การออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอล โดยใช้วิธีประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ

DESIGN OF A ROM-LESS DIRECT DIGITAL FREQUENCY SYNTHESIZER USING A POLYNOMIAL APPROXIMATION

ชรัณ มีนกาญจน์
CHARAN MEENAKARN

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมอิเล็กทรอนิกส์
บัณฑิตวิทยาลัย
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ. 2546

ISBN 974-324-667-3

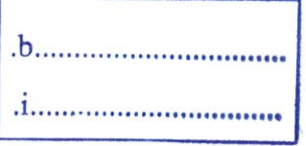
สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

การออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอล
โดยใช้วิธีประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ

DESIGN OF A ROM-LESS DIRECT DIGITAL FREQUENCY SYNTHESIZER USING A POLYNOMIAL APPROXIMATION

ชรัณ มีนกาญจน์
CHARAN MEENAKARN

เลขหมู่.................................
เลขทะเบียน 47584
วัน, เดือน, ปี 21 ส.ค. 2546



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมอิเล็กทรอนิกส์
บัณฑิตวิทยาลัย
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ.2546

ISBN 974-324-667-3

DESIGN OF A ROM-LESS DIRECT DIGITAL FREQUENCY SYNTHESIZER USING A POLYNOMIAL APPROXIMATION

CHARAN MEENAKARN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
MASTER OF ENGINEERING IN ELECTRONIC ENGINEERING
SCHOOL OF GRADUATE STUDIES
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
2003

ISBN 974-324-667-3

COPYRIGHT 2003
SCHOOL OF GRADUATE STUDIES
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

บัณฑิตวิทยาลัย
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ใบรับรองวิทยานิพนธ์

หัวข้อวิทยานิพนธ์ การออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีประมาณค่าโพลีเมียลแทนการใช้หน่วยความจำ
DESIGN OF A ROM-LESS DIRECT DIGITAL FREQUENCY SYNTHESIZER USING A POLYNOMIAL APPROXIMATION

ชื่อนักศึกษา นายชรัณ มีนกาญจน์

รหัสประจำตัว 43061316

ปริญญา วิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชา วิศวกรรมอิเล็กทรอนิกส์

อาจารย์ผู้ควบคุมวิทยานิพนธ์ ผศ.ดร.อภินันท์ ธนชยานนท์

คณะกรรมการสอบวิทยานิพนธ์		ลายมือชื่อ
ผศ.พิชัย	คูศิริวานิชกร	
ผศ.ดร.วรากร	เกษมสุวรรณ์	
ผศ.ประภากร	สุวรรณะ	
รศ.ดร.สมศักดิ์	ชุมช่วย	
ผศ.ดร.อภินันท์	ธนชยานนท์	

วัน/เดือน/ปี ที่สอบ 21 พฤษภาคม 2546 เวลา 10.30-12.30 น.

สถานที่สอบ ณ อาคาร 12 ชั้น ชั้น 4 (ห้อง E12-301)

บัณฑิตวิทยาลัยรับรองแล้ว

(รศ.ดร.บุญวัฒน์ อัตชู)
คณบดีบัณฑิตวิทยาลัย

วันที่......18.......**เดือน**.....กรกฎาคม.......**พ.ศ**.....2546....

หัวข้อวิทยานิพนธ์	การออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ
นักศึกษา	นายชรัณ มีนกาญจน์
รหัสนักศึกษา	43061316
ปริญญา	วิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชา	วิศวกรรมอิเล็กทรอนิกส์
พ.ศ.	2546
อาจารย์ผู้ควบคุมวิทยานิพนธ์	ผศ. ดร. อภินันท์ ธนชยานนท์

บทคัดย่อ

ในปัจจุบันมีการนำวงจรสังเคราะห์ความถี่แบบดิจิตอลมาใช้งานในระบบสื่อสารกันอย่างแพร่หลาย เนื่องจากวงจรสังเคราะห์ความถี่แบบดิจิตอลมีข้อได้เปรียบวงจรสังเคราะห์ความถี่แบบเฟสล็อกลูป หลายประการคือ มีความละเอียดของความถี่และเฟสสูง, สามารถเปลี่ยนความถี่ได้รวดเร็ว, สัญญาณรบกวนทางเฟสต่ำ และง่ายต่อการมอดูเลทสัญญาณ โดยทั่วไปวงจรสังเคราะห์ความถี่แบบดิจิตอลนิยมเก็บรูปสัญญาณที่ต้องการสังเคราะห์ไว้ในหน่วยความจำ ซึ่งจำเป็นที่จะต้องใช้หน่วยความจำขนาดใหญ่สำหรับสังเคราะห์สัญญาณที่มีคุณภาพสูง เป็นผลให้ความเร็วในการทำงานลดลง และการใช้กำลังงานสูงขึ้น วิทยานิพนธ์ฉบับนี้จึงนำเสนอการออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอลสำหรับสังเคราะห์สัญญาณซายน์ โดยใช้การประมาณค่าโพลีโนเมียล ซึ่งสามารถประมาณค่าของสัญญาณซายน์ได้ใกล้เคียงกับสัญญาณจริง มีค่าความผิดพลาดน้อยกว่า 2e-4 ทำให้ไม่จำเป็นต้องมีหน่วยความจำสำหรับเก็บรูปสัญญาณ หรือปรับปรุงคุณภาพของสัญญาณ ผลการทำงานและประสิทธิภาพของวงจรที่ออกแบบสามารถยืนยันได้โดยการจำลองการทำงาน และการทดสอบการทำงานในระดับฮาร์ดแวร์

Thesis Title	Design of a ROM-Less Direct Digital Frequency Synthesizer Using a Polynomial Approximation
Student	Mr. Charan Meenakarn
Student ID.	43061316
Degree	Master of Engineering
Programme	Electronic Engineering
Year	2003
Thesis Advisor	Assist. Prof. Dr. Apinunt Thanachayanont

ABSTRACT

Direct digital frequency synthesizers (DDFS) are playing an increasingly significant role in modern digital communication systems because of their characteristics of excellent frequency and phase resolution, fast frequency switching, low phase noise, and modulating capability. Typically, most DDFSs store a digital representation of a sinewave in a read-only memory (ROM). The spectral purity of a sinewave is determined by the resolution of the ROM. But larger ROM size means lower speed and higher power consumption. This thesis, thus, presents a design of a sine-output ROM-less direct digital frequency synthesizer by using a polynomial approximation. A maximum error of a sinewave approximated by a polynomial approximation is less than 2e-4. This means that no ROM is required. The functionality and the efficiency of the designed DDFS are guaranteed by simulation and hardware experimental results.

กิตติกรรมประกาศ

ผู้วิจัยขอกราบขอบพระคุณอาจารย์ผู้ควบคุมวิทยานิพนธ์ ผศ.ดร. อภินันท์ ธนชยานนท์ เป็นอย่างสูง ที่ให้คำปรึกษาและคำแนะนำให้การทำวิทยานิพนธ์สำเร็จลุล่วงไปด้วยดี

ขอขอบคุณห้องปฏิบัติการ MDRD RECCIT สำหรับสถานที่ เครื่องมือ และอุปกรณ์ในการทำวิทยานิพนธ์ฉบับนี้

ขอขอบคุณคุณธนพร ทองเผือก และคุณวิชัย แสงนาค สำหรับข้อมูลในการออกแบบชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำในบทที่ 3

ขอขอบคุณศูนย์พัฒนาธุรกิจออกแบบวงจรรวม ศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติสำหรับความอนุเคราะห์บอร์ดและชิปเอฟพีจีเอสำหรับการทดสอบวงจร

ขอขอบคุณศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติสำหรับการสนับสนุนทุนวิจัยในการออกแบบและเจือสารชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตคลที่ใช้หน่วยความจำ

ขอขอบคุณบิดา มารดา และพี่น้องและเพื่อนทุกๆ ท่าน สำหรับกำลังใจและความสนับสนุนที่มีให้มาโดยตลอด

คุณค่าและประโยชน์อันพึงได้จากวิทยานิพนธ์ฉบับนี้ ผู้วิจัยขอมอบแด่ผู้มีพระคุณทุกท่าน

ชรัณ มีนกาญจน์

สารบัญ

หน้า

บทคัดย่อภาษาไทย I

บทคัดย่อภาษาอังกฤษ II

กิตติกรรมประกาศ III

สารบัญ IV

สารบัญตาราง VII

สารบัญภาพ VIII

บทที่ 1 บทนำ 1

1.1 ความเป็นมาและความสำคัญของปัญหา 1

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา 3

1.3 สมมติฐานของการศึกษา 3

1.4 ทฤษฎีหรือแนวความคิดที่ใช้ในการวิจัย 4

1.5 ขอบเขตการวิจัย 4

1.6 ขั้นตอนการศึกษา 4

บทที่ 2 ทฤษฎีการสังเคราะห์ความถี่แบบดิจิตอล และงานวิจัยที่ผ่านมา 6

2.1 สถาปัตยกรรมของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล 7

2.1.1 วงจรสร้างเฟส 8

2.1.2 วงจรเปลี่ยนเฟสเป็นแอมปลิจูด 12

2.1.3 วงจรแปลงดิจิตอลเป็นอนาลอก 13

2.1.4 วงจรกรองความถี่ต่ำผ่าน 14

2.2 การสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ 16

2.2.1 การบีบอัดควอแดรนท์ 17

2.2.2 ขั้นตอนวิธีผลต่างซายน์-เฟส 19

2.2.3 ขั้นตอนวิธีซันเดอร์แลนด์ 20

2.2.4 การประมาณค่าอนุกรมเทย์เลอร์ 21

สารบัญ (ต่อ)

หน้า

2.3 การสังเคราะห์ความถี่แบบดิจิตอลที่ไม่ใช้หน่วยความจำ 23

2.3.1 ขั้นตอนวิธีคอร์ดิก 23

2.3.2 การคูณเชิงซ้อน 24

2.3.3 การประมาณค่าพาราโบลา 26

2.4 สรุปเปรียบเทียบ 28

บทที่ 3 วงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ 30

3.1 ข้อกำหนดการออกแบบสำหรับการประยุกต์ใช้งาน 30

3.2 รายละเอียดการออกแบบวงจร 31

3.2.1 ส่วนดิจิตอล 31

3.2.2 ส่วนอนาลอก 36

3.3 การออกแบบเลย์เอาต์ 39

3.4 ผลการทดสอบการทำงาน 40

3.5 สรุป 44

บทที่ 4 วงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ 45

4.1 สถาปัตยกรรมของวงจรที่นำเสนอ 46

4.1.1 การประมาณค่าโพลีโนเมียล 46

4.1.2 การออกแบบเป็นวงจรดิจิตอล 47

4.2 ผลการจำลองการทำงานของระบบ 49

4.3 วงจรสร้างเฟส 50

4.4 วงจรเปลี่ยนเฟสเป็นแอมปลิจูด 51

4.5 ผลการจำลองการทำงานและการทดสอบการทำงานของวงจร 53

4.6 สรุป 58

สารบัญ (ต่อ)

	หน้า
บทที่ 5 สรุปผลการวิจัยและข้อเสนอแนะ	59
เอกสารอ้างอิง	61
ภาคผนวก	64
ภาคผนวก ก. โค้ดวีเอชดีแอลของวงจรรวมสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ	65
ภาคผนวก ข. รายละเอียดข้อมูลของชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ	115
ภาคผนวก ค. คู่มือการใช้งานบอร์ดต้นแบบชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ	122
ภาคผนวก ง. โค้ดวีเอชดีแอลของวงจรรวมสังเคราะห์ความถี่แบบดิจิตอลที่ใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ.......	132
ผลงานวิจัยที่เกี่ยวข้องกับวิทยานิพนธ์และได้รับการตีพิมพ์	142
ประวัติผู้เขียน	169

สารบัญตาราง

ตารางที่ หน้า

2.1 ค่าเอาต์พุตของวงจรสร้างเฟสในแต่ละรอบสัญญาณนาฬิกา เมื่อ *N=4* และ *W=3* 10

2.2 ความสัมพันธ์ของบิตเอ็มเอสบีกับควอแดรนท์และสัญญาณซายน์ 18

2.3 สรุปคุณสมบัติของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล
แบบที่ใช้หน่วยความจำและไม่ใช้หน่วยความจำ 29

2.4 คุณสมบัติและประสิทธิภาพของวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ได้มีการวิจัยมา.. 29

3.1 คุณสมบัติของวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ 44

4.1 ค่าคงที่ที่ใช้ในสมการที่ (4.4) 47

4.2 ค่าของ k_2 และ k_4 ที่คำนวณจากการเลื่อน-บวก 47

4.3 คุณสมบัติของวงจรสังเคราะห์ความถี่แบบดิจิตอล
โดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ 58

4.4 เปรียบเทียบคุณสมบัติของวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ไม่ใช้หน่วยความจำ... 58

5.1 คุณสมบัติของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล
ที่ใช้การประมาณค่าโพลีโนเมียลที่นำเสนอเปรียบเทียบกับแบบที่ใช้หน่วยความจำ 60

สารบัญภาพ

ภาพที่	หน้า
2.1 สถาปัตยกรรมของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล	7
2.2 วงจรสร้างเฟส	8
2.3 ค่าเฟสเมื่อพิจารณาเป็นการหมุนรอบวงกลม	8
2.4 สัญญาณเอาต์พุตของวงจรสร้างเฟสในแต่ละรอบสัญญาณนาฬิกา เมื่อ $N=4$ และ $W=3$	11
2.5 วงจรแปลงดิจิตอลเป็นอนาลอกที่มีสถาปัตยกรรมแบบแยกส่วน	14
2.6 สเปกตรัมของสัญญาณเอาต์พุตของวงจรสังเคราะห์ความถี่แบบดิจิตอล	14
2.7 ผลตอบสนองทางความถี่ของวงจรกรองความถี่แอบแฝงในอุดมคติ	15
2.8 ผลตอบสนองทางความถี่ของวงจรกรองความถี่แอบแฝงทั่วไป	15
2.9 การชดเชยแอมปลิจูดของสัญญาณด้วยวงจรกรองอินเวอร์สซิงค์	16
2.10 หน่วยความจำสำหรับเปลี่ยนเฟสเป็นแอมปลิจูดสัญญาณซายน์	16
2.11 การสังเคราะห์สัญญาณซายน์เต็มรูปด้วยเทคนิคบีบอัดควอแดรนท์	18
2.12 ขั้นตอนวิธีผลต่างซายน์-เฟส	19
2.13 ขั้นตอนวิธีซันเดอร์แลนด์	21
2.14 การประมาณค่าอนุกรมเทย์เลอร์	22
2.15 การสร้างสัญญาณซายน์ด้วยขั้นตอนวิธีคอร์ดิก	23
2.16 การหมุนเวกเตอร์ของการคูณเชิงซ้อน	25
2.17 การสังเคราะห์สัญญาณซายน์ด้วยการคูณเชิงซ้อน	26
2.18 ค่าความผิดพลาดของการประมาณสัญญาณซายน์ ด้วยสมการพาราโบลาอันดับที่ 1	27
2.19 ค่าความผิดพลาดของการประมาณสัญญาณซายน์ ด้วยสมการพาราโบลาอันดับที่ 2	27
2.20 โครงสร้างของวงจรสังเคราะห์สัญญาณซายน์ด้วยสมการพาราโบลาอันดับที่ 2	28
3.1 สถาปัตยกรรมของชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ..	31
3.2 โครงสร้างของวงจรสร้างเฟสขนาด 32 บิต	32
3.3 โครงสร้างของวงจรบวก 4 บิต แบบริปเปิลแคร์รี่	33

สารบัญภาพ (ต่อ)

ภาพที่ หน้า

3.4 ขั้นตอนวิธีโมดิฟายซันเดอร์แลนด์ 34

3.5 การสังเคราะห์สัญญาณรูปฟันเลื่อย 34

3.6 รีจิสเตอร์เลื่อนย้อนกลับแบบเชิงเส้น 35

3.7 วงจรถอดรหัสเทอร์โมมิเตอร์ 35

3.8 สถาปัตยกรรมของวงจรแปลงดิจิตอลเป็นอนาลอก 10 บิต

ในโหมดกระแสแบบแยกส่วน 6/4 36

3.9 โครงสร้างของวงจรแปลงดิจิตอลเป็นอนาลอก 36

3.10 วงจรขับกระแส 37

3.11 การทำงานของวงจรกำเนิดกระแส 38

3.12 เลย์เอาต์ของวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ 40

3.13 การจำลองสัญญาณแอมปลิจูดดิจิตอลเอาต์พุตที่ความถี่ 4.032 เมกะเฮิร์ตซ์ 40

3.14 การจำลองสัญญาณเอาต์พุตของวงจรแปลงดิจิตอลเป็นอนาลอก

เมื่อมีการเปลี่ยนแปลงอุณหภูมิและแรงดัน 41

3.15 รูปถ่ายของวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ 41

3.16 ผังการทดสอบชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ 42

3.17 ภาพถ่ายบอร์ดทดสอบชิปวงจรรวมเพื่อสังเคราะห์ความถี่ 42

3.18 สเปกตรัมองค์ประกอบของสัญญาณซายน์ 5 เมกะเฮิรตซ์ 43

3.19 สัญญาณรบกวนทางเฟสของสัญญาณซายน์ 25 เมกะเฮิรตซ์ 43

3.20 ค่าเอสเอฟดีอาร์ (SFDR) ที่ความถี่เอาต์พุตต่างๆ

ขณะทำงานที่ความถี่สัญญาณนาฬิกา 100 เมกะเฮิรตซ์ 44

4.1 ค่าความผิดพลาดของการประมาณค่าซายน์ด้วยสมการต่างๆ 46

4.2 สถาปัตยกรรมของวงจรสังเคราะห์ความถี่แบบดิจิตอล

โดยใช้วิธีการประมาณค่าโพลีโนเมียล 48

4.3 โครงสร้างของวงจรสร้างโพลีโนเมียล 48

4.4 สเปกตรัมของสัญญาณซายน์ความถี่ 1/32 เท่าของสัญญาณนาฬิกา

จากการจำลองการทำงานของระบบ 49

สารบัญภาพ (ต่อ)

ภาพที่ หน้า

4.5 วงจรบวกแบบริปเปิลแครี่ขนาด 8 บิต 50

4.6 วงจรสร้างเฟสขนาด 32 บิต 51

4.7 วงจรคูณ 10x10 แบบบอว์-วูลีย์ 52

4.8 โครงสร้างการคำนวณ k_2m^2 52

4.9 โครงสร้างการคำนวณ k_4m^4 53

4.10 ผังการทดสอบวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล

โดยใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ 53

4.11 รูปสัญญาณซายน์ 1.5625 เมกะเฮิรตซ์ 54

4.12 รูปสัญญาณซายน์ 10 เมกะเฮิรตซ์ 54

4.13 สเปกตรัมของสัญญาณซายน์ 1.5625 เมกะเฮิรตซ์

ที่คำนวณจากโปรแกรมแมทแลบ 55

4.14 สเปกตรัมของสัญญาณซายน์ 1.5625 เมกะเฮิรตซ์ 56

4.15 สเปกตรัมของสัญญาณซายน์ 197 กิโลเฮิรตซ์ 56

4.16 สัญญาณรบกวนทางเฟสของสัญญาณซายน์ 15 เมกะเฮิรตซ์ 57

4.17 ค่าเอสเอฟดีอาร์ของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล

โดยใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำที่ความถี่ต่างๆ 57

บทที่ 1

บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

เครื่องมือสื่อสารประเภทต่างๆ มีส่วนช่วยอำนวยความสะดวกในการดำรงชีวิตประจำวันของมนุษย์เป็นอย่างมากไม่ว่าจะเป็น การสื่อสารดาวเทียม (Satellite communication), โทรศัพท์มือถือ (Mobile phone), วิทยุติดตามตัว (Radio pager) รวมไปถึงอุปกรณ์เครื่องใช้ไฟฟ้าตามบ้าน (Home appliance) ส่วนประกอบหลักที่สำคัญที่สุดในอุปกรณ์สื่อสารเหล่านี้ก็คือวงจรเพื่อสังเคราะห์ความถี่ (Frequency synthesizer) ที่ทำหน้าที่สังเคราะห์สัญญาณให้ได้ความถี่ตามต้องการ วงจรสังเคราะห์ความถี่ที่มีใช้กันอยู่ในปัจจุบันนั้นมีมากมายหลายประเภทด้วยกันสามารถแบ่งเป็นประเภทหลักๆ ได้ดังนี้

- วงจรเพื่อสังเคราะห์ความถี่โดยตรง (Direct frequency synthesizer) คือวงจรที่ทำหน้าที่สังเคราะห์ความถี่ได้โดยไม่ต้องมีสัญญาณย้อนกลับ (Feedback) แบ่งเป็นสองประเภทดังนี้
    - วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล (Direct digital frequency synthesizer, DDFS) เป็นการสังเคราะห์สัญญาณโดยใช้สัญญาณนาฬิกาอ้างอิง (Reference clock) สร้างรูปสัญญาณดิจิตอล (Digital pattern) ซึ่งจะถูกแปลงให้เป็นสัญญาณ อนาลอกโดยวงจรแปลงดิจิตอลเป็นอนาลอก (Digital-to-analog converter, DAC) และผ่านไปยังวงจรกรองความถี่ต่ำผ่าน (Low-pass filter) เพื่อกรองสัญญาณของความถี่แอบแฝง (Aliasing frequency) ออกและปรับปรุงคุณภาพสัญญาณให้ดีขึ้น
    - วงจรเพื่อสังเคราะห์ความถี่แบบอนาลอก (Direct analog frequency synthesizer, DAFS) เป็นวงจรที่สังเคราะห์ความถี่โดยใช้การผสม (Mix) และการคูณ (Multiplication) ของสัญญาณอนาลอก มีข้อดีคือสัญญาณที่สังเคราะห์มีคุณภาพสูง และสามารถเปลี่ยนความถี่ได้รวดเร็ว แต่มีข้อเสียคือวงจรจะมีขนาดใหญ่และกินไฟสูง จึงมักจะนำวงจรเพื่อสังเคราะห์ความถี่ชนิดนี้ไปใช้กับงานที่ต้องการสัญญาณคุณภาพสูงจริงๆ เท่านั้น

- วงจรเพื่อสังเคราะห์ความถี่โดยอ้อม (Indirect frequency synthesizer) คือวงจรที่สังเคราะห์ความถี่โดยอาศัยสัญญาณย้อนกลับ แบ่งเป็นสองประเภทคือวงจรเฟสล็อกลูปแบบอนาลอก (Analog phase-locked loop) และวงจรเฟสล็อกลูปแบบดิจิตอล (Digital phase-locked loop)

วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลมีข้อได้เปรียบวงจรเพื่อสังเคราะห์ความถี่แบบเฟสล็อกลูปและแบบอนาลอกหลายประการได้แก่ สามารถตั้งความถี่ได้แม่นยำ (Precise frequency setting), มีความละเอียดในการปรับความถี่และเฟสสูง (High frequency and phase resolutions), ใช้เวลาในการเปลี่ยนความถี่น้อย (Fast settling time), สัญญาณรบกวนทางเฟสต่ำ (Low phase noise), การเปลี่ยนเฟสเป็นไปอย่างต่อเนื่อง (Continuous-phase switching response) และทำการมอดูเลทสัญญาณได้ (Modulating capabilities) รวมไปถึงสามารถควบคุมการทำงานได้ง่ายเพราะเป็นวงจรดิจิตอลทั้งหมด จากข้อดีทั้งหมดของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลดังที่กล่าวมานี้ ทำให้วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลได้รับความนิยมนำไปประยุกต์ใช้ในงานต่างๆ อย่างแพร่หลาย

เราสามารถแบ่งงานวิจัยเกี่ยวกับการออกแบบวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลในปัจจุบันได้เป็นสองประเภทหลักๆ ดังนี้

(1) วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ (ROM-based DDFS)

(2) วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ไม่ใช้หน่วยความจำ (ROM-less DDFS)

ระหว่างวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลทั้งสองประเภทนี้ วงจรเพื่อสังเคราะห์ความถี่ที่ใช้หน่วยความจำจะได้รับความนิยมมากกว่า เนื่องจากสามารถออกแบบได้ง่าย แต่มีข้อจำกัดที่สำคัญคือ การเก็บรูปสัญญาณในหน่วยความจำนั้นจำเป็นต้องใช้หน่วยความจำขนาดใหญ่เพื่อที่จะสังเคราะห์สัญญาณให้มีคุณภาพสูง ซึ่งหน่วยความจำที่มีขนาดใหญ่จะทำให้วงจรกินกำลังงานมากขึ้น, ค่าใช้จ่ายในการเจือสารสูงขึ้น รวมไปถึงเวลาที่ใช้ในการเข้าถึงหน่วยความจำจะเพิ่มมากขึ้น เป็นผลทำให้ความเร็วในการทำงาน ของวงจรลดลง

เพื่อเป็นการหลีกเลี่ยงปัญหาที่จะเกิดจากการใช้หน่วยความจำในวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล วิทยานิพนธ์ฉบับนี้จึงนำเสนอการออกแบบวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลโดยใช้สมการโพลีโนเมียลประมาณค่าของรูปสัญญาณแทนการใช้หน่วยความจำ ซึ่งวงจรที่ออกแบบสามารถสังเคราะห์สัญญาณซายน์ได้ดีกว่าวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ใช้

หน่วยความจำ กล่าวคือมีค่าระดับฮาร์โมนิกสูงสุดไม่เกิด –64.87 เดซิเบล ซึ่งสามารถนำไปใช้ร่วมกับวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิต ได้โดยไม่ทำให้คุณภาพของสัญญาณเสื่อมลง

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

จากบทบาทที่สำคัญและข้อจำกัดในการใช้หน่วยความจำของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลตามที่ได้กล่าวมาแล้ว วิทยานิพนธ์ฉบับนี้จึงได้ถูกจัดทำขึ้น โดยมีความมุ่งหมายและวัตถุประสงค์แยกเป็นหัวข้อต่างๆ ได้ดังนี้

- ศึกษาทฤษฎีและหลักการทำงานของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล
- ออกแบบวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ
- ออกแบบวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ เพื่อพัฒนาวงจรให้มีประสิทธิภาพดีกว่าวงจรแบบที่ใช้หน่วยความจำ
- เปรียบเทียบประสิทธิภาพของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลชนิดที่ใช้หน่วยความจำและไม่ใช้หน่วยความจำ

1.3 สมมติฐานของการศึกษา

วิทยานิพนธ์ฉบับนี้ถูกจัดทำขึ้นโดยมีสมมติฐานคือสัญญาณรูปซายน์ (Sinewave) ที่สังเคราะห์ได้จากวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีประมาณค่าโพลีโนเมียลนั้น มีคุณภาพดีในระดับที่สามารถนำไปใช้งานร่วมกับวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิตได้ และความถี่สูงสุดของสัญญาณที่สังเคราะห์ได้สูงกว่าวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ (ใช้เทคโนโลยีในการเจือสารเหมือนกัน)

1.4 ทฤษฎีหรือแนวความคิดที่ใช้ในการวิจัย

วงจรสังเคราะห์ความถี่แบบดิจิตอลที่จะนำเสนอในวิทยานิพนธ์ฉบับนี้นั้นแตกต่างจากวงจรสังเคราะห์ความถี่แบบดิจิตอลทั่วๆ ไป คือวงจรที่ออกแบบจะประมาณค่าของสัญญาณรูปซายน์จากสมการโพลีโนเมียล โดยค่าของสัมประสิทธิ์ที่ใช้ในสมการนั้นได้ถูกคำนวณและลดรูปเพื่อให้ได้ค่าของสัญญาณซายน์ที่ใกล้เคียงกับสัญญาณจริงมากที่สุด โดยมีค่าความผิดพลาดน้อยกว่า 2e-4 นอกจากนี้การคำนวณค่าของสัมประสิทธิ์ในสมการโพลีโนเมียลนั้นยังคำนึงถึงความเหมาะสมในการนำไปออกแบบเป็นฮาร์ดแวร์จริงด้วยวิธีบนลงล่าง (Top-down design) อีกด้วย

1.5 ขอบเขตการวิจัย

วิทยานิพนธ์ฉบับนี้จะออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ โดยจะออกแบบวงจรสร้างเฟส (Phase accumulator) และวงจรเปลี่ยนเฟสเป็นแอมปลิจูด (Phase-to-amplitude converter) ด้วยวิธีการออกแบบจากบนลงล่าง (Top-down design) ซึ่งวงจรทั้งสองส่วนนี้จะถูกสังเคราะห์ (Synthesis) และจัดวาง-เชื่อมโยง (Place-and-route) ลงบนชิปเอฟพีจีเอ (Field Programmable Gate Array) แล้วนำไปทดสอบร่วมกับชิปแปลงดิจิตอลเป็นอนาลอก (DAC chip) และวงจรกรองความถี่ต่ำผ่าน (Low-pass filter) ภายนอก เพื่อตรวจสอบการทำงานและประสิทธิภาพของวงจร แล้วเปรียบเทียบผลการทำงานของวงจรที่ออกแบบกับวงจรสังเคราะห์ความถี่แบบที่ใช้หน่วยความจำ

1.6 ขั้นตอนการศึกษา

การวิจัยในวิทยานิพนธ์ฉบับนี้จะเริ่มจากการศึกษาทฤษฎีและหลักการทำงานของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลทั้งชนิดที่ใช้และไม่ใช้หน่วยความจำ และทำการเปรียบเทียบประสิทธิภาพของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ กับวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำที่ได้ออกแบบไป

โดยขั้นตอนในการออกแบบจะใช้วิธีการออกแบบจากบนลงล่าง โดยเริ่มจากการออกแบบและจำลองการทำงานในระดับระบบ (System-level simulation) และทำการออกแบบวงจรด้วยภาษาบรรยายฮาร์ดแวร์ (Hardware Description Language, HDL) ซึ่งในวิทยานิพนธ์นี้จะใช้ภาษาวีเอชดีแอล (VHDL) หลังจากนั้นจะทำการจำลองการทำงานระดับฟังก์ชั่น (Functional simulation) เพื่อตรวจสอบความถูกต้องในการทำงานของวงจร ก่อนที่จะนำไปสังเคราะห์

(Synthesis) และทำการจัดวางและเชื่อมโยง (Place-and-route) ให้ตรงกับเทคโนโลยีของชิปเอฟพีจีเอที่ทดสอบ แล้วนำผลที่ได้จากการจัดวางและเชื่อมโยงไปทำการจำลองการทำงานระดับเวลา (Timing simulation) เพื่อยืนยันความถูกต้องในการทำงานของวงจรอีกครั้งหนึ่ง เมื่อได้ตรวจสอบความถูกต้องในการจำลองการทำงานระดับเวลาของวงจรเรียบร้อยแล้ว ก็จะนำไปทดลองบนชิปเอฟพีจีเอหรือไอซีจริงอีกครั้งหนึ่ง

เนื้อหาในวิทยานิพนธ์จะแบ่งเป็นบทต่างๆ ตามรายละเอียดดังนี้

บทที่ 1 เป็นการเกริ่นนำถึงวัตถุประสงค์, สมมติฐาน, ขั้นตอนการศึกษาวิทยานิพนธ์ฉบับนี้

บทที่ 2 เกี่ยวกับทฤษฎีของการสังเคราะห์ความถี่แบบดิจิตอล และงานวิจัยที่ผ่านมา

บทที่ 3 นำเสนอการออกแบบวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

บทที่ 4 นำเสนอการออกแบบวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ

บทที่ 5 สรุปผลการวิจัย และเปรียบเทียบประสิทธิภาพของวงจรที่นำเสนอไว้ในบทที่ 3 และบทที่ 4

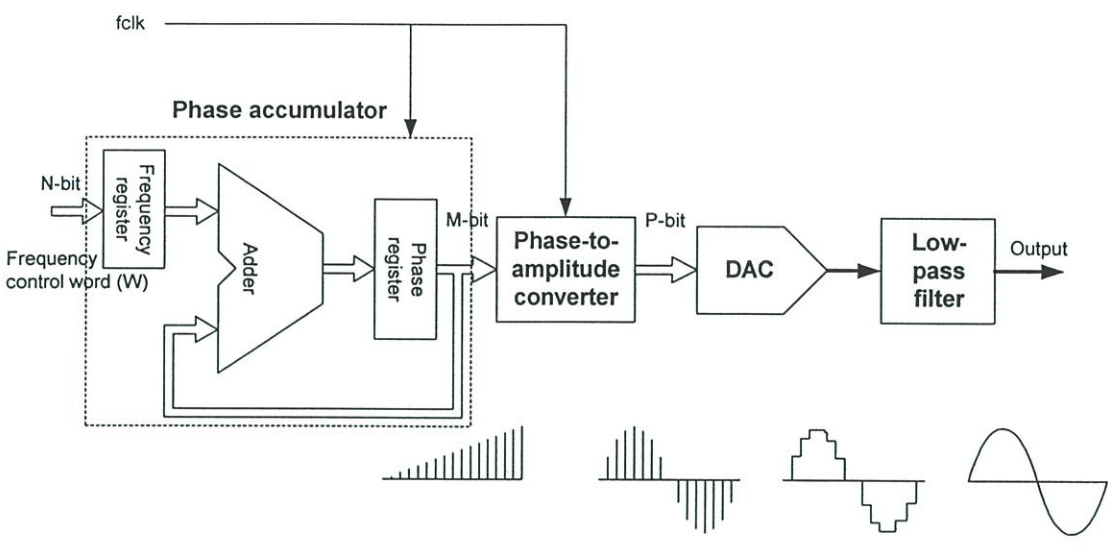
บทที่ 2

ทฤษฎีการสังเคราะห์ความถี่แบบดิจิตอล และงานวิจัยที่ผ่านมา

วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล (Direct digital frequency synthesizer, DDFS) ถูกคิดค้นและนำเสนอเป็นครั้งแรกในปี ค.ศ. 1971 โดย J. Tierney และคณะ [1] โดยมีหลักการคือใช้วงจรดิจิตอลสร้างสัญญาณเฟสเพื่อนำมาอ้างอิงเป็นค่าแอมปลิจูดของรูปสัญญาณ (Waveform) ที่ต้องการสังเคราะห์ ซึ่งในระยะแรกๆ นั้นนิยมใช้หน่วยความจำสำหรับเก็บรูปสัญญาณดิจิตอล (Digital pattern) ของสัญญาณที่ต้องการสังเคราะห์ ดังนั้นในระยะแรกๆ งานวิจัยเกี่ยวกับวงจรสังเคราะห์ความถี่แบบดิจิตอลจะมุ่งเน้นในเรื่องของเทคนิคการบีบอัดข้อมูลในหน่วยความจำเพื่อลดขนาดของหน่วยความจำ แต่ยังคงต้องรักษาคุณภาพของสัญญาณให้อยู่ในระดับที่นำไปใช้งานได้ แต่วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลแบบที่ใช้หน่วยความจำจะมีข้อจำกัดในเรื่องของเวลาในการเข้าถึงหน่วยความจำ, ขนาดวงจร และการกินกำลังงาน ในระยะหลังจึงได้มีงานวิจัยเกี่ยวกับวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลจำนวนมากที่ใช้การคำนวณค่าของสัญญาณแทนการเก็บค่าและบีบอัดรูปสัญญาณไว้ในหน่วยความจำ

เนื้อหาในบทนี้จะเกี่ยวกับทฤษฎีและหลักการทำงานของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล โดยจะแบ่งออกเป็นสองประเภทหลักๆ คือแบบที่ใช้หน่วยความจำ (ROM-based DDFS) และแบบที่ไม่ใช้หน่วยความจำ (ROM-less DDFS) ส่วนในหัวข้อสุดท้ายจะเป็นการเปรียบเทียบถึงคุณสมบัติและความเหมาะสมในการนำวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลทั้งสองประเภทไปประยุกต์ใช้ในงานต่างๆ เพื่อให้เกิดประโยชน์สูงสุด

2.1 สถาปัตยกรรมของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล

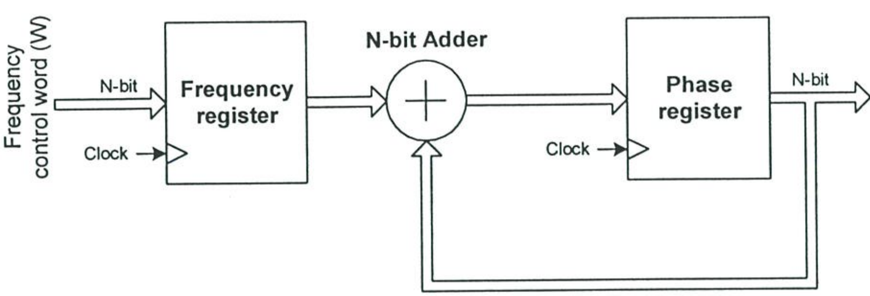


รูปที่ 2.1 สถาปัตยกรรมของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอล

รูปที่ 2.1 แสดงสถาปัตยกรรมของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลซึ่งประกอบด้วยวงจรสร้างเฟส (Phase accumulator), วงจรเปลี่ยนเฟสเป็นแอมปลิจูด (Phase-to-amplitude converter), วงจรแปลงดิจิตอลเป็นอนาลอก (Digital-to-analog converter) และวงจรกรองความถี่ต่ำผ่าน (Low-pass filter)

วงจรสร้างเฟสทำหน้าที่เป็นวงจรบวกที่มีอินพุตสองอินพุต คือรีจิสเตอร์ความถี่ (Frequency register) ขนาด N บิต และผลลัพธ์จากการบวกที่เกิดขึ้นในครั้งก่อนหน้าซึ่งถูกเก็บไว้ในรีจิสเตอร์เฟส (Phase register) ค่าของเฟสที่ถูกเก็บไว้ในรีจิสเตอร์เฟสจะถูกตัดทอน (Truncated) ให้เหลือ M บิต ซึ่งเฟสที่ถูกตัดทอนเหลือ M บิต นี้จะถูกวงจรเปลี่ยนเฟสเป็นแอมปลิจูดนำไปเปลี่ยนเป็นแอมปลิจูดของรูปสัญญาณ (Waveform) ที่ต้องการ ผลลัพธ์ที่ได้จากวงจรเปลี่ยนเฟสเป็นแอมปลิจูดจะมีลักษณะเป็นรูปสัญญาณดิจิตอล (Digital pattern) ณ เวลาต่างๆ ซึ่งรูปสัญญาณดิจิตอลเหล่านี้จะถูกเปลี่ยนเป็นสัญญาณอนาลอกโดยวงจรแปลงดิจิตอลเป็นอนาลอก จากนั้นก่อนที่จะนำสัญญาณที่สังเคราะห์ได้ไปใช้งานวงจรกรองความถี่ต่ำผ่านจะกรองเอาความถี่แอบแฝง (Aliasing frequency) ออกก่อน รายละเอียดการทำงานในแต่ละส่วนของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลจะอธิบายไว้ในหัวข้อ 2.1.1, 2.1.2, 2.1.3 และ 2.1.4 ตามลำดับ

2.1.1 **วงจรสร้างเฟส** (Phase accumulator)

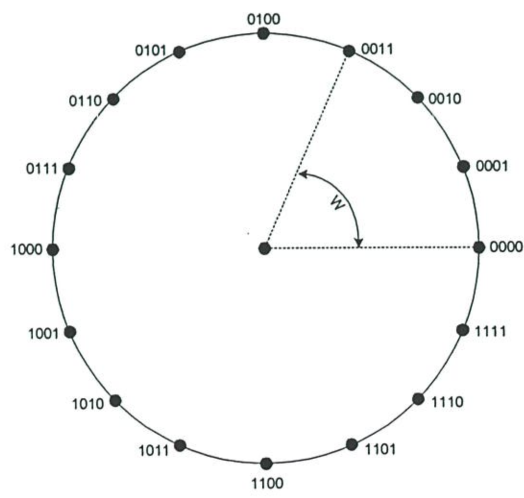


รูปที่ 2.2 วงจรสร้างเฟส

วงจรสร้างเฟสมีส่วนประกอบที่สำคัญคือรีจิสเตอร์ความถี่ (Frequency register), วงจรบวก (Adder) และรีจิสเตอร์เฟส (Phase register) ดังรูปที่ 2.2 โดยวงจรสร้างเฟสจะทำการบวกค่าควบคุมความถี่ (Frequency control word, W) ที่ถูกเก็บไว้ในรีจิสเตอร์ความถี่เข้ากับค่าของผลบวกในครั้งก่อนหน้าซึ่งถูกเก็บไว้ในรีจิสเตอร์เฟส มีความสัมพันธ์ตามสมการ

$$S(n) = S(n-1) + W \tag{2.1}$$

โดย $S(n)$ คือผลลัพธ์ของการบวกซึ่งจะถูกเก็บไว้ในรีจิสเตอร์เฟส, $S(n\text{-}1)$ คือค่าผลลัพธ์ที่เกิดจากการบวกครั้งก่อนหน้า และ W คือค่าควบคุมความถี่ โดยที่ $S(n)$, $S(n\text{-}1)$ และ W มีขนาด N บิต และอัตราการเปลี่ยนค่าของ $S(n)$ ถูกควบคุมด้วยจังหวะของสัญญาณนาฬิกา (Clock)



รูปที่ 2.3 ค่าเฟสเมื่อพิจารณาเป็นการหมุนรอบวงกลม

เพื่อเป็นการอธิบายให้เห็นภาพการทำงานของวงจรสร้างเฟสได้ชัดเจนยิ่งขึ้น เราจะสมมติให้การเพิ่มเฟสในแต่ละรอบสัญญาณนาฬิกาเสมือนกับการหมุนรอบวงกลม ซึ่งในเส้นรอบวงกลมจะมีจำนวนจุดทั้งหมด 2^N จุด และถ้าเราสมมติให้รีจิสเตอร์ความถี่ และรีจิสเตอร์เฟสมีขนาด 4 บิต นั่นคือ N=4 ดังนั้นเราจะได้จุดบนเส้นรอบวงกลมทั้งหมด 2^4 = 16 จุด ดังรูปที่ 2.3 ซึ่งค่าเฟสเหล่านี้จะถูกวงจรเปลี่ยนเฟสเป็นแอมปลิจูดนำไปใช้สร้างแอมปลิจูดของสัญญาณซายน์ที่ต้องการ จุดเริ่มต้นของค่าเฟสจะเริ่มที่ 0000 และในแต่ละรอบของสัญญาณนาฬิกาจะเพิ่มค่าขึ้นครั้งละ W และเมื่อค่าเฟสถูกเพิ่มค่าไปเรื่อยๆ จนมีค่ามากกว่า 1111_2 หรือ 15_{10} ผลลัพธ์ของการเพิ่มค่าในรอบสัญญาณนาฬิกาถัดไปจะมีค่ามากกว่า 15_{10} ซึ่งต้องแทนค่าด้วยรีจิสเตอร์ที่มีขนาดมากกว่า 4 บิต แต่เนื่องจากรีจิสเตอร์ความถี่และรีจิสเตอร์เฟสที่ใช้มีขนาด 4 บิต จะทำให้บิตบนถูกตัดทิ้งไป นั่นคือค่าเฟสที่ได้ก็จะวนซ้ำค่าเดิม เราจะถือว่าการเคลื่อนที่ของเฟสวนรอบวงกลมครบหนึ่งรอบนั้นก็คือการที่รูปคลื่นของสัญญาณซายน์ครบรอบนั่นเอง การเปลี่ยนค่าของ W นั้นจะมีผลต่อค่าของเฟสที่เกิดขึ้นในรอบสัญญาณนาฬิกาแต่ละรอบ และจำนวนรอบของสัญญาณนาฬิกาที่ใช้ในการวนเฟสครบหนึ่งรอบ (Cycle) ของสัญญาณ ซึ่งจำนวนรอบเหล่านี้จะเป็นตัวกำหนดค่าความถี่ของสัญญาณเอาต์พุตที่ต้องการ ความสัมพันธ์ของค่า W กับความถี่ของสัญญาณเอาต์พุตสามารถเขียนได้ดังนี้

$$f_{out} = \frac{W * f_{clk}}{2^N} \qquad (2.2)$$

โดย f_{out} คือความถี่ของสัญญาณเอาต์พุต, W คือค่าควบคุมความถี่, f_{clk} คือความถี่ของสัญญาณนาฬิกา และ N คือจำนวนบิตของรีจิสเตอร์ความถี่ จากสมการที่ 2.2 เราสามารถหาค่าความละเอียดในการปรับความถี่ (Frequency resolution) ของวงจรสังเคราะห์ความถี่แบบดิจิตอลได้โดยให้ W มีค่าเท่ากับ 1 ซึ่งสามารถเขียนเป็นสมการได้ดังนี้

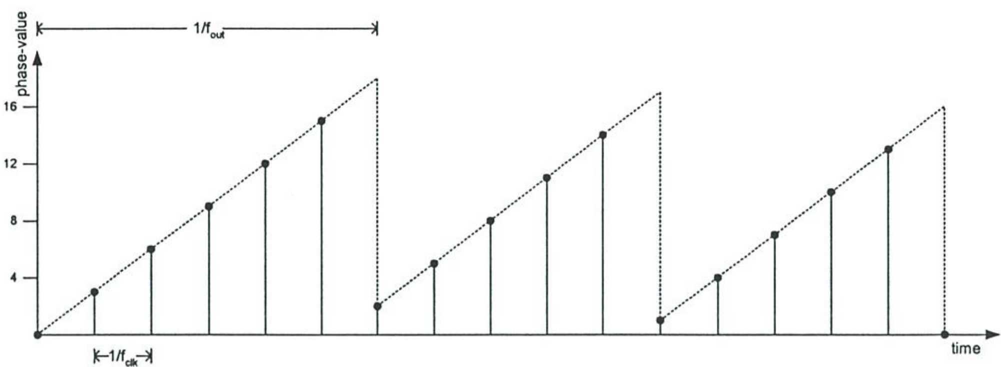
$$f_{res} = \frac{f_{clk}}{2^N} \qquad (2.3)$$

โดย f_{res} คือค่าความละเอียดในการปรับความถี่ ตัวอย่างเช่นในกรณีที่รีจิสเตอร์ความถี่มีขนาด 32 บิต และวงจรทำงานที่สัญญาณนาฬิกาความถี่ 100 เมกะเฮิรตซ์จะมีค่าความละเอียดในการปรับความถี่ประมาณ $(100*10^6)/2^{32}$ = 0.0233 เฮิรตซ์

จากสมการที่ (2.2) ถ้าลองพิจารณาถึงวงจรสร้างเฟสที่มีค่า N เท่ากับ 4 บิต และกำหนดให้ W มีค่าเท่ากับ 3 เราสามารถแสดงค่าเอาต์พุตและรูปสัญญาณของวงจรสร้างเฟสในแต่ละรอบจังหวะสัญญาณนาฬิกา f_{clk} ได้ดังตารางที่ 2.1 และรูปที่ 2.4 ตามลำดับ

ตารางที่ 2.1 ค่าเอาต์พุตของวงจรสร้างเฟสในแต่ละรอบสัญญาณนาฬิกา เมื่อ $N=4$ และ $W=3$

รอบที่	**เอาต์พุตของวงจรสร้างเฟส**	**สัญญาณทด** (Carry)
1	0000 (0)	1 (เริ่มต้นรอบ)
2	0011 (3)	0
3	0110 (6)	0
4	1001 (9)	0
5	1100 (12)	0
6	1110 (15)	0
7	0010 (2)	1
8	0101 (5)	0
9	1000 (8)	0
10	1011 (11)	0
11	1110 (14)	0
12	0001 (1)	1
13	0100 (4)	0
14	0111 (7)	0
15	1010 (10)	0
16	1101 (13)	0
17	0000 (0)	1



รูปที่ 2.4 สัญญาณเอาต์พุตของวงจรสร้างเฟสในแต่ละรอบสัญญาณนาฬิกา เมื่อ $N=4$ และ $W=3$

จากการที่โครงสร้างหลักของวงจรสร้างเฟสมีลักษณะเป็นวงจรบวก (Adder) ซึ่งจะมีสัญญาณค่าทด (Carry chain) เป็นผลทำให้เกิดเวลาประวิงการแพร่กระจาย (Propagation delay) ซึ่งถ้าวงจรสร้างเฟสมีเวลาประวิงการแพร่กระจายมากจะทำให้ความเร็วในการทำงานของวงจรลดลง นั่นคือความถี่ของสัญญาณนาฬิกาและสัญญาณเอาต์พุตจะลดลงไปด้วย

วิธีที่ง่ายที่สุดในการลดเวลาประวิงการแพร่กระจายก็คือการลดจำนวนบิตของรีจิสเตอร์ความถี่ลง แต่มีข้อเสียคือจำนวนบิตของรีจิสเตอร์ความถี่ที่ลดลงจะส่งผลทำให้การปรับความถี่มีความละเอียดต่ำลงไปด้วย วิธีที่นิยมใช้สำหรับการลดค่าเวลาประวิงก็คือการพัฒนาวงจรบวกให้มีเวลาประวิงการแพร่กระจายของสัญญาณค่าทดน้อยลง เช่น วงจรบวกแบบสร้างตัวทดล่วงหน้า (Carry-look-ahead adder) [2] เป็นต้น หรือใช้เทคนิคของการทำงานแบบสายท่อ (Pipelining technique) [3-4]

วงจรสร้างเฟสส่วนใหญ่มักจะประมวลผลด้วยระบบเลขฐานสองเพื่อให้ง่ายต่อการสร้างเป็นฮาร์ดแวร์ แต่การประมวลผลด้วยเลขฐานสองนั้นจะยากต่อการใช้งานของผู้ใช้จึงได้มีการพัฒนาวงจรสร้างเฟสที่ทำงานด้วยระบบเลขฐานสิบหรือที่เรียกว่าวงจรสร้างเฟสแบบบีซีดี (BCD phase accumulator) เพื่อให้สะดวกต่อการใช้งานของผู้ใช้มากยิ่งขึ้น [5]

2.1.2 วงจรเปลี่ยนเฟสเป็นแอมปลิจูด (Phase-to-amplitude converter)

ค่าของเฟสที่ได้จากวงจรสร้างเฟสจะถูกเปลี่ยนให้เป็นแอมปลิจูดของสัญญาณที่ต้องการสังเคราะห์โดยวงจรเปลี่ยนเฟสเป็นแอมปลิจูด สำหรับในกรณีอุดมคติที่ไม่มีการลดทอนค่าของแอมปลิจูด (Amplitude quantization) ค่าของแอมปลิจูดจะมีความสัมพันธ์กับค่าของรีจิสเตอร์เฟสดังนี้

$$\sin\left(2\pi\frac{P(n)}{2^N}\right) \tag{2.4}$$

โดย $P(n)$ คือค่าของรีจิสเตอร์เฟสที่สัญญาณนาฬิกาที่ n และ N คือขนาดของรีจิสเตอร์เฟส

วิธีการเปลี่ยนเฟสเป็นแอมปลิจูดนั้นสามารถกระทำได้หลายวิธี [5-7] สามารถสรุปเป็นประเภทหลักๆ ได้สองประเภทดังนี้

2.1.2.1 <u>วงจรเปลี่ยนเฟสเป็นแอมปลิจูดที่ใช้หน่วยความจำ</u>

วิธีการเปลี่ยนเฟสให้เป็นแอมปลิจูดของวงจรสังเคราะห์ความถี่แบบที่ใช้หน่วยความจำจะใช้วิธีการเก็บค่าแอมปลิจูดของสัญญาณที่ต้องการสังเคราะห์ลงในหน่วยความจำ เช่น หน่วยความจำแบบอ่านอย่างเดียว (Read-only memory, ROM) โดยค่าของแอมปลิจูดที่ออกจากหน่วยความจำจะมีขนาดความกว้างเท่ากับจำนวนบิตของวงจรแปลงดิจิตอลเป็นอนาลอก ค่าของแอมปลิจูดที่จะถูกดึงออกจากหน่วยความจำจะสัมพันธ์กับค่าของเฟสที่ได้รับจากวงจรสร้างเฟส กล่าวคือค่าของเฟสจะถูกใช้เป็นตำแหน่ง (Address) สำหรับอ้างอิงหน่วยความจำในการดึงค่าแอม-ปลิจูดออกมา ตัวอย่างและงานวิจัยเกี่ยวกับวงจรเปลี่ยนเฟสเป็นแอมปลิจูดชนิดที่ใช้หน่วยความจำจะถูกกล่าวถึงอย่างละเอียดในหัวข้อ 2.2

2.1.2.2 <u>วงจรเปลี่ยนเฟสเป็นแอมปลิจูดที่ไม่ใช้หน่วยความจำ</u>

วงจรเปลี่ยนเฟสเป็นแอมปลิจูดแบบที่ไม่ใช้หน่วยความจำจะใช้วิธีการคำนวณค่าเฟสที่รับมาจากวงจรสร้างเฟสให้เป็นค่าแอมปลิจูดของสัญญาณที่ต้องการสังเคราะห์ ซึ่งสามารถทำได้หลายวิธีเช่น การประมาณค่าโพลีโนเมียล (Polynomial approximation), การประมาณค่าพาราโบลา (Parabolic approximation), การคูณจำนวณเชิงซ้อน (Complex multiplication) ฯลฯ ตัวอย่างและงานวิจัยเกี่ยวกับวงจร

เปลี่ยนเฟสเป็นแอมปลิจูดชนิดที่ไม่ใช้หน่วยความจำจะถูกกล่าวถึงอย่างละเอียดในหัวข้อ 2.3

2.1.3 **วงจรแปลงดิจิตอลเป็นอนาลอก** (Digital-to-analog converter)

จากสถาปัตยกรรมของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลในรูปที่ 2.1 จะเห็นว่าสัญญาณแอมปลิจูดที่สร้างขึ้นจากวงจรเปลี่ยนเฟสเป็นแอมปลิจูดจะอยู่ในรูปของสัญญาณดิจิตอล ดังนั้นในกรณีที่ต้องการสังเคราะห์สัญญาณอนาลอกขึ้นใช้งานจึงจำเป็นต้องมีวงจรแปลงดิจิตอลเป็นอนาลอกทำหน้าที่เปลี่ยนแอมปลิจูดดิจิตอลให้อยู่ในรูปสัญญาณอนาลอก

สัญญาณเอาต์พุตของวงจรแปลงดิจิตอลเป็นอนาลอกจะมีความสัมพันธ์กับค่าแอมปลิจูดดิจิตอลอินพุตดังสมการ

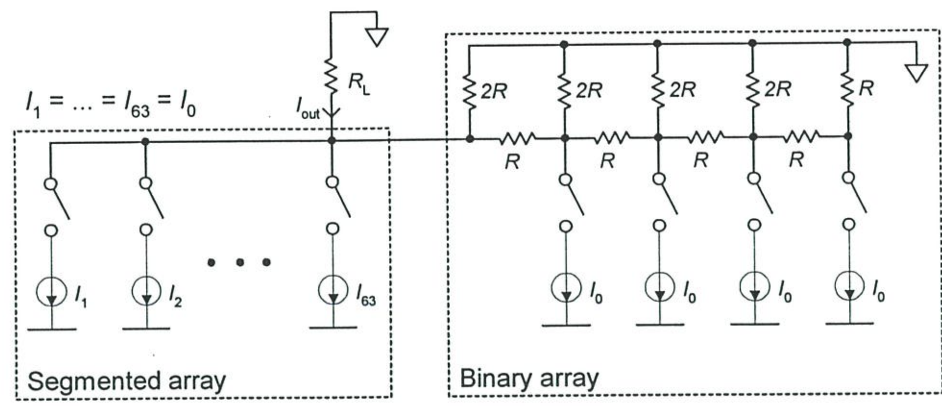
$$I_{out} = \frac{I_{ref} * W}{2^D}, 0 < W < 2^D - 1 \quad (2.5)$$

โดย I_{out} คือกระแสเอาต์พุตของวงจรแปลงดิจิตอลเป็นอนาลอก, W คือค่าแอมปลิจูดดิจิตอลอินพุต, D คือจำนวนบิตของวงจรแปลงดิจิตอลเป็นอนาลอก และ I_{ref} เป็นค่ากระแสอ้างอิงที่ป้อนให้กับวงจรแปลงดิจิตอลเป็นอนาลอก

ถ้าแทนค่า W ให้เท่ากับ 1 ลงในสมการที่ (2.5) เราจะได้ค่าความละเอียด (I_{res}) ของวงจรแปลงดิจิตอลเป็นอนาลอกดังนี้

$$I_{res} = \frac{I_{ref}}{2^D} \quad (2.6)$$

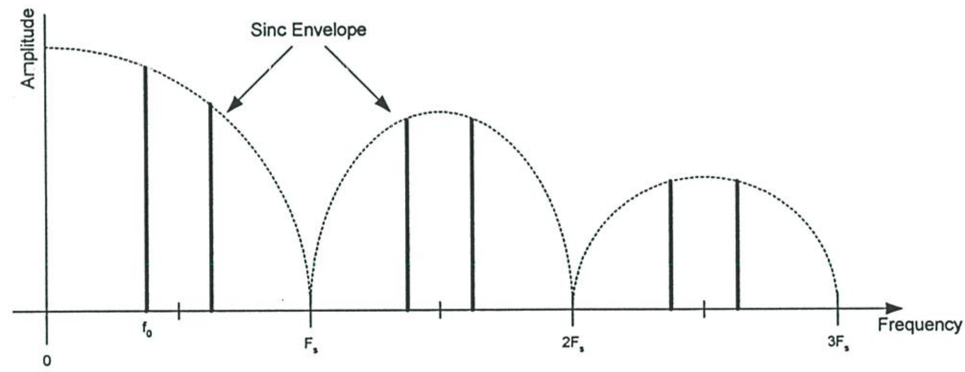
โดยมากแล้ววงจรแปลงดิจิตอลเป็นอนาลอกที่ใช้ในวงจรสังเคราะห์ความถี่แบบดิจิตอลมักจะมีสถาปัตยกรรมแบบแยกส่วน (Segmented architecture) [8] ดังแสดงในรูปที่ 2.5 เพราะการออกแบบวงจรแปลงดิจิตอลเป็นอนาลอกด้วยสถาปัตยกรรมแบบแยกส่วนนั้นจะทำให้การเข้ากัน (Matching) ในการออกแบบเลย์เอาต์ (Layout) ทำได้ง่ายขึ้น



รูปที่ 2.5 วงจรแปลงดิจิตอลเป็นอนาลอกที่มีสถาปัตยกรรมแบบแยกส่วน

2.1.4 **วงจรกรองความถี่ต่ำผ่าน** (Low-pass filter)

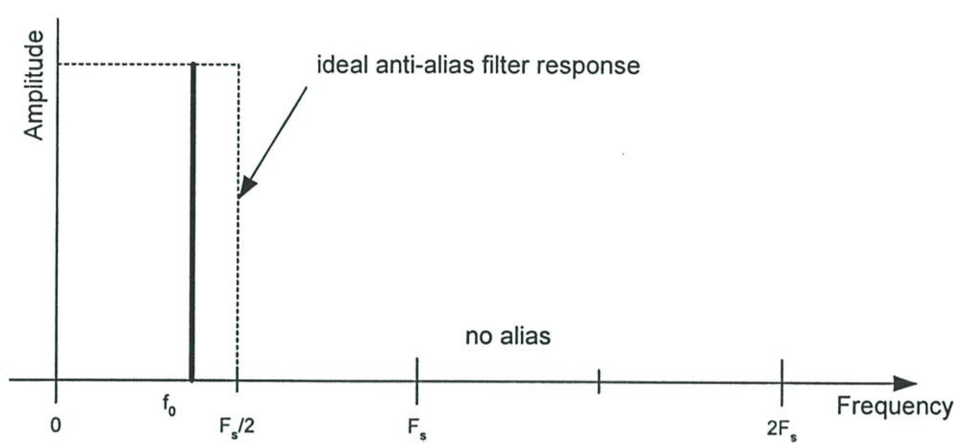
เนื่องจากการทำงานของวงจรสังเคราะห์ความถี่แบบดิจิตอลเป็นการทำงานโดยใช้หลักการสุ่มสัญญาณ (Sampling) ถ้ากำหนดให้วงจรสังเคราะห์ความถี่แบบดิจิตอลมีสัญญาณเอาต์พุต f_0 อยู่ในช่วง 0 ถึง $F_s/2$ (F_s คือความถี่ของสัญญาณนาฬิกาของระบบ) เราจะสามารถแสดงสเปกตรัมของสัญญาณเอาต์พุตได้ดังรูปที่ 2.6



รูปที่ 2.6 สเปกตรัมของสัญญาณเอาต์พุตของวงจรสังเคราะห์ความถี่แบบดิจิตอล

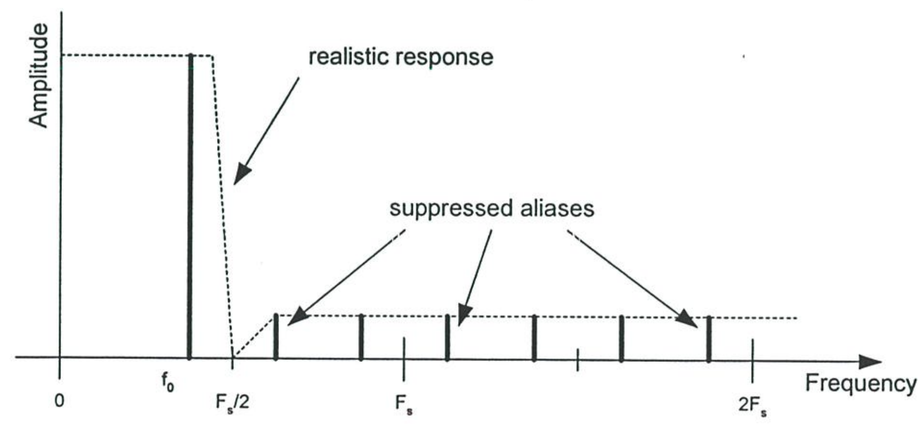
จะเห็นว่าสเปกตรัมของสัญญาณเอาต์พุตในรูปที่ 2.6 จะประกอบด้วยสเปกตรัมของสัญญาณความถี่ f_0 ที่ต้องการสังเคราะห์และยังมีสเปกตรัมของความถี่แอบแฝงอยู่ด้วย นอกจากนี้ขนาดของแอมปลิจูดของสัญญาณเอาต์พุตจะลดลงตามลักษณะการตอบสนองฟังก์ชั่น sin(x)/x [9] ซึ่งอันที่จริงแล้วสัญญาณที่เราต้องการสังเคราะห์จริงๆ นั้นคือสัญญาณที่ความถี่ f_0 เพียงสัญญาณเดียว ดังนั้นจึงจำเป็นอย่างยิ่งที่จะต้องมีการกำจัดสัญญาณความถี่แอบแฝงเหล่านี้ออกให้หมดก่อนที่จะนำสัญญาณเอาต์พุตไปใช้งานจริง

วงจรที่ใช้กำจัดความถี่แอบแฝงหรือวงจรกรองความถี่แอบแฝง (Anti-alias filter) นี้นิยมออกแบบด้วยวงจรกรองความถี่ต่ำผ่าน (Low-pass filter) ที่มีผลตอบสนองทางความถี่ในอุดมคติเท่ากับ 1 ในช่วง 0 ถึง $F_s/2$ และเท่ากับ 0 ในช่วงความถี่ $F_s/2$ ขึ้นไป ดังรูปที่ 2.7



รูปที่ 2.7 ผลตอบสนองทางความถี่ของวงจรกรองความถี่แอบแฝงในอุดมคติ

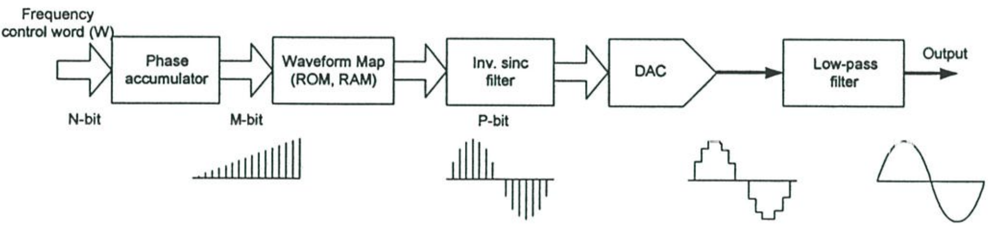
แต่ในความเป็นจริงแล้วการออกแบบวงจรกรองให้มีผลตอบสนองทางความถี่เป็นดังรูปที่ 2.7 นั้นเป็นไปไม่ได้ในทางปฏิบัติ จึงนิยมออกแบบวงจรกรองความถี่แอบแฝงด้วยวงจรกรองความถี่ต่ำผ่านที่มีผลตอบสนองทางความถี่เท่ากับ 1 ในช่วงความถี่ 0 ถึง 40 เปอร์เซนต์ของความถี่ระบบ ดังแสดงในรูปที่ 2.8



รูปที่ 2.8 ผลตอบสนองทางความถี่ของวงจรกรองความถี่แอบแฝงทั่วไป

สำหรับรูปแบบของวงจรกรองความถี่ต่ำผ่านให้ทำงานเป็นวงจรกรองความถี่แอบแฝงนั้น เราสามารถออกแบบได้หลายแบบขึ้นกับความต้องการและลักษณะของงานที่จะนำไปใช้ [10]

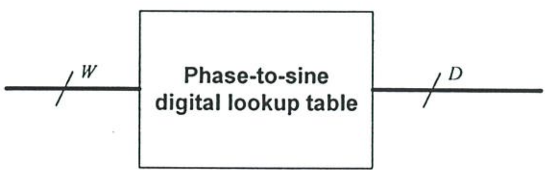
คุณสมบัติอีกประการหนึ่งของทฤษฎีการสุ่มสัญญาณ (Sampling theorem) ก็คือ แอมปลิจูดของสัญญาณเอาต์พุตของวงจรสังเคราะห์ความถี่แบบดิจิตอลจะมีขนาดสัมพันธ์กับผลการตอบสนองฟังก์ชั่น sin(x)/x ดังแสดงไว้แล้วในรูปที่ 2.6 ซึ่งเราสามารถชดเชยแอมปลิจูดของสัญญาณที่ต้องการสังเคราะห์ได้โดยใช้วงจรกรองอินเวอร์สซิงค์ (Inverse-sinc filter) [11] ชดเชยสัญญาณก่อนที่จะถูกแปลงเป็นสัญญาณอนาลอกโดยวงจรแปลงดิจิตอลเป็นอนาลอกดังรูปที่ 2.9



รูปที่ 2.9 การชดเชยแอมปลิจูดของสัญญาณด้วยวงจรกรองอินเวอร์สซิงค์

2.2 การสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

การสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำเป็นวิธีที่นิยมใช้กันมากที่สุด เนื่องจากเป็นวิธีที่สะดวกต่อการออกแบบ และเหมาะกับการสังเคราะห์รูปสัญญาณอื่นๆ นอกเหนือจากสัญญาณมาตรฐานเช่นสัญญาณรูปซายน์ (Sine) รูปสี่เหลี่ยม (Square) รูปฟันเลื่อย (Saw-tooth) ฯลฯ เพราะว่าการเปลี่ยนรูปสัญญาณสามารถทำได้อย่างง่ายๆ โดยการเปลี่ยนค่าแอมปลิจูดที่เก็บไว้ในหน่วยความจำให้เป็นไปตามรูปสัญญาณที่ต้องการสังเคราะห์



รูปที่ 2.10 หน่วยความจำสำหรับเปลี่ยนเฟสเป็นแอมปลิจูดสัญญาณซายน์

สำนักหอสมุดกลาง พระจอมเกล้าลาดกระบัง

โดยทั่วไปแล้วหน้าที่การทำงานหลักของหน่วยความจำก็คือการเปลี่ยนค่าของเฟส (φ) ให้เป็นแอมปลิจูดของสัญญาณซายน์ของเฟสนั้นๆ (sin φ) ถ้ากำหนดให้ตำแหน่งของหน่วยความจำมีขนาด W บิต (2^W ตำแหน่ง) และสัญญาณแอมปลิจูดเอาต์พุตของหน่วยความจำมีความกว้างทั้งหมด D บิต ดังรูปที่ 2.10 จะได้ว่าค่าแอมปลิจูด $s(j)$ ที่เก็บไว้ในตำแหน่ง (j) แต่ละตำแหน่งในหน่วยความจำมีค่าดังสมการ [5]

$$s(j) = 2^{D-1} + \text{int}\left[\left(2^{D-1} - 1\right)\sin\frac{2\pi j}{2^W} + 0.5\right] \qquad , j = 0 \text{ to } 2^W\text{-}1 \qquad (2.7)$$

ข้อเสียที่สำคัญของวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำก็คือการสังเคราะห์สัญญาณให้มีคุณภาพสูงนั้นจำเป็นต้องใช้หน่วยความจำที่มีขนาดใหญ่สำหรับเก็บรูปสัญญาณให้ได้มากที่สุด ซึ่งการใช้หน่วยความจำขนาดใหญ่จะทำให้ขนาดของวงจรใหญ่ขึ้น, การกินกำลังงานสูงขึ้น และความเร็วลดลง จากผลกระทบของการใช้หน่วยความจำที่มีขนาดใหญ่สังเคราะห์สัญญาณให้มีคุณภาพสูงนี้ จึงได้มีการวิจัยเกี่ยวกับการบีบอัดหน่วยความจำ (Memory compression) เพื่อสังเคราะห์สัญญาณให้มีคุณภาพเทียบเท่าหรือใกล้เคียงกับการสังเคราะห์สัญญาณโดยไม่มีการบีบอัดหน่วยความจำ [5-6] เราสามารถสรุปวิธีการบีบอัดหน่วยความจำที่นิยมใช้กันในการสังเคราะห์ความถี่แบบดิจิตอลได้ดังนี้

