารถดูได้ใน manuscript ที่แนบไว้ข้างท้ายรายงานเล่มนี้)

เพื่อที่จะพิสูจน์ว่า เราสามารถเพิ่ม stray capacitance ได้จากการเปลี่ยน configuration ของวงจร เรานำสาย BNC ยาว 1 เมตร มาต่อระหว่างกล่องตัวต้านทานและวงจรขยาย เราก็พบว่า ค่า stray capacitance นั้นเพิ่มขึ้นประมาณ 80 pF ซึ่งก็ใกล้เคียงกับ datasheet ของสายไฟที่เราใช้



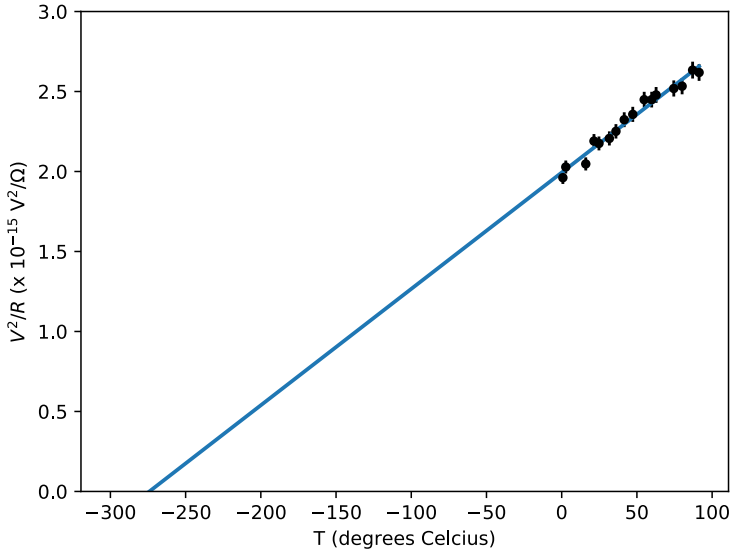
รูปที่ 2.7 ตัวอย่าง noise ที่วัดได้

กราฟข้างบนแสดงให้เห็นถึงผลการทดลองวัด noise ที่ค่าตัวต้านทานต่างๆค่ากัน สีฟ้าคือกรณีที่เราต่อสาย BNC เพิ่มเข้าไป ส่วนสีเขียวคือกรณีที่ไม่ได้ต่อ

2.5 แผนโครงการในอนาคต

จริง ๆ แล้ววงจรนี้ สามารถใช้เป็นวงจรวัด shot noise ได้ด้วย แต่ระหว่างดำเนินงานโครงการ เราไม่สามารถที่จะใช้งานในจุดนี้ได้ ซึ่งในปัจจุบันก็ยังไม่ทราบถึงสาเหตุ

นอกจากนี้เรามีแผนที่จะใช้วงจรนี้ในการวัดค่า absolute temperature scale ด้วยการวัดค่า noise ของ thermistor ที่อุณหภูมิต่างๆกันอีกด้วย ซึ่งในตอนนี้มีข้อมูลเบื้องต้นแล้ว ดังที่แสดงในรูปที่ 2.8 มีแผนที่จะเขียนเป็นผลงานส่งตีพิมพ์กลางปี 2562



รูปที่ 2.8 ตัวอย่างข้อมูลในการวัดหาค่า absolute temperature โดยวัด noise จาก thermistor

บทผนวก ก

โค้ด VHDL สำหรับวงจรให้กำเนิดสัญญาณวิทยุ

บทผนวกนี้เป็นโค้ด VHDL สำหรับวงจรให้กำเนิดสัญญาณวิทยุ จากประสบการณ์แล้วสิ่งที่จะเป็นประโยชน์ต่อผู้ที่จะทำโครงการวิจัยนี้ไปศึกษาต่อไปก็คือโค้ด VHDL ซึ่งผู้เขียนของนำโค้ดฉบับสมบูรณ์ซึ่งเป็นส่วนหนึ่งของงานวิจัยมาไว้ในภาคผนวกนี้

ก.1 FPGA สำหรับตัว motherboard

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_unsigned.all;
use work.FRONTPANEL.all;
library UNISIM;
use UNISIM.VComponents.all;

entity photon is
        port (
                ------- Opal Kelly Stuff --------
                hi_in     : in    STD_LOGIC_VECTOR(7 downto 0);
                hi_out    : out   STD_LOGIC_VECTOR(1 downto 0);
                hi_inout  : inout STD_LOGIC_VECTOR(15 downto 0);
                hi_muxsel : out   STD_LOGIC;
                hi_aa     : inout STD_LOGIC;
```

```
            i2c_sda   : out   STD_LOGIC;
            i2c_scl   : out   STD_LOGIC;
            -------- clock in from Cypress. Normally configured at 100 MHz
    ↪ -------
            clk1      : in    STD_LOGIC;
            -------- PMT input from the level translator. Note that the PMT pulse
    ↪ width is roughly 5-7 ns ----
            pmt_input : in  STD_LOGIC;
            -------- Logic In/Out -------
            logic_out: buffer STD_LOGIC_VECTOR (31 downto 0);
            logic_in:  in STD_LOGIC_VECTOR (3 downto 0);
            -------- LED ------------------------
            led       : out   STD_LOGIC_VECTOR(7 downto 0);
            -------- TO DDS ---------------------
            dds_logic_data_out : out STD_LOGIC_VECTOR (15 downto 0);
            dds_logic_fifo_rd_clk: in STD_LOGIC;
            dds_logic_fifo_rd_en: in STD_LOGIC;
            dds_logic_fifo_empty: out STD_LOGIC;
            dds_logic_ram_reset: out STD_LOGIC;
            dds_logic_step_to_next_value: out STD_LOGIC;
            dds_logic_reset_dds_chip: out STD_LOGIC;
            dds_logic_address : out STD_LOGIC_VECTOR (3 downto 0)

            --dds_logic : inout   STD_LOGIC_VECTOR(31 downto 0)
        );
end photon;

architecture arch of photon is
        ----- clocking pll component ------
        component clk_pll_100_in_200_out port (
        -- Clock in ports
            CLK_IN1           : in     std_logic;
    -- Clock out ports
            CLK_OUT1          : out    std_logic;
            CLK_OUT2          : out    std_logic;
            CLK_OUT3          : out    std_logic);
```

```
    end component;

    ----- fifo for dds -----

    component dds_fifo PORT (
            rst : IN STD_LOGIC;
            wr_clk : IN STD_LOGIC;
            rd_clk : IN STD_LOGIC;
            din : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
            wr_en : IN STD_LOGIC;
            rd_en : IN STD_LOGIC;
            dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            full : OUT STD_LOGIC;
            empty : OUT STD_LOGIC;
            wr_data_count: OUT STD_LOGIC_VECTOR(10 downto 0));
    end component;

    ------ FIFO for photon data --------
    ------ The time stamp of the photon is recorded into the fifo and ready to be
↪ read from the PC ------------
    ------ Due to the limitation in RAM size on-board, the number of photon
↪ tagged can be only 2^15 = 32768 ---

    component fifo_photon PORT (
            rst : IN STD_LOGIC;
            wr_clk : IN STD_LOGIC;
            rd_clk : IN STD_LOGIC;
            din : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
            wr_en : IN STD_LOGIC;
            rd_en : IN STD_LOGIC;
            dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            full : OUT STD_LOGIC;
            empty : OUT STD_LOGIC;
            rd_data_count : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
    end component;
```

```
  ------ block ram to store the pulses
↪ ------------------------------------------------------------------------
↪
  ------ The pulse data is first written into fifo (below). Then the fifo will transfer
↪ the data to ram. -----

  component pulser_ram PORT (
        clka : IN STD_LOGIC;
        wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
        addra : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
        dina : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
        clkb : IN STD_LOGIC;
        addrb : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
        doutb : OUT STD_LOGIC_VECTOR(63 DOWNTO 0));
  end component;

  ------- fifo to from pc to ram to store pulse -----

  component pulse_fifo PORT (
        rst : IN STD_LOGIC;
        wr_clk : IN STD_LOGIC;
        rd_clk : IN STD_LOGIC;
        din : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        wr_en : IN STD_LOGIC;
        rd_en : IN STD_LOGIC;
        dout : OUT STD_LOGIC_VECTOR(63 DOWNTO 0);
        full : OUT STD_LOGIC;
        empty : OUT STD_LOGIC;
        rd_data_count : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
  end component;

  --------- normal pmt fifo ----------

  component normal_pmt_fifo PORT (
        rst : IN STD_LOGIC;
        wr_clk : IN STD_LOGIC;
```

```
        rd_clk : IN STD_LOGIC;
        din : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        wr_en : IN STD_LOGIC;
        rd_en : IN STD_LOGIC;
        dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        full : OUT STD_LOGIC;
        empty : OUT STD_LOGIC;
        rd_data_count : OUT STD_LOGIC_VECTOR(10 DOWNTO 0));
    end component;

    --------- readout pmt fifo ----------

    component readout_count_fifo PORT (
        rst    : IN STD_LOGIC;
        wr_clk : IN STD_LOGIC;
        rd_clk : IN STD_LOGIC;
        din : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        wr_en : IN STD_LOGIC;
        rd_en : IN STD_LOGIC;
        dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        full : OUT STD_LOGIC;
        empty : OUT STD_LOGIC;
        rd_data_count : OUT STD_LOGIC_VECTOR(10 DOWNTO 0));
    end component;

    -- Target interface bus:
    signal ti_clk    : STD_LOGIC;
    signal ok1       : STD_LOGIC_VECTOR(30 downto 0);
    signal ok2       : STD_LOGIC_VECTOR(16 downto 0);
    signal ok2s      : STD_LOGIC_VECTOR(17*7-1 downto 0);

-- Endpoint connections:
    ------- configuration register -------
    signal ep00wire        : STD_LOGIC_VECTOR(15 downto 0)
  ↪ :="0000000000000000";
    ------- normal pmt measure period --------
```

```
  signal ep01wire        : STD_LOGIC_VECTOR(15 downto 0);

  ------- Manual overwrite of the output logic ------
  ------- Because there are 4 possible states for each channel of the logic out (
↪ there are
  ------- always on, always off, follow pulse, follow pulse with inverted), we need
↪ 2 bits of
  ------- information to store.

  signal ep02wire        : STD_LOGIC_VECTOR(15 downto 0);
  signal ep03wire        : STD_LOGIC_VECTOR(15 downto 0);
  ------- DDS channel ----------------------
  signal ep04wire        : STD_LOGIC_VECTOR(15 downto 0);
  ------- Number of loops wanted in the infinite loop
↪ ----------------------
  signal ep05wire        : STD_LOGIC_VECTOR(15 downto 0);
  ------- number of us delay in the line triggering
↪ ----------------------
  signal ep06wire        : STD_LOGIC_VECTOR(15 downto 0)
↪ :="0000000000000000";

  ------- output data to PC -------
  signal ep21wire        : STD_LOGIC_VECTOR(15 downto 0)
↪ :="0000000000000000";
  signal ep22wire        : STD_LOGIC_VECTOR(15 downto 0)
↪ :="0000000000000000";
  ------- Trigger in -------
  signal ep40wire        : STD_LOGIC_VECTOR(15 downto 0);

  ------ These are for pipe logic ----

  signal pipe_in_write   : STD_LOGIC;
  signal pipe_in_ready   : STD_LOGIC;
  signal pipe_in_data    : STD_LOGIC_VECTOR(15 downto 0);

  signal pipe_in_write_dds   : STD_LOGIC;
```

```
signal pipe_in_ready_dds   : STD_LOGIC;
signal pipe_in_data_dds    : STD_LOGIC_VECTOR(15 downto 0);

signal time_resolved_pipe_out_read   : STD_LOGIC;
signal time_resolved_pipe_out_valid  : STD_LOGIC;
signal time_resolved_pipe_out_data   : STD_LOGIC_VECTOR(15 downto 0);

signal normal_pmt_pipe_out_read   : STD_LOGIC;
signal normal_pmt_pipe_out_valid  : STD_LOGIC;
signal normal_pmt_pipe_out_data   : STD_LOGIC_VECTOR(15 downto 0);



signal bs_in, bs_out   : STD_LOGIC;
signal bs_in_dds, bs_out_dds   : STD_LOGIC;

--- CLOCKs -----

--- clk 100 MHz from PLL.
signal clk_100        : STD_LOGIC;
--- clk 200 MHz from PLL to sample the input PMT signal. Any slower clock
↪ will miss the pulse ---
signal clk_200        : STD_LOGIC;
--- clk 20 MHz from PLL. Not used for anything right now ----
signal clk_20         : STD_LOGIC;
--- slow clock at 1 MHz self-generated ---
signal clk_1                  : STD_LOGIC;

---- fifo photon signal ----

signal   fifo_photon_rst              : STD_LOGIC;
signal  fifo_photon_wr_clk     : STD_LOGIC;
signal  fifo_photon_din        : STD_LOGIC_VECTOR(31 downto 0);
signal  fifo_photon_wr_en             : STD_LOGIC;
signal  fifo_photon_full              : STD_LOGIC;
signal  fifo_photon_empty             : STD_LOGIC;
```

```
signal   fifo_photon_rd_data_count: STD_LOGIC_VECTOR(15 downto 0);
signal   photon_time_tag      : STD_LOGIC_VECTOR(31 downto 0);

----- fifo pulser signal -----

signal   fifo_pulser_rst              : STD_LOGIC;
signal  fifo_pulser_rd_clk      : STD_LOGIC;
signal  fifo_pulser_rd_en             : STD_LOGIC;
signal  fifo_pulser_dout              : STD_LOGIC_VECTOR(63 downto 0);
signal  fifo_pulser_full              : STD_LOGIC;
signal  fifo_pulser_empty             : STD_LOGIC;
signal   fifo_pulser_rd_data_count: STD_LOGIC_VECTOR(7 downto 0);

      ----- dds pulser signal -----

signal   fifo_dds_rst          : STD_LOGIC;
signal  fifo_dds_rd_clk : STD_LOGIC;
signal   fifo_dds_rd_clk_temp: STD_LOGIC;
signal  fifo_dds_rd_en         : STD_LOGIC;
signal  fifo_dds_dout          : STD_LOGIC_VECTOR(15 downto 0);
signal  fifo_dds_full          : STD_LOGIC;
signal  fifo_dds_empty         : STD_LOGIC;
signal  fifo_dds_wr_data_count         : STD_LOGIC_vector(10 downto 0);
signal   dds_ram_reset     : STD_LOGIC;

------ main signal route ------
signal  master_counter_hi_bit: STD_LOGIC_VECTOR (29 downto 0); -----
↪ this one is the counter for the pulser
signal  master_counter_low_bit: STD_LOGIC_VECTOR (1 downto 0); -----
↪ this one is the sub counter for the photon data. The combined is 32 bit
------ These two are the time variable of the evolution of the pulses.
------ Due to the limitation of the integer size in VHDL, the number has to
------ be divided into two separated numbers.
signal  master_counter_hi_int: integer range 0 to 1073741824 := 0;
signal  master_counter_low_int: integer range 0 to 3 := 0;
```

```
------ logic signal ------------------------------
------ This is the channels for the pulse sequence

signal master_logic :STD_LOGIC_VECTOR (31 downto 0);

------ pmt signal
↪ ------------------------------------------------

signal pmt_synced : STD_LOGIC; ------------

------ pulser ram -------

signal pulser_ram_clka : STD_LOGIC;
signal pulser_ram_wea : STD_LOGIC_VECTOR(0
↪ DOWNTO 0);
signal pulser_ram_addra : STD_LOGIC_VECTOR(9
↪ DOWNTO 0);
signal pulser_ram_dina : STD_LOGIC_VECTOR(63
↪ DOWNTO 0);
signal pulser_ram_clkb : STD_LOGIC;
signal pulser_ram_addrb : STD_LOGIC_VECTOR(9
↪ DOWNTO 0);
signal pulser_ram_doutb : STD_LOGIC_VECTOR(63
↪ DOWNTO 0);

------- This is the total number of sequence completed in the infinite loop mode
↪  of the pulser
signal seq_count_bit : STD_LOGIC_VECTOR(15
↪ downto 0);

------ various flag ------
signal pulser_counter_reset : STD_LOGIC; -----'0' = reset. '1
↪ ' = run
signal pulser_ram_reset : STD_LOGIC; -----'1' = reset
↪ pulser ram. '0' = normal operating state
```

```
  signal  pulser_infinite_loop                : STD_LOGIC; ----'1' = infinite
↪ loop. '0' = single shot
  signal  pulser_start_bit                    : STD_LOGIC; ----'1' = run
↪ sequence. '0' = pause sequence
  signal  pulser_sequence_done                : STD_LOGIC; ----'1' =
↪ sequence is done. '0' = seq is not yet done. In infinite mode it will always be '0'
  signal  pulser_flag_register                : STD_LOGIC_VECTOR (15 downto
↪ 0);---- this vector is to combine all above for convenience.