2.2.1 **การบีบอัดควอแดรนท์** (Quadrant compression)

การบีบอัดควอแดรนท์ใช้หลักการของการสมมาตร (Symmetry) ของสัญญาณซายน์ในแต่ละควอแดรนท์ [5-6] นั่นคือสำหรับ $0 \le a \le 90$ แล้วจะได้ว่า

$$\begin{aligned} &\sin(90-a) = \sin(90+a), \\ &\sin(270-a) = \sin(270+a), \\ &\sin a = -\sin(a), \\ &\sin a = -\sin(180+a) \end{aligned} \qquad (2.8)$$

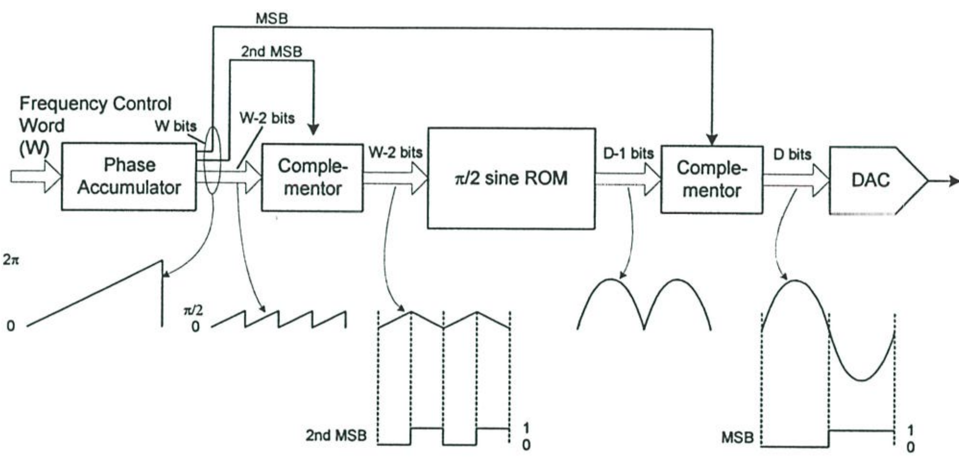
จากสมการ (2.8) จะเห็นได้ว่าเราสามารถหาค่าซายน์สำหรับทุกควอแดรนท์ ($0 < a < 360$) ได้จากค่า a ของควอแครนท์แรกเพียงควอแดรนท์เดียว ($0 < a < 90$) โดยในการการออกแบบจริงนั้นเราจะนำบิตที่มีนัยสำคัญสูงสุด (MSB) หรือเรียกว่าบิตเอ็มเอสบี 2 บิตแรกมากำหนดควอแดรนท์ของสัญญาณซายน์ที่จะสังเคราะห์ ความสัมพันธ์ของบิตเอ็มเอสบีกับควอแดรนท์และค่าแอมปลิจูดสัญญาณซายน์สามารถสรุปได้ดังตารางที่ 2.2 และรูปที่ 2.11 ตามลำดับ

47584

ตารางที่ 2.2 ความสัมพันธ์ของบิตเอ็มเอสบีกับควอแดรนท์และสัญญาณซายน์

Quadrant	MSB	MSB-1	Sine
Quadrant I	0	0	½ + ½ sin(I)
Quadrant II	0	1	½ + ½ sin(I complemented)
Quadrant III	1	0	½ - ½ sin(I complemented)
Quadrant IV	0	1	½ - ½ sin(I)

* I เป็นดัชนี (Index) เริ่มจาก 0 ถึง $2^{W-2}-1$



รูปที่ 2.11 การสังเคราะห์สัญญาณซายน์เต็มรูปด้วยเทคนิคบีบอัดควอแดรนท์

จากรูปที่ 2.11 หน่วยความจำ (ROM) จะเก็บแอมปลิจูดของสัญญาณซายน์ในช่วง 0 ถึง $\pi/2$ บิตเอ็มเอสบี 2 บิต บนจะถูกใช้สำหรับกำหนดควอแดรนท์ของสัญญาณซายน์ และบิตที่เหลือ W-2 บิต จะถูกใช้สำหรับชี้ตำแหน่ง ของหน่วยความจำเพื่อดึงแอมปลิจูดสัญญาณซายน์ที่ต้องการออกมา บิตเอ็มเอสบีบนสุด (MSB) ทำหน้าที่กำหนดเครื่องหมาย (Sign) ของสัญญาณเอาต์พุต และบิตเอ็มเอสบีที่สอง (2nd MSB) กำหนดว่าแอมปลิจูดของสัญญาณเอาต์พุตเป็นขาขึ้นหรือขาลง กล่าวคือในควอแดรนท์ที่ 1 และ 3 สัญญาณเอาต์พุตของวงจรสร้างเฟส (Phase accumulator) จะถูกนำมาใช้โดยตรง ส่วนควอแดรนท์ที่ 2 และ 4 นั้นสัญญาณเอาต์พุตของวงจรสร้างเฟสจะถูกคอมพลีเมนต์ (Complemented) สัญญาณแอมปลิจูดเอาต์พุตที่ออกจากหน่วยความจำจะมีลักษณะเป็นสัญญาณเต็มลูกคลื่น (Full wave) ซึ่งสามารถเปลี่ยนให้เป็นสัญญาณซายน์ได้โดยการคูณแอมปลิจูดของสัญญาณที่อยู่ในช่วง π ถึง 2π ด้วย -1

จะเห็นได้ว่าวิธีการบีบอัดควอแดรนท์นี้สามารถใช้ข้อมูลของแอมปลิจูดของสัญญาณซายน์เพียงควอแดรนท์เดียวสร้างสัญญาณซายน์ทั้งสี่ควอแดรนท์หรือสัญญาณซายน์เต็มลูกคลื่นได้ นั่นหมายถึงวิธีการบีบอัดควอแดรนท์นั้นสามารถลดขนาดของหน่วยความจำได้ถึง 75 เปอร์เซ็นต์

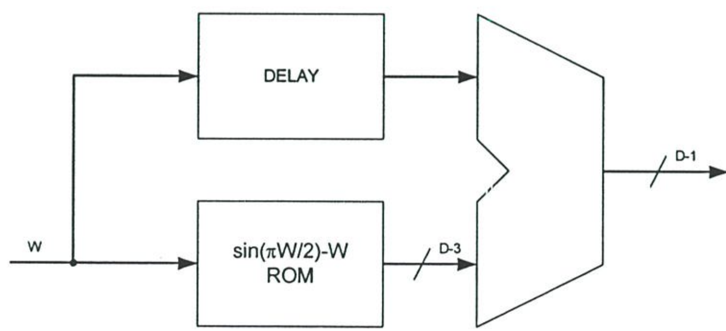
ด้วยเหตุนี้วิธีการบีบอัดควอแดรนท์จึงมักจะถูกนำมาใช้สำหรับการออกแบบวงจรสังเคราะห์ความถี่แบบที่ใช้หน่วยความจำอยู่เสมอ ซึ่งในการออกแบบจริงๆ แล้ว นอกจากการบีบอัดควอแดรนท์แล้วมักจะมีการบีบอัดข้อมูลเพิ่มเติมต่อไปอีก สรุปเป็นวิธีต่างๆ ได้ดังนี้

2.2.2 **ขั้นตอนวิธีผลต่างซายน์-เฟส** (Sine-phase difference algorithm)

ขั้นตอนวิธีผลต่างซายน์-เฟสนี้หน่วยความจำจะเก็บค่าของฟังก์ชั่น $f(W)$ ซึ่งเป็นค่าผลต่างของแอมปลิจูดซายน์กับค่าเฟสที่เฟสนั้นๆ ฟังก์ชั่น $f(W)$ สามารถเขียนเป็นสมการได้ดังนี้

$$f(W) = \sin\left(\frac{\pi W}{2}\right) - W \quad (2.9)$$

สัญญาณซายน์ที่ต้องการจะถูกสังเคราะห์โดยการนำค่าแอมปลิจูดจากหน่วยความจำมาบวกกับค่าของเฟสที่ต้องการสังเคราะห์ ดังแสดงในรูปที่ 2.12



รูปที่ 2.12 ขั้นตอนวิธีผลต่างซายน์-เฟส

วิธีการเก็บค่าของ *f(W)* แทนการเก็บค่า *sin(πW/2)* สามารถลดจำนวนบิตของแอมปลิจูดได้ 2 บิต [12] เพราะ

$$\max\left[\sin\left(\frac{\pi W}{2}\right)-W\right]\approx 0.21\max\left[\sin\left(\frac{\pi W}{2}\right)\right] \quad (2.10)$$

จะเห็นว่าในขั้นตอนวิธีผลต่างซายน์-เฟสนี้จะมีฮาร์ดแวร์ของวงจรบวก (Adder) สำหรับทำฟังก์ชั่น

$$\left[\sin\left(\frac{\pi W}{2}\right)-W\right]+W \quad (2.11)$$

จากการที่ขั้นตอนวิธีผลต่างซายน์-เฟสสามารถลดจำนวนบิตของแอมปลิจูดเอาต์พุตได้ นั่นคือขนาดของหน่วยความจำจะลดลง เป็นผลทำให้วงจรสังเคราะห์ความถี่จะสามารถทำงานได้เร็วขึ้น

2.2.3 ขั้นตอนวิธีซันเดอร์แลนด์ (Sunderland algorithm)

ขั้นตอนวิธีซันเดอร์แลนด์ [13] ถูกพัฒนาต่อจากขั้นตอนวิธีฮัทชิสัน (Hutchison algorithm) [5] ขั้นตอนวิธีซันเดอร์แลนด์มีหลักการทำงานโดยการแบ่งสัญญาณเฟส (x) ที่ได้รับจากวงจรสร้างเฟสออกเป็นสามส่วน (a, b, และ c) ดังนี้

$$\sin(x)=\sin(a+b+c) \quad (2.12)$$

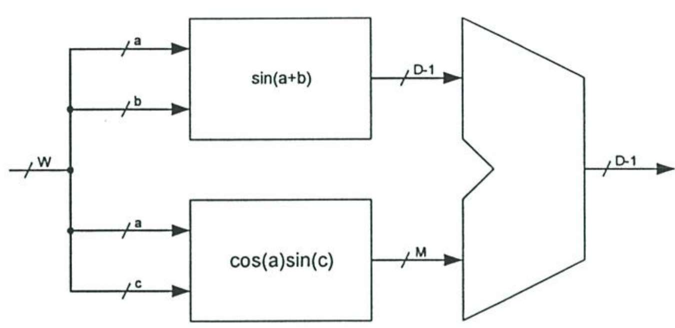
$$\sin(a+b+c)=\sin(a+b)\cos c+\cos a\cos b\sin c-\sin a\sin b\sin c \quad (2.13)$$

$$\sin(a+b+c)\approx\sin(a+b)+\cos a\sin c \quad (2.14)$$

และสัญญาณ $\sin(a+b+c)$ สามารถกระจายออกได้ดังสมการที่ (2.13) และประมาณค่าได้ดังสมการที่ (2.14) ตามลำดับ ซึ่งขั้นตอนวิธีซันเดอร์แลนด์นี้สามารถบีบอัดหน่วยความจำได้ประมาณ 11 เท่า [5]

จากสมการที่ (2.14) เราสามารถนำขั้นตอนวิธีซันเดอร์แลนด์มาสร้างเป็นฮาร์ดแวร์ได้โดยแบ่งหน่วยความจำออกเป็นสองส่วนคือหน่วยความจำหยาบ (Coarse ROM) และหน่วยความจำละเอียด (Fine ROM) โดยหน่วยความจำหยาบจะเก็บค่าแอมปลิจูดของ

$\sin(a+b)$ และหน่วยความจำละเอียดจะเก็บค่าแอมปลิจูดของ $\cos a \sin c$ ดังแสดงในรูปที่ 2.13



รูปที่ 2.13 ขั้นตอนวิธีซันเดอร์แลนด์

2.2.4 **การประมาณค่าอนุกรมเทย์เลอร์** (Taylor series approximation)

สำหรับการประมาณค่าอนุกรมเทย์เลอร์ สัญญาณเฟส W บิต จะถูกแบ่งเป็นสองส่วนคือส่วนบนขนาด u บิต และส่วนที่เหลือซึ่งเป็นส่วนล่างขนาด $W\text{-}u$ บิต [11] ค่าของแอมปลิจูดสัญญาณซายน์ที่ถูกประมาณจากอนุกรมเทย์เลอร์ของค่าเฟส u บิต จะเป็นดังนี้

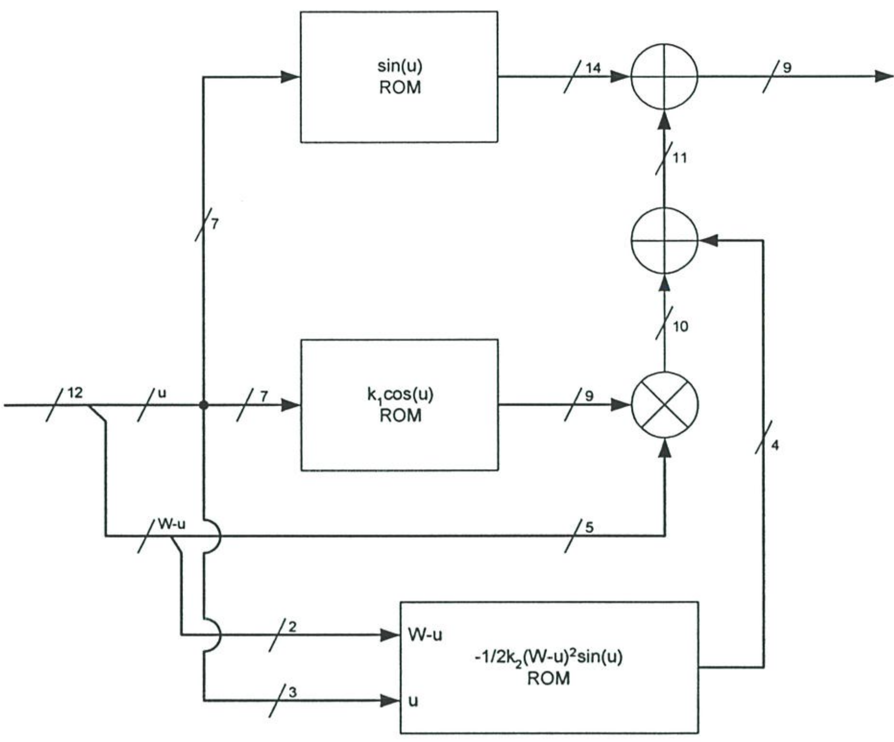
$$\sin(W) = \sin(u) + k_1(W-u)\cos(u) - \frac{k_2(W-u)^2\sin(u)}{2} + R_3 \qquad (2.15)$$

โดย k_n คือค่าคงที่สำหรับการปรับค่าของพจน์แต่ละพจน์ในอนุกรม และ

$$R_n = \frac{d^n(\sin(r))}{dr}\frac{(W-u)^n}{n!}, \qquad r \in [u,W] \qquad (2.16)$$

เนื่องจากค่าสูงสุดของฟังก์ชั่น sine และ cosine คือ 1 ดังนั้นเราสามารถสรุปได้ว่าค่าของความถูกต้องจะถูกกำหนดด้วย

$$|R_n| = \left|\frac{k_n(W-u)^n}{n!}\right| \le \left|\frac{k_n|W-u|^n_{\max}}{n!}\right| \qquad (2.17)$$



รูปที่ 2.14 การประมาณค่าอนุกรมเทย์เลอร์

รูปที่ 2.14 แสดงตัวอย่างของการสังเคราะห์สัญญาณซายน์ด้วยการประมาณค่าอนุกรมเทย์เลอร์ตามสมการที่ (2.15) ซึ่งจะประกอบด้วยหน่วยความจำซายน์ (Sine ROM) และหน่วยความจำโคซายน์ (Cosine ROM) สำหรับเก็บค่าสองพจน์แรกในสมการ และยังมีหน่วยความจำสำหรับเก็บพจน์ที่สามของสมการอีกด้วย สัญญาณเอาต์พุตของหน่วยความจำทั้งสามจะถูกนำมาประมวลผล เพื่อให้ได้ผลลัพธ์เป็นแอมปลิจูดของสัญญาณซายน์ตามที่กำหนดไว้ในสมการที่ (2.15)

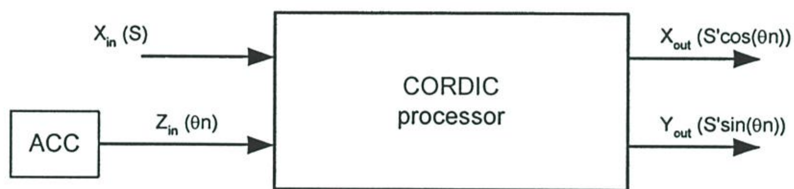
2.3 การสังเคราะห์ความถี่แบบดิจิตอลที่ไม่ใช้หน่วยความจำ

หัวข้อ 2.2 เป็นการศึกษารายละเอียดการทำงานของการสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ ซึ่งมีข้อเสียสำคัญคือคุณภาพของสัญญาณจะขึ้นกับความละเอียดของแอมปลิจูดและขนาดของหน่วยความจำ กล่าวคือถ้าต้องการสัญญาณที่มีคุณภาพสูงก็จำเป็นต้องใช้หน่วยความจำขนาดใหญ่เพื่อเก็บแอมปลิจูดของสัญญาณให้ได้มากและละเอียดที่สุด ซึ่งการใช้หน่วยความจำที่มีขนาดใหญ่จะมีผลให้ความเร็วในการเข้าถึงหน่วยความจำลดลง, วงจรที่ออกแบบมีขนาดใหญ่ขึ้น และการกินกำลังงานของวงจรเพิ่มขึ้น [6]

เพื่อเป็นการหลีกเลี่ยงข้อเสียของการสังเคราะห์ความถี่แบบดิจิตอลโดยใช้หน่วยความจำ ในหัวข้อนี้จึงจะอธิบายหลักการทำงานของวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ไม่ต้องใช้หน่วยความจำ ซึ่งสามารถสรุปเป็นวิธีต่างๆ ได้ดังนี้

2.3.1 ขั้นตอนวิธีคอร์ดิก (CORDIC algorithm)

ขั้นตอนวิธีคอร์ดิก (COordinate Rotation Digital Computer, CORDIC) [14] เป็นขั้นตอนวิธีที่ใช้ในการคำนวณฟังก์ชั่นตรีโกณมิติ (Trigonometric function) และเหมาะต่อการนำไปออกแบบเป็นฮาร์ดแวร์เนื่องจากการคำนวณที่ใช้ในขั้นตอนวิธีคอร์ดิกนั้นสามารถทำได้ด้วยวิธีการเลื่อนและบวก (Shift-and-add operation)



รูปที่ 2.15 การสร้างสัญญาณซายน์ด้วยขั้นตอนวิธีคอร์ดิก

รูปที่ 2.15 แสดงโครงสร้างของการสร้างสัญญาณซายน์โดยการใช้ขั้นตอนวิธีคอร์ดิก ซึ่งประกอบด้วยสัญญาณอินพุต X_{in}, Y_{in}, Z_{in} และสัญญาณเอาต์พุต X_{out}, Y_{out} โดยสัญญาณ Z_{in} เป็นสัญญาณเฟสที่ถูกสร้างขึ้นโดยวงจรสร้างเฟส เพื่อนำมาสังเคราะห์เป็นสัญญาณเอาต์พุตซายน์ (Sine) และโคซายน์ (Cosine) X_{out} และ Y_{out} และสัญญาณ X_{in} และ Y_{in} เป็นค่าข้อมูลแอมปลิจูด [15] เราสามารถเขียนความสัมพันธ์ระหว่างสัญญาณเอาต์พุตและเอาต์พุตของขั้นตอนวิธีคอร์ดิกได้ดังนี้

$$X_{out} = k[X_{in}\cos(Z_{in}) - Y_{in}\sin(Z_{in})] \tag{2.18}$$

$$Y_{out} = k[Y_{in}\cos(Z_{in}) + X_{in}\sin(Z_{in})] \quad (2.19)$$

ใน [15] Gielis และคณะได้ใช้คอร์ดิกโปรเซสเซอร์ (CORDIC processor) ทำหน้าที่แปลงค่าที่อยู่ในรูปโพลาร์ (Polar form) ให้เป็นค่าที่อยู่ในรูปคาร์ทีเซียน (Cartesian form) ถ้ากำหนดให้สัญญาณอินพุต Yin เท่ากับ 0 จะสามารถเขียนสมการความสัมพันธ์ระหว่างอินพุตกับเอาต์พุตได้ดังนี้

$$X_{out} = k[X_{in}\cos(Z_{in})] \quad (2.20)$$

$$Y_{out} = k[X_{in}\sin(Z_{in})] \quad (2.21)$$

ขนาดของแอมปลิจูดและความถี่ของสัญญาณที่สังเคราะห์สามารถควบคุมได้โดยสัญญาณ X_{in} และ Z_{in} ตามลำดับ

วงจรใน [15] ได้ถูกนำไปเจือสารด้วยเทคโนโลยีไบโพลาร์ขนาด 1 ไมครอน 13 กิกะเฮิรตซ์ โดยสัญญาณเฟสอินพุตมีขนาด 12 บิต และสัญญาณเอาต์พุตมีขนาด 10 บิต วงจรที่ออกแบบเสร็จแล้วสามารถทำงานได้ที่ความถี่สัญญาณนาฬิกาสูงสุด 540 เมกะเฮิรตซ์ และมีค่าอัตราส่วนของสัญญาณต่อสิ่งรบกวน (Signal-to-noise ratio) –60 เดซิเบล ซึ่งการหาค่าอัตราส่วนของสัญญาณต่อสิ่งรบกวนนี้หาจากวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 8 บิต เนื่องจากในสมัยนั้นยังไม่สามารถหาวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิต ได้

2.3.2 การคูณเชิงซ้อน (Complex multiplication)

การสังเคราะห์สัญญาณซายน์หรือโคซายน์ด้วยการคูณเชิงซ้อน [16] มีหลักการคือกำหนดให้มีจำนวนเชิงซ้อน (Complex number) สองจำนวนคือ a และ b มีค่าดังสมการ

$$a = r_\alpha(\cos\alpha + j\sin\alpha) \quad (2.22)$$

$$b = r_\beta(\cos\beta + j\sin\beta) \quad (2.23)$$

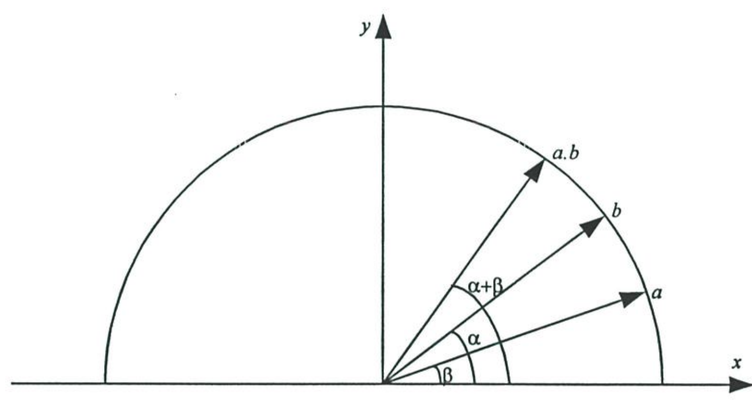
โดย r_α และ r_β เป็นความยาวของเวกเตอร์ และ α และ β เป็นมุมของเวกเตอร์ จากสมการที่ (2.22) และ (2.23) เราสามารถเขียนผลคูณ (Product) ของ a และ b เป็นสมการได้ดังนี้

$$a \bullet b = r_\alpha r_\beta[(\cos\alpha\cos\beta - \sin\alpha\sin\beta) + j(\cos\alpha\sin\beta + \cos\beta\sin\alpha)] \quad (2.24)$$

และเมื่อเปรียบเทียบความสัมพันธ์ของฟังก์ชั่นทางตรีโกณในสมการที่ (2.25) และ (2.26) กับสมการที่ (2.24) จะเห็นได้ว่าการคูณจำนวนเชิงซ้อนสองจำนวนเข้าด้วยกันจะทำให้เกิดจำนวนเชิงซ้อนใหม่ที่มีค่าของมุม (Angle) เท่ากับผลรวมของมุมของตัวคูณทั้งสอง และมีรัศมี (Radius) เท่ากับผลคูณของ r_α และ r_β ดังรูปที่ 2.16

$$\sin(\omega + \upsilon) = \cos\omega\sin\upsilon + \cos\upsilon\sin\omega \quad (2.25)$$

$$\cos(\omega + \upsilon) = \cos\omega\cos\upsilon - \sin\omega\sin\upsilon \quad (2.26)$$



รูปที่ 2.16 การหมุนเวกเตอร์ของการคูณเชิงซ้อน เมื่อ $r_\alpha = r_\beta$

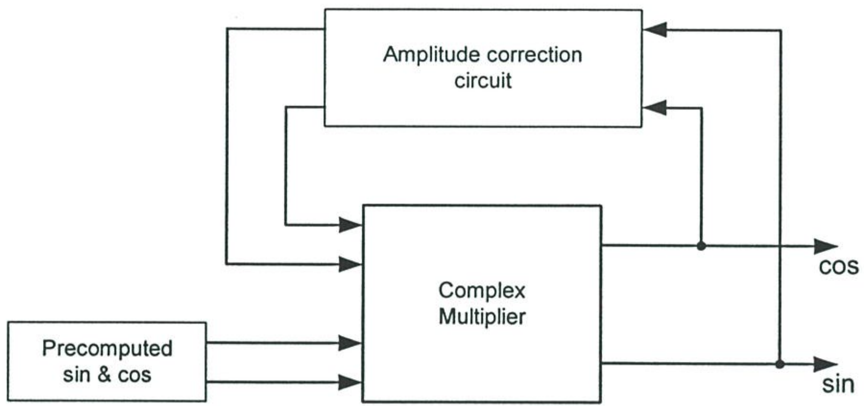
ถ้ากำหนดให้ตัวคูณตัวตั้ง (Multiplier) ในสมการที่ (2.24) คงที่และให้ตัวคูณตัวที่สอง (Multiplicand) เป็นค่าของผลลัพธ์ของการคูณครั้งก่อนหน้า จะได้ว่าเวกเตอร์ที่เกิดจากการคูณจะหมุนรอบวงกลมหนึ่งหน่วยด้วยความเร็วมุมคงที่ นั่นก็คือการสร้างสัญญาณซายน์และโคซายน์นั่นเอง ค่าความถี่ของสัญญาณเอาต์พุตจะถูกกำหนดโดยการเพิ่มของมุมดังสมการ

$$f_{out} = \frac{\beta}{2\pi} \bullet f_{clk} \quad (2.27)$$

โดย β คือมุมของเวกเตอร์ตัวคูณ และ f_{clk} คือความถี่ของสัญญาณนาฬิกาของระบบ

จากการที่ระบบเป็นระบบแบบย้อนกลับ (Feedback) จะทำให้เกิดการสะสมค่าความผิดพลาดในกรณีที่แอมปลิจูดของสัญญาณซายน์หรือโคซายน์ที่เกิดจากผลคูณเวก

เตอร์มีการตัดทอน ซึ่งจะสะสมเพิ่มขึ้นเรื่อยๆ จึงจำเป็นต้องมีการเพิ่มวงจรชดเชยแอมปลิจูด [17] สำหรับแก้ไขแอมปลิจูดให้ถูกต้องก่อนที่จะย้อนกลับไปคูณใหม่ ดังแสดงในรูปที่ 2.17



รูปที่ 2.17 การสังเคราะห์สัญญาณซายน์ด้วยการคูณเชิงซ้อน

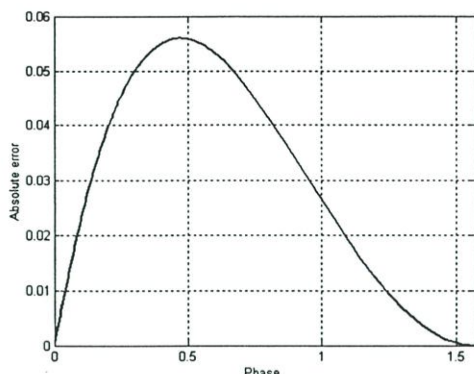
Kalle Palomaki และคณะ ได้ทำการออกแบบวงจรสังเคราะห์สัญญาณซายน์และโคซายน์ด้วยการคูณเชิงซ้อน [16] และเจือสารโดยใช้เทคโนโลยีซีมอส 0.35 ไมครอน วงจรรวมที่เจือสารเสร็จแล้วสามารถทำงานได้ที่สัญญาณนาฬิกาสูงสุด 100 เมกะเฮิรตซ์ และมีค่า SFDR (Spurious Free Dynamic Range) -82 เดซิเบล

2.3.3 **การประมาณค่าพาราโบลา** (Parabolic approximation)

การสังเคราะห์สัญญาณซายน์ด้วยวิธีประมาณค่าพาราโบลา [18] ใช้สมการพาราโบลาอันดับที่ 1 ประมาณค่าแอมปลิจูดของสัญญาณซายน์ดังนี้

$$pb1(\phi) = 4(\phi/\pi)(1-\phi/\pi), \qquad 0<\phi<\pi \qquad (2.28)$$

ซึ่งค่าความผิดพลาดของการประมาณค่าสัญญาณซายน์ด้วยสมการที่ (2.28) ในช่วง 0 ถึง $\pi/2$ นั้นสามารถพล็อตเป็นกราฟได้ดังรูปที่ 2.18



รูปที่ 2.18 ค่าความผิดพลาดของการประมาณสัญญาณซายน์ด้วยสมการพาราโบลาอันดับที่ 1

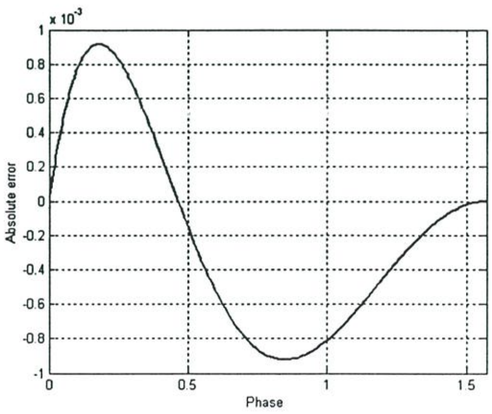
จากรูปที่ 2.18 จะเห็นว่ากราฟมีลักษณะเป็นพาราโบลาลักษณะเดียวกันกับ $pb1(\phi)$ ในสมการที่ (2.28) แต่จะต่างกันที่ขนาด ซึ่งสามารถแทนค่าได้ด้วยสมการ

$$er2(\phi) = 4K \bullet pb1(\phi) \bullet (1 - pb1(\phi)), \qquad 0 < \phi < \pi \qquad (2.29)$$

โดย K เป็นค่าคงที่สำหรับปรับขนาด มีค่าประมาณ 0.056 ดังนั้นเราสามารถเขียนสมการพาราโบลาอันดับที่ 2 ได้ดังนี้

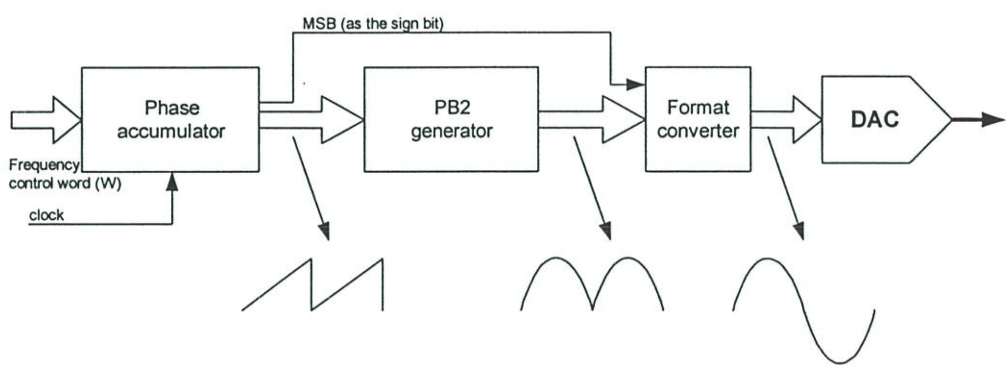
$$pb2(\phi) = pb1(\phi) - er2(\phi), \qquad 0 < \phi < \pi \qquad (2.30)$$

ค่าความผิดพลาดของการสร้างสัญญาณซายน์ด้วยสมการที่ (2.30) $(\sin(\phi) - pb2(\phi))$ สามารถพล็อตเป็นกราฟได้ดังรูปที่ 2.19



รูปที่ 2.19 ค่าความผิดพลาดของการประมาณสัญญาณซายน์ด้วยสมการพาราโบลาอันดับที่ 2

Amir M. Sodagar และ G. Roientan Lahiji ได้นำเสนอและออกแบบวงจรสังเคราะห์สัญญาณซายน์ด้วยสมการพาราโบลาอันดับที่ 2 ใน [18] ซึ่งมีโครงสร้างการทำงานดังรูปที่ 2.20 และได้นำวงจรที่ออกแบบไปจำลองการทำงานโดยใช้เทคโนโลยีซีมอสขนาด 0.6 ไมครอน และกำหนดให้สัญญาณเฟสมีขนาด 12 บิต และสัญญาณซายน์เอาต์พุตมีขนาด 10 บิต ซึ่งจากผลการจำลองการทำงานนั้นวงจรสามารถทำงานที่ความเร็วสัญญาณนาฬิกาสูงสุด 175 เมกะเฮิรตซ์ และมีค่า SFDR –64 เดซิเบล



รูปที่ 2.20 โครงสร้างของวงจรสังเคราะห์สัญญาณซายน์ด้วยสมการพาราโบลาอันดับที่ 2

2.4 สรุปเปรียบเทียบ

หลังจากที่ได้ศึกษาคุณสมบัติและหลักการทำงานของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลทั้งแบบที่ใช้หน่วยความจำและไม่ใช้หน่วยความจำ เราสามารถสรุปคุณสมบัติข้อแตกต่างระหว่างวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลทั้งสองประเภทได้ดังตารางที่ 2.3 และคุณสมบัติและประสิทธิภาพของวงจรสังเคราะห์ความถี่ที่ได้มีการวิจัยมาทั้งแบบใช้หน่วยความจำและไม่ใช้หน่วยความจำได้ดังตารางที่ 2.4 โดยที่สัญญาณเอาต์พุตของวิธีคูณเชิงซ้อนและวิธีประมาณค่าพาราโบลาอันดับสองมีขนาด 10 บิต ในขณะที่สัญญาณเอาต์พุตในวิธีอื่นนั้นมีขนาด 12 บิต

ตารางที่ 2.3 สรุปคุณสมบัติของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลแบบที่ใช้หน่วยความจำและไม่ใช้หน่วยความจำ

แบบใช้หน่วยความจำ	แบบไม่ใช้หน่วยความจำ
1. ออกแบบได้ง่าย	1. ใช้วงจรคณิตศาสตร์คำนวณรูปสัญญาณ
2. เหมาะกับการสังเคราะห์สัญญาณรูปแบบที่ไม่ใช่สัญญาณมาตรฐาน โดยการเปลี่ยนค่าแอมปลิจูดในหน่วยความจำให้เป็นตามสัญญาณที่ต้องการสังเคราะห์	2. การสังเคราะห์รูปสัญญาณที่ไม่เป็นมาตรฐานทำได้ยาก เพราะต้องสร้างวงจรสำหรับสร้างรูปสัญญาณนั้นๆ ขึ้นต่างหาก
3. ต้องใช้หน่วยความจำขนาดใหญ่สำหรับการสังเคราะห์สัญญาณให้มีคุณภาพสูง ทำให้ขนาดวงจรใหญ่ขึ้น, กินไฟสูง และความเร็วลดลง	3. การสังเคราะห์สัญญาณคุณภาพสูงสามารถทำได้โดยการสร้างวงจรเฉพาะสำหรับสังเคราะห์สัญญาณนั้นขึ้นมา ไม่ขึ้นกับขนาดของหน่วยความจำ

ตารางที่ 2.4 คุณสมบัติและประสิทธิภาพของวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ได้มีการวิจัยมา

วิธีการ	ขนาดหน่วยความจำ	อัตราการบีบอัดหน่วยความจำ	เอสเอฟดีอาร์ (SFDR) สูงสุด	วงจรที่ต้องใช้
ไม่บีบอัดหน่วยความจำ [6]	2^{14}x12 บิต	1:1	-97.23 dBc	-
โมดิฟายซันเดอร์แลนด์ [6]	2^{8}x9 บิต 2^{8}x4 บิต	59:1	-86.91 dBc	วงจรบวก
อนุกรมเทย์เลอร์สามพจน์ [6]	2^{7}x14 บิต 2^{7}x9 บิต	64:1	-97.04 dBc	วงจรบวก, วงจรคูณ
คอร์ดิก [6]	-	-	-84.25 dBc	ไปป์ไลน์ 14 สถานะ
การคูณเชิงซ้อน [16]	-	-	-69 dBc	วงจรบวก, วงจรเลื่อน, วงจรมัลติเพลกซ์, วงจรคอมพลีเมนต์
การประมาณค่าพาราโบลาอันดับสอง [18]	-	-	-64 dBc	วงจรบวก, วงจรลบ, วงจรคูณ, วงจรคอมพลีเมนต์

บทที่ 3

วงจรรวมเพื่อสังเคราะห์ความถี่ แบบดิจิตอลที่ใช้หน่วยความจำ

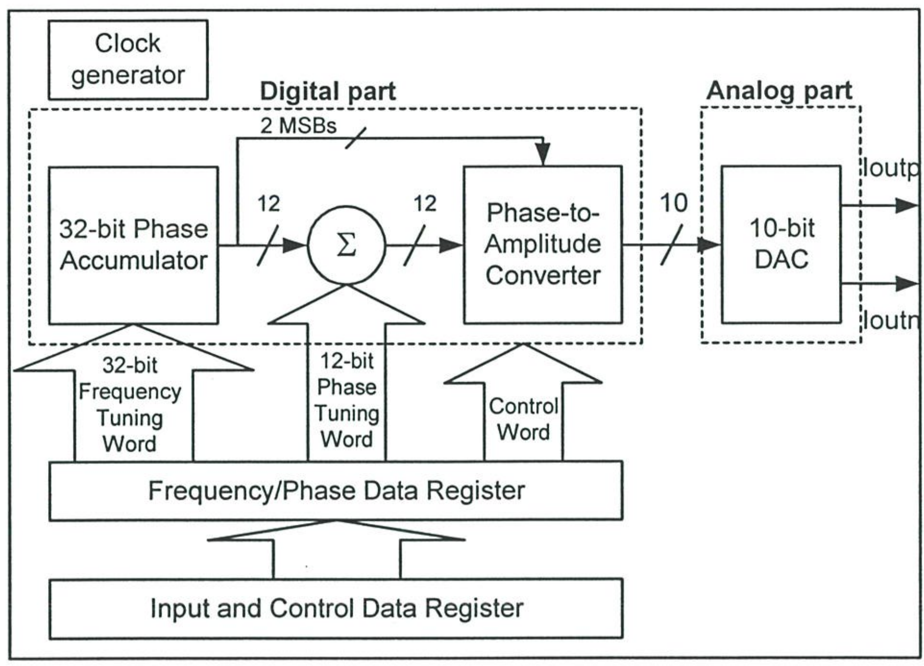
เนื้อหาในบทนี้จะเกี่ยวกับรายละเอียดการออกแบบและทดสอบชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำที่ได้ทำการออกแบบไว้ [19-20] โดยจะเริ่มจากการอธิบายถึงข้อกำหนดของชิปและการนำไปประยุกต์ใช้งาน ตามด้วยรายละเอียดของการออกแบบวงจรในแต่ละส่วนรวมไปถึงการออกแบบเลย์เอาต์ จากนั้นจะเป็นผลการทดสอบการทำงานของชิป และปิดท้ายด้วยบทสรุป

3.1 ข้อกำหนดการออกแบบสำหรับการประยุกต์ใช้งาน

ชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำถูกออกแบบและพัฒนาโดยมีจุดประสงค์เพื่อสร้างชิปสำหรับสังเคราะห์สัญญาณรูปซายน์ (Sine), รูปแรมป์ (Ramp), รูปฟันเลื่อย (Saw-tooth) และสัญญาณสุ่ม (Ramdom) เพื่อนำไปประยุกต์ใช้ในงานประเภทวิดีโอและการสื่อสารไร้สาย เช่น วิทยุ/โมเด็มแบบดิจิตอล เครื่องมือวัดแบบมือถือ (Handheld) ระบบวีดีโอ/ออดิโอแบบดิจิตอล เครื่องรับ/ส่งสัญญาณในย่านเบสแบนด์ รวมไปถึงการนำไปพัฒนาเป็นอุปกรณ์กำเนิดฟังก์ชั่น (Function generator) สำหรับใช้ในห้องปฏิบัติการตามสถานศึกษาและห้องปฏิบัติการวิจัยต่างๆ

เทคโนโลยีที่ใช้ในการเจือสารชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำนี้คือเทคโนโลยีซีมอส 0.5 ไมครอน แบบบ่อเอ็น (N-well) โพลีเดี่ยว (Single-poly) สามเมทัล (Triple-metal) ของบริษัทอัลคาเทล (Alcatel) ชิปที่ออกแบบสามารถสังเคราะห์สัญญาณได้ 4 รูปสัญญาณ และทำฟังก์ชั่นมอดูเลชั่นทางเฟสและความถี่ได้ สามารถทำงานได้ที่ความถี่สัญญาณนาฬิกาสูงสุด 100 เมกะเฮิรตซ์ โดยมีค่าควบคุมความถี่ (Frequency control word) ขนาด 32 บิต และมีความละเอียดในการปรับความถี่เท่ากับ 0.0233 เฮิรตซ์ ที่สัญญาณนาฬิกา 100 เมกะเฮิรตซ์

3.2 รายละเอียดการออกแบบวงจร



รูปที่ 3.1 สถาปัตยกรรมของชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

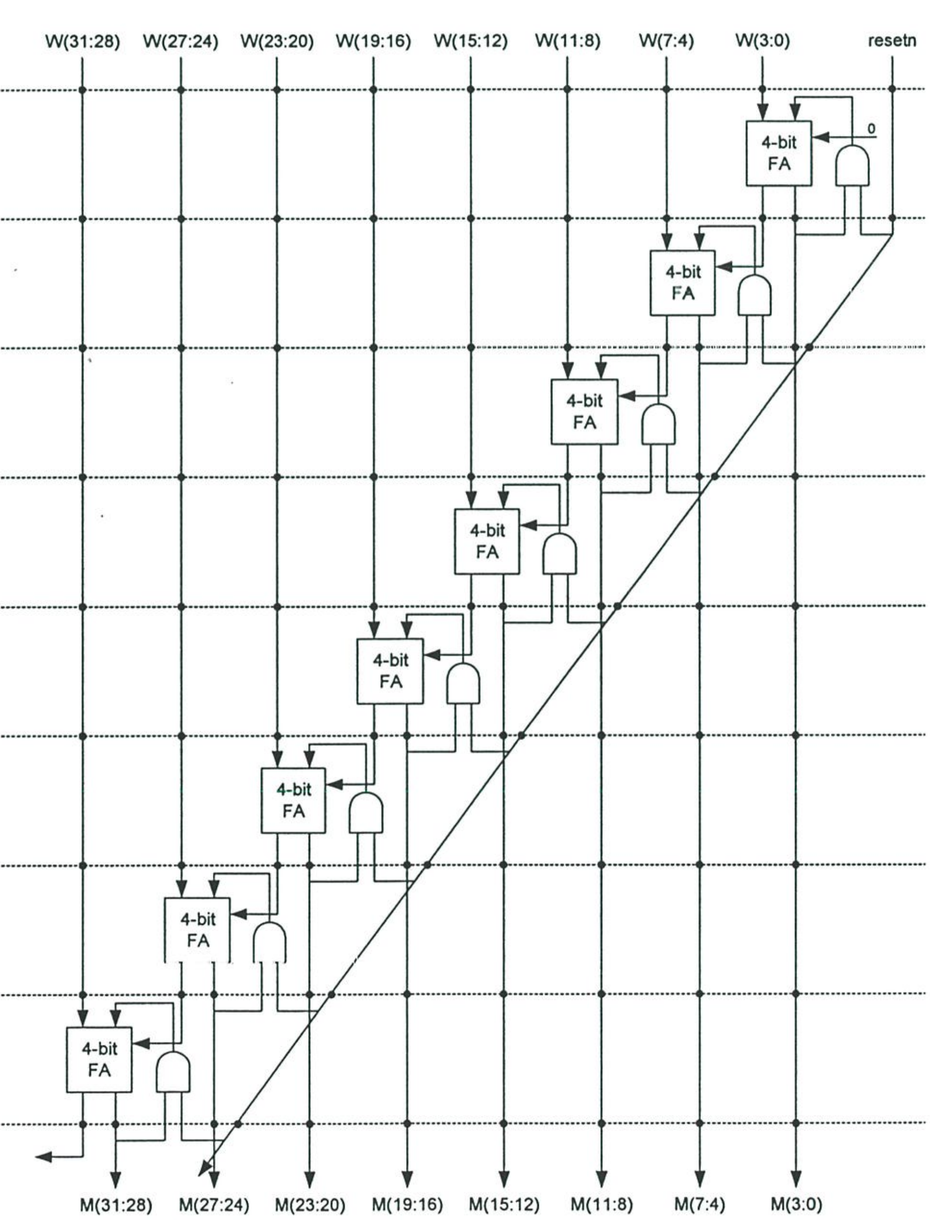
วงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำมีสถาปัตยกรรมตามรูปที่ 3.1 โดยจะแบ่งออกเป็นสองส่วนหลักๆ คือส่วนดิจิตอล (Digital part) และส่วนอนาลอก (Analog part) โดยส่วนดิจิตอลจะประกอบด้วยวงจรสร้างเฟสขนาด 32 บิต และวงจรเปลี่ยนเฟสเป็นแอมปลิจูด และส่วนประกอบหลักของส่วนอนาลอกคือวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิต เราสามารถสรุปรายละเอียดการออกแบบวงจรแยกตามส่วนประกอบหลักๆ ได้ดังนี้

3.2.1 **ส่วนดิจิตอล** (Digital part)

วงจรในส่วนดิจิตอลได้ถูกออกแบบด้วยวิธีออกแบบจากบนลงล่าง (Top-down design) ซึ่งมีขั้นตอนคือเริ่มจากการกำหนดข้อกำหนด (Specification) ของวงจร และเมื่อได้ข้อกำหนดของวงจรแล้วก็จะทำการเขียนโค้ดคำสั่งบรรยายพฤติกรรมการทำงานของวงจรด้วยภาษาวีเอชดีแอล เสร็จแล้วนำไปสังเคราะห์ (Synthesis) และจัดวางและเชื่อมโยง (Place-and-route) ด้วยเทคโนโลยีซีมอสขนาด 0.5 ไมครอน ของบริษัทอัลคาเทล

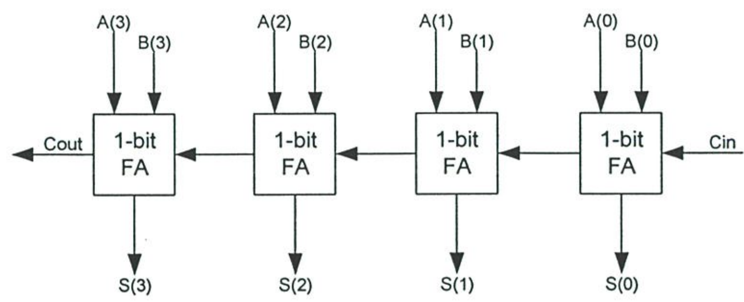
3.2.1.1 วงจรสร้างเฟส 32 บิต (32-bit phase accumulator)

วงจรสร้างเฟสขนาด 32 บิต มีโครงสร้างดังรูปที่ 3.2 ประกอบด้วยวงจรบวกขนาด 4 บิต (4-bit full adder) จำนวน 8 ชุด มาต่อกันในลักษณะสายท่อ (Pipeline) เพื่อเพิ่มความเร็วในการทำงานของวงจร นอกจากนี้ยังมีการนำสัญญาณรีเซตแบบสายท่อมาใช้เพื่อเคลียร์ค่าของรีจิสเตอร์เฟส [3] โดยที่จุดดำๆ ในรูปที่ 3.2 เป็นรีจิสเตอร์สำหรับปรับเวลา (Pre-skewing and De-skewing registers)



รูปที่ 3.2 โครงสร้างของวงจรสร้างเฟสขนาด 32 บิต

วงจรบวกขนาด 4 บิตที่นำมาใช้ออกแบบเป็นวงจรสร้างเฟส 32 บิต มีโครงสร้างเป็นวงจรบวกแบบริปเปิลแคร์รี่ [2] (Ripple-carry adder) ดังแสดงในรูปที่ 3.3



รูปที่ 3.3 โครงสร้างของวงจรบวก 4 บิต แบบริปเปิลแคร์รี่

3.2.1.2 วงจรเปลี่ยนเฟสเป็นแอมปลิจูด (Phase-to-amplitude converter)

วงจรเปลี่ยนเฟสเป็นแอมปลิจูดสามารถสังเคราะห์รูปสัญญาณได้ทั้งหมด 4 รูปสัญญาณ ได้แก่ รูปซายน์, รูปแรมป์, รูปฟันเลื่อย และสัญญาณสุ่ม วิธีการบีบอัดควอแดรนท์ [21] ได้ถูกนำมาใช้ในการสังเคราะห์สัญญาณทุกรูปสัญญาณ นอกจากวิธีการบีบอัดควอแดรนท์แล้วเราสามารถอธิบายขั้นตอนการสังเคราะห์สัญญาณแต่ละรูปสัญญาณได้ดังนี้

- รูปซายน์ (Sine)

ใช้หลักการของการเก็บรูปสัญญาณไว้ในหน่วยความจำแบบอ่านอย่างเดียว (ROM) โดยบีบอัดข้อมูลเพื่อลดขนาดของหน่วยความจำด้วยขั้นตอนวิธีโมดิฟายซันเดอร์แลนด์ [6] (Modified Sunderland algorithm)

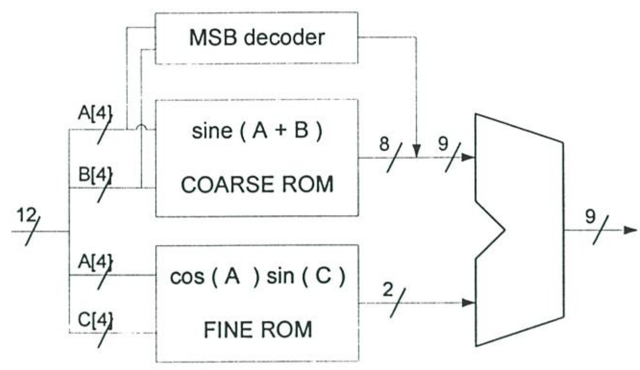
ในขั้นตอนวิธีซันเดอร์แลนด์ค่าของเฟสขนาด 12 บิต (ϕ) ที่ถูกสร้างจากวงจรสร้างเฟสจะถูกแบ่งออกเป็น 3 ส่วน เท่าๆ กัน ส่วนละ 4 บิต (*a, b, c*) ดังสมการที่ 3.1

$$\phi = a + b + c \tag{3.1}$$

และจากความสัมพันธ์ของฟังก์ชั่น $\sin(\phi) = \sin(a+b+c)$ เราสามารถประมาณค่าของฟังก์ชั่น $\sin(\phi)$ ได้ดังสมการที่ (3.2) และ (3.3) ตามลำดับ

$$\sin(a+b+c) = \sin(a+b)\cos c + \cos a \cos b \sin c - \sin a \sin b \sin c \tag{3.2}$$

$$\sin(a+b+c) \approx \sin(a+b) + \cos a \sin c \tag{3.3}$$



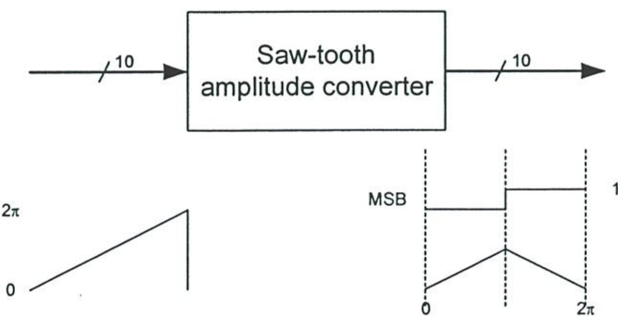
รูปที่ 3.4 ขั้นตอนวิธีโมดิฟายซันเดอร์แลนด์

สมการที่ (3.3) จะถูกใช้ในการออกแบบเป็นวงจรจริง โดย $\sin(a+b)$ จะถูกเก็บในหน่วยความจำหยาบ (Coarse ROM) และ $\cos a \sin c$ จะถูกเก็บในหน่วยความจำละเอียด (Fine ROM) สัญญาณเอาต์พุตของทั้งหน่วยความจำหยาบและหน่วยความจำละเอียดจะถูกนำมารวมกันเพื่อสร้างเป็นสัญญาณซายน์ดังแสดงในรูปที่ 3.4 โดยที่สัญญาณเอาต์พุตของวงจรเปลี่ยนเฟสเป็นแอมปลิจูดด้วยขั้นตอนวิธีซันเดอร์แลนด์จะมีขนาด 9 บิต ซึ่งจะถูกนำไปดีโคดควอแดรนท์และสร้างเป็นสัญญาณแอมปลิจูดเอาต์พุตของสัญญาณซายน์ขนาด 10 บิต ต่อไป

- รูปแรมป์ (Ramp)

สัญญาณรูปแรมป์นั้นสามารถสังเคราะห์ได้โดยตรงจากสัญญาณเอาต์พุตของวงจรสร้างเฟส โดยจะดึงเฉพาะส่วนบิตเอ็มเอสบี 10 บิต แรกมาเป็นสัญญาณเอาต์พุตของวงจรเปลี่ยนเฟสเป็นแอมปลิจูดเพื่อป้อนให้กับวงจรแปลงดิจิตอลเป็นอนาลอกต่อไป

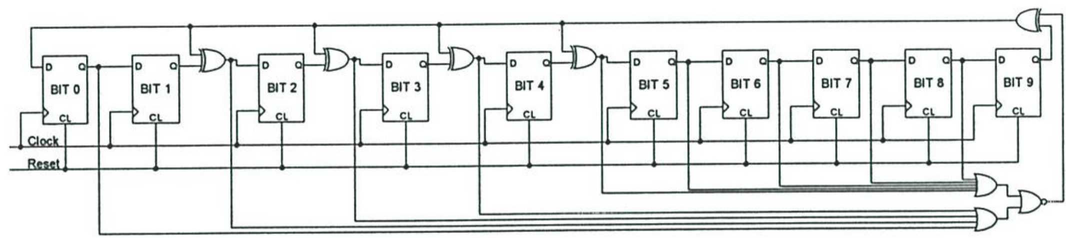
- รูปฟันเลื่อย (Saw-tooth)



รูปที่ 3.5 การสังเคราะห์สัญญาณรูปฟันเลื่อย

สัญญาณรูปฟันเลื่อยสามารถสังเคราะห์ได้จากบิตเอ็มเอสบี 10 บิต แรกที่ได้จากเอาต์พุตของวงจรสร้างเฟส โดยถ้าบิตเอ็มเอสบีบนสุดมีค่าเท่ากับ 0 สัญญาณ 10 บิต จากวงจรสร้างเฟสจะถูกส่งต่อโดยตรงไปยังวงจรแปลงดิจิตอลเป็นอนาลอก แต่ถ้าบิตเอ็มเอสบีบนสุดมีค่าเท่ากับ 1 สัญญาณ 10 บิตจากวงจรสร้างเฟสจะถูกคอมพลีเมนต์ก่อนที่จะส่งต่อไปยังวงจรแปลงดิจิตอลเป็นอนาลอก

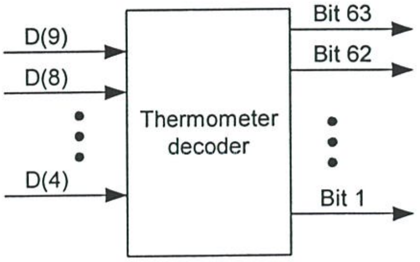
- สัญญาณสุ่ม (Random)



รูปที่ 3.6 รีจิสเตอร์เลื่อนย้อนกลับแบบเชิงเส้น

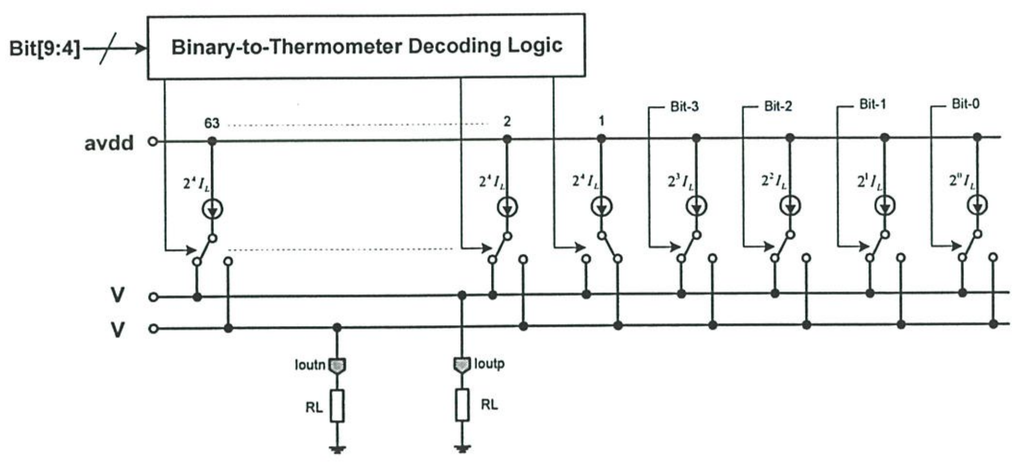
สัญญาณสุ่มถูกสังเคราะห์โดยใช้รีจิสเตอร์เลื่อนย้อนกลับแบบเชิงเส้น [22] (Linear Feedback Shift Register, LFSR) ขนาด 10 บิต ดังรูปที่ 3.6 ทำหน้าที่สังเคราะห์สัญญาณสุ่มเทียม (Pseudo-random) เพื่อป้อนให้กับวงจรแปลงดิจิตอลเป็นอนาลอก

นอกจากนี้ภายในวงจรเปลี่ยนเฟสเป็นแอมปลิจูดยังประกอบด้วยวงจรถอดรหัสเทอร์โมมิเตอร์ [8] สำหรับถอดรหัสข้อมูลบิตเอ็มเอสบี 6 บิต แรกของแอมปลิจูดของสัญญาณให้เป็นสัญญาณขนาด 63 บิต เพื่อควบคุมแหล่งจ่ายกระแสที่มีน้ำหนักกระแส 16 เท่าของแหล่งจ่ายกระแสที่มีน้ำหนักน้อยที่สุด ส่วนข้อมูล 4 บิต ที่เหลือจะถูกส่งต่อไปยังวงจรแปลงดิจิตอลเป็นอนาลอกโดยตรง โครงสร้างของวงจรถอดรหัสเทอร์โมมิเตอร์แสดงได้ดังรูปที่ 3.7



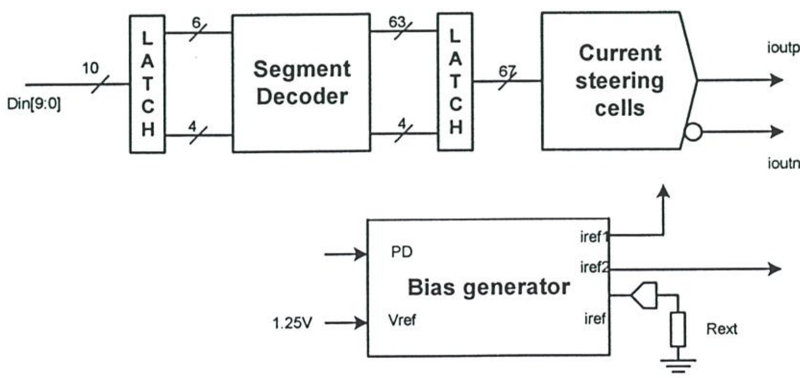
รูปที่ 3.7 วงจรถอดรหัสเทอร์โมมิเตอร์

3.2.2 **ส่วนอนาลอก** (Analog part)



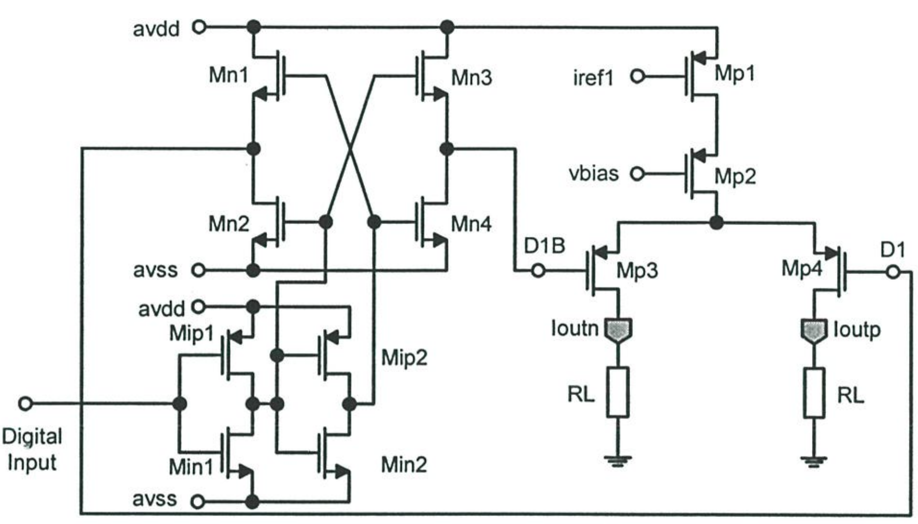
รูปที่ 3.8 สถาปัตยกรรมของวงจรแปลงดิจิตอลเป็นอนาลอก 10 บิต ในโหมดกระแสแบบแยกส่วน 6/4

วงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิต มีโครงสร้างการทำงานในโหมดกระแสแบบแยกส่วน 6/4 ดังแสดงในรูปที่ 3.8 โดยที่ ข้อมูลเอ็มเอสบี 6 บิตแรกถูกนำมาเข้ารหัสแบบเทอร์โมมิเตอร์ (Thermometer-coded) เพื่อควบคุมแหล่งจ่ายกระแสที่เหมือนกัน 63 ชุด โดยปริมาณกระแสแต่ละชุดมีค่าเท่ากับ 16 เท่าของขนาดกระแสสำหรับข้อมูลบิตที่สำคัญน้อยที่สุด (LSB) สำหรับข้อมูล 4 บิตที่สำคัญน้อยที่สุด ถูกนำมาควบคุมแหล่งจ่ายกระแส 4 ชุด ที่มีขนาดกระแสเท่ากับน้ำหนักไบนารี (Binary weighted) ของข้อมูลกระแสที่ไหลจากแหล่งจ่ายกระแสแต่ละตัว ซึ่งถูกควบคุมให้ 'ปิด' หรือ 'เปิด' ตามรหัสของสัญญาณดิจิตอลอินพุต กระแสเอาต์พุตทั้งหมดจะถูกนำมารวมกันและผ่านเข้าตัวต้านทานโหลดเพื่อสร้างแรงดันอนาลอกเอาต์พุต



รูปที่ 3.9 โครงสร้างของวงจรแปลงดิจิตอลเป็นอนาลอก

รูปที่ 3.9 แสดงโครงสร้างของวงจรแปลงดิจิตอลเป็นอนาลอก ซึ่งประกอบด้วย วงจรขับกระแส (Current-steering), วงจรกำเนิดไบแอส (Bias generator), วงจรเข้ารหัส (Segment decoder), และวงจรค้างสัญญาณ (Code latches)

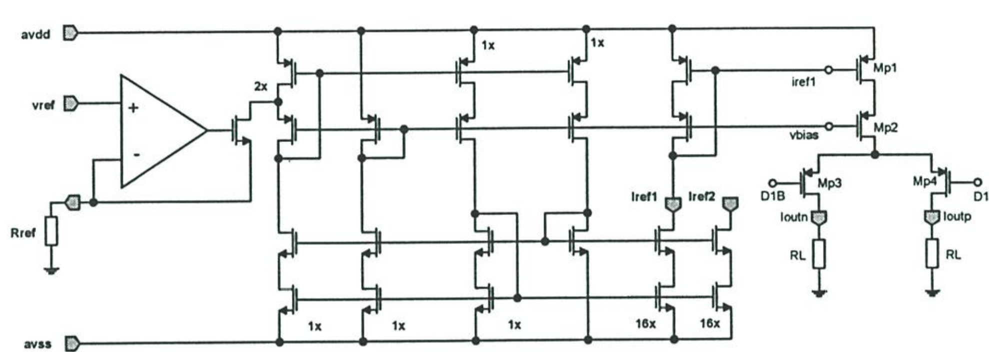


รูปที่ 3.10 วงจรขับกระแส

วงจรขับกระแสของวงจรแปลงดิจิตอลเป็นอนาลอก ในรูปที่ 3.10 ประกอบด้วย แหล่งจ่ายกระแสแบบคาสโคด (M_{p1} และ M_{p2}) และสวิตซ์กระแส (M_{p3} และ M_{p4}) ที่สร้างขึ้นจากมอสเฟทชนิดพี (p-channel MOSFET) ซึ่งทำให้ได้แรงดันเอาต์พุตที่มีค่าบวก แหล่งจ่ายกระแสแบบคาสโคดลดการเปลี่ยนแปลงค่าแรงดันที่ขาเดรนของมอสเฟท M_{p1} ทำให้แหล่งจ่ายกระแสมีความต้านทานสูง มอสเฟท M_{p1} มีความยาวมาก เพื่อให้กระแสจ่ายออกมาจากวงจรแต่ละชุดมีความเที่ยงตรงและแม่นยำสูง ในขณะที่ M_{p2} มีความยาวน้อย เพื่อลดตัวเก็บประจุแฝงที่ขาซอสของ M_{p3} และ M_{p4} ทำให้สามารถสวิตซ์กระแสได้เร็วขึ้น กระแสจากแหล่งจ่ายกระแสแบบคาสโคดถูกสวิตซ์ให้ไหลผ่านตัวต้านทานโหลดตามรหัสของสัญญาณดิจิตอลอินพุต D1 และ D1B สวิตซ์กระแส M_{p3} และ M_{p4} ใช้ความยาวน้อยที่สุดและความกว้างที่เหมาะสม เพื่อเพิ่มความเร็วในการสวิตซ์กระแส และรักษาความเที่ยงตรงและแม่นยำของกระแส

สัญญาณดิจิตอล D1 และ D1B กำเนิดจากวงจรขับสวิตซ์แบบไม่สมมาตร [23] M_{n1}-M_{n4}, เพื่อป้องกันไม่ให้สวิตซ์กระแส M_{p3} และ M_{p4} ทำงานในขณะ 'ปิด' พร้อมกัน ซึ่งอาจเกิดขึ้นได้เนื่องจากช่วงเวลาขอบขาขึ้นและขอบขาลงของสัญญาณ D1 และ D1B ไม่คล้องจองกัน มอสเฟท M_{n1}-M_{n4} ต้องมีขนาดเหมาะสมที่ทำให้ช่วงเวลาขอบขาลงของ

สัญญาณ D1 และ D1B (ขณะ 'เปิด') มีค่าน้อยกว่าช่วงเวลาขอบขาขึ้น (ขณะ 'ปิด') เพื่อป้องกันไม่ให้สวิตช์กระแส M_{p3} และ M_{p4} ทำงานในขณะ 'ปิด' พร้อมกัน ซึ่งจะลดแรงดันกลิทช์ที่เอาต์พุตได้มาก โดยยอมให้สวิตช์กระแส M_{p3} และ M_{p4} จะทำงานในขณะ 'เปิด' พร้อมกันในช่วงเวลาสั้นๆ ซึ่งจะทำให้ความเร็วในการสวิตช์กระแสลดลงเล็กน้อย นอกจากนี้วงจรขับสวิตช์แบบไม่สมมาตรยังลดขนาดของสัญญาณรบกวนที่เอาต์พุต ซึ่งเกิดจากการสวิตช์สัญญาณ



รูปที่ 3.11 การทำงานของวงจรกำเนิดกระแส

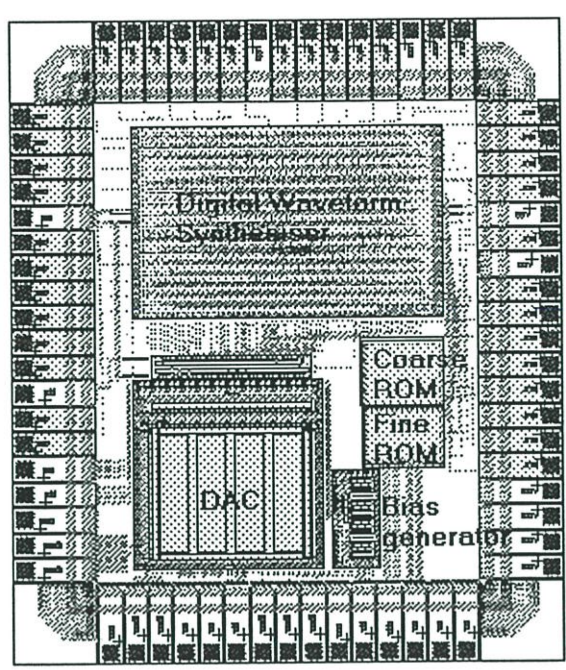
วงจรกำเนิดไบแอส ในรูปที่ 3.11 ทำหน้าที่ สร้างกระแสและแรงดันอ้างอิงให้กับของวงจรขับกระแส แรงดัน V_{ref} ถูกกำหนดด้วยวงจรแรงดันอ้างอิงชนิดแบนด์แกป (Bandgap) จากภายนอกชิป ขนาดของกระแสอ้างอิงที่สร้างขึ้นและจ่ายไปยังวงจรขับกระแส มีขนาดเป็นสี่เท่าของขนาดกระแสสำหรับข้อมูลบิตที่สำคัญน้อยที่สุด ซึ่งจะช่วยลดการคูณของสัญญาณรบกวนในวงจรสะท้อนกระแส ค่ากระแสเอาต์พุตสูงสุดของวงจรแปลงดิจิตอลเป็นอนาลอกซึ่งถูกกำหนดด้วยตัวต้านทานที่ต่อภายนอกชิป (R_{ref}) โดยมีค่าเท่ากับ $I_{out,FS}$ = $320/R_{ref}$ นอกจากนี้วงจรกำเนิดไบแอส ยังประกอบด้วยส่วนวงจรปิดการทำงาน (power-down) ที่สามารถควบคุมได้จากคำสั่งดิจิตอลภายนอก ในสภาวะการทำงานปกติ R_{ref} = 9.1 กิโลโอห์ม และ $I_{out,FS}$ = 35.16 มิลลิแอมป์

3.3 การออกแบบเลย์เอาต์

ประสิทธิภาพการทำงานของวงจรสังเคราะห์ความถี่จะขึ้นอยู่กับการทำงานของวงจรแปลงดิจิตอลเป็นอนาลอกเป็นส่วนใหญ่ ดังนั้นการออกแบบเลย์เอาต์สำหรับวงจรแปลงดิจิตอลเป็นอนาลอกจึงมีความสำคัญมาก ซึ่งต้องอาศัยฝีมือและประสบการณ์ ในขณะที่ส่วนของวงจรดิจิตอลนั้นจะใช้วิธีการทำเลย์เอาต์แบบออโตเมติก (Automatic place-and-route) โดยใช้เซลล์มาตรฐาน (Standard library cells) เลย์เอาต์ของวงจรอนาลอกกับดิจิตอล จะแยกให้อยู่คนละส่วน เพื่อป้องกันสัญญาณรบกวนที่เกิดจากการสวิตช์จากส่วนวงจรดิจิตอล เลย์เอาต์ของชิปที่สมบูรณ์มีขนาดประมาณ 12 ตารางมิลลิเมตร

เพื่อให้ได้ประสิทธิภาพการทำงานที่ดีที่สุดของวงจรแปลงดิจิตอลเป็นอนาลอก เลย์เอาต์ของส่วนวงจรขับกระแสที่ต่อแบบคาสโคดทั้งหมดถูกจัดแบบอาเรย์ โดยแยกจาก วงจรสวิตช์กระแสและวงจรขับสวิตช์ซึ่งถูกจัดวางให้อยู่รวมกันเป็นบล็อก ทำให้เลย์เอาต์ของส่วนวงจรขับกระแสที่ต่อแบบคาสโคดมีขนาดกะทัดรัด ช่วยลดความไม่สมดุลระหว่างวงจร การแยกส่วนเลย์เอาต์ของวงจรส่วนอนาลอกกับดิจิตอลออกจากกันเป็นสิ่งที่สำคัญและต้องคำนึงถึงอย่างมากในการออกแบบวงจรรวมแบบสัญญาณผสม (Mixed signal) ดังนั้นแหล่งจ่ายแรงดันไฟเลี้ยงของวงจรด้านดิจิตอลและอนาลอก ได้ถูกแยกออกจากกัน และได้มีส่วนของการป้องกันล้อมรอบ (Guard ring) แบบ 2 ชั้น ให้กับวงจรจ่ายกระแสและวงจรสวิตช์กระแส โดยจะช่วยป้องกันสัญญาณรบกวนที่เกิดจากการสวิตช์ของวงจรดิจิตอล ทำให้ส่วนของวงจรแปลงดิจิตอลเป็นอนาลอกมีเสถียรภาพมากขึ้น

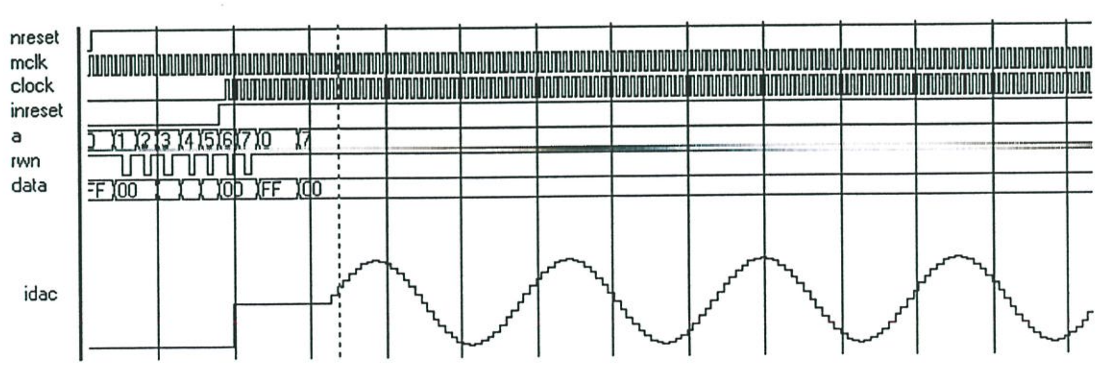
การเลย์เอาต์ของส่วนวงจรจ่ายกระแสจะแบ่งเป็น 4 ส่วนที่สมมาตรกัน โดยแต่ละส่วนจะประกอบด้วยแหล่งจ่ายกระแสจำนวน 63 ตัว ซึ่งใช้สำหรับข้อมูล 6 บิต ที่สำคัญมากที่สุด แหล่งจ่ายกระแสแต่ละตัวจ่ายกระแสมีขนาด 16 เท่าของขนาดกระแสสำหรับข้อมูลบิตที่สำคัญน้อยที่สุด (16-LSB) กระแสที่ได้จะเกิดจากปริมาณกระแสที่เท่ากันของทั้ง 4 ส่วนรวมกันได้ 16-LSB แต่ละส่วนจะจัดวางตำแหน่งของแหล่งจ่ายกระแสให้สมดุลกัน เพื่อให้สภาวะแวดล้อมของแหล่งจ่ายกระแสมีสภาวะคล้ายกัน เพื่อลดข้อผิดพลาดที่เกิดจากปริมาณกระแสไม่เท่ากัน เลย์เอาต์ของวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำแสดงในรูปที่ 3.12



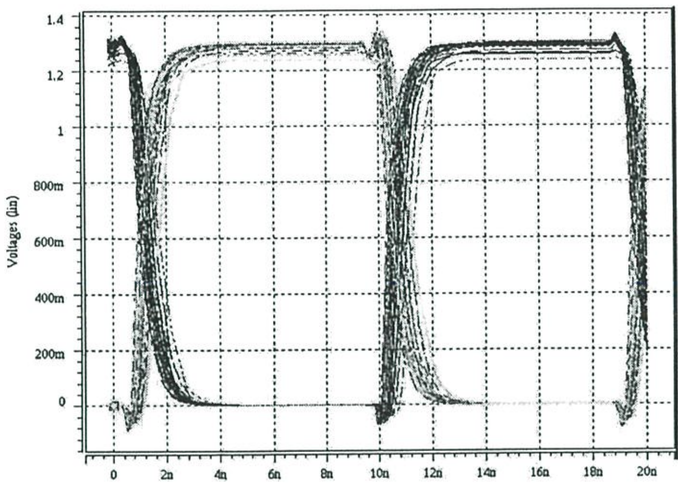
รูปที่ 3.12 เลย์เอาต์ของวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

3.4 ผลการทดสอบการทำงาน

ก่อนจะนำวงจรที่ออกแบบได้ทั้งในส่วนดิจิตอลและส่วนอนาลอกไปเจือสาร ได้มีการจำลองการทำงานเพื่อตรวจสอบความถูกต้องของวงจร รูปที่ 3.13 แสดงการจำลองการสังเคราะห์แอมปลิจูดดิจิตอลของสัญญาณซายน์ความถี่ 4.032 เมกะเฮิรตซ์ และรูปที่ 3.14 แสดงการจำลองการทำงานของวงจรแปลงดิจิตอลเป็นอนาลอกโดยการเปลี่ยนค่าของอุณหภูมิและแรงดันเพื่อตรวจสอบเวลาขาขึ้น (Rising time) และเวลาขาลง (Falling time)

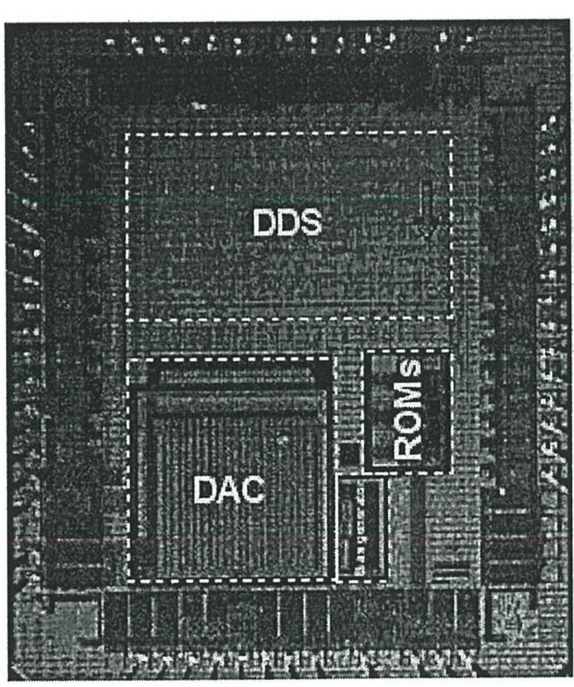


รูปที่ 3.13 การจำลองสัญญาณแอมปลิจูดดิจิตอลเอาต์พุตที่ความถี่ 4.032 เมกะเฮิร์ตซ์



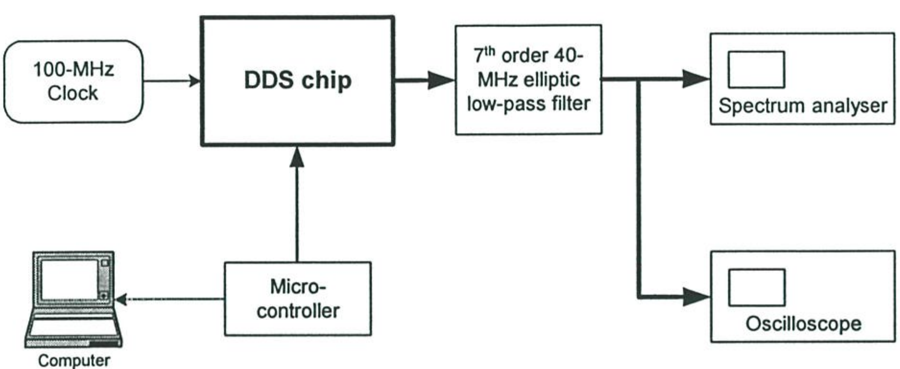
รูปที่ 3.14 การจำลองสัญญาณเอาต์พุตของวงจรแปลงดิจิตอลเป็นอนาลอกเมื่อมีการเปลี่ยนแปลงอุณหภูมิและแรงดัน

หลังจากที่ได้ตรวจสอบความถูกต้องในการทำงานของวงจรเรียบร้อยแล้ว ชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำได้ถูกส่งไปทำการเจือสาร และเข้าแพคเกจ ณ. ประเทศเบลเยี่ยม รูปถ่ายของชิปวงจรรวมเพื่อสังเคราะห์ความถี่ที่เจือสารเสร็จแล้วแสดงได้ดังรูปที่ 3.15

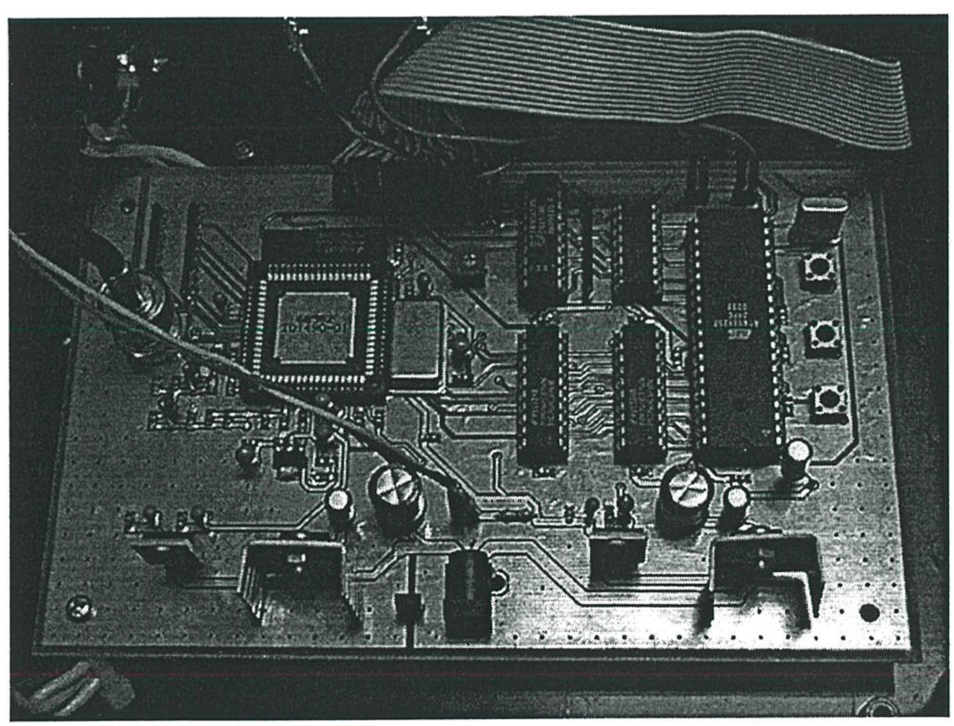


รูปที่ 3.15 รูปถ่ายของวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

หลังจากที่ได้ชิปสังเคราะห์ความถี่ที่เจือสารเรียบร้อยแล้วได้ทำการทดสอบการทำงานของชิป โดยมีแผนผังการทดสอบดังแสดงในรูปที่ 3.16 ประกอบด้วยเครื่องคอมพิวเตอร์, บอร์ดไมโครคอนโทรลเลอร์, วงจรความถี่ต่ำผ่านแบบอิลลิปติกอันดับที่เจ็ด 40 เมกะเฮิรตซ์, สเปกตรัมอนาไลเซอร์ , และดิจิตอลออสซิโลสโคป และรูปถ่ายของบอร์ดทดสอบแสดงในรูปที่ 3.17

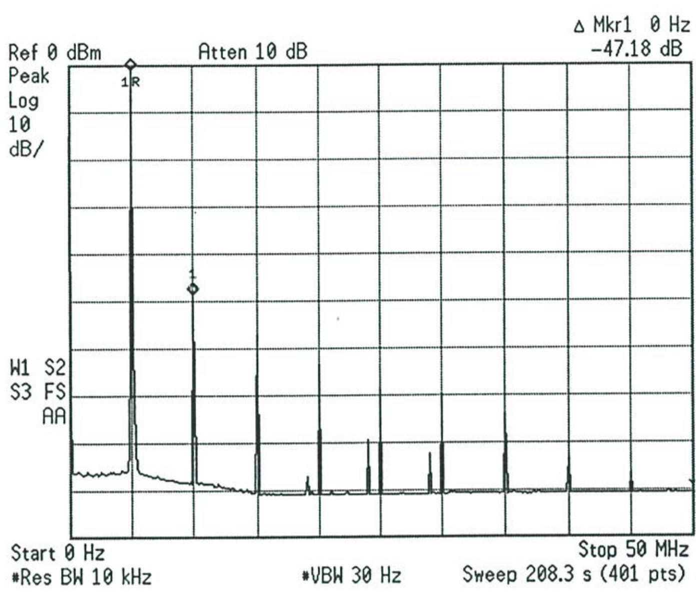


รูปที่ 3.16 ผังการทดสอบชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

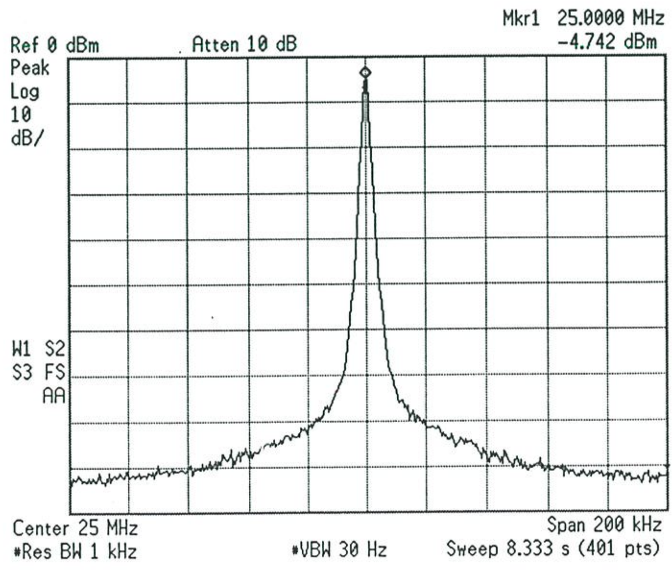


รูปที่ 3.17 ภาพถ่ายบอร์ดทดสอบชิปวงจรรวมเพื่อสังเคราะห์ความถี่

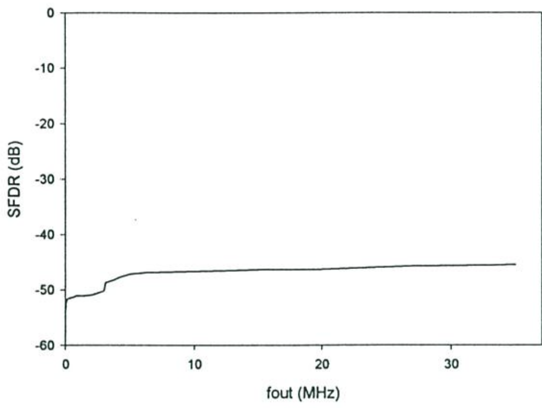
ความถี่ของสัญญาณนาฬิกาที่ป้อนให้กับชิปสังเคราะห์ความถี่คือ 100 เมกะเฮิรตซ์ และชิปสังเคราะห์ความถี่สามารถกำเนิดสัญญาณซายน์ได้ที่ความถี่ตั้งแต่ 0 ถึง 35 เมกะเฮิรตซ์ มีค่าเอสเอฟดีอาร์ (SFDR) ไม่เกิน –45 เดซิเบล และมีขนาดของสัญญาณรบกวนทางเฟสไม่เกิน –115 เดซิเบล ที่ความถี่ออฟเซ็ต 100 กิโลเฮิรตซ์ รูปที่ 3.18 และ 3.19 แสดงสเปกตรัมองค์ประกอบของสัญญาณซายน์ที่ 5 เมกะเฮิรตซ์ และสัญญาณรบกวนทางเฟสของสัญญาณซายน์ 25 เมกะเฮิรตซ์ ตามลำดับ และค่าเอสเอฟดีอาร์ (SFDR) ของสัญญาณเอาต์พุตความถี่ต่างๆ ขณะทำงานที่ความเร็วสัญญาณนาฬิกา 100 เมกะเฮิรตซ์สามารถพล็อตเป็นกราฟได้ดังรูปที่ 3.20



รูปที่ 3.18 สเปกตรัมองค์ประกอบของสัญญาณซายน์ 5 เมกะเฮิรตซ์



รูปที่ 3.19 สัญญาณรบกวนทางเฟสของสัญญาณซายน์ 25 เมกะเฮิรตซ์



รูปที่ 3.20 ค่าเอสเอฟดีอาร์ (SFDR) ที่ความถี่เอาต์พุตต่างๆ ขณะทำงานที่ความถี่สัญญาณนาฬิกา 100 เมกะเฮิรตซ์

3.5 สรุป

ชิปสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำที่ออกแบบสามารถทำงานได้ตามข้อกำหนดที่กำหนดไว้ และสามารถสรุปคุณสมบัติทั้งหมดของวงจรรวมเพื่อสังเคราะห์ความถี่ได้ดังตารางที่ 3.1

ตารางที่ 3.1 คุณสมบัติของวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

<table>
<tr><td colspan="2">Technology</td><td>Alcatel's single-poly, triple-metal 0.5-μm CMOS</td></tr>
<tr><td colspan="2">Power supply voltage</td><td>3.3 V</td></tr>
<tr><td colspan="2">Max. clock frequency</td><td>100 MHz</td></tr>
<tr><td colspan="2">Max. output frequency</td><td>35 MHz at 100-MHz clock rate</td></tr>
<tr><td colspan="2">Frequency resolution</td><td>32 bits (0.0233 Hz at 100 MHz)</td></tr>
<tr><td colspan="2">Phase tuning resolution</td><td>12 bits (0.0879 degrees)</td></tr>
<tr><td colspan="2">Amplitude resolution</td><td>10 bits (0.846 mV at R_l = 25 Ω)</td></tr>
<tr><td colspan="2">Modulation</td><td>Frequency and phase modulation</td></tr>
<tr><td rowspan="2">f_{out} = 25MHz</td><td>Worst-case spurious</td><td>-45 dBc</td></tr>
<tr><td>Phase noise</td><td>-115 dBc/Hz at 100-kHz offset</td></tr>
<tr><td colspan="2">Max. frequency switching speed</td><td>190 ns (19 x 1/100MHz)</td></tr>
<tr><td colspan="2">Die area (including pads)</td><td>3240x3830 μm^2</td></tr>
<tr><td rowspan="2">Power dissipation</td><td>Digital</td><td>270 mW at 100-MHz clock rate</td></tr>
<tr><td>Analog</td><td>126 mW at full-scale output level</td></tr>
</table>

บทที่ 4

วงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอล โดยใช้วิธีการประมาณค่าโพลีโนเมียล แทนการใช้หน่วยความจำ

ในบทนี้จะนำเสนอการออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ [24, 25] ซึ่งมีข้อได้เปรียบวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำในบทที่ 3 คือวงจรที่สร้างขึ้นจะไม่มีการใช้หน่วยความจำสำหรับเก็บรูปสัญญาณ นั่นคือข้อจำกัดของการใช้หน่วยความจำที่ได้กล่าวไว้แล้วในบทที่ผ่านมา ไม่ว่าจะเป็นความเร็วในการเข้าถึงหน่วยความจำ, ขนาดวงจร, คุณภาพสัญญาณ ตลอดจนการกินกำลังงานของวงจร จะถูกตัดทิ้งไป [6] ทำให้วงจรที่สังเคราะห์ความถี่ด้วยการประมาณค่าโพลีโนเมียลที่ออกแบบได้นั้นสามารถสังเคราะห์สัญญาณได้มีคุณภาพสูงกว่าวงจรสังเคราะห์ความถี่แบบที่ใช้หน่วยความจำ อีกทั้งยังสามารถเพิ่มความเร็วในการทำงานของวงจรด้วยการใช้เทคนิคการทำงานแบบสายท่อ (Pipeline) หรือออกแบบด้วยสถาปัตยกรรมของวงจรคูณที่มีความเร็วสูงได้อีกด้วย

หัวข้อ 4.1 จะเป็นการนำเสนอหลักการและสมการโพลีโนเมียลที่ใช้ในการออกแบบวงจรซึ่งจะถูกจำลองการทำงานในระดับระบบ (System-level simulation) ด้วยโปรแกรมแมทแลบ (MATLAB) ในหัวข้อที่ 4.2 ส่วนรายละเอียดการออกแบบวงจรสร้างเฟสและวงจรเปลี่ยนเฟสเป็นแอมปลิจูดจะถูกนำเสนอไว้ในหัวข้อ 4.3 และ 4.4 ตามลำดับ หลังจากออกแบบวงจรเสร็จแล้วผลการจำลองการทำงานและทดสอบการทำงานของวงจรจะถูกนำเสนอในหัวข้อที่ 4.5 และปิดท้ายด้วยบทสรุปในหัวข้อที่ 4.6

4.1 สถาปัตยกรรมของวงจรที่นำเสนอ (Proposed Architecture)

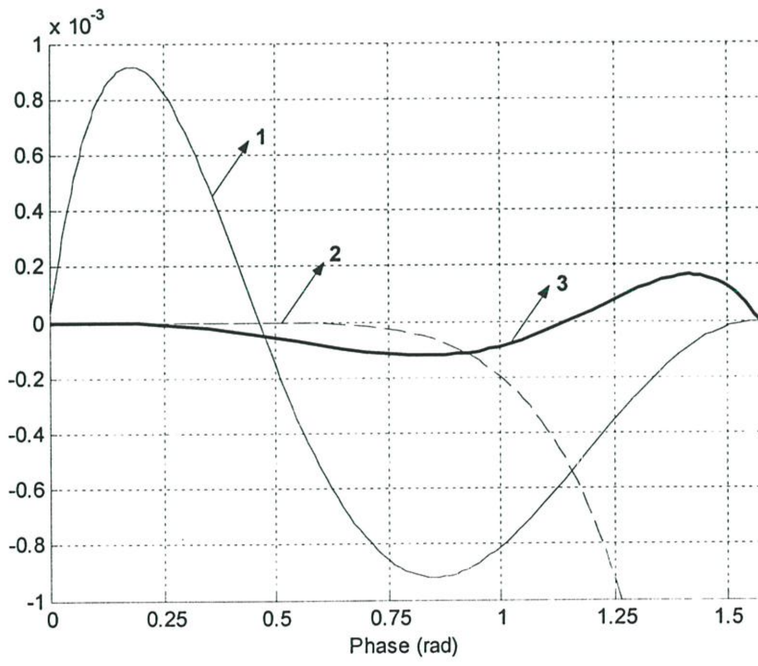
4.1.1 การประมาณค่าโพลีโนเมียล (The Polynomial Approximation)

วงจรสังเคราะห์ความถี่แบบดิจิตอลที่ออกแบบจะสังเคราะห์สัญญาณรูปซายน์ด้วยสมการอนุกรมเทย์เลอร์อันดับห้า

$$\sin(\phi) = \phi\left(1 + a_2\phi^2 + a_4\phi^4\right) + \varepsilon(\phi) \qquad ,0 \le \phi \le \frac{\pi}{2} \tag{4.1}$$

$$a_2 = -0.16605, a_4 = 0.00761 \tag{4.2}$$

โดยที่สัมประสิทธิ์ a_2 และ a_4 ที่ใช้ในการออกแบบนั้นจะแตกต่างจากสัมประสิทธิ์ที่ใช้ในอนุกรมเทย์เลอร์ [26] คือได้ถูกคำนวณให้มีค่าความผิดพลาดในช่วง 0 ถึง $\pi/2$ $|\varepsilon(\phi)|$ น้อยกว่า 2e-4 ซึ่งเมื่อเปรียบเทียบกับการประมาณค่าซายน์ด้วยสมการอนุกรมเทย์เลอร์อันดับห้า และการประมาณค่าพาราโบลาอันดับสอง [18] แล้วจะเห็นว่าการประมาณค่าด้วยสมการที่ (4.1) ที่มีสัมประสิทธิ์ดังสมการที่ (4.2) นั้นสามารถสังเคราะห์สัญญาณซายน์ได้ใกล้เคียงกับสัญญาณซายน์จริงมากที่สุด ดังแสดงในรูปที่ 4.1



รูปที่ 4.1 ค่าความผิดพลาดของการประมาณค่าซายน์ด้วยสมการต่างๆ: (1) สมการพาราโบลาอันดับสอง; (2) สมการอนุกรมเทย์เลอร์อันดับห้า (3) สมการโพลีโนเมียลที่นำเสนอ

4.1.2 การออกแบบเป็นวงจรดิจิตอล (Digital Realisation)

เราสามารถเขียนความสัมพันธ์ของเอาต์พุตของวงจรสร้างเฟสขนาด N บิต (m) กับค่าเฟส (ϕ) ของวงจรสังเคราะห์ความถี่แบบดิจิตอลที่มีสถาปัตยกรรมตามรูปที่ 2.1 ได้ดังนี้

$$m = \left(\frac{\phi}{2\pi}\right)2^N \tag{4.3}$$

เพื่อให้สะดวกต่อการออกแบบสมการที่ (4.1) จะถูกคูณด้วย 10^9 และจากความสัมพันธ์ของสมการที่ (4.3) เราสามารถเขียนสมการที่ (4.1) ให้อยู่ในรูปของเอาต์พุตของวงจรสร้างเฟสได้ดังนี้

$$y(m) = m\left(k_0 - k_2 m^2 + k_4 m^4\right) \tag{4.4}$$

โดยที่ค่า k_0, k_2, k_4 มีค่าดังตารางที่ 4.1

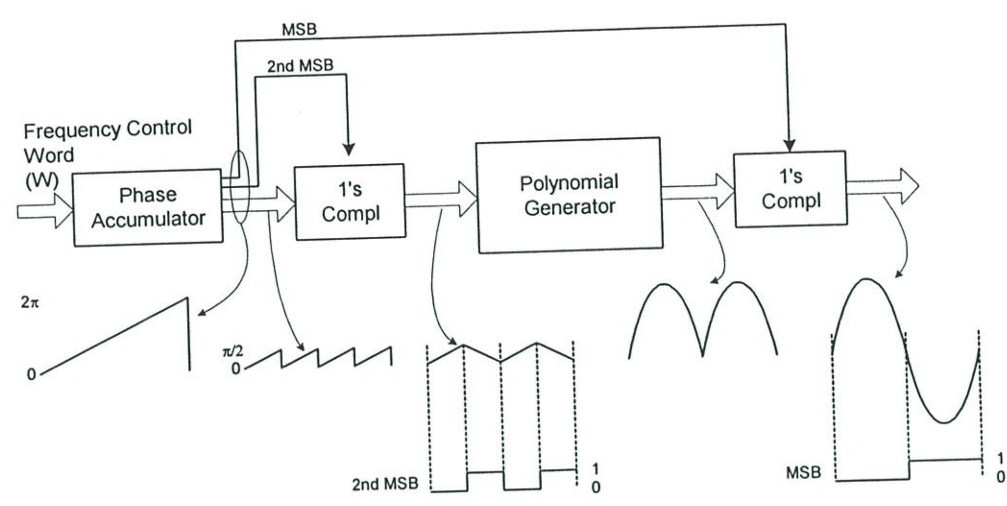
ตารางที่ 4.1 ค่าคงที่ที่ใช้ในสมการที่ (4.4)

k_0	1,000,000,000
k_2	391.046368345922...
k_4	4.22049520104...e-5

จะเห็นว่าถ้าเรานำสมการที่ (4.4) ไปออกแบบเป็นวงจรจะต้องมีการคูณกันถึงห้าครั้ง ซึ่งไม่เหมาะต่อการนำไปออกแบบเป็นวงจรทางฮาร์ดแวร์ ดังนั้นการคูณของ $k_2 m^2$ และ $k_4 m^4$ จึงถูกออกแบบด้วยการเลื่อน-บวก (Shift-and-add) ซึ่งจะได้ค่าของ k_2 และ k_4 จากการเลื่อน-บวกดังตารางที่ 4.2

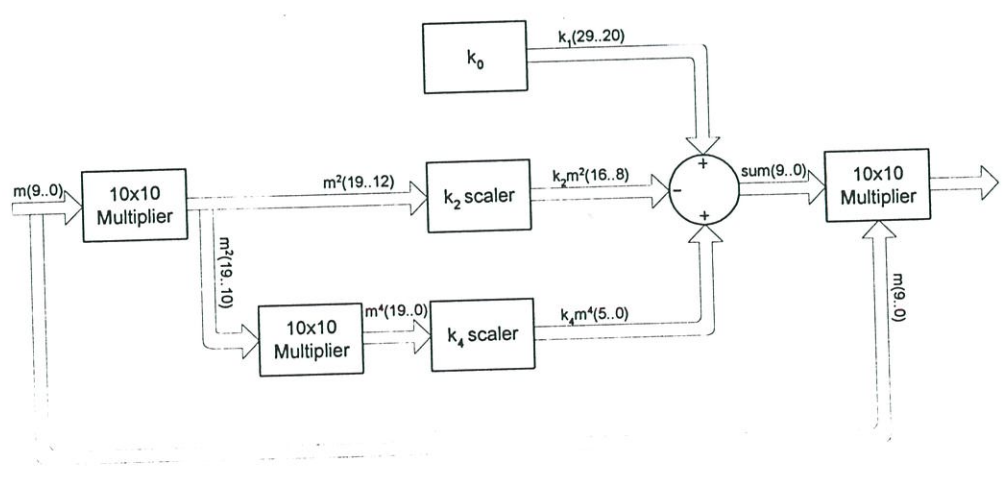
ตารางที่ 4.2 ค่าของ k_2 และ k_4 ที่คำนวณจากการเลื่อน-บวก

K	Value	Shift-add operation
k_2	392	$2^8 + 2^7 + 2^3$
k_4	4.1961669921875e-5	$1/2^{15} + 1/2^{16} - 1/2^{18}$



รูปที่ 4.2 สถาปัตยกรรมของวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียล

สถาปัตยกรรมของวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแสดงได้ดังรูปที่ 4.2 ประกอบด้วยวงจรสร้างเฟส, วงจรสร้างโพลีโนเมียล โดยวงจรสร้างเฟสจะทำหน้าที่สร้างสัญญาณเฟสสำหรับป้อนให้กับวงจรสร้างโพลีโนเมียล (Polynomial generator) และเนื่องจากค่าสัญญาณซายน์ที่ถูกประมาณค่าด้วยสมการที่ (4.1) นั้นมีค่าใกล้เคียงกับสัญญาณซายน์จริงในช่วง 0 ถึง π/2 ดังนั้นเทคนิคการบีบอัดควอแดรนท์ [21] จึงถูกนำมาใช้ โดยที่บิตเอ็มเอสบีบนสุด (MSB) ถูกใช้สำหรับกำหนดแอมปลิจูดของสัญญาณซายน์ว่าเป็นบวกหรือลบ และบิตเอ็มเอสบีถัดมา (2nd MSB) ถูกใช้สำหรับกำหนดว่าค่าของเฟสจะถูกคอมพลีเมนต์หรือไม่



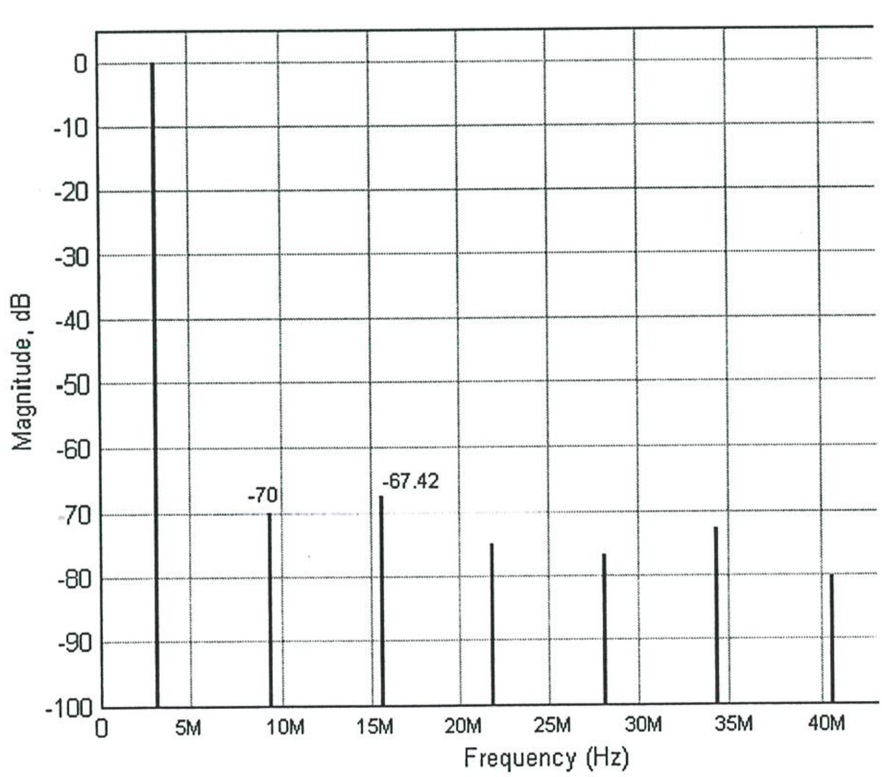
รูปที่ 4.3 โครงสร้างของวงจรสร้างโพลีโนเมียล

รูปที่ 4.3 แสดงโครงสร้างของวงจรสร้างโพลีโนเมียลที่ใช้สำหรับการประมาณค่าแอมปลิจูดของสัญญาณซายน์ในช่วง 0 ถึง $\pi/2$ ซึ่งประกอบด้วยวงจรคูณขนาด 10x10 บิต จำนวน 3 วงจร, วงจรเลื่อน-บวก (Shift-and-add) สำหรับคำนวณค่า k_2m^2 และ k_4m^4 และวงจรบวกและวงจรลบอย่างละหนึ่งตัว

สัญญาณเฟสขนาด 32 บิต จากวงจรสร้างเฟสจะถูกตัดให้เหลือเฉพาะบิตเอ็มเอสบีบนสุด 12 บิต *m(11:0)* โดยที่บิตเอ็มเอสบีสองบิตแรกถูกใช้สำหรับการกำหนดควอ-แดรนท์ และอีก 10 บิตที่เหลือ *m(9:0)* จะถูกนำไปใช้ในการคำนวณเป็นสัญญาณซายน์

4.2 ผลการจำลองการทำงานของระบบ (System-level simulation)

สถาปัตยกรรมของวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้การประมาณค่าโพลีโนเมียลที่นำเสนอในวิทยานิพนธ์ได้ถูกจำลองการทำงานในระดับระบบด้วยโปรแกรมแมทแลบ (MATLAB) เพื่อตรวจสอบความเป็นไปได้และความถูกต้องในการทำงานของวงจร และการจำลองการทำงานระดับระบบยังช่วยในการคำนวณหาขนาดบิตของสัญญาณที่จะใช้ในการออกแบบอีกด้วย

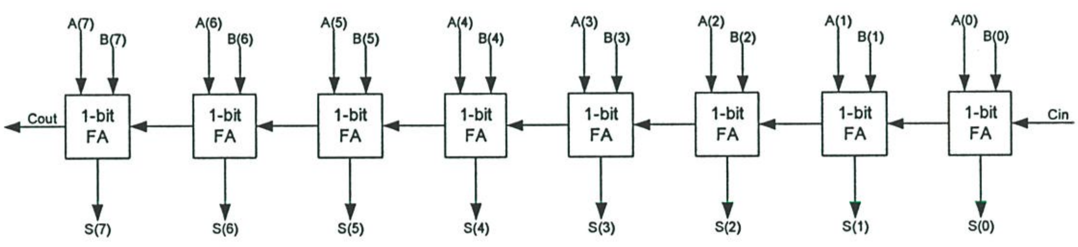


รูปที่ 4.4 สเปกตรัมของสัญญาณซายน์ความถี่ 1/32 เท่าของสัญญาณนาฬิกาจากการจำลองการทำงานของระบบ

รูปที่ 4.4 แสดงสเปกตรัมของสัญญาณซายน์ความถี่ 1/32 เท่าของสัญญาณนาฬิกาที่ได้จากการจำลองการทำงานในระดับระบบซึ่งมีค่าระดับเอสเอฟดีอาร์ (SFDR) สูงสุดเท่ากับ –67.42 เดซิเบล และจาก [9] ที่กล่าวไว้ว่าค่าเอสเอฟดีอาร์สูงสุดของการลดทอน (Truncate) สัญญาณแอมปลิจูดเอาต์พุตให้เหลือ 10 บิต มีค่าประมาณ –61.96 เดซิเบล ดังนั้นเราสามารถสรุปได้ว่าวิธีการประมาณค่าโพลีโนเมียลด้วยสมการโพลีโนเมียลนี้เหมาะสมกับการนำไปใช้ออกแบบวงจรแปลงเฟสเป็นแอมปลิจูดในวงจรสังเคราะห์ความถี่แบบดิจิตอล

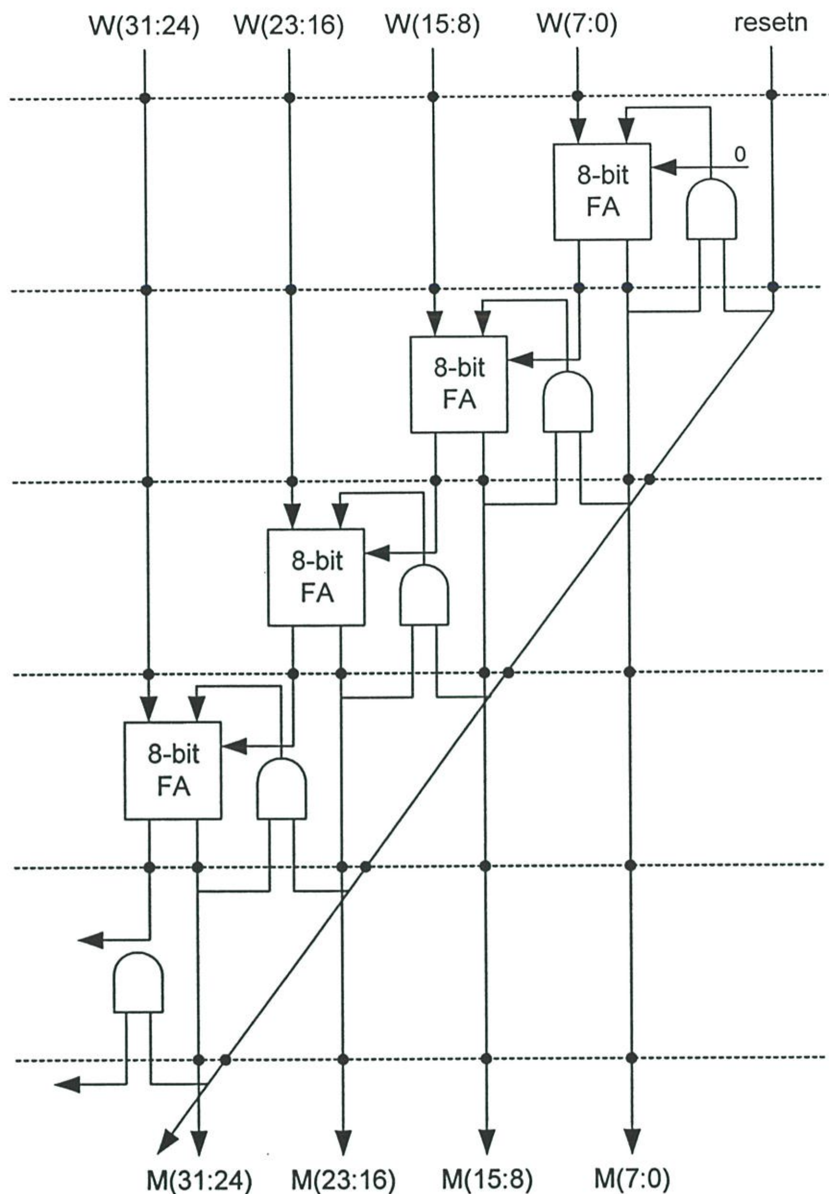
4.3 วงจรสร้างเฟส (Phase Accumulator)

วงจรสร้างเฟสถูกออกแบบด้วยวิธีบนลงล่าง (Top-down design) โดยเขียนโค้ดบรรยายพฤติกรรมการทำงานของวงจรด้วยภาษาวีเอชดีแอล แล้วจำลองการทำงานระดับฟังก์ชั่น และเมื่อตรวจสอบความถูกต้องของการทำงานในระดับฟังก์ชั่นเรียบร้อยแล้ว ก็จะนำไปสังเคราะห์ (Synthesis) และจัดวาง-เชื่อมโยง (Place-and-route) ตามเทคโนโลยีของชิปเอฟพีจีเอที่ใช้ในการทดสอบ ซึ่งในวิทยานิพนธ์นี้ใช้ชิปเอฟพีจีเอตระกูลสปาร์ทาน 2 ของบริษัทไซลิงซ์ (Xilinx) เบอร์ XC2S100PQ208-5 จากนั้นจึงนำเนทลิสต์ของวงจรที่จัดวางและเชื่อมโยงเรียบร้อยแล้วไปดาวน์โหลดลงในบอร์ดทดสอบเพื่อตรวจสอบความถูกต้องในการทำงานของวงจรในระดับฮาร์ดแวร์



รูปที่ 4.5 วงจรบวกแบบริปเปิลแครี่ขนาด 8 บิต

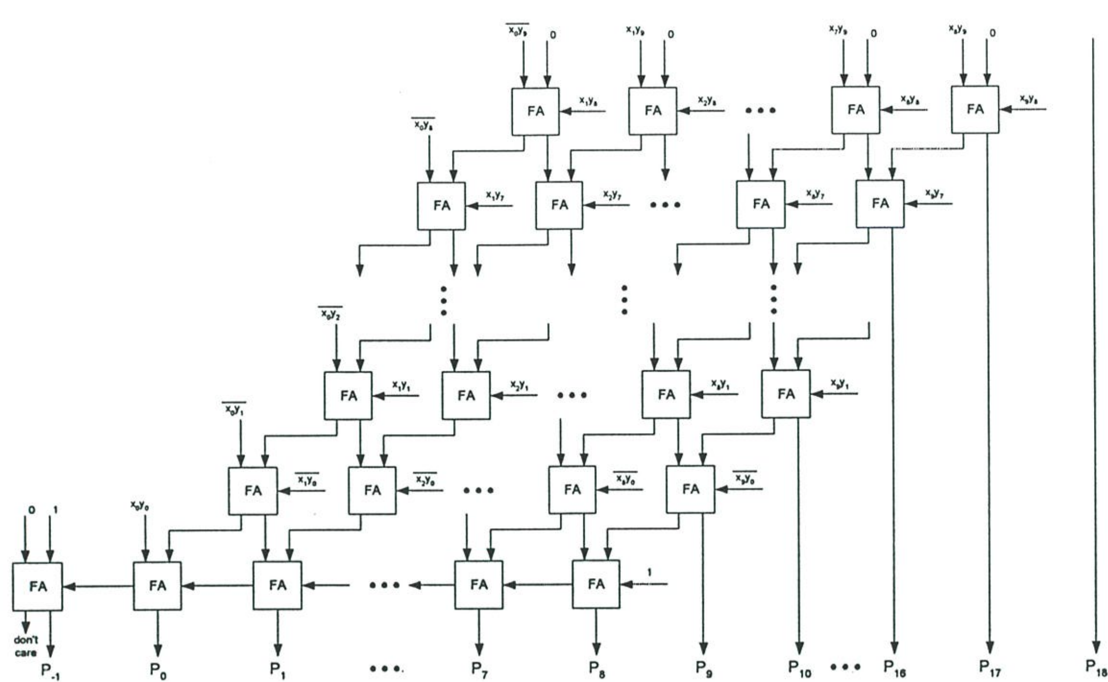
โครงสร้างของวงจรสร้างเฟสขนาด 32 บิต มีโครงสร้างดังรูปที่ 4.6 ประกอบด้วยวงจรบวกแบบริปเปิลแครี่ขนาด 8 บิต ที่มีโครงสร้างดังรูปที่ 4.5 และมีกระบวนการรีเซตค่าของรีจิสเตอร์เฟสแบบสายท่อ (Pipeline) โดยที่จุดดำในรูปที่ 4.5 เป็นรีจิสเตอร์ที่ใช้ในการปรับค่าเวลา (Pre-skewing and De-skewing registers)



รูปที่ 4.6 วงจรสร้างเฟสขนาด 32 บิต

4.4 **วงจรเปลี่ยนเฟสเป็นแอมปลิจูด** (Phase-to-Amplitude Converter)

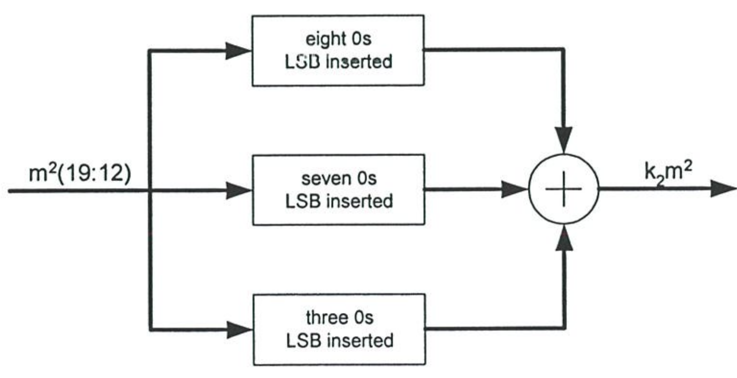
วงจรเปลี่ยนเฟสเป็นแอมปลิจูดถูกออกแบบด้วยวิธีบนลงล่างเช่นเดียวกับวงจรสร้างเฟส ในหัวข้อที่ 4.3 โดยการเปลี่ยนเฟสเป็นแอมปลิจูดของสัญญาณซายน์จะออกแบบตามสมการที่ (4.4) และมีโครงสร้างการทำงานดังรูปที่ 4.3 ประกอบด้วยวงจร 10x10 บิต จำนวน 3 ชุด วงจรเลื่อน-บวก (Shift-and-add) สำหรับคำนวณค่า k_2m^2 และ k_4m^4 และวงจรบวกและวงจรลบอย่างละหนึ่งตัว



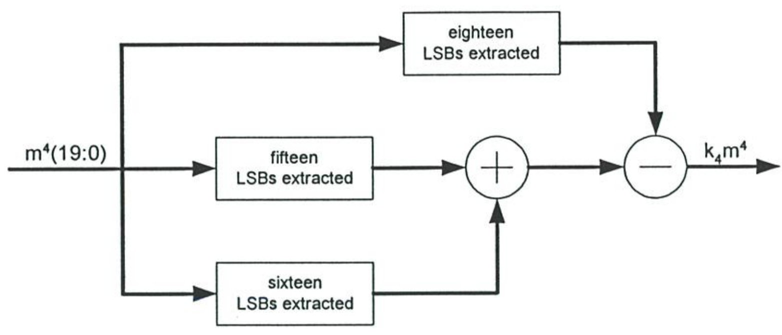
รูปที่ 4.7 วงจรคูณ 10x10 แบบบอว์-วูลีย์ (Baugh-Wooley's Multiplier)

วงจรคูณ 10x10 ทั้งสามวงจรถูกออกแบบด้วยสถาปัตยกรรมแบบบอว์-วูลีย์ [2] (Baugh-Wooley) ดังแสดงในรูปที่ 4.7 โดยมีสัญญาณอินพุตขนาด 10 บิต สองสัญญาณคือ x(9:0) และ y(9:0) และ P(-1:18) เป็นสัญญาณเอาต์พุตของการคูณขนาด 20 บิต

การคูณของ k_2m^2 และ k_4m^4 ในสมการที่ (4.4) และตารางที่ 4.2 ถูกออกแบบด้วยการเลื่อน-บวก (Shift-and-add) ที่มีโครงสร้างดังรูปที่ 4.8 และ 4.9 ตามลำดับ สำหรับการบวกและลบทั้งหมดในวงจรนั้นถูกออกแบบด้วยวงจรบวกแบบริปเปิลแคร์รี [2]



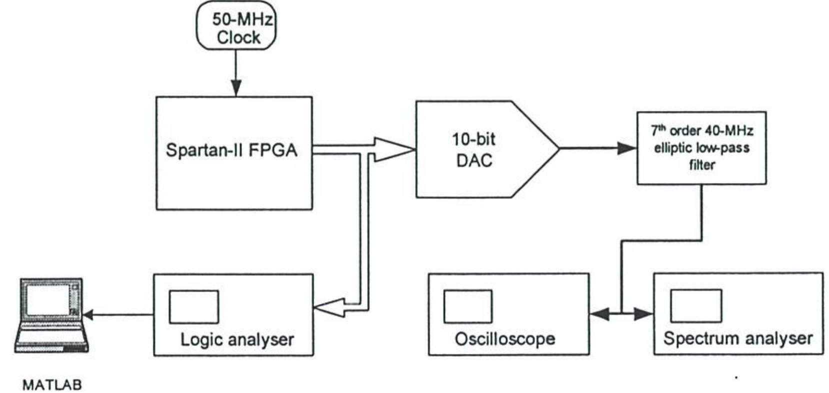
รูปที่ 4.8 โครงสร้างการคำนวณ k_2m^2



รูปที่ 4.9 โครงสร้างการคำนวณ k_4m^4

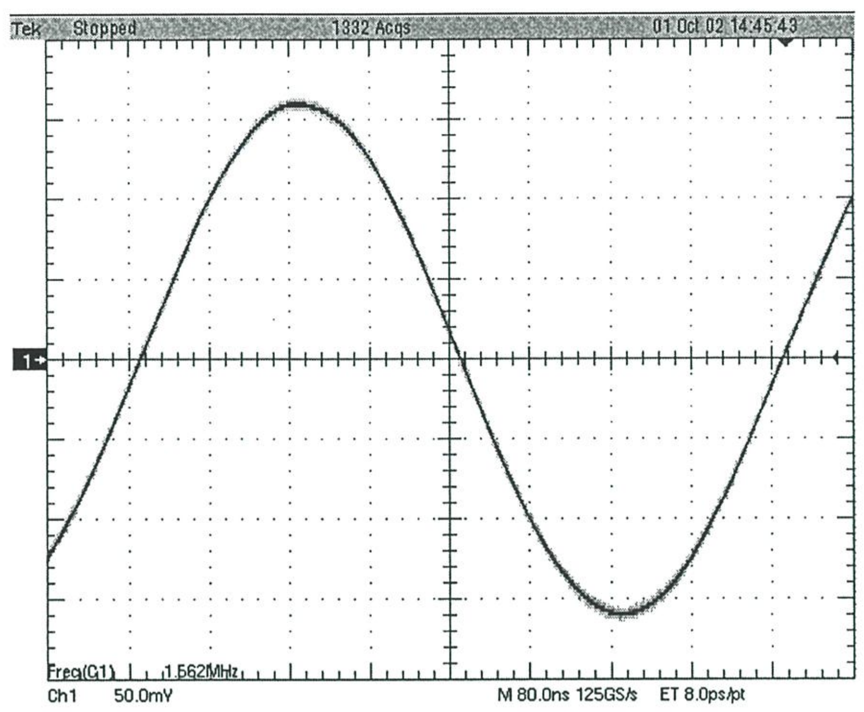
4.5 ผลการจำลองการทำงานและการทดสอบการทำงานของวงจร(Simulation and Experimental Results)

หลังจากที่เขียนโค้ดภาษาวีเอชดีแอล บรรยายพฤติกรรมการทำงานของวงจรสร้างเฟส เรียบร้อยแล้วได้ทำการสังเคราะห์และจัดวาง-เชื่อมโยงโดยใช้เทคโนโลยีของชิปเอฟพีจีเอตระกูล สปาร์ทาน 2 เบอร์ XC2S100PQ208-5 และดาวน์โหลดเนทลิสต์ของวงจรลงในบอร์ดทดสอบ โดย แผนผังของการทดสอบสามารถแสดงได้ดังรูปที่ 4.10 ประกอบด้วยชิปเอฟพีจีเอทำงานที่ความถี่ สัญญาณนาฬิกา 50 เมกะเฮิรตซ์ ทำหน้าที่สร้างแอมปลิจูดดิจิตอลของสัญญาณซายน์ขนาด 10 บิต, วงจรแปลงดิจิตอลเป็นอนาลอกทำหน้าที่แปลงแอมปลิจูดดิจิตอลให้เป็นสัญญาณอนาลอก, ลอจิกอนาไลเซอร์ถูกใช้ในการเก็บค่าแอมปลิจูดดิจิตอลของสัญญาณซายน์ที่สังเคราะห์ได้เพื่อนำ ไปคำนวณหาค่าระดับฮาร์โมนิกด้วยโปรแกรมแมทแลบ (MATLAB) และสัญญาณอนาลอกเอาต์ พุตจากวงจรแปลงดิจิตอลเป็นอนาลอกจะถูกวัดรูปสัญญาณ (Waveform) และสเปกตรัม (Spectrum) ด้วยออสซิโลสโคปและสเปกตรัมอนาไลเซอร์ตามลำดับ

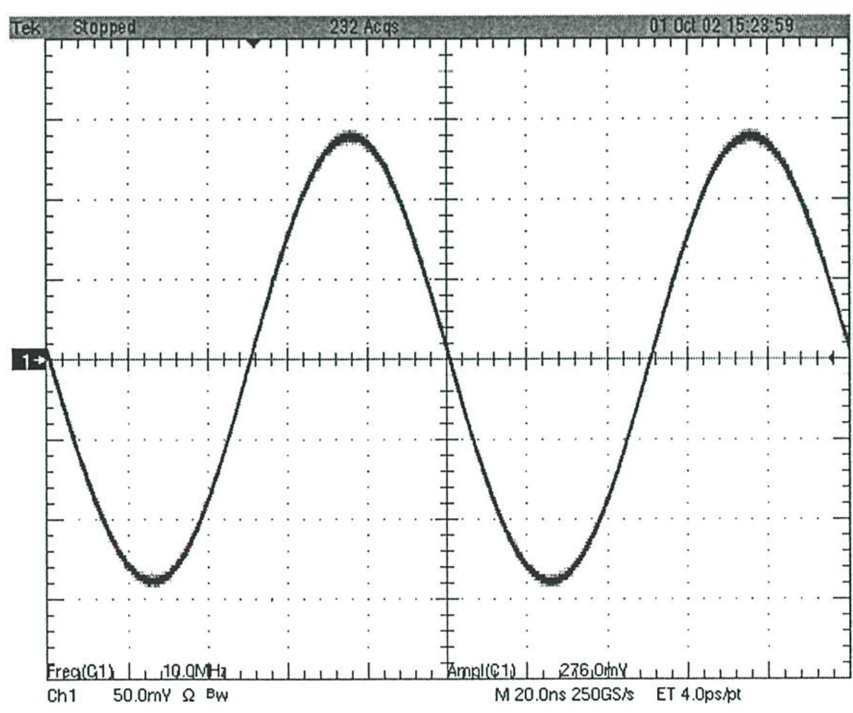


รูปที่ 4.10 ผังการทดสอบวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลโดยใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ

รูปสัญญาณซายน์ความถี่ 1.5625 เมกะเฮิรตซ์ และ 10 เมกะเฮิรตซ์ ที่วัดได้จากออสซิโลสโคปแสดงได้ดังรูปที่ 4.11 และ 4.12 ตามลำดับ



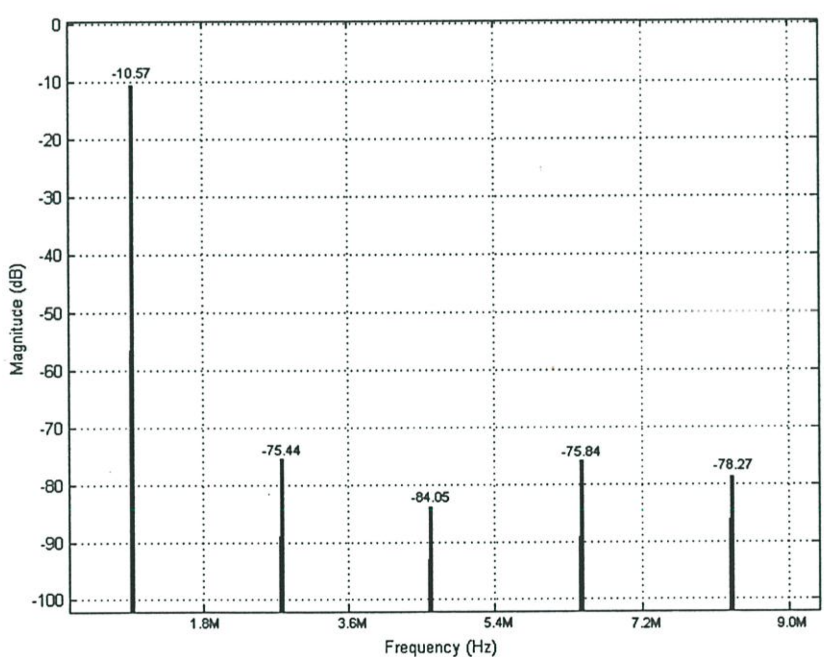
รูปที่ 4.11 รูปสัญญาณซายน์ 1.5625 เมกะเฮิรตซ์



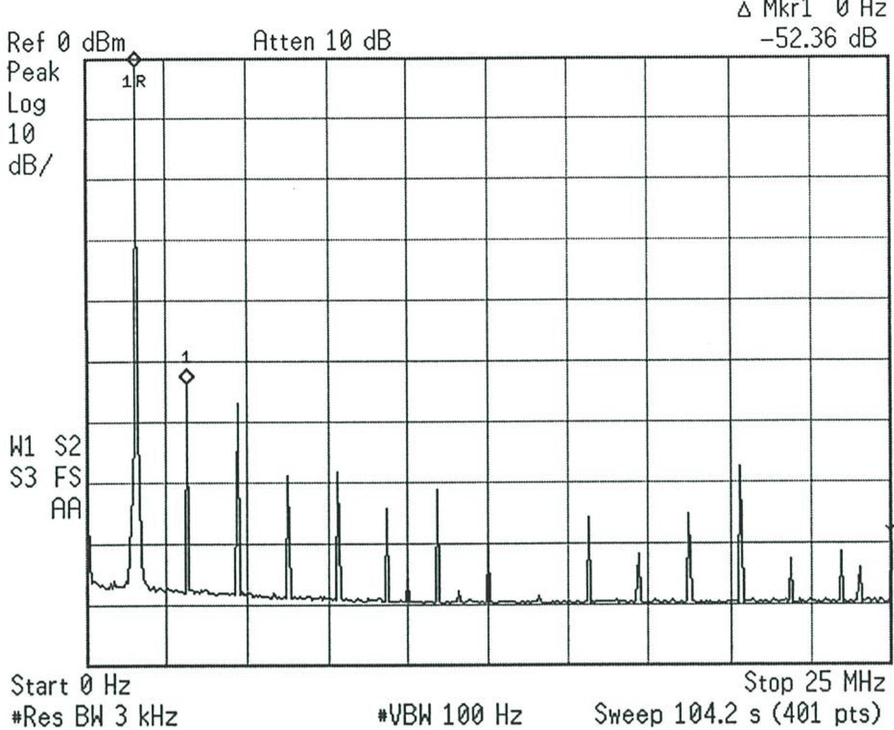
รูปที่ 4.12 รูปสัญญาณซายน์ 10 เมกะเฮิรตซ์

ค่าระดับเอสเอฟดีอาร์ (SFDR) สูงสุดของสัญญาณซายน์ 1.5625 เมกะเฮิรตซ์ ที่คำนวณจากโปรแกรมแมทแลบ (MATLAB) มีค่าต่ำกว่า –64.87 เดซิเบล ดังแสดงได้ดังรูปที่ 4.13 และสเปกตรัมของสัญญาณ 1.5625 เมกะเฮิรตซ์ ที่วัดได้จากสเปกตรัมอนาไลเซอร์ในรูปที่ 4.14 มีค่าระดับเอสเอฟดีอาร์สูงสุดเท่ากับ –52.36 เดซิเบล ซึ่งไม่เท่ากับที่คำนวณด้วยโปรแกรมแมทแลบ โดยอาจเป็นผลจากความไม่เป็นเชิงเส้นของวงจรแปลงดิจิตอลเป็นอนาลอก

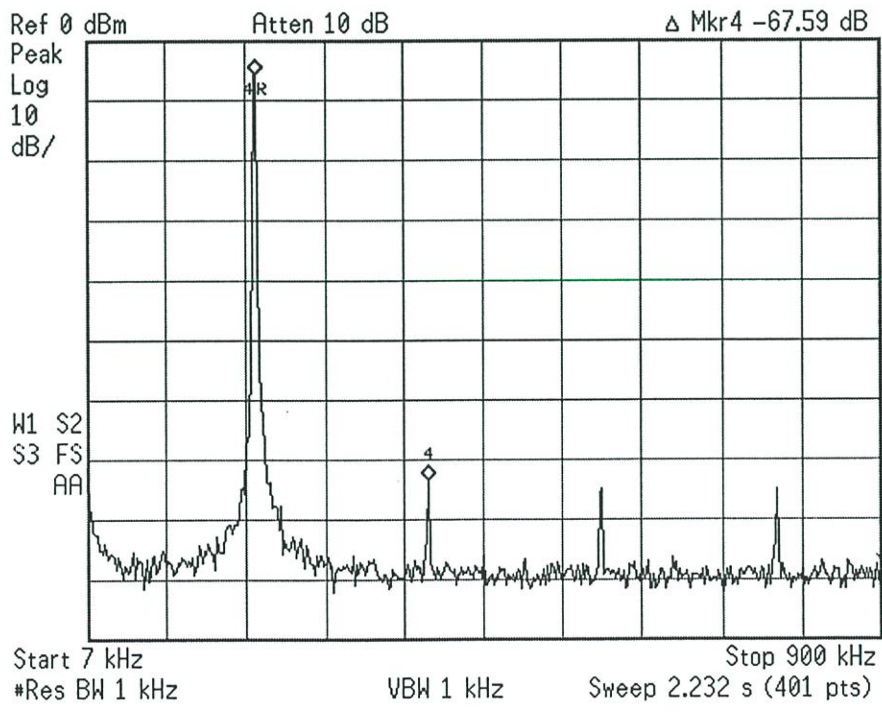
รูปที่ 4.15 แสดงสเปกตรัมของสัญญาณซายน์ที่ความถี่ 197 กิโลเฮิรตซ์ ที่วัดจากสเปกตรัมอนาไลเซอร์ ซึ่งมีค่าระดับเอสเอฟดีอาร์สูงสุดต่ำกว่า –67.59 เดซิเบล โดยที่เราถือว่าที่ความถี่ต่ำนั้นค่าเอสเอฟดีอาร์ที่วัดได้จะไม่ถูกกระทบจากความไม่เป็นเชิงเส้นของวงจรแปลงดิจิตอลเป็นอนาลอก และสัญญาณรบกวนทางเฟสของสัญญาณซายน์ 15 เมกะเฮิร์ตซ์ แสดงในรูปที่ 4.16



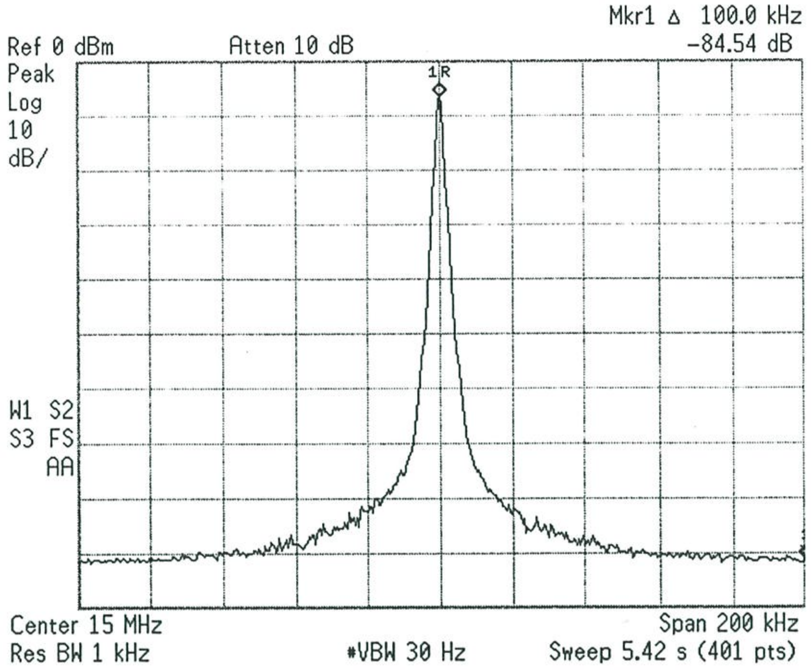
รูปที่ 4.13 สเปกตรัมของสัญญาณซายน์ 1.5625 เมกะเฮิรตซ์ ที่คำนวณจากโปรแกรมแมทแลบ



รูปที่ 4.14 สเปกตรัมของสัญญาณซายน์ 1.5625 เมกะเฮิรตซ์

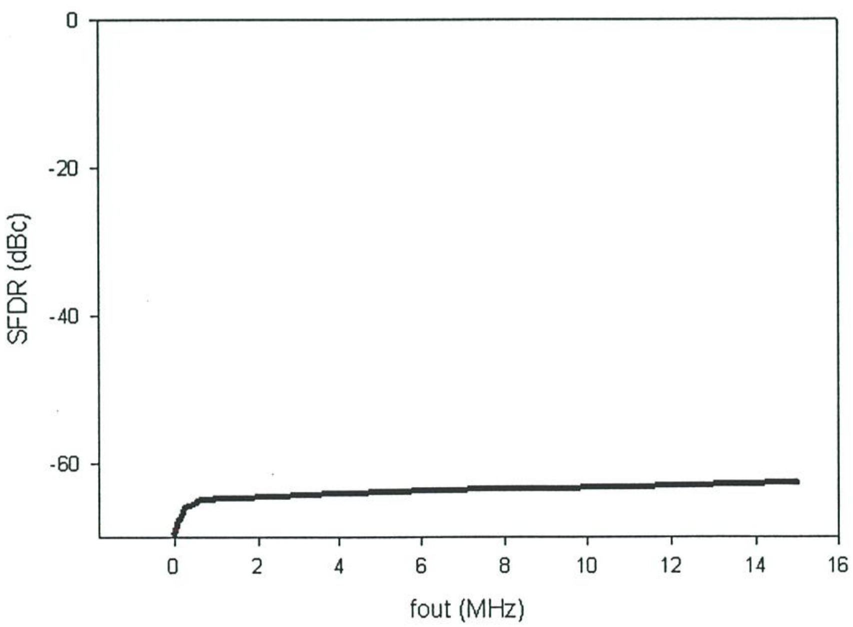


รูปที่ 4.15 สเปกตรัมของสัญญาณซายน์ 197 กิโลเฮิรตซ์



รูปที่ 4.16 สัญญาณรบกวนทางเฟสของสัญญาณซายน์ 15 เมกะเฮิรตซ์

เราสามารถพล็อตกราฟค่าเอสเอฟดีอาร์ที่คำนวณจากโปรแกรมแมทแลบสำหรับสัญญาณเอาต์พุตที่ความถี่ต่างๆ ได้ดังรูปที่ 4.17



รูปที่ 4.17 ค่าเอสเอฟดีอาร์ของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลโดยใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำที่ความถี่ต่างๆ

4.6 สรุป

วงจรสังเคราะห์ความถี่แบบดิจิตอลนำเสนอในบทนี้ เป็นการสังเคราะห์สัญญาณซายน์ด้วยสมการโพลีโนเมียล ซึ่งสัมประสิทธิ์ที่ใช้นั้นสามารถประมาณค่าสัญญาณซายน์ได้ใกล้เคียงกับสัญญาณซายน์จริงมาก มีค่าความผิดพลาดน้อยกว่า 2e-4 และเมื่อนำไปออกแบบเป็นวงจรและทดสอบการทำงานบนชิปเอฟพีจีเอแล้วสามารถสังเคราะห์สัญญาณซายน์ที่มีคุณภาพอยู่ในระดับที่สามารถจะนำไปใช้ร่วมกับวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิต ได้ [9] นอกจากนี้ยังสามารถพัฒนาวงจรในส่วนของวงจรคูณให้มีความเร็วในการทำงานสูงขึ้น [27] และนำวงจรกำลังสอง [28, 29] (Squarer) มาใช้ในการคำนวณค่า m^2 และ m^4 ในสมการที่ (4.4) ได้อีกด้วย คุณสมบัติการทำงานของวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำสามารถสรุปได้ดังตารางที่ 4.3 และในตารางที่ 4.4 เป็นการเปรียบเทียบคุณสมบัติของวงจรที่ออกแบบกับวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ไม่ใช้หน่วยความจำแบบการคูณเชิงซ้อน [16] และการประมาณค่าพาราโบลาอันดับสอง [18]

ตารางที่ 4.3 คุณสมบัติของวงจรสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ

FPGA Technology	Spartan-II (XC2S100PQ208-5)
Frequency control word	32 bits
Phase resolution	12 bits
No. of output bits	10 bits
Frequency switching latency	6 clock cycles
Maximum clock frequency	50 MHz
Worst case spurious	-64.87 dBc (excluding DAC)
Total no. of CLBs	266 CLBs

ตารางที่ 4.4 เปรียบเทียบคุณสมบัติของวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ไม่ใช้หน่วยความจำ

	วงจรที่ออกแบบ	[16]	[18]
ขนาดแอมปลิจูด	**10 บิต**	8 บิต	10 บิต
ค่าเอสเอฟดีอาร์ (SFDR)	**-64.87 dBc**	-44 dBc	-64 dBc

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

วิทยานิพนธ์ฉบับนี้ได้นำเสนอการออกแบบวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลโดยใช้วิธีการประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ โดยมีวัตถุประสงค์เพื่อหลีกเลี่ยงการใช้หน่วยความจำในการเก็บรูปสัญญาณ ซึ่งทำให้ประสิทธิภาพของวงจรที่ออกแบบได้นั้นจะไม่ถูกจำกัดด้วยขนาดและความเร็วของหน่วยความจำ อีกทั้งการใช้หน่วยความจำที่มีขนาดใหญ่ยังทำให้ค่าใช้จ่ายในการเจือสารสูงขึ้น และวงจรกินกำลังงานมากขึ้นอีกด้วย

สมการที่นำมาใช้ในการออกแบบวงจรสังเคราะห์ความถี่แบบดิจิตอลในวิทยานิพนธ์นี้เป็นสมการโพลีโนเมียลที่ถูกคำนวณค่าของสัมประสิทธิ์ในสมการให้สามารถประมาณรูปสัญญาณซายน์ได้ใกล้เคียงกับสัญญาณจริงมากกว่าการประมาณด้วยสมการพาราโบลาอันดับสอง [18] และสมการอนุกรมเทเลอร์อันดับห้า กล่าวคือมีค่าความผิดพลาดน้อยกว่า 2e-4 ซึ่งจากผลการจำลองการทำงานของสมการและระบบด้วยโปรแกรมแมทแลบ (MATLAB) บนคอมพิวเตอร์นั้นค่าของระดับฮาร์โมนิกที่สูงที่สุดมีค่าต่ำกว่า –67.42 เดซิเบล ซึ่งสามารถนำไปใช้ร่วมกับวงจรแปลงดิจิตอลเป็น อนาลอกขนาด 10 บิต ได้โดยไม่ทำให้คุณภาพของสัญญาณเสื่อมลง เพราะค่าระดับฮาร์โมนิกที่สูงที่สุดของวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิต จะมีค่าประมาณ –61.96 เดซิเบล [9] ซึ่งสูงกว่าฮาร์โมนิกที่สูงสุดของสถาปัตยกรรมที่นำเสนออยู่แล้ว และผลการจำลองการทำงานและการทดสอบการทำงานของวงจรบนชิปเอฟพีจีเอก็สามารถยืนยันได้ว่าวงจรที่ออกแบบสามารถสังเคราะห์สัญญาณได้มีคุณภาพสูงกว่าวงจรสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำในบทที่ 3 และค่าระดับฮาร์โมนิกที่สูงที่สุดมีค่าต่ำกว่า –64.87 เดซิเบล

คุณสมบัติของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้การประมาณค่าโพลีโนเมียลที่นำเสนอเปรียบเทียบกับวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำสามารถแสดงได้ดังตารางที่ 5.1

ตารางที่ 5.1 คุณสมบัติของวงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้การประมาณค่าโพลีโนเมียลที่นำเสนอเปรียบเทียบกับแบบที่ใช้หน่วยความจำ

	แบบประมาณค่าโพลีโนเมียล	แบบใช้หน่วยความจำ
Frequency resolution	32 bits	32 bits
Phase resolution	12 bits	12 bits
Amplitude resolution	10 bits	10 bits
Max frequency switching speed	6 clock cycles	19 clock cycles
Worst-case spurious	-64.87 dBc	-45 dBc

วงจรเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้การประมาณค่าโพลีโนเมียลในวิทยานิพนธ์ฉบับนี้ถูกออกแบบด้วยวิธีบนลงล่าง ซึ่งเป็นการใช้ภาษาวีเอชดีแอล (VHDL) บรรยายพฤติกรรมการทำงานของฮาร์ดแวร์ ซึ่งข้อดีของการออกแบบด้วยภาษาวีเอชดีแอล ก็คือวงจรที่ออกแบบได้สามารถนำไปสังเคราะห์ (Synthesis) และจัดวาง-เชื่อมโยง (Place-and-route) ให้ตรงกับเทคโนโลยีที่ต้องการใช้ในการเจือสารชิปวงจรรวมได้โดยไม่ต้องทำการออกแบบใหม่

สำหรับกรณีที่ต้องการพัฒนาให้ความเร็วในการทำงานของวงจรสูงขึ้นเราก็สามารถนำวงจรคูณหรือวงจรกำลังสองที่มีความเร็วสูงมาใช้ หรืออาจจะนำเทคนิคการทำงานแบบสายท่อ (Pipelining technique) มาใช้ด้วยก็ได้

เอกสารอ้างอิง

[1] J. Tierney, C. M. Rader and B. Gold. "A Digital Frequency Synthesizer." IEEE Trans. Audio Electroacoust., vol. AU-19, Mar. 1971. pp. 48-57.