  --======= NORMAL PMT ========--
  -- FIFO --
  signal   normal_pmt_rd_data_count: STD_LOGIC_VECTOR (10 DOWNTO 0);
  signal   normal_pmt_full: STD_LOGIC;
  signal   normal_pmt_fifo_reset: STD_LOGIC;
  signal   normal_pmt_fifo_data: STD_LOGIC_VECTOR (31 DOWNTO 0);
  signal   normal_pmt_empty: STD_LOGIC;
  signal   normal_pmt_wr_clk: STD_LOGIC:='0';
  signal   normal_pmt_wr_en: STD_LOGIC:='0';
  signal   normal_pmt_block_aval: STD_LOGIC:= '0';

  -- auto mode parameter --
  signal   normal_pmt_count_period : INTEGER RANGE 0 TO 65535:=1000;
↪ --- normal pmt period in ms ---
  signal   normal_pmt_auto_count_clk : STD_LOGIC:='0';
  signal   normal_pmt_count_trigger : STD_LOGIC := '0';

  -- PMT data --
  signal   pmt_count: INTEGER RANGE 0 TO 2147483647:=0;
  signal   pmt_count_reset: STD_LOGIC;
  signal   pmt_sampled: STD_LOGIC;

  --======= READOUT PMT ========--
  --FIFO--
  signal   readout_count_rd_data_count: STD_LOGIC_VECTOR (10 DOWNTO
↪ 0);
  signal   readout_pmt_full: STD_LOGIC;
```

```
    signal   readout_count_fifo_reset: STD_LOGIC;
    signal   readout_count_fifo_data: STD_LOGIC_VECTOR (31 DOWNTO 0);
    signal   readout_pmt_empty: STD_LOGIC;
    signal   readout_count_wr_clk: STD_LOGIC:='0';
    signal   readout_count_wr_en: STD_LOGIC:='0';
    signal   readout_count_pipe_out_read: STD_LOGIC := '0';
    signal   readout_count_pipe_out_valid: STD_LOGIC;
    signal   readout_count_pipe_out_data: STD_LOGIC_VECTOR (15 DOWNTO
  ↪ 0);
    --DATA--
    signal  pmt_readout_count: INTEGER RANGE 0 TO 2147483647:=0;
    signal  readout_should_count : STD_LOGIC := '0';



    ------ line triggering ------
    signal line_triggering_enabled: STD_LOGIC := '0'; ------ 1 means trigger
  ↪ with line
    signal line_triggering_pulse: STD_LOGIC := '0'; ------- line triggering pulse
  ↪  from some input
    signal line_triggering_conditioned: STD_LOGIC:= '0'; ------ conditioning of
  ↪ the 60 hz input

    ----------------------aux logic for      ------


begin


------------------------------------------------------------------------------
  ↪

---------- DDS fifo ---------------
fifo4: dds_fifo port map(
            rst=>fifo_dds_rst,
            wr_clk=>ti_clk,
            rd_clk=>fifo_dds_rd_clk,
```

```
            din=> pipe_in_data_dds,
            wr_en=> pipe_in_write_dds,
            rd_en=> fifo_dds_rd_en,
            dout=> fifo_dds_dout,
            full=> fifo_dds_full,
            empty=> fifo_dds_empty,
            wr_data_count=>fifo_dds_wr_data_count);

pipe_in_ready_dds <= '1'; ----- enable pipe in. The only pipe in used in this design
    ↪ is writing of the pulse into this fifo.
fifo_dds_rst <= ep40wire(7); --------- this fifo never gets reset because if there's
    ↪ anything in the fifo, it will get written into the ram right away
led(5) <= not fifo_dds_empty;
led(4 downto 0) <= not logic_out(4 downto 0);
--led(7 downto 4) <= not ep04wire(3 downto 0);
--led(3 downto 2) <= ep00wire(7 downto 6);
--led(1) <= not line_triggering_pulse;
--led(0) <= not logic_in(0);
--led <= not master_logic(7 downto 0);
led(7 downto 6) <= not ep04wire(1 downto 0);

-------------------------------------------------------------------------------
    ↪
---------- condition read clk ---------------------------
-------------------------------------------------------------------------------
    ↪

      process(clk_20, fifo_dds_rd_clk_temp)
      begin
            if (rising_edge(clk_20)) then
                  if (fifo_dds_rd_clk_temp = '1') then
                        fifo_dds_rd_clk <= '1';
                  else
                        fifo_dds_rd_clk <= '0';
                  end if;
            end if;
```

```
    end process;

---============== DDS stuff
    ↪ ================================
------------------------------------------------------------------------
    ↪

dds_logic_data_out <= not fifo_dds_dout;
fifo_dds_rd_clk_temp <= not dds_logic_fifo_rd_clk;
fifo_dds_rd_en <= not dds_logic_fifo_rd_en;
dds_logic_fifo_empty <= not fifo_dds_empty;
dds_logic_ram_reset <= not (master_logic(19) or ep40wire(4)); ----------dds
    ↪ reset----------
dds_logic_step_to_next_value <= not (master_logic(18) or ep40wire(5));
dds_logic_reset_dds_chip <= not (ep40wire(6));

dds_logic_address <= not (ep04wire(3 downto 0)); ----------------set dds
    ↪ channel


------------------------------------------------------------------------
    ↪

--    process (clk_200)
--    begin
--        IF rising_edge(clk_200) THEN
--            pmt_synced <= pmt_input;
--        END IF;
--    END PROCESS;


------------------------------------------------------------------------
    ↪
---------- general pll. This generated 200 MHz and 20 MHz from 100 MHz
    ↪ ------
pll: clk_pll_100_in_200_out port map(

    -- Clock in ports
```

```
            CLK_IN1 => clk1,
  -- Clock out ports
            CLK_OUT1 => clk_200,
            CLK_OUT2 => clk_100,
            CLK_OUT3 => clk_20);


------------------------------------------------------------------------------
    ↪
------------ this fifo is to store data from the pc before writing to the block ram
    ↪ for pulser --------
------------ this fifo is first-word-fall-through!!!
    ↪ ------------------------------------------------------
    ↪
------------------------------------------------------------------------------
    ↪

fifo2: pulse_fifo port map(
            rst=>fifo_pulser_rst,
            wr_clk=>ti_clk,
            rd_clk=>fifo_pulser_rd_clk,
            din=> pipe_in_data,
            wr_en=> pipe_in_write,
            rd_en=> fifo_pulser_rd_en,
            dout=> fifo_pulser_dout,
            full=> fifo_pulser_full,
            empty=> fifo_pulser_empty,
            rd_data_count=>fifo_pulser_rd_data_count);

pipe_in_ready <= '1'; ---- enable pipe in. The only pipe in used in this design is
    ↪ writing of the pulse into this fifo.
fifo_pulser_rst <= '0'; --------- this fifo never gets reseted because if there's
    ↪ anything in the fifo, it will get written into the ram right away

---------- RAM WRITER PROCESS-------------------
---------- write from fifo to block ram to store pulser
    ↪ ----------------
```

```
---------- this ram will store the pulse sequence
    ↪ ----------

ram1: pulser_ram port map (
            clka => pulser_ram_clka,
            wea => pulser_ram_wea,
            addra => pulser_ram_addra,
            dina => pulser_ram_dina,
            clkb => pulser_ram_clkb,
            addrb => pulser_ram_addrb,
            doutb => pulser_ram_doutb);


    process (clk_100,pulser_ram_reset)
            variable write_ram_address: integer range 0 to 1023:=0;
            variable ram_process_count: integer range 0 to 8:=0;
    begin
            ------ reset ram ------
            ------ This doesn't really reset the ram but only put the address to zero so
    ↪  that the next writing
            ------ from the fifo to the ram will start from the first address. Since each
    ↪ pulse will end with all zeros anyway
            ------ it's ok to have old information in the ram. The execution will never
    ↪ get past the end line.
            if (pulser_ram_reset = '1') then
                    write_ram_address := 0;
                    ram_process_count := 0;
            elsif rising_edge(clk_100) then
                    case ram_process_count is
                            ---------- first two prepare and check whether there is
    ↪ anything in the fifo. This can be done by looking at the pin
                            ---------- fifo_pulser empty.
                            when 0 => fifo_pulser_rd_clk <='1';
                                            fifo_pulser_rd_en <= '0';
                                            pulser_ram_wea <="0";
                                            ram_process_count := 1;
                            when 1 => fifo_pulser_rd_clk <='0';
```

```
                                    if (fifo_pulser_empty = '1') then ---- '1'
↪  is empty. Go back to case 0
                                        ram_process_count:=0;
                                    else
                                        ram_process_count := 2;  ---- if
↪ there's anything in the fifo, go to the next case
                                    end if;
                    --------- there's sth in the fifo ----------
                    when 2 => fifo_pulser_rd_en <= '1';
                                    ram_process_count:=3;
                    when 3 => fifo_pulser_rd_clk <= '1'; ------------
↪ read from fifo ----------
                                    pulser_ram_wea <="1";
                                    pulser_ram_clka <= '0';
                                    ram_process_count:=4;
                    when 4 => fifo_pulser_rd_clk <= '0';
                                    ram_process_count:=5;

                    ---------- prepare data and address that are about to be
↪ written to the ram------

                    when 5 => pulser_ram_addra <=
↪ CONV_STD_LOGIC_VECTOR(write_ram_address,10);
                                    pulser_ram_dina <= fifo_pulser_dout;
                                    ram_process_count:=6;
                    when 6 => pulser_ram_clka <= '1'; ----------write to
↪ ram
                                    ram_process_count:=7;
                    when 7 => write_ram_address:=write_ram_address+1;
↪ ----- increase address by one
                                    ram_process_count:=8;
                    ----- check again if the fifo is empty or not. Basically this
↪ whole process will
                    ----- keep writing to ram until fifo is empty.
                    when 8 => if (fifo_pulser_empty = '1') then
                                        ram_process_count:=0;
```

```
                                        else
                                            ram_process_count:=2;
                                        end if;
                    end case;
                end if;
            end process;


-------- generate slow clock at 1 MHz -------

            process (clk_20)
                variable count: integer range 0 to 21 :=0;
            begin
                if (rising_edge(clk_20)) then
                    count := count + 1;
                    if (count <= 10) then
                        clk_1 <= '1';
                    elsif (count <= 20) then
                        clk_1 <= '0';
                    elsif (count=21) then
                        count :=0;
                    end if;
                end if;
            end process;


-------------------------------------------------------------------------------
    ↪
-------------------------------------------------------------------------------
    ↪
------ line triggering generation conditioning ----------------

            process (clk_20, logic_in(0))
                variable duration_count: integer range 0 to 15:=0;
                variable delay_count: integer range 0 to 15:=0;
            begin
                if (logic_in(0) = '0') then
```

```
                duration_count := 0;
                delay_count := 0;
                line_triggering_conditioned <= '0';
            elsif (rising_edge(clk_20)) then
                if (delay_count < 7) then
                    delay_count := delay_count + 1;
                elsif (delay_count >= 7) then
                    if (duration_count < 7) then
                        duration_count := duration_count + 1;
                        line_triggering_conditioned <= '1';
                    elsif (duration_count >= 7) then
                        line_triggering_conditioned <= '0';
                    end if;
                end if;
            end if;
        end process;

------ line triggering generation ----------------

        process (clk_1, line_triggering_conditioned)
            variable duration_count: integer range 0 to 65535:=0;
            variable delay_count: integer range 0 to 65535:=0;
        begin
            if (line_triggering_conditioned = '1') then
                duration_count := 0;
                delay_count := 0;
                line_triggering_pulse <= '0';
            elsif (rising_edge(clk_1)) then
                if (delay_count < CONV_INTEGER(UNSIGNED (ep06wire(15
    ↪ downto 0)))) then
                    delay_count := delay_count + 1;
                elsif (delay_count >= CONV_INTEGER(UNSIGNED (ep06wire(15
    ↪ downto 0)))) then
                    if (duration_count < 15) then
                        duration_count := duration_count + 1;
                        line_triggering_pulse <= '1';
```

```
                elsif (duration_count >= 15) then
                    line_triggering_pulse <= '0';
                end if;
            end if;
        end if;
    end process;

---------------------------------- pulser module
    ↪ ----------------------------------
------ There is a main counter that go through each step in the RAM data. ------
------ The array of data of 64 bit has two portion. Time stamp and logic. ------
------ The convention is to specify which state of the logic at each time stamp
------ The first time stamp doesn't do anything. And the pulse will stop until
------ the end line is reached. The endline is specified from timestamp = 0. ---
------ The logic in the endline also doesn't do anything
    ↪ .-----------------------

    process (clk_100, pulser_counter_reset)
        variable seq_count: INTEGER range 0 to 65535;--- count number of seq
    ↪ run
        variable count1: INTEGER range 0 to 3:=0;
        variable time_count: INTEGER:=0;
        variable time_stamp: INTEGER:=0;
        variable ram_read_address: integer range 0 to 1023:=0;
        variable ram_process_count: INTEGER range 0 to 10:=0;
        variable ram_data_out_1: STD_LOGIC_VECTOR (63 downto 0);
        variable ram_data_out_2: STD_LOGIC_VECTOR (63 downto 0);
    begin
        IF (pulser_counter_reset = '1') then
            ram_process_count:=0;
            ram_read_address:=0;
            count1:=0;
            time_count:=0;
            time_stamp:=0;
            master_logic <= "00000000000000000000000000000000";
```

```
                pulser_sequence_done <= '0';----------------"seq done
↪ flag" is deasserted
                seq_count:=0;
            ELSIF (rising_edge(clk_100)) THEN
                IF (pulser_start_bit = '1') THEN ------- This means the "run"
↪ flaged is set such that the pulse runs.
                CASE ram_process_count IS

                    ------------- read initial configuration
↪ ---------------
                    when 0 =>   ram_read_address:=0;
                                ----------wait for line triggering
↪ ---------
                                IF ((ep00wire(3) = '1' and
↪ line_triggering_pulse = '1') or (ep00wire(3) = '0')) then
                                    ram_process_count :=
↪ ram_process_count + 1;
                                END IF;
                    WHEN 1 =>   pulser_ram_clkb <= '1';
                                ram_process_count :=
↪ ram_process_count + 1;
                    WHEN 2 =>   master_logic <= pulser_ram_doutb (31 downto
↪ 0);
                            ram_read_address:=1;
                            pulser_ram_clkb <= '0';
                            ram_process_count := ram_process_count
↪ + 1;
                    WHEN 3 =>   pulser_ram_clkb <= '1';
                                ram_process_count :=
↪ ram_process_count + 1;
                    WHEN 4 =>       ram_data_out_2:=pulser_ram_doutb;
                                ram_process_count := CONV_INTEGER(
↪ UNSIGNED (ram_data_out_2(61 downto 32)));
                                ram_read_address := 2;
                                ram_process_count :=
↪ ram_process_count + 1;
```

```
                    ----- read process ------
                    WHEN 5 => IF (time_count+1 = time_stamp) THEN
↪ ------------ approaching the time stamp
                                        IF (count1 = 0) THEN
                                            pulser_ram_clkb <= '0';
                                            count1 :=1;
                                        ELSIF (count1 = 1) THEN
                                            pulser_ram_clkb <= '1';
↪ ------------ram data is read
                                            count1 :=2;
                                        ELSIF (count1 = 2) THEN
                                            pulser_ram_clkb <= '0';
                                            ram_data_out_1 :=
↪ ram_data_out_2;
                                            ram_data_out_2 :=
↪ pulser_ram_doutb; ------------latch output from ram to ram_data_out
                                            count1:=3;
                                        ELSIF (count1 = 3) THEN
                                            count1:=0;
                                            time_count := time_count+1;
                                            ram_read_address :=
↪ ram_read_address+1;
                                            time_stamp := CONV_INTEGER
↪ (UNSIGNED (ram_data_out_2(61 downto 32)));
                                                IF (time_stamp = 0) THEN
↪  ------------ if the end line (specified by timestamp = 0) is reached.
                                                    IF (
↪ pulser_infinite_loop = '1') then ----- if it's in the infinite looped mode then
↪ jump back to the beginning

↪ ram_process_count:=0;

↪ ram_read_address:=0;
                                                    count1:=0;
                                                    time_count:=0;
```

```
                                                    time_stamp:=0;
                                                    master_logic <=
↪  ram_data_out_1(31 downto 0);
                                                    seq_count:=
↪ seq_count+1;--------- increase number of sequence count ---------
                                                    if (
↪ CONV_INTEGER(UNSIGNED (ep05wire(15 downto 0))) /= 0) then
                                                        if (
↪ seq_count = CONV_INTEGER(UNSIGNED (ep05wire(15 downto 0)))) then

↪ master_logic <= "00000000000000000000000000000000";

↪ ram_process_count := 6;
                                                        end if;
                                                    end if;
                                                else

↪ ---------------- one shot mode, after this go to limbo --------
                                                    master_logic <=
↪  "00000000000000000000000000000000";

↪ ram_process_count := ram_process_count+1;
                                                    end if;
                                                ELSE
                                                    master_logic <=
↪ ram_data_out_1(31 downto 0);
                                                END IF;
                                            END IF;
                                        ELSE
                                            ------------ if the time stamp is
↪ not yet reached, keeps counting --------
                                            IF (count1 = 0) THEN
                                                count1 :=1;
                                            ELSIF (count1 = 1) THEN
                                                count1 :=2;
                                            ELSIF (count1 = 2) THEN
```

```
                                        count1:=3;
                                    ELSIF (count1 = 3) THEN
                                        count1:=0;
                                        time_count := time_count+1;
                                    END IF;
                                END IF;
                    -------- this is limbo, you can't escape from here. To get out
↪  you need to reset the pulser. -----
                    WHEN 6 => pulser_sequence_done <= '1';
                    WHEN OTHERS => NULL;

                END CASE;
                END IF;
                ------- link the read address to the ram address in the process
                pulser_ram_addrb<=conv_std_logic_vector(ram_read_address,10);
                ------- get timestamp data ready. This is to be used to tag photon
↪ in the time resolved photon process ----
                photon_time_tag(31 downto 2) <= CONV_STD_LOGIC_VECTOR
↪ (time_count,30);
                photon_time_tag(1 downto 0) <= CONV_STD_LOGIC_VECTOR(
↪ count1,2);
                ------- The number of sequence looped
↪ -------------------------
                seq_count_bit <= CONV_STD_LOGIC_VECTOR(seq_count,16);
        END IF;

    end process;

    ----------------- testing process
↪ -------------------------------
    process (clk_200, pmt_sampled)
    begin
        IF rising_edge(clk_200) then
            IF (pmt_sampled = '1') then
                pmt_synced <= '1';
                fifo_photon_wr_clk <= '1';
```

```
                else
                        pmt_synced <= '0';
                        fifo_photon_din <= photon_time_tag;
                        fifo_photon_wr_clk <= '0';
                end if;
        end if;
    end process;

    process (clk_200, pmt_input)
    begin
        IF rising_edge(clk_200) then
                pmt_sampled <= pmt_input;-- and clk_100;
        END if;
    end process;