[2] L. Wanhammar. **DSP Integrated Circuits.** San Diego: Academic Press. 1999.

[3] F. Lu, H. Samueli, J. Yuan and C. Svensson. "A 700-MHz 24-b Pipelined Accumulator in 1.2-μm CMOS for Application as a Numerically Controlled Oscillator." IEEE J. Solid-State Circuits., vol. 28, no. 8, Aug. 1993. pp. 878-886.

[4] R. de J. Romero-Troncoso and G. Espinosa-Flores-Verdad. "Phase accumulator synthesis algorithm for DDS applications." Electronics Letters., vol. 35, no. 10, May 1999. pp. 770-771.

[5] B. Goldberg. **Digital Techniques in Frequency Synthesis.** New York: McGraw-Hill. 1996.

[6] J. Vankka. "Methods of Mapping from Phase to Sine Amplitude in Direct Digital Synthesis." IEEE Trans. Ultrasonics, Ferroelectrics and Freq. Control., vol. 44, no. 2, Mar. 1997. pp. 526-534.

[7] V. F. Kroupa, Editors. **Direct Digital Frequency Synthesizers.** New York: IEEE Press. 1999.

[8] B. Razavi. **Principles of Data Conversion System Design.** New York: IEEE Press. 1995.

[9] Analog Devices. **A Technical Tutorial on Digital Signal Synthesis.** Analog Devices. 1999.

[10] L. P. Huelsman. **Active and Passive Analog Filter Design.** International Edition. Singapore: McGraw-Hill. 1993.

[11] J. Vankka. "Direct Digital Synthesizers: Theory, Design and Applications." Ph.D. Thesis of Helsinki University of Technology. 2000.

[12] H. T. Nicholas, H. Samueli and B. Kim. "The Optimization of Direct Digital Frequency Synthesizer Performance in the Presence of Finite Word Length Effects." Proc. 42nd Annu. Freq. Contr. Symp., 1988. pp. 357-363.

[13] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson and C. R. Cole. "CMOS/SOS Frequency Synthesizer LSI Circuit for Spread Spectrum Communications." IEEE J. Solid-State Circuits., vol. SC-19, no. 4, Aug. 1984. pp. 497-505.

[14] J. E. Volder. "The CORDIC Trigonometric Computing Technique." IRE Trans. Electron. Comput., vol. EC-8, no. 3, Sept. 1959. pp. 330-334.

[15] G. C. Gielis, R. van de Plassche and J. van Valburg. "A 540-MHz 10-b Polar-to-Cartesian Converter." IEEE J. Solid-State Circuits., vol. 26, no. 11, Nov. 1991. pp. 1645-1650.

[16] K. Palomaki, J. Niittylahti and M. Renfors. "Numerical Sine and Cosine Synthesis Using a Complex Multiplier." Proc. IEEE-ISCAS, 1999., vol. 4 pp. 356-359.

[17] K. Palomaki. "A Digital Sinusoidal Signal Synthesizer Based on Feedback." M.Sc. Thesis of Tampere University of Technology. 1999.

[18] A. M. Sodagar and G. R. Lahiji. "A Pipelined ROM-Less Architecture for Sine-Output Direct Digital Synthesizers Using the Second-Order Parabolic Approximation." IEEE Trans. Circuits Syst. II., vol. 48, no. 9, Sept. 2001 pp. 850-857.

[19] A. Thanachayanont, C. Meenakarn, T. Thongphuak and W. Sangnak. "Design and Implementation of a 100-MHz CMOS Direct Digital Synthesiser with a 10-Bit On-Chip Digital-to-Analog Converter." Proc. ISIC-2001, Singapore, Sept. 2001. pp. 55-58.

[20] C. Meenakarn and A. Thanachayanont. "100-MHz CMOS Direct Digital Synthesizer with 10-Bit DAC." Proc. IEEE-APCCAS, Singapore, Sept. 2002. pp. 385-388.

[21] J. Vankka, M. Waltari, M. Kosunen and K. A. I. Halonen. "A Direct Digital Synthesizer with and On-Chip D/A-Converter." IEEE J. Solid-State Circuits., vol. 33, no. 2, Feb. 1998. pp. 218-227.

[22] D. J. Smith. **HDL Chip Design.** Madison: Doone Publications. 1998.

[23] T.-Y. Wu, C.-T. Jih, J.-C Chen and C.-Y Wu. "A Low Glitch 10-Bit 75-MHz CMOS Video D/A Converter." IEEE J. Solid-State Circuits., vol. 30, no. 5, Oct. 1989. pp. 68-72.

[24] C. Meenakarn and A. Thanachayanont. "A Sine-Output ROM-Less Direct Digital Frequency Synthesiser Using a Polynomial Approximation." to be appeared in Proc. IEEE-ISCAS, Bangkok THAILAND, May 2003.

[25] C. Meenakarn and A. Thanachayanont. "A ROM-Less Direct Digital Frequency Synthesiser Using a Polynomial Approximation." to be appeared in Proc. VLSI-TSA, Hsinchu TAIWAN, April 2003.

[26] M. Abramowitz and I. A. Stegun. **Handbook of Mathematical Functions.** New York: Dover Publications. 1970.

[27] S. Shah, A. J. Al-khalili and D. Al-khalili. "Comparison of 32-Bit Multipliers for Various Performance Measures." Proc. Int. Conf. on Microelectronics, Tehran, Nov. 2000. pp. 75-80.

[28] J.-T Yoo, K. F. Smith and G. Gopalakrishnan. "A Fast Parallel Squarer Based on Divide-and-Conquer." IEEE J. Solid-State Circuits., vol. 32, no. 6, June 1997. pp. 909-912.

[29] Y. B. Mahdy, S. A. Ali, K. M. Shaaban. "A Fast Scheme and Implementation for N-Bit Squarer." Proc. ICECS'99, vol. 1 pp. 25-28.

ภาคผนวก

ภาคผนวก ก.

โค้ดวิเอชดีแอลของวงจรรวมสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

1. DDS Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY DDS IS
port(   NRESET  : in    std_logic;  -- Active low reset
        MCLK            : in    std_logic;  -- Master clock
        SELCTRL : in    std_logic;  -- Select control (0 = Pallarel, 1 = serial)
        -- Pallarel control
        A               : in    std_logic_vector(2 downto 0);   -- Address
        RWN             : in    std_logic;              -- Rising edge write signal
        DATA            : in    std_logic_vector(7 downto 0);   -- Byte input
        -- Serial control
        SCLK            : in    std_logic;  -- Serial
        RX              : in    std_logic;  -- Serial Rx
        -- Test
        EXT_TSTEN       : in    std_logic;
        EXT_PWRDN       : in    std_logic;
        TST_BIN : in    std_logic_vector(9 downto 0);
        -- DAC output
        DDS_OUT         : out   std_logic_vector(9 downto 0);
        PHASE_OV        : out   std_logic;
        DAC_OUT         : out   std_logic
        );

END DDS;

ARCHITECTURE RTL OF DDS IS

component WAVE_CTRL
PORT(   ------------------------------------------------
        -- Wave generator
        ------------------------------------------------
        --      DDS_NRST        : in    std_logic;  -- System reset (before start pipeline)
        --      NRST            : in    std_logic_vector(19 downto 1);  -- Pipeline reset
                CLOCK           : in    std_logic;  -- Global clock
        --      WTYPE           : in    std_logic_vector( 1 downto 0);   -- wave type
        --      STARTQ  : in    std_logic_vector( 1 downto 0);   -- start quadrant
        --      INCREMENT       : in    std_logic_vector(31 downto 0);  -- phase increment
                                                        -- step
        -- Coarse rom interface
        CFADRH  : out   std_logic_vector(3 downto 0);
        CADRL           : out   std_logic_vector(3 downto 0);
        CDIN            : in    std_logic_vector(7 downto 0);
        -- Fine rom interface
        FADRL           : out   std_logic_vector(3 downto 0);
        FDIN            : in    std_logic_vector(1 downto 0);
        -- CTRL
        --              CTRL_BIN        : in    std_logic;
        --              DAC_BIN         : in    std_logic_vector(9 downto 0);
        --              DDSOUT_EN       : in    std_logic;
        -- TEST
        --      TSTEN           : in    std_logic;              -- Test enable
        XWAVE   : in    std_logic_vector(9 downto 0);
        -- decoder
        code            : out   std_logic_vector(63 downto 1);
        b               : out   std_logic_vector( 3 downto 0);
        -- Wave output
        WAVE_EXT        : out   std_logic_vector(9 downto 0);
```

```
        ------------------------------------------------
        -- Control unit
        ------------------------------------------------
        NRESET  : in        std_logic;  -- Active low reset
        MCLK                : in        std_logic;  -- MCLK
        SELCTRL : in        std_logic;  -- Select control (0 = Pallarel, 1 = serial)
        A                   : in        std_logic_vector(2 downto 0);   -- Address
        RWN                 : in        std_logic;  -- Rising edge write signal
        DATA                : in        std_logic_vector(7 downto 0);   -- Byte input
        -- Uart override control
        UA                  : in        std_logic_vector(2 downto 0);   -- Address
        URWN                : in        std_logic;  -- Rising edge write signal
        UDATA               : in        std_logic_vector(7 downto 0);   -- Byte input
        -- Control in
        EXT_TSTEN           : in        std_logic;  -- External Test enable
        EXT_PWRDN           : in        std_logic;  -- External DAC powerdown
        -- Control out
        DDS_NRST            : out       std_logic; -- internal
--      DDS_ENABLE          : out       std_logic;
--      DDS_STOP            : out       std_logic;
--      WTYPE               : out       std_logic_vector(1 downto 0);
--      STARTQ  : out       std_logic_vector(1 downto 0);
--      INCREMENT           : out       std_logic_vector(31 downto 0);
        PHASE_OV            : out       std_logic;
        ROMCSN : out        std_logic;
        CLKOUT  : out       std_logic;
        CLKLATCH            : out       std_logic;
-- Test DAC
        PWRDN   : out       std_logic
        --CTRL_BIN                      : out       std_logic; -- internal
        --DDSOUT_EN                     : out       std_logic; -- internal
        --DAC_BIN                                   : out       std_logic_vector(9 downto 0) -- internal
        );
end component;

component COARSE_ROM256X8
port(   A           : in        std_logic_vector(7 downto 0);
        CK          : in        std_logic;
        CSN         : in        std_logic;
        --OEN       : in        std_logic;
        Q           : out       std_logic_vector(7 downto 0)
        );
end component;

component FINE_ROM256X2
port(   A           : in        std_logic_vector(7 downto 0);
        CK          : in        std_logic;
        CSN         : in        std_logic;
        --OEN       : in        std_logic;
        Q           : out       std_logic_vector(1 downto 0)
        );
end component;
component CODE_LATCH IS
PORT(   NRESET  : in        std_logic;
        CLOCK               : in        std_logic;
        cin                 : in        std_logic_vector(63 downto 1);
        bin                 : in        std_logic_vector(3 downto 0);
        -- Wave output
        code                : out       std_logic_vector(63 downto 1);
        b                   : out       std_logic_vector( 3 downto 0)
        );
end component;

component DAC
PORT(   PWRDN   : in        std_logic;
        code                : in        std_logic_vector(63 downto 1);
        b                   : in        std_logic_vector( 3 downto 0);
        DAC_OUT             : out       std_logic
        );
end component;
```

```
component UART_INTF
PORT(   NRESET  : in        std_logic;
        SCLK                : in        std_logic;
        RX                  : in        std_logic;
        -- Control
        UA                  : out       std_logic_vector(2 downto 0);   -- Address
        URWN                : out       std_logic;  -- Rising edge write signal
        UDATA               : out       std_logic_vector(7 downto 0)    -- Byte input
        );
end component;

        signal  UA                          : std_logic_vector(2 downto 0);  -- Address
        signal  URWN                        : std_logic;                                    -- Rising edge
write signal
        signal  UDATA                       : std_logic_vector(7 downto 0);  -- Byte input

        signal  INRESET             : std_logic;
--      signal  DDS_ENABLE                  : std_logic;
--      signal  DDS_STOP                    : std_logic;
        -- Test DAC
        signal  DAC_PWRDN                   : std_logic;

        signal  CLOCK                       : std_logic;
        signal  CLOCK_LATCH         : std_logic;
--      signal  NRST                        : std_logic_vector(20 downto 1);

        -- Coarse rom interface
        signal  CFADRH              : std_logic_vector(3 downto 0);
        signal  CADR                        : std_logic_vector(7 downto 0);
        signal  CADRL                       : std_logic_vector(3 downto 0);
        signal  CDIN                        : std_logic_vector(7 downto 0);
        -- Fine rom interface
        signal  FADR                        : std_logic_vector(7 downto 0);
        signal  FADRL                       : std_logic_vector(3 downto 0);
        signal  FDIN                        : std_logic_vector(1 downto 0);
        signal  ROMCSN              : std_logic;
        -- Wave output

        signal  DEC_CODE                    : std_logic_vector(63 downto 1);
        signal  DEC_BIN             : std_logic_vector( 3 downto 0);

        -- Wave output
        signal  CODE                        : std_logic_vector(63 downto 1);
        signal  BIN                         : std_logic_vector( 3 downto 0);

BEGIN

        WAVE_CONTROL: WAVE_CTRL
        PORT MAP (
                NRESET  =>NRESET,
                MCLK            =>MCLK,
                SELCTRL =>SELCTRL,
                A               =>A,
                RWN             =>RWN,
                DATA            =>DATA,
                -- Uart override control
                UA              =>UA,
                URWN            =>URWN,
                UDATA           =>UDATA,
                -- Control in
                EXT_TSTEN       =>EXT_TSTEN,
                EXT_PWRDN       =>EXT_PWRDN,
                -- Control out
                DDS_NRST        =>INRESET,
--              DDS_ENABLE      =>DDS_ENABLE,
--              DDS_STOP        =>DDS_STOP,
--              WTYPE           =>WTYPE,
--              STARTQ  =>STARTQ,
--              INCREMENT       =>INCREMENT,
```

```
                PHASE_OV        =>PHASE_OV,
                ROMCSN =>ROMCSN,
                CLKOUT =>CLOCK,
                CLKLATCH        =>CLOCK_LATCH,
                -- Test DAC
                PWRDN  =>DAC_PWRDN,
--              CTRL_BIN        =>CTRL_BIN,
--              DDSOUT_EN       =>DDSOUT_EN,
--              DAC_BIN=>DAC_BIN

                --DDS_NRST       =>INRESET, -- CTRL
--              NRST            =>NRST(19 downto 1), -- RESET
                CLOCK           =>CLOCK, -- RESET
                --WTYPE =>WTYPE, -- CTRL
                --STARTQ         =>STARTQ, -- CTRL
                --INCREMENT      =>INCREMENT, -- CTRL
                -- Coarse rom interface
                CFADRH =>CFADRH,
                CADRL           =>CADRL,
                CDIN            =>CDIN,
                -- Fine rom interface
                FADRL           =>FADRL,
                FDIN            =>FDIN,
                -- Wave output
                --TSTEN  =>EXT_TSTEN, -- CTRL
                XWAVE  =>TST_BIN, -- EXT
                -- Wave output
                code            =>DEC_CODE,
                b               =>DEC_BIN,
                -- CTRL
                --CTRL_BIN       =>CTRL_BIN, -- CTRL
                --DAC_BIN        =>DAC_BIN, -- CTRL
                --DDSOUT_EN      =>DDSOUT_EN, -- CTRL
                -- Wave output
                WAVE_EXT        =>DDS_OUT
        );

        CADR <= CFADRH & CADRL;

        COARSE_ROM: COARSE_ROM256X8
        port map (
                A               =>CADR,
                CK              =>CLOCK,
                CSN             =>ROMCSN,
                --OEN            =>'0',
                Q               =>CDIN
        );

        FADR <= CFADRH & FADRL;

        FINE_ROM: FINE_ROM256X2
        port map (
                A               =>FADR,
                CK              =>CLOCK,
                CSN             =>ROMCSN,
                --OEN            =>'0',
                Q               =>FDIN
        );

        LATCH: CODE_LATCH
        PORT MAP (
                NRESET =>INRESET,
                CLOCK           =>CLOCK_LATCH,
                cin             =>DEC_CODE,
                bin             =>DEC_BIN,
                -- Wave output
                code            =>CODE,
                b               =>BIN
        );
```

```
DAC_UNIT: DAC
PORT MAP (
        PWRDN   =>DAC_PWRDN,
        code            =>CODE,
        b               =>BIN,
        DAC_OUT         =>DAC_OUT
);

UART_UNIT: UART_INTF
PORT MAP (
        NRESET  =>NRESET,
        SCLK            =>SCLK,
        RX              =>RX,
        -- Control
        UA              =>UA,
        URWN            =>URWN,
        UDATA           =>UDATA
);

END RTL;
```

2. WAVE_CTRL Module

```
------
--- W = Word size = 12
--- D = Output word = 9
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY WAVE_CTRL IS
PORT(   ------------------------------------------------
        -- Wave generator
        ------------------------------------------------
--      DDS_NRST        : in    std_logic;              -- System reset (before start pipeline)
--      NRST            : in    std_logic_vector(19 downto 1);  -- Pipeline reset
        CLOCK           : in    std_logic;              -- Global clock
--      WTYPE           : in    std_logic_vector( 1 downto 0);  -- wave type
--      STARTQ          : in    std_logic_vector( 1 downto 0);  -- start quadrant
--      INCREMENT       : in    std_logic_vector(31 downto 0);  -- phase increment step
-- Coarse rom interface
        CFADRH          : out   std_logic_vector(3 downto 0);
        CADRL           : out   std_logic_vector(3 downto 0);
        CDIN            : in    std_logic_vector(7 downto 0);
-- Fine rom interface
        FADRL           : out   std_logic_vector(3 downto 0);
        FDIN            : in    std_logic_vector(1 downto 0);
-- CTRL
--      CTRL_BIN        : in    std_logic;
--      DAC_BIN         : in    std_logic_vector(9 downto 0);
--      DDSOUT_EN       : in    std_logic;
-- TEST
--      TSTEN           : in    std_logic;              -- Test enable
        XWAVE           : in    std_logic_vector(9 downto 0);
-- decoder
        code            : out   std_logic_vector(63 downto 1);
        b               : out   std_logic_vector( 3 downto 0);
-- Wave output
        PHASE_OV        : out   std_logic;
        WAVE_EXT        : out   std_logic_vector(9 downto 0);
------------------------------------------------
-- Control unit
------------------------------------------------
        NRESET          : in    std_logic;  -- Active low reset
        MCLK            : in    std_logic;  -- MCLK
        SELCTRL         : in    std_logic;  -- Select control (0 = Pallarel, 1 = serial)
        A               : in    std_logic_vector(2 downto 0);   -- Address
        RWN             : in    std_logic;                      -- Rising edge write signal
        DATA            : in    std_logic_vector(7 downto 0);   -- Byte input
-- Uart override control
        UA              : in    std_logic_vector(2 downto 0);   -- Address
        URWN            : in    std_logic;                      -- Rising edge write signal
        UDATA           : in    std_logic_vector(7 downto 0);   -- Byte input
-- Control in
        EXT_TSTEN       : in    std_logic;                      -- External Test enable
        EXT_PWRDN       : in    std_logic;                      -- External DAC powerdown
-- Control out
        DDS_NRST        : out   std_logic; -- internal
--      DDS_ENABLE      : out   std_logic;
--      DDS_STOP        : out   std_logic;
--      WTYPE           : out   std_logic_vector(1 downto 0);
--      STARTQ          : out   std_logic_vector(1 downto 0);
--      INCREMENT       : out   std_logic_vector(31 downto 0);
        ROMCSN          : out   std_logic;
        CLKOUT          : out   std_logic;
        CLKLATCH        : out   std_logic;
-- Test DAC
        PWRDN           : out   std_logic
--      CTRL_BIN        : out   std_logic; -- internal
--      DDSOUT_EN       : out   std_logic; -- internal
--      DAC_BIN         : out   std_logic_vector(9 downto 0) -- internal
```

```
);
END WAVE_CTRL;

ARCHITECTURE RTL OF WAVE_CTRL IS

component CTRL_INTF
port(     NRESET   : in     std_logic;  -- Active low reset
          MCLK     : in     std_logic;  -- Master clock
          SELCTRL  : in     std_logic;  -- Select control (0 = Pallarel, 1 = serial)
          A        : in     std_logic_vector(2 downto 0);   -- Address
          RWN      : in     std_logic;  -- Rising edge write signal
          DATA     : in     std_logic_vector(7 downto 0);   -- Byte input
          -- Uart override control
          UA       : in     std_logic_vector(2 downto 0);   -- Address
          URWN     : in     std_logic;              -- Rising edge write signal
          UDATA    : in     std_logic_vector(7 downto 0);   -- Byte input
          -- Control in
          EXT_TSTEN     : in     std_logic;              -- External Test enable
          EXT_PWRDN     : in     std_logic;              -- External DAC powerdown
          -- Control out
          DDS_NRST      : out    std_logic;
          DDS_ENABLE    : out    std_logic;
          DDS_STOP      : out    std_logic;
          WTYPE         : out    std_logic_vector(1 downto 0);
          STARTQ        : out    std_logic_vector(1 downto 0);
          INCREMENT     : out    std_logic_vector(31 downto 0);
          ROMCSN        : out    std_logic;
          -- Test DAC
          PWRDN         : out    std_logic;
          CTRL_BIN      : out    std_logic;
          DDSOUT_EN     : out    std_logic;
          DAC_BIN       : out    std_logic_vector(9 downto 0);
          PHASE_MOD     : out    std_logic_vector(11 downto 0)
          );
end component;

component WAVE_GEN
PORT(     DDS_NRST      : in     std_logic;              -- System reset (before start pipeline)
          NRST          : in     std_logic_vector(19 downto 1);  -- Pipeline reset
          CLOCK         : in     std_logic;              -- Global clock
          WTYPE         : in     std_logic_vector( 1 downto 0);  -- wave type
          STARTQ        : in     std_logic_vector( 1 downto 0);  -- start quadrant
          INCREMENT     : in     std_logic_vector(31 downto 0);  -- phase increment step
          PHASE_MOD     : in     std_logic_vector(11 downto 0);
          -- Coarse rom interface
          CFADRH        : out    std_logic_vector(3 downto 0);
          CADRL         : out    std_logic_vector(3 downto 0);
          CDIN          : in     std_logic_vector(7 downto 0);
          -- Fine rom interface
          FADRL         : out    std_logic_vector(3 downto 0);
          FDIN          : in     std_logic_vector(1 downto 0);
          -- CTRL
          CTRL_BIN      : in     std_logic;
          DAC_BIN       : in     std_logic_vector(9 downto 0);
          DDSOUT_EN     : in     std_logic;
          -- TEST
          TSTEN         : in     std_logic;              -- Test enable
          XWAVE         : in     std_logic_vector(9 downto 0);
          -- decoder
          code          : out    std_logic_vector(63 downto 1);
          b             : out    std_logic_vector( 3 downto 0);
          -- Wave output
          PHASE_OV      : out    std_logic;
          WAVE_EXT      : out    std_logic_vector(9 downto 0)
          );
end component;

component RESET_STAGE
port(     NRESET        : in     std_logic;
          EXT_TSTEN     : in     std_logic;
```

```
        CLKIN           : in    std_logic;
        ENABLE          : in    std_logic;
        STOP            : in    std_logic;
        CLKOUT          : out   std_logic;
        CLKLATCH        : out   std_logic;
        NRST            : out   std_logic_vector(20 downto 1)
        );
end component;
        signal  NRST                    : std_logic_vector(20 downto 1);
        signal  DDS_ENABLE              : std_logic;
        signal  DDS_STOP                : std_logic;

        signal  INRESET                 : std_logic;
        signal  WTYPE                   : std_logic_vector(1 downto 0);
        signal  STARTQ                  : std_logic_vector(1 downto 0);
        signal  INCREMENT               : std_logic_vector(31 downto 0);
        -- Test DAC
        signal  CTRL_BIN                : std_logic;
        signal  DAC_BIN                 : std_logic_vector(9 downto 0);
        signal  DDSOUT_EN               : std_logic;
        signal  PHASE_MOD               : std_logic_vector(11 downto 0);

BEGIN

        CTRL_UNIT: CTRL_INTF
        port map (
                NRESET                  =>NRESET,
                MCLK                    =>MCLK,
                SELCTRL                 =>SELCTRL,
                A                       =>A,
                RWN                     =>RWN,
                DATA                    =>DATA,
                -- Uart override control
                UA                      =>UA,
                URWN                    =>URWN,
                UDATA                   =>UDATA,
                -- Control in
                EXT_TSTEN               =>EXT_TSTEN,
                EXT_PWRDN               =>EXT_PWRDN,
                -- Control out
                DDS_NRST                =>INRESET,
                DDS_ENABLE              =>DDS_ENABLE,
                DDS_STOP                =>DDS_STOP,
                WTYPE                   =>WTYPE,
                STARTQ                  =>STARTQ,
                INCREMENT               =>INCREMENT,
                ROMCSN                  =>ROMCSN,
                -- Test DAC
                PWRDN                   =>PWRDN,
                CTRL_BIN                =>CTRL_BIN,
                DDSOUT_EN               =>DDSOUT_EN,
                DAC_BIN                 =>DAC_BIN,
                PHASE_MOD               =>PHASE_MOD
        );

        DDS_NRST <= INRESET;

        WAVE_UNIT: WAVE_GEN
        PORT MAP (
                DDS_NRST                =>INRESET, -- CTRL
                NRST                    =>NRST(19 downto 1), -- RESET
                CLOCK                   =>CLOCK, -- RESET
                WTYPE                   =>WTYPE, -- CTRL
                STARTQ                  =>STARTQ, -- CTRL
                INCREMENT               =>INCREMENT, -- CTRL
                PHASE_MOD               =>PHASE_MOD,
                -- Coarse rom interface
                CFADRH                  =>CFADRH,
                CADRL                   =>CADRL,
                CDIN                    =>CDIN,
```

```
            -- Fine rom interface
            FADRL                   =>FADRL,
            FDIN                    =>FDIN,
            -- Wave output
            TSTEN                   =>EXT_TSTEN, -- CTRL
            XWAVE                   =>XWAVE, -- EXT
            -- Wave output
            code                    =>CODE,
            b                       =>B,
            -- CTRL
            CTRL_BIN                =>CTRL_BIN, -- CTRL
            DAC_BIN                 =>DAC_BIN, -- CTRL
            DDSOUT_EN               =>DDSOUT_EN, -- CTRL
            -- Wave output
            PHASE_OV                =>PHASE_OV,
            WAVE_EXT                =>WAVE_EXT
    );

    RESET_UNIT: RESET_STAGE
    port map (
            NRESET          =>INRESET,
            EXT_TSTEN       =>EXT_TSTEN,
            CLKIN           =>MCLK,
            ENABLE          =>DDS_ENABLE,
            STOP            =>DDS_STOP,
            CLKOUT          =>CLKOUT,
            CLKLATCH        =>CLKLATCH,
            NRST            =>NRST
    );

END RTL;
```

3. CTRL_INTF Module

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity CTRL_INTF is
port(   NRESET          : in    std_logic;  -- Active low reset
        MCLK            : in    std_logic;  -- MCLK
        SELCTRL         : in    std_logic;  -- Select control (0 = Pallarel, 1 = serial)
        A               : in    std_logic_vector(2 downto 0);   -- Address
        RWN             : in    std_logic;                      -- Rising edge write signal
        DATA            : in    std_logic_vector(7 downto 0);   -- Byte input
        -- Uart override control
        UA              : in    std_logic_vector(2 downto 0);   -- Address
        URWN            : in    std_logic;                      -- Rising edge write signal
        UDATA           : in    std_logic_vector(7 downto 0);   -- Byte input
        -- Control in
        EXT_TSTEN       : in    std_logic;                      -- External Test enable
        EXT_PWRDN       : in    std_logic;                      -- External DAC powerdown
        -- Control out
        DDS_NRST        : out   std_logic;
        DDS_ENABLE      : out   std_logic;
        DDS_STOP        : out   std_logic;
        WTYPE           : out   std_logic_vector(1 downto 0);
        STARTQ          : out   std_logic_vector(1 downto 0);
        INCREMENT       : out   std_logic_vector(31 downto 0);
        ROMCSN          : out   std_logic;
        -- Test DAC
        PWRDN           : out   std_logic;
        CTRL_BIN        : out   std_logic;
        DDSOUT_EN       : out   std_logic;
        DAC_BIN         : out   std_logic_vector(9 downto 0);
        PHASE_MOD       : out   std_logic_vector(11 downto 0)
        );
end CTRL_INTF;

architecture rtl of CTRL_INTF is

        constant  Inc0_adr      : integer := 1;
        constant  Inc1_adr      : integer := 2;
        constant  Inc2_adr      : integer := 3;
        constant  Inc3_adr      : integer := 4;
        constant  Ctrl_adr      : integer := 5;
        constant  Index_adr     : integer := 6;
        constant  Wdata_adr     : integer := 7;
        constant  DAC0_idx      : integer := 0;
        constant  DAC1_idx      : integer := 1;
        constant  PHASE0_idx    : integer := 2;
        constant  PHASE1_idx    : integer := 3;

        -- synopsys translate_off
        -- synopsys translate_on
        signal    RegCtrl  : std_logic_vector(7 downto 0);          -- ADDR 05h
                  -- Bit   : Name           [LOW -> HIGH]
                  ---------------------------------------------
                  -- 7     : NRST           [reset, not reset]
                  -- 6     : PWRDN          [DAC powerdown, DAC poweron]
                  -- 5     : ENABLE         [dds off, dds on]
                  -- 4     : STOP           [dds run, dds stop]
                  -- 3-2   : WTYPE          [SINE, RNG, RAMP, SAW]
                  -- 1-0   : STARTQ         [START QUADRANT 1,2,3,4]
        alias     Ctrl_NRST     : std_logic is RegCtrl(7);
        alias     Ctrl_PWRDN    : std_logic is RegCtrl(6);
        alias     Ctrl_ENABLE   : std_logic is RegCtrl(5);
        alias     Ctrl_STOP     : std_logic is RegCtrl(4);
        alias     Ctrl_WTYPE    : std_logic_vector(1 downto 0) is RegCtrl(3 downto 2);
        alias     Ctrl_STARTQ   : std_logic_vector(1 downto 0) is RegCtrl(1 downto 0);

        signal    RegINC0  : std_logic_vector(7 downto 0);  -- ADDR 01h
```

```
        signal      RegINC1  : std_logic_vector(7 downto 0);   -- ADDR 02h
        signal      RegINC2  : std_logic_vector(7 downto 0);   -- ADDR 03h
        signal      RegINC3  : std_logic_vector(7 downto 0);   -- ADDR 04h

        signal      RegINDEX          : std_logic_vector(1 downto 0);   -- ADDR 06h

        signal      RegDAC            : std_logic_vector(3 downto 0);   -- ADDR 00h
        alias       DAC_TSTEN         : std_logic is RegDAC(3);
        alias       DDS_WOUTEN        : std_logic is RegDAC(2);
                    -- Bit      : Name              [LOW -> HIGH]
                    ------------------------------------------------
                    -- 3        : TSTEN             [dds mode, dac mode]
                    -- 2        : WOUT_EN           [no wout, wout]
                    -- 1-0      : BINARY input [9:8]

        signal      RegDAC_BIN        : std_logic_vector(9 downto 0);              -- ADDR 01h (BINARY input [9:0]

        signal      RegPHASE1         : std_logic_vector(3 downto 0);              -- ADDR 02h
        signal      RegPHASE          : std_logic_vector(11 downto 0);  -- ADDR 03h

        signal      ADR               : std_logic_vector(2 downto 0);
        signal      RWCLK             : std_logic;
        signal      DIN               : std_logic_vector(7 downto 0);

begin

--                  SELCTRL           -- Select control (0 = Pallarel, 1 = serial)
--                  A                 -- Address
--                  RWN               -- Rising edge write signal
--                  DATA              -- Byte input
--                  -- Uart override control
--                  UA                -- Address
--                  URWN              -- Rising edge write signal
--                  UDATA             -- Byte input

        ADR     <= A      when SELCTRL = '0' else UA;
        RWCLK   <= RWN    when SELCTRL = '0' else URWN;
        DIN     <= DATA   when SELCTRL = '0' else UDATA;

        -----------------------------
        -- Write to Register --
        -----------------------------
        DATA_REG:
        process(NRESET, RWCLK)
        begin
                if NRESET = '0' then
                        RegINC0         <= (others=>'0');
                        RegINC1         <= (others=>'0');
                        RegINC2         <= (others=>'0');
                        RegINC3         <= (others=>'0');
                        RegCtrl         <= (others=>'0');
                        RegINDEX        <= (others=>'0');
                        RegDAC          <= (others=>'0');
                        RegDAC_BIN      <= (others=>'0');
                        RegPHASE1       <= (others=>'0');
                        RegPHASE        <= (others=>'0');
                elsif RWCLK'event and (RWCLK = '1') then
                        if to_integer(unsigned(ADR)) = Inc0_adr then
                                RegINC0  <= DIN;
                        end if;
                        if to_integer(unsigned(ADR)) = Inc1_adr then
                                RegINC1  <= DIN;
                        end if;
                        if to_integer(unsigned(ADR)) = Inc2_adr then
                                RegINC2  <= DIN;
                        end if;
                        if to_integer(unsigned(ADR)) = Inc3_adr then
                                RegINC3  <= DIN;
                        end if;
                        if to_integer(unsigned(ADR)) = Ctrl_adr then
```

```
                        RegCtrl     <= DIN;
                end if;
                if to_integer(unsigned(ADR)) = Index_adr then
                        RegINDEX <= DIN(1 downto 0);
                end if;

                if to_integer(unsigned(ADR)) = Wdata_adr then
                        case RegINDEX is
                                when "00" =>
                                        RegDAC    <= DIN(3 downto 0);
                                when "01" =>
                                        RegDAC_BIN <= RegDAC(1 downto 0) & DIN;
                                when "10" =>
                                        RegPHASE <= RegPHASE1 & DIN;
                                when others =>
                                        RegPHASE1 <= DIN(3 downto 0);
                        end case;
                end if;

        end if;
end process;

INCREMENT <= RegINC3 & RegINC2 & RegINC1 & RegINC0 ;

        -- 7     :      RST          [reset, not reset]
        -- 6     :      PWRDN        [DAC powerdown, DAC poweron]
        -- 5     :      ENABLE       [dds off, dds on]
        -- 4     :      STOP         [dds run, dds stop]
        -- 3-2   :      WTYPE        [SINE, RAMP, SAW, RNG]
        -- 1-0   :      STARTQ       [START QUADRANT 1,2,3,4]
DDS_NRST_BIT:
process(NRESET, MCLK)
begin
        if NRESET = '0' then
                DDS_NRST          <= '0';
        elsif MCLK'event and (MCLK = '1') then
                if EXT_TSTEN = '1' then
                        DDS_NRST <= NRESET;
                else
                        DDS_NRST <= (Ctrl_NRST    and NRESET);
                end if;
        end if;
end process;

DDS_ENABLE      <= Ctrl_ENABLE and (not EXT_TSTEN);
ROMCSN          <= not (Ctrl_ENABLE and (not EXT_TSTEN));

DDS_STOP        <= Ctrl_STOP;
WTYPE           <= Ctrl_WTYPE;
STARTQ          <= Ctrl_STARTQ;

PWRDN           <= Ctrl_PWRDN when EXT_TSTEN = '0' else EXT_PWRDN;

CTRL_BIN        <= DAC_TSTEN;
DAC_BIN         <= RegDAC_BIN;
PHASE_MOD       <= RegPHASE;

DDSOUT_EN       <= DDS_WOUTEN;

end rtl ;
```