------------------------------------------------------------------------------
    ↪
---------------------- logic_out table
    ↪ ----------------------
------------------------------------------------------------------------------
    ↪
------ 0 ------- 866DP
------ 1 ------- crystallization
------ 2 ------- bluePI
------ 3 ------- 110DP
------ 4 ------- axial
------ 5 ------- camera
------ 6
------ 7 ------- pump
------ 8
------ 9
------ 16 ------ pmt counter trigger for differential mode (DiffCountTrigger)
------ 17 ------ time resolved photon counting enable (TimeResolvedCount)
------ 18 ------ dds step to next value
------ 19 ------ reset dds
```

```
------ 20 ------ readout_should_count
------ 21 ------ advance dds 729
------ 22 ------ reset dds 729
-------------------------------------------------------------------------------
    ↪
------- This part: if ep02 = '0' and ep 03 = '0', then follow the logic
--------------------if ep02 = '0' and ep 03 = '1', then invert the logic
--------------------if ep02 = '1' and ep 03 = '0', then always '0'
--------------------if ep02 = '1' and ep 03 = '1', then always '1'



    LOGIC_OUT(0) <= master_logic(0)              WHEN (ep02wire(0)='0' AND
    ↪ ep03wire(0)='0') ELSE
                                    NOT master_logic(0)   WHEN (ep02wire(0)='0'
    ↪ AND ep03wire(0)='1') ELSE
                                    '0'                   WHEN (ep02wire(0)
    ↪ ='1' AND ep03wire(0)='0') ELSE
                                    '1';
    LOGIC_OUT(1) <= master_logic(1)              WHEN (ep02wire(1)='0' AND
    ↪ ep03wire(1)='0') ELSE
                                    NOT master_logic(1)   WHEN (ep02wire(1)='0'
    ↪ AND ep03wire(1)='1') ELSE
                                    '0'                   WHEN (ep02wire(1)
    ↪ ='1' AND ep03wire(1)='0') ELSE
                                    '1';
    LOGIC_OUT(2) <= master_logic(2)              WHEN (ep02wire(2)='0' AND
    ↪ ep03wire(2)='0') ELSE
                                    NOT master_logic(2)   WHEN (ep02wire(2)='0'
    ↪ AND ep03wire(2)='1') ELSE
                                    '0'                   WHEN (ep02wire(2)
    ↪ ='1' AND ep03wire(2)='0') ELSE
                                    '1';
    LOGIC_OUT(3) <= master_logic(3)              WHEN (ep02wire(3)='0' AND
    ↪ ep03wire(3)='0') ELSE
                                    NOT master_logic(3)   WHEN (ep02wire(3)='0'
    ↪ AND ep03wire(3)='1') ELSE
```

```
                                    '0'                    WHEN (ep02wire(3)
↪ ='1' AND ep03wire(3)='0') ELSE
                                    '1';
  LOGIC_OUT(4) <= master_logic(4)          WHEN (ep02wire(4)='0' AND
↪ ep03wire(4)='0') ELSE
                                    NOT master_logic(4)   WHEN (ep02wire(4)='0'
↪ AND ep03wire(4)='1') ELSE
                                    '0'                    WHEN (ep02wire(4)
↪ ='1' AND ep03wire(4)='0') ELSE
                                    '1';
  LOGIC_OUT(5) <= master_logic(5)          WHEN (ep02wire(5)='0' AND
↪ ep03wire(5)='0') ELSE
                                    NOT master_logic(5)   WHEN (ep02wire(5)='0'
↪ AND ep03wire(5)='1') ELSE
                                    '0'                    WHEN (ep02wire(5)
↪ ='1' AND ep03wire(5)='0') ELSE
                                    '1';
  LOGIC_OUT(6) <= master_logic(6)          WHEN (ep02wire(6)='0' AND
↪ ep03wire(6)='0') ELSE
                                    NOT master_logic(6)   WHEN (ep02wire(6)='0'
↪ AND ep03wire(6)='1') ELSE
                                    '0'                    WHEN (ep02wire(6)
↪ ='1' AND ep03wire(6)='0') ELSE
                                    '1';
  LOGIC_OUT(7) <= master_logic(7)          WHEN (ep02wire(7)='0' AND
↪ ep03wire(7)='0') ELSE
                                    NOT master_logic(7)   WHEN (ep02wire(7)='0'
↪ AND ep03wire(7)='1') ELSE
                                    '0'                    WHEN (ep02wire(7)
↪ ='1' AND ep03wire(7)='0') ELSE
                                    '1';
  LOGIC_OUT(8) <= master_logic(8)          WHEN (ep02wire(8)='0' AND
↪ ep03wire(8)='0') ELSE
                                    NOT master_logic(8)   WHEN (ep02wire(8)='0'
↪ AND ep03wire(8)='1') ELSE
```

```
                                    '0'                         WHEN (ep02wire(8)
    ↪ ='1' AND ep03wire(8)='0') ELSE
                                    '1';
      LOGIC_OUT(9) <= master_logic(9)          WHEN (ep02wire(9)='0' AND
    ↪ ep03wire(9)='0') ELSE
                                    NOT master_logic(9)  WHEN (ep02wire(9)='0'
    ↪ AND ep03wire(9)='1') ELSE
                                    '0'                         WHEN (ep02wire(9)
    ↪ ='1' AND ep03wire(9)='0') ELSE
                                    '1';
      LOGIC_OUT(10) <= master_logic(10)          WHEN (ep02wire(10)='0'
    ↪ AND ep03wire(10)='0') ELSE
                                    NOT master_logic(10)  WHEN (ep02wire(10)='0'
    ↪ AND ep03wire(10)='1') ELSE
                                    '0'                         WHEN (ep02wire(10)
    ↪ ='1' AND ep03wire(10)='0') ELSE
                                    '1';
      LOGIC_OUT(11) <= master_logic(11)          WHEN (ep02wire(11)='0'
    ↪ AND ep03wire(11)='0') ELSE
                                    NOT master_logic(11)  WHEN (ep02wire(11)='0'
    ↪ AND ep03wire(11)='1') ELSE
                                    '0'                         WHEN (ep02wire(11)
    ↪ ='1' AND ep03wire(11)='0') ELSE
                                    '1';
    ↪
------------------------------------------------------------------------------
    ↪
------ If more channels are needed, just copy above
    ↪ ----------------------------------------
      LOGIC_OUT(31 downto 16) <= master_logic(31 downto 16);

------------ 729 DDS BNC connections ------------
      LOGIC_OUT(12) <= master_logic(18);
      LOGIC_OUT(13) <= master_logic(19);
      LOGIC_OUT(14) <= '0';
      LOGIC_OUT(15) <= '0';
```

```
---- This is the data that indicates the number of photon tagged stored in the fifo
    ↪ -------
------ It will be twice the number of photon tagged because each photon tag requires
    ↪ 32 bit
------ but the fifo output is 16 bit wide

ep22wire <= fifo_photon_rd_data_count;

------------ Get flag from epwire  ----------------
------------ ep00wire(0) is the normal pmt or differential pmt mode --------
pulser_counter_reset <= ep40wire(0);
pulser_ram_reset <= ep40wire(1);
pulser_infinite_loop <= ep00wire(1);
pulser_start_bit <= ep00wire(2);
line_triggering_enabled <= ep00wire(3);

pulser_flag_register(0) <= pulser_sequence_done;




---------- this is to configure what to display on ep21wire ----
-------------------------------------------------------------------------------
    ↪

process (ti_clk)
begin
      if rising_edge(ti_clk) then
            if (ep00wire(7 downto 5) = "000") Then
                  ep21wire <= pulser_flag_register;
            elsif (ep00wire(7 downto 5) = "001") then
                  ep21wire <= seq_count_bit;
            elsif (ep00wire(7 downto 5) = "010") then
                  ep21wire(10 downto 0) <= normal_pmt_rd_data_count;
            elsif (ep00wire(7 downto 5) = "100") then
```

```
                ep21wire(10 downto 0) <= readout_count_rd_data_count;
            else
                ep21wire <= "0000000000000000";
            end if;
        end if;
end process;

------------ this fifo is for buffering the photon tagging data, when it's not
    ↪ enough the plan is to write to sdram----
---------- The logic(27) is to indicate which in the pulse sequece for the photon
    ↪ to get tagged. This is to -------
---------- have better control when there is time where no time resolved photon
    ↪ counting is needed -------

fifo_photon_wr_en <= master_logic(17);
-------------------- to have the writing clocked tied to the pmt
    ↪ ----------------------------
--fifo_photon_wr_clk <= pmt_synced;

normal_pmt_pipe_out_valid <= '1';
time_resolved_pipe_out_valid <= '1';

fifo1: fifo_photon port map (
            rst=>fifo_photon_rst,
            wr_clk=>fifo_photon_wr_clk,
            rd_clk=>ti_clk,
            din=> fifo_photon_din,
            wr_en=> fifo_photon_wr_en,
            rd_en=> time_resolved_pipe_out_read,
            --- the fifo is configured in a standard way that data is present one cycle
    ↪ after rd_en is asserted
            --- this is coincide with the way pipe_out_read is also asserted
            dout=> time_resolved_pipe_out_data,
            full=> fifo_photon_full,
            empty=> fifo_photon_empty,
            rd_data_count=>fifo_photon_rd_data_count);
```

```
fifo_photon_rst <= ep40wire(3);

i2c_sda    <= 'Z';
i2c_scl    <= 'Z';
hi_muxsel  <= '0';

------------------------------------------------------------------------
    ↪
---------------------------- NORMAL PMT
    ↪ ----------------------
------------------------------------------------------------------------
    ↪
---------- mode selection ------------
---------- This is to select whether it's a normal mode or differential mode
    ↪ ----------

normal_pmt_count_trigger <= normal_pmt_auto_count_clk WHEN ep00wire(0) = '
    ↪ 0' ELSE master_logic(16);
normal_pmt_count_period <= CONV_INTEGER(UNSIGNED (ep01wire (15
    ↪ DOWNTO 0)));

---------- generate auto clock where the user set the period --------

process (clk_100,normal_pmt_fifo_reset)
      variable count: integer range 0 to 2147483647:=0;
begin
            IF (normal_pmt_fifo_reset = '1') THEN
                  count:=0;
            ELSIF (rising_edge(clk_100)) THEN
                  count:=count+1;
                  IF (count = 1) THEN
                        normal_pmt_auto_count_clk <= '0';
                  ELSIF (count = normal_pmt_count_period*50000) THEN
                        normal_pmt_auto_count_clk <= '1';
                  ELSIF (count > normal_pmt_count_period*100000) THEN
```

```
                    count:=0;
                END IF;
            END IF;
    END PROCESS;

    ---------- trigger to reset FIFO -----
    normal_pmt_fifo_reset<=ep40wire(2);
    normal_pmt_block_aval <= '0' WHEN normal_pmt_rd_data_count =
↪ "00000000000" ELSE '1';


    -- FIFO for normal PMT: write in is 32 bit, read out 16 bit --
    fifo3: normal_pmt_fifo port map (rst => normal_pmt_fifo_reset,
                                                        wr_clk =>
↪  clk_100,
                                                        rd_clk =>
↪ ti_clk,
                                                        din =>
↪ normal_pmt_fifo_data,
                                                        wr_en =>
↪  normal_pmt_wr_en,
                                                        rd_en =>
↪ normal_pmt_pipe_out_read,
                                                        dout =>
↪ normal_pmt_pipe_out_data,
                                                        full =>
↪ normal_pmt_full,
                                                        empty =>
↪ normal_pmt_empty,

↪ rd_data_count=>normal_pmt_rd_data_count
                                                        );

    ------- generate timing sequece --------
    ------- write to fifo at the beginning of the count trigger ----
```

```
  ------- the dead time that we can't count is very low and can be ignored
↪ -------
  process (clk_100, normal_pmt_count_trigger)  ----
↪ count_trigger_active_high----
        variable count: integer range 0 to 6:=6;
        variable wr_en_var: STD_LOGIC:='0';
        variable fifo_data_var:STD_LOGIC_VECTOR(31 DOWNTO 0)
↪ :="00000000000000000000000000000000";
        variable pmt_count_reset_var: STD_LOGIC:='0';
  begin
        if (normal_pmt_count_trigger = '0') then
              count:=0;
        elsif (rising_edge(clk_100)) then
              case count IS
                    WHEN 0 =>
                          --define data--
                          wr_en_var := '0';
                          fifo_data_var (30 DOWNTO 0) :=
↪ CONV_STD_LOGIC_VECTOR(pmt_count,31);
                          fifo_data_var (31) := '0' WHEN ep00wire(0) = '0'
↪ ELSE
                                                                    '0'
↪    WHEN (master_logic(0) = '1' AND ep00wire(0) = '1') ELSE
                                                                    '1'
↪ ;
                          pmt_count_reset_var:='0';
                          count:=count+1;
                    WHEN 1 =>
                          --enable write
                          wr_en_var:='1';
                          count:=count+1;
                    WHEN 2 =>
                          --disable write
                          wr_en_var:='0';
                          count:=count+1;
                    WHEN 3 =>
```

```
                        --enable reset of pmt counting
                        pmt_count_reset_var:='1';
                        count := count+1;
                    WHEN 4 =>
                        count := count+1;
                    WHEN 5 =>
                        -- disable reset of pmt counting
                        pmt_count_reset_var:='0';
                        count := count+1;
                    WHEN 6 =>
                        NULL;
                end case;
                normal_pmt_wr_en<=wr_en_var;
                normal_pmt_fifo_data<=fifo_data_var;
                pmt_count_reset<=pmt_count_reset_var;
            end if;
        end process;

        -- count pmt by incresaing the value of pmt_count every time pmt_synced edge
        ↪ is detected
        process (pmt_count_reset, pmt_synced)
        begin
            if (pmt_count_reset = '1') then
                pmt_count<=0;
            elsif (rising_edge(pmt_synced)) then
                pmt_count<=pmt_count+1;
            end if;
        end process;

-- READOUT COUNTING:
readout_should_count <= master_logic(20);
readout_count_fifo_reset <= ep40wire(4);
readout_count_pipe_out_valid <= '1';

---------------------- readout_count FIFO
fifo5: readout_count_fifo port map (rst => readout_count_fifo_reset,
```

```
                                                wr_clk => clk_100,
                                                rd_clk => ti_clk,
                                                din => readout_count_fifo_data,
                                                wr_en => readout_count_wr_en,
                                                rd_en =>
↪ readout_count_pipe_out_read,
                                                dout =>
↪ readout_count_pipe_out_data,
                                                full =>readout_pmt_full,
                                                empty =>readout_pmt_empty,
                                                rd_data_count=>
↪ readout_count_rd_data_count
                                        );


  -- count readout counts by incresaing the value of pmt_readout_count every time
↪  pmt_synced edge is detected
  process (readout_should_count, pmt_synced)
  begin
        if (readout_should_count = '0') then
              pmt_readout_count<=0;
        elsif (rising_edge(pmt_synced)) then
              pmt_readout_count <= pmt_readout_count + 1;
        end if;
  end process;

  -- when readout_should_count is low, the counting is done and the result is
↪ written to the FIFO
  process(clk_100, readout_should_count)
        variable count: integer range 0 to 2:=2;
        variable wr_en_var: STD_LOGIC:='0';
        variable fifo_data_var:STD_LOGIC_VECTOR(31 DOWNTO 0)
↪ :="00000000000000000000000000000000";
        variable pmt_readout_count_var: INTEGER RANGE 0 TO 2147483647:=0;
  begin
        if (readout_should_count = '1') then
```

```
                pmt_readout_count_var := pmt_readout_count;
                wr_en_var := '0';
                count := 0;
            elsif (rising_edge(clk_100)) then
                case count IS
                    WHEN 0 =>
                        --define data--
                        wr_en_var := '0';
                        fifo_data_var (31 DOWNTO 0) :=
    ↪ CONV_STD_LOGIC_VECTOR(pmt_readout_count_var,32);
                        pmt_readout_count_var := 0; --avoids a latch of not
    ↪ always defining pmt_readout_count_var--
                        count:=count+1;
                    WHEN 1 =>
                        --enable write--
                        wr_en_var := '1';
                        count:=count+1;
                    WHEN 2 =>
                        --disable write--
                        wr_en_var := '0';
                end case;
                readout_count_wr_en<=wr_en_var;
                readout_count_fifo_data<=fifo_data_var;
            end if;
        end process;
-- END READOUT COUNTING.