4. WAVE_GEN Module

```
------
--- W = Word size = 12
--- D = Output word = 9
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY WAVE_GEN IS
PORT(   DDS_NRST        : in    std_logic;              -- System reset (before start pipeline)
        NRST            : in    std_logic_vector(19 downto 1);  -- Pipeline reset
        CLOCK           : in    std_logic;                      -- Global clock
        WTYPE           : in    std_logic_vector( 1 downto 0);  -- wave type
        STARTQ          : in    std_logic_vector( 1 downto 0);  -- start quadrant
        INCREMENT       : in    std_logic_vector(31 downto 0);  -- phase increment step
        PHASE_MOD       : in    std_logic_vector(11 downto 0);
        -- Coarse rom interface
        CFADRH          : out   std_logic_vector(3 downto 0);
        CADRL           : out   std_logic_vector(3 downto 0);
        CDIN            : in    std_logic_vector(7 downto 0);
        -- Fine rom interface
        FADRL           : out   std_logic_vector(3 downto 0);
        FDIN            : in    std_logic_vector(1 downto 0);
        -- CTRL
        CTRL_BIN        : in    std_logic;
        DAC_BIN         : in    std_logic_vector(9 downto 0);
        DDSOUT_EN       : in    std_logic;
        -- TEST
        TSTEN           : in    std_logic;                      -- Test enable
        XWAVE           : in    std_logic_vector(9 downto 0);
        -- decoder
        code            : out   std_logic_vector(63 downto 1);
        b                       : out   std_logic_vector( 3 downto 0);
        -- Wave output
        PHASE_OV        : out   std_logic;
        WAVE_EXT        : out   std_logic_vector(9 downto 0)
        );

END WAVE_GEN;

ARCHITECTURE RTL OF WAVE_GEN IS

component DAC_DECODER
PORT(   TSTEN           : in    std_logic;                      -- Test enable
        IWAVE           : in    std_logic_vector(9 downto 0);
        XWAVE           : in    std_logic_vector(9 downto 0);
        -- Wave output
        code            : out   std_logic_vector(63 downto 1);
        b               : out   std_logic_vector( 3 downto 0)
        );
end component;

component LFSR
generic(Width: positive := 8);          -- length of pseudo-random sequence
port (  clock           : in std_logic;
        nreset          : in std_logic;         -- active low reset
        load            : in std_logic;         -- active high load
        seed            : in std_logic_vector(Width-1 downto 0);   -- parallel seed input
        parallel_out    : out std_logic_vector(Width-1 downto 0); -- parallel data out
        serial_out      : out std_logic     -- serial data out (From last shift register)
        );
end component;

component ACC4
port(   NRESET          : in    std_logic;
        CLOCK           : in    std_logic;
        LOAD            : in    std_logic;
        CIN             : in    std_logic;
        DIN             : in    std_logic_vector( 3 downto 0);
```

```
            DOUT            : out    std_logic_vector( 3 downto 0);
            CO              : out    std_logic
            );
end component;

component ACCP4
port(       NRESET          : in     std_logic;
            CLOCK           : in     std_logic;
            CIN             : in     std_logic;
            DIN             : in     std_logic_vector( 3 downto 0);
            PDIN            : in     std_logic_vector( 3 downto 0);
            DOUT            : out    std_logic_vector( 3 downto 0);
            CO              : out    std_logic
            );
end component;

component ADDER4
port (      A               : in     std_logic_vector(3 downto 0);
            B               : in     std_logic_vector(3 downto 0);
            CIN             : in     std_logic;
            S               : out    std_logic_vector(3 downto 0);
            COUT            : out    std_logic
            );
end component;

component ADDER2
port (      A               : in     std_logic_vector(1 downto 0);
            B               : in     std_logic_vector(1 downto 0);
            CIN             : in     std_logic;
            S               : out    std_logic_vector(1 downto 0);
            COUT            : out    std_logic
            );
end component;

component INC2
port (      A               : in     std_logic_vector(1 downto 0);
            CIN             : in     std_logic;
            S               : out    std_logic_vector(1 downto 0);
            COUT            : out    std_logic
            );
end component;

component INC5
port (      A               : in     std_logic_vector(4 downto 0);
            CIN             : in     std_logic;
            S               : out    std_logic_vector(4 downto 0);
            COUT            : out    std_logic
            );
end component;

signal      ACC_CIN : std_logic_vector(8 downto 0);
signal      ACCR4, ACCR5, ACCR6, ACCR7 : std_logic_vector( 3 downto 0);
-- synopsys translate_off
signal      ACCR0, ACCR1, ACCR2, ACCR3 : std_logic_vector( 3 downto 0);
signal      ACC      : std_logic_vector(31 downto 0);
signal      ACCR0_1, ACCR0_2, ACCR0_3, ACCR0_4, ACCR0_5, ACCR0_6, ACCR0_7: std_logic_vector( 3 downto 0);
signal      ACCR1_1, ACCR1_2, ACCR1_3, ACCR1_4, ACCR1_5, ACCR1_6        : std_logic_vector( 3 downto 0);
signal      ACCR2_1, ACCR2_2, ACCR2_3, ACCR2_4, ACCR2_5                 : std_logic_vector( 3 downto 0);
signal      ACCR3_1, ACCR3_2, ACCR3_3, ACCR3_4                          : std_logic_vector( 3 downto 0);
signal      ACCR4_1, ACCR4_2, ACCR4_3                                   : std_logic_vector( 3 downto 0);

signal      ACCR4_D, ACCR4_L1,ACCR4_L2      : std_logic_vector( 1 downto 0);
signal      ACCR5_1, ACCR5_D                : std_logic_vector( 3 downto 0);
signal      ACCR6_D                         : std_logic_vector( 3 downto 0);

signal      S_ACC_CIN                               : std_logic_vector(8 downto 1);
signal      S_ACCR4, S_ACCR5, S_ACCR6, S_ACCR7      : std_logic_vector( 3 downto 0);
signal      S_ACCR0, S_ACCR1, S_ACCR2, S_ACCR3      : std_logic_vector( 3 downto 0);
```

```
signal   S_ACCR0_1, S_ACCR0_2, S_ACCR0_3, S_ACCR0_4, S_ACCR0_5, S_ACCR0_6, S_ACCR0_7 : std_logic_vector( 3
         downto 0);
signal   S_ACCR1_1, S_ACCR1_2, S_ACCR1_3, S_ACCR1_4, S_ACCR1_5, S_ACCR1_6: std_logic_vector( 3 downto 0);
signal   S_ACCR2_1, S_ACCR2_2, S_ACCR2_3, S_ACCR2_4, S_ACCR2_5          : std_logic_vector( 3 downto 0);
signal   S_ACCR3_1, S_ACCR3_2, S_ACCR3_3, S_ACCR3_4                     : std_logic_vector( 3 downto 0);
signal   S_ACCR4_1, S_ACCR4_2, S_ACCR4_3                                : std_logic_vector( 3 downto 0);

signal   S_ACCR4_D, S_ACCR4_L1,  S_ACCR4_L2      : std_logic_vector( 1 downto 0);
signal   S_ACCR5_1, S_ACCR5_D               : std_logic_vector( 3 downto 0);
signal   S_ACCR6_D                          : std_logic_vector( 3 downto 0);

         signal   ACC_TEST        : std_logic_vector(31 downto 0);
         signal   SUM_TEST        : std_logic_vector(32 downto 0);
         signal   ERR_TEST        : std_logic;
         signal   ROM_TEST        : std_logic_vector(8 downto 0);
         signal   ROM_ERR         : std_logic;
-- synopsys translate_on

         signal   ADIN0, ADIN1, ADIN2, ADIN3, ADIN4, ADIN5, ADIN6, ADIN7 : std_logic_vector( 3 downto 0);

         signal   ACIN    : std_logic_vector(8 downto 0);

         signal   PHASE   : std_logic_vector(13 downto 0);
         alias    WAVE_LIN: std_logic_vector(9 downto 0) is PHASE(13 downto 4);

         signal   PCIN            : std_logic_vector(4 downto 0);
         signal   PHASE_MOD0      : std_logic_vector(3 downto 0);
         signal   PHASE_MOD1      : std_logic_vector(3 downto 0);
         signal   PHASE_MOD2      : std_logic_vector(3 downto 0);
         signal   PHASE_MOD3      : std_logic_vector(3 downto 0);
         signal   ADDPR4          : std_logic_vector(3 downto 0);
         signal   ADDPR5          : std_logic_vector(3 downto 0);
         signal   ADDPR6          : std_logic_vector(3 downto 0);
         signal   ADDPR7          : std_logic_vector(3 downto 0);
         signal   ADDPR6_D        : std_logic_vector(3 downto 0);
         signal   ADDPR5_1        : std_logic_vector(3 downto 0);
         signal   ADDPR5_D        : std_logic_vector(3 downto 0);
-- synopsys translate_off
         signal   ADDPR4_1        : std_logic_vector(3 downto 0);
         signal   ADDPR4_2        : std_logic_vector(3 downto 0);
         signal   ADDPR4_D        : std_logic_vector(3 downto 0);
-- synopsys translate_on
         signal   ADDPR4_L1       : std_logic_vector(1 downto 0);
         signal   ADDPR4_L2       : std_logic_vector(1 downto 0);
         signal   ADDPR4_LD       : std_logic_vector(1 downto 0);

         signal   PHASE_OUT       : std_logic_vector(13 downto 0);

         signal   ROM_ADR13       : std_logic;
         signal   ROM_ADR         : std_logic_vector(11 downto 0);
         alias    WAVE_TRI: std_logic_vector(9 downto 0) is ROM_ADR(11 downto 2);

         signal   Q_1, Q_2, Q_3, Q_4  : std_logic;

         signal   MSB_OUT         : std_logic;
--       signal   COARSE_OUT      : std_logic_vector(7 downto 0);
         signal   COARSE_BUF      : std_logic_vector(8 downto 0);
--       signal   FINE_OUT        : std_logic_vector(1 downto 0);
         signal   FINE_BUF        : std_logic_vector(1 downto 0);
         signal   COARSE_H        : std_logic_vector(4 downto 0);

         signal   ROM_SUM         : std_logic_vector(8 downto 0);
         signal   RSUM0   : std_logic_vector(3 downto 0);
--       signal   RSUM1   : std_logic_vector(4 downto 0);
         signal   RSUM0_CIN, RSUM0_CO, RSUM1_CIN  : std_logic;
         signal   ROM_SUM0        : std_logic_vector(3 downto 0);
         signal   ROM_SUM1        : std_logic_vector(4 downto 0);

         signal   SINE_OUT        : std_logic_vector(9 downto 0);
```

```
        signal    WAVE_IN         : std_logic_vector(9 downto 0);
        signal    WAVE_I0 : std_logic_vector(9 downto 0);
        signal    WAVE_I1 : std_logic_vector(9 downto 0);
        signal    WAVE_OUT        : std_logic_vector(9 downto 0);

        signal    LOAD_ACC        : std_logic;
        signal    PHASE_RUN       : std_logic_vector(7 downto 1);
        signal    RNG, RNG_BUF: std_logic_vector(9 downto 0);
        signal    SAVE_RNG, LOAD_RNG      : std_logic;

-- synopsys translate_off
        signal    IN_FRQ : std_logic_vector(31 downto 0);
-- synopsys translate_on
        signal    INC_FRQ : std_logic_vector(31 downto 0);

        signal    IN_PHASE : std_logic_vector(11 downto 0);
        subtype BYTE is std_logic_vector(3 downto 0);
        Type BYTE4 is array (3 downto 0) of BYTE;
        Type BYTE8 is array (7 downto 0) of BYTE;

        signal    PINC, PDIN_INC : BYTE8;
        signal    IL1S : BYTE8;
        signal    IL1C : std_logic_vector(7 downto 0);
        signal    IL2S : BYTE8;
        signal    IL2C : std_logic_vector(7 downto 0);
        signal    IL3S : BYTE8;
        signal    IL3C : std_logic_vector(7 downto 0);
        signal    IL4S : BYTE8;
        signal    IL4C : std_logic_vector(7 downto 0);
        signal    IL5S : BYTE8;
        signal    IL5C : std_logic_vector(7 downto 0);
        signal    IL6S : BYTE8;
        signal    IL6C : std_logic_vector(7 downto 0);
        signal    IL7S : BYTE8;
        signal    IL7C : std_logic_vector(7 downto 0);
        signal    IL8S : BYTE8;
--      signal    IL8C : std_logic_vector(7 downto 0);

        signal    DACCR4_0        : std_logic_vector(3 downto 0);
        signal    DACCR4_1        : std_logic_vector(3 downto 0);
        signal    DACCR4_2        : std_logic_vector(3 downto 0);
        signal    DACCR4          : std_logic_vector(3 downto 0);

        signal    LOAD_ACC5       : std_logic;
        signal    DACCR5_0        : std_logic_vector(3 downto 0);
        signal    DACCR5_1        : std_logic_vector(3 downto 0);
        signal    DACCR5_2        : std_logic_vector(3 downto 0);
        signal    DACCR5          : std_logic_vector(3 downto 0);

        signal    LOAD_ACC6       : std_logic;
        signal    DACCR6_0        : std_logic_vector(3 downto 0);
        signal    DACCR6_1        : std_logic_vector(3 downto 0);
        signal    DACCR6_2        : std_logic_vector(3 downto 0);
        signal    DACCR6          : std_logic_vector(3 downto 0);

        signal    LOAD_ACC7       : std_logic;
        signal    DACCR7_0        : std_logic_vector(3 downto 0);
        signal    DACCR7_1        : std_logic_vector(3 downto 0);
        signal    DACCR7_2        : std_logic_vector(3 downto 0);
        signal    DACCR7          : std_logic_vector(3 downto 0);

        signal    FRQ     : std_logic_vector(31 downto 0);

        signal    ZERO    : std_logic;

BEGIN

        PHASE_OV <= LOAD_ACC5;

        --ZERO    <= not DDS_NRST;
```

```
        ZERO        <= '0';

        --FRQ       <= IN_FRQ;
        FRQ         <= INCREMENT;

        INC_FRQ_BIT:
        process(DDS_NRST, CLOCK)
        begin
                if DDS_NRST = '0' then
                        INC_FRQ <= (others=>'0');
                elsif CLOCK'event and (CLOCK = '1') then
                        if ((DDS_NRST = '1') and (NRST(1) = '0')) or (LOAD_ACC = '1') then
                                INC_FRQ <= FRQ;
                        end if;
                end if;
        end process;

-- synopsys translate_off
        IN_FRQ <=       ("0001" & "0010" & "0101" & "0100" & "0000000000000000"),
                        ("0000" & "0101" & "1000" & "0100" & "0000000000000000") after 692 ns,
                        ("0001" & "0010" & "0101" & "0100" & "0000000000000000") after 1433 ns
                        ;
--      IN_FRQ <=       ("0011" & "0011" & "0011" & "0011" & "0011" & "0011" & "0011" & "0011");
-- synopsys translate_on

--- STAGE 1

        PINC(0)  <= FRQ( 3 downto 0); PINC(1)  <= FRQ( 7 downto 4); PINC(2)  <= FRQ(11 downto 8);
        PINC(3)  <= FRQ(15 downto 12); PINC(4)<= FRQ(19 downto 16); PINC(5)<= FRQ(23 downto 20);
        PINC(6)  <= FRQ(27 downto 24); PINC(7)<= FRQ(31 downto 28);

        PDIN_INC(0)     <= INC_FRQ( 3 downto 0);
        PDIN_INC(1)     <= INC_FRQ( 7 downto 4);
        PDIN_INC(2)     <= INC_FRQ(11 downto 8);
        PDIN_INC(3)     <= INC_FRQ(15 downto 12);
        PDIN_INC(4)     <= INC_FRQ(19 downto 16);
        PDIN_INC(5)     <= INC_FRQ(23 downto 20);
        PDIN_INC(6)     <= INC_FRQ(27 downto 24);
        PDIN_INC(7)     <= INC_FRQ(31 downto 28);

        ACC_CIN(0)      <= ZERO;


        ACC0_UNIT: ACC4 port map (
                NRESET=>NRST(1), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(0),
                DIN=>ADIN0, DOUT=>OPEN, CO=>ACC_CIN(1) );

        ACC1_UNIT: ACC4 port map (
                NRESET=>NRST(2), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(1),
                DIN=>ADIN1, DOUT=>OPEN, CO=>ACC_CIN(2) );

        ACC2_UNIT: ACC4 port map (
                NRESET=>NRST(3), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(2),
                DIN=>ADIN2, DOUT=>OPEN, CO=>ACC_CIN(3) );

        ACC3_UNIT: ACC4 port map (
                NRESET=>NRST(4), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(3),
                DIN=>ADIN3, DOUT=>OPEN, CO=>ACC_CIN(4) );

        ACC4_UNIT: ACC4 port map (
                NRESET=>NRST(5), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(4),
                DIN=>ADIN4, DOUT=>ACCR4, CO=>ACC_CIN(5) );

        ACC5_UNIT: ACC4 port map (
                NRESET=>NRST(6), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(5),
                DIN=>ADIN5, DOUT=>ACCR5, CO=>ACC_CIN(6) );

        ACC6_UNIT: ACC4 port map (
                NRESET=>NRST(7), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(6),
                DIN=>ADIN6, DOUT=>ACCR6, CO=>ACC_CIN(7) );
```

```
        ACC7_UNIT: ACC4 port map (
                NRESET=>NRST(8), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(7),
                DIN=>ADIN7, DOUT=>ACCR7, CO=>ACC_CIN(8) );

        LOAD_ACC        <= ACC_CIN(8);

-- synopsys translate_off
        TACC0_UNIT: ACC4 port map (
                NRESET=>NRST(1), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(0),
                DIN=>ADIN0, DOUT=>ACCR0, CO=>OPEN );

        TACC1_UNIT: ACC4 port map (
                NRESET=>NRST(2), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(1),
                DIN=>ADIN1, DOUT=>ACCR1, CO=>OPEN );

        TACC2_UNIT: ACC4 port map (
                NRESET=>NRST(3), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(2),
                DIN=>ADIN2, DOUT=>ACCR2, CO=>OPEN );

        TACC3_UNIT: ACC4 port map (
                NRESET=>NRST(4), CLOCK=>CLOCK, LOAD=>LOAD_ACC, CIN=>ACIN(3),
                DIN=>ADIN3, DOUT=>ACCR3, CO=>OPEN );
-- synopsys translate_on


-- Restore value
        ADIN0   <= PDIN_INC(0) when LOAD_ACC = '0' else IL1S(6);
        ADIN1   <= PDIN_INC(1) when LOAD_ACC = '0' else IL2S(5);
        ADIN2   <= PDIN_INC(2) when LOAD_ACC = '0' else IL3S(4);
        ADIN3   <= PDIN_INC(3) when LOAD_ACC = '0' else IL4S(3);
        ADIN4   <= PDIN_INC(4) when LOAD_ACC = '0' else IL5S(2);
        ADIN5   <= PDIN_INC(5) when LOAD_ACC = '0' else IL6S(1);
        ADIN6   <= PDIN_INC(6) when LOAD_ACC = '0' else IL7S(0);
        ADIN7   <= PDIN_INC(7) when LOAD_ACC = '0' else PINC(7);

        ACIN(8) <= ACC_CIN(8)   when LOAD_ACC = '0' else ZERO;
        ACIN(7) <= ACC_CIN(7)   when LOAD_ACC = '0' else ZERO;
        ACIN(6) <= ACC_CIN(6)   when LOAD_ACC = '0' else IL7C(0);
        ACIN(5) <= ACC_CIN(5)   when LOAD_ACC = '0' else IL6C(1);
        ACIN(4) <= ACC_CIN(4)   when LOAD_ACC = '0' else IL5C(2);
        ACIN(3) <= ACC_CIN(3)   when LOAD_ACC = '0' else IL4C(3);
        ACIN(2) <= ACC_CIN(2)   when LOAD_ACC = '0' else IL3C(4);
        ACIN(1) <= ACC_CIN(1)   when LOAD_ACC = '0' else IL2C(5);
        ACIN(0) <= ACC_CIN(0)   when LOAD_ACC = '0' else IL1C(6);

        DACCR4_REG:
        process(NRST(2), CLOCK)
        begin
                if NRST(2) = '0' then
                        DACCR4_0        <= (others=>'0');
                        DACCR4_1        <= (others=>'0');
                        DACCR4_2        <= (others=>'0');
                        DACCR4          <= (others=>'0');
                        LOAD_ACC5       <= '0';
                elsif CLOCK'event and (CLOCK = '1') then
                        LOAD_ACC5       <= LOAD_ACC;
                        if LOAD_ACC = '0' then
                                DACCR4          <= DACCR4_2;
                                DACCR4_2        <= DACCR4_1;
                                DACCR4_1        <= DACCR4_0;
                                DACCR4_0        <= ACCR4;
                        else
                                DACCR4          <= (others=>'0');
                                DACCR4_2        <= PINC(4);
                                DACCR4_1        <= IL5S(0);
                                DACCR4_0        <= IL5S(1);
                        end if;

                end if;
```

```
end process;

DACCR5_REG:
process(NRST(3), CLOCK)
begin
        if NRST(3) = '0' then
                DACCR5_0        <= (others=>'0');
                DACCR5_1        <= (others=>'0');
                DACCR5_2        <= (others=>'0');
                DACCR5          <= (others=>'0');
                LOAD_ACC6       <= '0';
        elsif CLOCK'event and (CLOCK = '1') then
                LOAD_ACC6       <= LOAD_ACC5;
                if LOAD_ACC5 = '0' then
                        DACCR5          <= DACCR5_2;
                        DACCR5_2        <= DACCR5_1;
                        DACCR5_1        <= DACCR5_0;
                        DACCR5_0        <= ACCR5;
                else
                        DACCR5          <= (others=>'0');
                        DACCR5_2        <= PINC(5);
                        DACCR5_1        <= IL6S(0);
                        DACCR5_0        <= IL6S(1);
                end if;

        end if;
end process;

DACCR6_REG:
process(NRST(4), CLOCK)
begin
        if NRST(4) = '0' then
                DACCR6_0        <= (others=>'0');
                DACCR6_1        <= (others=>'0');
                DACCR6_2        <= (others=>'0');
                DACCR6          <= (others=>'0');
                LOAD_ACC7       <= '0';
        elsif CLOCK'event and (CLOCK = '1') then
                LOAD_ACC7       <= LOAD_ACC6;
                if LOAD_ACC6 = '0' then
                        DACCR6          <= DACCR6_2;
                        DACCR6_2        <= DACCR6_1;
                        DACCR6_1        <= DACCR6_0;
                        DACCR6_0        <= ACCR6;
                else
                        DACCR6          <= (others=>'0');
                        DACCR6_2        <= PINC(6);
                        DACCR6_1        <= IL7S(0);
                        DACCR6_0        <= IL7S(1);
                end if;

        end if;
end process;

DACCR7_REG:
process(NRST(5), CLOCK)
begin
        if NRST(5) = '0' then
                DACCR7_0        <= (others=>'0');
                DACCR7_1        <= (others=>'0');
                DACCR7_2        <= (others=>'0');
                DACCR7          <= (others=>'0');
        elsif CLOCK'event and (CLOCK = '1') then
                if LOAD_ACC7 = '0' then
                        DACCR7          <= DACCR7_2;
                        DACCR7_2        <= DACCR7_1;
                        DACCR7_1        <= DACCR7_0;
                        DACCR7_0        <= ACCR7;
                else
                        DACCR7          <= (others=>'0');
```

```
                DACCR7_2        <= PINC(7);
                DACCR7_1        <= IL8S(0);
                DACCR7_0        <= IL8S(1);
            end if;

        end if;
    end process;

-- INC LEVEL 1
    ADD_L1_0: ADDER4 port map (A=>PINC(0), B=>PINC(0), CIN=>ZERO,  S=>IL1S(0), COUT=>IL1C(0));
    ADD_L1_1: ADDER4 port map (A=>IL1S(0), B=>PINC(0), CIN=>ZERO,  S=>IL1S(1), COUT=>IL1C(1));
    ADD_L1_2: ADDER4 port map (A=>IL1S(1), B=>PINC(0), CIN=>ZERO,  S=>IL1S(2), COUT=>IL1C(2));
    ADD_L1_3: ADDER4 port map (A=>IL1S(2), B=>PINC(0), CIN=>ZERO,  S=>IL1S(3), COUT=>IL1C(3));
    ADD_L1_4: ADDER4 port map (A=>IL1S(3), B=>PINC(0), CIN=>ZERO,  S=>IL1S(4), COUT=>IL1C(4));
    ADD_L1_5: ADDER4 port map (A=>IL1S(4), B=>PINC(0), CIN=>ZERO,  S=>IL1S(5), COUT=>IL1C(5));
    ADD_L1_6: ADDER4 port map (A=>IL1S(5), B=>PINC(0), CIN=>ZERO,  S=>IL1S(6), COUT=>IL1C(6));
    ADD_L1_7: ADDER4 port map (A=>IL1S(6), B=>PINC(0), CIN=>ZERO,  S=>IL1S(7), COUT=>IL1C(7));
-- INC LEVEL 2
    ADD_L2_0: ADDER4 port map (A=>PINC(1), B=>PINC(1), CIN=>IL1C(0), S=>IL2S(0), COUT=>IL2C(0));
    ADD_L2_1: ADDER4 port map (A=>IL2S(0), B=>PINC(1), CIN=>IL1C(1), S=>IL2S(1), COUT=>IL2C(1));
    ADD_L2_2: ADDER4 port map (A=>IL2S(1), B=>PINC(1), CIN=>IL1C(2), S=>IL2S(2), COUT=>IL2C(2));
    ADD_L2_3: ADDER4 port map (A=>IL2S(2), B=>PINC(1), CIN=>IL1C(3), S=>IL2S(3), COUT=>IL2C(3));
    ADD_L2_4: ADDER4 port map (A=>IL2S(3), B=>PINC(1), CIN=>IL1C(4), S=>IL2S(4), COUT=>IL2C(4));
    ADD_L2_5: ADDER4 port map (A=>IL2S(4), B=>PINC(1), CIN=>IL1C(5), S=>IL2S(5), COUT=>IL2C(5));
    ADD_L2_6: ADDER4 port map (A=>IL2S(5), B=>PINC(1), CIN=>IL1C(6), S=>IL2S(6), COUT=>IL2C(6));
    ADD_L2_7: ADDER4 port map (A=>IL2S(6), B=>PINC(1), CIN=>IL1C(7), S=>IL2S(7), COUT=>IL2C(7));
-- INC LEVEL 3
    ADD_L3_0: ADDER4 port map (A=>PINC(2), B=>PINC(2), CIN=>IL2C(0), S=>IL3S(0), COUT=>IL3C(0));
    ADD_L3_1: ADDER4 port map (A=>IL3S(0), B=>PINC(2), CIN=>IL2C(1), S=>IL3S(1), COUT=>IL3C(1));
    ADD_L3_2: ADDER4 port map (A=>IL3S(1), B=>PINC(2), CIN=>IL2C(2), S=>IL3S(2), COUT=>IL3C(2));
    ADD_L3_3: ADDER4 port map (A=>IL3S(2), B=>PINC(2), CIN=>IL2C(3), S=>IL3S(3), COUT=>IL3C(3));
    ADD_L3_4: ADDER4 port map (A=>IL3S(3), B=>PINC(2), CIN=>IL2C(4), S=>IL3S(4), COUT=>IL3C(4));
    ADD_L3_5: ADDER4 port map (A=>IL3S(4), B=>PINC(2), CIN=>IL2C(5), S=>IL3S(5), COUT=>IL3C(5));
    ADD_L3_6: ADDER4 port map (A=>IL3S(5), B=>PINC(2), CIN=>IL2C(6), S=>IL3S(6), COUT=>IL3C(6));
    ADD_L3_7: ADDER4 port map (A=>IL3S(6), B=>PINC(2), CIN=>IL2C(7), S=>IL3S(7), COUT=>IL3C(7));
-- INC LEVEL 4
    ADD_L4_0: ADDER4 port map (A=>PINC(3), B=>PINC(3), CIN=>IL3C(0), S=>IL4S(0), COUT=>IL4C(0));
    ADD_L4_1: ADDER4 port map (A=>IL4S(0), B=>PINC(3), CIN=>IL3C(1), S=>IL4S(1), COUT=>IL4C(1));
    ADD_L4_2: ADDER4 port map (A=>IL4S(1), B=>PINC(3), CIN=>IL3C(2), S=>IL4S(2), COUT=>IL4C(2));
    ADD_L4_3: ADDER4 port map (A=>IL4S(2), B=>PINC(3), CIN=>IL3C(3), S=>IL4S(3), COUT=>IL4C(3));
    ADD_L4_4: ADDER4 port map (A=>IL4S(3), B=>PINC(3), CIN=>IL3C(4), S=>IL4S(4), COUT=>IL4C(4));
    ADD_L4_5: ADDER4 port map (A=>IL4S(4), B=>PINC(3), CIN=>IL3C(5), S=>IL4S(5), COUT=>IL4C(5));
    ADD_L4_6: ADDER4 port map (A=>IL4S(5), B=>PINC(3), CIN=>IL3C(6), S=>IL4S(6), COUT=>IL4C(6));
    ADD_L4_7: ADDER4 port map (A=>IL4S(6), B=>PINC(3), CIN=>IL3C(7), S=>IL4S(7), COUT=>IL4C(7));
-- INC LEVEL 5
    ADD_L5_0: ADDER4 port map (A=>PINC(4), B=>PINC(4), CIN=>IL4C(0), S=>IL5S(0), COUT=>IL5C(0));
    ADD_L5_1: ADDER4 port map (A=>IL5S(0), B=>PINC(4), CIN=>IL4C(1), S=>IL5S(1), COUT=>IL5C(1));
    ADD_L5_2: ADDER4 port map (A=>IL5S(1), B=>PINC(4), CIN=>IL4C(2), S=>IL5S(2), COUT=>IL5C(2));
    ADD_L5_3: ADDER4 port map (A=>IL5S(2), B=>PINC(4), CIN=>IL4C(3), S=>IL5S(3), COUT=>IL5C(3));
    ADD_L5_4: ADDER4 port map (A=>IL5S(3), B=>PINC(4), CIN=>IL4C(4), S=>IL5S(4), COUT=>IL5C(4));
    ADD_L5_5: ADDER4 port map (A=>IL5S(4), B=>PINC(4), CIN=>IL4C(5), S=>IL5S(5), COUT=>IL5C(5));
    ADD_L5_6: ADDER4 port map (A=>IL5S(5), B=>PINC(4), CIN=>IL4C(6), S=>IL5S(6), COUT=>IL5C(6));
    ADD_L5_7: ADDER4 port map (A=>IL5S(6), B=>PINC(4), CIN=>IL4C(7), S=>IL5S(7), COUT=>IL5C(7));
-- INC LEVEL 6
    ADD_L6_0: ADDER4 port map (A=>PINC(5), B=>PINC(5), CIN=>IL5C(0), S=>IL6S(0), COUT=>IL6C(0));
    ADD_L6_1: ADDER4 port map (A=>IL6S(0), B=>PINC(5), CIN=>IL5C(1), S=>IL6S(1), COUT=>IL6C(1));
    ADD_L6_2: ADDER4 port map (A=>IL6S(1), B=>PINC(5), CIN=>IL5C(2), S=>IL6S(2), COUT=>IL6C(2));
    ADD_L6_3: ADDER4 port map (A=>IL6S(2), B=>PINC(5), CIN=>IL5C(3), S=>IL6S(3), COUT=>IL6C(3));
    ADD_L6_4: ADDER4 port map (A=>IL6S(3), B=>PINC(5), CIN=>IL5C(4), S=>IL6S(4), COUT=>IL6C(4));
    ADD_L6_5: ADDER4 port map (A=>IL6S(4), B=>PINC(5), CIN=>IL5C(5), S=>IL6S(5), COUT=>IL6C(5));
    ADD_L6_6: ADDER4 port map (A=>IL6S(5), B=>PINC(5), CIN=>IL5C(6), S=>IL6S(6), COUT=>IL6C(6));
    ADD_L6_7: ADDER4 port map (A=>IL6S(6), B=>PINC(5), CIN=>IL5C(7), S=>IL6S(7), COUT=>IL6C(7));
-- INC LEVEL 7
    ADD_L7_0: ADDER4 port map (A=>PINC(6), B=>PINC(6), CIN=>IL6C(0), S=>IL7S(0), COUT=>IL7C(0));
    ADD_L7_1: ADDER4 port map (A=>IL7S(0), B=>PINC(6), CIN=>IL6C(1), S=>IL7S(1), COUT=>IL7C(1));
    ADD_L7_2: ADDER4 port map (A=>IL7S(1), B=>PINC(6), CIN=>IL6C(2), S=>IL7S(2), COUT=>IL7C(2));
    ADD_L7_3: ADDER4 port map (A=>IL7S(2), B=>PINC(6), CIN=>IL6C(3), S=>IL7S(3), COUT=>IL7C(3));
    ADD_L7_4: ADDER4 port map (A=>IL7S(3), B=>PINC(6), CIN=>IL6C(4), S=>IL7S(4), COUT=>IL7C(4));
    ADD_L7_5: ADDER4 port map (A=>IL7S(4), B=>PINC(6), CIN=>IL6C(5), S=>IL7S(5), COUT=>IL7C(5));
    ADD_L7_6: ADDER4 port map (A=>IL7S(5), B=>PINC(6), CIN=>IL6C(6), S=>IL7S(6), COUT=>IL7C(6));
```

```
        ADD_L7_7: ADDER4 port map (A=>IL7S(6), B=>PINC(6), CIN=>IL6C(7), S=>IL7S(7), COUT=>IL7C(7));
-- INC LEVEL 8
        ADD_L8_0: ADDER4 port map (A=>PINC(7), B=>PINC(7), CIN=>IL7C(0), S=>IL8S(0), COUT=>OPEN);
        ADD_L8_1: ADDER4 port map (A=>IL8S(0), B=>PINC(7), CIN=>IL7C(1), S=>IL8S(1), COUT=>OPEN);
        ADD_L8_2: ADDER4 port map (A=>IL8S(1), B=>PINC(7), CIN=>IL7C(2), S=>IL8S(2), COUT=>OPEN);
        ADD_L8_3: ADDER4 port map (A=>IL8S(2), B=>PINC(7), CIN=>IL7C(3), S=>IL8S(3), COUT=>OPEN);
        ADD_L8_4: ADDER4 port map (A=>IL8S(3), B=>PINC(7), CIN=>IL7C(4), S=>IL8S(4), COUT=>OPEN);
        ADD_L8_5: ADDER4 port map (A=>IL8S(4), B=>PINC(7), CIN=>IL7C(5), S=>IL8S(5), COUT=>OPEN);
        ADD_L8_6: ADDER4 port map (A=>IL8S(5), B=>PINC(7), CIN=>IL7C(6), S=>IL8S(6), COUT=>OPEN);
        ADD_L8_7: ADDER4 port map (A=>IL8S(6), B=>PINC(7), CIN=>IL7C(7), S=>IL8S(7), COUT=>OPEN);

-- synopsys translate_off

        ACC_REG:
        process(DDS_NRST, CLOCK)
        begin
                if DDS_NRST = '0' then
                        ACCR6_D <= (others=>'0');

                        ACCR5_1 <= (others=>'0');
                        ACCR5_D <= (others=>'0');

                        ACCR4_L1 <= (others=>'0');
                        ACCR4_L2 <= (others=>'0');
                        ACCR4_D <= (others=>'0');

                        ACCR4_1 <= (others=>'0');
                        ACCR4_2 <= (others=>'0');
                        ACCR4_3 <= (others=>'0');

                        ACCR3_1 <= (others=>'0');
                        ACCR3_2 <= (others=>'0');
                        ACCR3_3 <= (others=>'0');
                        ACCR3_4 <= (others=>'0');

                        ACCR2_1 <= (others=>'0');
                        ACCR2_2 <= (others=>'0');
                        ACCR2_3 <= (others=>'0');
                        ACCR2_4 <= (others=>'0');
                        ACCR2_5 <= (others=>'0');

                        ACCR1_1 <= (others=>'0');
                        ACCR1_2 <= (others=>'0');
                        ACCR1_3 <= (others=>'0');
                        ACCR1_4 <= (others=>'0');
                        ACCR1_5 <= (others=>'0');
                        ACCR1_6 <= (others=>'0');

                        ACCR0_1 <= (others=>'0');
                        ACCR0_2 <= (others=>'0');
                        ACCR0_3 <= (others=>'0');
                        ACCR0_4 <= (others=>'0');
                        ACCR0_5 <= (others=>'0');
                        ACCR0_6 <= (others=>'0');
                        ACCR0_7 <= (others=>'0');
                elsif CLOCK'event and (CLOCK = '1') then
                        if LOAD_ACC     = '0' then
                                ACCR0_7 <= ACCR0_6;
                                ACCR0_6 <= ACCR0_5;
                                ACCR0_5 <= ACCR0_4;
                                ACCR0_4 <= ACCR0_3;
                                ACCR0_3 <= ACCR0_2;
                                ACCR0_2 <= ACCR0_1;
                                ACCR0_1 <= ACCR0;

                                ACCR1_6 <= ACCR1_5;
                                ACCR1_5 <= ACCR1_4;
                                ACCR1_4 <= ACCR1_3;
                                ACCR1_3 <= ACCR1_2;
                                ACCR1_2 <= ACCR1_1;
```

```
                    ACCR1_1 <= ACCR1;

                    ACCR2_5 <= ACCR2_4;
                    ACCR2_4 <= ACCR2_3;
                    ACCR2_3 <= ACCR2_2;
                    ACCR2_2 <= ACCR2_1;
                    ACCR2_1 <= ACCR2;

                    ACCR3_4 <= ACCR3_3;
                    ACCR3_3 <= ACCR3_2;
                    ACCR3_2 <= ACCR3_1;
                    ACCR3_1 <= ACCR3;

                    ACCR4_3 <= ACCR4_2;
                    ACCR4_2 <= ACCR4_1;
                    ACCR4_1 <= ACCR4;

                    ACCR4_D  <= ACCR4_L2;
                    ACCR4_L2 <= ACCR4_L1;
                    ACCR4_L1 <= ACCR4(3 downto 2);

                    ACCR5_D <= ACCR5_1;
                    ACCR5_1 <= ACCR5;

                    ACCR6_D <= ACCR6;
                else
                    ACCR0_7 <= S_ACCR0_7;
                    ACCR0_6 <= S_ACCR0_6;
                    ACCR0_5 <= S_ACCR0_5;
                    ACCR0_4 <= S_ACCR0_4;
                    ACCR0_3 <= S_ACCR0_3;
                    ACCR0_2 <= S_ACCR0_2;
                    ACCR0_1 <= S_ACCR0_1;

                    ACCR1_6 <= S_ACCR1_6;
                    ACCR1_5 <= S_ACCR1_5;
                    ACCR1_4 <= S_ACCR1_4;
                    ACCR1_3 <= S_ACCR1_3;
                    ACCR1_2 <= S_ACCR1_2;
                    ACCR1_1 <= S_ACCR1_1;

                    ACCR2_5 <= S_ACCR2_5;
                    ACCR2_4 <= S_ACCR2_4;
                    ACCR2_3 <= S_ACCR2_3;
                    ACCR2_2 <= S_ACCR2_2;
                    ACCR2_1 <= S_ACCR2_1;

                    ACCR3_4 <= S_ACCR3_4;
                    ACCR3_3 <= S_ACCR3_3;
                    ACCR3_2 <= S_ACCR3_2;
                    ACCR3_1 <= S_ACCR3_1;

                    ACCR4_3 <= S_ACCR4_3;
                    ACCR4_2 <= S_ACCR4_2;
                    ACCR4_1 <= S_ACCR4_1;

                    ACCR4_D  <= S_ACCR4_D ;
                    ACCR4_L2 <= S_ACCR4_L2;
                    ACCR4_L1 <= S_ACCR4_L1;

                    ACCR5_D <= S_ACCR5_D;
                    ACCR5_1 <= S_ACCR5_1;

                    ACCR6_D <= S_ACCR6_D;

                end if;
        end if;
end process;

S_ACC_REG:
```

```
process(NRST(9), CLOCK)
begin
	if NRST(9) = '0' then
		S_ACC_CIN		<= (others=>'0');

		S_ACCR6_D <= (others=>'0');
		S_ACCR5_1 <= (others=>'0');
		S_ACCR5_D <= (others=>'0');
		S_ACCR4_L1 <= (others=>'0');
		S_ACCR4_L2 <= (others=>'0');
		S_ACCR4_D <= (others=>'0');

		S_ACCR7 <= (others=>'0');
		S_ACCR6 <= (others=>'0');
		S_ACCR5 <= (others=>'0');
		S_ACCR4 <= (others=>'0');
		S_ACCR3 <= (others=>'0');
		S_ACCR2 <= (others=>'0');
		S_ACCR1 <= (others=>'0');
		S_ACCR0 <= (others=>'0');

		S_ACCR4_1 <= (others=>'0');
		S_ACCR4_2 <= (others=>'0');
		S_ACCR4_3 <= (others=>'0');
		S_ACCR3_1 <= (others=>'0');
		S_ACCR3_2 <= (others=>'0');
		S_ACCR3_3 <= (others=>'0');
		S_ACCR3_4 <= (others=>'0');
		S_ACCR2_1 <= (others=>'0');
		S_ACCR2_2 <= (others=>'0');
		S_ACCR2_3 <= (others=>'0');
		S_ACCR2_4 <= (others=>'0');
		S_ACCR2_5 <= (others=>'0');
		S_ACCR1_1 <= (others=>'0');
		S_ACCR1_2 <= (others=>'0');
		S_ACCR1_3 <= (others=>'0');
		S_ACCR1_4 <= (others=>'0');
		S_ACCR1_5 <= (others=>'0');
		S_ACCR1_6 <= (others=>'0');
		S_ACCR0_1 <= (others=>'0');
		S_ACCR0_2 <= (others=>'0');
		S_ACCR0_3 <= (others=>'0');
		S_ACCR0_4 <= (others=>'0');
		S_ACCR0_5 <= (others=>'0');
		S_ACCR0_6 <= (others=>'0');
		S_ACCR0_7 <= (others=>'0');
	elsif CLOCK'event and (CLOCK = '1') then
		if NRST(10)		= '0' then
			S_ACC_CIN		<= ACC_CIN(8 downto 1);

			S_ACCR7 <= ACCR7;
			S_ACCR6 <= ACCR6;
			S_ACCR5 <= ACCR5;
			S_ACCR4 <= ACCR4;
			S_ACCR3 <= ACCR3;
			S_ACCR2 <= ACCR2;
			S_ACCR1 <= ACCR1;
			S_ACCR0 <= ACCR0;
			S_ACCR0_7 <= ACCR0_7;
			S_ACCR0_6 <= ACCR0_6;
			S_ACCR0_5 <= ACCR0_5;
			S_ACCR0_4 <= ACCR0_4;
			S_ACCR0_3 <= ACCR0_3;
			S_ACCR0_2 <= ACCR0_2;
			S_ACCR0_1 <= ACCR0_1;
			S_ACCR1_6 <= ACCR1_6;
			S_ACCR1_5 <= ACCR1_5;
			S_ACCR1_4 <= ACCR1_4;
			S_ACCR1_3 <= ACCR1_3;
			S_ACCR1_2 <= ACCR1_2;
```

```
                                S_ACCR1_1 <= ACCR1_1;
                                S_ACCR2_5 <= ACCR2_5;
                                S_ACCR2_4 <= ACCR2_4;
                                S_ACCR2_3 <= ACCR2_3;
                                S_ACCR2_2 <= ACCR2_2;
                                S_ACCR2_1 <= ACCR2_1;
                                S_ACCR3_4 <= ACCR3_4;
                                S_ACCR3_3 <= ACCR3_3;
                                S_ACCR3_2 <= ACCR3_2;
                                S_ACCR3_1 <= ACCR3_1;
                                S_ACCR4_3 <= ACCR4_3;
                                S_ACCR4_2 <= ACCR4_2;
                                S_ACCR4_1 <= ACCR4_1;
                                S_ACCR4_D  <= ACCR4_D;
                                S_ACCR4_L2 <= ACCR4_L2;
                                S_ACCR4_L1 <= ACCR4_L1;
                                S_ACCR5_D <= ACCR5_D;
                                S_ACCR5_1 <= ACCR5_1;
                                S_ACCR6_D <= ACCR6_D;
                        end if;
                end if;
end process;

ACC        <= ACCR7 & ACCR6_D & ACCR5_D & ACCR4_3 & ACCR3_4 & ACCR2_5 & ACCR1_6 & ACCR0_7 ;
SUM_TEST <= std_logic_vector(UNSIGNED('0' & ACC_TEST) + UNSIGNED(INCREMENT));

ACC_TEST_REG:
process(DDS_NRST, CLOCK)
variable delay : integer := 0;
begin
        if DDS_NRST = '0' then
                ACC_TEST <= (others=>'0');
        elsif CLOCK'event and (CLOCK = '1') then
                if delay < 8 then
                        delay := delay + 1;
                else
                        if SUM_TEST(32) = '1' then
                                ACC_TEST <= (others=>'0');
                        else
                                ACC_TEST <= SUM_TEST(31 downto 0);
                        end if;
                end if;
        end if;
end process;

ERR_TEST_REG:
process(DDS_NRST, CLOCK)
variable delay : integer := 0;
begin
        if DDS_NRST = '0' then
                ERR_TEST <= '0';
        elsif CLOCK'event and (CLOCK = '0') then
                if delay < 8 then
                        delay := delay + 1;
                else
                        if ACC_TEST = ACC then
                                ERR_TEST <= '0';
                        else
                                ERR_TEST <= '1';
                        end if;
                end if;
        end if;
end process;

-- synopsys translate_on

--IN_PHASE <= (others=>'0'), "010001000100" after 280 ns;
IN_PHASE <= PHASE_MOD;

PHASE_MOD_REG:
```

```
	process(DDS_NRST, CLOCK)
	begin
		if DDS_NRST = '0' then
			PHASE_MOD0 <= (others=>'0');
			PHASE_MOD1 <= (others=>'0');
			PHASE_MOD2 <= (others=>'0');
			PHASE_MOD3 <= (others=>'0');
		elsif CLOCK'event and (CLOCK = '1') then
			--INC_PHASE <= PHASE_MOD;
			if ((DDS_NRST = '1') and (NRST(1) = '0')) or (LOAD_ACC = '1') then
				PHASE_MOD0 <= (IN_PHASE(1 downto 0) & "00");
			end if;
			if ((DDS_NRST = '1') and (NRST(1) = '0')) or (LOAD_ACC5 = '1') then
				PHASE_MOD1 <= IN_PHASE(5 downto 2);
			end if;
			if ((DDS_NRST = '1') and (NRST(1) = '0')) or (LOAD_ACC6 = '1') then
				PHASE_MOD2 <= IN_PHASE(9 downto 6);
			end if;
			if ((DDS_NRST = '1') and (NRST(1) = '0')) or (LOAD_ACC7 = '1') then
				PHASE_MOD3 <= "00" & IN_PHASE(11 downto 10);
			end if;
		end if;
	end process;

	PCIN(0)	<= ZERO;

	ADDP4_UNIT: ACCP4 port map (
		NRESET=>NRST(10), CLOCK=>CLOCK, CIN=>PCIN(0),
		DIN=>DACCR4, PDIN=>PHASE_MOD0,
		DOUT=>ADDPR4, CO=>PCIN(1) );

	ADDP5_UNIT: ACCP4 port map (
		NRESET=>NRST(11), CLOCK=>CLOCK, CIN=>PCIN(1),
		DIN=>DACCR5, PDIN=>PHASE_MOD1,
		DOUT=>ADDPR5, CO=>PCIN(2) );

	ADDP6_UNIT: ACCP4 port map (
		NRESET=>NRST(12), CLOCK=>CLOCK, CIN=>PCIN(2),
		DIN=>DACCR6, PDIN=>PHASE_MOD2,
		DOUT=>ADDPR6, CO=>PCIN(3) );

	ADDP7_UNIT: ACCP4 port map (
		NRESET=>NRST(13), CLOCK=>CLOCK, CIN=>PCIN(3),
		DIN=>DACCR7, PDIN=>PHASE_MOD3,
		DOUT=>ADDPR7, CO=>PCIN(4) );

	ADDP_REG:
	process(DDS_NRST, CLOCK)
	begin
		if DDS_NRST = '0' then
			ADDPR6_D <= (others=>'0');

			ADDPR5_1 <= (others=>'0');
			ADDPR5_D <= (others=>'0');
-- synopsys translate_off
			ADDPR4_1 <= (others=>'0');
			ADDPR4_2 <= (others=>'0');
			ADDPR4_D <= (others=>'0');
-- synopsys translate_on
			ADDPR4_L1 <= (others=>'0');
			ADDPR4_L2 <= (others=>'0');
			ADDPR4_LD <= (others=>'0');

		elsif CLOCK'event and (CLOCK = '1') then
-- synopsys translate_off
				ADDPR4_D <= ADDPR4_2;
				ADDPR4_2 <= ADDPR4_1;
				ADDPR4_1 <= ADDPR4;
-- synopsys translate_on
				ADDPR4_LD <= ADDPR4_L2;
```

```
                              ADDPR4_L2 <= ADDPR4_L1;
                              ADDPR4_L1 <= ADDPR4(3 downto 2);

                              ADDPR5_D <= ADDPR5_1;
                              ADDPR5_1 <= ADDPR5;

                              ADDPR6_D <= ADDPR6;

                    end if;
          end process;

          PHASE_OUT          <= ADDPR7 & ADDPR6_D & ADDPR5_D & ADDPR4_LD;

          PHASE    <= PHASE_OUT;

          PHASE_RUN_REG:
          process(NRST(13), CLOCK)
          begin
                    if NRST(13) = '0' then
                              PHASE_RUN <= (others=>'0');
                    elsif CLOCK'event and (CLOCK = '1') then
                              PHASE_RUN(7 downto 2) <= PHASE_RUN(6 downto 1);
                              if STARTQ = ADDPR7(3 downto 2) then
                                        PHASE_RUN(1)        <= '1';
                              end if;
                    end if;
          end process;


--- STAGE 2
          -- ROM ADR SETUP
          ROM_ADR_REG:
          process(NRST(14), CLOCK)
          begin
                    if NRST(14) = '0' then
                              ROM_ADR13           <= '0';
                              ROM_ADR <= (others=>'0');
                    elsif CLOCK'event and (CLOCK = '1') then
                              if PHASE_RUN(1) = '1' then
                                        ROM_ADR13           <= PHASE(13);
                                        if PHASE(12) = '1' then             -- Invert ROM_ADR in QUADRANT 1,4
                                                  ROM_ADR(11 downto 0)          <= not PHASE(11 downto 0);
                                        else
                                                  ROM_ADR(11 downto 0)          <= PHASE(11 downto 0);
                                        end if;
                              end if;
                    end if;
          end process;

          -- Duplicate address
          --CADR <= ROM_ADR(11 downto 8) & ROM_ADR(7 downto 4);
          --FADR <= ROM_ADR(11 downto 8) & ROM_ADR(3 downto 0);

          CFADRH  <= ROM_ADR(11 downto 8);
          CADRL   <= ROM_ADR(7 downto 4);
          FADRL   <= ROM_ADR(3 downto 0);

--- STAGE 3
          -- ROM READ

          MSB_OUT_BIT:
          process(NRST(15), CLOCK)
          begin
                    if NRST(15) = '0' then
                              MSB_OUT <= '0';
                              Q_1          <= '0';
                    elsif CLOCK'event and (CLOCK = '1') then
                              Q_1          <= ROM_ADR13;
                              if to_integer(unsigned(ROM_ADR(11 downto 4))) > 85 then
                                        MSB_OUT             <= '1';
```

```
                    else
                              MSB_OUT <= '0';
                    end if;
          end if;
end process;

--- STAGE 4
     -- ROM RESULT

     ROM_BUF_REG:
     process(NRST(16), CLOCK)
     begin
               if NRST(16) = '0' then
                         COARSE_BUF        <= (others=>'0');
                         FINE_BUF          <= (others=>'0');
                         Q_2       <= '0';
               elsif CLOCK'event and (CLOCK = '1') then
                         Q_2       <= Q_1;
                         COARSE_BUF        <= MSB_OUT & CDIN;
                         FINE_BUF          <= FDIN;
               end if;
     end process;

     RADD0_REG: ADDER2 port map
               (         A=>COARSE_BUF(1 downto 0), B=>FINE_BUF(1 downto 0), CIN=>ZERO,
                         S=>RSUM0(1 downto 0), COUT=>RSUM0_CIN);

     RINC0_REG: INC2 port map
               (         A=>COARSE_BUF(3 downto 2),          CIN=>RSUM0_CIN,
                         S=>RSUM0(3 downto 2), COUT=>RSUM0_CO);

     ROM_SUM0_REG:
     process(NRST(17), CLOCK)
     begin
               if NRST(17) = '0' then
                         ROM_SUM0 <= (others=>'0');
                         COARSE_H <= (others=>'0');
                         RSUM1_CIN         <= '0';
                         Q_3       <= '0';
               elsif CLOCK'event and (CLOCK = '1') then
                         Q_3       <= Q_2;
                         ROM_SUM0 <= RSUM0;
                         COARSE_H <= COARSE_BUF(8 downto 4);
                         RSUM1_CIN <= RSUM0_CO;
               end if;
     end process;

     RINC1_REG: INC5 port map
               (         A=>COARSE_H, CIN=>RSUM1_CIN, S=>ROM_SUM1, COUT=>OPEN);

     ROM_SUM_REG:
     process(NRST(18), CLOCK)
     begin
               if NRST(18) = '0' then
                         ROM_SUM <= (others=>'0');
                         Q_4       <= '0';
               elsif CLOCK'event and (CLOCK = '1') then
                         Q_4       <= Q_3;
                         ROM_SUM <= ROM_SUM1 & ROM_SUM0;
               end if;
     end process;

     -- synopsys translate_off
     ACC       <= ACCR7 & ACCR6_D & ACCR5_D & ACCR4_3 & ACCR3_4 & ACCR2_5 & ACCR1_6 & ACCR0_7 ;

     ROM_TEST_REG:
     process(NRST(16), CLOCK)
     variable ROM_D1, ROM_D2, ROM_D3, ROM_D4, ROM_D5 : std_logic_vector(8 downto 0);
```

```
	begin
		if NRST(16) = '0' then
			ROM_D1  := (others=>'0');
			ROM_D2  := (others=>'0');
			ROM_D3  := (others=>'0');
			ROM_D4  := (others=>'0');
			ROM_D5  := (others=>'0');
			ROM_TEST <= (others=>'0');
		elsif CLOCK'event and (CLOCK = '1') then
			ROM_D5  := ROM_D4;
			ROM_D4  := ROM_D3;
			ROM_D3  := ROM_D2;
			ROM_D2  := ROM_D1;
			ROM_D1  := std_logic_vector(unsigned(COARSE_BUF) + unsigned(FINE_BUF));
			ROM_TEST 		<= ROM_D2;
		end if;
	end process;

	ROM_ERR_REG:
	process(NRST(16), CLOCK)
	variable delay : integer := 0;
	begin
		if NRST(16) = '0' then
			ROM_ERR <= '0';
		elsif CLOCK'event and (CLOCK = '0') then
			if ROM_TEST = ROM_SUM then
				ROM_ERR <= '0';
			else
				ROM_ERR <= '1';
			end if;
		end if;
	end process;

	-- synopsys translate_on


--- STAGE 5
	-- SINE OUTPUT
	SINE_OUT_REG:
	process(NRST(19), CLOCK)
	begin
		if NRST(19) = '0' then
			SINE_OUT <= "1000000000";
		elsif CLOCK'event and (CLOCK = '1') then
			if Q_4 = '1' then -- Invert ROM OUTPUT in QUADRANT 3,4
				SINE_OUT <= '0' & not ROM_SUM;
			else
				SINE_OUT <= '1' & ROM_SUM;
			end if;
		end if;
	end process;

	LOAD_RNG 	<= '1' when (NRST(1) = '1') and (NRST(2) = '0') else '0';

	-- instantiate a 10 bit LFSR
	LFSR10: LFSR generic map (	Width => 10		)
	port map   (		clock => CLOCK,
				nreset => DDS_NRST,
				load => LOAD_RNG,
				seed => "0010101001",
				parallel_out => RNG,
				serial_out => OPEN
		);

	SAVE_RNG_BIT:
	process(DDS_NRST, CLOCK)
	begin
		if DDS_NRST = '0' then
			SAVE_RNG <= '0';
		elsif CLOCK'event and (CLOCK = '1') then
```

```
                        SAVE_RNG <= SINE_OUT(9);
                end if;
        end process;

        RNG_BUF_REG:
        process(NRST(19), CLOCK)
        begin
                if NRST(19) = '0' then
                        RNG_BUF         <= (others=>'0');
                elsif CLOCK'event and (CLOCK = '1') then
                        if (SINE_OUT(9) = '0') and (SAVE_RNG = '1') then
                                RNG_BUF <= RNG;
                        end if;
                end if;
        end process;

        WAVE_IN_REG:
        process(NRST(19), CLOCK)
        begin
                if NRST(19) = '0' then
                        WAVE_IN  <= "0000000000";
                elsif CLOCK'event and (CLOCK = '1') then
                        if CTRL_BIN = '1' then
                                WAVE_IN  <= DAC_BIN;
                        else
                                WAVE_IN  <= RNG_BUF;
                        end if;
                end if;
        end process;

--- STAGE 6
        --
        WAVE_I0 <= SINE_OUT     when WTYPE(0) = '0' else WAVE_IN;
        WAVE_I1 <= WAVE_LIN     when WTYPE(0) = '0' else WAVE_TRI;

        WAVE_REG:
        process(DDS_NRST, CLOCK)
        begin
                if DDS_NRST = '0' then
                        WAVE_OUT        <= "0000000000";
                        WAVE_EXT        <= "0000000000";
                elsif CLOCK'event and (CLOCK = '1') then
                        if NRST(12) = '0' then
                                case STARTQ is
                                        when "00" =>        WAVE_OUT <= "1000000000";
                                        when "01" =>        WAVE_OUT <= "1111111111";
                                        when "10" =>        WAVE_OUT <= "1000000000";
                                        when others =>      WAVE_OUT <= "0000000000";
                                end case;
                                WAVE_EXT        <= "0000000000";
                        else
                                if WTYPE(1) = '0' then
                                        if PHASE_RUN(7) = '1' then
                                                WAVE_OUT        <= WAVE_I0;
                                        end if;
                                else
                                        WAVE_OUT        <= WAVE_I1;
                                end if;
                                if DDSOUT_EN = '1' then
                                        WAVE_EXT <= WAVE_OUT;
                                end if;
                        end if;
                end if;
        end process;

        DECODER: DAC_DECODER
        PORT MAP (
                TSTEN           =>TSTEN,
                IWAVE           =>WAVE_OUT,
                XWAVE           =>XWAVE,
```

```
                -- Wave output
                code                =>code,
                b                           =>b
        );

END RTL;
```

5. DAC_DECODER Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY DAC_DECODER IS
PORT(   TSTEN           : in    std_logic;                              -- Test enable
        IWAVE           : in    std_logic_vector(9 downto 0);
        XWAVE           : in    std_logic_vector(9 downto 0);
        -- Wave output
        code            : out   std_logic_vector(63 downto 1);
        b               : out   std_logic_vector( 3 downto 0)
        );
END DAC_DECODER;

ARCHITECTURE RTL OF DAC_DECODER IS

        signal  thm_part  : std_logic_vector(5 downto 0);
        signal  icode     : std_logic_vector(63 downto 1);
        -- synopsys translate_off
        signal  ib        : std_logic_vector(3 downto 0);
        signal  IDAC      : integer;
        -- synopsys translate_on

BEGIN

        b          <= IWAVE(3 downto 0) when TSTEN = '0' else XWAVE(3 downto 0);

        thm_part   <= IWAVE(9 downto 4) when TSTEN = '0' else XWAVE(9 downto 4);

        thm_decoder:
        process(thm_part)
        variable thm_value : integer;
        begin
                thm_value := to_integer(unsigned(thm_part));
                ------------------------
                --                      MSB LSB
                -- 00100   0000001111 --
                for i in 1 to 63 loop
                        if i > thm_value then
                                icode(i) <= '0';
                        else
                                icode(i) <= '1';
                        end if;
                end loop;
        end process;

        code       <= icode;

        -- synopsys translate_off
        ib         <= IWAVE(3 downto 0) when TSTEN = '0' else XWAVE(3 downto 0);

        DAC_PROC:
        process(icode, ib)
        variable current : integer := 0;
        begin
                current := 0;
                for i in 1 to 63 loop
                        if icode(i) = '1' then
                                        current := current + 1;
                        end if;
                end loop;
                current := current * 16;
                IDAC <= current + to_integer(unsigned(ib));
        end process;
        -- synopsys translate_on

END RTL;
```

6. LFSR Module

```
library IEEE;
use IEEE.STD_Logic_1164.all;

entity LFSR is

generic(Width: positive := 10);          -- length of pseudo-random sequence
port       (          clock              : in std_logic;
                      nreset             : in std_logic;          -- active low reset
                      load               : in std_logic;                    -- active high load
                      seed               : in std_logic_vector(Width-1 downto 0);     -- parallel seed input
                      parallel_out       : out std_logic_vector(Width-1 downto 0); -- parallel data out
                      serial_out         : out std_logic          -- serial data out (From last shift register)
           );
end LFSR;

architecture RTL of LFSR is

-- Look-Up Table for Tap points to insert XOR gates as feedback into D-FF
-- outputs.  Taps are designed so that 2^N-1 (N=Width of Register) numbers
-- are cycled through before the sequence is repeated

type TapsArrayType is array(2 to 32) of std_logic_vector(31 downto 0);
constant TapsArray: TapsArrayType := TapsArrayType'(
          ("00000000000000000000000000000011"), --2
          ("00000000000000000000000000000101"), --3
          ("00000000000000000000000000001001"), --4
          ("00000000000000000000000000010010"), --5
          ("00000000000000000000000000100001"), --6
          ("00000000000000000000000001000001"), --7
          ("00000000000000000000000010001110"), --8
          ("00000000000000000000000100001000"), --9
          ("00000000000000000000001000000100"), --10
          ("00000000000000000000010000000010"),
          ("00000000000000000000100000101001"),
          ("00000000000000000001000000001101"),
          ("00000000000000000010000000010101"),
          ("00000000000000000100000000000001"),
          ("00000000000000001000000000010110"),
          ("00000000000000010000000000000100"),
          ("00000000000000100000000001000000"),
          ("00000000000001000000000000010011"),
          ("00000000000010000000000000000100"),
          ("00000000000100000000000000000010"),
          ("00000000001000000000000000000001"),
          ("00000000010000000000000000010000"),
          ("00000000100000000000000000001101"),
          ("00000001000000000000000000000100"),
          ("00000010000000000000000000100011"),
          ("00000100000000000000000000010011"),
          ("00001000000000000000000000000100"),
          ("00010000000000000000000000000010"),
          ("00100000000000000000000000101001"),
          ("01000000000000000000000000000100"),
          ("10000000000000000000000001100010")
);

constant Taps: std_logic_vector(Width-1 downto 0) := TapsArray(Width)(Width-1 downto 0);
signal     LFSR_Reg: std_logic_vector(Width-1 downto 0);

begin

          LFSR_UNIT:
          process (nreset, clock)
          -- internal registers and signals
          begin

                    if nreset='0' then
```

```
                    LFSR_Reg <= (others=>'1');

        elsif rising_edge(clock) then
                -- load signal asserted, use seed value on port to determine where
                -- to start in the pseudo-random sequence
                if load = '1' then
                        --for index in seed'range loop
                        --      if seed(index) = '1' then
                                        LFSR_Reg <= seed;
                        --      end if;
                        --end loop;
                else

                -- Use RTL to construct linear feedback shift register network as described in
                -- appnote diagram

                        for N in Width-1 downto 1 loop

                                if (Taps(N-1)='1') then

                                        LFSR_Reg(N) <= LFSR_Reg(N-1) xor LFSR_Reg(Width-1);

                                else

                                        LFSR_Reg(N) <= LFSR_Reg(N-1);

                                end if;

                        end loop;

                        LFSR_Reg(0) <= LFSR_Reg(Width-1);

                end if;

        end if;

    end process;

    parallel_out <= LFSR_Reg;       -- parallel data out
    serial_out <= LFSR_Reg(Width-1);        -- serial data out

end RTL;
```

7. ACC4 Module

```
--A simple example using the package TextIO is:
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ACC4 is
port(   NRESET          : in    std_logic;
        CLOCK           : in    std_logic;
        LOAD            : in    std_logic;
        CIN             : in    std_logic;
        DIN             : in    std_logic_vector( 3 downto 0);
        DOUT            : out   std_logic_vector( 3 downto 0);
        CO              : out   std_logic
        );
end ACC4;

architecture RTL of ACC4 is

component ADDER4
port (  A               : in    std_logic_vector(3 downto 0);
        B               : in    std_logic_vector(3 downto 0);
        CIN             : in    std_logic;
        S               : out   std_logic_vector(3 downto 0);
        COUT            : out   std_logic
        );
end component;

        signal  ACC     : std_logic_vector(4 downto 0);
        signal  ACC_S   : std_logic_vector(3 downto 0);
        signal  ACC_CO  : std_logic;

BEGIN

        ADD_UNIT: ADDER4 port map
                (A=>DIN, B=>ACC(3 downto 0), CIN=>CIN, S=>ACC_S, COUT=>ACC_CO);

        ACC_REG:
        process(NRESET, CLOCK)
        begin
                if NRESET = '0' then
                        ACC <= (others=>'0');
                elsif CLOCK'event and (CLOCK = '1') then
                        if LOAD = '1' then
                                ACC     <= CIN & DIN;
                        else
                                ACC <= (ACC_CO & ACC_S);
                        end if;
                end if;
        end process;

        DOUT    <= ACC(3 downto 0);
        CO              <= ACC(4);

END RTL;
```

8. ADDER4 Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ADDER4 is
port (     A            : in    std_logic_vector(3 downto 0);
           B            : in    std_logic_vector(3 downto 0);
           CIN          : in    std_logic;
           S            : out   std_logic_vector(3 downto 0);
           COUT         : out   std_logic
           );
end ADDER4;

library mietec05_lib;
architecture RTL of ADDER4 is

begin

           ADDER_UNIT:
           process(A, B, CIN)
           variable SUM : UNSIGNED(4 downto 0);
           begin
                      SUM     := ('0' & UNSIGNED(A)) + UNSIGNED(B) + CIN;
                      COUT    <= SUM(4);
                      S                 <= std_logic_vector(SUM(3 downto 0));
           end process;

end RTL;
```

9. ADDER2 Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ADDER2 is
port (     A            : in    std_logic_vector(1 downto 0);
           B            : in    std_logic_vector(1 downto 0);
           CIN          : in    std_logic;
           S            : out   std_logic_vector(1 downto 0);
           COUT         : out   std_logic
           );
end ADDER2;

library mietec05_lib;
architecture RTL of ADDER2 is

begin

           ADDER_UNIT:
           process(A, B, CIN)
           variable SUM : UNSIGNED(2 downto 0);
           begin
                      SUM     := ('0' & UNSIGNED(A)) + UNSIGNED(B) + CIN;
                      COUT    <= SUM(2);
                      S                 <= std_logic_vector(SUM(1 downto 0));
           end process;

end RTL;
```

10. ACCP4 Module

```
--A simple example using the package TextIO is:
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ACCP4 is
port(   NRESET      : in    std_logic;
        CLOCK       : in    std_logic;
        CIN         : in    std_logic;
        DIN         : in    std_logic_vector( 3 downto 0);
        PDIN        : in    std_logic_vector( 3 downto 0);
        DOUT        : out   std_logic_vector( 3 downto 0);
        CO          : out   std_logic
        );
end ACCP4;

architecture RTL of ACCP4 is

component ADDER4
port (  A           : in    std_logic_vector(3 downto 0);
        B           : in    std_logic_vector(3 downto 0);
        CIN         : in    std_logic;
        S           : out   std_logic_vector(3 downto 0);
        COUT        : out   std_logic
        );
end component;

        signal  ACC             : std_logic_vector(4 downto 0);
        signal  ACC_S   : std_logic_vector(3 downto 0);
        signal  ACC_CO  : std_logic;

BEGIN

        ADD_UNIT: ADDER4 port map
                (A=>DIN, B=>PDIN, CIN=>CIN, S=>ACC_S, COUT=>ACC_CO);

        ACC_REG:
        process(NRESET, CLOCK)
        begin
                if NRESET = '0' then
                        ACC <= (others=>'0');
                elsif CLOCK'event and (CLOCK = '1') then
                        ACC <= (ACC_CO & ACC_S);
                end if;
        end process;

        DOUT    <= ACC(3 downto 0);
        CO              <= ACC(4);

END RTL;
```

11. INC2 Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity INC2 is
port (      A           : in    std_logic_vector(1 downto 0);
            CIN         : in    std_logic;
            S           : out   std_logic_vector(1 downto 0);
            COUT        : out   std_logic
            );
end INC2;

architecture RTL of INC2 is

begin

            ADDER_UNIT:
            process(A, CIN)
            variable SUM : UNSIGNED(2 downto 0);
            begin
                        SUM     := ('0' & UNSIGNED(A)) + CIN;
                        COUT    <= SUM(2);
                        S               <= std_logic_vector(SUM(1 downto 0));
            end process;

end RTL;
```

12. INC5 Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity INC5 is
port (      A           : in    std_logic_vector(4 downto 0);
            CIN         : in    std_logic;
            S           : out   std_logic_vector(4 downto 0);
            COUT        : out   std_logic
            );
end INC5;

architecture RTL of INC5 is

begin

            ADDER_UNIT:
            process(A, CIN)
            variable SUM : UNSIGNED(5 downto 0);
            begin
                        SUM     := ('0' & UNSIGNED(A)) + CIN;
                        COUT    <= SUM(5);
                        S               <= std_logic_vector(SUM(4 downto 0));
            end process;

end RTL;
```

13. RESET_STAGE Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity RESET_STAGE is
port(     NRESET          : in     std_logic;
          EXT_TSTEN       : in     std_logic;
          CLKIN           : in     std_logic;
          ENABLE          : in     std_logic;
          STOP            : in     std_logic;
          CLKOUT          : out    std_logic;
          CLKLATCH        : out    std_logic;
          NRST            : out    std_logic_vector(20 downto 1)
          );
end RESET_STAGE;

architecture RTL of RESET_STAGE is

          signal    RST_QUEUE : std_logic_vector(NRST'HIGH downto 1);
          signal    CLOCK     : std_logic;
          signal    EN        : std_logic;
          signal    CLKEN     : std_logic;
          signal    NRESET_ACC       : std_logic;

BEGIN
          CLOCK    <= CLKIN;
          EN_BIT:
          process(NRESET, CLOCK)
          begin
                    if NRESET = '0' then
                              EN        <= '0';
                    elsif CLOCK'event and (CLOCK = '1') then
                              if EN = '0' then
                                        EN <= ENABLE;
                              else
                                        if RST_QUEUE(NRST'HIGH-1) = '0' then
                                                  EN <= '1';
                                        else
                                                  EN <= ENABLE;
                                        end if;
                              end if;
                    end if;
          end process;
          CLKEN_BIT:
          process(NRESET, CLKIN)
          begin
                    if NRESET = '0' then
                              CLKEN    <= '0';
                    elsif CLKIN'event and (CLKIN = '0') then
                              if (EN = '1') or (EXT_TSTEN = '1') then
                                        CLKEN <= not STOP;
                              else
                                        CLKEN <= '0';
                              end if;
                    end if;
          end process;

          CLKOUT              <= CLKEN and CLKIN;
          CLKLATCH            <= CLKEN and CLKIN;

          RST_QUEUE_REG:
          process(NRESET, CLOCK)
          begin
                    if NRESET = '0' then
                              RST_QUEUE          <= (others=>'0');
                    elsif CLOCK'event and (CLOCK = '1') then
                              RST_QUEUE          <= RST_QUEUE(NRST'HIGH-1 downto 1) & EN;
                    end if;
```

```
        end process;
        -- Must drive by output buffer
        NRST       <= RST_QUEUE;

END RTL;
```

14. COARSE_ROM256x8 Module

```
--A simple example using the package TextIO is:
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity COARSE_ROM256X8 is
port(   A       : in    std_logic_vector(7 downto 0);
        CK      : in    std_logic;
        CSN     : in    std_logic;
--      OEN     : in    std_logic;
        Q       : out   std_logic_vector(7 downto 0)
        );
end COARSE_ROM256X8;

architecture behave of COARSE_ROM256X8 is

subtype COARSE_WORD is std_logic_vector(7 downto 0);
type COARSE_TABLE is array (0 to 255) of COARSE_WORD;
constant COARSE_ROM : COARSE_TABLE := COARSE_TABLE'(
  COARSE_WORD'("00000000"),-- 0
  COARSE_WORD'("00000011"),-- 1
  COARSE_WORD'("00000110"),-- 2
  COARSE_WORD'("00001001"),-- 3
  COARSE_WORD'("00001101"),-- 4
  COARSE_WORD'("00010000"),-- 5
  COARSE_WORD'("00010011"),-- 6
  COARSE_WORD'("00010110"),-- 7
  COARSE_WORD'("00011001"),-- 8
  COARSE_WORD'("00011100"),-- 9
  COARSE_WORD'("00011111"),-- 10
  COARSE_WORD'("00100010"),-- 11
  COARSE_WORD'("00100110"),-- 12
  COARSE_WORD'("00101001"),-- 13
  COARSE_WORD'("00101100"),-- 14
  COARSE_WORD'("00101111"),-- 15
  COARSE_WORD'("00110010"),-- 16
  COARSE_WORD'("00110101"),-- 17
  COARSE_WORD'("00111000"),-- 18
  COARSE_WORD'("00111011"),-- 19
  COARSE_WORD'("00111110"),-- 20
  COARSE_WORD'("01000010"),-- 21
  COARSE_WORD'("01000101"),-- 22
  COARSE_WORD'("01001000"),-- 23
  COARSE_WORD'("01001011"),-- 24
  COARSE_WORD'("01001110"),-- 25
  COARSE_WORD'("01010001"),-- 26
  COARSE_WORD'("01010100"),-- 27
  COARSE_WORD'("01010111"),-- 28
  COARSE_WORD'("01011010"),-- 29
  COARSE_WORD'("01011101"),-- 30
  COARSE_WORD'("01100001"),-- 31
  COARSE_WORD'("01100100"),-- 32
  COARSE_WORD'("01100111"),-- 33
  COARSE_WORD'("01101010"),-- 34
  COARSE_WORD'("01101101"),-- 35
  COARSE_WORD'("01110000"),-- 36
  COARSE_WORD'("01110011"),-- 37
  COARSE_WORD'("01110110"),-- 38
  COARSE_WORD'("01111001"),-- 39
  COARSE_WORD'("01111100"),-- 40
  COARSE_WORD'("01111111"),-- 41
  COARSE_WORD'("10000010"),-- 42
  COARSE_WORD'("10000101"),-- 43
  COARSE_WORD'("10001000"),-- 44
  COARSE_WORD'("10001011"),-- 45
  COARSE_WORD'("10001110"),-- 46
  COARSE_WORD'("10010001"),-- 47
  COARSE_WORD'("10010100"),-- 48
```

```
COARSE_WORD'("10010111"),-- 49
COARSE_WORD'("10011010"),-- 50
COARSE_WORD'("10011101"),-- 51
COARSE_WORD'("10100000"),-- 52
COARSE_WORD'("10100011"),-- 53
COARSE_WORD'("10100110"),-- 54
COARSE_WORD'("10101001"),-- 55
COARSE_WORD'("10101100"),-- 56
COARSE_WORD'("10101111"),-- 57
COARSE_WORD'("10110010"),-- 58
COARSE_WORD'("10110101"),-- 59
COARSE_WORD'("10111000"),-- 60
COARSE_WORD'("10111011"),-- 61
COARSE_WORD'("10111110"),-- 62
COARSE_WORD'("11000001"),-- 63
COARSE_WORD'("11000100"),-- 64
COARSE_WORD'("11000111"),-- 65
COARSE_WORD'("11001001"),-- 66
COARSE_WORD'("11001100"),-- 67
COARSE_WORD'("11001111"),-- 68
COARSE_WORD'("11010010"),-- 69
COARSE_WORD'("11010101"),-- 70
COARSE_WORD'("11011000"),-- 71
COARSE_WORD'("11011011"),-- 72
COARSE_WORD'("11011101"),-- 73
COARSE_WORD'("11100000"),-- 74
COARSE_WORD'("11100011"),-- 75
COARSE_WORD'("11100110"),-- 76
COARSE_WORD'("11101001"),-- 77
COARSE_WORD'("11101011"),-- 78
COARSE_WORD'("11101110"),-- 79
COARSE_WORD'("11110001"),-- 80
COARSE_WORD'("11110100"),-- 81
COARSE_WORD'("11110110"),-- 82
COARSE_WORD'("11111001"),-- 83
COARSE_WORD'("11111100"),-- 84
COARSE_WORD'("11111110"),-- 85
COARSE_WORD'("00000001"),-- 86
COARSE_WORD'("00000100"),-- 87
COARSE_WORD'("00000111"),-- 88
COARSE_WORD'("00001001"),-- 89
COARSE_WORD'("00001100"),-- 90
COARSE_WORD'("00001111"),-- 91
COARSE_WORD'("00010001"),-- 92
COARSE_WORD'("00010100"),-- 93
COARSE_WORD'("00010110"),-- 94
COARSE_WORD'("00011001"),-- 95
COARSE_WORD'("00011100"),-- 96
COARSE_WORD'("00011111"),-- 97
COARSE_WORD'("00100001"),-- 98
COARSE_WORD'("00100100"),-- 99
COARSE_WORD'("00100110"),-- 100
COARSE_WORD'("00101001"),-- 101
COARSE_WORD'("00101011"),-- 102
COARSE_WORD'("00101110"),-- 103
COARSE_WORD'("00110000"),-- 104
COARSE_WORD'("00110011"),-- 105
COARSE_WORD'("00110101"),-- 106
COARSE_WORD'("00111000"),-- 107
COARSE_WORD'("00111010"),-- 108
COARSE_WORD'("00111101"),-- 109
COARSE_WORD'("00111111"),-- 110
COARSE_WORD'("01000010"),-- 111
COARSE_WORD'("01000100"),-- 112
COARSE_WORD'("01000111"),-- 113
COARSE_WORD'("01001001"),-- 114
COARSE_WORD'("01001011"),-- 115
COARSE_WORD'("01001110"),-- 116
COARSE_WORD'("01010000"),-- 117
COARSE_WORD'("01010011"),-- 118
```

```
COARSE_WORD'("01010101"),-- 119
COARSE_WORD'("01010111"),-- 120
COARSE_WORD'("01011001"),-- 121
COARSE_WORD'("01011100"),-- 122
COARSE_WORD'("01011110"),-- 123
COARSE_WORD'("01100000"),-- 124
COARSE_WORD'("01100011"),-- 125
COARSE_WORD'("01100101"),-- 126
COARSE_WORD'("01100111"),-- 127
COARSE_WORD'("01101001"),-- 128
COARSE_WORD'("01101100"),-- 129
COARSE_WORD'("01101110"),-- 130
COARSE_WORD'("01110000"),-- 131
COARSE_WORD'("01110010"),-- 132
COARSE_WORD'("01110100"),-- 133
COARSE_WORD'("01110110"),-- 134
COARSE_WORD'("01111001"),-- 135
COARSE_WORD'("01111011"),-- 136
COARSE_WORD'("01111101"),-- 137
COARSE_WORD'("01111111"),-- 138
COARSE_WORD'("10000001"),-- 139
COARSE_WORD'("10000011"),-- 140
COARSE_WORD'("10000101"),-- 141
COARSE_WORD'("10000111"),-- 142
COARSE_WORD'("10001001"),-- 143
COARSE_WORD'("10001011"),-- 144
COARSE_WORD'("10001101"),-- 145
COARSE_WORD'("10001111"),-- 146
COARSE_WORD'("10010001"),-- 147
COARSE_WORD'("10010011"),-- 148
COARSE_WORD'("10010101"),-- 149
COARSE_WORD'("10010111"),-- 150
COARSE_WORD'("10011001"),-- 151
COARSE_WORD'("10011011"),-- 152
COARSE_WORD'("10011100"),-- 153
COARSE_WORD'("10011110"),-- 154
COARSE_WORD'("10100000"),-- 155
COARSE_WORD'("10100010"),-- 156
COARSE_WORD'("10100100"),-- 157
COARSE_WORD'("10100101"),-- 158
COARSE_WORD'("10100111"),-- 159
COARSE_WORD'("10101001"),-- 160
COARSE_WORD'("10101011"),-- 161
COARSE_WORD'("10101101"),-- 162
COARSE_WORD'("10101110"),-- 163
COARSE_WORD'("10110000"),-- 164
COARSE_WORD'("10110010"),-- 165
COARSE_WORD'("10110011"),-- 166
COARSE_WORD'("10110101"),-- 167
COARSE_WORD'("10110110"),-- 168
COARSE_WORD'("10111000"),-- 169
COARSE_WORD'("10111010"),-- 170
COARSE_WORD'("10111011"),-- 171
COARSE_WORD'("10111101"),-- 172
COARSE_WORD'("10111110"),-- 173
COARSE_WORD'("11000000"),-- 174
COARSE_WORD'("11000001"),-- 175
COARSE_WORD'("11000011"),-- 176
COARSE_WORD'("11000100"),-- 177
COARSE_WORD'("11000110"),-- 178
COARSE_WORD'("11000111"),-- 179
COARSE_WORD'("11001001"),-- 180
COARSE_WORD'("11001010"),-- 181
COARSE_WORD'("11001011"),-- 182
COARSE_WORD'("11001101"),-- 183
COARSE_WORD'("11001110"),-- 184
COARSE_WORD'("11001111"),-- 185
COARSE_WORD'("11010001"),-- 186
COARSE_WORD'("11010010"),-- 187
COARSE_WORD'("11010011"),-- 188
```

```
  COARSE_WORD'("11010100"),-- 189
  COARSE_WORD'("11010110"),-- 190
  COARSE_WORD'("11010111"),-- 191
  COARSE_WORD'("11011000"),-- 192
  COARSE_WORD'("11011001"),-- 193
  COARSE_WORD'("11011011"),-- 194
  COARSE_WORD'("11011100"),-- 195
  COARSE_WORD'("11011101"),-- 196
  COARSE_WORD'("11011110"),-- 197
  COARSE_WORD'("11011111"),-- 198
  COARSE_WORD'("11100000"),-- 199
  COARSE_WORD'("11100001"),-- 200
  COARSE_WORD'("11100010"),-- 201
  COARSE_WORD'("11100011"),-- 202
  COARSE_WORD'("11100100"),-- 203
  COARSE_WORD'("11100101"),-- 204
  COARSE_WORD'("11100110"),-- 205
  COARSE_WORD'("11100111"),-- 206
  COARSE_WORD'("11101000"),-- 207
  COARSE_WORD'("11101001"),-- 208
  COARSE_WORD'("11101010"),-- 209
  COARSE_WORD'("11101011"),-- 210
  COARSE_WORD'("11101100"),-- 211
  COARSE_WORD'("11101101"),-- 212
  COARSE_WORD'("11101110"),-- 213
  COARSE_WORD'("11101111"),-- 214
  COARSE_WORD'("11101111"),-- 215
  COARSE_WORD'("11110000"),-- 216
  COARSE_WORD'("11110001"),-- 217
  COARSE_WORD'("11110010"),-- 218
  COARSE_WORD'("11110010"),-- 219
  COARSE_WORD'("11110011"),-- 220
  COARSE_WORD'("11110100"),-- 221
  COARSE_WORD'("11110100"),-- 222
  COARSE_WORD'("11110101"),-- 223
  COARSE_WORD'("11110101"),-- 224
  COARSE_WORD'("11110110"),-- 225
  COARSE_WORD'("11110111"),-- 226
  COARSE_WORD'("11110111"),-- 227
  COARSE_WORD'("11111000"),-- 228
  COARSE_WORD'("11111000"),-- 229
  COARSE_WORD'("11111001"),-- 230
  COARSE_WORD'("11111001"),-- 231
  COARSE_WORD'("11111010"),-- 232
  COARSE_WORD'("11111010"),-- 233
  COARSE_WORD'("11111011"),-- 234
  COARSE_WORD'("11111011"),-- 235
  COARSE_WORD'("11111011"),-- 236
  COARSE_WORD'("11111100"),-- 237
  COARSE_WORD'("11111100"),-- 238
  COARSE_WORD'("11111100"),-- 239
  COARSE_WORD'("11111100"),-- 240
  COARSE_WORD'("11111100"),-- 241
  COARSE_WORD'("11111100"),-- 242
  COARSE_WORD'("11111100"),-- 243
  COARSE_WORD'("11111101"),-- 244
  COARSE_WORD'("11111101"),-- 245
  COARSE_WORD'("11111101"),-- 246
  COARSE_WORD'("11111101"),-- 247
  COARSE_WORD'("11111101"),-- 248
  COARSE_WORD'("11111110"),-- 249
  COARSE_WORD'("11111110"),-- 250
  COARSE_WORD'("11111110"),-- 251
  COARSE_WORD'("11111110"),-- 252
  COARSE_WORD'("11111110"),-- 253
  COARSE_WORD'("11111110"),-- 254
  COARSE_WORD'("11111110") -- 255
);

        signal      DO, DD   : std_logic_vector(7 downto 0) := (others=>'0');
```

```
BEGIN

    READ_FILE:
    PROCESS(CK)
    BEGIN
                if CK'event and (CK = '1') then
                        if CSN = '0' then
                                DO          <= COARSE_ROM(to_integer(unsigned(A)));
                        end if;
                end if;
    END PROCESS;

    DD      <= DO after 2 ns;

        --Q         <= DD when OEN = '0' else (others=>'Z');
        Q           <= DD;

END behave;
```

15. FINE_ROM256x2 Module

```
--A simple example using the package TextIO is:
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity FINE_ROM256X2 is
port(   A       : in    std_logic_vector(7 downto 0);
        CK      : in    std_logic;
        CSN     : in    std_logic;
        --OEN   : in    std_logic;
        Q       : out   std_logic_vector(1 downto 0)
        );
end FINE_ROM256X2;

architecture behave of FINE_ROM256X2 is

subtype FINE_WORD is std_logic_vector(1 downto 0);
type FINE_TABLE is array (0 to 255) of FINE_WORD;
constant FINE_ROM : FINE_TABLE := FINE_TABLE'(
  FINE_WORD'("00"), -- 0
  FINE_WORD'("00"), -- 1
  FINE_WORD'("00"), -- 2
  FINE_WORD'("01"), -- 3
  FINE_WORD'("01"), -- 4
  FINE_WORD'("01"), -- 5
  FINE_WORD'("01"), -- 6
  FINE_WORD'("01"), -- 7
  FINE_WORD'("10"), -- 8
  FINE_WORD'("10"), -- 9
  FINE_WORD'("10"), -- 10
  FINE_WORD'("10"), -- 11
  FINE_WORD'("10"), -- 12
  FINE_WORD'("11"), -- 13
  FINE_WORD'("11"), -- 14
  FINE_WORD'("11"), -- 15
  FINE_WORD'("00"), -- 16
  FINE_WORD'("00"), -- 17
  FINE_WORD'("00"), -- 18
  FINE_WORD'("01"), -- 19
  FINE_WORD'("01"), -- 20
  FINE_WORD'("01"), -- 21
  FINE_WORD'("01"), -- 22
  FINE_WORD'("01"), -- 23
  FINE_WORD'("10"), -- 24
  FINE_WORD'("10"), -- 25
  FINE_WORD'("10"), -- 26
  FINE_WORD'("10"), -- 27
  FINE_WORD'("10"), -- 28
  FINE_WORD'("11"), -- 29
  FINE_WORD'("11"), -- 30
  FINE_WORD'("11"), -- 31
  FINE_WORD'("00"), -- 32
  FINE_WORD'("00"), -- 33
  FINE_WORD'("00"), -- 34
  FINE_WORD'("01"), -- 35
  FINE_WORD'("01"), -- 36
  FINE_WORD'("01"), -- 37
  FINE_WORD'("01"), -- 38
  FINE_WORD'("01"), -- 39
  FINE_WORD'("10"), -- 40
  FINE_WORD'("10"), -- 41
  FINE_WORD'("10"), -- 42
  FINE_WORD'("10"), -- 43
  FINE_WORD'("10"), -- 44
  FINE_WORD'("10"), -- 45
  FINE_WORD'("11"), -- 46
  FINE_WORD'("11"), -- 47
  FINE_WORD'("00"), -- 48
```

```
FINE_WORD'("00"), -- 49
FINE_WORD'("00"), -- 50
FINE_WORD'("01"), -- 51
FINE_WORD'("01"), -- 52
FINE_WORD'("01"), -- 53
FINE_WORD'("01"), -- 54
FINE_WORD'("01"), -- 55
FINE_WORD'("01"), -- 56
FINE_WORD'("10"), -- 57
FINE_WORD'("10"), -- 58
FINE_WORD'("10"), -- 59
FINE_WORD'("10"), -- 60
FINE_WORD'("10"), -- 61
FINE_WORD'("11"), -- 62
FINE_WORD'("11"), -- 63
FINE_WORD'("00"), -- 64
FINE_WORD'("00"), -- 65
FINE_WORD'("00"), -- 66
FINE_WORD'("01"), -- 67
FINE_WORD'("01"), -- 68
FINE_WORD'("01"), -- 69
FINE_WORD'("01"), -- 70
FINE_WORD'("01"), -- 71
FINE_WORD'("01"), -- 72
FINE_WORD'("10"), -- 73
FINE_WORD'("10"), -- 74
FINE_WORD'("10"), -- 75
FINE_WORD'("10"), -- 76
FINE_WORD'("10"), -- 77
FINE_WORD'("10"), -- 78
FINE_WORD'("10"), -- 79
FINE_WORD'("01"), -- 80
FINE_WORD'("01"), -- 81
FINE_WORD'("01"), -- 82
FINE_WORD'("01"), -- 83
FINE_WORD'("01"), -- 84
FINE_WORD'("01"), -- 85
FINE_WORD'("01"), -- 86
FINE_WORD'("01"), -- 87
FINE_WORD'("01"), -- 88
FINE_WORD'("10"), -- 89
FINE_WORD'("10"), -- 90
FINE_WORD'("10"), -- 91
FINE_WORD'("10"), -- 92
FINE_WORD'("10"), -- 93
FINE_WORD'("10"), -- 94
FINE_WORD'("11"), -- 95
FINE_WORD'("00"), -- 96
FINE_WORD'("00"), -- 97
FINE_WORD'("00"), -- 98
FINE_WORD'("00"), -- 99
FINE_WORD'("01"), -- 100
FINE_WORD'("01"), -- 101
FINE_WORD'("01"), -- 102
FINE_WORD'("01"), -- 103
FINE_WORD'("01"), -- 104
FINE_WORD'("01"), -- 105
FINE_WORD'("10"), -- 106
FINE_WORD'("10"), -- 107
FINE_WORD'("10"), -- 108
FINE_WORD'("10"), -- 109
FINE_WORD'("10"), -- 110
FINE_WORD'("10"), -- 111
FINE_WORD'("00"), -- 112
FINE_WORD'("00"), -- 113
FINE_WORD'("00"), -- 114
FINE_WORD'("00"), -- 115
FINE_WORD'("01"), -- 116
FINE_WORD'("01"), -- 117
FINE_WORD'("01"), -- 118
```

```
FINE_WORD'("01"), -- 119
FINE_WORD'("01"), -- 120
FINE_WORD'("01"), -- 121
FINE_WORD'("01"), -- 122
FINE_WORD'("10"), -- 123
FINE_WORD'("10"), -- 124
FINE_WORD'("10"), -- 125
FINE_WORD'("10"), -- 126
FINE_WORD'("10"), -- 127
FINE_WORD'("00"), -- 128
FINE_WORD'("00"), -- 129
FINE_WORD'("00"), -- 130
FINE_WORD'("00"), -- 131
FINE_WORD'("01"), -- 132
FINE_WORD'("01"), -- 133
FINE_WORD'("01"), -- 134
FINE_WORD'("01"), -- 135
FINE_WORD'("01"), -- 136
FINE_WORD'("01"), -- 137
FINE_WORD'("01"), -- 138
FINE_WORD'("01"), -- 139
FINE_WORD'("10"), -- 140
FINE_WORD'("10"), -- 141
FINE_WORD'("10"), -- 142
FINE_WORD'("10"), -- 143
FINE_WORD'("00"), -- 144
FINE_WORD'("00"), -- 145
FINE_WORD'("00"), -- 146
FINE_WORD'("00"), -- 147
FINE_WORD'("00"), -- 148
FINE_WORD'("01"), -- 149
FINE_WORD'("01"), -- 150
FINE_WORD'("01"), -- 151
FINE_WORD'("01"), -- 152
FINE_WORD'("01"), -- 153
FINE_WORD'("01"), -- 154
FINE_WORD'("01"), -- 155
FINE_WORD'("01"), -- 156
FINE_WORD'("01"), -- 157
FINE_WORD'("01"), -- 158
FINE_WORD'("01"), -- 159
FINE_WORD'("00"), -- 160
FINE_WORD'("00"), -- 161
FINE_WORD'("00"), -- 162
FINE_WORD'("00"), -- 163
FINE_WORD'("00"), -- 164
FINE_WORD'("00"), -- 165
FINE_WORD'("01"), -- 166
FINE_WORD'("01"), -- 167
FINE_WORD'("01"), -- 168
FINE_WORD'("01"), -- 169
FINE_WORD'("01"), -- 170
FINE_WORD'("01"), -- 171
FINE_WORD'("01"), -- 172
FINE_WORD'("01"), -- 173
FINE_WORD'("01"), -- 174
FINE_WORD'("01"), -- 175
FINE_WORD'("00"), -- 176
FINE_WORD'("00"), -- 177
FINE_WORD'("00"), -- 178
FINE_WORD'("00"), -- 179
FINE_WORD'("00"), -- 180
FINE_WORD'("00"), -- 181
FINE_WORD'("01"), -- 182
FINE_WORD'("01"), -- 183
FINE_WORD'("01"), -- 184
FINE_WORD'("01"), -- 185
FINE_WORD'("01"), -- 186
FINE_WORD'("01"), -- 187
FINE_WORD'("01"), -- 188
```

```
    FINE_WORD'("01"), -- 189
    FINE_WORD'("01"), -- 190
    FINE_WORD'("01"), -- 191
    FINE_WORD'("00"), -- 192
    FINE_WORD'("00"), -- 193
    FINE_WORD'("00"), -- 194
    FINE_WORD'("00"), -- 195
    FINE_WORD'("00"), -- 196
    FINE_WORD'("00"), -- 197
    FINE_WORD'("00"), -- 198
    FINE_WORD'("00"), -- 199
    FINE_WORD'("01"), -- 200
    FINE_WORD'("01"), -- 201
    FINE_WORD'("01"), -- 202
    FINE_WORD'("01"), -- 203
    FINE_WORD'("01"), -- 204
    FINE_WORD'("01"), -- 205
    FINE_WORD'("01"), -- 206
    FINE_WORD'("01"), -- 207
    FINE_WORD'("00"), -- 208
    FINE_WORD'("00"), -- 209
    FINE_WORD'("00"), -- 210
    FINE_WORD'("00"), -- 211
    FINE_WORD'("00"), -- 212
    FINE_WORD'("00"), -- 213
    FINE_WORD'("00"), -- 214
    FINE_WORD'("00"), -- 215
    FINE_WORD'("00"), -- 216
    FINE_WORD'("00"), -- 217
    FINE_WORD'("00"), -- 218
    FINE_WORD'("00"), -- 219
    FINE_WORD'("00"), -- 220
    FINE_WORD'("00"), -- 221
    FINE_WORD'("00"), -- 222
    FINE_WORD'("00"), -- 223
    FINE_WORD'("00"), -- 224
    FINE_WORD'("00"), -- 225
    FINE_WORD'("00"), -- 226
    FINE_WORD'("00"), -- 227
    FINE_WORD'("00"), -- 228
    FINE_WORD'("00"), -- 229
    FINE_WORD'("00"), -- 230
    FINE_WORD'("00"), -- 231
    FINE_WORD'("00"), -- 232
    FINE_WORD'("00"), -- 233
    FINE_WORD'("00"), -- 234
    FINE_WORD'("00"), -- 235
    FINE_WORD'("00"), -- 236
    FINE_WORD'("00"), -- 237
    FINE_WORD'("00"), -- 238
    FINE_WORD'("00"), -- 239
    FINE_WORD'("01"), -- 240
    FINE_WORD'("01"), -- 241
    FINE_WORD'("01"), -- 242
    FINE_WORD'("01"), -- 243
    FINE_WORD'("01"), -- 244
    FINE_WORD'("01"), -- 245
    FINE_WORD'("01"), -- 246
    FINE_WORD'("01"), -- 247
    FINE_WORD'("01"), -- 248
    FINE_WORD'("01"), -- 249
    FINE_WORD'("01"), -- 250
    FINE_WORD'("01"), -- 251
    FINE_WORD'("01"), -- 252
    FINE_WORD'("01"), -- 253
    FINE_WORD'("01"), -- 254
    FINE_WORD'("01")  -- 255
);

        signal      DO, DD    : std_logic_vector(1 downto 0) := (others=>'0');
```

```
BEGIN

   READ_FILE:
   PROCESS(CK)
   BEGIN
                  if CK'event and (CK = '1') then
                         if CSN = '0' then
                                DO        <= FINE_ROM(to_integer(unsigned(A)));
                         end if;
                  end if;
   END PROCESS;

   DD      <= DO after 2 ns;

         --Q         <= DD when OEN = '0' else (others=>'Z');
         Q           <= DD;

END behave;
```

16. CODE_LATCH Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY CODE_LATCH IS
PORT(   NRESET          : in    std_logic;
        CLOCK           : in    std_logic;
        cin             : in    std_logic_vector(63 downto 1);
        bin             : in    std_logic_vector(3 downto 0);
        -- Wave output
        code            : out   std_logic_vector(63 downto 1);
        b               : out   std_logic_vector( 3 downto 0)
        );
END CODE_LATCH;

ARCHITECTURE RTL OF CODE_LATCH IS
BEGIN
        LATCH_REG:
        process(NRESET, CLOCK)
        begin
                if NRESET = '0' then
                        code    <= (others=>'0');
                        b               <= (others=>'0');
                elsif CLOCK'event and (CLOCK = '1') then
                        code    <= cin;
                        b               <= bin;
                end if;
        end process;
END RTL;
```

17. DAC Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

ENTITY DAC IS
PORT(   PWRDN           : in    std_logic;
        code            : in    std_logic_vector(63 downto 1);
        b               : in    std_logic_vector( 3 downto 0);
        DAC_OUT         : out   std_logic
        );
END DAC;

ARCHITECTURE RTL OF DAC IS
        signal  IDAC    : integer;
BEGIN
        DAC_PROC:
        process(PWRDN, code, b)
        variable current : integer := 0;
        begin
                if PWRDN = '0' then -- powerdown mode
                        IDAC    <= 0;
                else
                        if code'event or b'event then
                                current := 0;
                                for i in 1 to 63 loop
                                        if code(i) = '1' then
                                                current := current + 1;
                                        end if;
                                end loop;
                                current := current * 16;
                                IDAC <= current + to_integer(unsigned(b));
                        end if;
                end if;
        end process;
        DAC_out  <= '1' when IDAC > 511 else '0';
END RTL;
```

ภาคนวก ข.

รายละเอียดข้อมูลของชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

Technical Information

Features

+3.3 V power supply voltage

100 MHz Speed

Sine, random, ramp, and sawtooth output

Analog output with 10-bit amplitude resolution

10-bit digital output

Frequency and phase modulation with 32-bit and 12-bit resolution respectively

Power-down function

Parallel loading

Narrowband SFDR > 72 dB

400 mW power consumption at 100 MHz

64-pin PLCC

Applications

Test equipment

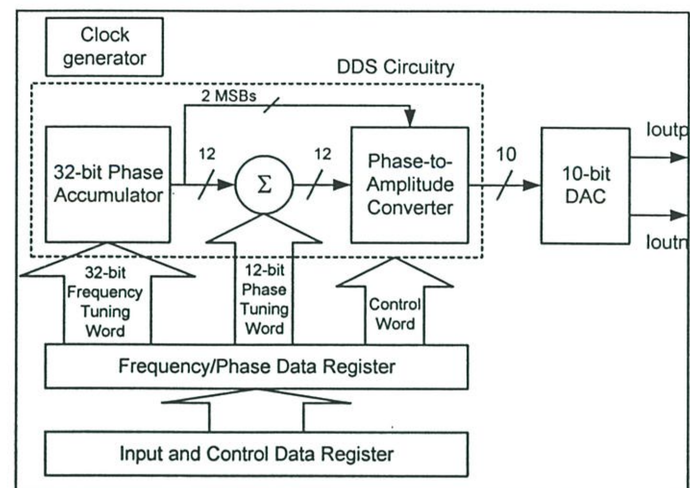
Slow Sweep Generator

DDS Tuning

Digital Modulation

General Description

This DDS device is a numerically controlled oscillator employing a phase accumulator, a sine look-up table and a D/A converter integrated on a single CMOS chip. Clock rates up to 100 MHz are supported with 3.3 V power supply voltage. Modulation capabilities are provided for phase modulation and frequency modulation. Frequency accuracy can be controlled to one part in 0.25 billion. Modulation is effected by loading registers through the serial interface. The SIN ROM can be bypassed so that a linear up/down ramp is output from the DAC. Also, if a clock output is required, the sign data bit can be output. The digital section is driven by an on-board regulator. The analog and digital sections are independent and can be run from different power supplies. A power-down pin allows external control of a power-down mode. In addition, sections of the device, which are not being used can be powered down to minimise the current consumption. For example, the DAC can be powered down when a clock output is being generated. The part is available in a 64-pin PLCC package.



รูปที่ ข.1 โครงสร้างของชิปต้นแบบวงจรรวมเพื่อสังเคราะห์ความถี่

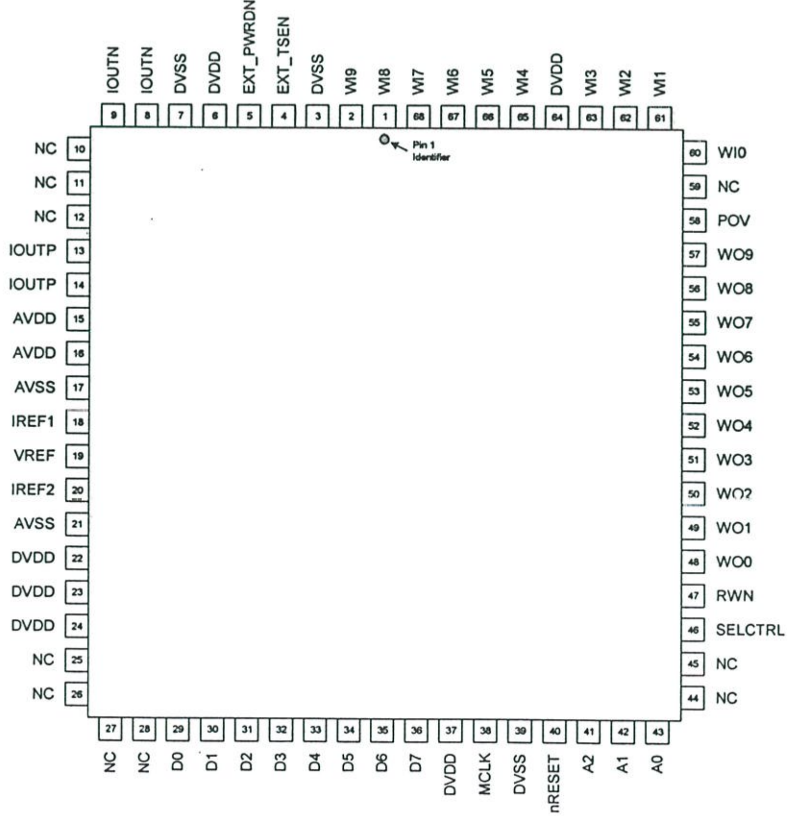
Specifications

(DV_{DD} = AV_{DD} = 3.3 V, AGND = DGND = 0 V, R_{ext} = 9.1 kΩ, R_L = 25 Ω for Ioutn and Ioutp unless stated otherwise)

Parameter	*DDS*	*Units*	*Test conditions/comments*
DAC Specifications			
Resolution	10	Bits	
Update rate (f_{max})	100	MSPS max	
I_{out} Full Scale	35.16	mA max	
Output compliance	1.3	V max	
DC Accuracy			
Integral Nonlinearity	±1	LSB typ	
Differential Nonlinearity	±0.5	LSB typ	

Parameter	*DDS*	*Units*	*Test conditions/comments*
DDS Specifications			
Dynamic Specifications			
Signal to Noise Ratio	50	dB min	$f_{CLK} = f_{MAX}$, f_{OUT} = 3 kHz
Total Harmonic Distortion	-53	dBc max	$f_{CLK} = f_{MAX}$, f_{OUT} = 3 kHz
Spurious Free Dynamic Range (SFDR)			
Wideband (± 2 MHz)	50	dBc min	
	55	dBc min	$f_{CLK} = f_{MAX}$, $f_{OUT} = f_{CLK}/3$
NarrowBand (± 50 kHz)	72	dBc min	$f_{CLK} = f_{MAX}$, f_{OUT} = 1 MHz
	75	dBc min	$f_{CLK} = f_{MAX}$, $f_{OUT} = f_{CLK}/3$
Clock Feedthrough	–55	dBc typ	$f_{CLK} = f_{MAX}$, f_{OUT} = 1 MHz
Power-Down Option	Yes		
Power Supplies			
AVDD	3.3 (±10%)	V typ	
DVDD	3.3 (±10%)	V typ	

Pin configuration



NC = No connect

Pin Description

Mnemonic	Function
Power Supply	
AVDD	Positive power supply for the analog section. A 0.1 μF decoupling capacitor should be connected between AVDD and AGND.
AGND	Analog ground.
DVDD	Positive power supply for the digital section. A 0.1 μF decoupling capacitor should be connected between DVDD and DGND.
DGND	Digital Ground.
Analog Signal and Reference	
IOUTN, IOUTP	Current Output. This is a high impedance current source. A load resistor should be connected between IOUTN, IOUTP and AGND.
IREF1	Full-Scale Adjust Control. A resistor (REXT) is connected between this pin and AGND. This determines the magnitude of the full-scale DAC current. The relationship between REXT and the fullscale current is as follows: $IOUT_{FULL\text{-}SCALE} = 256 \times V_{REFIN}/R_{EXT}$ $V_{REFIN} = 1.25\ V$ *nominal*, $R_{EXT} = 9.1\ k\Omega$ *typical*
IREF2	
VREF	Voltage reference input. This pin is connected to an external reference DC voltage, of which nominal value is 1.25 V.
Digital Interface and Control	
MCLK	Digital Clock Input. DDS output frequencies are expressed as a binary fraction of the frequency of CLK. The output frequency accuracy and phase noise are determined by this clock.
EXT_TSTEN	External control-enable input. When EXT_TSTEN goes high, DDS uses external data (WI9-WI0) for generating output (IOUTP and IOUTN). On the contrary when EXT_TSTEN goes low, DDS uses data from RegCtrl0 – RegCtrl3.
EXT_PWRDN	External power-down input. When EXT_PWRDN goes high, DDS enters power-down mode.
nRESET	Reset input. Low-level logic forces all registers to be reset.
SELCTRL	Parallel-loading selected input. SELCTRL must be held low for parallel loading.
RWN	Data is written to registers on the rising-edge of RWN.
A2-A0	Three-bit address. See Table 1 and Table 2 for register mapping.

Mnemonic	Function
D7-D0	Eight-bit data used for programming DDS registers.
WI9-WI0	Ten-bit digital inputs, used for external control. EXT_TSTEN must be held high when using WI9-WI0
WO9-WO0	Ten-bit digital outputs, used for debugging or controlling external DAC. These pins can be enabled or disabled by setting or clearing RegDAC(2) respectively.
POV	Phase-overflow output.

Direct addressing register map

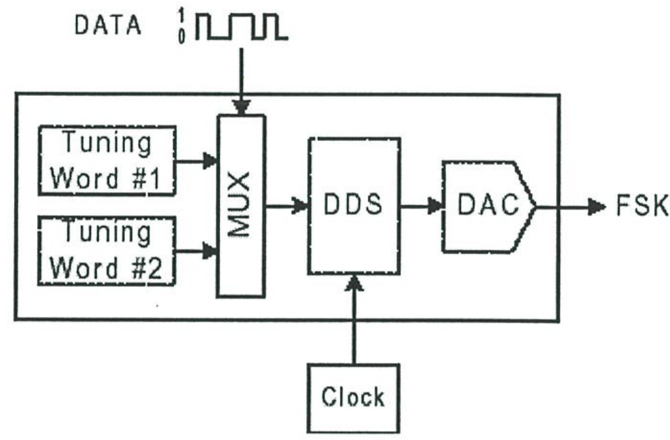
Addr (hex)	Register Name	Description		
01	RegINC0	Bit7 – Bit0 of frequency control word (W). The frequency control word is used for calculating output frequency. The frequency is calculated by an equation $f_{out} = (f_{clk} * W)/2^{32}$ where fout is output frequency fclk is MCLK frequency W is 32-bit frequency control word		
02	RegINC1	Bit15 – bit8 of frequency control word (W).		
03	RegINC2	Bit23 – bit16 of frequency control word (W).		
04	RegINC3	Bit31 – bit24 of frequency control word (W).		
05	RegCTRL	DDS control register, used for controlling DDS operation.		
		Bit 7	nRST	DDS reset bit. 0: Reset 1: Not reset
		Bit 6	PWRDN	DAC power-down bit. 0: Power-down is active. 1: Power-down is inactive.
		Bit 5	ENABLE	DDS-enable bit. 0: DDS is disabled (off). 1: DDS is enabled (on).
		Bit 4	STOP	DDS stop bit. 0: DDS is not stopped. 1: DDS is stopped.

Addr (hex)	Register Name	Description		
		Bit 3..2	WTYPE	Select types of output waveform. 00: Sine output 01: Random output 10: Ramp output 11: Saw-tooth output
		Bit 1..0	STARTQ	Select starting quadrant. 00: Start at quadrant 1 01: Start at quadrant 2 10: Start at quadrant 3 11: Start at quadrant 4
06	RegINDEX	DDS index register, used for indirect addressing.		

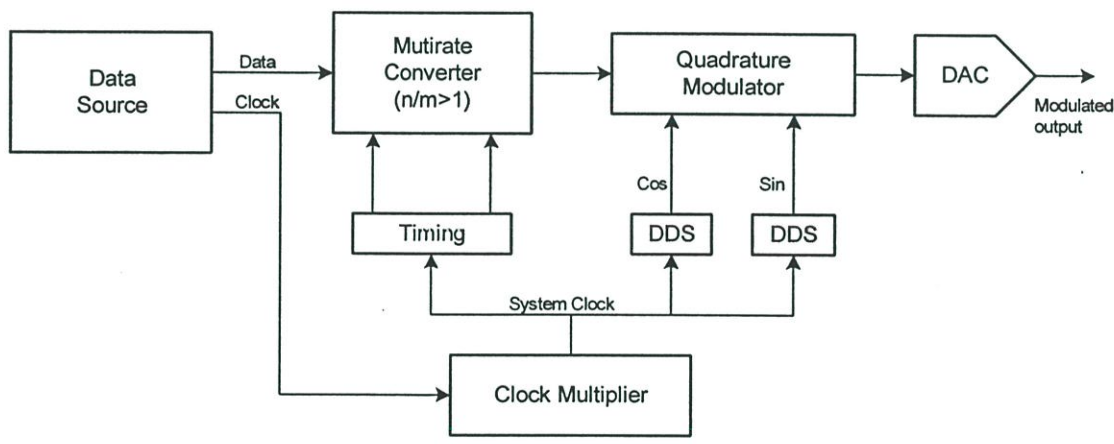
Indirect addressing register map

Addr (hex)	Register Name	Description		
00	RegDAC	DAC control register and two MSBs of DAC inputs.		
		Bit 7..4		Not used
		Bit 3	TSTEN	Selects between DDS and DAC mode. 0: DDS mode (output is generated from DDS circuit) 1: DAC mode (output is generated from external input, WI9 – WI0)
		Bit 2	Wout_EN	Wout-enable bit. Wout (WO9 – WO0) is a 10-bit digital signal, which is generated by DDS circuitry. 0: Wout is disabled. 1: Wout is enabled.
		Bit1..0	DAC_in	Two MSBs of DAC 10-bit digital inputs.
01	RegDAC_BIN	Eight LSBs of DAC 10-bit digital inputs.		
02	RegPHASE1	Four MSBs of 12-bit phase tuning word for phase modulation.		
03	RegPHASE0	Eight LSBs of 12-bit phase tuning word for phase modulation.		

ตัวอย่างการประยุกต์ใช้งานชิป DDS



รูปที่ ข.2 การเข้ารหัสเอฟเคสเค (FSK)



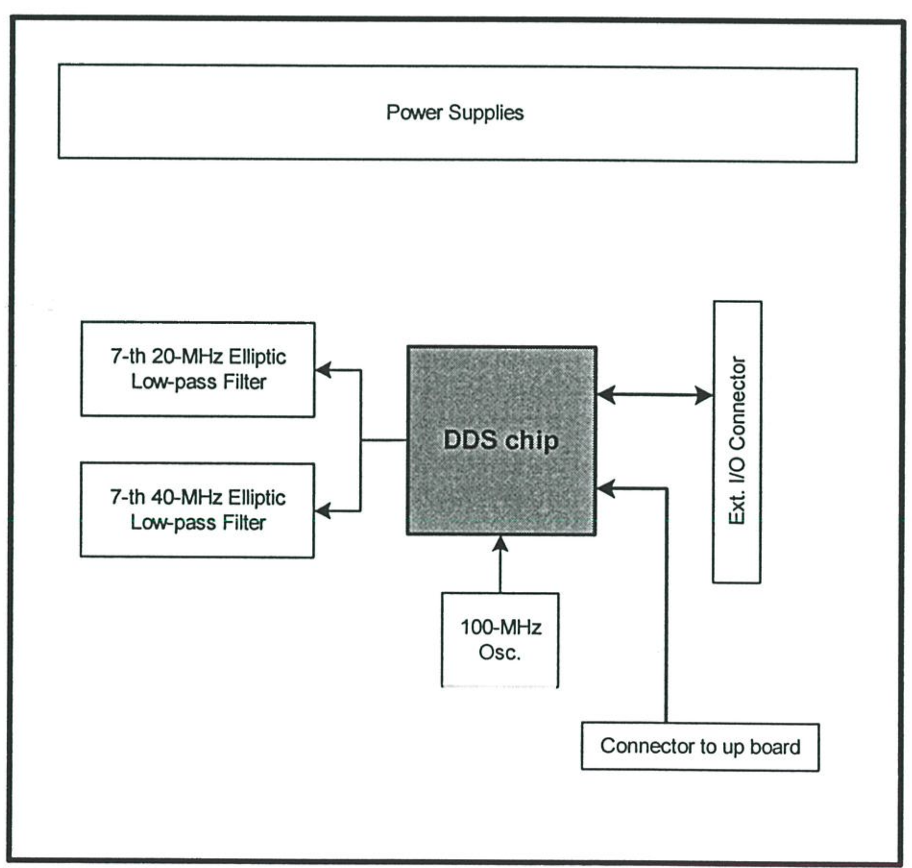
รูปที่ ข.3 การมอดดูเลทสัญญาณ แบบคิวเอเอ็ม (QAM)

ภาคผนวก ค.

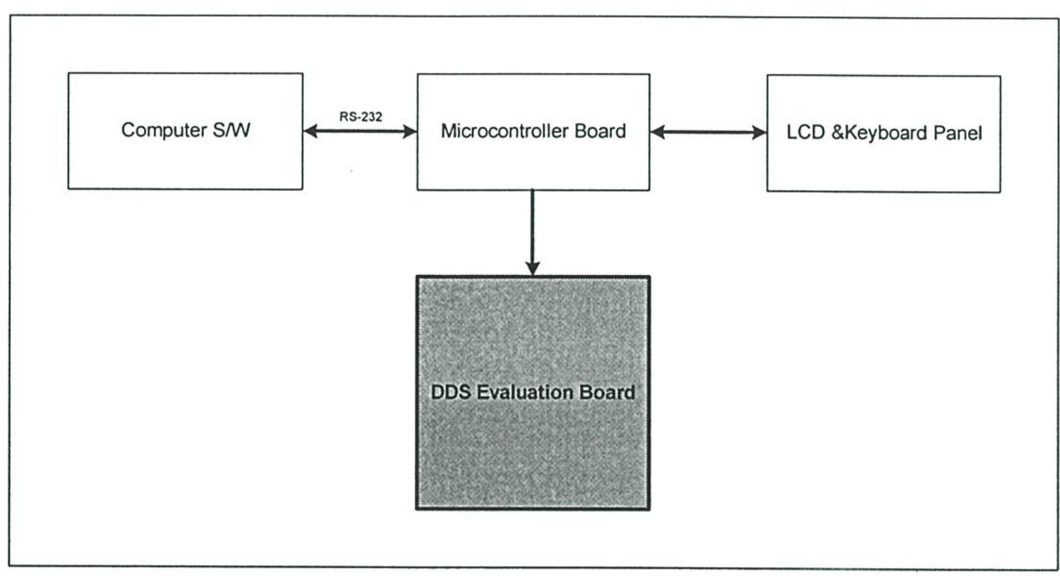
คู่มือการใช้งานบอร์ดต้นแบบชิปวงจรรวมเพื่อสังเคราะห์ความถี่แบบดิจิตอลที่ใช้หน่วยความจำ

DDS evaluation board ได้ถูกพัฒนาขึ้นเพื่อใช้เป็นเครื่องกำเนิดฟังก์ชั่น (Function generator) สำหรับสร้างสัญญาณรูปซายน์, แรมป์, ฟันเลื่อย และสัญญาณสุ่ม ที่ความถี่ 0-35 เมกะเฮิรตซ์ บล็อกไดอะแกรมและการใช้งานบอร์ดต้นแบบแสดงได้ดังรูปที่ ค.1 และ ค.2 ตามลำดับ

การควบคุมการทำงานของบอร์ดต้นแบบสามารถทำได้สองวิธีคือ (1) ใช้โปรแกรมคอมพิวเตอร์ควบคุมการทำงานผ่านทางพอร์ตอนุกรม (RS-232) หรือ (2) ควบคุมผ่านทาง LCD & keyboard panel ซึ่งจะกล่าวถึงรายละเอียดของการทำงานในแต่ละแบบในหัวข้อถัดไป



รูปที่ ค.1 DDS Evaluation board block diagram



รูปที่ ค.2 การใช้งาน DDS Evaluation board

1. การใช้โปรแกรมคอมพิวเตอร์ควบคุมการทำงานผ่านทางพอร์ตอนุกรม (RS-232)

บอร์ดต้นแบบนี้สามารถเชื่อมต่อกับโปรแกมคอมพิวเตอร์ผ่านทางพอร์ตอนุกรมเพื่อควบคุมการทำงาน หน้าต่าง (Windows) ของโปรแกรมที่ใช้ควบคุมการทำงานแสดงได้ดังรูปที่ ค.3

รูปที่ ค.3 หน้าต่างของโปรแกรมสำหรับควบคุม DDS Evaluation Board

ขั้นตอนการใช้งานโปรแกรมสำหรับควบคุมบอร์ดต้นแบบ (DDS Evaluation Board) มีดังนี้

- ตั้งค่าพอร์ตที่ต่อกับบอร์ดต้นแบบ โดย Click ที่ COM1 หรือ COM2

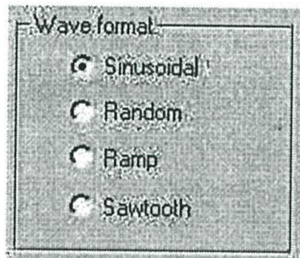
- ในช่อง Ref Clock Freq ให้ใส่ความถี่ของ Oscillator ที่ใช้ป้อนให้กับ DDS Chip หน่วยเป็น MHz

Ref Clock Freq 100 MHz

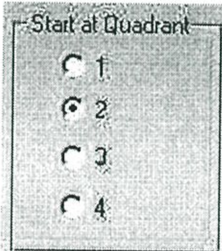
- ใส่ค่าของความถี่ Output ที่ต้องการในช่อง Output Freq (หน่วยเป็น MHz, ไม่เกิน 25 MHz)

Output Freq 10 MHz

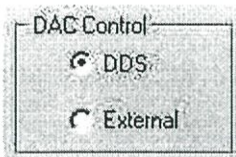
- เลือกรูปแบบของ Waveform ที่ต้องการ



- เลือก Starting quadrant



- ตั้งค่าในส่วนของ DAC Control ว่าจะควบคุมจาก DDS หรือจากภายนอก (External)



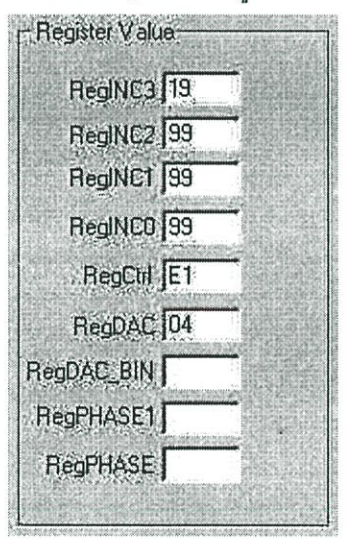
- ถ้าต้องการให้ Output 10-bit ของ DDS ออกมาภายนอกให้เลือกที่ Enable Wout

Enable Wout

- หลังจากตั้งค่าต่างๆ ข้างต้นเรียบร้อยแล้วถ้าต้องการดูค่าของ Register ต่างๆ ให้ Click ที่

View Register

ค่าของ Register จะถูกแสดงในส่วนของ Register Value ดังนี้

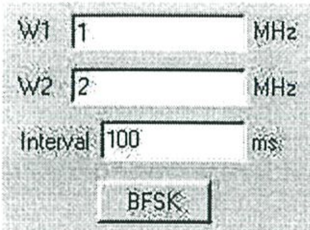


- ถ้าต้องการส่งค่าไปยัง DDS Evaluation Board ให้ Click ที่ Transmit ค่าของ Register ต่างๆ จะถูกส่งไปยัง DDS Evaluation Board เพื่อสร้างเป็นสัญญาณ Output ที่ต้องการ

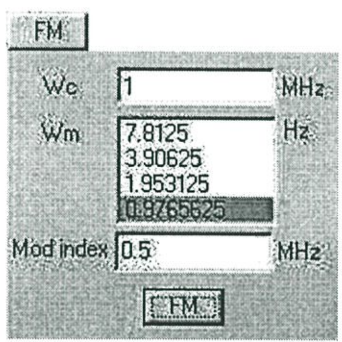
- ในขณะที่ DDS Chip กำลังทำงานอยู่ถ้าต้องการสั่ง Power-down ให้ Click ที่

Power Down

- ถ้าต้องการทำ Bi-phase Shift Keying (BFSK) ให้ใส่ข้อมูลของความถี่ที่ต้องการ (W1, W2) และช่วงเวลาในการเปลี่ยนระหว่าง W1 และ W2 เสร็จแล้ว Click ที่ BFSK

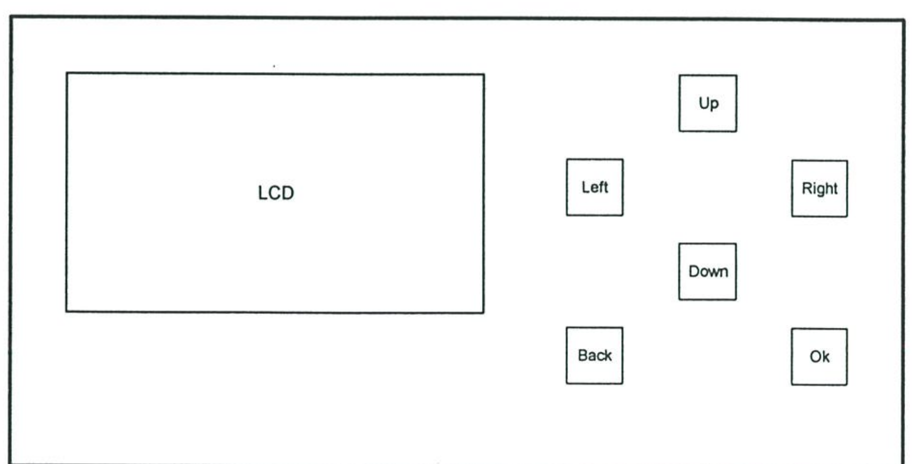


- ถ้าต้องการทำ Frequency Modulation (FM) ให้ใส่ค่า Carrier Frequency (Wc), เลือกค่า Modulating Frequency (Wm), ใส่ค่า Modulation index (Mod index) แล้ว Click ที่



2. การควบคุมผ่านทาง LCD & keyboard panel

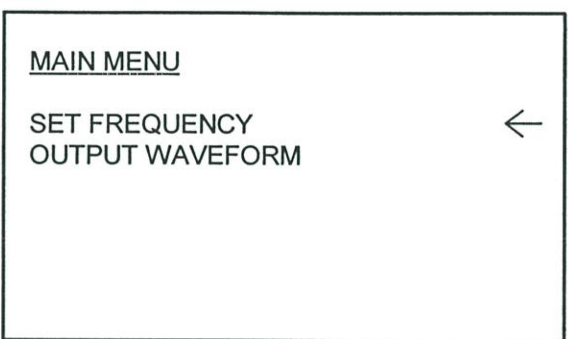
เราสามารถใช้ LCD & keyboard panel ควบคุมการทำงานของ DDS Evaluation board แทนการใช้โปรแกรมบนคอมพิวเตอร์ได้ โครงสร้างของ LCD & keyboard panel แสดงได้ดังรูปที่ ค.4



รูปที่ ค.4 LCD & Keyboard panel

ตัวเลือกต่างๆ จะถูกแสดงบนจอ LCD โดยมีปุ่มขึ้น (Up), ลง (Down), ซ้าย (Left) และ ขวา (Right) สำหรับเลื่อนตำแหน่ง ปุ่ม Ok สำหรับป้อนค่าตัวเลือก และปุ่ม Back สำหรับกลับไปยังหน้าจอหลัก (Main menu) การทำงานของแต่ละหน้าจออธิบายได้ดังนี้

1. หน้าจอหลัก (Main menu)



ใช้ปุ่มขึ้น-ลง เลื่อนตำแหน่งของลูกศรให้ตรงกับ Menu ที่ต้องการแล้วกดปุ่ม Ok เพื่อไปยังหน้าจอ Set Frequency หรือหน้าจอ Output Waveform

2. หน้าจอ SET FREQUENCY

SET FREQUENCY

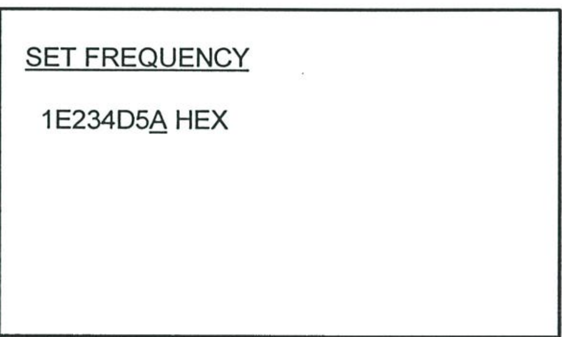
00000000 HEX

หน้าจอนี้ใช้สำหรับป้อนค่า 32 บิท ของ Frequency Control Word โดยค่าที่ใส่จะอยู่ในรูปเลขฐาน 16 มีความสัมพันธ์ตามสมการ

$$W = \frac{f_{out} * 2^{32}}{f_{clk}}$$

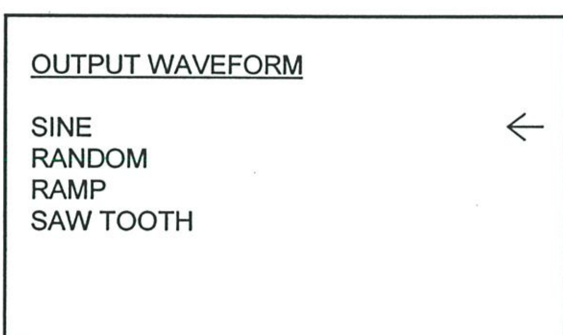
โดย W คือ Frequency Control Word, f_{out} คือความถี่ของสัญญาณเอาท์พุทที่ต้องการ และ f_{clk} คือความถี่ของ Oscillator ที่ใช้

ใช้ปุ่มซ้าย-ขวา เลื่อน Cursur ไปยังตำแหน่งที่ต้องการ แล้วใช้ปุ่มขึ้น-ลง เพื่อเพิ่มหรือลดค่าในตำแหน่งที่ Cursur อยู่ ตัวอย่างเช่น



เสร็จแล้วกดปุ่ม Ok เพื่อส่งค่าที่ตั้งไปยัง DDS Chip เพื่อสร้างเป็นสัญญาณ Output ต่อไป และถ้าต้องการกลับไปหน้าจอหลักให้กดปุ่ม Back

3. หน้าจอ OUTPUT WAVEFORM



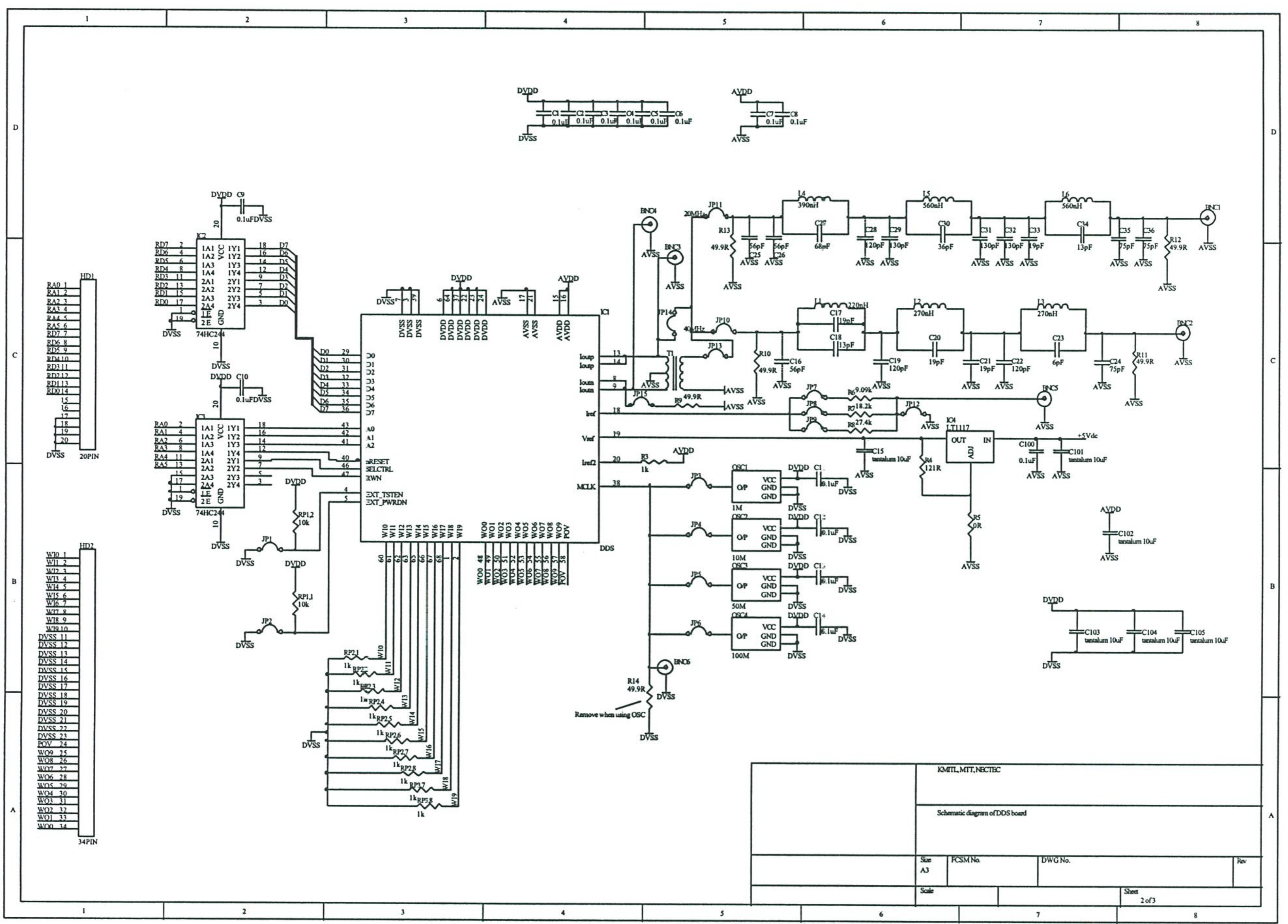
หน้าจอนี้จะแสดงรูปแบบของ Output waveform ที่ถูกเลือกไว้ โดยลูกศรจะชี้ไปยังตำแหน่งของ Waveform ที่กำลังถูกใช้งานอยู่ ถ้าต้องการเปลี่ยน Waveform ให้กดปุ่มขึ้น-ลงเพื่อเลื่อนลูกศรไปยังตำแหน่งของ Waveform ที่ต้องการ แล้วกดปุ่ม Ok ตัวอย่างเช่นต้องการเลือก Waveform ให้เป็น RAMP จะได้หน้าจอดังนี้

OUTPUT WAVEFORM

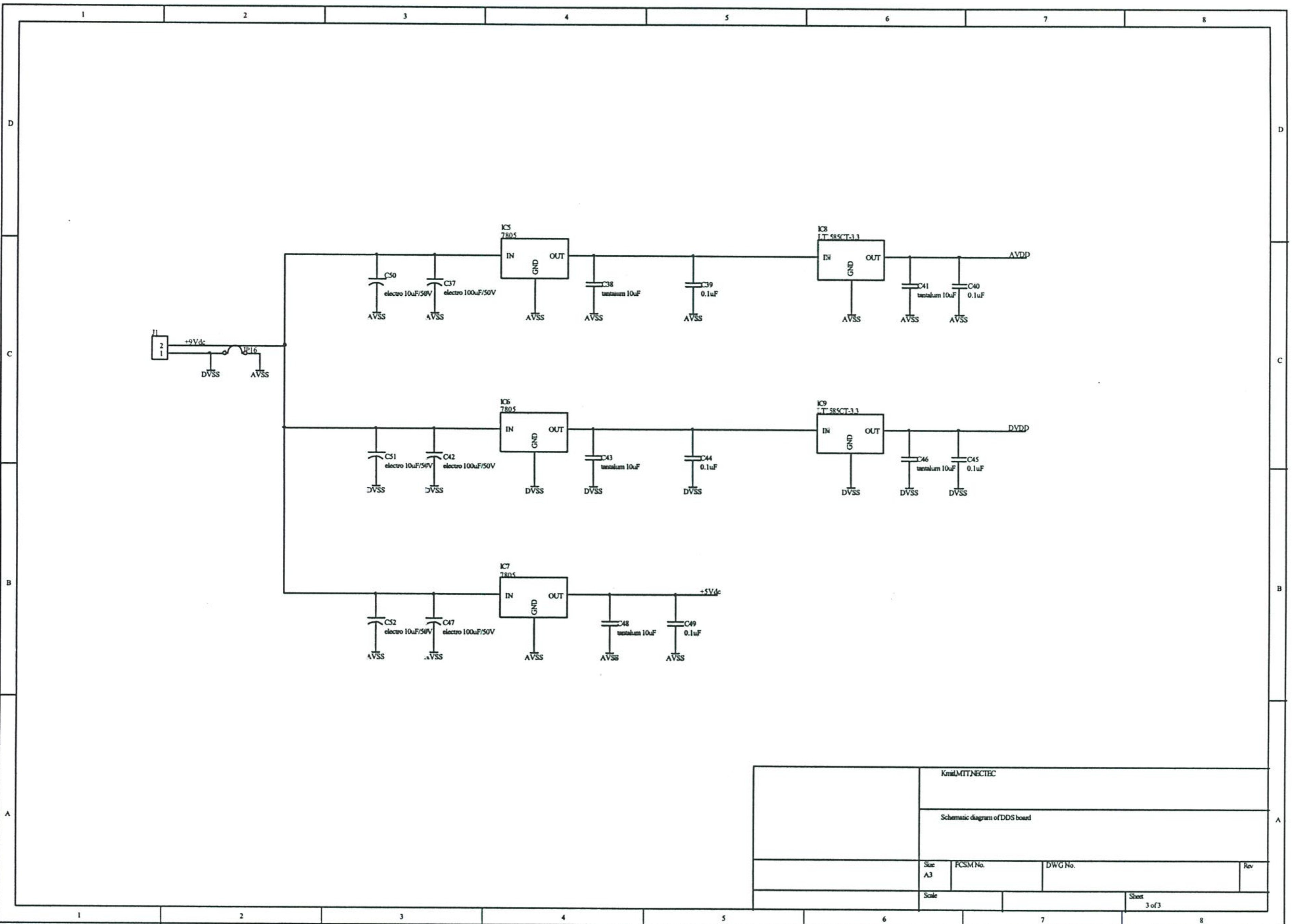
SINE
RANDOM
RAMP ←
SAW TOOTH

ถ้าต้องการกลับไปหน้าจอหลักให้กดปุ่ม Back

วงจรของ DDS Evaluation Board

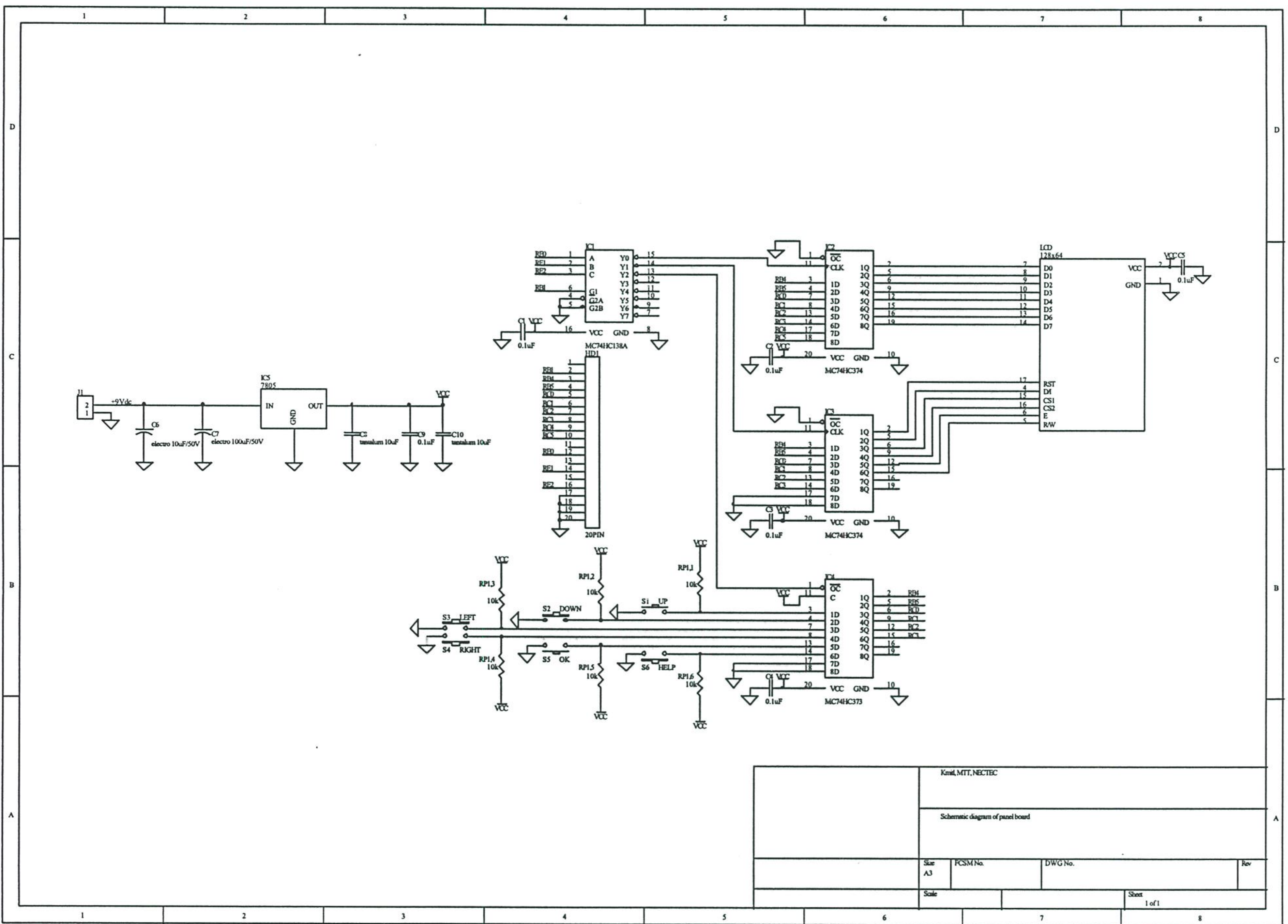


KMITL, MTT, NECTEC
Schematic diagram of DDS board
Size
A3
FCSM No.
DWG No.
Rev
Scale
Sheet
2 of 3
DVDD
DVSS
AVDD
AVSS
HD1
20PIN
HD2
34PIN
IC2
74HC244
IC3
74HC244
IC1
DDS
nRESET
SELCTRL
RWN
EXT_TSTEN
EXT_PWRDN
MCLK
Iref
Vref
Iref2
IC4
LT1117
OUT
IN
ADJ
+5Vdc
T1
BNC1
BNC2
BNC3
BNC4
BNC5
BNC6
20MHz
40MHz
L4
390nH
L5
560nH
L6
560nH
L1
220nH
L2
270nH
L3
270nH
OSC1
OSC2
OSC3
OSC4
1M
10M
50M
100M
O/P
VCC
GND
R14
49.9R
Remove when using OSC
C102
tantalum 10uF
C103
C104
C105
tantalum 10uF



IC5
7805
IN
OUT
GND
IC8
LT1585CT-3.3
AVDD
C50
electro 10uF/50V
C37
electro 100uF/50V
C38
tantalum 10uF
C39
0.1uF
C41
tantalum 10uF
C40
0.1uF
AVSS
J1
+9Vdc
JP16
DVSS
IC6
7805
IC9
LT1585CT-3.3
DVDD
C51
electro 10uF/50V
C42
electro 100uF/50V
C43
tantalum 10uF
C44
0.1uF
C46
tantalum 10uF
C45
0.1uF
IC7
7805
+5Vdc
C52
electro 10uF/50V
C47
electro 100uF/50V
C48
tantalum 10uF
C49
0.1uF
Kmitl,MTT,NECTEC
Schematic diagram of DDS board
Size
A3
FCSM No.
DWG No.
Rev
Scale
Sheet
3 of 3

วงจรของ LCD & Keyboard Panel



IC1
MC74HC138A
IC2
MC74HC374
IC3
MC74HC374
IC4
MC74HC373
IC5
7805
LCD
128x64
HD1
20PIN
J1
+9Vdc
C6
electro 10uF/50V
C7
electro 100uF/50V
C8
tantalum 10uF
C9
0.1uF
C10
tantalum 10uF
VCC
GND
IN
OUT
S1 UP
S2 DOWN
S3 LEFT
S4 RIGHT
S5 OK
S6 HELP
RP1,1
RP1,2
RP1,3
RP1,4
RP1,5
RP1,6
10k
D0
D1
D2
D3
D4
D5
D6
D7
RST
DI
CS1
CS2
E
R/W
Kmitl, MTT, NECTEC
Schematic diagram of panel board
Size
A3
FCSM No.
DWG No.
Rev
Scale
Sheet
1 of 1

ภาคผนวก ง.

โค้ดวิเอชดีแอลของวงจรรวมสังเคราะห์ความถี่แบบดิจิตอลที่ใช้การประมาณค่าโพลีโนเมียลแทนการใช้หน่วยความจำ

1. DDFS Module

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

entity ddfs is
port (
        resetn          : in std_logic;
        clock           : in std_logic;
        w               : in std_logic_vector (31 downto 0);
        phase_2msb_out  : out std_logic_vector (1 downto 0);        -- MSB, MSB-1 of phase
        wave_out        : out std_logic_vector (9 downto 0)
);
end ddfs;

architecture rtl of ddfs is

component acm32
port (
        resetn    : in std_logic;
        clock     : in std_logic;
        w         : in std_logic_vector (31 downto 0);
        phase_out : out std_logic_vector (31 downto 0)
);
end component;

component mul10
port (
        resetn  : in std_logic;
        clock   : in std_logic;

        x       : in std_logic_vector (9 downto 0);
        y       : in std_logic_vector (9 downto 0);
        p       : out std_logic_vector (19 downto 0)
);
end component;

component a1
port (
        resetn  : in std_logic;
        clock   : in std_logic;
        x       : in std_logic_vector (19 downto 0);
        y       : out std_logic_vector (28 downto 0)
);
end component;

component a2
port (
        resetn  : in std_logic;
        clock   : in std_logic;
        x       : in std_logic_vector (19 downto 0);
        y       : out std_logic_vector (5 downto 0)
);
end component;

signal phase              : std_logic_vector (31 downto 0);
signal phase_2msb         : std_logic_vector (1 downto 0);  -- MSB, MSB-1 of phase
signal phase_9downto0     : std_logic_vector (9 downto 0);  -- phase(9 downto 0);

signal decoded_phase      : std_logic_vector (9 downto 0);

signal m2                 : std_logic_vector (19 downto 0);
```

```
signal m4               : std_logic_vector (19 downto 0);
signal a1_out           : std_logic_vector (28 downto 0);
signal a2_out           : std_logic_vector (5 downto 0);
signal ten_power9       : std_logic_vector (29 downto 0);
signal s                : unsigned (9 downto 0);
signal sum              : std_logic_vector (9 downto 0);

signal wave             : std_logic_vector (19 downto 0);

signal msb_d1,msb_d2,msb_d3,msb_d4,msb_d5       : std_logic;
signal m2_d1                                    : std_logic_vector (19 downto 0);
signal m_d1,m_d2,m_d3,m_d4                      : std_logic_vector (9 downto 0);

begin
phase_acm: acm32
port map (
        resetn      => resetn,
        clock       => clock,
        w           => w,
        phase_out => phase
);

phase_2msb <= phase (31 downto 30);
phase_9downto0 <= phase (29 downto 20);

phase_2msb_out <= phase_2msb;

delay_msb: process (resetn,clock)
begin
  if resetn = '0' then
    msb_d1 <= '0'; msb_d2 <= '0'; msb_d3 <= '0'; msb_d4 <= '0'; msb_d5 <= '0';
  elsif clock'event and clock = '1' then
    msb_d1 <= phase_2msb(1);
    msb_d2 <= msb_d1;
    msb_d3 <= msb_d2;
    msb_d4 <= msb_d3;
    msb_d5 <= msb_d4;
  end if;
end process delay_msb;

delay_m: process (resetn,clock)
begin
  if resetn = '0' then
    m_d1 <= (others => '0');
    m_d2 <= (others => '0');
    m_d3 <= (others => '0');
    m_d4 <= (others => '0');
  elsif clock'event and clock = '1' then
    m_d1 <= decoded_phase;
    m_d2 <= m_d1;
    m_d3 <= m_d2;
    m_d4 <= m_d3;
  end if;
end process delay_m;

m_power2: mul10
port map (
        resetn      => resetn,
        clock       => clock,
        x           => decoded_phase,
        y           => decoded_phase,
        p           => m2
);

delay_m2: process (resetn,clock)
begin
  if resetn = '0' then
    m2_d1 <= (others => '0');
  elsif clock'event and clock = '1' then
    m2_d1 <= m2;      -- (19 downto 12);
  end if;
end process delay_m2;
```

```
m_power4: mul10
port map (
            resetn      => resetn,
            clock       => clock,
            x           => m2 (19 downto 10),
            y           => m2 (19 downto 10),
            p           => m4
);

a1_m_power2: a1
port map (
            resetn      => resetn,
            clock       => clock,
            x           => m2_d1,               --m2 (19 downto 12),
            y           => a1_out
);

a2_m_power4: a2
port map (
            resetn      => resetn,
            clock       => clock,
            x           => m4,
            y           => a2_out
);

               --123456789012345678901234567890 => 30 bits
ten_power9 <= "111011100110101100101000000000";

process (resetn,clock)
begin
  if resetn = '0' then
    s <= (others => '0');
  elsif clock'event and clock = '1' then
    s <= unsigned(ten_power9(29 downto 20)) - unsigned(a1_out(28 downto 20)) + unsigned(a2_out);
  end if;
end process;

sum <= std_logic_vector (s (9 downto 0));

m_mul_m: mul10
port map (
            resetn      => resetn,
            clock       => clock,
            x           => sum,
            y           => m_d4,                --decoded_phase,
            p           => wave
);

decode_waveout: process (resetn,clock)
begin
  if resetn = '0' then
    wave_out <= (others => '0');
  elsif clock'event and clock = '1' then
    if msb_d5 = '0' then              -- quadrant 1 or 2
      wave_out <= '1' & wave (19 downto 11);
    else
      wave_out <= '0' & not (wave (19 downto 11));
    end if;
  end if;
end process decode_waveout;

--wave_out <= wave (19 downto 10);

end rtl;
```

2. ACM32 Module

```
-- 32-bit accumulator
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity acm32 is
port (
            resetn    : in std_logic;
            clock     : in std_logic;
            w         : in std_logic_vector (31 downto 0);
            phase_out : out std_logic_vector (31 downto 0)
);
end acm32;

architecture rtl of acm32 is
component adder8
port (
            in1       : in std_logic_vector (7 downto 0);
            in2       : in std_logic_vector (7 downto 0);
            cin       : in std_logic;

            res_out   : out std_logic_vector (7 downto 0);
            cout      : out std_logic
);
end component;

signal w_d1,w_d2,w_d3,w_d4,w_d5,w_d6: std_logic_vector (31 downto 0);
signal to_gnd: std_logic;

signal acc0_in,acc1_in,acc2_in,acc3_in: std_logic_vector (7 downto 0);
signal acc0_out,acc1_out,acc2_out,acc3_out: std_logic_vector (7 downto 0);
signal c_acc0,c_acc1,c_acc2,c_acc3: std_logic;
signal cl_acc0,cl_acc1,cl_acc2,cl_acc3: std_logic;

signal accp0_in,accp1_in,accp2_in,accp3_in: std_logic_vector (7 downto 0);
signal accp0_out,accp1_out,accp2_out,accp3_out: std_logic_vector (7 downto 0);
signal c_accp0,c_accp1,c_accp2,c_accp3: std_logic;
signal cl_accp0,cl_accp1,cl_accp2,cl_accp3: std_logic;

signal c_d1,c_d2,c_d3: std_logic;

signal pre0_1,pre0_2,pre0_3: std_logic_vector (7 downto 0);
signal pre1_1,pre1_2: std_logic_vector (7 downto 0);
signal pre2_1: std_logic_vector (7 downto 0);

begin

to_gnd <= '0';

delay_w: process (resetn,clock)
begin
  if resetn = '0' then
    w_d1 <= (others => '0');
    w_d2 <= (others => '0');
    w_d3 <= (others => '0');
    w_d4 <= (others => '0');
    w_d5 <= (others => '0');
    w_d6 <= (others => '0');
  elsif clock'event and clock = '1' then
    w_d1 <= w;
    w_d2 <= w_d1;
    w_d3 <= w_d2;
    w_d4 <= w_d3;
    w_d5 <= w_d4;
    w_d6 <= w_d5;
  end if;
end process delay_w;

delay_c: process (resetn,clock)
begin
  if resetn = '0' then
    c_d1 <= '0';
    c_d2 <= '0';
    c_d3 <= '0';
```

```
  elsif clock'event and clock = '1' then
    c_d1 <= c_acc3;
    c_d2 <= c_d1;
    c_d3 <= c_d2;
  end if;
end process delay_c;

acc0_unit: adder8
port map (
            in1 => w (7 downto 0), in2 => acc0_out,
            cin => to_gnd, res_out => acc0_in, cout => c_acc0);

acc1_unit: adder8
port map (
            in1 => w_d1 (15 downto 8), in2 => acc1_out,
            cin => cl_acc0, res_out => acc1_in, cout => c_acc1);

acc2_unit: adder8
port map (
            in1 => w_d2 (23 downto 16), in2 => acc2_out,
            cin => cl_acc1, res_out => acc2_in, cout => c_acc2);

acc3_unit: adder8
port map (
            in1 => w_d3 (31 downto 24), in2 => acc3_out,
            cin => cl_acc2, res_out => acc3_in, cout => c_acc3);

reg_acc0: process (resetn,clock)
begin
  if resetn = '0' then
    acc0_out <= (others => '0');
    cl_acc0 <= '0';
  elsif clock'event and clock = '1' then
--    if c_acc3 = '0' then
      acc0_out <= acc0_in;
      cl_acc0 <= c_acc0;
--    else
--      acc0_out <= (others => '0');
--      cl_acc0 <= '0';
--    end if;
  end if;
end process reg_acc0;

reg_acc1: process (resetn,clock)
begin
  if resetn = '0' then
    acc1_out <= (others => '0');
    cl_acc1 <= '0';
  elsif clock'event and clock = '1' then
--    if c_acc3 = '0' then
      acc1_out <= acc1_in;
      cl_acc1 <= c_acc1;
--    else
--      acc1_out <= (others => '0');
--      cl_acc1 <= '0';
--    end if;
  end if;
end process reg_acc1;

reg_acc2: process (resetn,clock)
begin
  if resetn = '0' then
    acc2_out <= (others => '0');
    cl_acc2 <= '0';
  elsif clock'event and clock = '1' then
--    if c_acc3 = '0' then
      acc2_out <= acc2_in;
      cl_acc2 <= c_acc2;
--    else
--      acc2_out <= (others => '0');
--      cl_acc2 <= '0';
--    end if;
  end if;
end process reg_acc2;
```

```
reg_acc3: process (resetn,clock)
begin
  if resetn = '0' then
    acc3_out <= (others => '0');
    cl_acc3 <= '0';
  elsif clock'event and clock = '1' then
--   if c_acc3 = '0' then
     acc3_out <= acc3_in;
     cl_acc3 <= c_acc3;
--   else
--     acc3_out <= (others => '0');
--     cl_acc3 <= '0';
--   end if;
  end if;
end process reg_acc3;

accp0_unit: adder8
port map (
          in1 => w_d3 (7 downto 0), in2 => accp0_out,
          cin => to_gnd, res_out => accp0_in, cout => c_accp0);

accp1_unit: adder8
port map (
          in1 => w_d4 (15 downto 8), in2 => accp1_out,
          cin => cl_accp0, res_out => accp1_in, cout => c_accp1);

accp2_unit: adder8
port map (
          in1 => w_d5 (23 downto 16), in2 => accp2_out,
          cin => cl_accp1, res_out => accp2_in, cout => c_accp2);

accp3_unit: adder8
port map (
          in1 => w_d6 (31 downto 24), in2 => accp3_out,
          cin => cl_accp2, res_out => accp3_in, cout => c_accp3);

reg_accp0: process (resetn,clock)
begin
  if resetn = '0' then
    accp0_out <= (others => '0');
    cl_accp0 <= '0';
  elsif clock'event and clock = '1' then
--   if c_acc3 = '0' then
     accp0_out <= accp0_in;
     cl_accp0 <= c_accp0;
--   else
--     accp0_out <= (others => '0');
--     cl_accp0 <= '0';
--   end if;
  end if;
end process reg_accp0;

reg_accp1: process (resetn,clock)
begin
  if resetn = '0' then
    accp1_out <= (others => '0');
    cl_accp1 <= '0';
  elsif clock'event and clock = '1' then
--   if c_d1 = '0' then
     accp1_out <= accp1_in;
     cl_accp1 <= c_accp1;
--   else
--     accp1_out <= (others => '0');
--     cl_accp1 <= '0';
--   end if;
  end if;
end process reg_accp1;

reg_accp2: process (resetn,clock)
begin
  if resetn = '0' then
    accp2_out <= (others => '0');
    cl_accp2 <= '0';
  elsif clock'event and clock = '1' then
--   if c_d2 = '0' then
```

```
      accp2_out <= accp2_in;
      cl_accp2 <= c_accp2;
--    else
--      accp2_out <= (others => '0');
--      cl_accp2 <= '0';
--    end if;
  end if;
end process reg_accp2;

reg_accp3: process (resetn,clock)
begin
  if resetn = '0' then
    accp3_out <= (others => '0');
    cl_accp3 <= '0';
  elsif clock'event and clock = '1' then
--    if c_d3 = '0' then
      accp3_out <= accp3_in;
      cl_accp3 <= c_accp3;
--    else
--      accp3_out <= (others => '0');
--      cl_accp3 <= '0';
--    end if;
  end if;
end process reg_accp3;

reg_pre0_1: process (resetn,clock)
begin
  if resetn = '0' then
    pre0_1 <= (others => '0');
  elsif clock'event and clock = '1' then
    pre0_1 <= accp0_out;
  end if;
end process reg_pre0_1;

reg_pre0_2: process (resetn,clock)
begin
  if resetn = '0' then
    pre0_2 <= (others => '0');
  elsif clock'event and clock = '1' then
    pre0_2 <= pre0_1;
  end if;
end process reg_pre0_2;

reg_pre0_3: process (resetn,clock)
begin
  if resetn = '0' then
    pre0_3 <= (others => '0');
  elsif clock'event and clock = '1' then
--    if c_accp3 = '0' then
      pre0_3 <= pre0_2;
--    else
--      pre0_3 <= (others => '0');
--    end if;
  end if;
end process reg_pre0_3;

reg_pre1_1: process (resetn,clock)
begin
  if resetn = '0' then
    pre1_1 <= (others => '0');
  elsif clock'event and clock = '1' then
    pre1_1 <= accp1_out;
  end if;
end process reg_pre1_1;

reg_pre1_2: process (resetn,clock)
begin
  if resetn = '0' then
    pre1_2 <= (others => '0');
  elsif clock'event and clock = '1'then
--    if c_accp3 = '0' then
      pre1_2 <= pre1_1;--accp1_out;
--    else
--      pre1_2 <= (others => '0');
--    end if;
```

```
  end if;
end process reg_pre1_2;

reg_pre2_1: process (resetn,clock)
begin
  if resetn = '0' then
    pre2_1 <= (others => '0');
  elsif clock'event and clock = '1' then
--    if c_accp3 = '0' then
      pre2_1 <= accp2_out;
--    else
--      pre2_1 <= (others => '0');
--    end if;
  end if;
end process reg_pre2_1;

phase_out <= accp3_out & pre2_1 & pre1_2 & pre0_3;

end rtl;
```

3. ADDER8 Module

```
-- 8-bit adder
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity adder8 is
port (      in1        : in std_logic_vector (7 downto 0);
            in2        : in std_logic_vector (7 downto 0);
            cin        : in std_logic;
            res_out    : out std_logic_vector (7 downto 0);
            cout       : out std_logic );
end adder8;

architecture rtl of adder8 is

signal in_cout: std_logic_vector (7 downto 0);
signal in_xor: std_logic_vector (7 downto 0);

begin
        in_xor(0) <= in1(0) xor in2(0);
        in_xor(1) <= in1(1) xor in2(1);
        in_xor(2) <= in1(2) xor in2(2);
        in_xor(3) <= in1(3) xor in2(3);
        in_xor(4) <= in1(4) xor in2(4);
        in_xor(5) <= in1(5) xor in2(5);
        in_xor(6) <= in1(6) xor in2(6);
        in_xor(7) <= in1(7) xor in2(7);

        res_out(0) <= not cin when in_xor(0) = '1' else cin;
        in_cout(0) <= cin when in_xor(0) = '1' else in1(0);

        res_out(1) <= not in_cout(0) when in_xor(1) = '1' else in_cout(0);
        in_cout(1) <= in_cout(0) when in_xor(1) = '1' else in1(1);

        res_out(2) <= not in_cout(1) when in_xor(2) = '1' else in_cout(1);
        in_cout(2) <= in_cout(1) when in_xor(2) = '1' else in1(2);

        res_out(3) <= not in_cout(2) when in_xor(3) = '1' else in_cout(2);
        in_cout(3) <= in_cout(2) when in_xor(3) = '1' else in1(3);

        res_out(4) <= not in_cout(3) when in_xor(4) = '1' else in_cout(3);
        in_cout(4) <= in_cout(3) when in_xor(4) = '1' else in1(4);

        res_out(5) <= not in_cout(4) when in_xor(5) = '1' else in_cout(4);
        in_cout(5) <= in_cout(4) when in_xor(5) = '1' else in1(5);

        res_out(6) <= not in_cout(5) when in_xor(6) = '1' else in_cout(5);
        in_cout(6) <= in_cout(5) when in_xor(6) = '1' else in1(6);

        res_out(7) <= not in_cout(6) when in_xor(7) = '1' else in_cout(6);
        in_cout(7) <= in_cout(6) when in_xor(7) = '1' else in1(7);

        cout <= in_cout(7);
end rtl;
```

4. MUL10 Module

```
-- 10x10 Baugh Wooley's multiplier
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

entity mul10 is
port (
        resetn   : in std_logic;
        clock    : in std_logic;

        x        : in std_logic_vector (9 downto 0);
        y        : in std_logic_vector (9 downto 0);
        p        : out std_logic_vector (19 downto 0)
);
end mul10;

architecture rtl of mul10 is

signal m: unsigned (19 downto 0);

begin

process (resetn,clock)

begin
  if resetn = '0' then
    m <= (others => '0');
  elsif clock'event and clock = '1' then
    m <= UNSIGNED(x) * UNSIGNED(y);
  end if;

end process;

  p <= std_logic_vector (m);

end rtl;
```

5. A1 Module

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

entity a1 is
port (
            resetn      : in std_logic;
            clock       : in std_logic;
            x           : in std_logic_vector (19 downto 0);
            y           : out std_logic_vector (28 downto 0)
);
end a1;

architecture rtl of a1 is
            signal      m: unsigned (28 downto 0);
            signal      k1: unsigned (27 downto 0);
            signal      k2: unsigned (26 downto 0);
            signal      k3: unsigned (22 downto 0);
begin
            k1 <= unsigned(x) & "00000000";
            k2 <= unsigned(x) & "0000000";
            k3 <= unsigned(x) & "000";
process (resetn,clock)
begin
  if resetn = '0' then
    m <= (others => '0');
  elsif clock'event and clock = '1' then
     m <= ('0' & k1) + k2 + k3;
    m <= ('0' & k1) + k2 - k3;
  end if;
end process;

  y <= std_logic_vector (m);

end rtl;
```