-- Instantiate the okHost and connect endpoints.
okHI : okHost port map (hi_in=>hi_in, hi_out=>hi_out, hi_inout=>hi_inout, hi_aa
    ↪ =>hi_aa, ti_clk=>ti_clk, ok1=>ok1, ok2=>ok2);
okWO : okWireOR    generic map (N=>7) port map (ok2=>ok2, ok2s=>ok2s);
wi00 : okWireIn    port map (ok1=>ok1,                                ep_addr=>x"00",
    ↪ ep_dataout=>ep00wire);
wi01 : okWireIn    port map (ok1=>ok1,                                ep_addr=>x"01",
    ↪ ep_dataout=>ep01wire);
```

```
wi02 : okWireIn    port map (ok1=>ok1,                                    ep_addr=>x"02",
    ↪ ep_dataout=>ep02wire);
wi03 : okWireIn    port map (ok1=>ok1,                                    ep_addr=>x"03",
    ↪ ep_dataout=>ep03wire);
wi04 : okWireIn    port map (ok1=>ok1,                                    ep_addr=>x"04",
    ↪ ep_dataout=>ep04wire);
wi05 : okWireIn    port map (ok1=>ok1,                                    ep_addr=>x"05",
    ↪ ep_dataout=>ep05wire);
wi06 : okWireIn    port map (ok1=>ok1,                                    ep_addr=>x"06",
    ↪ ep_dataout=>ep06wire);
ep40 : okTriggerIn  port map (ok1=>ok1,                                    ep_addr=>x"40",
    ↪ ep_clk=>clk_1, ep_trigger=>ep40wire);
wo21 : okWireOut   port map (ok1=>ok1, ok2=>ok2s( 1*17-1 downto 0*17 ), ep_addr
    ↪ =>x"21", ep_datain=>ep21wire);
wo22 : okWireOut   port map (ok1=>ok1, ok2=>ok2s( 4*17-1 downto 3*17 ), ep_addr
    ↪ =>x"22", ep_datain=>ep22wire);
ep80 : okBTPipeIn  port map (ok1=>ok1, ok2=>ok2s( 2*17-1 downto 1*17 ), ep_addr
    ↪ =>x"80",
                          ep_write=>pipe_in_write, ep_blockstrobe=>bs_in, ep_dataout
    ↪ =>pipe_in_data, ep_ready=>pipe_in_ready);
ep81 : okBTPipeIn  port map (ok1=>ok1, ok2=>ok2s( 6*17-1 downto 5*17 ), ep_addr
    ↪ =>x"81",
                          ep_write=>pipe_in_write_dds, ep_blockstrobe=>bs_in_dds,
    ↪ ep_dataout=>pipe_in_data_dds, ep_ready=>pipe_in_ready_dds);
----time resolved----
epA0 : okBTPipeOut port map (ok1=>ok1, ok2=>ok2s( 3*17-1 downto 2*17 ),
    ↪ ep_addr=>x"A0",
                          ep_read=>time_resolved_pipe_out_read, ep_blockstrobe=>
    ↪ bs_out, ep_datain=>time_resolved_pipe_out_data, ep_ready=>
    ↪ time_resolved_pipe_out_valid);
----normal pmt ----
epA1 : okBTPipeOut port map (ok1=>ok1, ok2=>ok2s( 5*17-1 downto 4*17 ),
    ↪ ep_addr=>x"A1",
                          ep_read=>normal_pmt_pipe_out_read, ep_blockstrobe=>
    ↪ bs_out, ep_datain=>normal_pmt_pipe_out_data, ep_ready=>
    ↪ normal_pmt_pipe_out_valid);
```

```
----readout pmt ----
epA2 : okBTPipeOut port map (ok1=>ok1, ok2=>ok2s( 7*17-1 downto 6*17 ),
    ↪ ep_addr=>x"A2",
                    ep_read=>readout_count_pipe_out_read, ep_blockstrobe=>
    ↪ bs_out, ep_datain=>readout_count_pipe_out_data, ep_ready=>
    ↪ readout_count_pipe_out_valid);

end arch;
```

ก.2 FPGA สำหรับตัว DDS

[basicstyle=]

```
library ieee;
use ieee.std_logic_1164.all;
--USE ieee.std_logic_arith.all;
use ieee.numeric_std.all;
USE work.all;

entity DDS_RIKEN is
  port(
    clk_dds : in std_logic; -- clock in from DDS divided clock
         clk_in0: in std_logic; --clock in from local oscillator of 25 MHz

    -- LED driver ---
         LED_CLK: OUT STD_LOGIC;
         LED_SDI: OUT STD_LOGIC_VECTOR (0 DOWNTO 0);
         LED_LE: OUT STD_LOGIC;
         LED_OE: OUT STD_LOGIC;

         -- DDS comminucation port --
         dds_port: out std_logic_vector (31 downto 0);
         dds_master_reset: out std_logic;
         dds_osk: out std_logic;
         dds_io_update: out std_logic;
```

```
        dds_drover: in std_logic;
        dds_drhold: out std_logic;
        dds_drctl: out std_logic;

        -- function pins for dds --
        f_pin: out std_logic_vector (3 downto 0);
        --profile select pins--
        ps: out std_logic_vector (2 downto 0);

        -- pulser bus --
        dds_bus_in: in std_LOGIC_vector(31 downto 0);
        dds_bus_out: out std_LOGIC_vector(7 downto 0);
        tx_enable: out std_LOGIC_vector(1 downto 0);


        -- DAC control pin --

        dac_out : out std_LOGIC_VECTOR (13 downto 0);
        dac_wr_pin: out std_logic;


        ----address set pins----
        add_in: in std_logic_vector (3 downto 0) -- set address of the DDS
    ↪  board (4bit)
    );

end DDS_RIKEN;

architecture behaviour of DDS_RIKEN is
        ---- various constants ----
        constant CRF2_modulus: std_LOGIC_vector(15 downto 0) :=
    ↪ "0000000010001001";
        constant CRF2_profile: std_LOGIC_vector(15 downto 0) :=
    ↪ "0000000010000000";
        constant CRF2_address: std_LOGIC_vector(7 downto 0) := "00000111";
```

```
    constant CRF1_address: std_LOGIC_vector(7 downto 0) :="00000001";
    constant CRF1_enable_amp_scale: std_LOGIC_vector(15 downto 0) :=
↪ "0000000100001000";
    constant CRF1_disable_amp_scale: std_LOGIC_vector(15 downto 0) :=
↪ "0000000000001000";

    signal led_value: STD_LOGIC_VECTOR (7 downto 0);
    signal clk_system: STD_LOGIC;

    ---- declare signal for use in parallel programming mode ----
    signal par_16_bit: STD_LOGIC_vector(0 downto 0); -- '0' = 8 bit; '1
↪ ' = 16 bit
    signal par_rd: STD_LOGIC_vector(0 downto 0); -- read pin
    signal par_wr: STD_LOGIC_vector(0 downto 0); -- write pin
    signal par_add: STD_LOGIC_VECTOR(7 downto 0); -- parallel protocol
↪ address
    signal par_data: STD_LOGIC_VECTOR(15 downto 0); -- parallel
↪ protocol data

    signal dds_address: std_logic_vector(3 downto 0);

    ---- amplitude for gain variable amp ----

    signal main_amplitude: std_logic_vector(13 downto 0);
    signal main_frequency: std_LOGIC_vector(63 downto 0);
    signal main_phase:        std_LOGIC_vector(15 downto 0);
    signal target_frequency: std_LOGIC_vector(63 downto 0);
    signal target_amplitude: std_LOGIC_vector(13 downto 0);
    signal target_phase:     std_LOGIC_vector(15 downto 0);

    --- signal for bus talking to the pulser ----
    signal bus_in_address: std_LOGIC_vector(3 downto 0);
    signal bus_in_fifo_rd_clk: std_logic;
    signal bus_in_fifo_rd_en: std_logic;
    signal bus_in_fifo_empty: std_logic;
    signal bus_in_ram_reset: std_logic;
```

```
    signal bus_in_step_to_next_value: std_logic;
    signal bus_in_reset_dds_chip: std_logic;

    signal reset_fpga: std_logic;

    ---- fifo reading from pulser
    signal   fifo_dds_dout                  : STD_LOGIC_VECTOR (15
↪ downto 0);
    signal  fifo_dds_empty                  : STD_LOGIC;
    signal  fifo_dds_rd_clk      : STD_LOGIC;
    signal  fifo_dds_rd_en                  : STD_LOGIC;

    ---- ram stuff
    signal  dds_ram_data_in         : STD_LOGIC_VECTOR (15 DOWNTO 0);
    signal  dds_ram_rdaddress               : STD_LOGIC_VECTOR (11
↪ DOWNTO 0);
    signal  dds_ram_rdclock         : STD_LOGIC;
    signal  dds_ram_wraddress               : STD_LOGIC_VECTOR (14
↪ DOWNTO 0);
    signal  dds_ram_wrclock         : STD_LOGIC := '1';
    signal  dds_ram_wren               : STD_LOGIC;
    signal  dds_ram_data_out                : STD_LOGIC_VECTOR (127
↪ DOWNTO 0);
    signal  dds_ram_reset        : STD_LOGIC;
    signal   dds_step_to_next_freq : STD_LOGIC;
    signal  dds_step_to_next_freq_sampled: STD_LOGIC;

    signal clk_50: std_logic;

    signal clk_slow: std_logic;

    ----- frequency and amplitude sweeping related signals ----
    signal   ramp_enable          : std_logic:='0';
    signal   amp_ramp_enable      : std_logic:='0';
    signal   amp_ramp_rate          : std_logic_vector(15 downto 0):= x
↪ "0000";
```

```
    ----- comparer1
    signal   comparer_dataa         : std_LOGIC_vector (63 downto 0);
    signal   comparer_datab         : std_logic_vector (63 downto 0);
    signal   comparer_aeb                   : std_logic;
    signal   comparer_agb           : std_logic;
    signal   comparer_alb           : std_logic;

    ----- comparer2
    signal   comparer_dataa2        : std_LOGIC_vector (63 downto 0);
    signal   comparer_datab2        : std_logic_vector (63 downto 0);
    signal   comparer_aeb2                  : std_logic;
    signal   comparer_agb2          : std_logic;
    signal   comparer_alb2          : std_logic;


    ---- amp comparer
    signal   amp_comparer_dataa1    : std_logic_vector (13 downto 0);
    signal   amp_comparer_datab1    : std_logic_vector (13 downto 0);
    signal  amp_comparer_dataa2     : std_logic_vector (13 downto 0);
    signal   amp_comparer_datab2    : std_logic_vector (13 downto 0);
    signal   amp_agb1                                       : std_logic
↪ ;
    signal   amp_agb2               : std_logic;


    ----- adder
    signal  adder_input                     : std_LOGIC_vector (63
↪ downto 0);
    signal  adder_direction                 : std_logic;
    signal   adder_output                   : std_logic_vector (63
↪ downto 0);
    signal   adder_step_buffer    : std_logic_vector (63 downto 0);
    signal   adder_step             : std_logic_vector (63 downto 0);

    ---- amp adder ---
```

```
        signal   amp_adder_direction    : std_logic;
        signal   amp_adder_output       : std_logic_vector (13 downto 0);
        signal   amp_adder_input        : std_logic_vector (13 downto 0);

        signal   ramping_flag                   : std_logic;
        signal   amp_ramping_flag     : std_logic;



begin
        pll: dds_pll port map (inclk0=>clk_dds, c0=>clk_50, c1 => clk_slow)
    ↪ ;

        --- assignment of dds bus in to various pins ---
        bus_in_address                          <= dds_bus_in(31 downto 28)
    ↪ ;
        bus_in_fifo_empty                       <= dds_bus_in(27);
        bus_in_ram_reset                                <= dds_bus_in(26);
        bus_in_step_to_next_value       <= dds_bus_in(25);
        bus_in_reset_dds_chip           <= dds_bus_in(24);
        dds_bus_out(0)                          <= bus_in_fifo_rd_en;
        dds_bus_out(1)                          <= bus_in_fifo_rd_clk;

        bus_in_fifo_rd_clk                      <= fifo_dds_rd_clk WHEN
    ↪ bus_in_address = dds_address else 'Z';
        bus_in_fifo_rd_en                               <= fifo_dds_rd_en
    ↪ WHEN bus_in_address = dds_address else 'Z';
        tx_enable                                               <= "11"
    ↪ WHEN bus_in_address = dds_address else "00";
        reset_fpga                                              <=
    ↪ bus_in_reset_dds_chip;
        fifo_dds_dout                                   <= dds_bus_in(15
    ↪ downto 0);
        fifo_dds_empty                          <= bus_in_fifo_empty;

        dds_ram_rdclock                         <= clk_dds;
```

```
    dds_ram_reset                                   <= bus_in_ram_reset
↪ ;
    dds_step_to_next_freq           <= bus_in_step_to_next_value;

    dds_address                                     <= add_in;

    led_VALUE (2 downto 0)          <= dds_ram_rdaddress(2 downto 0);
    led_VALUE (5 downto 3)          <= bus_in_address(2 downto 0);
    led_VALUE (6)                                   <=
↪ bus_in_fifo_empty;
    led_value(7)                                    <= ramping_flag or
↪ amp_ramping_flag;

    --- ground unused pins ---
    dds_osk                                                         <=
↪ '0';
    dds_drhold                                              <= '0';
    dds_drctl                                               <= '0';

    ----- Test DDS functionality ------
    f_pin                                                   <= "0000";
↪ --- Parallel programming mode
    ----- assign various data to the dds bus ---
    dds_port(31 downto 16)          <= par_data(15 downto 0);
    dds_port(15 downto 8)           <= par_add(7 downto 0);
    dds_port(0 downto 0)            <= par_16_bit;
    dds_port(1 downto 1)            <= par_rd;
    dds_port(2 downto 2)            <= par_wr;
    dds_port(7 downto 3)            <= "00000";

    target_amplitude <= dds_ram_data_out(31 downto 18);
    target_phase <= dds_ram_data_out(15 downto 0);
    main_phase <= target_phase;
    target_frequency <= dds_ram_data_out(127 downto 64);
```

```
    ---- frequency step for ramping
    adder_step_buffer (42 downto 27) <= dds_ram_data_out(63 downto 48);
    adder_step_buffer (63 downto 43) <= "00000000000000000000";
    adder_step_buffer (26 downto 0)  <= "00000000000000000000000000000";

    ramp_enable <= '0' WHEN dds_ram_data_out(63 downto 48) = x"0000"
↪ ELSE '1';

    --------- amplitude sweeper ------
    -- main amplitude is 14 bit --
    amp_ramp_rate                   <= dds_ram_data_out(47 downto 32);
↪
    --amp_ramp_rate <= x"4000";
    amp_ramp_enable                 <= '0' WHEN amp_ramp_rate = x"0000"
↪  ELSE '1';

    comparer_14bit_1: dds_14bit_compare port map (dataa=>
↪ amp_comparer_dataa1, datab=>amp_comparer_datab1, agb=>amp_agb1);
    adder_14bit:            dds_14bit_adder         port map (add_sub=>
↪ amp_adder_direction, dataa=>amp_adder_input, result=>amp_adder_output
↪ );

    --- amplitude ramper always add 1 to the amplitude scaling. We
↪ change the ramp rate by the chaning the timing ---
    process (clk_50)
            variable count:                         integer range 0 to
↪ 7:=0;
            variable sub_count:             integer range 0 to
↪ 65535:=0;
            variable delay:                         integer range 0 to
↪ 65535:=0;
            variable old_amp:               std_logic_vector(13 downto
↪ 0):="00000000000000";
            variable main_amplitude_var: std_logic_vector(13 downto 0)
↪ :="00000000000000";
            variable amp_ramp_up:   std_logic := '0';
```

```
    begin
            if rising_edge(clk_50) then
                    if amp_ramp_enable = '0' then
                            main_amplitude <= target_amplitude;
                            count := 0;
                            sub_count := 0;
                    else
                            CASE count IS
                                    --- update amplitude ---
                                    WHEN 0 =>       old_amp :=
↪ main_amplitude;
                                                                    ---
↪  feed into the comparer

↪ amp_comparer_dataa1 <= main_amplitude;

↪ amp_comparer_datab1 <= target_amplitude;

↪ count := count+1;
                                    --- do comparison ---
                                    WHEN 1 =>       if (amp_agb1='1')
↪ then

↪           amp_ramp_up := '0';

↪           amp_adder_direction <= '0';

↪ else

↪           amp_ramp_up := '1';

↪           amp_adder_direction <= '1';
                                                                    end
↪  if;

↪ count := count + 1;
```

```
↪ ---- update delay ---
↪ delay:= to_integer(unsigned(amp_ramp_rate));
                                        --- update adder input and delay
↪ ---
                                        WHEN 2 =>       amp_adder_input <=
↪ old_amp;
                                                                        --
↪ sub_count := sub_count + 1;
                                                                        if
↪ (sub_count = delay) then
↪         sub_count := 0;
↪         count := count + 1;
↪ else
↪         sub_count := sub_count + 1;
                                                                        end
↪  if;
                                                                        --
↪ count := count + 1;
                                        --- read from adder output
                                        WHEN 3 =>   old_amp:=
↪ amp_adder_output;
↪ count := count + 1;
                                        ---- test for overflow
                                        WHEN 4 =>   if (old_amp =
↪ target_amplitude) then
↪         main_amplitude_var := target_amplitude;
```

```
↪          amp_ramping_flag <= '0';
↪          count := 6;
↪ else
↪          main_amplitude_var := old_amp;
↪          amp_ramping_flag <= '1';
↪          count := count + 1;
                                                                    end
↪  if;
                                    ---- update amplitude output ---
                                    WHEN 5 =>       main_amplitude <=
↪ main_amplitude_var;
↪ count := 2;
                                    WHEN 6 =>       main_amplitude <=
↪ main_amplitude_var;
↪ count := count + 1;
                                    WHEN 7 =>       if (main_amplitude
↪ = target_amplitude) then
↪          null;
↪ else
↪          count := 0;
                                                                    end
↪  if;
                               end case;
```

```
                    end if;
            end if;
    end process;


    --------- frequency sweeper process ----------

    comparer_1: dds_compare port map (dataa=> comparer_dataa, datab=>
↪ comparer_datab, aeb=>comparer_aeb, agb=>comparer_agb, alb=>
↪ comparer_alb);
    comparer_2: dds_compare port map (dataa=> comparer_dataa2, datab=>
↪ comparer_datab2, aeb=>comparer_aeb2, agb=>comparer_agb2, alb=>
↪ comparer_alb2);
    adder: dds_add_subtract port map (add_sub => adder_direction, dataa
↪ => adder_input, datab=>adder_step ,result=> adder_output);

    process (clk_50)
            variable count: integer range 0 to 7:=0;
            variable old_freq: std_logic_vector(63 downto 0):=x
↪ "0000000000000000";
            variable main_frequency_var: std_logic_vector(63 downto 0)
↪ :=x"0000000000000000";
            variable ramp_up: std_logic := '0';
    begin
            if rising_edge(clk_50) then
                    if ramp_enable = '0' then  ---- normal no-ramp
↪ operation
                            main_frequency <= target_frequency;
                            count := 0;
                            ---led_value(2 downto 0) <= "000";
                    else                        ---- ramp operation
↪
                            CASE count IS
                                    ---- update frequency
                                    WHEN 0 => old_freq :=
↪ main_frequency;
```