6. A2 Module

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

entity a2 is
port (
            resetn      : in std_logic;
            clock       : in std_logic;
            x           : in std_logic_vector (19 downto 0);
            y           : out std_logic_vector (5 downto 0)
);
end a2;

architecture rtl of a2 is
            signal      m: unsigned (5 downto 0);
            signal      k1: unsigned (4 downto 0);
            signal      k2: unsigned (3 downto 0);
            signal      k3: unsigned (1 downto 0);
begin
            k1 <= unsigned(x (19 downto 15));
            k2 <= unsigned(x (19 downto 16));
            k3 <= unsigned(x (19 downto 18));
process (resetn,clock)
begin
  if resetn = '0' then
    m <= (others => '0');
  elsif clock'event and clock = '1' then
    m <= ('0' & k1) + k2 - k3;
  end if;
end process;

  y <= std_logic_vector (m);

end rtl;
```

ผลงานวิจัยที่เกี่ยวข้องกับวิทยานิพนธ์และได้รับการตีพิมพ์

1. A. Thanachayanont, C. Meenakarn, T. Thongphuak and W. Sangnak. "Design and Implementation of a 100-MHz CMOS Direct Digital Synthesiser with a 10-Bit On-Chip Digital-to-Analog Converter." Proc. ISIC-2001, Singapore, Sept. 2001. pp. 55-58.
2. ชรัณ มีนกาญจน์ วิชัย แสงนาค ธนพร ทองเผือก และ อภินันท์ ธนชยานนท์ "การออกแบบวงจรรวมเพื่อสังเคราะห์ความถี่ขนาด 100 เมกะเฮิร์ตซ์ พร้อมด้วยวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิท โดยใช้เทคโนโลยีซีมอส." การประชุมวิชาการวิศวกรรมไฟฟ้าครั้งที่ 24 พฤศจิกายน 2544 ณ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง หน้า 962-967.
3. C. Meenakarn and A. Thanachayanont. "A Single-Chip CMOS Digitally Synthesized 0-35 MHz Agile Function Generator." Proc. ITC-CSCC, Phuket THAILAND, July 2002. pp. 1984-1987.
4. C. Meenakarn and A. Thanachayanont. "100-MHz CMOS Direct Digital Synthesizer with 10-Bit DAC." Proc. IEEE-APCCAS, Singapore, Sept. 2002. pp. 385-388.
5. C. Meenakarn and A. Thanachayanont. "A ROM-Less Direct Digital Frequency Synthesiser Using a Polynomial Approximation." to be appeared in Proc. VLSI-TSA, Hsinchu TAIWAN, April 2003.
6. C. Meenakarn and A. Thanachayanont. "A Sine-Output ROM-Less Direct Digital Frequency Synthesiser Using a Polynomial Approximation." to be appeared in Proc. IEEE-ISCAS, Bangkok THAILAND, May 2003.

Design and Implementation of a 100-MHz CMOS Direct Digital Synthesiser with a 10-bit On-chip Digital-to-Analog Converter

A. Thanachayanont, C. Meenakarn**, T. Thongphuak***, and W. Sangnak**

Research Center of Communications and Information Technology, and Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, THAILAND, E-mail: ktapinun@kmitl.ac.th

**Thai Microelectronics Center, National Electronics and Computer Technology Center, National Science and Technology Development Agency, Bangkok, THAILAND, E-mail: charan@nectec.or.th*

***Microelectronics Technology, Bangkok, THAILAND, E-mail: logic@bitsmart.com*

ABSTRACT

This paper presents the design and implementation of a CMOS integrated direct digital synthesiser (DDS) with a 10-bit on-chip digital-to-analog converter (DAC) using an n-well single-poly triple-metal 0.5-µm CMOS technology. Main features of the DDS chip include maximum clock frequency of 100-MHz at 3.3-V single supply voltage, 32-bit frequency tuning word resolution, 12-bit phase tuning word resolution, and an on-chip 10-bit DAC. The chip provides sinusoidal, triangular, sawtooth, square, and random waveforms with phase and frequency modulation, and power-down function. The complete chip occupies 10-mm^2 die area and dissipates 0.4 W at 100-MHz clock frequency.

1. INTRODUCTION

For many applications, direct digital synthesiser (DDS) easily provide many distinct advantages over the traditional frequency-agile phased-locked loop (PLL) based synthesiser, including fast settling time, digitally-controlled sub-hertz frequency and sub-degree phase resolutions, continuous-phase switching response, and low phase noise [1]. In recent years, DDS have enjoyed an increasingly significant role in wideband frequency generation due to advances in digital logic and digital-to-analog converter (DAC) IC technologies. The integration of a high-speed, high-performance DAC and DDS circuitry onto a single chip enables low-cost, high-performance, functionally-integrated, and small package-sized DDS products that target a wide range of applications such as time division multiple access/code division multiple access digital cellular systems and spread-spectrum wireless LANs.

A number of high-speed DDSs with on-chip DAC have been reported. In [2], a CMOS DDS implemented in a 1.0-µm process was reported with 50-MHz maximum operating clock frequency. In [3], a BiCMOS DDS implemented in 0.8-µm double-metal double-poly process was reported with the maximum operating clock frequency of 170 MHz. The gap in the maximum operating clock frequency of the two designs mainly arises from the difficulty in the design of high-speed DAC in CMOS technology. Therefore the design of high-speed DDS with an on-chip DAC in standard CMOS process still poses a real challenge. This paper describes the design and implementation of a 100-MHz DDS with a 10-bit on-chip DAC using an n-well, single-poly, triple-metal (1P3M) 0.5-µm CMOS technology.

2. DESIGN SPECIFICATIONS

The DDS reported in this paper is designed to generate various types of waveform for general video and wireless applications such as digital radios/modems, handheld test and measurement equipment, digital video/audio, baseband transmitters/receivers, PC-based instrumentation cards. The typical DDS architecture, shown in Figure 1, is employed. The phase accumulator word length was chosen to be 32 bits to achieve a frequency tuning resolution of 0.0233 Hz at the clock rate of 100 MHz. Only 14 of the most significant bits (MSBs) are used to calculate the sine-wave samples in order to reduce size and power dissipation of the table look-up ROM. Two MSBs of these 14 bits are used to decode the quadrant of the sinewave, while the remaining 12 bits are used to address the sinewave samples in the ROM. This achieves a 12-bit phase tuning resolution that yields a spurious performance due to the phase accumulator truncation of –72 dBc [5], which is well below the spur level of the on-chip 10-bit DAC at 100 MHz.

Phase modulation of the DDS is achieved by adding a 12-bit phase tuning word to the phase accumulator output before entering the waveform generator. The 10-bit digital output data is also available for applications that do not require the conversion to analog, such as tuneable digital bandpass filters, mixers for digital receivers, and real-time digital spectrum analysis. The design of the DDS chip, described in the following sections, is divided into two main parts: (1) the digital part including an I/O control interface, coarse and fine ROMs, a waveform synthesiser, a clock generator and a thermometer decoder, and (2) the analog part including a 10-bit DAC and a bias current generator.

3. DESIGN OF DDS CIRCUITRY

The DDS chip was first simulated at the system level to check its functionality. Then the DDS was designed and implemented by digital circuitry and simulated at the gate level to check timing constraints. For maximum conversion speed, the digital part contains 19 pipelined stages. The main building block is the waveform synthesiser that contains an increment, a 32-bit phase accumulator, and a 14-bit phase-to-amplitude converter. The modified Sunderland algorithm [4] is chosen for the phase-to-amplitude conversion. Latches are employed to strobe the digital control signals that are applied to the 10-bit DAC to eliminate signal skew and thus reducing output glitches.

The DDS can be controlled by an external controller via an 8-bit parallel interface for frequency update and waveform pattern. The maximum update determined by the speed of the control interface

and the 19 pipeline stages is 38 MS/s. The DDS is designed to generate 5 waveform patterns including sinusoidal, triangular, sawtooth, ramp, and random. The 10-bit amplitude for triangular, sawtooth, and ramp is generated directly from the phase register, while a linear feedback shift register is used to generate the random values. The 10b sinewave amplitude is generated from a sine lookup table, while the square wave can be generated from the MSB of the sinewave.

Using the well-known quarter-wave symmetry technique, the sine-wave samples for the full range of 2π are generated from the 0 to $\pi/2$ rads of sine information stored in the ROM [4]. The two MSBs of the phase accumulator are used to decode the quadrant of the sine function. Thus the MSB is used as the sign bit, while the next MSB controls whether the phase between 0 and $\pi/2$ should be increasing or decreasing. This reduces the capacity of the look-up table with the penalty of additional logic circuits required to generate the complements of the accumulator and the look-up table outputs. The width of the one-quadrant look-up table is reduced by compression of the quarter-wave sine information using the modified Sunderland algorithm [4], shown in Figure 2. The 12- bit of the phase address of the quarter of the sine wave is divided into three 4-bit fractions, i.e. $\phi = a + b + c$. Using proper sizes of a, b, and c, the sine function is approximately given by (1).

$$\sin(a + b + c) = \sin(a+b) + \cos(a)\sin(c) \qquad (1)$$

The coarse ROM, 2^8x8 bits, provides low resolution phase samples, while the fine ROM, 2^8x2 bits, gives additional phase resolution by interpolating between the low resolution samples. Note that this gives the total ROM compression ratio of 64:1.

4. DESIGN OF 10-BIT DAC

4.1 DAC Architecture

The integrated high-speed 10-bit DAC is implemented by using the 6/4 segmented current-steering architecture, in which the 6 MSBs of the digital binary inputs are thermometer-decoded to control 63 identical current sources, each having 16-LSB current weighting, and the remaining 4 LSBs control 4 binary-weighted current sources. The output currents of all current sources, which are either switched ON or OFF according to the digital input codes, are summed and driven into any resistive load in order to generate the required analog output voltage. The DAC comprises an array of current-steering cells, a bias generator, a segment decoder, and code latches as shown in Figure 3.

4.2 Current-Steering Cell

The schematic diagram of the current-steering cell of the 10-bit DAC is shown in Figure 4. The cascode current source (M_{p1} and M_{p2}) and the differential current-steering switches (M_{p3} and M_{p4}) are implemented by using p-channel MOSFETs in order to obtain the output voltage referred to ground. The cascode configuration suppresses the voltage fluctuation at the drain of the transistor M_{p1} thereby enhancing the output impedance of the current source. Long channel length is required for the transistor M_{p1} in order to achieve good accuracy and matching of the output current. The cascode transistor M_{p2} should have a short channel length in order to minimise the parasitic capacitance at the coupled sources of the differential switches, thus improving the switching speed during the output transition. The current from the cascode circuit is steered by the differential switches into two resistive loads, according to the true and complementary digital control signals, D1 and D1B. The differential switches employ the minimum channel length, and the width is optimised toward the switching speed while maintaining the accuracy of the output currents.

The two complementary digital control signals, D1 and D1B, are generated by the asymmetrical switch driver [6], M_{n1}-M_{n4}, in order to withdraw any possibility that both switches are turned OFF simultaneously due to inevitable delay in both control signals. The switch driver is simply a differential drive circuit with n-channel transistors for both pull-up and pull-down. The dimension of M_{n1}-M_{n4} must be chosen properly such that the fall-time of D1 and D1B (turning ON) will be faster than the rise-time (turning OFF). This ensures that both switches are never turned off simultaneously, but allowing a simultaneous turn-on for a short period of time. Turning-on both switches simultaneously will degrade the switching speed a little, but reducing glitches substantially. Furthermore, the HIGH level output is V_{TH} lower than V_{DD}, minimising the switching swing thus reducing the switching feedthrough to the output current. This intentional asymmetrical switching circuit guarantees that one switch is ON, even for a small skew. HSPICE simulation using the process parameters from the Alcatel's 0.5-µm CMOS technology suggests an extra delay of 0.2 ns due to the switch driver circuit.

The two complementary DAC output currents are uni-polar. Thus the voltages developed across the load resistors range from 0 V to a positive full-scale value. In practice, a center-tapped RF transformer is used to combine these two complementary currents and produce a bi-polar, zero DC-offset symmetrical output current. Additionally the transformer is beneficial in both providing the DAC outputs a suitable load resistance (via an impedance transformation) in order not to violate the DAC output voltage compliance, and coupling the DAC output currents to the reactive input of the subsequent LC lowpass filter. This also maintains a constant current flowing in each current source during switching, thus allowing fast settling time and reducing output glitches.

Linearity of the DAC is determined by the matching of current sources. In practice, the total current error is due to the variation in threshold voltage and process geometry [7]. The dominant source of error is likely to be the threshold voltage variation provided that the devices are well above the minimum geometry. Thus large gate-source voltage and large LSB current weighting are required to reduce the mismatch. Measured mismatch data from the 0.5-µm CMOS technology suggest that the channel length of the p-channel MOSFET current source should be no shorter than 4 µm in order to satisfy the current matching requirements of the 10-bit DAC. Other circuit parameters are then calculated accordingly to meet the DAC specifications.

4.3 Bias Generator

The simplified schematic diagram of the bias generator that provides the reference current and the bias voltage to the DAC current sources is shown in Figure 5. The voltage V_{ref} is provided by an external bandgap voltage generator. The reference current distributed to the current source array is 16 times larger than the current of the unit current-steering cell (i.e. $4I_{LSB}$), this reduces

noise multiplication in the current mirror. The full-scale DAC output current is determined by an off-chip resistor R_{ref} and is given by the following equation, $I_{out,FS} = 320/R_{ext}$. The bias generator also includes the necessary power-down circuitry, which can be digitally controlled.

The complete 10-bit DAC combines the current-steering cells and the bias generator with the digital logic and thermometer decoder circuit, which are automatically synthesised from a VHDL description. Monte Carlo simulations showed that the complete DAC functioned satisfactorily under worst-case process parameters and variations in power supply voltage (+/- 10%), temperature (-30 to +70 °C), and threshold voltage (+/-5 %).

5. LAYOUT CONSIDERATIONS

The performance of the complete DDS is largely determined by the on-chip DAC. Thus much attention were paid to the layout and routing of the DAC including the bias generator which were done by hand, while all digital circuits were implemented by using automatic place-and-route with standard library cells. The total chip area is about 10 mm^2.

To obtain the optimum performance, the DAC cascode current sources are drawn in an array, isolating from their associated current-steering switches and asymmetrical switch drivers that are placed together in a separate block. This allows a compact and uniform layout of the array, reducing mismatches between the current sources. Isolation of sensitive analog circuits from digital switching noise is very important in mixed signal IC. Therefore analog and digital power supplies are separated, and double guard rings surround both the current source array and the switches and drivers block in order to prevent digital switching noise coupling to the DAC output currents.

The current source array is separated into 4 quadrants. Each of the 63 current sources with 16-LSB weighting for the 6 MSBs is implemented by 4 identical paralleled current sources of 4-LSB weighting; each of which is placed into one different quadrant. A two-dimensional common-centroid switching sequence is used in each of the four quadrants, which are mirrored with respect to the vertical and horizontal axes. The spatial symmetry of the array and the common-centroid switching scheme simultaneously compensate for both two-dimensional linear and parabolic systematic errors due to process, temperature, and electrical gradients over the current source array [8].

The 4-LSB cascode current source is laid out elegantly such that all cells can be 'butt-end' connected. This reduces both layout time and chip area, and is very well suited for design automation. The current sources in each quadrant are routed by using overlaid metal-1 and metal-2 grids. Wide metal-2 grid is also used for power supply routings in order to reduce the effect of 'IR' voltage dropped. The binary-weighted current sources for the 4 LSBs are located in the unused space in the centre of the array. Finally, the array is surrounded by dummy cells to reduce the edge effects.

6. CONCLUSION

This paper has described the design and implementation of a 100-MHz CMOS DDS including a 10-bit on-chip DAC. The DDS provides sinusoidal, triangular, sawtooth, square, and random waveforms with phase and frequency modulation. The complete chip occupies 10-mm^2 die area and dissipates 0.4 W at 100-MHz.. The DDS chip is in fabrication and preliminary measured results should be available at the time of presentation.

Acknowledgements: Financial support of the Tri-Partite research grant from the National Electronics and Computers Technology Centre (NECTEC), Thailand is gratefully acknowledged.

7. REFERENCES

[1] B. Goldberg, Digital techniques in frequency synthesis, McGraw Hill, 1996, New York.

[2] G. Chang, A. Rofougaran, M. Ku, A. A. Abidi, and H. Samueli, "A low-power CMOS digitally synthesised 0-13 MHz agile sinewave generator," in *Proc. Int. Solid-State Circuits Conf.*, 1994, pp. 32-33.

[3] J. Vankka, M. Waltari, M. Kosunen, and K. Halonen, "A direct digital synthesizer with an on chip D/A converter," *IEEE J. Solid-State Circuits*, vol. 33, no. 2, pp. 218-227, Feb. 1998.

[4] J. Vankka, "Methods of mapping from phase to sine amplitude in direct digital synthesis," in *Proc. 1996 IEEE Int. Frequency Control Symp.*, pp. 942-950.

[5] H. T. Nicholas, III, H. Samueli, and B. Kim, "The optimization of direct digital synthesizer performance in the presence of finite word length effects," *in Proc. 42nd Annu. Frequency Control Symp.*, 1988, pp. 357-363.

[6] T.-Y. Wu, C.-T. Jih, J.-C. Chen, and C.-Y. Wu, "A low glitch 10-bit 75-MHz CMOS video D/A converter," *IEEE J. Solid-State Circuits*, vol. 30, no. 1, pp. 68-72, Jan. 1995.

[7] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers, "Matching properties of MOS transistors," *IEEE J. Solid-State Circuits*, vol. 24, no. 5, pp. 1433-1440, Oct. 1989.

[8] A. Marques, *et al* "A 12b accuracy 300MSamples/s update rate CMOS DAC," in *ISSCC* Dig. Tech. Papers, 1998.

Table 1. Designed DDS chip specifications

IC Technology	n-well 1P3M 0.5-µm CMOS
Max. clock frequency	100 MHz at Vdd = 3.3 V
Max. output frequency	40 MHz (0.4 x 100 MHz)
Frequency tuning resolution	0.0233 Hz (at 100 MHz)
Phase tuning resolution	0.0879 degrees (12-bits)
Amplitude resolution	0.846 mV @ $R_L = 25\ \Omega$
Frequency switching time	190 ns (19 x 1/100MHz)
Power dissipation	0.4 W @ 100 MHz, 3.3V
Die area	10 mm^2

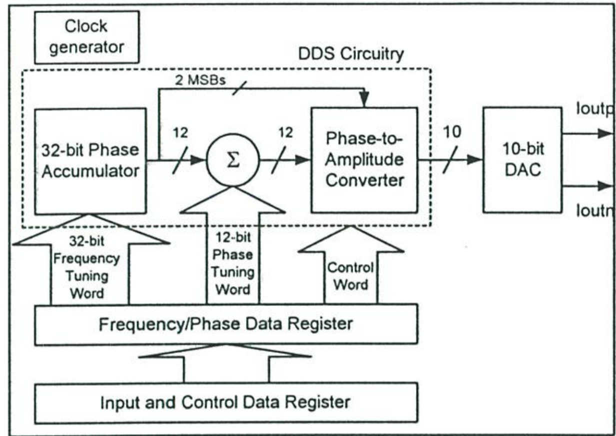


Figure 1. Architecture of the DDS chip

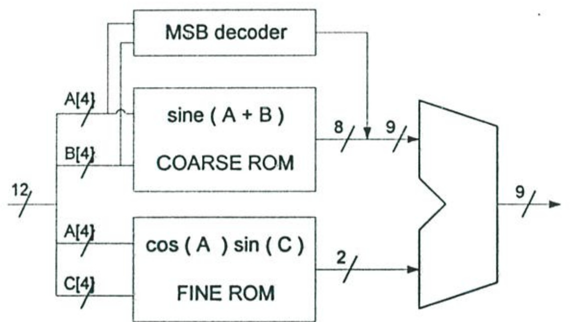


Figure 2. Modified Sunderland sine mapping

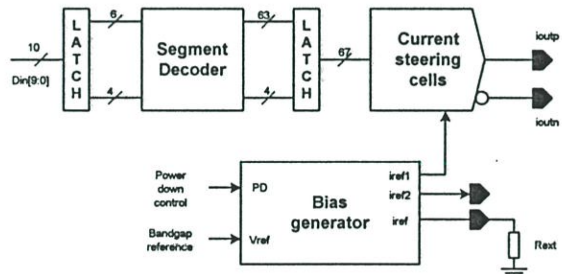


Figure 3. Block diagram of the 10-bit DAC

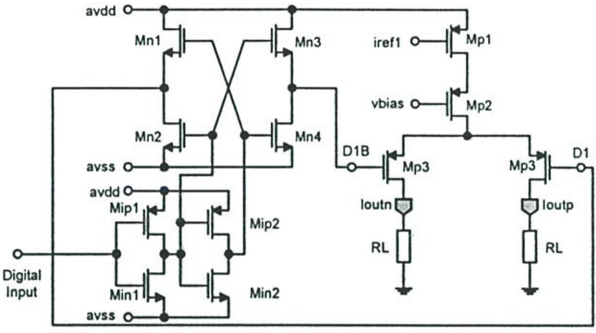


Figure 4. DAC's current-steering cell

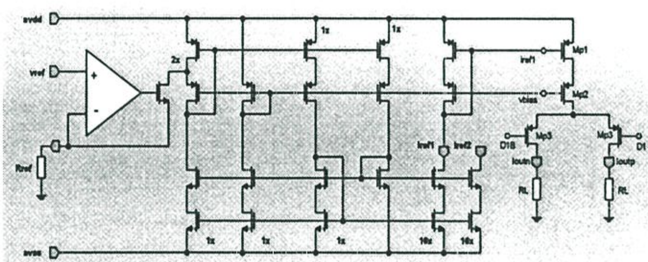


Figure 5. Simplified circuit diagram of the DAC's bias generator

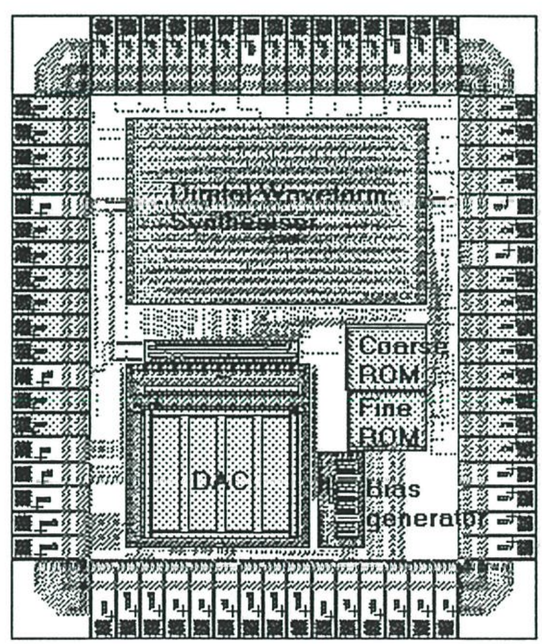


Figure 6. Layout of the complete DDS

การออกแบบวงจรรวมเพื่อสังเคราะห์ความถี่ขนาด 100 เมกะเฮิร์ตซ์ พร้อมด้วยวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิท โดยใช้เทคโนโลยีซีมอส

Design and Implementation of a 100-MHz CMOS Direct Digital Synthesiser With a 10-bit On-chip Digital-to-Analog Converter

ชรัณ มีนกาญจน์ [*] วิชัย แสงนาค [**] ธนพร ทองเผือก [***] และ อภินันท์ ธนชยานนท์ [**]

[*] งานวิจัยการออกแบบวงจรรวม ศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติ กรุงเทพฯ 10400, E-mail: charan@nectec.or.th

[**] ภาควิชาวิศวกรรมอิเล็กทรอนิกส์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง กรุงเทพฯ 10520,

E-mail: s3061315@kmitl.ac.th , ktapinun@kmitl.ac.th

[***] บริษัท ไมโครอิเล็กทรอนิกส์เทคโนโลยี (ประเทศไทย) จำกัด 33 เอสเอ็นซี ทาวเวอร์ ชั้น 13 สุขุมวิท 4 คลองเตย กรุงเทพฯ 10110,

E-mail: logic@bitsmart.com

บทคัดย่อ

บทความวิจัยนี้นำเสนอการออกแบบวงจรรวมเพื่อสังเคราะห์ความถี่ พร้อมด้วยวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิท โดยใช้เทคโนโลยีซีมอสขนาด 0.5 ไมครอน แบบบ่อเอ็น (N-well) ซิงเกิลโพลี (Single-poly) ทริปเปิลเมทัล (Triple-metal) โดยวงจรรวมสังเคราะห์ความถี่ที่ได้สามารถทำงานที่ความถี่สัญญาณนาฬิกาสูงสุด 100 เมกะเฮิร์ตซ์ ที่ไฟเลี้ยง 3.3 โวลท์ มีความละเอียดในการปรับความถี่ 32 บิท ความละเอียดในการปรับเฟส 12 บิท พร้อมทั้งได้รวมวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิท ไว้ด้วย สัญญาณที่วงจรรวมเพื่อสังเคราะห์ความถี่นี้สามารถสังเคราะห์ได้ ได้แก่ สัญญาณรูปซายน์ สัญญาณสามเหลี่ยม สัญญาณรูปฟันเลื่อย สัญญาณสี่เหลี่ยม และสัญญาณแบบสุ่ม คุณสมบัติอีกประการหนึ่งของวงจรรวมเพื่อสังเคราะห์ความถี่นี้คือการทำมอดูเลชั่นทางเฟส และความถี่ รวมไปถึงโหมดการทำงานแบบประหยัดพลังงาน (Power down) วงจรรวมสังเคราะห์ความถี่นี้ใช้พื้นที่ทั้งหมด 10 ตารางมิลลิเมตร ใช้พลังงาน 0.4 วัตต์ ที่ความถี่สัญญาณนาฬิกา 100 เมกะเฮิร์ตซ์

Abstract

This paper presents the design and implementation of a CMOS integrated direct digital synthesiser (DDS) with a 10-bit on-chip digital-to-analog converter (DAC) using an n-well single-poly triple-metal 0.5-μm CMOS technology. Main features of the DDS chip include maximum clock frequency of 100-MHz at 3.3-V single supply voltage, 32-bit frequency tuning word resolution, 12-bit phase tuning word resolution, and an on-chip 10-bit DAC. The chip provides sinusoidal, triangular, sawtooth, square, and random waveforms with phase and frequency modulation, and power-down function. The complete chip occupies 10-mm^2 die area and dissipates 0.4-W at 100-MHz clock frequency.

Keywords: CMOS, Direct digital synthesiser, DDS, Digital-to-analog converter, DAC, N-well, Single-poly, Triple-metal.

1. คำนำ

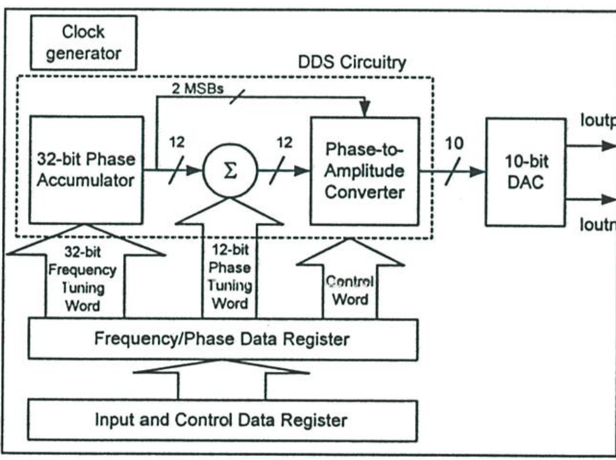
วงจรรวมเพื่อสังเคราะห์ความถี่แบบดีดีเอส (DDS) มีข้อได้เปรียบวงจรสังเคราะห์ความถี่ประเภทเฟสล็อกลูปหลายประการ เช่นเวลาที่ใช้ในการตั้งค่า (Settling time) ต่ำ การควบคุมความถี่และเฟสเป็นแบบดิจิตอลซึ่งจะให้ความละเอียดมากกว่า และมีสัญญาณรบกวนทางเฟสต่ำ [1] ในปัจจุบันวงจรดีดีเอสมีบทบาทสำคัญเป็นอย่างยิ่งในการสังเคราะห์สัญญาณความถี่ในย่านกว้าง (Wideband frequency) ซึ่งเป็นผลมาจากความก้าวหน้าทางเทคโนโลยีของวงจรดิจิตอล และการแปลงดิจิตอลเป็นอนาลอก ซึ่งการรวมวงจรแปลงดิจิตอลเป็นอนาลอกความเร็วสูง และวงจรดีดีเอสเข้าไว้ในชิพเดียวกันนั้นจะส่งผลให้ชิพที่ได้มีต้นทุนต่ำ ขนาดเล็ก อีกทั้งยังมีความเร็วและประสิทธิภาพสูง ซึ่งชิพที่ออกแบบนี้สามารถนำไปประยุกต์ใช้ในงานต่างๆ ได้ เช่น ระบบดิจิตอลเซลลูลาร์แบบ TDMA/CDMA ระบบเครือข่ายแบบไร้สาย (Spread-spectrum wireless LANs) ฯลฯ

ที่ผ่านมาได้มีการศึกษาวิจัยเกี่ยวกับวงจรดีดีเอสที่มีการรวมวงจรแปลงดิจิตอลเป็นอนาลอกกันอย่างแพร่หลาย ดังเช่นในงานวิจัยวงจรดีดีเอสความเร็ว 50 เมกะเฮิร์ตซ์ โดยใช้เทคโนโลยีซีมอสขนาด 1.0 ไมครอน [2] หรืองานวิจัยวงจรดีดีเอสความเร็ว 170 เมกะเฮิร์ตซ์ โดยใช้เทคโนโลยีไบซีมอสขนาด 0.8 ไมครอน [3] จะเห็นว่าวงจรรวมที่ได้จาก

งานวิจัยทั้งสองนี้มีความเร็วแตกต่างกันมากเนื่องจากการออกแบบวงจรดิจิตอลเป็นอนาลอกความเร็วสูงโดยใช้เทคโนโลยีซีมอสจะทำได้ยากกว่าเทคโนโลยีไบซีมอส ทำให้การออกแบบวงจรแปลงดิจิตอลเป็นอนาลอกความเร็วสูงด้วยเทคโนโลยีซีมอสนั้นยังคงเป็นสิ่งที่ท้าทายอยู่เสมอ ซึ่งในบทความนี้จะนำเสนอและอธิบายการออกแบบวงจรรวมเพื่อสังเคราะห์ความถี่ที่ความเร็ว 100 เมกะเฮิร์ตซ์ พร้อมด้วยวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิท โดยใช้เทคโนโลยีซีมอสขนาด 0.5 ไมครอน แบบบ่อเอ็น (N-well) ซิงเกิลโพลี (Single-poly) ทริปเปิลเมทัล (Triple-metal)

2. ข้อกำหนดการออกแบบ (Design Specifications)

วงจรรวมเพื่อสังเคราะห์ความถี่นี้ถูกออกแบบขึ้นเพื่อสังเคราะห์ความถี่หลายๆ รูปแบบ สำหรับนำไปประยุกต์ใช้ในงานประเภทวีดีโอ และการสื่อสารแบบไร้สาย เช่น วิทยุ/โมเด็มแบบดิจิตอล เครื่องมือวัดแบบมือถือ (Handheld) ระบบวีดีโอ/ออดิโอแบบดิจิตอล เครื่องรับ/ส่งสัญญาณในย่านเบสแบนด์ ฯลฯ



รูปที่ 1 สถาปัตยกรรมของวงจรดีดีเอส

เราสามารถแสดงสถาปัตยกรรมของวงจรดีดีเอสได้ดังรูปที่ 1 ซึ่งในการออกแบบนี้เรากำหนดให้เฟสแอคคิวมูเลเตอร์มีขนาด 32 บิท เพื่อให้ได้ความละเอียดในการปรับความถี่เท่ากับ 0.0233 เฮิร์ตซ์ ที่ความถี่สัญญาณนาฬิกา 100 เมกะเฮิร์ตซ์ เพื่อเป็นการลดขนาดและการใช้พลังงานของหน่วยความจำ (ROMs) เราจะใช้แค่เอ็มเอสบี (MSB) 14 บิท ในการคำนวณรูปสัญญาณ โดยเอ็มเอสบี 2 บิท จะถูกใช้สำหรับการกำหนดควอทแดรนท์ (Quadrant) ของสัญญาณ ส่วน 12 บิท ที่เหลือจะใช้สำหรับคำนวณรูปสัญญาณ ซึ่งจะทำให้ความละเอียดในการปรับเฟสของวงจรดีดีเอสนี้มีขนาด 12 บิท

การทำเฟสมอดูเลชั่นของวงจรดีดีเอสนี้สามารถทำได้โดยบวกเฟสจูนนิ่งเวิร์ด (Phase Tuning Word) ขนาด 12 บิท กับเอาต์พุตของเฟสแอคคิวมูเลเตอร์ แล้วส่งผลลัพธ์ให้กับตัวเปลี่ยนเฟสเป็นแอมปลิจูด (Phase-to-amplitude converter) เพื่อสร้างสัญญาณดิจิตอลขนาด 10 บิท โดยสัญญาณดิจิตอล 10 บิท นี้จะถูกเปลี่ยนเป็นสัญญาณอนาลอกด้วย วงจรแปลงดิจิจอลเป็นอนาลอกความเร็วสูงขนาด 10 บิท นอกจากนั้นเราได้เตรียมขาสำหรับสัญญาณดิจิตอลทั้ง 10 บิท นี้ไว้สำหรับแอปพลิเคชั่นที่ไม่ต้องการการแปลงสัญญาณเป็นอนาลอก ก็สามารถนำสัญญาณดิจิตอลนี้ไปใช้งานได้โดยตรง ตัวอย่างของแอปพลิเคชั่นที่ไม่ต้องการแปลงสัญญาณเป็นอนาลอก เช่น ตัวกรองแถบผ่านแบบดิจิตอล เป็นต้น

ในการออกแบบนี้เราแบ่งวงจรดีดีเอสออกเป็นสองส่วนหลักๆ คือ (1) ส่วนดิจิตอล ประกอบด้วยวงจรควบคุมการอินเทอร์เฟสหน่วยความจำแบบหยาบและละเอียด (Coarse and fine ROMs) วงจรสังเคราะห์รูปสัญญาณ (Waveform synthesiser) วงจรสร้างสัญญาณนาฬิกา (Clock generator) และเทอร์โมมิเตอร์ดีโคดเดอร์ (Thermometer decoder) และส่วนที่ (2) ส่วนอนาลอก ประกอบด้วยวงจรแปลงดิจิตอลเป็นอนาลอกความเร็วสูงขนาด 10 บิท (10-bit DAC) และวงจรสร้างกระแสไบอัส (Bias current generator) ซึ่งการทำงานในแต่ละส่วนจะกล่าวถึงในหัวข้อถัดไป

3. การออกแบบวงจรส่วนดิจิตอล

วงจรส่วนดิจิตอลทั้งหมดถูกออกแบบโดยใช้ภาษาวีเอชดีแอล (VHDL) เพื่ออธิบายการทำงาน จำลองการทำงาน และตรวจสอบความถูกต้อง หลังจากเขียนเสร็จแล้วใช้โปรแกรมจำลองการทำงาน (Simulator) จำลองการทำงานของวงจร จากนั้นจึงใช้โปรแกรมสังเคราะห์วงจร (Logic synthesis) สังเคราะห์วงจรออกมาเป็นเนทลิสต์ (Netlist) เพื่อนำไปจำลองการทำงานในระดับเกต (Gate-level simulation) ขั้นตอนต่อไปคือนำเนทลิสต์นี้ไปทำ Standard Cell Place & Route เพื่อสร้างเลย์เอาต์ของวงจรรวม (IC layout) ส่วนต่างๆ แล้วทำการวิเคราะห์ทางเวลา (Timing Analysis) เพื่อวิเคราะห์เวลาประวิง (Delay) ที่เกิดขึ้นในส่วนต่างๆ แล้วนำไปจำลองการทำงานระดับเวลา (Timing simulation) อีกครั้งหนึ่ง

ในการออกแบบวงจรส่วนดิจิตอลนี้ได้นำเทคนิคไพพ์ไลน์เข้ามาช่วยเพื่อให้ความเร็วในการทำงานสูงขึ้น โดยไพพ์ไลน์ที่ใช้มีทั้งหมด 19 สถานะ สำหรับวงจรเปลี่ยนเฟสเป็นแอมปลิจูด (Phase-to-amplitude converter) นั้น ใช้อัลกอริธึ่ม Modified Sunderland [4]

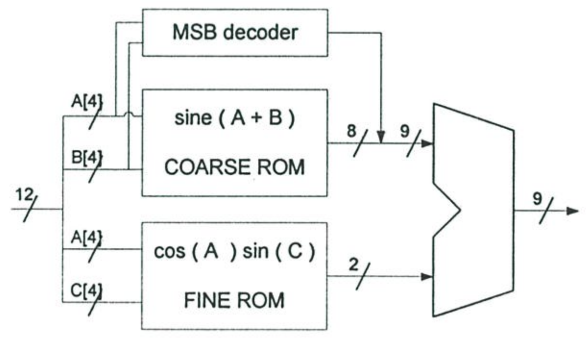
เนื่องจากสัญญาณที่ได้จากวงจรเปลี่ยนเฟสเป็นแอมปลิจูดจะมีเวลาประวิงเกิดขึ้น จึงต้องทำการแลทช์ (Latch) สัญญาณทั้งหมดเพื่อปรับให้มีเวลาประวิงเท่ากันและลดกลิทช์ (Glitch) ที่จะเกิดขึ้นก่อนเข้าสู่วงจรแปลงดิจิตอลเป็นอนาลอก

เราสามารถกำหนดค่าความถี่ และรูปแบบสัญญาณได้โดยเชื่อมต่อชิพดีดีเอสกับไมโครคอนโทรลเลอร์ภายนอกผ่านสัญญาณขนานขนาด 8 บิท โดยค่าความถี่สูงสุดในการปรับค่าเท่ากับ 38 เมกะแซมเพิลต่อวินาที ซึ่งถูกกำหนดโดยความเร็วของวงจรควบคุมการอินเทอร์เฟสและไพพ์ไลน์ 19 สถานะ ชิพดีดีเอสที่ออกแบบนี้สามารถสร้างรูปคลื่น

สัญญาณได้ทั้งหมด 5 รูปแบบ คือ (1) ไซน์ (2) สามเหลี่ยม (3) ฟันเลื่อย (Sawtooth) (4) Ramp และ (5) แบบสุ่ม สำหรับแอมปลิจูดขนาด 10 บิท ของสัญญาณรูปสามเหลี่ยม ฟันเลื่อย และ Ramp นั้น จะถูกสร้างโดยตรงจากรีจิสเตอร์เฟส แอมปลิจูดของสัญญาณแบบสุ่มถูกสร้างโดยรีจิสเตอร์เลื่อนย้อนกลับแบบตรง (Linear feedback shift register) ส่วนสัญญาณรูปไซน์นั้นจะสร้างแอมปลิจูดขนาด 10 บิท จากตารางค่าไซน์ (Sine lookup table) สำหรับแอมปลิจูดของสัญญาณสี่เหลี่ยม (Square) จะถูกสร้างโดยบิทเอ็มเอสบี (MSB) ของรูปคลื่นสัญญาณไซน์

ในการสร้างรูปคลื่นสัญญาณไซน์ของวงจรแปลงเฟสเป็นแอมปลิจูดนั้น เราแบ่งรูปคลื่นสัญญาณออกเป็น 4 ควอทแดรนท์ เพื่อลดขนาดของหน่วยความจำ (ROMs) โดยจะเก็บเฉพาะค่าตั้งแต่ 0 ถึง $\pi/2$ เรเดียน [4] แล้วใช้เอ็มเอสบีสองบิทแรกของเฟสแอคคิวมูเลเตอร์กำหนด ควอทแดรนท์ของคลื่นสัญญาณ โดยบิทเอ็มเอสบีแรกใช้กำหนดเครื่องหมาย ส่วนบิทเอ็มเอสบีถัดมาใช้กำหนดว่าค่าในช่วง 0 ถึง $\pi/2$ เรเดียนกำลังลดลงหรือเพิ่มขึ้น การแบ่งออกเป็น 4 ควอทแดรนท์นี้นอกจากจะลดขนาดของหน่วยความจำ (ROMs) แล้วยังเป็นการลดความซับซ้อนของวงจรอีกด้วย

เราสามารถลดขนาดของลุกอัพเทเบิลได้โดยใช้อัลกอริธึ่ม Modified Sunderland [4] ดังรูปที่ 2



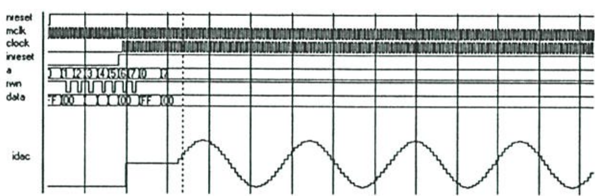
รูปที่ 2 Modified Sunderland Algorithm

จากรูปที่ 2 แอดเดรสของเฟสขนาด 12 บิท จะถูกแบ่งออกเป็น 3 ส่วน ส่วนละ 4 บิท เท่าๆ กัน คือ $\phi = a + b + c$ จากนั้นเราจะได้ฟังก์ชั่นไซน์ดังสมการที่ (1)

$$\sin(a + b + c) = \sin(a+b) + \cos(a)\sin(c) \qquad (1)$$

เทคนิคของอัลกอริธึ่ม Modified Sunderland นี้สามารถบีบอัดหน่วยความจำได้ 64:1

หลังจากที่ได้ทำการออกแบบวงจรส่วนดิจิตอลเสร็จแล้วได้ทดสอบการทำงานของวงจรโดยการจำลองการทำงาน โดยใช้ข้อมูลเวลาประวิงที่เกิดขึ้นในระดับเลย์เอาต์ ซึ่งให้ผลการทำงานที่ถูกต้อง รูปที่ 3 เป็นตัวอย่างของสัญญาณคลื่นไซน์ที่ได้จากวงจรส่วนดิจิตอล ก่อนเข้าสู่วงจรแปลงดิจิตอลเป็นอนาลอก

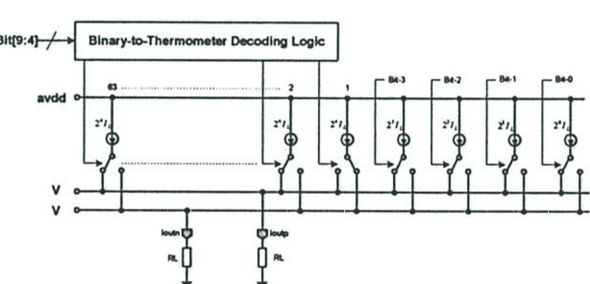


รูปที่ 3 สัญญาณเอาต์พุตที่ความถี่ 4.032 เมกะเฮิร์ตซ์

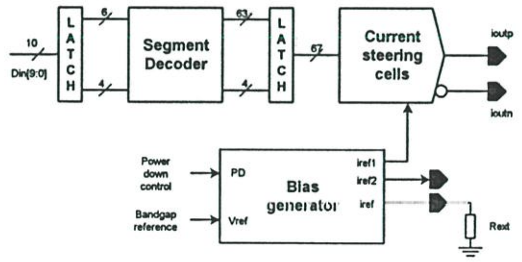
4. การออกแบบวงจรแปลงดิจิตอลเป็นอนาลอก

4.1 โครงสร้างวงจรแปลงดิจิตอลเป็นอนาลอก (DAC)

วงจรรวมความเร็วสูง แบบ 10-บิท ดิจิตอลเป็นอนาลอก โครงสร้างการทำงานแบบจ่ายกระแสที่เอาต์พุต 6/4 segment โดยแบ่งการทำงานเป็น 6 MSBs แบบ thermometer-decoded ควบคุมการจ่ายกระแส 63 ชุด ปริมาณกระแสชุดละ 16-LSB ส่วนของ 4 LSBs เป็นการทำงานแบบน้ำหนักกระแส(binary-weighted) จ่ายกระแส 4 ชุด มีปริมาณกระแสตามบิท เอาต์พุตของวงจรคือผลรวมของปริมาณกระแสที่ไหลจากแหล่งจ่ายกระแสแต่ละตัว จ่ายกระแสจากการ 'ON'หรือ 'OFF' ตามรหัสของสัญญาณอินพุตแสดงในรูปที่ 4



รูปที่ 4 การทำงานวงจรแปลงดิจิตอลเป็นอนาลอก (DAC)

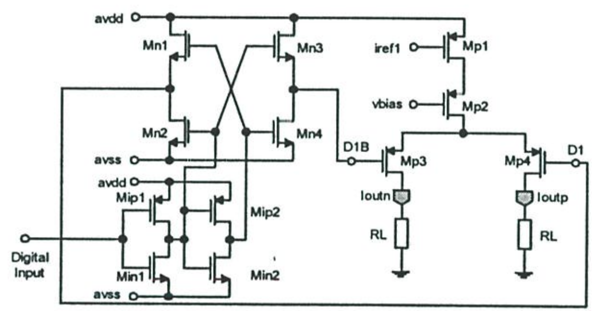


รูปที่ 5 โครงสร้างของวงจรแปลงดิจิตอลเป็นอนาลอก (DAC)

เอาต์พุตของวงจรจะจ่ายให้กับโหลดความต้านทาน ทำให้เกิดระดับแรงดัน ที่โหลดตามความต้องการ วงจรแปลงดิจิตอลเป็นอนาลอกประกอบด้วย วงจรแทนแหล่งจ่ายกระแส, วงจรสวิตช์กระแส, วงจรกำเนิดกระแส (bias generator), วงจรถอดรหัส (segment decoder), และวงจรค้างสัญญาณ (code latches) ซึ่งแสดงในรูปที่ 5

4.2 วงจรสวิตช์กระแส(Current-Steering Cell)

วงจรสวิตช์กระแสของวงจรแปลงดิจิตอลเป็นอนาลอก ขนาด 10-bit แสดงในรูปที่ 6 โดยวงจรแทนแหล่งจ่ายกระแส ใช้มอสเฟทต่อแบบคาสโคด (M_{p1} และ M_{p2}) และวงจรสวิตช์กระแส (M_{p3} และ M_{p4}) สร้างขึ้นจาก พี-แชนแนล มอสเฟท โดยสวิตช์ทิศทางของกระแสไปที่เอาต์พุต วงจรแทนแหล่งจ่ายกระแสต่อลักษณะคาสโคด เพื่อทำให้มีความต้านทานของแหล่งจ่ายกระแสสูง เพื่อลดผลกระทบที่เกิดจาก แรงดันที่เปลี่ยนแปลงที่เดรนของมอสเฟท M_{p1}, วงจร M_{p1} ซึ่งเป็นองค์ประกอบสำคัญในการจ่ายกระแสให้สมดุลกันทุกๆ ส่วน จึงต้องมีความสมดุลกันทุกๆ เซล, M_{p2} เป็นมอสเฟทที่ต่อคาสโคด ได้ทำการออกแบบให้มีอัตราส่วน (W/L) เหมาะสมเพื่อลดผลของตัวเก็บประจุแฝง ซึ่งอาจจะทำให้เกิดปัญหากับช่วงเวลาที่ใช้ในการสวิตช์กระแส ซึ่งก็จะมีผลกับความไวในการจ่ายกระแสที่เอาต์พุต

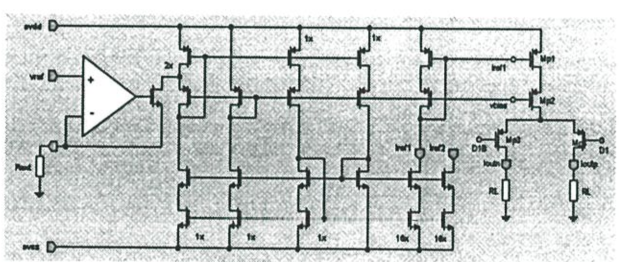


รูปที่ 6 ลักษณะของวงจรสวิตช์กระแส

กระแสที่ออกจากวงจรจ่ายกระแสจะไหลผ่านส่วนของวงจรสวิตช์กระแส และถูกจ่ายให้กับความต้านทานที่เอาต์พุต ทิศทางของกระแสจะเกิดจากดิจิตอลอินพุต (D1 และ D1B) วงจรสวิตช์กระแสต้องออกแบบให้เหมาะสมเพื่อลดผลที่เกิดจากตัวเก็บประจุแฝงซึ่งจะทำให้ลดความเร็วในการสวิตช์ การสวิตช์กระแสจะได้รับการควบคุมจากวงจรควบคุมเพิ่มและลดแรงดันที่ D1 และ D1B สัญญาณดิจิตอลที่ควบคุมที่ D1 และ D1B กำเนิดจากวงจรสวิตช์ [6] M_{n1}-M_{n4}, โดยรับสัญญาณอินพุตดิจิตอล ขนาดของ M_{n1}-M_{n4} จะเป็นส่วนสำคัญซึ่งทำให้เกิด ช่วงเวลาของสัญญาณขอบขาขึ้นและขอบขาลงที่ D1 และ D1B ซึ่งอาจจะทำงานไม่พร้อมกัน (ในขณะ 'ON' และ 'OFF'), กระแสที่เอาต์พุตของวงจรแปลงดิจิตอลเป็นอนาลอกเป็นแบบ 2 ขั้ว หากจะนำเอาต์พุตของวงจรไปใช้แบบขั้วเดียว คือการต่อเอาต์พุตที่ตกคร่อมความต้านทานทั้งสองด้าน , ในทางปฏิบัติ อาจใช้ center-tapped RF transformer, ความเป็นเชิงเส้นของวงจรแปลงดิจิตอลเป็นอนาลอก ขึ้นอยู่กับความสมมาตรเท่ากันของวงจรจ่ายกระแสทุกๆ ตัว ในทางปฏิบัติย่อมมีความผิดพลาดอยู่บ้าง โดยกระแสที่ผิดพลาด จะเกิดจากความไม่สมมาตรกันของแรงดันขีดเริ่ม (threshold voltage)ในมอสเฟทแต่ละตัว และข้อผิดพลาดที่เกิดจากการผลิต [7] ความไม่สมมาตรของแรงดันขีดเริ่ม (threshold voltage) จะเกิดจาก ขนาดความยาวของมอสเฟท (L) ซึ่งมีขนาดที่เล็กมากจะทำให้มีข้อผิดพลาดมาก ความผิดพลาดตรงนี้จะทำให้แรงดันขีดเริ่มของมอสเฟทแต่ละตัวไม่เท่ากัน จึงทำให้แรงดัน เกต-ซอร์ส มอสเฟท ไบอัสค่ากระแสที่ไม่เท่ากัน ตรวจสอบความไม่สมมาตรจากข้อมูลของเทคโนโลยี CMOS 0.5-μm เพื่อลดผลของความไม่สมมาตรจึงกำหนดขนาด ความยาวของพี-แชนแนล มอสเฟท ให้มีขนาดไม่น้อยกว่า 4μm และคำนวณ คุณสมบัติอื่นๆ ตามความต้องการของวงจร

4.3 วงจรกำเนิดกระแส (Bias Generator)

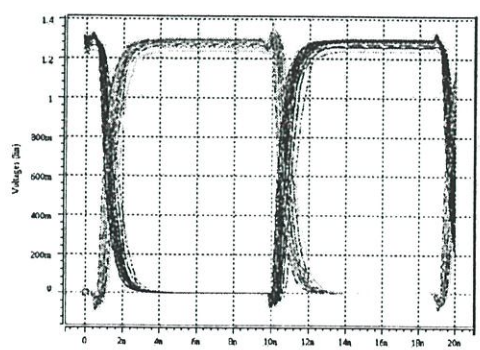
วงจรกำเนิดกระแสในรูปที่ 7 เป็นลักษณะวงจรอย่างง่าย โดยวงจรทำหน้าที่ เป็นระดับกระแสมาตรฐานและ มีระดับแรงดันมาตรฐาน เพื่อจ่ายให้กับส่วนของวงจรแหล่งจ่ายกระแส ของวงจรแปลงดิจิตอลเป็นอนาลอก จุด V_{ref} เป็นจุดต่อเพื่อรับแรงดันอ้างอิงจากภายนอกวงจร มีค่ากระแสมาตรฐานที่ต้องการหรือเท่ากับปริมาณกระแสใน 4 LSB ค่ากระแสมาตรฐานที่วงจรแปลงดิจิตอลเป็นอนาลอกใช้ จะถูกกำหนดด้วยตัวต้านทานที่ต่อภายนอกวงจร (R_{ext}) โดยสามารถกำหนดและควบคุมได้จากสมการ $I_{out,FS} = 320/R_{ext}$ วงจรกำเนิดกระแส ประกอบด้วยวงจรแบบประหยัดพลังงาน (power-down) ที่สามารถควบคุมจากคำสั่งดิจิตอลภายนอก



รูปที่ 7 การทำงานของวงจรกำเนิดกระแส

4.4 ผลจำลองการทำงานวงจรแปลงดิจิตอลเป็นอนาลอก

วงจรแปลงดิจิตอลเป็นอนาลอก 10-บิท ประกอบด้วยส่วนของวงจรสวิตช์จ่ายกระแส (current-steering) วงจรกำเนิดกระแส (bias generator) ส่วนของดิจิตอลลอจิก และ วงจรถอดรหัส การจำลองการทำงานของวงจร แปลงดิจิตอลเป็นอนาลอกโดยใช้โปรแกรม HSPICE โดยต่อโหลดที่เป็นตัวต้านทานขนาด 25Ω ผลการจำลองการทำงานแสดงในรูปที่ 8 และ ตารางที่ 1



รูปที่ 8 สัญญาณที่เอาต์พุตเมื่อมีการเปลี่ยนแปลงอุณหภูมิและแรงดัน

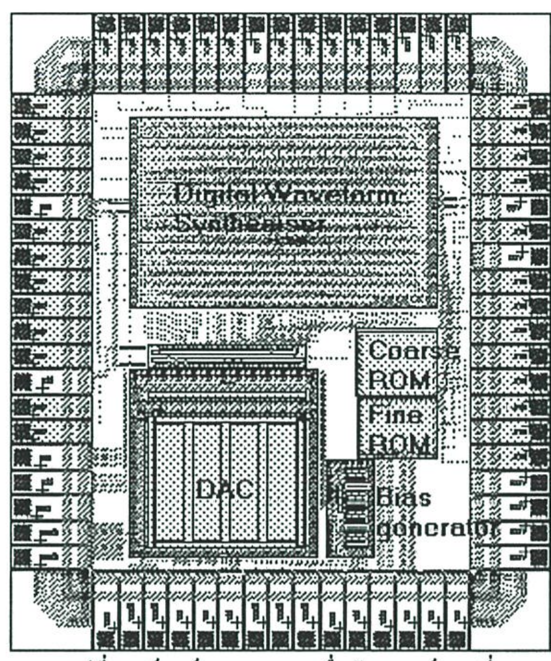
ทดสอบความไวในการทำงานโดยตรวจสอบค่าเวลาสัญญาณขอบขาขึ้น และ สัญญาณขอบขาลง โดยการทดสอบวงจรได้ปรับเปลี่ยนสภาวะให้วงจรทำงานในองค์ประกอบ ที่เกิดการเปลี่ยนแปลงของแหล่งจ่ายแรงดัน (+/-10%), อุณหภูมิ (0 ถึง +100°C) และ Vth (+/- 5%)

ตารางที่ 1 สรุปเวลาของการเกิดสัญญาณขาขึ้นและขาลง ของวงจร DAC

Temp. (°C)		25			100			0		
Supply (V)		3.30	3.63	2.97	3.30	3.63	2.97	3.30	3.63	2.97
TYP.	Tr nS.	0.46	0.471	0.527	0.580	0.497	0.656	0.486	0.446	0.513
	Tf nS.	0.705	0.722	0.792	0.870	0.883	1.01	0.766	0.754	0.790
FAST	Tr nS.	0.563	0.427	0.510	0.484	0.453	0.604	0.460	0.394	0.708
	Tf nS.	0.735	0.733	0.764	0.795	0.742	0.787	0.713	0.745	0.458
SLOW	Tr nS.	0.558	0.473	0.630	0.661	0.585	1.01	0.512	0.463	0.601
	Tf nS.	0.839	0.773	0.912	0.974	0.887	1.46	0.790	0.767	0.848

5. การออกแบบเลย์เอาต์ของวงจรรวม

ในการออกแบบเลย์เอาต์จะวาดเลย์เอาต์ของวงจรแปลงดิจิตอลเป็นอนาลอกและวงจรสร้างกระแสไบอัสด้วยมือ (Manual Low-Level Design) และแยกส่วนของอนาลอกกับดิจิตอล ให้อยู่คนละส่วน เพื่อป้องกันสัญญาณรบกวนที่เกิดจากการสวิตช์จากส่วนวงจรดิจิตอล ส่วนของวงจรดิจิตอลนั้นจะใช้วิธีการทำเลย์เอาต์แบบออโตเมติก (Automatic place-and-route) โดยใช้ไลบรารีเซลล์มาตรฐาน (Standard library cells) เลย์เอาต์ของชิพที่สมบูรณ์มีขนาด 10 ตารางมิลลิเมตร วงจรแปลงดิจิตอล เป็นอนาลอก (DAC) ทำการเลย์เอาต์วงจรจ่ายกระแสที่ต่อแบบคาสโคด (cascode current sources) โดยทำการเลย์เอาต์ แบบอาเรย์ โดยจัดให้อยู่คนละส่วนกับ วงจรสวิตช์กระแสและเป็นสัดส่วนกับการสวิตช์ให้กระแสไหลผ่าน ซึ่งเลย์เอาต์โดยวางตำแหน่งแยกจากกัน ลดความไม่สมดุลระหว่างวงจรแหล่งจ่ายกระแส ซึ่งเป็นสิ่งที่ต้องคำนึงถึงอย่างมากในการออกแบบวงจรรวมแบบผสม นอกจากนี้แหล่งจ่ายแรงดันของวงจรด้านดิจิตอล และ อนาลอก ได้แยกให้ใช้งานคนละส่วนอีกด้วย วงจรได้เลย์เอาต์ส่วนของการป้องกันล้อมรอบ (Guard ring) แบบ 2 ชั้น ให้กับวงจรจ่ายกระแส และ วงจรสวิตช์กระแส โดยจะช่วยป้องกันสัญญาณรบกวนที่เกิดจากการ สวิตช์ของวงจรดิจิตอลซึ่งเป็นสิ่งที่ต้องคำนึงถึงอย่างมากในการออกแบบวงจรรวมแบบผสม ทำให้ส่วนของวงจรแปลงดิจิตอลเป็น อนาลอกมีเสถียรภาพมากขึ้น การวางตำแหน่งของวงจรจ่ายกระแสแบ่งเป็น 4 ส่วน ซึ่งแต่ละส่วนจะประกอบด้วยแหล่งจ่ายกระแสจำนวน 63 ตัว น้ำหนักกระแสในการสวิตช์คือบิตละ 16-LSB ใช้ในการเพื่อสำหรับสวิตช์ส่วนของ 6 MSBs ที่ทำงานแบบเทอร์โมมิเตอร์โคด กระแสที่ได้จะเกิดจากปริมาณกระแสที่เท่ากันของทั้ง 4 ส่วนรวมกันได้ 16-LSB แต่ละส่วนจะจัดว่างตำแหน่งของแหล่งจ่ายกระแสให้สมดุลกัน เพื่อให้สภาวะแวดล้อมของแหล่งจ่ายกระแสมีสภาวะคล้ายกัน เพื่อลดข้อผิดพลาดที่เกิดจากปริมาณกระแสไม่เท่ากัน, ตำแหน่งของวงจรในชิพ แสดงในรูปที่ 9



รูปที่ 9 เลย์เอาต์ของวงจรรวมเพื่อสังเคราะห์ความถี่

6. สรุป

บทความวิจัยนี้ได้นำเสนอการออกแบบวงจรรวมเพื่อสังเคราะห์ความถี่ พร้อมด้วยวงจรแปลงดิจิตอลเป็นอนาลอกขนาด 10 บิท โดยใช้เทคโนโลยีซีมอสขนาด 0.5 ไมครอน คุณสมบัติของวงจรได้สรุปไว้ในตารางที่ 2 ซึ่งได้ส่งเลย์เอาต์ของวงจรรวมไปทำการเจือสารและเข้าแพคเกจ (Fabrication) เรียบร้อยแล้ว คาดว่าจะได้ตัวชิพกลับมาทดสอบภายในอนาคตอันใกล้นี้

ตารางที่ 2 คุณสมบัติของชิพดีดีเอส

IC Technology	1P3M 0.5-μm CMOS n-well
Max. clock frequency	100 MHz at Vdd = 3.3 V
Max. output frequency	40 MHz (0.4 x 100 MHz)
Frequency tuning resolution	0.0233 Hz (at 100 MHz)
Phase tuning resolution	0.0879 degrees (12-bits)
Amplitude resolution	0.846 mV @ R_L = 25 Ω
Frequency switching time	190 ns (19 x 1/100MHz)
Power dissipation	0.4 W @ 100 MHz, 3.3V
Die area	10 mm^2

7. กิตติกรรมประกาศ

คณะผู้วิจัยขอแสดงความขอบคุณฝ่ายไตรภาคี ศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติ ที่ให้ทุนสนับสนุนงานวิจัยนี้ ให้สำเร็จลุล่วงลงด้วยดี

เอกสารอ้างอิง

[1] B. Goldberg, Digital techniques in frequency synthesis, McGraw Hill, 1996, New York.

[2] G. Chang, A. Rofougaran, M. Ku, A. A. Abidi, and H. Samueli, "A low-power CMOS digitally synthesised 0-13 MHz agile sinewave generator," in *Proc. Int. Solid-State Circuits Conf.*, 1994, pp. 32-33.

[3] J. Vankka, M. Waltari, M. Kosunen, and K. Halonen, "A direct digital synthesizer with an on-chip D/A-converter," *IEEE J. Solid-State Circuits*, vol. 33, no. 2, pp. 218-227, Feb. 1998.

[4] J. Vankka, "Methods of mapping from phase to sine amplitude in direct digital synthesis," in *Proc. 1996 IEEE Int. Frequency Control Symp.*, pp. 942-950.

[5] H. T. Nicholas, III, H. Samueli, and B. Kim, "The optimization of direct digital synthesizer performance in the presence of finite word length effects," *in Proc. 42nd Annu. Frequency Control Symp.*, 1988, pp. 357-363.

[6] T.-Y. Wu, C.-T. Jih, J.-C. Chen, and C.-Y. Wu, "A low glitch 10-bit 75-MHz CMOS video D/A converter," *IEEE J. Solid-State Circuits*, vol. 30, no. 1, pp. 68-72, Jan. 1995.

[7] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers, "Matching properties of MOS transistors," *IEEE J. Solid-State Circuits*, vol. 24, no. 5, pp. 1433-1440, Oct. 1989.

ประวัติผู้เขียนบทความ

ชรัณ มีนกาญจน์ ปี 2540 สำเร็จการศึกษาในระดับปริญญาตรี สาขาวิศวกรรมคอมพิวเตอร์ จาก คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ปัจจุบันศึกษาในหลักสูตรวิศวกรรมศาสตร์มหาบัณฑิต สาขาวิศวกรรมอิเล็กทรอนิกส์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ศึกษาและทำงานวิจัยในส่วนของการออกแบบวงจรดิจิตอล ปีพ.ศ. 2540-2452 ทำงานตำแหน่งวิศวกรฝ่ายวิจัยและพัฒนาที่บริษัทชินโดมอิเล็กทรอนิกส์อินดัสตรี จนถึงปัจจุบัน ปี พ.ศ. 2542-ปัจจุบัน ทำงานตำแหน่งผู้ช่วยนักวิจัย สังกัดงานวิจัยการออกแบบวงจรรวม ศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติ รับผิดชอบงานวิจัยเกี่ยวกับการออกแบบวงจรรวมทางด้านดิจิตอล และการทดสอบวงจรรวมต้นแบบ

วิชัย แสงนาค ปี 2543 สำเร็จการศึกษาในระดับปริญญาตรี สาขาวิศวกรรมอิเล็กทรอนิกส์ จาก คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีมหานคร ปัจจุบันศึกษาในหลักสูตรวิศวกรรมศาสตร์มหาบัณฑิต สาขาวิศวกรรมอิเล็กทรอนิกส์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ศึกษาและทำงานวิจัยด้านการออกแบบวงจรรวมอนาลอก สำหรับใช้งานในระบบสื่อสาร

ธนพร ทองเผือก ปี 2540 สำเร็จการศึกษาในระดับปริญญาตรี สาขาวิศวกรรมคอมพิวเตอร์ จาก คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ปัจจุบันศึกษาในหลักสูตรวิศวกรรมศาสตร์มหาบัณฑิต สาขาวิศวกรรมอิเล็กทรอนิกส์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ศึกษาและทำงานวิจัยในส่วนของการออกแบบวงจรดิจิตอล ปีพ.ศ. 2540-ปัจจุบัน ทำงานตำแหน่งวิศวกรออกแบบวงจรรวม ที่บริษัท ไมโครอิเล็กทรอนิกส์ เทคโนโลยี (ประเทศไทย) จำกัด จนถึงปัจจุบัน รับผิดชอบงานวิจัยเกี่ยวกับการออกแบบวงจรรวม

อภินันท์ ธนชยานนท์ สำเร็จการศึกษาในระดับปริญญาตรี โท และเอก สาขาวิศวกรรมไฟฟ้าและอิเล็กทรอนิกส์ จากอิมพีเรียลคอลเลจ มหาวิทยาลัยลอนดอน ในปี พ.ศ.2538 และ 2542 ตามลำดับ, พ.ศ. 2542-ปัจจุบัน ดำรงตำแหน่ง อาจารย์ประจำภาควิชาวิศวกรรมอิเล็กทรอนิกส์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ตั้งแต่ปี ทำงานวิจัยทางด้านการออกแบบวงจรรวม สำหรับการนำไปใช้งานในระบบสื่อสารและทางชีวภาพการแพทย์

A Single-Chip CMOS Digitally Synthesized 0-35 MHz Agile Function Generator

C. Meenakarn[1,2] and A. Thanachayanont[1]

[1] Research Center of Communications and Information Technology, and Faculty of Engineering,
King Mongkut's Institute of Technology Ladkrabang,
3 Chalongkrung Rd, Bangkok 10520, THAILAND
Phone: +66-2-7373000 ext 3309, Fax: +66-2-7392429, E-mail: s3061316@kmitl.ac.th, ktapinun@kmitl.ac.th
[2] Thailand IC Design Incubator (TIDI), National Electronics and Computer Technology Center (NECTEC)
99/25, 10th floor, Software Park Bldg, Chaengwattana Rd, Nonthaburi 11120, THAILAND
Phone: +66-2-5822560 ext 213, Fax: +66-2-5822562, E-mail: charan@nectec.or.th

Abstract: This paper describes the design and implementation of a single-chip digitally synthesized 0-35 MHz agile function generator. The chip comprises an integrated direct digital synthesizer (DDS) with a 10-bit on-chip digital-to-analog converter (DAC) using an n-well single-poly triple-metal 0.5-μm CMOS technology. The main features of the chip include maximum clock frequency of 100 MHz at 3.3-V supply voltage, 32-bit frequency tuning word resolution, 12-bit phase tuning word resolution, and an on-chip 10-bit DAC. The chip provides sinusoidal, ramp, saw-tooth, and random waveforms with phase and frequency modulation, and power-down function. At 100-MHz clock frequency, the chip covers a bandwidth from dc to 35 MHz in 0.0233-Hz frequency steps with 190-ns frequency switching speed. The complete chip occupies 12-mm^2 die area and dissipates 0.4 W at 100-MHz clock frequency.

1. Introduction

There are many significant advantages of direct digital synthesizer (DDS) over the traditional frequency-agile phase-locked loop (PLL)-based synthesizer, including fast settling time, digitally-controlled sub-hertz frequency and sub-degree phase resolutions, continuous-phase switching response, and low phase noise [1]. Due to advances in digital logic and digital-to-analog converter (DAC) IC technologies, DDS is now enjoying an increasingly significant role in wideband frequency generation. The integration of a high-speed, high-performance DAC and DDS circuitry onto a single chip enables low-cost, high-performance, functionally-integrated, and small package-sized DDS products, which target a wide range of applications such as time division multiple access/code division multiple access (TDMA/CDMA) digital cellular systems and spread-spectrum wireless LANs.

A number of high-speed DDSs with on-chip DAC have been reported. In [2], a CMOS DDS implemented in a 1.0-μm process was reported with 50-MHz maximum operating clock frequency. In [3], a BiCMOS DDS implemented in 0.8-μm double-metal double-poly process was reported with the maximum operating clock frequency of 170 MHz. The gap in the maximum operating clock frequency of the two designs mainly arises from the difficulty in the design of high-speed DAC in CMOS technology. Therefore the design of a high-speed DDS with an on-chip DAC in standard CMOS process still poses a real challenge. This paper describes the design and implementation of a single-chip digitally synthesized 0-35 MHz agile function generator using an n-well single-poly triple-metal (1P3M) 0.5-μm CMOS technology. The objective of the chip reported in this paper is to generate various types of waveforms for general video, wireless applications such as digital radios/modems, test and measurement equipment, digital video/audio, baseband transceivers, and PC-based instrumentation cards.

2. Chip Architecture

The architecture of the chip, shown in Figure 1, is employed. The phase accumulator word length is chosen to be 32 bits to achieve a frequency tuning resolution of 0.0233 Hz at the clock rate of 100 MHz. Only 14 of the most significant bits (MSBs) are used to calculate the sine wave samples in order to reduce size and power dissipation of the table look-up ROM. Two MSBs of these 14 bits are used to decode the quadrant of the sine wave samples in the ROM. This achieves a 12-bit phase tuning resolution that yields a spurious performance due to the phase accumulator truncation of –72 dBc [4], which is well below the spur level of the on-chip 10-bit DAC at 100 MHz.

Phase modulation of the chip is achieved by adding 12-bit phase tuning word to the phase accumulator output before entering the phase-to-amplitude converter. The 10-bit digital output data is also available for applications that do not require the conversion to analog, such as tuneable digital band-pass filter, mixers for digital receivers, and real-time digital spectrum analysis. The design of the chip, described in the following sections, is divided into two main parts: (1) the DDS circuitry (digital part) and (2) the analog part. The layout considerations and the experimental results are described after the design of digital and analog parts.

3. Design of DDS Circuitry

3.1 DDS Circuitry Architecture

The DDS circuitry is designed and implemented by using hardware description language (HDL). In this design, VHDL language is chosen. The VHDL code is first simulated at the functional level. Then the functionally verified VHDL code is synthesized into the gate-level

circuitry. The gate-level circuitry is re-simulated in order to verify the timing constraints. The main building blocks of the DDS circuitry are a 32-bit phase accumulator, coarse and fine ROMs, a 14-bit phase-to-amplitude converter and a clock generator. In order to achieve maximum conversion speed, the 19-stage pipeline is employed. The modified Sunderland algorithm [5] is chosen for phase-to-amplitude conversion. Latches are implemented to strobe the digital samples of the waveform before entering the 10-bit DAC in order to eliminate signal skews and reduce output glitches.

The frequency and pattern of the output waveform can be controlled by an external micro-controller via an 8-bit parallel interface. The maximum update rate, determined by the speed of the control interface and the 19-stage pipeline, is 38 MS/s. The DDS circuitry can provide 4 waveforms including sinusoidal, ramp, saw-tooth and random. The 10-bit amplitude of ramp and saw-tooth waveforms is generated directly from the phase register, while a linear feedback shift register is used to generate random values for random waveform. The 10-bit amplitude of sinusoidal waveform is generated from a sine look-up table.

3.2 Memory Compression

Using the well-known quarter-wave symmetry technique, the sine wave samples for the full range of 2π are generated from 0 to $\pi/2$ rads of sine information stored in the ROM [5]. The two MSBs of the phase accumulator are used to decode the quadrant of the sine function. Thus the MSB is used as the sign bit, while the next MSB controls whether the phase between 0 to $\pi/2$ should be increased or decreased. This reduces the capacity of the look up table with the penalty of additional logic circuits required to generate the complements of the accumulator and the look-up table outputs. The width of the one-quadrant look-up table is reduced by compression of the quarter-wave sine information using the Modified Sunderland algorithm [5]. The use of Modified Sunderland algorithm gives the total ROMS compression ratio of 64:1.

4. Design of 10-Bit DAC

4.1 DAC Architecture

The integrated high-speed 10-bit DAC is implemented by using the 6/4 segmented current-steering architecture, in which the 6 MSBs of the digital binary inputs are thermometer-decoded to control 63 identical current sources, each having 16-LSB current weighting, and the remaining 4 LSBs control 4 binary-weighted current sources. The output currents of all current sources, which are either switched ON or OFF according to the digital input codes, are summed and driven into any resistive load in order to generate the required analog output voltage. The DAC comprises an array of current-steering cells, a bias generator, a segment decoder, and code latches as shown in Figure 2.

4.2 Current-Steering Cell

The schematic diagram of the current-steering cell of the 10-bit DAC is shown in Figure 3. The cascode current source (M_{p1} and M_{p2}) and the differential current-steering switches (M_{p3} and M_{p4}) are implemented by using p-channel MOSFETs in order to obtain the output voltage referred to ground. The cascode configuration suppresses the voltage fluctuation at the drain of the transistor M_{p1} thereby enhancing the output impedance of the current source. Long channel length is required for the transistor M_{p1} in order to achieve good accuracy and matching of the output current. The cascode transistor M_{p2} should have a short channel length in order to minimize the parasitic capacitance at the coupled sources of the differential switches, thus improving the switching speed during the output transition. The current from the cascode circuit is steered by the differential switches into two resistive loads, according to the true and complementary digital control signals, D1 and D1B. The differential switches employ the minimum channel length, and the width is optimized toward the switching speed while maintaining the accuracy of the output currents.