```
                                                                --- feed
↪ into comparer ---
↪ comparer_dataa <= main_frequency;
↪ comparer_datab <= target_frequency;
                                                                count :=
↪ count + 1;
                                        ---- do comparision
↪
                                        WHEN 1 => if (comparer_agb='1')
↪ then
↪ ramp_up := '0';
                                                                    --
↪ led_value(3)<='0';
↪ adder_direction <= '0'; --- do subtraction
                                                                else
↪ ramp_up := '1';
                                                                    --
↪ led_value(3)<='1';
↪ adder_direction <= '1'; --- do addition
                                                                end if;
                                                                count :=
↪ count + 1;
                                                                --
↪ led_value(1)<='0';
                                                                adder_step
↪  <= adder_step_buffer;
                                        ---- update adder input
                                        WHEN 2 => adder_input<=old_freq;
```

```
                                                                count :=
↪ count + 1;
                                        ---- read from adder output
                                        WHEN 3 => old_freq:=adder_output;

↪ comparer_dataa2<=adder_output;

↪ comparer_datab2<=target_frequency;
                                                                count :=
↪ count + 1;
                                        ---- for over flow
                                        WHEN 4 => if (ramp_up = '0') then
↪ --- test for ramp down case

↪         if (comparer_agb2 = '1') then --- updated frequency is still
↪ larger than the target

↪                 main_frequency_var:=old_freq;

↪                 ramping_flag <= '1';

↪                 count := count + 1;

↪         else

↪                 main_frequency_var:=target_frequency;

↪                 ramping_flag <= '0';

↪                 count := 6;

↪         end if;
                                                                else ---
↪ test for ramp up case

↪         if (comparer_agb2 = '1') then --- updated frequency is
```

```
↪ already larger than the targer frequency
↪                 main_frequency_var:=target_frequency;
↪                 ramping_flag <= '0';
↪                 count :=6;
↪         else
↪                 main_frequency_var:=old_freq;
↪                 ramping_flag <= '1';
↪                 count := count + 1;
↪         end if;
                                                        end if;
                                    ---- update frequency output
                                    WHEN 5 => main_frequency<=
↪ main_frequency_var;
                                                        count :=
↪ 2;
                                    ---- update frequency then go to
↪ idle
                                    WHEN 6 => main_frequency<=
↪ main_frequency_var;
                                                        count :=
↪ count+1;

                                    ---- idle
                                    WHEN 7 => if (main_frequency =
↪ target_frequency) then
↪ null;
```

```
↪ --led_value(1) <= '1';
                                                                    else
↪          count := 0;
                                                                    end if;
                              END CASE;
                        end if;
              end if;
     end process;


     ------------------------------------------------




     par_16_bit <= "1";

     ps <= "000"; --- select profile 0
     par_rd <= "0";

     ram1: dds_ram port map (data=>dds_ram_data_in,
↪ rdaddress=>dds_ram_rdaddress,
↪ rdclock=>dds_ram_rdclock,
↪ wraddress=>dds_ram_wraddress,
↪ wrclock=>dds_ram_wrclock,
↪ wren=>dds_ram_wren,
                                                                    q=>
↪ dds_ram_data_out);

     ---- sample and condition the "step_to_next_value" signal from the
```

```
↪ pulser

    process (dds_step_to_next_freq, clk_dds)
    begin
            if rising_edge(clk_dds) then
                    if (dds_step_to_next_freq = '1') then
                            dds_step_to_next_freq_sampled <= '1';
                    else
                            dds_step_to_next_freq_sampled <= '0';
                    end if;
            end if;
    end process;

    ---dds_step_to_next_freq_sampled <= dds_step_to_next_freq;

    process (dds_step_to_next_freq_sampled, dds_ram_reset)
            variable dds_step_count: integer range 0 to 4095:=0;
    begin
                    if (dds_ram_reset = '1') then
                            dds_step_count:=0;
                    elsif (rising_edge(dds_step_to_next_freq_sampled))
↪ then
                            dds_step_count := dds_step_count+1;
                    end if;
                    dds_ram_rdaddress<=std_LOGIC_vector(to_unsigned(
↪ dds_step_count,12));
    end process;

    ---- read from pulser and write to RAM ---
    process (clk_system,dds_ram_reset)
            variable write_ram_address: integer range 0 to 32767:=0;
            variable ram_process_count: integer range 0 to 9:=0;
    begin
            ----- reset ram -----
            ----- This doesn't really reset the ram but only put the
↪ address to zero so that the next writing
```

```
            ----- from the fifo to the ram will start from the first
↪ address. Since each pulse will end with all zeros anyway
            ----- it's ok to have old information in the ram. The
↪ execution will never get past the end line.
            if (dds_ram_reset = '1') then
                    write_ram_address := 0;
                    ram_process_count := 0;
            elsif rising_edge(clk_system) then
                    case ram_process_count is
                            --------- first two prepare and check
↪ whether there is anything in the fifo. This can be done by looking at
↪  the pin
                            --------- fifo_pulser empty.
                            when 0 => fifo_dds_rd_clk <='1';
                                                    fifo_dds_rd_en <=
↪ '0';
                                                    dds_ram_wren <='0'
↪ ;
                                                    ram_process_count
↪ := 1;

                            when 1 => fifo_dds_rd_clk <='0';
                                                    ram_process_count
↪ := 2;

                            when 2 => if (bus_in_address = dds_address)
↪  then
                                                            if (
↪ fifo_dds_empty = '1') then ---- '1' is empty. Go back to case 0

↪ ram_process_count:=0;
                                                            else

↪ ram_process_count := 3; --2 ---- if there's anything in the fifo, go
↪ to the next case
                                                            end if;
```

```
                                                    else
↪ ram_process_count:=0;
                                                    end if;

                            -------- there's sth in the fifo ---------
                            when 3 => fifo_dds_rd_en <= '1';
                                                    ram_process_count
↪ :=4;
                            when 4 => fifo_dds_rd_clk <= '1';
↪ ------------- read from fifo --------------
                                                    dds_ram_wren <='1'
↪ ;
                                                    dds_ram_wrclock <=
↪  '1';
                                                    ram_process_count
↪ :=5;
                            when 5 => fifo_dds_rd_clk <= '0';
                                                    ram_process_count
↪ :=6;

                            ---------- prepare data and address that
↪ are about to be written to the ram------

                            when 6 => dds_ram_wraddress <=
↪ std_LOGIC_vector(to_unsigned(write_ram_address,15));
                                                    dds_ram_data_in <=
↪  fifo_dds_dout;
                                                    ram_process_count
↪ :=7;

                            when 7 => dds_ram_wrclock <= '0';
↪ ----------write to ram
                                                    ram_process_count
↪ :=8;
                            when 8 => write_ram_address:=
```

```
↪ write_ram_address+1; ----- increase address by one
                                                    ram_process_count
↪ :=9;
                            ----- check again if the fifo is empty or
↪ not. Basically this whole process will
                            ----- keep writing to ram until fifo is
↪ empty.
                            when 9 => if (fifo_dds_empty = '1') then

↪ ram_process_count:=0;
                                                    else

↪ ram_process_count:=3;
                                                    end if;
                    end case;
            end if;
    end process;

    ---- write instruction to DDS ---
    PROCESS (clk_50, reset_fpga)
            variable main_count: integer range 0 to 18:=0;
            variable sub_count: integer range 0 to 3:=0;
            variable main_frequency_var: std_LOGIC_VECTOR (63 downto 0)
↪ ;
            variable count_delay: integer range 0 to 65535:=0;
            variable main_amplitude_var: std_logic_vector(13 downto 0);
            variable main_phase_var: std_logic_vector(15 downto 0);
    BEGIN
            IF (reset_fpga = '1') then
                    main_count := 0;
                    count_delay :=0;
                    dds_master_reset <= '1';
            ELSIF (clk_50'event and clk_50='0') then
                    CASE main_count IS
                            ---- initialization. DDS chip reset ----
                            WHEN 0 => dds_io_update <= '0';
```

```
                                        dds_master_reset <='0';
                                                    main_count :=
↪ main_count+1;
                            WHEN 1 => dds_io_update <= '0';
                                        dds_master_reset <='1';
                                                    main_count :=
↪ main_count+1;
                            WHEN 2 => dds_io_update <= '0';
                                        dds_master_reset <='0';
                                                    main_count :=
↪ main_count+1;

                            ---- DAC calibration -----
                            WHEN 3 => CASE sub_count IS

↪ WHEN 0 => par_wr <= "1";

↪                           sub_count:=sub_count+1;

↪ WHEN 1 => par_add <=x"0F"; --- address for initial DAC calibration
↪ ---

↪        par_data <=x"0105";

↪                           sub_count:=sub_count+1;

↪ WHEN 2 => par_wr <= "0";

↪        sub_count:=sub_count+1;

↪ WHEN 3 => par_wr <= "1";

↪                           sub_count:=0;

↪                           main_count:=main_count+1;
                                                    END CASE;
```

```
↪

                            ----- delay -- wait 1 ms for DAC
↪ calibration to finish ----
                            WHEN 4 => if (count_delay = 50000) then
                                                          main_count
↪ := main_count+1;
                                                          count_delay
↪  := 0;
                                                    else
                                                      count_delay :=
↪ count_delay +1;
                                                    end if;
                            ---- clear DAC calibration ----

                            WHEN 5 => CASE sub_count IS

↪ WHEN 0 => par_wr <= "1";

↪                           sub_count:=sub_count+1;

↪ WHEN 1 => par_add <=x"0F"; --- address for initial DAC calibration
↪ ---

↪        par_data <=x"0005";

↪                           sub_count:=sub_count+1;

↪ WHEN 2 => par_wr <= "0";

↪        sub_count:=sub_count+1;

↪ WHEN 3 => par_wr <= "1";

↪                           sub_count:=0;
```

```
↪                          main_count:=main_count+1;
                                                    END CASE;
↪


                           ---- set up modlus mode ----

                           WHEN 6 => CASE sub_count IS

↪ WHEN 0 => par_wr <= "1";

↪                          sub_count:=sub_count+1;

↪ WHEN 1 => par_add <=CRF2_address;

↪        par_data <=CRF2_modulus;

↪                          sub_count:=sub_count+1;

↪ WHEN 2 => par_wr <= "0";

↪        sub_count:=sub_count+1;

↪ WHEN 3 => par_wr <= "1";

↪                          sub_count:=0;

↪                          main_count:=main_count+1;
                                                    END CASE;
↪


                           ----- enable amplitude tuning ---

                           WHEN 7 => CASE sub_count IS
```

```
↪ WHEN 0 => par_wr <= "1";
↪                          sub_count:=sub_count+1;
↪ WHEN 1 => par_add <=CRF1_address;
↪        par_data <=CRF1_enable_amp_scale;
↪                          sub_count:=sub_count+1;
↪ WHEN 2 => par_wr <= "0";
↪        sub_count:=sub_count+1;
↪ WHEN 3 => par_wr <= "1";
↪                          sub_count:=0;
↪                          main_count:=main_count+1;
                                                  END CASE;


                            ---- B -> 2^32 - 1 ---- fixed ----
                            WHEN 8 => CASE sub_count IS
↪ WHEN 0 => par_wr <= "1";
↪                          sub_count:=sub_count+1;
↪ WHEN 1 => par_add <=x"15";
↪        --par_data <=x"FFFF";
↪                          par_data <=x"0000";
```

```
                        sub_count:=sub_count+1;
WHEN 2 => par_wr <= "0";
        sub_count:=sub_count+1;
WHEN 3 => par_wr <= "1";
                        sub_count:=0;
                        main_count:=main_count+1;
                                                    END CASE;

                         WHEN 9 => CASE sub_count IS
WHEN 0 => par_wr <= "1";
                        sub_count:=sub_count+1;
WHEN 1 => par_add <=x"17";
        --par_data <=x"FFFF";
                        par_data <=x"8000";
                        sub_count:=sub_count+1;
WHEN 2 => par_wr <= "0";
        sub_count:=sub_count+1;
WHEN 3 => par_wr <= "1";
                        sub_count:=0;
```

```
↪                          main_count:=main_count+1;
                                                    END CASE;


                           -------- program A modulus mode testing

                           ---- A --> 1
                           WHEN 10 => CASE sub_count IS

↪ WHEN 0 => par_wr <= "1";

↪                          sub_count:=sub_count+1;

↪ WHEN 1 => par_add <=x"19";---0x19

↪        par_data <=main_frequency_var(16 downto 1);---

↪                          sub_count:=sub_count+1;

↪ WHEN 2 => par_wr <= "0";

↪        sub_count:=sub_count+1;

↪ WHEN 3 => par_wr <= "1";

↪                          sub_count:=0;

↪                          main_count:=main_count+1;
                                                    END CASE;

                           WHEN 11 => CASE sub_count IS

↪ WHEN 0 => par_wr <= "1";

↪                          sub_count:=sub_count+1;
```

```
↪ WHEN 1 => par_add <=x"1B";---0x1B
↪                          par_data(15) <='0';
↪         par_data(14 downto 0) <=main_frequency_var(31 downto 17);---
↪                          sub_count:=sub_count+1;
↪ WHEN 2 => par_wr <= "0";
↪         sub_count:=sub_count+1;
↪ WHEN 3 => par_wr <= "1";
↪                          sub_count:=0;
↪                          main_count:=main_count+1;
                                                    END CASE;

                           ----- set frequency tuning word ----

                           WHEN 12 => CASE sub_count IS
↪ WHEN 0 => par_wr <= "1";
↪                          sub_count:=sub_count+1;
↪ WHEN 1 => par_add <=x"11";
↪         par_data <=main_frequency_var(47 downto 32);---
↪                          sub_count:=sub_count+1;
↪ WHEN 2 => par_wr <= "0";
↪         sub_count:=sub_count+1;
```

```
↪ WHEN 3 => par_wr <= "1";
↪                          sub_count:=0;
↪                          main_count:=main_count+1;
                                                    END CASE;

                            WHEN 13 => CASE sub_count IS
↪ WHEN 0 => par_wr <= "1";
↪                          sub_count:=sub_count+1;
↪ WHEN 1 => par_add <=x"13";
↪        par_data <=main_frequency_var(63 downto 48);---
↪                          sub_count:=sub_count+1;
↪ WHEN 2 => par_wr <= "0";
↪        sub_count:=sub_count+1;
↪ WHEN 3 => par_wr <= "1";
↪                          sub_count:=0;
↪                          main_count:=main_count+1;
                                                    END CASE;

                            ----- set phase ----

                            WHEN 14 => CASE sub_count IS
↪ WHEN 0 => par_wr <= "1";
```

```
                        sub_count:=sub_count+1;
WHEN 1 => par_add <=x"31";
        par_data <=main_phase_var;---
                        sub_count:=sub_count+1;
WHEN 2 => par_wr <= "0";
        sub_count:=sub_count+1;
WHEN 3 => par_wr <= "1";
                        sub_count:=0;
                        main_count:=main_count+1;
                                                END CASE;


                         ----- set amplitude -----
                         WHEN 15 => CASE sub_count IS
WHEN 0 => par_wr <= "1";
                        sub_count:=sub_count+1;
WHEN 1 => par_add <="00110011";
                        if (main_amplitude_var = "00000000000000")
then
                                    par_data <= x"0000";
```

```
↪                          else
↪                                       par_data <= x"0FFF";
↪                          end if;
↪                          sub_count:=sub_count+1;
↪ WHEN 2 => par_wr <= "0";
↪        sub_count:=sub_count+1;
↪ WHEN 3 => par_wr <= "1";
↪                          sub_count:=0;
↪                          main_count:=main_count+1;
                                                    END CASE;

                            WHEN 16 => dds_io_update <='0';
                                                    main_count :=
↪ main_count+1;
                            WHEN 17 => dds_io_update <='1';
                                                    main_count :=
↪ main_count+1;
                            WHEN 18 =>      dds_io_update <='0';
                                                        if (
↪ main_frequency_var = main_frequency) then
                                                                if
↪ (main_phase_var = main_phase) then
↪         if (main_amplitude_var = main_amplitude) then
↪                 null;
↪         else
```

```
↪                 main_amplitude_var := main_amplitude;
↪                 main_count:=15;
↪         end if;
↪ else
↪         main_amplitude_var:=main_amplitude;
↪         main_phase_var:=main_phase;
↪         main_count:=14;
                                                                    end
↪  if;
                                                          else
↪ main_frequency_var:=main_frequency;
↪ main_amplitude_var:=main_amplitude;
↪ main_phase_var:=main_phase;
↪ main_count:=10;
                                                          end if;
                    end case;
            END IF;
    END PROCESS;

    ---- write to DAC for amplitude tuning ----

    PROCESS (clk_50)
            VARIABLE main_count: INTEGER range 0 to 3:=0;
            VARIABLE main_amplitude_var : STD_LOGIC_VECTOR (13 downto
↪ 0);
```

```
    BEGIN
            IF (clk_50'event and clk_50='0') then
                    CASE main_count IS
                            WHEN 0 => dac_out <= main_amplitude_var;
↪ -----set DAC amplitude
                                                            dac_wr_pin <= '0';
                                                            main_count:=1;
                            WHEN 1 => dac_wr_pin <= '1'; -------------
↪ write to dac for amplitude
                                            main_count:=2;
                            WHEN 2 => dac_wr_pin <= '0'; -------------
↪ write to dac for amplitude
                                            main_count:=3;
                            WHEN 3 => if (main_amplitude_var =
↪ main_amplitude) then
                                                              null;
                                                            else

↪ main_amplitude_var := main_amplitude;
                                                                    main_count
↪ :=0;
                                                          end if;
                    END CASE;
            END IF;
    END PROCESS;




    ------- generate slower clock --------
    process (clk_50)
            variable count: integer range 0 to 21 :=0;
    begin
            if (rising_edge(clk_50)) then
```

```
                        count := count + 1;
                        if (count <= 10) then
                                clk_system <= '1';
                        elsif (count <= 20) then
                                clk_system <= '0';
                        elsif (count=21) then
                                count :=0;
                        end if;
                end if;
        end process;

        --- Write LED data to the TI converter chip --

        PROCESS
                VARIABLE count_serial: INTEGER RANGE 0 to 19:=0;
        BEGIN
                WAIT UNTIL (clk_system'EVENT AND clk_system='1');
                CASE count_serial IS
                        WHEN 0  => LED_OE <= '0'; LED_LE <= '0'; LED_CLK <=
↪  '0';
                        WHEN 1  => LED_SDI <= LED_VALUE (7 DOWNTO 7);
↪ LED_CLK <= '0';---- first----
                        WHEN 2  => LED_CLK <= '1';
                        WHEN 3  => LED_SDI <= LED_VALUE (6 DOWNTO 6);
↪ LED_CLK <= '0';
                        WHEN 4  => LED_CLK <= '1';
                        WHEN 5  => LED_SDI <= LED_VALUE (5 DOWNTO 5);
↪ LED_CLK <= '0';
                        WHEN 6  => LED_CLK <= '1';
                        WHEN 7  => LED_SDI <= LED_VALUE (4 DOWNTO 4);
↪ LED_CLK <= '0';
                        WHEN 8  => LED_CLK <= '1';
                        WHEN 9  => LED_SDI <= LED_VALUE (3 DOWNTO 3);
↪ LED_CLK <= '0';
                        WHEN 10  => LED_CLK <= '1';
                        WHEN 11  => LED_SDI <= LED_VALUE (2 DOWNTO 2);
```

```
    ↪ LED_CLK <= '0';
                        WHEN 12  => LED_CLK <= '1';
                        WHEN 13  => LED_SDI <= LED_VALUE (1 DOWNTO 1);
    ↪ LED_CLK <= '0';
                        WHEN 14  => LED_CLK <= '1';
                        WHEN 15  => LED_SDI <= LED_VALUE (0 DOWNTO 0);
    ↪ LED_CLK <= '0';---- last bit----
                        WHEN 16  => LED_CLK <= '1';
                        WHEN 17 => LED_OE <= '0';LED_LE <= '1';
                        WHEN 18 => LED_OE <= '0';LED_LE <= '0';
                        WHEN 19 => LED_OE <= '0';LED_LE <= '0';
                END CASE;
                count_serial := count_serial +1;
                IF (count_serial = 18) THEN
                        count_serial :=0;
                END IF;
        END PROCESS;