The two complementary digital control signals, D1 and D1B, are generated by the asymmetrical switch driver [6], M_{n1}-M_{n4}, in order to withdraw any possibility that both switches are turned OFF simultaneously due to inevitable delay in both control signals. The switch driver is simply a differential drive circuit with n-channel transistors for both pull-up and pull-down. The dimension of M_{n1}-M_{n4} must be chosen properly such that the fall-time of D1 and D1B (turning ON) will be faster than the rise-time (turning OFF). This ensures that both switches are never turned off simultaneously, but allowing a simultaneous turn-on for a short period of time. Turning-on both switches simultaneously will degrade the switching speed a little, but reducing glitches substantially. Furthermore, the HIGH level output is V_{TH} lower than V_{DD}, minimizing the switching swing thus reducing the switching feedthrough to the output current. This intentional asymmetrical switching circuit guarantees that one switch is ON, even for a small skew. HSPICE simulation using the process parameters from the Alcatel's 0.5-µm CMOS technology suggests an extra delay of 0.2 ns due to the switch driver circuit.

The two complementary DAC output currents are uni-polar. Thus the voltages developed across the load resistors range from 0 V to a positive full-scale value. In practice, a center-tapped RF transformer is used to combine these two complementary currents and produce a bi-polar, zero DC-offset symmetrical output current. Additionally the transformer is beneficial in both providing the DAC outputs a suitable load resistance (via an impedance transformation) in order not to violate the DAC output voltage compliance, and coupling the DAC output currents to the reactive input of the subsequent LC low-pass filter. This also maintains a constant current flowing in each current source during switching, thus allowing fast settling time and reducing output glitches.

Linearity of the DAC is determined by the matching of current sources. In practice, the total current error is due to

the variation in threshold voltage and process geometry [7]. The dominant source of error is likely to be the threshold voltage variation provided that the devices are well above the minimum geometry. Thus large gate-source voltage and large LSB current weighting are required to reduce the mismatch. Measured mismatch data from the 0.5-μm CMOS technology suggest that the channel length of the p-channel MOSFET current source should be no shorter than 4 μm in order to satisfy the current matching requirements of the 10-bit DAC. Other circuit parameters are then calculated accordingly to meet the DAC specifications.

5. Layout Considerations

The performance of the complete chip is largely determined by the on-chip DAC. Thus much attention were paid to the layout and routing of the DAC including the bias generator which were done by hand, while all digital circuits were implemented by using automatic place-and-route with standard library cells. The total chip area about is 12 mm^2.

To obtain the optimum performance, the DAC cascode current sources are drawn in an array, isolating from their associated current-steering switches and asymmetrical switch drivers that are placed together in a separate block. This allows a compact and uniform layout of the array, reducing mismatches between the current sources. Isolation of sensitive analog circuits from digital switching noise is very important in mixed signal IC. Therefore analog and digital power supplies are separated, and double guard rings surround both the current source array and the switches and drivers block in order to prevent digital switching noise coupling to the DAC output currents.

6. Experimental Results

A test system, shown in Figure 4, was developed in order to evaluate the chip. The software on a personal computer is used to load the frequency control word into the test board. The micro-controller on the test board generates controlling data and signal for the chip according to the frequency control word received from the computer. A 7th-order 40-MHz low-pass filter is used to filter high-frequency components from the output of the chip.

The chip was tested at 3.3-V supply voltage and 100-MHz clock frequency. The chip can generate an output from dc to 35 MHz in 0.0233-Hz frequency steps with 190-ns frequency switching speed. The spurious free dynamic range (SFDR) is better than 45 dBc. A spectrum shown in Figure 5 is the spectrum of 2-MHz sinusoidal output. Figure 6 shows the relation between SFDR and output frequencies at 100-MHz clock frequency. The phase noise is below –115 dBc/Hz at 100-kHz offset. The closed-in spectrum of the 25-MHz sinusoidal output at 100-MHz clock frequency is shown in Figure 7.

7. Conclusions

The design and implementation of a single-chip CMOS digitally synthesized agile function generator is described in this paper. The chip covers a bandwidth from dc to 35 MHz with 0.0233-Hz resolution. At 100-MHz clock frequency, the SFDR is better than 45 dBc. Table 1 summarizes the specifications of the chip. The photomicrograph of the chip is shown in Figure 8.

Acknowledgements: Financial support of the Tri-Partite research grant from National Electronics and Computer Technology Center (NECTEC), Thailand, is gratefully acknowledged.

References

[1] B. Goldberg, Digital techniques in frequency synthesis, McGraw Hill, 1996, New York.

[2] G. Chang, A. Rofougaran, M. Ku, A. A. Abidi, and H. Samueli, "A low-power CMOS digitally synthesised 0-13 MHz agile sinewave generator," in *Proc. Int. Solid-State Circuits Conf.*, 1994, pp. 32-33.

[3] J. Vankka, M. Waltari, M. Kosunen, and K. Halonen, "A direct digital synthesizer with an on-chip D/A-converter," *IEEE J. Solid-State Circuits*, vol. 33, no. 2, pp. 218-227, Feb. 1998.

[4] H. T. Nicholas, III, H. Samueli, and B. Kim, "The optimization of direct digital synthesizer performance in the presence of finite word length effects," *in Proc. 42nd Annu. Frequency Control Symp.*, 1988, pp. 357-363.

[5] J. Vankka, "Methods of mapping from phase to sine amplitude in direct digital synthesis," in *Proc. 1996 IEEE Int. Frequency Control Symp.*, pp. 942-950.

[6] T.-Y. Wu, C.-T. Jih, J.-C. Chen, and C.-Y. Wu, "A low glitch 10-bit 75-MHz CMOS video D/A converter," *IEEE J. Solid-State Circuits*, vol. 30, no. 1, pp. 68-72, Jan. 1995.

[7] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers, "Matching properties of MOS transistors," *IEEE J. Solid-State Circuits*, vol. 24, no. 5, pp. 1433-1440, Oct. 1989.

Table 1. Chip specifications

Parameter		Specification
Technology		Single-poly, triple-metal 0.5-μm CMOS
Power supply voltage		3.3 V
Max. clock frequency		100 MHz
Max. output frequency		35 MHz at 100-MHz clock rate
Frequency resolution		32 bits (0.0233 Hz at 100 MHz)
Phase tuning resolution		12 bits (0.0879 degrees)
Amplitude resolution		10 bits (0.846 mV at R_L = 25 Ω)
f_{out} = 25MHz	Worst-case spurious	-45 dBc
	Phase noise	-115 dBc/Hz at 100-kHz offset
Max. frequency switching speed		190 ns (19 x 1/100MHz)
Die area (including pads)		3240x3830 μm^2
Power dissipation	Digital	270 mW at 100-MHz clock rate
	Analog	126 mW at full-scale output level

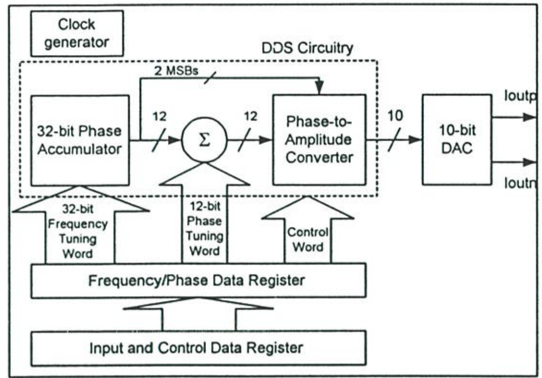


Figure 1. Block diagram of the chip

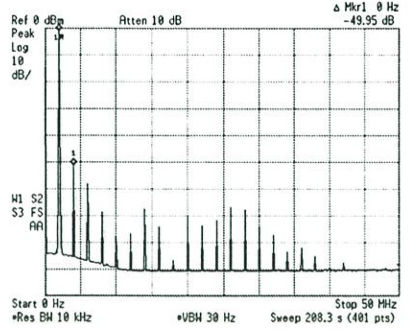


Figure 5. Spectrum of 2-MHz sinusoidal output @ 100-MHz clock frequency

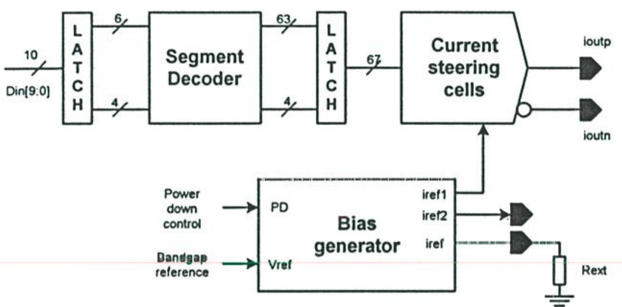


Figure 2. Block diagram of the 10-bit DAC

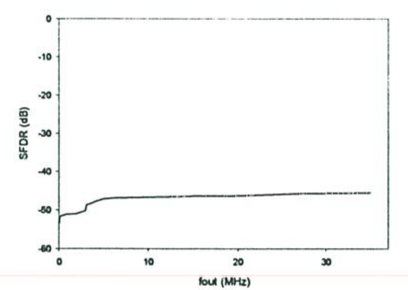


Figure 6. SFDR as a function of output frequency @ 100-MHz clock frequency

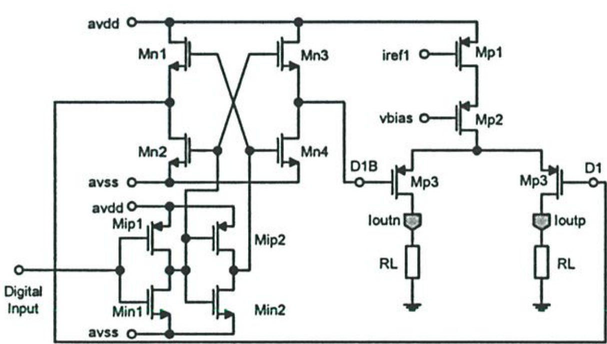


Figure 3. DAC's current-steering cell

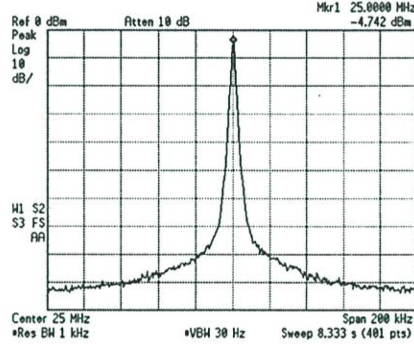


Figure 7. The close-in spectrum of 25-MHz sinusoidal output @ 100-MHz clock frequency

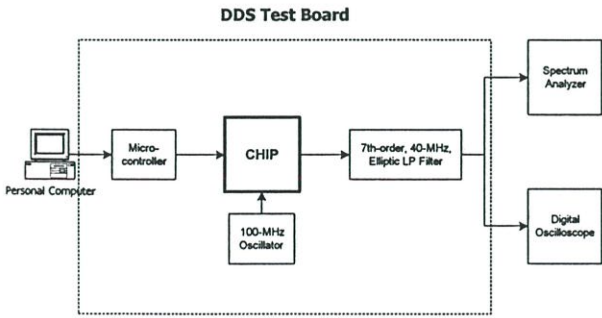


Figure 4. Test system

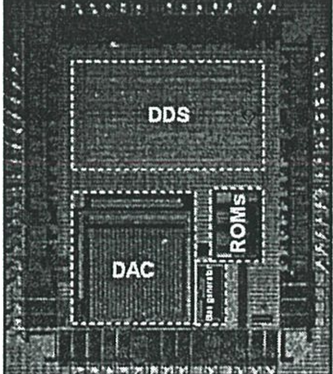


Figure 8. The photomicrograph of the chip

100-MHZ CMOS DIRECT DIGITAL SYNTHESIZER WITH 10-BIT DAC

C. Meenakarn[1,2] *and A. Thanachayanont*[1]

[1]Research Center of Communications and Information Technology, and Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, THAILAND, E-mail: ktapinun@kmitl.ac.th
[2]National Electronics and Computer Technology Center, National Science and Technology Development Agency, Bangkok, THAILAND, E-mail: charan@nectec.or.th

ABSTRACT

This paper describes the design and implementation of an integrated direct digital synthesizer with a 10-bit on-chip digital-to-analog converter using a 0.5-μm CMOS technology. The DDS chip operates at 100-MHz maximum clock frequency under 3.3-V supply voltage, with 32-bit frequency, 12-bit phase and 10-bit amplitude resolution. The chip provides sinusoidal, sawtooth, ramp, square and random waveforms with phase and frequency modulation, and power-down function, occupies 12-mm^2 die area, and dissipates 0.4 W at 100-MHz clock rate. At 25-MHz sinusoidal output, the measured worst-case spurious is –65 dBc and the phase noise is -119 dBc/Hz at 100-kHz frequency offset.

1. INTRODUCTION

For many applications, direct digital synthesizer (DDS) provide distinct advantages over the traditional frequency-agile phase-locked loop (PLL) based synthesizer, including fast settling time, sub-hertz frequency and sub-degree phase resolutions, continuous-phase switching response, and good spectral purity [1]. In recent years, DDS have enjoyed an increasingly significant role in wideband frequency generation due to advances in digital logic and digital-to-analog converter (DAC) IC technologies. The integration of a high-speed, high-performance DAC and DDS circuitry onto a single chip enables low-cost, high-performance, functionally-integrated, and small package-sized DDS products that target a wide range of applications such as time division multiple access/code division multiple access digital cellular systems and spread-spectrum wireless LANs.

High-speed DDSs with on-chip DAC have been reported recently. In [1], a 40-MHz DDS using a 1.0-μm CMOS process was reported, while a 170-MHz DDS using a 0.8-μm BiCMOS process was reported in [2]. The gap in the maximum operating clock frequency of the two designs mainly arises from the difficulty in the design of high-speed DAC in CMOS technology. Therefore the design of high-speed DDS with on-chip DAC in standard CMOS process still poses a real challenge. This paper describes the design and implementation of a 100-MHz DDS with 10-bit on-chip DAC using a 0.5-μm digital CMOS process.

2. DDS ARCHITECTURE

The architecture of the DDS chip is shown in Fig. 1. The phase accumulator word length is 32 bits, rendering 0.0233-Hz a frequency tuning resolution at 100-MHz sample rate. Fourteen most significant bits (MSBs) are used to calculate the sine-wave samples in order to reduce size and power dissipation of the ROM look-up table. Two MSBs of the 14 bits are used to decode the quadrant of the sinewave, while the remaining 12 bits are used to address the sinewave samples in the ROM. This achieves a 12-bit phase tuning resolution that yields a spurious performance, due to the phase accumulator truncation of –72 dBc [5], which is below the spur level of the 10-bit DAC.

Phase modulation of the DDS is achieved by adding a 12-bit phase tuning word to the phase accumulator output before entering the waveform generator. The 10-bit digital output is also made available for applications that do not require the conversion to analog, such as tuneable digital bandpass filters, mixers for digital receivers, and real-time digital spectrum analysis. The design of the DDS chip, described in the following sections, is divided into two main parts: (1) the DDS circuitry, including an I/O control interface, coarse and fine ROMs, a waveform synthesizer, and (2) the 10-bit DAC, including a bias current generator and a thermometer decoder.

3. DESIGN OF DDS CIRCUITRY

For maximum conversion speed, the DDS digital circuit employs 19 pipeline stages. The main building block is the waveform synthesizer that contains an increment, a 32-bit phase accumulator, and a 14-bit phase-to-amplitude converter. The modified Sunderland algorithm [4] is chosen for the phase-to-amplitude conversion. Latches are employed to strobe the digital control signals that are applied to the 10-bit DAC to eliminate signal skew and thus reducing output glitches. The DDS is controlled by an external controller via an 8-bit parallel interface for frequency update and waveform pattern. The maximum update rate determined by the speed of the control interface and the 19 pipeline stages is 38 MS/s. The waveform generator can generate sinusoidal, sawtooth, ramp, square and random waveforms. The sawtooth and ramp are generated directly from the phase register, while a linear feedback shift register is used to generate the random waveform. The sinewave is generated from a lookup table, while the MSB renders the square waveform.

Using the well-known quarter-wave symmetry technique, the sine-wave samples for the full range of 2π

are generated from the 0 to $\pi/2$ rads of sine information stored in the ROM [4]. The two MSBs of the phase accumulator are used to decode the quadrant of the sine function. Thus the MSB is used as the sign bit, while the next MSB controls whether the phase between 0 and $\pi/2$ should be increasing or decreasing. This reduces the capacity of the look-up table with the penalty of additional logic circuits required to generate the complements of the accumulator and the look-up table outputs. The width of one-quadrant look-up table is reduced by compression of the quarter-wave sine information using the modified Sunderland algorithm [4]. A 'coarse' ROM, 2^8x8 bits, provides low resolution phase samples, while a 'fine' ROM, 2^8x2 bits, gives additional phase resolution by interpolating between the low resolution samples, rendering a compression ratio of 64:1.

4. DESIGN OF 10-BIT DAC

The on-chip 10-bit DAC is implemented by using the 6/4 segmented current-steering architecture. The 6 MSBs of the digital binary inputs are thermometer-decoded to control 63 identical current sources, each having 16-LSB current weighting, while the remaining 4 LSBs control 4 binary-weighted current sources. The output currents of all current sources, which are either switched ON or OFF according to the digital input codes, are summed and driven into any resistive load in order to generate the required analog output voltage. The DAC comprises an array of current-steering cells, a bias generator, a segment decoder, and code latches as shown in Fig. 2. The maximum output compliance is 1.3 V under 3.3-V power supply voltage.

4.1 Current-Steering Cell

The schematic diagram of the current-steering cell of the 10-bit DAC is shown in Fig. 3. The cascode configuration suppresses voltage fluctuation at the drain of the transistor M_{p1} thereby enhancing the output impedance of the current source. Long channel length is required for the transistor M_{p1} in order to achieve good accuracy and matching of the output current. The cascode transistor M_{p2} has a short channel length to minimise parasitic capacitance at the sources of the differential switches, thus improving the switching speed during the output transition. The current is steered by the differential switches into two resistive loads, according to the true and complementary digital control signals, D1 and D1B. The differential switches employ minimum channel length, and their width is optimised toward the switching speed while maintaining the accuracy of the output currents.

The complementary control signals, D1 and D1B, are obtained from the asymmetrical switch driver [6], M_{n1}-M_{n4}, in order to withdraw any possibility that both switches are turned OFF simultaneously, due to inevitable delay in both control signals. The switch driver is simply a differential drive circuit with n-channel transistors for both pull-up and pull-down. The dimension of M_{n1}-M_{n4} must be chosen properly such that the fall-time of D1 and D1B (turning ON) will be faster than the rise-time (turning OFF). This ensures that both switches are never OFF simultaneously, but allowing a simultaneous ON for a short period of time. Turning-on both switches simultaneously will degrade the switching speed a little, but reducing glitches substantially. Furthermore, the HIGH output level is V_{TH} lower than V_{DD}, reducing the switching feedthrough to the output current. The switch driver guarantees that one switch is always ON, even for a small skew. HSPICE simulation suggests a 0.2-ns delay due to the switch driver circuit.

The complementary DAC output currents are uni-polar. Thus the voltages developed across the load resistors range from 0 volt to a positive full-scale value. In practice, a center-tapped RF transformer is used to combine these two complementary currents and produce a bi-polar, zero DC-offset symmetrical output current. Additionally the transformer is beneficial in both providing the DAC outputs a suitable load resistance (via an impedance transformation) in order not to violate the DAC output voltage compliance, and coupling the DAC output currents to the reactive input of the subsequent LC lowpass filter. This also maintains a constant current flowing in each current source during switching, thus allowing fast settling time and reducing output glitches.

The DAC's linearity is determined by mismatches in the current sources. In practice, the total current error is due to variations in threshold voltage and process geometry [7]. The dominant source of error is likely to be the threshold voltage variation provided that the devices are well above the minimum geometry. Thus large gate-source voltage and large LSB current weighting are required to reduce mismatches. For this process, the channel length of the p-channel MOSFET current source should be greater than 4 μm in order to satisfy the current matching requirement. Other circuit parameters are then calculated accordingly to meet the DAC specifications.

4.2 Bias Generator

Fig. 4 shows the simplified schematic diagram of the bias generator that provides the reference current and the bias voltage to the DAC current steering cells. The voltage V_{ref} is provided by an external bandgap voltage generator. The reference current distributed to the current source array is 16 times larger than the current of the unit current-steering cell (i.e. $4I_{LSB}$), this reduces noise multiplication in the current mirror. The full-scale DAC output current is determined by an off-chip resistor R_{ext} and is given by, $I_{out,FS} = 320/R_{ext}$. The bias generator also includes the necessary power-down circuitry, which can be digitally controlled. Nominally, $R_{ext} = 9.1\text{k}\Omega$ and $I_{out,FS} = 35.16$ mA. Monte Carlo simulations showed that the DAC performed satisfactorily under worst-case parameters and variations in power supply voltage (+/- 10%), temperature (-30 to +70 °C), and threshold voltage (+/-5 %).

5. LAYOUT CONSIDERATIONS

The layout and routing of the DAC including the bias generator were done by hand while all digital circuits were laid out by using automatic place-and-route with standard library cells. To obtain the optimum performance, the DAC

cascode current sources are drawn in an array, isolating from their associated current-steering switches and asymmetrical switch drivers that are placed together in a separate block. This allows a compact and uniform layout of the array, reducing mismatches between the current sources. Isolation of sensitive analog circuits from digital switching noise is very important in mixed signal IC. Therefore analog and digital power supplies are separated, and double guard rings surround both the current source array and the switches and drivers block in order to prevent digital switching noise coupling to the DAC output currents.

The current source array is separated into 4 quadrants. Each of the 63 current sources with 16-LSB weighting for the 6 MSBs is implemented by 4 identical parallel current sources of 4-LSB weighting; each of which is placed into one different quadrant. A two-dimensional common-centroid switching sequence is used in each of the four quadrants, which are mirrored with respect to the vertical and horizontal axes. The spatial symmetry of the array and the common-centroid switching scheme simultaneously compensate for both two-dimensional linear and parabolic systematic errors due to process, temperature, and electrical gradients over the current source array [7].

The 4-LSB cascode current source is laid out elegantly such that all cells can be 'butt-end' connected. This reduces both layout time and chip area, and is very well suited for design automation. The current sources in each quadrant are routed by using overlaid metal-1 and metal-2 grids. Wide metal-2 grid is also used for power supply routings in order to reduce the effect of 'IR' voltage dropped. The binary-weighted current sources for the 4 LSBs are located in the vacant space in the centre of the array. Dummy current cells surround the array in order to reduce the edge effects.

6. EXPERIMENTAL RESULTS

A photograph of the DDS chip is shown in Fig. 5. A 6th-order 40-MHz elliptic lowpass filter is employed to remove aliasing harmonics of the DAC output. At 100 MHz with 3.3 V power supply voltage, the DDS logic, the worst case spurious products in a DDS system operate a sample rate f_s appears at $f_s/4$ and $f_s/3$. shows the measured spurious products at 25-MHz output frequency with 100-MHz clock frequency. Fig. 6 and Fig. 7 show the measured frequency spectra of the 25-MHz sinewave output, the spurious products are below –65 dBc while the measured phase noise is –119 dBc/Hz at 100-kHz frequency offset.

7. CONCLUSION

This paper has described the design and implementation of a 100-MHz CMOS DDS including a 10-bit on-chip DAC. The complete chip occupies 12-mm^2 die area and dissipates 0.4 W at 100-MHz. At 25-MHz output, the measured worst-case spurious is –65 dBc and the phase noise is -119 dBc at 100-kHz offset.

Acknowledgements: The financial support from the National Electronics and Computers Technology Center, Thailand is gratefully acknowledged.

REFERENCES

[1] B. Goldberg, Digital techniques in frequency synthesis, McGraw Hill, 1996, New York.

[2] G. Chang, A. Rofougaran, M. Ku, A. A. Abidi, and H. Samueli, "A low-power CMOS digitally synthesised 0-13 MHz agile sinewave generator," in Proc. Int. Solid-State Circuits Conf., 1994, pp. 32-33.

[3] J. Vankka, M. Waltari, M. Kosunen, and K. Halonen, "A direct digital synthesizer with an on-chip D/A-converter," IEEE J. Solid-State Circuits, vol. 33, no. 2, pp. 218-227, Feb. 1998.

[4] J. Vankka, "Methods of mapping from phase to sine amplitude in direct digital synthesis," in Proc. 1996 IEEE Int. Frequency Control Symp., pp. 942-950.

[5] H. T. Nicholas, III, H. Samueli, and B. Kim, "The optimization of direct digital synthesizer performance in the presence of finite word length effects," in Proc. 42nd Annu. Frequency Control Symp., 1988, pp. 357-363.

[6] T.-Y. Wu, C.-T. Jih, J.-C. Chen, and C.-Y. Wu, "A low glitch 10-bit 75-MHz CMOS video D/A converter," IEEE J. Solid-State Circuits, vol. 30, no. 1, pp. 68-72, Jan. 1995.

[7] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers, "Matching properties of MOS transistors," IEEE J. Solid-State Circuits, vol. 24, no. 5, pp. 1433-1440, Oct. 1989.

[8] A. Marques, et al "A 12b accuracy 300MSamples/s update rate CMOS DAC," in ISSCC Dig. Tech. Papers, 1998.

Table 1. Measured performance of the DDS

Technology		0.5-μm CMOS
Power supply voltage		3.3 V
Max. clock frequency		100 MHz
Max. output frequency		34 MHz
Frequency resolution		32 bits (0.0233 Hz at 100 MHz)
Phase tuning resolution		12 bits (0.0879 degrees)
Amplitude resolution		10 bits (0.846 mV at R_L =25 Ω)
f_{out} = 25 MHz	Spurious	-65 dBc (worst case)
	Phase noise	-119 dBc/Hz at 100-kHz offset
Frequency switching speed		190 ns (maximum)
Die area (including pads)		3240x3830 μm^2
Power dissipation	Digital	270 mW at 100-MHz
	Analog	126 mW at full-scale output

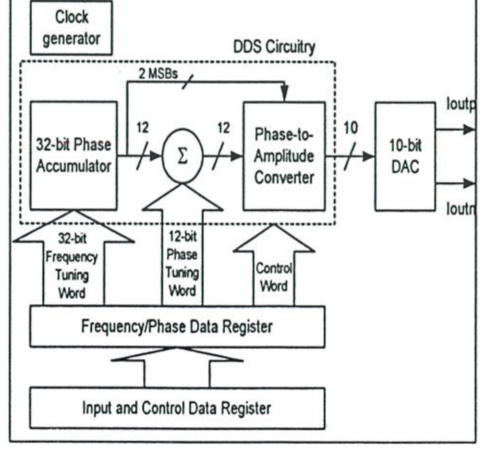


Fig. 1. Architecture of the DDS chip

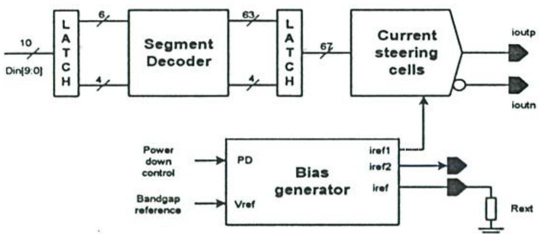


Fig. 2. Block diagram of the 10-bit DAC

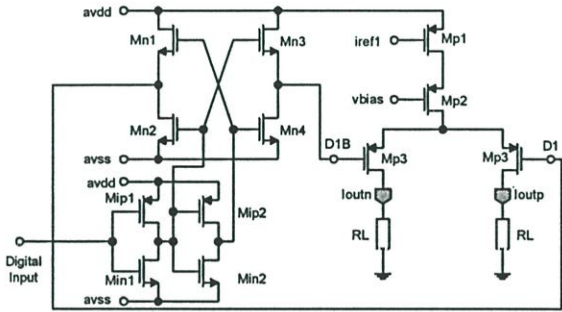


Fig. 3. DAC current-steering cell

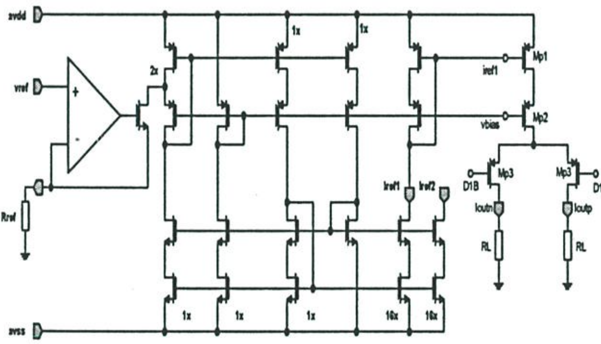


Fig. 4. The DAC bias generator

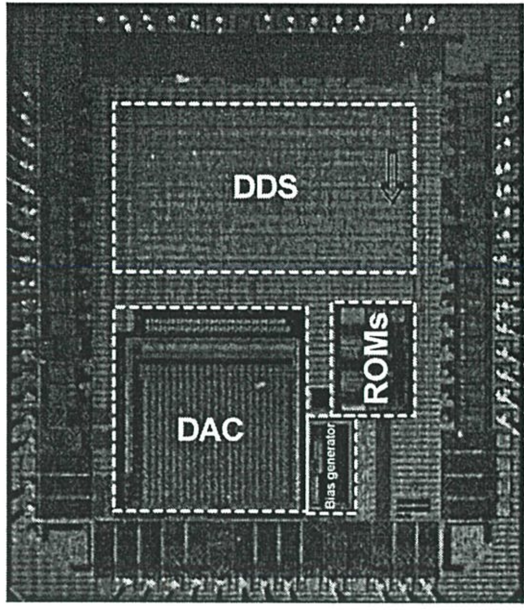


Fig. 5. DDS chip photograph

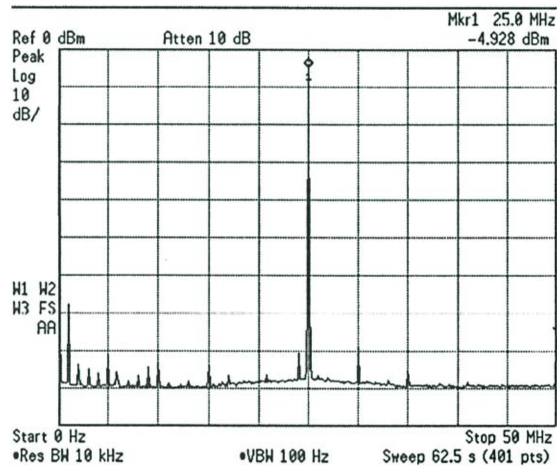


Fig. 6. Spurious products at 25-MHz sinusoidal output

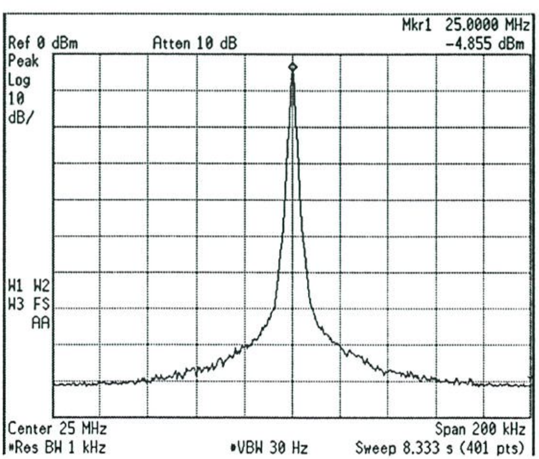


Fig. 7. Spectrum of 25-MHz sine output

A ROM-Less Direct Digital Frequency Synthesiser Using a Polynomial Approximation

C. Meenakarn[*,**], and A. Thanachayanont[*]

[*] Research Center of Communications and Information Technology, and Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, THAILAND
[**] Thailand IC Design Incubator (TIDI), National Electronics and Computer Technology Center (NECTEC), Nonthaburi, THAILAND

ABSTRACT

This paper describes the design and implementation of a sine-output ROM-less direct digital frequency synthesiser by using a polynomial approximation of which a maximum error less than 2e-4 can be achieved for phase values between 0 to $\pi/2$. The full-period sinewave is generated by using a quadrant-decode technique. This yields a very high spectral purity sinewave output. System-level approximation and experimental results using an FPGA show that the maximum harmonic distortion is less than –67.4 dBc and –64.8 dBc respectively.

INTRODUCTION

Direct digital frequency synthesisers (DDFSs) are now playing an increasingly significant role in modern digital communication systems because of their characteristics of fast frequency switching and excellent frequency and phase resolution [1]. Almost all DDFSs are designed by using an architecture developed by Tierney *et. al.* [2], as shown in Fig. 1. The phase argument to the sine-computation block is generated by exploiting the modulo 2^N overflow of an N-bit accumulator, which can be adjusted by a frequency control word (W). The sine-computation block converts the phase values into a digital representation of a sinewave. The digital sinewave can be converted into its analog counterpart by a digital-to-analog converter and a low-pass filter. The 2^N binary words of the accumulator are mapped into phase values such that

$$m = \left(\frac{\phi}{2\pi}\right)2^N \qquad (1)$$

where m is the binary word of the accumulator. The output frequency of the DDFS can thus be expressed as,

$$f_{out} = \frac{W * f_{clk}}{2^N}, W < 2^{N-1} \qquad (2)$$

where W is the frequency control word, f_{clk} is the clock frequency, N is the bit-size of the frequency control word and f_{out} is the output frequency. The frequency resolution (Δf) of the DDFS can be derived from (2) by setting W=1 as,

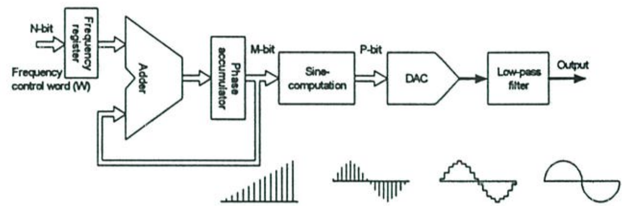


Figure 1. Typical DDFS architecture

$$\Delta f = \frac{f_{clk}}{2^N} \qquad (3)$$

Typically, the sine function generator is realised by using a look-up table provided by a read-only memory (ROM). The phase and amplitude quantisation errors are determined, respectively, by the number of words and the number of bits in each word in the ROM. Thus it is desirable to increase the resolution of the ROM for a high spectral purity sine output. This leads to larger ROM size, which means higher power consumption, low reliability, lower speed and increased costs.

There are a number of techniques [3], which can reduce the ROM size while retaining the spectral purity and frequency resolution. These methods include exploitation of trigonometric identities [3] and approximation of the sine function [4, 5]. An alternative approach, which avoids using the ROM, is to compute the samples of sine amplitude from the digital phase contents. Examples of this approach include Taylor series approximation [6], CORDIC algorithm, and parabolic approximation [7]. In particular, the DDFS based on the parabolic approximation in [7] can achieved high speed and high spectral purity.

This paper describes a ROM-less sine-output DDFS by using a polynomial approximation, which can achieve higher accuracy comparing to other reported sine approximations. Next section describes the polynomial approximation, followed by an implementation of the DDFS. Then experimental results and a conclusion are given.

THE POLYNOMIAL APPROXIMATION

The polynomial approximation for sine function computation is given by

$$\sin(x) = x\left(1 + a_2x^2 + a_4x^4\right) + \varepsilon(x) \quad ,0 \le x \le \frac{\pi}{2} \quad (4)$$

where the error $|\varepsilon(x)| \le 2\times10^{-4}$

$$a_2 = -0.16605, a_4 = 0.00761$$

Equation (4) is similar to the series expansion of sin x except for the coefficients a_2 and a_4 which are optimised to minimise the error. The maximum error of the polynomial approximation in (4) is less than 2e-4, which is much smaller than those obtained from the series expansion and the 2nd-order parabolic approximation [7]. This is shown in Fig. 2. Therefore the polynomial approximation in (4) can be used to generate a digital representation of a sinewave with higher spectral purity.

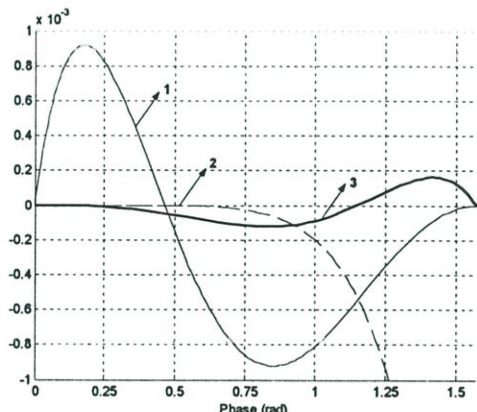


Figure 2. Errors from three approximations: (1) the second-order parabolic approximation; (2) the series expansion; and (3) the polynomial approximation.

DIGITAL REALISATION

Recall that the phase values are mapped into the sine function by using the relationship in (1). Thus the polynomial approximation in (4) can be expressed in terms of the phase values as

$$y(m) = m\left(k_1 - k_2m^2 + k_3m^4\right) \qquad (5)$$

where the values of k_1, k_2 and k_3 are shown in Table 1.

Table 1. Constants in equation (5)

k_1	1,000,000,000
k_2	391.046368345922...
k_3	4.22049520104...e-5

It can be seen that equation (5) requires five multiplications, which are undesirable in terms of hardware implementation. The multiplications appeared in k_2m^2 and k_3m^4 are replaced with shift-and-add operation. The values of k_1 and k_2, computed by shift-and-add operation, are shown in Table 2.

Table 2. Values of k_2, and k_3 computed by shift-and-add operation

K	Value	Shift/add operation
k_2	392	$2^8 + 2^7 + 2^3$
k_3	4.1961669921875e-5	$1/2^{15} + 1/2^{16} - 1/2^{18}$

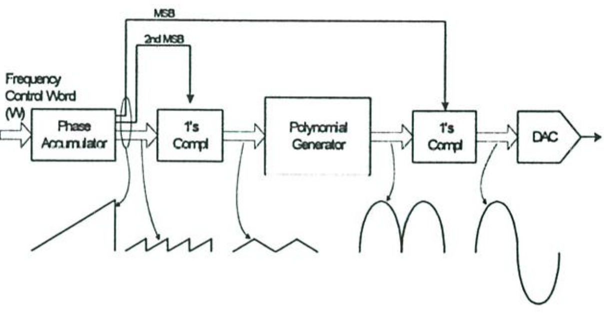


Figure 3. Architecture of the polynomial approximation direct digital frequency synthesiser

The architecture of the DDFS using a polynomial approximation is shown in Fig. 3. The phase accumulator generates a digital-sweep phase, which is converted to sine-amplitude by the polynomial generator. As previously stated, the accuracy of the polynomial approximation is excellent between 0 to π/2. Thus the quadrant-decode technique [8] is employed in order to generate a full sinewave. The second MSB is used to determine whether the output of phase accumulator has to be inverted, and the first MSB determines whether the sine-amplitude output of the polynomial generator has to be inverted.

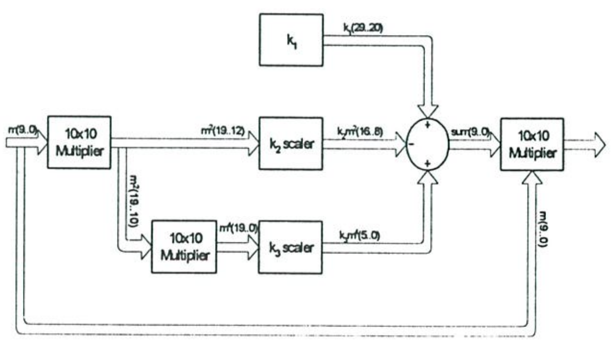


Figure 4. Block diagram of the polynomial generator

The block diagram of the polynomial generator, shown in Fig. 4, comprises three 10x10-bit multipliers, two shift-and-add scalers, and one adder/subtracter. The 32-bit output of phase accumulator is truncated to 12 bits, m(11..0). The two most significant bits are used for quadrant decoding, and the ten remaining bits, m(9..0), are converted to sine amplitude by the polynomial generator. The value of k is 10^9 as indicated in Table 1, and the k_2 and k_3 scalers are implemented with shift-and-add operation previously stated in Table 2. The bit-size of each internal block is derived from a system-level simulation.

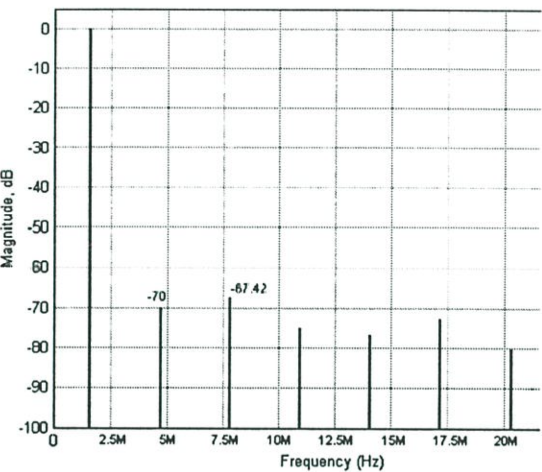


Figure 5. Power spectrum of the simulated 1.5625-MHz (fclk/32) sinewave

In order to verify the functionality of the proposed architecture, a system-level simulation has been conducted. Fig. 5 shows the power spectrum of the simulated 1.5625-MHz sinewave, which has a maximum harmonic level of –67.42 dBc. As we have already known that the spectrum level of the truncated 10-bit output is about –61.96 dBc [9], it can be confirmed that the polynomial approximation method is very suitable for phase to sine-amplitude conversion in direct digital frequency synthesis.

EXPERIMENTAL RESULTS

An RTL-level VHDL-model was developed and implemented on an FPGA in order to verify the operation of the proposed DDFS. The testing environment comprises a Spartan-II FPGA (XC2S100PQ208 – 5), a 10-bit digital-to-analog converter, a logic analyzer, an oscilloscope and a spectrum analyser as shown in Fig. 6. The DDFS operates at 50-MHz clock frequency. Fig. 7 shows a sinewave output at 1.5625 MHz (flck/32). To exclude the non-linearity effects of the digital-to-analog converter, the 10-bit digital data from the logic analyzer was used to calculate the output spectrum of the synthesised sinewave. Fig. 8 shows that the maximum harmonic distortion is below –64.87 dBc for 1.5625-MHz output. Fig. 9 shows the spectral components, obtained from the spectrum analyzer, at 197-kHz, where the DAC's non-linearity is negligible. The maximum harmonic level is below –67.59 dBc. Measured performances of the proposed DDFS are summarized in Table 3.

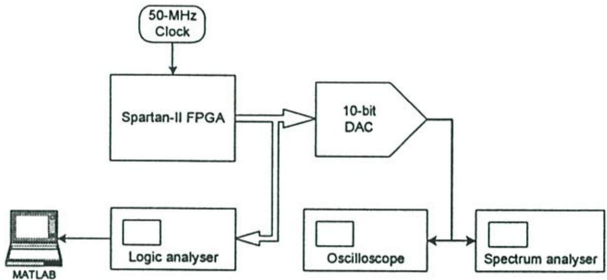


Figure 6. Test system of the DDFS

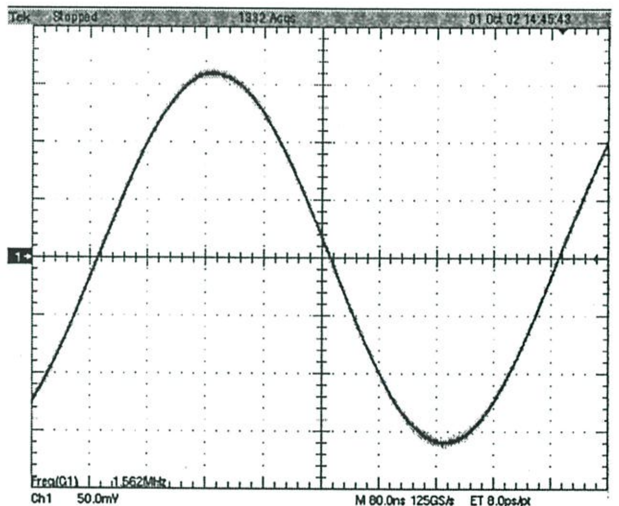


Figure 7. Waveform of 1.5625-MHz sinewave

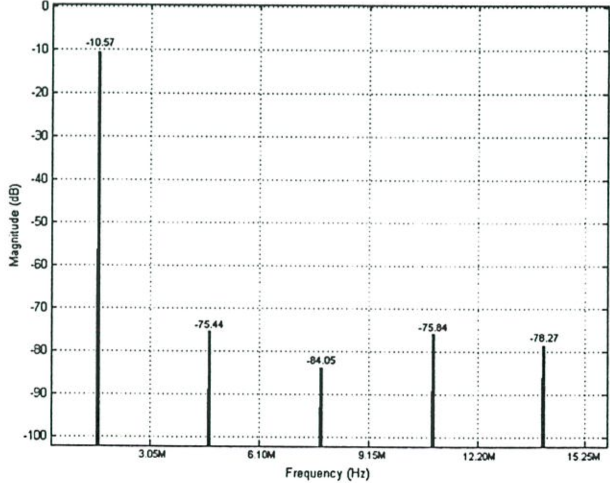


Figure 8. Power spectrum of 1.5625-MHz sinewave

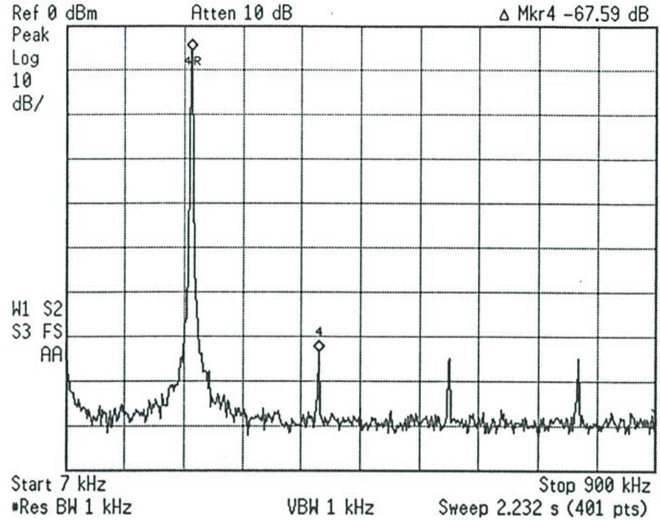


Figure 9. Power spectrum of 197-kHz sinewave

Table 3. Measured performances of the proposed DDFS

FPGA Technology	Spartan-II (XC2S100PQ208-5)
Frequency control word	32 bits
Phase resolution	12 bits
No. of output bits	10 bits
Frequency switching latency	6 clock cycles
Maximum clock frequency	50 MHz
Worst case spurious	-64.8 dBc
Total equivalent gate count	5,611 gates

CONCLUSION

The DDFS using a polynomial approximation has been proposed, simulated and implemented on an FPGA. Experimental results are very close to the system-level simulation. This confirms the functionality and feasibility of a polynomial approximation approach. Note that the experiment conducted in this paper was only aimed for evaluating the function of the DDFS. Thus, for further optimisation, optimised multipliers [10], or even specific squarers [11, 12] can be employed, including a pipeline if high operating speed is required.

REFERENCES

[1] V. F. Kroupa, *Direct digital frequency synthesizers*, IEEE Press, 1999.

[2] J. Tierney, C. Rader, and B. Gold, "A digital frequency synthesizer," *IEEE Trans. Audio Electroacoust.*, vol. AU-19, pp. 48-57, Mar. 1971.

[3] J. Vankka, "Methods of mapping from phase to sine amplitude in direct digital synthesis," *in Proc. 1996 IEEE Int. Frequency Control Symp.*, pp. 942-950.

[4] K. I. Palomaki, and J. Niittylahti, "Direct digital frequency synthesizer architecture based on Chebyshev approximation," *in Proc. 34th Asilomar Conf. on Signals, Systems and Computers*, 29 Oct.-1 Nov. 2000, vol.2 pp. 1639-1643.

[5] K. I. Palomaki, *et al.*, "Numerical sine and cosine synthesis using a complex multiplier," *in Proc. 1999 IEEE Int. Symp. on Circuits and Syst. (ISCAS'99)*, vol. 4, pp. 356-359.

[6] K. I. Palomaki, and J. Niittylahti, "Methods to improve the performance of quadrature phase-to-amplitude conversion based on Taylor series approximation," *in Proc. 43rd IEEE Midwest Symp. on Circuits and Syst.*, vol. 1, pp. 14-17.

[7] A. M. Sodagar, and G. R. Lahiji, "A pipelined ROM-less architecture for sine-output direct digital frequency synthesizers using the second-order parabolic approximation," *IEEE Trans. Circuits and Syst.-II*, vol. 48, no. 9, pp. 850-857, Sep 2001.

[8] B. Goldberg, *Digital techniques in frequency synthesis*, McGraw Hill, 1996, New York.

[9] Analog Devices, *A technical tutorial on digital signal synthesis*, 1999.

[10] S. Shah, *et al.*, "Comparison of 32-bit multipliers for various performance measures," *in Proc. 12th Int. Conference on Microelectronics*, Tehran, pp. 75-80, Nov. 2000.

[11] J. Yoo, *et al.*, "A fast parallel squarer based on divide-and-conquer," *IEEE J. Solid-State Circuits*, vol. 32, no. 6, pp. 909-912, Jun. 1997.

[12] Y. B. Mahdy, *et al.*, "A fast scheme and implementation for n-bit squarer," *in Proc. 6th IEEE Int. Conference on Electronics, Circuits and Systems (ICECS'99)*, vol. 1, pp. 25-28, 1999.

A Sine-Output ROM-Less Direct Digital Frequency Synthesiser Using a Polynomial Approximation

C. Meenakarn*,**, and A. Thanachayanont*

* Research Center of Communications and Information Technology, and Faculty of Engineering,
King Mongkut's Institute of Technology Ladkrabang, Bangkok, THAILAND
** Thailand IC Design Incubator (TIDI),
National Electronics and Computer Technology Center (NECTEC), Nonthaburi, THAILAND

ABSTRACT

This paper describes the design and implementation of a ROM-less sine-output direct digital frequency synthesiser by using a polynomial approximation, which provides a maximum error less than 2e-4 for phase values between 0 to $\pi/2$. The full-period sinewave is generated by using a quadrant-decode technique. This yields a very high spectral purity sinewave output. System-level approximation and experimental results using an FPGA show that the maximum harmonic distortion is less than –67.4 dBc and –64.8 dBc respectively.

1. INTRODUCTION

Direct digital frequency synthesisers (DDFS) are now playing an increasingly significant role in modern digital communication systems because of their characteristics of fast frequency switching and excellent frequency and phase resolution [1]. Almost all DDFSs are designed by using an architecture developed by Tierney *et. al.* [2], as shown in Fig. 1. The phase argument to the sine-computation block is generated by exploiting the modulo 2^N overflow of an N-bit accumulator, which can be adjusted by a frequency control word (W). The sine-computation block converts the phase values into a digital representation of a sinewave. The digital sinewave can be converted into its analog counterpart by a digital-to-analog converter and a low-pass filter. The 2^N binary words of the accumulator are mapped into phase values such that

$$m = \left(\frac{\phi}{2\pi}\right)2^N \quad (1)$$

where m is the binary word of the accumulator, and ϕ is the phase value between 0 to 2π. The output frequency of the DDFS can thus be expressed as,

$$f_{out} = \frac{W * f_{clk}}{2^N}, W < 2^{N-1} \quad (2)$$

where W is the frequency control word, f_{clk} is the clock frequency, N is the bit-size of the frequency control word and f_{out} is the output frequency. The frequency resolution (Δf) of the DDFS can be derived from (2) by setting W=1 as,

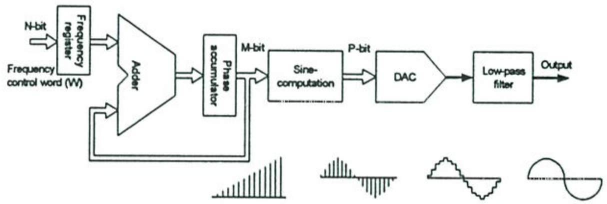


Figure 1. Typical DDFS architecture

$$\Delta f = \frac{f_{clk}}{2^N} \quad (3)$$

Typically, the sine function generator is realised by using a look-up table provided by a read-only memory (ROM). The phase and amplitude quantisation errors are determined, respectively, by the number of words and the number of bits in each word in the ROM. Thus it is desirable to increase the resolution of the ROM for a high spectral purity sine output. This leads to larger ROM size, which means higher power consumption, low reliability, lower speed and increased costs.

There are a number of techniques [3], which can reduce the ROM size while retaining the spectral purity and frequency resolution. These methods include exploitation of trigonometric identities [3] and approximation of the sine function [4, 5]. An alternative approach, which avoids using the ROM, is to compute the samples of sine amplitude from the digital phase contents. Examples of this approach include Taylor series approximation [6], CORDIC algorithm, and parabolic approximation [7]. In particular, the DDFS based on the parabolic approximation in [7] can achieve high speed and high spectral purity.

This paper describes a ROM-less sine-output DDFS by using a polynomial approximation, which can achieve higher accuracy comparing to other reported sine approximations. Section 2 describes the polynomial approximation. Section 3 describes the DDFS implementation. Experimental results and conclusion are given in sections 4 and 5 respectively.

2. THE POLYNOMIAL APPROXIMATION

The polynomial approximation for sine function computation is given by

$$\sin(\phi) = \phi\left(1 + a_2\phi^2 + a_4\phi^4\right) + \varepsilon(\phi) \quad ,0 \le \phi \le \frac{\pi}{2} \qquad (4)$$

where the error $|\varepsilon(\phi)| \le 2\times10^{-4}$

$$a_2 = -0.16605, a_4 = 0.00761$$

Equation (4) is similar to the Taylor series expansion of sin ϕ except for the coefficients a_2 and a_4 which are optimised to minimise the error. For phase values between 0 to $\pi/2$, the maximum error of the polynomial approximation in (4) is less than 2e-4, which is much smaller than those obtained from the 5th-order Taylor series expansion and the 2nd-order parabolic approximation [7]. This is shown in Fig. 2. Therefore the polynomial approximation in (4) can be used to generate a digital representation of a sinewave with higher spectral purity.

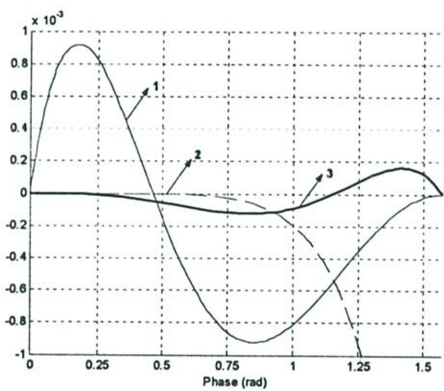


Figure 2. Errors from three approximations: (1) the 2nd–order parabolic approximation; (2) the 5th–order Taylor series expansion; and (3) the polynomial approximation (used in this paper).

3. DIGITAL REALISATION

For practical implementation, the mapped polynomial approximation is scaled up by a factor of 10^9, and the resulting function can be expressed in terms of the binary word of the accumulator as

$$y(m) = m\left(k_1 - k_2 m^2 + k_3 m^4\right) \qquad (5)$$

where the values of k_1, k_2 and k_3 are shown in Table 1.

Table 1. Constants in equation (5)

k_1	1,000,000,000
k_2	391.046368345922...
k_3	4.22049520104...e-5

It can be seen that equation (5) requires five multiplications, which are undesirable in terms of hardware implementation. Therefore the multiplications appeared in k_2m^2 and k_3m^4 are replaced with shift-and-add operation. The values of k_2 and k_3, computed by shift-and-add operation, are shown in Table 2.

Table 2. Values of k_2, and k_3 computed by shift-and-add operation

K	Value	Shift/add operation
k_2	392	$2^8 + 2^7 + 2^3$
k_3	4.1961669921875e-5	$1/2^{15} + 1/2^{16} - 1/2^{18}$

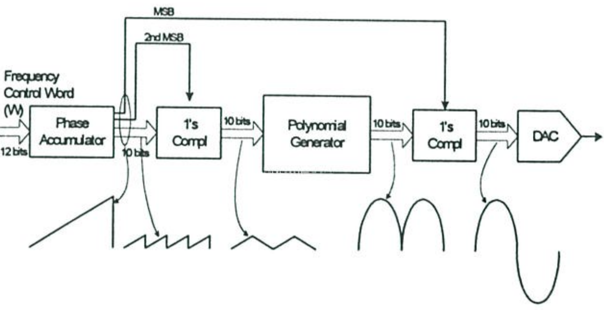


Figure 3. Architecture of the polynomial approximation direct digital frequency synthesiser

The architecture of the DDFS using a polynomial approximation is shown in Fig. 3. The phase accumulator generates a digital-sweep phase, which is converted to sine-amplitude by the polynomial generator. As previously stated, the accuracy of the polynomial approximation is excellent between 0 to $\pi/2$. Thus the quadrant-decode technique [8] is employed in order to generate a full sinewave. The second MSB is used to determine whether the output of phase accumulator has to be inverted, and the first MSB determines whether the sine-amplitude output of the polynomial generator has to be inverted.

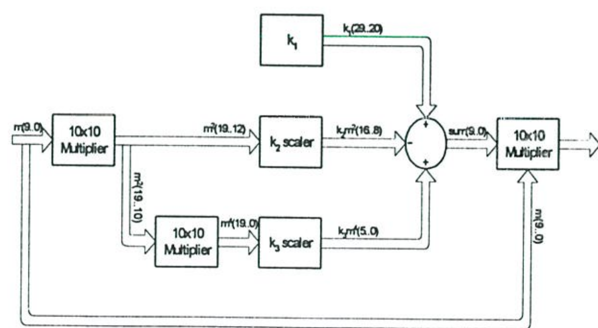


Figure 4. Block diagram of the polynomial generator

The block diagram of the polynomial generator, shown in Fig. 4, comprises three 10x10-bit multipliers, two shift-and-add scalers, and one adder/subtracter. The 32-bit output of phase accumulator is truncated to 12 bits, m(11..0). The two most significant bits are used for quadrant decoding, and the ten remaining bits, m(9..0), are converted to sine amplitude by the polynomial generator. The

value of k_1 is 10^9 as indicated in Table 1, and the k_2 and k_3 scalers are implemented with shift-and-add operation previously stated in Table 2. The bit-size of each internal block is derived from a system-level simulation.

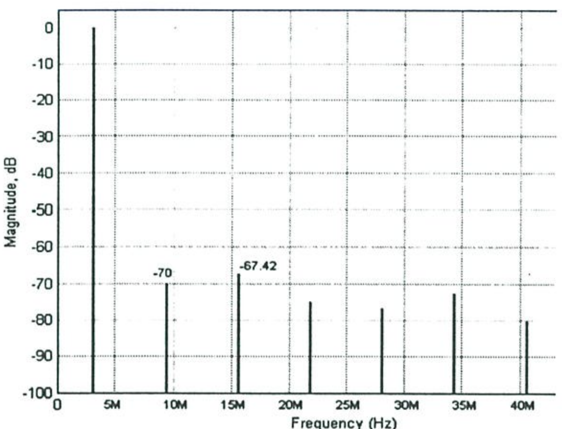


Figure 5. Power spectrum of the simulated 3.125-MHz (Fclk/32) sinewave

In order to verify the functionality of the proposed architecture, a system-level simulation has been conducted. Fig. 5 shows the power spectrum of the simulated 3.125-MHz sinewave, which has a maximum harmonic level of –67.42 dBc. As we have already known that the spectrum level of the truncated 10-bit output is about –61.96 dBc [9], it can be confirmed that the polynomial approximation method is very suitable for phase to sinc-amplitude conversion in direct digital frequency synthesis.

4. EXPERIMENTAL RESULTS

An RTL-level VHDL-model was developed and implemented on an FPGA in order to verify the operation of the proposed DDFS. The testing environment comprises a Spartan-II FPGA (XC2S100PQ208 –5), a 10-bit digital-to-analog converter, a logic analyser, an oscilloscope and a spectrum analyser as shown in Fig. 6. The DDFS operates at 50-MHz clock frequency. Fig. 7 shows a sinewave output at 1.5625 MHz (flck/32). To exclude the non-linearity effects of the digital-to-analog converter, the 10-bit digital data from the logic analyzer was used to calculate the output spectrum of the synthesised sinewave. Fig. 8 shows that the maximum harmonic distortion is below –64.87 dBc for 1.5625-MHz output. Fig. 9 shows the spectral components, obtained from the spectrum analyzer, at 197-kHz, where the DAC's non-linearity is negligible. The maximum harmonic level is below –67.59 dBc. Measured performances of the proposed DDFS are summarized in Table 3.

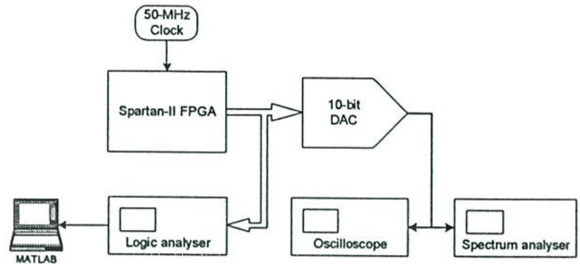


Figure 6. Test system of the DDFS

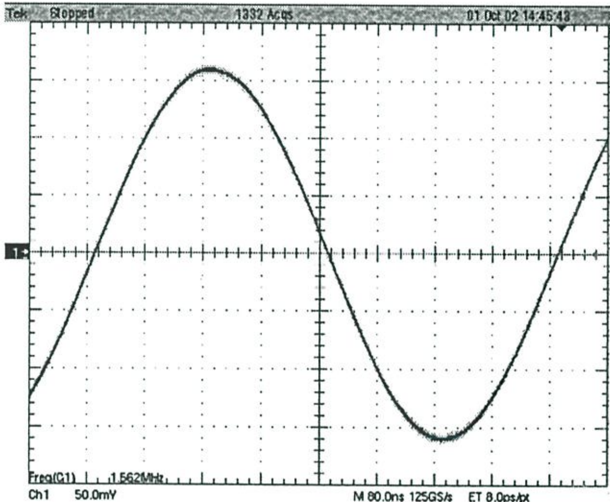


Figure 7. Waveform of 1.5625-MHz sinewave

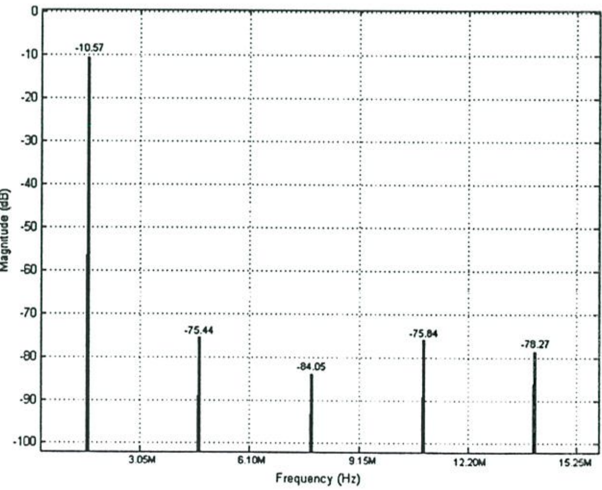


Figure 8. Power spectrum of 1.5625-MHz sinewave

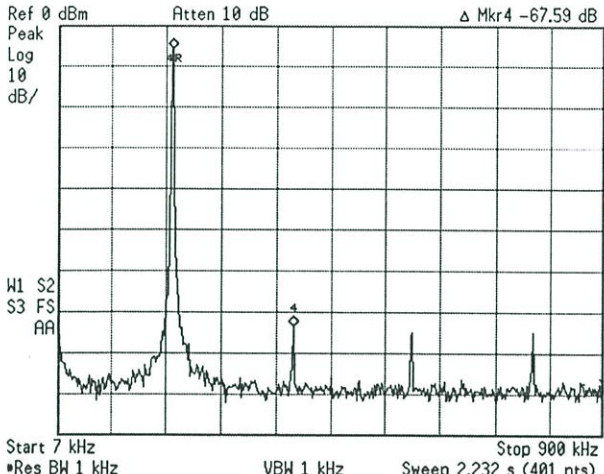


Figure 9. Power spectrum of 197-kHz sinewave

Table 3. Specifications of the DDFS

FPGA Technology	Spartan-II (XC2S100PQ208-5)
Frequency control word	32 bits
Phase resolution	12 bits
No. of output bits	10 bits
Frequency switching latency	6 clock cycles
Maximum clock frequency	50 MHz
Worst case spurious	-64.87 dBc
Total no. of CLBs	266 CLBs

5. CONCLUSION

The DDFS using a polynomial approximation has been proposed, simulated and implemented on an FPGA. Experimental results are very close to the system-level simulation. This confirms the functionality and feasibility of a polynomial approximation approach. Note that an experiment conducted in this paper was only aimed for evaluating the function of the DDFS. Thus, for further opimisation, a number of optimised multipliers [10], or even specific squarers [11, 12] can be employed, including a pipeline if high operating speed is required.

6. REFERENCES

[1] V. F. Kroupa, *Direct digital frequency synthesizers*, IEEE Press, 1999.

[2] J. Tierney, C. Rader, and B. Gold, "A digital frequency synthesizer," *IEEE Trans. Audio Electroacoust.*, vol. AU-19, pp. 48-57, Mar. 1971.

[3] J. Vankka, "Methods of mapping from phase to sine amplitude in direct digital synthesis," *in Proc. 1996 IEEE Int. Frequency Control Symp.*, pp. 942-950.

[4] K. I. Palomaki, and J. Niittylahti, "Direct digital frequency synthesizer architecture based on Chebyshev approximation," *in Proc. 34th Asilomar Conf. on Signals, Systems and Computers*, 29 Oct.-1 Nov. 2000, vol.2 pp. 1639-1643.

[5] K. I. Palomaki, *et al.*, "Numerical sine and cosine synthesis using a complex multiplier," *in Proc. 1999 IEEE Int. Symp. on Circuits and Syst. (ISCAS'99)*, vol. 4, pp. 356-359.

[6] K. I. Palomaki, and J. Niittylahti, "Methods to improve the performance of quadrature phase-to-amplitude conversion based on Taylor series approximation," *in Proc. 43rd IEEE Midwest Symp. on Circuits and Syst.*, vol. 1, pp. 14-17.

[7] A. M. Sodagar, and G. R. Lahiji, "A pipelined ROM-less architecture for sine-output direct digital frequency synthesizers using the second-order parabolic approximation," *IEEE Trans. Circuits and Syst.-II*, vol. 48, no. 9, pp. 850-857, Sep 2001.

[8] B. Goldberg, *Digital techniques in frequency synthesis*, McGraw Hill, 1996, New York.

[9] Analog Devices, *A technical tutorial on digital signal synthesis*, 1999.

[10] S. Shah, *et al.*, "Comparison of 32-bit multipliers for various performance measures," *in Proc. 12th Int. Conference on Microelectronics*, Tehran, pp. 75-80, Nov. 2000.

[11] J. Yoo, *et al.*, "A fast parallel squarer based on divide-and-conquer," *IEEE J. Solid-State Circuits*, vol. 32, no. 6, pp. 909-912, Jun. 1997.

[12] Y. B. Mahdy, *et al.*, "A fast scheme and implementation for n-bit squarer," *in Proc. 6th IEEE Int. Conference on Electronics, Circuits and Systems (ICECS'99)*, vol. 1, pp. 25-28, 1999.

ประวัติผู้เขียน

นายชรัณ มีนกาญจน์ สำเร็จการศึกษาวิศวกรรมศาสตรบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์ (วศ.บ.) จากสถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง ปีการศึกษา 2540

หลังจากจบการศึกษาได้เข้าทำงานในตำแหน่งวิศวกรฝ่ายวิจัยและพัฒนา บริษัท ซินโดม อิเล็คทรอนิคส์ อินดัสตรี จำกัด และในปี พ.ศ. 2543 ถึง ปัจจุบัน ได้เข้าทำงานในตำแหน่งผู้ช่วยนักวิจัย ศูนย์พัฒนาธุรกิจออกแบบวงจรรวม ศูนย์เทคโนโลยีอิเล็กทรอนิกส์และคอมพิวเตอร์แห่งชาติ