end behaviour;
```

บทผนวก ข

คู่มือการทดลองที่ใช้ในการเรียนการสอนเรื่อง thermal noise

บทผนวกนี้เป็นคู่มือการทดลองเรื่อง thermal noise ที่ใช้ในการเรียนการสอนวิชาปฏิบัติการฟิสิกส์ สำหรับนักศึกษา ป.ตรี (ปี 3) ภาควิชาฟิสิกส์ คณะวิทยาศาสตร์ มหาวิทยาลัยมหิดล

CHAPTER 3

ปฏิบัติการ 1: การวัดค่า k_B จากตัวต้านทาน

3.1 บทนำ

อนุภาคที่อยู่ในวัตถุที่มีอุณหภูมิที่มากกว่าศูนย์เคลวิน จะมีการเคลื่อนที่แบบสุ่ม (random) ซึ่งในบางกรณี การเคลื่อนที่แบบสุ่มนี้ มีผลมากพอที่จะทำให้เราสังเกตมันได้โดยไม่ต้องใช้เครื่องมือราคาแพงหรือ มีความซับซ้อนสูง

ในตัวนำไฟฟ้า การเคลื่อนที่แบบสุ่มของอิเลคตรอน ทำให้เกิด noise หรือ สัญญาณรบกวนในวงจรไฟฟ้า ซึ่ง noise ที่เกิดจากอุณหภูมิที่ไม่เป็นศูนย์นี้ (เรียกว่า thermal noise หรือ Johnson noise) เราไม่สามารถกำจัดมันให้หมดไปได้ เพราะเป็น fundamental noise นอกเหนือจากการลดอุณหภูมิของระบบเท่านั้น นี่เป็นเหตุผลว่า ทำไมในหลายๆกรณี การลดอุณหภูมิของวงจรไฟฟ้าเป็นสิ่งที่จำเป็น เช่น การถ่ายภาพของท้องฟ้าในเวลากลางคืน ในหลายๆครั้ง เราจะลดอุณหภูมิของตัวรับแสงเพื่อลด noise ในการถ่ายภาพ

การทดลองนี้ เราจะวัด noise (สัญญาณรบกวน) ของความต่างศักย์คร่อมตัวต้านทาน โดยสัญญาณรบกวนนี้ เป็นผลเนื่องมาจากการเคลื่อนที่แบบสุ่มของอิเล็กตรอนในตัวต้านทาน เราจะวัดขนาดของ noise สำหรับตัวต้านทานที่มีค่าต่างๆกัน เพื่อที่จะหาว่าค่าคงที่ของ Boltzmann (k_B) นั้น มีค่าเท่าใด

3.2 รายละเอียดการทดลอง

3.2.1 ทฤษฎีที่เกี่ยวข้อง

เราสามารถพิสูจน์ได้ว่า (อ้างอิงจาก [1] หรือหนังสือเรื่องฟิสิกส์เชิงสถิติทั่วไป) ถ้าเราวัดความต่างศักย์คร่อมตัวต้านทานที่มีความต้านทาน R เราจะวัดค่าความต่างศักย์ root-mean-square ของ noise ที่เกิดจากตัวต้านทาน ได้

$$V_{\text{RMS}} = \sqrt{4k_B T R \Delta f} \tag{3.2.1}$$

โดยที่ k_B คือค่าคงที่ของ Boltzmann $\approx 1.3806 \times 10^{-23}$ J/K และ T คืออุณหภูมิของตัวต้านทาน ส่วนปริมาณ Δf นั้น เรียกว่า bandwidth ซึ่งเป็นช่วงของความถี่ของสัญญาณที่เราวัด มีหน่วยเดียวกันกับความถี่ สาเหตุที่มีปริมาณนี้อยู่ในสมการเป็นเพราะว่า ตอนที่เราคำนวน noise ในระบบนั้น เราแบ่งแยกพิจารณาทีละความถี่ของสัญญาณไฟฟ้า ในตอนท้ายสุดเราต้องรวมสัญญาณไฟฟ้าของทุกความถี่ที่เราสนใจเข้าด้วยกัน เวลาที่เราพูดถึงปริมาณที่อธิบายถึง noise โดยมากเราจะเขียนอยู่ในรูปของ ความต่างศักย์กำลังสอง ต่อ ช่วงความถี่ 1 Hertz หรือค่ารากที่สองของปริมาณที่ว่านี้ อย่างเช่น ในกรณีที่ R = 1 kΩ และ T = 300 K เราจะใช้ $\Delta f = 1$ Hz เราจะเขียนว่า ความต่างศักย์รบกวน (voltage noise) ของตัวต้านทาน คือ

$$V_{\text{RMS}} = 4.07 \times 10^{-9} \text{ V}/\sqrt{\text{Hz}} \tag{3.2.2}$$

ความคุ้นเคยต่อการเขียน noise โดยที่หน่วยเป็นปริมาณบางอย่าง ต่อ $\sqrt{\text{Hz}}$ แบบนี้เป็นสิ่งที่สำคัญมากๆ เพราะเราต้องเข้าใจว่า เวลาที่บอกว่า noise = 4.1 nV/$\sqrt{\text{Hz}}$ ไม่ได้แปลว่า ตอนที่เราเอามัลติมิเตอร์ไปวัดความต่างศักย์คร่อมตัวต้านทานแล้วเราจะวัดได้ 4.1 nV แต่ว่าเราต้องกำหนดก่อนว่า เราจะรวม noise ตั้งแต่ความถี่เท่าไหร่ถึงความถี่เท่าไหร่ก่อน สมมุติว่า เราสนใจ noise ตั้งแต่ $f = 0$ Hz ถึง 1 kHz ปริมาณของ noise ทั้งหมด จะคิดได้โดยแทน $\Delta f = 1$ kHz ในสมการที่ (3.2.1)

สมการที่ (3.2.1) นั้น เป็นผลที่ได้จากการประมาณในกรณีที่ ค่าความต้านทานไม่สูงจนเกินไป เพราะในความเป็นจริงแล้ว ต้องคิดถึงค่าความจุไฟฟ้า (capacitance) ของตัวต้านทานด้วย

3.2.2 การทดลอง

บทนำการทดลอง

จากผลที่ได้ในสมการที่ (3.2.2) เราจะเห็นได้ว่า ค่าความต่างศักย์ที่เกิดจาก thermal noise นั้นมีค่าน้อยมากๆ อยู่ในระดับ nV เลยทีเดียว เครื่องมือวัดทั่วๆไปที่เราใช้ในห้องทดลองที่วัดความต่างศักย์ได้ ก็จะมีมัลติมิเตอร์ หรือ ออสซิลโลสโคป (เรียกย่อๆว่า สโคป) โดยเครื่องมือสองประเภทนี้ วัดค่าความต่างศักย์ได้อย่างน้อยที่สุดก็อยู่ในระดับ mV ดังนั้น การที่เราจะวัดค่าความต่างศักย์ของ thermal noise เราต้องมีวงจรขยายสัญญาณจาก nV มาอยู่ที่ระดับ mV

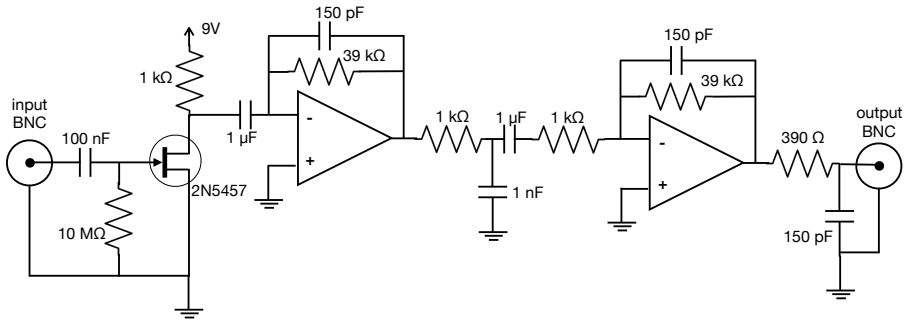


Figure 3.1: วงจรขยายสัญญาณที่ใช้ในการทดลองโดยปรับเปลี่ยนจากวงจรต้นแบบใน [2] ตัวต้านทานในส่วนที่เป็น input BNC เป็นตัวกำเนินสัญญาณ noise ที่เราต้องการวัด

วงจรขยายสัญญาณที่เราใช้นั้น มีต้นแบบมาจาก [2] โดยที่ตัววงจรนั้นแสดงอยู่ในรูปที่ (3.1) เราไม่จำเป็นต้องเข้าใจการทำงานของวงจรทั้งหมด แต่เราต้องทราบว่า วงจรขยายของเรานั้นมีอัตราขยาย (gain) เท่าไหร่ และ มีการตอบสนองเชิงความถี่เท่าไหร่ ในความเป็นจริง แทนที่จะใช้ปริมาณ Δf ในสมการที่ (3.2.1) โดยตรง เราจะคำนวนปริมาณที่เรียกว่า effective gain bandwidth ซึ่งเขียนได้ว่า

$$\Delta f' = \int_0^\infty [G(f)]^2 df \tag{3.2.3}$$

โดยที่ $G(f)$ เป็นอัตราขยายของวงจรขยายที่ขึ้นอยู่กับความถี่ ดังนั้น ความต่างศักย์ของ noise ที่เกิดจากตัวต้านทานหลังจากที่ผ่านวงจรขยายแล้ว มีค่าเท่ากับ

$$V_{\text{RMS}} = \sqrt{4k_B T R \int_0^\infty [G(f)]^2 df} \tag{3.2.4}$$

จุดมุ่งหมายของการทดลองนี้คือ เราจะทำการวัดความต่างศักย์ของ noise ของตัวต้านทานที่มีค่าต่างๆกัน เพื่อที่จะดูว่า มีความสัมพันธ์แบบ $V_{\text{RMS}} \propto \sqrt{R}$ อย่างที่แสดงในสมการที่ (3.2.1) หรือไม่ จากนั้นเราจะคำนวนจากการทดลองของเราเพื่อดูว่าค่าคงที่ของ Boltzmann k_B ที่เราวัดได้จากการทดลอง มีค่าเท่าไรเทียบกับค่าจริง

3.2.3 ขั้นตอนการทดลอง

วัดการตอบสนองเชิงความถี่และอัตราการขยายของวงจรขยายสัญญาณ

ในขั้นตอนแรก เราจะวัดการตอบสนองเชิงความถี่ของวงจรขยายโดยการใส่สัญญาณคลื่น sine ที่เราทราบค่าแอมปลิจูดที่แน่นอนจาก function generator ไปยัง input ของเครื่องขยายสัญญาณ แล้วดูสัญญาณหลังจากที่ขยายแล้วโดยใช้สโคป

ขั้นตอนคร่าวๆมีดังนี้

- สร้างความคุ้นเคยในการใช้งานเครื่อง function generator โดยการต่อสัญญาณเข้ากับสโคปโดยตรง ดูว่า ค่าความถี่ที่ปรับได้ อยู่ในช่วงไหน แอมปลิจูดที่เราปรับได้ อยู่ในระดับกี่ V
- ลองทำความคุ้นเคยกับฟังก์ชั่นการวัดค่า RMS ของสโคป โดยการกดปุ่ม measure บนสโคป
- ขั้นตอนไปเราจะใส่สัญญาณคลื่น sine จาก function generator เข้าไปใน input ของวงจรขยาย และใช้ สโคปวัด output ของวงจรขยาย เพื่อที่จะวัดอัตราขยายของวงจร โดยที่วงจรขยายของเรานั้น ใช้ถ่านไฟฉาย 9V สองก้อนเป็น แหล่งกำเนิดไฟฟ้า ดังนั้น **สัญญาณ output สุดท้ายของวงจรนั้น จะไม่สามารถมีค่าสูงไปกว่า $\pm$ 9 V หรือ 18 $V_{p\text{-}p}$** อัตราขยายของวงจรของเราจะอยู่ที่ประมาณ 3,000 ให้ลองพิจารณาดูว่า แอมปลิจูดของสัญญาณที่เราใส่เข้าไปใน input ของวงจรขยายควรจะมีค่าไม่มากไปกว่าเท่าใด
- เมื่อเราปรับแอมปลิจูดที่ใส่ใน input ของวงจรขยายให้พอเหมาะแล้ว ลองดู output ของวงจรขยายโดยใช้สโคป (ระวังไม่ให้ output มีค่ามากเกินไปจนทำให้สัญญาณถูกขลิบออก ในกรณีสัญญาณ output จะมีหน้าตาเหมือน square wave แทนที่จะเป็น sine wave) แล้วลองปรับความถี่ของคลื่นดู แอมปลิจูดของ output ควรจะคงที่จนถึงค่าความถี่ค่านึงแล้วจะลดลงเรื่อยๆ จนเข้าใกล้ศูนย์ **อย่าลืมบันทึกว่า ขนาดของสัญญาณ input มีค่าเท่าใด เพราะว่าต้องใช้ในการคำนวนหาอัตราขยาย**

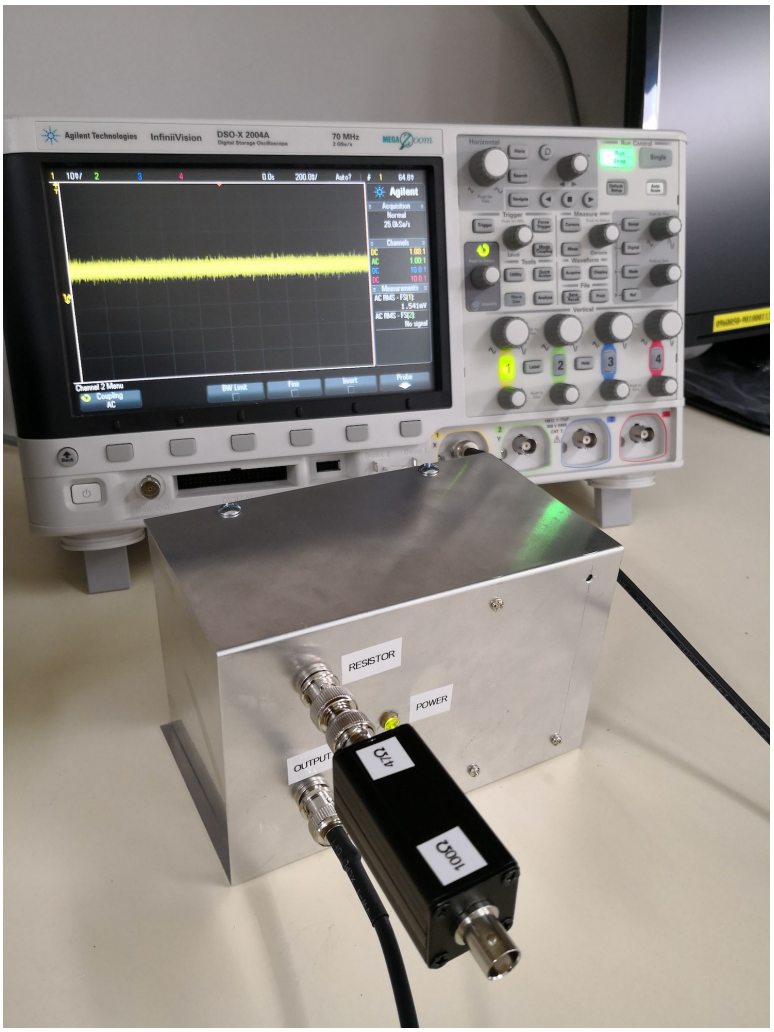


Figure 3.2: ตัวอย่างการวัด

- บันทึกค่าแอมปลิจูดของ output ที่ค่าความถี่ต่างๆ ช่วงความถี่ที่ควรจะวัดควรจะครอบคลุมระหว่าง 0 Hz ถึงประมาณ 100 kHz **เนื่องจากช่วงความถี่ที่วัดมีค่าค่อนข้างกว้าง คำแนะนำคือให้วัดที่ประมาณ 10, 20, 40, 70, 100, 200, 400, 700, 1000 Hz ไปตามลำดับ จนกว่าจะถึง 100 kHz หรือ 200 kHz**
- พล็อตกราฟค่าอัตราขยาย (gain = output / input) และความถี่ โดยที่ให้แกนนอน เป็น scale แบบ logarithm
- คำนวนหา effective gain bandwidth โดยคร่าวๆด้วยการหาพื้นที่ใต้กราฟระหว่างอัตราขยายยกกำลังสอง และความถี่ หรืออีกนัยหนึ่งคือการคำนวนหา $\int_0^\infty [G(f)]^2 df$ นั่นเอง

Figure 3.3: ตัวอย่างการวัด

วัดค่าความต่างศักย์ที่เกิดจาก noise ของตัวต้านทาน

ในขั้นตอนนี้ เราจะวัด noise จากตัวต้านทานจริงๆ (ตัวอย่างในรูปที่ (3.2 และ 3.3)) โดยตัวต้านทานค่าต่างๆนั้น ได้ใส่อยู่ในกล่องสีดำ อย่างที่แสดงในรูปที่ (3.4)

- เริ่มต้นด้วยการใส่ $R = 0\ \Omega$ (ดังที่แสดงในรูปที่ (3.5)) ที่ input ของเครื่อง แล้ว วัด output ของวงจรขยายโดยใช้สโคป เราจะเห็นว่า noise จะไม่เป็นศูนย์ (ทำไม?) ให้เราวัดค่าความต่างศักย์ RMS โดยใช้ฟังก์ชั่นการวัด V_{RMS} ของสโคปโดยตรง ค่านี้จะเป็นสิ่งที่เรียกว่า background noise (V_{bg}) ที่เราจะต้องลบออกสำหรับ noise ที่ความต้านทานค่าอื่นๆ
- วัดค่า noise สำหรับตัวต้านทานให้ครบทุกค่า

การวิเคราะห์ข้อมูล

หลังจากที่เสร็จสิ้นการวัดแล้ว ที่เหลือเป็นการวิเคราะห์ข้อมูล สิ่งที่ควรจะทำ(อย่างน้อย) คือ

- พล็อตกราฟระหว่าง V_{RMS}^2 และ R (อย่าลืมลบ V_{bg}^2 ออกจาก V_{RMS}^2 เพื่อแสดง noise ที่มาจากตัวต้านทานจริงๆ) นักศึกษาควรจะพล็อตกราฟโดยใช้ scale แบบ logarithm ทั้งสองแกน เนื่องจากข้อมูลที่เรามี อยู่ในช่วงที่กว้างมาก
- จาก effective gain bandwidth ที่คำนวนได้ในขั้นตอนแรก ให้วิเคราะห์ว่าค่าคงที่ของ Boltzmann ที่เราวัดได้มีค่าเท่าไหร่ และมีค่าความคลาดเคลื่อนเท่าไหร่

Figure 3.4: กล่องใส่ตัวต้านทาน

3.2.4 คำแนะนำทั่วไปและข้อควรระวัง

- วงจรขยายนั้น มีสวิตช์เปิดปิดอยู่ ตอนที่เปิด หลอด LED สีเหลืองจะติด ถ้าเกิด หลอด LED ไม่ติด ให้ลองเปลี่ยนถ่านดู ให้ปิดสวิตช์ไว้ตอนที่ไม่ใช้วงจรเพื่อประหยัดถ่าน
- กล่องโลหะของวงจรขยายมีไว้เพื่อกักไม่ให้คลื่นรบกวนจากภายนอกมีผลต่อสัญญาณที่เราจะวัด ดังนั้นถ้าเป็นไปให้ ให้ปิดกล่องให้เรียบร้อยในขณะที่วัด
- ในเมื่อค่าความต้านทานที่เราใช้วัดมีค่าตั้งแต่ 100 Ω ถึง 100 kΩ การพล็อตกราฟระหว่างค่าความต่างศักย์ของ noise กับ ค่าความต้านทานนั้น ไม่ควรจะใช้สเกลที่เป็นเชิงเส้น เพราะจะทำให้ดูข้อมูลยาก
- สังเกตดูสัญญาณที่วัดได้บนสโคปว่าขึ้นกับอะไรบ้าง เช่น ตำแหน่งของมือเราว่าได้แตะกล่องของตัวขยายสัญญาณหรือไม่ พยายามทำให้การวางของอุปรณ์หรือเงื่อนไขต่างๆคงเดิมเวลาที่เปลี่ยนตัวต้านทาน

3.3 คำถามก่อนการทดลอง

1. ค่าคงที่มูลฐานที่เราจะวัดในการทดลองคืออะไร มีค่าประมาณเท่าใด
2. V_{RMS} มีนิยามว่าอะไร ทำไมเราถึง V_{RMS} ของ noise แทนที่จะวัดค่าความต่างศักย์เฉลี่ยของ noise
3. จากสมการ (3.2.1) เราควรจะพล็อตกราฟระหว่างปริมาณใดเพื่อที่จะหาค่า k_B

Figure 3.5: ตัวต้านทาน $R = 0\ \Omega$

3.4 คำถามเพิ่มเติม

- อ้างอิงจากสมการที่ (3.2.1) ตอนที่เราไม่ได้ต่อตัวต้านทานใดๆไปที่ input ของวงจรขยาย หมายความว่า $R = \infty$ ซึ่งแปลว่า noise ก็ควรจะมีค่าเป็นอนันต์เหมือนกัน (แล้วควรจะเกิดระเบิดอย่างรุนแรง) ทำไมเราถึงไม่ได้วัดค่า noise เป็นอนันต์
- ตอนที่เรา short input โดยใช้ตัวต้านทาน $R = 0$ เราก็ยังเห็น noise ที่วัดได้บนสโคปอยู่ noise พวกนี้มาจากไหน
- เราสามารถใช้การทดลองนี้ หาค่าอุณหภูมิศูนย์สมบูรณ์ (absolute zero) ได้อย่างไร

3.5 Checklist

3.5.1 เมื่อเสร็จสิ้นสัปดาห์ที่ 1

- เก็บข้อมูลอัตราขยาย (gain) สำหรับสัญญาณ input ที่ความถี่ต่างๆ

3.5.2 เมื่อเสร็จสิ้นสัปดาห์ที่ 2

- เก็บข้อมูล noise สำหรับตัวต้านทานทุกตัวเสร็จ

บทผนวก ฆ

Reprint ผลงานตีพิมพ์

บทผนวกนี้เป็น reprint ของผลงานตีพิมพ์ในเรื่อง “A robust experimental setup for Johnson noise measurement suitable for advanced undergraduate students” ซึ่งตีพิมพ์ใน European Journal of Physics

European Journal of Physics

PAPER

A robust experimental setup for Johnson noise measurement suitable for advanced undergraduate students

To cite this article: Thaned Pruttivarasin 2018 *Eur. J. Phys.* **39** 065102

IOP Publishing European Journal of Physics
Eur. J. Phys. **39** (2018) 065102 (10pp) https://doi.org/10.1088/1361-6404/aadc8b

A robust experimental setup for Johnson noise measurement suitable for advanced undergraduate students

Thaned Pruttivarasin

Department of Physics, Faculty of Science, Mahidol University, 272 Rama VI Rd., Ratchathewi District, Bangkok, 10400, Thailand
Thailand Center of Excellence in Physics, Commission on Higher Education, 328 Si Ayutthaya Road, Bangkok, 10400, Thailand

E-mail: thaned.pru@mahidol.edu

Received 22 May 2018, revised 11 August 2018
Accepted for publication 23 August 2018
Published 24 September 2018

Abstract

We present a durable and inexpensive construction of a Johnson noise experiment suitable for advanced undergraduate students. By measuring the root-mean-square voltage noises for different resistors and taking into account the input capacitance of the amplifier circuit, we demonstrate that the experimental setup is capable of measuring the value of the Boltzmann's constant to be $k_B = (1.381 \pm 0.009) \times 10^{-23}$ J K^{-1}. The setup also allows us to determine the capacitance of an RG-58/U cable to be 80 ± 3 pF m^{-1}. We also provide observations from two years of teaching this experiment, containing common student pitfalls. Finally, we discuss possible adjustments of the experiment according to a time constraint.

Keywords: thermal noise, Boltzmann's constant, undergraduate physics

(Some figures may appear in colour only in the online journal)

1. Introduction

Johnson noise or thermal noise in a resistor is one of the earliest kinds of noise that students encounter in their undergraduate course in physics [1–5]. The derivation of the noise is within the scope of the first course in statistical mechanics. To be able to relate the amplitude of the noise to a fundamental constant such as the Boltzmann's constant is rather extraordinary [6]. It shows students that one can extract useful information from noise, which is usually regarded as an unwanted signal from any measurements.

Since the resistor thermal noise is quite small, we usually need an amplifier to be able to measure the noise comfortably. A beautifully designed amplifier circuit by John Geller [7] achieves this task by a high-input-impedance triple stage amplification (JFET-Op Amp-Op Amp) [8], and the signal is large enough to be measurable by either a decent voltmeter or an oscilloscope.

In this work, we construct an experiment utilizing a high-input-impedance amplifier circuit [7], and design the apparatus to withstand weekly abusing from undergraduate students by means of separated resistor boxes and Bayonet NeillConcelman (BNC) connectors. The construction allows us to conveniently insert an RG-58/U cable to vary the input capacitance of the circuit. By fitting the result using a model including an input capacitance, we are able to directly see the effect of the input capacitance on the response of the amplifier circuit and extract the value of the input capacitance. Changing the length of the input RG-58/U cable allows us to measure the capacitance of the RG-58/U cable. We also provide observations and suggestions from two years (2016–2017) of teaching this experiment in a 3rd-year undergraduate course at the Physics Department, Faculty of Science, Mahidol University.

2. Theoretical considerations

A resistor placed in a thermal equilibrium at temperature T produces voltage noise across its two terminals. This so-called Johnson (or Johnson-Nyquist) noise originates from random motions of electrons in the resistor. The root-mean-square of this voltage noise is given by a simple formula:

$$V_{\mathrm{RMS}}^2 = 4k_{\mathrm{B}}TR\Delta f, \tag{1}$$

where k_{B} is the Boltzmann's constant, R is the resistance of the resistor and Δf is the noise bandwidth given by

$$\Delta f = \int_0^\infty \frac{[G(f)]^2}{1 + (2\pi fRC)^2}\mathrm{d}f \tag{2}$$

where C is the combined input capacitance of the amplifier and $G(f)$ is the gain profile of the circuit which depends on the frequency, f, of the noise. The derivation of this formula can be found in many standard textbooks [9–11].

Our goal is to perform a measurement to determine the values of k_{B} and C. The experimental procedure involves first independently determining the gain profile of the circuit, $G(f)$, using a known source of sinusoidal signal with varying frequency. Then, the root-mean-square noise is measured for different values of resistors. The values of k_{B} and C can be extracted by fitting the data using equations (1) and (2).

3. Construction of the device

The amplifier circuit is based on the circuit by Geller [7] with the main JFET replaced by a different part number, as shown in figure 1. The original JFET in Geller's circuit, which is BF244A, has become obsolete. We have found a suitable replacement to be 2N5457, which is used in our circuit.

The circuit and a two-9V-battery power-supply are enclosed within a folded aluminum box as shown in figure 2. The input and output of the circuit are connected to panel-mounted grounded BNC connectors. (It is important that the BNC connectors are grounded.) The panel of the box in shown in figure 3. The input BNC connector to the amplifier allows us to

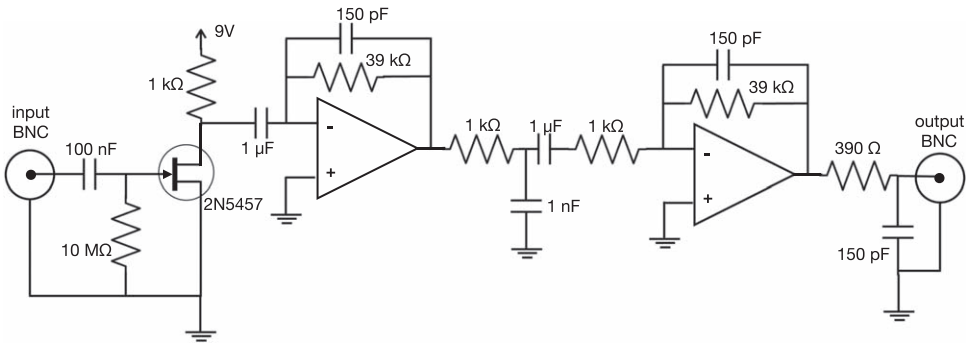


Figure 1. The amplifier circuit used in the experiment.

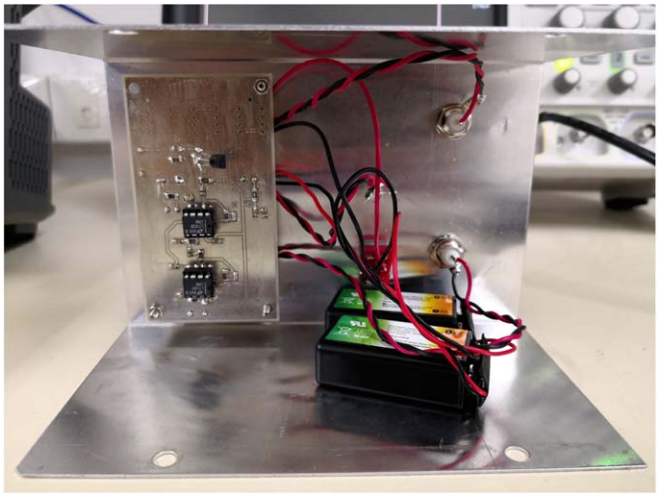


Figure 2. Inside the amplifier box. The ground of the circuit must be grounded to the box.

directly connect either a function generator (to measure the gain profile) or resistor boxes (to measure the noise of resistors).

We chose regular metal-film resistors ranging from 50 Ω to 1 MΩ for the measurement. Each aluminum resistor box (as shown in figure 4) contains two resistors to reduce the number of the boxes needed. Although this results in a system that is not completely enclosed inside a metal cage when measuring a single side of the resistor box, we did not find any influence on the measurement. One can also short the unused side of the resistor box when performing a measurement.

To change the input capacitance, we can insert an RG-58/U cable between the resistor box and the amplifier circuit. This allows us to fine tune the input capacitance of the amplifier circuit.

We found that BNC connectors can withstand multiple connections and disconnections from our students. In our first attempt with the device, we used alligator clips for the input resistors and the connections failed once every 2–3 weeks of repeated usage.

We note that the total cost of the construction of the experiment (not including the oscilloscope and the function generator) is less than 1,500 bahts (∼45 USD). All components are standard components available in common electronics shops.

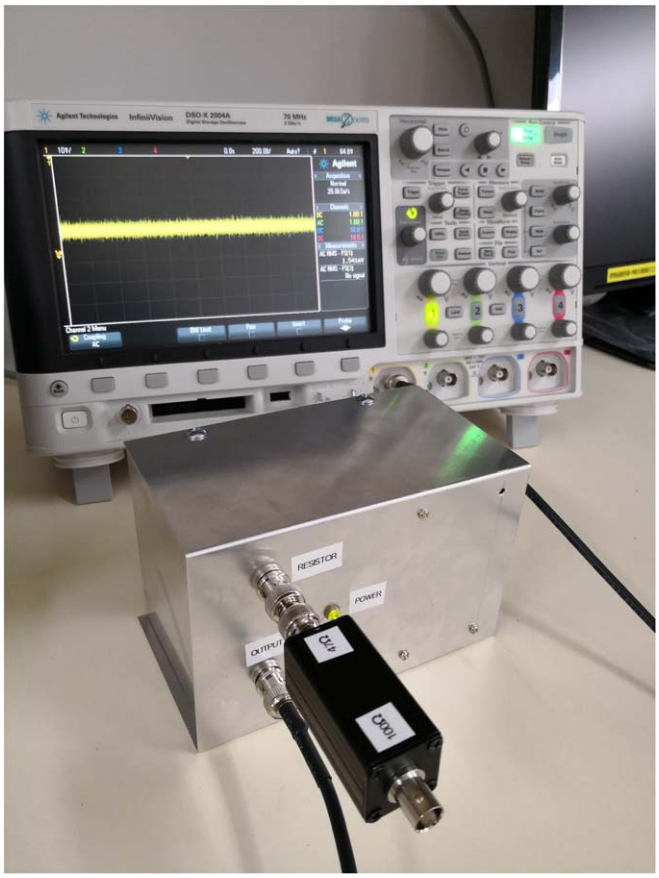


Figure 3. Measurement of the resistor noise using an oscilloscope.

Figure 4. A set of resistor boxes with various resistors inside. It is important to cover the range of the resistance from 50 Ω to 1 MΩ.

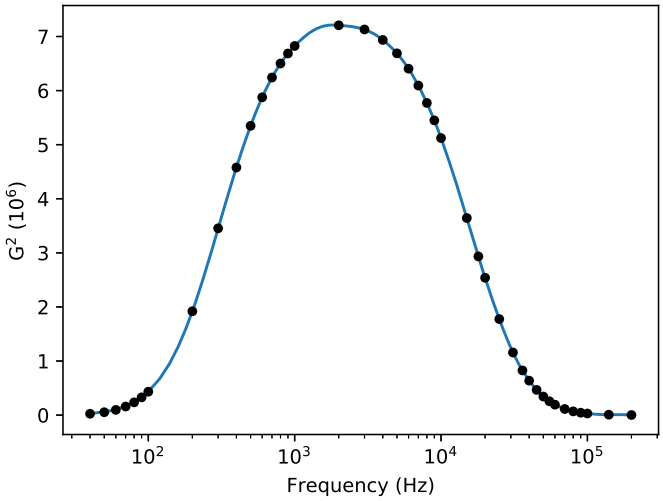


Figure 5. The gain profile measured using a function generator and an oscilloscope. The line is a cubic interpolation of the data points.

4. Data taking and analysis

4.1. Measurement of the gain profile

We measure the gain profile of the amplifier by connecting a function generator (Rigol DG1022) to the input of the amplifier circuit. We measure both the input and the output voltages using an oscilloscope (Agilent DSO-X 2004A) while changing the frequency of the signal from 40 Hz to 200 kHz. The gain is calculated from $G = V_{\text{out}}/V_{\text{in}}$ and the result is presented in figure 5 where we plot G^2 versus the frequency of the input signal, f. We obtain the fit to the data points by employing the cubic interpolation function of the SciPy (Python) package. We found that the peak occurs at around 2 kHz with a gain of approximately 2,700. The uncertainty of the voltage measurement using an oscilloscope is about 0.1%.

4.2. Measurement of k_B and the input capacitance

Once we have obtained $G(f)$, we measure the root-mean-square voltage noise (V_{RMS}) for all the resistors. V_{RMS} can be measured directly using an on-screen measurement function of the oscilloscope. Alternatively, one can save traces from the oscilloscope and calculate V_{RMS} using a personal computer. We found that the uncertainty from the on-screen measurement (which is approximately 0.1%, judging from the fluctuation of the number displayed) is good enough for our experiment.

Each value for V_{RMS} is background subtracted using

$$V_{\text{RMS}}^2 = V_{\text{RMS measured}}^2 - V_{\text{background}}^2, \tag{3}$$

where $V_{\text{background}} = 1.490 \pm 0.002$ mV is measured with the input of the amplifier shorted.

We measure two sets of data. The first set is measured without any RG-58/U cable between the resistor box and the amplifier circuit, as shown with the black circles (•) data set in figure 6. The second set is measured with a 1-meter RG-58/U cable inserted between the resistor boxes and the amplifier, as shown with the black crosses (×) data set in figure 6.

Each set of data is fitted using equations (1) and (2). Since for each value of R, the frequency bandwidth, Δf, has to be reevaluated, a simple ready-to-use fitting routine is usually insufficient. We use a package called `lmfit` in Python to fit the data. The flexibility of the package allows us to manually define the fit function and accommodate our needs. We provide a Python script used in this manuscript at [12]. A detailed fitting procedure can be found in the appendix.

For the first set of data (no RG-58/U cable), we obtain $k_B = (1.374 \pm 0.010) \times 10^{-23}$ J K^{-1} and $C = 41.1 \pm 0.6$ pF. For the second set of data (1-m long RG-58/U cable), we obtain $k_B = (1.387 \pm 0.015) \times 10^{-23}$ J K^{-1} and $C_1 = 121 \pm 3$ pF. The temperature for both data sets is measured to be $T = 23.8 \pm 0.2$ °C.

The increase in the input capacitance in the second data set when compared to the first is due to the presence of the RG-58/U cable. We conclude that a 1-meter RG-58/U cable has a capacitance of 80 ± 3 pF. This is in agreement with the specification of 82 pF m^{-1} (see [13]).

The final value for the Boltzmann's constant is obtained from a weighted average of the values from the two data sets. We obtain $k_B = (1.381 \pm 0.009) \times 10^{-23}$ J K^{-1} which agrees with the CODATA value of $1.380\,648\,52(79) \times 10^{-23}$ J K^{-1} (see [14]).

5. Observations

We included this experiment as a part of the 3rd-year experimental physics course for the Physics Department, Faculty of Science, Mahidol University from 2016 to 2017. A pair of students had three hours to complete the experiment. Because of the time limit, we asked the students to disregard data points for $R > 10\,\text{k}\Omega$ and fit the data using a simple linear relation (see appendix). In this case, the measurement of the input capacitance is not included in our teaching of this experiment.

We found a few common pitfalls that the students experienced:

- Without being told explicitly, the students did not realize that the graph in figure 6 should be plotted in a log-log scale. In fact, only 1 out of 50 students chose appropriate axes to display the data.
- The students spent too much time on the measurement of the gain profile due to lack of planning. It is crucial for the students to understand a multiple-octave-spanning range of the signal frequency.
- During the measurement of the gain, the input signal to the amplifier is usually set to be too high. This results in the output being a square wave (due to clipping). Some students failed to recognize the cause of this problem. For some function generators, the signal cannot be adjusted to be small enough. A simple voltage divider can remedy this problem.
- Some students were not fluent with the oscilloscope and were not able to use the oscilloscope to measure the voltage properly.
- Most of the students failed to realize that the amplifier is extremely sensitive. Any talking during the measurement affects the fluctuation of the signal tremendously, especially at small values of R where the noise amplitude under measurement is low.

6. Summary and outlook

We constructed a high-durability Johnson noise experiment that allows a measurement of the Boltzmann's constant with an accuracy of better than 1%. Because of the tunability of the input capacitance, we can directly observe the effect of the input capacitance on the bandwidth of the amplifier circuit. This allows us to measure the capacitance of an RG-58/U cable which agrees with the specification.

Because of the simplicity of the construction, this experiment is highly suitable as a part of an experimental physics course for undergraduate students. The difficulty of the experiment can be adjusted according to the time allocated for the experiment. Many important skills such as low-noise measurement and non-trivial data analysis are required to obtain an accurate measurement of the Boltzmann's constant.

Acknowledgments

This project is supported by the Thailand Research Fund (MRG6080221) and the Faculty of Science, Mahidol University. The author would like to thank Piyapong Rahut and Nopphrakorn Kerdsamut for assistance during the design of the construction.

Appendix A. Fitting procedure to determine the Boltzmann's constant

Data shown in figure 5 allows us to determine the gain profile function $G(f)$. This can be done by fitting to a combined high-pass/low-pass filter response function according to the circuit shown in figure 1. However, we found that using a cubic interpolation function (SciPy) is simpler and works better. The script `f2 = interp1d(x,y, kind='cubic')` allows us to construct a callable function `f2` that can take any input frequency as an argument.

Next is to fit the V_{RMS}^2 versus R data to determine k_{B} with the input capacitance of the amplifier taken into account. We have the full equation

$$V_{\mathrm{RMS}}^2 = 4k_{\mathrm{B}}TR\int_0^\infty \frac{[G(f)]^2}{1+(2\pi fRC)^2}\mathrm{d}f. \tag{4}$$

It is convenient to define an effective resistance R_{eff} as

$$R_{\mathrm{eff}} = R\int_0^\infty \frac{[G(f)]^2}{1+(2\pi fRC)^2}\mathrm{d}f. \tag{5}$$

So equation (4) becomes

$$V_{\mathrm{RMS}}^2 = 4k_{\mathrm{B}}TR_{\mathrm{eff}}, \tag{6}$$

as k_{B} can be found by fitting a linear relationship between V_{RMS}^2 and R_{eff}.

The `lmfit` package allows us to define an arbitrary fit function. We define a function that takes V_{RMS}^2 and R as input data, and the capacitance C and slope $A = 4k_{\mathrm{B}}T$ as fitting parameters. Within this function, R_{eff} is calculated numerically using

$$R_{\mathrm{eff}} \approx R\sum_{i=1}^{N}\left(\frac{[G(f_i)]^2}{1+(2\pi f_i RC)^2}\Delta f\right), \tag{7}$$

where $G(f)$ is the function `f2` found earlier. For our analysis, we chose $\Delta f = 20\,\mathrm{Hz}$. Since the fitting routine will have to search for C and A that minimize the error between the model and the data set, the sum in equation (7) is evaluated every time the fitting function is called and the value of C has changed.

Appendix B. Tables for numerical data

Table B1 shows the measured output voltage as a function of signal frequency. The uncertainty is determined by the fluctuation of the reading of the oscilloscope.

Table B1. Measured gain of the amplifier for different signal frequencies. V_{in} is measured independently to be 1.690 ± 0.005 mV.

f (Hz)	V_{out} (V)	ΔV_{out} (V)	$G = V_{out}/V_{in}$	f (Hz)	V_{out} (V)	ΔV_{out} (V)	$G = V_{out}/V_{in}$
40	0.2659	0.0003	157.3	7k	4.172	0.001	2468
50	0.3902	0.0003	230.9	8k	4.060	0.001	2402
60	0.5251	0.0003	310.7	9k	3.945	0.001	2334
70	0.6731	0.0003	398.3	10k	3.825	0.001	2263
80	0.8210	0.0003	485.8	15k	3.226	0.001	1909
90	0.9685	0.0005	573.1	18k	2.895	0.001	1713
100	1.112	0.001	658.1	20k	2.693	0.001	1593
200	2.341	0.001	1386	25k	2.252	0.001	1333
300	3.141	0.001	1859	31k	1.818	0.001	1076
400	3.616	0.001	2140	36k	1.536	0.001	908.9
500	3.909	0.001	2313	40k	1.349	0.001	798.2
600	4.097	0.001	2424	45k	1.153	0.001	682.5
700	4.222	0.001	2498	50k	0.990	0.001	585.8
800	4.309	0.001	2550	55k	0.855 0	0.000 5	505.9
900	4.370	0.001	2586	60k	0.742 0	0.000 5	439.1
1k	4.415	0.001	2612	70k	0.568 0	0.000 5	336.1
2k	4.537	0.001	2685	80k	0.443 8	0.000 5	262.6
3k	4.513	0.001	2670	90k	0.355 0	0.000 5	210.1
4k	4.451	0.001	2634	100k	0.289 5	0.000 5	171.3
5k	4.371	0.001	2586	140k	0.148 6	0.000 5	87.93
6k	4.277	0.001	2531	200k	0.074 0	0.001	43.8

Table B2. Measured V_{RMS} for different resistor values.

R (Ω)	$V_{RMS\ meassured}$ (mV) (no RG-58/U)	$V_{RMS\ meassured}$ (mV) (with 1m-RG-58/U)	ΔV_{RMS} (mV)
0	1.490	1.490	0.005
47	1.530	1.528	0.005
100	1.570	1.560	0.005
220	1.650	1.659	0.005
470	1.824	1.836	0.005
1k	2.14	2.13	0.01
2.2k	2.72	2.71	0.01
4.7k	3.63	3.61	0.02
10k	4.98	4.95	0.02
22k	7.06	6.83	0.03
47k	10.02	9.07	0.03
100k	13.70	11.30	0.05
220k	17.8	13.0	0.1
470k	21.1	13.8	0.1
1M	22.9	14.1	0.1

Table B2 shows the root-mean-square voltage noise of the amplifier as a function of input resistance. Similarly, the uncertainty is determined by the fluctuation of the reading of the oscilloscope.

Appendix C. Simplification of the data analysis

It is possible to disregard the effect of the input capacitance of the amplifier if the students measure the noise up to $R = 10\,\mathrm{k}\Omega$. We justify this procedure by fitting a simple linear function to the data set with resistance larger than $10\,\mathrm{k}\Omega$. Because of the roll-off effect as seen in figure 6, data points for $R > 10\,\mathrm{k}\Omega$ decrease the slope and hence decrease the measured value of k_B. We show in table C1 the effect of data points for $R > 10\,\mathrm{k}\Omega$. We conclude that the students can safely disregard $R > 10\,\mathrm{k}\Omega$ and use a simple linear relationship between V_RMS^2 and R to find a reasonable value of k_B.

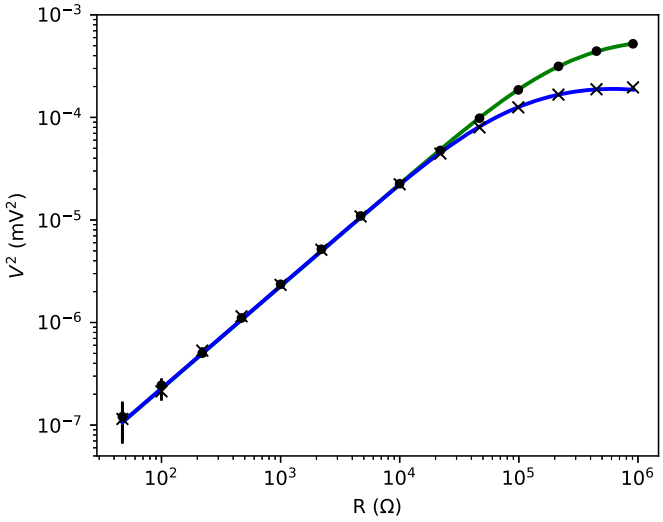


Figure 6. A plot of V_RMS^2 versus the resistance, R. Black circles (•) denote measurements without an extra RG-58/U cable. Black crosses (×) denote measurements with a 1-meter-long RG-58/U cable between the resistor box and the input of the amplifier. Each data set is shown with a curve fitted as discussed in the text.

Table C1. Measured k_B for different numbers of data points for $R > 10\,\mathrm{k}\Omega$.

k_B from a linear fit ($\times 10^{-23}$ J K^{-1})	Number of data points for $R > 10\,\mathrm{k}\Omega$ included
0.6087	6
0.9604	5
1.191	4
1.269	3
1.330	2
1.370	1
1.405	0

ORCID iDs

Thaned Pruttivarasin https://orcid.org/0000-0001-9957-2719

References

[1] Earl J A 1966 Undergraduate experiment on thermal and shot noise *Am. J. Phys.* **34** 575
[2] Engelberg S and Bendelac Y 2003 A topic for an undergraduate laboratory experiment in signal processing *IEEE Instrum. Meas. Mag.* **2003** 49–52
[3] Kraftmakher Y 1995 Two student experiments on electrical fluctuations *Am. J. Phys.* **63** 932
[4] Livesey D L and McLeod D L 1973 An experiment on electronic noise in the freshman laboratory *Am. J. Phys.* **41** 1364
[5] Mishonov T M, Yordanov V G and Varonov A M Measurement of electron charge q_e and Boltzmann's constant k_B by a cheap do-it-yourself undergraduate experiment arXiv:1703.05224
[6] Johnson J B 1971 Electronic noise: the first two decades *IEEE Spectr.* **8** 42–6
[7] Geller J 2007 Build the jcan to measure resistor noise *Nuts and Volts* **2007** 54–62
[8] Horowitz P and Hill W 1989 *The Art of Electronics* 2nd edn (New York: Cambridge University Press) p 1031
[9] Kittel C 1958 *Elementary Statistical Physics* (New York: Wiley)
[10] MIT Department of Physics, Johnson noise and shot noise: the determination of the Boltzmann constant, absolute zero temperature and the charge of the electron.
[11] Abbott D, Davis B R, Phillips N J and Eshraghian K 1996 Simple derivation of the thermal noise formula using window-limited Fourier transforms and other conundrums *IEEE Trans. Educ.* **39** 1–13
[12] The Python code used for data analysis can be found at https://github.com/thetorque/ion_clock_mahidol/blob/master/DataAnalysis/Johnson_noise/2018_data/johnson_noise_capacitance_2018.py.
[13] We use C3519 RG 58/U cable. The specification can be obtained at https://alliedelec.com/m/d/b6728848b37672a49202d050a0ff49b5.pdf.
[14] Mohr P J, Newell D B and Taylor B N 2016 *The 2014 CODATA Recommended Values of the Fundamental Physical Constants' (Web Version 7.2).* ed J Baker, M Douma and S Kotochigova (Gaithersburg, MD: National Institute of Standards and Technology) 20899 http://physics.nist.gov/constants [Tuesday, 22-May-2018 06:46:55 EDT]

บรรณานุกรม

[1] Pruttivarasin, T. and Katori, H., Compact FPGA-based pulse-sequencer and radio-frequency generator for experiments with trapped atoms, Rev. Sci. Instrum. 86, 115106 (2015).

[2] Pruttivarasin, T., A robust experimental setup for Johnson noise measurement suitable for ad- vanced undergraduate students, Eur. J. Phys. 39, 065102 (2018).

[3] J. B. Johnson,"Electronic noise: The first two decades," IEEE Spectr., **8** 42–46, (1971).

[4] J. Geller, "Build the Jcan to Measure Resistor Noise," Nuts and Volts, July 2007, 52-62 (2007).

Output จากโครงการวิจัยที่ได้รับทุนจาก สกว.

1. ผลงานตีพิมพ์ในวารสารวิชาการนานาชาติ (ระบุชื่อผู้แต่ง ชื่อเรื่อง ชื่อวารสาร ปี เล่มที่ เลขที่ และหน้า) หรือผลงานตามที่คาดไว้ในสัญญาโครงการ

ตีพิมพ์ใน Pruttivarasin, T., A robust experimental setup for Johnson noise measurement suitable for advanced undergraduate students, Eur. J. Phys. 39, 065102 (2018).

2. การนำผลงานวิจัยไปใช้ประโยชน์
   - เชิงพาณิชย์ (มีการนำไปผลิต/ขาย/ก่อให้เกิดรายได้ หรือมีการนำไปประยุกต์ใช้โดยภาคธุรกิจ/บุคคลทั่วไป)

เครื่องกำเนิดความถี่วิทยุที่ใช้งานที่สถาบันมาตรวิทยาเพื่อรองรับโครงการวิจัยด้านนาฬิกาอะตอมเชิงแสง

   - เชิงนโยบาย (มีการกำหนดนโยบายอิงงานวิจัย/เกิดมาตรการใหม่/เปลี่ยนแปลงระเบียบข้อบังคับหรือวิธีทำงาน)
   - เชิงสาธารณะ (มีเครือข่ายความร่วมมือ/สร้างกระแสความสนใจในวงกว้าง)
   - เชิงวิชาการ (มีการพัฒนาการเรียนการสอน/สร้างนักวิจัยใหม่)
3. อื่นๆ (เช่น ผลงานตีพิมพ์ในวารสารวิชาการในประเทศ การเสนอผลงานในที่ประชุมวิชาการ หนังสือ การจดสิทธิบัตร)